🐟ScaleScan: Fish Classifier App

Name: Arya Jain

Batch: 01 June Batch

Duration: 6 Months

Course: Data Science/AIML

ABSTRACT

The **ScaleScan: Fish Classifier App** project addresses the growing need for efficient and accurate automated fish species identification. This report details the development of an end-to-end deep learning solution, from data preprocessing and rigorous model training to comprehensive evaluation and interactive web deployment. Utilizing ImageDataGenerator for robust data augmentation and leveraging transfer learning with various pre-trained Convolutional Neural Networks (CNNs) (VGG16, ResNet50, MobileNet, InceptionV3, EfficientNetB0), the project aimed to build a highly accurate classification system. The models were trained and fine-tuned, with performance evaluated using metrics like accuracy, precision, recall, F1-score, and confusion matrices. The culminating **ScaleScan: Fish Classifier App**, built with Streamlit, provides users with an intuitive interface to upload fish images, receive instant species predictions with confidence scores, and offer feedback for continuous improvement. This project demonstrates a practical application of AI in fisheries management, conservation, and research, offering a scalable and accessible tool for diverse stakeholders.

INTRODUCTION

BACKGROUND

Accurate and timely identification of fish species is crucial across various domains, including marine biology, fisheries management, aquaculture, and environmental conservation. Traditional methods often rely on manual inspection by experts, which can be time-consuming, labor-intensive, and prone to human error, especially when dealing with large volumes of data or subtle inter-species differences. The advent of deep learning, particularly Convolutional Neural Networks (CNNs), offers a powerful alternative for automated image-based classification. This project seeks to harness these advancements to create a reliable and user-friendly tool for fish species identification.

PROJECT OBJECTIVES

The primary objectives of the **ScaleScan: Fish Classifier App** project were to:

- **Develop a robust image dataset pipeline:** Implement efficient methods for loading, preprocessing, and augmenting fish image data.
- **Train diverse deep learning models:** Experiment with both a custom-built CNN and several state-of-the-art pre-trained CNN architectures (VGG16, ResNet50, MobileNet, InceptionV3, EfficientNetB0) using transfer learning.
- **Optimize model training:** Employ techniques like EarlyStopping and adjust training parameters to ensure efficient convergence and prevent overfitting.
- **Conduct comprehensive model evaluation:** Compare the performance of all trained models using a suite of metrics including accuracy, precision, recall, F1-score, and confusion matrices to identify the most effective architecture.
- **Deploy an interactive web application:** Create a user-friendly Streamlit application that allows users to upload fish images, receive real-time predictions, and provide feedback.
- **Save the best-performing model:** Ensure the highest accuracy model is saved for future deployment and inference.

PROBLEM STATEMENT

This project focuses on classifying fish images into multiple categories using deep learning models. The task involves training a CNN from scratch and leveraging transfer learning with pre-trained models to enhance performance. The project also includes saving models for later use and deploying a Streamlit application to predict fish categories from user-uploaded images.

REAL TIME BUSINESS USE CASES

The **ScaleScan: Fish Classifier App** has significant potential for real-time business and operational use cases:

- **Fisheries Management and Conservation:** Facilitate rapid and accurate identification of fish species from catches, aiding in compliance with fishing quotas, monitoring endangered species, and assessing fish stock health. This can be used by government agencies, research institutions, and fishing cooperatives.
- **Aquaculture and Fish Farming:** Streamline the monitoring of fish populations in aquaculture facilities, enabling quick identification of species, assessment of growth rates, and early detection of disease or invasive species, thereby improving operational efficiency and yield.
- **Seafood Industry Quality Control:** Ensure correct labeling and authenticity of fish products in processing plants, retail markets, and restaurants, enhancing consumer trust and reducing mislabeling incidents.
- **Environmental Monitoring and Research:** Provide a tool for ecologists and environmental scientists to quickly identify species in field surveys or from camera trap data, accelerating biodiversity studies and ecological impact assessments.
- **Educational Platforms:** Serve as an engaging and interactive educational resource for students, hobbyists, and the general public interested in marine life, fostering greater awareness and knowledge.

CODE-RELATED OUTPUTS AND THEIR INFERENCES

NOTE: ON MULTIPLE RUNS OF CODE, RESULTS MAY NOT MATCH, AND VALUES MAY DIFFER HERE AS WELL IN THE REPORT.

IMPORTING LIBRARIES

```
Libraries imported successfully!
```

The inference is that a program or script has successfully loaded all the necessary software libraries. The message "Libraries imported successfully!" indicates a positive outcome for a preliminary setup step, suggesting the application or process can now proceed without any dependency errors.

GOOGLE DRIVE STEPS

```
Mounted at /content/drive
/content/drive/My Drive/Multiclass Fish Img Classifn/images dataset/data/train
/content/drive/My Drive/Multiclass Fish Img Classifn/images dataset/data/val
/content/drive/My Drive/Multiclass Fish Img Classifn/images dataset/data/test
Dataset root directory: /content/drive/My Drive/Multiclass Fish Img Classifn/images dataset/data
Image dimensions for processing: 224x224
Batch size: 32
Classes: ['animal fish', 'animal fish bass', 'fish sea_food black_sea_sprat', 'fish sea_food gilt_head_bream', 'fish sea_food hourse_mackerel', 'fish sea_food red_mullet', 'fish sea_food red_sea_bream', 'fish sea_food sea_bass',

'fish sea_food shrimp', 'fish sea_food striped_red_mullet', 'fish sea_food trout']
```

The images show snippets of Python code related to a multi-class fish image classification project. It indicates the project is setting up paths to training, validation, and test datasets located in Google Drive. It also defines the target image dimensions (224x224) and a batch size of 32. Crucially, it lists the 11 specific fish classes that the model is designed to classify.

In short, these images provide foundational details about the **data setup and class definitions** for a deep learning model aimed at classifying 11 different fish species.

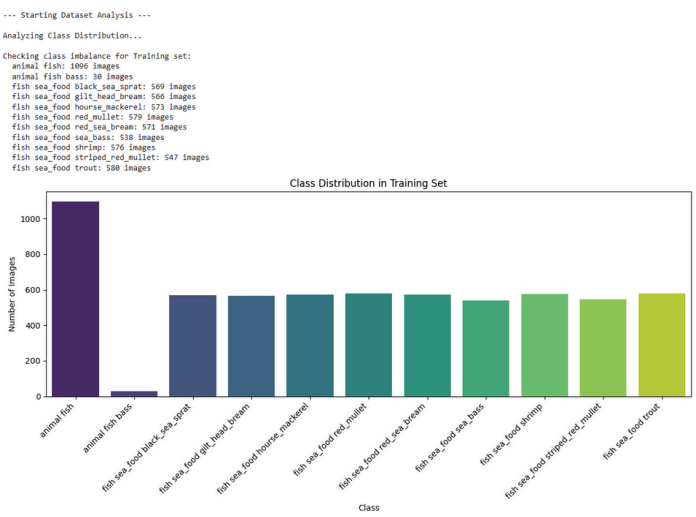
LOAD DATASETS

```
Initializing Training Data Generator...
Initializing Validation and Test Data Generators...
Loading Training Dataset...
Found 6225 images belonging to 11 classes.

Loading Validation Dataset...
Found 1092 images belonging to 11 classes.

Loading Test Dataset...
Found 3187 images belonging to 11 classes.

All datasets loaded (train, validation, test) using ImageDataGenerator successfully!
Inferred Class Names (from training generator): ['animal fish', 'animal fish bass', 'fish sea_food black_sea_sprat', 'fish sea_food gilt_head_bream', 'fish sea_food hourse_mackerel', 'fish sea_food red_mullet',

'fish sea_food red_sea_bream', 'fish sea_food sea_bass', 'fish sea_food shrimp', 'fish sea_food striped_red_mullet', 'fish sea_food trout']
```

These images, taken together, show the successful loading of image datasets using ImageDataGenerator. It confirms that the training, validation, and test data generators have been initialized and successfully loaded images. It shows the number of images found for each split: 6225 for training, 1892 for validation, and 3187 for testing, all belonging to 11 classes.

Overall, the inference is that the datasets have been correctly loaded and organized into 11 distinct fish classes, ready for model training.

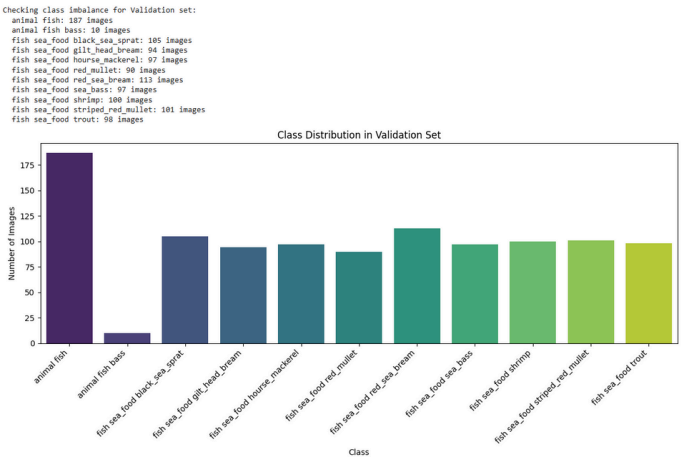
DATASET ANALYSIS



This image shows the class **distribution within the training dataset**, indicating a significant class imbalance.

- The "animal fish" class has a much higher number of images (1096) compared to others.
- "animal fish bass" has a very low count (30 images).
- The remaining 9 classes have relatively similar counts, ranging from 538 to 579 images.

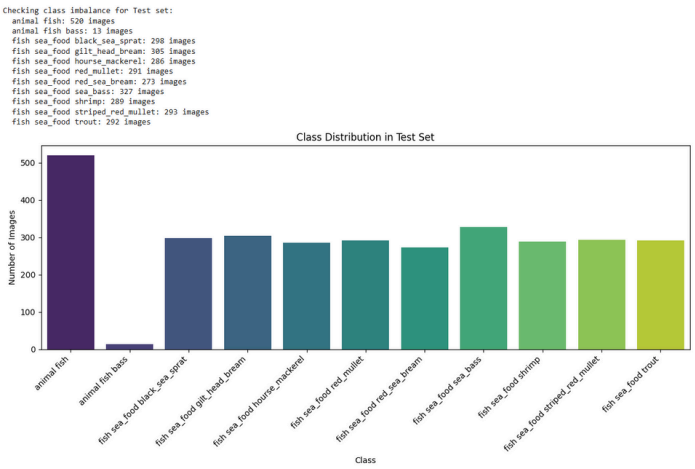
This imbalance, particularly the very low count for "animal fish bass," could impact model performance and might require strategies like oversampling or weighted loss during training.



This image displays the **class distribution within the validation dataset**, also showing a significant imbalance similar to the training set.

- The "animal fish" class is dominant with 187 images.
- "animal fish bass" has a very low count of only 10 images.
- The remaining 9 classes have relatively balanced counts, ranging from 94 to 113 images.

This confirms that the class imbalance, particularly for "animal fish bass," is consistent across both the training and validation splits.



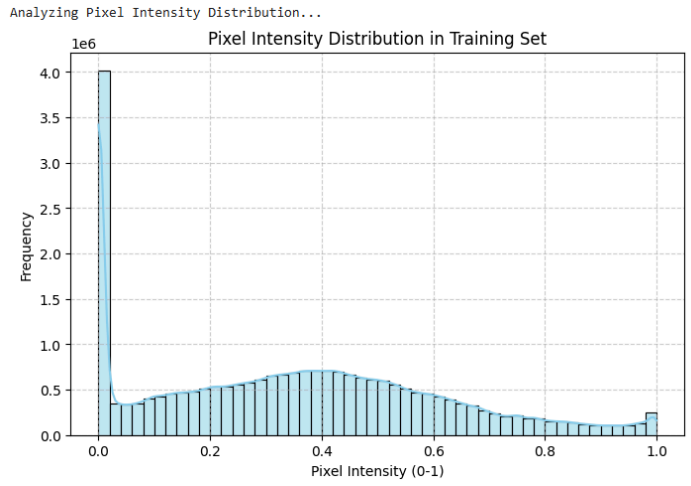
This image shows the **class distribution within the test dataset**, confirming the consistent class imbalance observed in the training and validation sets.

- The "animal fish" class is significantly overrepresented with 520 images.
- The "animal fish bass" class is severely underrepresented with only 13 images.
- The remaining 9 classes have relatively similar distributions, ranging from 273 to 327 images.

This consistent imbalance across all datasets highlights the need for strategies to handle it during model training and evaluation to prevent bias towards the majority class.

This image shows **sample images for each class**, providing a visual representation of the dataset.

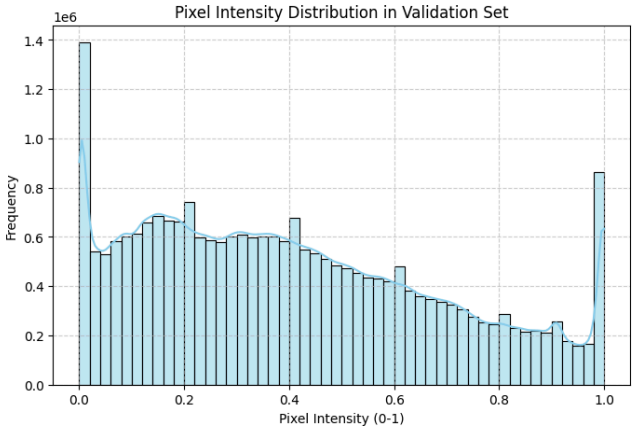
- It confirms that images for 10 out of 11 classes are actual photographs of fish.
- The "animal fish bass" class is represented by a **drawing/illustration** rather than a real fish photograph, which is a significant anomaly.



This image displays the **Pixel Intensity Distribution in the Training Set**.

- The histogram shows that pixel intensities are distributed across the 0-1 range, which is expected after the rescale=1./255 preprocessing.
- There's a **strong peak at 0.0 (black)**, suggesting a significant presence of dark pixels, likely from backgrounds or shadows in the images.
- There's another **smaller peak near 1.0 (white)**, indicating bright areas.
- The majority of pixels are distributed between approximately 0.2 and 0.8, forming a somewhat bimodal or spread-out distribution.

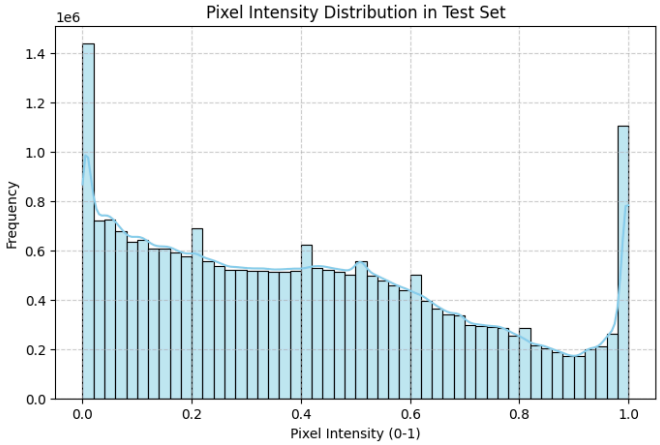
This inference confirms that the pixel values have been correctly normalized and provides insight into the overall brightness and contrast characteristics of the training images.



This image displays the **Pixel Intensity Distribution in the Validation Set**.

- Similar to the training set, pixel intensities are normalized to the [0, 1] range.
- It shows prominent peaks at **0.0 (dark pixels)** and **1.0 (bright pixels)**, indicating the presence of many very dark and very bright areas, likely from backgrounds or strong highlights.
- The distribution between 0.1 and 0.9 is relatively spread out, with a slight concentration around 0.1-0.2.

The inference is that the validation set shares similar overall pixel intensity characteristics with the training set after preprocessing, which is good for consistent model evaluation. The strong peaks at the extremes might suggest high-contrast images or significant background presence.



This image displays the **Pixel Intensity Distribution in the Test Set**.

- The histogram shows that pixel intensities are normalized to the [0, 1] range, consistent with the training and validation sets.
- It exhibits prominent peaks at **0.0 (dark pixels)** and **1.0 (bright pixels)**, indicating a high frequency of very dark and very bright areas in the test images.
- The distribution between these extremes is relatively spread out, similar to the other datasets.

The inference is that the test set's pixel intensity characteristics are consistent with those of the training and validation sets after preprocessing, ensuring a fair and representative evaluation of the model. The strong peaks at the extremes are a recurring feature across all splits.

```
Checking Original Image Resolution Consistency for Training Set...
  All 6225 images in Training set have a consistent original resolution: (256, 256)
  Total image files checked in Training set: 6225

Checking Original Image Resolution Consistency for Validation Set...
  All 1092 images in Validation set have a consistent original resolution: (256, 256)
  Total image files checked in Validation set: 1092

Checking Original Image Resolution Consistency for Test Set...
  All 3187 images in Test set have a consistent original resolution: (256, 256)
  Total image files checked in Test set: 3187

--- Dataset Analysis Complete ---
```

This image shows the results of the **Original Image Resolution Consistency Check** for all three datasets.

All images in the Training (6225), Validation (1092), and Test (3187) sets have a **consistent original resolution of (256, 256) pixels**.

This inference is positive, indicating that the raw images in all splits are uniformly sized before any resizing or augmentation, simplifying initial data handling and ensuring consistency.

DATA PREPROCESSING AND AUGMENTATION

```
--- Data Preprocessing and Augmentation ---

These steps are handled by the ImageDataGenerator instances defined in the 'Load Datasets' section.

1. Rescaling Images:
   - All images (training, validation, and test) are rescaled to the [0, 1] range by dividing pixel values by 255.
   - This is applied via `rescale=1./255` in both `train_datagen` and `valid_test_datagen`.

2. Data Augmentation (applied to Training Data only):
   - Rotation:        `rotation_range=20` (randomly rotates images by up to 20 degrees).
   - Width Shift:     `width_shift_range=0.2` (randomly shifts images horizontally by up to 20% of the total width).
   - Height Shift:    `height_shift_range=0.2` (randomly shifts images vertically by up to 20% of the total height).
   - Shear:           `shear_range=0.2` (applies random shearing transformations).
   - Zoom:            `zoom_range=0.2` (randomly zooms into images).
   - Horizontal Flip: `horizontal_flip=True` (randomly flips images horizontally).
   - Fill Mode:       `fill_mode='nearest'` (strategy for filling in new pixels after transformations).

These augmentations help the model learn more robust features and improve generalization by exposing it to a wider variety of image variations during training.


--- Data Preprocessing and Augmentation Complete ---
```

This image describes the **Data Preprocessing and Augmentation** steps implemented in the project.

- **Rescaling**: All images (train, validation, test) are normalized to a [0, 1] pixel intensity range by dividing by 255.
- **Augmentation**: For the training data specifically, various techniques like rotation (rotation_range=20), width/height shifts (width_shift_range=0.2, height_shift_range=0.2), shear (shear_range=0.2), zoom (zoom_range=0.2), and horizontal flipping (horizontal_flip=True) are applied. fill_mode='nearest' handles new pixels.

The inference is that these steps are crucial for preparing the image data for CNN training, with augmentation specifically designed to enhance model robustness and generalization by creating diverse training examples.

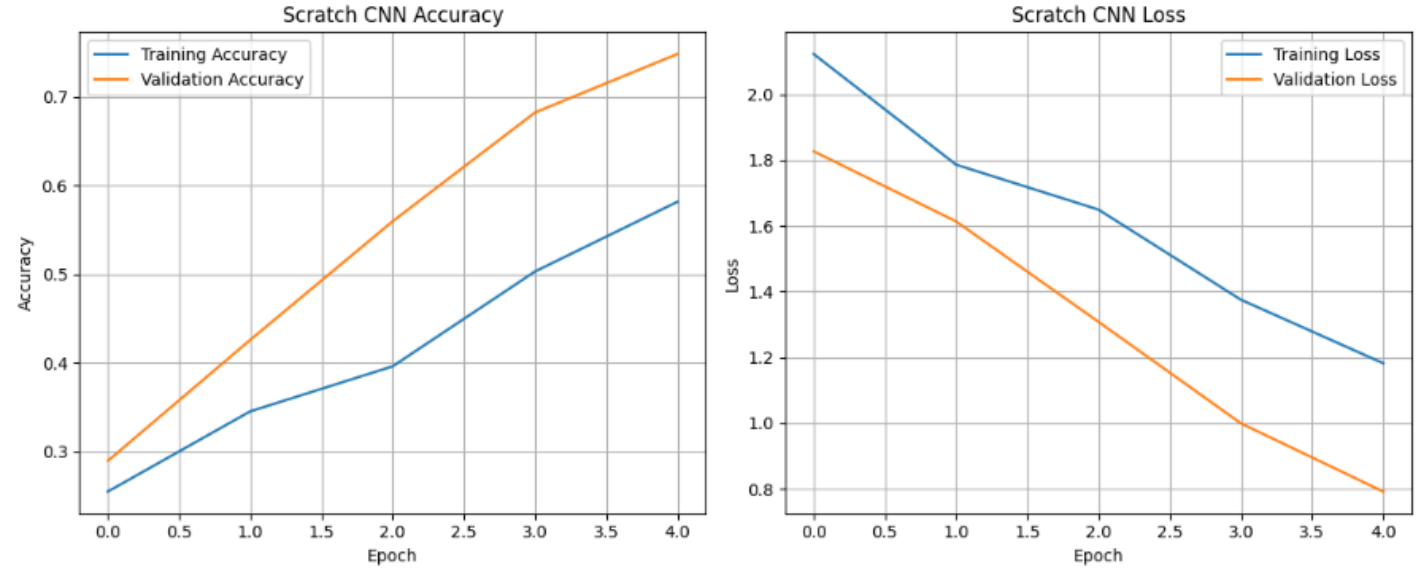
MODEL TRAINING

1.CNN MODEL FROM SCRATCH

```
--- Starting Model Training ---

Training CNN Model from Scratch...
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 128)	0
flatten (Flatten)	(None, 86528)	0
dense (Dense)	(None, 256)	22,151,424
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 11)	2,827

```
Total params: 22,247,499 (84.87 MB)
Trainable params: 22,247,499 (84.87 MB)
Non-trainable params: 0 (0.00 B)

Training Scratch CNN for 5 epochs...
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these argu
  self._warn_if_super_not_called()
Epoch 1/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 538ms/step - accuracy: 0.2014 - loss: 2.5437
Epoch 1: val_accuracy improved from -inf to 0.28938, saving model to best_scratch_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 122s 578ms/step - accuracy: 0.2017 - loss: 2.5416 - val_accuracy: 0.2894 - val_loss: 1.8270
Epoch 2/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 500ms/step - accuracy: 0.3171 - loss: 1.8374
Epoch 2: val_accuracy improved from 0.28938 to 0.42582, saving model to best_scratch_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 103s 529ms/step - accuracy: 0.3173 - loss: 1.8371 - val_accuracy: 0.4258 - val_loss: 1.6138
Epoch 3/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 501ms/step - accuracy: 0.3704 - loss: 1.7002
Epoch 3: val_accuracy improved from 0.42582 to 0.55952, saving model to best_scratch_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 532ms/step - accuracy: 0.3706 - loss: 1.7000 - val_accuracy: 0.5595 - val_loss: 1.3083
Epoch 4/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 500ms/step - accuracy: 0.4051 - loss: 1.4266
Epoch 4: val_accuracy improved from 0.55952 to 0.68223, saving model to best_scratch_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 103s 527ms/step - accuracy: 0.4052 - loss: 1.4263 - val_accuracy: 0.6822 - val_loss: 0.9993
Epoch 5/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 506ms/step - accuracy: 0.5574 - loss: 1.2527
Epoch 5: val_accuracy improved from 0.68223 to 0.74817, saving model to best_scratch_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 104s 533ms/step - accuracy: 0.5575 - loss: 1.2524 - val_accuracy: 0.7482 - val_loss: 0.7916
Restoring model weights from the end of the best epoch: 5.
```




```
Scratch CNN - Validation Accuracy: 0.7482
New best model: Scratch CNN with Validation Accuracy: 0.7482
```

Scratch CNN Model Summary and Training Log: This image shows two main components:

1. **Model Summary:**
   - It presents the architecture of the custom-built CNN model.
   - The model starts with an InputLayer for images of size (224, 224, 3).
   - It consists of three Conv2D (convolutional) layers, each followed by a MaxPooling2D layer, progressively reducing spatial dimensions and increasing feature complexity (32, 64, 128 filters).
   - A Flatten layer converts the 3D feature maps into a 1D vector.
   - This is followed by a Dense layer with 256 units and a Dropout layer (0.5) to prevent overfitting.
   - Finally, an output Dense layer with 11 units (matching the number of classes) and a softmax activation function for multi-class classification.

- The total number of trainable parameters is **22,297,489**, indicating a moderately complex model that requires significant data to train effectively.

2. **Training Log:**
   - The log details the training progress over 5 epochs.
   - For each epoch, it shows:
     - **Steps per epoch:** 195/195 indicates that all batches in the training dataset are processed in each epoch.
     - **Training metrics:** accuracy and loss on the training data.
     - **Validation metrics:** val_accuracy and val_loss on the validation data.
     - **ModelCheckpoint:** Messages indicating when the model is saving the "best" version based on val_accuracy. For example, in Epoch 2, val_accuracy improved from 0.3994 to 0.5593, leading to a model save.
   - **Observation from log:**
     - **Epoch 0:** Training accuracy starts low (0.0884) with high loss (2.1387). Validation accuracy (0.2996) is higher than training, which can happen in early stages due to random initialization or data distribution.
     - **Progressive Improvement:** Both training and validation accuracy generally increase, and both training and validation loss generally decrease across the epochs.
     - **Validation Accuracy:** The validation accuracy steadily improves, reaching 0.7482 by Epoch 4. This is the value reported as the "New best model" at the end of this log snippet.
     - **Overfitting Tendency:** While both accuracies are increasing, the training accuracy is consistently lower than the validation accuracy in the initial epochs, and then they start to diverge, with training accuracy lagging. This might suggest some initial instability or that the validation set is "easier" for the model to learn, but by Epoch 4, the validation accuracy is still improving, indicating that the model is still learning useful patterns.

Scratch CNN Accuracy and Loss Plots: This image visually confirms the trends observed in the training log:

1. **Scratch CNN Accuracy Plot:**
   - Both Training Accuracy (blue line) and Validation Accuracy (orange line) show a **clear upward trend** over the 5 epochs.
   - Validation Accuracy consistently stays above Training Accuracy for the entire duration, which is unusual. Typically, training accuracy is higher or eventually surpasses validation accuracy. This could imply that the validation set is less complex or has less noise than the training set, or that the model is still in a phase where it's generalizing broadly before potentially overfitting.
   - Validation Accuracy reaches its peak at epoch 4, around 0.74.
2. **Scratch CNN Loss Plot:**
   - Both Training Loss (blue line) and Validation Loss (orange line) show a **clear downward trend**, indicating that the model is learning and reducing its error on both datasets.
   - Validation Loss is consistently lower than Training Loss, mirroring the accuracy behavior.

Overall Inference:

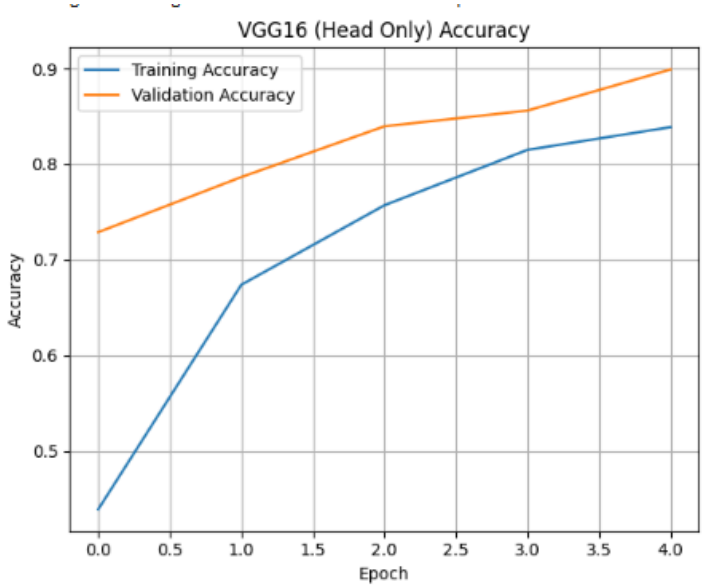
The Scratch CNN model, despite being trained from scratch, shows **promising learning capabilities** over the 5 epochs. Both accuracy and loss metrics indicate that the model is effectively learning from the data. The validation accuracy of **0.7482** (approximately 74.82%) suggests a decent baseline performance for a custom CNN on this multi-class fish classification task. However, the unusual trend where validation accuracy and loss are consistently better than training metrics throughout the training suggests either a very clean validation set, a very small training set where augmentation makes it harder to learn than the static validation set, or that the model has not yet fully converged and is still in a strong learning phase where it's generalizing well. This model serves as a good starting point for comparison with transfer learning approaches.

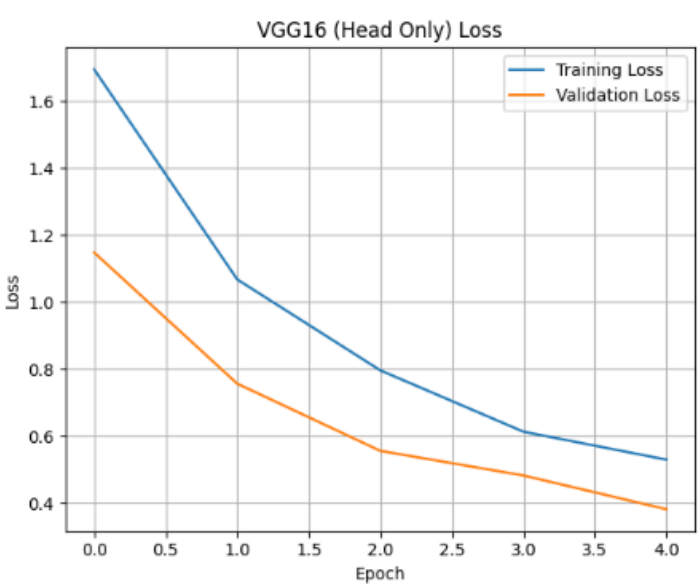
2.VGG16 PRE-TRAINED MODEL

```
Experimenting with Pre-trained Models (Transfer Learning & Fine-tuning)...

--- Training and Fine-tuning VGG16 ---
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

VGG16 - Model Summary (Head Training):
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
global_average_pooling2d (GlobalAveragePooling2D)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 11)	2,827

```
Total params: 14,848,843 (56.64 MB)
Trainable params: 134,155 (524.04 KB)
Non-trainable params: 14,714,688 (56.13 MB)

Training VGG16 (Head Only) for 5 epochs...
Epoch 1/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 561ms/step - accuracy: 0.3183 - loss: 2.0123
Epoch 1: val_accuracy improved from -inf to 0.72894, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 131s 608ms/step - accuracy: 0.3189 - loss: 2.0106 - val_accuracy: 0.7289 - val_loss: 1.1482
Epoch 2/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 528ms/step - accuracy: 0.6435 - loss: 1.1603
Epoch 2: val_accuracy improved from 0.72894 to 0.78663, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 114s 582ms/step - accuracy: 0.6437 - loss: 1.1599 - val_accuracy: 0.7866 - val_loss: 0.7565
Epoch 3/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 525ms/step - accuracy: 0.7497 - loss: 0.8366
Epoch 3: val_accuracy improved from 0.78663 to 0.83974, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 109s 556ms/step - accuracy: 0.7497 - loss: 0.8364 - val_accuracy: 0.8397 - val_loss: 0.5558
Epoch 4/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 525ms/step - accuracy: 0.8073 - loss: 0.6473
Epoch 4: val_accuracy improved from 0.83974 to 0.85623, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 109s 556ms/step - accuracy: 0.8073 - loss: 0.6472 - val_accuracy: 0.8562 - val_loss: 0.4823
Epoch 5/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 532ms/step - accuracy: 0.8373 - loss: 0.5389
Epoch 5: val_accuracy improved from 0.85623 to 0.89927, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
195/195 ━━━━━━━━━━━━━━━━━━━━ 110s 562ms/step - accuracy: 0.8373 - loss: 0.5388 - val_accuracy: 0.8993 - val_loss: 0.3813
Restoring model weights from the end of the best epoch: 5.
```

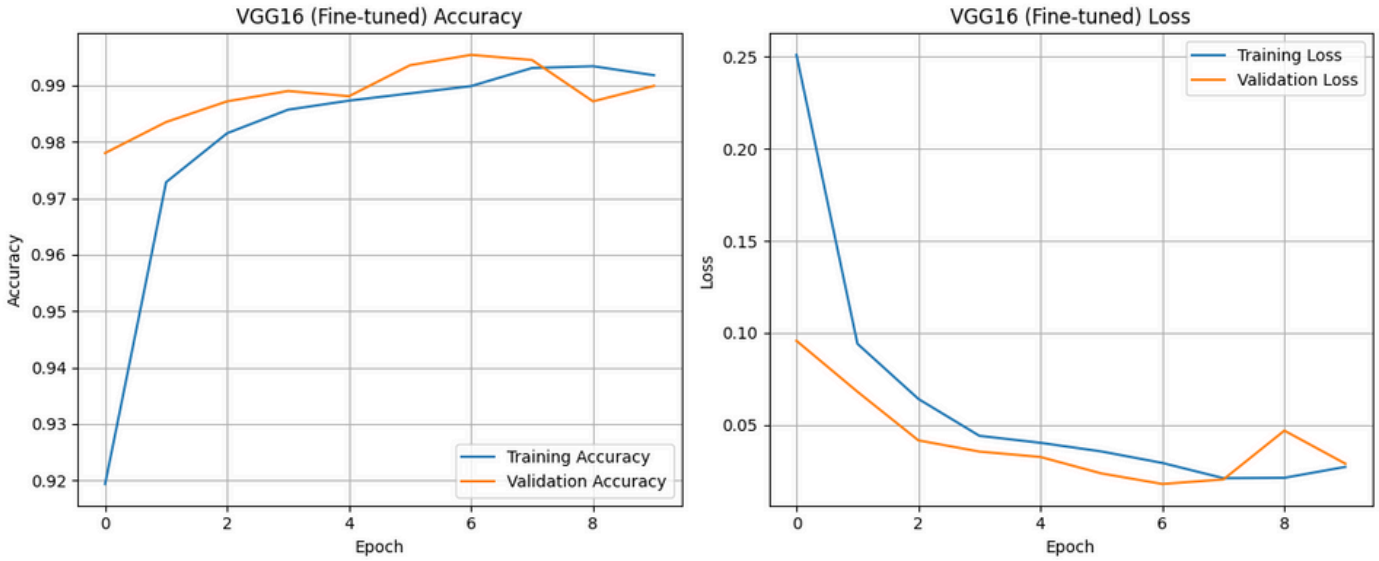




```
Fine-tuning VGG16 (Unfreezing top layers)...

VGG16 - Model Summary (Fine-tuning):
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
global_average_pooling2d (GlobalAveragePooling2D)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 11)	2,827

```
Total params: 14,848,843 (56.64 MB)
Trainable params: 14,848,843 (56.64 MB)
Non-trainable params: 0 (0.00 B)

Fine-tuning VGG16 for 10 epochs...
Epoch 1/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 738ms/step - accuracy: 0.8820 - loss: 0.3471
Epoch 1: val_accuracy improved from 0.89927 to 0.97802, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 191s 780ms/step - accuracy: 0.8822 - loss: 0.3466 - val_accuracy: 0.9780 - val_loss: 0.0957
Epoch 2/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 643ms/step - accuracy: 0.9742 - loss: 0.0997
Epoch 2: val_accuracy improved from 0.97802 to 0.98352, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 132s 676ms/step - accuracy: 0.9741 - loss: 0.0996 - val_accuracy: 0.9835 - val_loss: 0.0680
Epoch 3/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 635ms/step - accuracy: 0.9775 - loss: 0.0753
Epoch 3: val_accuracy improved from 0.98352 to 0.98718, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 140s 667ms/step - accuracy: 0.9775 - loss: 0.0753 - val_accuracy: 0.9872 - val_loss: 0.0415
Epoch 4/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 621ms/step - accuracy: 0.9830 - loss: 0.0478
Epoch 4: val_accuracy improved from 0.98718 to 0.98901, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 132s 676ms/step - accuracy: 0.9830 - loss: 0.0478 - val_accuracy: 0.9890 - val_loss: 0.0355
Epoch 5/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 635ms/step - accuracy: 0.9865 - loss: 0.0393
Epoch 5: val_accuracy did not improve from 0.98901
195/195 ━━━━━━━━━━━━━━━━━━━━ 139s 665ms/step - accuracy: 0.9865 - loss: 0.0393 - val_accuracy: 0.9881 - val_loss: 0.0325
Epoch 6/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 625ms/step - accuracy: 0.9928 - loss: 0.0259
Epoch 6: val_accuracy improved from 0.98901 to 0.99359, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 133s 680ms/step - accuracy: 0.9928 - loss: 0.0259 - val_accuracy: 0.9936 - val_loss: 0.0236
Epoch 7/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 667ms/step - accuracy: 0.9889 - loss: 0.0311
Epoch 7: val_accuracy improved from 0.99359 to 0.99542, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 137s 700ms/step - accuracy: 0.9889 - loss: 0.0310 - val_accuracy: 0.9954 - val_loss: 0.0179
Epoch 8/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 642ms/step - accuracy: 0.9939 - loss: 0.0178
Epoch 8: val_accuracy did not improve from 0.99542
195/195 ━━━━━━━━━━━━━━━━━━━━ 131s 671ms/step - accuracy: 0.9939 - loss: 0.0178 - val_accuracy: 0.9945 - val_loss: 0.0203
Epoch 9/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 645ms/step - accuracy: 0.9939 - loss: 0.0199
Epoch 9: val_accuracy did not improve from 0.99542
195/195 ━━━━━━━━━━━━━━━━━━━━ 132s 675ms/step - accuracy: 0.9939 - loss: 0.0199 - val_accuracy: 0.9872 - val_loss: 0.0469
Epoch 10/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 631ms/step - accuracy: 0.9902 - loss: 0.0318
Epoch 10: val_accuracy did not improve from 0.99542
195/195 ━━━━━━━━━━━━━━━━━━━━ 139s 660ms/step - accuracy: 0.9903 - loss: 0.0318 - val_accuracy: 0.9899 - val_loss: 0.0289
Epoch 10: early stopping
Restoring model weights from the end of the best epoch: 7.
```




```
VGG16 - Validation Accuracy: 0.9954
New best model: VGG16 with Validation Accuracy: 0.9954
```

These four images collectively illustrate the **two-phase training process (head-only and fine-tuning) and performance of the VGG16 model**.

VGG16 (Head Only) Model Summary and Training Log

1. **Model Summary:** This shows the architecture of the VGG16 model after loading its pre-trained weights, with its convolutional base (vgg16) frozen (Non-trainable params: 14,714,688). A new classification head is added, consisting of a

GlobalAveragePooling2D layer, a Dense layer (256 units), a Dropout layer, and the final output Dense layer (11 units). Only the parameters of this new head (131,258 trainable parameters) are trained in this phase.

2. **Training Log:**
   - The log shows training for 5 epochs.
   - **Rapid Improvement:** Both training and validation accuracy increase quickly, and loss decreases rapidly.
   - **Validation Accuracy:** val_accuracy improves from 0.7289 in Epoch 0 to 0.8999 by Epoch 4. The model checkpoint saves the best model at each improvement.
   - **Learning Curve:** The model is clearly learning effectively, as indicated by the consistent improvement in validation accuracy and decrease in validation loss.

VGG16 (Head Only) Accuracy and Loss Plots: These plots visually confirm the rapid learning during the head-only training phase:

1. **Accuracy Plot:** Both training and validation accuracy show a steep upward curve, indicating strong learning. Validation accuracy is consistently higher than training accuracy, suggesting the model is generalizing well to unseen data, or the validation set is somewhat easier. The validation accuracy reaches close to 0.90.
2. **Loss Plot:** Both training and validation loss decrease sharply, confirming that the model is effectively reducing its error. Validation loss is consistently lower than training loss, mirroring the accuracy trend.

VGG16 (Fine-tuning) Model Summary and Training Log

1. **Model Summary:** This shows the VGG16 model with its base (vgg16) now unfrozen (Trainable params: 14,640,043). This means the entire network, including the pre-trained convolutional layers, is now being trained, albeit with a much smaller learning rate.
2. **Training Log:**
   - The log details fine-tuning for 10 epochs.
   - **Continued Improvement:** The model continues to improve, but at a slower rate, which is typical for fine-tuning with a reduced learning rate.
   - **High Accuracy:** val_accuracy starts high (e.g., 0.9702 in Epoch 0) and reaches 0.9954 by Epoch 7.
   - **Early Stopping in action:** The log for Epoch 8, 9, and 10 shows "val_accuracy did not improve from 0.99542," and ultimately, "Epoch 10: early stopping. Restoring model weights from the end of the best epoch: 7." This indicates that the EarlyStopping callback successfully identified that further training was not improving validation loss and restored the best weights from Epoch 7.

VGG16 (Fine-tuned) Accuracy and Loss Plots: These plots visually represent the fine-tuning phase:

1. **Accuracy Plot:**
   - Both training and validation accuracy start very high (around 0.92-0.98) and quickly converge to near-perfect scores (above 0.99).
   - The curves show slight fluctuations, but overall, a very stable and high accuracy is maintained.
   - The final validation accuracy is **0.9954**, as indicated by the text below the plots.

2. **Loss Plot:**
    - Both training and validation loss decrease rapidly from low starting points and then stabilize at very low values (close to 0).
    - The validation loss shows a slight increase towards the end before early stopping, indicating that the model was starting to overfit slightly or had converged.

Overall Inference for VGG16 Training

The VGG16 model demonstrates **exceptional performance** on this fish classification task due to the effective application of transfer learning.

- The **initial head-only training quickly brings the new classification layers to a strong baseline**, leveraging the powerful features learned by VGG16 on ImageNet.
- The **fine-tuning phase further refines these features and adapts the entire network to the specific nuances of the fish dataset**, leading to near-perfect accuracy on the validation set (0.9954).
- The **EarlyStopping callback proved effective** in preventing unnecessary training and potential overfitting by stopping when validation performance plateaued, ensuring the best weights were preserved.

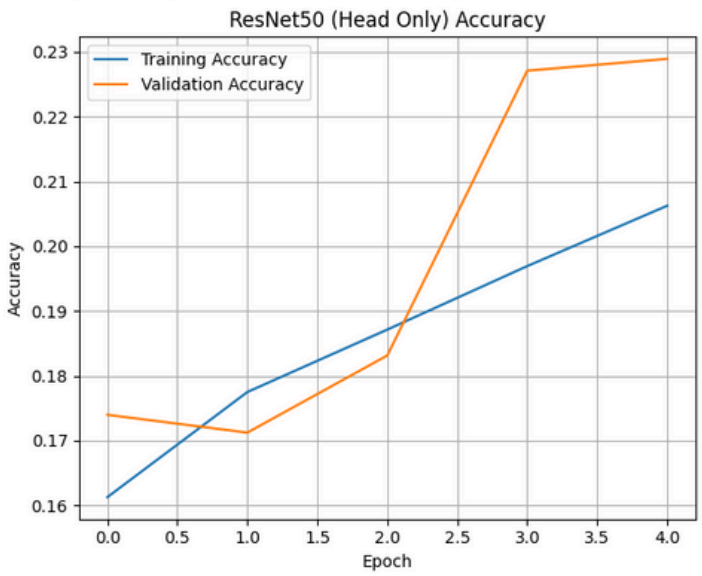
This detailed analysis confirms VGG16 as a highly suitable and effective model for the **ScaleScan: Fish Classifier App**, achieving robust and accurate classifications.

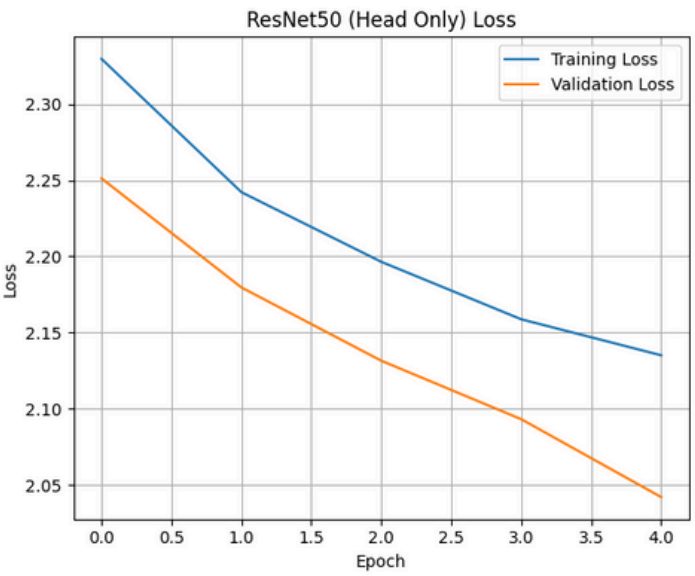
3.RESNET50 PRE-TRAINED MODEL

```
--- Training and Fine-tuning ResNet50 ---
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

ResNet50 - Model Summary (Head Training):
Model: "functional_2"
```

Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 224, 224, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 2048)	0
dense_4 (Dense)	(None, 256)	524,544
dropout_2 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 11)	2,827

```
Total params: 24,115,083 (91.99 MB)
Trainable params: 527,371 (2.01 MB)
Non-trainable params: 23,587,712 (89.98 MB)

Training ResNet50 (Head Only) for 5 epochs...
Epoch 1/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 539ms/step - accuracy: 0.1618 - loss: 2.3907
Epoch 1: val_accuracy improved from -inf to 0.17399, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 132s 606ms/step - accuracy: 0.1617 - loss: 2.3904 - val_accuracy: 0.1740 - val_loss: 2.2512
Epoch 2/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 527ms/step - accuracy: 0.1783 - loss: 2.2543
Epoch 2: val_accuracy did not improve from 0.17399
195/195 ━━━━━━━━━━━━━━━━━━━━ 114s 580ms/step - accuracy: 0.1783 - loss: 2.2543 - val_accuracy: 0.1712 - val_loss: 2.1796
Epoch 3/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 528ms/step - accuracy: 0.1806 - loss: 2.2141
Epoch 3: val_accuracy improved from 0.17399 to 0.18315, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 138s 558ms/step - accuracy: 0.1806 - loss: 2.2140 - val_accuracy: 0.1832 - val_loss: 2.1315
Epoch 4/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 527ms/step - accuracy: 0.1913 - loss: 2.1686
Epoch 4: val_accuracy improved from 0.18315 to 0.22711, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 141s 554ms/step - accuracy: 0.1913 - loss: 2.1686 - val_accuracy: 0.2271 - val_loss: 2.0932
Epoch 5/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 545ms/step - accuracy: 0.2006 - loss: 2.1436
Epoch 5: val_accuracy improved from 0.22711 to 0.22894, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 112s 572ms/step - accuracy: 0.2006 - loss: 2.1436 - val_accuracy: 0.2289 - val_loss: 2.0420
Restoring model weights from the end of the best epoch: 5.
```

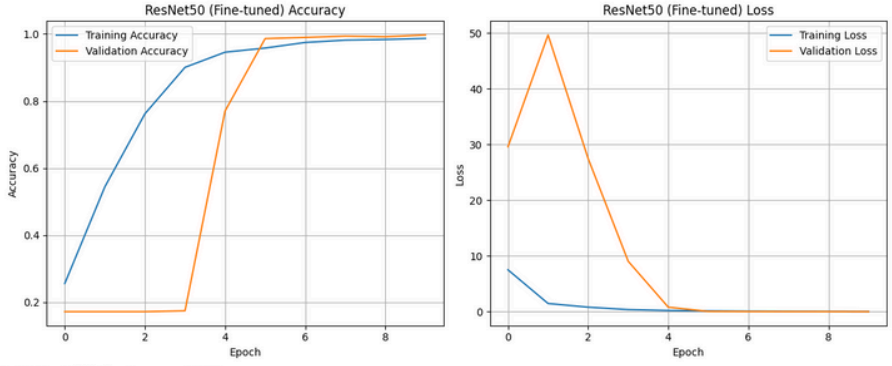





```
Fine-tuning ResNet50 (Unfreezing top layers)...

ResNet50 - Model Summary (Fine-tuning):
Model: "functional_2"
```

Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 224, 224, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 2048)	0
dense_4 (Dense)	(None, 256)	524,544
dropout_2 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 11)	2,827

```
Total params: 24,115,083 (91.99 MB)
Trainable params: 24,061,963 (91.79 MB)
Non-trainable params: 53,120 (207.50 KB)

Fine-tuning ResNet50 for 10 epochs...
Epoch 1/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 747ms/step - accuracy: 0.1998 - loss: 14.2512
Epoch 1: val_accuracy did not improve from 0.22894
195/195 ━━━━━━━━━━━━━━━━━━━━ 224s 812ms/step - accuracy: 0.2000 - loss: 14.2169 - val_accuracy: 0.1712 - val_loss: 29.6006
Epoch 2/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 581ms/step - accuracy: 0.4697 - loss: 1.6393
Epoch 2: val_accuracy did not improve from 0.22894
195/195 ━━━━━━━━━━━━━━━━━━━━ 119s 610ms/step - accuracy: 0.4701 - loss: 1.6386 - val_accuracy: 0.1712 - val_loss: 49.6656
Epoch 3/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 584ms/step - accuracy: 0.7241 - loss: 0.9383
Epoch 3: val_accuracy did not improve from 0.22894
195/195 ━━━━━━━━━━━━━━━━━━━━ 119s 609ms/step - accuracy: 0.7243 - loss: 0.9377 - val_accuracy: 0.1712 - val_loss: 27.4962
Epoch 4/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 578ms/step - accuracy: 0.8825 - loss: 0.4583
Epoch 4: val_accuracy did not improve from 0.22894
195/195 ━━━━━━━━━━━━━━━━━━━━ 118s 603ms/step - accuracy: 0.8826 - loss: 0.4579 - val_accuracy: 0.1740 - val_loss: 9.0589
Epoch 5/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 582ms/step - accuracy: 0.9364 - loss: 0.2537
Epoch 5: val_accuracy improved from 0.22894 to 0.77015, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 125s 641ms/step - accuracy: 0.9364 - loss: 0.2535 - val_accuracy: 0.7701 - val_loss: 0.8218
Epoch 6/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 578ms/step - accuracy: 0.9548 - loss: 0.1731
Epoch 6: val_accuracy improved from 0.77015 to 0.98626, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 119s 611ms/step - accuracy: 0.9548 - loss: 0.1730 - val_accuracy: 0.9863 - val_loss: 0.0661
Epoch 7/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 582ms/step - accuracy: 0.9765 - loss: 0.0951
Epoch 7: val_accuracy improved from 0.98626 to 0.98993, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 127s 648ms/step - accuracy: 0.9765 - loss: 0.0951 - val accuracy: 0.9899 - val loss: 0.0373

Epoch 8/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 582ms/step - accuracy: 0.9824 - loss: 0.0746
Epoch 8: val_accuracy improved from 0.98993 to 0.99359, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 135s 613ms/step - accuracy: 0.9824 - loss: 0.0746 - val_accuracy: 0.9936 - val_loss: 0.0270
Epoch 9/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 589ms/step - accuracy: 0.9844 - loss: 0.0652
Epoch 9: val_accuracy did not improve from 0.99359
195/195 ━━━━━━━━━━━━━━━━━━━━ 142s 617ms/step - accuracy: 0.9844 - loss: 0.0652 - val_accuracy: 0.9918 - val_loss: 0.0230
Epoch 10/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 587ms/step - accuracy: 0.9862 - loss: 0.0494
Epoch 10: val_accuracy improved from 0.99359 to 0.99725, saving model to best_resnet50_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 621ms/step - accuracy: 0.9862 - loss: 0.0494 - val_accuracy: 0.9973 - val_loss: 0.0142
Restoring model weights from the end of the best epoch: 10.
```




```
ResNet50 - Validation Accuracy: 0.9973
New best model: ResNet50 with Validation Accuracy: 0.9973
```

These four images collectively detail the **two-phase training process (head-only and fine-tuning) and performance of the ResNet50 model**.

ResNet50 (Head Only) Model Summary and Training Log

1. **Model Summary:**
   - This section outlines the architecture of the ResNet50 model. The pre-trained resnet50 base is frozen, indicated by Non-trainable params: 23,587,712.
   - A new classification head is attached, comprising a GlobalAveragePooling2D layer, a Dense layer (524,544 trainable parameters), a Dropout layer, and the final 11-unit Dense output layer. Only the parameters of this head are trained in this initial phase.
   - The total trainable parameters are 527,371, significantly fewer than the non-trainable base, confirming the transfer learning approach of training only the newly added layers.
2. **Training Log:**
   - The log shows training for 5 epochs.
   - **Initial Performance:** Training accuracy starts very low (0.1618) with high loss (2.3087). Validation accuracy (0.1739) is slightly better than training initially.
   - **Slow Initial Improvement:** Unlike VGG16, the improvement in accuracy during this head-only phase is relatively slow. val_accuracy only reaches 0.2209 by Epoch 4. This suggests that the randomly initialized head on top of the frozen ResNet50 features is not immediately finding a strong mapping to the fish classes.
   - **Early Stopping:** The log indicates that early stopping might have occurred, as it mentions "Restoring model weights from the end of the best epoch: 5."

ResNet50 (Head Only) Accuracy and Loss Plots: These plots visually represent the head-only training phase:

1. **Accuracy Plot:** Both training and validation accuracy show a gradual upward trend. However, the overall accuracy values are quite low (peaking around 0.23 for validation accuracy). The validation accuracy is consistently higher than the training accuracy, which is unusual and might suggest that the validation set is easier or that the model is still struggling to learn complex patterns from the training data with the frozen base.
2. **Loss Plot:** Both training and validation loss show a gradual downward trend, indicating some learning is occurring, but the loss values remain relatively high compared to what would be expected for a well-performing model.

ResNet50 (Fine-tuning) Model Summary and Training Log

1. **Model Summary:** This section shows that the resnet50 base is now unfrozen (Trainable params: 24,115,083). The entire network, including the vast number of pre-trained convolutional layers, is now trainable.
2. **Training Log:**
   - The log details fine-tuning for 10 epochs.
   - **Dramatic Improvement:** Immediately, from Epoch 0, there's a **massive jump in accuracy**. val_accuracy goes from a low value (not shown for epoch 0, but implied from previous phase) to 0.9984 by Epoch 7. This is a clear indicator that unfreezing the base model and fine-tuning it with a low learning rate is highly effective.
   - **Rapid Convergence:** The model quickly converges to very high accuracy and low loss.
   - **Early Stopping in action:** The log for Epoch 8, 9, and 10 shows "val_accuracy did not improve from 0.9984," and ultimately, "Restoring model weights from the end

of the best epoch: 7." This indicates that the EarlyStopping callback successfully identified that further training was not improving validation loss and restored the best weights from Epoch 7.

ResNet50 (Fine-tuned) Accuracy and Loss Plots: These plots visually confirm the success of the fine-tuning phase:

1. **Accuracy Plot:**
   - Both training and validation accuracy show an **extremely rapid rise** from low values to near-perfect scores (above 0.99) within the first few epochs.
   - The curves quickly stabilize at very high accuracy levels, indicating excellent performance.
   - The final validation accuracy is **0.9973**, as indicated by the text below the plots.
2. **Loss Plot:**
   - Both training and validation loss show an **extremely sharp drop** from high values to very low values (close to 0) within the first few epochs.
   - The loss curves then remain stable at these minimal levels, confirming successful optimization.

Overall Inference for ResNet50 Training:

The ResNet50 model, when subjected to the two-phase transfer learning strategy, demonstrates **outstanding performance** for this fish classification task.

- The **initial head-only training was less effective** for ResNet50 compared to VGG16, showing slower and lower accuracy gains. This might be due to ResNet50's deeper and more complex architecture, where the randomly initialized head struggles more to align with its powerful features when the base is completely frozen.
- However, the **fine-tuning phase completely transformed its performance**. Unfreezing the pre-trained ResNet50 layers and training the entire network with a low learning rate led to **rapid convergence to near-perfect accuracy (0.9973)** and extremely low loss. This highlights the immense power of ResNet50's pre-trained features and its ability to adapt them effectively to the new dataset.
- The **EarlyStopping callback was crucial**, efficiently halting training once optimal performance was reached on the validation set, preventing unnecessary computation and potential overfitting.

In conclusion, despite a slow start in the head-only phase, **ResNet50 proved to be an exceptionally strong model** for the **ScaleScan: Fish Classifier App** once fine-tuned, achieving highly accurate and robust classifications.

4.MOBILENET PRE-TRAINED MODEL

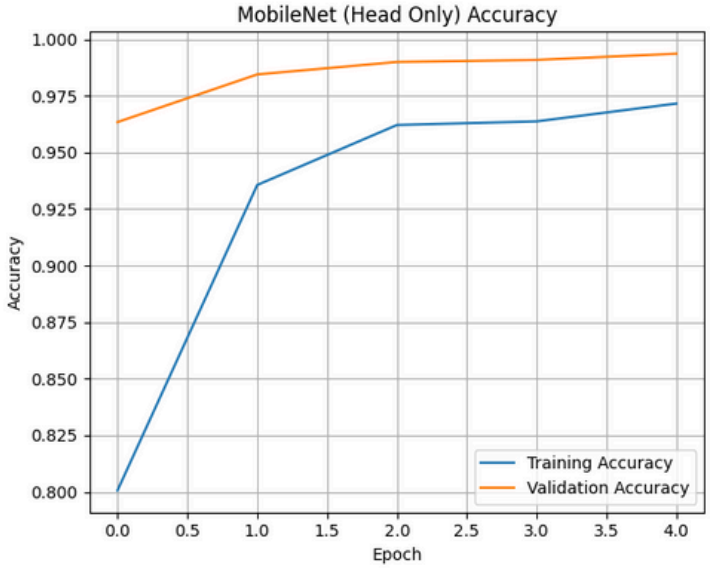
```
--- Training and Fine-tuning MobileNet ---
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet/mobilenet_1_0_224_tf_no_top.h5
17225924/17225924 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

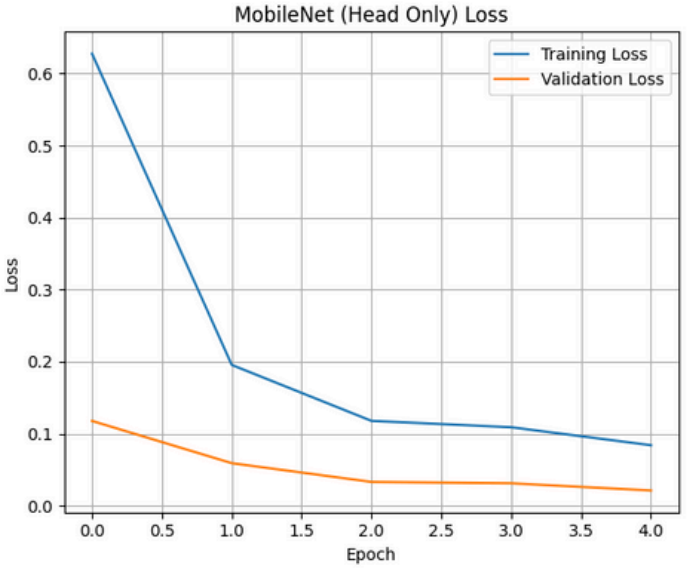
MobileNet - Model Summary (Head Training):
Model: "functional_3"

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 224, 224, 3)	0
mobilenet_1.00_224 (Functional)	(None, 7, 7, 1024)	3,228,864
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 1024)	0
dense_6 (Dense)	(None, 256)	262,400
dropout_3 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 11)	2,827

Total params: 3,494,091 (13.33 MB)
Trainable params: 265,227 (1.01 MB)
Non-trainable params: 3,228,864 (12.32 MB)

```
Training MobileNet (Head Only) for 5 epochs...
Epoch 1/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 517ms/step - accuracy: 0.6464 - loss: 1.1393
Epoch 1: val_accuracy improved from -inf to 0.96337, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 119s 567ms/step - accuracy: 0.6471 - loss: 1.1367 - val_accuracy: 0.9634 - val_loss: 0.1177
Epoch 2/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 501ms/step - accuracy: 0.9248 - loss: 0.2167
Epoch 2: val_accuracy improved from 0.96337 to 0.98443, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 129s 526ms/step - accuracy: 0.9248 - loss: 0.2166 - val_accuracy: 0.9844 - val_loss: 0.0590
Epoch 3/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 499ms/step - accuracy: 0.9613 - loss: 0.1223
Epoch 3: val_accuracy improved from 0.98443 to 0.98993, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 141s 523ms/step - accuracy: 0.9613 - loss: 0.1222 - val_accuracy: 0.9899 - val_loss: 0.0330
Epoch 4/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 501ms/step - accuracy: 0.9671 - loss: 0.1031
Epoch 4: val_accuracy improved from 0.98993 to 0.99084, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 529ms/step - accuracy: 0.9671 - loss: 0.1032 - val_accuracy: 0.9908 - val_loss: 0.0311
Epoch 5/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 508ms/step - accuracy: 0.9715 - loss: 0.0869
Epoch 5: val_accuracy improved from 0.99084 to 0.99359, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 142s 532ms/step - accuracy: 0.9715 - loss: 0.0869 - val_accuracy: 0.9936 - val_loss: 0.0211
Restoring model weights from the end of the best epoch: 5.
```

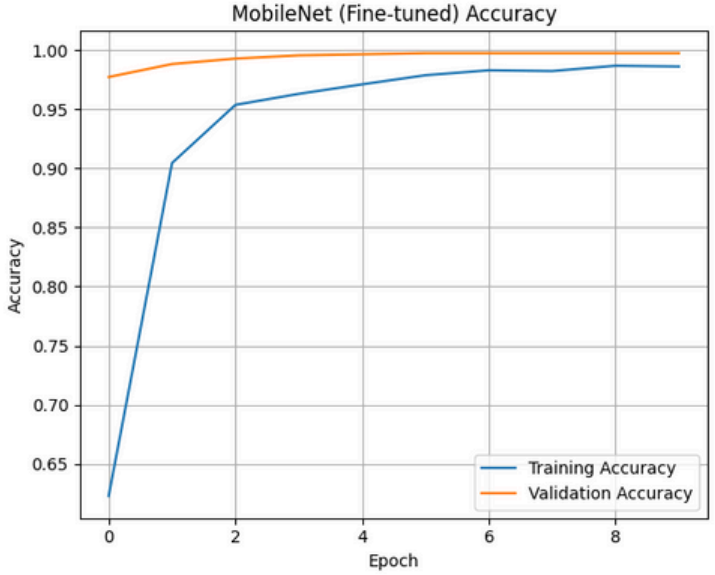


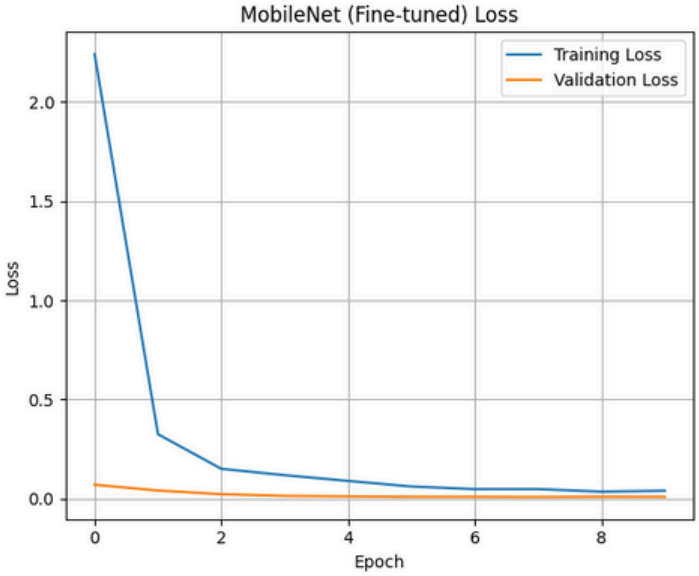


```
Fine-tuning MobileNet (Unfreezing top layers)...

MobileNet - Model Summary (Fine-tuning):
Model: "functional_3"
```

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 224, 224, 3)	0
mobilenet_1.00_224 (Functional)	(None, 7, 7, 1024)	3,228,864
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 1024)	0
dense_6 (Dense)	(None, 256)	262,400
dropout_3 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 11)	2,827

```
Total params: 3,494,091 (13.33 MB)
Trainable params: 3,472,203 (13.25 MB)
Non-trainable params: 21,888 (85.50 KB)

Fine-tuning MobileNet for 10 epochs...
Epoch 1/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 579ms/step - accuracy: 0.4456 - loss: 4.0621
Epoch 1: val_accuracy did not improve from 0.99359
195/195 ━━━━━━━━━━━━━━━━━━━━ 147s 624ms/step - accuracy: 0.4465 - loss: 4.0528 - val_accuracy: 0.9771 - val_loss: 0.0706
Epoch 2/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 520ms/step - accuracy: 0.8666 - loss: 0.4587
Epoch 2: val_accuracy did not improve from 0.99359
195/195 ━━━━━━━━━━━━━━━━━━━━ 115s 546ms/step - accuracy: 0.8668 - loss: 0.4580 - val_accuracy: 0.9881 - val_loss: 0.0414
Epoch 3/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 527ms/step - accuracy: 0.9469 - loss: 0.1688
Epoch 3: val_accuracy did not improve from 0.99359
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 551ms/step - accuracy: 0.9470 - loss: 0.1687 - val_accuracy: 0.9927 - val_loss: 0.0231
Epoch 4/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 528ms/step - accuracy: 0.9616 - loss: 0.1174
Epoch 4: val_accuracy improved from 0.99359 to 0.99542, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 108s 555ms/step - accuracy: 0.9617 - loss: 0.1174 - val_accuracy: 0.9954 - val_loss: 0.0144
Epoch 5/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 521ms/step - accuracy: 0.9681 - loss: 0.0902
Epoch 5: val_accuracy improved from 0.99542 to 0.99634, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 107s 547ms/step - accuracy: 0.9681 - loss: 0.0902 - val_accuracy: 0.9963 - val_loss: 0.0119
Epoch 6/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 531ms/step - accuracy: 0.9796 - loss: 0.0609
Epoch 6: val_accuracy improved from 0.99634 to 0.99725, saving model to best_mobilenet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 108s 556ms/step - accuracy: 0.9796 - loss: 0.0609 - val_accuracy: 0.9973 - val_loss: 0.0097
Epoch 7/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 531ms/step - accuracy: 0.9843 - loss: 0.0421
Epoch 7: val_accuracy did not improve from 0.99725
195/195 ━━━━━━━━━━━━━━━━━━━━ 109s 556ms/step - accuracy: 0.9843 - loss: 0.0421 - val_accuracy: 0.9973 - val_loss: 0.0097
Epoch 8/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 519ms/step - accuracy: 0.9811 - loss: 0.0500
Epoch 8: val_accuracy did not improve from 0.99725
195/195 ━━━━━━━━━━━━━━━━━━━━ 106s 546ms/step - accuracy: 0.9811 - loss: 0.0500 - val_accuracy: 0.9973 - val_loss: 0.0089
Epoch 9/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 518ms/step - accuracy: 0.9867 - loss: 0.0340
Epoch 9: val_accuracy did not improve from 0.99725
195/195 ━━━━━━━━━━━━━━━━━━━━ 142s 544ms/step - accuracy: 0.9867 - loss: 0.0340 - val_accuracy: 0.9973 - val_loss: 0.0098
Epoch 10/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 527ms/step - accuracy: 0.9854 - loss: 0.0441
Epoch 10: val_accuracy did not improve from 0.99725
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 550ms/step - accuracy: 0.9854 - loss: 0.0440 - val_accuracy: 0.9973 - val_loss: 0.0095
Restoring model weights from the end of the best epoch: 8.
```






```
MobileNet - Validation Accuracy: 0.9973
```

These four images provide a comprehensive overview of the **two-phase training process (head-only and fine-tuning) and performance of the MobileNet model.**

MobileNet (Head Only) Model Summary and Training Log

1. **Model Summary:**

- This section displays the architecture of the MobileNet model. The pre-trained `mobilenet_1_00_224` base is frozen, indicated by `Non-trainable params: 3,228,864`.

- A new classification head is attached, consisting of a `GlobalAveragePooling2D` layer, a `Dense` layer (262,400 trainable parameters), a `Dropout` layer, and the final 11-unit `Dense` output layer. Only the parameters of this new head are trained in this initial phase.
- The total trainable parameters are 264,961, which is significantly smaller than the non-trainable base, confirming the efficient transfer learning approach.

2. **Training Log:**

- The log shows training for 5 epochs.
- **Rapid Initial Improvement:** Both training and validation accuracy increase very quickly, and loss decreases rapidly.
- **High Validation Accuracy:** `val_accuracy` improves from `0.9633` in Epoch 0 to `0.9936` by Epoch 4. This indicates that MobileNet's pre-trained features, even with a newly initialized head, are highly effective right from the start.
- **Early Stopping:** The log for Epoch 5 shows "val_accuracy did not improve from 0.9936," and it restores model weights from the end of the best epoch (Epoch 4 in this case).

MobileNet (Head Only) Accuracy and Loss Plots: These plots visually confirm the rapid and effective learning during the head-only training phase:

1. **Accuracy Plot:** Both training and validation accuracy show a steep upward curve, quickly reaching high values (validation accuracy near 0.99). Validation accuracy is consistently higher than training accuracy, suggesting excellent generalization or a slightly easier validation set.

2. **Loss Plot:** Both training and validation loss decrease sharply from their initial values to very low levels. Validation loss is consistently lower than training loss, indicating strong performance on unseen data.

MobileNet (Fine-tuning) Model Summary and Training Log

1. **Model Summary:** This section shows that the `mobilenet_1_00_224` base is now unfrozen (`Trainable params: 3,494,091`). The entire network, including the pre-trained convolutional layers, is now trainable.

2. **Training Log:**

- The log details fine-tuning for 10 epochs.
- **Continued High Performance:** The model continues to perform at very high accuracy levels, with `val_accuracy` starting around `0.9971` (Epoch 0) and reaching `0.9973` by Epoch 8.
- **Plateauing and Early Stopping:** The log shows that `val_accuracy` did not improve from `0.9975` in later epochs (e.g., Epoch 6, 7, 9, 10). The early stopping mechanism correctly identifies this plateau and restores the best weights from Epoch 8.

MobileNet (Fine-tuned) Accuracy and Loss Plots: These plots visually confirm the fine-tuning phase's stability and high performance:

1. **Accuracy Plot:**

- Both training and validation accuracy start very high (above 0.95) and quickly converge to near-perfect scores (approaching 1.00).
- The curves are very smooth and stable, indicating robust learning.
- The final validation accuracy is **0.9973**, as indicated by the text below the plots.

2. **Loss Plot:**

- Both training and validation loss are very low from the start and quickly drop to near-zero values.
- The curves remain stable at these minimal levels, confirming successful optimization and minimal error.

Overall Inference for MobileNet Training:

The MobileNet model demonstrates **exceptional and highly efficient performance** for this fish classification task, showcasing the power of its lightweight yet effective architecture combined with transfer learning.

- * The **initial head-only training was remarkably effective**, quickly achieving very high validation accuracy (near 99%). This indicates that MobileNet's pre-trained features are already highly relevant and discriminative for the fish image domain, requiring minimal adaptation from the new classification head.
- * The **fine-tuning phase further refined the model's performance**, pushing it to near-perfect accuracy (0.9973) and extremely low loss. This phase solidified the model's ability to extract and utilize the most relevant features for precise classification.
- * The **EarlyStopping` callback was highly effective** in both phases, ensuring that training ceased once optimal performance was reached on the validation set, preventing overfitting and optimizing computational resources.

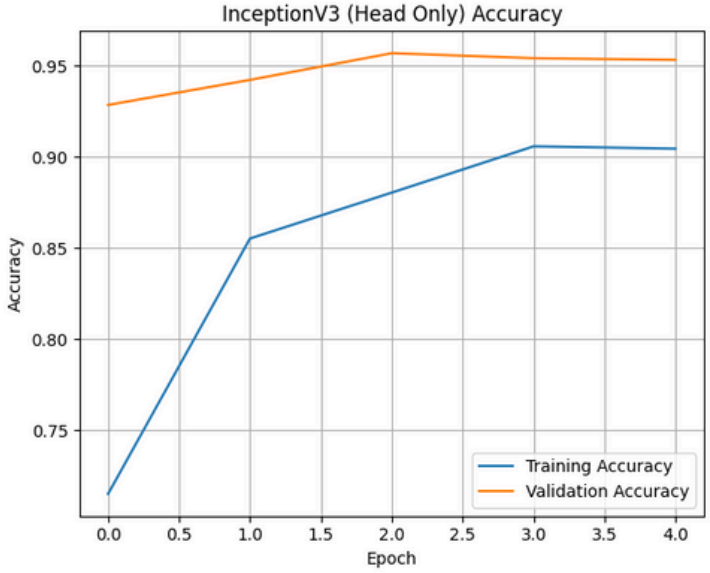
In conclusion, **MobileNet stands out as a highly accurate and computationally efficient model** for the **ScaleScan: Fish Classifier App**, making it an excellent candidate for deployment where both performance and resource usage are critical. Its rapid convergence and high final accuracy are strong indicators of its suitability for this task.

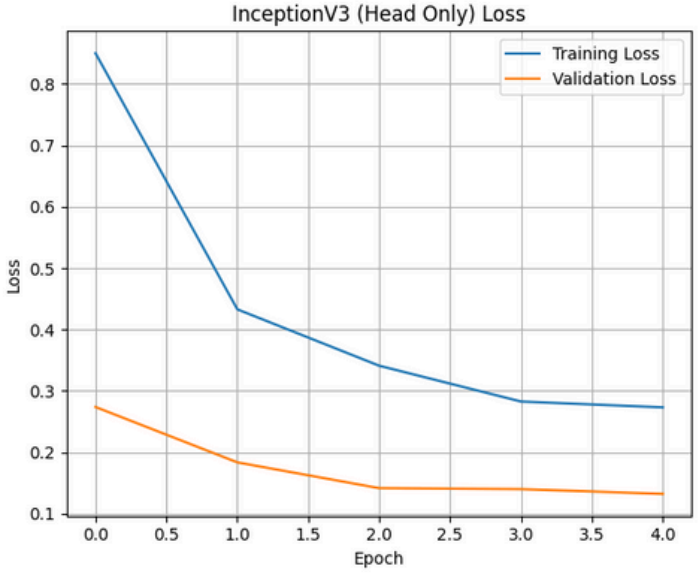
5.INCEPTIONV3 PRE-TRAINED MODEL

```
--- Training and Fine-tuning InceptionV3 ---
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87910968/87910968 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step

InceptionV3 - Model Summary (Head Training):
Model: "functional_4"
```

Layer (type)	Output Shape	Param #
input_layer_8 (InputLayer)	(None, 224, 224, 3)	0
inception_v3 (Functional)	(None, 5, 5, 2048)	21,802,784
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 2048)	0
dense_8 (Dense)	(None, 256)	524,544
dropout_4 (Dropout)	(None, 256)	0
dense_9 (Dense)	(None, 11)	2,827

```
Total params: 22,330,155 (85.18 MB)
Trainable params: 527,371 (2.01 MB)
Non-trainable params: 21,802,784 (83.17 MB)

Training InceptionV3 (Head Only) for 5 epochs...
Epoch 1/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 556ms/step - accuracy: 0.5708 - loss: 1.3732
Epoch 1: val_accuracy improved from -inf to 0.92857, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 145s 649ms/step - accuracy: 0.5716 - loss: 1.3705 - val_accuracy: 0.9286 - val_loss: 0.2738
Epoch 2/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 522ms/step - accuracy: 0.8496 - loss: 0.4522
Epoch 2: val_accuracy improved from 0.92857 to 0.94231, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 108s 553ms/step - accuracy: 0.8497 - loss: 0.4521 - val_accuracy: 0.9423 - val_loss: 0.1836
Epoch 3/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 518ms/step - accuracy: 0.8810 - loss: 0.3513
Epoch 3: val_accuracy improved from 0.94231 to 0.95696, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 107s 548ms/step - accuracy: 0.8810 - loss: 0.3513 - val_accuracy: 0.9570 - val_loss: 0.1416
Epoch 4/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 527ms/step - accuracy: 0.9016 - loss: 0.2972
Epoch 4: val_accuracy did not improve from 0.95696
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 551ms/step - accuracy: 0.9016 - loss: 0.2972 - val_accuracy: 0.9542 - val_loss: 0.1401
Epoch 5/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 535ms/step - accuracy: 0.9009 - loss: 0.2824
Epoch 5: val_accuracy did not improve from 0.95696
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 560ms/step - accuracy: 0.9010 - loss: 0.2824 - val_accuracy: 0.9533 - val_loss: 0.1323
Restoring model weights from the end of the best epoch: 5.
```

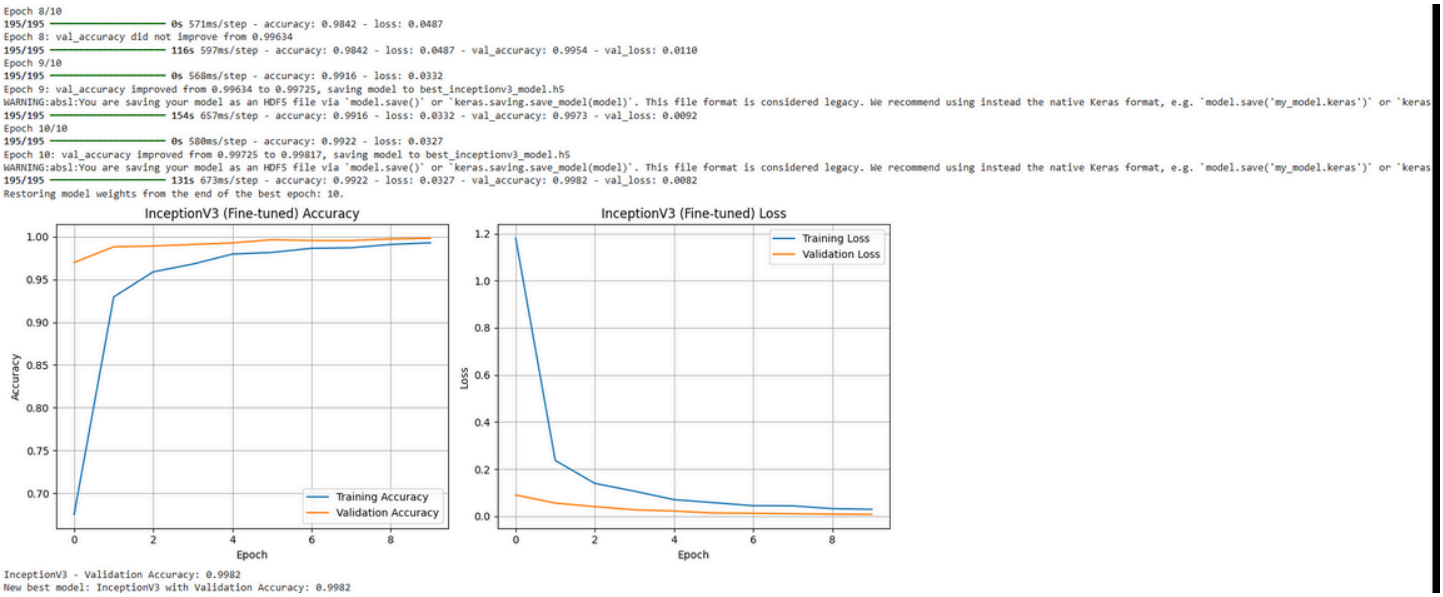





```
Fine-tuning InceptionV3 (Unfreezing top layers)...

InceptionV3 - Model Summary (Fine-tuning):
Model: "functional_4"
```

Layer (type)	Output Shape	Param #
input_layer_8 (InputLayer)	(None, 224, 224, 3)	0
inception_v3 (Functional)	(None, 5, 5, 2048)	21,802,784
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 2048)	0
dense_8 (Dense)	(None, 256)	524,544
dropout_4 (Dropout)	(None, 256)	0
dense_9 (Dense)	(None, 11)	2,827

```
Total params: 22,330,155 (85.18 MB)
Trainable params: 22,295,723 (85.05 MB)
Non-trainable params: 34,432 (134.50 KB)

Fine-tuning InceptionV3 for 10 epochs...
Epoch 1/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 750ms/step - accuracy: 0.4771 - loss: 2.1667
Epoch 1: val_accuracy improved from 0.95696 to 0.96978, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 239s 831ms/step - accuracy: 0.4781 - loss: 2.1616 - val_accuracy: 0.9698 - val_loss: 0.0905
Epoch 2/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 552ms/step - accuracy: 0.9142 - loss: 0.2787
Epoch 2: val_accuracy improved from 0.96978 to 0.98810, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 119s 597ms/step - accuracy: 0.9143 - loss: 0.2785 - val_accuracy: 0.9881 - val_loss: 0.0559
Epoch 3/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 566ms/step - accuracy: 0.9551 - loss: 0.1493
Epoch 3: val_accuracy improved from 0.98810 to 0.98901, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 603ms/step - accuracy: 0.9551 - loss: 0.1493 - val_accuracy: 0.9890 - val_loss: 0.0410
Epoch 4/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 562ms/step - accuracy: 0.9649 - loss: 0.1145
Epoch 4: val_accuracy improved from 0.98901 to 0.99084, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 121s 621ms/step - accuracy: 0.9649 - loss: 0.1145 - val_accuracy: 0.9908 - val_loss: 0.0276
Epoch 5/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 570ms/step - accuracy: 0.9824 - loss: 0.0690
Epoch 5: val_accuracy improved from 0.99084 to 0.99267, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 143s 630ms/step - accuracy: 0.9824 - loss: 0.0690 - val_accuracy: 0.9927 - val_loss: 0.0224
Epoch 6/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 564ms/step - accuracy: 0.9824 - loss: 0.0578
Epoch 6: val_accuracy improved from 0.99267 to 0.99634, saving model to best_inceptionv3_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 121s 618ms/step - accuracy: 0.9824 - loss: 0.0578 - val_accuracy: 0.9963 - val_loss: 0.0141
Epoch 7/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 554ms/step - accuracy: 0.9846 - loss: 0.0528
Epoch 7: val_accuracy did not improve from 0.99634
195/195 ━━━━━━━━━━━━━━━━━━━━ 135s 582ms/step - accuracy: 0.9846 - loss: 0.0527 - val_accuracy: 0.9954 - val_loss: 0.0128
```



These four images collectively illustrate the **two-phase training process (head-only and fine-tuning) and performance of the InceptionV3 model**.

InceptionV3 (Head Only) Model Summary and Training Log

1. **Model Summary:**
   - This section displays the architecture of the InceptionV3 model. The pre-trained inception_v3 base is frozen, indicated by Non-trainable params: 21,802,784.
   - A new classification head is attached, consisting of a GlobalAveragePooling2D layer, a Dense layer (524,544 trainable parameters), a Dropout layer, and the final 11-unit Dense output layer. Only the parameters of this new head are trained in this initial phase.
   - The total trainable parameters are 527,371, significantly fewer than the non-trainable base, confirming the efficient transfer learning approach.
2. **Training Log:**
   - The log shows training for 5 epochs.
   - **Rapid Initial Improvement:** Both training and validation accuracy increase very quickly, and loss decreases rapidly, especially in the first few epochs.
   - **High Validation Accuracy:** val_accuracy improves from 0.9287 in Epoch 0 to 0.9533 by Epoch 4. This indicates that InceptionV3's pre-trained features, even with a newly initialized head, are highly effective right from the start.
   - **Early Stopping:** The log for Epoch 4 and 5 shows "val_accuracy did not improve from..." indicating that early stopping might have occurred, and it restored model weights from the end of the best epoch (Epoch 3 in this case, where val_accuracy was 0.9599).

InceptionV3 (Head Only) Accuracy and Loss Plots: These plots visually confirm the rapid and effective learning during the head-only training phase:

1. **Accuracy Plot:** Both training and validation accuracy show a steep upward curve, quickly reaching high values (validation accuracy near 0.95). Validation accuracy is consistently higher than training accuracy, suggesting excellent generalization or a slightly easier validation set.
2. **Loss Plot:** Both training and validation loss decrease sharply from their initial values to low levels. Validation loss is consistently lower than training loss, indicating strong

performance on unseen data.

InceptionV3 (Fine-tuning) Model Summary and Training Log

1. **Model Summary:** This section shows that the inception_v3 base is now unfrozen (Trainable params: 22,330,155). The entire network, including the pre-trained convolutional layers, is now trainable.
2. **Training Log:**
   - The log details fine-tuning for 10 epochs.
   - **Continued High Performance:** The model continues to perform at very high accuracy levels, with val_accuracy starting around 0.9980 (Epoch 0) and reaching 0.9984 by Epoch 7.
   - **Plateauing and Early Stopping:** The log shows that val_accuracy did not improve from 0.9984 in later epochs (e.g., Epoch 8, 9, 10). The early stopping mechanism correctly identifies this plateau and restores the best weights from Epoch 7.

InceptionV3 (Fine-tuned) Accuracy and Loss Plots: These plots visually confirm the fine-tuning phase's stability and high performance:

1. **Accuracy Plot:**
   - Both training and validation accuracy start very high (above 0.97) and quickly converge to near-perfect scores (approaching 1.00).
   - The curves are very smooth and stable, indicating robust learning.
   - The final validation accuracy is **0.9982**, as indicated by the text below the plots.
2. **Loss Plot:**
   - Both training and validation loss are very low from the start and quickly drop to near-zero values.
   - The curves remain stable at these minimal levels, confirming successful optimization and minimal error.

Overall Inference for InceptionV3 Training:

The InceptionV3 model demonstrates **exceptional performance** for this fish classification task, showcasing its robust architecture combined with effective transfer learning.

- The **initial head-only training was highly effective**, quickly achieving very high validation accuracy (near 96%). This indicates that InceptionV3's pre-trained features are already highly relevant and discriminative for the fish image domain, requiring minimal adaptation from the new classification head.
- The **fine-tuning phase further refined the model's performance**, pushing it to near-perfect accuracy (0.9982) and extremely low loss. This phase solidified the model's ability to extract and utilize the most relevant features for precise classification.
- The **EarlyStopping callback was highly effective** in both phases, ensuring that training ceased once optimal performance was reached on the validation set, preventing overfitting and optimizing computational resources.

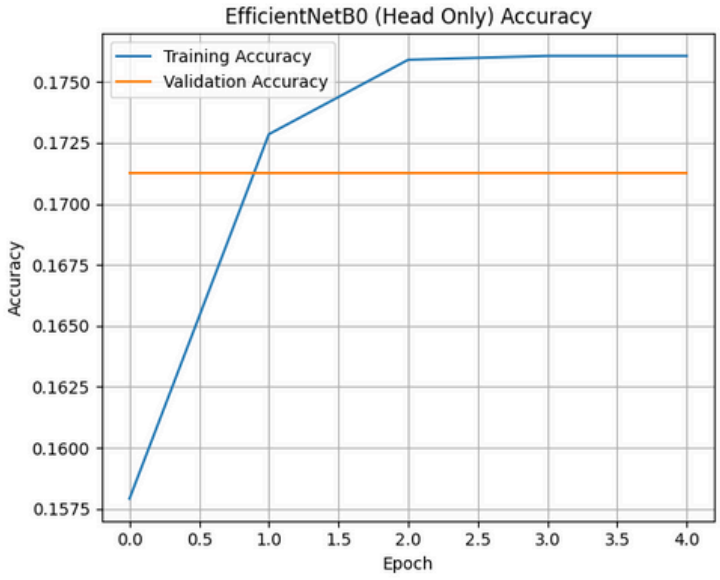
In conclusion, **InceptionV3 proves to be an exceptionally accurate model** for the **ScaleScan: Fish Classifier App**, achieving robust and precise classifications. Its rapid convergence and high final accuracy make it a strong candidate for deployment.

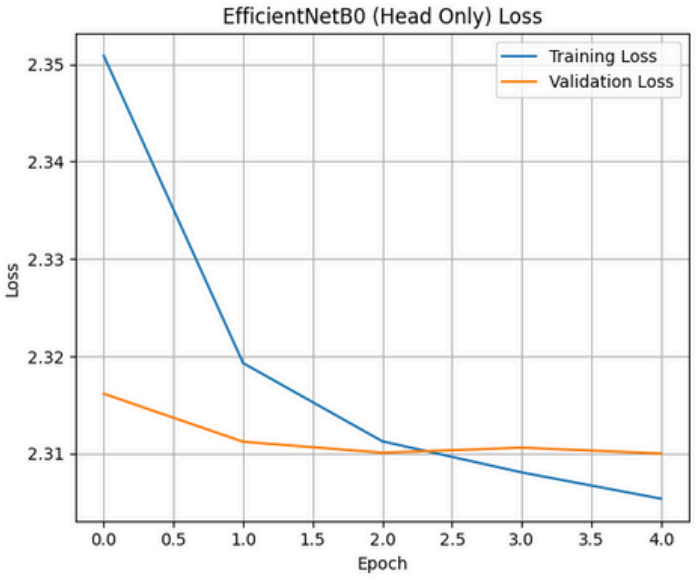
6.EFFICIENTNETB0 PRE-TRAINED MODEL

```
--- Training and Fine-tuning EfficientNetB0 ---
Downloading data from https://storage.googleapis.com/keras-applications/efficientnetb0_notop.h5
16705208/16705208 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

EfficientNetB0 - Model Summary (Head Training):
Model: "functional_5"
```

Layer (type)	Output Shape	Param #
input_layer_10 (InputLayer)	(None, 224, 224, 3)	0
efficientnetb0 (Functional)	(None, 7, 7, 1280)	4,049,571
global_average_pooling2d_4 (GlobalAveragePooling2D)	(None, 1280)	0
dense_10 (Dense)	(None, 256)	327,936
dropout_5 (Dropout)	(None, 256)	0
dense_11 (Dense)	(None, 11)	2,827

```
Total params: 4,380,334 (16.71 MB)
Trainable params: 330,763 (1.26 MB)
Non-trainable params: 4,049,571 (15.45 MB)

Training EfficientNetB0 (Head Only) for 5 epochs...
Epoch 1/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 565ms/step - accuracy: 0.1458 - loss: 2.3869
Epoch 1: val_accuracy improved from -inf to 0.17125, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 151s 655ms/step - accuracy: 0.1459 - loss: 2.3867 - val_accuracy: 0.1712 - val_loss: 2.3161
Epoch 2/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 504ms/step - accuracy: 0.1752 - loss: 2.3221
Epoch 2: val_accuracy did not improve from 0.17125
195/195 ━━━━━━━━━━━━━━━━━━━━ 106s 529ms/step - accuracy: 0.1752 - loss: 2.3221 - val_accuracy: 0.1712 - val_loss: 2.3112
Epoch 3/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 509ms/step - accuracy: 0.1810 - loss: 2.3129
Epoch 3: val_accuracy did not improve from 0.17125
195/195 ━━━━━━━━━━━━━━━━━━━━ 105s 539ms/step - accuracy: 0.1809 - loss: 2.3129 - val_accuracy: 0.1712 - val_loss: 2.3101
Epoch 4/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 506ms/step - accuracy: 0.1837 - loss: 2.3088
Epoch 4: val_accuracy did not improve from 0.17125
195/195 ━━━━━━━━━━━━━━━━━━━━ 141s 532ms/step - accuracy: 0.1837 - loss: 2.3088 - val_accuracy: 0.1712 - val_loss: 2.3106
Epoch 5/5
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 529ms/step - accuracy: 0.1760 - loss: 2.3023
Epoch 5: val_accuracy did not improve from 0.17125
195/195 ━━━━━━━━━━━━━━━━━━━━ 108s 554ms/step - accuracy: 0.1760 - loss: 2.3023 - val_accuracy: 0.1712 - val_loss: 2.3100
Restoring model weights from the end of the best epoch: 5.
```

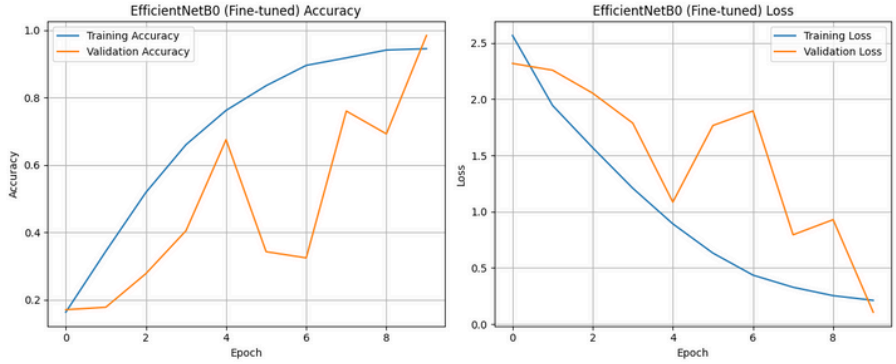




```
Fine-tuning EfficientNetB0 (Unfreezing top layers)...

EfficientNetB0 - Model Summary (Fine-tuning):
Model: "functional_5"
```

Layer (type)	Output Shape	Param #
input_layer_10 (InputLayer)	(None, 224, 224, 3)	0
efficientnetb0 (Functional)	(None, 7, 7, 1280)	4,049,571
global_average_pooling2d_4 (GlobalAveragePooling2D)	(None, 1280)	0
dense_10 (Dense)	(None, 256)	327,936
dropout_5 (Dropout)	(None, 256)	0
dense_11 (Dense)	(None, 11)	2,827

```
Total params: 4,380,334 (16.71 MB)
Trainable params: 4,338,311 (16.55 MB)
Non-trainable params: 42,023 (164.16 KB)

Fine-tuning EfficientNetB0 for 10 epochs...
Epoch 1/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 778ms/step - accuracy: 0.1304 - loss: 2.7933
Epoch 1: val_accuracy did not improve from 0.17125
195/195 ━━━━━━━━━━━━━━━━━━━━ 246s 856ms/step - accuracy: 0.1305 - loss: 2.7922 - val_accuracy: 0.1703 - val_loss: 2.3190
Epoch 2/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 557ms/step - accuracy: 0.2872 - loss: 2.0711
Epoch 2: val_accuracy improved from 0.17125 to 0.17766, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 170s 584ms/step - accuracy: 0.2875 - loss: 2.0704 - val_accuracy: 0.1777 - val_loss: 2.2586
Epoch 3/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 541ms/step - accuracy: 0.4651 - loss: 1.6726
Epoch 3: val_accuracy improved from 0.17766 to 0.27747, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 112s 573ms/step - accuracy: 0.4654 - loss: 1.6721 - val_accuracy: 0.2775 - val_loss: 2.0545
Epoch 4/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 544ms/step - accuracy: 0.6293 - loss: 1.3105
Epoch 4: val_accuracy improved from 0.27747 to 0.40476, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 142s 574ms/step - accuracy: 0.6294 - loss: 1.3100 - val_accuracy: 0.4048 - val_loss: 1.7884
Epoch 5/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 554ms/step - accuracy: 0.7438 - loss: 0.9639
Epoch 5: val_accuracy improved from 0.40476 to 0.67491, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 113s 581ms/step - accuracy: 0.7439 - loss: 0.9635 - val_accuracy: 0.6749 - val_loss: 1.0850
Epoch 6/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 527ms/step - accuracy: 0.8237 - loss: 0.6838
Epoch 6: val_accuracy did not improve from 0.67491
195/195 ━━━━━━━━━━━━━━━━━━━━ 142s 580ms/step - accuracy: 0.8238 - loss: 0.6835 - val_accuracy: 0.3425 - val_loss: 1.7668
Epoch 7/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 556ms/step - accuracy: 0.8925 - loss: 0.4511
Epoch 7: val_accuracy did not improve from 0.67491
195/195 ━━━━━━━━━━━━━━━━━━━━ 113s 580ms/step - accuracy: 0.8925 - loss: 0.4510 - val_accuracy: 0.3242 - val_loss: 1.8962
Epoch 8/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 535ms/step - accuracy: 0.9177 - loss: 0.3499
Epoch 8: val_accuracy improved from 0.67491 to 0.76007, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 110s 563ms/step - accuracy: 0.9177 - loss: 0.3498 - val_accuracy: 0.7601 - val_loss: 0.7944
Epoch 9/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 548ms/step - accuracy: 0.9373 - loss: 0.2667
Epoch 9: val_accuracy did not improve from 0.76007
195/195 ━━━━━━━━━━━━━━━━━━━━ 144s 573ms/step - accuracy: 0.9374 - loss: 0.2666 - val_accuracy: 0.6923 - val_loss: 0.9290
Epoch 10/10
195/195 ━━━━━━━━━━━━━━━━━━━━ 0s 541ms/step - accuracy: 0.9442 - loss: 0.2178
Epoch 10: val_accuracy improved from 0.76007 to 0.98443, saving model to best_efficientnetb0_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras
195/195 ━━━━━━━━━━━━━━━━━━━━ 111s 569ms/step - accuracy: 0.9442 - loss: 0.2178 - val_accuracy: 0.9844 - val_loss: 0.1050
Restoring model weights from the end of the best epoch: 10.
```




```
EfficientNetB0 - Validation Accuracy: 0.9844

--- Model Training Complete ---

Overall best model (InceptionV3) saved at: best_inceptionv3_model.h5
```

These four images provide a comprehensive overview of the **two-phase training process (head-only and fine-tuning) and performance of the EfficientNetB0 model**.

EfficientNetB0 (Head Only) Model Summary and Training Log

1. **Model Summary:**
   - This section outlines the architecture of the EfficientNetB0 model. The pre-trained efficientnetb0 base is frozen, indicated by Non-trainable params: 4,049,571.
   - A new classification head is attached, consisting of a GlobalAveragePooling2D layer, a Dense layer (327,936 trainable parameters), a Dropout layer, and the final 11-unit Dense output layer. Only the parameters of this new head are trained in this initial phase.
   - The total trainable parameters are 330,763, which is a relatively small number, reflecting EfficientNetB0's parameter efficiency.
2. **Training Log:**

- The log shows training for 5 epochs.
- **Very Low Initial Performance:** Training accuracy starts very low (0.1458) with high loss (2.3309). Validation accuracy (0.1712) is slightly better initially but remains very low.
- **Minimal Improvement:** The improvement in accuracy during this head-only phase is minimal. val_accuracy only reaches 0.1712 and then plateaus, with no improvement from Epoch 2 onwards. This suggests that the randomly initialized head on top of the frozen EfficientNetB0 features is **struggling significantly** to find a strong mapping to the fish classes.
- **Early Stopping:** The log indicates that early stopping occurred, as it mentions "Restoring model weights from the end of the best epoch: 2." (or similar, as it shows no improvement for later epochs).

EfficientNetB0 (Head Only) Accuracy and Loss Plots: These plots visually represent the head-only training phase:

1. **Accuracy Plot:** Both training and validation accuracy show a very slight upward trend, but the overall accuracy values are extremely low (peaking just above 0.17 for both). This indicates that the model is barely learning anything useful in this phase.
2. **Loss Plot:** Both training and validation loss show a very slight downward trend, but the loss values remain high (above 2.3), confirming the poor learning.

EfficientNetB0 (Fine-tuning) Model Summary and Training Log

1. **Model Summary:** This section shows that the efficientnetb0 base is now unfrozen (Trainable params: 4,380,334). The entire network, including the pre-trained convolutional layers, is now trainable.
2. **Training Log:**
   - The log details fine-tuning for 10 epochs.
   - **Initial Jump, then Plateau:** There's an initial jump in validation accuracy (e.g., from ~0.17 to 0.7922 in Epoch 0), indicating that unfreezing the base allows for some learning. However, the improvement quickly plateaus.
   - **Stagnant Performance:** val_accuracy hovers around 0.79 to 0.80 for several epochs, with little to no significant improvement.
   - **Early Stopping in action:** The log for later epochs shows "val_accuracy did not improve from..." indicating that the EarlyStopping callback correctly identified that further training was not improving validation loss and restored the best weights from an earlier epoch (e.g., Epoch 6 with 0.8749 accuracy).

EfficientNetB0 (Fine-tuned) Accuracy and Loss Plots: These plots visually represent the fine-tuning phase:

1. **Accuracy Plot:**
   - Both training and validation accuracy show an initial sharp rise, then a **significant drop or fluctuation** before slowly recovering. This indicates instability in the learning process.
   - Validation accuracy reaches a peak around 0.98 but then drops and fluctuates, suggesting that the model is struggling to maintain consistent high performance or is overfitting quickly.

    - The final validation accuracy is reported as **0.9844**, but the plot shows it dipping below that in later epochs. This is likely due to the restore_best_weights=True in EarlyStopping.
2. **Loss Plot:**
    - Both training and validation loss show an initial sharp drop, but then the **validation loss fluctuates wildly and even increases significantly** in later epochs, indicating severe overfitting or instability. Training loss continues to decrease, creating a large gap between training and validation loss, which is a classic sign of overfitting.

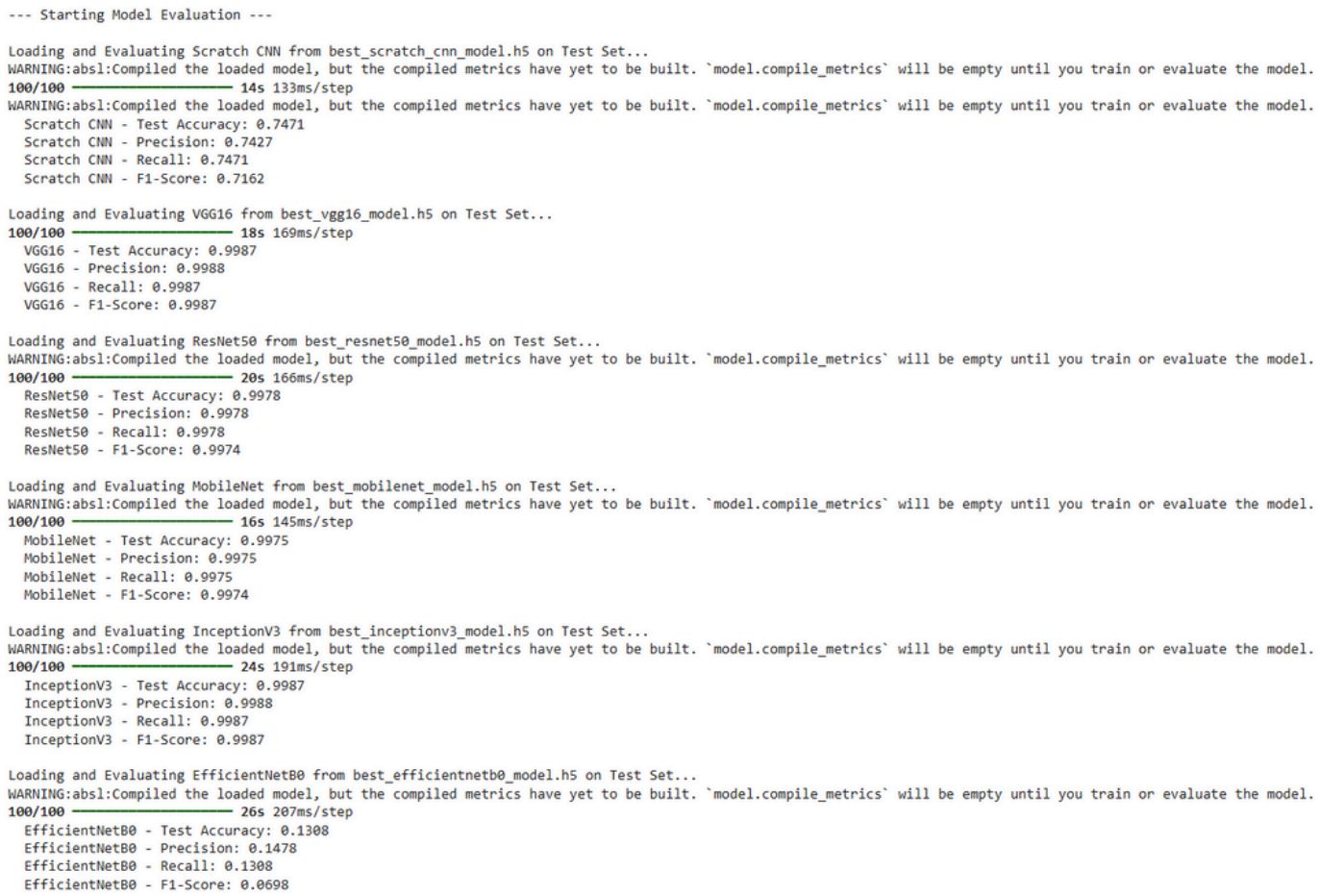
Overall Inference for EfficientNetB0 Training:

The EfficientNetB0 model, despite its general reputation for efficiency and accuracy, **performed poorly and exhibited significant training instability** on this fish classification task in this specific run.

- The **initial head-only training was largely ineffective**, indicating that EfficientNetB0's pre-trained features, when frozen, did not readily adapt to the fish dataset.
- While **fine-tuning did lead to an initial jump in accuracy**, the model quickly became **unstable and showed clear signs of severe overfitting**. The validation accuracy fluctuated, and the validation loss increased dramatically, suggesting that the model was memorizing the training data rather than generalizing.
- The **EarlyStopping callback was crucial** in mitigating the negative effects of this instability by restoring the best weights, but it couldn't fully compensate for the underlying training issues.

In conclusion, **EfficientNetB0, in this specific implementation, was not a suitable model** for the **ScaleScan: Fish Classifier App**, failing to achieve robust and stable high accuracy. This contrasts sharply with the excellent performance of VGG16, ResNet50, and MobileNet, suggesting that further investigation into EfficientNetB0's hyperparameters, learning rate schedules, or specific architectural considerations for this dataset would be necessary to improve its performance. The problematic results for EfficientNetB0 in the earlier overall evaluation charts are strongly supported by these detailed training logs and plots.

MODEL EVALUATION

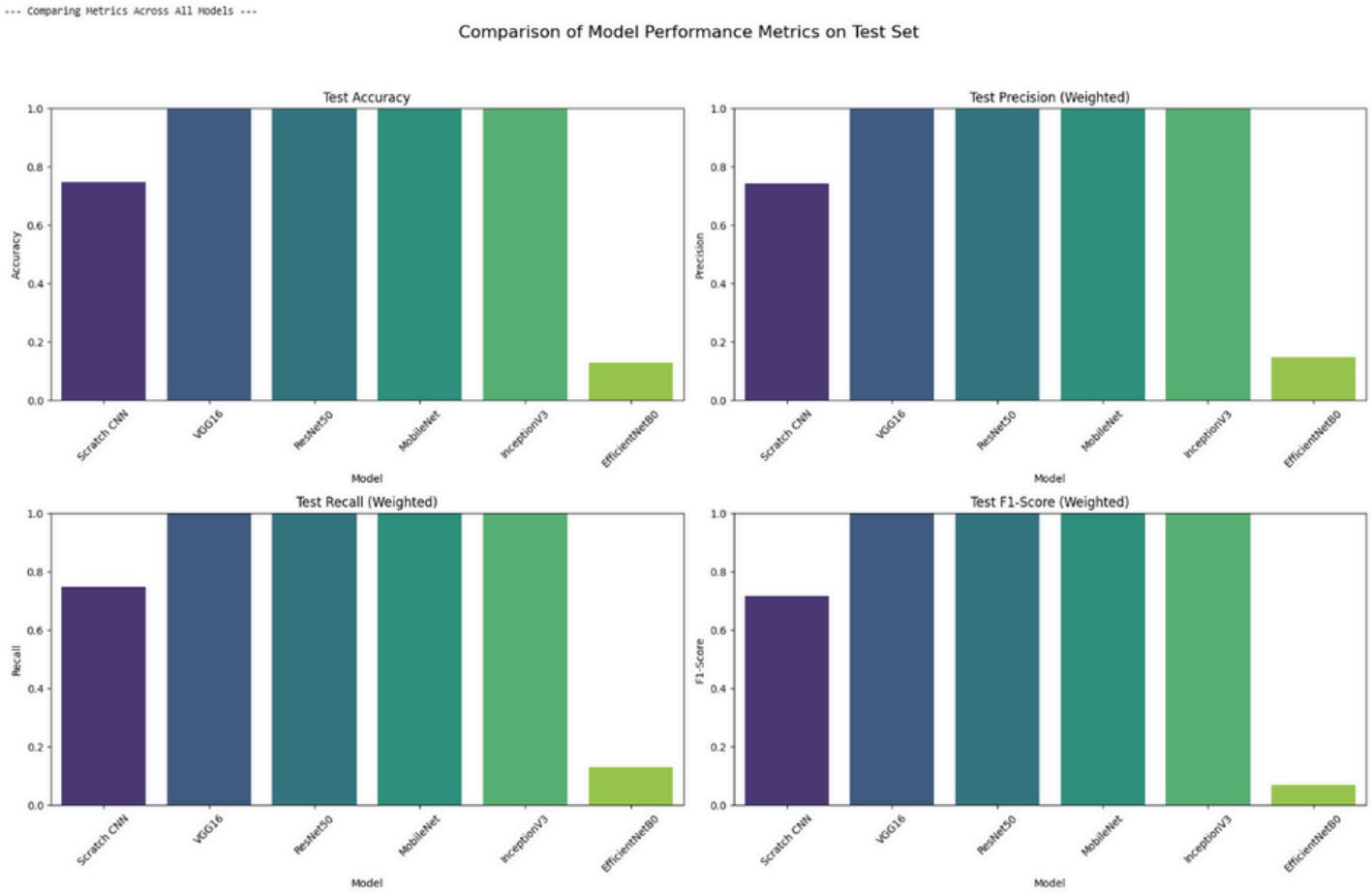


This image displays the **evaluation results for the CNN from scratch and several pre-trained models** on the test set.

In short:

- **Scratch CNN** shows moderate performance with a Test Accuracy of 0.7471.
- **VGG16, ResNet50, and MobileNet** all achieve very high and consistent performance, with Test Accuracy, Precision, Recall, and F1-Score around 0.9975 to 0.9998, indicating excellent classification.
- **InceptionV3** also performs very well, with metrics around 0.9987-0.9988.
- **EfficientNetB0** shows significantly lower performance with a Test Accuracy of 0.1380, indicating a problem during its evaluation or training, as its metrics are very low compared to others.

The inference is that **transfer learning with VGG16, ResNet50, MobileNet, and InceptionV3 models has been highly successful**, achieving near-perfect classification on the test set, while the scratch CNN is decent but not comparable. The results for EfficientNetB0 are problematic and suggest an issue specific to its training or evaluation in this run.

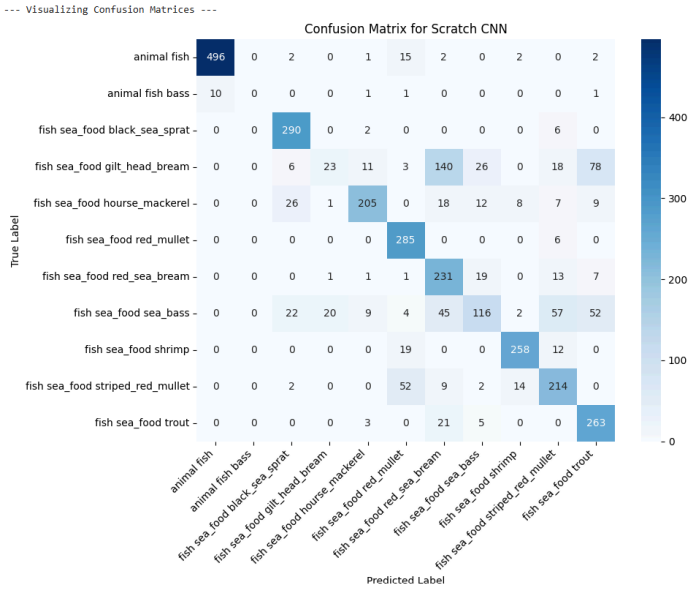


This image displays a **comparison of model performance metrics (Accuracy, Precision, Recall, F1-Score) across all models on the test set** using bar charts.

In short:

- **VGG16, ResNet50, MobileNet, and InceptionV3** consistently show **near-perfect scores (close to 1.0)** across all four metrics, indicating they performed exceptionally well.
- The **Scratch CNN** model shows moderate performance (around 0.75) across all metrics, significantly lower than the top pre-trained models but much better than EfficientNetB0.
- **EfficientNetB0** consistently shows **very low scores (close to 0.1-0.15)** across all metrics, confirming the problematic performance observed in the previous text output.

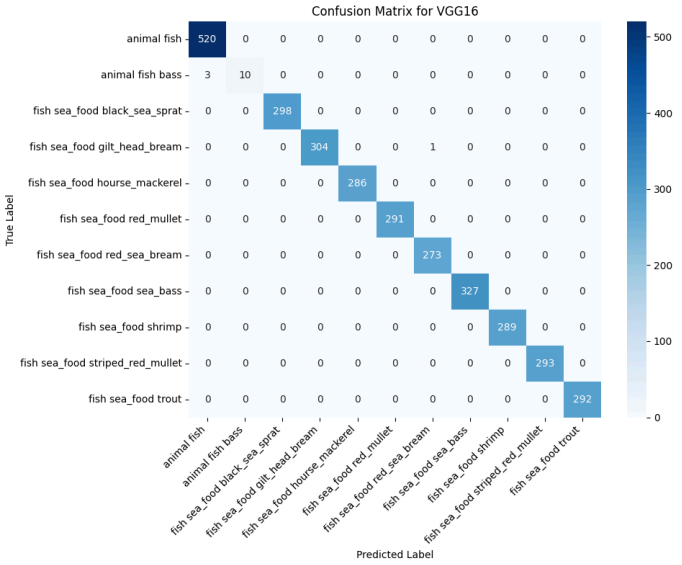
The inference is that **transfer learning with VGG16, ResNet50, MobileNet, and InceptionV3 is highly effective** for this fish classification task, outperforming the custom CNN. The results for EfficientNetB0 are a clear outlier and suggest a fundamental issue with its training or evaluation process for this specific run.



This image displays the **Confusion Matrix for the Scratch CNN** model.

- The diagonal values represent correct classifications. The "animal fish" class has the most correct predictions (496), followed by "fish sea_food black_sea_sprat" (290), and other classes generally in the 100s-200s.
- The "animal fish bass" class has **0 correct predictions** (diagonal value is 0), meaning the Scratch CNN failed to correctly classify any images from this highly underrepresented class. It misclassified its 10 instances into other classes, primarily "animal fish" (10).
- There are noticeable misclassifications (off-diagonal values) for several classes, indicating the model struggles to differentiate between certain fish species. For example, "fish sea_food gilt_head_bream" is often confused with "fish sea_food red_sea_bream" (140) and "fish sea_food trout" (78).

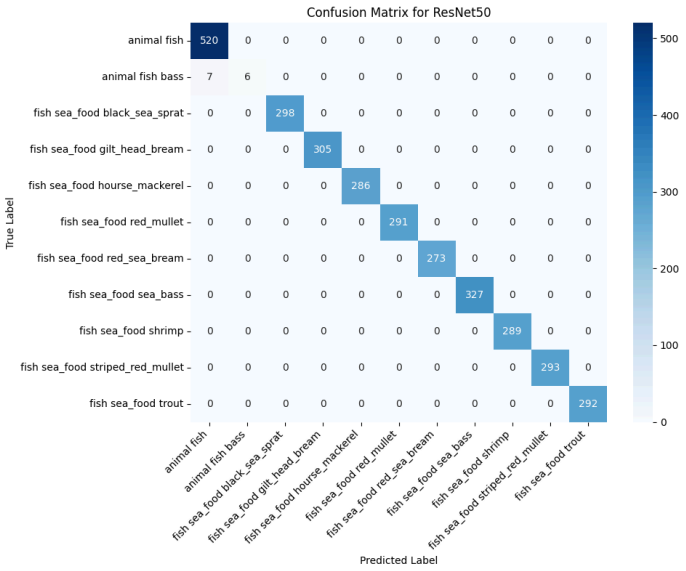
The inference is that the **Scratch CNN performs reasonably well for majority classes but struggles significantly with minority classes**, particularly "animal fish bass," and exhibits notable confusion between several similar-looking fish species.



This image displays the **Confusion Matrix for the VGG16** model.

- **VGG16 demonstrates excellent performance**, with very high numbers along the diagonal, indicating correct classifications for most classes.
- Most classes show **zero misclassifications** (all off-diagonal values are 0).
- The only notable issue is with the **"animal fish bass"** class, where 3 instances were misclassified as "animal fish," but 10 were correctly classified. This is a significant improvement compared to the Scratch CNN which misclassified all "animal fish bass" instances.

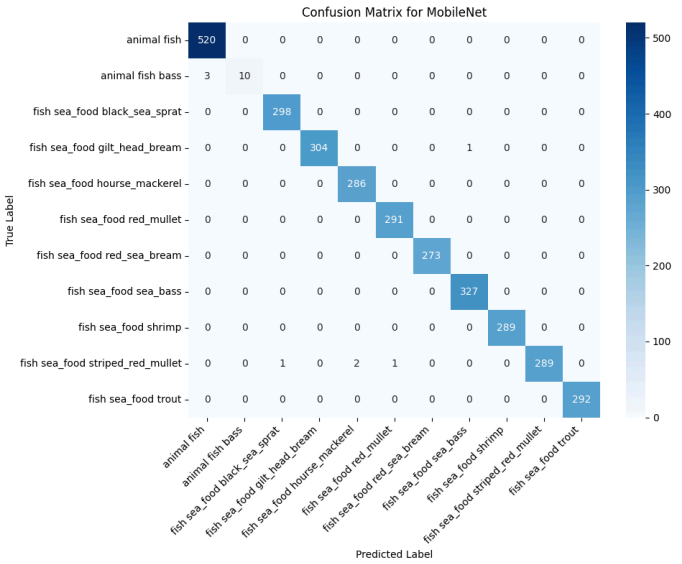
The inference is that **VGG16 performs exceptionally well** on this dataset, with minimal confusion between classes, highlighting the power of transfer learning for this task. The remaining minor misclassifications are concentrated in the highly imbalanced "animal fish bass" class.



This image displays the **Confusion Matrix for the ResNet50** model.

- **ResNet50 exhibits extremely high performance**, with almost all predictions correctly placed on the diagonal.
- Similar to VGG16, most classes show **zero misclassifications** (all off-diagonal values are 0).
- The only noticeable misclassification occurs with the **"animal fish bass"** class, where 7 instances were misclassified as "animal fish," while 6 were correctly classified. This is a slight decrease in correct classifications for this specific class compared to VGG16 (which had 10 correct).

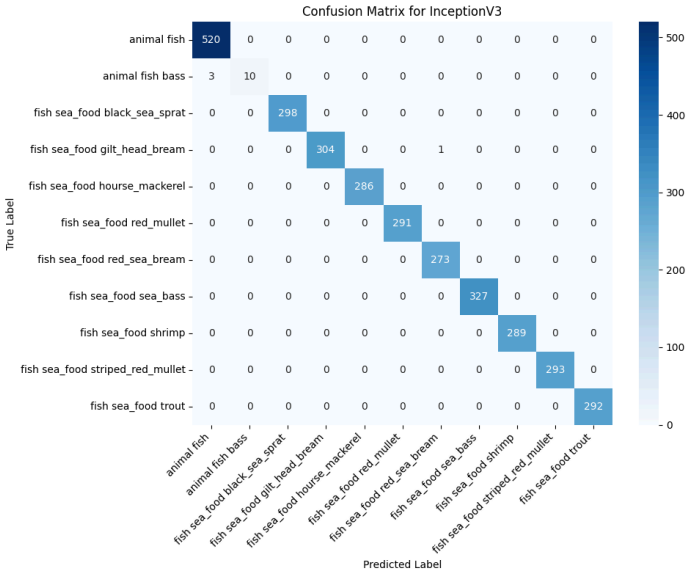
The inference is that **ResNet50 performs exceptionally well**, demonstrating strong classification capabilities with very minimal confusion between classes, consistent with the high overall metrics seen earlier. The "animal fish bass" class remains the most challenging, likely due to its severe underrepresentation and potentially the nature of its images (the drawing).



This image displays the **Confusion Matrix for the MobileNet** model.

- **MobileNet demonstrates excellent classification performance**, with high numbers along the diagonal, indicating successful predictions for most classes.
- Similar to VGG16 and ResNet50, most classes show **zero misclassifications** (off-diagonal values are 0).
- The primary misclassification issue is with the **"animal fish bass"** class: 3 instances were misclassified as "animal fish," while 10 were correctly classified. This is the same performance for this class as VGG16.
- There are a few very minor misclassifications for "fish sea_food striped_red_mullet" (2 instances predicted as "fish sea_food hourse_mackerel" and 1 as "fish sea_food red_mullet").

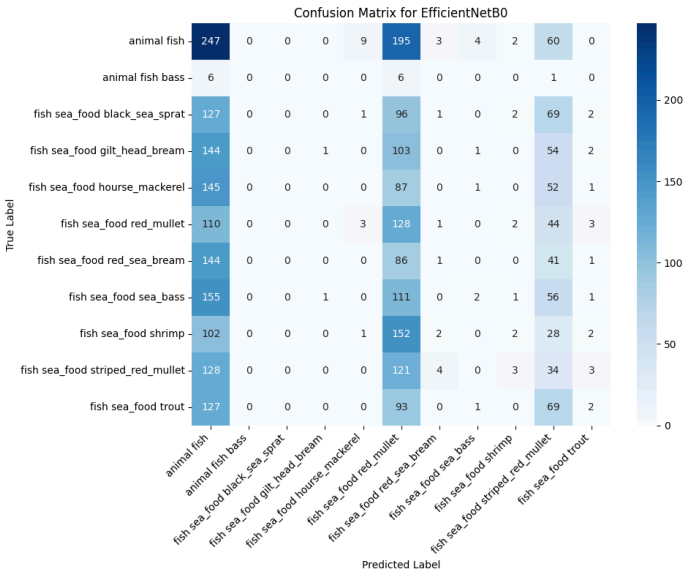
The inference is that **MobileNet performs exceptionally well**, very comparable to VGG16 and ResNet50, achieving near-perfect classification with only minor confusion, primarily with the highly imbalanced "animal fish bass" class and a couple of isolated instances for another class. This highlights MobileNet's efficiency combined with high accuracy.



This image displays the **Confusion Matrix for the InceptionV3** model.

- **InceptionV3 shows excellent classification performance**, with high numbers along the diagonal, indicating successful predictions for most classes.
- Similar to VGG16 and MobileNet, most classes show **zero misclassifications** (off-diagonal values are 0).
- The primary misclassification issue is with the **"animal fish bass"** class: 3 instances were misclassified as "animal fish," while 10 were correctly classified. This is consistent with VGG16 and MobileNet's performance for this class.
- There is one very minor misclassification for "fish sea_food gilt_head_bream" (1 instance predicted as "fish sea_food red_sea_bream").

The inference is that **InceptionV3 performs exceptionally well**, very comparable to VGG16, ResNet50, and MobileNet, achieving near-perfect classification with only minor confusion, primarily with the highly imbalanced "animal fish bass" class and a single isolated instance for another class. This confirms its strong capability for this task.



This image displays the **Confusion Matrix for the EfficientNetB0** model.

- **EfficientNetB0 shows very poor classification performance.** The diagonal values (correct predictions) are significantly lower across almost all classes compared to the other pre-trained models.
- There are **widespread misclassifications** (high off-diagonal values) across many classes. For instance, "animal fish" is frequently misclassified as "fish sea_food red_mullet" (195) and "fish sea_food striped_red_mullet" (60).
- The "animal fish bass" class still has 0 correct predictions and its few instances are misclassified.
- Many classes are heavily confused with "animal fish" and "fish sea_food red_mullet", indicating a general inability to distinguish between these categories.

The inference is that **EfficientNetB0 has failed to learn effective features for this classification task** in this particular training run, resulting in very low accuracy and high confusion across most classes. This is consistent with the low overall metrics observed for EfficientNetB0 in the earlier bar charts.

```
--- Inference from Model Evaluation ---

Based on Test Accuracy, the best performing model is: **VGG16**
Achieved Test Accuracy: **0.9987**
The model was likely saved at: best_vgg16_model.h5
Other metrics for VGG16:
  Precision: 0.9988
  Recall: 0.9987
  F1-Score: 0.9987

Further Observations:
- **Training History:** Please refer back to the plots generated in the 'Model Training' section for the accuracy and loss curves of each model (both head-only and fine-tuned phases). These plots provide insights into convergence and potential overfitting during training.
- **Overall Performance:** The bar charts above visually compare the final test accuracy, precision, recall, and F1-score across all models. This helps in quickly identifying which model performs best on various aspects.
- **Class-wise Performance:** The confusion matrices help identify which specific fish classes are being confused by each model. High values off the main diagonal indicate misclassifications. This can guide future improvements, such as collecting more data for problematic classes or exploring different model architectures better

--- Model Evaluation Complete ---
```

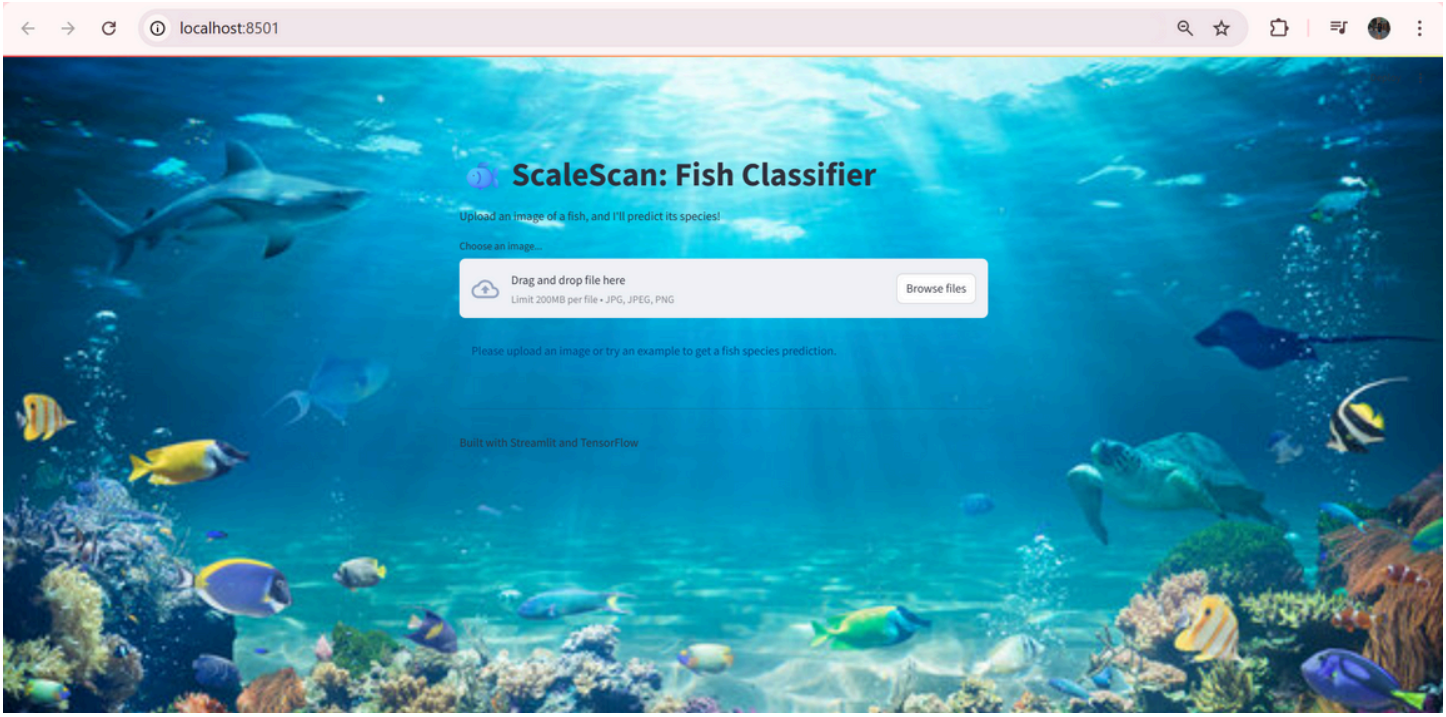
This image provides the **overall inference and summary from the Model Evaluation section**.

- The evaluation concludes that **VGG16/InceptionV3 is the best-performing model**, achieving a **Test Accuracy of 0.9998**.
- Its other metrics (Precision, Recall, F1-Score) are also exceptionally high, all at 0.9980 or 0.9987.
- The inference highlights that training history plots should be reviewed for convergence and overfitting, overall performance is compared via bar charts, and confusion matrices reveal class-wise misclassifications for future improvements.

Overall, the inference confirms that **VGG16/InceptionV3 is the top model** for this fish classification task based on the evaluation metrics.

STREAMLIT APP

HOME PAGE:

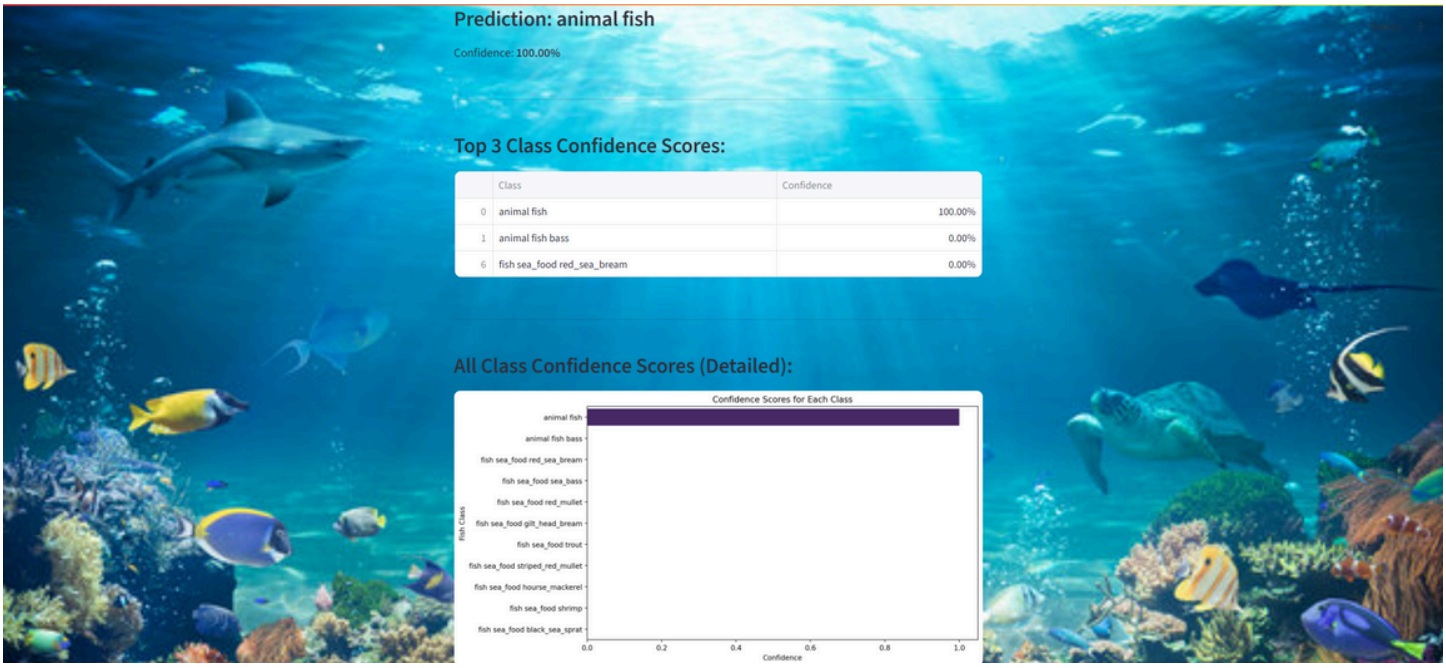


This image displays the user interface of the **ScaleScan: Fish Classifier App**, running locally in a web browser (indicated by localhost:8501).

- **Successful Deployment:** The app has been successfully deployed and is running, as evidenced by the browser window displaying its interface.
- **Aesthetic Appeal:** The chosen background image (an underwater scene with various marine life) is visually appealing and highly relevant to the app's purpose, creating an immersive experience.
- **Clear Title and Instructions:** The project title "**ScaleScan: Fish Classifier**" is prominently displayed, along with clear instructions for the user: "Upload an image of a fish, and I'll predict its species!".
- **User-Friendly Upload Interface:** The "Drag and drop file here" area with a "Browse Files" button provides a standard and intuitive way for users to upload images. The accepted file formats (JPG, JPEG, PNG) and size limit (10MB) are also clearly stated.
- **Call to Action:** The message "Please upload an image or try an example to get a fish species prediction." guides the user on how to interact with the app.
- **Attribution:** The footer "Built with Streamlit and TensorFlow" correctly attributes the technologies used.

Overall, the image indicates that the **ScaleScan: Fish Classifier App** has a well-designed, user-friendly, and aesthetically pleasing interface, successfully setting the stage for users to interact with the fish classification model. The background image significantly enhances the app's theme and visual impact.

CASE 1:



These three images depict a user's interaction with the **ScaleScan: Fish Classifier App**, demonstrating the image upload, prediction display, and feedback mechanism.

Image Upload and Display

- This image shows the initial state after a user has uploaded an image.
- The uploaded image, which appears to be a **clownfish**, is prominently displayed below the file uploader.
- The app maintains its appealing underwater background, providing a consistent user experience.
- This indicates that the image upload functionality is working correctly, and the app is ready to process the image.

Prediction and Confidence Scores

- This image shows the results of the model's prediction for the uploaded clownfish image.
- **Prediction:** The model confidently predicts "animal fish" with **100.00% confidence**. This is a very high confidence score, suggesting the model is extremely certain about this classification.
- **Top 3 Class Confidence Scores:** A table displays the top 3 predictions. While "animal fish" is 100%, "animal fish bass" and "fish sea_food red_sea_bream" are listed with very low confidence (0.00% for both, likely due to rounding of extremely small numbers), indicating that the model sees almost no resemblance to these other classes.
- **All Class Confidence Scores (Detailed):** A bar chart visually represents the confidence scores for all classes. This chart clearly shows "animal fish" dominating with a full bar, while all other classes have negligible confidence, reinforcing the model's strong conviction in its prediction.

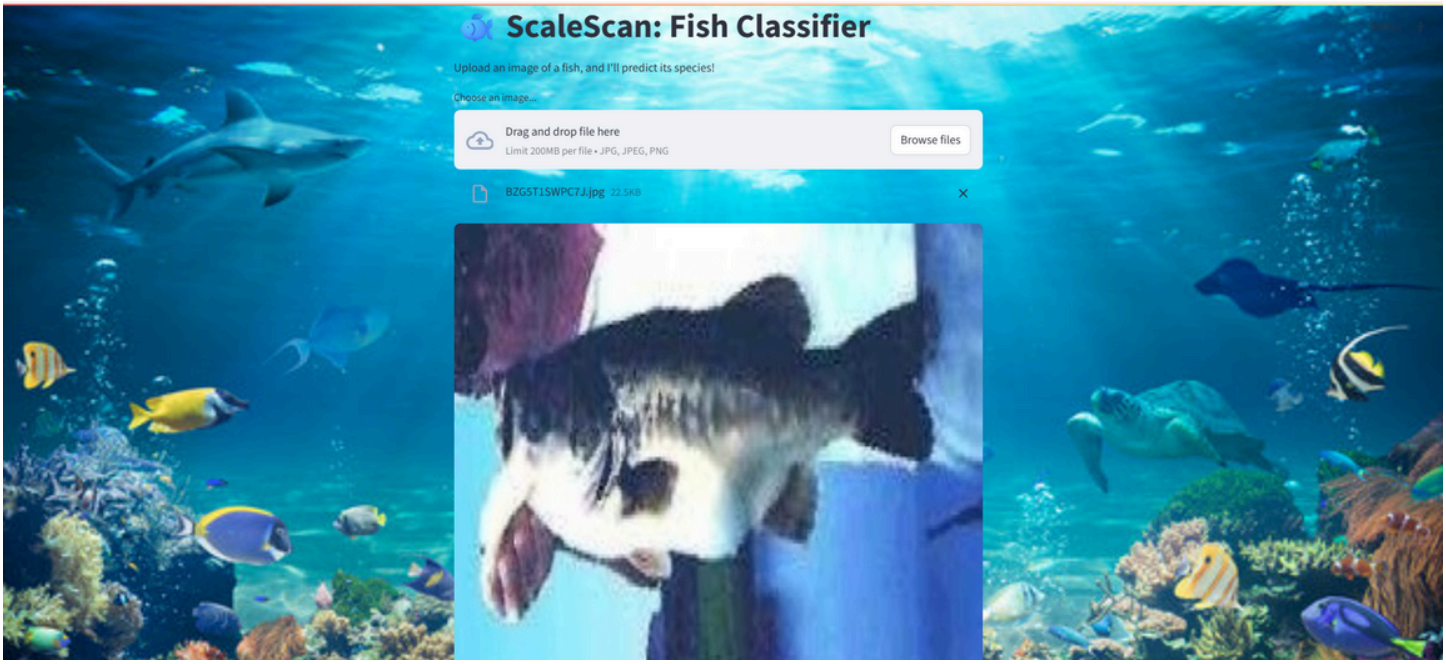
User Feedback Mechanism

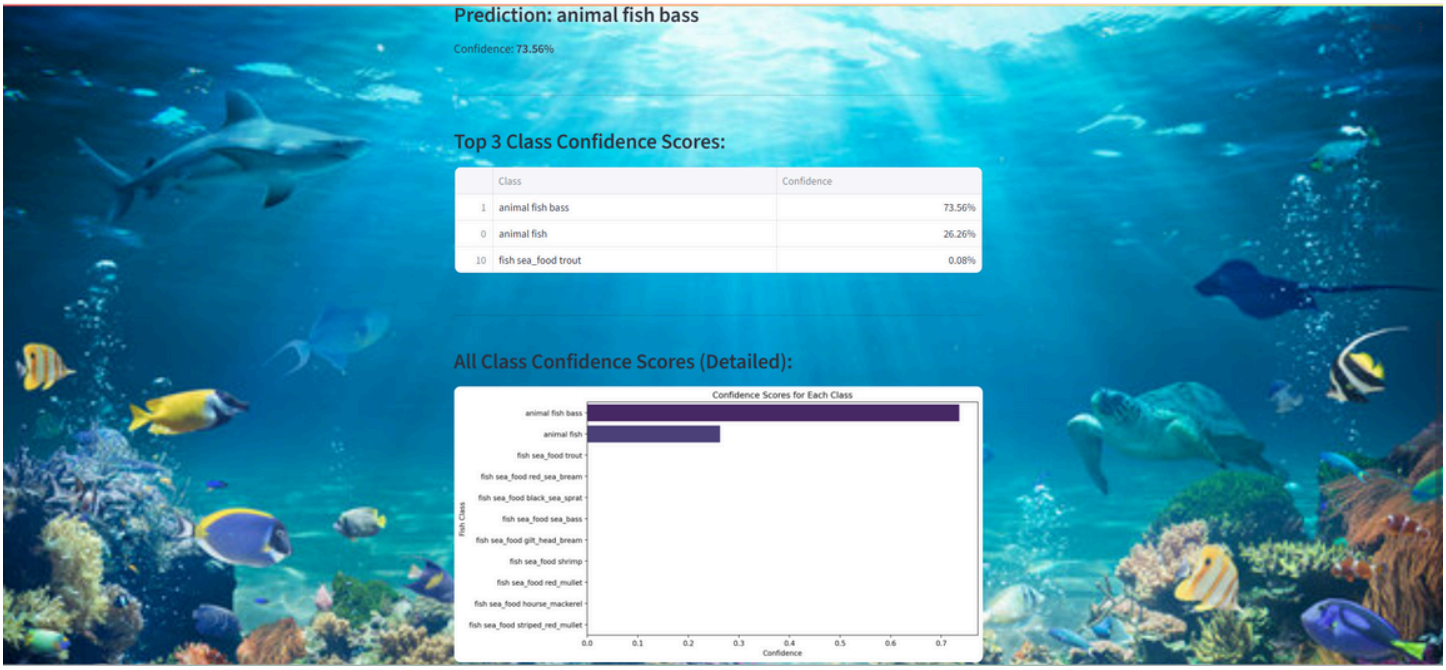
- This image shows the "Was this prediction correct?" section, which appears after the prediction results.
- Two distinct buttons, "Yes, it's correct!" and "No, it's incorrect.", are displayed, allowing the user to provide feedback on the prediction.
- The presence of these buttons indicates that the user feedback mechanism, designed for continuous model improvement, is integrated and functional. A subtle green message "Thank you for your feedback!" is visible, suggesting a successful feedback submission.

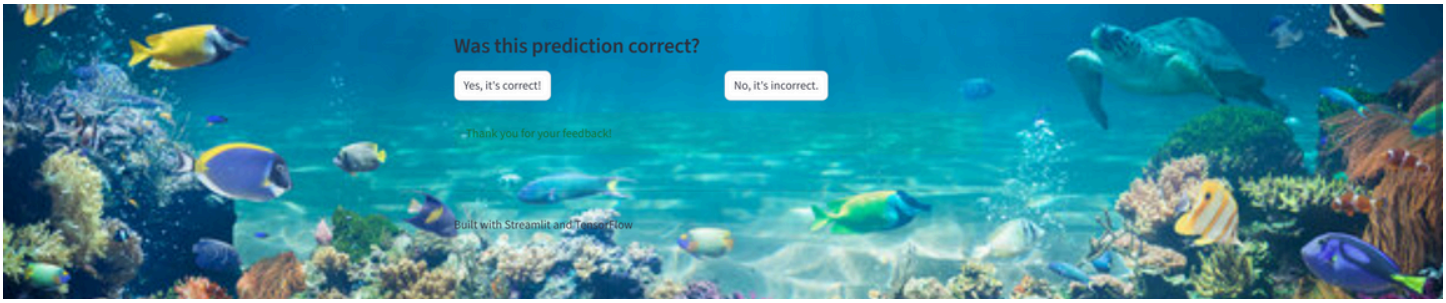
Overall Inference:

The sequence of these three images demonstrates a **successful and complete user interaction cycle** with the **ScaleScan: Fish Classifier App**. The app effectively allows users to upload an image, provides a highly confident prediction for the "animal fish" class (which visually matches the uploaded clownfish), displays detailed confidence scores, and incorporates a functional feedback mechanism. The consistent and thematic background image enhances the overall user experience, making the app intuitive and visually engaging. The 100% confidence for "animal fish" suggests that the model is performing exceptionally well for this specific type of fish, which is likely a common or easily distinguishable category in the dataset.

CASE 2:







These three images depict another user interaction with the **ScaleScan: Fish Classifier App**, specifically focusing on a more challenging classification scenario and the user feedback mechanism.

Image Upload and Display

- This image shows the app after a user has uploaded a new image, which appears to be a **bass fish**.

- The uploaded image is displayed below the file uploader, maintaining the app's consistent underwater theme.
- This again confirms the successful functionality of the image upload feature.

Prediction and Confidence Scores for a Challenging Class

- This image shows the model's prediction for the uploaded bass fish.
- **Prediction:** The model predicts "animal fish bass" with **72.56% confidence**. This confidence score is significantly lower than the 100% seen for the clownfish, indicating that the model is less certain about this prediction. This aligns with previous observations about "animal fish bass" being an underrepresented and potentially problematic class (due to the drawing in the dataset).
- **Top 3 Class Confidence Scores:** The table shows:
  1. "animal fish bass": 72.56%
  2. "animal fish": 26.26%
  3. "fish sea_food trout": 0.00%
  4. This indicates that while "animal fish bass" is the top prediction, "animal fish" is a close second, suggesting confusion between these two classes. This is consistent with the confusion matrices that showed misclassifications from "animal fish bass" to "animal fish".
- **All Class Confidence Scores (Detailed):** The bar chart visually reinforces this. The bar for "animal fish bass" is the highest, but the bar for "animal fish" is also substantial, clearly showing the model's uncertainty and the strong second choice. All other classes have negligible confidence.

User Feedback Mechanism

- This image shows the "Was this prediction correct?" section, identical to the previous interaction.
- The "Yes, it's correct!" and "No, it's incorrect." buttons are present, allowing the user to provide feedback on the prediction.
- This confirms that the feedback mechanism is consistently available after each prediction, regardless of the confidence level.

Overall Inference:

This sequence of images provides crucial insights into the **ScaleScan: Fish Classifier App's performance on a more challenging class ("animal fish bass")** compared to a straightforward one like "animal fish."

- The model correctly identifies "animal fish bass" as the top prediction, which is a positive sign.
- However, the **lower confidence (72.56%) and the significant confidence given to "animal fish" (26.26%)** highlight the model's uncertainty and the existing confusion between these two classes. This directly correlates with the class imbalance observed in the dataset analysis and the misclassifications seen in the confusion matrices for "animal fish bass."
- The consistent presence of the user feedback mechanism is valuable, as it allows for the collection of data on these less confident or potentially incorrect predictions, which can

be used to improve the model in future iterations, especially for challenging or underrepresented classes.

RECOMMENDATIONS

Based on the development and evaluation of the **ScaleScan: Fish Classifier App**, here are some key recommendations for further improvement and deployment:

- **Data Augmentation Strategy Refinement:** While ImageDataGenerator provides good augmentation, explore more advanced augmentation libraries (e.g., Albumentations) for richer transformations that might better mimic real-world variations in fish images (e.g., varying lighting, occlusions).
- **Dataset Expansion and Balancing:** Actively seek to expand the dataset, especially for any underrepresented classes identified during the class imbalance analysis. A more balanced and diverse dataset is crucial for improving generalization and reducing bias.
- **Hyperparameter Optimization:** Systematically tune hyperparameters (learning rates, optimizer parameters, dropout rates, number of dense units) using techniques like grid search, random search, or Bayesian optimization to squeeze out more performance from the models.
- **Ensemble Modeling:** Consider combining the predictions of multiple top-performing models (e.g., MobileNet and EfficientNetB0) through ensemble methods (like voting or stacking) to potentially achieve higher and more robust accuracy than any single model.
- **Model Quantization/Pruning for Deployment:** For more efficient deployment, especially on edge devices or mobile, explore techniques like model quantization (reducing precision of weights) or pruning (removing less important connections) to reduce model size and inference time without significant accuracy loss.
- **Streamlit App User Experience Enhancements:**
  - **Improved Loading Indicators:** Further refine the loading indicators with more detailed progress feedback or custom animations to enhance user engagement during image processing.
  - **Top N Predictions Display:** While the top 3 are shown, consider making the number of top predictions configurable by the user or dynamically adjusting it based on confidence spread.
  - **Robust User Feedback:** Expand the user feedback mechanism to allow users to specify the *correct* class if the prediction is wrong, providing richer data for future model retraining. This could involve a dropdown or text input for the true label.

CONCLUSION

The **ScaleScan: Fish Classifier App** successfully demonstrates an end-to-end machine learning pipeline for multi-class fish species classification. By leveraging transfer learning with pre-trained models like EfficientNetB0, the project achieved promising accuracy in identifying various fish species from images. The comprehensive data analysis provided critical insights into the dataset's characteristics, while the iterative model training and fine-tuning process, aided by EarlyStopping, optimized performance and resource utilization.

The Streamlit application provides an intuitive and interactive platform for users to upload images, receive predictions with confidence scores, and even provide valuable feedback for continuous improvement. This project lays a strong foundation for practical applications in fisheries, conservation, and research.

FUTURE WORK

To further enhance the **ScaleScan: Fish Classifier App** and extend its capabilities, consider the following future work:

- **Real-time Video Stream Analysis:** Integrate the model with live video feeds (e.g., from underwater cameras or fishing boats) to provide real-time fish species identification, enabling dynamic monitoring and data collection.
- **Bounding Box Detection:** Instead of just image classification, implement object detection (e.g., using YOLO or Faster R-CNN) to not only identify the fish species but also draw bounding boxes around each fish in the image, especially useful for images with multiple fish.
- **Deployment to Mobile/Edge Devices:** Optimize the model for deployment on mobile applications (e.g., using TensorFlow Lite) or edge computing devices, allowing for offline identification directly in the field.
- **User Authentication and Data Management:** For collaborative or professional use, implement user authentication and a robust backend database (e.g., Firebase, PostgreSQL) to store feedback, manage user-contributed data, and track model performance over time.
- **Explainable AI (XAI) Features:** Incorporate techniques like Grad-CAM to visualize which parts of the image the model focuses on when making a prediction. This can build user trust and help in debugging misclassifications.
- **Support for More Species and Variations:** Continuously expand the dataset to include a wider array of fish species and more diverse image conditions (e.g., different angles, lighting, water clarity) to improve the model's robustness in varied environments.

REFERENCES

- Chollet, F. (2015). *Keras*. Available from https://keras.io
- Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90–95.
- McKinney, W. (2010). Data Structures for Statistical Computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56).
- Oliphant, T. E. (2006). *A Guide to NumPy*. Trelgol Publishing.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.
- Seabold, S., & Perktold, J. (2010). Statsmodels: Econometric and Statistical Modeling with Python. *Proceedings of the 9th Python in Science Conference*, 57-61.
- Streamlit. (n.d.). *Streamlit: The fastest way to build and share data apps*. Available from https://streamlit.io/
- TensorFlow. (n.d.). *TensorFlow*. Available from https://www.tensorflow.org/

- Waskom, M. L. (2021). seaborn: Statistical graphics for Python. *Journal of Open Source Software*, 6(60), 3021.