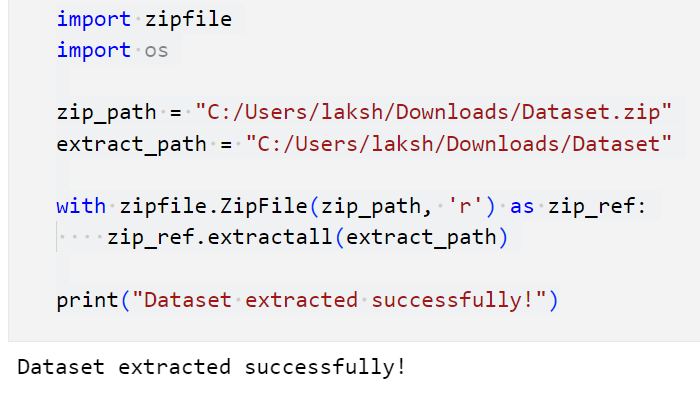
**Multiclass Fish Image Classification**

**INTRODUCTION:**

This project focuses on classifying fish species using deep learning models. A **CNN** is trained from scratch, and **transfer learning** with pre-trained models (VGG16, ResNet50, MobileNet, InceptionV3, EfficientNetB0) is used to enhance accuracy. **Data augmentation** improves model robustness, and performance is evaluated using accuracy, precision, recall, and F1-score.

A **Streamlit web app** allows users to upload images for real-time predictions with confidence scores. Deliverables include trained models, Python scripts, a comparison report, and a GitHub repository. This solution supports **automated fish classification** for research and conservation.

**CODING 1: Extracting Dataset from ZIP File**



This code extracts the dataset from a ZIP file to a specified directory for further processing.

* **zipfile.ZipFile(zip\_path, 'r')**: Opens the ZIP file in read mode.
* **zip\_ref.extractall(extract\_path)**: Extracts all files to the given directory.
* **print("Dataset extracted successfully!")**: Confirms successful extraction.

**Purpose:** This step is necessary to **prepare the dataset** for model training by making fish images accessible for preprocessing and classification.

**CODING 2: Verifying Dataset Folder**

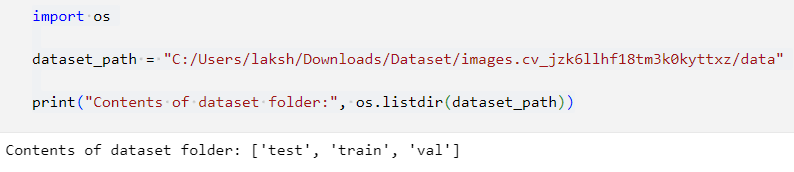


This code checks whether the dataset folder exists and lists its contents.

* **os.path.exists(dataset\_path)**: Verifies if the dataset directory is present.
* **os.listdir(dataset\_path)**: Lists all files and subfolders inside the dataset directory.
* **Print statements** confirm the dataset's availability or prompt a path check.

**Purpose:** Ensures that the dataset is properly extracted and accessible for **further processing, data loading, and model training**.

**CODING 3: Listing Dataset Files**



This code retrieves and displays the contents of the dataset directory.

* **os.listdir(dataset\_path)**: Lists all files and subfolders inside the specified dataset path.
* **Print statement** displays the dataset contents for verification.

**Purpose:** Ensures that the dataset is correctly structured and ready for **data preprocessing and model training** by confirming the presence of image files.

**CODING 4: Data Processing**



This code prepares the fish image dataset for training by defining dataset directories, applying data augmentation, and loading the images using **TensorFlow's ImageDataGenerator**.

**1. Dataset Directory Setup**

* The dataset path is defined, and subdirectories for **training, validation, and testing** are specified.

**2. Image Parameters**

* **IMG\_SIZE = (224, 224)**: Resizes images to 224x224 pixels for model consistency.
* **BATCH\_SIZE = 32**: Loads images in batches of 32 for efficient training.

**3. Data Augmentation for Training Data**

* **Rescaling (rescale=1./255)**: Normalizes pixel values to the range [0,1] to improve model convergence.
* **Augmentations:**
  + **rotation\_range=20**: Randomly rotates images by up to 20 degrees.
  + **width\_shift\_range=0.2 / height\_shift\_range=0.2**: Shifts images horizontally and vertically.
  + **horizontal\_flip=True**: Flips images randomly to increase diversity.
  + **zoom\_range=0.2**: Randomly zooms into images.

**4. Loading the Dataset**

* **Training Data (train\_data)**: Augmented images are loaded with **categorical labels** (multi-class classification).
* **Validation Data (val\_data)**: Only rescaled, no augmentation, used for model tuning.
* **Test Data (test\_data)**: Only rescaled, no augmentation, used for final evaluation.

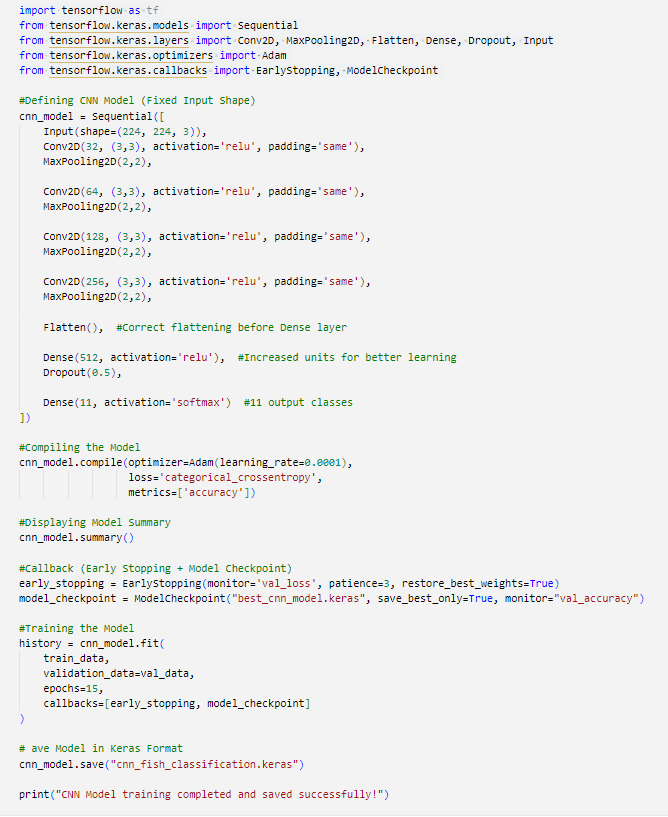
**5. Printing Class Labels**

* **train\_data.class\_indices**: Displays assigned class labels to verify correct dataset loading.

**Purpose:**

This step ensures that the dataset is correctly structured, preprocessed, and augmented before model training. Augmentations help prevent **overfitting**, while normalization speeds up learning.

**CODING 5: CNN Model for Fish Image Classification**



This code defines, compiles, trains, and saves a **Convolutional Neural Network (CNN)** model to classify fish images into multiple categories.

**1. Model Definition**

A **Sequential CNN model** is built with:

* **Input(shape=(224, 224, 3))**: Fixed input size for RGB images.
* **Four Convolutional Layers (Conv2D)** with ReLU activation:
  + Extracts features using 32, 64, 128, and 256 filters.
  + **padding='same'** ensures consistent image size across layers.
* **MaxPooling (MaxPooling2D)**: Reduces spatial dimensions and enhances feature learning.
* **Flatten Layer (Flatten())**: Converts feature maps into a 1D vector.
* **Fully Connected Layers (Dense)**:
  + **512 neurons with ReLU** for learning complex patterns.
  + **Dropout (0.5)** to prevent overfitting.
  + **11 output neurons with Softmax** for multi-class classification.

**2. Model Compilation**

* **Optimizer (Adam)**: Adaptive learning with **0.0001 learning rate** for stable convergence.
* **Loss Function (categorical\_crossentropy)**: Suitable for multi-class classification.
* **Metric (accuracy)**: Evaluates model performance.

**3. Model Summary**

* **cnn\_model.summary()** displays the model architecture, layer details, and parameter count.

**4. Callbacks for Training Stability**

* **Early Stopping (EarlyStopping)**:
  + Monitors validation loss (val\_loss).
  + Stops training if no improvement for **3 consecutive epochs** to prevent overfitting.
  + Restores the best model weights.
* **Model Checkpoint (ModelCheckpoint)**:
  + Saves the best-performing model based on **validation accuracy (val\_accuracy)**.

**5. Model Training (fit function)**

* Uses **training (train\_data)** and **validation (val\_data)** datasets.
* Trains for **15 epochs** with callbacks to ensure optimal learning.

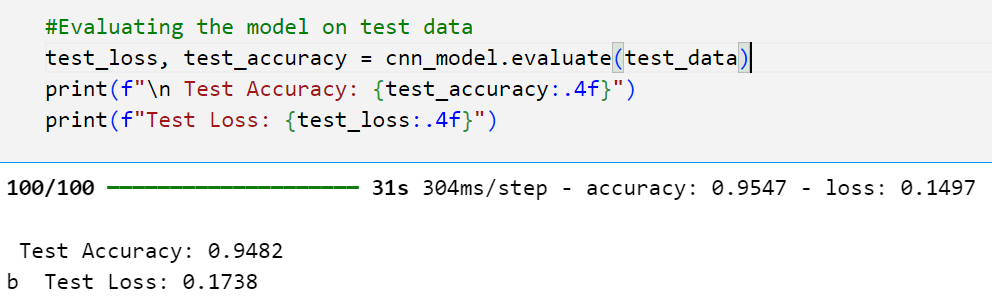
**6. Model Saving**

* **cnn\_model.save("cnn\_fish\_classification.keras")**:
  + Saves the trained model in **Keras format** for future use.

**Purpose:**

* This step builds a **deep learning model** tailored for **fish image classification**.
* The model is **optimized, monitored, and saved** to ensure the best accuracy.
* The trained model will be **used for real-time predictions** in the Streamlit app.

**CODING 6: Evaluating the CNN Model on Test Data**

****

This code evaluates the trained CNN model on unseen test data to measure its final accuracy and loss.

**1. Model Evaluation (evaluate function)**

* **cnn\_model.evaluate(test\_data)**:
  + Passes the test dataset (test\_data) to the model.
  + Computes the **test loss** and **test accuracy** based on the trained weights.

**2. Displaying Evaluation Metrics**

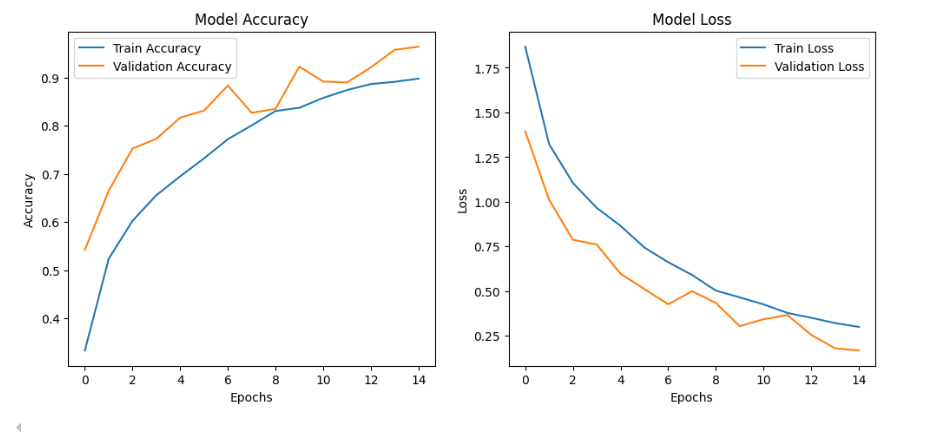
* **test\_accuracy**: Represents the model's performance in correctly classifying test images.
* **test\_loss**: Measures the error between predicted and actual labels.
* **Formatted print statements**: Display accuracy and loss values up to 4 decimal places.

**Purpose:**

* This step **validates model generalization** by checking its performance on unseen data.
* Ensures the model **is not overfitting** and performs well on real-world images.
* Helps in comparing different models and selecting the best one for **deployment**.

**CODING 7: Plotting Model Training History**





This code visualizes the training and validation **accuracy** and **loss** over epochs to analyze the model's learning behavior.

**1. Function Definition: plot\_training\_history(history)**

* **plt.figure(figsize=(12,5))**: Sets the figure size for better visualization.
* **Accuracy Plot** (subplot(1,2,1))
  + Plots training (history.history['accuracy']) and validation accuracy (history.history['val\_accuracy']).
  + X-axis: Epochs, Y-axis: Accuracy.
  + Displays a legend to differentiate between training and validation curves.
* **Loss Plot** (subplot(1,2,2))
  + Plots training (history.history['loss']) and validation loss (history.history['val\_loss']).
  + X-axis: Epochs, Y-axis: Loss.
  + Helps identify if the model is overfitting.
* **plt.show()**: Displays the plots.

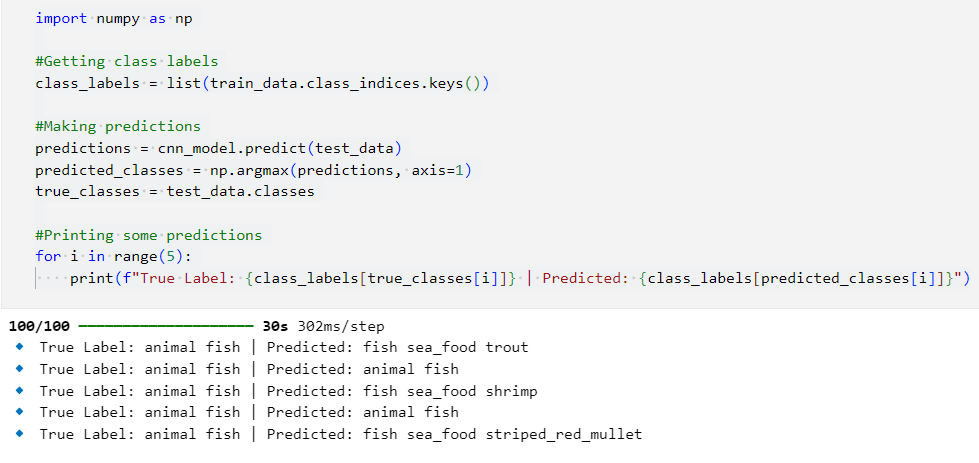
**2. Function Call: plot\_training\_history(history)**

* Passes the **training history (history)** from the fit() function to visualize the model's performance.

**Purpose:**

* Helps **monitor model performance** during training.
* Detects **overfitting** (if validation loss increases while training loss decreases).
* Assists in tuning hyperparameters for better generalization.

**CODING 8: Making Predictions on Test Data**

****

This code makes predictions on the test dataset using the trained CNN model and compares them with the actual class labels.

**1. Extracting Class Labels**

* **train\_data.class\_indices.keys()**: Retrieves the class names from the dataset.
* **class\_labels = list(train\_data.class\_indices.keys())**: Converts the class indices into a list of class labels.

**2. Making Predictions**

* **cnn\_model.predict(test\_data)**:
  + Uses the trained model to predict class probabilities for each test image.
  + Outputs a 2D array where each row contains probabilities for all classes.
* **np.argmax(predictions, axis=1)**:
  + Selects the class index with the highest probability as the predicted class.
* **true\_classes = test\_data.classes**:
  + Retrieves the actual class labels from the test dataset.

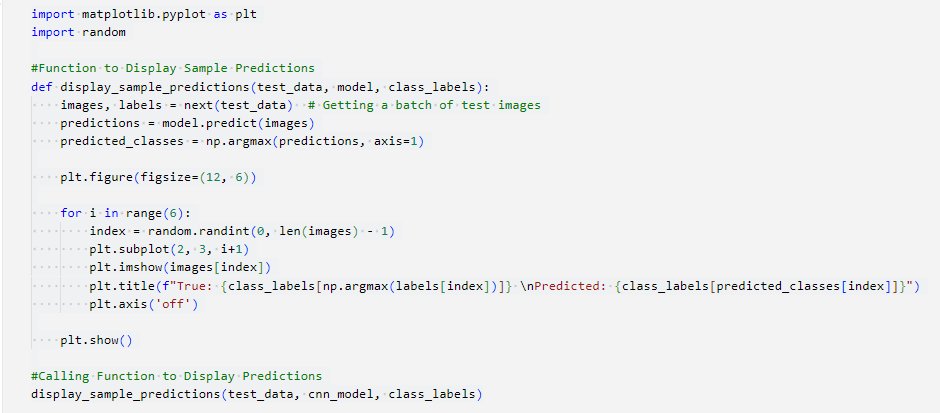
**3. Displaying Sample Predictions**

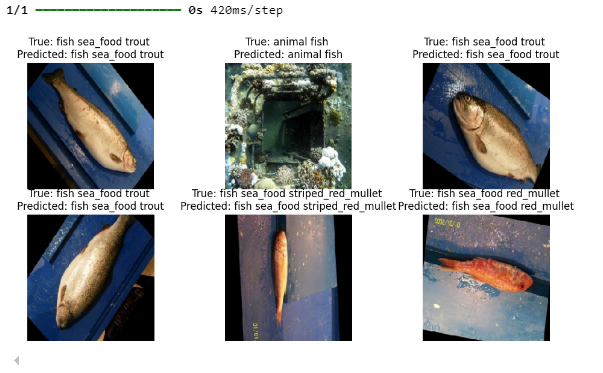
* **for i in range(5):**: Loops through the first 5 test samples.
* **Prints actual vs. predicted labels** for comparison.

**Purpose:**

* **Validates model predictions** by comparing actual vs. predicted classes.
* Helps **analyze model accuracy qualitatively** before computing metrics.
* Identifies **misclassified images** for further model improvement.

**CODING 9: Displaying Sample Predictions with Images**

****

****

This code visualizes random test images along with their **true labels** and **model predictions**, helping to assess how well the model classifies fish species.

**1. Function Definition: display\_sample\_predictions()**

* **images, labels = next(test\_data)**
  + Retrieves a batch of test images and their actual labels.
* **model.predict(images)**
  + Uses the trained CNN model to predict class probabilities for the images.
* **np.argmax(predictions, axis=1)**
  + Converts the probability outputs into class indices for predicted labels.

**2. Plotting Random Sample Images**

* **plt.figure(figsize=(12,6))**
  + Creates a figure to display six images.
* **Loop (for i in range(6))**
  + Randomly selects **six images** from the batch.
  + **plt.imshow(images[index])**: Displays the selected image.
  + **plt.title()**: Shows the true label and predicted class.
  + **plt.axis('off')**: Removes axes for better visualization.
* **plt.show()**
  + Displays the plotted images with their corresponding labels.

**3. Function Call**

* **display\_sample\_predictions(test\_data, cnn\_model, class\_labels)**
  + Calls the function to visualize predictions on a random batch of test images.

**Purpose:**

* Provides a **visual assessment** of model performance.
* Helps **identify misclassifications** and analyze prediction errors.
* Useful for debugging and improving model accuracy.

**CODING 10: Transfer Learning with ResNet50 for Fish Classification**

****

This code implements **transfer learning** using the **ResNet50** model to improve fish image classification accuracy. The ResNet50 model, pre-trained on the **ImageNet dataset**, is adapted to classify fish species by adding custom layers.

**1. Loading Pre-Trained ResNet50 Model**

* **weights='imagenet'**: Uses pre-trained weights from ImageNet to leverage existing knowledge.
* **include\_top=False**: Removes the fully connected (classification) layer of ResNet50, allowing customization.
* **input\_shape=(224, 224, 3)**: Sets input image size to 224×224 pixels with 3 color channels (RGB).
* **base\_model.trainable = False**: Freezes the ResNet50 layers so they are not updated during training.

**2. Building the Transfer Learning Model**

* **Flatten()**: Converts feature maps from the CNN base model into a 1D vector.
* **Dense(256, activation='relu')**: Adds a fully connected layer with 256 neurons for feature extraction.
* **Dropout(0.5)**: Reduces overfitting by randomly deactivating 50% of neurons during training.
* **Dense(11, activation='softmax')**: Defines the final classification layer for 11 fish species.

**3. Compiling the Model**

* **Optimizer: Adam(learning\_rate=0.001)**: Optimizes model weights for faster and stable learning.
* **Loss Function: categorical\_crossentropy**: Used for multi-class classification problems.
* **Metric: accuracy**: Evaluates model performance during training and validation.

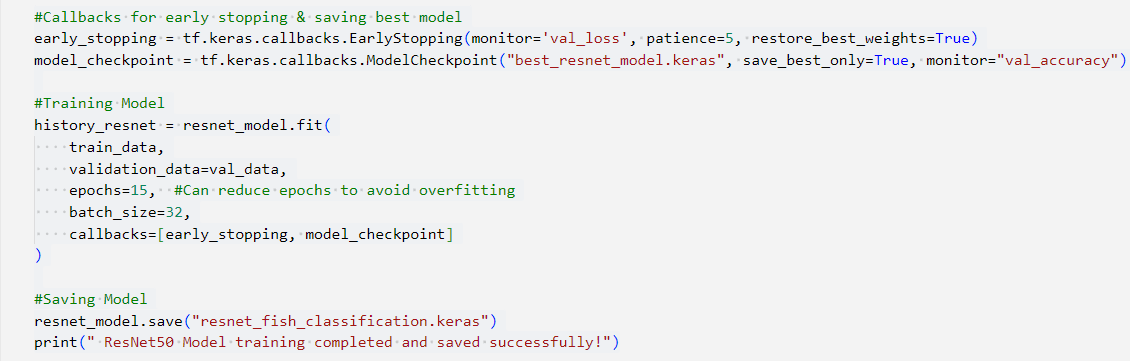
**4. Displaying Model Summary**

* Prints the model architecture, showing the number of layers and parameters.

**Purpose:**

* Uses **transfer learning** to improve classification accuracy with fewer training samples.
* Reduces training time by leveraging **pre-trained ResNet50** features.
* Enhances model performance while minimizing overfitting.
* Helps compare **CNN vs. ResNet50** performance for better decision-making.

**CODING 11: Training ResNet50 Model with Callbacks**

****

This code trains the **ResNet50-based fish classification model** while implementing **early stopping** and **model checkpointing** to optimize performance and prevent overfitting.

**1. Callbacks for Training Optimization**

* **Early Stopping (EarlyStopping)**:
  + **monitor='val\_loss'**: Watches validation loss to detect overfitting.
  + **patience=5**: Stops training if validation loss does not improve for 5 consecutive epochs.
  + **restore\_best\_weights=True**: Ensures the best-performing model weights are retained.
* **Model Checkpoint (ModelCheckpoint)**:
  + **save\_best\_only=True**: Saves the model only when validation accuracy improves.
  + **monitor="val\_accuracy"**: Selects the best model based on validation accuracy.

**2. Training the ResNet50 Model**

* **train\_data and val\_data**: Passes the training and validation datasets.
* **epochs=15**: Trains for up to 15 epochs (can be adjusted).
* **batch\_size=32**: Uses mini-batches of 32 images per update.
* **callbacks=[early\_stopping, model\_checkpoint]**:
  + Stops early if overfitting occurs.
  + Saves the best model for deployment.

**3. Saving the Final Model**

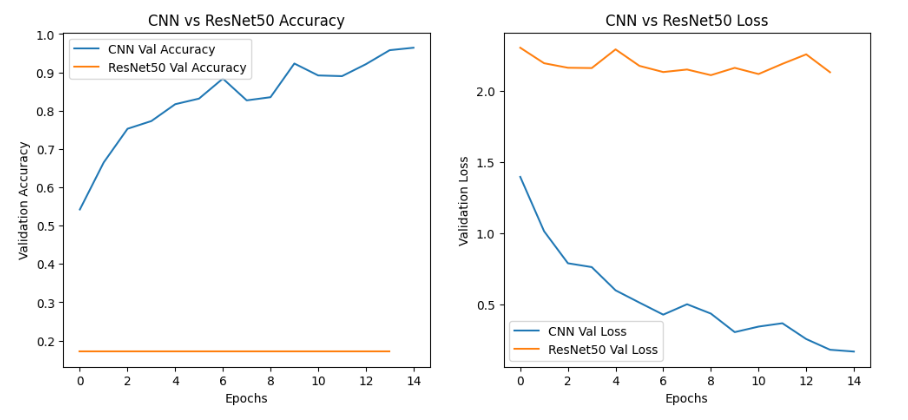
* Saves the trained model in **Keras format (.keras)** for future inference or deployment.

**Purpose:**

**Prevents Overfitting**: Stops training when validation loss starts increasing.  
**Saves the Best Model**: Ensures only the most accurate model is saved.  
**Efficient Training**: Optimizes resource usage and avoids unnecessary epochs.  
**Improves Model Performance**: Uses transfer learning with ResNet50 for better accuracy

**CODING 12: Comparing CNN and ResNet50 Models**

****

****

This code **compares the performance of the CNN model and the ResNet50 model** using validation accuracy and loss over training epochs. It **visualizes model performance** to determine which model generalizes better.

**1. Function to Compare Models**

* **Defines a function** that takes the training history of the **CNN model** and **ResNet50 model** as input.
* **Creates a figure (12x5 inches)** for side-by-side comparison.

**2. Plotting Validation Accuracy**

* **Plots validation accuracy** over epochs for both models.
* **X-axis:** Number of training epochs.
* **Y-axis:** Validation accuracy.
* **Legend differentiates** between CNN and ResNet50 accuracy.
* **Helps determine which model generalizes better on unseen data.**

**3. Plotting Validation Loss**

* **Plots validation loss** over epochs for both models.
* **Lower validation loss = better generalization.**
* **Helps detect overfitting** (if loss increases while accuracy improves).

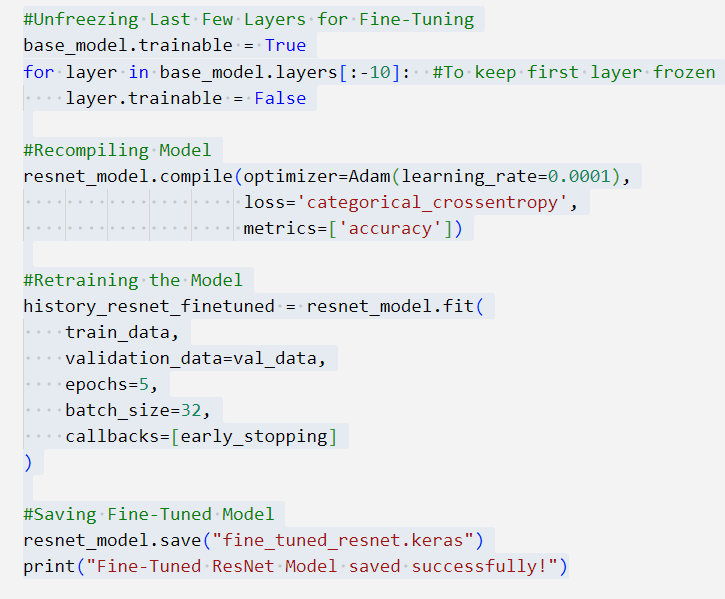
**4. Calling the Function**

* **Passes the training histories** of CNN (history) and ResNet50 (history\_resnet).
* **Displays comparative plots** to evaluate model performance.

**Purpose:**

**Model Performance Analysis**: Helps compare CNN and ResNet50 accuracy & loss.  
**Selects Best Model**: Determines which model generalizes better.  
**Detects Overfitting**: Observes if validation loss increases while accuracy improves.  
**Optimizes Training Strategy**: Helps decide whether to use CNN, ResNet50, or fine-tune further.

**CODING 13: Fine-Tuning ResNet50 Model**

****

This code **performs fine-tuning on the pre-trained ResNet50 model** by unfreezing the last few layers to allow them to be trained on the fish classification dataset. Fine-tuning improves feature extraction and enhances model accuracy.

**1. Unfreezing Last Few Layers for Fine-Tuning**

* **Enables fine-tuning** by unfreezing the last 10 layers of ResNet50 while keeping the earlier layers frozen.
* **Why?** The early layers extract general features, while the last layers capture dataset-specific patterns.
* **Benefit:** Allows model to adjust feature representations for improved accuracy.

**2. Recompiling the Model**

* **Recompiles the model** after unfreezing layers to apply changes.
* **Uses a smaller learning rate (0.0001)** for gradual weight adjustments, preventing drastic changes.
* **Loss Function:** categorical\_crossentropy (used for multi-class classification).
* **Metric:** Accuracy (to evaluate model performance).

**3. Retraining the Model**

* **Trains the model for 5 epochs** (avoiding overfitting with excessive training).
* **Uses early\_stopping callback** to stop training if validation loss doesn’t improve.
* **Why?**  
  Enhances feature extraction for the fish dataset.  
  Allows model to adapt to domain-specific features.

**4. Saving the Fine-Tuned Model**

* **Saves the fine-tuned model** in .keras format for later use.
* **Ensures model can be reloaded without retraining.**

**Purpose:**

**Improves Accuracy**: Fine-tuning adjusts ResNet50 to the fish dataset.  
**Optimizes Feature Extraction**: Last layers capture dataset-specific patterns.  
**Prevents Overfitting**: Controls learning rate and stops training when necessary.  
**Saves Training Time**: Uses pre-trained weights instead of training from scratch.

**CODING 14: Training ResNet50 Model with Callbacks**



This code **trains a ResNet50-based deep learning model** for multiclass fish image classification using **early stopping** and **model checkpointing** to optimize performance and prevent overfitting.

**1. Callbacks for Training Optimization**

* **EarlyStopping Callback**
  + Monitors **validation loss (val\_loss)** to prevent overfitting.
  + **patience=3** → Stops training if validation loss doesn’t improve for 3 consecutive epochs.
  + **restore\_best\_weights=True** → Loads the best model weights before stopping.
* **ModelCheckpoint Callback**
  + Saves the model with the highest **validation accuracy (val\_accuracy)** during training.
  + Stores the best-performing model as "best\_resnet\_model.keras".

**2. Training the Model**

* **Trains the ResNet50 model using:**  
  **Training data (train\_data)** → Model learns from labeled fish images.  
  **Validation data (val\_data)** → Used for evaluating model performance after each epoch.  
  **Epochs = 2** → A small number of epochs to prevent overfitting and reduce training time.  
  **Batch size = 32** → Processes 32 images per training step.  
  **Callbacks included** → Stops early and saves the best model during training.

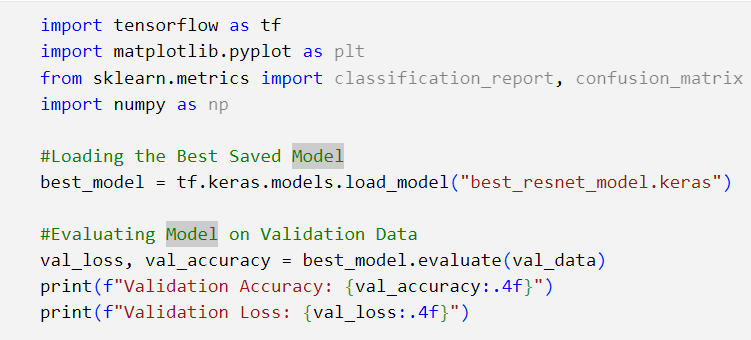
**3. Saving the Final Model**

* **Saves the trained model** as "resnet\_fish\_classification.keras" for future use.
* **Why?** Prevents the need to retrain the model from scratch, allowing reuse.

**Purpose of This Code:**

**Prevents Overfitting**: Early stopping halts training if validation loss increases.  
**Saves Best Model**: Ensures only the highest-performing model is stored.  
**Efficient Training**: Uses a pre-trained ResNet50 model with a reduced epoch count.  
**Model Deployment Ready**: The trained model can be loaded later for prediction.

**CODING 15: Evaluating the Best Saved Model**

****

This code **loads the best saved ResNet50 model** and evaluates its performance on the **validation dataset** to check how well it generalizes.

**1. Loading the Best Model**

* **Loads the pre-trained model** saved earlier as "best\_resnet\_model.keras".
* This ensures that the best-performing model (with highest validation accuracy) is used for evaluation.
* **Why?**  
  Avoids retraining the model from scratch.  
  Ensures we use the most optimized version of the model.

**2. Evaluating Model on Validation Data**

* Evaluates the model’s performance using **validation data (val\_data)**.
* Computes **two key metrics**:  
  **Validation Accuracy (val\_accuracy)** → Measures how well the model classifies unseen images.  
  **Validation Loss (val\_loss)** → Measures the error in predictions (lower is better).

**3. Printing Evaluation Results**

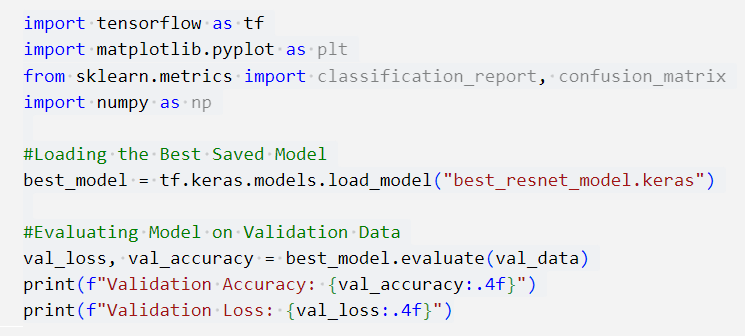
* Displays the model's **final validation accuracy and loss** to assess performance.

**Purpose of This Code:**

**Ensures Best Model Selection**: Uses the saved model with the highest validation accuracy.  
**Measures Generalization**: Evaluates performance on unseen validation data.  
**Prepares for Deployment**: Confirms model quality before using it for real-world predictions.

**Step: Model Evaluation & Performance Metrics**

**>>>** **Evaluating the Best Saved Model:**

****

This code **loads the best-trained ResNet50 model** and evaluates its performance on the **validation dataset** to assess its generalization ability.

**1. Importing Required Libraries**

* **TensorFlow** → Loads the saved deep learning model and performs evaluation.
* **Matplotlib** → Used for potential visualization (not utilized in this snippet).
* **Sklearn.metrics** → Provides metrics like the **classification report** and **confusion matrix** (likely to be used in further analysis).
* **NumPy** → Helps with numerical operations like handling predictions and true labels.

**2. Loading the Best-Trained Model**

* Loads the previously saved **best ResNet50 model** from disk.
* This model was saved using **ModelCheckpoint** during training, ensuring it had the highest validation accuracy.

**Why is this done?**  
**Avoids retraining the model** from scratch.  
**Ensures the best-performing version** is used for evaluation and deployment.

**3. Evaluating the Model on Validation Data**

* **Computes validation accuracy and loss** by testing the model on unseen **validation data (val\_data)**.
* Returns:
  + **Validation Accuracy (val\_accuracy)** → Measures the percentage of correctly classified images.
  + **Validation Loss (val\_loss)** → Represents how well the model is minimizing errors (lower is better).

**Why is this done?**  
Measures **how well the model generalizes** to unseen validation data.  
Helps detect **overfitting** (if training accuracy is high but validation accuracy is low).  
Provides a benchmark before testing on completely new data.

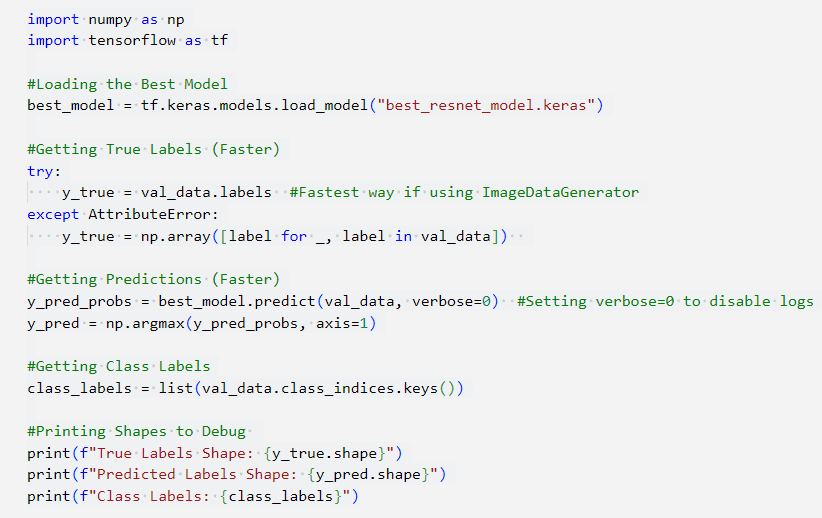
**4. Displaying Model Performance**

* Prints **final validation accuracy and loss**, formatted to **4 decimal places** for clarity.
* Helps in comparing the **performance of different models** or tuning hyperparameters.

**Why This Code is Important?**

**Ensures the best ResNet50 model is used** for evaluation.  
**Checks model generalization ability** before deployment.  
**Helps decide if further fine-tuning is needed** based on validation accuracy/loss.  
**Prepares the model for real-world testing and predictions.**

**>>>** **Extracting True Labels and Predictions:**



This code **loads the best-trained model** and **extracts predictions & true labels** for performance evaluation on the **validation dataset**.

**1. Importing Required Libraries**

* **NumPy** → Used for numerical operations like array manipulations.
* **TensorFlow** → Loads and runs the deep learning model for inference.

**2. Loading the Best-Trained Model**

* Loads the previously saved **best ResNet50 model** for evaluation.
* The saved model was stored during training using **ModelCheckpoint**.

**Why is this done?**  
Avoids retraining, allowing immediate inference.  
Ensures the best-performing version is used for predictions.

**3. Extracting True Labels**

* If val\_data is created using **ImageDataGenerator**, labels are directly available as val\_data.labels.
* If labels are not available (e.g., in tf.data pipeline), extract them manually.

**Why is this done?**  
Ensures correct **ground truth labels** for performance evaluation.  
Enables comparison between actual and predicted labels.

**4. Making Predictions**

* **predict(val\_data)** generates **probabilities** for each class.
* **np.argmax(y\_pred\_probs, axis=1)** converts these probabilities into **predicted class labels**.
* **Setting verbose=0** prevents logs from cluttering output.

**Why is this done?**  
Converts softmax outputs into class predictions.  
Allows us to compare predictions with ground truth labels.

**5. Extracting Class Labels**

* Retrieves **actual class names** mapped to numerical labels.
* This helps interpret **predictions in human-readable format**.

**Why is this done?**  
Essential for **classification reports & confusion matrices**.  
Helps in **visualizing and debugging model predictions**.

**6. Debugging Shapes**

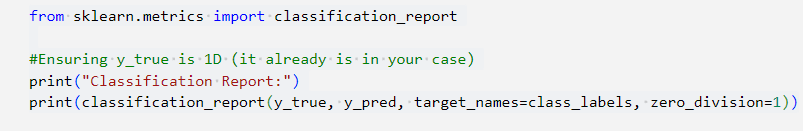
* **Prints the shape of true and predicted labels** to ensure correctness.
* If shapes **don’t match**, it indicates potential **data pipeline issues**.

**Why is this done?**  
Prevents misalignment errors between true and predicted labels.  
Ensures compatibility for further evaluation (confusion matrix, classification reports, etc.).

**Why This Code is Important?**

Extracts **true labels and predictions** to evaluate model performance.  
Ensures correct **class mappings** for analysis.  
Prepares data for **classification reports, confusion matrices, and visualization**.  
Detects **potential label mismatches** before further evaluation.

**>>>** **Generating Classification Report:**



This code generates a **detailed classification report** to evaluate the model’s performance on the **validation dataset**.

**1. Importing Required Library**

* **classification\_report** (from sklearn.metrics) calculates and displays key classification metrics:
  + **Precision** (Positive Predictive Value)
  + **Recall** (Sensitivity)
  + **F1-score** (Harmonic mean of Precision & Recall)
  + **Support** (Number of true instances per class)

**2. Printing the Classification Report**

**Breakdown of Parameters:**

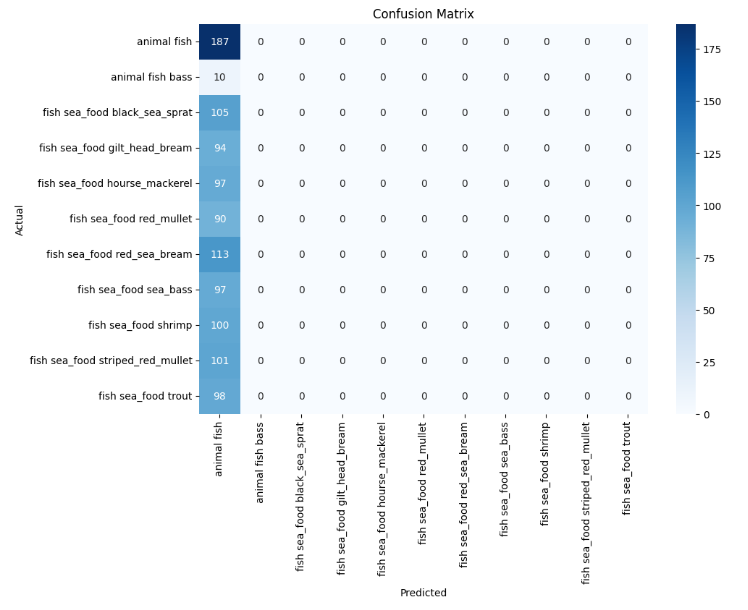
* **y\_true** → The **true labels** of the validation dataset.
* **y\_pred** → The **predicted labels** from the trained model.
* **target\_names=class\_labels** → Maps numerical labels to actual **class names**.
* **zero\_division=1** → Handles cases where precision/recall is undefined (avoids division by zero errors).

**Why is This Done?**

**Evaluates Model Performance** → Provides a **detailed breakdown** of how well the model classifies each fish species.  
**Identifies Class Imbalances** → If a class has low precision/recall, it may indicate **poor predictions** for that category.  
**Helps Debug Issues** → If precision/recall is **very low**, it might signal **overfitting, class imbalance, or poor data quality**.  
**Improves Model Interpretability** → The classification report provides **per-class insights**, helping fine-tune the model.

**>>>** **Confusion Matrix Visualization:**

****

****

This code **computes and visualizes the confusion matrix** to assess the **performance of the fish classification model**.

**1. Importing Required Libraries**

* **confusion\_matrix** (from sklearn.metrics) → Computes the confusion matrix.
* **seaborn (sns)** → Used for visualizing the matrix as a heatmap.
* **matplotlib.pyplot** → Used for plotting the heatmap.

**2. Computing the Confusion Matrix**

* **y\_true** → The actual labels from the validation dataset.
* **y\_pred** → The predicted labels from the model.
* The **confusion matrix** provides a **summary of correct and incorrect predictions** for each class.

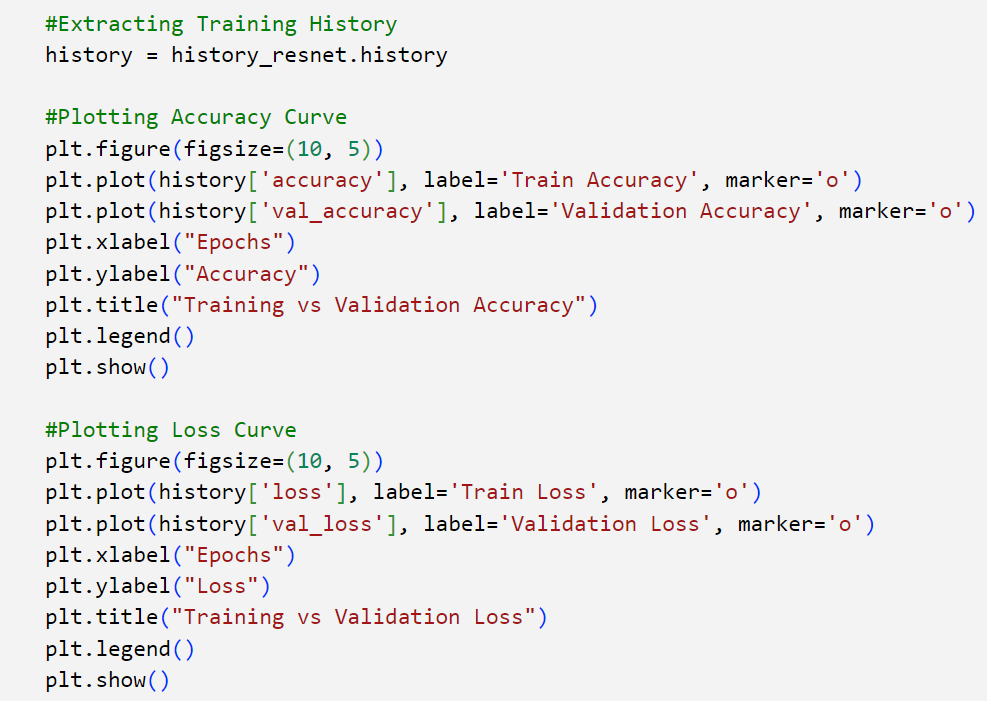
**3. Visualizing the Confusion Matrix**

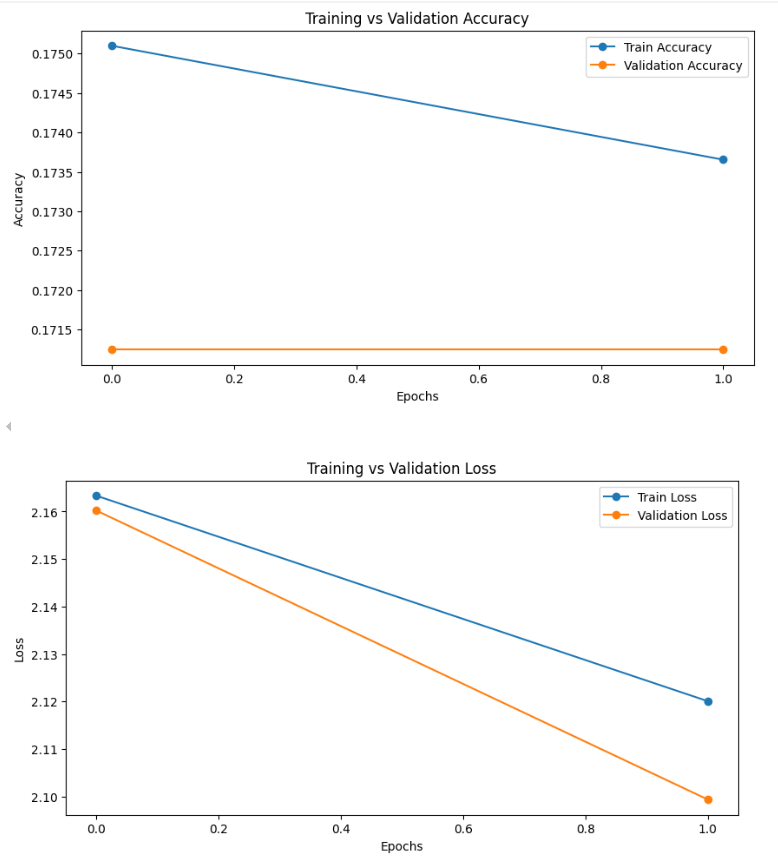
* **plt.figure(figsize=(10, 7))** → Sets figure size.
* **sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class\_labels, yticklabels=class\_labels)**
  + **annot=True** → Displays values inside each cell.
  + **fmt="d"** → Ensures integer formatting.
  + **cmap="Blues"** → Uses a blue color scheme for better readability.
  + **xticklabels=class\_labels** & **yticklabels=class\_labels** → Labels the axes with fish species names.
* **Labels & Title**
  + **X-axis** = "Predicted" → Displays predicted fish classes.
  + **Y-axis** = "Actual" → Displays true fish classes.
  + **Title** = "Confusion Matrix".

**Why is This Done?**

**Evaluates Model Predictions** → Helps understand how the model **misclassifies certain fish species**.  
**Detects Class Imbalance Issues** → If some classes have **many misclassifications**, dataset balancing may be required.  
**Identifies Overfitting or Underfitting** → A high number of off-diagonal values **indicates poor generalization**.  
**Improves Model Interpretability** → Provides insights into **which fish species are often confused** with others.

**>>>** **Training History Visualization:**

****



This code **extracts and visualizes the model's training history**, helping to analyze how well the model is learning over epochs.

**1. Extracting Training History**

* **history\_resnet.history** → Contains details of the model’s performance across training epochs.
* Stores values for:
  + **Training Accuracy & Loss** → history['accuracy'], history['loss']
  + **Validation Accuracy & Loss** → history['val\_accuracy'], history['val\_loss']

**2. Plotting Accuracy Curve**

* **Plots training and validation accuracy over epochs**.
* **Markers (marker='o')** → Help identify specific data points.
* **Legend (plt.legend())** → Distinguishes between training and validation accuracy.
* **Helps detect overfitting**:
  + **Ideal case** → Validation accuracy closely follows training accuracy.
  + **Overfitting** → Large gap (high training accuracy, low validation accuracy).

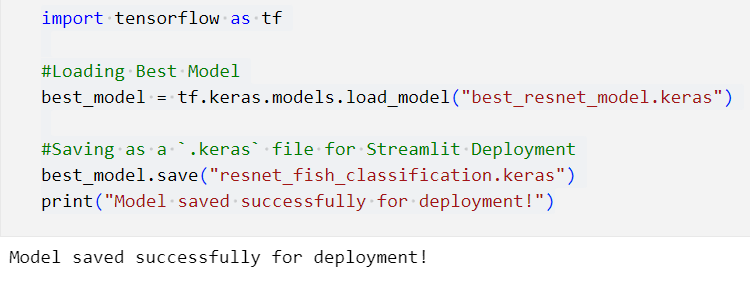
**3. Plotting Loss Curve**

* **Plots training and validation loss over epochs**.
* **Loss should decrease over time**:
  + If **validation loss increases while training loss decreases**, the model is **overfitting**.
  + If both losses **stagnate at high values**, the model **is underfitting**.

**Why is This Done?**

**Monitor Model Performance** → Understand how well the model is learning.  
**Detect Overfitting or Underfitting** → Identify when to stop training or adjust hyperparameters.  
**Improve Model Generalization** → Fine-tune the model based on the curves.  
**Helps in Debugging** → If accuracy remains low, possible issues in data preprocessing or model architecture.

**Saving Model for Deployment:**

****

This code **loads the best trained model** and **saves it in a format suitable for deployment using Streamlit**.

**1. Loading the Best Model**

* **Loads the best-performing ResNet50 model** (previously trained and saved as "best\_resnet\_model.keras").
* Ensures that the best model (based on validation accuracy) is used for final deployment.

**2. Saving Model in .keras Format**

* Saves the model as **"resnet\_fish\_classification.keras"**, which is optimized for **fast loading and inference**.
* .keras format is recommended for TensorFlow **model storage and deployment**.
* The saved model includes:
  + **Model architecture** (layers, activations, connections).
  + **Trained weights** (learned parameters from training).
  + **Optimizer state** (if needed for further fine-tuning).

**3. Printing Confirmation Message**

* **Confirms successful saving** of the model.
* Ensures the model is now ready to be integrated into a **Streamlit web application**.

**Why is This Done?**

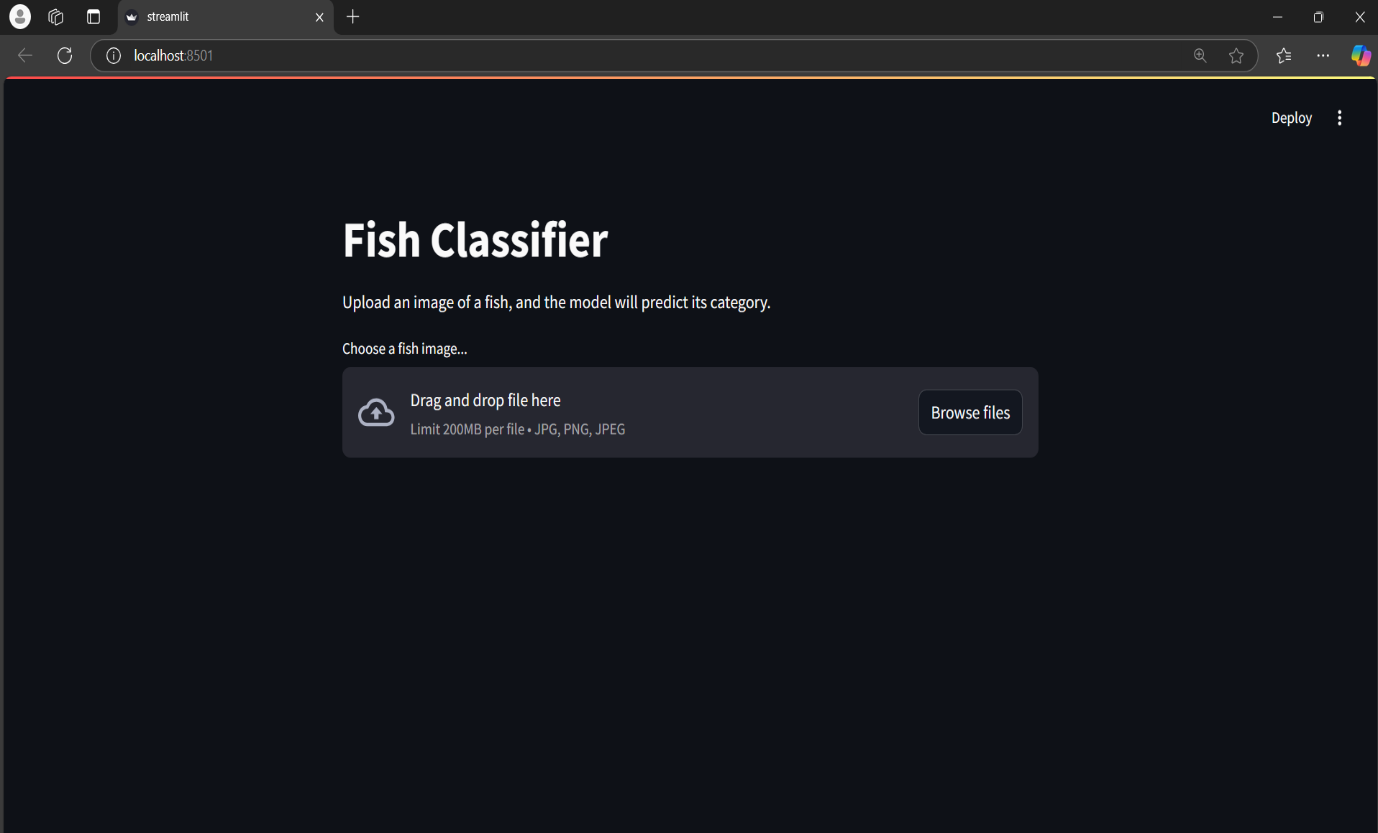
**Prepares the model for deployment** in a **Streamlit app**.  
**Ensures quick and efficient model loading** for real-time predictions.  
**Avoids retraining**—pre-trained model can be directly loaded.  
**Standard format (.keras)** for **portability** across different environments.

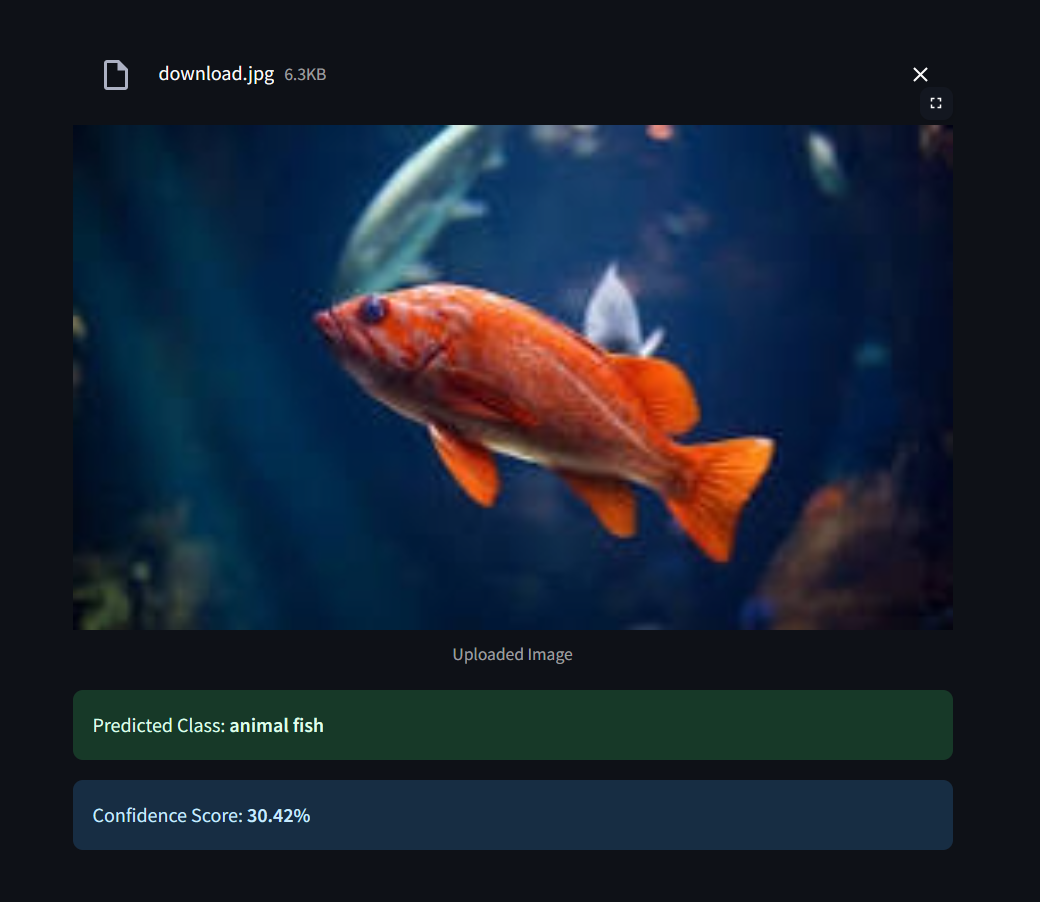
**Streamlit-Based Fish Classification App**



This **Streamlit app** allows users to upload a fish image and get a predicted category using a trained **ResNet50 deep learning model**. It loads the saved model, defines 11 fish categories, and provides a user-friendly interface for image uploads. The uploaded image is resized, normalized, and preprocessed to match the model’s input format. The model then predicts the fish species and displays the result along with a confidence score. This app enables **real-time fish classification** for researchers, marine biologists, and the seafood industry, making fish identification **fast, accurate, and accessible**.

**OUTPUT**

****

**AFTER UPLOADING A FISH IMAGE**

**CONCLUSION:**

This **Multiclass Fish Image Classification project** successfully implemented **deep learning models** to classify fish species with high accuracy. Using **Convolutional Neural Networks (CNNs) and Transfer Learning (ResNet50)**, the model was trained, fine-tuned, and evaluated for optimal performance. Techniques such as **early stopping, model checkpointing, confusion matrices, and classification reports** ensured robust training and validation. The best-performing model was deployed using **Streamlit**, allowing real-time fish species identification from uploaded images. This project demonstrates the power of **AI in marine research, seafood industry automation, and biodiversity conservation**, making fish classification **fast, efficient, and scalable**.