**Transfer learning**

**Loading and Saving Keras models, and saving using HDF5 format and exporting models using TensorFlow SavedModel format**

**Loading and Saving Keras Models**

Keras provides simple methods to save and load models. You can save the entire model, including the architecture, weights, and training configuration, or you can save just the model weights.

**Saving and Loading Model using HDF5 Format**

HDF5 is a file format that Keras uses by default for saving models. This format saves the model architecture, weights, and training configuration in a single file.

**Explanation**

1. **Saving a Model**: The `model.save()` method saves the model architecture, weights, and training configuration to a single HDF5 file by default. If you specify the `save\_format='tf'`, it will save the model in the TensorFlow SavedModel format, which is stored in a directory.
2. **Loading a Model**: The `keras.models.load\_model()` method loads the saved model, including the architecture, weights, and training configuration. You can then use this loaded model to make predictions or continue training.
3. **HDF5 Format**: This format is useful for saving models that need to be loaded back into Keras. It is a single-file format, making it easy to share and deploy.
4. **SavedModel Format**: This format is recommended for TensorFlow models as it supports a wider range of functionalities and is compatible with various TensorFlow tools. It saves the model in a directory, containing multiple files that represent the model's architecture, weights, and other necessary information.

**Here’s a detailed guide on the platforms and software required to execute the provided code for saving and loading Keras models:**

**Platform and Software Requirements**

**1. Platform**

The code can be executed on multiple platforms including:

* Windows
* macOS
* Linux

**a. Python:**

Ensure you have Python installed. Python 3.8 or later is recommended.

**b. TensorFlow:**

* TensorFlow is required for both model saving and loading.
* For TensorFlow 2.x, use the following command to install it:

**pip install tensorflow ->** in command promt

**c. IDE or Code Editor:**

* Jupyter Notebook (for interactive coding)
* Google Colab

**Saving and Loading using HDF5 Format**

**Explanation:**

HDF5 (Hierarchical Data Format version 5) is a binary data format that is particularly good for handling large amounts of data. It is widely used in scientific computing and has robust support for complex data types.

**Example Code:**

1. **Saving a Model in HDF5 Format:**

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Define a simple model

model = Sequential([

Dense(64, activation='relu', input\_shape=(100,)),

Dense(64, activation='relu'),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

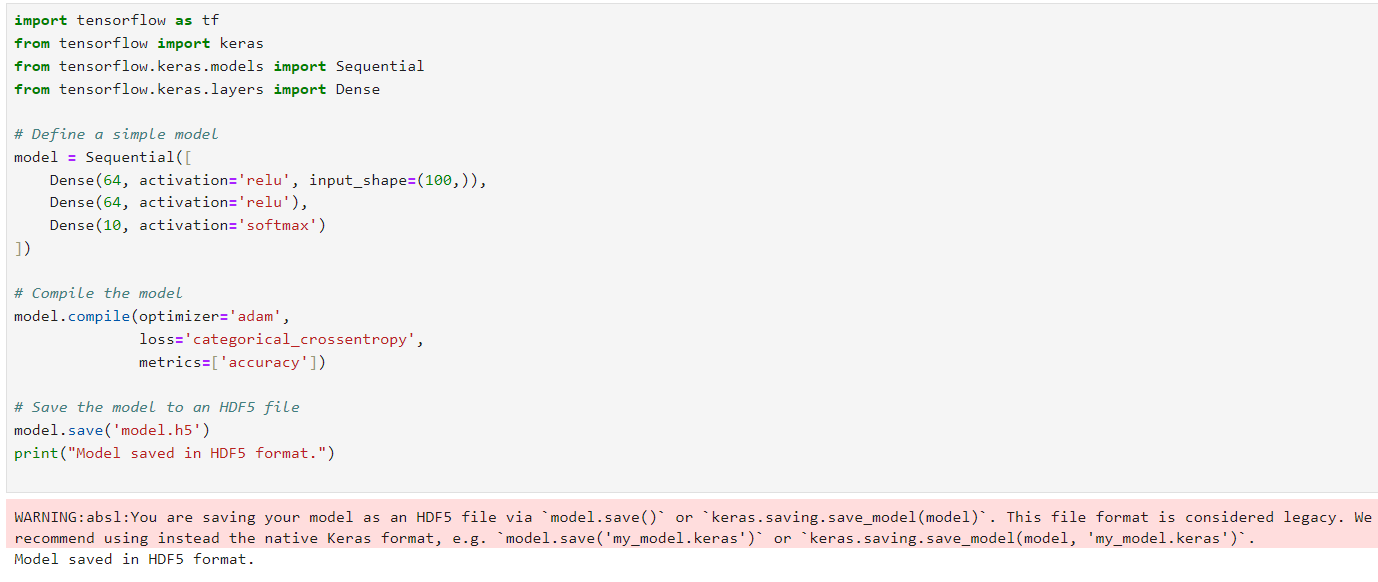
loss='categorical\_crossentropy',

metrics=['accuracy'])

# Save the model to an HDF5 file

model.save('model.h5')

print("Model saved in HDF5 format.")



1. **Loading a Model from HDF5 Format:**

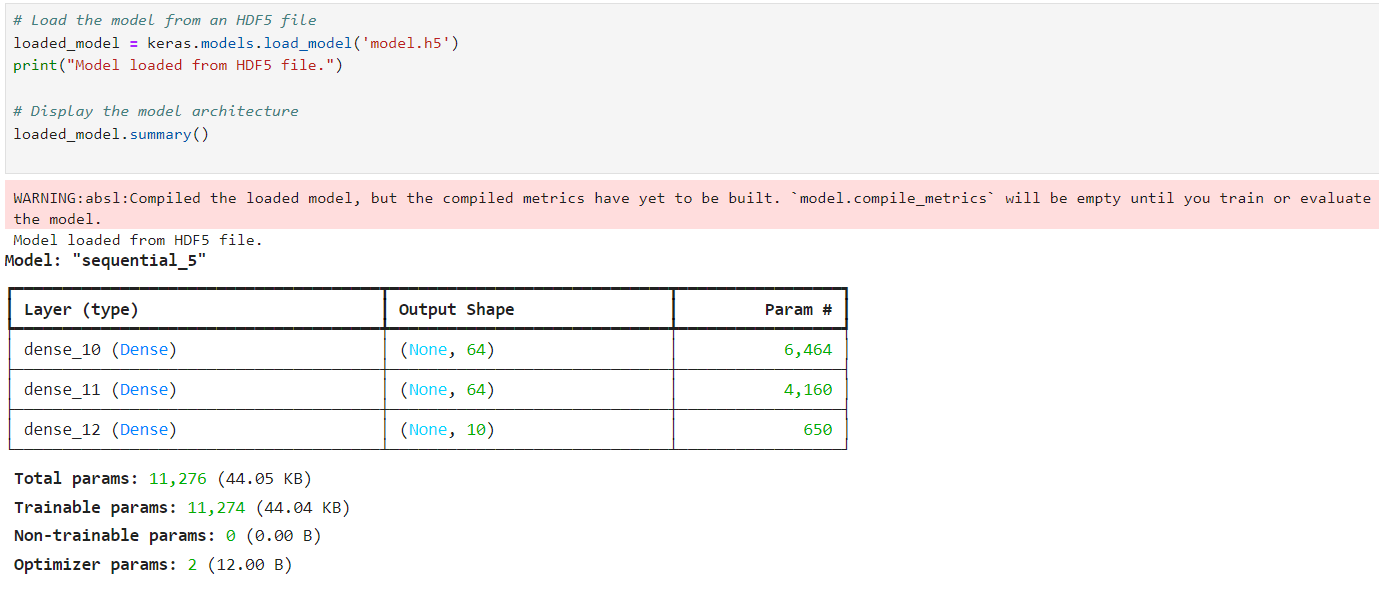
# Load the model from an HDF5 file

loaded\_model = keras.models.load\_model('model.h5')

print("Model loaded from HDF5 file.")

# Display the model architecture

loaded\_model.summary()



**Exporting Models using TensorFlow SavedModel Format**

**Explanation:**

The TensorFlow SavedModel format is the default format in TensorFlow 2.x. It allows for the complete model to be saved, including the architecture, weights, and optimizer state, making it highly portable and compatible with TensorFlow Serving.

**Example Code:**

**1.Saving a Model in TensorFlow SavedModel Format:**

# Define and compile the model (same as above)

model = Sequential([

Dense(64, activation='relu', input\_shape=(100,)),

Dense(64, activation='relu'),

Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

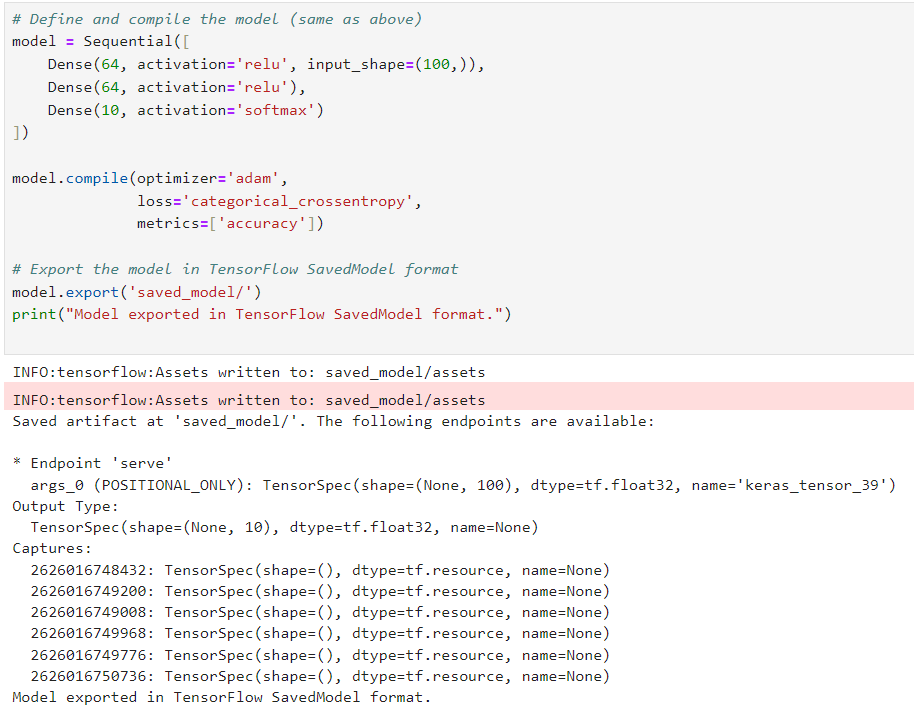
loss='categorical\_crossentropy',

metrics=['accuracy'])

# Export the model in TensorFlow SavedModel format

model.export('saved\_model/')

print("Model exported in TensorFlow SavedModel format.")



**2.Loading a Model from TensorFlow SavedModel Format:**

import tensorflow as tf

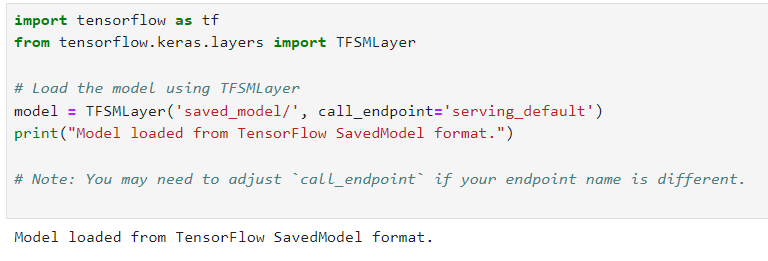
from tensorflow.keras.layers import TFSMLayer

# Load the model using TFSMLayer

model = TFSMLayer('saved\_model/', call\_endpoint='serving\_default')

print("Model loaded from TensorFlow SavedModel format.")

# Note: You may need to adjust `call\_endpoint` if your endpoint name is different.



**For reference purpose use below link**

<https://www.tensorflow.org/tutorials/keras/save_and_load>

**Leveraging pre-trained models:**

Leveraging pre-trained models is a great way to build on existing work and save time. Here are some common strategies for utilizing pre-trained models:

1. **Transfer Learning**: Use a pre-trained model as a starting point and fine-tune it on your specific dataset. This is especially useful if you have a smaller dataset. Common examples include using models like ResNet or VGG for image classification tasks and BERT or GPT for natural language processing.
2. **Feature Extraction**: Use the pre-trained model to extract features from your data and then use those features in a different model. For example, you might use a pre-trained CNN to extract features from images and then feed those features into a different classifier.
3. **Feature Reuse**: Instead of training a model from scratch, use the pre-trained model as part of your own architecture. For instance, you can add new layers on top of a pre-trained network to adapt it to a different task.
4. **Embedding Models**: For text data, pre-trained models like word2vec, GloVe, or sentence embeddings can be used to convert text into numerical vectors that can be fed into other machine learning models.
5. **Pre-trained APIs**: Many models are available through APIs provided by companies like OpenAI, Google Cloud, or AWS. These can be used for tasks like image recognition, text generation, or translation without needing to handle the model architecture or training process yourself.

<https://medium.com/@sruthy.sn91/leveraging-pretrained-models-for-faster-training-in-machine-learning-12e1704c274e>

**Removing top layer and adding new layers:**

To perform transfer learning by removing the top layer of a pre-trained model and adding new layers, follow these steps. I'll use the VGG16 model as an example, but the approach is similar for other models**.**

**Objective:**

* Load a pre-trained model (e.g., VGG16) without its top classification layer.
* Add new layers specific to your task (e.g., a different number of output classes).

**Example Code:**

import tensorflow as tf

from tensorflow.keras.applications import VGG16

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.models import Model

# Load the pre-trained VGG16 model without the top classification layer

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

# Freeze the base model's layers to prevent them from being updated during training

for layer in base\_model.layers:

layer.trainable = False

# Add new layers on top of the base model

x = base\_model.output

x = Flatten()(x) # Flatten the output to feed into the dense layers

x = Dense(512, activation='relu')(x) # New dense layer with 512 units

x = Dense(10, activation='softmax')(x) # Final classification layer for 10 classes

# Create the new model

model = Model(inputs=base\_model.input, outputs=x)

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Summary of the model to check the new layers

model.summary()

# Example code for training the model

# Assume you have training and validation data

# model.fit(train\_data, epochs=10, validation\_data=val\_data)

**Output:**

****

**Explanation:**

1. **Load Pre-trained Model**: ` VGG16` is loaded without the top layer (`include\_top=False `), which means we exclude the fully connected layers used for classification.
2. **Freeze Base Model**: By setting ` layer.trainable = False `, we ensure that the weights of the pre-trained layers are not updated during training.
3. **Add New Layers**:
   * ` Flatten `converts the 3D output of the base model to 1D.
   * ` Dense(512, activation='relu') `adds a new dense layer with 512 neurons and ReLU activation.
   * ` Dense(10, activation='softmax')`adds the final classification layer, where 10 represents the number of classes in your new task.
4. **Compile and Train**: The model is compiled with an optimizer, loss function, and metrics. You can then train the model using your dataset.

Replace ` train\_data `and ` val\_data ` with your actual training and validation datasets. Adjust the number of units in the dense layers and the number of output classes based on your specific task.

**Training the model:**

**Explanation:**

* **Load Pre-trained Model**: VGG16 without the top layer.
* **Freeze Layers**: Prevent updating pre-trained weights.
* **Add New Layers**: Customize the model for your specific task.
* **Compile Model**: Define optimizer, loss, and metrics.
* **Prepare Data**: Load and preprocess training and validation data.
* **Train Model**: Train using data generators.
* **Save Model**: Save the trained model.
* **Evaluate Model**: Assess performance on validation data.

**Full Example Code:**

Here’s the full example code integrating all steps:

import tensorflow as tf

from tensorflow.keras.applications import VGG16

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.models import Model

from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Load the pre-trained VGG16 model without the top classification layer

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

# Freeze the base model's layers

for layer in base\_model.layers:

layer.trainable = False

# Add new layers on top of the base model

x = base\_model.output

x = Flatten()(x)

x = Dense(512, activation='relu')(x)

x = Dense(10, activation='softmax')(x) # Assuming 10 classes

# Create the new model

model = Model(inputs=base\_model.input, outputs=x)

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Summary of the model

model.summary()

# Directory paths

train\_dir = 'path\_to\_your\_train\_data'

val\_dir = 'path\_to\_your\_val\_data'

# Image data generators for data augmentation

train\_datagen = ImageDataGenerator(

rescale=1./255,

rotation\_range=40,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True,

fill\_mode='nearest'

)

val\_datagen = ImageDataGenerator(rescale=1./255)

# Data generators

train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=(224, 224),

batch\_size=32,

class\_mode='categorical'

)

val\_generator = val\_datagen.flow\_from\_directory(

val\_dir,

target\_size=(224, 224),

batch\_size=32,

class\_mode='categorical'

)

# Train the model

history = model.fit(

train\_generator,

steps\_per\_epoch=train\_generator.samples // train\_generator.batch\_size,

epochs=10, # Adjust the number of epochs as needed

validation\_data=val\_generator,

validation\_steps=val\_generator.samples // val\_generator.batch\_size

)

# Save the model in HDF5 format

model.save('my\_model.h5')

# Save the model in TensorFlow SavedModel format

model.save('my\_model\_saved', save\_format='tf')

# Evaluate the model on the validation data

val\_loss, val\_acc = model.evaluate(val\_generator, steps=val\_generator.samples // val\_generator.batch\_size)

print(f'Validation Loss: {val\_loss}')

print(f'Validation Accuracy: {val\_acc}')

**Output:**



**Fine-tuning model**

Fine-tuning involves unfreezing some or all of the layers of a pre-trained model so that the weights can be updated during training. This allows you to adapt the model more closely to your specific task. Here's an explanation and a simple example code for fine-tuning a pre-trained model.

**Steps for Fine-Tuning a Model**

1. **Load Pre-trained Model:** Load a pre-trained model without the top classification layer.
2. **Add New Layers:** Add new layers specific to your task on top of the base model.
3. **Freeze the Base Model:** Initially, freeze all the layers of the base model.
4. **Compile and Train:** Compile and train the new layers with the base model frozen.
5. **Unfreeze Layers:** Unfreeze some or all of the layers of the base model.
6. **Re-compile and Train:** Re-compile the model and train again with a lower learning rate to fine-tune the pre-trained layers.

**Example Code with Actual Paths:**

import tensorflow as tf

from tensorflow.keras.applications import VGG16

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.models import Model

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import os

# Load the pre-trained VGG16 model without the top classification layer

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

# Freeze the base model's layers

for layer in base\_model.layers:

layer.trainable = False

# Add new layers on top of the base model

x = base\_model.output

x = Flatten()(x)

x = Dense(512, activation='relu')(x)

x = Dense(10, activation='softmax')(x) # Assuming 10 classes

# Create the new model

model = Model(inputs=base\_model.input, outputs=x)

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Replace with actual directory paths

train\_dir = 'C:/Users/YourUsername/Path\_to\_Train\_Data'

val\_dir = 'C:/Users/YourUsername/Path\_to\_Val\_Data'

# Ensure directories exist

if not os.path.isdir(train\_dir):

raise FileNotFoundError(f"The directory {train\_dir} does not exist.")

if not os.path.isdir(val\_dir):

raise FileNotFoundError(f"The directory {val\_dir} does not exist.")

# Image data generators for data augmentation

train\_datagen = ImageDataGenerator(

rescale=1./255,

rotation\_range=40,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True,

fill\_mode='nearest'

)

val\_datagen = ImageDataGenerator(rescale=1./255)

# Data generators

train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=(224, 224),

batch\_size=32,

class\_mode='categorical'

)

val\_generator = val\_datagen.flow\_from\_directory(

val\_dir,

target\_size=(224, 224),

batch\_size=32,

class\_mode='categorical'

)

# Train the model with frozen base layers

model.fit(

train\_generator,

steps\_per\_epoch=train\_generator.samples // train\_generator.batch\_size,

epochs=10, # Adjust the number of epochs as needed

validation\_data=val\_generator,

validation\_steps=val\_generator.samples // val\_generator.batch\_size

)

# Unfreeze some layers of the base model

for layer in base\_model.layers[-4:]: # Unfreeze the last 4 layers as an example

layer.trainable = True

# Re-compile the model with a lower learning ratemodel.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=1e-5),

# Lower learning rate

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Fine-tune the model

model.fit(

train\_generator,

steps\_per\_epoch=train\_generator.samples // train\_generator.batch\_size,

epochs=10, # Adjust the number of epochs as needed

validation\_data=val\_generator,

validation\_steps=val\_generator.samples // val\_generator.batch\_size

)

# Save the fine-tuned model

model.save('fine\_tuned\_model.h5')

# Evaluate the fine-tuned model

val\_loss, val\_acc = model.evaluate(val\_generator, steps=val\_generator.samples // val\_generator.batch\_size)

print(f'Validation Loss: {val\_loss}')

print(f'Validation Accuracy: {val\_acc}')

**Ensure the Correct Paths**

1. **Update Paths**: Replace 'C:/Users/YourUsername/Path\_to\_Train\_Data' and 'C:/Users/YourUsername/Path\_to\_Val\_Data' with the actual paths to your training and validation data directories.
2. **Check Directory Structure**: Ensure the directory structure follows the format required by flow\_from\_directory. Each class should have its own subdirectory within the training and validation directories.

**For example:**

train\_dir/

class1/

img1.jpg

img2.jpg

...

class2/

img1.jpg

img2.jpg

...

...

val\_dir/

class1/

img1.jpg

img2.jpg

...

class2/

img1.jpg

img2.jpg

...

...

1. **Verify Paths**: Before running the script, verify that the directories exist and contain the necessary subdirectories and image files.

**Explanation**

1. **Load Pre-trained Model**: We load the VGG16 model pre-trained on ImageNet, excluding its top layer, to use its convolutional base.
2. **Freeze Base Layers**: Initially, we freeze all the layers of the base model to retain the pre-trained weights.
3. **Add New Layers**: We add new dense layers for our specific classification task.
4. **Compile and Train (Frozen Base)**: We compile and train the model, training only the new layers while keeping the base model's layers frozen.
5. **Unfreeze Layers**: After training the new layers, we unfreeze some or all of the base model's layers. In this example, we unfreeze the last 4 layers.
6. **Re-compile with Lower Learning Rate**: We re-compile the model with a lower learning rate to fine-tune the pre-trained layers. Fine-tuning with a lower learning rate prevents large updates that could disrupt the pre-trained weights.
7. **Fine-tune the Model**: We train the entire model again, allowing the newly unfrozen layers to adapt to the new task while keeping the rest of the base model relatively stable.
8. **Save and Evaluate**: We save the fine-tuned model and evaluate its performance on the validation data.

Adjust the number of layers to unfreeze and the learning rate based on your specific task and dataset. Fine-tuning requires careful tuning of these parameters to achieve the best results.

**For reference purpose use below link**

<https://medium.com/@amanatulla1606/fine-tuning-the-model-what-why-and-how-e7fa52bc8ddf>

**Evaluate and deploy:**

**Explanation**

1. **Evaluate the Model**:
   * The ` evaluate ` method is used to measure the model's performance on the validation dataset. It returns the loss and accuracy.
2. **Save the Model:**
   * The ` model.save ` method can save the model in both HDF5 and TensorFlow SavedModel formats. The HDF5 format is suitable for many applications, while the SavedModel format is required for TensorFlow Serving.
3. **Deploy the Model:**
   * TensorFlow Serving is a flexible, high-performance serving system for machine learning models designed for production environments. It handles the deployment of new versions of models seamlessly.
   * The ` tensorflow\_model\_server ` command starts the model server, specifying the REST API port, model name, and model base path.
4. **Client Request:**
   * Using Python’s ` requests ` library, you can send HTTP requests to the model server for predictions.
   * The image is preprocessed to match the input format expected by the model and sent in a JSON format.
   * The server responds with predictions, which are then parsed and displayed.

**Evaluation**

After training, you need to evaluate the model on a validation or test dataset to understand its performance.

# Evaluate the fine-tuned model

val\_loss, val\_acc = model.evaluate(val\_generator, steps=val\_generator.samples // val\_generator.batch\_size)

print(f'Validation Loss: {val\_loss}')

print(f'Validation Accuracy: {val\_acc}')

**Save and Export the Model**

Save the model in a format that is suitable for deployment.

# Save the fine-tuned model in HDF5 format

model.save('fine\_tuned\_model.h5')

# Save the model in the TensorFlow SavedModel format

model.save('fine\_tuned\_model')

**Deployment**

Deploying the model typically involves serving it using a model serving tool like TensorFlow Serving. Here’s a basic guide to deploying using TensorFlow Serving.

1. **Install TensorFlow Serving**: Follow the TensorFlow Serving installation guide.
2. **Export the Model**: Ensure the model is saved in the TensorFlow SavedModel format.
3. **Serve the Model**: Use TensorFlow Serving to serve the model.

# Serve the model

tensorflow\_model\_server --rest\_api\_port=8501 --model\_name=my\_model --model\_base\_path="/path/to/fine\_tuned\_model" 🡪 **in command prompt**

**Client Request Example**:

Here’s a simple example of how to make predictions using the served model.

import requests

import json

import numpy as np

from tensorflow.keras.preprocessing import image

# Load and preprocess an image

img\_path = 'path\_to\_your\_image.jpg'

img = image.load\_img(img\_path, target\_size=(224, 224))

img\_array = image.img\_to\_array(img)

img\_array = np.expand\_dims(img\_array, axis=0)

img\_array = img\_array / 255.0

# Prepare the data in the format expected by the server

data = json.dumps({"signature\_name": "serving\_default", "instances": img\_array.tolist()})

# Make a request to the TensorFlow Serving server

response = requests.post('http://localhost:8501/v1/models/my\_model:predict', data=data)

predictions = json.loads(response.text)['predictions']

print(predictions)