**TF Hub**

**Accessing Tensorflow Hub using PIP and the Python API**

TensorFlow Hub is a repository of pre-trained models that can be easily used for various machine learning tasks. To access TensorFlow Hub using PIP and the Python API, follow these steps:

**Install TensorFlow Hub**

First, you need to install TensorFlow Hub. You can do this using pip:

**pip install tensorflow-hub 🡪** in command prompt

**Example: Using a Pre-trained Model from TensorFlow Hub**

Here’s an example of how to use a pre-trained model from TensorFlow Hub for image classification:

**1.Import Necessary Libraries**:

import tensorflow as tf

import tensorflow\_hub as hub

import numpy as np

from tensorflow.keras.preprocessing import image

import matplotlib.pyplot as plt

import json

import requests

**2.Load a Pre-trained Model**:

# Load a pre-trained model from TensorFlow Hub

model\_url = "https://tfhub.dev/google/imagenet/mobilenet\_v2\_100\_224/classification/5"

model = tf.keras.Sequential([

hub.KerasLayer(model\_url, input\_shape=(224, 224, 3))

])

**3.Load and Preprocess an 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 # Normalize the image

**4.Make Predictions**:

# Make predictions

predictions = model.predict(img\_array)

predicted\_class = np.argmax(predictions[0], axis=-1)

# Get human-readable labels

labels\_path = tf.keras.utils.get\_file(

'ImageNetLabels.txt',

'https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')

with open(labels\_path) as f:

labels = f.read().splitlines()

# Print the predicted class

print(f'Predicted class: {labels[predicted\_class]}')

**5.Visualize the Image and Prediction**:

# Plot the image and predicted class

plt.imshow(img)

plt.title(f'Predicted class: {labels[predicted\_class]}')

plt.axis('off')

plt.show()

**Full Example Code**

Here’s the complete code to load a pre-trained model from TensorFlow Hub, make a prediction on an image, and visualize the output.

import tensorflow as tf

import tensorflow\_hub as hub

import numpy as np

from tensorflow.keras.preprocessing import image

import matplotlib.pyplot as plt

import json

# Load a pre-trained model from TensorFlow Hub

model\_url = "https://tfhub.dev/google/imagenet/mobilenet\_v2\_100\_224/classification/5"

model = tf.keras.Sequential([

hub.KerasLayer(model\_url, input\_shape=(224, 224, 3))

])

# Load and preprocess an image

img\_path = 'path\_to\_your\_image.jpg' # Replace with the actual path to your image

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 # Normalize the image

# Make predictions

predictions = model.predict(img\_array)

predicted\_class = np.argmax(predictions[0], axis=-1)

# Get human-readable labels

labels\_path = tf.keras.utils.get\_file(

'ImageNetLabels.txt',

'https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')

with open(labels\_path) as f:

labels = f.read().splitlines()

# Print the predicted class

print(f'Predicted class: {labels[predicted\_class]}')

# Plot the image and predicted class

plt.imshow(img)

plt.title(f'Predicted class: {labels[predicted\_class]}')

plt.axis('off')

plt.show()

**Explanation**

1. **Install Dependencies**:
   * Ensure TensorFlow Hub is installed using ` pip install tensorflow-hub `.
2. **Load Pre-trained Model**:
   * The MobileNet V2 model trained on ImageNet is loaded from TensorFlow Hub.
3. **Load and Preprocess Image**:
   * The image is loaded, resized to 224x224 pixels, converted to a NumPy array, expanded to add a batch dimension, and normalized.
4. **Make Predictions**:
   * The model predicts the class of the image, and the predicted class index is extracted.
5. **Get Human-readable Labels**:
   * ImageNet class labels are downloaded and read.
6. **Display Prediction**:
   * The image is displayed with the predicted class label as the title.

**OUTPUT:**

When you run the above code, you should see the following:

1. **Printed Output**:
   * The predicted class will be printed to the console. For example:

Predicted class: Labrador retriever

1. **Visual Output**:

A window will pop up displaying the image with the predicted class label as the title.

**Predicted class: Labrador retriever**

(Note: Replace the placeholder image URL with the path to your actual image to see the real output.)

**Model formats i.e. TensorFlow SavedModel vs. TF2 SavedModel**

The terms **TensorFlow SavedModel** and **TF2 SavedModel** often come up in discussions about TensorFlow model formats. Understanding the differences can help in choosing the right format for saving and deploying models. Here’s a clear breakdown:

**TensorFlow SavedModel vs. TF2 SavedModel**

**1. TensorFlow SavedModel (TF1.x)**

* **Version**: Used in TensorFlow 1.x.
* **Format**: The SavedModel format for TensorFlow 1.x is designed to store models for both training and inference. It includes:
  + A ` saved\_model.pb ` file which contains the model architecture and computation graph.
  + A ` variables ` directory containing checkpoint files with model weights.
* **Graph Structure**: It uses the TensorFlow 1.x graph-based approach where operations and tensors are explicitly defined as a static computation graph.
* **Serialization**: The SavedModel format allows for serialization of the entire model, including the graph, weights, and metadata, which can be restored later for inference or continued training.

**Example code for Saving and Loading a Model in TensorFlow 1.x**

import tensorflow as tf

# Define a simple model

class SimpleModel(tf.Module):

def \_\_init\_\_(self):

self.w = tf.Variable(tf.random.normal([32, 10]), name='w')

self.b = tf.Variable(tf.random.normal([10]), name='b')

@tf.function(input\_signature=[tf.TensorSpec(shape=[None, 32], dtype=tf.float32)])

def \_\_call\_\_(self, x):

return tf.matmul(x, self.w) + self.b

# Instantiate and save the model

model = SimpleModel()

tf.saved\_model.save(model, 'path\_to\_saved\_model')

# Load the model

loaded\_model = tf.saved\_model.load('path\_to\_saved\_model')

# Test the model

test\_input = tf.random.normal([1, 32])

print(loaded\_model(test\_input))

**OUTPUT:**



1. **TF2 SavedModel (TF2.x)**

* **Version**: Introduced with TensorFlow 2.x.
* **Format**: The TF2 SavedModel format is an evolution of the original SavedModel format. It maintains compatibility with TensorFlow 1.x SavedModel but includes enhancements for TF2.x features. It includes:
  + A ` saved\_model.pb ` file (or ` saved\_model.pbtxt ` for a text version) containing the model architecture and computation graph.
  + A variables directory with checkpoint files.
* **Eager Execution**: TF2.x emphasizes eager execution, where operations are executed immediately as they are called within Python. The SavedModel format supports both eager execution and graph-based execution.
* **Keras Integration**: TensorFlow 2.x introduces tighter integration with Keras, allowing for easier model saving and loading using the Keras API. Models created with tf.keras.Model can be saved and loaded seamlessly using the SavedModel format.
* **Enhanced Features**: It includes support for newer TensorFlow features, such as custom training loops, the use of ` tf.function` for graph execution, and better integration with TensorFlow Hub and TensorFlow Serving.

**Example Code for Saving and Loading Models Saving and Loading a Model in TensorFlow 2.x (TF2 SavedModel)**

import tensorflow as tf

from tensorflow.keras import layers, models

# Create a simple model

model = models.Sequential([

layers.Dense(10, activation='relu', input\_shape=(32,)),

layers.Dense(3, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy')

# Save the model using .keras extension

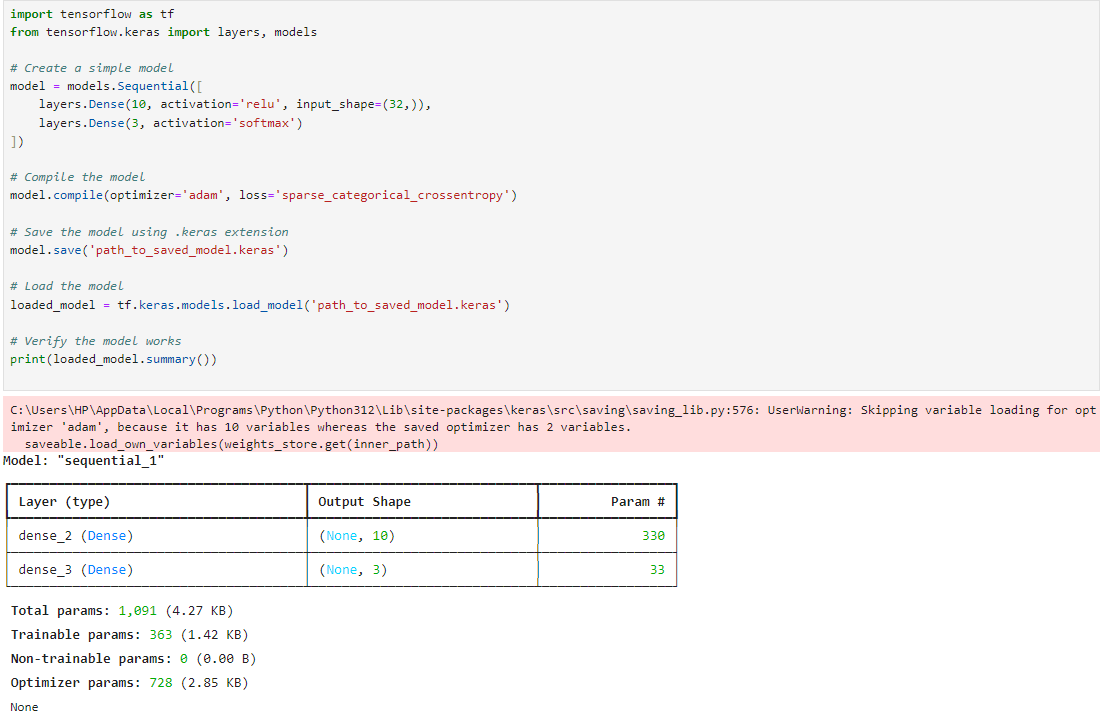
model.save('path\_to\_saved\_model.keras')

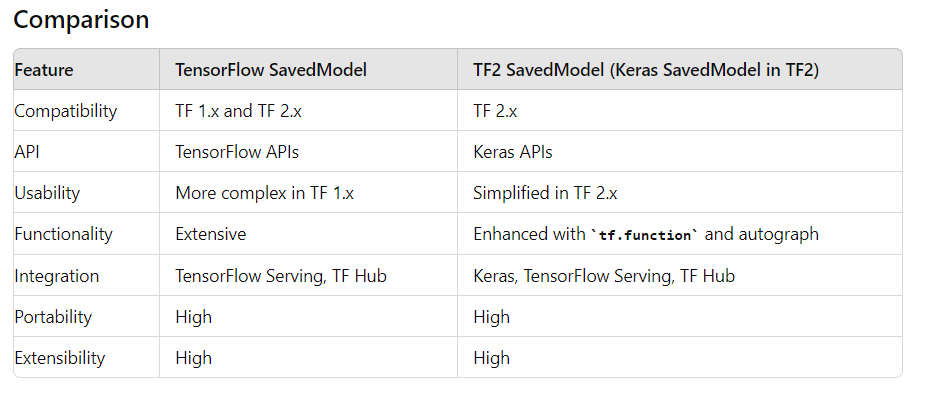
# Load the model

loaded\_model = tf.keras.models.load\_model('path\_to\_saved\_model.keras')

# Verify the model works

print(loaded\_model.summary())

**OUTPUT:**



**Key features of the TensorFlow SavedModel format**:

* **Compatibility**: Supports both TensorFlow 1.x and TensorFlow 2.x.
* **Flexibility**: Can store the complete TensorFlow program, including variables, assets, and the computation graph.
* **Portability**: Allows models to be shared and loaded across different environments.
* **Extensibility**: Supports various TensorFlow components like Estimators, Keras models, and custom models.

**Understanding input and output requirements using Model Signature and inspecting model layers using summary()**

Understanding the input and output requirements of a model using its Model Signature and inspecting the model layers using the ` summary() `method are important steps in working with TensorFlow/Keras models.

**Model Signature**

A **Model Signature** provides a clear specification of the inputs and outputs of a TensorFlow model, including their shapes and types. It is particularly useful when exporting a model for serving or deployment. In TensorFlow 2.x, the ` tf.saved\_model.save ` function allows you to define a signature.

**Key Points:**

* **Input Signature**: Defines the expected input tensor(s), including their shapes and data types.
* **Output Signature**: Defines the output tensor(s) in a similar manner.
* **Usage**: Signatures help ensure that the model receives and produces the expected data formats, which is crucial for deployment in production environments.

**Here's an example of how to save a model with a signature:**

import tensorflow as tf

# Define a simple model

class SimpleModel(tf.Module):

def \_\_init\_\_(self):

self.w = tf.Variable(tf.random.normal([32, 10]), name='w')

self.b = tf.Variable(tf.random.normal([10]), name='b')

@tf.function(input\_signature=[tf.TensorSpec(shape=[None, 32], dtype=tf.float32)])

def \_\_call\_\_(self, x):

return tf.matmul(x, self.w) + self.b

# Instantiate and save the model

model = SimpleModel()

tf.saved\_model.save(model, 'path\_to\_saved\_model')

# Load the model

loaded\_model = tf.saved\_model.load('path\_to\_saved\_model')

# Print the signature

print(list(loaded\_model.signatures.keys()))

print(loaded\_model.signatures['serving\_default'])

**OUTPUT:**



**Inspecting Model Layers Using summary()**

The ` summary() `method is available for Keras models and provides a concise overview of the model's architecture, including the layers, output shapes, and the number of parameters.

**Key Points:**

* **Layer Types**: Identifies each layer type in the model (e.g., Dense, Conv2D).
* **Output Shapes**: Shows the shape of the output tensor(s) for each layer.
* **Parameter Counts**: Displays the number of parameters (weights and biases) in each layer, including trainable and non-trainable parameters.

**Here's an example of how to use the ` summary() `method with a Keras model:**

import tensorflow as tf

from tensorflow.keras import layers, models

# Create a simple model

model = models.Sequential([

layers.Dense(10, activation='relu', input\_shape=(32,)),

layers.Dense(3, activation='softmax')

])

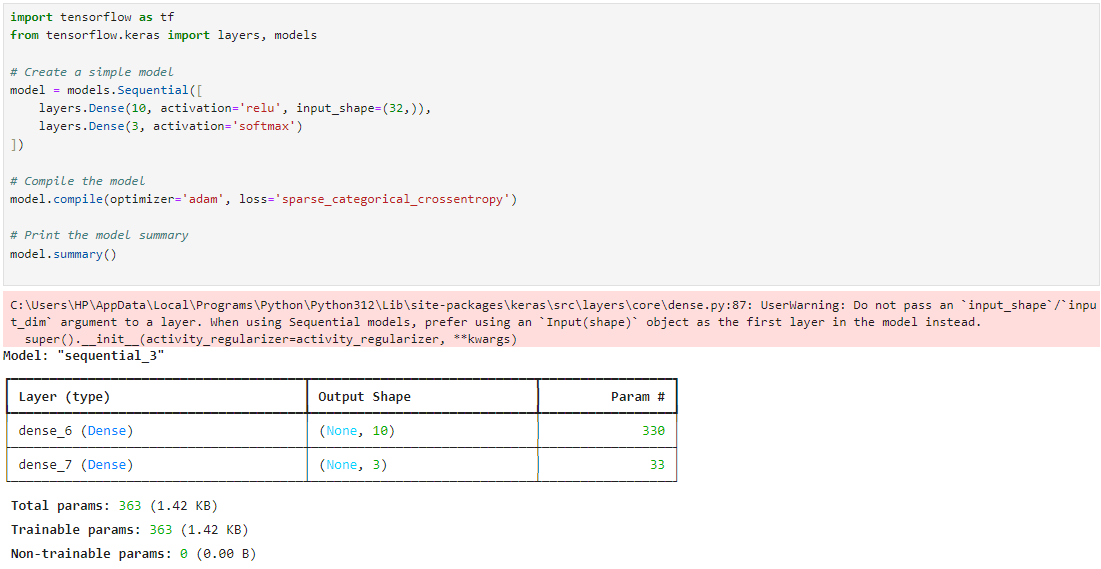
# Compile the model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy')

# Print the model summary

model.summary()

**OUTPUT:**



**Notes**

1. **Model Signature**: When saving a model with a signature, you specify the expected input and output tensors, which helps in deploying the model correctly.
2. **Model Summary**: The ` summary() `method is useful for getting an overview of the model architecture and ensuring that all layers are defined as expected.