Coral dev Board

Introducción

La Coral Dev Board es una computadora de placa única que contiene un coprocesador Edge TPU y está diseñada para la prototipación de productos de aprendizaje automático en dispositivos locales. El Edge TPU es un procesador de unidades de procesamiento tensorial (TPU) que está optimizado para ejecutar modelos de aprendizaje automático de forma eficiente. La Coral Dev Board es versátil y puede utilizarse en una variedad de aplicaciones.

En el ámbito de las unidades de procesamiento tensorial (TPU), la inferencia se refiere al proceso de usar un modelo de aprendizaje automático entrenado para predecir un resultado a partir de una nueva entrada. Por ejemplo, un modelo de reconocimiento de imágenes entrenado en un conjunto de datos de gatos y perros podría usarse para inferir si una nueva imagen muestra un gato o un perro.

Las TPU están diseñadas específicamente para la inferencia de aprendizaje automático. Son mucho más eficientes que las CPU y las GPU tradicionales para realizar operaciones matemáticas de gran escala, que son necesarias para las redes neuronales artificiales. Esto significa que las TPU pueden realizar inferencias a velocidades mucho más rápidas, lo que las hace ideales para aplicaciones que requieren un procesamiento de inferencia en tiempo real, como la detección de objetos en cámaras de seguridad o la traducción automática.

Hay dos tipos principales de inferencia de TPU:

- **Inferencia en línea:** La inferencia en línea se realiza en un servidor central. El modelo de aprendizaje automático se almacena en el servidor y se usa para realizar inferencias a medida que se reciben nuevas entradas.

- **Inferencia fuera de línea:** La inferencia fuera de línea se realiza en un dispositivo perimetral, como un teléfono inteligente o una cámara de seguridad. El modelo de aprendizaje automático se descarga al dispositivo y se usa para realizar inferencias localmente.

Las TPU se pueden usar para una amplia gama de aplicaciones de inferencia de aprendizaje automático, incluyendo:

- **Reconocimiento de imágenes:** Las TPU se pueden usar para reconocer objetos, personas y rostros en imágenes.
- **Reconocimiento de voz:** Las TPU se pueden usar para reconocer palabras y frases habladas.
- **Traducción automática:** Las TPU se pueden usar para traducir texto de un idioma a otro.
- **Recomendación de productos:** Las TPU se pueden usar para recomendar productos a los usuarios en función de sus intereses.
- **Clasificación de spam:** Las TPU se pueden usar para clasificar el correo electrónico como spam o no spam.

Las TPU están revolucionando el campo del aprendizaje automático, haciendo que las aplicaciones de inferencia sean más rápidas, eficientes y asequibles.

Características Clave de la Tarjeta Coral dev Board:

- **Procesador Edge TPU de Google:** El Edge TPU es un procesador especializado para la visión artificial que ofrece un rendimiento líder en el sector. Puede realizar hasta 4 billones de operaciones por segundo (TOPS) con un consumo de energía de solo 0,5 vatios por TOPS.
- **Módulo de Sistema Extraíble (SoM):** La Dev Board cuenta con un módulo de sistema extraíble que permite escalar desde el prototipo hasta la producción. Este módulo se puede quitar de la placa base, ordenarse en grandes cantidades y luego integrarse en hardware personalizado.
- **Conectividad Inalámbrica:** Ofrece conectividad Wi-Fi 2x2 MIMO (802.11b/g/n/ac 2.4/5 GHz) y Bluetooth 4.2, lo que facilita la comunicación inalámbrica para una variedad de aplicaciones.

- **Sistema Operativo y Framework Soportados:** Ejecuta una versión derivada de Debian Linux llamada Mendel y es compatible con el framework TensorFlow Lite. Esto proporciona a los desarrolladores la flexibilidad de utilizar herramientas familiares de Linux y compilar modelos de TensorFlow Lite para ejecutarse en el Edge TPU.
- **Desempeño:** La Dev Board puede ejecutar modelos de visión móvil de última generación, como MobileNet v2, a casi 400 fotogramas por segundo, manteniendo una eficiencia energética destacada.

Componentes Claves de la Tarjeta Coral dev Board:

1. Edge TPU System-on-Module (SoM).
2. Puertos de Entrada/Salida (GPIO) que permiten la conexión de periféricos y dispositivos externos. Estos puertos son esenciales para la expansión y personalización de la funcionalidad de la tarjeta según las necesidades del proyecto.
3. Alimentación y puerto USB: La Dev Board cuenta con una interfaz de alimentación para suministrar energía al sistema. También incluye puertos USB para conectar dispositivos periféricos, como cámaras, teclados o unidades de almacenamiento externas.
4. Botones de Encendido y reinicio
5. LED 's para el control y la indicación de estado del sistema.
6. Un conector HDMI de salida para la reproducción de vídeo.
7. Un conector de audio estéreo para la entrada y salida de audio.
8. Un conector micro SD para almacenar el sistema operativo.
9. Un puerto Ethernet Gigabit para la comunicación de red y la transferencia de datos.

Coral vs Raspberry Pi

La Coral Dev Board y la Raspberry Pi son dos de los llamados computadores monoplaca (SBC) que se pueden utilizar para una variedad de tareas, incluyendo aprendizaje automático e inteligencia artificial. Sin embargo, tienen diferentes

fortalezas y debilidades, lo que las hace más adecuadas para diferentes aplicaciones.

Coral Dev Board

La Coral Dev Board es una SBC especializada que está diseñada específicamente para aplicaciones de IA y aprendizaje automático. Cuenta con un coprocesador Google Edge TPU, que es un ASIC personalizado que puede acelerar las cargas de trabajo de IA. Esto hace que la Coral Dev Board sea ideal para tareas como el reconocimiento de imágenes, la detección de objetos y el procesamiento del lenguaje natural.

Raspberry Pi

La Raspberry Pi es una SBC más generalista que se puede utilizar para una gama más amplia de tareas, incluyendo la navegación web, la reproducción de medios y los juegos. También es una opción popular para proyectos de domótica y robótica.

Tabla de comparación

Característica	Coral Dev Board	Raspberry Pi
Procesador	Google Edge TPU, quad-core Cortex-A53	Quad-core Cortex-A72
RAM	4GB	1-4GB
Almacenamiento	8 GB eMMC	16-32 GB eMMC
Puertos	USB 2.0, USB 3.0, USB-C, GPIO, HDMI	USB 2.0, USB 3.0, HDMI, GPIO
Precio	$ 630.000 COP	$ 200.000- 350.000 COP

¿Cuál elegir?

La mejor SBC para ti dependerá de tus necesidades específicas. Si estás principalmente interesado en aplicaciones de aprendizaje automático e IA, entonces la Coral Dev Board es una buena opción. Sin embargo, si necesitas una SBC más generalista, entonces la Raspberry Pi es una mejor opción.

- **Elige la Coral Dev Board si:**
  - Necesitas ejecutar aplicaciones de IA y aprendizaje automático
  - Necesitas el mejor rendimiento posible para tareas de IA
  - Estás dispuesto a pagar un precio premium por una SBC especializada

- **Elige la Raspberry Pi si:**
  - Necesitas una SBC generalista para una variedad de tareas
  - Estás ajustado de presupuesto
  - Necesitas una SBC con una amplia gama de puertos y opciones de conectividad

Aplicaciones

- **Salud:** Google Coral puede utilizarse para desarrollar dispositivos médicos para diagnósticos, monitoreo de pacientes y descubrimiento de medicamentos. El Edge TPU puede procesar grandes cantidades de datos médicos localmente, garantizando la privacidad y reduciendo la necesidad de conectividad a la nube.
- **Agricultura:** Los dispositivos Coral pueden utilizarse para crear sistemas de agricultura inteligente para monitorear la salud de los cultivos, rastrear las condiciones del suelo y optimizar la irrigación. El Edge TPU puede procesar datos en tiempo real de sensores y cámaras, permitiendo una toma de decisiones más rápida y prácticas agrícolas más eficientes.
- **Ciudades Inteligentes:** Google Coral puede utilizarse para desarrollar aplicaciones para ciudades inteligentes, como la gestión del tráfico, el manejo de residuos y el monitoreo del consumo de energía. El Edge TPU puede procesar grandes cantidades de datos de diversos sensores y

cámaras, permitiendo la toma de decisiones en tiempo real y mejorando la eficiencia general en entornos urbanos.

- ➔ **Visión por Computadora:** Los dispositivos de Google Coral, como la Coral Dev Board y la Coral Dev Board Mini, pueden utilizarse para construir aplicaciones de visión por computadora para la detección de objetos, reconocimiento facial y análisis de expresiones faciales. El Edge TPU puede procesar datos visuales localmente, permitiendo un procesamiento más rápido y consciente de la privacidad.
- ➔ **Aplicaciones de IA Personalizadas:** Google Coral puede utilizarse para desarrollar aplicaciones de inteligencia artificial personalizadas para diversas industrias, incluyendo manufactura, venta al por menor y sistemas autónomos.

Estos son solo algunos ejemplos de la amplia gama de aplicaciones que se pueden construir utilizando Google Coral. El Edge TPU y las herramientas de software de la plataforma facilitan la prototipación y escalado de productos de IA local en diversas industrias.

Conectar la Coral dev Board

Necesitarás:

- ➢ Una tarjeta Coral dev Board.
- ➢ Una Micro SD mínima de 8 GB.
- ➢ Un cable Ethernet o (conexión Wi-Fi disponible).
- ➢ Una fuente de alimentación USB-C (2-3 A / 5 V), como un cargador de teléfono.
- ➢ Un cable USB-C a USB-A (para conectarse a su computadora).
- ➢ Un computador.

- **Ahora vamos a preparar la tarjeta SD**

Al utilizar la tarjeta por primera vez o en caso de querer borrar todo el sistema y los datos del usuario se tiene que instalar la imagen del sistema a la placa.

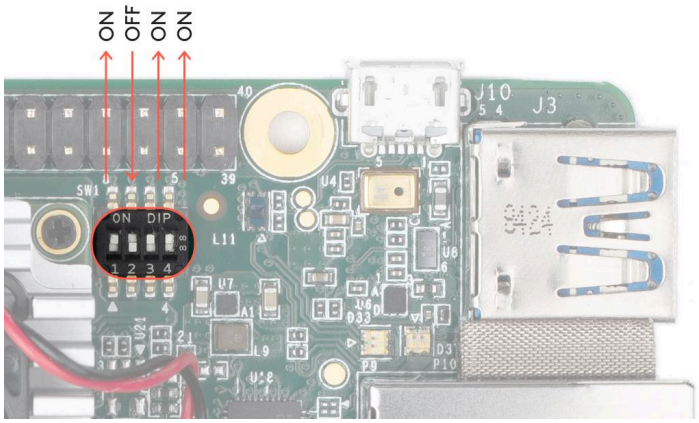
La tarjeta SD se utiliza para alojar el sistema operativo Mendel Linux que se ejecuta en la tarjeta, esto permite que la tarjeta funcione como una computadora embebida, lo que facilita la ejecución de aplicaciones y la interacción con la tarjeta, además del almacenamiento de los programas y proyectos que tengamos. Así que comencemos:

- Descargue el archivo de Imagen (https://dl.google.com/coral/mendel/enterprise/enterprise-eagle-flashcard-20211117215217.zip) y descomprimalo. El ZIP contiene un archivo nombrado `flashcard_arm64.img`.

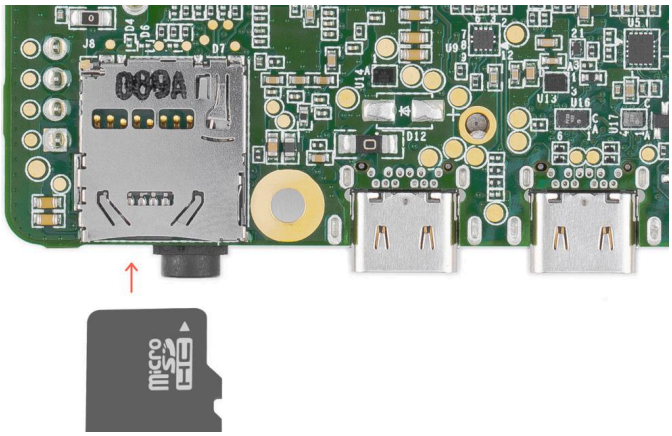
- Utilice un programa como balenaEtcher para flashear el archivo flashcard_arm64.img a su tarjeta microSD. Esto toma 5-10 minutos, dependiendo de la velocidad de su tarjeta microSD y adaptador.

- **Iniciar la Tarjeta**

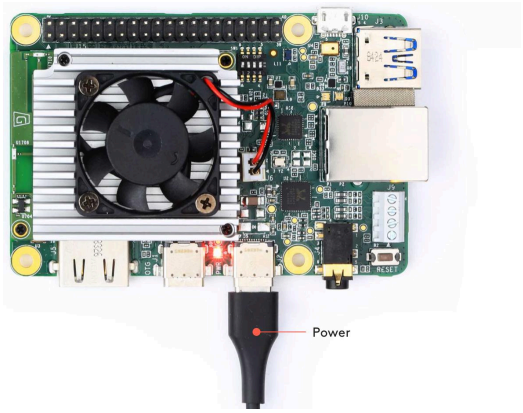
Con la tarjeta apagada y desconectada cambie los interruptores a modo boot, como se indica en la imagen:



Inserte la SD flasheada en la placa de desarrollo (las clavijas de la tarjeta miran hacia la placa). Aún **no** encienda la placa.



Encienda la placa conectando el cable de alimentación de 2-3 A al puerto USB-C etiquetado como 'PWR' (No alimente la placa con un computador). El LED rojo de la placa debería encenderse. Cuando la placa se inicia, carga la imagen de la tarjeta SD y comienza a actualizarse. Debería finalizar en 5 a 10 minutos, dependiendo de la velocidad de su tarjeta microSD.

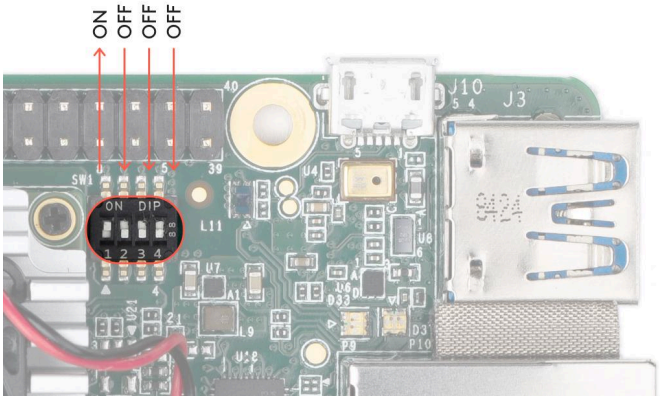


Ha terminado cuando el ventilador se apague y el LED rojo se apague, cuando esto suceda, desconecte la alimentación y retire la tarjeta microSD.

**Recuerde que esto solo se realiza la primera vez.

- **Trabajar con la Tarjeta**

Con la tarjeta apagada y desconectada cambie los interruptores del modo boot, como se indica en la imagen:



Conecte la placa a la alimentación y ahora debería iniciar Mendel Linux. El arranque por primera vez después del flasheo tarda unos 3 minutos (después será mucho más rápido).

Para conectarse a la Coral necesita usar la terminal Git Bash, esta se incluye en Git para Windows (https://git-scm.com/downloads) , también debe tener instalado Python en su ordenador, después de instalar estos dos, ábra Git Bash y ejecute los siguientes comandos para que Python sea accesible:

```
echo "alias python3='winpty python3.exe'" >> ~/.bash_profile
```

```
source ~/.bash_profile
```

Mientras la placa se inicia, puede instalar MDT en su computadora, MDT es una herramienta de línea de comandos que le ayuda a interactuar con la Dev Board. Por ejemplo, MDT puede enumerar los dispositivos conectados, instalar paquetes de Debian en la placa y abrir una terminal shell. Instale MDT de la siguiente manera, en un terminal de Git Bash:

```
python3 -m pip install --user mendel-development-tool
```

Es posible que vea una advertencia de que el mdt se instaló en algún lugar que no está en su variable de entorno PATH. Si es así, asegúrese de agregar la ubicación proporcionada a su RUTA, según corresponda para su sistema operativo. En Windows:

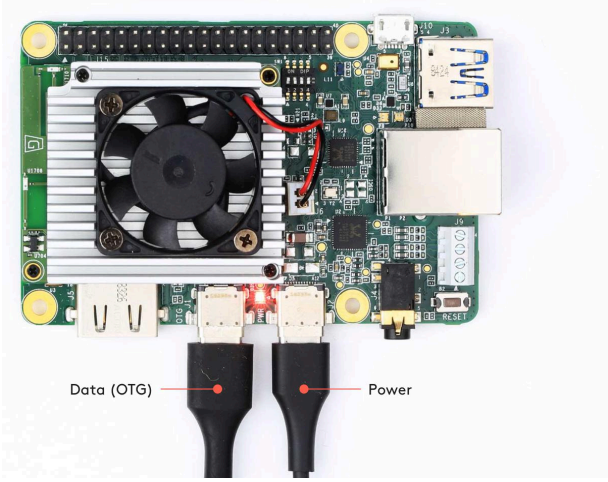
```
echo "alias mdt='winpty mdt'" >> ~/.bash_profile

source ~/.bash_profile
```

En Linux:

```
echo 'export PATH="$PATH:$HOME/.local/bin"' >> ~/.bash_profile

source ~/.bash_profile
```

En macOS es diferente así que por favor siga las instrucciones del siguiente link: https://coral.ai/docs/dev-board/mdt/#mdt-on-macos

Ya con el sistema Mendel corriendo en la placa y una computadora con MDT, conecte un cable USB-C desde su computadora al otro puerto USB de la placa (etiquetado como 'OTG')



En un terminal Git Bash compruebe de qué MDT pueda ver su dispositivo con el siguiente comando:

```
mdt devices
```

Deberías ver como resultado el nombre de host y la dirección IP de tu placa:

Ej: `orange-horse (192.168.100.2)`

Si no ve su dispositivo, probablemente se deba a que el sistema aún está configurando Mendel. En su lugar, ejecute `mdt wait-for-device`. Cuando su tablero esté listo, imprimirá "`Found 1 devices`".

El nombre de host de su placa se genera aleatoriamente la primera vez que arranca desde una nueva actualización. Puede cambiar este nombre utilizando las herramientas estándar de nombre de host de Linux (como `hostname`).

Ahora, para abrir el shell del dispositivo (El shell es el programa que provee una interfaz de usuario para acceder a los servicios del sistema operativo) , ejecute este comando:

```
mdt shell
```

Después de un momento, deberías ver el indicador de conexión con la placa. Asegúrese de que su placa solo esté conectada a través de USB antes de usar MDT por primera vez.

Puede que surja un problema y salga el siguiente mensaje de error:

```
It looks like you're trying to connect to a device that isn't connected to your workstation via USB and doesn't have the SSH key this MDT generated. To connect with `mdt shell` you will need to first connect to your device ONLY via USB.
```

Ante este proveemos dos soluciones posibles:

1.

Conéctese a su placa a través del puerto micro USB usando la pantalla:

```
screen /dev/ttyUSB0 115200
```

Log in:

```
Login: mendel

Password: mendel
```

Comprueba tus conexiones activas:

```
nmcli connection show
```

Deberías ver tus conexiones de red enumeradas en el resultado. Las conexiones activas son aquellas que tienen un dispositivo asociado. Por ejemplo:

```
NAME        UUID                                  TYPE              DEVICE
 usb0        cee59267-da42-443f-a8ca-cb94bccf1ad9  802-3-ethernet    usb0
 STRAVINSKY  71db4358-29d4-4da3-96d1-f76baf6a8f88  802-11-wireless   wlan0
```

Desactiva cualquier conexión que no sea USB. En este caso, `wlan0`. Para hacer esto, use `nmtui`, seleccione `Activate a connection`, busque la red que desea deshabilitar y marque y seleccione `<Deactivate>` en ella.

Ahora ya puede volver a intentar utilizar `mdt shell` en su máquina host, debería conectarse correctamente.

2.

También el problema puede ser causado por el paquete PyPI *mendel-development-tool* donde siempre que la IP del dispositivo no comienza con 192.168.100 se genera una excepción en sshclient.py, puede comprobar la IP con:

mdt devices

El problema se puede arreglar de la siguiente forma, abra el archivo del problema:

```
vim $HOME/.local/lib/python3.6/site-packages/mdt/sshclient.py +86
```

Sustituir la línea 86, que debe decir:

```
if not self.address.startswith('192.168.100'):
        raise NonLocalDeviceError()
```

Con:

```
if not self.address.startswith('192.168.10'):

    raise NonLocalDeviceError()
```

Una vez que el paquete ha sido actualizado sólo tienes que actualizarlo en tu host:

```
pip3 install --upgrade mendel-development-tool
```

Ahora ya puede volver a intentar utilizar `mdt shell` en su máquina host, debería conectarse correctamente.

Cuando haya terminado con la placa de desarrollo, no la desconecte simplemente de la alimentación, hacerlo podría dañar la imagen del sistema si hay alguna operación de escritura en curso. En su lugar, apague el sistema de forma segura con el siguiente comando:

```
sudo shutdown now
```

Cuando el LED rojo de la placa de desarrollo se apague, puedes desconectar la alimentación.

- **Ejemplos**

La página de Coral provee ejemplos para ver el funcionamiento de la tarjeta ante una aplicación de clasificación, detección y el uso de una cámara:

https://coral.ai/docs/dev-board/get-started/#run-demo

La Coral dev Board proporciona acceso a varias interfaces periféricas mediante 40 pines disponibles, donde se incluyen GPIO, I2C, UART y SPI; puede interactuar con los pines desde el espacio del usuario utilizando interfaces de Linux, como archivos de dispositivo (/dev) y archivos sysfs (/sys). También hay varias bibliotecas API que puedes usar para programar los periféricos conectados a estos pines, incluidas python-periphery, Adafruit Blinka y libgpiod.

Todos los pines de E/S en el conector están alimentados a 3,3 V, con una impedancia programable de 40-255 ohmios y una corriente máxima de 82 mA. Para conocer más información vaya al siguiente link:

https://coral.ai/docs/dev-board/gpio/#uart

Crear una red

Afortunadamente para simplificar el desarrollo con Coral, Edge TPU es compatible con la API estándar de TensorFlow Lite. Entonces, si ya tiene código que utiliza la API de TensorFlow Lite, puede ejecutar un modelo en Edge TPU cambiando solo un par de líneas de código.

La compatibilidad del Edge TPU se limita a ciertos modelos de TensorFlow Lite, los cuales deben ser compilados específicamente para su ejecución en Edge TPU. Entre los modelos disponibles se incluyen opciones como EfficientNet-EdgeTpu e Inception para la clasificación de imágenes, MobileNet y EfficientDet para la detección de objetos, ResNet para la segmentación semántica, PoseNet MobileNet para la estimación de pose, y YamNet para la clasificación de audio. Para conocer todos los modelos disponibles vaya al siguiente link:

https://coral.ai/models/all/

- **Model Maker**

En teoría existe una forma muy simple de crear un modelo tflite y es usando la biblioteca TensorFlow Lite Model Maker, que simplifica el proceso de entrenamiento de un modelo con un conjunto de datos personalizado en solo unas pocas líneas de código.

Es compatible con las tareas de clasificación de imágenes, detección de objetos, clasificación de texto, BERT Pregunta Respuesta y clasificación de audio.

Por ejemplo, estos son los pasos para entrenar un modelo de clasificación de imágenes:

```
from tflite_model_maker import image_classifier

from tflite_model_maker.image_classifier import DataLoader

# Load input data specific to an on-device ML app.

data = DataLoader.from_folder('flower_photos/')

train_data, test_data = data.split(0.9)

# Customize the TensorFlow model.

model = image_classifier.create(train_data)

# Evaluate the model.

loss, accuracy = model.evaluate(test_data)

# Export to Tensorflow Lite model and label file in `export_dir`.

model.export(export_dir='/tmp/')
```

Donde flower_photos es una carpeta que contiene otras subcarpetas que contienen imágenes con distintos tipos de flores, los nombres de las carpetas que contienen las imágenes de las flores serán los labels.

Lamentablemente a la fecha Diciembre del 2023 a causa de una actualización de python del 3.9 al 3.10 Model Maker dejó de funcionar, están buscando la solución por lo que esperamos que en el momento que estés leyendo esto, ya se haya arreglado.

- **TensorFlow**

Dado que no podemos generar directamente un archivo TensorFlow Lite, procedemos a entrenar un modelo utilizando un conjunto de datos en TensorFlow, el cual posteriormente convertiremos a TensorFlow Lite mediante cuantificación después del entrenamiento. Para concluir, compilamos el modelo para asegurar su compatibilidad con Edge TPU.

El modelo se basa en una versión previamente entrenada de MobileNet V2. Comenzaremos reentrenando sólo las capas de clasificación, reutilizando las capas de extracción de características previamente entrenadas de MobileNet. Luego si vemos necesario, ajustaremos el modelo actualizando los pesos en algunas de las capas. Este tipo de aprendizaje por transferencia es mucho más rápido que entrenar todo el modelo desde cero.

Una vez entrenado, usaremos la cuantificación posterior al entrenamiento para convertir todos los parámetros al formato int8, lo que reduce el tamaño del modelo y aumenta la velocidad de inferencia. Recomendamos usar Colab por su facilidad de uso.

```
#Importamos las librerías necesarias

import tensorflow as tf

import os

import numpy as np

import matplotlib.pyplot as plt
```

Sugerimos subir un archivo comprimido con la base de datos, seguiremos la misma lógica hablada anteriormente de tener una carpeta que contiene otras subcarpetas que contienen imágenes de los distintos tipos de objetos que quieras clasificar, los nombres de las carpetas que contienen las imágenes de las flores serán los labels. Recomendamos subir un archivo comprimido.

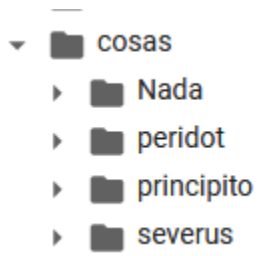
```
#Para subir los archivos al Colab

from google.colab import files

uploaded = files.upload()
```

Para descomprimir el archivo previamente subido, si es .zip:

```
!unzip nombre_del_archivo.zip
```

Si es .rar

```
!unrar nombre_del_archivo.rar
```



Le asignamos la dirección de nuestro archivo a una variable llamada directorio:

```
dir = '/content/nombre_del_archivo'
```

A continuación, usamos `ImageDataGenerator` para reescalar los datos de la imagen en valores flotantes (dividir por 255 para que los valores del tensor estén entre 0 y 1) y llamamos a `flow_from_directory()` para crear dos generadores: uno para el conjunto de datos de entrenamiento y otro para el conjunto de datos de validación.

```
IMAGE_SIZE = 224
BATCH_SIZE = 64
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2)

train_generator = datagen.flow_from_directory(
    dir,
    target_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE,
    subset='training')

val_generator = datagen.flow_from_directory(
    dir,
    target_size=(IMAGE_SIZE, IMAGE_SIZE),
```

```
    batch_size=BATCH_SIZE,
    subset='validation')
```

```
Found 461 images belonging to 4 classes.
Found 113 images belonging to 4 classes.
```

En cada iteración, estos generadores proporcionan un lote de imágenes leyendo imágenes del disco y procesándolas al tamaño de tensor adecuado (224 x 224). La salida es una tupla de (imágenes, etiquetas). Por ejemplo, puedes ver las formas resultantes aquí:

```
image_batch, label_batch = next(val_generator)

image_batch.shape, label_batch.shape
```

```
((64, 224, 224, 3), (64, 4))
```

Ahora guardamos las etiquetas de clase en un archivo de texto, crea un archivo .txt:

```
print (train_generator.class_indices)


labels = '\n'.join(sorted(train_generator.class_indices.keys()))


with open('labels.txt', 'w') as f:
  f.write(labels)
```

```
{'Nada': 0, 'peridot': 1, 'principito': 2, 'severus': 3}
```

Ahora crearemos un modelo, aquí crearemos un MobileNet V2 de Keras como modelo base, este nos proporciona un excelente extractor de funciones para la clasificación de imágenes.

Al crear una instancia del modelo MobileNetV2, especificamos el argumento **`include_top=False`** para cargar la red sin las capas de clasificación en la parte

superior. Luego configuramos trainable false para congelar todos los pesos en el modelo base. Esto convierte efectivamente el modelo en un extractor de características porque todos los pesos y sesgos previamente entrenados se conservan en las capas inferiores cuando comencemos a entrenar nuestro encabezado de clasificación. Recuerde que existen más modelos puede consultarlos aqui: https://www.tensorflow.org/api_docs/python/tf/keras/applications

```
IMG_SHAPE = (IMAGE_SIZE, IMAGE_SIZE, 3)

# Cree el modelo base a partir de MobileNet V2

base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,

                                               include_top=False,

                                               weights='imagenet')

base_model.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_t
9406464/9406464 [==============================] - 0s 0us/step
```

Ahora crearemos el encabezado de clasificación, creamos un modelo `Sequential`, le pasamos el modelo MobileNet como base y agregamos nuevas capas de clasificación para que podamos establecer la dimensión de salida final para que coincida con la cantidad de clases en nuestro conjunto de datos:

```
model = tf.keras.Sequential([

  base_model,

  tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'),

  tf.keras.layers.Dropout(0.2),

  tf.keras.layers.GlobalAveragePooling2D(),

  #units debe concordar con el número de clases que tengo

  tf.keras.layers.Dense(units=4, activation='softmax')

])
```

Aunque este método se llama `compile()`, es básicamente un paso de configuración necesario antes de que podamos comenzar a entrenar:

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Puedes ver un resumen de cadena de la red final con el método summary():

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Func  (None, 7, 7, 1280)        2257984
 tional)

 conv2d (Conv2D)             (None, 5, 5, 32)          368672

 dropout (Dropout)           (None, 5, 5, 32)          0

 global_average_pooling2d (  (None, 32)                0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 4)                 132

=================================================================
Total params: 2626788 (10.02 MB)
Trainable params: 368804 (1.41 MB)
Non-trainable params: 2257984 (8.61 MB)
_________________________________________________________________
```

Ahora podemos entrenar el modelo utilizando los datos proporcionados por `train_generator` y `val_generator` que creamos al principio:

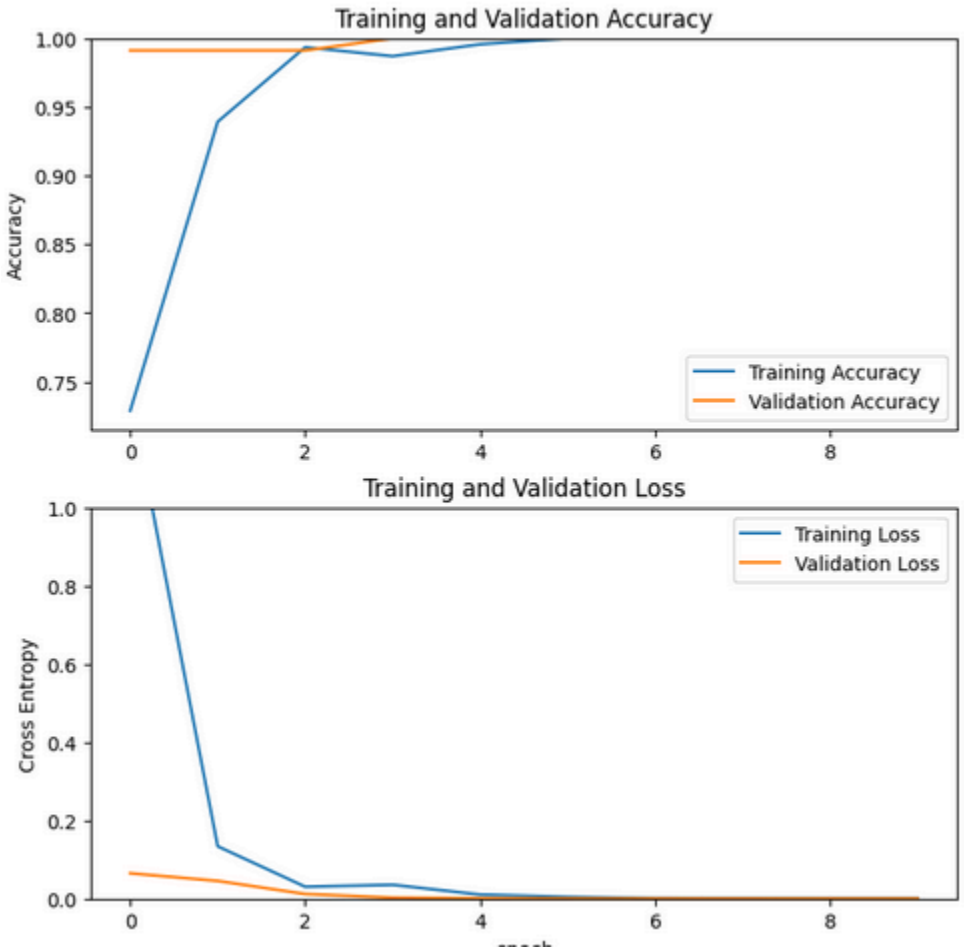
```
history = model.fit(train_generator,
                    steps_per_epoch=len(train_generator),
                    epochs=10,
                    validation_data=val_generator,
                    validation_steps=len(val_generator))
```

```
Epoch 1/10
8/8 [==============================] - 77s 10s/step - loss: 1.2899 - accuracy: 0.7289 - val_loss: 0.0648 - val_accuracy: 0.9912
Epoch 2/10
8/8 [==============================] - 69s 9s/step - loss: 0.1337 - accuracy: 0.9393 - val_loss: 0.0452 - val_accuracy: 0.9912
Epoch 3/10
8/8 [==============================] - 59s 7s/step - loss: 0.0303 - accuracy: 0.9935 - val_loss: 0.0117 - val_accuracy: 0.9912
Epoch 4/10
8/8 [==============================] - 69s 10s/step - loss: 0.0354 - accuracy: 0.9870 - val_loss: 0.0024 - val_accuracy: 1.0000
Epoch 5/10
8/8 [==============================] - 66s 8s/step - loss: 0.0102 - accuracy: 0.9957 - val_loss: 6.1119e-04 - val_accuracy: 1.0000
Epoch 6/10
8/8 [==============================] - 64s 8s/step - loss: 0.0045 - accuracy: 1.0000 - val_loss: 1.7151e-04 - val_accuracy: 1.0000
Epoch 7/10
8/8 [==============================] - 69s 9s/step - loss: 7.3650e-04 - accuracy: 1.0000 - val_loss: 1.0058e-04 - val_accuracy: 1.0000
Epoch 8/10
8/8 [==============================] - 60s 7s/step - loss: 5.9029e-04 - accuracy: 1.0000 - val_loss: 8.0250e-05 - val_accuracy: 1.0000
Epoch 9/10
8/8 [==============================] - 70s 9s/step - loss: 5.1365e-04 - accuracy: 1.0000 - val_loss: 6.9841e-05 - val_accuracy: 1.0000
Epoch 10/10
8/8 [==============================] - 68s 9s/step - loss: 3.7490e-04 - accuracy: 1.0000 - val_loss: 6.5550e-05 - val_accuracy: 1.0000
```

Podemos ver las curvas de aprendizaje:

```
acc = history.history['accuracy']

val_acc = history.history['val_accuracy']


loss = history.history['loss']

val_loss = history.history['val_loss']


plt.figure(figsize=(8, 8))

plt.subplot(2, 1, 1)

plt.plot(acc, label='Training Accuracy')

plt.plot(val_acc, label='Validation Accuracy')

plt.legend(loc='lower right')

plt.ylabel('Accuracy')

plt.ylim([min(plt.ylim()),1])

plt.title('Training and Validation Accuracy')


plt.subplot(2, 1, 2)

plt.plot(loss, label='Training Loss')

plt.plot(val_loss, label='Validation Loss')

plt.legend(loc='upper right')
```

```
plt.ylabel('Cross Entropy')
plt.ylim([0,1.0])
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```



Si queremos aumentar la precisión, podemos entrenar (o 'afinar') más capas del modelo previamente entrenado. Es decir, descongelaremos algunas capas del modelo base y ajustaremos esos pesos (que originalmente se entrenaron con 1000 clases de ImageNet) para que estén mejor adaptados a las características que se encuentran en nuestro conjunto de datos. Primero, veamos cuántas capas hay en el modelo base:

```
print("Number of layers in the base model: ", len(base_model.layers))
```

```
Number of layers in the base model:  154
```

Intentemos solo las 100 capas inferiores:

```
base_model.trainable = True

fine_tune_at = 100


# Freeze all the layers before the `fine_tune_at` layer

for layer in base_model.layers[:fine_tune_at]:

  layer.trainable =  False
```

Ahora configure el modelo nuevamente, pero esta vez con una tasa de aprendizaje más baja (el valor predeterminado es 0,001):

```
model.compile(optimizer=tf.keras.optimizers.Adam(1e-5),

              loss='categorical_crossentropy',

              metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Func  (None, 7, 7, 1280)        2257984
 tional)

 conv2d (Conv2D)             (None, 5, 5, 32)          368672

 dropout (Dropout)           (None, 5, 5, 32)          0

 global_average_pooling2d (  (None, 32)                0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 4)                 132

=================================================================
Total params: 2626788 (10.02 MB)
Trainable params: 2230244 (8.51 MB)
Non-trainable params: 396544 (1.51 MB)
_________________________________________________________________
```

Ahora ajustamos todas las capas entrenables. Esto comienza con los pesos que ya entrenamos en las capas de clasificación, por lo que no necesitamos tantas épocas:

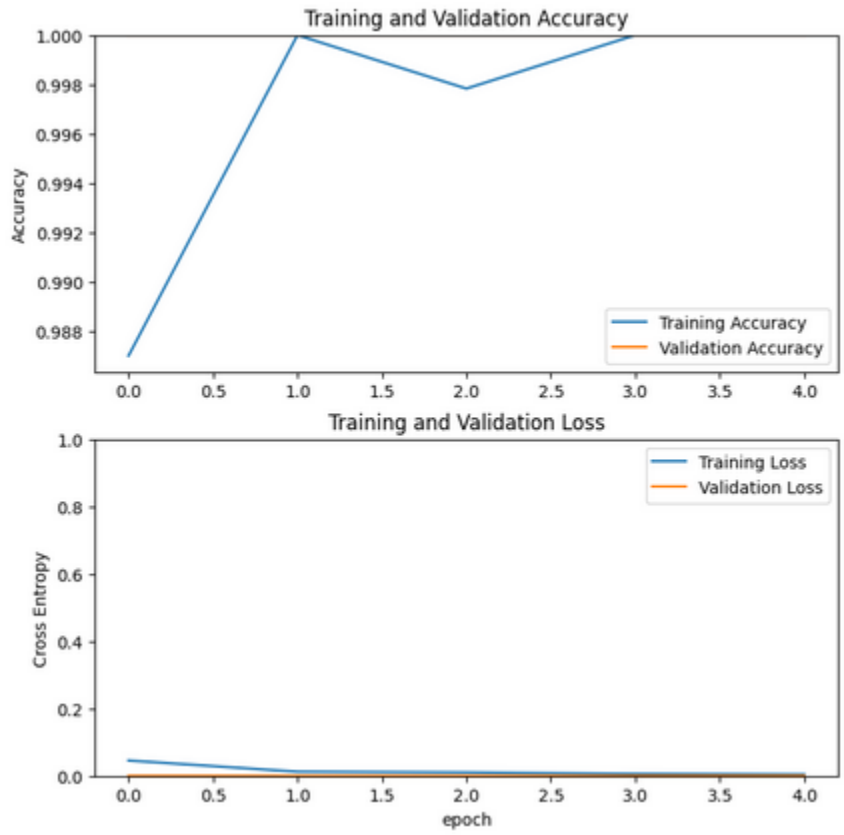
```
history_fine = model.fit(train_generator,
                         steps_per_epoch=len(train_generator),
                         epochs=5,
                         validation_data=val_generator,
                         validation_steps=len(val_generator))
```

```
Epoch 1/5
8/8 [==============================] - 98s 11s/step - loss: 0.0458 - accuracy: 0.9870 - val_loss: 6.8637e-05 - val_accuracy: 1.0000
Epoch 2/5
8/8 [==============================] - 75s 9s/step - loss: 0.0125 - accuracy: 1.0000 - val_loss: 8.5443e-05 - val_accuracy: 1.0000
Epoch 3/5
8/8 [==============================] - 81s 10s/step - loss: 0.0104 - accuracy: 0.9978 - val_loss: 1.0043e-04 - val_accuracy: 1.0000
Epoch 4/5
8/8 [==============================] - 71s 8s/step - loss: 0.0062 - accuracy: 1.0000 - val_loss: 1.1840e-04 - val_accuracy: 1.0000
Epoch 5/5
8/8 [==============================] - 73s 9s/step - loss: 0.0046 - accuracy: 1.0000 - val_loss: 1.3330e-04 - val_accuracy: 1.0000
```

Observamos las curvas de aprendizaje nuevamente:

```
acc = history_fine.history['accuracy']
val_acc = history_fine.history['val_accuracy']


loss = history_fine.history['loss']
val_loss = history_fine.history['val_loss']


plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.ylabel('Accuracy')
plt.ylim([min(plt.ylim()),1])
plt.title('Training and Validation Accuracy')
```

```
plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.ylabel('Cross Entropy')
plt.ylim([0,1.0])
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```



Este proceso de ir ajustando puede hacerlo la veces que sea necesario hasta que obtenga un resultado que le parezca aceptable.

Ahora convertiremos el modelo a un modelo de TensorFlow Lite:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

with open('mobilenet_v2_1.0_224.tflite', 'wb') as f:
  f.write(tflite_model)
```

Sin embargo, este archivo .tflite es un modelo básico y todavía usa valores de punto flotante para los datos de los parámetros y necesitamos cuantificar completamente el modelo al formato int8. Para cuantificar completamente el modelo, necesitamos realizar una cuantificación posterior al entrenamiento con un conjunto de datos representativo, lo que requiere algunos argumentos más para TFLiteConverter y una función que cree un conjunto de datos que sea representativo del conjunto de datos de entrenamiento. Entonces, convirtamos el modelo nuevamente con cuantificación posterior al entrenamiento:

```
# A generator that provides a representative dataset
def representative_data_gen():
  dataset_list = tf.data.Dataset.list_files(flowers_dir + '/*/*')
  for i in range(100):
    image = next(iter(dataset_list))
    image = tf.io.read_file(image)
    image = tf.io.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, [IMAGE_SIZE, IMAGE_SIZE])
    image = tf.cast(image / 255., tf.float32)
    image = tf.expand_dims(image, 0)
    yield [image]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
# This enables quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# This sets the representative dataset for quantization
```

```
converter.representative_dataset = representative_data_gen

# This ensures that if any ops can't be quantized, the converter throws an
error

converter.target_spec.supported_ops                                    =
[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

# For full integer quantization, though supported types defaults to int8
only, we explicitly declare it for clarity.

converter.target_spec.supported_types = [tf.int8]

# These set the input and output tensors to uint8 (added in r2.3)

converter.inference_input_type = tf.uint8

converter.inference_output_type = tf.uint8

tflite_model = converter.convert()


with open('mobilenet_v2_1.0_224_quant.tflite', 'wb') as f:

  f.write(tflite_model)
```

Ahora tenemos un modelo TensorFlow Lite completamente cuantificado. Para asegurarnos de que la conversión haya ido bien, evaluemos tanto el modelo sin formato como el modelo de TensorFlow Lite. Primero verifique la precisión del modelo sin procesar:

```
batch_images, batch_labels = next(val_generator)


logits = model(batch_images)

prediction = np.argmax(logits, axis=1)

truth = np.argmax(batch_labels, axis=1)


keras_accuracy = tf.keras.metrics.Accuracy()

keras_accuracy(prediction, truth)


print("Raw model accuracy: {:.3%}".format(keras_accuracy.result()))
```

```
Raw model accuracy: 100.000%
```

Ahora verifiquemos la precisión del archivo .tflite, usando el mismo conjunto de datos. Sin embargo, no existe una API conveniente para evaluar la precisión de un modelo de TensorFlow Lite, por lo que este código ejecuta varias inferencias y compara las predicciones con la verdad básica:

```
def set_input_tensor(interpreter, input):
  input_details = interpreter.get_input_details()[0]
  tensor_index = input_details['index']
  input_tensor = interpreter.tensor(tensor_index)()[0]
   # Inputs for the TFLite model must be uint8, so we quantize our input data.
   # NOTE: This step is necessary only because we're receiving input data from
  # ImageDataGenerator, which rescaled all image data to float [0,1]. When using
   # bitmap inputs, they're already uint8 [0,255] so this can be replaced with:
  #   input_tensor[:, :] = input
  scale, zero_point = input_details['quantization']
  input_tensor[:, :] = np.uint8(input / scale + zero_point)


def classify_image(interpreter, input):
  set_input_tensor(interpreter, input)
  interpreter.invoke()
  output_details = interpreter.get_output_details()[0]
  output = interpreter.get_tensor(output_details['index'])
  # Outputs from the TFLite model are uint8, so we dequantize the results:
  scale, zero_point = output_details['quantization']
```

```
  output = scale * (output - zero_point)
  top_1 = np.argmax(output)
  return top_1

interpreter = tf.lite.Interpreter('mobilenet_v2_1.0_224_quant.tflite')
interpreter.allocate_tensors()

# Collect all inference predictions in a list
batch_prediction = []
batch_truth = np.argmax(batch_labels, axis=1)

for i in range(len(batch_images)):
  prediction = classify_image(interpreter, batch_images[i])
  batch_prediction.append(prediction)

# Compare all predictions to the ground truth
tflite_accuracy = tf.keras.metrics.Accuracy()
tflite_accuracy(batch_prediction, batch_truth)
print("Quant TF Lite accuracy: {:.3%}".format(tflite_accuracy.result()))
```

```
Quant TF Lite accuracy: 100.000%
```

Es posible que vea algo, pero con suerte no mucha, caída de precisión entre el modelo sin formato y el modelo TensorFlow Lite. Pero nuevamente, estos resultados no son adecuados para la implementación en producción, por lo que debemos compilar el modelo para Edge TPU. Primero descargue el compilador Edge TPU:

```
! curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

```
! echo "deb https://packages.cloud.google.com/apt coral-edgetpu-stable main" | sudo tee /etc/apt/sources.list.d/coral-edgetpu.list


! sudo apt-get update


! sudo apt-get install edgetpu-compiler
```

Luego compile el modelo:

```
! edgetpu_compiler mobilenet_v2_1.0_224_quant.tflite
```

```
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 1891 ms.

Input model: mobilenet_v2_1.0_224_quant.tflite
Input size: 2.94MiB
Output model: mobilenet_v2_1.0_224_quant_edgetpu.tflite
Output size: 3.12MiB
On-chip memory used for caching model parameters: 3.33MiB
On-chip memory remaining for caching model parameters: 4.36MiB
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 1
Total number of operations: 72
Operation log: mobilenet_v2_1.0_224_quant_edgetpu.log
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!
```

Eso sería todo, el modelo compilado utiliza el mismo nombre de archivo pero con '_edgetpu' añadido al final. Puedes descargar el modelo y el archivo de etiquetas desde Colab de esta manera:

```
from google.colab import files

files.download('mobilenet_v2_1.0_224_quant_edgetpu.tflite')

files.download('labels.txt')
```

Puede llegar a salir el error '`Failed to fetch`', probablemente se deba a que los archivos no se han terminado de guardar, así que espere un momento y vuelve a intentarlo. También esté atento a una ventana emergente del navegador que podría necesitar aprobación para descargar los archivos.

Ahora puede ejecutar el modelo en su dispositivo Coral, para comenzar, intente usar su modelo .tflite con este código para clasificar imágenes con la API de TensorFlow Lite (https://github.com/google-coral/tflite/tree/master/python/examples/classification). Simplemente siga las instrucciones en esa página para configurar su dispositivo, copie los archivos `mobilenet_v2_1.0_224_quant_edgetpu.tflite` y `labels.txt` a su Coral Dev Board o dispositivo con Coral Accelerator y pásele una foto que sea parte de su base de datos como esta:

```
python3 classify_image.py \
  --model mobilenet_v2_1.0_224_quant_edgetpu.tflite \
  --labels labels.txt \
  --input A.jpg
```

Nota: Los resultados que mostramos son acorde a nuestra base de datos, estos cambian según los datos de entrada que utilicen y a los cambios que realicen para que el código se adapte a ellos.

Nota: Recomendamos que para subir los archivos a la coral se realice desde un repositorio en github y así solo tienes que usar el comando `git clone`.