Google Coral Edge

Table of Contents

Part 1- Readings

Google Coral Edge TPU - Architecture

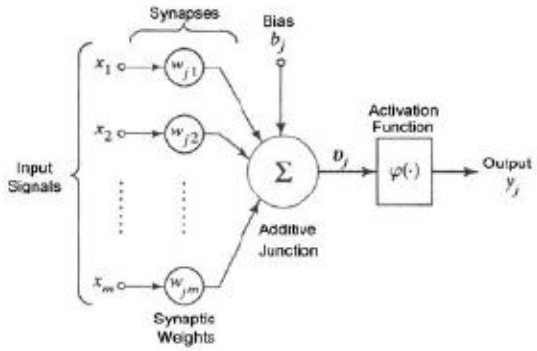
0.1 Introduction

The Google Coral has a TPU on board which speeds up the tensor calculations enormously. These tensor calculations are used in deep learning and neural networks. How does such a Tensor Processing Unit work?

Both the **Google Coral Dev** board and the Coral USB Accelerator **use** an **ASIC** made by the Google team called the **Edge TPU**. It is a much lighter version of the well-known TPUs used in Google's datacenter. It also consumes very little power, so it is ideal for small embedded systems. Nevertheless, the similarities in applied technology are significant. They both use a systolic array to do the tensor operations. First, let's start with neural nodes.

Neural networks used in deep learning consists of many neural nodes. They are all connected together in a defined way. The way these nodes are wired is called the topology of the network. This topology determines the function the network performs.

Each node has always three basic components. A multiplier multiplies all the inputs with their respective so-called weight, the synapses. An adder who accumulates all the individual multiplications. And an activation function that shapes the output given the addition. A schematic view below.

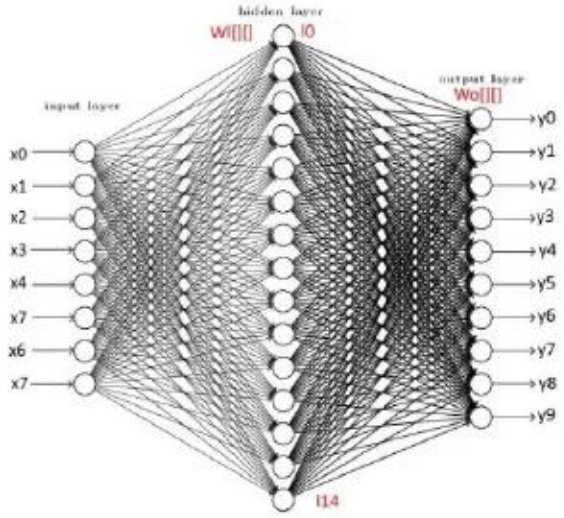


The corresponding formula can be written as follows.

$$y_i = \phi\left(\sum_{i=1}^{n} w_{ij} \cdot x_i\right)$$

The bias is a constant that gives the network much greater flexibility. Mathematically it can be included in the mul-add summation by giving an extra input the value 1.0 and the associated weight the value of the bias. Below a three-layer neural network is shown.

The corresponding algorithm of this network is easy to program; two nested for loops and you have your outputs. See the C code. It is all quite simple, no rocket science at all.




```
int j,i;
double sum;
 //x[] inputs
 //l[] hidden layer outputs
 //y[] outputs
 //Wl[][] weights hidden layer
 //Wo[][] weights output layer
 for(j=0;j<NoHiddenLayer;j++) {
   sum=0;
   for(i=0;i<NoInput;i++) sum+=x[i]*Wl[j][i];
   sum+=1.0*Wh[j][i]; //the bias
   l[j]=tanh(sum);
   }
 for(j=0;j<NoOutput;j++){
   sum=0;
   for(i=0;i<NoHiddenLayer;i++)sum+=l[i]*Wo[j][i];
   sum+=1.0*Wo[j][i]; //the bias
   y[j]=tanh(sum);
                                                    }
```

There is one problem, the computing time. A deep learning network consists of millions of neural nodes distributed over many layers. Despite the simplicity of the code, only one multiplication and one addition, it still takes a lot of time to calculate all the intermediate layers and outputs. On a fast PC, it can take a few seconds. Keep also in mind that training a network requires thousands of epochs and it becomes obvious that it all takes far too much time to be practical.

Fortunately, the algorithm is **well suited for parallel execution**. If you want to calculate an intermediate value, say `l0`, you do not need to know the other values on the same layer (`l1` to `l14`). All **values are independent** of each other.

A **first step to speed up** the algorithm could be to distribute the calculations over different cores if you have a multi-core CPU. In case of four cores, the first calculates `W0` to `W3`, while **core 2** simultaneously performs `W4` to `W7`.

At the same time generates **core 3** `W8` to `W11` and **core 4** calculates `W12` to `W14`, given the example above. In theory, a reduction of the execution time of 75% is possible. However, programming is very difficult. It makes the network topology rigid and the transfer of values to and from the cores usually ends up in a bottleneck. All this leads to disappointing results.

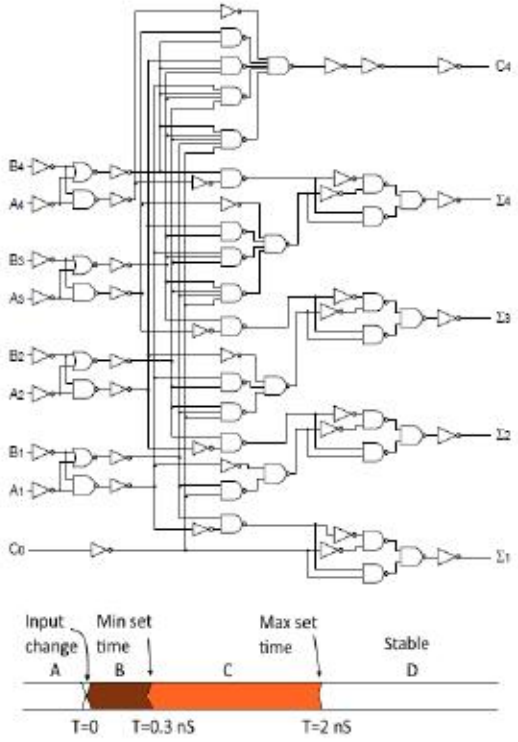
By the way, distributing the algorithm over the processes and threads won't take you any further, since they are all time-sliced by the operating system. In other words, every process and their threads are placed after one and each other in time, each getting a little time frame (slice) to do their calculations before moving to the other thread or process.

A second option is the use of the Graphical Processing Unit, the GPU on a video card. These are made up with a lot of little parallel cores designed for very fast matrix calculations. You can read more about this topic here. The best choice is the use of the **Tensor Processing Unit**, the **TPU**. This device has been specially designed for the above neural node algorithm.

0.1.1 The adder

When software is too slow, hardware is the answer. Let see how the Google's Edge TPU hardware is structured. The three main components of the neural node, the **multiplier**, the **adder** and the **activation function** must be included in the **hardware**. Let start with the adder.

Below a schematic diagram of the hardware of a **4-bit adder**. **A** and **B** are the inputs. If the output overflows **C4** the carry out is set. **C0** is the carry-in of a previous phase.



Every **basic digital gate** (AND, OR, NOT) has its own symbol. They usually consist of two or three transistors. Signals A and B propagate through the circuit and generate the result A + B. Changing one of them alters almost immediately the output. This happens extremely very fast, within a few nanoseconds.

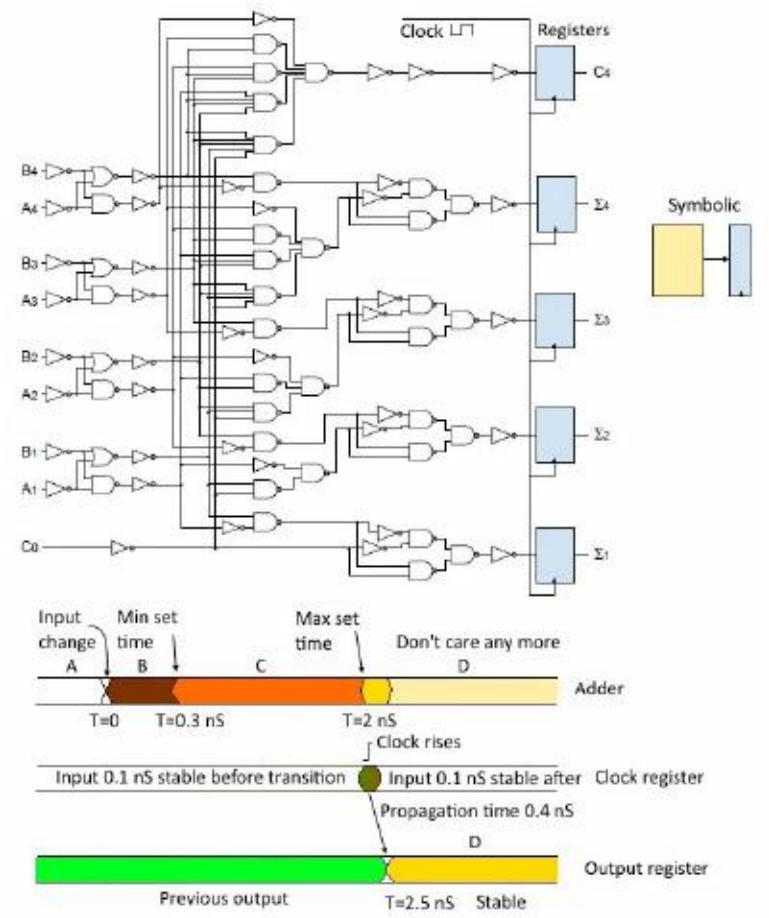
This so-called propagation time depends on the number of digital ports whose output changes. Sometimes two gates only change their state. Sometimes more the six gates in a chain must alter their output. The propagation time is therefore not fixed, but lies between two limits, a minimum and a maximum time, see diagram at the bottom of the drawing. By the way, al mentioned times are illustrative and have no relationship to any device.

0.1.2 Pipeline

If the propagation time of one adder lies at 2 nSec, the maximum clock rate can be 500 MHz. As mentioned earlier, a neural node can have hundreds of inputs, all of which must be accumulated. Designing a chain of adders is not a technical problem. However, the propagation time increases dramatically.

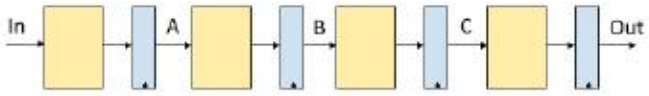
The last adder in the chain must wait for all intermediate results before its output becomes stable. With a chain of 250 inputs, each with a 2 nSec delay, the total time is now 500 nSec. Which gives a very slow clock of 2 MHz.

The solution here is a pipeline structure. A memory element is placed after each adder that keeps the result stable for the next adder.



The output of the registers is updated at the rising edge of the clock signal. That is the only time the output can change. When the clock signal is high or low or falls, the inputs cannot manipulate the output, it remains stable. Just before the clock rises, the input must be stable for a minimum time, also just after the rise. Here an illustrative 0.1 nS.

The register has its own propagation time (0.4 nS). The total propagation for a new stable signal is now 2.5 nS, which results in a maximum clock speed of 400 MHz. When placing the adders after each, you get the following situation.



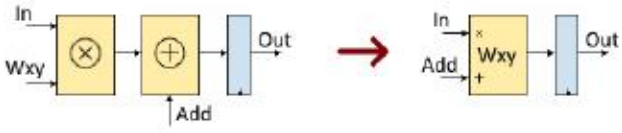
Clk	In	A	B	C	Out
⎍					
0		?	?	?	?
1			?	?	?
2				?	?
3					?
4					
5					
6					

Each color represents a value. After four clock cycles, the value at the input has propagated through the network and appears at the output. Because the registers are updated simultaneously a new input is accepted every 2.5 nS. The propagation time has been restored. The time it takes to travel through the whole pipeline is called latency time and is 10 nS in the above diagram (4 x 2.5nS). This pipeline technology is the backbone of modern computer technology and can be found everywhere. Every digital component is built on large pipelines which guarantee the required speed.

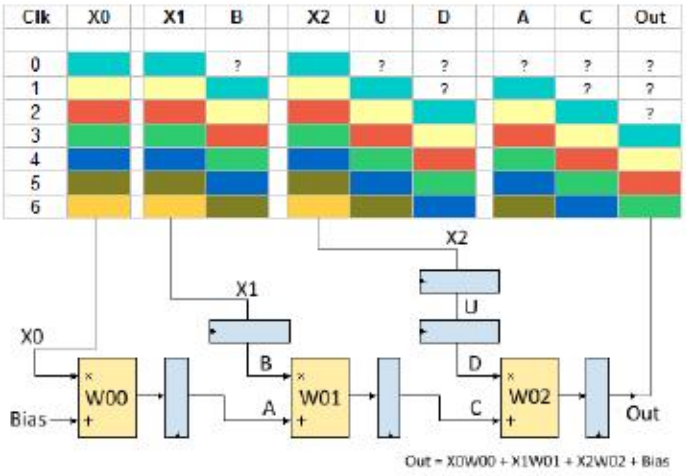
0.1.3 The mul-add cell

Every input signal of the neural node has its own effect on the final result. This gives one simple multiplication per input with a weight value. Just like an adder, a multiplication can build with the same basic digital gates. As you can imagine, more gates are now needed to perform a multiplication than a summation.

In the formula of the neural node, each input is multiplied by its own weight value.
The symbol can therefore even more straightforward.



For the sake of simplicity, no additional propagation time fixing registers have been added.
With the formula of the neural node in mind, it is now relatively easy to design such a neuron with a few mul-add cells. Below the schematic for a three input neuron.



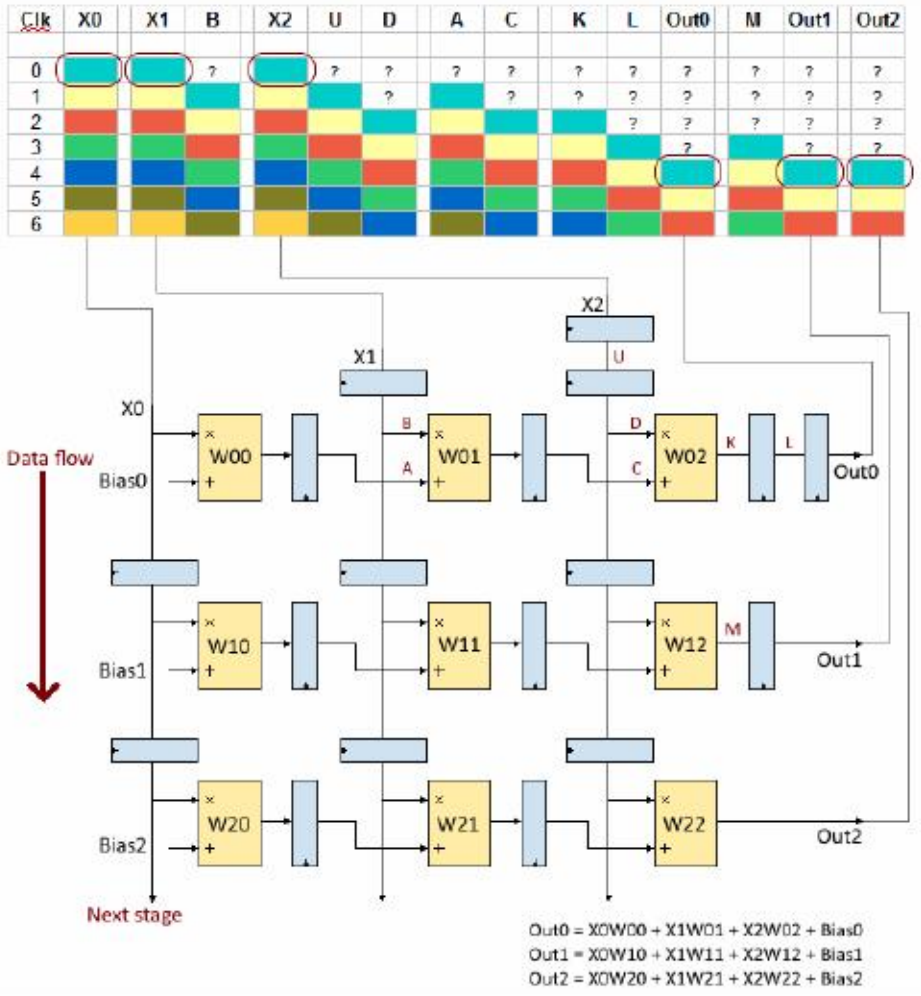
One important point to mention, are the registers at input **`X1`** and **`X2`**. They create a delay line and are placed here to synchronize the necessary latency in the mul-add chain. When the second mul-add cell accumulates the output from the first cell (signal **`A`**) it is delayed one clock cycle.

Therefore the multiplication of **`X1`** with its weight need also be delayed one cycle in order to have the results at the same time.

The color scheme above the design makes it all clear. Every color is now an element of the input vector [**`X0, X1, X2`**]. Looking at the horizontal lines, inputs **`A`** and **`B`** have always identical colors. Meaning they appear simultaneous, they are synchronized. The same applies to inputs **`C`** and **`D`**.

0.1.4 Systolic array

Once a neural node has been created, it is easy to extend this scheme for the other neurons on the same layer. Given that each individual input is connected to all neurons in the next layer, the diagram will look like this.



This type of design is called a systolic array. All values are pumped through the stages from top to bottom, hence the name. Looking at the timing, the array can accept a complete input vector every clock cycle. So, the propagation is still intact with 2.5 nS in the above-mentioned examples. At the same speed, the **array generates complete output vectors**.

If the systolic array expands in depth or width, the **propagation time still remains the same**.

Only the latency time increases. This enormous parallel calculation capacity is the reason why systolic arrays are widely used in neural network hardware.

The sizes of the systolic array in the Edge TPU are not known yet. The first TPU Google uses in its datacenter contains 256 x 256 mul-add cells. Running at 700 MHz it can theoretical performs 256 x 256 x 700.000.000 = 46 trillion mul-adds per second. Or if you look to individual operations 92 trillion (92 TOPS). However, this impressive figure is purely theoretical. In practice, there are some factors that slow performance.

The systolic array is made up of hardware. This means that it has fixed dimensions. Therefore, the input and output vectors also have fixed dimensions. However, the numbers of neurons per layer are determined by the design and are certainly not constant. If the number is smaller than the size of the array, it is, of course, possible to expand an

input vector with leading zeros, so that the dimension equals the array size. The same technique can be applied to an oversized output vector.

If the input vector contains more elements than the width of the systolic array, the entire arithmetic calculation must be performed in more than one call. To this end, the input vector is cut into a series of smaller vectors that match the width of the series. They are then processed one after the other. The intermediate results must be stored in a buffer.

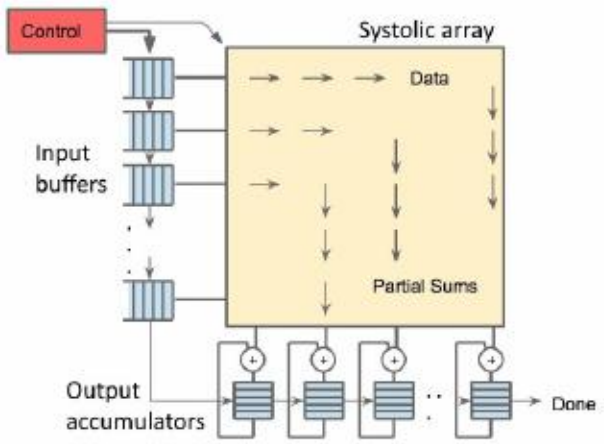
These values must be added after completion of all the systolic calls.

Note that not only the input vector is split into smaller parts, but all weights used in the multiplication must be updated according to the formula's processing section. This may require a massive memory transfer.

Below an example of a systolic array with only four inputs processing a vector of size 12.

$$
\begin{aligned}
Out_A &= X_0 \cdot W_0^0 + X_1 \cdot W_0^1 + X_2 \cdot W_0^2 + X_3 \cdot W_0^3 \quad \rightarrow \quad Out = Out_A + Out \\
Out_B &= X_4 \cdot W_0^4 + X_5 \cdot W_0^5 + X_6 \cdot W_0^6 + X_7 \cdot W_0^7 \quad \rightarrow \quad Out = Out_B + Out \\
Out_C &= X_8 \cdot W_0^8 + X_9 \cdot W_0^9 + X_{10} \cdot W_0^{10} + X_{11} \cdot W_0^{11} \quad \rightarrow \quad Out = Out_C + Out
\end{aligned}
$$

Looking at Google's schematic overview of the TPU, these output buffers and accumulators are drawn at the bottom. (In their diagram, the output is at the bottom instead of the right as on our drawing).



Buffers have also been placed on the input vector side in the diagram above. They act as a first-in-first-out buffer, a FIFO. This guarantees continuous input of the systolic array with input values. It may also have some rotating operations, which increases the performance of CNN networks. It is unclear whether Google uses this method here.

0.1.5 Activation unit

Once the output is available, it is sent to the activation unit. This module within the Edge TPU applies the activation function to the output. It is **hardwired**.
In other words, **you cannot alter the function**, it works like a ROM. Probably it is a ReLU function as it is nowadays the most used activation function and it is very easy to implement in hardware.

0.2 TPU versus Edge TPU

The Google's **Edge TPU** is a much smaller device than the **TPUs** that Google uses in its **data centers**.

It is obvious that such a small device cannot have the same functions as its much larger ancestor. The floor plan on the die does not allow this to happen. Nor can the systolic array have the **256 x 256 dimension** as in the **original TPU**.

Google has so far not revealed the **measures in the Edge**, but a well-founded **estimate** is **64 x 64** with a clock of **480 MHz**, resulting in **4 TOPS**.

Memory is also an issue. In the original chip, it takes around 29% of the total floor plan. It is impossible that the same amount can be found on the Edge TPU die. Moreover, the chip is very energy efficient. It requires no cooling.

This means that almost certainly all **the buffering is outside the chip**. It leaves only a simple transfer of input vectors, output vectors and weights to the chip.

0.2.1 Practice

In fact, that is the **whole concept of the Google Coral Dev board**. **Transfer all tensor calculations to the TPU** as quickly as possible, fetch the results, fill them in the next layer and start the cycle again with this layer. Process all layers one by one until ready.

To speed up the process, TensorFlow uses a special back end compiler, the Edge TPU compiler. The main task of this Edge TPU compiler is to partition the TensorFlow Lite file into more suitable TPU transfer packages.

If for any reason the TPU cannot process the TensorFlow Lite file or part of it, the CPU will take care of it. However, if this happens, your application will, of course, work extremely slowly.

0.2.2 8 bits integers

Another way of speeding up the calculations is the use of 8 bit signed integers instead of floats. A floating number occupies 32 bits in memory whereas the integer only needs 8.

Because a neural network is fairly insensitive to number accuracy, it will also work well with 8-bit integers. **It will still remain accurate**.

Not only does this technology save approximately 75% memory, but it also significantly reduces the number of transistors on the chip. If you look at the adder at the top of the page, you can imagine how many transistors a floating-point adder needs.

Without this compression, the Edge TPU would not be as powerful, small and energy efficient.

The converting algorithm from floating point to 8 bit signed integer is quite simple. First, it determines which maximum and minimum a variable will take in the model. The larger of the two is then taken and scaled to 127.

Suppose minimum is -1205.8 and maximum is 646.8. The larger is the min value of 1205.8. So that number becomes -127. Zero remains zero. That gives 646.8 a integer value of 127 x (646.8 / 1205.8) = 68. Every number in the TensorFlow model is converted in this way. Not only for the Edge TPU, by the way.

All TensorFlow Lite models for embedded deep learning are processed in this way.

A rather remarkable detail is the output of the Edge TPU, which does consist of a floating point.
Because the **activation function is embedded in a ROM**, it is just as easy to store floating numbers as integers.

One last tip in this regard. Apply quantization-aware training in TensorFlow. This simulates the effect of the 8 bit signed integers. Therefore generating a more precise model. It makes the model also more tolerant for low precision variables.

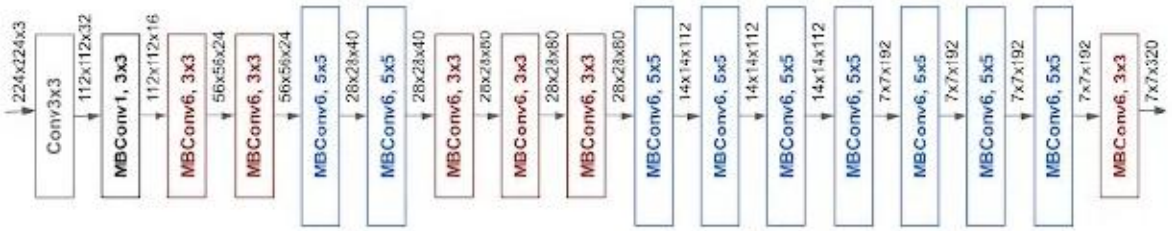
0.2.3 EfficientNet

EfficientNets are a **family of network topologies exclusive tailored for the Coral Edge TPU**. As explained above, the Edge TPU hardware is specially designed to accelerate MAC (multiply-accumulate) operations.

Only this specific operation can be done amazingly fast on a TPU. All other operations like **loading weights**, subtractions, additions or **dimension reduction are all time-consuming**. To get the maximum out of the TPU hardware, these operations must be reduced to the bare minimum.

And that is exactly the strategy behind **EfficientNets**. Rather a **large 3x3 convolution**, then a small 1x1 and 3x3 convolution sequence, which need double loading times. Don't re-use outputs of previous layers like **ResNet**, which use single additions. Use only simple activation functions like **ReLU** which are **hardcoded** implemented in the TPU architecture.

All resulting in a **fast deep learning network**.



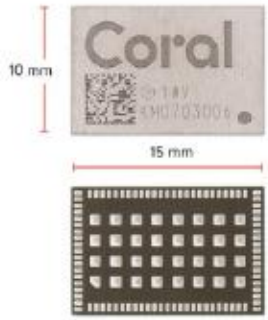
Part 1 - Lab 1

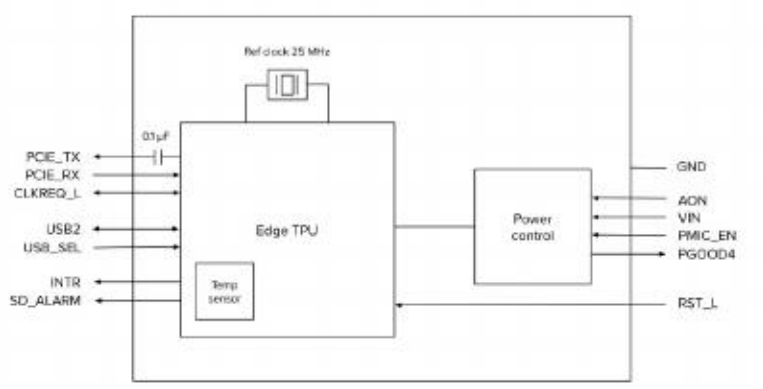
Get started with Coral Dev Board Mini for high speed ML inferencing

1.1 Introduction

The **Coral Dev Board Mini** is a single-board computer that provides fast machine learning (ML) inferencing in a small form factor. It's primarily designed as an evaluation device for the Accelerator Module (a surface-mounted module that provides the Edge TPU), but it's also a fully-functional embedded system you can use for various on-device ML projects.

Performs high-speed ML inferencing
The on-board **Edge TPU coprocessor** is capable of performing 4 trillion operations (tera-operations) per second (TOPS), using 0.5 watts for each TOPS (2 TOPS per watt). For example, it can execute state-of-the-art mobile vision models such as MobileNet v2 at almost 400 FPS, in a power efficient manner. See more performance benchmarks.





Provides a complete system
A single-board computer with SoC + ML + wireless connectivity, all on the board running a derivative of Debian Linux we call Mendel, so you can run your favorite Linux tools with this board.

Supports TensorFlow Lite
No need to build models from the ground up. TensorFlow Lite models can be compiled to run on the Edge TPU.

Tech specs

CPU	MediaTek 8167s SoC (Quad-core Arm Cortex-A35)
GPU	IMG PowerVR GE8300 (integrated in SoC)
ML accelerator	Google Edge TPU coprocessor: 4 TOPS (int8); 2 TOPS per watt
RAM	2 GB LPDDR3
Flash memory	8 GB eMMC
Wireless	Wi-Fi 5 (802.11a/b/g/n/ac); Bluetooth 5.0
Audio/video	3.5mm audio jack; digital PDM microphone; 2.54mm 2-pin speaker terminal; micro HDMI (1.4); 24-pin FFC connector for MIPI-CSI2 camera (4-lane); 24-pin FFC connector for MIPI-DSI display (4-lane)
Input/output	40-pin GPIO header; 2x USB Type-C (USB 2.0)

1.2 Get started with the Dev Board Mini

This section shows you how to set up the Coral Dev Board Mini, a single-board computer that accelerates machine learning models with the on-board Edge TPU.
The board in your box is already flashed with Mendel Linux, so the setup simply requires connecting to the board's shell console and updating some software. Then we'll show you how to run a TensorFlow Lite model on the board. From there, you can try any of our other examples—perhaps connect a camera and run an object detection model, pose detection model, keyphrase detector, and more.

On YouTube

https://www.youtube.com/watch?v=uF5FEzwOgzM

1.2.1 Gather requirements

To get started, be sure you have the following:

- A host computer running Linux (recommended) or Mac
    - Python 3 installed
- One USB-C power supply (2 A / 5 V), such as a phone charger
- One USB-C to USB-A cable (to connect to your computer)
- An available Wi-Fi internet connection

Notice that this board is intended as a **headless device**. That is, you typically do not connect a keyboard and monitor, and use it as a desktop computer. So, this guide is focused on connecting to the board remotely with a secure shell terminal.

However, the **Mendel OS** does include a simple desktop with a terminal application. So if you prefer, you can connect a monitor to the **micro-HDMI** port, and connect a **keyboard** and **mouse** to the USB OTG port.

1.2.2 Install MDT

MDT is a **command line tool** for your host computer that helps you interact with the **Dev Board Mini**.
For example, **MDT** can list connected devices, install Debian packages on the board, and open a shell terminal on the board.
You can install **MDT** on **your host** computer follows:

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

You might see a warning that **mdt** was installed somewhere that's not in your PATH environment variable. If so, be sure you add the given location to your **PATH**, as appropriate for your operating system. If you're on Linux, you can add it like this:

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

1.1.3 Plug in the board

1. Connect your power supply to the board's USB power plug (the left plug, as shown in figure 1) and connect it to an outlet.
2. Connect your USB data cable to the other USB plug and to your host computer.

When you connect the USB cable to your computer, the board automatically boots up, and the board's LED turns green. It then takes 20-30 seconds for the system to boot up.

Note: We've released a system update that instead turns the LED orange during boot-up, and then green when the system is booted and ready for login. (Further below, you'll find instruction to install this update.)

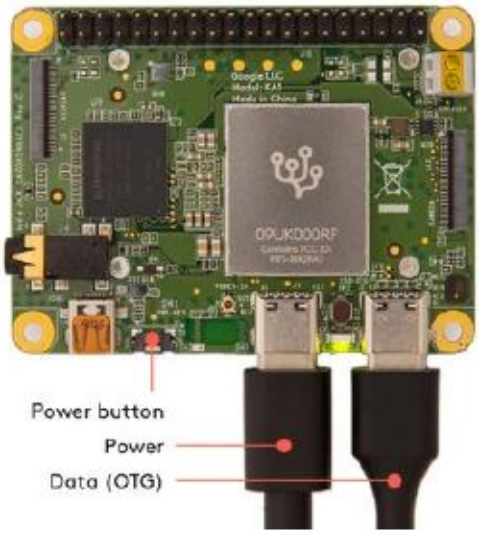


Fig 1. The USB power and data cables connected

About the USB ports

Both USB ports can power the board, but they behave differently:

- **USB power port (left):** This port supports power input only (no data). If you connect only this port, you must firmly press the power button to boot the board. Once the board is booted, you can press the power button to safely shut it down.
- **USB OTG port (right):** This port supports power input and USB data (as host or device). If you connect power here, the board boots up automatically. If you press the power button, the board reboots—you cannot fully shut down the board when the OTG port is connected to power (even if you shut down the operating system, it will reboot).

So to safely power-off the board, you must deliver power with the power port only, and then you can toggle power with the power button. Whereas, if you want the board always on, deliver power with the OTG port, but beware that you then cannot safely power off the board unless you transition power to the power port.

1.1.4 Connect to the board's shell via MDT

MDT provides an easy way to establish an **SSH** connection with the board over USB, as follows.
Mac users: Beginning with macOS Catalina (10.15), you cannot create an SSH connection over USB, so the following MDT commands won't work until you get the board online and install your SSH key. See the instructions to use MDT on macOS, and then return here to complete the setup.

1. Make sure MDT can see your device by running this command from your host computer:
   ```
   mdt devices
   ```
   You should see output showing your board hostname and IP address (your name will be different):
   ```
   fun-zebra          (192.168.100.2)
   ```
   If you don't see your device, it might be because the system is still booting up. Wait a moment and try again.
2. Now to open the device shell, run this command:
   ```
   mdt shell
   ```
   After a moment, you should see the board's shell prompt, which looks like this:
   ```
   mendel@fun-zebra:~$
   ```

Now you're in the board!

More on MDT

MDT allows us to execute a numbe of additional commands:

```
ubuntu@bako:~$ mdt

Usage: mdt <subcommand> [<options>]

Where <subcommand> may be one of the following:
    help              - this command, gets help on another command.
    devices           - lists all detected devices.
    wait-for-device   - waits for a device to be discovered on the network
    get               - gets an MDT variable value
    set               - sets an MDT variable value
    clear             - clears an MDT variable
    genkey            - generates an SSH key for connecting to a device
    pushkey           - pushes an SSH public key to a device
    setkey            - imports a PEM-format SSH private key into the MDT
                        keystore
    resetkeys         - removes all keys from the given board and resets key
                        authentication to factory defaults
    shell             - opens an interactive shell to a device
    exec              - runs a shell command and returns the output and the
                        exit code
    install           - installs a Debian package using mdt-install-package on
                        the device
    push              - pushes a file (or files) to the device
    pull              - pulls a file (or files) from the device
    reboot            - reboots a device
    reboot-bootloader - reboots a device into the bootloader
    version           - prints which version of MDT this is

Use "mdt help <subcommand>" for more details.
```

For example you can send or receive (push-pull) the files betweent the host computer and the device.

```
ubuntu@bako:~$ ls *.txt
test.txt
ubuntu@bako:~$ mdt push test.txt
Waiting for a device...
Connecting to coy-pig at 192.168.100.2
100% |>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| test.txt
ubuntu@bako:~$
ubuntu@bako:~$ rm test.txt
ubuntu@bako:~$ mdt pull test.txt .
Waiting for a device...
Connecting to coy-pig at 192.168.100.2
100% |<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<| test.txt
ubuntu@bako:~$
```

Execution example:

```
ubuntu@bako:~$ python3 -m pip install --user mendel-development-tool
Collecting mendel-development-tool
  Downloading mendel_development_tool-1.5.2-py3-none-any.whl (26 kB)
Collecting zeroconf>=0.27.0
  Downloading zeroconf-0.38.7-py3-none-any.whl (106 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 106.6/106.6 KB 827.2 kB/s eta 0:00:00
Requirement already satisfied: paramiko>=2.0.0 in /usr/lib/python3/dist-packages (from mendel-
development-tool) (2.6.0)
Collecting ifaddr>=0.1.7
  Downloading ifaddr-0.2.0-py3-none-any.whl (12 kB)
Installing collected packages: ifaddr, zeroconf, mendel-development-tool
Successfully installed ifaddr-0.2.0 mendel-development-tool-1.5.2 zeroconf-0.38.7
WARNING: You are using pip version 22.0.4; however, version 22.1.2 is available.
You should consider upgrading via the '/usr/bin/python3 -m pip install --upgrade pip' command.
ubuntu@bako:~$ echo 'export PATH="$PATH:$HOME/.local/bin"' >> ~/.bash_profile
ubuntu@bako:~$ source ~/.bash_profile
```

```
ubuntu@bako:~$ mdt devices
coy-pig         (192.168.100.2)
ubuntu@bako:~$ mdt shell
Waiting for a device...
Connecting to coy-pig at 192.168.100.2
Looks like you don't have a private key yet. Generating one.
Key not present on coy-pig -- pushing
Linux coy-pig 4.19.125-mtk #1 SMP PREEMPT Thu Nov 12 23:33:16 UTC 2020 aarch64

The programs included with the Mendel GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Mendel GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Feb 14 10:12:05 2019
mendel@coy-pig:~$ ls
```

1.1.5 Connect to the `internet`link

You'll need the board online to download system updates, TensorFlow models, and code examples.

1. Select a Wi-Fi network on the board. You can use one of two option:
   - Enter your network name and password with this command:

     ```
     nmcli dev wifi connect <NETWORK_NAME> password <PASSWORD> ifname wlan0
     ```

   - Or use an interactive menu to select the network:
     1. Launch the network manager menu: `nmtui`
     2. Use your keyboard to select **Activate a connection**.
     3. Select a network from the list, enter a password if required, and then quit the menu.

2. Verify your connection with this command:

   ```
   nmcli connection show
   ```

3. You should see your selected network listed in the output.
   For example:

   ```
   NAME                UUID                                  TYPE             DEVICE
   MyNetworkName       61f5d6b2-5f52-4256-83ae-7f148546575a  wifi wlan0
   ```

   You might see other connections listed, such as the **Ethernet-over-USB** connection you're using for **MDT**.

Tip:
Next time you boot up the board, you can connect to the shell over Wi-Fi instead of connecting the USB cable. Just power the board and it will log onto your Wi-Fi automatically, and then you can connect wirelessly with `mdt shell`.

Execution example with (`ip a` command):

```
ip a

..

7: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether f4:f5:e8:86:14:12 brd ff:ff:ff:ff:ff:ff
    inet 192.168.43.226/24 brd 192.168.43.255 scope global dynamic noprefixroute wlan0
       valid_lft 3591sec preferred_lft 3591sec
    inet6 fe80::5926:c27a:f454:6416/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
8: p2p0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 02:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
9: usb0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 02:22:78:0d:f6:df brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.2/24 brd 192.168.100.255 scope global noprefixroute usb0
```

```
       valid_lft forever preferred_lft forever
    inet6 fe80::28c7:c84e:76a9:a28b/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
10: usb1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN group default qlen
1000
    link/ether 02:22:78:0d:f6:de brd ff:ff:ff:ff:ff:ff
    inet 192.168.101.2/24 brd 192.168.101.255 scope global noprefixroute usb1
       valid_lft forever preferred_lft forever
    inet6 fe80::434c:55f:a852:4eee/64 scope link tentative noprefixroute
       valid_lft forever preferred_lft forever
mendel@coy-pig:~$ nmcli connection show
NAME     UUID                                  TYPE      DEVICE
gadget0  c2979f0f-45f3-4159-9ad9-9e8db5aaa626  ethernet  usb0
gadget1  b6f52a8a-a842-446a-a6ae-2aa4a209e8f4  ethernet  usb1
PhoneAP  c2a75c42-645f-40bf-ae71-6404eed5e154  wifi      wlan0
mendel@coy-pig:~$
```

1.1.6 Update the Mendel software

Some of our software updates are delivered with Debian packages separate from the system image, so make sure you have the latest software by running the following commands in the board's shell terminal:

```
sudo date +%Y%m%d -s "20220714"
sudo apt-get update
sudo apt-get dist-upgrade
sudo reboot now
```

Reboot is required for some kernel changes to take effect.

Help!
If you see an error that certificate verification failed, it's probably because the system has not fetched the correct date yet. You can manually set the date with a command like this: **`sudo date +%Y%m%d -s "20210121"`**. Then try to upgrade again.
When the board's LED turns green after reboot, it should be ready for login, so you can reconnect from your computer with MDT:

```
mdt shell
```

Now you're ready to run a TensorFlow Lite model on the Edge TPU!

Note: The **`dist-upgrade`** command updates all system packages for your current Mendel version. If you want to upgrade to a newer version of Mendel, you need to flash a new system image.

1.1.7 Mount an SD card

The board integrates SD card reader that can be used to store local data and/or load the data from other sources. Insert the SD card and check the disk devices available.
Note that **`Disk /dev/mmcblk0: 7.3 GiB, 7818182656 bytes, 15269888 sectors`** is the main EEPROM disk of the device and **`/dev/mmcblk2: 29.7 GiB, 31914983424 bytes, 62333952 sectors`** is the SD card memory.

```
sudo fdisk -l
Disk /dev/ram0: 4 MiB, 4194304 bytes, 8192 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
..
Disk /dev/mmcblk0: 7.3 GiB, 7818182656 bytes, 15269888 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 534A56F4-400F-4E75-AF83-D5E9621BDB42

Device           Start      End  Sectors  Size Type
/dev/mmcblk0p1    1024     9215     8192    4M unknown
/dev/mmcblk0p2    9216   271359   262144  128M unknown
```

```
/dev/mmcblk0p3  271360  4465663  4194304    2G unknown
/dev/mmcblk0p4 4465664 15268863 10803200  5.2G unknown


Disk /dev/mmcblk2: 29.7 GiB, 31914983424 bytes, 62333952 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xc61d35e7
```

Now we can prepare the mount directory (**/sd**) and execute the **mount** command:

```
mendel@coy-pig:~$ mkdir /sd
mkdir: cannot create directory '/sd': File exists
mendel@coy-pig:~$ sudo mount /dev/mmcblk2p1 /sd
mendel@coy-pig:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       5.1G  2.6G  2.3G  53% /
devtmpfs        967M     0  967M   0% /dev
tmpfs           999M     0  999M   0% /dev/shm
tmpfs           999M   17M  983M   2% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           999M     0  999M   0% /sys/fs/cgroup
tmpfs           999M  1.5M  998M   1% /var/log
/dev/mmcblk0p3  2.0G  1.4G  458M  76% /home
/dev/mmcblk0p2  124M   28M   91M  24% /boot
tmpfs           200M  4.6M  196M   3% /run/user/1000
/dev/mmcblk2p1   27G  279M   26G   2% /sd
mendel@coy-pig:~$ cd /sd
mendel@coy-pig:/sd$ ls
coral  lost+found
mendel@coy-pig:/sd$
```

1.1.8 Run a model using the PyCoral API

To get you started with inferncing examples, there are some simple examples that demonstrate how to perform an inference on the Edge TPU using the TensorFlow Lite API (assisted by the PyCoral API).

Execute the following commands from the Dev Board Mini shell to run our image classification example:

Download the example code from **GitHub**:

```
mkdir coral && cd coral
git clone https://github.com/google-coral/pycoral.git
cd pycoral
```

Download the model, labels, and bird photo:

```
bash examples/install_requirements.sh classify_image.py
```

Run the image classifier with the bird photo (shown in figure 2):

```
python3 examples/classify_image.py \
--model test_data/mobilenet_v2_1.0_224_inat_bird_quant_edgetpu.tflite \
--labels test_data/inat_bird_labels.txt \
--input test_data/parrot.jpg
```

Fig 2. `parrot.jpg`

You should see results like this:

```
----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading the model into Edge TPU memory.
158.3ms
14.3ms
14.3ms
14.5ms
14.3ms
-------RESULTS--------
Ara macao (Scarlet Macaw): 0.75781
```

Congrats!
You just performed an **inference on the Edge TPU** using **TensorFlow Lite**.
To demonstrate varying inference speeds, the example repeats the same inference five times. It prints the time to perform each inference and then the top classification result (the label name and the confidence score, from 0 to 1.0).
To learn more about how the code works, take a look at the classify_image.py source code and read our guide about how to run inference with TensorFlow Lite.

1.1.9 Safely shut down the board

Caution: Do not unplug the board while it is running. Doing so could corrupt the system image.

You should shut down the Dev Board Mini as follows:

1. Be sure the board is connected to power through the USB power port (see figure 1).
2. Unplug the USB OTG cable (if connected). This is important, because if the OTG port is connected, the board will immediately reboot upon shutdown.
3. Either press the board power button or run `sudo shutdown now` from the board terminal.
4. When the board's LED turns off, you can unplug the power supply.

1.3 Coral Camera

To perform real-time inferencing with a vision model, you can connect the Dev Board Mini to the Coral Camera. Once you connect your camera, try the demo scripts below.

Note:

- USB cameras are currently not supported with the Dev Board Mini. Because the board's USB port supports USB 2.0 only, the limited bandwidth makes it difficult to sustain both a high frame-rate and high image quality. We're working to support USB cameras, but for optimal performance, we recommend using the CSI camera interface.
- The CSI cable connector on the Dev Board Mini is designed to be compatible with the Coral Camera only.

Connect the Coral Camera

The Coral Camera connects to the CSI connector on the top of the Dev Board Mini.

You can connect the camera to the Dev Board Mini as follows:

- Make sure the board is powered off and unplugged.
- On the top of the Dev Board Mini, locate the "Camera" connector and flip the small black latch so it's facing upward, as shown in figure 1.

The board's camera connector with the latch open

- Slide the flex cable into the connector with the contact pins facing toward the board (the blue strip is facing away from the board), as shown in figure 2.
- Close the black latch.

- The cable inserted to the board and the latch closed
- Likewise, ensure that the other end is secured on the camera module.

The camera cable connected to the camera module

Power on the board and verify it detects the camera by running this command:

```
v4l2-ctl --list-devices
You should see the camera listed as /dev/video1:
platform:mt8167 (platform:mt8167):
        /dev/video0
        /dev/video1

14001000.rdma (platform:mt8173):
        /dev/video2
```

Note: Even when the camera is not connected, you will see `video0` and `video2` listed, because they represent the MediaTek SoC's video decoder and scaler/converter, respectively.

Camera resolution

To fix correct camera resolution for your screen modify **`weston.ini`** file:

`/etc/xdg/weston/weston.ini`

```
[core]
shell=desktop-shell.so
idle-time=0
xwayland=true

[shell]
background-image=/usr/share/weston/background.png
background-type=scale
allow-zap=false
locking=false

[keyboard]
vt-switching=true

[output]
name=HDMI-A-1
mode=1920x1080
```

You may also delete this line completely, and it will then use the highest resolution supported by the monitor (but doing so can degrade the overall system performance if it is higher than **1920x1080**).
For a quick camera test, connect to the board's shell terminal and run the snapshot tool:

`snapshot`

If you have a monitor attached to the board, you'll see the live camera feed. You can press **Spacebar** to save an image to the home directory

Run a demo with the camera

The Mendel system image on the Dev Board Mini includes two demos that perform real-time image classification and object detection.

First, connect a monitor to the board's **micro HDMI** port so you can see the video results.

Then log on to the board (using MDT or the serial console) and run these commands to be sure you have the latest software:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

Note:
The following demo code is optimized for performance on the Dev Board Mini, and you can get the source code from the edgetpuvision Git repo. If you'd like to see other examples using a camera, which are more broadly applicable for other devices (not just the Dev Board Mini), see the examples-camera GitHub repo.

Download the model files

Before you run either demo, you'll need to download the model files on your board.

First, set this environment variable:

```
export DEMO_FILES="$HOME/demo_files"
```

Note:

To keep this **export** permanent append the above line to **.bashrc** file and **source .bashrc** for immediate use.

Then download the following files on your board (be sure you're connected to the internet):

```
# The image classification model and labels file
wget -P ${DEMO_FILES}/
https://github.com/google-coral/test_data/raw/master/mobilenet_v2_1.0_224_quant_edgetpu.tflite

wget -P ${DEMO_FILES}/ https://raw.githubusercontent.com/google-coral/test_data/release-frogfish/
imagenet_labels.txt

# The face detection model (does not require a labels file)
wget -P ${DEMO_FILES}/
https://github.com/google-coral/test_data/raw/master/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tf
lite
```

Note:

The **depo** files are registered in **/usr/bin** directory

```
mendel@coy-pig:/usr/bin$ ls edgetpu_*
edgetpu_classify  edgetpu_classify_server  edgetpu_demo  edgetpu_detect  edgetpu_detect_server
```

Run a classification model

This demo classifies 1,000 different objects shown to the camera:

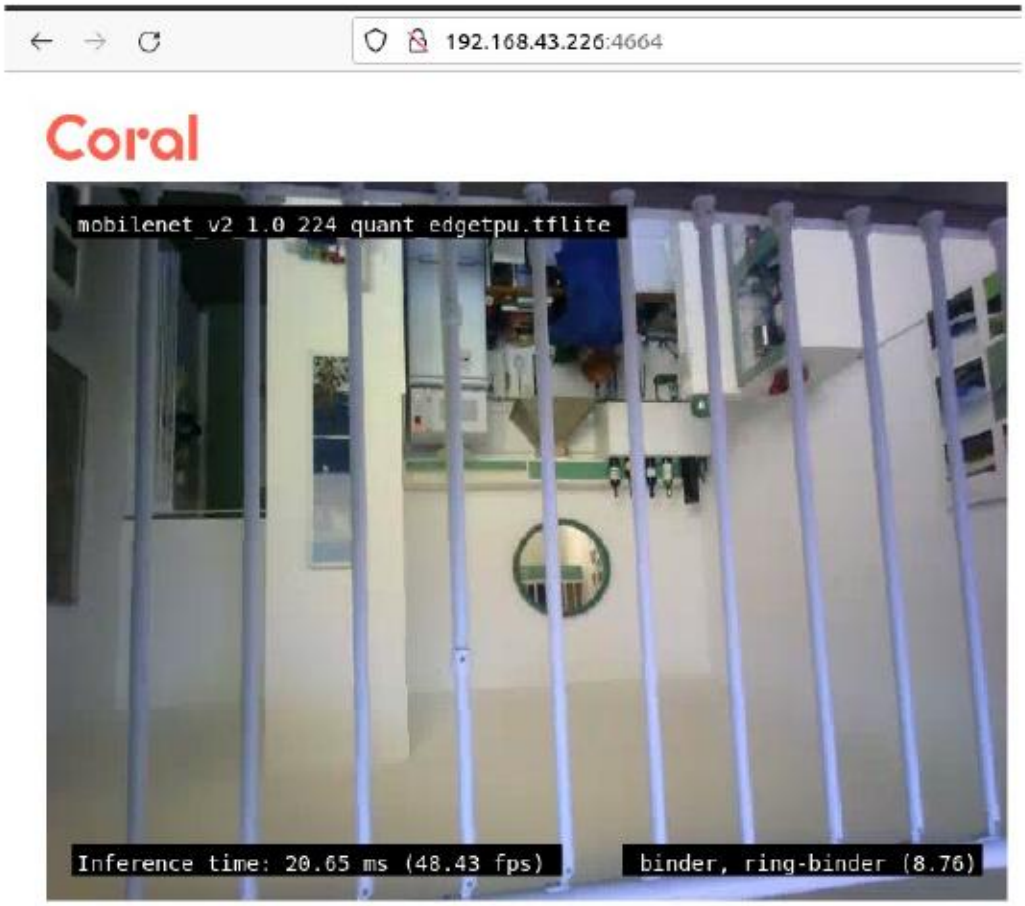
```
edgetpu_classify \
--model ${DEMO_FILES}/mobilenet_v2_1.0_224_quant_edgetpu.tflite \
--labels ${DEMO_FILES}/imagenet_labels.txt
```

If your display is not connected to the Coral Board you can try the **server** version of the same program:

```
mendel@coy-pig:/usr/bin$ edgetpu_classify_server  --model
${DEMO_FILES}/mobilenet_v2_1.0_224_quant_edgetpu.tflite --labels ${DEMO_FILES}/imagenet_labels.txt
INFO:edgetpuvision.streaming.server:Listening on ports tcp: 4665, web: 4664, annexb: 4666
```

Now you can connect to the server with IP address of your card and **WEB port** `4664`.

```
mendel@coy-pig:/usr/bin$ ip a | grep wlan0
7: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    inet 192.168.43.226/24 brd 192.168.43.255 scope global dynamic noprefixroute wlan0
```



Run a face detection model

This demo draws a box around any detected human faces:

```
edgetpu_detect \
--model ${DEMO_FILES}/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

Then try the same application with **server** version:

```
edgetpu_detect_server --model ${DEMO_FILES}/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

Try other example code

We have several other examples that are compatible with almost any camera and any Coral device with an Edge TPU (including the Dev Board). They each show how to stream images from a camera and run classification or detection models . Each example uses a different camera library, such as GStreamer, OpenCV, and PyGame.

To explore the code and run them, see the instructions at:

github.com/google-coral/examples-camera.

https://github.com/google-coral

1.4 Next steps

To run some other types of neural networks, check out our example projects, including examples that perform real-time object detection, pose estimation, keyphrase detection, on-device transfer learning, and more.

If you want to train your own TensorFlow model for the Edge TPU, try these tutorials:

- Retrain an image classification model (MobileNet) (runs in Google Colab)
- Retrain an object detection model (EfficientDet) (runs in Google Colab)
- More model retraining tutorials are available on GitHub.
- **Or to build your own model that's compatible with the Edge TPU, read TensorFlow Models on the Edge TPU**

Part 2 - Readings

TensorFlow models on the Edge TPU

0.0 Introduction

In order for the Edge TPU to provide high-speed neural network performance with a low-power cost, the Edge TPU supports a specific set of neural network operations and architectures. This page describes what types of models are compatible with the Edge TPU and how you can create them, either by compiling your own TensorFlow model or retraining an existing model with transfer-learning.

If you're looking for information about how to run a model, read the Edge TPU inferencing overview.
Or if you just want to try some models, check out our trained models.

0.0.1 Compatibility overview

The Edge TPU is capable of executing deep feed-forward neural networks such as convolutional neural networks (CNN). It supports only TensorFlow Lite models that are fully 8-bit quantized and then compiled specifically for the Edge TPU.

If you're not familiar with TensorFlow Lite, it's a lightweight version of TensorFlow designed for mobile and embedded devices. It achieves low-latency inference in a small binary size—both the TensorFlow Lite models and interpreter kernels are much smaller. TensorFlow Lite models can be made even smaller and more efficient through quantization, which converts 32-bit parameter data into 8-bit representations (which is required by the Edge TPU).

You cannot train a model directly with TensorFlow Lite; instead you must convert your model from a TensorFlow file (such as a `.pb` file) to a TensorFlow Lite file (a **`.tflite`** file), using the TensorFlow Lite converter.
Figure 1 illustrates the basic process to create a model that's compatible with the Edge TPU. Most of the workflow uses standard TensorFlow tools. Once you have a TensorFlow Lite model, you then use our Edge TPU compiler to create a **`.tflite`** file that's compatible with the Edge TPU.

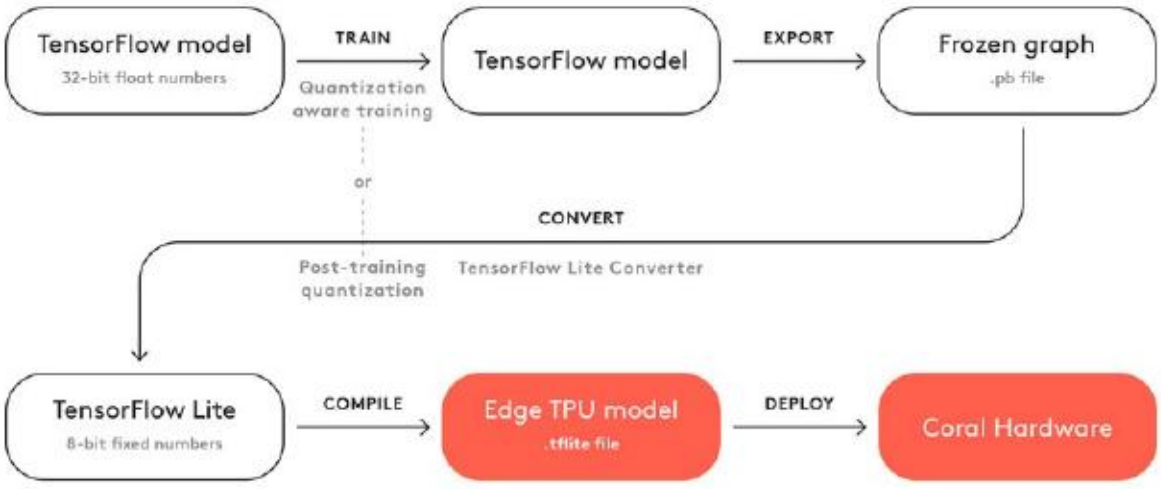


Figure 1. The basic workflow to create a model for the Edge TPU

However, you don't need to follow this whole process to create a good model for the Edge TPU. Instead, you can leverage existing TensorFlow models that are compatible with the Edge TPU by retraining them with your own dataset.

For example, MobileNet is a popular image classification/detection model architecture that's compatible with the Edge TPU. We've created several versions of this model that you can use as a starting point to create your own model that recognizes different objects. To get started, see the next section about how to retrain an existing model with transfer learning.

If you have designed—or plan to design—your own model architecture, then you should read the section below about model requirements.

0.1 Transfer learning

Instead of building your own model and then training it from scratch, you can retrain an existing model that's already compatible with the Edge TPU, using a technique called transfer learning (sometimes also called "fine tuning").

Training a neural network from scratch (when it has no computed weights or bias) can take days-worth of computing time and requires a vast amount of training data. But transfer learning allows you to start with a model that's already trained for a related task and then perform further training to teach the model new classifications using a smaller training dataset. You can do this by retraining the whole model (adjusting the weights across the whole network), but you can also achieve very accurate results by simply removing the final layer that performs classification, and training a new layer on top that recognize your new classes.

Using this process, with sufficient training data and some adjustments to the hyperparameters, you can create a highly accurate TensorFlow model in a single sitting. Once you're happy with the model's performance, simply convert it to TensorFlow Lite and then compile it for the Edge TPU. And because the model architecture doesn't change during transfer learning, you know it will fully compile for the Edge TPU (assuming you start with a compatible model).

To get started without any setup, try our Google Colab retraining tutorials. All these tutorials perform transfer learning in cloud-hosted Jupyter notebooks.

0.1.1 Transfer learning on-device

If you're using an image classification model, you can also perform accelerated transfer learning on the Edge TPU.

Our Python and C++ APIs offer two different techniques for on-device transfer learning:

- Weight imprinting on the last layer (`ImprintingEngine` in Python or C++)
- Backpropagation on the last layer (`SoftmaxRegression` in Python or C++)

In both cases, you must provide a model that's specially designed to allow training on the last layer. The required model structure is different for each API, but the result is basically the same: the last fully-connected layer where classification occurs is separated from the base of the graph. Then only the base of the graph is compiled for the Edge TPU, which leaves the weights in the last layer accessible for training.

More detail about the model architecture is available in the corresponding documents below. For now, let's compare how retraining works for each technique:

Weight imprinting takes the output (the embedding vectors) from the base model, adjusts the activation vectors with L2-normalization, and uses those values to compute new weights in the final layer—it averages the new vectors with those already in the last layer's weights. This allows for effective training of new classes with very few sample images.

- **Backpropagation** is an abbreviated version of traditional backpropagation. Instead of backpropagating new weights to all layers in the graph, it updates only the fully-connected layer at the end of the graph with new weights. This is the more traditional trWeight imprinting takes the output (the embedding vectors) from the base model, adjusts the activation vectors with L2-normalization, and uses those values to compute new weights in the final layer—it averages the new vectors with those already in the last layer's weights. This allows for effective training of new classes with very few sample images.

- **Backpropagation** is an abbreviated version of traditional backpropagation. Instead of backpropagating new weights to all layers in the graph, it updates only the fully-connected layer at the end of the graph with new weights. This is the more traditional training strategy that generally achieves higher accuracy, but it requires more images and multiple training iterations.

When choosing between these training techniques, you might consider the following factors:

- **Training sample size:** Weight imprinting is more effective if you have a relatively small set of training samples: anywhere from 1 to 200 sample images for each class (as few as 5 can be effective and the API sets a maximum of 200). If you have more samples available for training, you'll likely achieve higher accuracy by using them all with backpropagation.
- **Training sample variance:** Backpropagation is more effective if your dataset includes large intra-class variance. That is, if the images within a given class show the subject in significantly different ways, such as in angle or size, then backpropagation probably works better. But if your application operates in an environment where such variance is low, and your training samples thus also have little intra-class variance, then weight imprinting can work very well.
- **Adding new classes:** Only weight imprinting allows you to add new classes to the model after you've begun training. If you're using backpropagation, adding a new class after you've begun training requires that you restart training for all classes. Additionally, weight imprinting allows you to retain the classes from the pre-trained model (those trained before converting the model for the Edge TPU); whereas backpropagation requires all classes to be learned on-device.
- **Model compatibility:** Backpropagation is compatible with more model architectures "out of the box"; you can convert existing, pre-trained MobileNet and Inception models into embedding extractors that are compatible with on-device backpropagation. To use weight imprinting, you must use a model with some very specific layers and then train it in a particular manner before using it for on-device training (currently, we offer a version of MobileNet v1 with the proper modifications).

In both cases, the vast majority of the training process is accelerated by the Edge TPU. And when performing inferences with the retrained model, the Edge TPU accelerates everything except the final classification layer, which runs on the CPU. But because this last layer accounts for only a small portion of the model, running this last layer on the CPU should not significantly affect your inference speed.

To learn more about each technique and try some sample code, see the following pages:

- Retrain a classification model on-device with weight imprinting
- Retrain a classification model on-device with backpropagation

0.1.2 Model requirements

If you want to build a TensorFlow model that takes full advantage of the Edge TPU for accelerated inferencing, the model must meet these basic requirements:

- **Tensor parameters are quantized** (8-bit fixed-point numbers; int8 or uint8).
- **Tensor sizes are constant at compile-time** (no dynamic sizes).
- **Model parameters (such as bias tensors) are constant at compile-time**.
- **Tensors are either 1-, 2-, or 3-dimensional**. If a tensor has more than 3 dimensions, then only the 3 innermost dimensions may have a size greater than 1.
- **The model uses only the operations supported by the Edge TPU** (see table 1 below).

Failure to meet these requirements could mean your model cannot compile for the Edge TPU at all, or only a portion of it will be accelerated, as described in the section below about compiling.

Note:
If you pass a model to the Edge TPU Compiler that uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). So as long as your tensor parameters are quantized, it's okay if the input and output tensors are float because they'll be converted on the CPU

0.1.2.1 Supported operations

When building your own model architecture, be aware that only the operations in the following table are supported by the Edge TPU. If your architecture uses operations not listed here, then only a portion of the model will execute on the Edge TPU.

Note:
When creating a new TensorFlow model, also refer to the list of operations compatible with TensorFlow Lite.

Table 1. All operations supported by the Edge TPU and any known limitations

Operation nQuantizationame	Runtime version*	Known limitations
Add	All	
AveragePool2d	All	No fused activation function.
Concatenation	All	No fused activation function. If any input is a compile-time constant tensor, there must be only 2 inputs, and this constant tensor must be all zeros (effectively, a zero-padding op).
Conv2d	All	Must use the same dilation in x and y dimensions.
DepthwiseConv2d	≤12	Dilated conv kernels are not supported.
	≥13	Must use the same dilation in x and y dimensions.
ExpandDims	≥13	
FullyConnected	All	Only the default format is supported for fully-connected weights. Output tensor is one-dimensional.
L2Normalization	All	
Logistic	All	
LSTM	≥14	Unidirectional LSTM only.
Maximum	All	
MaxPool2d	All	No fused activation function.
Mean	≤12	No reduction in batch dimension. Supports reduction along x- and/or y-dimensions only.
	≥13	No reduction in batch dimension. If a z-reduction, the z-dimension must be multiple of 4.
Minimum	All	
Mul	All	
Pack	≥13	No packing in batch dimension.
Pad	≤12	No padding in batch dimension. Supports padding along x- and/or y-dimensions only.
	≥13	No padding in batch dimension.
PReLU	≥13	Alpha must be 1-dimensional (only the innermost dimension can be >1 size). If using Keras PReLU with 4D input (batch, height, width, channels), then shared_axes must be [1,2] so each filter has only one set of parameters.
Quantize	≥13	
ReduceMax	≥14	Cannot operate on the batch dimension.
ReduceMin	≥14	Cannot operate on the batch dimension.
ReLU	All	
ReLU6	All	
ReLUN1To1	All	
Reshape	All	Certain reshapes might not be mapped for large tensor sizes.
ResizeBilinear	All	Input/output is a 3-dimensional tensor. Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision.
ResizeNearestNeighbor	All	Input/output is a 3-dimensional tensor. Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision.
Rsqrt	≥14	
Slice	All	
Softmax	All	Supports only 1-D input tensor with a max of 16,000 elements.
SpaceToDepth	All	
Split	All	No splitting in batch dimension.
Squeeze	≤12	Supported only when input tensor dimensions that have leading 1s (that is, no relayout needed). For example input tensor with [y][x][z] = 1,1,10 or 1,5,10 is ok. But [y][x][z] = 5,1,10 is not supported.

Operation nQuantizationame	Runtime version*	Known limitations
	≥13	None.
StridedSlice	All	Supported only when all strides are equal to 1 (that is, effectively a Stride op), and with ellipsis-axis-mask == 0, and new-axis-max == 0.
Sub	All	
Sum	≥13	Cannot operate on the batch dimension.
Squared-difference	≥14	
Tanh	All	
Transpose	≥14	
TransposeConv	≥13	

0.1.2.2 Quantization

Quantizing your model means converting all the 32-bit floating-point numbers (such as weights and activation outputs) to the nearest 8-bit fixed-point numbers. This makes the model smaller and faster. And although these 8-bit representations can be less precise, the inference accuracy of the neural network is not significantly affected.
For compatibility with the Edge TPU, you must use either quantization-aware training (recommended) or full integer post-training quantization.

Quantization-aware training (for TensorFlow 1) uses "fake" quantization nodes in the neural network graph to simulate the effect of 8-bit values during training. Thus, this technique requires modification to the network before initial training. This generally results in a higher accuracy model (compared to post-training quantization) because it makes the model more tolerant of lower precision values, due to the fact that the 8-bit weights are learned through training rather than being converted later. It's also currently compatible with more operations than post-training quantization.

Note:
As of August, 2021, the quantization-aware training API with TF2 is not compatible with the object detection API; it is compatible with image classification models only. For more compatibility,z including with the object detection API, you may use quantiation-aware training with TF1 or use post-training quantization with TF2.

Full integer post-training quantization doesn't require any modifications to the network, so you can use this technique to convert a previously-trained network into a quantized model. However, this conversion process requires that you supply a representative dataset. That is, you need a dataset that's formatted the same as the original training dataset (uses the same data range) and is of a similar style (it does not need to contain all the same classes, though you may use previous training/evaluation data). This representative dataset allows the quantization process to measure the dynamic range of activations and inputs, which is critical to finding an accurate 8-bit representation of each weight and activation value.

However, not all TensorFlow Lite operations are currently implemented with an integer-only specification (they cannot be quantized using post-training quantization). By default, the TensorFlow Lite converter leaves those operations in their float format, which is not compatible with the Edge TPU. As described below, the Edge TPU Compiler stops compiling when it encounters an incompatible operation (such as a nonQuantization-quantized op), and the remainder of the model executes on the CPU. So to enforce integer-only quantization, you can instruct the converter to throw an error if it encounters a non-quantizable operation. Read more about integer-only quantization.

For examples of each quantization strategy, see our Google Colab tutorials for model training.
For more details about how quantization works, read the TensorFlow Lite 8-bit quantization spec.

0.1.2.3 Float input and output tensors

As mentioned in the model requirements, the Edge TPU requires 8-bit quantized input tensors. However, if you pass the Edge TPU Compiler a model that's internally quantized but still uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). Likewise, the output is dequantized at the end.

So it's okay if your TensorFlow Lite model uses float inputs/outputs. But beware that if your model uses float input and output, then there will be some amount of latency added due to the data format conversion, though it should be negligible for most models (the bigger the input tensor, the more latency you'll see).

To achieve the best performance possible, we recommend fully quantizing your model so the input and output use `int8` or `uint8` data, which you can do by setting the input and output type with the TF Lite converter, as shown in the TensorFlow docs for integer-only quantization.

0.1.3 Compiling

After you train and convert your model to TensorFlow Lite (with quantization), the final step is to compile it with the Edge TPU Compiler.

If your model does not meet all the requirements listed at the top of this section, it can still compile, but only a portion of the model will execute on the Edge TPU. At the first point in the model graph where an unsupported operation occurs, the compiler partitions the graph into two parts. The first part of the graph that contains only supported operations is compiled into a custom operation that executes on the Edge TPU, and everything else executes on the CPU, as illustrated in figure 2.

Note:
Currently, the Edge TPU compiler cannot partition the model more than once, so as soon as an unsupported operation occurs, that operation and everything after it executes on the CPU, even if supported operations occur later.

Figure 2. The compiler creates a single custom op for all Edge TPU compatible ops, until it encounters an unsupported op; the rest stays the same and runs on the CPU

If you inspect your compiled model (with a tool such as `visualize.py`), you'll see that it's still a TensorFlow Lite model except it now has a custom operation at the beginning of the graph. This custom operation is the only part of your model that is actually compiled—it contains all the operations that run on the Edge TPU. The rest of the graph (beginning with the first unsupported operation) remains the same and runs on the CPU.

If part of your model executes on the CPU, you should expect a significantly degraded inference speed compared to a model that executes entirely on the Edge TPU. We cannot predict how much slower your model will perform in this situation, so you should experiment with different architectures and strive to create a model that is 100% compatible with the Edge TPU. That is, your compiled model should contain only the Edge TPU custom operation.

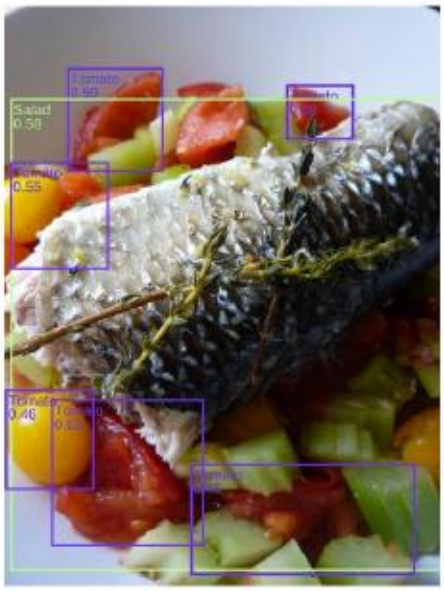
Note: When compilation completes, the Edge TPU compiler tells you how many operations can execute on the Edge TPU and how many must instead execute on the CPU (if any at all). But beware that the percentage of operations that execute on the Edge TPU versus the CPU does not correspond to the overall performance impact—if even a small fraction of your model executes on the CPU, it can potentially slow the inference speed by an order of magnitude (compared to a version of the model that runs entirely on the Edge TPU).

Part 2 - Lab 1

Retrain `EfficientDet` for the Edge TPU with TensorFlow Lite `Model Maker`

In this lab, we'll retrain the **EfficientDet-Lite** object detection model (derived from EfficientDet) using the TensorFlow Lite Model Maker library, and then compile it to run on the Coral Edge TPU. All in about 30 minutes. By default, w'll retrain the model using a publicly available dataset of salad photos, teaching the model to recognize a salad and some of the ingredients.

Here's an example of the salad training results:



If you want to run the notebook with the salad dataset, you can run the whole thing now by clicking **Runtime > Run all** in the **Colab** toolbar. But if you want to use your own dataset, then continue down to Load the training data and follow the instructions there.

Note: If using a custom dataset, beware that if your dataset includes more than 20 classes, you'll probably have slower inference speeds compared to if you have fewer classes. This is due to an aspect of the **EfficientDet** architecture in which a certain layer cannot compile for the Edge TPU when it carries more than 20 classes.

1.1 Import the required packages

```
!pip install -q tflite-model-maker
```

```
     |████████████████████████████████| 577 kB 36.3 MB/s
     |████████████████████████████████| 1.3 MB 54.7 MB/s
     |████████████████████████████████| 1.1 MB 49.2 MB/s
     |████████████████████████████████| 87 kB 6.7 MB/s
     |████████████████████████████████| 3.4 MB 53.1 MB/s
     |████████████████████████████████| 60.2 MB 1.4 MB/s
     |████████████████████████████████| 128 kB 53.3 MB/s
     |████████████████████████████████| 840 kB 69.0 MB/s
     |████████████████████████████████| 77 kB 6.2 MB/s
     |████████████████████████████████| 10.9 MB 56.0 MB/s
     |████████████████████████████████| 238 kB 66.8 MB/s
     |████████████████████████████████| 25.3 MB 1.4 MB/s
     |████████████████████████████████| 352 kB 70.6 MB/s
     |████████████████████████████████| 99 kB 8.7 MB/s
     |████████████████████████████████| 40 kB 4.0 MB/s
     |████████████████████████████████| 1.1 MB 60.9 MB/s
     |████████████████████████████████| 213 kB 58.6 MB/s
  Building wheel for fire (setup.py) ... done
  Building wheel for py-cpuinfo (setup.py) ... done
```

```
import numpy as np
import os

from tflite_model_maker.config import ExportFormat
from tflite_model_maker import model_spec
from tflite_model_maker import object_detector

import tensorflow as tf
assert tf.__version__.startswith('2')

tf.get_logger().setLevel('ERROR')
from absl import logging
logging.set_verbosity(logging.ERROR)
```

1.2 Load the training data

To use the **default salad training dataset**, just run all the code below as-is.
But if you want to train with **your own image dataset**, follow these steps:

1. Be sure your dataset is annotated in Pascal VOC XML (various tools can help create VOC annotations, such as LabelImg). Then create a ZIP file with all your JPG images and XML files (JPG and XML files can all be in one directory or in separate directories).
2. Click the **Files** tab in the left panel and just drag-drop your ZIP file there to upload it.
3. Use the following drop-down option to set **`use_custom_dataset`** to **`True`**.
4. If your dataset is already split into separate directories for training, validation, and testing, also set **`dataset_is_split`** to **`True`**. (If your dataset is not split, leave it False and we'll split it below.)
5. Then skip to Load your own Pascal VOC dataset and follow the rest of the instructions there.

```
use_custom_dataset = False #@param ["False", "True"] {type:"raw"}
dataset_is_split = False #@param ["False", "True"] {type:"raw"}
```

1.3 Load the salads CSV dataset

Model Maker requires that we load our dataset using the **`DataLoader`** API. So in this case, we'll load it from a CSV file that defines 175 images for training, 25 images for validation, and 25 images for testing.

```
if not use_custom_dataset:
        train_data, validation_data, test_data = object_detector.DataLoader.from_csv('gs://cloud-ml-data/img/openimage/csv/salads_ml_use.csv')
```

1.4 Select the model spec

Model Maker supports the **EfficientDet-Lit**e family of object detection models that are compatible with the Edge TPU.
(EfficientDet-Lite is derived from **`EfficientDet`**, which offers state-of-the-art accuracy in a small model size).

There are several model sizes you can choose from:

Model architecture	Size(MB)*	Latency(ms)**	Average Precision***
`EfficientDet-Lite0`	**`5.7`**	**`37.4`**	**`30.4%`**
`EfficientDet-Lite1`	**`7.6`**	**`56.3`**	**`34.3%`**
`EfficientDet-Lite2`	**`10.2`**	**`104.6`**	**`36.0%`**
`EfficientDet-Lite3`	**`14.4`**	**`107.6`**	**`39.4%`**

** File size of the compiled Edge TPU models.*
*** Latency measured on a desktop CPU with a Coral USB Accelerator.*
**** Average Precision is the mAP (mean Average Precision) on the COCO 2017 validation dataset.*

Beware that the `Lite2` and `Lite3` models do **not fit onto the Edge TPU's onboard memory (8MB)**, so you'll see even greater latency when using those, due to the cost of fetching data from the host system memory. Maybe this extra latency is okay for your application, but if it's not and you require the precision of the larger models, then you can pipeline the model across multiple Edge TPUs (more about this when we compile the model below).

For this lab, we'll use `Lite0`:

```
spec = object_detector.EfficientDetLite0Spec()
```

The `EfficientDetLite0Spec` constructor also supports several arguments that specify training options, such as the max number of detections (default is 25 for the TF Lite model) and whether to use Cloud TPUs for training. You can also use the constructor to specify the number of training epochs and the batch size, but you can also specify those in the next step.

1.5 Create and train the model

Now we need to create our model according to the model spec, load our dataset into the model, specify training parameters, and begin training.

Using **Model Maker**, we accomplished all of that with `create()`:

```
model = object_detector.create(train_data=train_data,
                               model_spec=spec,
                               validation_data=validation_data,
                               epochs=50,
                               batch_size=10,
                               train_whole_model=True)
```

```
Epoch 1/10
17/17 [==============================] - 172s 8s/step - det_loss: 1.7640 - cls_loss: 1.1348 - box_loss: 0.0126 - reg_l2_loss: 0.0635 -
loss: 1.8275 - learning_rate: 0.0102 - gradient_norm: 0.6630 - val_det_loss: 1.6719 - val_cls_loss: 1.1088 - val_box_loss: 0.0113 -
val_reg_l2_loss: 0.0635 - val_loss: 1.7355
Epoch 2/10
17/17 [==============================] - 127s 7s/step - det_loss: 1.6282 - cls_loss: 1.0866 - box_loss: 0.0108 - reg_l2_loss: 0.0635 -
loss: 1.6918 - learning_rate: 0.0116 - gradient_norm: 0.7075 - val_det_loss: 1.5651 - val_cls_loss: 1.0278 - val_box_loss: 0.0107 -
val_reg_l2_loss: 0.0635 - val_loss: 1.6286
Epoch 3/10
17/17 [==============================] - 119s 7s/step - det_loss: 1.4867 - cls_loss: 0.9938 - box_loss: 0.0099 - reg_l2_loss: 0.0635 -
loss: 1.5503 - learning_rate: 0.0103 - gradient_norm: 1.2108 - val_det_loss: 1.4812 - val_cls_loss: 0.9758 - val_box_loss: 0.0101 -
val_reg_l2_loss: 0.0635 - val_loss: 1.5447
Epoch 4/10
17/17 [==============================] - 122s 7s/step - det_loss: 1.3540 - cls_loss: 0.8835 - box_loss: 0.0094 - reg_l2_loss: 0.0635 -
loss: 1.4175 - learning_rate: 0.0084 - gradient_norm: 1.5609 - val_det_loss: 1.3610 - val_cls_loss: 0.8809 - val_box_loss: 0.0096 -
val_reg_l2_loss: 0.0635 - val_loss: 1.4246
Epoch 5/10
17/17 [==============================] - 143s 8s/step - det_loss: 1.2162 - cls_loss: 0.7792 - box_loss: 0.0087 - reg_l2_loss: 0.0636 -
loss: 1.2798 - learning_rate: 0.0063 - gradient_norm: 1.8290 - val_det_loss: 1.2999 - val_cls_loss: 0.8335 - val_box_loss: 0.0093 -
val_reg_l2_loss: 0.0636 - val_loss: 1.3635
Epoch 6/10
17/17 [==============================] - 125s 7s/step - det_loss: 1.1629 - cls_loss: 0.7289 - box_loss: 0.0087 - reg_l2_loss: 0.0636 -
loss: 1.2265 - learning_rate: 0.0041 - gradient_norm: 1.7212 - val_det_loss: 1.2628 - val_cls_loss: 0.8070 - val_box_loss: 0.0091 -
val_reg_l2_loss: 0.0636 - val_loss: 1.3264
Epoch 7/10
17/17 [==============================] - 123s 7s/step - det_loss: 1.1153 - cls_loss: 0.7037 - box_loss: 0.0082 - reg_l2_loss: 0.0636 -
loss: 1.1789 - learning_rate: 0.0023 - gradient_norm: 1.7256 - val_det_loss: 1.2004 - val_cls_loss: 0.7532 - val_box_loss: 0.0089 -
val_reg_l2_loss: 0.0636 - val_loss: 1.2640
Epoch 8/10
17/17 [==============================] - 128s 8s/step - det_loss: 1.1370 - cls_loss: 0.7149 - box_loss: 0.0084 - reg_l2_loss: 0.0636 -
loss: 1.2005 - learning_rate: 8.7051e-04 - gradient_norm: 1.8087 - val_det_loss: 1.1611 - val_cls_loss: 0.7198 - val_box_loss: 0.0088 -
val_reg_l2_loss: 0.0636 - val_loss: 1.2246
Epoch 9/10
17/17 [==============================] - 122s 7s/step - det_loss: 1.0974 - cls_loss: 0.6846 - box_loss: 0.0083 - reg_l2_loss: 0.0636 -
loss: 1.1609 - learning_rate: 1.2989e-04 - gradient_norm: 1.6174 - val_det_loss: 1.1483 - val_cls_loss: 0.7096 - val_box_loss: 0.0088 -
val_reg_l2_loss: 0.0636 - val_loss: 1.2119
Epoch 10/10
17/17 [==============================] - 131s 8s/step - det_loss: 1.1133 - cls_loss: 0.7055 - box_loss: 0.0082 - reg_l2_loss: 0.0636 -
loss: 1.1769 - learning_rate: 1.2745e-04 - gradient_norm: 1.7946 - val_det_loss: 1.1361 - val_cls_loss: 0.6994 - val_box_loss: 0.0087 -
val_reg_l2_loss: 0.0636 - val_loss: 1.1997
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)     multiple                  3234464
```

```
 class_net/class-predict (Se  multiple                  3501
 parableConv2D)

 box_net/box-predict (Separa  multiple                  2916
 bleConv2D)

=================================================================
Total params: 3,240,881
Trainable params: 3,193,745
Non-trainable params: 47,136
```

1.6 Evaluate the model

Now we'll use the test dataset to evaluate how well the model performs with data it has never seen before. The **`evaluate()`** method provides output in the style of COCO evaluation metrics:

```
model.evaluate(test_data)
```

```
1/1 [==============================] - 12s 12s/step

{'AP': 0.06736898,
 'AP50': 0.15404154,
 'AP75': 0.048867878,
 'APs': -1.0,
 'APm': 0.12764111,
 'APl': 0.06746608,
 'ARmax1': 0.06685495,
 'ARmax10': 0.15428698,
 'ARmax100': 0.19536653,
 'ARs': -1.0,
 'ARm': 0.21666667,
 'ARl': 0.19543435,
 'AP_/Baked Goods': 0.0,
 'AP_/Salad': 0.16721539,
 'AP_/Cheese': 0.009718822,
 'AP_/Seafood': 0.0007750775,
 'AP_/Tomato': 0.1591356}
```

Because the default batch size for EfficientDetLite models is 64, this needs only 1 step to go through all 25 images in the salad test set. You can also specify the **`batch_size`** argument when you call **`evaluate()`**.

1.7 Export to TensorFlow Lite

Next, we'll export the model to the **TensorFlow Lite** format. By default, the **`export()`** method performs full integer post-training quantization, which is exactly what we need for compatibility with the Edge TPU. (Model Maker uses the same dataset we gave to our model spec as a representative dataset, which is required for full-int quantization.) We just need to specify the **export directory** and **format**. By default, it exports to **TF Lite**, but we also want a labels file, so we declare both:

```
TFLITE_FILENAME = 'efficientdet-lite-salad.tflite'
LABELS_FILENAME = 'salad-labels.txt'

model.export(export_dir='.', tflite_filename=TFLITE_FILENAME, label_filename=LABELS_FILENAME,
             export_format=[ExportFormat.TFLITE, ExportFormat.LABEL])
```

1.7.1 Evaluate the TF Lite model

Exporting the model to TensorFlow Lite can affect the model accuracy, due to the reduced numerical precision from quantization and because the original TensorFlow model uses per-class non-max supression (NMS) for post-processing, while the TF Lite model uses global NMS, which is faster but less accurate.

Therefore you should always evaluate the exported TF Lite model and be sure it still meets your requirements:

```
model.evaluate_tflite(TFLITE_FILENAME, test_data)
```

```
25/25 [==============================] - 73s 3s/step

{'AP': 0.048213303,
 'AP50': 0.106587596,
 'AP75': 0.041628063,
 'APs': -1.0,
 'APm': 0.10148624,
 'APl': 0.04836065,
 'ARmax1': 0.041458335,
 'ARmax10': 0.09552807,
 'ARmax100': 0.11094474,
 'ARs': -1.0,
 'ARm': 0.13333334,
 'ARl': 0.11216566,
 'AP_/Baked Goods': 0.0,
 'AP_/Salad': 0.073322594,
 'AP_/Cheese': 0.00259901,
 'AP_/Seafood': 0.000990099,
 'AP_/Tomato': 0.1641548}
```

1.7.2 Try the TFLite model

Just to be sure of things, let's run the model ourselves with an image from the test set.

```
# If you're using a custom dataset, we take a random image from the test set:
if use_custom_dataset:
        images_path = test_images_dir if dataset_is_split else os.path.join(test_dir, "images")
        filenames = os.listdir(os.path.join(images_path))
        random_index = random.randint(0,len(filenames)-1)
        INPUT_IMAGE = os.path.join(images_path, filenames[random_index])
else:
        # Download a test salad image
        INPUT_IMAGE = 'salad-test.jpg'
        DOWNLOAD_URL =
        "https://storage.googleapis.com/cloud-ml-data/img/openimage/3/2520/3916261642_0a504acd60_o.jpg"
        !wget -q -O $INPUT_IMAGE $DOWNLOAD_URL
```

To simplify our code, we'll use the PyCoral API:

```
! python3 -m pip install --extra-index-url https://google-coral.github.io/py-repo/ pycoral
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/, https://google-coral.github.io/py-repo/
Collecting pycoral
  Downloading https://github.com/google-coral/pycoral/releases/download/v2.0.0/pycoral-2.0.0-cp37-cp37m-linux_x86_64.whl (373 kB)
     |████████████████████████████████| 373 kB 8.9 MB/s
Requirement already satisfied: numpy>=1.16.0 in /usr/local/lib/python3.7/dist-packages (from pycoral) (1.21.6)
Requirement already satisfied: Pillow>=4.0.0 in /usr/local/lib/python3.7/dist-packages (from pycoral) (7.1.2)
Collecting tflite-runtime==2.5.0.post1
  Downloading https://github.com/google-coral/pycoral/releases/download/v2.0.0/tflite_runtime-2.5.0.post1-cp37-cp37m-linux_x86_64.whl (1.5 MB)
     |████████████████████████████████| 1.5 MB 57.7 MB/s
Installing collected packages: tflite-runtime, pycoral
Successfully installed pycoral-2.0.0 tflite-runtime-2.5.0.post1
```

```
from PIL import Image
from PIL import ImageDraw
from PIL import ImageFont

import tflite_runtime.interpreter as tflite
from pycoral.adapters import common
from pycoral.adapters import detect
from pycoral.utils.dataset import read_label_file
```

```
def draw_objects(draw, objs, scale_factor, labels):
        """Draws the bounding box and label for each object."""
        COLORS = np.random.randint(0, 255, size=(len(labels), 3), dtype=np.uint8)
        for obj in objs:
                bbox = obj.bbox
                color = tuple(int(c) for c in COLORS[obj.id])
                draw.rectangle([(bbox.xmin * scale_factor, bbox.ymin * scale_factor),
                        (bbox.xmax * scale_factor, bbox.ymax * scale_factor)],
                        outline=color, width=3)
        font = ImageFont.truetype("LiberationSans-Regular.ttf", size=15)
        draw.text((bbox.xmin * scale_factor + 4, bbox.ymin * scale_factor + 4),
                '%s\n%.2f' % (labels.get(obj.id, obj.id), obj.score),
                fill=color, font=font)

# Load the TF Lite model
labels = read_label_file(LABELS_FILENAME)
interpreter = tflite.Interpreter(TFLITE_FILENAME)
interpreter.allocate_tensors()
# Resize the image for input
image = Image.open(INPUT_IMAGE)
_, scale = common.set_resized_input(
        interpreter, image.size, lambda size: image.resize(size, Image.ANTIALIAS))
# Run inference
interpreter.invoke()
objs = detect.get_objects(interpreter, score_threshold=0.4, image_scale=scale)
# Resize again to a reasonable size for display
display_width = 500
scale_factor = display_width / image.width
height_ratio = image.height / image.width
image = image.resize((display_width, int(display_width * height_ratio)))
draw_objects(ImageDraw.Draw(image), objs, scale_factor, labels)
image
```

1.8 Compile for the Edge TPU

First we need to download the Edge TPU Compiler:

```
! 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
```

Before compiling the `.tflite` file for the Edge TPU, it's important to consider whether your model will fit into the Edge TPU memory.
The Edge TPU has approximately **8 MB of SRAM** for caching model paramaters, so any model close to or over 8 MB will not fit onto the Edge TPU memory. That means the inference times are longer, because some model parameters must be fetched from the host system memory.
One way to elimiate the extra latency is to use model pipelining, which splits the model into segments that can run on separate Edge TPUs in series. This can significantly reduce the latency for big models.

The following table provides recommendations for the number of Edge TPUs to use with each **`EfficientDet-Lite`** model.

Model architecture	Minimum TPUs	Recommended TPUs
`EfficientDet-Lite0`	1	1
`EfficientDet-Lite1`	1	1
`EfficientDet-Lite2`	1	2
`EfficientDet-Lite3`	2	2
`EfficientDet-Lite4`	2	3

If you need extra Edge TPUs for your model, then update **`NUMBER_OF_TPUS`** here:

```
NUMBER_OF_TPUS = 1

!edgetpu_compiler $TFLITE_FILENAME
```

```
Edge TPU Compiler version 16.0.384591198
Searching for valid delegate with step 1
Try to compile segment with 267 ops
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 4575 ms.

Input model: efficientdet-lite-salad.tflite
Input size: 4.24MiB
Output model: efficientdet-lite-salad_edgetpu.tflite
Output size: 5.61MiB
On-chip memory used for caching model parameters: 4.24MiB
On-chip memory remaining for caching model parameters: 3.27MiB
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 1
Total number of operations: 267
Operation log: efficientdet-lite-salad_edgetpu.log

Model successfully compiled but not all operations are supported by the Edge TPU. A percentage of the model will instead run on the CPU,
which is slower. If possible, consider updating your model to use only operations supported by the Edge TPU. For details, visit
g.co/coral/model-reqs.
Number of operations that will run on Edge TPU: 264
Number of operations that will run on CPU: 3
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!
```

Beware when using multiple segments: The Edge TPU Comiler divides the model such that all segments have roughly equal amounts of parameter data, but that does not mean all segments have the same latency. Especially

when dividing an SSD model such as EfficientDet, this results in a latency-imbalance between segments, because SSD models have a large post-processing op that actually executes on the CPU, not on the Edge TPU.
So although segmenting your model this way is better than running the whole model on just one Edge TPU, we recommend that you segment the EfficientDet-Lite model using our profiling-based partitioner tool, which measures each segment's latency on the Edge TPU and then iteratively adjusts the segmentation sizes to provide balanced latency between all segments.

1.8.1 Download the files

```
from google.colab import files

files.download(TFLITE_FILENAME)
files.download(TFLITE_FILENAME.replace('.tflite', '_edgetpu.tflite'))
files.download(LABELS_FILENAME)
```

You will find in **Downloads** directory:

1.8.2 Run the model on the Edge TPU

You can now run the model with acceleration on the Edge TPU.
First, download an image of a salad on your device with an Edge TPU. For example, you can use the same one we tested above:

```
wget https://storage.googleapis.com/cloud-ml-data/img/openimage/3/2520/3916261642_0a504acd60_o.jpg -O
salad.jpg
```

And then make sure you have installed the PyCoral API.
Now run an inference using this example code for the PyCoral API. Just clone the **GitHub** repo and run the example, passing it the model files from this tutorial:

```
mendel@coy-pig:mkdir coral
mendel@coy-pig:cd ~/coral
mendel@coy-pig:git clone https://github.com/google-coral/pycoral
mendel@coy-pig:cd ~/coral/pycoral
mendel@coy-pig:~/coral/pycoral$mkdir test_data
mendel@coy-pig:~/coral/pycoral/test_data$ wget
https://storage.googleapis.com/cloud-ml-data/img/openimage/3/2520/3916261642_0a504acd60_o.jpg -O salad.jpg
--2022-09-03 13:12:53--  https://storage.googleapis.com/cloud-ml-data/img/openimage/3/2520/3916261642_0a504acd60_o.jpg
Resolving storage.googleapis.com (storage.googleapis.com)... 216.58.209.240, 216.58.215.48, 216.58.213.80, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|216.58.209.240|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4328316 (4.1M) [image/jpeg]
Saving to: 'salad.jpg'

salad.jpg                          100%[=============================================================================>]   4.13M  93.0KB/s
in 28s

2022-09-03 13:13:24 (150 KB/s) - 'salad.jpg' saved [4328316/4328316]

mendel@coy-pig:~/coral/pycoral$ python3 examples/detect_image.py --model test_data/efficientdet-lite-
salad_edgetpu.tflite --labels test_data/salad-labels.txt --input test_data/salad.jpg --output
test_data/salad_result.jpg
----INFERENCE TIME----
Note: The first inference is slow because it includes loading the model into Edge TPU memory.
340.24 ms
179.97 ms
180.73 ms
181.02 ms
173.09 ms
-------RESULTS--------
Tomato
  id:     4
  score:  0.546875
  bbox:   BBox(xmin=-22, ymin=2403, xmax=934, ymax=3251)
```

```
Tomato
  id:     4
  score:  0.52734375
  bbox:   BBox(xmin=352, ymin=2408, xmax=1216, ymax=3287)
Tomato
  id:     4
  score:  0.51953125
  bbox:   BBox(xmin=447, ymin=397, xmax=1144, ymax=1111)
Tomato
  id:     4
  score:  0.5
  bbox:   BBox(xmin=1817, ymin=481, xmax=2214, ymax=844)
Tomato
  id:     4
  score:  0.4921875
  bbox:   BBox(xmin=820, ymin=486, xmax=1517, ymax=1155)
Tomato
  id:     4
  score:  0.4609375
  bbox:   BBox(xmin=195, ymin=653, xmax=2883, ymax=3415)
Tomato
  id:     4
  score:  0.4140625
  bbox:   BBox(xmin=-59, ymin=2420, xmax=598, ymax=3077)
```

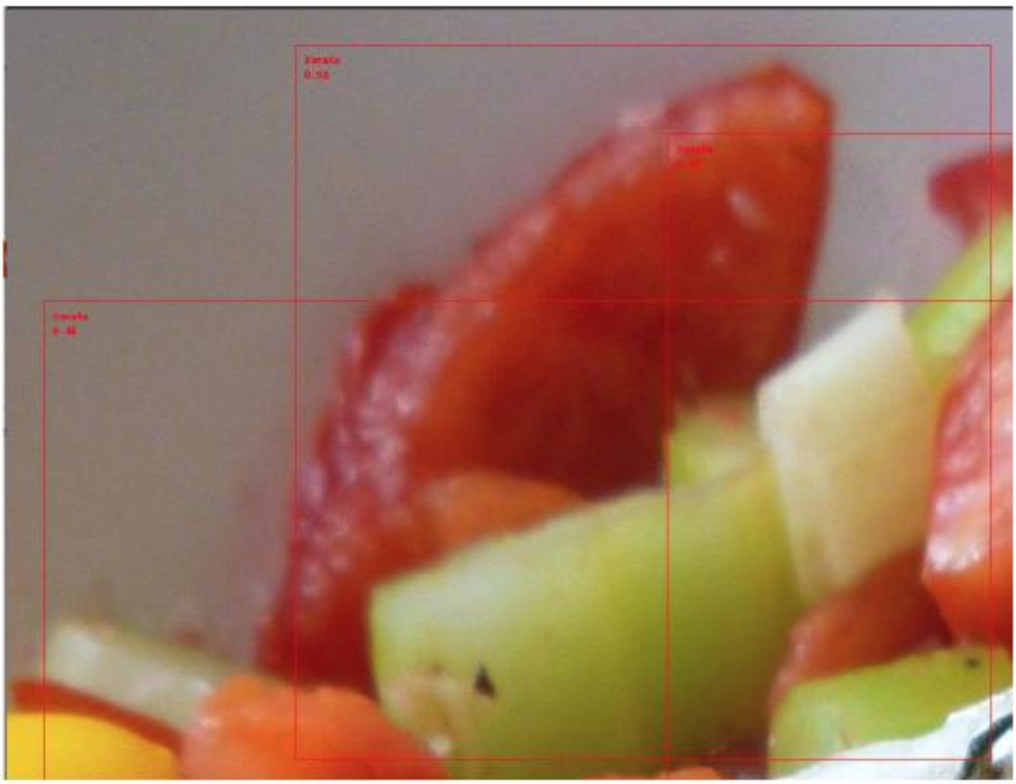


Fig Output image file: --output test_data/salad_result.jpg

1.9 More resources

- For more information about the Model Maker library used in this tutorial, see the TensorFlow Lite Model Maker guide and API reference.
- For other transfer learning tutorials that are compatible with the Edge TPU, see the Colab tutorials for Coral.
- You can also find more examples that show how to run inference on the Edge TPU at coral.ai/examples.

Part 2 - Lab 2

Training and retraining DL models for Edge TPU with TF2

2.0 Introduction

In this lab , we'll use TensorFlow 2 to create an image classification model, train it with a flowers dataset, and convert it to TensorFlow Lite using post-training quantization. Finally, we compile it for compatibility with the Edge TPU (available in Coral devices).

The model is based on a pre-trained version of MobileNet V2. We'll start by retraining only the classification layers, reusing MobileNet's pre-trained feature extractor layers. Then we'll fine-tune the model by updating weights in some of the feature extractor layers. This type of transfer learning is much faster than training the entire model from scratch.
Once it's trained, we'll use post-training quantization to convert all parameters to int8 format, which reduces the model size and increases inferencing speed. This format is also required for compatibility on the Edge TPU.
For more information about how to create a model compatible with the Edge TPU, see the documentation at coral.ai.

Note: This tutorial requires TensorFlow 2.3+ for full quantization, which currently does not work for all types of models. In particular, this tutorial expects a Keras-built model and this conversion strategy currently doesn't work with models imported from a frozen graph.

To start running all the code in this tutorial, select **Runtime > Run all** in the Colab toolbar.

2.1 Import the required libraries

In order to quantize both the input and output tensors, we need `TFLiteConverter` APIs that are available in TensorFlow r2.3 or higher:

```
import tensorflow as tf
assert float(tf.__version__[:3]) >= 2.3

import os
import numpy as np
import matplotlib.pyplot as plt
```

2.2 Prepare the training data

First let's download and organize the flowers dataset we'll use to retrain the model (it contains 5 flower classes). Pay attention to this part so you can reproduce it with your own images dataset. In particular, notice that the "flower_photos" directory contains an appropriately-named directory for each class. The following code randomizes and divides up the photos into training and validation sets, and generates a labels file based on the photo folder names.

```
_URL = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
zip_file = tf.keras.utils.get_file(origin=_URL, fname="flower_photos.tgz", extract=True)
flowers_dir = os.path.join(os.path.dirname(zip_file), 'flower_photos')
```

Next, we use `ImageDataGenerator` to rescale the image data into float values (divide by 255 so the tensor values are between 0 and 1), and call `flow_from_directory()` to create two generators: one for the training dataset and one for the validation dataset.

```
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(
        flowers_dir,
        target_size=(IMAGE_SIZE, IMAGE_SIZE),
        batch_size=BATCH_SIZE,
        subset='training')

val_generator = datagen.flow_from_directory(
        flowers_dir,
        target_size=(IMAGE_SIZE, IMAGE_SIZE),
        batch_size=BATCH_SIZE,
        subset='validation')
```

Run:

```
Found 2939 images belonging to 5 classes.
Found 731 images belonging to 5 classes.
```

On each iteration, these generators provide a batch of images by reading images from disk and processing them to the proper tensor size (224 x 224). The output is a tuple of (images, labels). For example, you can see the shapes here:

```
image_batch, label_batch = next(val_generator)
image_batch.shape, label_batch.shape
```

Run:

```
((64, 224, 224, 3), (64, 5))
```

Now save the class labels to a text file:

```
print (train_generator.class_indices)
labels = '\n'.join(sorted(train_generator.class_indices.keys()))
with open('flower_labels.txt', 'w') as f:
f.write(labels)
```

Run:

```
{'daisy': 0, 'dandelion': 1, 'roses': 2, 'sunflowers': 3, 'tulips': 4}
```

```
!cat flower_labels.txt
```

Run:

```
daisy
dandelion
roses
sunflowers
tulips
```

2.3 Build the model

Now we'll create a model that's capable of transfer learning on just the last fully-connected layer.
We'll start with MobileNet V2 from Keras as the base model, which is pre-trained with the ImageNet dataset (trained to recognize 1,000 classes). This provides us a great feature extractor for image classification and we can then train a new classification layer with our flowers dataset.

2.3.1 Create the base model – feature extractor

When instantiating the **`MobileNetV2`** model, we specify the **`include_top=False`** argument in order to load the network ***without*** **the classification layers at the top**.
Then we set **`trainable`** **false to freeze all the weights in the base model**. This effectively converts the model into a **feature extractor** because all the pre-trained weights and biases are preserved in the lower layers when we begin training for our classification head.

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

# Create the base model from the pre-trained MobileNet V2
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
               include_top=False,
               weights='imagenet')
base_model.trainable = False
```

Run:

```
Downloading data from
https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/
mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9412608/9406464 [==============================] - 0s 0us/step
9420800/9406464 [==============================] - 0s 0us/step
```

2.3.2 Add a classification head

Now we create a **new** **`Sequential`** **model** and pass the frozen **`MobileNet`** model as the base of the graph, and append **new classification layers** so we can set the final output dimension to match the number of classes in our dataset (5 types of flowers).

```
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(),
        tf.keras.layers.Dense(units=5, activation='softmax')
])
```

2.4 Configure the model

Although this method is called **`compile()`**, it's basically a **configuration step** that's required before we can start training.

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

You can see a string summary of the final network with the **`summary()`** method:

```
model.summary()
```

Run:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
```

```
=================================================================
 mobilenetv2_1.00_224 (Funct  (None, 7, 7, 1280)       2257984
 ional)

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

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

 global_average_pooling2d (G  (None, 32)               0
 lobalAveragePooling2D)

 dense (Dense)               (None, 5)                 165

=================================================================
Total params: 2,626,821
Trainable params: 368,837
Non-trainable params: 2,257,984
```

And because the majority of the model graph is frozen in the base model, weights from only the last convolution and dense layers are trainable:

```
print('Number of trainable weights = {}'.format(len(model.trainable_weights)))
```

Run:

```
Number of trainable weights = 4
```

2.5 Train the model

Now we can train the model using data provided by the `train_generator` and `val_generator` that we created at the beginning.
This should take **less than 10 minutes**. (on Colab choose GPU or TPU execution mode)

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

Run:

```
Epoch 1/10
46/46 [==============================] - 29s 318ms/step - loss: 0.6477 - accuracy: 0.7839 - val_loss: 0.4129 - val_accuracy: 0.8454
Epoch 2/10
46/46 [==============================] - 13s 287ms/step - loss: 0.2478 - accuracy: 0.9085 - val_loss: 0.4412 - val_accuracy: 0.8372
Epoch 3/10
46/46 [==============================] - 13s 285ms/step - loss: 0.1659 - accuracy: 0.9425 - val_loss: 0.4201 - val_accuracy: 0.8591
Epoch 4/10
46/46 [==============================] - 16s 340ms/step - loss: 0.1018 - accuracy: 0.9687 - val_loss: 0.4551 - val_accuracy: 0.8577
Epoch 5/10
46/46 [==============================] - 13s 284ms/step - loss: 0.0596 - accuracy: 0.9843 - val_loss: 0.5097 - val_accuracy: 0.8495
Epoch 6/10
46/46 [==============================] - 13s 284ms/step - loss: 0.0386 - accuracy: 0.9935 - val_loss: 0.4970 - val_accuracy: 0.8605
Epoch 7/10
46/46 [==============================] - 13s 285ms/step - loss: 0.0285 - accuracy: 0.9939 - val_loss: 0.5009 - val_accuracy: 0.8755
Epoch 8/10
46/46 [==============================] - 13s 280ms/step - loss: 0.0240 - accuracy: 0.9959 - val_loss: 0.5316 - val_accuracy: 0.8687
Epoch 9/10
46/46 [==============================] - 13s 281ms/step - loss: 0.0167 - accuracy: 0.9976 - val_loss: 0.5328 - val_accuracy: 0.8796
Epoch 10/10
46/46 [==============================] - 13s 280ms/step - loss: 0.0126 - accuracy: 0.9983 - val_loss: 0.5537 - val_accuracy: 0.8714
```

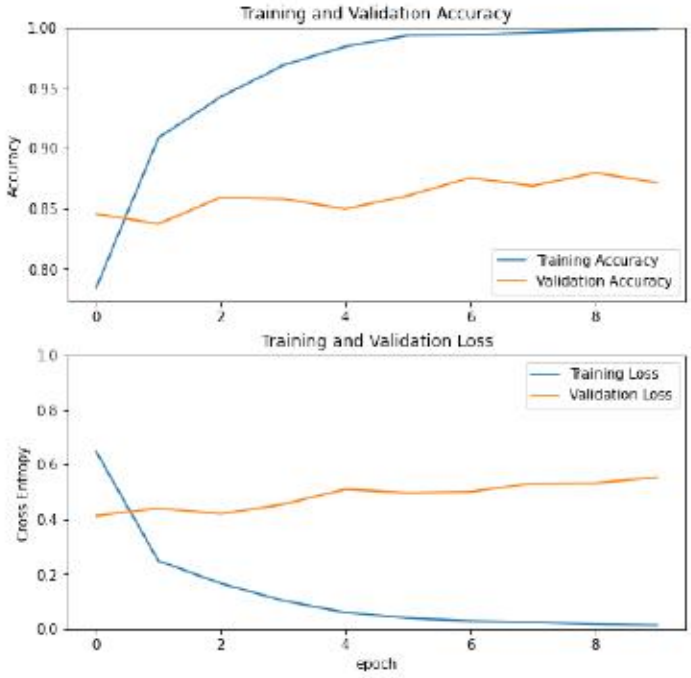
2.5.1 Review the learning curves

```
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()
```



2.6 Fine tune the base model

So far, we've only trained the classification layers—the weights of the pre-trained network were *not* changed. One way we can increase the accuracy is to train (or "fine-tune") more layers from the pre-trained model. That is, we'll **un-freeze some layers** from the base model and adjust those weights (which were originally trained with 1,000 ImageNet classes) so they're better tuned for features found in our flowers dataset.

2.6.1 Un-freeze more layers

So instead of freezing the entire base model, we'll freeze individual layers. First, let's see **how many layers** are in the base model:

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

Run:

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

Let's try freezing just the bottom 100 layers.

```
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
```

2.6.2 Reconfigure the model

Now configure the model again, but this time with a lower learning rate (the default is 0.001).

```
model.compile(optimizer=tf.keras.optimizers.Adam(1e-5),
        loss='categorical_crossentropy',
        metrics=['accuracy'])

model.summary()
```

Run:

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

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

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

 global_average_pooling2d (G  (None, 32)               0
 lobalAveragePooling2D)

 dense (Dense)               (None, 5)                 165

=================================================================
Total params: 2,626,821
Trainable params: 2,230,277
Non-trainable params: 396,544
```

```
print('Number of trainable weights = {}'.format(len(model.trainable_weights)))
```

Run:

```
Number of trainable weights = 58
```

2.6.3 Continue training

Now let's **fine-tune all trainable layers**. This starts with the weights we already trained in the classification layers, so we don't need as many epochs.

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

Run:

```
Epoch 1/5
46/46 [==============================] - 18s 309ms/step - loss: 0.4235 - accuracy: 0.8585 - val_loss: 0.5513 - val_accuracy: 0.8687
Epoch 2/5
46/46 [==============================] - 13s 287ms/step - loss: 0.2064 - accuracy: 0.9279 - val_loss: 0.5645 - val_accuracy: 0.8673
Epoch 3/5
46/46 [==============================] - 13s 285ms/step - loss: 0.1266 - accuracy: 0.9537 - val_loss: 0.5610 - val_accuracy: 0.8687
Epoch 4/5
46/46 [==============================] - 13s 285ms/step - loss: 0.1025 - accuracy: 0.9660 - val_loss: 0.5668 - val_accuracy: 0.8714
Epoch 5/5
46/46 [==============================] - 13s 285ms/step - loss: 0.0760 - accuracy: 0.9741 - val_loss: 0.5668 - val_accuracy: 0.8673
```

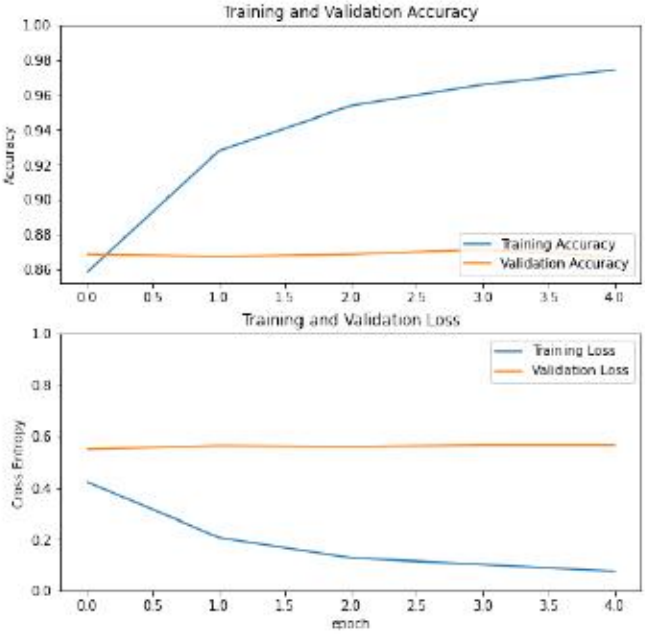
2.6.4 Review the new learning curves

```
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()
```



This is better, but it's not ideal.

The **validation loss** is still higher than the training loss, so there could be some **overfitting** during training.

The overfitting might also be because the new training set is relatively small with less intra-class variance, compared to the original **`ImageNet`** dataset used to train **`MobileNet V2`**.

So this model isn't trained to an accuracy that's production ready, but it works well enough as a demonstration.

Let's move on and convert the model to **TensorFlow Lite**.

2.7 Convert to `TFLite`

Ordinarily, creating a TensorFlow Lite model is just a few lines of code with **`TFLiteConverter`**. For example, this creates a **basic** (**un-quantized**) **TensorFlow Lite model**:

```
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)
```

Run:

```
WARNING:absl:Function `_wrapped_model` contains input name(s) mobilenetv2_1.00_224_input with unsupported characters which will be renamed
to mobilenetv2_1_00_224_input in the SavedModel.
INFO:tensorflow:Assets written to: /tmp/tmp7ej5sgho/assets
INFO:tensorflow:Assets written to: /tmp/tmp7ej5sgho/assets
WARNING:absl:Buffer deduplication procedure will be skipped when flatbuffer library is not properly loaded
```

However, this **`.tflite`** file still uses **floating-point values** for the parameter data, and we need to fully **quantize the model** to **`int8` format**.
To fully **quantize** the model, we need to perform **post-training quantization** with a representative dataset, which requires a few more arguments for the **`TFLiteConverter`**, and a function that builds a dataset that's representative of the training dataset.

So let's convert the model again with **post-training quantization**:

```
# 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)
```

Run:

```
WARNING:absl:Function `_wrapped_model` contains input name(s) mobilenetv2_1.00_224_input with
unsupported characters which will be renamed to mobilenetv2_1_00_224_input in the SavedModel.
INFO:tensorflow:Assets written to: /tmp/tmpypcs37fx/assets
INFO:tensorflow:Assets written to: /tmp/tmpypcs37fx/assets
/usr/local/lib/python3.7/dist-packages/tensorflow/lite/python/convert.py:746: UserWarning: Statistics
for quantized inputs were expected, but not specified; continuing anyway.
  warnings.warn("Statistics for quantized inputs were expected, but not "
WARNING:absl:Buffer deduplication procedure will be skipped when flatbuffer library is not properly
loaded
```

2.7.1 Compare the accuracy

So now we have a **fully quantized `TensorFlow Lite` model**. To be sure the conversion went well, let's evaluate both the raw model and the TensorFlow Lite model.
First check the accuracy of the raw model:

```
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()))
```

Run:

```
Raw model accuracy: 81.250%
```

Now let's check the accuracy of the **`.tflite`** file, using the same dataset.
However, there's no convenient API to evaluate the accuracy of a TensorFlow Lite model, so this code runs several inferences and compares the predictions against ground truth:

```
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()))
```

Run:

```
Quant TF Lite accuracy: 85.938%
```

You might see some, but hopefully not very much accuracy drop between the raw model and the TensorFlow Lite model. But again, **these results are not suitable for production deployment**.

2.8 Compile for the Edge TPU

Finally, we're ready to compile the model for the Edge TPU.
First download the Edge TPU Compiler:

```
! 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
```

Then **compile** the model:

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

Run:

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

Model compiled successfully in 974 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: 3.34MiB
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!
```

That's it.
The compiled model uses the same filename but with "_edgetpu" appended at the end.

2.8.1 Download the model

You can download the converted model and labels file from **Colab** like this:

```
from google.colab import files

files.download('mobilenet_v2_1.0_224_quant_edgetpu.tflite')
files.download('flower_labels.txt')
```

If you get a "Failed to fetch" error here, it's probably because the files weren't done saving. So just **wait a moment and try again**.
Also look out for a browser popup that might need approval to download the files.

2.8.2 Transfer the model to the device with SD card or with `push` command

Steps:

1. Insert SD card
2. Now get ID of SD card

```
mendel@coy-pig:~$ sudo fdisk -l

..
Disk /dev/mmcblk2: 29.7 GiB, 31914983424 bytes, 62333952 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xc61d35e7

mkdir /sd
mendel@coy-pig:~$ sudo mount /dev/mmcblk2p1 /sd
mendel@coy-pig:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       5.1G  2.5G  2.3G  53% /
devtmpfs        967M     0  967M   0% /dev
tmpfs           999M     0  999M   0% /dev/shm
tmpfs           999M  8.7M  991M   1% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           999M     /dev/mmcblk2p10  999M   0% /sys/fs/cgroup
tmpfs           999M  852K  998M   1% /var/log
/dev/mmcblk0p2  124M   28M   91M  24% /boot
/dev/mmcblk0p3  2.0G  1.4G  483M  74% /home
tmpfs           200M  4.6M  196M   3% /run/user/1000
/dev/mmcblk2p1   27G  277M   26G   2% /sd
mendel@coy-pig:~$ cd /sd

mendel@coy-pig:/sd$ ls
coral  lost+found
mendel@coy-pig:/sd$ cd coral
mendel@coy-pig:/sd/coral$ ls
demo_files
mendel@coy-pig:/sd/coral$ cd demo_files/
mendel@coy-pig:/sd/coral/demo_files$ ls
flower_labels.txt  flower_photos  mobilenet_v2_1.0_224_quant_edgetpu.tflite
```

Now you can **copy** the required files to yout working directory.

2.8.3 `ssh` and data transfer with `mdt push/pull` commands

If you prefer to use other **SSH** tools, simply generate your own key and then push that one to the device with mdt pushkey. For example, run the following on your host computer:
Note: This command requires that you already established a secure connection with `mdt shell`.

2.8.3.1 `ssh`

```
ubuntu@bako:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:9zMRH8tERXUOL6QdjvqjoNKCj/1HS4/DQ0W9KhexGDM ubuntu@bako
The key's randomart image is:
+---[RSA 3072]----+
|           .  =o*|
|        E o .B =.|
|         * o+.* o|
|        . +..= + |
|        S.oo. +  |
```

```
|        =.oo .    |
|   . . =.*  *     |
|  .oo ..B... +    |
|  ..o+o. o.       |
+----[SHA256]-----+
ubuntu@bako:~$
ubuntu@bako:~$ mdt pushkey ~/.ssh/id_rsa.pub
Waiting for a device...
Connecting to coy-pig at 192.168.100.2
Pushing /home/ubuntu/.ssh/id_rsa.pub
Key /home/ubuntu/.ssh/id_rsa.pub pushed.
ubuntu@bako:~$ ssh mendel@192.168.100.2
The authenticity of host '192.168.100.2 (192.168.100.2)' can't be established.
ECDSA key fingerprint is SHA256:sgQh23+ygUJgjJwhdq328Odk2ADf4VLH7HfoPb2bLwA.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.100.2' (ECDSA) to the list of known hosts.
Linux coy-pig 4.19.125-mtk #1 SMP PREEMPT Thu Dec 10 02:36:13 UTC 2020 aarch64

The programs included with the Mendel GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Mendel GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Aug 13 12:39:24 2022 from 192.168.100.84
mendel@coy-pig:~$
```

Connection via WiFi link (note that the **-X** parameter is not accepted):

```
ubuntu@bako:~/Téléchargements$ ssh -X mendel@192.168.43.226
X11 forwarding request failed on channel 0
Linux coy-pig 4.19.125-mtk #1 SMP PREEMPT Thu Dec 10 02:36:13 UTC 2020 aarch64

The programs included with the Mendel GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Mendel GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Aug 13 12:45:48 2022 from 192.168.100.84
mendel@coy-pig:~$
```

2.8.3.2 push & pull mdt commands

```
ubuntu@bako:~/Téléchargements$ mdt push  mobilenet_v2_1.0_224_quant_edgetpu.tflite
Waiting for a device...
Connecting to coy-pig at 192.168.100.2
100% |>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| mobilenet_v2_1.0_224_quant_...
ubuntu@bako:~/Téléchargements$
```

push result:

```
mendel@coy-pig:~$ ls
END_USER_LICENSE  data        google-coral  mobilenet_v2_1.0_224_quant_edgetpu.tflite  wifista.sh
coral             demo_files  gpio          project-keyword-spotter
mendel@coy-pig:~$

ubuntu@bako:~/AI$ mdt pull wifista.sh .
Waiting for a device...
Connecting to coy-pig at 192.168.100.2
100% |<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<| wifista.sh
ubuntu@bako:~/AI$ ls
AI.Labs.2021  books  docs  images  lecture  notebooks  projects  python  wifista.sh
ubuntu@bako:~/AI$
```

Now prepare (select) some photos and copy the model , the labels and the prepared photos to mendel@coy-pig:~/coral/pycoral directory.

2.8.4 Run the model on the Edge TPU

You can now run the model on your Coral device with acceleration on the Edge TPU.
To get started, try using your **`.tflite`** model with this code for image classification with the TensorFlow Lite API.

Just follow the instructions on that page to set up your device, copy the **`mobilenet_v2_1.0_224_quant_edgetpu.tflite`** and **`flower_labels.txt`** files to your **Coral Dev. Board at `~/coral/pycoral`**, and pass it a flower photo like this:

```
mendel@coy-pig:~/coral/pycoral$ ls test_data
bird.bmp                                                    mobilenet_v2_1.0_224_inat_bird_quant.tflite
cat.bmp                                                     mobilenet_v2_1.0_224_quant_edgetpu.tflite
coco_labels.txt                                             movenet_single_pose_lightning_ptq_edgetpu.tflite
daisy1.jpg                                                  parrot.jpg
daisy2.jpg                                                  pipeline
dandelion1.jpg                                              rose1.jpg
dandelion2.jpg                                              rose2.jpg
deeplabv3_mnv2_pascal_quant_edgetpu.tflite                  squat.bmp
flower_labels.txt                                           ssd_mobilenet_v2_coco_quant_no_nms_edgetpu.tflite
grace_hopper.bmp                                            ssd_mobilenet_v2_coco_quant_postprocess_edgetpu.tflite
imagenet_labels.txt                                         ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
inat_bird_labels.txt                                        sunflower1.jpg
kite_and_cold.jpg                                           sunflower2.jpg
mobilenet_v1_1.0_224_l2norm_quant_edgetpu.tflite            sunflower.bmp
mobilenet_v1_1.0_224_quant_embedding_extractor_edgetpu.tflite  tulip1.jpg
mobilenet_v2_1.0_224_inat_bird_quant_edgetpu.tflite         tulip2.jpg
mendel@coy-pig:~/coral/pycoral$
```

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

The base of flower photos may be found here:

```
"https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
```

Check out more examples for running inference at coral.ai/examples.

2.8.4.1 Example of inference

```
mendel@coy-pig:~/coral/pycoral$ python3 examples/classify_image.py --model
test_data/mobilenet_v2_1.0_224_quant_edgetpu.tflite --labels test_data/flower_labels.txt --input
test_data/tulip1.jpg

----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading the model into Edge TPU
memory.
129.0ms
14.5ms
14.5ms
14.5ms
14.6ms
-------RESULTS--------
tulips: 0.99609
mendel@coy-pig:~/coral/pycoral$
```

```
  id:     4
  score:  0.4609375
  bbox:   BBox(xmin=195, ymin=653, xmax=2883, ymax=3415)
Tomato
  id:     4
  score:  0.4140625
  bbox:   BBox(xmin=-59, ymin=2420, xmax=598, ymax=3077)
```

2.8.4.2 Source code of `classify_image.py`

```
import argparse
import time
from PIL import Image
from pycoral.adapters import classify
from pycoral.adapters import common
from pycoral.utils.dataset import read_label_file
from pycoral.utils.edgetpu import make_interpreter

def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument('-m', '--model', required=True,
                      help='File path of .tflite file.')
  parser.add_argument('-i', '--input', required=True,
                      help='Image to be classified.')
  parser.add_argument('-l', '--labels',
                      help='File path of labels file.')
  parser.add_argument('-k', '--top_k', type=int, default=1,
                      help='Max number of classification results')
  parser.add_argument('-t', '--threshold', type=float, default=0.0,
                      help='Classification score threshold')
  parser.add_argument('-c', '--count', type=int, default=5,
                      help='Number of times to run inference')
  args = parser.parse_args()

  labels = read_label_file(args.labels) if args.labels else {}
  interpreter = make_interpreter(*args.model.split('@'))
  interpreter.allocate_tensors()
  size = common.input_size(interpreter)
  image = Image.open(args.input).convert('RGB').resize(size, Image.ANTIALIAS)
  common.set_input(interpreter, image)
  print('----INFERENCE TIME----')
  print('Note: The first inference on Edge TPU is slow because it includes',
        'loading the model into Edge TPU memory.')
  for _ in range(args.count):
    start = time.perf_counter()
    interpreter.invoke()
    inference_time = time.perf_counter() - start
    classes = classify.get_classes(interpreter, args.top_k, args.threshold)
    print('%.1fms' % (inference_time * 1000))

  print('-------RESULTS--------')
  for c in classes:
    print('%s: %.5f' % (labels.get(c.id, c.id), c.score))

if __name__ == '__main__':
  main()
```

Part 3 - Lab 1 (audio)

Coral Keyphrase Detector

1.0 Introduction

A keyphrase detector, often referred to a keyword spotter (KWS) is a simple speech processing application that detects the presence of a predefined word or short phrase in stream of audio. This is commonly encountered nowadays with *hotwords* (or wake words) such as *"OK Google"* or *"Alexa"* that are used by digital assistants to tell them when to start listening.

The repo contains a keyphrase detection model that can detect about **140 short keyphrases** such as **`move left`**, **`position four`** in a **two second window of audio**.

Each **output neuron** of the neural network corresponds to **a keyphrase**, the full list can be found here.

The repo also contains all the necessary wrapper code and three example programs:

- **`run_model.py`**: Prints out the top couple of detections each time inference is performed,
- **`run_hearing_snake.py`**: Launches a simple voice controlled version of the game snake.
- **`run_yt_voice_control.py`**: Allows you to control a YouTube player. Requires YouTube to be running in a browser tab and focus to be on the YouTube player.

1.1 Quick Start

Make sure you've got your **Edge TPU device** is correctly configured. If you encounter any problems with the model please ensure that you are running the latest version:

- For Coral DevBoard
- For USB Accelerator
-

To install all the requirements, simply run.

```
sh install_requirements.sh
```

On both the Coral DevBoard and USB Accelerator the model can be tested by executing:

```
python3 run_model.py
```

After a couple of seconds the following output should be visible:

```
ALSA lib pcm_usb_stream.c:486:(_snd_pcm_usb_stream_open) Invalid type for card
Cannot connect to server socket err = No such file or directory
Cannot connect to server request channel
jack server is not running or cannot be started
JackShmReadWritePtr::~JackShmReadWritePtr - Init not done for -1, skipping unlock
JackShmReadWritePtr::~JackShmReadWritePtr - Init not done for -1, skipping unlock

Input microphone devices:
  ID:  1  -  excelsior-card: - (hw:0,1)
  ID:  2  -  excelsior-card: - (hw:0,2)
  ID:  8  -  pulse
  ID:  12  -  default
Using audio device 'default' for index 12
..
        negative (0.980)
        negative (0.996)
        negative (0.996)
        negative (0.996)
        negative (0.996)
```

```
..
```

Say some of the keyphrases the model can understand and you should see them being recognized.

```
        negative (0.992)
        negative (0.680)              no (0.305)
*launch_application* (0.980)        negative (0.020)
*launch_application* (0.996)
*launch_application* (0.996)
        negative (0.996)
        negative (0.996)
        negative (0.914)    position_two (0.039)              no (0.016)
*      move_left* (0.895)         negative (0.082)      move_right (0.016)
*      move_left* (0.977)         negative (0.020)
*      move_left* (0.684)         negative (0.309)
        negative (0.988)
        negative (0.762)       move_right (0.070)       move_down (0.070)
*     move_right* (0.996)
*     move_right* (0.996)
*     move_right* (0.961)         negative (0.039)
        negative (0.992)
        negative (0.949)        volume_up (0.039)     volume_down (0.008)
*      volume_up* (0.977)         negative (0.020)
*      volume_up* (0.996)
*      volume_up* (0.980)         negative (0.020)
        negative (0.906)          turn_up (0.035)         move_up (0.035)
..
```

The default display class shows up to the top 3 detections with their confidences. If something isn't working as expected then check out the troubleshooting section below before trying the other examples.

The full list of commands:

```
what_can_i_say, what_can_you_do, yes, no
start_window, start_application, start_game, start_program,
start_task, start_tab, begin_window, begin_application, begin_game, begin_program, begin_task,
begin_tab, launch_window, launch_application, launch_game, launch_program, launch_task,launch_tab,
open_window, open_application, open_game, open_program, open_task,open_tab,
close_window, close_application, close_game, close_program, close_task, close_tab,
stop_window, stop_application, stop_game, stop_program, stop_task, stop_tab,
terminate_window, terminate_application, terminate_game, terminate_program, terminate_task,
terminate_tab,
exit_window, exit_application, exit_game, exit_program, exit_task, exit_tab,
kill_window, kill_application, kill_game, kill_program, kill_task, kill_tab,
engage, target, switch_on, switch_off, pick_up, volume_up, volume_down,
remove, delete, mute, unmute, silence, reverse,
next_song, next_video, next_game,
last_song, last_video, last_game,
random_song, random_video, random_game,
pause_song, pause_video, pause_game,
stop_song stop_video, start_song, start_video,
previous_song, previous_video,
insert, select, unselect,
move_up, move_down, move_left, move_right, move_backwards, move_forwards,
turn_up, turn_down, turn_left, turn_right, turn_backwards, turn_forwards,
go_up, go_down, go_left, go_right, go_backwards, go_forwards,
channel_zero, position_zero, one_o_clock,channel_one, position_one,
two_o_clock, channel_two, position_two, three_o_clock, channel_three, position_three,
four_o_clock, channel_four, position_four,
five_o_clock, channel_five, position_five,
six_o_clock, channel_six, position_six,
seven_o_clock, channel_seven, position_seven,
eight_o_clock, channel_eight, position_eight,
nine_o_clock, channel_nine, position_nine,
ten_o_clock, channel_ten, position_ten,
eleven_o_clock,channel_eleven position_eleven,
twelve_o_clock, channel_twelve, position_twelve
```

1.2 Hearing Snake

Hearing snake is a version of the classic snake game that can be controlled using voice commands. It **requires a display**, so if you're using the **Coral DevBoard attach a monitor first**.
Be sure **`pygame`** is installed:

```
sudo apt-get install python3-pygame
```

If you have the **USB Accelerator** plugged into a Linux system with an X windows system with simply run:

```
python3 run_hearing_snake.py
```

On the **Coral DevBoard** run:

```
bash run_snake.sh
```

A game window should appear with instructions on how to play.

Part 3 – Lab 2

Programming with Dev Board Mini I/O pins

2.0 Introduction

The Dev Board Mini provides access to several peripheral interfaces through the 40-pin expansion header, including GPIO, I2C, UART, and SPI. This page describes how you can interact with devices connected to these pins.

Because the Dev Board Mini runs Mendel Linux, you can interact with the pins from user space using Linux interfaces such as device files (`/dev`) and sysfs files (`/sys`). There are also several API libraries you can use to program the peripherals connected to these pins. This page describes a few API options, including python-periphery, Adafruit Blinka, and libgpiod.

Note:
If most of your GPIO experience comes from boards like the Raspberry Pi, beware that—although the pin layout on this board might be the same—the vast majority of GPIO APIs out there are board-specific or must be updated for new boards. So the APIs you've used before probably won't work with this board.

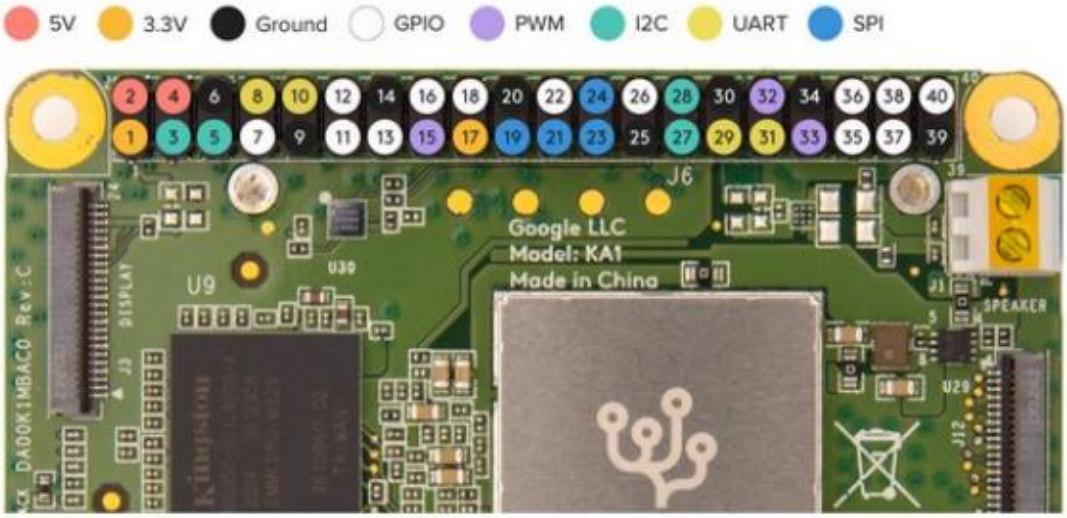


Figure 1. Default pin functions on the 40-pin header

Warning:

When handling the GPIO pins, be cautious to avoid electrostatic discharge or contact with conductive materials (metals). Failure to properly handle the board can result in a short circuit, electric shock, serious injury, death, fire, or damage to your board and other property.

2.0.1 Header pinout

Table 1 shows the header pinout, including the device or sysfs file for each pin, plus the character device numbers.

If you'd like to see the SoC pin names instead, refer to section 4.9 in the Dev Board Mini datasheet. You can also see the pinout by typing `pinout` from the board's shell terminal.

All pins are powered by the 3.3 V power rail, with a max current of ~16 mA on most pins (although the default configuration is 2-4 mA max for most pins).

Chip, line	Device path	Pin function	Pin		Pin function	Device path	Chip, line
		+3.3 V	1	2	+5 V		
	/dev/i2c-3	I2C1_SDA	3	4	+5 V		
	/dev/i2c-3	I2C1_SCL	5	6	Ground		
0, 22	/sys/class/gpio/gpio409	GPIO22	7	8	UART0_TX	/dev/ttyS0	
		Ground	9	10	UART0_RX	/dev/ttyS0	
0, 9	/sys/class/gpio/gpio396	GPIO9	11	12	GPIO36	/sys/class/gpio/gpio423	0, 36
0, 10	/sys/class/gpio/gpio397	GPIO10	13	14	Ground		
0, 2	/sys/class/pwm/pwmchip0/pwm2	PWM_C	15	16	GPIO0	/sys/class/gpio/gpio387	0, 0
		+3.3 V	17	18	GPIO1	/sys/class/gpio/gpio388	0, 1
	/dev/spidev0	SPI_MO	19	20	Ground		
	/dev/spidev0	SPI_MI	21	22	GPIO7	/sys/class/gpio/gpio394	0, 7
	/dev/spidev0	SPI_CLK	23	24	SPI_CSB	/dev/spidev0.0	
		Ground	25	26	GPIO8	/sys/class/gpio/gpio395	0, 8
	/dev/i2c-0	I2C2_SDA	27	28	I2C2_SCL	/dev/i2c-0	
	/dev/ttyS1	UART1_TX	29	30	Ground		
	/dev/ttyS1	UART1_RX	31	32	PWM_A	/sys/class/pwm/pwmchip0/pwm0	0, 0
0, 1	/sys/class/pwm/pwmchip0/pwm1	PWM_B	33	34	Ground		
0, 37	/sys/class/gpio/gpio424	GPIO37	35	36	GPIO13	/sys/class/gpio/gpio400	0, 13
0, 45	/sys/class/gpio/gpio432	GPIO45	37	38	GPIO38	/sys/class/gpio/gpio425	0, 38
		Ground	39	40	GPIO39	/sys/class/gpio/gpio426	0, 39

2.0.2 Program with python-periphery

The python-periphery library provides a generic Linux interface that's built atop the sysfs and character device interface, providing APIs to control GPIO, PWM, I2C, SPI, and UART pins.

By default, the python-periphery package is included with the Mendel system image on the Dev Board Mini. So no installation is required.

The following sections show how to instantiate an object for each pin on the Dev Board Mini header.

2.0.2.1 GPIO

You can instantiate a GPIO object using either the sysfs path (deprecated) or the character device numbers. The following code instantiates each GPIO pin as input **using the character devices**:

```
gpio22 = GPIO("/dev/gpiochip0", 22, "in")  # pin 7
gpio9 = GPIO("/dev/gpiochip0", 9, "in")    # pin 11
gpio36 = GPIO("/dev/gpiochip0", 36, "in")  # pin 12
gpio10 = GPIO("/dev/gpiochip0", 10, "in")  # pin 13
gpio0 = GPIO("/dev/gpiochip0", 0, "in")    # pin 16
gpio1 = GPIO("/dev/gpiochip0", 1, "in")    # pin 18
gpio7 = GPIO("/dev/gpiochip0", 7, "in")    # pin 22
gpio8 = GPIO("/dev/gpiochip0", 8, "in")    # pin 26
```

```
gpio37 = GPIO("/dev/gpiochip0", 37, "in")  # pin 35
gpio13 = GPIO("/dev/gpiochip0", 13, "in")  # pin 36
gpio45 = GPIO("/dev/gpiochip0", 45, "in")  # pin 37
gpio38 = GPIO("/dev/gpiochip0", 38, "in")  # pin 38
gpio39 = GPIO("/dev/gpiochip0", 39, "in")  # pin 40
```

Note: Do not use pin 37 (gpio432) to drive resistive loads directly, due to weak drive strength.

For example, here's how to turn on an LED when you push a button:

```
from periphery import GPIO

led = GPIO("/dev/gpiochip0", 39, "out")  # pin 40
button = GPIO("/dev/gpiochip0", 13, "in")  # pin 36

try:
  while True:
    led.write(button.read())
finally:
  led.write(False)
  led.close()
  button.close()
```

For more examples, see the periphery GPIO documentation

PWM

The following code instantiates each of the PWM pins:

```
pwm_a = PWM(0, 0)  # pin 32
pwm_b = PWM(0, 1)  # pin 33
pwm_c = PWM(0, 2)  # pin 15
```

For usage examples, see the periphery PWM documentation.

I2C

The following code instantiates each of the I2C ports:

```
i2c1 = I2C("/dev/i2c-3")  # pins 3/5
i2c2 = I2C("/dev/i2c-0")  # pins 27/28
```

For usage examples, see the periphery I2C documentation.

Code Example

```
from periphery import I2C

# Open i2c-0 controller
i2c = I2C("/dev/i2c-0")

# Read byte at address 0x100 of EEPROM at 0x50
msgs = [I2C.Message([0x01, 0x00]), I2C.Message([0x00], read=True)]
i2c.transfer(0x50, msgs)
print("0x100: 0x{:02x}".format(msgs[1].data[0]))

i2c.close()
```

SPI

The following code instantiates the SPI port:

```
spi0 = SPI("/dev/spidev0.0", 0, 10000000)  # pins 19/21/23/14 (Mode 0, 10MHz)
```

For usage examples, see the periphery SPI documentation.

Code Example

```
from periphery import SPI

# Open spidev1.0 with mode 0 and max speed 1MHz
spi = SPI("/dev/spidev1.0", 0, 1000000)

data_out = [0xaa, 0xbb, 0xcc, 0xdd]
data_in = spi.transfer(data_out)

print("shifted out [0x{:02x}, 0x{:02x}, 0x{:02x}, 0x{:02x}]".format(*data_out))
print("shifted in  [0x{:02x}, 0x{:02x}, 0x{:02x}, 0x{:02x}]".format(*data_in))

spi.close()
```

UART

The following code instantiates each of the UART ports:

```
uart0 = Serial("/dev/ttyS0", 115200)  # pins 8/10 (115200 baud)
uart1 = Serial("/dev/ttyS1", 9600)    # pins 29/31 (9600 baud)
```

For usage examples, see the periphery Serial documentation.

Code Example

```
from periphery import Serial

# Open /dev/ttyUSB0 with baudrate 115200, and defaults of 8N1, no flow control
serial = Serial("/dev/ttyUSB0", 115200)

serial.write(b"Hello World!")

# Read up to 128 bytes with 500ms timeout
buf = serial.read(128, 0.5)
print("read {:d} bytes: _{:s}_".format(len(buf), buf))

serial.close()
```

2.1 Program with Adafruit Blinka

The Blinka library not only offers a simple API for **GPIO**, **PWM**, **I2C**, and **SPI**, but also provides compatibility with a long list of sensor libraries built for **`CircuitPython`**. That means you can reuse **`CircuitPython`** code for peripherals that was originally used on microcontrollers or other boards such as Raspberry Pi.

To get started, install **`Blinka`** and **`libgpiod`** on your **Dev Board Min**i as follows:

```
sudo apt-get install python3-libgpiod
python3 -m pip install adafruit-blinka
```

Then you can turn on an LED when you push a button as follows (notice this uses pin names from the pinout above):

```
import board
import digitalio

led = digitalio.DigitalInOut(board.GPIO39)  # pin 40
led.direction = digitalio.Direction.OUTPUT

button = digitalio.DigitalInOut(board.GPIO13)  # pin 36
button.direction = digitalio.Direction.INPUT

try:
  while True:
    led.value = button.value
finally:
  led.value = False
  led.deinit()
  button.deinit()
```

For more information, including example code using I2C and SPI, see the Adafruit guide for CircuitPython libraries on Coral. But we suggest you skip their setup guide and install Blinka as shown above.

And beware that their guide was written for the Coral Dev Board, so some pin names are different on the Dev Board Mini. Also check out the Blinka API reference.

2.2 Program with `CircuitPython`

`CircuitPython` is a variant of **`MicroPython`**, a very small version of Python that can fit on a microcontroller.

CircuitPython adds the Circuit part to the Python part. Letting you program in Python and talk to Circuitry like sensors, motors, and LEDs!

For a couple years now we've had CircuitPython for microcontrollers like our SAMD21 series with Feather/Trinket/CircuitPlayground/Metro M0, as well as the ESP8266 WiFi microcontroller, nRF52 bluetooth microcontroller and SAMD51 series.

All of these chips have something in common - they are *microcontrollers* with hardware peripherals like SPI, I2C, ADCs etc. We squeeze Python into 'em and can then make the project portable.
But...sometimes you want to do more than a microcontroller can do. Like HDMI video output, or camera capture, or serving up a website, or just something that takes more memory and computing than a microcontroller board can do.

2.2.1 CircuitPython Libraries on Linux & Google Coral

The next obvious step is to bring CircuitPython ease of use **back** to 'desktop Python'. We've got tons of projects, libraries and example code for CircuitPython on microcontrollers, and thanks to the flexibility and power of Python its pretty easy to get it working with micro-computers like Google Coral or other 'Linux with **GPIO** pins available' **single board computers**.

We'll use a special library called **adafruit_blinka** (named after Blinka, the CircuitPython mascot) to provide the layer that translates the **`CircuitPython`** hardware API to whatever library the Linux board provides. For example, on **Coral** we use the python **`libgpiod`** bindings. For any I2C interfacing we'll use **`ioctl`** messages to the **`/dev/i2c`** **device**.

For SPI we'll use the spidev python library, etc. These details don't matter so much because they all happen underneath the **adafruit_blinka** layer.
The upshot is that any code we have for **`CircuitPython`** will be instantly and easily runnable on Linux computers like **Google Coral**.
In particular, we'll be able to use all of our device drivers - the sensors, led controllers, motor drivers, HATs, bonnets, etc.

And nearly all of these use **I2C** or **SPI**!

Wait, isn't there already something that does this - Periphery?

Periphery is a pure python hardware interface class for Coral, it works just fine for I2C, SPI and GPIO but doesn't work with our drivers as its a different API

By letting you use **`CircuitPython`** libraries on Raspberry Pi via **adafruit_blinka**, **you can unlock all of the drivers** and example code we wrote! **And** you can keep using **periphery** if you like. We save time and effort so we can focus on getting code that works in one place, and you get to reuse all the code we've written already.

2.3 Setting CORAL

Once you've got the Coral flashed, you will be able to set up and test WiFi.
Verify you have a WiFi connection with **sudo ping 8.8.8.8**
The good news about the mendel distribution is it already has Python3 installed by default.

4.3.1 Install `libgpiod`

`libgpiod` is what we use for **`gpio`** toggling. To install run this command:

```
sudo apt-get install libgpiod2
```

After installation you should be able to **`import gpiod`** from within Python3

4.3.2 Update Your Board and Python

```
sudo apt-get update
sudo apt-get upgrade
```

and

```
sudo pip3 install --upgrade setuptools
```

4.3.3 Check UART, I2C and SPI

A vast number of our CircuitPython drivers use UART, I2C and SPI for interfacing. Luckily, they're already enabled. Check by running: 'adafruit/Adafruit_CircuitPython_RFM9x - GitHub)

```
 ls /dev/i2c* /dev/spi*
```

Install the support software with:

```
sudo apt-get install -y python3-smbus python3-dev i2c-tools
sudo adduser mendel i2c
```

ou can get info about the I2C interfaces with:

```
 sudo i2cdetect -l
```

You can test to see what I2C addresses are connected by running:

```
 sudo i2cdetect -y 0 (internal I2C)
```

or

```
sudo i2cdetect -y 1 (pins #3 and #5)
```

and

```
sudo i2cdetect -y 1 (pins #27 and #28)
```

You'll note there looks like an RTC on address **`0x68`** (a common RTC address). Some addresses are pre-allocated by the kernel (unavailable **UU**)
For the GPIO interface chips, you can check with **`sudo gpiodetect`** and **`sudo gpioinfo`** to see the 5 32-pin busses.

To enable **UART1**, you can use the following command:

```
systemctl stop serial-getty@ttymxc0.service
```

2.3.4 Install Python libraries

Now you're ready to install all the python support
Run the following command to install **adafruit_blinka**

```
pip3 install adafruit-blinka
```

The computer will install a few different libraries such as **adafruit-pureio** (our **ioctl-only i2c** library), **spidev** (for SPI interfacing), **Adafruit-GPIO** (for detecting your board) and of course **adafruit-blinka**

That's pretty much it! You're now ready to test.

Create a new file called **blinkatest.py** with **nano** or your favorite text editor and put the following in:

```
import board
import digitalio
import busio

print("Hello blinka!")

# Try to great a Digital input
pin = digitalio.DigitalInOut(board.GPIO_P13)
print("Digital IO ok!")

# Try to create an I2C device : I2C-1
i2c = busio.I2C(board.SCL1, board.SDA1)
print("I2C ok!")

# Try to create an SPI device
spi = busio.SPI(board.SCLK, board.MOSI, board.MISO)
print("SPI ok!")

print("done!")
```

Save it and run at the command line with:

```
sudo python3 blinkatest.py
```

You should see the following, indicating digital i/o, I2C and SPI all worked.

Note that the GPIO pins on the Coral roughly correspond to the Raspberry Pi GPIO

Similarities:

- 5V, 3.3V and GND pins are all in the same locations
- Main I2C port is on pins #3 and #5
- Main SPI port is on pins #19, 21, 23, 24 and 26
- Main UART port is on pins #8 and #10 **note that this is shared with the console so you will conflict with the built in serial console unless you disable the console on these pins**
- I2S is available on same pins as Raspberry Pi

Differences:

- There's a UART3 on pins #7 and #11 but these don't seem to be available (**/dev/ttymxc2** does not exist) so these pins cannot be used for GPIO
- I2S is enabled by default so you cannot use pins #12, 35, 38, 40 for GPIO
- PWM is enabled on pins #15, #32 and #33 so these pins cannot be used for GPIO but you can use them to create PWM outputs
- The secondary I2C port is available for you to use (pins #27 and #28)
- Raspberry Pi **GPIO #22** is known as **GPIO_P13**
- Raspberry Pi **GPIO #23** is known as **GPIO_P16** (output only)
- Raspberry Pi **GPIO #24** is known as **GPIO_P18**

- Raspberry Pi **GPIO #25** is known as **GPIO_P22**
- Raspberry Pi **GPIO #5** is known as **GPIO_P29**
- Raspberry Pi **GPIO #6** is known as **GPIO_P31**
- Raspberry Pi **GPIO #16** is known as **GPIO_P36**
- Raspberry Pi **GPIO #26** is known as **GPIO_P37** (output only)

2.4 Using `CircuitPython` libraries

https://github.com/adafruit/Adafruit_CircuitPython_Bundle/tree/main/libraries/drivers

2.4.1 Test BH1750

```
sudo pip3 install adafruit-circuitpython-bh1750
```

```
import time
import board
import busio
import adafruit_bh1750

i2c = busio.I2C(board.SCL1, board.SDA1)

sensor = adafruit_bh1750.BH1750(i2c)

while True:
    print("%.2f Lux"%sensor.lux)
    time.sleep(1)
```

2.4.2 Test ssd1306

```
sudo pip3 install adafruit-circuitpython-ssd1306
```

```
# Import all board pins.
from board import SCL1, SDA1
import busio
import digitalio
from PIL import Image, ImageDraw, ImageFont

# Import the SSD1306 module.
import adafruit_ssd1306

# Create the I2C interface.
i2c = busio.I2C(SCL1, SDA1)

# Create the SSD1306 OLED class.
# The first two parameters are the pixel width and pixel height.  Change these
# to the right size for your display!
oled = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c)
# Alternatively you can change the I2C address of the device with an addr parameter:
# display = adafruit_ssd1306.SSD1306_I2C(128, 32, i2c, addr=0x31)

# Clear the display.  Always call show after changing pixels to make the display
# update visible!
# Create blank image for drawing.
image = Image.new("1", (oled.width, oled.height))
draw = ImageDraw.Draw(image)

# Load a font in 2 different sizes.
font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf", 28)
font2 = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf", 14)

# Draw the text
```

```
draw.text((0, 0), "Hello!", font=font, fill=255)
draw.text((0, 30), "Hello!", font=font2, fill=255)
draw.text((34, 46), "Hello!", font=font2, fill=255)

# Display image
oled.image(image)
oled.show()
```

2.4.3 Test `vl53l0x` lidar sensor

```
sudo pip3 install adafruit-circuitpython-vl53l0x
```

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

# Simple demo of the VL53L0X distance sensor.
# Will print the sensed range/distance every second.
import time

import board
import busio

import adafruit_vl53l0x

# Initialize I2C bus and sensor.
i2c = busio.I2C(board.SCL1, board.SDA1)
vl53 = adafruit_vl53l0x.VL53L0X(i2c)

# Optionally adjust the measurement timing budget to change speed and accuracy.
# See the example here for more details:
#   https://github.com/pololu/vl53l0x-arduino/blob/master/examples/Single/Single.ino
# For example a higher speed but less accurate timing budget of 20ms:
# vl53.measurement_timing_budget = 20000
# Or a slower but more accurate timing budget of 200ms:
# vl53.measurement_timing_budget = 200000
# The default timing budget is 33ms, a good compromise of speed and accuracy.

# Main loop will read the range and print it every second.
while True:
    print("Range: {0}mm".format(vl53.range))
    time.sleep(1.0)
```

2.4.4 Test `bmp280` sensor

```
sudo pip3 install adafruit-circuitpython-bmp280
```

Attention:

(uses I2C **`0x77`** address needs modification to **`0x76`**)

`line 451:`

```
    def __init__(self, i2c: I2C, address: int = 0x77) -> None:
        from adafruit_bus_device import (  # pylint: disable=import-outside-toplevel
            i2c_device,
        )
```

Code:

```
"""Simpletest Example that shows how to get temperature,
   pressure, and altitude readings from a BMP280"""
import time
import board
```

```
import busio
# in mybmp280  modified  address to 0x76 in place of 0x77 in original adafruit_bmp280 file
import  mybmp280 as adafruit_bmp280

# Create sensor object, communicating over the board's default I2C bus
# Initialize I2C bus and sensor.
i2c = busio.I2C(board.SCL1, board.SDA1)
bmp280 = adafruit_bmp280.Adafruit_BMP280_I2C(i2c)

# OR Create sensor object, communicating over the board's default SPI bus
# spi = board.SPI()
# bmp_cs = digitalio.DigitalInOut(board.D10)
# bmp280 = adafruit_bmp280.Adafruit_BMP280_SPI(spi, bmp_cs)

# change this to match the location's pressure (hPa) at sea level
bmp280.sea_level_pressure = 1022.25

while True:
    print("\nTemperature: %0.1f C" % bmp280.temperature)
    print("Pressure: %0.1f hPa" % bmp280.pressure)
    print("Altitude = %0.2f meters" % bmp280.altitude)
    time.sleep(2)
```

2.5 Using `ThingSpeak` (library)

```
sudo pip3 install thingspeak
sudo pip3 install psutil
```

```
from time import localtime, strftime
import psutil
import time

import thingspeak

channel_id = 1538804  # PUT CHANNEL ID HERE
key = 'YOX31M0EDKO0JATK' # PUT YOUR WRITE KEY HERE

cpu_pc = psutil.cpu_percent()
mem_avail = psutil.virtual_memory().percent

channel = thingspeak.Channel(id=channel_id, api_key=key)
time.sleep(2)
while(True):
        try:
                response = channel.update({'field1': cpu_pc, 'field2': mem_avail})
                print(cpu_pc)
                print(mem_avail)
                print(strftime("%a, %d %b %Y %H:%M:%S", localtime()))
                print(response)
        except:
                print("connection failed")
        time.sleep(20)
```

Table of Contents