Software for Coral

In addition to devices being able to run ML, there's also the ability to extend them with some more ML power. The Coral product is a USB accelerator that allows you to deploy models and execute them on devices that may not have enough power to run them themselves. They come with an Edge TPU built in, and that's a processor that's specifically designed to run TensorFlow based models. The accelerator on the right here is a standalone USB powered device, and the Dev Board on the left is a single-board computer containing an Edge TPU processor.

Mendel OS which is a fork of Debian to carry out development on Coral with Mendel you can build on device machine learning applications with accelerated inference Coral includes an edge TPU compiler for tensorflow light models to be made compatible with the TPU on board the device and ultimately run these compiled models on it in order to make overall processing of humming various actions such as opening a shell installing packages pushing and pulling files to and from the device Coral comes with a command line tool known as the Mendel development tool or MDT, which lets you directly interface with the dev board or the USB accelerator. Google has provided several pre-trained Edge TPU compiled here flight models that you can quickly do prototyping with On a related machine learning tasks.

- Mendel OS
- Edge TPU Compiler
- Mendel Development Tool (`mdt`)
- Edge TPU models : https://coral.withgoogle.com/models/

Raspberry Pi

- Small sized
- Low cost
- Just like a computer
- Accessibility
- Raspbian (OS)

There's the Raspberry Pi which you'll be spending some time with this week. The Raspberry Pi is a low-cost credit card sized computer that plugs into a computer monitor or TV and can use a standard keyboard or Mouse. It's capable of doing almost everything you'd expect a desktop computer to do from browsing the internet to playing high definition video to making spreadsheets word processing even playing games its development is carried out by the Raspberry Pi Foundation, which is a uk-based charity. That works to put the power of computing into the hands of people all over the world just like Carl's Mendel raspberry has its own debian-based operating system known as raspbian.

While the Raspberry Pi can be used for running most tensorflow light models with its accelerator missing. It may not be good at running some resource-hungry models. Reliably this however can be overcome by pairing a coral USB accelerator with the device. So you get accelerated inference.

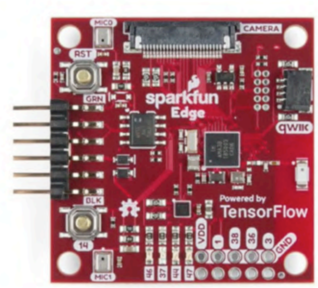
Image credits:https://en.wikipedia.org/wiki/Raspberry_Pi#/media/File:Raspberry_Pi_3_B+_(39906369025).png

Microcontrollers are small low-power devices which can often be embedded within Hardware. This Hardware can be in the form of an appliance or an Internet of Things device since billions of these are manufactured every year. This adds to the trends that we discussed earlier in this lesson microcontrollers and also come with their own share of benefits such as low energy consumption a small form factor, but it comes at the cost of reduced processing power memory and storage.

Some of them are designed to be Optimized for machine learning tasks specifically you may have already seen the connecting to the internet and using it too often can take up a lot of bandwidth and this can result in increased power consumption and higher latency has lower powered devices like microcontrollers are typically designed to work offline. So your model should be planned and implemented accordingly the image on the right here. Is that of a SparkFun Edge microcontroller, which was built by Google in collaboration with SparkFun.

Micro-controllers

- Low-powered
- Small form-factor
- Some specially designed for ML tasks
- No reliance on network connectivity



SparkFun Edge

Image credits: https://www.sparkfun.com/products/15170

Options

- Compile TensorFlow from Source
- Install TensorFlow with pip
- Use TensorFlow Lite Interpreter directly

Build From Source

https://www.tensorflow.org/install/source_rpi

```
curl -sSL https://get.docker.com | sh

sudo docker run hello-world
```

If you want to build from source, the first thing that you'll need is Docker. The build libraries use this. A nice script for installing Docker and making sure that it works is here. Be sure after you've run this to log out and log back in again to get the Docker demand to run.

Build From Source

https://www.tensorflow.org/install/source_rpi

```
git clone https://github.com/tensorflow/tensorflow.git

cd tensorflow
```

Next you'll need to get the TensorFlow source code from GitHub and you clone that with the following command. Make sure you're in the TensorFlow directory after doing that by changing into that directory.

Build From Source

https://www.tensorflow.org/install/source_rpi

```
sudo CI_DOCKER_EXTRA_PARAMS= \
        "-e CI_BUILD_PYTHON=python3 \
         -e CROSSTOOL_PYTHON_INCLUDE_PATH=/usr/include/python3.4" \
         tensorflow/tools/ci_build/ci_build.sh PI-PYTHON3 \
         tensorflow/tools/ci_build/pi/build_raspberry_pi.sh
```

The Python build tools are in the TensorFlow/tools/CI build directory. The docker image handles the dependencies but there are some parameters that you need to send to it. So in order to build, you pseudo the build.shell and the raspberry_ pi.shell commands giving them these extra parameters.

Build From Source

https://www.tensorflow.org/install/source_rpi

```
pip install <your wheel name>
```

It will take some time, but you should end up with a wheel file after that. Then you can pip install that on your Raspberry Pi.

Use Pre-built Packages

https://www.tensorflow.org/install/pip

```
sudo apt update
sudo apt install python3-dev python3-pip
sudo apt install libatlas-base-dev
pip install --upgrade tensorflow
```

If you don't want to build it for yourself, you could also try to use the pre-built packages. Getting these is pretty straight forward, and you can follow the instructions at this URL if you want to see how to do it in the virtual environment on your Pi.

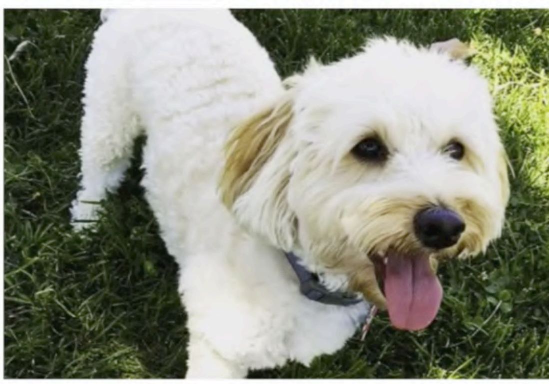
The one dependency that you should always install on a Pi first is the libatlas-base-dev. So be sure to act install that.

Use Interpreter Only

https://www.tensorflow.org/lite/guide/python

```
pip3 install tflite_runtime-1.14.0-cp37-cp37m-linux_armv7l.whl
```

The third option is to use the Python Quick-start, where you just install the TensorFlow interpreter. This is a small and lightweight solution to give you inference alone. In some ways, it's my favorite, but you do have to keep in mind that all you will get is the interpreter. So you might have to do a little bit of low-level work in dealing with the input and output. On the page at this URL, you can see various wheel files for different systems. If you have one that fits your system, you can download and try it out.



dog	0.91
rabbit	0.07
hamster	0.02

Identify classes of different objects in the image

Model

Pre-quantized MobileNet trained on ImageNet

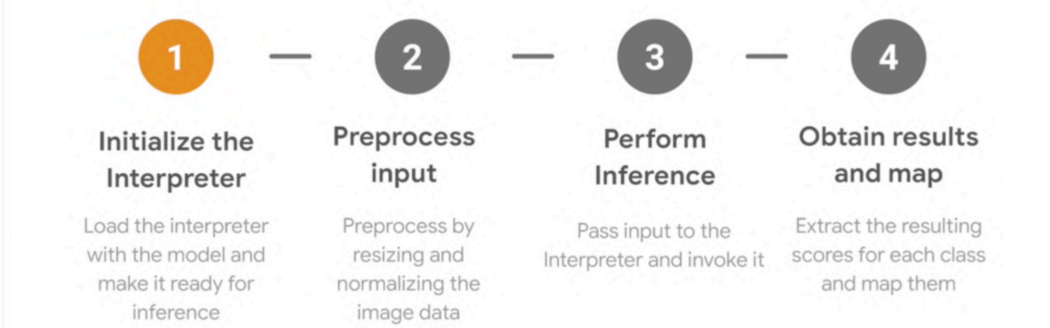
1000 different classes of objects

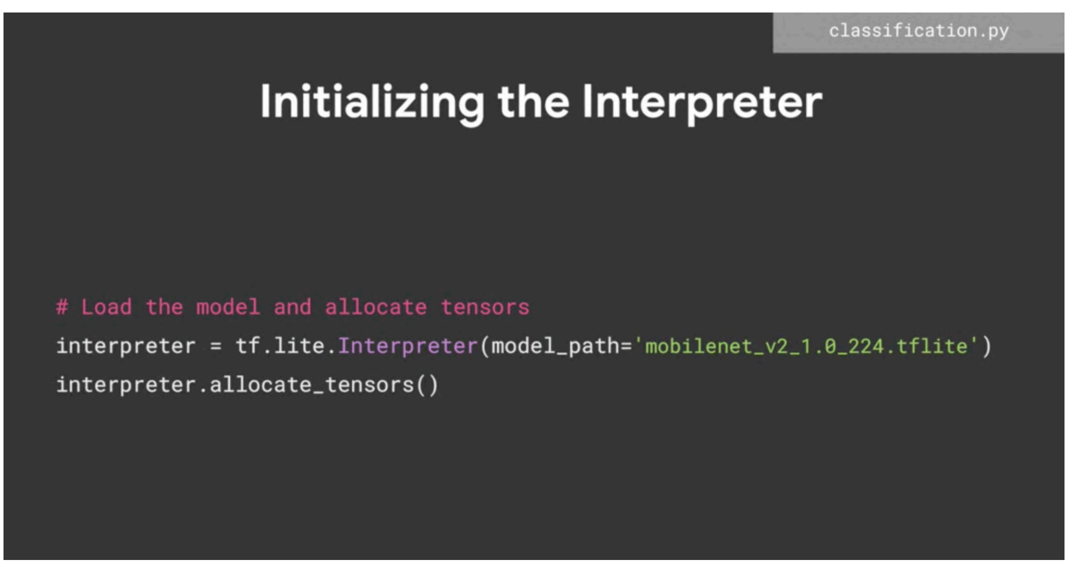
More details on the model can be found at

https://www.tensorflow.org/lite/models/image_classification/overview

Let's see how we would run this on a pie. The model that we'll be using for doing image classification is a pre-optimized MobileNet. We recommend going with a quantized MobileNet V2 model for the ImageNet dataset. You can use this model to classify around 1,000 different classes including people, activities, animals, plants, and places You're always free to experiment with it or try other models.

The best way to get started is to download the course repository and you can see the scripts there. Then in the image classification folder, there's a read me that contains the URLs for the startup model and labels. Add a picture to the directory, download them to the same directory as the script, and then run the script with the command provided in the read me. Be sure to make sure the name of the image you are using in the script matches the one in the folder.





classification.py

Get the model's tensors

```
# Load the model and allocate tensors
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
...
```

1 — Initialize the Interpreter

Load the interpreter with the model and make it ready for inference

2 — Preprocess input

Preprocess by resizing and normalizing the image data

3 — Perform Inference

Pass input to the Interpreter and invoke it

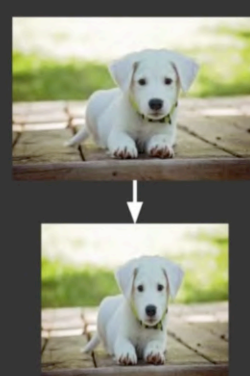
4 — Obtain results and map

Extract the resulting scores for each class and map them

classification.py

Preprocess the image

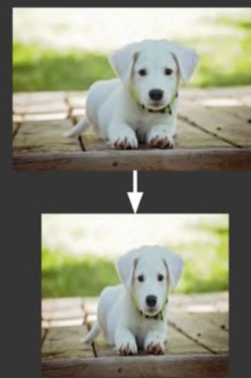
```
# Read image and decode
img = tf.io.read_file(filename)
img_tensor = tf.image.decode_image(img)

# Preprocess image
img_tensor = tf.image.resize(img_tensor, size)
img_tensor = tf.cast(img_tensor, tf.uint8)

# Add a batch dimension
input_data = tf.expand_dims(img_tensor, axis=0)
```



classification.py

Preprocess the image

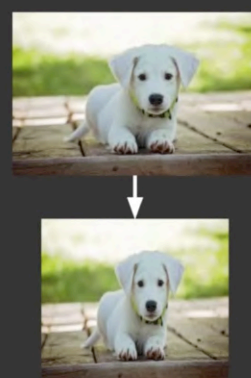
```
# Read image and decode
img = tf.io.read_file(filename)
img_tensor = tf.image.decode_image(img)

# Preprocess image
img_tensor = tf.image.resize(img_tensor, size)
img_tensor = tf.cast(img_tensor, tf.uint8)

# Add a batch dimension
input_data = tf.expand_dims(img_tensor, axis=0)
```



classification.py

Preprocess the image

```
# Read image and decode
img = tf.io.read_file(filename)
img_tensor = tf.image.decode_image(img)

# Preprocess image
img_tensor = tf.image.resize(img_tensor, size)
img_tensor = tf.cast(img_tensor, tf.uint8)

# Add a batch dimension
input_data = tf.expand_dims(img_tensor, axis=0)
```



1 — Initialize the Interpreter
Load the interpreter with the model and make it ready for inference

2 — Preprocess input
Preprocess by resizing and normalizing the image data

3 — Perform Inference
Pass input to the Interpreter and invoke it

4 — Obtain results and map
Extract the resulting scores for each class and map them

classification.py

Perform inference

```python
# Point data to be used for testing the interpreter and run it
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
```

1 — 2 — 3 — 4

1 Initialize the Interpreter

Load the interpreter with the model and make it ready for inference

2 Preprocess input

Preprocess by resizing and normalizing the image data

3 Perform Inference

Pass input to the Interpreter and invoke it

4 Obtain results and map

Extract the resulting scores for each class and map them

classification.py

Report only the top results

```python
# Obtain results
predictions = interpreter.get_tensor(output_details[0]['index'])

# Get indices of the top k results
_, top_k_indices = tf.math.top_k(predictions, k=top_k_results)
top_k_indices = np.array(top_k_indices)[0]

for i in range(top_k_results):
    print(labels[top_k_indices[i]],
          predictions[top_k_indices[i]] / 255.0)
```

Label	Probability
dog	0.91
rabbit	0.07
hamster	0.02

classification.py

Report only the top results

```python
# Obtain results
predictions = interpreter.get_tensor(output_details[0]['index'])

# Get indices of the top k results
_, top_k_indices = tf.math.top_k(predictions, k=top_k_results)
top_k_indices = np.array(top_k_indices)[0]

for i in range(top_k_results):
    print(labels[top_k_indices[i]],
          predictions[top_k_indices[i]] / 255.0)
```

Label	Probability
dog	0.91
rabbit	0.07
hamster	0.02

classification.py

Report only the top results

```python
# Obtain results
predictions = interpreter.get_tensor(output_details[0]['index'])

# Get indices of the top k results
_, top_k_indices = tf.math.top_k(predictions, k=top_k_results)
top_k_indices = np.array(top_k_indices)[0]

for i in range(top_k_results):
    print(labels[top_k_indices[i]],
          predictions[top_k_indices[i]] / 255.0)
```

Label	Probability
dog	0.91
rabbit	0.07
hamster	0.02

Object Detection

Detect multiple objects within an image

Recognize different classes of objects

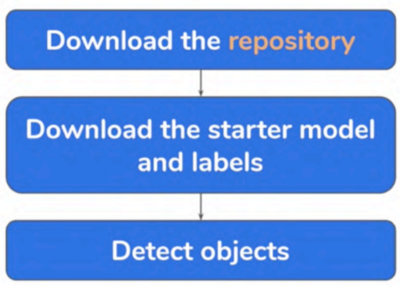
Model

- Pre-optimized MobileNet SSD trained on COCO dataset
- COCO dataset has 80 common object categories
- A labels file to map the model's outputs
- You can download the model (.tflite) and labels (.txt)

http://storage.googleapis.com/download.tensorflow.org/models/tflite/coco_ssd_mobilenet_v1_1.0_quant_2018_06_29.zip

Here, the concept is to detect multiple objects in an image and recognize different classes of objects. The object detection model is a MobileNet SSD trained on the COCO dataset. You can find more details about the model at the URL at this slide. COCO has about 80 different classes of objects, so this app can be used to classify those objects. The only real difference when it comes to running this one as opposed to the image classification is that you have to deal with a greater number of outputs.

Quickstart



First, you start by downloading the starter model and labels along with the script to run it. I'd recommend starting with this pre-trained quantized SSD MobileNet V1 model trained on the COCO dataset. Once you have the model, you're ready to go.

1 — Initialize the Interpreter: Load the interpreter with the model and make it ready for inference by getting the input and output tensors

2 — Preprocess input: Center crop the input image so that the model can accept the input

3 — Perform Inference: Pass the preprocessed input to the Interpreter and invoke it

4 — Fetch the outputs: Extract the outputs as locations, classes, scores and number of detections

detector.py

Initializing the Interpreter

```python
# Load the model and allocate tensors
interpreter = tf.lite.Interpreter(model_path='detect.tflite')
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

Next up you have to get the camera feed and preprocess it. If you don't have access to a Raspberry Pi with a camera, you can follow the steps from the previous tutorial to just use a static image and detect the objects within it.

1 — 2 — 3 — 4

1. Initialize the Interpreter

Load the interpreter with the model and make it ready for inference

2. Get cam feed and preprocess

Gather images from the camera feed and center crop the input image so that the model can accept the input

3. Perform Inference

Pass the preprocessed input to the Interpreter and invoke it

4. Fetch the outputs

Extract the outputs as locations, classes, scores and number of detections

main.py

Raspberry Pi Camera

```python
with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    while True:
        # Input image
        image = np.empty((480, 640, 3), dtype=np.uint8)
        camera.capture(image, 'rgb')
        # Use the frame captured from the stream
```

The Pi Camera is super simple to use in Python, you simply use the Pi Camera Library and in a loop you call camera.capture to get an image. You specify that you want it as RGB, and you can initialize the size that you want in a way that's super close to the input for the tensor with three separate arrays for the RGB channels.

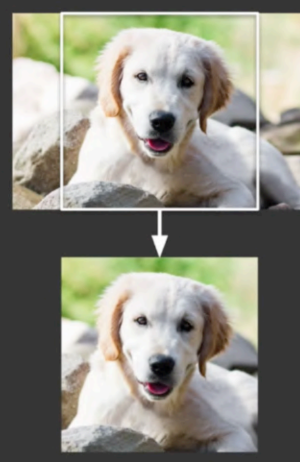
Preprocessing

Image credits: https://www.rspcansw.org.au/what-we-do/adoptions/dogs-and-puppies/

utils.py

Center and Crop

```
# Get the dimensions
height, width, _ = frame.shape # Image shape
new_width, new_height = (300, 300) # Target shape

# Calculate offsets between heights and widths
offset_height = (height - new_height) // 2
offset_width = (width - new_width) // 2

# Crop to the biggest square in the center
image = tf.image.crop_to_bounding_box(frame,
                                      offset_height,
                                      offset_width,
                                      new_height,
                                      new_width)
```



Finally, you crop the image to that square. It's up to you how to do it, some apps will just resize the image into a square potentially distorting it, others will crop like this. They each have their advantages and disadvantages

1 — 2 — 3 — 4

1. Initialize the Interpreter

Load the interpreter with the model and make it ready for inference

2. Get cam feed and preprocess

Gather images from the camera feed and center crop the input image so that the model can accept the input

3. Perform Inference

Pass the preprocessed input to the Interpreter and invoke it

4. Fetch the outputs

Extract the outputs as locations, classes, scores and number of detections

detector.py

Perform inference

```python
def detect(self, image, threshold=0.1):
    # Add a batch dimension
    frame = np.expand_dims(image, axis=0)

    # run model
    self.interpreter.set_tensor(self.input_details[0]['index'], frame)
    self.interpreter.invoke()
```

Now that you have the data in the correct shape, it's time to pass it to the interpreter for inference. The source code in the download, this is in a function called Detect, but the role of the function is exactly the same as you saw in image classification. Expand the dimensions of the image to add a new dimension at the beginning and then pass it to the input tensor at zero index before invoking the interpreter.

The interpreter will run inference on the image using the model and then give you results. Because these results can contain multiple classes and probabilities and bounding boxes for each, processing them is a little more complex than what you saw earlier

1 — 2 — 3 — 4

1. Initialize the Interpreter
Load the interpreter with the model and make it ready for inference

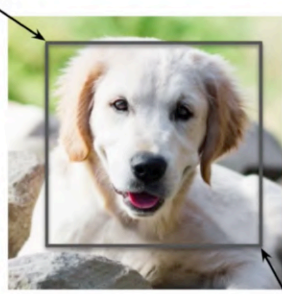
2. Get cam feed and preprocess
Gather images from the camera feed and center crop the input image so that the model can accept the input

3. Perform Inference
Pass the preprocessed input to the Interpreter and invoke it

4. Fetch the outputs
Extract the outputs as locations, classes, scores and number of detections

How a detected object is represented

(top, left)



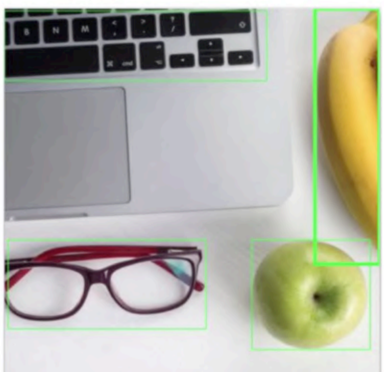
(bottom, right)

Class	Score	Location
Dog	0.92	top, left, bottom, right

How the COCO model sees outputs

index	name	
0	locations	A list of floats in [0, 1] representing ***normalized bounding boxes*** [top left bottom right]
1	classes	A list of integers (output as float) each indicating the ***index of a class label*** from the labels file
2	scores	Array of floats in [0, 1] representing ***probability*** that a class was detected
3	number of detections	Floating point value expressing ***total number of results***

Interpreting the number of results



Class	Score	Location (top, left, bottom, right)
Apple	0.96	275, 257, 407, 379
Glasses	0.89	4, 257, 224, 356
Computer Keyboard	0.77	0, 2, 292, 80
Banana	0.67	345, 0, 417, 284

detector.py

Fetching the outputs

```
# Normalized coordinates of the detected objects
boxes = interpreter.get_tensor(output_details[0]['index'])[0]

# Recognized classes of objects
classes = interpreter.get_tensor(output_details[1]['index'])[0]

# Probabilities of the detected classes
scores = interpreter.get_tensor(output_details[2]['index'])[0]

# Maximum number of results
num_detections = interpreter.get_tensor(output_details[3]['index'])[0]
```

Let's take a look at the code. Earlier, we got output details index for all of the results, but now there are four tensors, we simply get the output details number index. This gives us our boxes, classes, scores and number of detections for zero, one, two and three respectively.

detector.py

Discarding less relevant results

```
min_score_thresh = 0.6
number_boxes = boxes.shape[0]
detected_boxes = []
probabilities = []
categories = []

for i in range(number_boxes):
    if scores is None or scores[i] > min_score_thresh:
        box = tuple(boxes[i].tolist()) # [top, left, bottom, right]
        detected_boxes.append(box)
        probabilities.append(scores[i])
        categories.append(self.category_index[classes[i]])
```

If we want to discard lower quality results, we can do so by setting a threshold for example, 0.6 and then iterating through the list and throwing away those that are either empty or below this threshold.

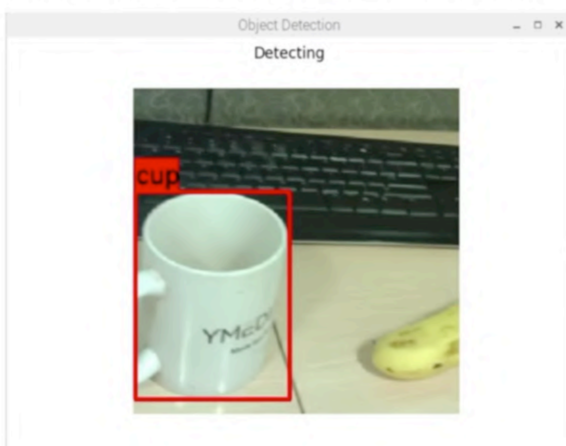
visualization_utils.py

If we want to discard lower quality results, we can do so by setting a threshold for example, 0.6 and then iterating through the list and throwing away those that are either empty or below this threshold.

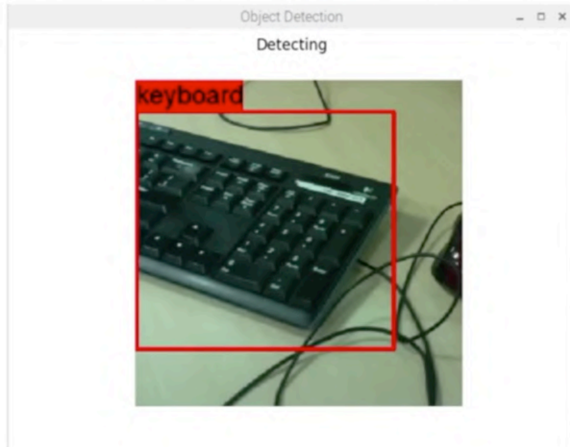
Reporting results

```
# Convert normalized boxes to regular bounding boxes
(top, bottom, left, right) = (top * im_height, bottom * im_height,
                              left * im_width, right * im_width)

# Draw lines for the detected boxes
draw.line([(left, top), (left, bottom), (right, bottom), (right, top),
          (left, top)], width=thickness, fill=color)

# Draw the display string (predicted class)
draw.text(
    # Calculate position of text to be placed at the top-left corner
    (left + margin, text_bottom - text_height - margin),
    display_str,
    fill='black',
    font=font)
```

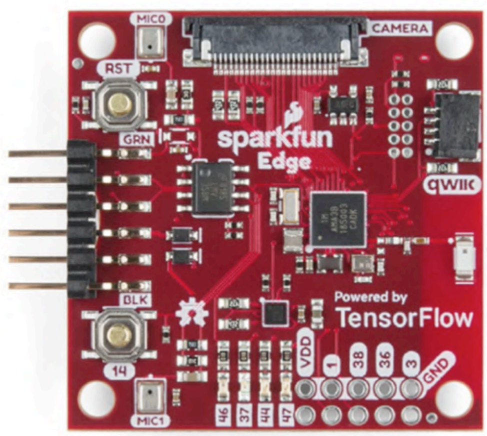
https://www.youtube.com/watch?v=e-Gu6Sp4OUQ

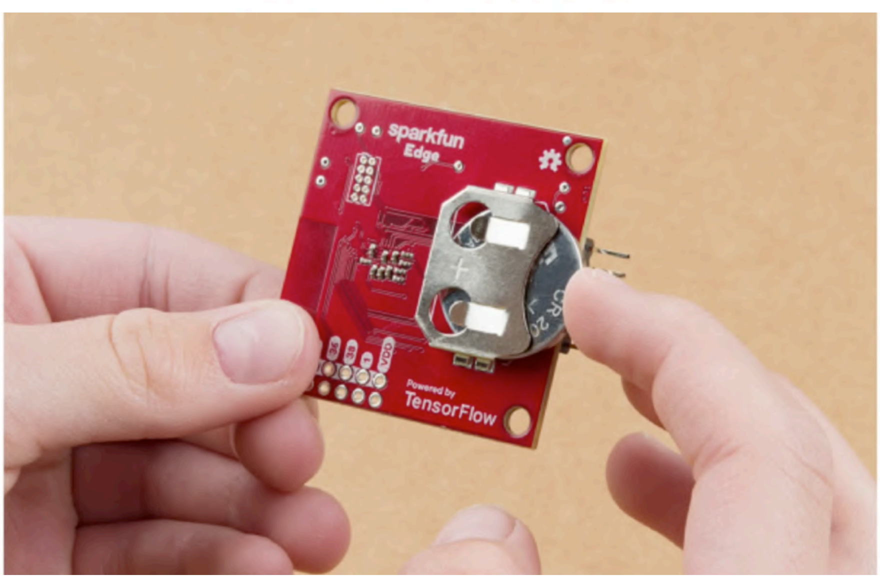


https://www.youtube.com/watch?v=e-Gu6Sp4OUQ

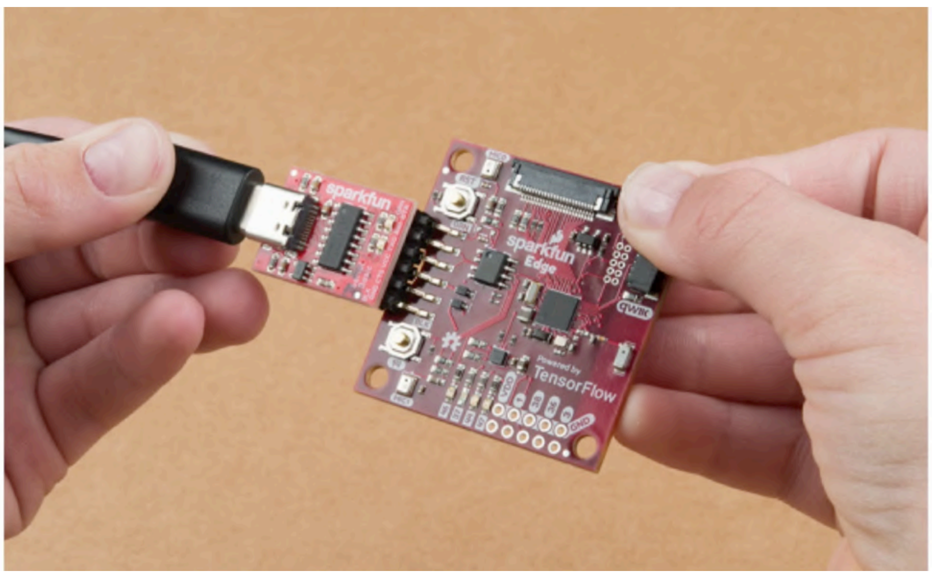


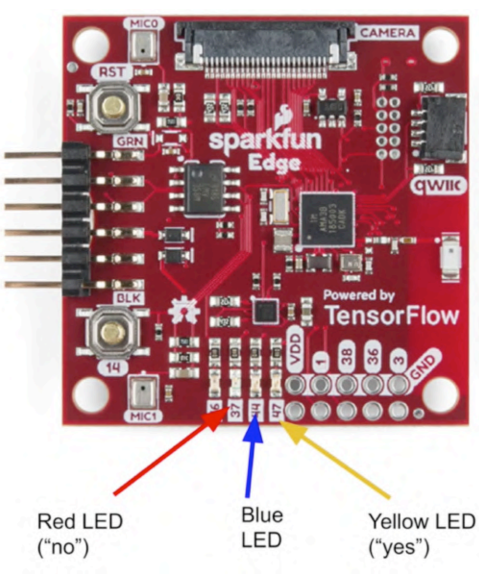
https://www.sparkfun.com/products/15170

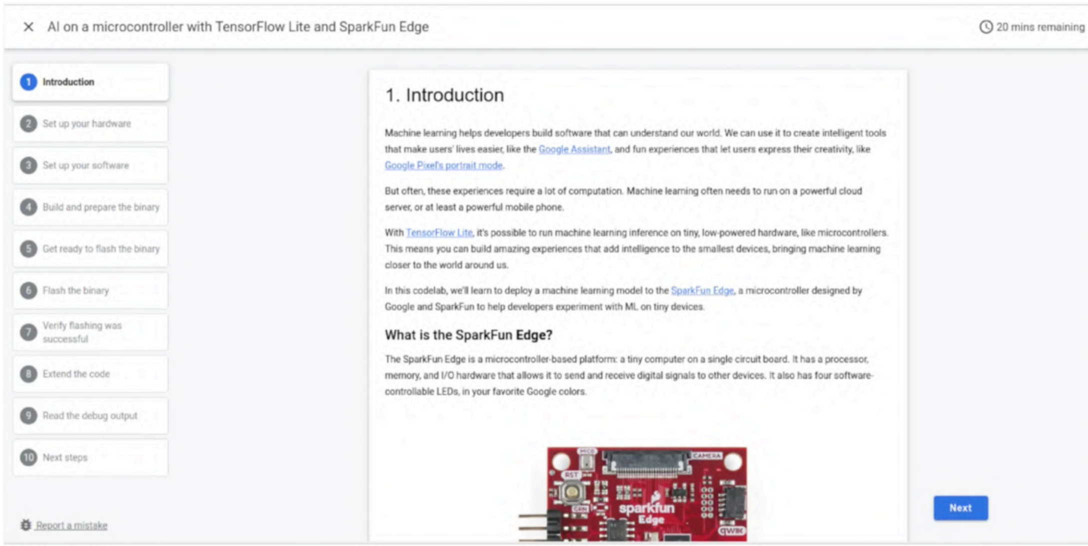




https://www.sparkfun.com/products/15096







AI on a microcontroller with TensorFlow Lite and SparkFun Edge

20 mins remaining

1. Introduction
2. Set up your hardware
3. Set up your software
4. Build and prepare the binary
5. Get ready to flash the binary
6. Flash the binary
7. Verify flashing was successful
8. Extend the code
9. Read the debug output
10. Next steps

Report a mistake

1. Introduction

Machine learning helps developers build software that can understand our world. We can use it to create intelligent tools that make users' lives easier, like the Google Assistant, and fun experiences that let users express their creativity, like Google Pixel's portrait mode.

But often, these experiences require a lot of computation. Machine learning often needs to run on a powerful cloud server, or at least a powerful mobile phone.

With TensorFlow Lite, it's possible to run machine learning inference on tiny, low-powered hardware, like microcontrollers. This means you can build amazing experiences that add intelligence to the smallest devices, bringing machine learning closer to the world around us.

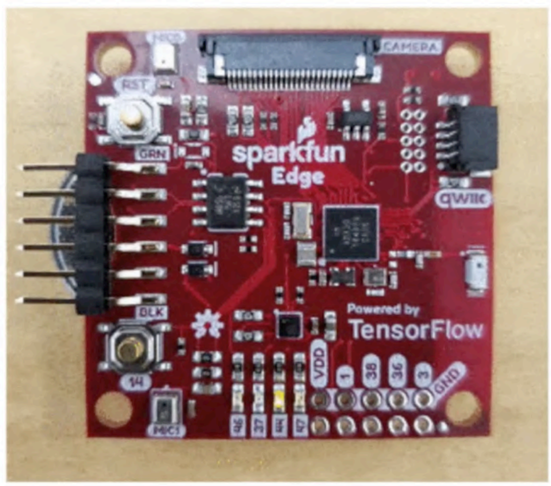
In this codelab, we'll learn to deploy a machine learning model to the SparkFun Edge, a microcontroller designed by Google and SparkFun to help developers experiment with ML on tiny devices.

What is the SparkFun Edge?

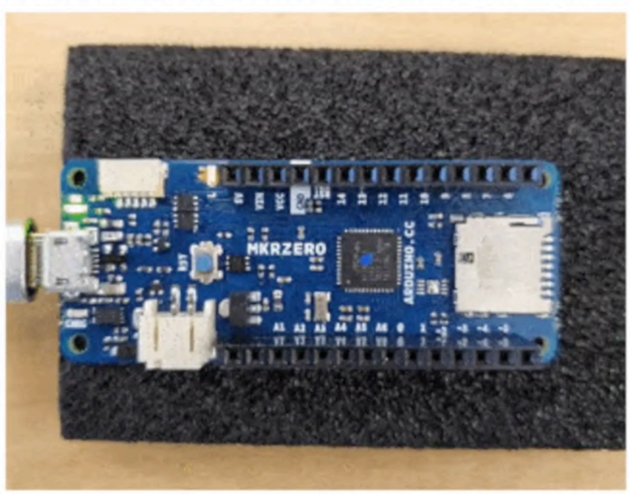
The SparkFun Edge is a microcontroller-based platform: a tiny computer on a single circuit board. It has a processor, memory, and I/O hardware that allows it to send and receive digital signals to other devices. It also has four software-controllable LEDs, in your favorite Google colors.

Next

https://www.tensorflow.org/lite/microcontrollers/get_started



https://www.tensorflow.org/lite/microcontrollers/get_started



https://www.tensorflow.org/lite/microcontrollers/get_started