POLITECNICO
MILANO 1863

Hardware Architectures for Embedded and Edge AI

Prof Manuel Roveri – manuel.roveri@polimi.it
Massimo Pavan – massimo.pavan@polimi.it

Exercise session 4 – TinyML and TensorFlow lite (for Microcontrollers)

What is TinyML?

- Fast growing field of Machine Learning
- Algorithms, hardware, and software
- **On-device** sensor data analytics
- Extreme **low power** consumption
- **Always on** ML use-cases
- **Battery operated** devices

What are the goals of tinyML?

- We want to perform inference on an embedded/iot device
- We want to be able to perform computation completely on-device, for efficiency, privacy and latency reasons.
- We want it to solve simple tasks, with respect to big ML pipelines/algorithms
- We want TinyML algorithms to have a low power consumption, in order to be able to function continously for days, weeks or even a year without human aid and without changing batteries
- Very often, we want it to be «embeddable» in larger cascade architectures.

What are the application of tinyML now…

Wake-word detection

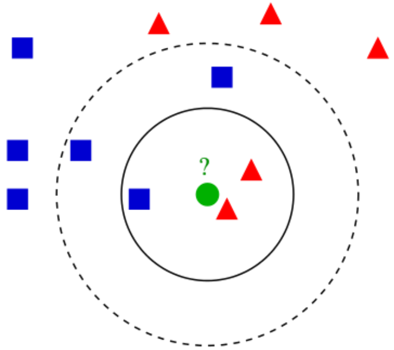
Presence detection

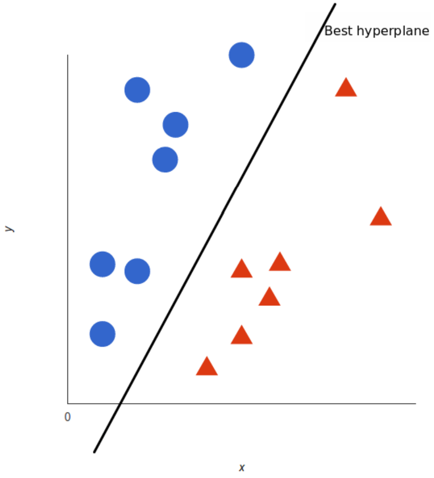
Anomaly detection

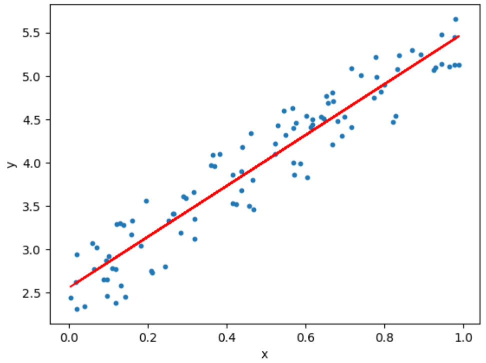
Health parameter monitoring

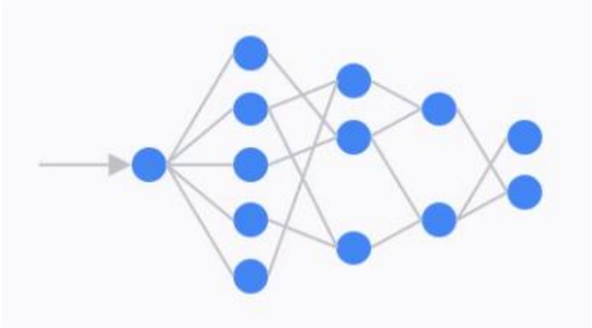
… and what will they be in the next future

TinyML is not only Neural Networks, nor only TFmicro









TensorFlow Lite (Micro)

CMSIS CMSIS-NN

CMSIS NN Software Library

STM32 Cube.AI

Endpoints have sensors, a lot of sensors

Motion sensors
Gyroscope, Radar, Accelerometer

Acustic sensors
Ultrasonic, Microphones, Vibrometers …

Environmental sensors
Temperature, Humidity, Pressure, IR …

Touchscreen sensors
Capacitive, IR

Image sensors
Thermal, Image

Biometric sensors
Fingerprint, Heart rate …

Force sensors
Pressure, Strain

Rotation sensors
Encoders

Endpoints have sensors, a lot of sensors

Motion sensors
Gyroscope, Radar, **Accelerometer**

Acustic sensors
Ultrasonic, **Microphones**, Vibrometers …

Environmental sensors
Temperature, Humidity, Pressure, IR …

Touchscreen sensors
Capacitive, IR

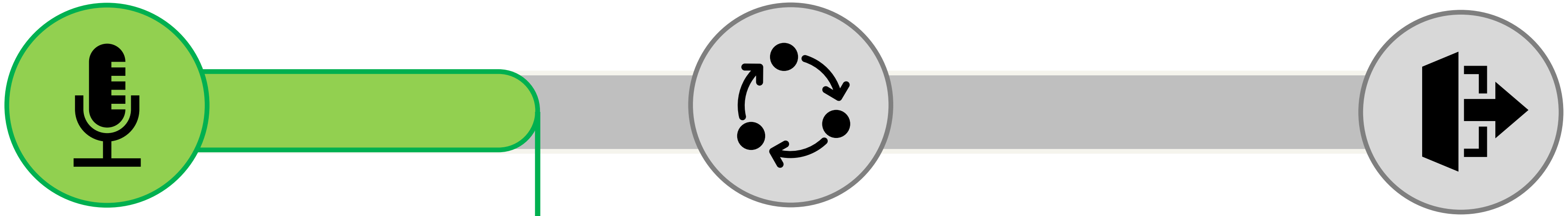
Image sensors
Thermal, **Image**

Biometric sensors
Fingerprint, Heart rate …

Force sensors
Pressure, Strain

Rotation sensors
Encoders

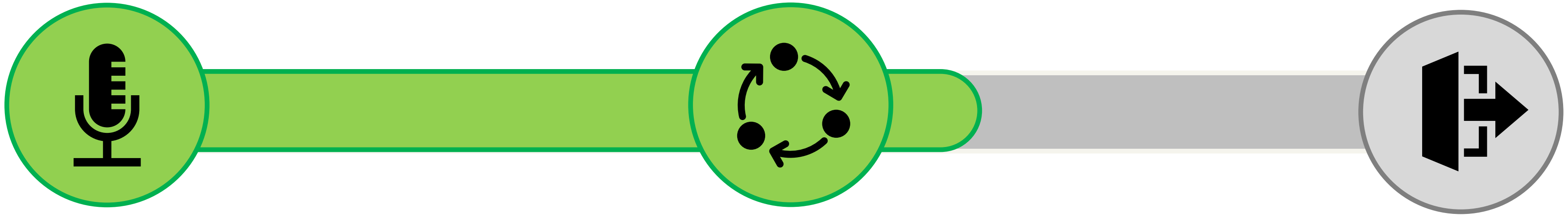
A complete ML pipeline: wake-word detection example



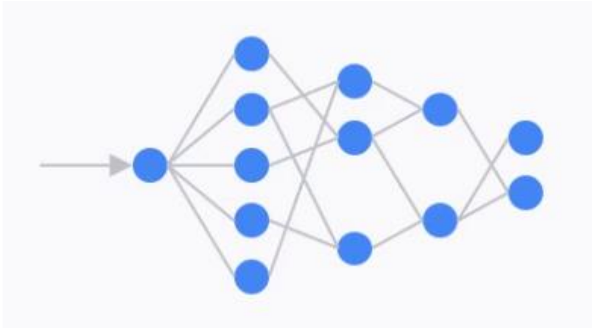
Step 1:
audio input
From microphone (sensor)

Input complete
Translation to mp3, waw…

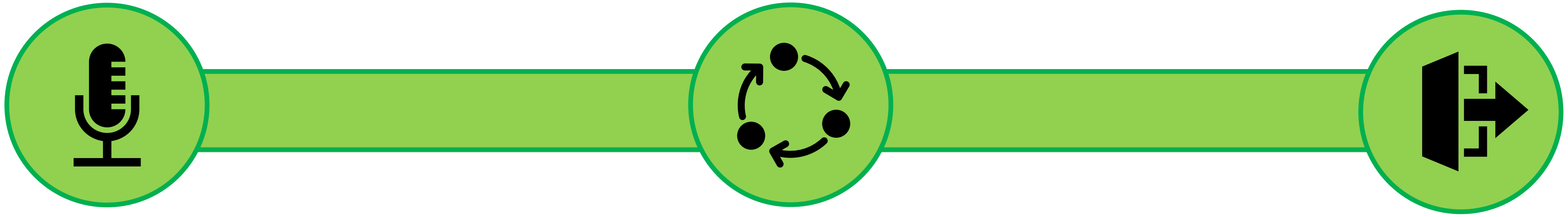
A complete ML pipeline: wake-word detection example



Step 2:
Process input
Translation, than
execute command



A complete ML pipeline: wake-word detection example

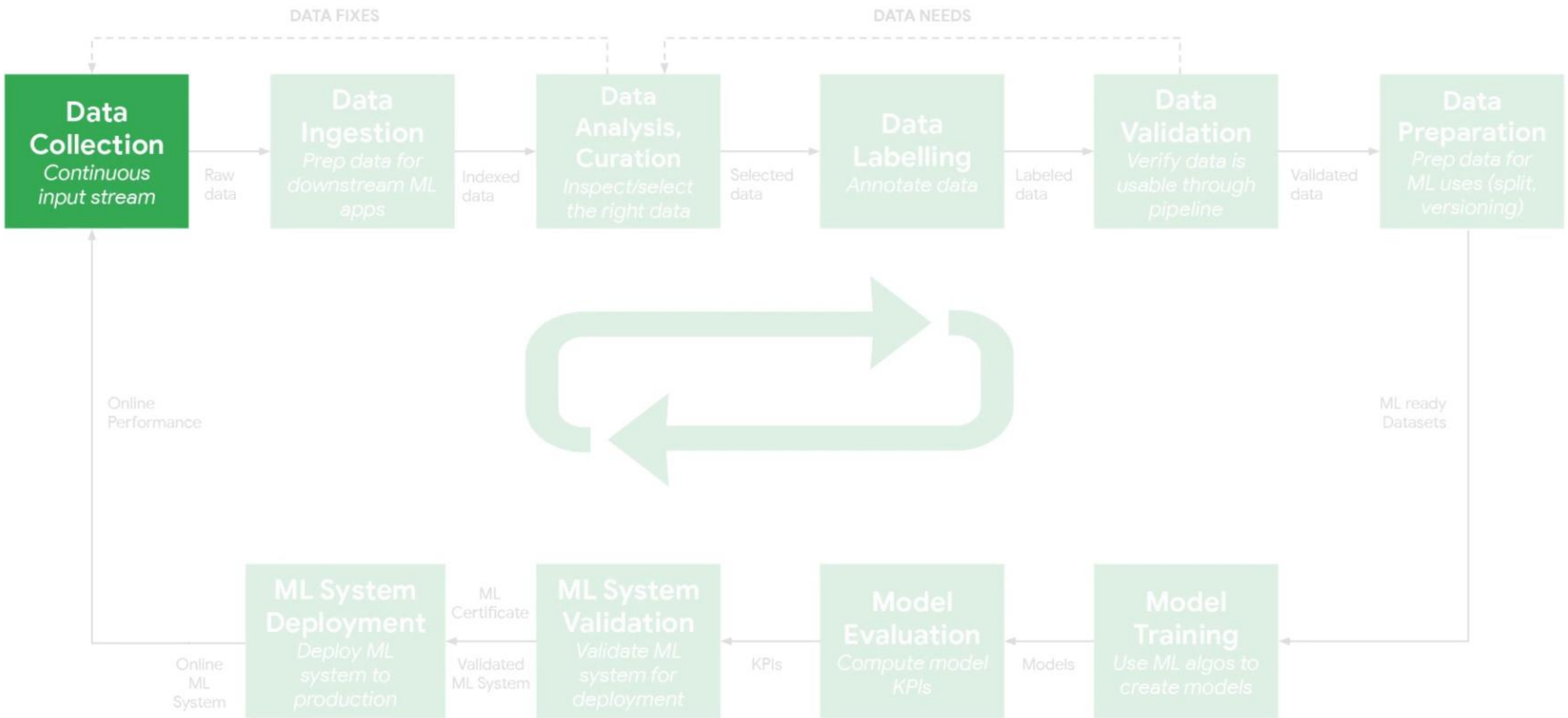


Step 3:
Generate output
Play response trough embedded speakers

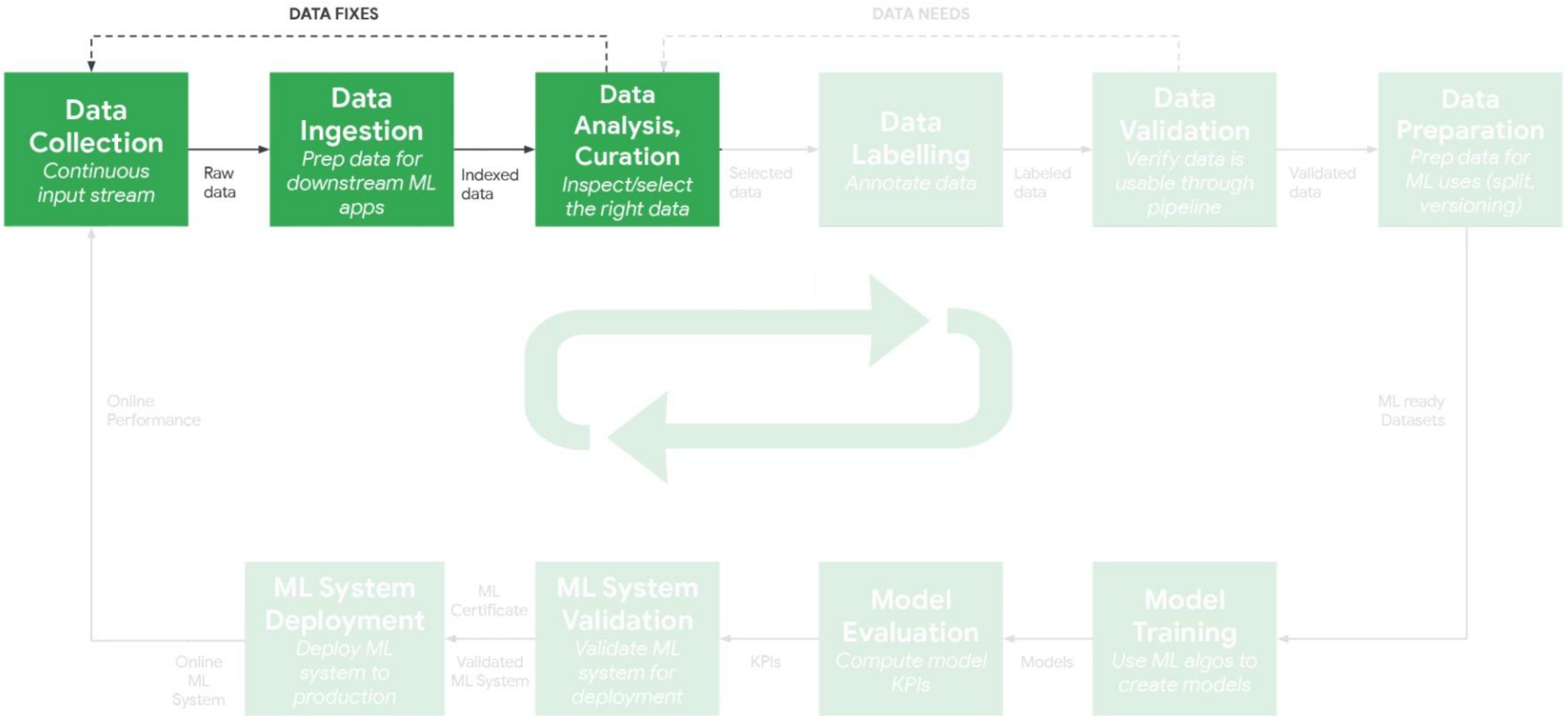
Questions

- How do we capture the data to feed into the network?
- How do you design the neural network to take in the speech signal?
- What dataset does the network need to be trained on?
- How do we pre-process the data for neural network inference?
- How do you post-process the neural network output?
- How do you deploy this on the microcontroller?
- How do we ensure that the neural network is resilient?
- How do we get the neural network to train faster?

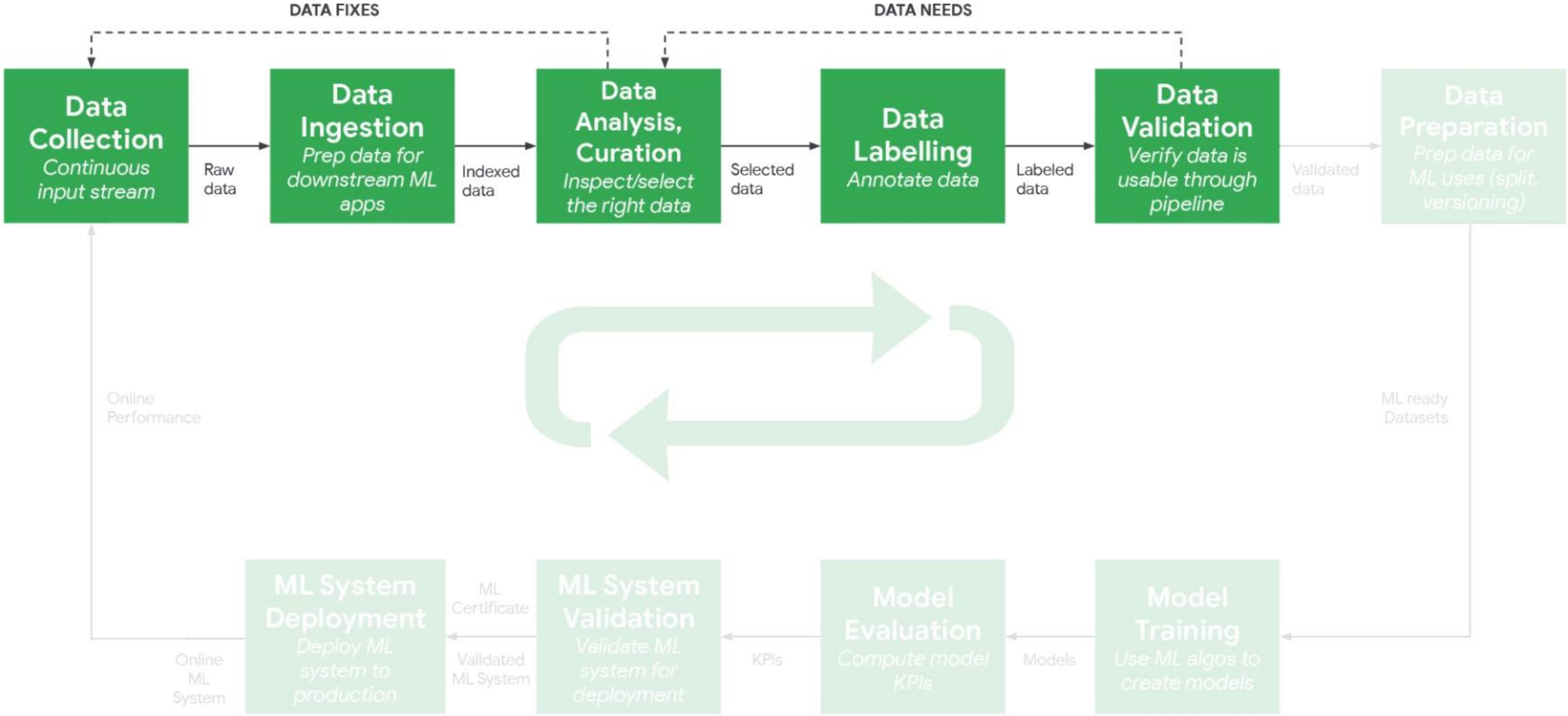
ML real life cycle



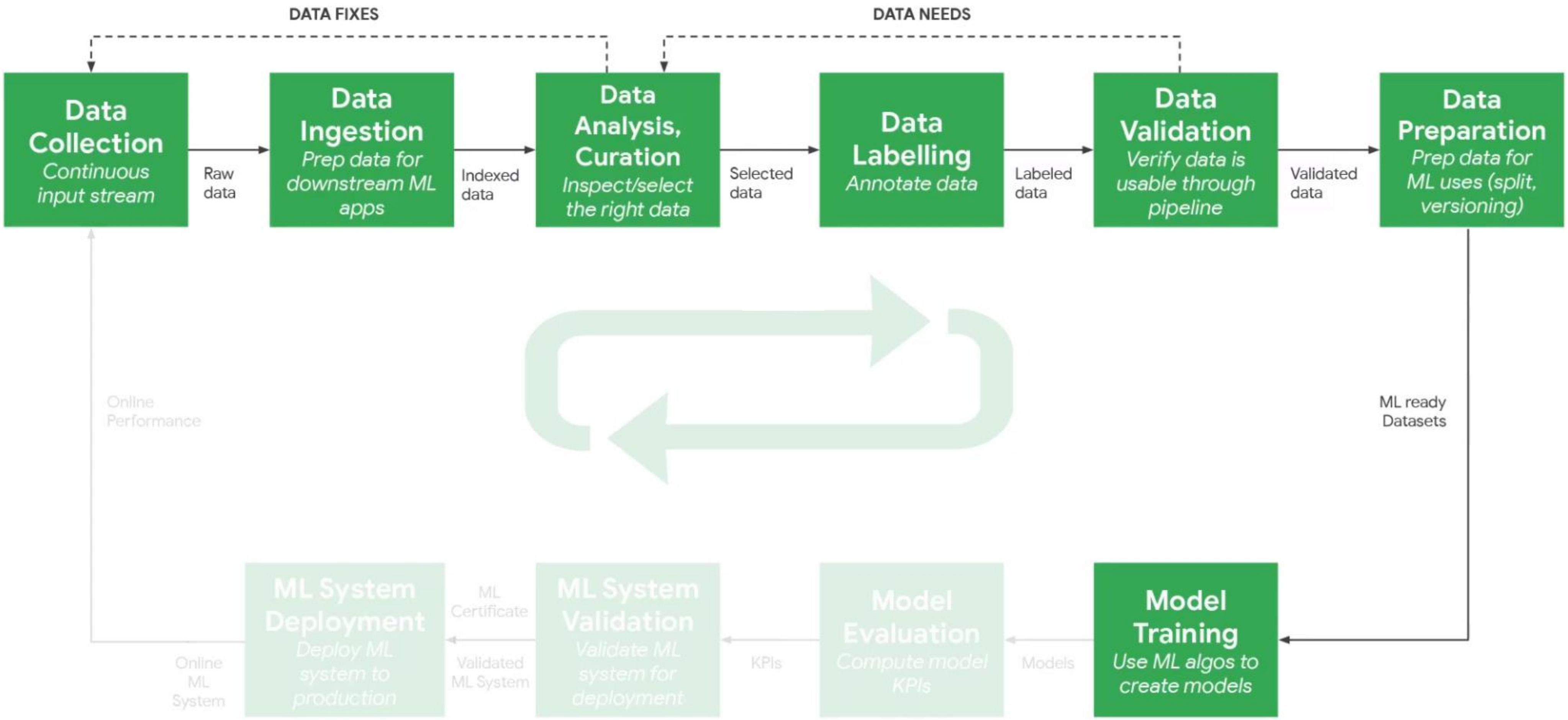
ML real life cycle



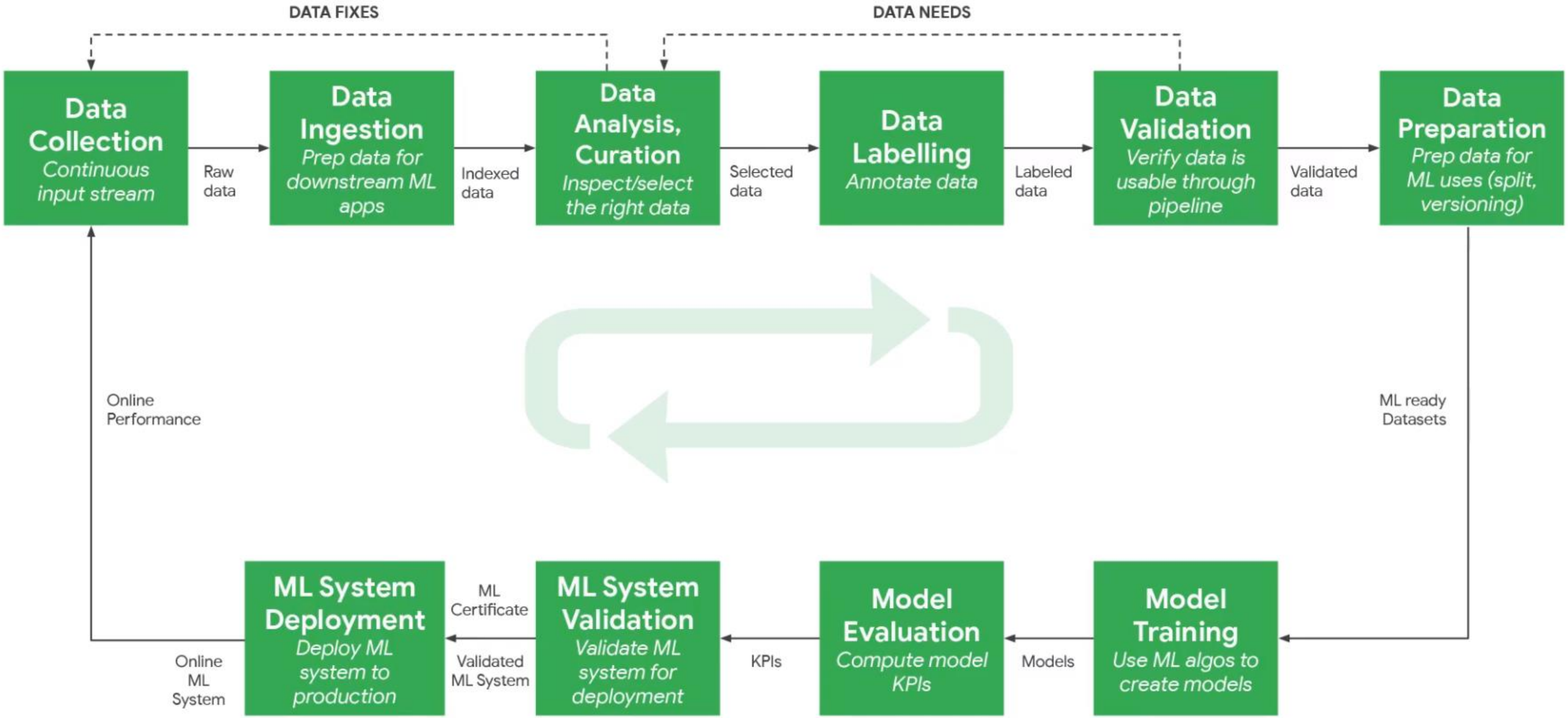
ML real life cycle



ML real life cycle



ML real life cycle



(Tiny) Machine Learning Workflow

AI Infrastructure

- Data Engineering
- Model Engineering
- Model Deployment
- Product Analytics

Collect Data → Pre-process data → Design Model → Train a Model → Evaluate optimize → Convert Model → Deploy Model → Make inference

Data Engineering

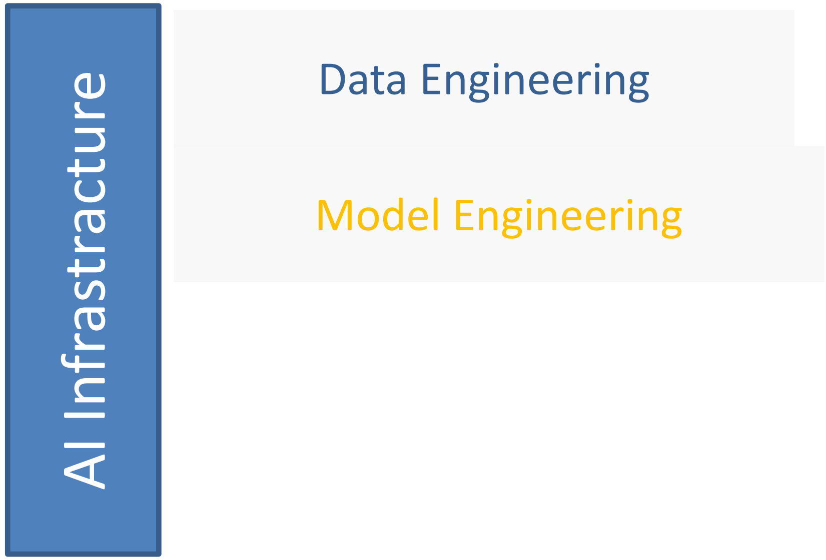
- Defining data **requirements**
- **Collecting** data
- **Labelling** the data
- Inspect and **clean** the data
- Prepare data for **training**
- **Augment** the data
- **Add more data**

AI Infrastructure

Data Engineering

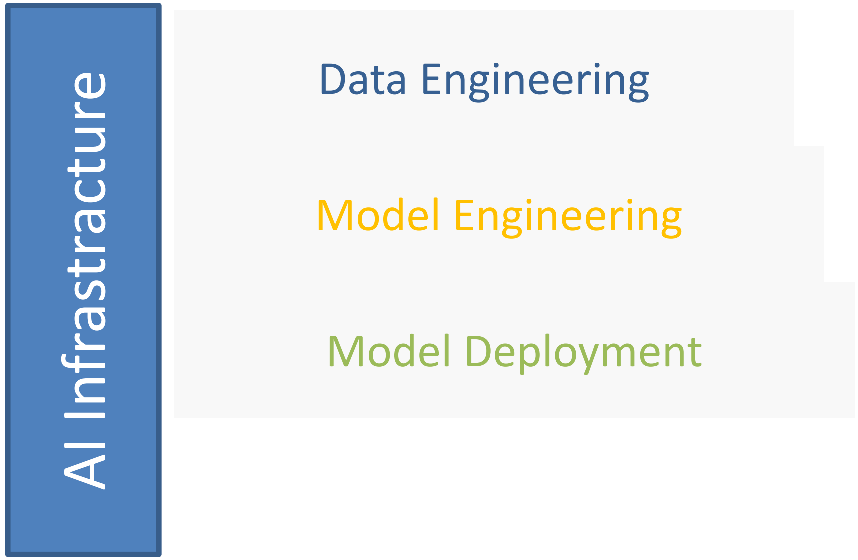
Model Engineering

- **Training** ML models
- Improving training **speed**
- Setting target **metrics**
- Evaluating against metrics
- **Optimizing** model training
- Keeping up with **SOTA**



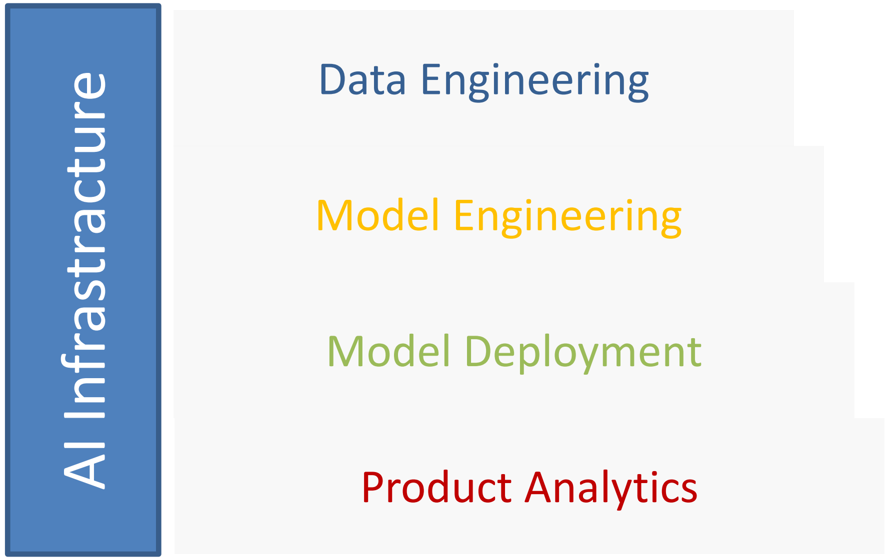
Model Deployment

- Model **conversion**
- Performance **optimization**
- **Energy-aware** optimizations
- **Security and privacy**
- **Inference** serving APIs
- **On-device** fine-tuning

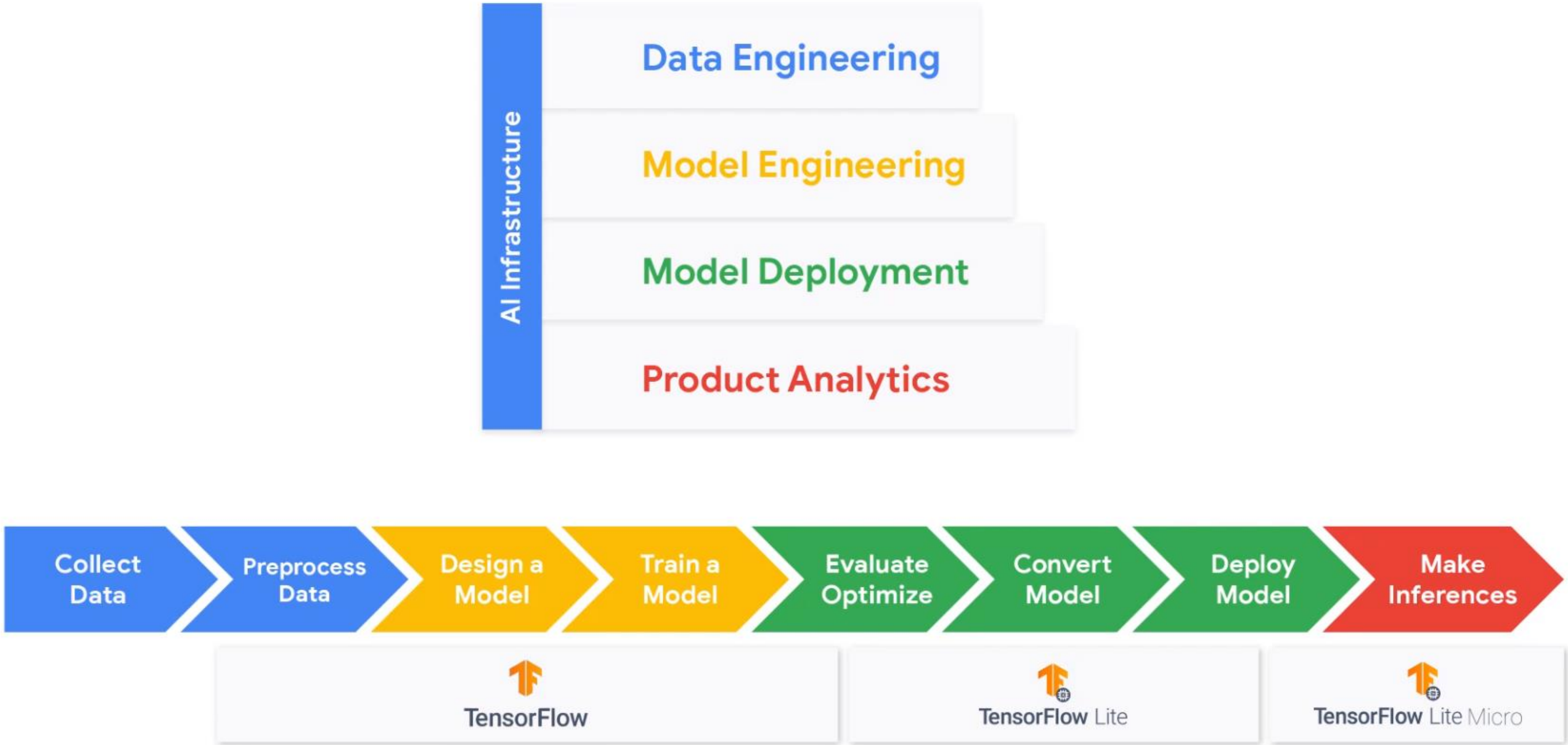


Product Analysis

- **Dashboards**
- **Field** data evaluation
- **Value-added** for business
- Opportunities for **advancement** and improvements



TinyML workflow



Differences between targets and operative frameworks

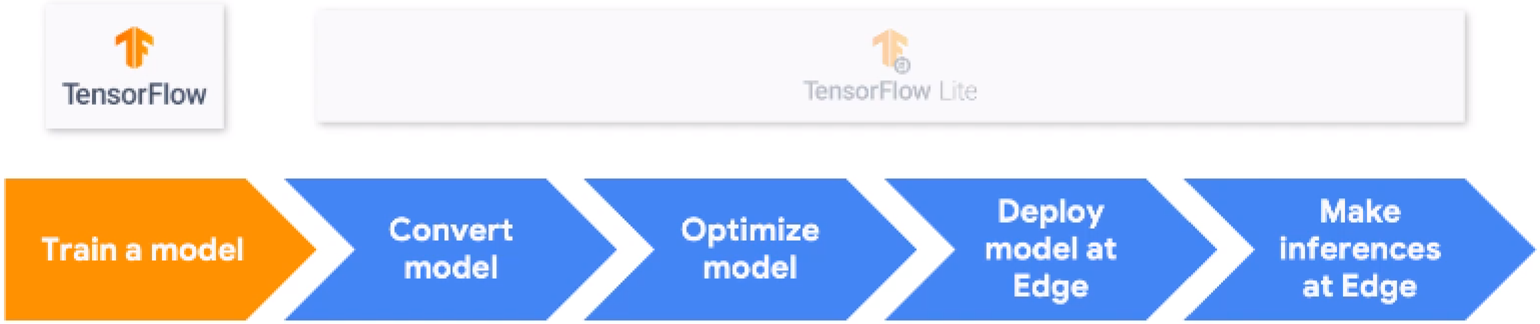
	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Training	Yes	No	No
Inference	Yes *(but inefficient on edge)*	Yes *(and efficient)*	Yes *(and even more efficient)*
How Many Ops	~1400	~130	~50
Native Quantization Tooling + Support	No	Yes	Yes

	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Needs an OS	Yes	Yes	No
Memory Mapping of Models	No	Yes	Yes
Delegation to accelerators	Yes	Yes	No

	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Base Binary Size	3MB+	100KB	~10 KB
Base Memory Footprint	~5MB	300KB	20KB
Optimized Architectures	X86, TPUs, GPUs	Arm Cortex A, x86	Arm Cortex M, DSPs, MCUs

Edge Devices

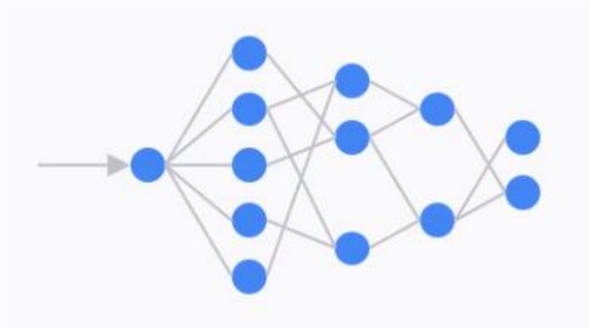
Let’s start with EDGE



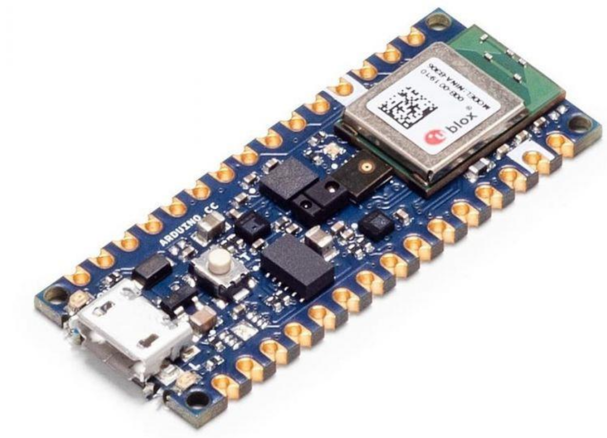
A comparison

- MobileNet (2015)
  - MobileNetv1
    - 70.6% top-1 accuracy
    - 16.9MB in size

- Arduino Nano BLE 33 sense
  - Has only 256KB of RAM!
  - 1 MB of flash



?????



Using the TFLITE converter

- Export saved model:

```
export_dir = 'saved_model/1'
tf.saved_model.save(model, export_dir)
```

- Use the TFLite converter:

```
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()
```

- Save the TFLite model:

```
import pathlib
tflite_model_file = pathlib.Path('model.tflite')
tflite_model_file.write_bytes(tflite_model)
```

- Create TFLite interpreter:

```
interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
```

- Test inference:

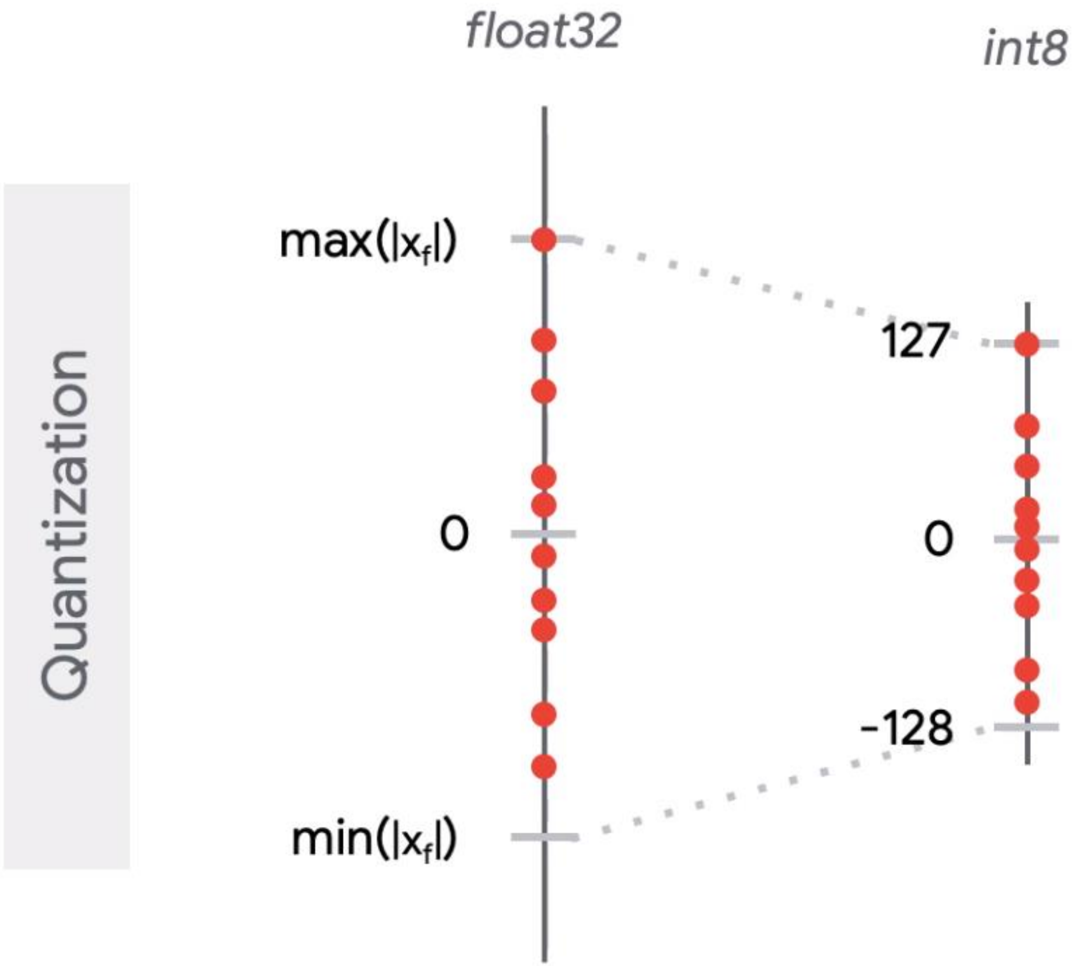
```
# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
to_predict = # Input data in the same shape as what the model expects
interpreter.set_tensor(input_details[0]['index'], to_predict)

tflite_results = interpreter.get_tensor(output_details[0]['index'])
```

Using the TFLite converter

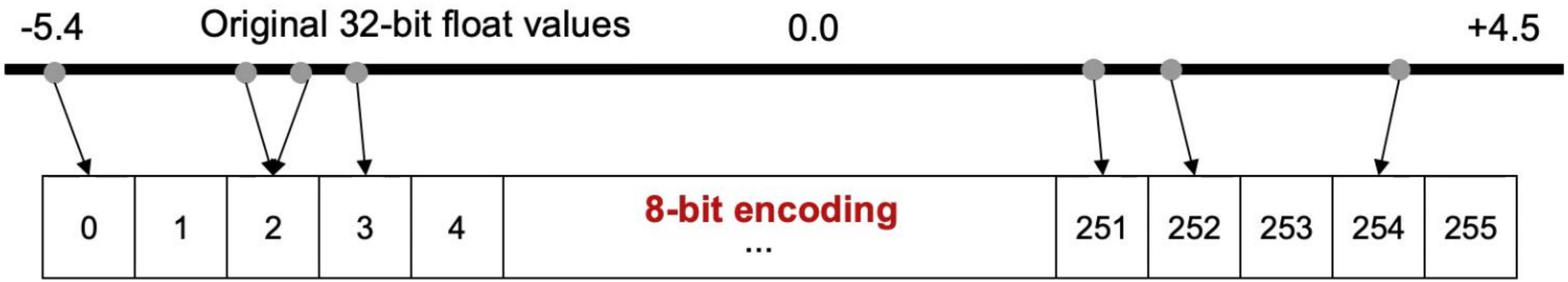
COLAB:

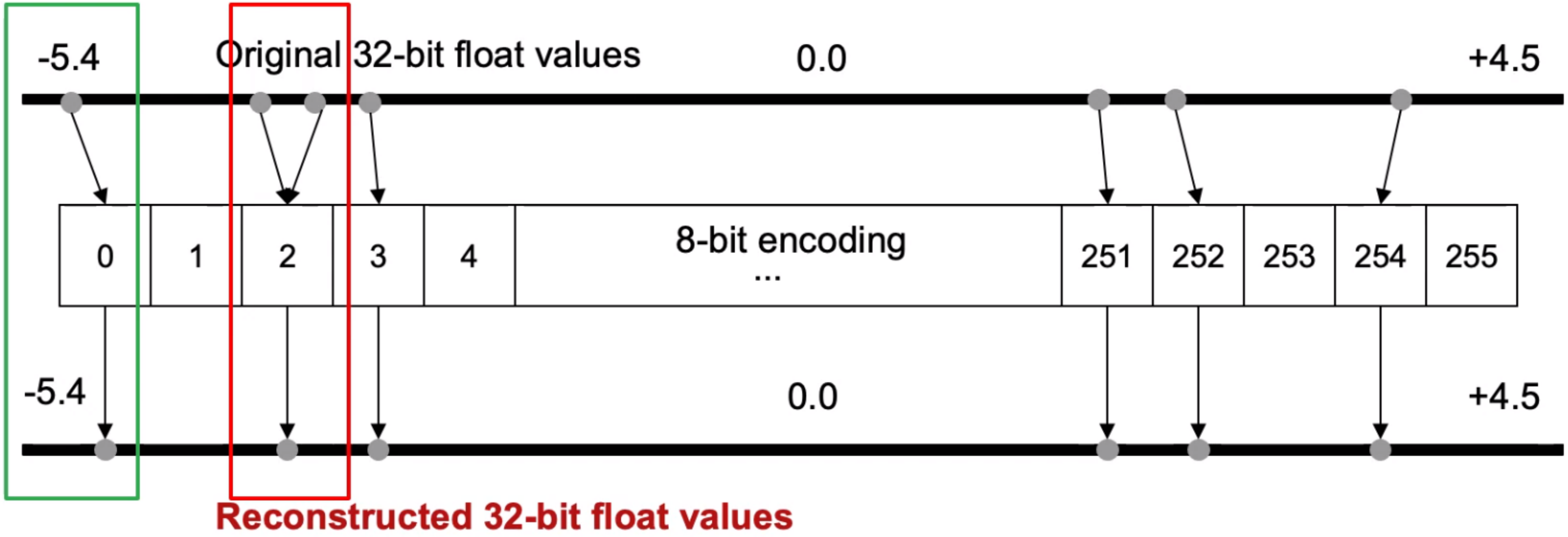
https://colab.research.google.com/github/tinyMLx/colabs/blob/master/3-3-7-RunningTFLiteModels.ipynb

Quantization



Quantization

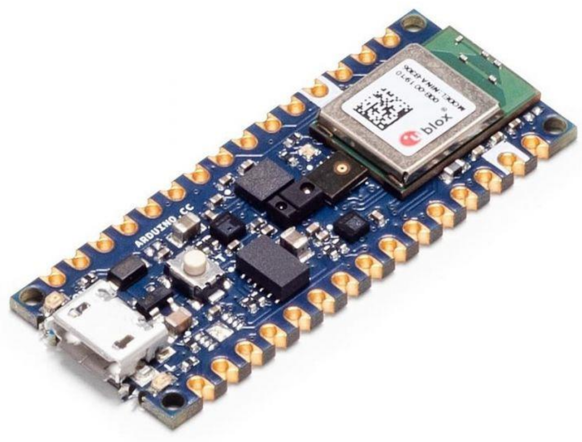


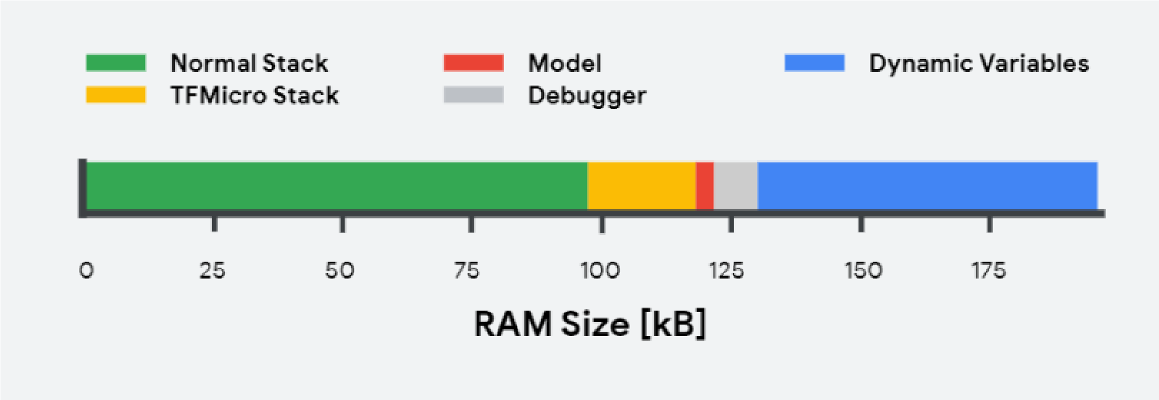

-5.4
Original 32-bit float values
0.0
+4.5
0
1
2
3
4
8-bit encoding
...
251
252
253
254
255
-5.4
0.0
+4.5
Reconstructed 32-bit float values

Why quantizing is (almost) always a good idea

- **Storage and Memory size**: when quantizing we have an immediate ≈4x reduction on the storage Memory, and a 2/4x reduction in peak RAM usage depending on the type of quantization you choose

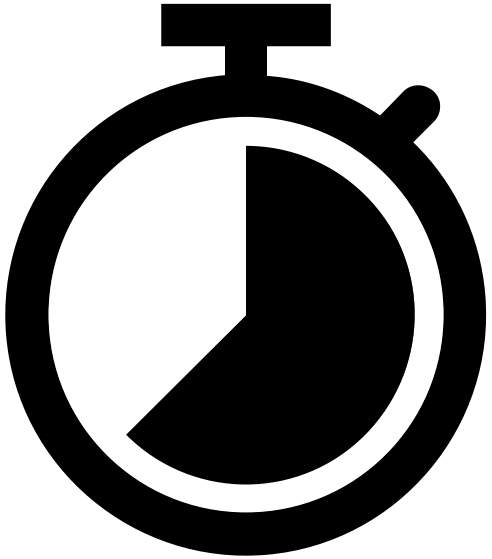
- Arduino Nano BLE 33 sense has 1 MB of flash memory





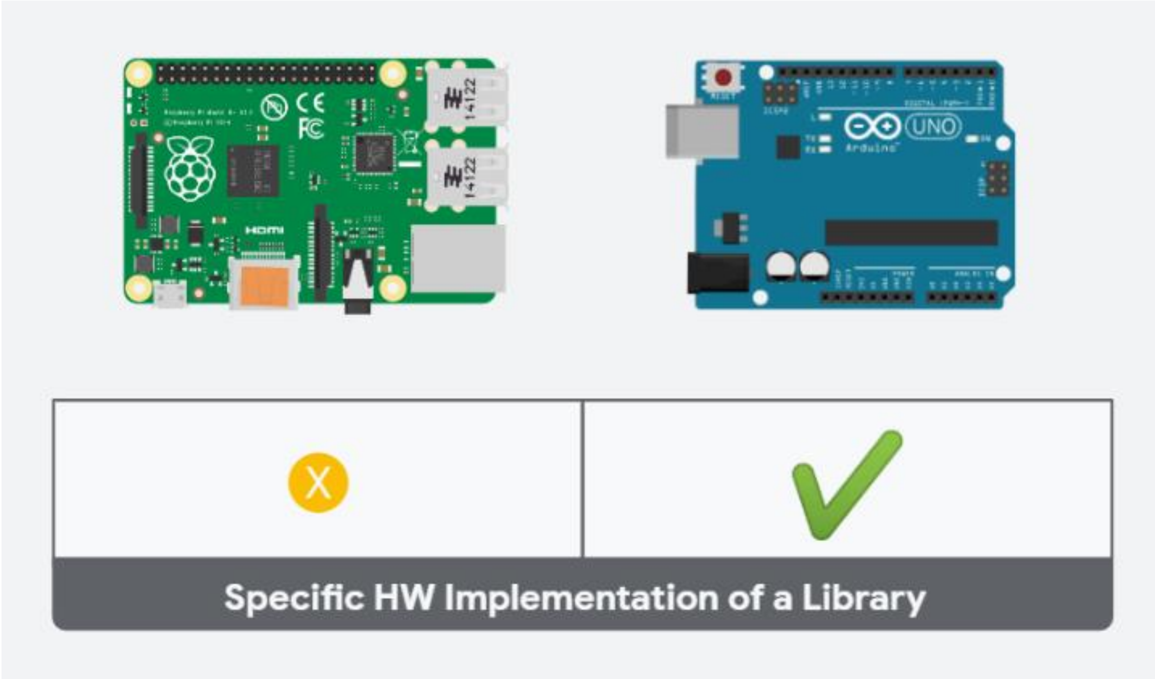
Why quantizing is (almost) always a good idea

- **Inference Latency**: by performing integer arithmetics instead of floating point arithmetics all the computations on the device are much faster. The expected reduction in time is in the order of 2/4x
- **Energy saving**: for the same reasons performing 8bit operations is much less energy-hungry than 32bit floating point arithmetics.



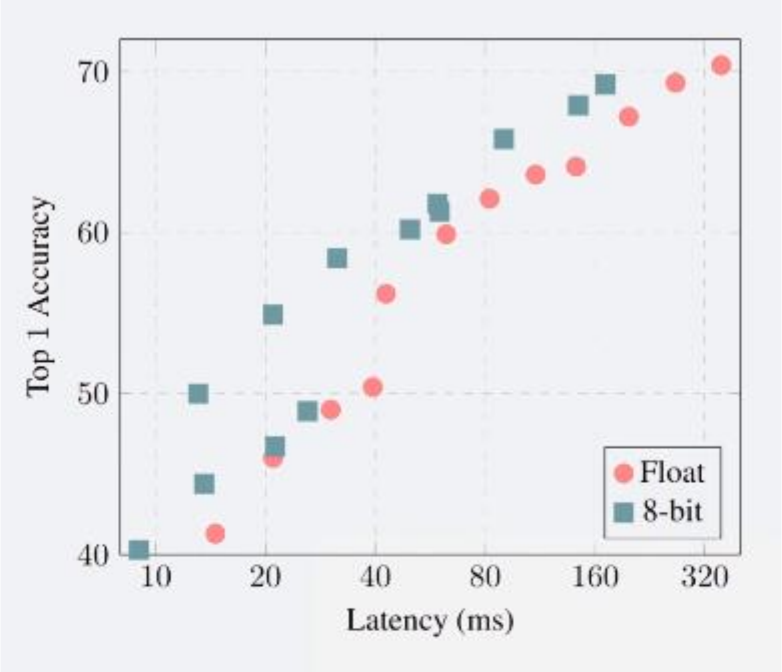
Why quantizing is (almost) always a good idea

- **Portability**: not every device can perform floating point 32 bits arithmetics. Everyone instead can perform 8bit integer arithmetics.



Accuracy-latency(/memory) trade-off

Quantization works well but performance can suffer of **accuracy-loss** during inference.



Enabling conversions optimization

Default optimization:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]

tflite_model = converter.convert()
tflite_model_file = 'converted_model.tflite'

with open(tflite_model_file, "wb") as f:
    f.write(tflite_model)
```

Quantization with representative dataset optimization:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]

def representative_data_gen():
    for input_value, _ in test_batches.take(100):
        yield [input_value]

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

tflite_model = converter.convert()
tflite_model_file = 'converted_model.tflite'

with open(tflite_model_file, "wb") as f:
    f.write(tflite_model)
```

Full integer optimization

```
def representative_data_gen():
  for input_value in tf.data.Dataset.from_tensor_slices(train_images).batch(1).take(100):
    yield [input_value]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set the input and output tensors to uint8 (APIs added in r2.3)
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8

tflite_model_quant = converter.convert()
```

In case input and output type == uint8, remember to quantize the input before feeding it to the network! (and de-quantize the output, in case you need it)

```
# Check if the input type is quantized, then rescale input data to uint8
if input_details['dtype'] == np.uint8:
  input_scale, input_zero_point = input_details["quantization"]
  test_image = test_image / input_scale + input_zero_point
```

Optimizing the network

COLAB:

https://colab.research.google.com/github/tinyMLx/colabs/blob/master/3-3-10-TFLiteOptimizations.ipynb#scrollTo=0RTZmndkcZFP

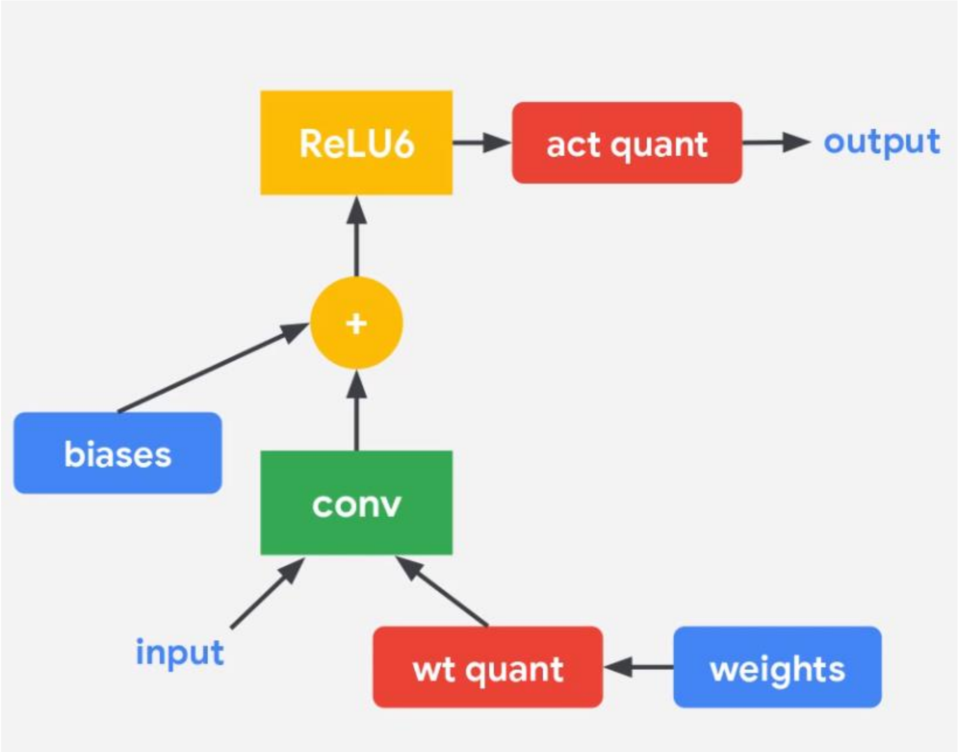
Quantization-aware training

Quantization aware training emulates inference-time quantization during training, creating a model that downstream tools will use to produce actually quantized models. The quantized models use lower-precision (e.g. 8-bit instead of 32-bit float), leading to benefits during deployment.

•Mimic the inference path during the training phase.

•Expose the training pipeline to the errors introduced by quantization

•Allow the training phase to recover the error «naturally»



Results comparison

	Floating-point Baseline	Post-training Quantization (PTQ)	Quantization-Aware Training (QAT)
MobileNet v1 1.0 224	71.03%	69.57%	71.06%
MobileNet v2 1.0 224	70.77%	70.20%	70.01%
Resnet v1 50	76.30%	75.95%	76.10%

How to quantization aware-training

```
import tensorflow_model_optimization as tfmot

quantize_model = tfmot.quantization.keras.quantize_model

# q_aware stands for for quantization aware.
q_aware_model = quantize_model(model)

# `quantize_model` requires a recompile.
q_aware_model.compile(optimizer='adam',
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=['accuracy'])
```

Quantization aware training

COLAB:

https://colab.research.google.com/github/tinyMLx/colabs/blob/master/3-3-12-QAT.ipynb#scrollTo=w7fztWsAOHTz

Hands on: rock paper scissor

https://colab.research.google.com/drive/1vAXuU9bDbD90W6fnpx1crvbQrzpnOcaU?usp=sharing

Appendix

Credits and reference

- "TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers", Daniel Situnayake, Pete Warden, O'Reilly Media, Inc.
- Online course:
  - https://www.edx.org/professional-certificate/harvardx-tiny-machine-learning
- A lot more material on TinyML:
  - http://tinyml.seas.harvard.edu/