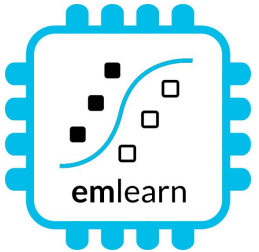
Sensor data processing on microcontrollers

with MicroPython and emlearn

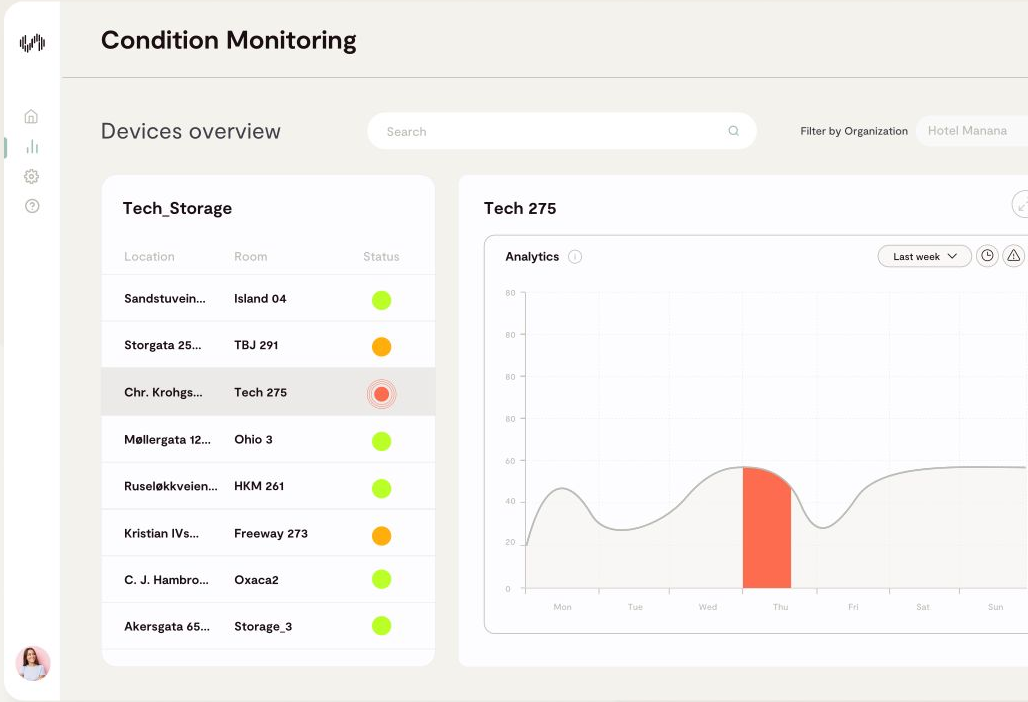
https://github.com/emlearn/emlearn-micropython

Jon Nordby jononor@gmail.com
PyCon ZA 2024



We utilise sound and vibration analysis to detect and warn you of upcoming errors in your technical infrastructure before they happen.

soundsensing



Trusted by Nordic market leaders

anthon eiendom · RAGDE EIENDOM · NORDR · SELVAAG · AF EIENDOM · OBOS · THON EIENDOM · EIENDOMSSPAR · Oslo · WÄRTSILÄ · ASPELIN RAMM · MALLING & CO · STATSBYGG

Goal

Purpose of this presentation

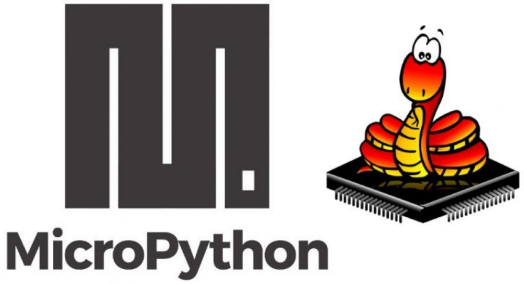
You, as a Python developer,

can build an IoT sensor

on a microcontroller

using MicroPython

Background

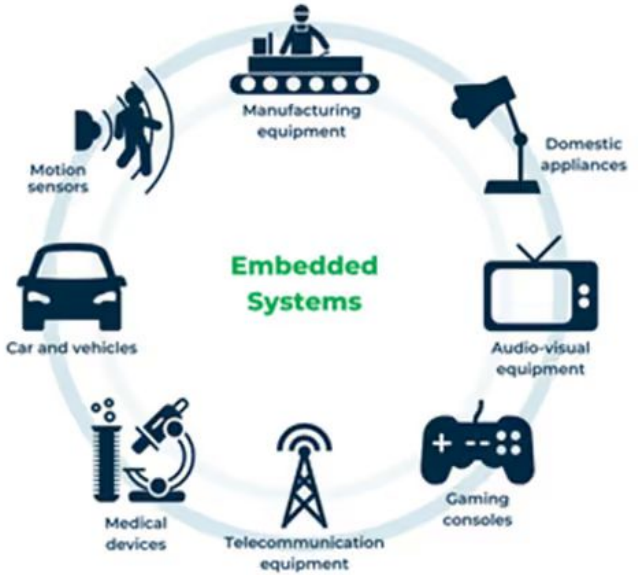


Microcontrollers &
MicroPython

Microcontroller - tiny programmable chip

Compute power: 1 / 1000x of a smartphone

- **RAM: 0.10 - 1 000 kB**
- **Program space: 1.0 - 10 000 kB**
- **Compute 10 - 1 000 DMIPS**
- **Price: 0.10 - 10 USD**
- **Energy: 1 - 1 000 milliWatt**

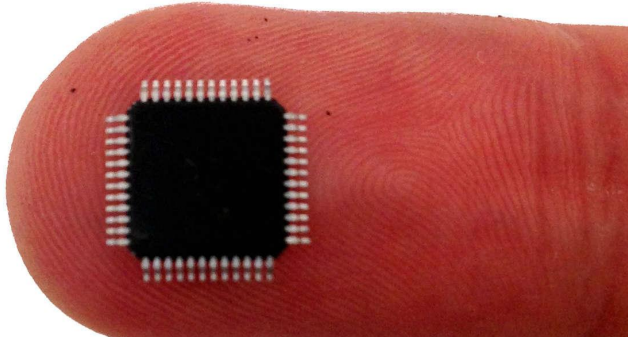


Over 20 *billions* shipped *per year*!

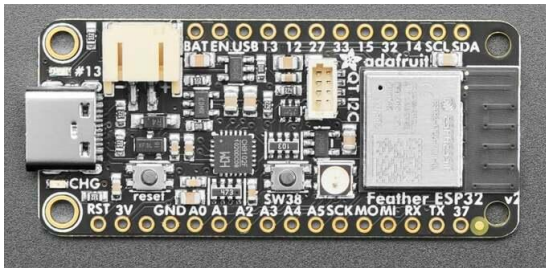
Increasingly accessible for hobbyists

2010: Arduino Uno
2014: MicroPython
2019: MicroPython 1.10 - ESP32 PSRAM

Efficiency is key !
Memory, compute, power

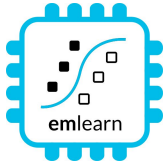


Hardware recommendation - start with a **ESP32** device



Complete device with sensors etc.:	20 - 50 USD
Development boards:	5 - 20 USD
Chips / modules	1 - 5 USD

MicroPython introduction



Implements a subset of Python 3.x for devices with 16 kB+ RAM
Supports 8+ microcontroller families

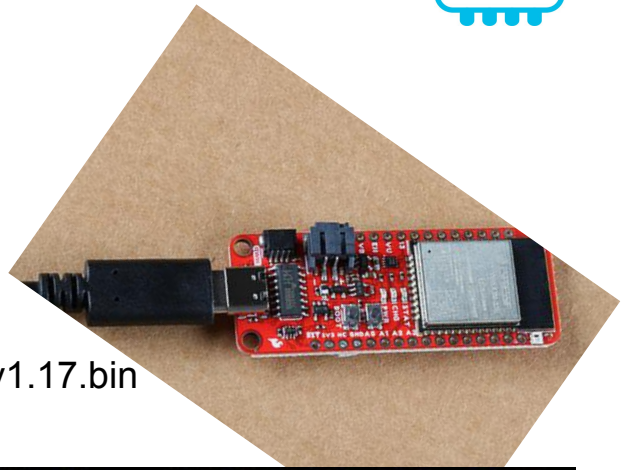
Tries to be as compatible with CPython as possible, within constraints.
Python 3.6 mostly implemented, partial after that.

No CFFI or C module compatibility!
TLDR: Small scripts will mostly work with minor mods. But not PyData stack

Package manager "**mip install**"
Has support for loading C modules at runtime!

More info: https://micropython.org

Installing MicroPython



Download prebuilt firmware

https://micropython.org/download/?port=esp32

Flash firmware to device

pip install esptool

esptool.py --chip esp32 --port … erase_flash
esptool.py --chip esp32 --port … write_flash -z 0 micropython-v1.17.bin

Connect to device

pip install mpremote

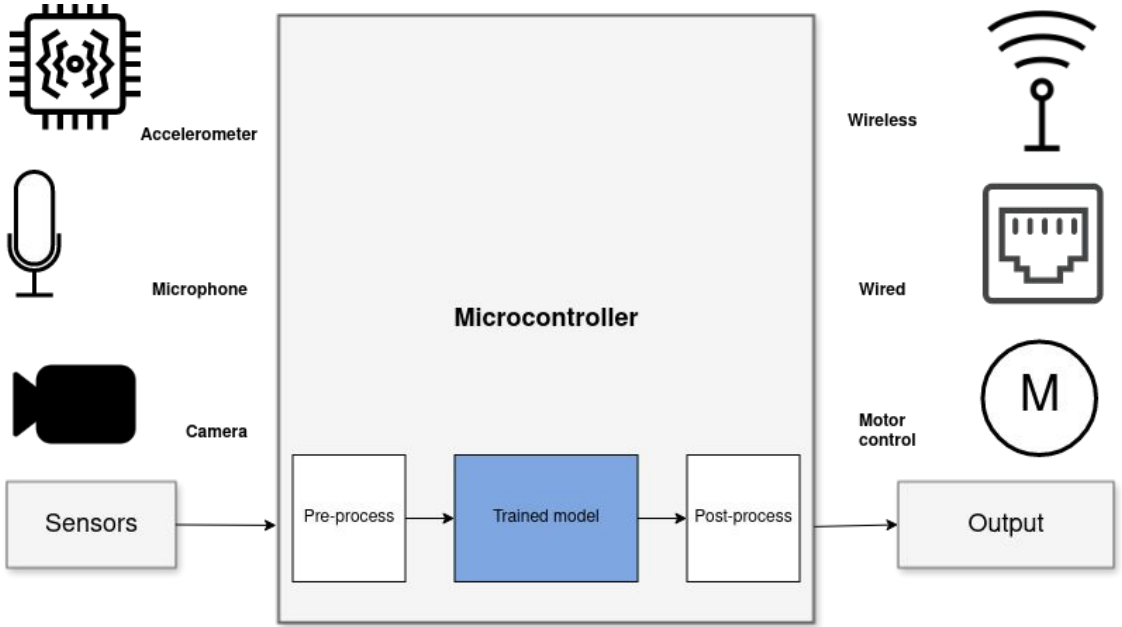
mpremote repl

```
MicroPython v1.8.3-24-g095e43a on 2016-08-16; ESP module
Type "help()" for more information.
>>> print('Hello world!')
Hello world!
>>>
```

IDE (optional): Thonny, VS Code, et.c.

Sensor node systems

Including on-sensor data processing
with Digital Signal Processing (DSP) and Machine Learning (ML)



“Internet of Things”

“Wearable devices”

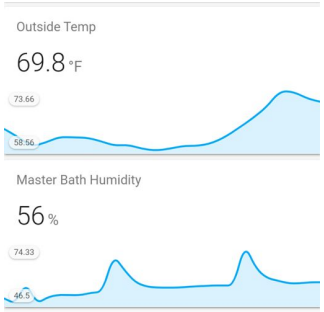
Robotics,
Industrial automation

Sensor data processing

What can be done on an ESP32 microcontroller with MicroPython and how to make it work

Temperature logger

Temperature
0.1 Hz



Activity tracker

Accelerometer
100 Hz

Noise monitor

Microphone
16000 Hz

Cat Detector

Camera
27000 bytes/s

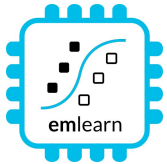

```
if is_MyCat(img):
    open_door()
```

Make it Work,
Make it Right,
Make it Fast

– Ken Beck

→

Write simple automated tests,
Code in straightforward Python,
Measure performance with benchmarks



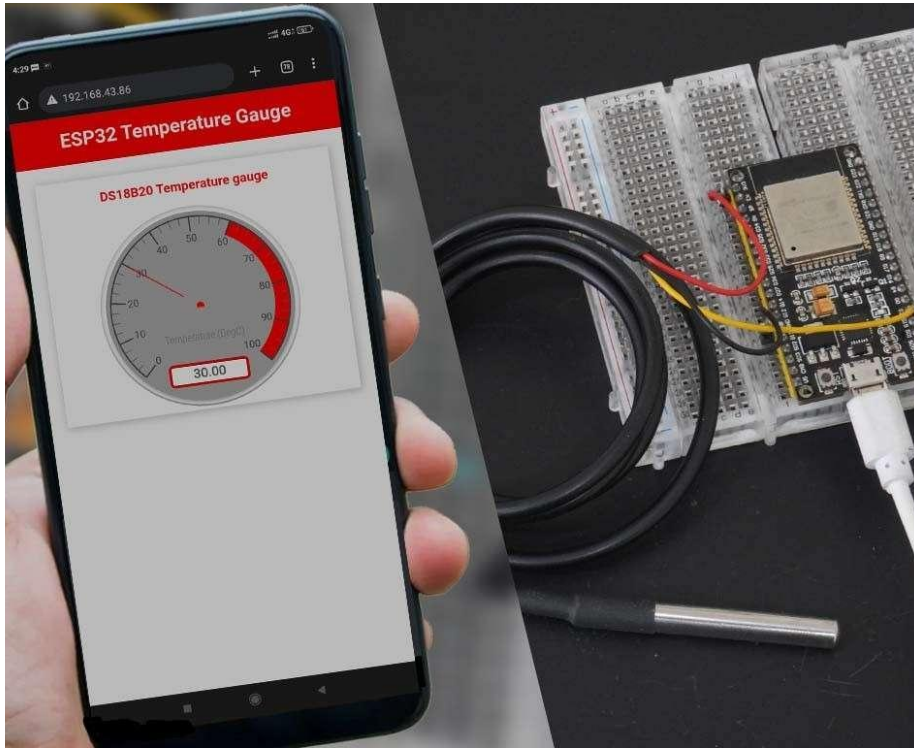
Optimize *if needed*

- Start with simple techniques
- Go more advanced *if needed*

time.time and **assert**
for benchmark and tests

```
import time

repeats = 100
expect = 18965.39
input = ....

start = time.ticks_us()
for r in range(repeats):
    out = rms_python(input)
    assert p == expect, (out, expect)
t = time.ticks_diff(time.ticks_us(), start) / repeats
print('python', t)

start = time.ticks_us()
for r in range(repeats):
    out = rms_micropython_native(input)
    assert out == expect, (out, expect)
t = time.ticks_diff(time.ticks_us(), start) / repeats
print('native', t)
```

Temperature sensor



Temperature is a **slow changing phenomena.**
Want: **1 value for longish time interval**
(ex: 5 min)

! Sensor can be noisy. Filtering advised

Will show **analog sensor** example.
Same approach can be used for a **digital sensor**.
And for **other slow-changing** phenomena.

Temperature sensor

Solution: Read the sensor in a loop

Filter noise using use a median filter.

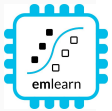
= Trivially doable in straightforward Python

Efficiency techniques (non-critical)

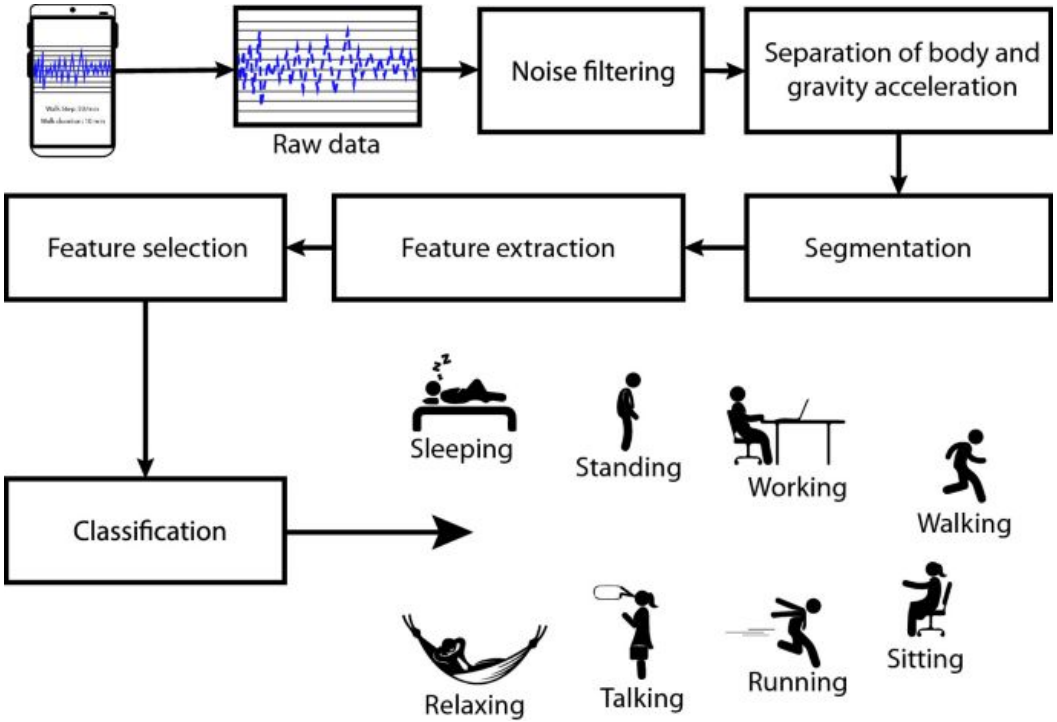
- Using *lightsleep()* to conserve power.
- Pre-allocate fixed **array.array**, instead of over mutating a **list** (append) - to reduce allocations

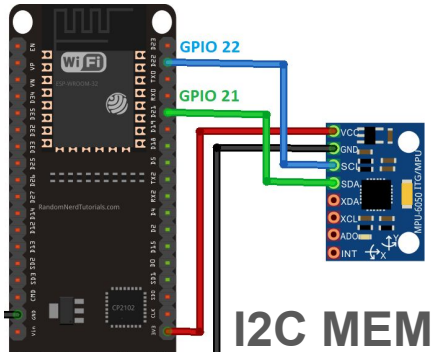
```
import machine

N_SAMPLES = 10
SAMPLE_INTERVAL = 1.0
SLEEP_INTERVAL = 60.0

analog_input = machine.ADC(machine.Pin(22))
samples = array.array('H') # raw values from ADC, as uint16
measurement_no = 0

while True:

    # collect data for measurement
    for i in range(N_SAMPLES):
        samples[i] = adc.read_u16()
        time.sleep(SAMPLE_INTERVAL)

    # aggregate samples, convert to temperature
    raw = median(samples)
    temperature = (raw * 0.323) - 50.0 # calculation depends on

    # Do something with the measurement
    send_bluetooth_le(measurement_no, temperature)

    # sleep until next time to collect new measurement
    machine.lightsleep(int(SLEEP_INTERVAL*1000))
    measurement_no += 1
```

Sending data as Bluetooth Low Energy advertisements, see
https://github.com/orgs/micropython/discussions/15926



Activity tracker - concept





I2C MEMS IMU accelerometer/gyro
(Example wiring)

Complete hardware unit
LilyGo
T-Watch S3

Activity Tracker - data acquisition

Want data chunks of 1 second.

25 Hz sample rate.
4 ms between samples
OK for basic Human Activity Detection
Might for some tasks want 100+ Hz.

Efficiency techniques (recommended)

- Utilizing FIFO buffer in sensor
- Using *lightsleep()* to conserve power.

```
THRESHOLD = 25
sensor = SomeIMU(i2c=...)
samples = []
while True:
    t = time.ticks_ms()
    samples.append(sensor.get_xyz())

    if len(samples) == THRESHOLD:
        process_samples(samples)
        samples = []

    # Try compensate for execution time data sampling time
    time_spent = time.ticks_diff(time.ticks_ms(), t)
    wait_time = max(1000/SAMPLERATE - time_spent, 0)
    time.sleep_ms(wait_time)
```

Polling individual samples
Must always use < 4 ms, otherwise will drop sample

Limits what process_samples() can do

! GC can easily take 10-100 ms

```
THRESHOLD = 25 # NOTE: max 75% of FIFO capacity
imu = SuperIMU2k(i2c=..., odr=SAMPLERATE)

while True:
    level = imu.get_fifo_level()
    if level >= THRESHOLD:
        samples = imu.read_fifo_data(THRESHOLD)

        process_samples(samples)

    machine.lightsleep(100)
```

Using sensor FIFO buffer
Can spend 100 ms+ on processing

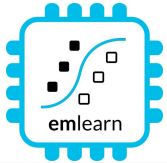
Can sleep 90%+ of the time

Perfectly spaced samples, always

Implementing an IMU/accelerometer/gyro driver? Use the FIFO!
https://github.com/orgs/micropython/discussions/15512

emlearn

Activity Tracker - Feature Extraction



Basic statistical summarizations are useful.
Can be sufficient for Human Activity Recognition

At 25 Hz, 3 channels, every 1 second.
Doable in pure Python
Estimated processing time 10-100 ms

Can be optimized using specialized code emitters.
More later…

```
l = sorted(list(v))
l2 = [x*x for x in l]
sm = sum(l)
sqs = sum(l2)
avg = sum(l) / len(l)

median = l[MEDIAN]
q25 = l[Q1]
q75 = l[Q3]
iqr = (l[Q3] - l[Q1])

energy = ((sqs / len(l2)) ** 0.5) # rms
std = ((sqs - avg * avg) ** 0.5)
```

Time-based features extraction
Are Microcontrollers Ready for Deep Learning-Based Human Activity Recognition?
Atis Elsts, and Ryan McConville
https://www.mdpi.com/2079-9292/10/21/2640

Activity Tracker

Classification using Random Forest

everywhereml / m2cgen:

Generate Python .py code module

emltrees from emlearn-micropython

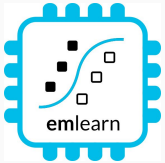
Dynamic native module .mpy

Implemented in C

Load model from .csv file

10 trees sufficient for OK performance on common HAR datasets (PAMAP2/UCI)

```
import emltrees
model = emltrees.new(10, 1000, 10)
with open('eml_digits.csv', 'r') as f:
    emltrees.load_model(model, f)

features = array.array('h', ...)
out = model.predict(features)
```



```
from everywhere_digits import RandomForestClassifier
model = RandomForestClassifier()
out = model.predict(x)
```

10 trees, max 100 leaf nodes, “digits” dataset

	Inference time	Program space
m2cgen	60.1 ms	179 kB
everywhereml	17.7 ms	154 kB
emlearn	1.3 ms	15 kB

10x faster

10x more space efficient

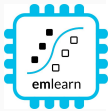
C modules

Written in C.
Defines a Python module with API. functions/classes et.c.

Can be implemented by users, libraries or be part of MicroPython core.

Can be portable or specific to one hardware/platform

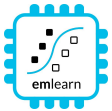
```c
// Include the header file to get access to the MicroPython API
#include "py/dynruntime.h"

// Helper C function to compute factorial
static mp_int_t factorial_helper(mp_int_t x) {
    if (x == 0) {
        return 1;
    }
    return x * factorial_helper(x - 1);
}

// DEFINE FUNCTION. Callable from Python
static mp_obj_t factorial(mp_obj_t x_obj) {
    mp_int_t x = mp_obj_get_int(x_obj);
    mp_int_t result = factorial_helper(x);
    return mp_obj_new_int(result);
}
static MP_DEFINE_CONST_FUN_OBJ_1(factorial_obj, factorial);


// MODULE ENTRY
mp_obj_t mpy_init(mp_obj_fun_bc_t *self, size_t n_args, size_t n_kw, mp_obj_t *args) {
    // Must be first, it sets up the globals dict and other things
    MP_DYNRUNTIME_INIT_ENTRY

    // Register function in the module's namespace
    mp_store_global(MP_QSTR_factorial, MP_OBJ_FROM_PTR(&factorial_obj));

    // This must be last, it restores the globals dict
    MP_DYNRUNTIME_INIT_EXIT
}
```



Native module (.mpy) VS External C module

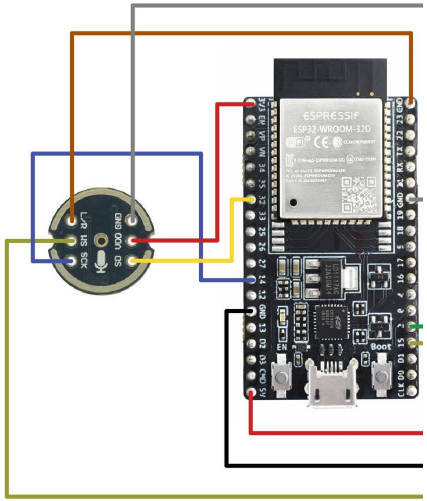
	Native module	**External C module**
Installable at runtime	Yes, as .mpy file	No. Must be included in firmware image
Requires SDK/toolchain	No (only to build)	Yes
Code executes from	RAM	FLASH
Limitations	No libc / libm linked * No static BSS *	None
Maturity	Low *	Excellent
Documentation	https://docs.micropython.org/en/latest/develop/natmod.html	https://docs.micropython.org/en/latest/develop/cmodules.html

* Improved greatly in upcoming MicroPython (1.24+).
Contributions by Volodymyr Shymanskyy, Alessandro Gatti, Damien George, and others



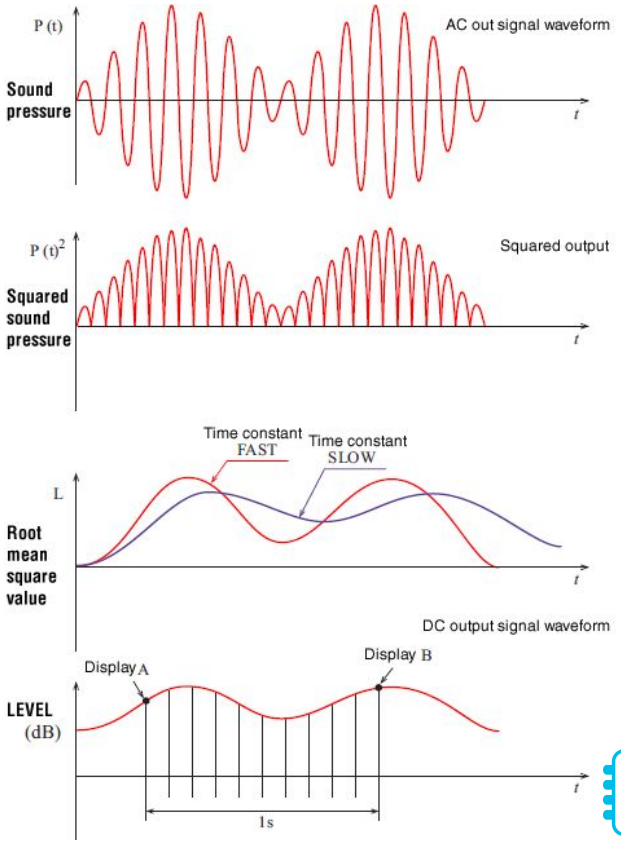
Sound sensor

I2S digital microphone
(Example wiring)



Complete hardware unit
LilyGo TTGO
T-Camera Mic v1.6

Soundlevel calculation
Processing steps



Sound sensor - Data Acquisition

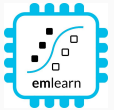
16 kHz samplerate - no problem to read
48 kHz would probably also read fine
- but 3x the load for processing.

Efficiency techniques (critical)

- Using dedicated I2S hardware for audio input
- memoryview() to allow bytes-based API to write into our int16 array, without needing conversion

Rendering text/widgets to screen
https://github.com/peterhinch/micropython-nano-gui

```
# I2S audio input
from machine import I2S
audio_in = I2S(0, sck=Pin(26), ws=Pin(32), sd=Pin(33),
    mode=I2S.RX, bits=16,format=I2S.MONO, rate=16000,
)

# allocate sample arrays
chunk_samples = int(AUDIO_SAMPLERATE * 0.125)
mic_samples = array.array('h', (0 for _ in range(chunk_samples))) # int16
# memoryview used to reduce heap allocation in while loop
mic_samples_mv = memoryview(mic_samples)
# global to share state between callback and main
soundlevel_db = 0.0

meter = SoundlevelMeter(buffer_size=chunk_samples, samplerate=16000)

def audio_ready_callback(arg):
    # compute soundlevel
    global soundlevel_db
    soundlevel_db = meter.process(mic_samples)
    # re-trigger audio callback
    _ = audio_in.readinto(mic_samples_mv)

def main():
    # Use Non-Blocking I/O with callback
    audio_in.irq(audio_ready_callback)
    # Trigger first audio readout
    audio_in.readinto(mic_samples_mv)

    while True:
        render_display(db=soundlevel_db)
        time.sleep_ms(200)

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



Sound sensor - RMS calculation

125ms @ 16kHz = 2000 samples
Processing deadline: 125 ms

Root Mean Square (energy calculation)

1. Plain Python **= too slow!**
2. Native emitter + trivial change
   = runs fine
3. Viper emitter + low-level rewrite
   for integer-based arithmetic
   = add 10x efficiency

```python
def rms_python(arr):
    acc = 0.0
    for i in range(len(arr)):
        v = float(arr[i])
        acc += (v * v)
    mean = acc / len(arr)
    out = math.sqrt(mean)
    return out
```

1. 113 ms 95% CPU
Plain Python
All floats

```python
@micropython.native
def rms_micropython_native(arr):
    acc = 0
    for i in range(len(arr)):
        v = arr[i]
        acc += (v * v)
    mean = acc / len(arr)
    out = math.sqrt(mean)
    return out
```

2. 35 ms 28% CPU
Native emitter
Integer squaring Float accumulator

```python
@micropython.viper
def rms_micropython_viper(arr) -> object:
    buf = ptr16(arr) # XXX: input MUST BE h/uint16 array
    l = int(len(arr))
    cumulated_average : int = 0
    cumulated_remainder : int = 0
    addendum : int = 0
    n_values : int = 0
    for i in range(l):
        v = int(buf[i])
        value = (v * v) # square it
        n_values += 1
        addendum = value - cumulated_average + cumulated_remainder
        cumulated_average += addendum // n_values
        cumulated_remainder = addendum % n_values
    out = math.sqrt(cumulated_average)
    return out
```

3. 2 ms 2% CPU
Viper emitter
Integer-based
Cumulative Moving Average[1]

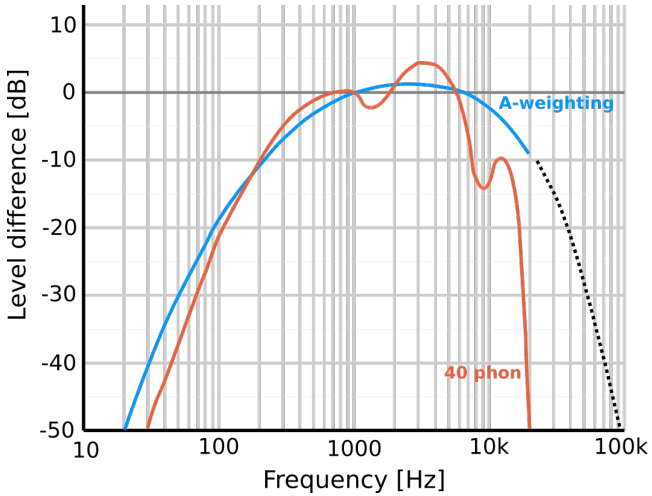
1. CodeProject.com: Stefan Lang (2014)
Precise and safe calculation method for the average of large integer arrays
https://www.codeproject.com/Articles/807195/Precise-and-safe-calculation-method-for-the-averag

emlearn

Sound sensor - IIR filter

Standard sound level measurements are **A-weighted**. To approximate human hearing.

Typically, implemented using **Infinite Impulse Response (IIR) filters**.




```
class IIRFilter():
    def __init__(self, coefficients : array.array):
        stages = len(coefficients)//6
        self.sos = coefficients
        self.state = array.array('f', [0.0]*(2*stages))

    @micropython.native
    def process(self, samples : array.array):
        stages = len(self.sos)//6
        for i in range(len(samples)):
            x = samples[i]
            for s in range(stages):
                b0, b1, b2, a0, a1, a2 = self.sos[s*6:(s*6)+6]
                # compute difference equations of transposed direct form II
                y = b0*x + self.state[(s*2)+0]
                self.state[(s*2)+0] = b1*x - a1*y + self.state[(s*2)+1]
                self.state[(s*2)+1] = b2*x - a2*y
                x = y
            samples[i] = x
```

1100 ms 900% CPU
Native emitter
Float

Too slow by ~10x

```
# 6th order filter. 3x Second Order Sections "biquad"
a_filter_16k = [
    1.0383002230320646, 0.0, 0.0, 1.0, -0.016647242439959593, 6.928267
    1.0, -2.0, 1.0, 1.0, -1.7070508390293027, 0.7174637059318595,
    1.0, -2.0, 1.0, 1.0, -1.9838868447331497, 0.9839517531763131
]
self.frequency_filter = IIRFilter(a_filter_16k)
...
self.frequency_filter.process(samples)
```

Sound sensor - IIR filter

Using **emliir.mpy** native module helps a lot.

BUT - conversion from float/int16 too slow
Also needs a native module

```
import emliir

class IIRFilterEmlearn:

    def __init__(self, coefficients):
        c = array.array('f', coefficients)
        self.iir = emliir.new(c)
    def process(self, samples):
        self.iir.run(samples)
```

IIR filter only
Using emliir.mpy
native module
30 ms 20% CPU - OK

```
@micropython.native
def float_to_int16(inp, out):
    for i in range(len(inp)):
        out[i] = int(inp[i]*32768)

@micropython.native
def int16_to_float(inp, out):
    for i in range(len(inp)):
        out[i] = inp[i]/32768.0

int16_to_float(samples, self.float_array)
self.frequency_filter.process(self.float_array)
float_to_int16(self.float_array, samples)
```

But need to convert data types
Adds 70ms+ with micropython.native
Too slow! Total > 100 ms
Must create native module

emliir.mpy from emlearm-micropython
https://github.com/emlearn/emlearn-micropython/

Cat Detector - Data Acquisition

Want: Classify *low-res* images, at *1 FPS+*

Using mp_camera by cnadler86
https://github.com/cnadler86/micropython-camera-API

96x96 px grayscale.
Takes ~100 ms - OK

Cat Detector - Classification with CNN

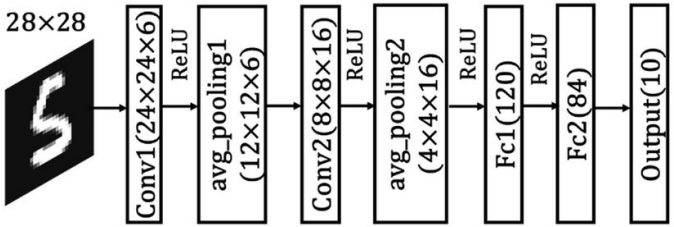
Go-to solution for image classification:
Convolutional Neural Network (CNN)

tinymaix_cnn from emlearn-micropython
32x32 px input. 3 layers. Under 100 ms - OK

Preprocessing (untested)
- Downscaling 96 px -> 32 px
- Image brightness normalization

= Image classification is doable (with native C modules)

See also OpenMV - MicroPython for Computer Vision- https://docs.openmv.io/index.html



https://github.com/emlearn/emlearn-micropython/tree/master/examples/mnist_cnn

```
import tinymaix_cnn # from emlearn-micropython

with open('cat_classifier.tmdl', 'rb') as f:
    model_data = array.array('B', f.read())
    model = tinymaix_cnn.new(model_data)

while True:

    raw = read_camera()
    img = preprocess(raw)
    classification = model.predict(img)

    if classification == MY_CAT:
       open_door()

    machine.lightsleep(500)
```

Inline Assembly

MicroPython can expose Assembler opcodes as Python statements.

Allows to write a function in Assembler
inline in the Python program
Can compile and execute on device

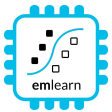
Supported on ARM Cortex M chips
Not supported (yet) on ESP32

For the most hardcore hackers!

Official Documentation:
https://docs.micropython.org/en/latest/reference/asm_thumb2_index.html

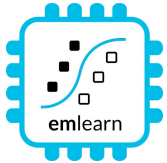
```
@micropython.asm_thumb
def fir(r0, r1, r2):
    mov(r3, r8)         # For Pico: can't push({r8}). r0-r7 only.
    push({r3})
    ldr(r7, [r0, 0])    # Array length
    mov(r6, r7)         # Copy for filter
    mov(r3, r0)
    add(r3, 12)         # r3 points to ring buffer start
    sub(r7, 1)
    add(r7, r7, r7)
    add(r7, r7, r7)     # convert to bytes
    add(r5, r7, r3)     # r5 points to ring buffer end (last valid address)
    ldr(r4, [r0, 8])    # Current insertion point address
    cmp(r4, 0)          # If it's zero we need to initialise
    bne(INITIALISED)
    mov(r4, r3)         # Initialise: point to buffer start
    label(INITIALISED)
    str(r2, [r4, 0])    # put new data in buffer and post increment
    add(r4, 4)
    cmp(r4, r5)         # Check for buffer end
    ble(BUFOK)
    mov(r4, r3)         # Incremented past end: point to start
    label(BUFOK)
    str(r4, [r0, 8])    # Save the insertion point for next call
                        # *** Filter ***
    ldr(r0, [r0, 4])    # Bits to shift
```

Example: FIR filter implementation (cut out)
https://github.com/peterhinch/micropython-filters/blob/master/fir.py



Conclusions

Feasibility and techniques for sensor data processing



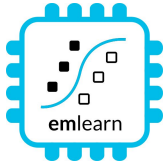
Accelerometer data et.c

Up to 1000 samples/second
= doable with plain Python
+ Benefits from @micropython.native / viper

Sound and low-res/low-rate video

Over 10’000 samples/second
= use of C modules becomes key

Other resources



MicroPython docs: Maximising MicroPython speed
https://docs.micropython.org/en/latest/reference/speed_python.html

PyConAU 2018: Writing fast and efficient MicroPython (Damien George)
https://www.youtube.com/watch?v=hHec4qL00x0

PyConAU 2019: Extending MicroPython: Using C for good!" (Matt Trentini)
https://www.youtube.com/watch?v=437CZBnK8vI

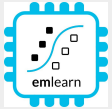
MicroPython docs: Extending MicroPython in C
https://docs.micropython.org/en/latest/develop/extendingmicropython.html

PyCon Berlin 2024: Machine Learning on microcontrollers
using MicroPython and emlearn
https://www.youtube.com/@PyDataTV

Summary

Learn more:
https://github.com/emlearn/emlearn-micropython

1. **Current microcontrollers can run Python** thanks to MicroPython
2. **MicroPython is a powerful platform** for building sensor nodes
3. Wide **range of performance tools available** for creating advanced and efficient sensor nodes
4. **emlearn-micropython** is a collection of native modules for Machine Learning in MicroPython

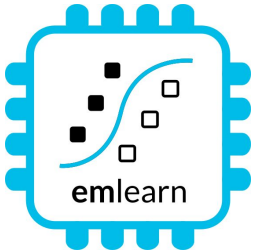


Sensor data processing on microcontrollers

with MicroPython and emlearn

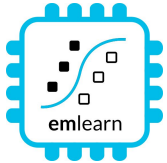
https://github.com/emlearn/emlearn-micropython

Jon Nordby jononor@gmail.com
PyCon ZA 2024



Bonus

TinyML for MicroPython - comparisons



Project	Deployment	Models	Size	Compute time
emlearn -micropython	Easy. Native mod .mpy	DT, RF, KNN, CNN	Good	Good
everywhereml	Easy. Pure Python .py	DT, RF, SVM, KNN,	High with large models	Poor
m2cgen	Easy. Pure Python .py	DT, RF, SVM, KNN, MLP	High with large models	Poor
OpenMV.tf	Hard. Custom Fork	CNN	High initial size	Good
ulab	Hard. User C module	(build-your-own) Using ndarray primitives	High initial size	Unknown (assume good)