

Lab 0: Sine Inference on an STM32F407

ECEN 5003 Embedded AI, Fall 2026. Written up 2026-09-15.

What the project actually is

Despite the repo being named “hello world,” this is a full embedded machine learning pipeline. The job was to train a neural network that approximates $y = \sin(x)$, export it to ExecuTorch, run inference on a bare metal Cortex M4 with no operating system, drive the brightness of an LED from the model’s output using PWM, and print the results over a serial port.

The point of the lab is that the microcontroller never computes a sine. It runs a trained network, on device, at 16 MHz, out of flash.

Board	STM32F4DISCOVERY (STM32F407VG, Cortex M4, 1 MB flash, 128 KB SRAM, 64 KB CCM RAM)
Runtime	ExecuTorch release/1.2
Toolchain	arm-none-eabi-gcc 15.3, CMake, Ninja
Clock	16 MHz HSI, no PLL
LED	PD12, driven by TIM4 channel 1 PWM on AF2
Serial	USART2, PA2 TX, 115200 8N1

What we built, in order

Phase	Deliverable	Result
1	models/train_and_export.py and sine_model.pt	0.000448 mean error, 4100 byte model
2	lib/libexecutorch_selected_kernels.a	8000s, 5.3 KB
3	ops_manifest.json	generated by the training script so it can’t drift
4	src/main_host.cpp	PASS on host, no hardware needed
5	src/main.cpp	firmware, 7.5% of flash
6	Hardware verification	flashed, measured, confirmed

The design decisions that mattered

tanh instead of ReLU. Sine is smooth and bounded in $[-1, 1]$, and so is tanh, so a 1 to 16 to 16 to 1 network fits it closely. ReLU is piecewise linear and would need far more width to hit the same error.

Input shape [1, 1] is baked into the .pte at export time. Both the host harness and the firmware have to build a tensor with exactly that shape or inference fails at `set_input`.

Only 3 operators. `nn.Linear` compiles down to `addmm` plus `permute_copy`, and the activation is `tanh`. That short list is what keeps the firmware at 7.5% of flash: `EXECUTORCH_SELECT_OPS_LIST` controls which kernels get *registered*, and the linker discards everything unregistered. The selected kernels library is only 5.3 KB because it holds nothing but a name to function pointer table.

Both ExecuTorch memory pools live in CCM RAM. That’s 64 KB of CPU only memory at 0x10000000 that DMA can’t reach. Nothing here needs DMA, so putting the 62 KB of pools there leaves essentially all 128 KB of main SRAM free for the stack, which matters because `load_method()` recurses deeply.

The float formatter is written by hand. `newlib nano`’s `printf` has no floating point support and fails silently rather than erroring, so `snprintf("%.4f", y)` would have quietly destroyed 20 points of the grade.

Steps are scheduled against an absolute deadline, not a fixed delay after the work. Inference and UART time would otherwise stretch every step past 100 ms and drift the cadence.

Duty cycle is $(y + 1) / 2$. A full sine period averages zero, so the duty averages 50%, and because the model reaches plus and minus 0.999 the LED hits both full on and full off.

Problems we hit

1. **Op list read at the wrong time.** Reading the operator list *after* `to_executorch()` picked up `alloc` nodes, which are memory planning plumbing, not kernels. Fixed by reading the edge graph before that call.
2. **CMake picked the wrong Python.** The kernel codegen step failed with `No module named 'yaml'` because CMake found Homebrew's python3.14 instead of the ExecuTorch venv. Activating the venv doesn't help, since CMake caches the interpreter path at configure time. Fixed with `-DPYTHON_EXECUTABLE`.
3. **The flash CMake target breaks on paths with spaces.** "Fall 2026" in the path made OpenOCD's TCL program command split the filename. Worked around by running OpenOCD from the build directory with a relative path, since `CMakeLists.txt` is marked `do not modify`.
4. **Wrong header in my wiring instructions.** PA2 was called out as header P1; the lab README says P2.
5. **The UART capture silently ran at 9600 baud.** On macOS, `stty -f` opens the port, applies settings, then closes it, which resets them. The reader then opened it fresh at the default speed. This produced zero output and looked like a hardware fault. Fixed by configuring and reading inside a single file descriptor.
6. **A loose header connection.** Reseating the jumpers was what finally made the UART appear.

Worth noting that problems 4, 5 and 6 all presented identically as "no UART output," which is why the debugging went through the SWD probe first. Reading `CFSR`, `g_ticks` and `TIM4->CCR1` off the running chip proved the firmware was healthy and narrowed the fault to the wire and the capture script.

Final measured results

All of these were taken off the real board, not an emulator.

Measurement	Value
Mean inference error vs <code>sinf()</code>	0.000448
Max inference error	0.001704
Flash used	79 KB of 1 MB (7.5%)
SRAM used	3.1 KB of 128 KB
CFSR / HFSR	0 / 0, no faults after minutes of running
PWM duty range	0.0% to 100.0%
PWM duty mean	brackets 50%
SysTick accuracy	997 ticks per second
UART cadence	1.00 line per second
UART format	15 of 15 lines well formed

Sampled LED brightness across one full 36 second sweep:



Sample UART output:

`x=340 y=-0.3412 angle=5.9341`
`x=350 y=-0.1730 angle=6.1087`

x=0 y=0.0000 angle=0.0000
x=10 y=0.1738 angle=0.1745

Submitted files

CMakeLists.txt, CMakePresets.json, arm-none-eabi.cmake (provided)
startup_stm32f407.s, stm32f407.ld, stm32f407.h (provided)
models/train_and_export.py
models/sine_model.ptc
src/syscalls.c (provided)
src/main.cpp
src/main_host.cpp
lib/libexecutorch_selected_kernels.a
ops_manifest.json
AI-use-statement.md
platforms/ is kept locally for Renode testing but not submitted, per the lab README.