EECE-4710 "IoT and TinyML"

TFL Micro "Hello World" Code Walkthrough

Cris Ababei

MARQUETTE UNIVERSITY

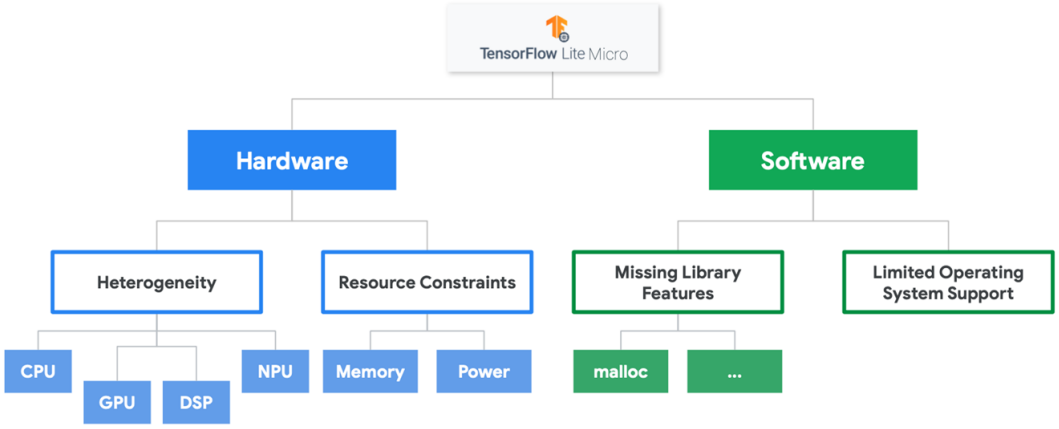
BE THE DIFFERENCE.

1

TFLite Micro - Recap

Inference at MCU level

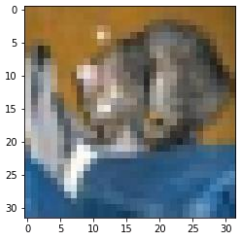
2



3

TensorFlow Lite - Recall

```
1. interpreter = tf.lite.Interpreter("/content/cifar10_quant_model.tflite")
2. interpreter.allocate_tensors()

3. set_input_tensor(interpreter, image)

4. interpreter.invoke()

5. output_details = interpreter.get_output_details()[0]
6. img_pred =np.argmax(interpreter.get_tensor(output_details['index'][0]))
```



image

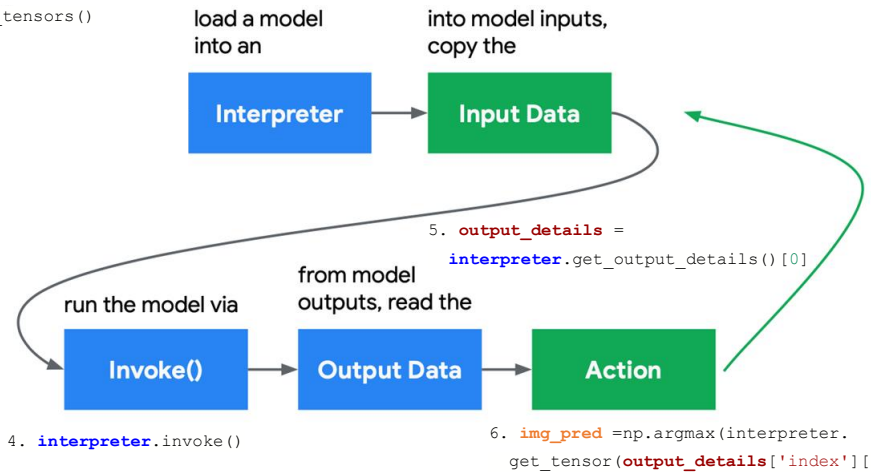
→ 3 ⇒ "CAT"

4

TensorFlow Lite - Recall

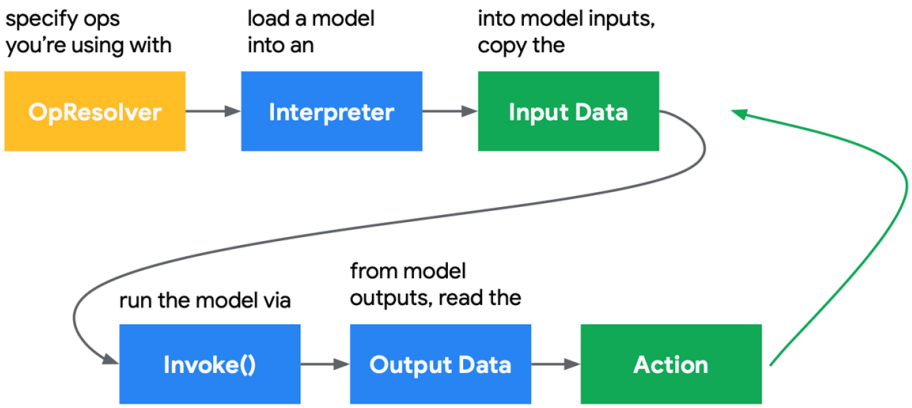
```
1. interpreter = tf.lite.Interpreter("cifar10_quant_model.tflite")
2. interpreter.allocate_tensors()
```

```
3. set_input_tensor(interpreter, image)
```



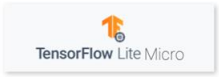
5

How do you use **TFL Micro**?



6

Recap: What is TensorFlow Lite Micro?



Compatible with the TensorFlow training environment.

Built to fit on **embedded systems**:

- Very **small binary footprint**
- **No** dynamic memory allocation
- **No** dependencies on complex parts of the standard C/C++ libraries
- **No** operating system dependencies, **can run on bare metal**
- Designed to be **portable** across a wide variety of systems

7

TensorFlow Lite Micro Hello World Model
Code Walkthrough!

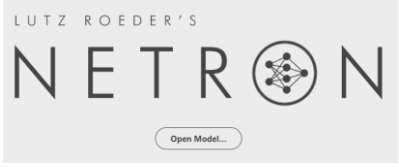
hello_word_V2.ino

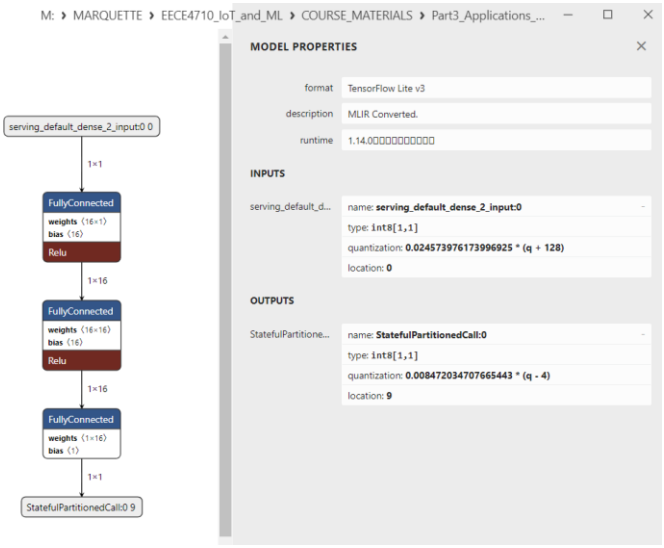
8

8

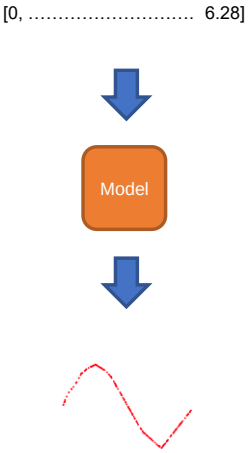
Netron

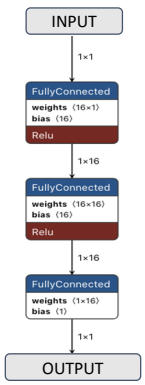
- Netron is a viewer for neural network, deep learning and machine learning models.
- Download and install it on your computer from (if you do not have it already):
  - https://github.com/lutzroeder/netron



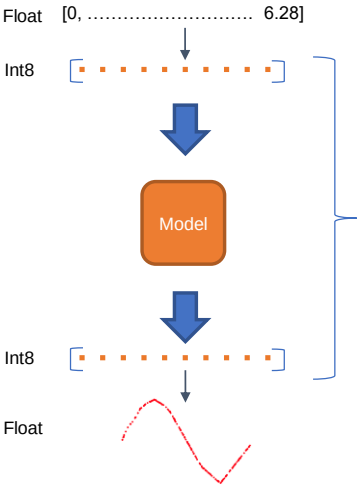


Hello World TFLM model

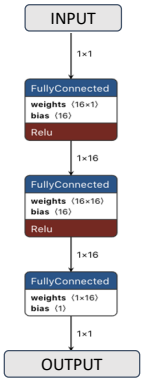




Hello World TFLM model



Model.tflite

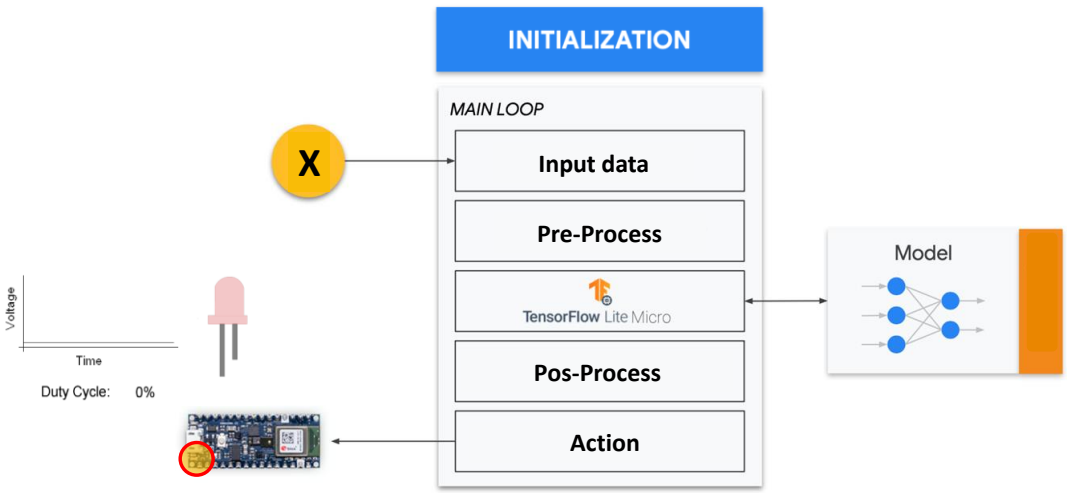


Model.cc

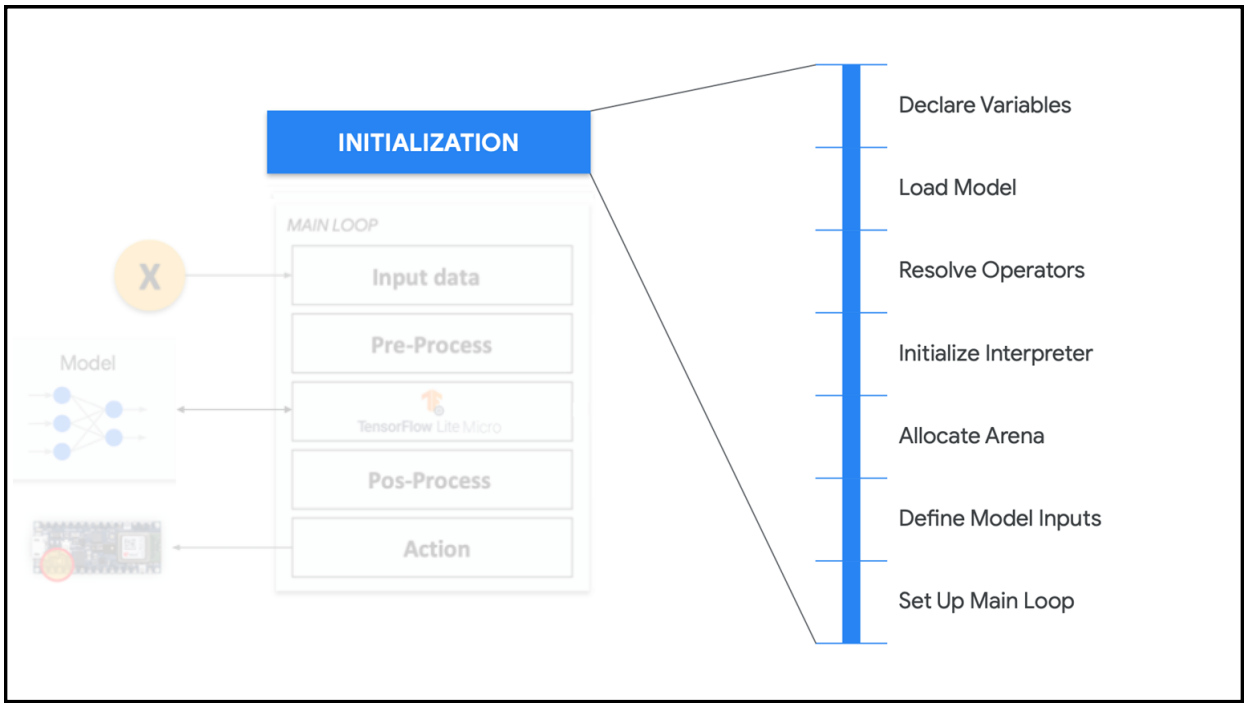
```
1 !cat {MODEL_TFLITE_MICRO}
 0x02, 0x15, 0x01, 0xd1, 0x02, 0xe9, 0xee, 0x07, 0x2d, 0x18, 0xfe, 0x01,
 0x1c, 0xfa, 0x03, 0xf6, 0x0c, 0xf2, 0xed, 0xed, 0x06, 0xf2, 0xfa, 0xda,
 0x0f, 0xf1, 0x06, 0x0e, 0xee, 0xf8, 0x01, 0x0e, 0x07, 0x03, 0xf7, 0x30,
 0xf7, 0xfa, 0xf7, 0x0a, 0xd9, 0xff, 0x12, 0x02, 0xfb, 0x01, 0x14, 0xf8,
 0x07, 0xd8, 0xfd, 0x0b, 0x01, 0x1e, 0xc3, 0x10, 0x20, 0x2c, 0x0f, 0xf1,
 0x04, 0x10, 0x05, 0x2a, 0xd9, 0xf3, 0x0a, 0x00, 0xfd, 0xe0, 0xda, 0x1a,
 0xfb, 0xea, 0xfd, 0xf5, 0x0a, 0x00, 0xff, 0xe8, 0xf3, 0xe4, 0x03, 0x15,
 0x04, 0x0d, 0xff, 0xdb, 0xd9, 0x06, 0x0b, 0xda, 0xdb, 0xf9, 0x00, 0x03,
 0x0b, 0x08, 0x03, 0x03, 0x25, 0xf9, 0xd5, 0x02, 0x0e, 0x0a, 0xf1, 0xf7,
 0x09, 0x0d, 0x0c, 0xb6, 0x12, 0x08, 0x02, 0xf8, 0x04, 0x02, 0x17, 0x10,
 0x0e, 0xdf, 0x01, 0xd0, 0xff, 0x00, 0xfd, 0x0f, 0x1c, 0x02, 0x17, 0x0a,
 0x05, 0xf0, 0xfb, 0xed, 0x21, 0xfe, 0xfd, 0xec, 0xdf, 0x04, 0x03, 0xf9,
...
 0x04, 0x00, 0x00, 0x00, 0x0c, 0x00, 0x00, 0x00, 0x63, 0x6f, 0x6e, 0x76,
 0x32, 0x64, 0x5f, 0x69, 0x6e, 0x70, 0x75, 0x74, 0x00, 0x00, 0x00, 0x00,
 0x04, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00,
 0x20, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00,
 0x60, 0x00, 0x00, 0x00, 0x44, 0x00, 0x00, 0x00, 0x28, 0x00, 0x00, 0x00,
 0x14, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0xd8, 0xff, 0xff, 0xff,
 0x00, 0x00, 0x00, 0x19, 0x19, 0x00, 0x00, 0x00, 0xcc, 0xff, 0xff, 0xff,
 0x00, 0x00, 0x00, 0x09, 0x09, 0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00,
 0xf4, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x16, 0x16, 0x00, 0x00, 0x00,
 0x0c, 0x00, 0x0c, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00,
 0x0c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x11, 0x11, 0x00, 0x00, 0x00,
 0x0c, 0x00, 0x10, 0x00, 0x07, 0x00, 0x00, 0x00, 0x08, 0x00, 0x0c, 0x00,
 0x0c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x05, 0x00, 0x00, 0x00,
 0x03, 0x00, 0x00, 0x00
};
unsigned int g_model_len = 177232;
```

11

Hello World TFLM Components



12


INITIALIZATION
MAIN LOOP
Input data
Pre-Process
TensorFlow Lite Micro
Pos-Process
Action
Model
Declare Variables
Load Model
Resolve Operators
Initialize Interpreter
Allocate Arena
Define Model Inputs
Set Up Main Loop

13


Declare Variables
Load Model
Resolve Operators
Initialize Interpreter
Allocate Arena
Define Model Inputs
Set Up Main Loop
#include "main_functions.h"
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "constants.h"
#include "model.h"
#include "output_handler.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
// Globals, used for compatibility with Arduino-style sketches.
namespace {
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* input = nullptr;
TfLiteTensor* output = nullptr;
int inference_count = 0;
constexpr int kTensorArenaSize = 2000;
uint8_t tensor_arena[kTensorArenaSize];
} // namespace

14

- **Declare Variables**
- Load Model
- Resolve Operators
- Initialize Interpreter
- Allocate Arena
- Define Model Inputs
- Set Up Main Loop

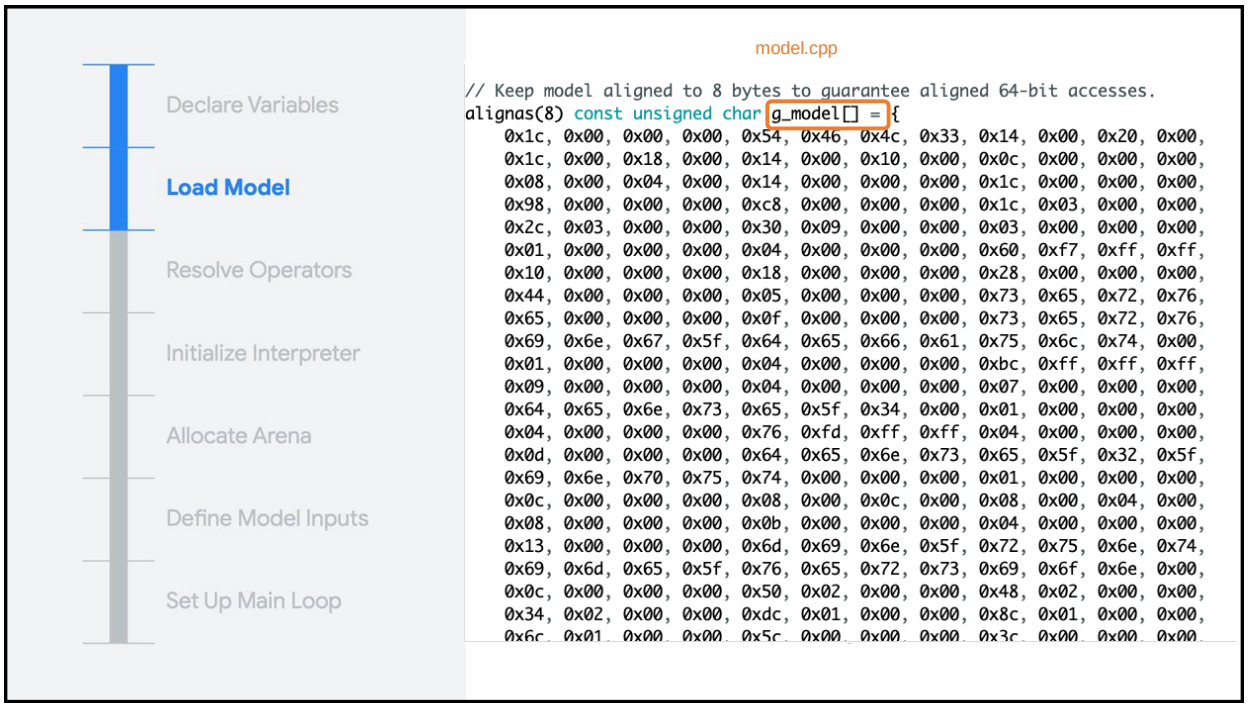
```
#include "main_functions.h"

#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "constants.h"
#include "model.h"
#include "output_handler.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"

// Globals, used for compatibility with Arduino-style sketches.

namespace {
  tflite::ErrorReporter* error_reporter = nullptr;
  const tflite::Model* model = nullptr;
  tflite::MicroInterpreter* interpreter = nullptr;
  TfLiteTensor* input = nullptr;
  TfLiteTensor* output = nullptr;
  int inference_count = 0;

constexpr int kTensorArenaSize = 2000;
uint8_t tensor_arena[kTensorArenaSize];
}  // namespace
```

15

- Declare Variables
- **Load Model**
- Resolve Operators
- Initialize Interpreter
- Allocate Arena
- Define Model Inputs
- Set Up Main Loop

```
void setup() {
  // Set up logging. Google style is to avoid globals or statics because of
  // lifetime uncertainty, but since this has a trivial destructor it's okay.
  // NOLINTNEXTLINE(runtime-global-variables)
  static tflite::MicroErrorReporter micro_error_reporter;
  error_reporter = &micro_error_reporter;

  // Map the model into a usable data structure. This doesn't involve any
  // copying or parsing, it's a very lightweight operation.

  model = tflite::GetModel(g_model);

  if (model->version() != TFLITE_SCHEMA_VERSION) {
    TF_LITE_REPORT_ERROR(error_reporter,
                         "Model provided is schema version %d not equal "
                         "to supported version %d.",
                         model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }

  // This pulls in all the operation implementations we need.
  // NOLINTNEXTLINE(runtime-global-variables)
  static tflite::AllOpsResolver resolver;
```

16



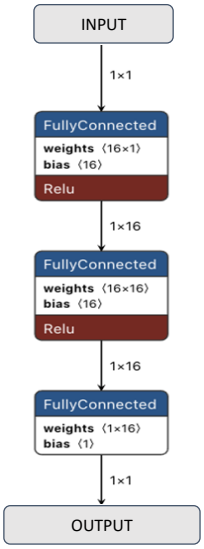
17

What is g_model?

- **Array of bytes**, and acts as the equivalent of a file on disk
- Holds **all** of the **information about the model**, its **operators**, their **connections**, and the trained **weights**

18

- Declare Variables
- Load Model
- **Resolve Operators**
- Initialize Interpreter
- Allocate Arena
- Define Model Inputs
- Set Up Main Loop



19

- Declare Variables
- Load Model
- **Resolve Operators**
- Initialize Interpreter
- Allocate Arena
- Define Model Inputs
- Set Up Main Loop

Load ALL Ops

```
// This pulls in all the operation implementations we need.
// NOLINTNEXTLINE(runtime-global-variables)
static tflite::AllOpsResolver resolver;

// Build an interpreter to run the model with.
static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors.
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
  TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
  return;
}

// Obtain pointers to the model's input and output tensors.
input = interpreter->input(0);
output = interpreter->output(0);

// Keep track of how many inferences we have performed.
inference_count = 0;
```

Used if you do not have problem with memory

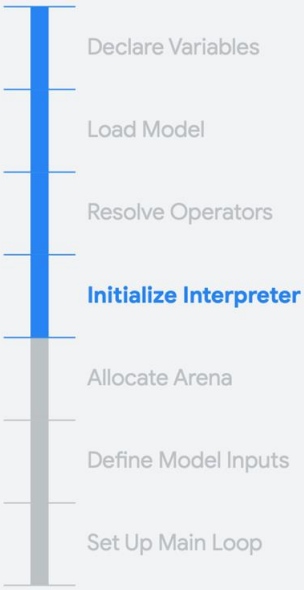
20

Load only the needed Ops

```
static tflite::MicroMutableOpResolver <3>
micro_op_resolver(error_reporter);
if (micro_op_resolver.AddBuiltin(
  tflite::BuiltinOperator_FULLY_CONNECTED,
  tflite::ops::micro::Register_FULLY_CONNECTED()) != kTfLiteOk)
{
  return;
}
if (micro_op_resolver.AddBuiltin(
  tflite::BuiltinOperator_FULLY_CONNECTED,
  tflite::ops::micro::Register_FULLY_CONNECTED()) != kTfLiteOk)
{
  return;
}
if (micro_op_resolver.AddBuiltin(
  tflite::BuiltinOperator_FULLY_CONNECTED,
  tflite::ops::micro::Register_FULLY_CONNECTED()) != kTfLiteOk)
{
  return;
}
```

Used if you have problem with memory

21

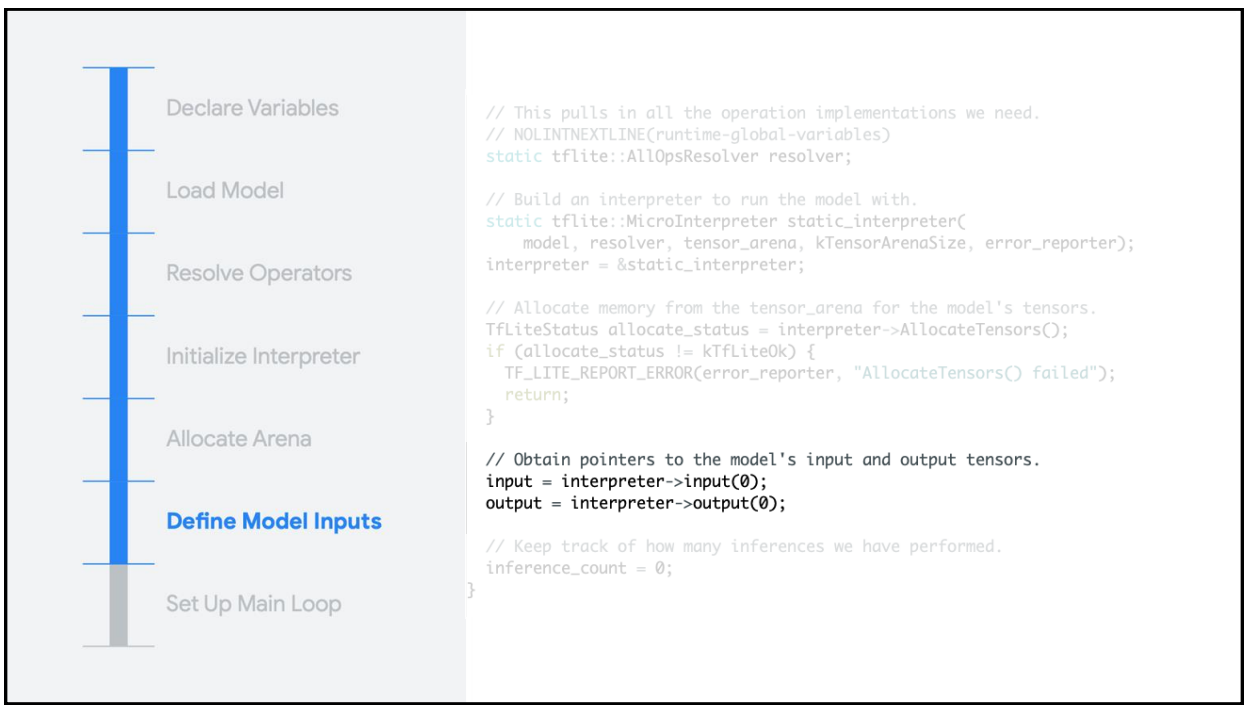



```
// This pulls in all the operation implementations we need.
// NOLINTNEXTLINE(runtime-global-variables)
static tflite::AllOpsResolver resolver;

// Build an interpreter to run the model with.
static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors.
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
  TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
  return;
}

// Obtain pointers to the model's input and output tensors.
input = interpreter->input(0);
output = interpreter->output(0);

// Keep track of how many inferences we have performed.
inference_count = 0;
}
```

22


Declare Variables
Load Model
Resolve Operators
Initialize Interpreter
Allocate Arena
Define Model Inputs
Set Up Main Loop
// This pulls in all the operation implementations we need.
// NOLINTNEXTLINE(runtime-global-variables)
static tflite::AllOpsResolver resolver;
// Build an interpreter to run the model with.
static tflite::MicroInterpreter static_interpreter(
model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
interpreter = &static_interpreter;
// Allocate memory from the tensor_arena for the model's tensors.
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
return;
}
// Obtain pointers to the model's input and output tensors.
input = interpreter->input(0);
output = interpreter->output(0);
// Keep track of how many inferences we have performed.
inference_count = 0;
}

23

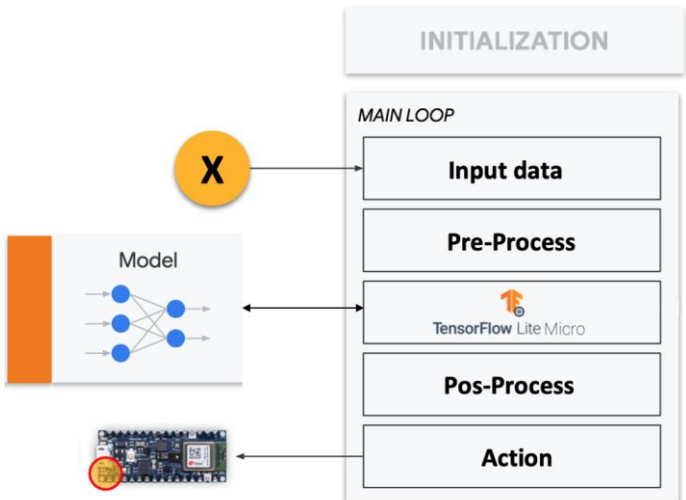

Declare Variables
Load Model
Resolve Operators
Initialize Interpreter
Allocate Arena
Define Model Inputs
Set Up Main Loop
// This pulls in all the operation implementations we need.
// NOLINTNEXTLINE(runtime-global-variables)
static tflite::AllOpsResolver resolver;
// Build an interpreter to run the model with.
static tflite::MicroInterpreter static_interpreter(
model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
interpreter = &static_interpreter;
// Allocate memory from the tensor_arena for the model's tensors.
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
return;
}
// Obtain pointers to the model's input and output tensors.
input = interpreter->input(0);
output = interpreter->output(0);
// Keep track of how many inferences we have performed.
inference_count = 0;
}

24

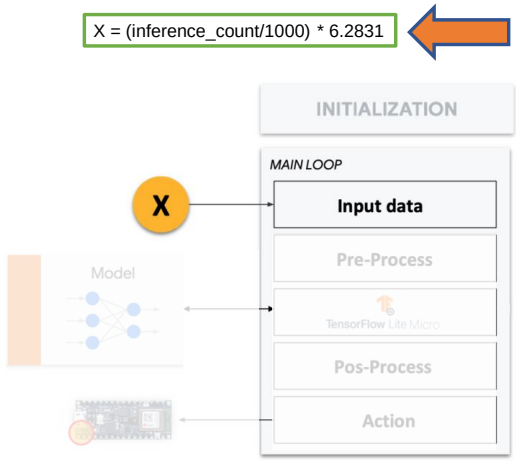
- Declare Variables
- Load Model
- Resolve Operators
- Initialize Interpreter
- Allocate Arena
- Define Model Inputs
- **Set Up Main Loop**

```
// This pulls in all the operation implementations we need.
// NOLINTNEXTLINE(runtime-global-variables)
static tflite::AllOpsResolver resolver;

// Build an interpreter to run the model with.
static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors.
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
  TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
  return;
}

// Obtain pointers to the model's input and output tensors.
input = interpreter->input(0);
output = interpreter->output(0);

// Keep track of how many inferences we have performed.
inference_count = 0;
}
```

25

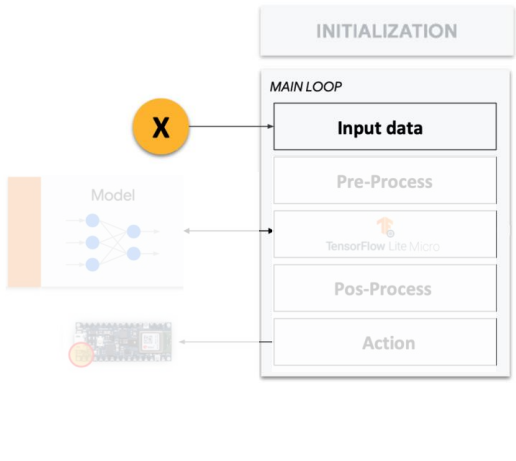


26




```
void loop() {
  // Calculate an x value to feed into the model. We compare the current
  // inference_count to the number of inferences per cycle to determine
  // our position within the range of possible x values the model was
  // trained on, and use this to calculate a value.
  float position = static_cast<float>(inference_count) /
                   static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input->params.scale + input->params.zero_point;
  // Place the quantized input in the model's input tensor
  input->data.int8[0] = x_quantized;

  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x: %f\n",
                         static_cast<double>(x));
    return;
  }
  // Obtain the quantized output from model's output tensor
  int8_t y_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y = (y_quantized - output->params.zero_point) * output->params.scale;

  // Output the results. A custom HandleOutput function can be implemented
  // for each supported hardware target.
  HandleOutput(error_reporter, x, y);

  // Increment the inference_counter, and reset it if we have reached
  // the total number per cycle
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle) inference_count = 0;
}
```

27



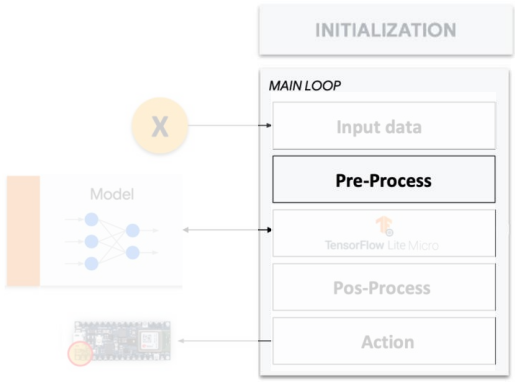
arduino_constants.cpp

```
#include "constants.h"

// This is tuned so that a full cycle takes ~4 seconds on an Arduino MKRZERO.
const int kInferencesPerCycle = 1000;
```

constants.h

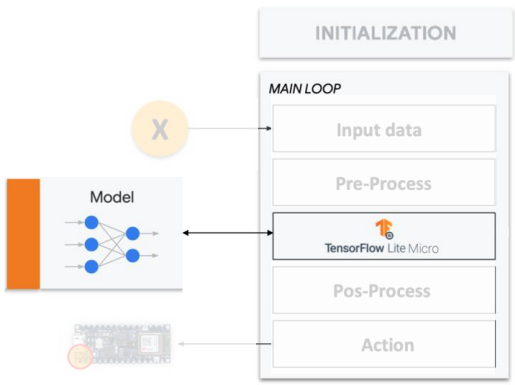
```
#ifndef TENSORFLOW_LITE_MICRO_EXAMPLES_HELLO_WORLD_CONSTANTS_H_
#define TENSORFLOW_LITE_MICRO_EXAMPLES_HELLO_WORLD_CONSTANTS_H_

// This constant represents the range of x values our model was trained on,
// which is from 0 to (2 * Pi). We approximate Pi to avoid requiring additional
// libraries
const float kXrange = 2.f * 3.14159265359f;

// This constant determines the number of inferences to perform across the range
// of x values defined above. Since each inference takes time, the higher this
// number, the more time it will take to run through the entire range. The value
// of this constant can be tuned so that one full cycle takes a desired amount
// of time. Since different devices take different amounts of time to perform
// inference, this value should be defined per-device.
extern const int kInferencesPerCycle;

#endif  // TENSORFLOW_LITE_MICRO_EXAMPLES_HELLO_WORLD_CONSTANTS_H_
```

28



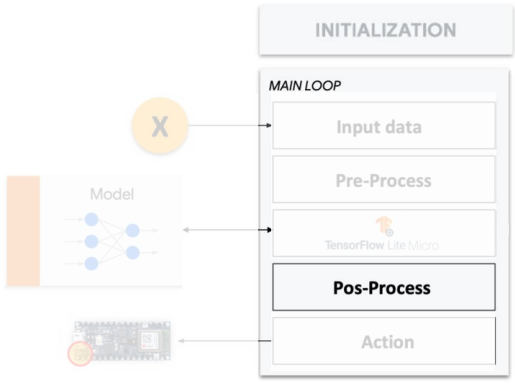

```
void loop() {
  // Calculate an x value to feed into the model. We compare the current
  // inference_count to the number of inferences per cycle to determine
  // our position within the range of possible x values the model was
  // trained on, and use this to calculate a value.
  float position = static_cast<float>(inference_count) /
                   static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input->params.scale + input->params.zero_point;
  // Place the quantized input in the model's input tensor
  input->data.int8[0] = x_quantized;

  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x: %f\n",
                         static_cast<double>(x));
    return;
  }
  // Obtain the quantized output from model's output tensor
  int8_t y_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y = (y_quantized - output->params.zero_point) * output->params.scale;

  // Output the results. A custom HandleOutput function can be implemented
  // for each supported hardware target.
  HandleOutput(error_reporter, x, y);

  // Increment the inference_counter, and reset it if we have reached
  // the total number per cycle
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle) inference_count = 0;
}
```

29



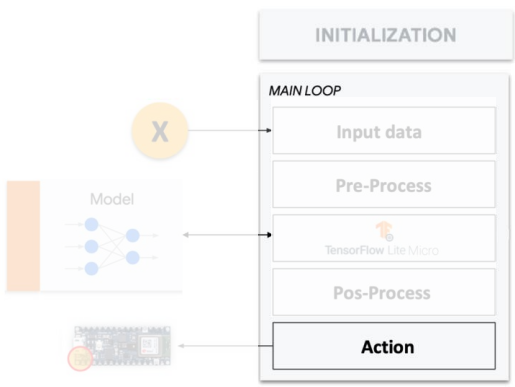

```
void loop() {
  // Calculate an x value to feed into the model. We compare the current
  // inference_count to the number of inferences per cycle to determine
  // our position within the range of possible x values the model was
  // trained on, and use this to calculate a value.
  float position = static_cast<float>(inference_count) /
                   static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input->params.scale + input->params.zero_point;
  // Place the quantized input in the model's input tensor
  input->data.int8[0] = x_quantized;

  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x: %f\n",
                         static_cast<double>(x));
    return;
  }
  // Obtain the quantized output from model's output tensor
  int8_t y_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y = (y_quantized - output->params.zero_point) * output->params.scale;

  // Output the results. A custom HandleOutput function can be implemented
  // for each supported hardware target.
  HandleOutput(error_reporter, x, y);

  // Increment the inference_counter, and reset it if we have reached
  // the total number per cycle
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle) inference_count = 0;
}
```

30



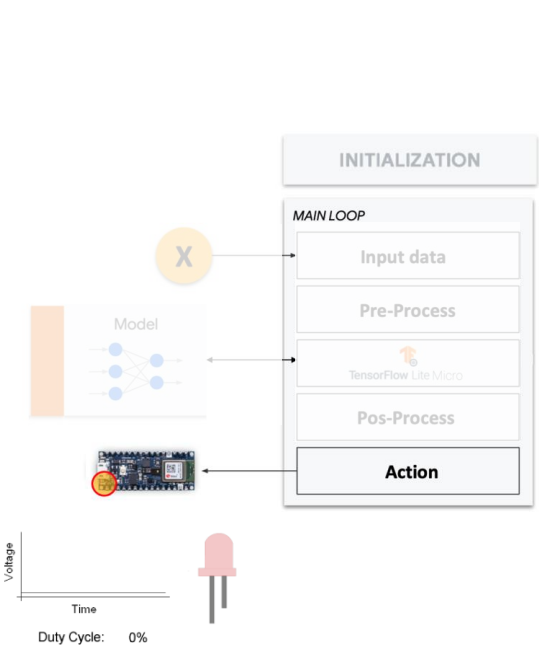

```
void loop() {
  // Calculate an x value to feed into the model. We compare the current
  // inference_count to the number of inferences per cycle to determine
  // our position within the range of possible x values the model was
  // trained on, and use this to calculate a value.
  float position = static_cast<float>(inference_count) /
                   static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input->params.scale + input->params.zero_point;
  // Place the quantized input in the model's input tensor
  input->data.int8[0] = x_quantized;

  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x: %f\n",
                         static_cast<double>(x));
    return;
  }
  // Obtain the quantized output from model's output tensor
  int8_t y_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y = (y_quantized - output->params.zero_point) * output->params.scale;

  // Output the results. A custom HandleOutput function can be implemented
  // for each supported hardware target.
  HandleOutput(error_reporter, x, y);

  // Increment the inference_counter, and reset it if we have reached
  // the total number per cycle
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle) inference_count = 0;
}
```

31




```
void loop() {
  // Calculate an x value to feed into the model. We compare the current
  // inference_count to the number of inferences per cycle to determine
  // our position within the range of possible x values the model was
  // trained on, and use this to calculate a value.
  float position = static_cast<float>(inference_count) /
                   static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input->params.scale + input->params.zero_point;
  // Place the quantized input in the model's input tensor
  input->data.int8[0] = x_quantized;

  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x: %f\n",
                         static_cast<double>(x));
    return;
  }
  // Obtain the quantized output from model's output tensor
  int8_t y_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y = (y_quantized - output->params.zero_point) * output->params.scale;

  // Output the results. A custom HandleOutput function can be implemented
  // for each supported hardware target.
  HandleOutput(error_reporter, x, y);

  // Increment the inference_counter, and reset it if we have reached
  // the total number per cycle
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle) inference_count = 0;
}
```

32



arduino_output_handler.cpp

```
#include "output_handler.h"

#include "Arduino.h"
#include "constants.h"

// The pin of the Arduino's built-in LED
int led = LED_BUILTIN;

// Track whether the function has run at least once
bool initialized = false;

// Animates a dot across the screen to represent the current x and y values
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
  // Do this only once
  if (!initialized) {
    // Set the LED pin to output
    pinMode(led, OUTPUT);
    initialized = true;
  }

  // Calculate the brightness of the LED such that y=-1 is fully off
  // and y=1 is fully on. The LED's brightness can range from 0-255.
  int brightness = (int)(127.5f * (y_value + 1));

  // Set the brightness of the LED. If the specified pin does not support PWM,
  // this will result in the LED being on when y > 127, off otherwise.
  analogWrite(led, brightness);

  // Log the current brightness value for display in the Arduino plotter
  TF_LITE_REPORT_ERROR(error_reporter, "%d\n", brightness);
}
```

33

Credits

- A previous edition of this course was developed in collaboration with Dr. Susan C. Schneider of Marquette University.
- We are very grateful and thank all the following professors, researchers, and practitioners for jump-starting courses on TinyML and for sharing their teaching materials:

- Prof. Marcelo Rovai - TinyML - Machine Learning for Embedding Devices, UNIFEI
  - https://github.com/Mjrovai/UNIFEI-IESTI01-TinyML-2022.1
- Prof. Vijay Janapa Reddi - CS249r: Tiny Machine Learning, Applied Machine Learning on Embedded IoT Devices, Harvard
  - https://sites.google.com/g.harvard.edu/tinyml/home
- Prof. Rahul Mangharam – ESE3600: Tiny Machine Learning, Univ. of Pennsylvania
  - https://tinyml.seas.upenn.edu/#
- Prof. Brian Plancher - Harvard CS249r: Tiny Machine Learning (TinyML), Barnard College, Columbia University
  - https://a2r-lab.org/courses/cs249r_tinyml/

34

34

References

- Additional references from where information and other teaching materials were gathered include:

- Applications & Deploy textbook: “TinyML” by Pete Warden, Daniel Situnayake
  - https://www.oreilly.com/library/view/tinyml/9781492052036/
- Deploy textbook “TinyML Cookbook” by Gian Marco Iodice
  - https://github.com/PacktPublishing/TinyML-Cookbook
- Jason Brownlee
  - https://machinelearningmastery.com/
- TinyMLedu
  - https://tinyml.seas.harvard.edu/
- Professional Certificate in Tiny Machine Learning (TinyML) – edX/Harvard
  - https://www.edx.org/professional-certificate/harvardx-tiny-machine-learning
- Introduction to Embedded Machine Learning - Coursera/Edge Impulse
  - https://www.coursera.org/learn/introduction-to-embedded-machine-learning
- Computer Vision with Embedded Machine Learning - Coursera/Edge Impulse
  - https://www.coursera.org/learn/computer-vision-with-embedded-machine-learning

35

35