Module 2

Model Deployment on MCUs

Learning Objectives

- Convert a trained neural network into a TensorFlow Lite model.
- Optimize models for size and speed.
- Understand quantization and its role in TinyML.

2.1 Introduction

Neural networks trained on desktops or cloud servers are often too large for microcontrollers. Neural networks trained on desktops or cloud servers (e.g., using TensorFlow or PyTorch) are typically too large and computationally expensive for microcontrollers. A typical pre-trained CNN (e.g., MobileNet) may require:

- Hundreds of MBs of storage (vs. <256 KB on many MCUs).
- Floating-point operations (FP32) (vs. 8-bit integer (INT8) ops on low-end MCUs).
- High memory bandwidth (vs. limited SRAM/Flash on embedded devices).

Table 2.1: Key Challenges in TinyML Deployment

Challenge	Cloud/PC Model	Microcontroller Constraint
Model Size	10–100 MB	<256 KB (often <64 KB)
Compute Precision	32-bit floating point (FP32)	8-bit integers (INT8)
Memory Usage	GBs of RAM	<100 KB SRAM
Power Consumption	Watts	Milliwatts (mW)
Latency	10–100 ms	<10 ms (real-time)
Supported Operations	Full TensorFlow/PyTorch ops	Limited TFLite ops

Solution TensorFlow Lite (TFLite) for Microcontrollers

TensorFlow Lite (TFLite) is an optimized framework for on-device ML, with a specialized runtime (TFLite for Microcontrollers) for bare-metal embedded systems.

Why TFLite?

- Smaller models (via quantization & pruning).
- Faster inference (optimized kernels for ARM Cortex-M).
- Low memory footprint (works with <16 KB RAM).
- Cross-platform support (STM32, Arduino, ESP32, RP2040).

2.2 Workflow: From Training to Deployment

Step 1: Train a Model

Before deployment, we need a **trained model**. This model is usually built with frameworks such as `TensorFlow` or `Keras`. Examples include a CNN for image classification or an LSTM for time-series data.

```
import tensorflow as tf
from tensorflow.keras import layers

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
    load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize

# Define a simple CNN
model = tf.keras.Sequential([
    layers.Reshape((28, 28, 1), input_shape=(28, 28)),
    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)
])

model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
        True),
    metrics=['accuracy']
)

model.fit(x_train, y_train, epochs=5)
```

Key Considerations for TinyML:

- Model architecture should be lightweight (fewer layers, smaller filters).
- Avoid unsupported ops (e.g., LSTM may not work on all MCUs; use Dense or Conv2D instead).
- Input shape must match sensor data (e.g., 28×28 for MNIST, but 32×32 or 96×96 for camera inputs).

Step 2: Convert to TensorFlow Lite

Once the model is trained, we convert it to **TensorFlow Lite (TFLite)** format for deployment on microcontrollers and other edge devices. TFLite models are optimized for small size and efficient inference on low-power hardware.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
```

```
# Save the model
with open("mnist_model_float32.tflite", "wb") as f:
    f.write(tflite_model)
```

Output:

- `mnist_model_float32.tflite` (~1.2 MB for this CNN).
- Still too large for most MCUs (needs quantization).

Step 3: Apply Quantization

Quantization converts 32-bit floating-point (FP32) weights to 8-bit integers (INT8), shrinking model size, boosting inference speed, and cutting power use on microcontrollers.

Benefits:

- **Model size:** up to 4× smaller.
- **Compute load:** INT8 operations are faster on MCUs.
- **Power consumption:** fewer memory accesses.

Post-Training Quantization (PTQ)

PTQ converts a trained model to INT8 without retraining. It's the simplest way to optimize for microcontrollers, typically with $< 1\%$ accuracy loss if well-calibrated.

```
  import tensorflow as tf
  converter = tf.lite.TFLiteConverter.from_keras_model(model)
  converter.optimizations = [tf.lite.Optimize.DEFAULT]
  tflite_model = converter.convert()

  with open("mnist_model_int8.tflite", "wb") as f:
  f.write(tflite_model)
```

Listing 2.1: Post-training quantization with TensorFlow Lite

Output:

- File: `mnist_model_int8.tflite` (~300 KB, 4× smaller).
- Accuracy: Typically $< 1\%$ drop.

Quantization-Aware Training (QAT)

QAT simulates INT8 quantization during training to minimize accuracy loss. Ideal for critical applications (e.g., medical, industrial).

```
import tensorflow as tf
import tensorflow_model_optimization as tfmot
```

```
model = tf.keras.Sequential([...])
model.compile(...)

# Apply quantization-aware training
quant_aware_model = tfmot.quantization.keras.quantize_model(model)
quant_aware_model.fit(x_train, y_train, epochs=5)

# Convert to TFLite
converter = tf.lite.TFLiteConverter.from_keras_model(
    quant_aware_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
```

Listing 2.2: Quantization-aware training with TensorFlow Lite

Is this method for you?

- Minimizes accuracy loss via INT8 simulation.
- Best for critical applications.
- More complex; not all quantization backends supported.

Note: For best PTQ results, provide representative calibration data to the converter (e.g., `converter.representative_dataset`). This ensures optimal quantization scaling.

Step 4: Verify Model Compatibility with TFLite for Microcontrollers

Not all TensorFlow operations are supported on MCUs. Check compatibility before deployment:

```
import tensorflow as tf

# Load TFLite model and check ops
interpreter = tf.lite.Interpreter(model_content=tflite_model)
print("Supported ops:", interpreter.get_tensor_details())

# List of supported ops for ARM Cortex-M:
# https://www.tensorflow.org/lite/microcontrollers/supported_ops
```

Listing 2.3: Checking TFLite model operations on MCUs

Table 2.2: Common Unsupported Ops and Workarounds:

Unsupported Op	**Workaround**
LSTM	Use SimpleRNN or Conv1D
String operations	Preprocess data before inference
Custom layers	Replace with TFLite-compatible layers
Dynamic shapes	Fix input shape (e.g., (1, 28, 28, 1))

Step 5: Deploy to a Microcontroller

Now that the model is trained and optimized, we can deploy the `.tflite` file onto a microcontroller (MCU) such as STM32, Arduino, or ESP32.

Option 1: Edge Impulse (No-Code Deployment)

For those who prefer a no-code workflow:

1. Upload the `.tflite` model to **Edge Impulse**.
2. Select target hardware (e.g., STM32Cube.AI, Arduino, ESP32).
3. Download the auto-generated firmware and flash it onto the device.

Option 2: STM32Cube.AI (STMicroelectronics)

STM32Cube.AI provides an official toolchain for running ML on STM32 MCUs:

1. Import the `.tflite` model into **STM32CubeMX**.
2. Analyze the model (RAM/Flash usage, performance).
3. Generate C code and integrate it into your STM32CubeIDE project.

Option 3: TensorFlow Lite for Microcontrollers (C++)

First, convert the `.tflite` model into a C header that can be compiled into firmware:

```
xxd -i model.tflite > model.h  # Converts binary to C array
```

Listing 2.4: Convert TFLite model into a C array

The `xxd` command is is part of the Vim package and is usually pre-installed on most Linux distributions.

Integrate the generated `model.h` file into an embedded project (e.g., STM32CubeIDE or Arduino IDE). Inference can be performed as follows:

```
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "model.h"  // Generated from xxd

// Load model
const tflite::Model* model = tflite::GetModel(g_model);
static tflite::MicroInterpreter static_interpreter(
model, tflite::AllOpsResolver(), tensor_arena, kTensorArenaSize);

// Run inference
TfLiteStatus invoke_status = static_interpreter.Invoke();
if (invoke_status != kTfLiteOk) { /* Handle error */ }
```

Listing 2.5: Running inference on MCU with TFLite Micro

2.3 Hands-On 1: MNIST Classifier on Portenta H7

Goal: Deploy a quantized MNIST model on Arduino Portenta H7 for fast digit recognition.

Steps

1. Train a simple CNN on MNIST
2. Quantize to INT8 with TensorFlow Lite using a representative dataset
3. Convert to C array for Arduino
4. Generate properly quantized sample digits (1–5) as `digits.h`
5. Flash to Portenta H7 via Arduino IDE
6. Test with simulated or camera input

Python: Train & Quantize

```
# Simple CNN for MNIST
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

# Define model architecture
model = tf.keras.Sequential([
layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
layers.MaxPooling2D(),
layers.Conv2D(64, 3, activation='relu'),
layers.MaxPooling2D(),
layers.Flatten(),
layers.Dense(64, activation='relu'),
layers.Dense(10, activation='softmax')
])

# Load and preprocess MNIST data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
    load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28, 28, 1).astype(np.float32)
x_test = x_test.reshape(-1, 28, 28, 1).astype(np.float32)

# Compile and train model
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, validation_data=(x_test,
    y_test))

# Create representative dataset for quantization
def representative_data_gen():
for i in range(100):
yield [x_test[i:i+1]]
```

```
# Quantize to INT8
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.
    TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

tflite_model = converter.convert()

# Save the quantized model
with open("mnist_portenta.tflite", "wb") as f:
f.write(tflite_model)
```

Listing 2.6: MNIST training and INT8 quantization with representative dataset

Note: Using a representative dataset ensures proper quantization of both weights and activations to INT8.

Python: Generate Sample Digits

To test on Portenta without a camera, export digits 1–5 into a header with proper quantization:

```
import numpy as np
from tensorflow.keras.datasets import mnist

# Load MNIST test data
(_, _), (x_test, y_test) = mnist.load_data()

# Quantization parameters (from previous step)
input_scale = 0.003921568859368563  # Replace with your actual
    values
input_zero_point = -128              # Replace with your actual
    values

with open("digits.h", "w") as f:
f.write("// Sample MNIST digits 1--5 (quantized)\n")
f.write("#ifndef DIGITS_H\n")
f.write("#define DIGITS_H\n\n")

f.write("// Quantization parameters\n")
f.write(f"#define INPUT_SCALE {input_scale}f\n")
f.write(f"#define INPUT_ZERO_POINT {input_zero_point}\n\n")

for d in range(1, 6):
idx = np.where(y_test == d)[0][0]
arr = x_test[idx].astype(np.float32).flatten() / 255.0

# Quantize the data
quantized_arr = np.round((arr / input_scale) + input_zero_point)
quantized_arr = np.clip(quantized_arr, -128, 127).astype(np.int8)

f.write(f"const int8_t digit{d}[28*28] = {{\n")
    for i, val in enumerate(quantized_arr):
    f.write(f"0x{val & 0xFF:02X}, ")
    if (i+1) % 28 == 0:
    f.write("\n")
    f.write("};\n\n")

  f.write("#endif // DIGITS_H\n")
```

Listing 2.7: Exporting quantized sample digits to C header

Note: Include `digits.h` in your Arduino sketch and copy one of the arrays into the input tensor before running inference.

Arduino: Deploy on Portenta H7

Convert `mnist_portenta.tflite` to C header using `xxd`:

```
xxd -i mnist_portenta.tflite > model_data.h
```

```
// Include necessary libraries
#include "TensorFlowLite.h"
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "model_data.h"  // Generated from .tflite
#include "digits.h"      // Quantized sample digits

// Globals
namespace {
  tflite::MicroErrorReporter micro_error_reporter;
  tflite::ErrorReporter* error_reporter = &micro_error_reporter;
  const tflite::Model* model = tflite::GetModel(
      g_mnist_portenta_tflite);
  tflite::MicroInterpreter* interpreter = nullptr;
  TfLiteTensor* input = nullptr;
  TfLiteTensor* output = nullptr;

  constexpr int kTensorArenaSize = 100 * 1024;
  uint8_t tensor_arena[kTensorArenaSize];
} // namespace

void setup() {
  Serial.begin(115200);
  while (!Serial) { ; }

  if (model->version() != TFLITE_SCHEMA_VERSION) {
    error_reporter->Report("Model version does not match Schema")
        ;
    return;
  }

  static tflite::AllOpsResolver resolver;
  static tflite::MicroInterpreter static_interpreter(
  model, resolver, tensor_arena, kTensorArenaSize, error_reporter
      );
  interpreter = &static_interpreter;

  if (interpreter->AllocateTensors() != kTfLiteOk) {
    error_reporter->Report("AllocateTensors() failed");
    return;
  }

  input = interpreter->input(0);
  output = interpreter->output(0);
  Serial.println("Portenta H7 MNIST ready!");
}

void loop() {
  const int8_t* test_image = digit1;  // Change to digit2, digit3
      , etc.

  for (int i = 0; i < 784; i++) {
    input->data.int8[i] = test_image[i];
  }

  if (interpreter->Invoke() != kTfLiteOk) {
    error_reporter->Report("Invoke failed");
    return;
  }

  int8_t max_val = output->data.int8[0];
  int max_idx = 0;
  for (int i = 1; i < 10; i++) {
    if (output->data.int8[i] > max_val) {
      max_val = output->data.int8[i];
      max_idx = i;
    }
  }

  float confidence = (max_val - output->params.zero_point) *
      output->params.scale;
  Serial.print("Predicted digit: ");
  Serial.print(max_idx);
  Serial.print(", Confidence: ");
  Serial.println(confidence, 4);

  delay(2000);
}
```

Listing 2.8: MNIST inference on Portenta H7 with quantized model

Testing

- Flash to Portenta H7 using Arduino IDE (select *Arduino Mbed OS Portenta Boards > Portenta H7 (M7 core)*)
- Open Serial Monitor (115200 baud)
- Expected: *Predicted digit: 1* with high confidence when using `digit1` array
- Change the `test_image` pointer to `digit2`, `digit3`, etc. to test other numbers

Next: Connect a camera module to capture handwritten digits in real-time!

2.4 Generating Header Files on Windows

To deploy the quantized model on Portenta H7, convert the `.tflite` file to a C header (`.h`) file. The `xxd` tool is common but not native to Windows. Use this Python script instead, which requires only Python (already needed for TensorFlow).

```
import tensorflow as tf
from tensorflow.lite.python.util import convert_bytes_to_c_source

# Load .tflite file
with open('mnist_model_int8.tflite', 'rb') as f:
tflite_model = f.read()

# Generate C source and header
source_text, header_text = convert_bytes_to_c_source(tflite_model,
    'mnist_model_int8')

# Save files
with open('mnist_model_int8.cc', 'w') as f:
f.write(source_text)
with open('mnist_model_int8.h', 'w') as f:
f.write(header_text)
```

Listing 2.9: Convert .tflite to C header on Windows

Steps:

1. Run the script in Python (e.g., `python convert.py`).
2. Include `mnist_model_int8.h` in your Arduino sketch.
3. Use `tflite::GetModel(mnist_model_int8)` to load the model.

Alternative: If you prefer a standalone tool, download `bin2h.exe` from `https://www.codeproject.com/articles/22001` and run:

```
bin2h.exe -i mnist_model_int8.tflite -o mnist_model_int8.h
```

Both methods produce a header file compatible with TensorFlow Lite Micro for Portenta H7 deployment.