TinyML on the Edge: A Practical Guide with TFLite and Colab

A TinyML Enthusiast

October 31, 2025

Contents

Preface **1**

1 TinyML - The Future is on the Edge **3**
1.1 The Core Challenge: Severe Constraints 3
1.2 The TinyML Workflow 3

2 Fault Detection - A Practical Application **5**
2.1 Anomaly Detection vs. Classification 5
2.2 Use Case: Gesture Recognition 5

3 Sensors and Microcontrollers - The Building Blocks **7**
3.1 The "Senses" of TinyML: Common Sensors 8
3.2 The "Brain" of TinyML: Microcontrollers 8

4 Object Detection with Computer Vision **9**
4.1 Vision Tasks: From Simple to Complex 9
4.2 Building an Object Detection Model 9

5 Practical Considerations: Data Quality and Model Selection **11**
5.1 The "Garbage In, Garbage Out" Principle 11
5.2 The Accuracy vs. Efficiency Trade-off 11

6 F1 Score: Understanding Model Performance **13**
6.1 Precision: The Quality of Positive Predictions 13
6.2 Recall: The Completeness of Positive Predictions 13
6.3 F1 Score: The Harmonic Mean 14

7 Real-World Examples: Where TinyML is Used **15**

8 From Training to Deployment **17**
8.1 The TFLite Micro Model 17
8.2 The Deployment Workflow 17

9 Practical Projects: Putting Your Knowledge to the Test **19**
9.0.1 Hand Gesture Control for Phone Functions 19

9.0.2 Drowsiness Detection 19
9.0.3 Mask Detection 19
9.0.4 Road Safety Assistance 20
9.0.5 Plant Disease Detection 20

Conclusion **21**

Preface: Bringing Intelligence to the Edge

Imagine you're driving home late at night. You're tired, and your eyelids are getting heavy. Suddenly, a subtle alert on your dashboard wakes you up. A small, inexpensive device, using a low-power camera sensor, has detected the micro-expressions and head-nod patterns associated with drowsiness. It knows this because it's running an optimized neural network trained to recognize the subtle patterns of fatigue. This isn't science fiction; it's TinyML in action.

Tiny Machine Learning (TinyML) represents a paradigm shift in artificial intelligence. It's not about building bigger models in the cloud; it's about bringing the power of AI to small, low-power devices that operate at the "edge"—close to the data source and the real-time environment. Think sensors, microcontrollers (MCUs), and other embedded systems.

The drive for edge intelligence stems from three core needs:

- **Low Latency:** Decisions must be made in milliseconds, without the delay (latency) of a round trip to a server. The drowsiness detector must react instantly, not after a few seconds of buffering.
- **Low Power:** Many edge devices run on batteries for months or years. They must perform complex tasks while consuming only milliwatts of power.
- **Privacy & Security:** By processing data locally (e.g., your camera feed or voice), sensitive information never has to leave the device, dramatically enhancing user privacy.

This book will guide you through a practical journey of building and deploying these powerful TinyML models using TensorFlow Lite (TFLite) and Google Colaboratory (Colab). Let's start the journey.

Chapter 1

TinyML - The Future is on the Edge

In a world increasingly driven by data, the bottleneck is no longer just collecting data, but processing it intelligently and efficiently. While cloud AI is powerful, it's not suitable for every application. TinyML is emerging as a game-changing technology by filling this gap, enabling real-time decision-making without relying on constant cloud connectivity.

1.1 The Core Challenge: Severe Constraints

The "tiny" in TinyML refers to the hardware constraints. We are not working with multi-gigabyte GPUs; we are working with:

- **Limited Memory (SRAM):** Often less than 512 Kilobytes. The model and all its intermediate calculations must fit within this space.
- **Limited Storage (Flash):** The model itself must be stored in flash memory, often just 1-2 Megabytes.
- **Low-Power Processors:** We use Microcontroller Units (MCUs) like the ARM Cortex-M series, which are designed for efficiency, not raw speed.

1.2 The TinyML Workflow

To overcome these constraints, we use a specialized workflow:

1. **Data Collection:** Gathering high-quality sensor data (audio, images, motion).

2. **Model Design:** Choosing or designing a neural network architecture that is small and efficient (e.g., MobileNet).
3. **Model Training:** Training the model in a high-power environment like Google Colab using TensorFlow/Keras.
4. **Model Quantization:** This is the critical step. We convert the model's 32-bit floating-point weights into 8-bit integers. This makes the model 4x smaller and significantly faster on MCUs.
5. **Conversion:** Using the TensorFlow Lite converter to create a flat-buffer ('.tflite') model file.
6. **On-Device Inference:** Deploying this file to a microcontroller and using the TFLite for Microcontrollers library to run the model.

By 2025 and beyond, proficiency in this workflow will be a key skill for engineers and developers, shaping the future of smart devices.

Chapter 2

Fault Detection - A Practical Application

One of the most practical applications of TinyML is building fault detection systems, often framed as **anomaly detection**. The goal is to identify when a system's behavior deviates from the norm.

2.1 Anomaly Detection vs. Classification

In a typical classification problem, you train a model to recognize several distinct classes (e.g., "cat," "dog," "car"). In anomaly detection, you often only have data for one class: "normal." The system's job is to recognize anything that is *not* normal.

Imagine a system monitoring the vibrations of a factory machine. You can train an autoencoder model on thousands of hours of normal vibration data. The model learns to "reconstruct" this normal data very well. When a fault occurs (e.g., a bearing starts to fail), the vibration pattern changes. The model, having never seen this pattern, will do a poor job of reconstructing it. By measuring this "reconstruction error," the system can immediately detect the deviation and trigger an alert for predictive maintenance.

2.2 Use Case: Gesture Recognition

This same principle applies to gesture recognition. You can train a system to recognize a pre-programmed set of gestures (e.g., "up," "down," "left," "right") using data from an accelerometer. This is a time-series classification problem. When the user repeats one of

those motions, the model identifies which gesture it is. This is the basis for applications like controlling music with a wave of your hand.

Chapter 3

Sensors and Microcontrollers - The Building Blocks

The foundation of any TinyML application lies in its hardware: the sensors that collect data and the microcontrollers that process it.

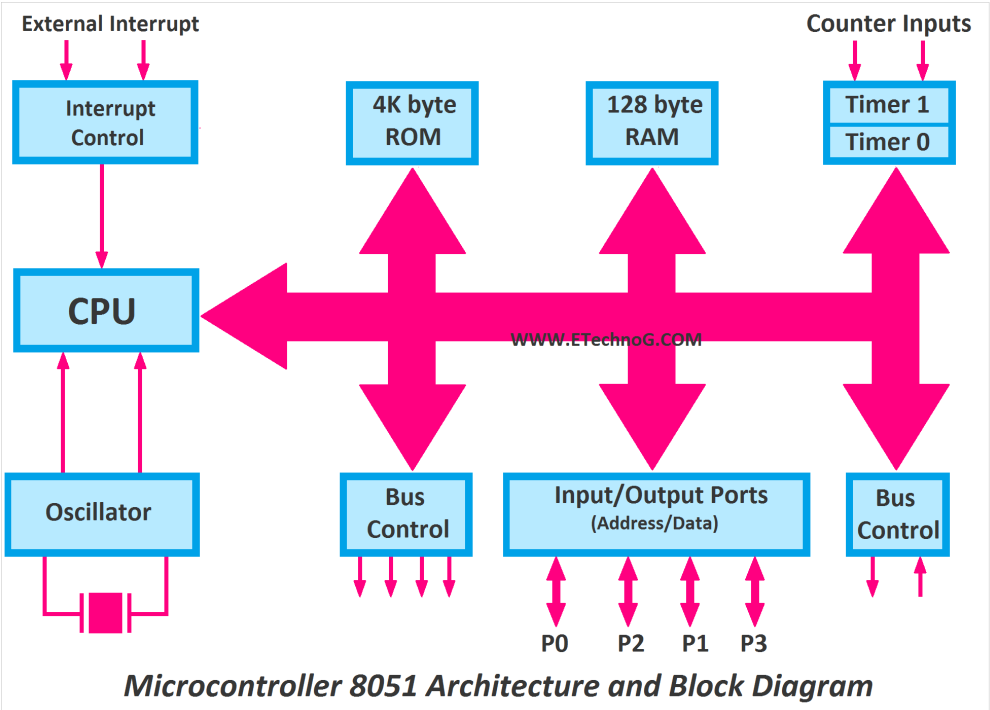

Figure 3.1: Microcontrollers

3.1 The "Senses" of TinyML: Common Sensors

- **IMU (Inertial Measurement Unit):** This sensor typically combines an accelerometer (measures linear acceleration) and a gyroscope (measures angular velocity). It's the key to understanding movement, vibration, orientation, and gestures.
- **Camera:** Low-power image sensors (e.g., OV2640) capture images and videos, enabling object detection, person detection, and image classification.
- **Microphone:** PDM or I2S microphones capture audio data. They are used for keyword spotting ("Hey Google"), sound classification (e.g., glass breaking, baby crying), and audio-based fault detection.
- **Environmental Sensors:** These sensors measure temperature, humidity, pressure, and volatile organic compounds (VOCs) for applications in smart homes, agriculture, and air quality monitoring.

3.2 The "Brain" of TinyML: Microcontrollers

The microcontroller (MCU) runs the TFLite model. The choice of MCU is critical and depends on the application's needs (power, memory, cost).

- **Arduino Nano 33 BLE Sense:** A popular beginner board packed with sensors (IMU, microphone, humidity, etc.) and a capable ARM Cortex-M4 MCU.
- **ESP32 Series:** Extremely popular for its powerful dual-core processor, low cost, and built-in Wi-Fi and Bluetooth. The ESP32-S3 has vector extensions for accelerating ML tasks.
- **Raspberry Pi Pico:** A low-cost, flexible board based on the RP2040 chip.
- **STM32 Series:** A vast family of powerful ARM Cortex-M MCUs used widely in industrial and commercial products.

Chapter 4

Object Detection with Computer Vision

Another area where TinyML shines is computer vision. While running complex vision models on an MCU was once impossible, optimized models now make it feasible.

4.1 Vision Tasks: From Simple to Complex

- **Image Classification:** Answers "What is in this image?" (e.g., "This is a can").
- **Object Detection:** Answers "What is in this image *and where*?" It draws a bounding box around each object (e.g., "There is a can at [x,y,w,h]").
- **Segmentation:** Answers "What is the exact-pixel-level-border of each object?" (Generally too complex for most MCUs).

Imagine you have a camera pointed at a conveyor belt. You can train a TinyML object detection model to identify and locate different types of objects passing by—bottles, cans, or other items. This can be used for automated sorting or quality control.

4.2 Building an Object Detection Model

The process involves several key steps:

1. **Data Collection:** Gathering hundreds or thousands of images containing the objects you want to detect, in various lighting conditions, angles, and backgrounds.
2. **Labeling:** Manually drawing bounding boxes around every object in every image and assigning a class label (e.g., "bottle"). This is often the most time-consuming part.

3. **Model Training:** Training an efficient model architecture, such as **MobileNetV2-SSDLite**, using your labeled dataset. This model is specifically designed for high speed on mobile and edge devices.

4. **Model Conversion & Quantization:** Converting the trained model to the TFLite format and applying **post-training quantization**. This step converts the model's 32-bit floating-point numbers to 8-bit integers, making it ∼4x smaller and much faster on MCUs, with only a small drop in accuracy.

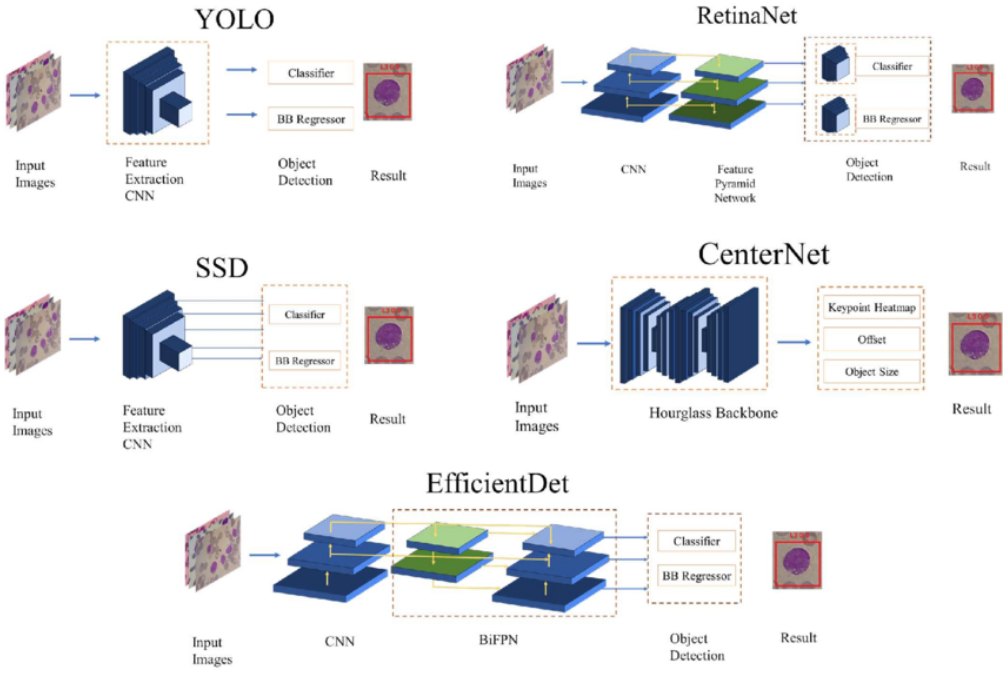

Figure 4.1: Object Detection Processing

Chapter 5

Practical Considerations: Data Quality and Model Selection

Several practical considerations can make or break a TinyML project.

5.1 The "Garbage In, Garbage Out" Principle

The quality of your training data is paramount. Clear, well-labeled images and sensor data are crucial. If your dataset is "messy" (e.g., poorly lit images, incorrect labels, or sensor data that doesn't represent the real world), your model will perform poorly, no matter how good its architecture is. **Data augmentation** (e.g., randomly rotating, scaling, or adding noise to your data) is a key pre-processing step to make your model more robust.

5.2 The Accuracy vs. Efficiency Trade-off

In TinyML, you must always balance three competing factors:

- **Model Accuracy:** How well does it perform the task?
- **Memory Footprint:** Will the model fit in the MCU's flash and SRAM?
- **Inference Speed (Latency):** Can the model run fast enough for the application?

A highly accurate model might be too large or too slow. A tiny model might be fast but not accurate enough. The art of TinyML is finding the right balance for your specific application.

It's important to remember that partial success is still progress. For example, if you are

training a model to detect three different objects and it is only able to detect one of them correctly, that itself is a valuable baseline. It proves your data pipeline, training, and deployment process are working. From there, you can iterate to improve performance on the other classes.

Chapter 6

F1 Score: Understanding Model Performance

When evaluating a model, simple "accuracy" (percent correct) can be very misleading, especially if your data is *imbalanced* (e.g., 99% "normal" data and 1% "fault" data). A model that just guesses "normal" every time would be 99% accurate but completely useless. This is where Precision, Recall, and the F1 Score become essential.

6.1 Precision: The Quality of Positive Predictions

Precision measures, **out of all the times the model predicted a positive class (e.g., "fault"), what percentage was correct?** It answers: "Is the model trustworthy when it raises an alert?"

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

A high precision means a low false positive rate.

6.2 Recall: The Completeness of Positive Predictions

Recall measures, **out of all the actual positive classes in the data, what percentage did the model find?** It answers: "Is the model finding all the faults it should?"

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

A high recall means a low false negative rate (i.e., it doesn't miss many faults).

6.3 F1 Score: The Harmonic Mean

The F1 score provides a single, balanced measure that combines precision and recall. It is the *harmonic mean* of the two, which means it heavily penalizes extreme, unbalanced values.

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

A high F1 score indicates that the model has both high precision and high recall, meaning it is both trustworthy in its predictions and finds most of the relevant cases.

Chapter 7

Real-World Examples: Where TinyML is Used

TinyML is already making a significant impact in various industries. Here are a few examples:

- **Audio & Keyword Spotting:** When you train Alexa or Google Home to recognize your voice, you're creating a tiny, personalized model. The main device is always listening (using a TinyML model) for a single "wake word" (e.g., "Alexa"). This low-power listening model runs 24/7; only when it detects the wake word does it power up the main processor to send your command to the cloud.
- **Biometric Authentication:** When you set up fingerprint recognition on your phone, you're training a TinyML model to recognize your unique fingerprint patterns. Each scan is compared against this local model.
- **Industrial IoT:** Using TinyML on an MCU with an IMU to monitor a machine for vibrations that signal an impending failure (predictive maintenance).
- **Smart Agriculture:** Low-power cameras in fields use TinyML to detect pests or identify plant diseases, so farmers only apply pesticides where needed.
- **Wearable Health:** Smartwatches use TinyML to detect falls, analyze sleep patterns, or monitor for irregular heartbeats (arrhythmia).

Chapter 8

From Training to Deployment

After completing the training process in Colab, you need to deploy your model to a microcontroller. This is where the "embedded" part of embedded systems comes in.

8.1 The TFLite Micro Model

The ‘.tflite‘ model file generated by the converter cannot be read from a filesystem on most MCUs. Instead, we use a tool (like ‘xxd‘ on Linux) to convert the binary model file into a large ‘unsigned char‘ array in a C/C++ header file. This array is compiled directly into your program's firmware and stored in the MCU's flash memory.

8.2 The Deployment Workflow

1. **Model Conversion:** Convert the ‘.tflite‘ file into a C array (e.g., ‘model.h‘).
2. **Integrate the Interpreter:** Add the ‘TensorFlow Lite for Microcontrollers‘ library to your project (e.g., as an Arduino library). This library provides the "engine" to run the model.
3. **Hardware Abstraction:** Write C/C++ code to read data from your sensor (e.g., the IMU) and place it into the model's **input tensor**.
4. **Run Inference:** Call the ‘interpreter->Invoke()‘ function. This runs the model.
5. **Post-process:** Read the prediction from the model's **output tensor** and use it to take action (e.g., light an LED, send a Bluetooth message, or trigger an alert).

You can also deploy your model using QR codes, which is a common method for deploying TFLite models to *mobile phones*, where the phone's camera app can download the model. For MCUs, the C++ array method is standard.

Chapter 9

Practical Projects: Putting Your Knowledge to the Test

Now that you have a solid foundation, here are some practical projects you can tackle:

9.0.1 Hand Gesture Control for Phone Functions

- **Goal:** Control phone functions (e.g., skip music track) by moving your device.
- **Sensor:** IMU (accelerometer).
- **Model:** Time-series classifier trained to recognize "left," "right," "up," and "down" motion patterns.

9.0.2 Drowsiness Detection

- **Goal:** Detect signs of driver drowsiness.
- **Sensor:** IMU (to detect sharp head nods) or a low-power camera (to detect eye-blink rate).
- **Model:** A classifier trained on "drowsy" vs. "alert" patterns.

9.0.3 Mask Detection

- **Goal:** Identify if a person is wearing a mask.
- **Sensor:** Camera.

- **Model:** An object detection model (like MobileNet-SSDLite) trained on "mask" and "no-mask" classes.

9.0.4 Road Safety Assistance

- **Goal:** Warn pedestrians or cyclists of approaching vehicles.
- **Sensor:** Camera.
- **Model:** An object detection model trained to recognize "pedestrian," "bicycle," and "car."

9.0.5 Plant Disease Detection

- **Goal:** Identify plant health from a picture of a leaf.
- **Sensor:** Camera.
- **Model:** An image classifier trained on "healthy_leaf," "leaf_rust," and "powdery_mildew" classes.

Conclusion

TinyML opens up a world of possibilities for creating intelligent, low-power, and private devices that can operate at the edge. The barrier to entry has never been lower, thanks to powerful tools like TensorFlow Lite and accessible hardware platforms. By understanding the fundamental concepts of model design, quantization, and deployment—and by applying the practical techniques outlined in this book—you can harness the power of TinyML to build innovative solutions for a wide range of applications. The future is not just in the cloud; it's in your hand, on your wrist, and in the environment all around you.