Hello World TinyML: Recognizing Objects with Limited Resources

28th Oct. 2025

Contents

Hello World TinyML: Recognizing Objects with Limited Resources

0.1 Introduction

Tiny Machine Learning (TinyML) is the discipline of running machine learning models on ultra-low-power devices such as microcontrollers and embedded sensors. These systems often have limited RAM (tens of kilobytes), low processing speed, and no access to cloud computing resources.

In this document, we will build a foundational understanding of TinyML using an example: recognizing simple objects like a mouse, bottle, and pen using minimal computational resources. Through this, we'll explore data preprocessing, feature extraction, model training, quantization, and deployment.

0.2 Simplifying Images: Seeing in Gray

0.2.1 Why Simplify Images?

Image data is computationally heavy. A $96 \times 96 \times 3$ RGB image has over 27,000 pixel values. This is excessive for a microcontroller. Simplifying such data allows our model to run efficiently.

0.2.2 Grayscale Conversion

A grayscale image captures brightness intensity without color. The conversion formula is:

$$Gray = 0.299R + 0.587G + 0.114B$$

This reduces the data size by one-third while preserving shape information — critical for recognizing object contours.

0.2.3 Grayscale vs Black and White

A black-and-white (binary) image uses only two levels (0 and 1), whereas grayscale allows continuous shades (0–255). Grayscale maintains richer detail, improving the model's ability to detect edges and shapes while remaining compact.

0.3 Feature Generation: Teaching the Machine to See

0.3.1 Training Sets and Labels

To train a model, we must prepare a dataset of labeled images. For example:

- Class 0 – Mouse
- Class 1 – Bottle
- Class 2 – Pen
- Class 3 – Background

Each image in the training set has a corresponding label. These serve as the "answers" during training.

0.3.2 Feature Extraction

The goal of feature extraction is to convert pixel data into meaningful patterns. Classical methods used edge detection filters like Sobel or Laplacian. TinyML often relies on lightweight neural architectures, where convolutional layers learn features automatically.

A convolution operation is given by:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

This allows the network to detect patterns like edges and corners.

0.4 Visualizing Features

Feature visualization helps us understand if the model distinguishes classes effectively. Using PCA or t-SNE, we can project high-dimensional features into 2D space.

0.4.1 Clusters and Overlaps

When plotted, similar objects form clusters. If two clusters (e.g., bottles and pens) overlap, it indicates confusion. Improving lighting, adding data, or cleaning noisy samples can help separate clusters.

0.5 Model Training and Evaluation

0.5.1 Model Architecture

A lightweight CNN architecture suitable for TinyML might be:

```
Conv2D(8 filters, 3x3) → ReLU
MaxPooling2D(2x2)
Conv2D(16 filters, 3x3) → ReLU
Flatten
Dense(32) → ReLU
Dense(4) → Softmax
```

This architecture balances accuracy and resource efficiency.

0.5.2 Training Process

Training adjusts network parameters to minimize loss. For multi-class classification, we use categorical cross-entropy:

$$Loss = -\sum_i y_i \log(\hat{y}_i)$$

Common optimizers: SGD, Adam. Hyperparameters include learning rate, batch size, and number of epochs.

0.5.3 Performance Metrics

To measure model quality, we use:

$$Precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

A higher F1 score (ideally > 0.9) indicates robust performance.

0.6 Model Quantization and Deployment

0.6.1 Why Quantization?

Trained models use 32-bit floating-point numbers. Microcontrollers typically can't handle such precision efficiently. Quantization converts weights and activations to 8-bit integers, reducing model size by $4\times$ and improving speed.

$$w_q = \text{round}\left(\frac{w_f}{S}\right)$$

where S is the scale factor.

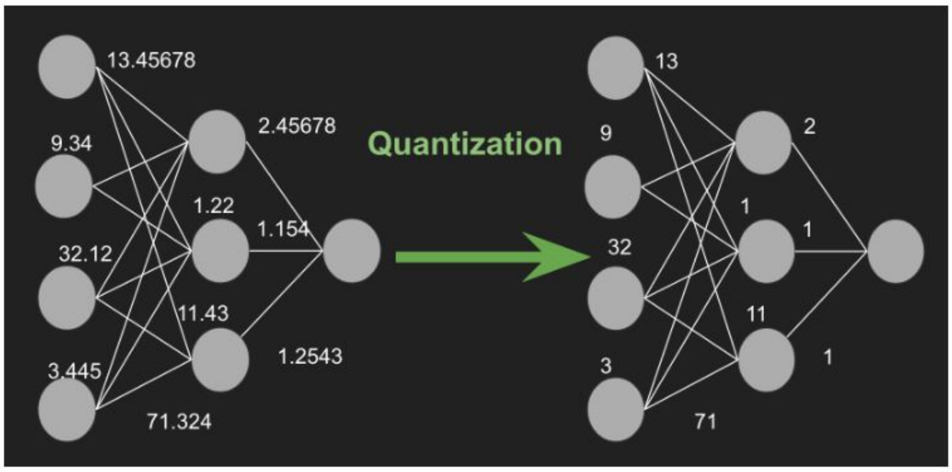

Figure 1: Quantization

0.6.2 TensorFlow Lite Workflow

Example conversion code:

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model('model')
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

This produces a compact '.tflite' model ready for deployment.

0.6.3 Deployment on Microcontrollers

We deploy using TensorFlow Lite for Microcontrollers (TFLM):

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

TFLM executes the model within a small memory "tensor arena" (typically a few KB).

0.7 Optimizing and Debugging TinyML Models

0.7.1 Optimization Techniques

- Use **depthwise separable convolutions** instead of full convolutions.
- Apply **pruning** to remove redundant weights.
- Reduce input image size or model depth.

0.7.2 Common Debugging Steps

- If accuracy is low: increase dataset diversity or adjust learning rate.
- If overfitting occurs: add dropout or L2 regularization.
- If RAM overflows: reduce batch size or quantize further.

0.8 Real-World Applications

TinyML brings AI capabilities to devices that operate offline:

- **Healthcare:** Detecting falls or heart rate anomalies.
- **Agriculture:** Pest detection, soil condition analysis.
- **Industrial IoT:** Predictive maintenance using vibration data.
- **Environment:** Smart sensors for air or noise pollution.

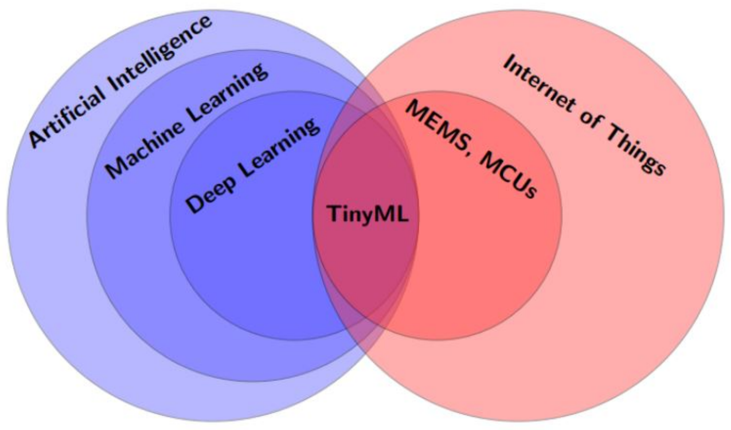

Figure 2: Where is TinyML

0.9 Conclusion

TinyML enables AI at the edge by combining efficient model design, quantization, and embedded deployment. From converting images to grayscale to deploying quantized models on microcontrollers, we can achieve real-time inference using minimal resources.

This approach makes machine learning ubiquitous — running not only in data centers but everywhere, even inside devices that fit in your pocket.