Minor in AI

TinyML

Tiny Model Development

June 03, 2025

1 Introduction to TinyML

Tiny Machine Learning (TinyML) refers to deploying machine learning models on extremely low-power devices such as microcontrollers (MCUs) with limited memory and computation resources. Unlike traditional ML applications that rely on cloud-based or high-end hardware, TinyML aims to enable on-device intelligence for use cases like sensor data monitoring, gesture recognition, and predictive maintenance.

2 Model Creation and Compression

The first stage in a TinyML pipeline is designing a compact and efficient neural network. A smaller model implies:

- Fewer parameters
- Lower memory requirements
- Faster inference on limited hardware

2.1 Compression Techniques

There are three main strategies to reduce model size:

1. **Pruning** involves removing unnecessary weights or neurons from the network. These are typically the weights with the smallest magnitudes, as they contribute the least to the final prediction. Types of pruning include:
    - **Weight Pruning:** Eliminating individual low-magnitude weights.
    - **Neuron Pruning:** Removing entire neurons (and associated weights) that minimally affect performance.
    - **Structured Pruning:** Removing groups of neurons or channels for optimized hardware acceleration.
2. **Quantization** transforms 32-bit floating-point weights and activations to lower precision formats, such as 8-bit integers (int8). This reduces both memory usage and computational overhead.
3. **Knowledge Distillation** transfers knowledge from a large, well-trained model (teacher) to a smaller model (student), maintaining accuracy while reducing size.

3 Neural Network Implementation: `Hello World`

To illustrate the basics of neural networks, the session implemented a simple model that learned to approximate the sine function.

3.1 Model Architecture

The architecture consists of:

- A single input representing the x-value
- A hidden layer (e.g., with 16 neurons and ReLU activation)
- An output layer yielding the predicted $\sin(x)$

The forward pass of the network can be described as:

$$h = \text{ReLU}(W_1 x + b_1), \quad \hat{y} = W_2 h + b_2$$

where W_1, W_2 are weights and b_1, b_2 are biases.

Training Objective: Minimize the Mean Squared Error (MSE) between predicted and actual sine values.

Learning Outcome: Students gained hands-on experience with defining, training, evaluating, and visualizing a compact regression model.

4 Hardware Constraints in Microcontrollers

Microcontrollers typically offer:

- RAM: 2KB to 256KB
- Flash: Up to 1MB
- Clock speed: 16–200MHz

These limitations make it infeasible to deploy large models. Thus, optimizing model size is critical.

4.1 TF Lite vs Manual Implementation

TensorFlow Lite for Microcontrollers (TFLM): Allows deploying converted models directly to embedded devices.

Manual Implementation: Offers finer control by translating weights and operations directly into C/C++ arrays and logic, bypassing interpreter overhead.

U0001F4A1 Insight:

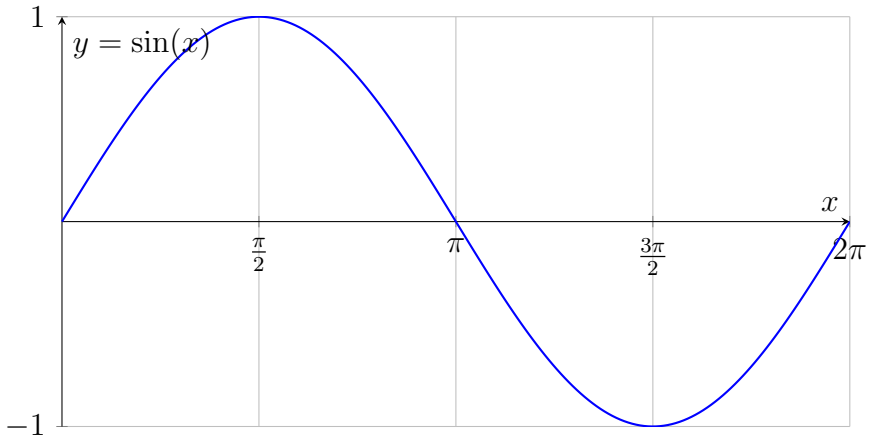
MicroPython is easier to prototype with, but C/C++ offers better speed and memory efficiency in production environments.

5 Sine Wave Prediction: A Case Study

Let us dive deeper into the practical implementation of sine wave prediction using a neural network:

5.1 Graph

The sine function is a periodic, smooth function commonly used in signal processing, mathematics, and neural network training examples.



This graph shows one complete cycle of the sine wave from 0 to 2π.

5.2 Model Design

A fully connected neural network was designed with one hidden layer to approximate the function $\sin(x)$.

5.3 Training and Testing

The model was trained on uniformly sampled values of x over an interval, with $\sin(x)$ as the target output. The students observed the learning progression and evaluated the performance visually.

5.4 Conversion and Deployment

After training, the model was converted to the TensorFlow Lite format. It was then compiled for deployment on a microcontroller where it inferred sine values for new inputs.

5.5 Reflection

This example illustrates the end-to-end lifecycle of a TinyML model: from training to quantization to deployment.

6 Quantization and Model Size Optimization

6.1 Float32 vs Int8 Models

In many embedded systems, using 32-bit float operations is expensive. Converting model parameters to 8-bit integers greatly reduces:

- Storage (4x reduction in size)

- Inference time
- Power consumption

Conversion Pipeline:

1. Train full precision (float32) model
2. Use TensorFlow's post-training quantization API
3. Convert to int8 using representative datasets

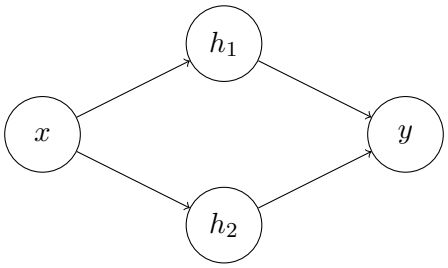
6.2 Manual Weight and Bias Integration

For ultimate control, weights and biases can be exported and embedded into C/C++ code directly:

```
const int8_t weights[16] = { ... };
const int8_t bias[16] = { ... };
int output = relu(dot_product(input, weights) + bias);
```

This technique eliminates all external dependencies and allows you to fine-tune memory and performance manually.

7 Visualizing a Small Neural Network



Key Takeaways

1. TinyML allows ML model deployment on microcontrollers with limited memory and compute power.
2. Compression techniques such as pruning, quantization, and knowledge distillation are essential to fit models on edge devices.
3. Simple neural network architectures can be used to approximate functions like sine waves.
4. Hardware constraints necessitate careful design and optimization, favoring int8 quantization and manual implementations.
5. TF Lite simplifies deployment but manual C/C++ implementations offer greater control and efficiency.

Questions to Ponder

1. Why is it necessary to use quantization in TinyML applications?
2. What trade-offs exist when choosing manual implementation over TF Lite?
3. Could pruning hurt accuracy? When would you avoid pruning?
4. Train a small model on a different function (e.g., cosine), quantize it, and manually export the weights. Write inference logic in C and benchmark execution speed.