TinyML: Use Cases and Concepts

November 5, 2025

1 Introduction to TinyML

Tiny Machine Learning (TinyML) is a field of machine learning focused on deploying and running ML models on extremely resource-constrained devices, primarily microcontrollers (MCUs). These devices operate under severe limitations:

- **Memory:** Typically in the range of kilobytes (kB) to a few megabytes (MB). For example, an Arduino Nano 33 BLE Sense has only 256 kB of RAM and 1MB of Flash storage.
- **Compute:** Low-power processors (e.g., ARM Cortex-M series) with clock speeds often below 100 MHz and lacking advanced instruction sets or a Floating Point Unit (FPU).
- **Energy:** Designed to run for months or even years on a small battery, with power budgets measured in milliwatts (mW) or even microwatts (μW).

TinyML enables on-device sensor data processing and intelligence, often called "edge intelligence." This approach provides significant benefits over traditional cloud-based ML:

- **Low Latency:** No network round-trip is required. Inference happens in milliseconds, which is critical for real-time applications like collision avoidance or activity recognition.
- **Low Bandwidth:** Only high-level insights (e.g., "anomaly detected") are sent to the cloud, not raw sensor data. This saves costs and is ideal for remote areas with poor connectivity.
- **Privacy & Security:** Sensitive data (like audio from a home or biometric data from a wearable) never leaves the device, drastically reducing privacy risks.
- **Low Power:** On-device processing avoids the high energy cost of Wi-Fi or cellular data transmission, enabling battery-powered applications.

Applications of TinyML are vast and growing, including predictive maintenance in industrial settings, smart agriculture (monitoring soil and crop health), keyword spotting in smart assistants, wearable health devices (ECG monitoring, fall detection), and intelligent wildlife tracking.

2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep neural networks that are particularly effective for analyzing visual imagery. They are inspired by the organization of the animal visual cortex and are designed to automatically and adaptively learn spatial hierarchies of features.

Unlike a standard Fully Connected Network (FCN), where every neuron connects to every neuron in the previous layer, CNNs use two key principles:

- **Local Receptive Fields:** Each neuron in a convolutional layer only responds to a small, local region of the input.
- **Parameter (Weight) Sharing:** The same set of weights (a "kernel" or "filter") is applied across the entire input. This allows the network to detect the same feature (e.g., a vertical edge) regardless of its position in the image, a property known as **translation invariance**.

This architecture makes CNNs far more parameter-efficient than FCNs for tasks like image processing, making them a feasible (though challenging) choice for TinyML.

2.1 Key Concepts

- **Hierarchical Feature Extraction:** CNNs learn features in a hierarchy. Early layers detect simple features like edges and corners. Deeper layers combine these to detect more complex features like shapes, textures, and eventually object parts.
- **End-to-End Feature Learning:** CNNs do not require manual feature engineering (e.g., manually defining SIFT or HOG features). Given enough data, the network learns the optimal features for the task directly from the raw pixel data.
- **Classification:** After feature extraction, the final high-level features are typically passed to one or more fully connected layers, which act as a classifier. A **softmax** activation function is often used in the final layer to output a probability distribution over the possible classes.

CNNs consistently outperform traditional ML for vision-based tasks such as object detection, gesture recognition, and face detection. In the TinyML context, they are also adapted for 1D data, such as audio (for keyword spotting) and time-series data (for activity recognition).

2.2 Real-world Examples

- **Visual Wake Words:** A low-power camera on a smart home device runs a tiny CNN to detect the presence of a person. It only "wakes up" the main, power-hungry processor when a person is detected, saving significant energy.
- **Keyword Spotting:** Smart speakers and phones use a small 1D CNN (or a similar model) to constantly listen for a "wake word" (e.g., "Hey Google"). This model runs entirely on a low-power digital signal processor (DSP).

3 Stages of a CNN

A typical CNN architecture is a sequence of layers. The main building blocks are:

3.1 Input Layer

This layer holds the raw pixel data of the image. Its dimensions are typically $height \times width \times channels$.

- For a **grayscale** image, $channels = 1$.
- For a **color (RGB)** image, $channels = 3$.

3.2 Convolution (Conv) Layer

This is the core building block. It applies a set of learnable **kernels** (or filters) to the input.

- **Kernel:** A small matrix of weights (e.g., 3×3 or 5×5).
- **Operation:** The kernel "slides" (convolves) across the input image. At each position, it computes the element-wise multiplication and sum (a **dot product**) between the kernel weights and the input pixels it covers.
- **Feature Map:** The 2D output of this operation is called a **feature map** or **activation map**. It highlights the areas of the image where the feature (defined by the kernel) was detected.
- **Parameters:** Two key hyperparameters are:
  - **Stride:** The number of pixels the kernel jumps as it slides. A stride of 1 moves pixel-by-pixel. A stride of 2 skips every other pixel.
  - **Padding:** Adding zeros around the border of the input. "Same" padding ensures the output feature map has the same $height \times width$ as the input, while "valid" padding (no padding) results in a smaller output.

- **Activation:** After the convolution, a bias is added, and the result is passed through a non-linear **activation function**, most commonly the Rectified Linear Unit (**ReLU**), defined as $f(x) = \max(0, x)$. This introduces non-linearity, allowing the network to learn complex patterns.

The mathematical operation for a single output value is:

$$Output = f((\sum(Input \cdot Kernel)) + bias) \tag{1}$$

where f is the activation function (e.g., ReLU).

3.3 Pooling (Subsampling) Layer

The pooling layer's main purpose is to **downsample** the feature maps, reducing their spatial dimensions (*height* and *width*).

- **Benefit 1 (Computation):** Smaller feature maps require less computation in subsequent layers.
- **Benefit 2 (Invariance):** It provides a small amount of translation invariance, making the network more robust to the exact position of features.
- **Max Pooling:** The most common form. It takes a window (e.g., 2×2) and outputs only the *maximum* value from that window.
- **Average Pooling:** An alternative that outputs the *average* value from the window.

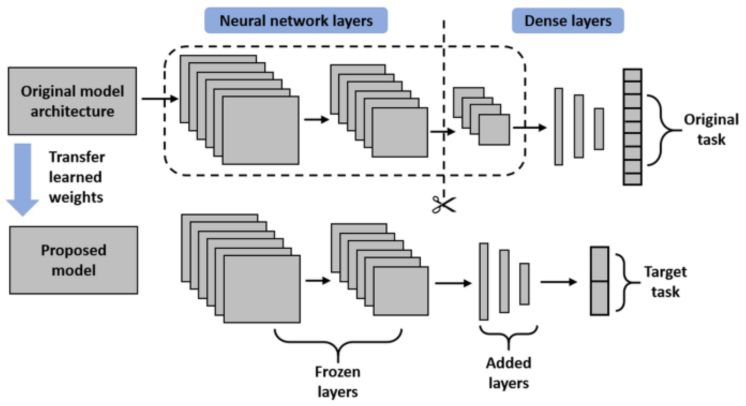

Figure 1: Architecture

3.4 Flattening

After the final convolution and pooling layers, the resulting 3D feature maps (e.g., $7 \times 7 \times 256$) are unrolled into a single, long 1D vector. This is done to transition from the feature extraction part of the network to the classification part.

3.5 Fully Connected (FC) Layer

Also known as a "Dense" layer, this is a traditional Multi-Layer Perceptron (MLP). Every neuron in the FC layer is connected to every activation in the previous (flattened) layer. Its job is to perform high-level reasoning and combine the extracted features to make a final classification decision. The last FC layer uses a **softmax** activation for multi-class classification, outputting a probability for each class.

4 Transfer Learning for TinyML

Transfer learning is a technique where a model trained on a large, general-purpose dataset (e.g., ImageNet, with 1.4 million images) is adapted for a new, specific task. This is **critical** for TinyML for two reasons:

1. TinyML datasets are often small and specialized (e.g., "my voice" or "vibrations of this machine"). Training a deep CNN from scratch on such little data would lead to severe overfitting.
2. Training large models is computationally expensive and impossible to do on the target microcontroller.

The pre-trained model acts as an excellent feature extractor.

4.1 Steps

1. **Select Pre-trained Model:** Choose a model that is already optimized for efficiency, such as **MobileNetV2** or **EfficientNetLite**. These models use efficient blocks like *depthwise separable convolutions*.
2. **Remove Top Layer:** Remove the original, task-specific classification layer (e.g., the 1000-class ImageNet classifier).
3. **Add Custom Layer:** Add a new, small classification head (one or two FC layers) tailored to your specific task (e.g., 3 classes: "person," "car," "background").
4. **Freeze Base Layers (Feature Extraction):** Initially, "freeze" the weights of all the original convolutional layers. This prevents them from being updated during training. Only the weights of your new custom layers are trained.

5. **Fine-Tune (Optional):** After the new head is trained, you can "unfreeze" the top few layers of the base model and retrain the entire network at a very low learning rate. This "fine-tunes" the high-level features to be more specific to your new dataset, often yielding a boost in accuracy.

5 Model Optimization Techniques

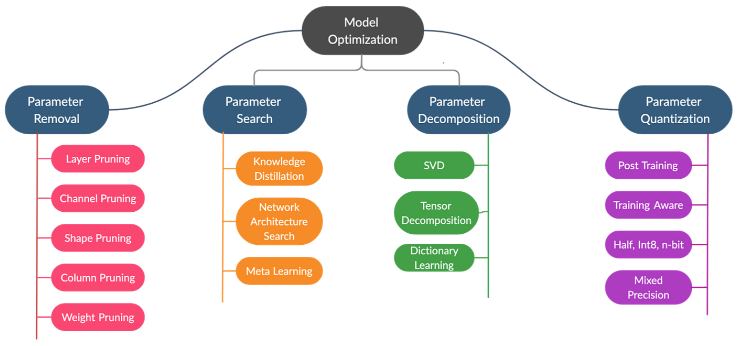

Figure 2:

To fit a trained model into a few hundred kilobytes, optimisation is not optional—it's essential.

5.1 Quantization

Quantization is the process of reducing the numerical precision of a model's weights and, optionally, its activations.

- **The Problem:** Standard models use 32-bit floating-point numbers (**Float32**) for all weights and biases.
- **The Solution:** Convert these numbers to 8-bit signed integers (**Int8**).
- **Benefits:**
  - **Size:** $\approx 4\times$ reduction in model size (8 bits vs. 32 bits).
  - **Speed:** Integer arithmetic is significantly faster and more energy-efficient on MCUs, especially those without an FPU.
  - **Memory:** Activations (the outputs of intermediate layers) can also be stored as Int8, reducing peak RAM usage.
- **Methods:**
  - **Post-Training Quantization:** The easiest method. A trained Float32 model is converted to Int8 after training. It requires a small "calibration" dataset to determine the dynamic range of activations.

  - **Quantization-Aware Training (QAT):** Simulates Int8 quantization during the training process. This allows the model to adapt to the precision loss, resulting in much higher accuracy for the final quantized model.

5.2 Pruning

Pruning involves identifying and removing individual weights or even entire neurons that are "unimportant" to the model's predictions.

- **The Idea:** In a large network, many weights are very close to zero and contribute little.
- **The Process:** These low-magnitude weights are set to zero, creating a **sparse** model. This model is then typically retrained for a few epochs to let the remaining weights "compensate" for the pruned ones, recovering lost accuracy.
- **Benefits:** Can significantly reduce model size (when compressed) and, on specialized hardware or libraries that support sparse operations, can also speed up inference.

A common approach is **hybrid optimization**, where pruning is applied first, followed by quantization, to achieve maximum model compression.

5.3 Knowledge Distillation

This technique involves training a small "student" model (the one for the MCU) to mimic the behavior of a much larger, high-performance "teacher" model.

- **Process:** The student model is trained not just to match the final labels (e.g., "cat"), but to match the full probability distribution (the "soft labels") produced by the teacher (e.g., "95% cat, 4% dog, 1% truck").
- **Benefit:** The student learns the more nuanced "dark knowledge" from the teacher, achieving higher accuracy than if it were trained on the hard labels alone.

6 Deploying TinyML Models

The most common framework for deploying models on MCUs is **TensorFlow Lite for Microcontrollers (TFLM)**.

6.1 Workflow

1. **Train and Optimize Model:** Train a model in a standard framework like TensorFlow/Keras. Apply optimizations like Quantization-Aware Training.

2. **Convert to TFLite Format:** Use the TensorFlow Lite Converter tool to convert the trained model into a ‘.tflite‘ flat-buffer file. This step applies post-training quantization if not already done.

3. **Export as C Header File:** Use a command-line tool (like ‘xxd‘) to convert the binary ‘.tflite‘ file into a ‘const unsigned char[]‘ array inside a C/C++ header file (‘.h‘).

4. **Integrate and Flash:**

    - In your C/C++ microcontroller project (e.g., in the Arduino IDE), include this header file. This compiles the model weights *directly into the program's binary.*
    - Include the TFLM library, which provides the C++ *interpreter* to run the model.
    - Write code to load the model from the array, allocate memory for input/output tensors (the "Tensor Arena"), run inference, and interpret the results.
    - Flash the final compiled application to the microcontroller's flash memory.

6.2 Example Use Case: Human Activity Recognition (HAR)

Using accelerometer and gyroscope data from an IMU (Inertial Measurement Unit) sensor, a wearable device runs a 1D CNN or RNN.

- **Input:** A "window" of sensor data (e.g., 2 seconds of 3-axis accelerometer data).
- **Output:** A classification of the user's current motion (e.g., "walking," "sitting," "running," "fall detected").
- **Benefit:** This runs entirely on the device, enabling real-time fall detection alerts or activity tracking without needing a smartphone connection.

6.3 Example Use Case: Predictive Maintenance

A small microcontroller with a vibration sensor (accelerometer) is attached to an industrial motor.

- **Input:** A window of vibration data.
- **Output:** A classification of the machine's state (e.g., "normal," "bearing wear," "imbalance").
- **Benefit:** The system can detect anomalies that predict a failure *before* it happens, allowing for maintenance to be scheduled proactively, all without streaming continuous vibration data to the cloud.

7 Conclusion

TinyML represents a significant shift in computing, moving intelligence from the cloud to the extreme edge. It makes AI ubiquitous by enabling low-power, low-cost, and privacy-preserving solutions. By understanding the fundamentals of efficient architectures like CNNs, mastering optimization techniques like quantization and pruning, and leveraging frameworks like TFLM, developers can build the next generation of intelligent edge devices.

Future challenges and opportunities include developing more efficient model architectures, enabling on-device (or federated) learning to adapt models in the field, and creating hardware specifically co-designed for TinyML workloads.