Practical TinyML

A Hands-On Guide

1 Hello, Tiny World!

Imagine you're teaching a robot to recognize your voice. You could load a massive, sophisticated AI model onto it, requiring a powerful and expensive computer just to understand simple commands. But what if you wanted this voice-controlled robot to be small, energy-efficient, and cheap – something you could easily build and deploy?

That's where Tiny Machine Learning (TinyML) comes in. TinyML is all about shrinking machine learning models down to a size where they can run on tiny, low-power devices called microcontrollers. These are the brains inside everyday objects like smartwatches, appliances, and IoT sensors. TinyML enables machine learning inference on devices with severe resource constraints, typically less than 1 MB of RAM and operating at under 1 mW of power. This allows for applications in edge computing, where data processing happens locally without relying on cloud servers, improving privacy, latency, and energy efficiency. Common use cases include wake-word detection, gesture recognition, anomaly detection in sensors, and predictive maintenance in industrial IoT.

Instead of a complex voice recognition system, think about something simpler, like predicting a sine wave. That might seem abstract, but it demonstrates the core principle: getting a small, efficient model to learn a pattern and make predictions. The sine wave prediction serves as the "Hello World" of TinyML because it's a straightforward regression task that illustrates the end-to-end workflow—from data preparation and model training to deployment and inference—while fitting within the constraints of microcontrollers. It helps verify that the model can approximate a known mathematical function, providing a controlled environment to test performance before tackling real-world problems. We'll learn how to do this step by step.

Just like a programmer's first program is often "Hello, World!", we're going to create a "Hello, Tiny World!" experience by building a small neural network and training it to predict a sine wave. Let's dive in!

2 Tiny Models: Making Machine Learning Small

TinyML is about making machine learning models that can fit and run on very small devices. These devices, powered by microcontrollers, often have limited memory (RAM). Unlike large servers or even your smartphone, these devices might only have a few hundred kilobytes of RAM. So, the size of the models we use *really* matters. Microcontrollers like those based on Arm Cortex-M series are common in TinyML, offering 32-bit processing at low cost (under $1) and power consumption.

This means we need to consider techniques to compress and optimize models. In essence, we are aiming to achieve the same level of performance from a much smaller device. Model compression is crucial for TinyML to reduce memory footprint and computational requirements. Key techniques include quantization (reducing precision from 32-bit floats to 8-bit integers), pruning (removing less important weights), and knowledge distillation (training a smaller model to mimic a larger one). These methods can shrink models by 4x to 100x while maintaining acceptable accuracy, enabling deployment on devices with kilobytes of RAM.

3 Our Tiny Sine Wave Prediction Model

Instead of just printing "Hello, World!" to the screen, we are trying to have some experience with a smaller model. So, we aim to build a neural network to generate y values for given x values.

We'll be using the sine function: $y = \sin(x)$. This is a regression problem where the model learns to approximate the continuous sine wave, which is periodic and nonlinear, making it a good test for neural networks' ability to capture patterns.

Our goal is to create an *untrained* neural network, feed it with x values, train it with our sine function, and, through training, make it generate the corresponding y values. In practice, this involves using frameworks like TensorFlow and Keras to define the model, then training it on a dataset of x-y pairs.

Here's how it works:

1. We input an x value into the neural network.
2. The neural network generates a predicted y value, which we'll call y'.

Figure 1: Image of a sine wave

3. We compare y' to the actual y value (calculated using the sine function).
4. The neural network then adjusts its internal parameters (weights) to minimize the difference between y' and y.

We can repeat this process many times to train our model. During training, backpropagation updates weights using an optimizer like RMSprop and a loss function such as mean squared error (MSE), which measures the average squared difference between predictions and actual values.

For example, a basic code snippet in Python using TensorFlow might look like this:

```
import tensorflow as tf
import numpy as np

# Generate data
x_values = np.random.uniform(0, 2*np.pi, 1000)
y_values = np.sin(x_values)

# Define model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, activation='relu', input_shape=(1,)),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='rmsprop', loss='mse')
model.fit(x_values, y_values, epochs=500)
```

This trains a simple model to predict sine values.

4 Neural Network Size and Structure

Our microcontrollers come with RAMs in the range of 100-300KB, so the size of the neural network really matters. A larger neural network will typically have more parameters (weights and biases) and require more memory to store and compute. For TinyML, models must be designed with minimal parameters to fit in limited flash and RAM, often under 100 KB total.

Our network will have the following structure:

- **Input Layer:** Takes one input value (x).
- **Hidden Layer 1:** A "dense" layer with 16 neurons.
- **Hidden Layer 2:** A "dense" layer with 16 neurons.

- **Output Layer:** Outputs one value (y').

Here, "dense" simply means each neuron is connected to every neuron in the previous layer. We need to be careful to keep our layers small, to keep the model size down. This architecture typically results in around 321 parameters (weights and biases). The ReLU (Rectified Linear Unit) activation function is used in hidden layers to introduce nonlinearity: $\text{ReLU}(z) = \max(0, z)$. Without nonlinearity, the network would only learn linear functions, incapable of approximating sine. For sine prediction, two hidden layers provide sufficient capacity; more layers could lead to overfitting or increased size.

The total parameters can be calculated as: Input to Hidden1: $1 \times 16 + 16 = 32$, Hidden1 to Hidden2: $16 \times 16 + 16 = 272$, Hidden2 to Output: $16 \times 1 + 1 = 17$, summing to 321.

5 Data Preparation: Introducing Noise

In the real world, data is rarely perfect. Input data is often noisy, that is, it is corrupted by errors. To prepare our data for TinyML, we'll start with our $y = \sin(x)$ function, and will add some noise to it. We can introduce noise by adding a uniformly distributed random number ranging from 0 to 2π to the sine of that random number. More commonly, Gaussian noise is added to simulate sensor inaccuracies: $y = \sin(x) + \mathcal{N}(0, \sigma^2)$, where σ is the standard deviation (e.g., 0.1). This helps the model learn robust features and prevents overfitting to perfect data.

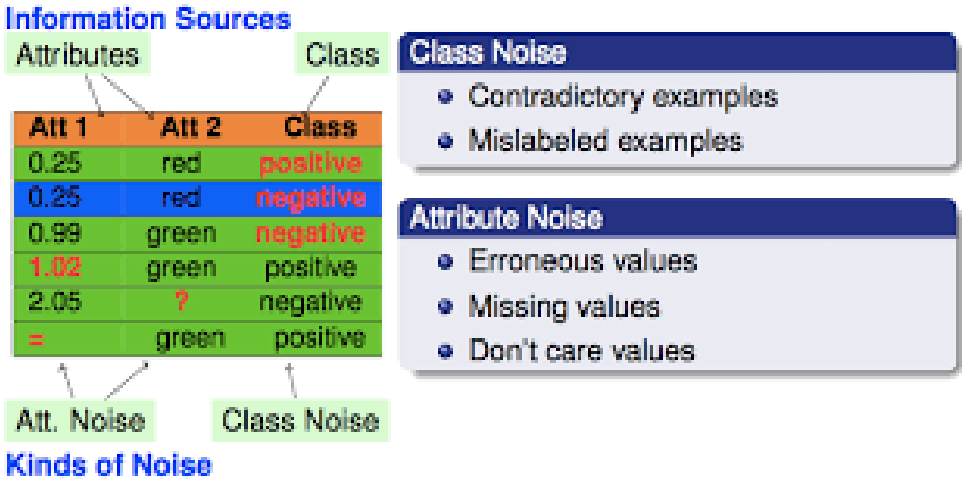

Figure 2: Image of a sine wave

Then, we'll split this data into three sets:

- **Training Set:** Used to train the neural network.
- **Testing Set:** Used to evaluate the performance of the trained network.
- **Validation Set:** Used to tune the hyper-parameters of the model, while training.

A typical split might be 70% for training, 15% for testing, and 15% for validation. This ensures that the network isn't just memorizing the training data but can generalize to new, unseen data. In practice, for 1000 samples, this could be 600 train, 200 validate, 200 test. Data should be shuffled to avoid order bias.

Example code for data preparation:

```
import numpy as np
import matplotlib.pyplot as plt

SAMPLES = 1000
x_values = np.random.uniform(low=0, high=2*np.pi, size=SAMPLES).astype(np.float32)
np.random.shuffle(x_values)
y_values = np.sin(x_values).astype(np.float32)
y_values += 0.1 * np.random.randn(*y_values.shape)

plt.plot(x_values, y_values, 'b.')
plt.show()
```

This generates and visualizes noisy data.

6 Model Training and Compression

Training a neural network involves feeding it with data, calculating the errors, and adjusting the internal parameters. Usually, the weights and biases of the models are floating-point numbers, typically using 32 or 64 bits of precision. Training uses backpropagation to minimize loss over epochs (full passes through data), with batch sizes (e.g., 16) for efficiency. For sine prediction, mean squared error (MSE) is common: $\text{MSE} = \frac{1}{n}\sum(y' - y)^2$. Monitor validation loss to detect overfitting.

To compress the model, we can perform quantization and pruning. Quantization can convert floating-point numbers to fixed-point numbers or convert the model to smaller integer numbers, and Pruning can remove various edges and neurons to reduce the network complexity. The aim is to reduce the model size, even at the cost of losing precision. Quantization (e.g., post-training to 8-bit) can reduce size by 4x with minimal accuracy drop, using representative datasets for calibration. Pruning removes weights below a threshold, leading to sparse models (up to 10-100x compression), though hardware support for sparsity varies. Other techniques include knowledge distillation and low-rank approximation.

In TensorFlow, quantization is applied during conversion:

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

This produces a compressed .tflite file.

7 Deployment Options

After training, we need to deploy our model to a microcontroller. We'll have three options to go about this:

- **Using TensorFlow Lite Micro with a .hex file:** The first way is by taking the help of external library, like a link to your tiny m l, that will take input in terms of TensorFlow micro. This generates a .h file containing a .hex file, which has the model loaded into it, and then interpreted by LNQT. More precisely, convert the .tflite to a C array using xxd: xxd −i model.tflite > model.h, then include in C++ code for TensorFlow Lite Micro interpreter.
- **LNQ Tiny with Micro Models:** Here, the micro model file is loaded into a header file, number of inputs, number of outputs, arena size, object creation, and other variables, are passed in. This will read the hex file and load the model. Alternatives like Edge Impulse or SensiML provide platforms for easier deployment.
- **Driver program with the C programming language:** A driver program written in C will perform matrix multiplication addition, then radio activation before that add the bias, bias and propagate to the next layer. This is a from-scratch implementation without libraries, useful for understanding but less efficient.

Common platforms include Arduino Nano 33 BLE Sense, SparkFun Edge, and STM32 boards. Deployment uses tools like Arduino IDE, Make, or Mbed CLI to build and flash binaries. For example, on Arduino: Install TensorFlow Lite library, upload sketch with model header. On STM32: Use Cube.AI or TensorFlow Lite Micro with Makefile builds.

8 Inside the Black Box: Weights, Biases, and Inference

Even though our Sine Wave prediction model is super simple, we can write our own driver program and the Python program or the colab environment, which will then generate the Wilson biases, and we can take those biases and process time input, and then discover the output with this driver program. In total, there are 321 parameters. If we store these as 32 bit floating point number, then they will take up to 1284 bytes.

Then, we need our own inference logic. So, the model needs to perform these the different tasks for inference. So, first one we have to do is scalar vector multiplication for the first layer. Scalars value

versus vector. One value has to be multiplied by across the all the values in the vector. Then what we have to have is we have to have vector matrix multiplication for the middle layer. So, we have a vector and we have matrix. So, we have to do the multiplication for that day. Now, we have to have the dot product for the last layer.

Finally, addition of vector for the addition means like once you multiply as take a weighted sum, then you have to add the bias value to it. And then you will pass it to the relu. So, that thing has to happen at the end. So, once I lose 10, it will be propagated to the next step. So, then all the steps will be also again repeated.

In TinyML, inference on microcontrollers uses a lightweight interpreter like TensorFlow Lite Micro. It allocates a tensor arena (e.g., 2KB buffer) for activations, loads weights/biases (constant in flash), sets input tensor, invokes operations (matrix multiplies, adds, ReLU), and reads output. Weights are matrices connecting layers; biases are added post-multiplication. For efficiency, use CMSIS-NN for Arm MCUs to accelerate dot products. Manual implementation involves loops for $z = W \cdot x + b$, then ReLU(z).

Example C++ inference snippet:

```
TfLiteTensor* input = interpreter.input(0);
input->data.f[0] = x_value;
interpreter.Invoke();
TfLiteTensor* output = interpreter.output(0);
float y_value = output->data.f[0];
```

This runs the forward pass.

9 Beyond Sine: Exploring Other Functions

Instead of $y = \sin(x)$, we can try other trigonometric functions like the tangent function: $y = \tan(x)$. But, the tangent function has a vertical asymptote, that is, it tends towards infinity at some points. Can we train our machine learning models with the tangent function? This is left as an exercise for the reader. Training on tan(x) is challenging due to discontinuities and unbounded values near asymptotes (e.g., at $\pi/2$), requiring data clipping or specialized handling to avoid exploding gradients.

Other functions for TinyML exploration include cosine ($y = \cos(x)$), which is similar to sine but phase-shifted; exponential ($y = e^x$), testing growth patterns; or polynomials like $y = x^2$. Real-world extensions include time-series prediction, such as approximating sensor waveforms (e.g., audio signals for wake words) or periodic sensor data (e.g., vibration monitoring). Advanced examples: Kolmogorov-Arnold Networks for function approximation, or fuzzy logic for anomaly detection in road data.

10 Final Thoughts

By building a small neural network and training it to predict a sine wave, we took our first step in the practical realm of TinyML. We can compress the model, deploy it, and perform inferences. As we build and deploy these models, we need to ask ourselves:

- Can we train a smaller model for the same levels of performance? (E.g., reduce neurons or use pruning.)
- Can we get better performance, by adding more training data? (More samples improve generalization, but watch for diminishing returns.)
- Are our assumptions right? Are the models behaving as we are thinking? (Validate with metrics like MAE, check for overfitting via loss curves.)

Additionally, consider power consumption during inference, hardware-specific optimizations (e.g., using DSP instructions), and ethical aspects like data privacy in edge devices. Experiment with tools like Edge Impulse for end-to-end workflows, and explore applications beyond toys, such as environmental monitoring or health wearables.