# EML

## Introduction

Embedded Machine Learning (ML) refers to the integration of machine learning models into embedded systems. These are typically resource-constrained devices such as microcontrollers, Internet of Things (IoT) devices, edge devices, and even wearables. Embedded ML is an exciting frontier that brings the power of machine learning to the edge, right where the data is generated. As advancements in hardware and software continue, the capabilities of embedded ML are expected to grow, making it an essential tool for modern embedded systems development.

### Motivation:

* Traditional cloud-based machine learning solutions require data to be sent to the cloud for inference. This can lead to latency, require a continuous internet connection, and pose privacy concerns. Embedded ML solves these problems by processing data locally and performing inference directly on the device.

### Applications:

ML inference on TinyML systems has facilitated the development of technologies in low-power devices such as wakeword detection (Gruenstein et al., 2017), predictive maintenance (Susto et al., 2015), anomaly detection (Koizumi et al., 2019), visual object detection (Chowdhery et al., 2019), and human activity recognition (Chavarriaga et al., 2013).[[1]](#footnote-1)

* IoT Devices: For example, a smart thermostat might use ML to predict when to adjust temperatures based on user behavior.
* Wearables: Health monitoring devices can predict abnormal heart rhythms.
* Smart Cameras: Can process images and videos for object detection, face recognition, etc.
* Voice Assistants: Local voice recognition on devices without requiring a cloud connection.

### Challenges:

* Resource Constraints: Embedded systems often have limited memory, storage, and computational power. This means traditional, large-scale ML models are not suitable. Instead, specialized models and optimization techniques are required to fit the constraints.
* TinyML: This is a term frequently associated with embedded ML, emphasizing the development of machine learning models that can run on extremely constrained devices, typically with memory measured in kilobytes (KB) rather than megabytes (MB) or gigabytes (GB).
* Power Efficiency: Since many embedded devices are battery-powered, ML models need to be optimized not just for size and speed but also for power efficiency.
* Lifecycle Management: Deploying updates and maintaining models on a vast array of devices can be challenging.
* Data Skew: If a model is trained on data from one set of devices and deployed on another, the variations between devices can lead to performance issues.
* Hardware Heterogeneity: Despite resource limitations being fairly constant across TinyML hardware, the embedded computing systems themselves are quite diverse. Each hardware platform supports different deployment processes, model types, numerical formats, and memory access patterns, which often makes TinyML applications difficult to port across devices. This complexity is exacerbated when a creating an application at scale, which must be deployed to a wide variety of devices, each with their own libraries and deployment method.

## Frameworks and tools

### TensorFlow Lite for Microcontrollers

A version of TensorFlow designed to run on microcontrollers.

Pros:

* Integration with TensorFlow: If you're already using TensorFlow, transitioning to TensorFlow Lite for microcontrollers is easier.
* Supported Operations: Contains optimized operations for many popular neural network layers.
* Community Support: TensorFlow's large community means more tutorials, forums, and other resources are available.
* Versatility: Designed to support multiple platforms including various microcontrollers and edge devices.

Cons:

* Memory Consumption: Despite being lightweight, it might still be too large for extremely constrained devices.
* Model Conversion: Models sometimes need modification to be fully compatible with TensorFlow Lite.

### ONNX (Open Neural Network Exchange)

An open standard for representing ML models, which allows for easier deployment on a variety of platforms, including embedded systems.

Pros:

* Interoperability: ONNX provides a platform-neutral way to represent ML models, allowing for easier deployment across different platforms.
* Support: Supported by many popular ML frameworks, including PyTorch, TensorFlow, and Microsoft's Cognitive Toolkit.

Cons:

* Limited Operations: While ONNX supports a broad range of operations, it might not cover very new or exotic layers.
* Overhead: Converting to and from ONNX can introduce some overhead and sometimes lead to minor performance discrepancies.

### Edge Impulse

A platform designed specifically for developing embedded ML solutions.

[GitHub - edgeimpulse/courseware-embedded-machine-learning](https://github.com/edgeimpulse/courseware-embedded-machine-learning?utm_campaign=University%20Program&utm_medium=email&_hsmi=203658073&_hsenc=p2ANqtz-_TwsiAP4Mjzt1slWjMaSobriFrXFRyqTS1Da0tSvUEmwIwhQWtILqlXYCOa8SD-4JU4DGmVAwE7oqGV8NhH73kPeDN1w&utm_content=203658073&utm_source=hs_automation)

[[2212.03332] Edge Impulse: An MLOps Platform for Tiny Machine Learning (arxiv.org)](https://arxiv.org/abs/2212.03332?utm_campaign=University%20Program&utm_medium=email&_hsmi=203658073&_hsenc=p2ANqtz-9MwxW2R3a8FFzj_na7IHTk9PPnm7n8VVmFCMQk3uRroPnBk4jW4UdA4PHLQ-bYOXZ5FhVMs9xJJriczr2mnW2XMdgFLw&utm_content=203658073&utm_source=hs_automation)

Pros:

* End-to-end Platform: Offers a comprehensive solution from data collection to deployment.
* Optimized for Edge Devices: Specifically designed for edge and embedded devices, making it highly efficient.
* User-friendly Interface: Provides a cloud-based interface for building, training, and testing models.

Cons:

* Less Flexibility: While user-friendly, it might not be as flexible for those wishing to perform very custom operations or modifications.
* Dependency: Reliance on a specific platform could be a downside if the platform changes its policies or pricing.

### CMSIS-NN

Provides neural network kernels optimized for ARM Cortex-M processors.

Pros:

* Optimized for ARM Cortex-M: Specifically designed for ARM's microcontroller architectures.
* Efficient: Provides highly optimized neural network functions.

Cons:

* Limited to ARM: While great for ARM Cortex-M devices, it's not suitable if you're targeting other architectures.
* Less Comprehensive: It's more of a collection of kernels rather than a full-fledged ML framework, so you may need other tools in conjunction.
* When evaluating these tools and frameworks, it's essential to consider the specific requirements and constraints of your project. Factors such as device architecture, memory availability, desired model complexity, and the need for future updates can all influence which tool is most appropriate for your needs.

### TinyML

A term often associated with running ML models on ultra-low-power microcontrollers.

Pros:

* Community and Ecosystem: The TinyML term represents a growing ecosystem, and with its increasing popularity, more resources, tools, and support become available.
* Focused on Ultra-Low-Power Devices: Specifically designed for microcontrollers and extremely resource-constrained devices.
* Integration with TensorFlow Lite for Microcontrollers: This offers a smooth transition for developers familiar with TensorFlow.

Cons:

* General Term: TinyML isn't a tool per se, but rather a term that encompasses many tools, techniques, and platforms. This might make it confusing for newcomers to identify what exactly they need.

### MicroML

A tool to generate C code from trained TensorFlow Lite models, making them ready to run on microcontrollers.

Pros:

* Simple Deployment: Helps generate C code from TensorFlow Lite models, simplifying deployment on microcontrollers.
* Resource-Constrained Devices: Focus on extremely limited devices.

Cons:

* Narrow Scope: Primarily focused on generating C code from TFLite models, which might not cater to broader ML needs.
* Less Comprehensive: Doesn’t provide a full end-to-end solution like some commercial platforms.

### Cubemos

Offers an SDK for skeleton tracking, and while it's focused on a specific application (vision-based human pose estimation), it is an example of specialized tools that can be used for embedded ML applications.

Pros:

* Specialized: Focused on skeleton tracking, which can be an advantage for projects that specifically need this capability.
* SDK Availability: Provides an SDK for easier development and deployment.

Cons:

* Niche Focus: Its focus on skeleton tracking means it might not be suitable for broader machine learning tasks.
* Less Information on Open Source: At the time of my last update in September 2021, Cubemos' broader open-source offerings and community contributions weren't clear.

### EloquentTinyML

This is a library to simplify running TinyML models on Arduino and other platforms.

Pros:

* Arduino Friendly: Designed to simplify running TinyML models on Arduino and similar platforms.
* Ease of Use: Provides a simpler interface compared to some more general tools.

Cons:

* Limited to Arduino Ecosystem: While Arduino is popular, this tool might not cater to developers working with other platforms.
* Dependency on Libraries: Relies on specific libraries, which could pose issues if they get deprecated or aren't updated.

### Conclusion

* Versatility: Tools like TensorFlow Lite for Microcontrollers offer broader versatility and support for various platforms, while some tools like EloquentTinyML or Cubemos are more specialized.
* End-to-End Solutions: Edge Impulse provides a more comprehensive, end-to-end platform, while many of the open-source alternatives focus on specific parts of the machine learning pipeline.
* Community and Support: Popular tools like TensorFlow Lite for Microcontrollers have robust community support, extensive documentation, and resources, which might be lacking in more niche or newer tools.
* Performance and Optimization: Some tools might have better optimization techniques or cater more specifically to certain hardware architectures. For instance, CMSIS-NN is tailored for ARM Cortex-M processors.

## Data collection

Data collection challenge. There is no large-scale, curated, public sensor data set for the embedded ecosystem. Currently, it is difficult to efficiently collect, and

analyze such datasets from a rich variety of sensors.

Additionally, data cleaning and labeling are essential

to ML development but are expensive, labor intensive

processes without tooling or automation.

* Sensor data is challenging to work with, and Python models are hard to deploy to the edge. Edge Impulse handles the tough parts of edge AI, so every ML practitioner can feel confident solving problems on the edge.
* Build high-quality sensor datasets
* Collect and visualize your sensor data to uncover critical insights that will enable you to build a high-quality datasets to achieve your target outcomes faster and in a cost-effective way.
* Edge impulse: Active learning (Build a high-quality dataset with sophisticated tools that guide data collection and labeling for complex sensor data).

### Data Security

Combining multiple layers of security—physical, technical, and administrative—provides a comprehensive defense against potential threats.

Physical Security, Access Controls,Data Encryption: At Rest: Encrypt data when it's stored on disks using technologies like BitLocker, LUKS, or hardware encryption. In Transit: Use protocols like HTTPS, SSL/TLS, and VPNs to ensure data is encrypted when being transmitted.

Secure Passwords and Authentication:Complex Passwords: Encourage the use of strong, unique passwords.Two-Factor Authentication (2FA): Implement 2FA for an extra layer of security.

Password Managers: Promote the use of password managers to store and generate strong passwords.

Regular Backups:Create regular backups of essential data.Ensure backups are encrypted and stored securely, both onsite for quick recovery and offsite for disaster recovery.

Firewalls and Intrusion Detection/Prevention Systems (IDS/IPS):Install firewalls to monitor and control incoming and outgoing network traffic.Use IDS/IPS systems to detect and prevent potential threats in real-time.Regular Updates:Frequently update operating systems, applications, and any software to patch vulnerabilities.Use automated tools for vulnerability scanning and patch management.

Secure Networking:VPNs: Use Virtual Private Networks for secure remote access.

Segmentation: Segment the network to ensure sensitive data is separated and more secure.

Wireless Security: Ensure Wi-Fi networks are secured with strong passwords and the latest encryption standards like WPA3.Limit User Access:Use the principle of least privilege (PoLP) to ensure users only have access to data they need.Regularly review permissions and revoke unnecessary access.

Employee Training:Train employees about the importance of data security, phishing threats, and best practices.Foster a culture of security awareness.Endpoint Security:Secure all endpoints with updated antivirus and antimalware solutions.Consider Mobile Device Management (MDM) solutions for mobile devices.Data Disposal:Securely dispose of old hardware using techniques like disk wiping or physical destruction.Implement secure methods for data deletion.Incident Response Plan:Have a plan in place for data breaches or security incidents.Regularly review and practice this plan.Regular Audits:Conduct regular security audits to identify vulnerabilities.Use third-party services for unbiased security assessments.Data Minimization:Only collect and store data that is absolutely necessary.

Regularly purge unneeded or outdated data.

Reduce attack surface

We don’t use any third party software to run your models, removing unseen security vulnerabilities.

Air gap for greater security

Our models run completely offline to keep your data safe and all data is anonymized (no personally identifiable information is used).

Reduce attack surface

We don’t use any third party software to run your models, removing unseen security vulnerabilities.

SOC 2 Type 2 compliance?

## Feature engineering for sensor data

Data preprocessing challenge. Digital signal processing (DSP) is a critical stage of the ML stack and has strong interactions with the ML model, that are sometimes hard to quantify. Yet there are a lack of automated machine learning tools for the embedded ecosystem that include the DSP component, which hinders the development of efficient preprocessing methods for these systems by non-domain experts

* Digital signal processing that can improve your on-device inference performance.
* Digital signal processing (DSP) algorithms, available during training and optimized for on-device performance.
* Sensor data guidance: Use data exploration tools to visualize and uncover critical insights that can help you improve your dataset.
* Edge impulse: DSP autotuning (Perform automatic DSP parameter tuning to discover the best configuration to use based on your dataset), EON Tuner (Leverage an AutoML tool able to find the right balance between DSP and ML models for your dataset and device).

## ML models

Build your own model or leverage ground-breaking models like FOMO (Faster Objects More Objects) built to bring real-time object detection, tracking and counting to microcontrollers for the first time.

### K nearest neighbors (KNN)

### Support vector machine (SVM)

### Decision forests

[Introducing TensorFlow Decision Forests — The TensorFlow Blog](https://blog.tensorflow.org/2021/05/introducing-tensorflow-decision-forests.html)

https://www.tensorflow.org/decision\_forests

[Training tree-based models with TensorFlow in just a few lines of code — The TensorFlow Blog](https://blog.tensorflow.org/2022/08/training-tree-based-models-with-TensorFlow.html)

## Deployment

Estimate the on-device performance of your model continually during model development, and generate a portable and optimized C++ library ready to be deployed to any edge device.

## Optimization Techniques:

* Model pruning: Removes unnecessary weights or neurons.
* Quantization: Reduces the precision of the model's weights and sometimes activations.
* Knowledge distillation: Trains a smaller model (student) to imitate a larger, more complex model (teacher).
* Efficient architectures: Like MobileNets, TinyML, or EfficientNet for deep learning tasks.

Complement your workflow with tools built for dealing with complex sensor data and with edge devices, from data collection, to profiling, to optimization and on-device model deployment.

## IDE integration

Integrate Edge Impulse in your environment with the Python SDK and into deployable C++ libraries optimized for any edge device.

Do all this from within your existing Python scripts and notebooks by leveraging Edge Impulse's Python SDK and the Python API bindings.

#### Fundamentals of TinyML ([1.1: Welcome to Deploying TinyML | Deploying TinyML | Deploying TinyML | edX](https://learning.edx.org/course/course-v1:HarvardX+TinyML3+1T2023/block-v1:HarvardX+TinyML3+1T2023+type@sequential+block@1d7335df2a1a49089ca2fe814f1ff98b/block-v1:HarvardX+TinyML3+1T2023+type@vertical+block@3705041f953543e88a436162bc79c488))

Before we could understand TinyML, we needed to understand its underpinnings, namely, machine learning. The utility of TinyML only becomes apparent once we understand the machine learning workflow, the different supervised learning algorithms, how they are structured, and the tasks they are able to solve (and what data they act on). To this end, the first course provided a crash course in these topics, showing that models can be trained on existing data, allowing the model to learn important associations between its parameters and the data. These trained models could then be exposed to new information, and make predictions about the data based on these associations. The two main tasks that these models perform are called regression and classification. Regression algorithms try to approximate the mapping function (f) from the input variables (x) to numerical or continuous output variables (y), whereas classification algorithms try to approximate the mapping function from the input variables to different discrete or categorical output variables.

#### Machine Learning

We have seen that machine learning entails the development of data-driven models to make predictions or decisions for a particular task. Supervised learning takes labeled data, meaning that both the input and output variables are known, and attempts to produce a model that can manipulate the input variables in such a way that the output variable can be effectively predicted.

Depending on the curated dataset, there may be a large number of input variables or very few; there may be very many data points for us to train our model with or only a small number; we may need to preprocess the data in order to extract more relevant feature information, known as **feature engineering**, or simply leave the input data unperturbed. Thus, the first important steps are **dataset collection** and **dataset preprocessing**. The collection stage is often the most time-consuming unless an existing dataset is available. Determining how much data is necessary to perform a robust analysis, the best method to obtain the data (e.g., crowd-sourcing, external data providers, simulated data), as well as the number of features and their relevance, are challenging steps that will often require multiple iterations to get right.

Once our dataset is curated and preprocessed, we are ready to begin developing and training machine learning models. The training process is where a model tunes its parameters based on associations learned from a particular dataset. Different tasks lend themselves better to certain machine learning methods. For example, convolutional neural networks often work best for image data - a standard fully connected neural network can also be used, but will likely perform worse at the same task. Usually, models are selected based purely on their performance, but other metrics may also be important, such as interpretability and model complexity.

Different machine learning algorithms have different **hyperparameters**, pre-specified values which characterize the configuration of the algorithm. During the training of a machine learning model, often through the use of a **stochastic gradient descent** algorithm on a chosen **loss function**, these parameters are often altered to find the values that lead to the highest-performing model. This process is known as **hyperparameter optimization**. Some platforms, known as AutoML tools, have **automated** this activity, making it easier for the user to optimize their models during the training process (e.g., [Google’s Cloud AutoML](https://cloud.google.com/automl)). The optimization process performs the **model evaluation**, which assesses the performance of a model using metrics of interest, such as [false-positive rate](https://en.wikipedia.org/wiki/False_positive_rate) and [F1 score](https://en.wikipedia.org/wiki/F-score). [Cross-validation](https://machinelearningmastery.com/k-fold-cross-validation/#:~:text=Cross%2Dvalidation%20is%20a%20resampling,on%20a%20limited%20data%20sample.&text=That%20is%2C%20to%20use%20a,the%20training%20of%20the%20model.) is often used to improve the robustness of the algorithm by preventing overfitting from occurring.

Recall that once our model has been suitably trained and optimized, we test our model on a hold-out dataset known as the test set. This simulates an unseen dataset, like our model would see in the production environment. If our model works on this set of data, it should work in our production environment! After a positive result here, we are then ready to deploy our model, which, depending on the application, may involve integration into existing infrastructure. Models are often **monitored** after deployment to ensure that they are functioning as expected.

#### Embedded Systems

The main embedded system used in this course is the Arduino Nano 33 BLE Sense, which is a relatively small microcontroller equipped with a 64 MHz processor, 1 MB of flash memory, and 256 KB of SRAM. While this system is highly resource-constrained, it is low cost and can be purchased for about $30. Clearly, this system is very different from the device you are using to view this reading. Laptops and smartphones have processors in the range of GHz, which is hundreds, if not thousands of times faster than the Arduino Nano. Furthermore, the Nano does not have any input or output peripherals that we are mostly accustomed to, such as a trackpad, keyboard, and screen. Finally, the small amounts of program memory and RAM available mean that the system has very little capacity to perform complex tasks. In fact, our particular embedded system is designed to only run one task at a time, as compared to your laptop which probably currently has multiple tabs of Chrome open!

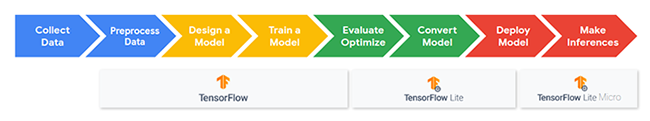
However, we have seen that there are multiple benefits despite the severe resource constraints. Since there is no (or a very limited and lightweight) operating system, we have minimal **computational overhead**, which improves inference speed (i.e., how many times our model can run per second) and reduces system **latency** (i.e., communication delay between sending and receiving data).

**Privacy** is another important benefit. Embedded systems that can perform machine learning do not need to communicate data, which reduces network loads and also prevents the possibility of man-in-the-middle attacks of transmitted data. Perhaps more important, only localized data is stored on the devices, meaning that there is no risk of data being stolen from a central repository.

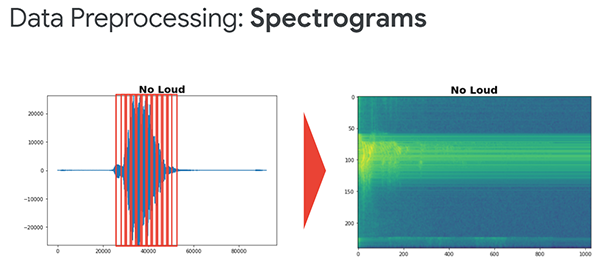
Perhaps the greatest benefit of embedded systems is their small **power consumption**. Modern devices such as laptops and smartphones consume a great deal of power, and running large machine learning algorithms often has a high computational cost. In comparison, embedded systems use very little power, typically on the order of mW. This is no small feat since such devices can run on a simple coin cell battery for an entire year without needing to be recharged - just imagine not having to charge your laptop for a year!

#### Applications of TinyML

After learning the ropes of machine learning and motivating the utility of performing machine learning on embedded systems, we started to look at some archetypal examples of TinyML applications and some of the specifics of how models are ported from frameworks such as TensorFlow to embedded systems. We studied all of this in the context of the machine learning flow moving from collecting and preprocessing data to designing, training, evaluating, and converting models. We’ll cover deployment in this course!



The first example we looked at was **keyword spotting**, which involved extracting the presence of specific keywords from a short voice recording. This example already exists in smartphones such as with Apple’s “Hey Siri” and Google’s “OK Google”. We performed **feature extraction** of the voice recording by using spectrograms, which were then used to train our model.



Following feature extraction, we performed **post-training quantization** of model weights and inference calculations to allow our model to run using the 8-bit arithmetic available on most embedded systems¹. In the penultimate stage, we took the quantized model and converted it to a more suitable file format, which, in this case, was a TFLite model. After converting this model to a binarized format, the model was then ready to be deployed in an embedded system.

We then went on to look at further examples, such as **visual wake words**. In the visual wake words application, we trained a model to determine the presence of a person in an image and saw how transfer learning could be used to retrain models for similarly related applications, such as detecting the presence of masks on a person’s face.

In the final parts of Course 2, we looked at more advanced industry-oriented applications that began to move towards unsupervised learning algorithms, such as **anomaly detection**.

We have already done a considerable amount in these first two courses. In Course 3, we will go one step further still, and provide you with all the tools necessary to develop, build, and troubleshoot your own TinyML systems. Let’s jump in!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

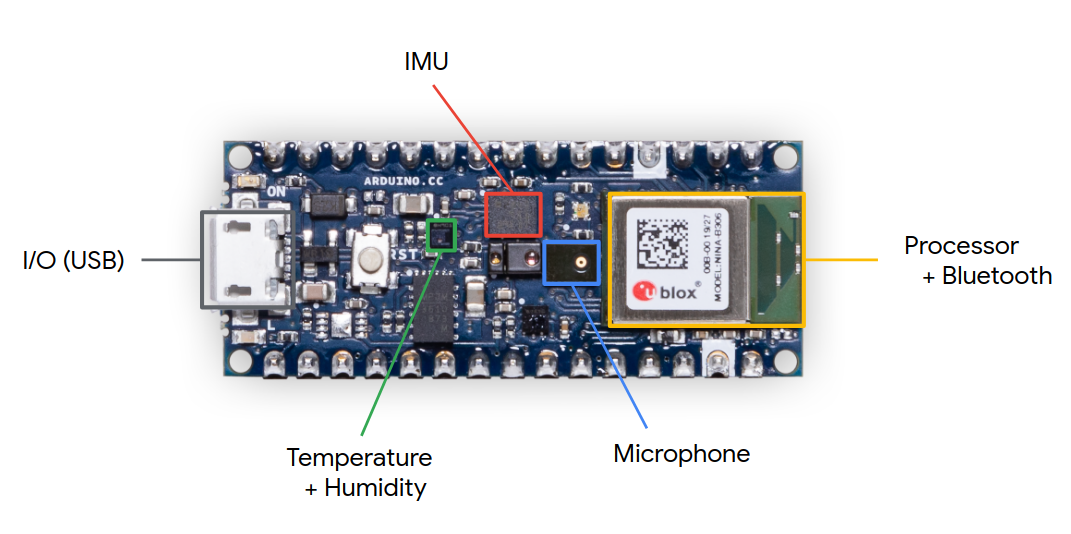
¹We must also note that we explored how **Quantization Aware Training** can reduce the errors introduced by quantization by allowing the model to adjust the weights to the quantization scheme during training.

### Embedded Systems

In this reading, we will explore the diversity of embedded systems.

If you take a few minutes to look around the room you’re in, you’ll no doubt recognize the proliferation of embedded hardware within our world. Embedded systems are increasingly ubiquitous because they serve to complete specific computational tasks within their environment, allowing us to not only collect all sorts of data from distributed sensors but also to process the resulting information locally and act accordingly. That we can accomplish this in situ, or within the environment that this hardware is embedded within, at a fraction of the cost and physical scale of most general-purpose computing hardware, opens the door to many advancements in automation and generally to the creation of smart, connected things.

While the specifications and capabilities of a given embedded system ought to be tailored for its application, in general, we can look to the common construct for an embedded system that links sensing to processing and later actuation through the process of transduction, the conversion of one form of energy into another, for throughlines. In sense, we convert energy from a physical phenomenon into an electrical signal we can go on to digitize and compute with. In actuation, the converse. In this course, we’ll focus predominantly on the central element of said construct: the computing hardware, and review specifications and technical considerations that vary between implementations of embedded systems.



#### Development Boards

In the context of this course, we have talked about and deployed a microcontroller unit (or MCU) development board from Arduino, the Nano 33 BLE Sense. Here, we distinguish the nRF52840 MCU from the development board it is soldered to. What’s interesting about this board is that despite its relatively small size, it is jam-packed with many of the representative elements of an embedded system, all on one printed circuit board (PCB): a collection of surface-mount sensors, a form of an actuator in the on-board programmable RGB LED, and an integrated MCU-BLE module, where BLE is an acronym for a branch of the Bluetooth standard called [Bluetooth Low Energy](https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-low-energy/). Further still, the PCB that the MCU is soldered to serves to ‘break out’ additional IO from the controller to provide interfaces for external, off-board modules and connections to daughter boards. So, while in many ways, the Nano 33 BLE Sense is representative of an embedded system, it is separated from commercial implementations in that it serves no particular purpose, other than the open-ended development of an application that is. Put another way, a development board has the advantage of enabling many potential use cases. Further, you don’t have to go through the process of designing circuitry, capturing schematics, or physical layout for such a board to get started prototyping a concept that could go on to be manufactured with specific intent later. The tradeoffs you make in employing a development board often involve board size and cost.

#### Board Size

Board size, and shape, have fairly meaningful implications for use in the field, given these simple parameters constrain where and how a system can be deployed. Some environments or contexts within which we’d like to embed a system are forgiving (like the Google Home, say), but other manifestations (smart glasses, for example) depend wholly on the requisite hardware being quite tiny and perhaps of unique form. Somewhat obviously, board size is determined by the number and physical size, or package, of the components that live upon it. At a minimum, an embedded system must include a MCU chip alongside a power source, often a lithium polymer battery, and power circuitry. The [nRF52840](https://courses.edx.org/assets/courseware/v1/0c9851bb6f6b06104a552daff589ed1e/asset-v1:HarvardX+TinyML3+1T2023+type@asset+block/nRF52840_PB_v1.0.pdf) MCU on the [Nano 33 BLE Sense](https://courses.edx.org/assets/courseware/v1/4d30cc2dd8a8d0b07aade8501c8a5f29/asset-v1:HarvardX+TinyML3+1T2023+type@asset+block/NANO33BLE_V2.0_sch.pdf) lives within a MCU-BLE module, the [U-Blox NINA-B306](https://courses.edx.org/assets/courseware/v1/573b32f0932af0c4b18b4ba9b83e3972/asset-v1:HarvardX+TinyML3+1T2023+type@asset+block/NINA-B3_DataSheet_UBX-17052099_C1-Public.pdf), which spans 10 by 15 mm, dimensions that include a trace antenna. The [MPM3610](https://courses.edx.org/assets/courseware/v1/db91acf50c0f8d470b3ef738ddc634ae/asset-v1:HarvardX+TinyML3+1T2023+type@asset+block/MPM3610AGQV.pdf) step-down converter used to down-regulate the 5V delivered to the board over USB occupies 3 by 5 mm, alongside small SMD passives. The entire board, meanwhile, spans 18 by 45 mm. While impressive, clearly a purpose-designed PCB could remove some of the sensors and IO breakout unnecessary for a specific application to reduce the overall scale even further, perhaps by about 60 to 70%.

#### Cost

Unsurprisingly, the cost of a development board scales with its MCU’s compute capability and general feature set. In selecting the appropriate MCU for an application, it is important to remember that MCUs are often deployed to complete specific tasks, where the complexity they will face is predetermined or constrained. For very simple tasks, we highlight the ATTINY85 (an 8-bit processor with 8 kB of flash) that can, depending on the package selected, cost well less than $1 EA and even less at scale, perfect for simple computing requirements. In the context of TinyML, the AI-capable NINA-B306 module featuring the Nordic Semiconductor nRF52840 chip (an ARM Cortex M4) and all-in-one BLE hardware (including antenna) costs about $10 EA and is more generally representative. In view of the on-board sensors and design costs, the Arduino Nano 33 BLE Sense costs just over $30, not even an hour's time of an electrical engineer who might design such a board — a tremendous value. In general, the boards we originally considered for this class span from $2 ([BluePill](https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html" \t "_blank)) to $54 ([Disco-F746NG](https://os.mbed.com/platforms/ST-Discovery-F746NG/)).

You can find a list of the boards we considered in the table below. Ultimately, our staff selected the Arduino Nano 33 BLE Sense for its versatility, in providing a large selection of on-board sensors, accessible IO via breakout pins, and a BLE module for projects that involve wireless communication, with reasonably representative compute specifications for resource-constrained hardware. We’ll explore the compute specifications in a bit more detail in the next video and reading.

#### TinyML Development Board Comparison                    ◽officially TFLM supported   ▢ unofficially compatible boards

| **0Board** | **MCU** | **CPU** | **Clock** | **Memory** | **IO** | **Sensor(s)** | **Radio** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Arduino Nano 33 BLE Sense | Nordic nRF52840 | 32-bit ARM Cortex-M4F | 64 MHz | 1 MB flash 256 kB RAM | x8 12-bit ADCs x14 DIO UART, I2C, SPI | Mic, IMU, temp, humidity, gesture, pressure, proximity, brightness, color | BLE |
| Espressif ESP32-DevKitC | ESP32 D0WDQ6 | 32-bit, 2-core Xtensa LX6 | 240 MHz | 4 MB flash 520 kB RAM | x18 12-bit ADCs x34 DIO\*\* UART, I2C, SPI | Hall effect, capacitive touch\*\*\* | WiFi, BLE |
| Espressif EYE | ESP32 D0WD | 32-bit, 2-core Xtensa LX6 | 240 MHz | 4 MB flash\* 520 kB RAM | SPI via surface pads | Mic, camera | WiFi, BLE |
| Teensy 4.0 | NXP iMXRT1062 | 32-bit ARM Cortex-M7 | 600 MHz | 2 MB flash 1 MB RAM | x14 10-bit ADCs x40 DIO\*\* UART, I2C, SPI | Internal temperature, capacitive touch | None |
| MAX32630FTHR | Maxim MAX32620 | 32-bit ARM Cortex-M4F | 96 MHz | 2 MB flash 512 kB RAM | x4 10-bit ADCs x16 DIO UART, I2C, SPI | Accelerometer, gyroscope | BLE |

\*This board also features 4 MB flash and 8 MB of PSRAM external to the MCU, \*\*shared programmable functions, \*\*\*with external touchpads

| **Board** | **ASIC** | **DSP** | **Clock** | **Memory** | **IO** | **Sensor(s)** | **Radio** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Himax WiseEye WE-I Plus EVB | HX6537-A | 32-bit ARC EM9D DSP | 400 MHz | 2 MB flash 2 MB RAM | x3 DIO I2C | Mic, accelerometer, camera | None |

We include the Himax WiseEye as an officially supported example of hardware optimization for TensorFlow Lite that calls on an application-specific integrated circuit (ASIC).

### What is TF Lite Micro?

As a future TinyML engineer, it is essential to understand the inner workings of the software you use to know its capabilities and limitations. So, in the upcoming series of videos and readings, you will learn more about the challenges that led to the development of TF Lite Micro, straight from the source. Pete Warden from Google, who leads the team that works on TF Lite Micro, will introduce TF Lite Micro and give us a sneak peek into its internal workings. Here is a preview of what is to come next.

TensorFlow has become the most popular deep learning framework, superseding other popular frameworks such as PyTorch and Keras. TensorFlow, developed by Google, contains a Python frontend with highly optimized C++ code at its core, making it simple to program, fast, and efficient. The library has a large developer community and is now seen as the de facto standard for most machine learning applications.

Despite this, TensorFlow is not suitable for every scenario. The standard TensorFlow library is ~400 MB in size, and even running a relatively small model (e.g., 200 MB) can take up a considerable amount of random access memory (> 1 GB). Such large storage and memory requirements make running simple models on lightweight systems largely intractable.

Recognizing this issue, Google developed a more lightweight framework, TensorFlow Lite, also sometimes referred to as TensorFlow Mobile. The TFLite binary is approximately 1 MB in size, considerably more compact than the original library, making it possible to run deep learning models on mobile devices such as smartphones. This compression was achieved by removing superfluous functionality that is largely unnecessary for mobile deployment.

While this is an improvement, our problem still remains: even TFLite is not suitable for every scenario. Many important deep learning applications exist at the microcontroller-level, which are significantly more resource-constrained than mobile devices, often equipped with less than 1 MB of storage and 256 KB RAM. Clearly, deploying TFLite models is not feasible for microcontrollers, so an alternative solution was needed.

Enter TF Lite Micro. TF Lite Micro takes the compression of the TensorFlow library to the extreme, removing all but essential functionality. In fact, the core runtime of the library takes up only 16 KB, several orders of magnitude smaller than TFLite. With such a small memory footprint, this lightweight framework makes it possible to deploy deep learning models on the smallest of microcontrollers, such as an Arduino Nano.

However,  this is not without its complications.  Deploying models with TF Lite Micro is fraught with new and unique challenges when building models. For example, since all functionality for plotting and debugging is removed, troubleshooting model issues is difficult. Additionally, since many microcontrollers do not have floating-point units or use 8-bit arithmetic, the model weights and activations must be suitably quantized on the microcontroller system. Since model training requires near-machine precision to perform gradient descent, this largely precludes on-device training. Thus, TF Lite Micro models must first be trained on a device with greater computational resources before being ported to the microcontroller, adding an additional stage to the machine learning workflow.

Despite this, the benefits provided by TF Lite Micro - the ability to perform machine learning inference on microcontroller devices - far exceed the challenges, heralding a new era of machine learning that is often referred to as tiny machine learning.

### TFLite Micro Developer Design Principles

There are four overarching design principles that TFMicro was built upon in order to address some of the challenges faced by developers when working with tinyML for embedded systems. This reading provides a synopsis of these core principles, as outlined in further detail in the TensorFlow Lite Micro [paper](https://courses.edx.org/assets/courseware/v1/8973f2e4df755f03a7ddadbabc88b477/asset-v1:HarvardX+TinyML3+1T2023+type@asset+block/2010.08678.pdf).

#### Principle 1: Minimize Feature Scope for Portability

This principle proposes that an embedded machine learning (ML) framework should assume, by default, that the model, input data, and output arrays are in memory, and do not need to be loaded into memory. In addition, accessing peripherals, such as an on-device camera, should not be the job of the ML framework. These functions still need to be fulfilled, but principally should not be fulfilled by the ML framework.

While this may seem unimportant, some microcontrollers do not have memory management (e.g. malloc) and other capabilities. Thus, trying to accommodate all varieties of platforms would bloat the library in an attempt to provide sufficient portability. Fortunately, due to the self-contained nature of machine learning models, the model can be run on-device without the need to access peripherals and system functions.

#### Principle 2: Enable Vendor Contributions to Span Ecosystem

Embedded devices come in all shapes and sizes, and require kernels to perform tinyML functions. The more optimized these kernels are for a particular device, the better performance will be achieved. However, because of the many differences between device platforms, there is no one-size-fits-all optimization solution. Consequently, the TFMicro team by itself is unable to support the wide variety of platforms that may want to run tinyML, and thus, vendors with strong motivation (i.e., those involved in microcontroller development) are encouraged to contribute to help bridge the gap. These vendors often have little experience with deep learning, and thus, TFMicro must provide sufficient resources to allow these teams to easily contribute. One way this is accomplished is by encouraging vendors to submit to a library repository and to provide tests and benchmarks for vendors to assess their hardware performance.

#### Principle 3: Reuse TensorFlow Tools for Scalability

The third principle focuses on scalability. More than 1,400 operations (e.g. CONV2D) are supported by TensorFlow and other machine learning training frameworks. However, inference frameworks (i.e., those actually deploying the model) typically only support a fraction of these operations. For most use-cases, this will likely not cause issues since the most commonly used operations will likely be supported, but this inherent difference leads to a mismatch between the set of potential models produced by the training framework and the set of potential models that can be deployed by the inference framework.

An exporter is used to convert a model from a training framework, such as TensorFlow, to a model for an inference framework, such as TFLite or TFLite for Microcontrollers. This model can then be deployed directly to a device and run using the library interpreter. Often, the training and inference frameworks are developed by different entities, which can present difficulties for developers when there are compatibility issues between the various stages of the developmental pipeline. This may render otherwise functional models unusable when trying to be deployed to a client device, especially when the incompatibilities are abstracted in high-level libraries such as Keras.

Due to these concerns, the TFMicro developers decided to reuse as many TensorFlow tools available as possible to help minimize such complications and compatibility issues.

#### Principle 4: Build System for Heterogeneous Support

The last principle focuses on promoting a flexible build environment. There are a large number of different types of embedded devices that may wish to use tinyML, and thus TFMicro should be designed without preference to any particular platform. This prevents vendor lock-in and also attracts a larger developer ecosystem due to improved portability. To combat this, TFMicro prioritizes code that can be built across a wide variety of integrated development environments (IDEs) and toolchains.

These four principles help to facilitate a developer ecosystem that is oriented towards maximizing portability between various hardware platforms, architectures, frameworks, and toolchains.

1. [Edge Impulse: An MLOps Platform for Tiny Machine Learning (arxiv.org)](https://arxiv.org/pdf/2212.03332.pdf) [↑](#footnote-ref-1)