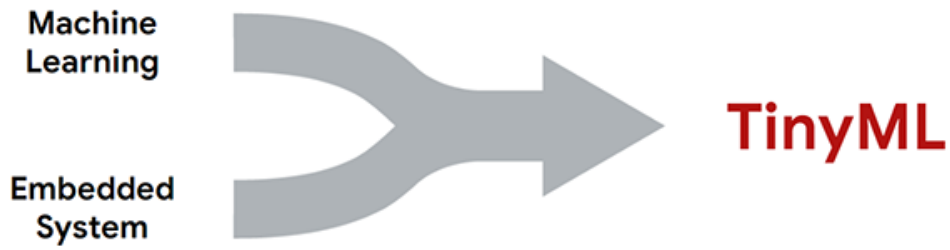
TinyML1 and TinyML2 Recap



Over the first two Modules, you have been exposed to various facets of the machine learning ecosystem. You have seen how models are developed using the machine learning workflow, from the data collection phase through to the training and deployment phases. You have seen how models are produced using Tensorflow and then ported into a lighter-weight framework such as Tensorflow Lite. You have also seen how these models can interact with various forms of input data, such as time-series data from sensors and microphones, to images from an onboard camera. Throughout both Modules, we also made sure to keep in mind the core principles of responsible AI. We talked you through how to design products with all users in mind. We discussed various error types that (can) occur when we are not careful in designing and developing AI models. We explored these concepts using various real-world case studies. We then considered how to ensure that models and datasets are designed with the goals of maximizing fairness and minimizing bias in mind. Finally, we explored Google's What-If Tool which allowed us to explore and analyze fairness and bias issues in real datasets.

In this Module, you will go one step further still, and learn to deploy TinyML models responsibly on your own embedded system. Let's first revise some of the key concepts in more detail.

Module 1: Fundamentals of TinyML

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 Module provided a crash Module 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). The optimization process performs the **model evaluation**, which assesses the performance of a model using metrics of interest, such as false-positive rate and F1 score. Cross-validation 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 Module 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!

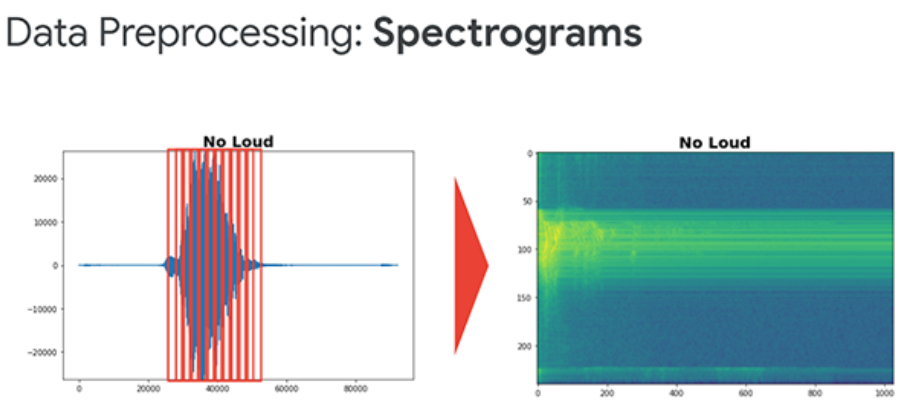
High power
High bandwidth
High latency

Low power
Low bandwidth
Low latency

Module 2: 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 Module!

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 systems[1]. 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 Module 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 Modules. In Module 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!

[1]We 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.

***