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.

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.

***