Chapter 13:
TensorFlow Lite for Microcontrollers

Mijin An
meeeeeejin@gmail.com

TensorFlow

- Google's open source machine learning library (first released to the public 2015)
  - **"An Open Source Machine Learning Library Framework for Everyone"**

- Target: Linux, Windows, macOS **desktop and server** platforms
  - Having **hundreds of megabytes of binaries** is not a problem when using a cloud server with gigabytes of RAM and terabytes of storage

- But, adding **even a few megabytes** to an app on **Android and iOS** devices can decrease the number of downloads/customer satisfaction dramatically

TensorFlow Lite

- To meet these **lower size requirements for mobile platforms**, Google started a companion project, TensorFlow Lite (in 2017)

- To reduce the size and complexity of the framework, it drops features that are less common on the platform

- With the trade-offs, it can fit within just **a few hundred KB**

TF Lite for Microcontrollers

- On **embedded** platforms, the biggest constraint is binary size
  - They need something that would fit within **20KB or less**

- Google started experimenting TF Lite for the embedded platforms (in 2018)

- Main Goals:
  - To reuse as much of the code, tooling, and doc from the mobile project as possible
  - To focus on the read-world use case (e.g., wake-word such as “Hey Google”)

Key Requirements (1)

- **No operating system dependencies**
  - A ML model is fundamentally a mathematical black box
  - So, it don't need to access to the rest of the system
  - Some platforms don't have an OS at all
- **No standard C or C++ library dependencies at linker time**
  - The purpose is to deploy on devices with only a few tens of KB of memory
  - So, the binary size is very important
  - It is important to avoid anything (e.g., `sprintf()`) that hold the C/C++ std libraries

Key Requirements (2)

- **No floating-point hardware expected**
  - Many embedded platforms don't support for floating-point arithmetic in HW

- **No dynamic memory allocation**
  - A lot of apps using microcontrollers need to run for months or years
  - So, the dynamic memory allocation (e.g., `malloc()`/`free()`) can make the heap fragmented, causing an allocation failure or a crash

Why Is the Model Interpreted?

- **Code generation** (like compile languages)
  - Advantages: Ease of building, Modifiability, Inline data, Code size
  - Disadvantages:
    - Upgradability
    - Multiple models
    - Replacing models

- So, the TF Lite team uses **project generation** (like interpreter languages) to get a lot of the benefits of code generation, without incurring the drawbacks

Project Generation

- A process that
  - Creates a **copy** of the source files you need to build a particular model
  - And optionally sets up any **IDE-specific** project files to build them easily

- Key advantages:
  - Upgradability
  - Multiple and replacement models
  - Inline data
  - External dependencies

Build Systems

- TF Lite was originally developed in a Linux environment
  - So, a lot of tooling is based on traditional UNIX tools

- You can build for a lot of platforms using a standard Makefile approach:

```
$ make –f tensorflow/lite/micro/tools/make/Makefile test
```

- You can build a specific target:

```
$ make –f tensorflow/lite/micro/tools/make/Makefile \
TARGET="sparkfun_edge" micro_speech_bin
```

Supporting a New Hardware Platform

- Printing to a log
  - The only platform dependency for inspecting externally from a host machine
- Implementing `DebugLog()`
- Running all the targets
- Integrating with the Makefile build

Supporting a New IDE or Build System

- First, see whether the raw set of files that generated when you generate a Make project can be imported into your IDE

- If not, modify the Makefile to support some transformations for your IDE

Supporting New Hardware Accelerators

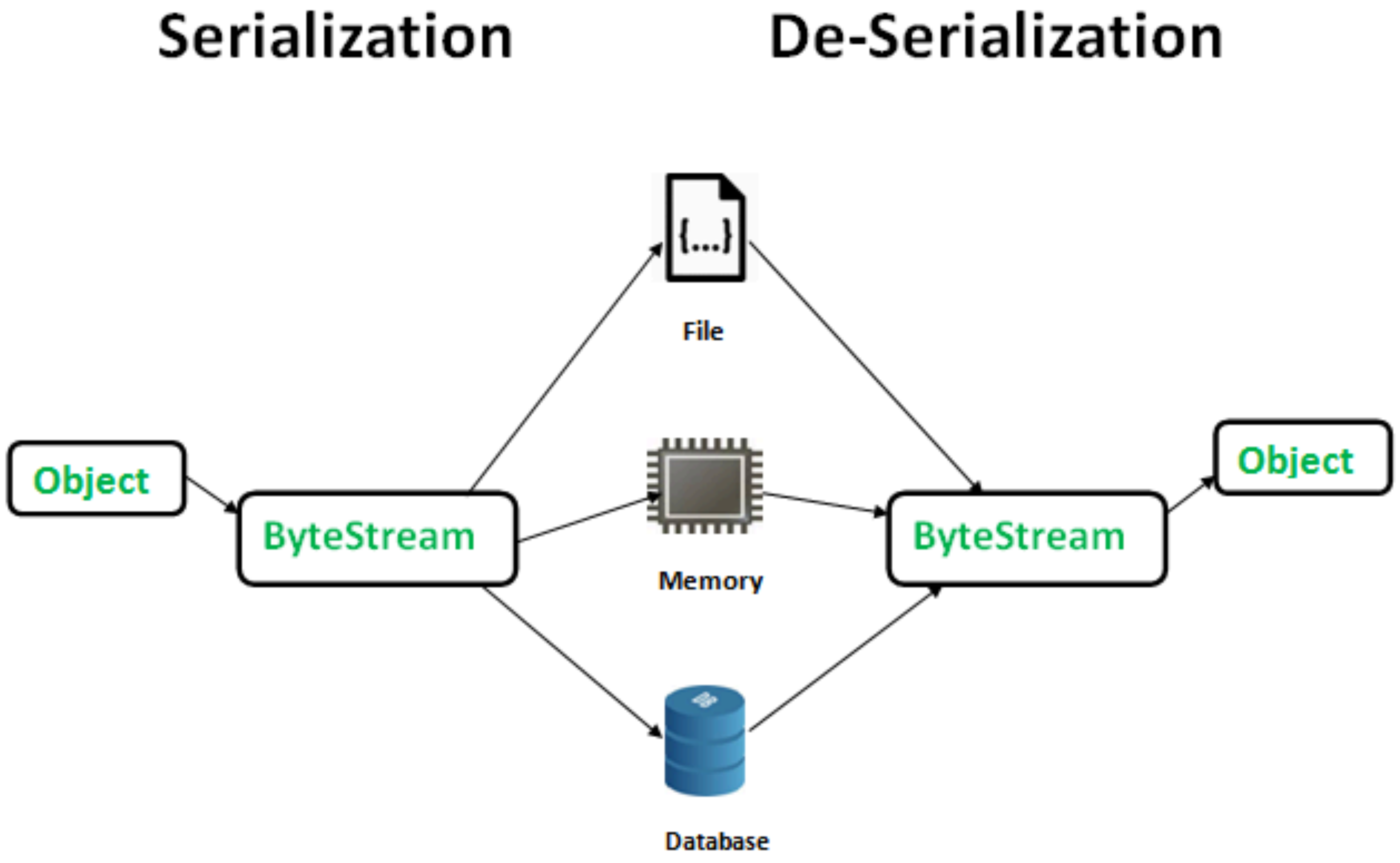
- One of the goals of TF Lite for Microcontrollers is to be a reference SW platform to help HW developers make faster progress with their design

- The tricky details like quantization are not good candidates for HW optimization because they take so little time

- What HW developers can do:
  - To get the **unoptimized code** for TF Lite for Microcontrollers running on their platform and producing the correct results
  - To **replace individual operator implementations** at the kernel level with calls to any specialized HW

FlatBuffers (1)

- A **serialization** library for applications for which performance is critical
  - Good for embedded systems



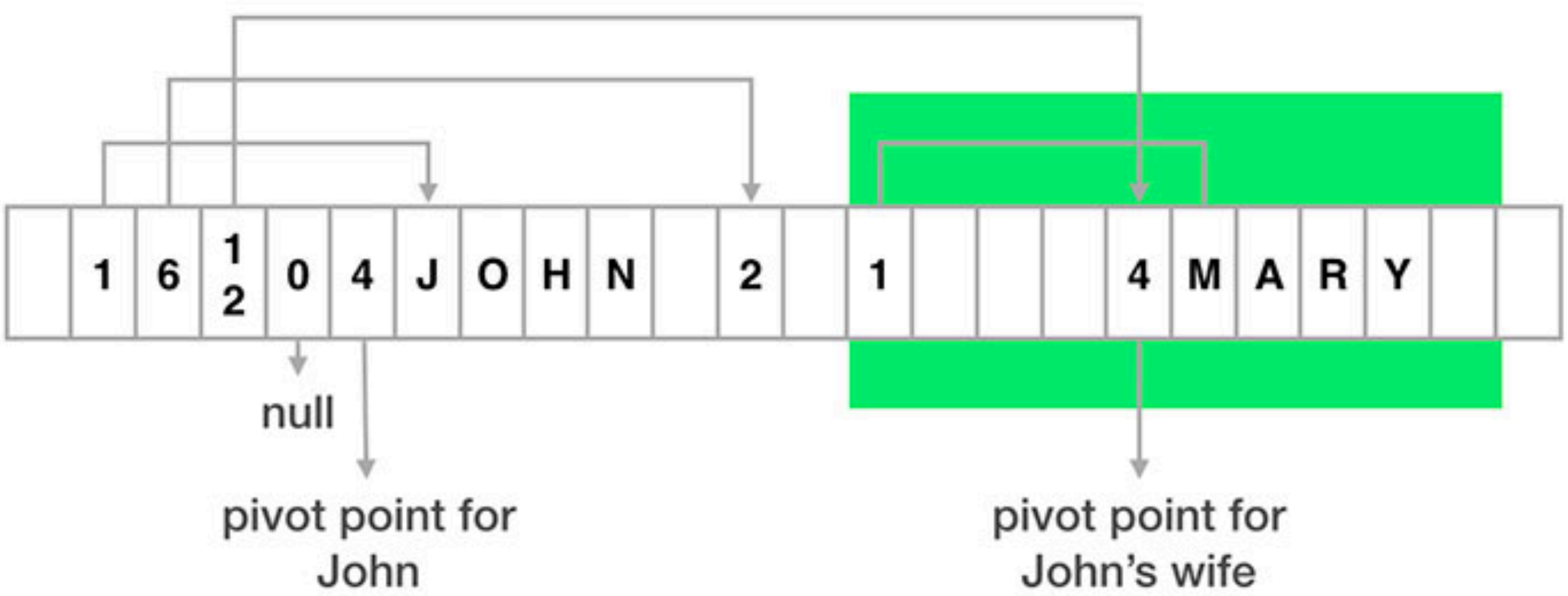
FlatBuffers (2)

- Its **runtime in-memory representation** is exactly the **same as its serialized form**
  - So, models can be embedded directly into flash memory and accessed immediately
  - **No** need for any **parsing** or **copying**
  - Machine-friendly (Hard to understand for human)

- It works using:
  - A **schema** that defines the data structure to serialize
    - `tensorflow/lite/schema/schema.fbs`
  - A **compiler** that turns the schema into native C++ (or C, Python, etc.) code for reading and writing the information

FlatBuffers: Example (1)

```
table Person {
   String name;
   int friendshipStatus;
   Person spouse;
   List<Person> friends;
}
```



- A FlatBuffer for a person, John, and his wife, Mary
- Each object is separated into two parts:
  - The metadata part (**vtable**) on the left and the **real data** part on the right
- Each field corresponds to a slot in vtable

FlatBuffers: Example (2)

```
// Root object position is normally stored at beginning of flatbuffer.
int johnPosition = FlatBufferHelper.getRootObjectPosition(flatBuffer);
int maryPosition = FlatBufferHelper.getChildObjectPosition(
   flatBuffer,
   johnPosition, // parent object position
   2 // field number for spouse field);
String maryName = FlatBufferHelper.getString(flatBuffer, johnPosition, 2);
```

- There is **no intermediate object creation** involved
  - Saving transient memory allocations
- We can optimize this more by storing the FlatBuffer data directly in a file and `mmap`-ing it

FlatBuffers in TF Lite

tensorflow/lite/schema/schema.fbs

```
table Model {
  version:uint;
  operator_codes:[OperatorCode];
  subgraphs:[SubGraph];
  description:string;
  buffers:[Buffer];
  metadata_buffer:[int];
  metadata:[Metadata];
  signature_defs:[SignatureDef];
}

root_type Model;
```

tensorflow/lite/micro/examples/micro_speech/micro_speech_test.cc

```
const tflite::Model* model =
  ::tflite::GetModel(g_tiny_conv_micro_features_model_data);

if (model->version() != TFLITE_SCHEMA_VERSION) {
  TF_LITE_REPORT_ERROR(&micro_error_reporter,
  ”Model provided is schema version %d not equal “
  "to supported version %d.\n”,
  model->version(), TFLITE_SCHEMA_VERSION); }
```

Chapter 14:
Designing Your Own TinyML Applications

Mijin An
meeeeeejin@gmail.com

The Design Process

- Do you need a **microcontroller**, or would a **larger device** work?
- Understanding what’s **possible**
- Follow in someone else’s **footsteps**
- Find some **similar models** to train
- **Look at the data**
- Get it working on the **desktop first**

Wizard of OZ-ing

- This approach is:
  - To **flush out** the requirements
  - To make sure you have the **specification well tested** before you bake them into your design

1

참가자를 프로토 타입 앞에 위치 시키고, 참가자가 볼 수 없는 곳에 마법사(시스템 역할 대행자)가 될 연구자를 위치시킴

2

마법사(시스템 역할 대행자)가 다양한 역할과 행동을 연기힘

3

유저가 취한 행동들을 기록함

4

기록된 결과를 분석하고 공유함

Reference

[1] Pete Warden and Daniel Situnayake, “TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers”, O'Reilly Media, 1st Edition (December 31, 2019)

[2] Dilanka Muthukumarana, “Java Serialization”, Medium, https://medium.com/@dilankam/java-serialization-4ff2d5cf5fa8

[3] “Improving Facebook’s performance on Android with FlatBuffers”, Facebook Engineering, https://engineering.fb.com/android/improving-facebook-s-performance-on-android-with-flatbuffers/

[4] “Wizard of OZ”, KOREA UNIVERSITY EXPERIENCE DESIGN METHODOLOGY, http://designmethod.korea.ac.kr/design-method/wizard-of-oz/