第二十三讲
基于ESP32的TFLite（I）
Lecture 23
TFLite for ESP32 (I)

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit for the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for remedy including deletion have you had any concern.

背景
Background

- TensorFlow Lite或简称TFLite是谷歌TensorFlow团队提供的一套深度学习工具，用于转换和优化TensorFlow模型，使其在移动和边缘设备上运行。TensorFlow Lite最初仅在谷歌内部使用，后来谷歌选择将其开源。如今，TFLite已在上十亿台设备上运行。作为边缘AI实现，TensorFlow Lite大大地降低了在设备端部署神经网络的障碍，使得在移动与嵌入式设备上运行机器学习成为可能。

  TensorFlow Lite (TFLite) is a collection of tools developed by TensorFlow team in Google, in order to convert and optimize TensorFlow models to run on mobile and edge devices. Google developed TensorFlow for internal use, but later chose to open-source it. Today, TFLite is running on more than billions of devices. As an Edge AI implementation, TensorFlow Lite greatly reduces the barriers to deploy neural network models on edge devices, making it possible to run machine learning on mobile or embedded systems.

- TensorFlow Lite 特别针对设备端机器学习 (Edge ML) 进行了优化。作为Edge ML模型，它适合部署到资源受限的边缘设备，从嵌入式Linux、Android，到各种MCU等平台。

  TensorFlow Lite is specially optimized for on-device machine learning (Edge ML). As an Edge ML model, it is suitable for deployment to resource-constrained edge devices, ranging from embedded Linux, up to a wide variety of MCUs.

背景
Background

- 注意，尽管TensorFlow Lite可视为TensorFlow的轻量级版本，但其原则上仅提供对已训练模型执行预测的能力（推理任务），这是因为TF Lite专为移动计算平台、边缘设备如嵌入式设备等而设计的，这些系统计算能力较为有限。

  Notably, although TensorFlow Lite is regarded as a lighter version of TensorFlow (TF), it only provides the ability to perform predictions on an already trained model (inference tasks). This is because TF Lite is specifically designed for mobile computing platforms, edge devices such as embedded devices, etc. These devices are with rather limited computing capabilities, and disqualified for the high computing load during training.

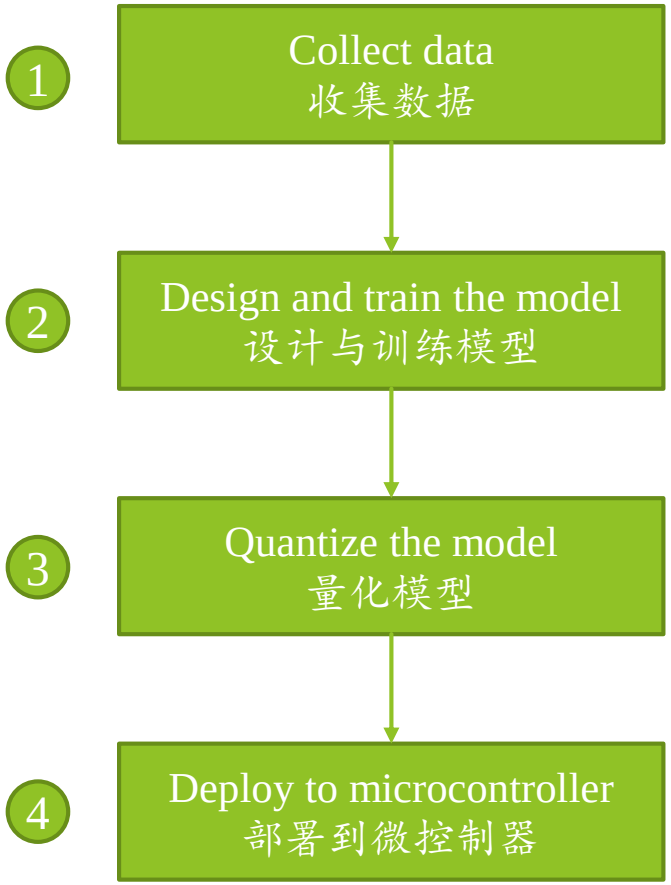
- 这意味着我们通常使用TensorFlow构建和训练机器学习模型。而用TensorFlow Lite在边缘设备上进行推理。同时，TensorFlow Lite提供了相关工具，包括使用量化技术优化训练后的模型等，从而减少必要的内存使用量以及利用神经网络的计算成本。

  The above indicates that we generally use TensorFlow to build and train the ML model, then shift to TensorFlow Lite for inference on edge devices. TensorFlow Lite also provides toolkits, such as tools for optimizing the trained model using quantization techniques (discussed later in this article), which consequently reduces the necessary memory usage as well as the computational cost of utilizing neural networks.

整体流程
Overall Process

- 利用TensorFlow Lite实施基于嵌入式系统的项目整体流程如右所示。由于项目设计的初衷是利用机器学习解决实际问题，因此对于前置的训练工作不再赘述。

  It is recommended to follow the right procedures in order to harness TensorFlow Lite to implement project for embedded systems. Due to the fact that the motivations of these projects are to solve real problems by machine learning, the prerequisite work such as model training will not be elaborated here.



准备工作
Preparation

- TFLite使用特定格式的模型文件。有两种方法可以将普通TF模型转成TFLite格式文件：
  - 利用tf.lite.TFLiteConverter此Python函数将业已存在的TF模型转为.tflite文件；
  - 直接使用在tfhub.dev托管的.tflite格式的模型。

  There are two ways to convert the general TF models to specific TFLite models to comply with the requirement by TensorFlow Lite on microcontrollers:

  - Use tf.lite.TFLiteConverter Python API to convert the existing TensorFlow model to a .tflite file;
  - Use pre-build .tffile models optimized for mobile on tfhub.dev
- 由于模型的构建与训练既可以基于Keras，又可以基于TensorFlow，因此又有两种方式将原生的不同格式的TF模型转为TFL格式。在转化过程中，用户可以指定是否优化。

  Due to the construction and training of TensorFlow models can be based on Keras or TensorFlow, subsequently, there are two ways to convert the TF models in different native formats into TFL format. During conversion, users can specify further optimizing the model or not.

准备工作
Preparation

- 假设用户基于Keras构建与训练模型，假设构建的模型为example_model，则如下步骤提供了如何将模型转为TFL要求的格式：

  Suppose users utilize Keras for constructing and training the model, which is named example_model, the following procedures show how to convert it into the format required by TFL:

  - 从模型实例构建converter对象，然后调用转换方法：

    Construct the converter object from model instance, followed by invoking convert method:

    ```
    converter = tf.lite.TFLiteConverter.from_keras_model(example_model)
    tflite_model = converter.convert()
    ```

  - 将转换后的模型永久存储：

    Save the model to disk:

    ```
    open("sine_model.tflite," "wb").write(tflite_model)
    ```

准备工作
Preparation

- 在上面的转换过程中，没有对模型进行优化，如果需要优化，特别是量化，需要添加下面额外的步骤：

  There is no extra optimization of the model during the conversion. If optimization is required, specifically, quantization, extra steps need to be done as follows:

  - 在从模型实例构建converter对象后，添加优化约束：

    Add the optimization constraints after instantiating converter object from model instance:

    ```
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    ```

  - 定义表征数据集，以便选择最优量化：

    Define the representative dataset to select the best quantization strategy:

    ```
    def representative_dataset_generator():
        for value in x_test:
            yield [np.array(value, dtype=np.float32, ndmin=2)]
    converter.representative_dataset = representative_dataset_generator
    ```

准备工作
Preparation

- 在上面定义的表征数据生成函数中，x_test表示测试数据集。转换程式会运行原来的模型与以某种方式量化的模型来比较误差，以确定当前的量化方式是否合适。显然这要求测试数据与实际部署后采集的数据具有同分布，否则量化后的模型与实际模型相比，可能会有较大的误差。

In the representative dataset generator, the variable x_test represents the test set. The converter will use data sampled from the representative dataset to drive both the original model and the quantized model to compare the error difference, to assess the suitability of the current quantizing way. It obviously requires that data in x_test and data captured from the deployed environment are of the same distribution. Otherwise, it will cause eminent discrepancy between the original model and the quantized one.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
def representative_dataset_generator():
    for value in x_test:
        yield [np.array(value, dtype=np.float32, ndmin=2)]
converter.representative_dataset =
representative_dataset_generator
tflite_model = converter.convert()
open("model_quantized.tflite," "wb").write(tflite_model)
```

准备工作
Preparation

- 对于直接使用TensorFlow而非Keras的情况，原则上，上面的过程同样适用。但由于TensorFlow并没有Keras中类似容器的模型对象，能同时容纳网络架构与参数，因此需要先把模型架构与参数存下来，再从保存目录进行加载，然后按上面的方式进行转换。

For the case of using TensorFlow directly instead of Keras, the above procedures apply in principle. However, since TensorFlow does not come with a model object acting as a container which simultaneously captures the network structure and weights. Therefore, we need to save the model structure and parameters first, then load them from the specified folder to instantiate the converter object, followed by converting to the TFL format.

```
import tensorflow as tf

model_path = 'saved_model'
converter =
tf.lite.TFLiteConverter.from_saved_model(model_path)
tflite_model = converter.convert()
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Save the converted model to a file
tflite_model_path = 'model.tflite'
with open(tflite_model_path, 'wb') as f:
    f.write(tflite_model)
```

准备工作
Preparation

- 由于嵌入式设备比较简单，因此开发与运行环境也会比较有限。通常用于嵌入式设备开发的语言仍是C语言。另外，尽管具有基本的文件读写功能，但为了简单起见，选择将转化后的模型转成字符串，直接编译进最后的可执行文件是更为简单的做法。

  Due to the simplicity of embedded systems, the development tools and runtime libraries are quite limited. Generally, it is the C language still dominates the development of embedded devices. In addition, although embedded systems come with a basic file system, to make it simple, conventionally it is still to choose converting the TFL model into strings and compiling it into the final executable.

- 可以用命令xxd将TFL模型转为C/C++源文件（在Windows平台上可以借助Git Bash完成）：

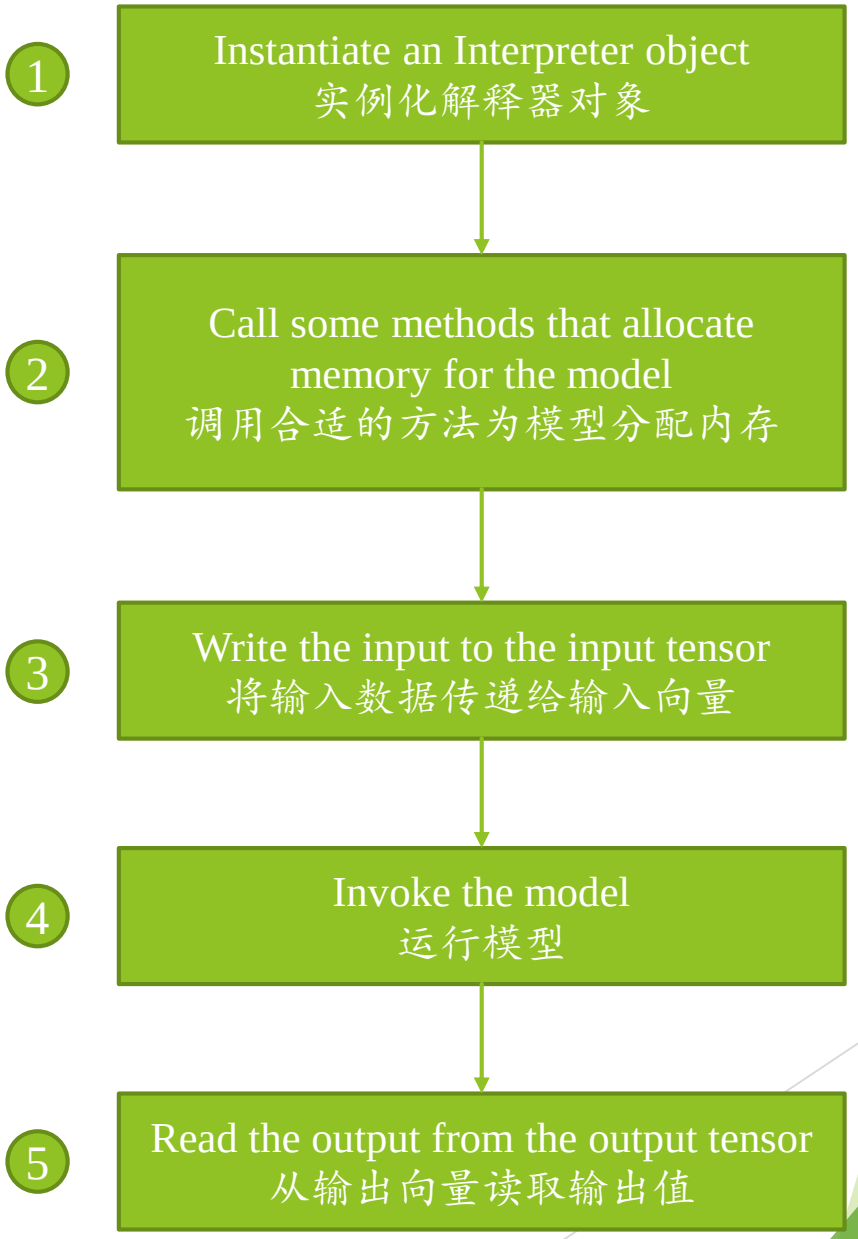
  The xxd command can be used to convert the TFL model into C/C++ source file (On the Windows platform, the Git Bash can be used for such a purpose).

  ```
  !xxd -i model_quantized.tflite > model_quantized.cc
  ```

部署流程
Deployment

- 当模型就绪之后，具体部署到设备上还需要经过编程、烧录等一系列流程。针对编程，需要一系列编码为利用模型进行推断做准备，基本做法如右流程图所示：

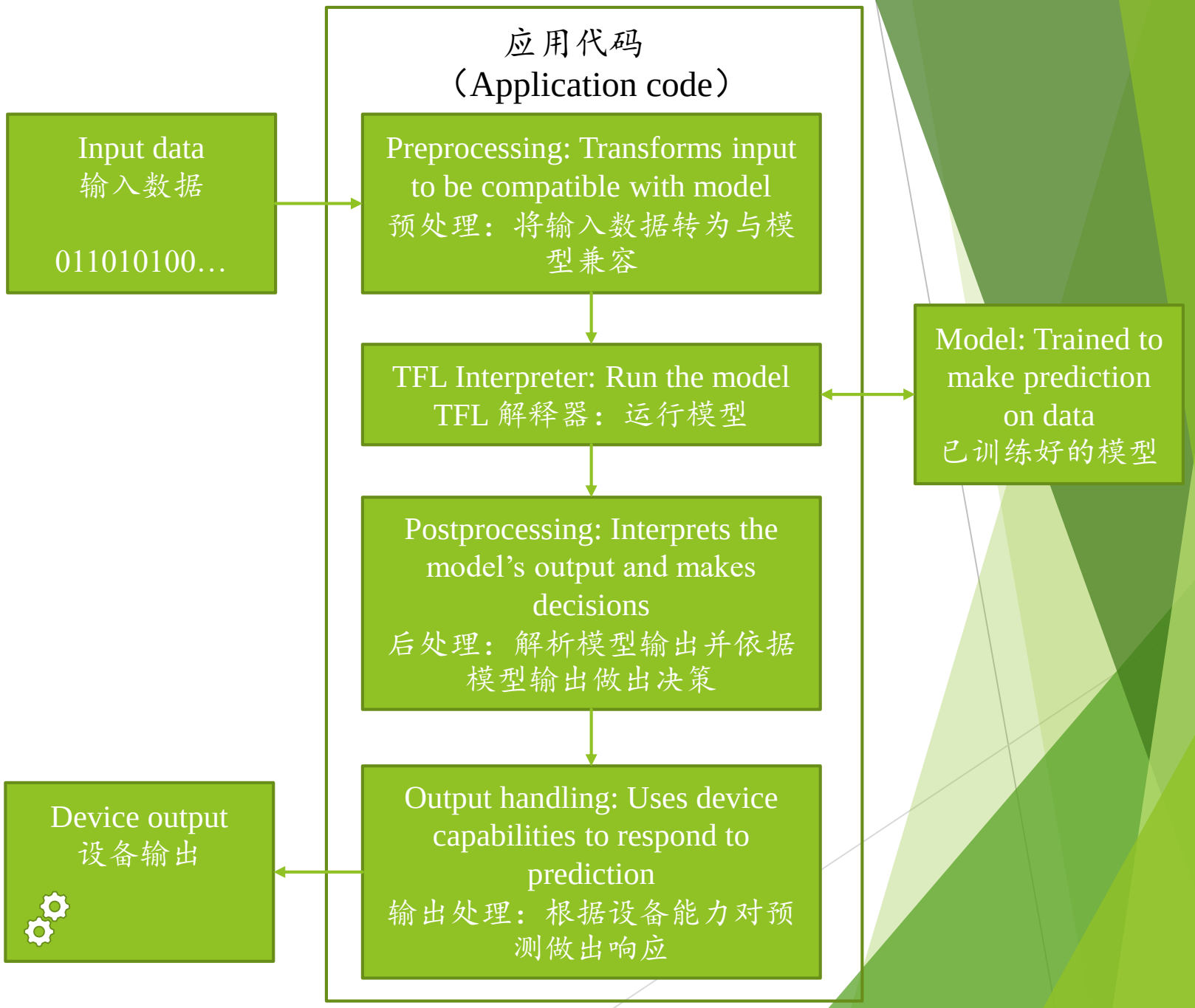
  When the model is ready, to deploy it to the device requires a series of steps such as programming, flashing/downloading, etc. Speaking of programming, users need to write pieces of codes for inference. The right diagram shows the actions need to take in sequence:



部署流程
Deployment

- 上页的流程更侧重于围绕模型论述，从系统的观点看上述工作的框图如右所示：

  The diagram in the last slide mainly depicts from the perspective of the model itself. The blocks show right are from the system perspective:



代码解析
Code Analysis

- 我们以具体代码为例子，讲解上述应用代码框架里各框图的具体实现。实际上，谷歌最开始构建TensorFlow Lite的目的是在Android手机上运行模型，但基于ARM的CPU与其他指令集的MCU相比，在一定程度上仍有不错的计算能力。为进一步适配MCU更加严格的计算限制，谷歌进一步开发了TensorFlow Lite for Microcontrollers。在此基础上，乐鑫为了进一步适配自家ESP32的硬件特性，将TensorFlow Lite for Microcontrollers客化到自家的产品上，创建了TensorFlow Lite for ESP32，其链接如下所示：

  We show by example to dissect the implementations of blocks which reside in the application code framework. Actually, the motivation for Google to initially launched the TensorFlow Lite project was to run models in Android mobile phones. But compared with MCUs based on proprietary instructions set, the ARM based CPU for mobile phones definitely have some superiority. However, to meet the rigorous computing constraints of MCUs, Google further developed TensorFlow Lite for Microcontrollers. To explore and harvest the hardware features and potentials of ESP32 families, Espressif ported TensorFlow Lite for Microcontrollers to their products and rename it as TensorFlow Lite for ESP32. The link is as follows:

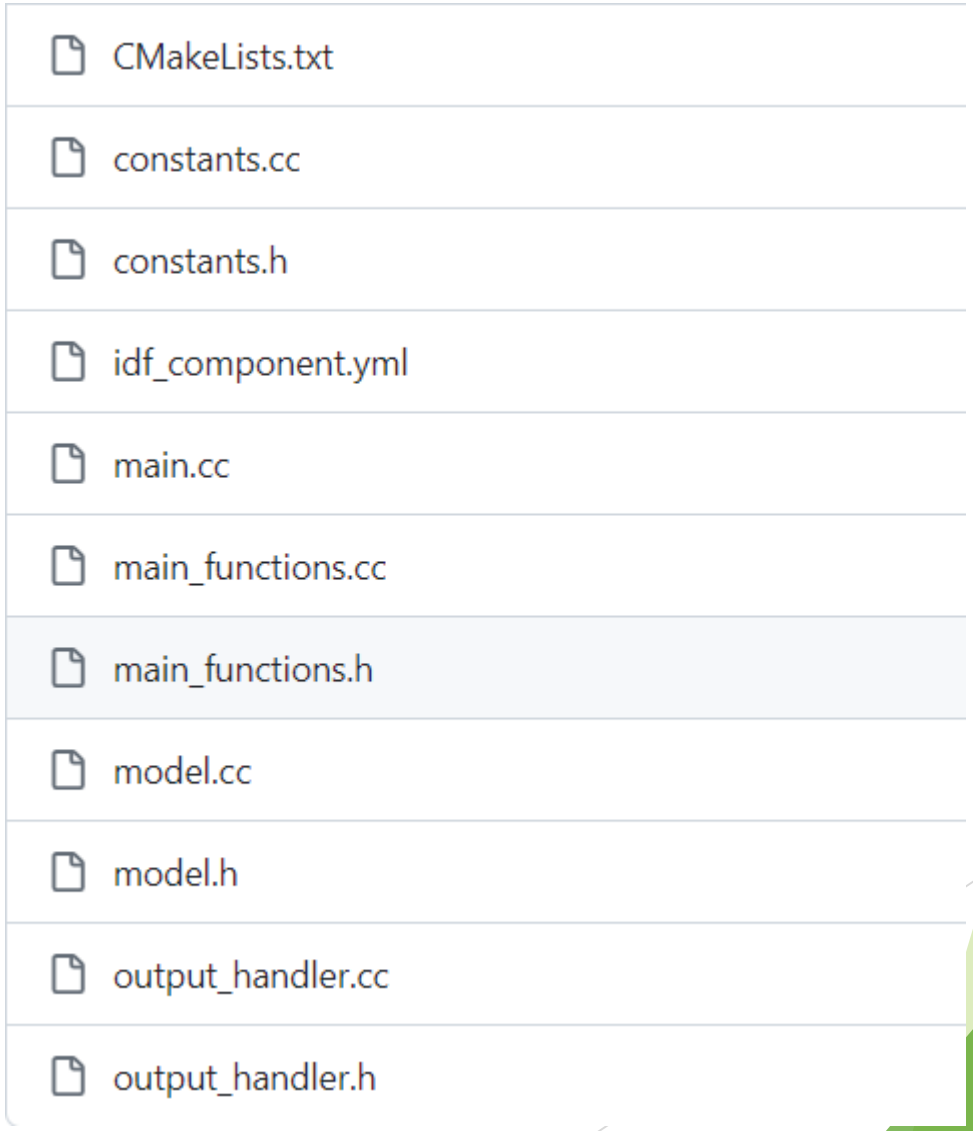
  https://github.com/tensorflow/tflite-micro

代码解析
Code Analysis

- 我们以hello_world项目代码为例，来讲解如何进行实践。该项目的链接地址与所包含的代码如下所示：

  We explain how to proceed by exemplification of the hello_world project. The link and the source files in this project are as follows:

  https://github.com/espressif/esp-tflite-micro/tree/master/examples/hello_world/



代码解析
Code Analysis

- 首先是model.h与model.cc两个文件。这两个文件内容原则上较为简单，其是由xxd工具直接生成的：

  The first are the two files model.h and model.cc. These two files are quite simple in principle, actually they are generated by the xxd command:

model.h中所含全局变量，其均由xxd命令生成↓
Global variables declared in model.h, all generated via the xxd command ↓

```
#ifndef TENSORFLOW_LITE_MICRO_EXAMPLES_HELLO_WORLD_MODEL_H_
#define TENSORFLOW_LITE_MICRO_EXAMPLES_HELLO_WORLD_MODEL_H_

extern const unsigned char g_model[];
extern const int g_model_len;

#endif  // TENSORFLOW_LITE_MICRO_EXAMPLES_HELLO_WORLD_MODEL_H_
```

model.cc中全局变量及其初始化，其均由xxd命令生成→
Global variables and their initializations in model.cc, all generated by the xxd command →

```
#include "model.h"

// Keep model aligned to 8 bytes to guarantee aligned 64-bit accesses.
alignas(8) const unsigned char g_model[] = {
    0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
    0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
    0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
    0x98, 0x00, 0x00, 0x00, 0xc8, 0x00, 0x00, 0x00, 0x1c, 0x03, 0x00, 0x00,
    0x2c, 0x03, 0x00, 0x00, 0x30, 0x09, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x60, 0xf7, 0xff, 0xff,
```

代码解析
Code Analysis

- 接下来是constants.h与constants.cc两个文件。这两个文件内容包含一些全局性的常量，是否放在单独的头文件与源文件中可能依据编程习惯：

  Now are the constants and constants.cc two files. These two files contain some global constants. Whether gathering all the constants into separate header and source files depends on the programming habit.

- 对于output_handler.h与output_handler.cc两个文件，其实质上是对TFL提供的MicroPrintf函数的简单包装，使得使用起来更简洁。

  For the output_handler.h and output_handler.cc two files, they are nothing more than just wrapper of the MicroPrintf function provided by TFL, to make the invoking simpler.

```
#include "output_handler.h"
#include "tensorflow/lite/micro/micro_log.h"

void HandleOutput(float x_value, float y_value) {
  // Log the current X and Y values
  MicroPrintf("x_value: %f, y_value: %f", static_cast<double>(x_value),
              static_cast<double>(y_value));
}
```

代码解析
Code Analysis

- 在深入解析剩余文件之前，我们先来看一下主文件main.cc。main.cc文件相当简单，只是按照RTOS实时操作系统的要求，定义了入口函数app_main，并调用两个典型例程setup与loop：

  Before diving into the remaining files, let's have a brief look of the main.cc file. Actually, it is quite simple, just conforming to the requirement of RTOS, one of the real-time operating system, by defining the app_main entry function, in which invoking the typical setup and loop functions:

```
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

#include "main_functions.h"

extern "C" void app_main(void) {
  setup();
  while (true) {
    loop();

    // trigger one inference every 500ms
    vTaskDelay(pdMS_TO_TICKS(500));
  }
}
```

代码解析
Code Analysis

- 下面，我们来分析两个最重要的文件，即main_functions.h与main_functions.cc，这两个文件主要是对main.cc中的两个典型例程或函数的实现，我们首先来看一下头文件所包含的内容，可以看出其是用extern “C” 包装的声明，即虽然在源代码中用了C++的语法，但链接时依然要链到C库：

  Now we turn to the two most important files, aka., the main_functions.h and main_functions.cc. These two source files mainly implements the two typical functions in main.cc, aka., setup() and loop(). We firstly check what’s contained in the header file. Actually it is quite simple, just the declarations of two functions wrapped by the extern “C” keywords. It means although C++ syntax is used to write the program, however, still the C library is linked against:

```
#ifdef __cplusplus
extern "C" {
#endif

// Initializes all data neede
// to be setup() for Arduino
void setup();

// Runs one iteration of data
// repeatedly from the applic
// compatibility.
void loop();

#ifdef __cplusplus
}
#endif
```

代码解析
Code Analysis

- 下面，我们主要看一下实现文件main_functions.cc。首先是包含文件部分，包括一些系统头文件与本地头文件：

  Now we turn to main_functions.cc , the implementation file. The first is the include section, which includes some system header files and local header files:

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"

#include "main_functions.h"
#include "model.h"
#include "constants.h"
#include "output_handler.h"
```

模型的运行依赖于特定于平台的算子，该头文件提供了相关算子的支持
The running of the model depends on the operators provided by specific MCUs. This header file specifies the operators supported by given platform

模型推断需要将加载的模型、所依赖的算子、输入输出张量结合起来，Interpreter对象类似TF里的Session对象，可以将相关对象结合起来进行推断
Model inference needs the combination of model loaded, operators implemented, input/output tensors allocated, etc. Just as the session object in TF, the interpreter object connects all together for inference.

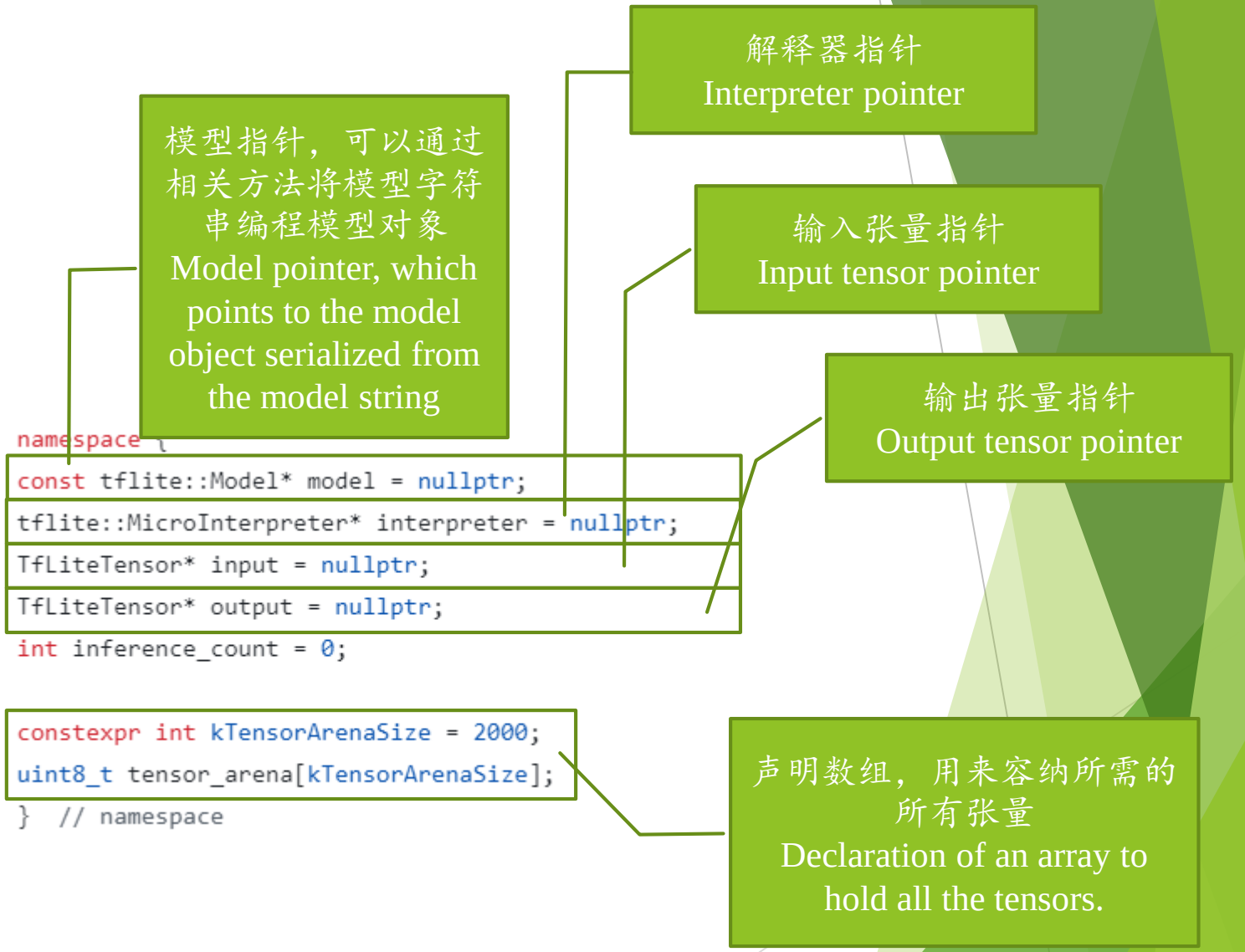
包含系统的初始化例程（特定于平台或MCU）
Contain initialization routines for system startup (pertaining to the platform or MCU).

包含对象序列化例程
Contain object serialization/de-serialization procedures

代码解析
Code Analysis

- 接下来，是变量声明部分，注意TFL的编程范式是以初始化的方式声明一个大的数组，然后模型所用的张量数据从该数组中分配，因此，大家可以看到在命名空间里变量声明的形式：

  Now is the variable declaration section. Note the programming paradigm of TFL is to declare a array sufficient enough to hold all the tensors involved, all the tensors demanded will be allocated from this array. This is the declaration we see inside the namespace enclosure:



代码解析
Code Analysis

- 接下来，我们静态实例化Interpreter对象，并将其赋给之前声明的Interpreter指针：

  Now we instantiate an Interpreter object in a static way and assign its address to the former declared Interpreter pointer variable:

创建interpreter对象，将模型、resolver对象、tensor对象关联
Instantiate the interpreter object, associating model, resolver, tensor object together

```
static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tenso
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
  MicroPrintf("AllocateTensors() failed");
  return;
}

// Obtain pointers to the model's input and output tensors.
input = interpreter->input(0);
output = interpreter->output(0);

// Keep track of how many inferences we have performed.
inference_count = 0;
}
```

为张量分配内存空间，并且测试是否成功。由于较难预测分配空间的大小，通常从较大值开始，然后通过降低该值并测试是否分配成功，来选择最佳值
Allocate the memory for tensors, and test it is successful or not. Usually, it is pretty hard at the beginning to guess the best value, so the practice is to gradually reduce the value to see whether it still succeeds or not in order to choose the optimal value

获取输入张量与输出向量
Acquire the tensors for input and output

代码解析
Code Analysis

- 本例中因为是通过全连接网络逼近正弦波函数，因此，需要通过一系列参数构建输入值，然后在相关工作就绪之后，进行推断：

  In this example it is to use fully-connected neural network to approximate the sine function, so we need to construct the input. When the preparation is done, just invoke the corresponding method for inference.

```
void loop() {
  // Calculate                                    compare the current
  // inference_                                   cycle to determine
  // our positi                                   es the model was
  // trained on, and use this to calculate a value.
  float position = static_cast<float>(inference_count) /
                   static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input->params.scale + input->params.zero_point;
  // Place the quantized input in the model's input tensor
  input->data.int8[0] = x_quantized;

  // Run inference, and report any error
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    MicroPrintf("Invoke failed on x: %f\n",
                         static_cast<double>(x));
    return;
  }
```

根据相关参数计算输入x的值
Calculate the x value based on some parameters

由于模型在转换的过程中被量化，因此输入也要被量化。
Since the model is quantized during converting, so the input.

当所有工作准备就绪之后，调用Interpreter对象的Invoke方法进行推断
When all are ready, call the Invoke method of the Interpretor object for inference

代码解析
Code Analysis

- 在推断之后，获取结果，然后根据对结果的处理，决定系统响应。这里采用的是较为简单的处理：

  After inference, to retrieve the result and based on the result to make the response according to the logic. Here is processing the inference result is quite simple:

获取结果，并进行反量化，得到输出值
Retrieve the result and do the dequantization to obtain the output value

处理输出，这里的处理非常简单，只是打印输出
Processing the output. Here it is quite simple, just print it out.

```
  // Obtain the quantized output from model's output tensor
  int8_t y_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y = (y_quantized - output->params.zero_point) * output->params.scale;

  // Output the results. A custom HandleOutput function can be implemented
  // for each supported hardware target.
  HandleOutput(x, y);

  // Increment the inference_counter, and reset it if we have reache
  // the total number per cycle
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle) inference_count = 0;
}
```

更新参数值，为下一次推断做准备。在实际应用中，输入一般是传感器获得的值
Update some parameters for the next round inference. In practice, the inputs are data from sensors