附录

附录A: 大模型推理与 llama.cpp 简介

欢迎来到实验的AI探索部分！在本部分，我们将初步接触人工智能（AI）在端侧设备（如我们的DAYU200开发板）上进行推理的概念和实践。由于大家可能之前没有系统学习过AI，我们会从最基本的概念开始。（由于我们是操作系统课程，所以不会很详细(°°)～）

1.1 人工智能（AI）原理简介

你可能经常听到“人工智能”或“AI”这个词，它听起来可能很复杂，但其核心思想其实并不难理解。

1.1.1 什么是人工智能?

简单来说，人工智能的目标是让计算机能够像人一样思考、学习、决策和解决问题。我们希望机器能够执行那些通常需要人类智能才能完成的任务，比如识别图像、理解语言、下棋、驾驶汽车等。

1.1.2 机器学习 (Machine Learning, ML) - AI的核心驱动力

目前实现AI最主要和最成功的方法之一是机器学习 (ML)。与传统编程中我们明确告诉计算机每一步该怎么做不同，机器学习的核心思想是让计算机从数据中“学习”。

- 打个比方： 想象一下教一个小孩识别猫。你不会去精确描述猫的每一根胡须或毛发的数学模型，而是会给她看很多猫的图片（这些图片就是“数据”）。通过观察足够多的例子，小孩的大脑会自己总结出猫的共同特征（比如有尖耳朵、胡须、特定的脸型等——这就是“学习”或“训练模型”的过程）。之后，当你给她看一张新的、她以前没见过的猫的图片时，她也能认出来（这就是“预测”或“推理”）。
- 机器学习就是让计算机做类似的事情：我们给计算机提供大量的数据（比如成千上万张标记为“猫”或“狗”的图片），并使用特定的算法，让计算机自动从这些数据中找出规律和模式，形成一个“模型 (Model)”。

1.1.3 深度学习 (Deep Learning, DL) - 更强大的机器学习

深度学习是机器学习的一个分支，它使用一种叫做“人工神经网络 (Artificial Neural Networks)”的复杂数学结构，特别是包含许多层（所以叫“深度”）的神经网络。这些网络的设计灵感来源于人脑神经元的连接方式。深度学习在处理复杂数据（如图像、语音、自然语言文本）方面表现得尤其出色，近年来许多AI的突破都得益于深度学习。我们本次实验接触的语言模型就是一种基于深度学习的大语言模型。

总而言之，AI（尤其是通过机器学习和深度学习）使计算机能够通过分析数据来学习和做出智能的响应，而不仅仅是执行预先编程好的指令。

1.2 什么是 AI 推理 (Inference)?

在机器学习（包括深度学习）的生命周期中，通常有两个主要阶段：

1. **训练 (Training)**：

- 这是“学习”的阶段，就像前面例子里小孩看图识猫，或者学生备考学习知识的过程。

- 在这个阶段，我们会用海量的标注数据（例如，数百万张图片及其对应的标签，或者TB级的文本数据）来“喂”给一个初始的、未学习的AI模型。
- 通过特定的学习算法，模型会调整其内部参数，以使其能够对输入数据做出正确的响应（例如，正确分类图片，或理解文本的含义）。
- 训练过程通常需要非常强大的计算资源（如高性能GPU集群）和很长的时间（几天、几周甚至几个月）。
- 训练的最终产出是一个“训练好的模型 (Trained Model)” ——可以把它看作是包含了从数据中学到的所有知识和规律的“大脑”文件。

2. **推理 (Inference)**：

- 这是“使用”或“应用”的阶段，就像小孩识别一张新的猫图片，或者学生运用所学知识解答考试题目。
- 在这个阶段，我们把新的、未曾见过的数据（例如，用户上传的一张照片，或者用户输入的一段文字）输入到已经训练好的模型中。
- 模型会利用它在训练阶段学到的知识和规律，对这些新数据进行处理，并给出一个预测、判断或输出（例如，识别出照片中的物体，或者根据输入的文字生成一段新的文字）。
- 推理过程通常比训练过程快得多，对计算资源的要求也低得多，使其可以在计算能力相对较弱的设备上运行。

本实验的核心在于“推理”：我们将使用一个已经训练好的型语言模型 (TinyStory/Qwen)，并在我们的DAYU200开发板上运行它，让它对我们输入的文本进行处理并生成回应。我们不涉及模型的训练过程。

1.3 什么是 AI 推理框架 (Inference Framework)?

当你有了一个训练好的AI模型后，要在实际设备上高效地运行它进行推理，直接操作原始模型文件可能会非常复杂和低效。这时，AI推理框架就派上了用场。

AI推理框架是一套专门设计用来简化和优化在各种硬件平台上部署和执行已训练模型进行推理的软件工具包或库。

它的主要作用包括：

- 模型加载与解析： 能够读取和理解不同格式的训练好的模型文件（例如，TensorFlow产生的.pb文件，PyTorch的.pt文件，MindSpore的.ms文件，或者像Llama.cpp使用的.gguf格式）。
- 性能优化： 针对目标硬件（如CPU、GPU、NPU——DAYU200上就有NPU）进行深度优化，以加快推理速度、减少内存占用和降低功耗。这可能包括图优化、算子融合、量化、使用特定硬件加速指令等技术。
- 硬件抽象： 为上层应用提供统一的API接口，屏蔽底层不同硬件的差异。开发者不需要针对每种硬件都写一套不同的代码。
- 跨平台支持： 很多框架支持将模型部署到多种操作系统（Windows, Linux, Android, iOS, OpenHarmony等）和多种硬件架构上。
- 易用性： 简化了模型部署的流程，让应用开发者可以更容易地将AI能力集成到自己的应用中。

1.4 我们实验中接触到的推理工具/框架：

- Llama.cpp: https://github.com/ggml-org/llama.cpp
    - 这是一个非常优秀的开源项目，专门用于在CPU上高效运行Llama系列及其他大型语言模型 (LLM)。
    - 它的主要特点是用纯C/C++编写，追求极致的性能和最小的依赖，非常适合交叉编译到各种端侧设备和嵌入式系统上（比如我们的DAYU200）。

    - 它支持多种量化技术，可以将原本非常庞大的LLM模型压缩到几GB甚至几百MB，同时尽量保持较好的性能，使得在普通PC和资源受限的设备上运行LLM成为可能。
    - 在本次实验中，我们将把Llama.cpp编译成一个库，并在OpenHarmony应用中调用它来实现语言模型的推理。
- MindSpore Lite：
    - 这是华为昇思MindSpore AI计算框架的轻量化版本，专门用于在端侧设备和物联网设备上进行高效推理。
    - 我们会提供一个使用MindSpore Lite的Demo。MindSpore Lite支持多种硬件后端（CPU、GPU、以及华为自研的NPU），能够加载MindSpore训练出的模型（.ms格式）以及转换自其他框架（如TensorFlow Lite、ONNX）的模型。
    - 它提供了Java和C++等多种语言的API，方便开发者在Android、iOS、OpenHarmony等系统上集成AI能力。

可以把推理框架想象成一个高度专业的“引擎室管理员”，它知道如何最高效地启动和运行一个复杂的“AI引擎”（即训练好的模型），并确保它在特定的“船只”（即你的硬件设备）上表现最佳。

1.5 端侧 AI (On-Device AI)推理 与应用开发简介

1.5.1 什么是端侧AI推理?

“端侧AI” (On-Device AI 或 Edge AI) 指的是直接在终端用户设备（如智能手机、平板电脑、智能手表、物联网设备、或者我们实验中的DAYU200开发板）上本地执行AI模型的推理过程，而不是将数据发送到远程的云服务器进行处理。

> 这也是我们课题组的一个研究方向，欢迎感兴趣的同学联系 **李永坤老师** ，然后加入我们(●'◡'●)

1.5.2 端侧AI推理的优势

- 低延迟 (Low Latency)： 数据处理在本地进行，无需网络传输，响应速度更快，对于需要实时反馈的应用（如实时语音识别、动态手势识别）至关重要。
- 隐私保护 (Privacy Protection)： 敏感数据（如个人照片、录音）无需上传到云端，保留在用户本地设备上，更好地保护了用户隐私。
- 离线可用 (Offline Capability)： 即使在没有网络连接或网络不稳定的情况下，AI功能依然可以正常使用。
- 节省带宽与成本 (Bandwidth & Cost Saving)： 减少了大量数据上传下载所需的网络带宽，也可能降低了对云端计算资源的依赖，从而节省成本。

> 其实最主要的是省钱💴还有隐私保护。

1.5.3 端侧AI推理的挑战：

- 资源限制： 端侧设备的计算能力（CPU、GPU、NPU性能）、内存、存储空间以及电池续航都远不如云服务器。
- 模型大小与效率： 需要对AI模型进行高度优化和压缩（如量化），使其能够在资源受限的设备上高效运行，同时尽量不损失太多精度。

> 在本次实验中，我们将体验到量化对模型大小和性能的影响,也会体会到内存和计算能力带来的限制。

1.5.4 端侧AI应用开发流程（简化版）

1. 选择或训练AI模型： 根据应用需求选择合适的预训练模型（例如，是图像分类、目标检测、还是自然语言理解）。在本次实验中，我们直接使用已经训练好的TinyStories模型。
2. 优化与量化： 对模型进行量化（如INT8量化），减小模型大小，提高在端侧设备上的运行效率。
3. 应用集成与开发：

- 在你的应用程序代码中（对于Llama.cpp，我们将在C++层面操作），调用推理框架提供的API：
- 加载优化后的模型文件。
- 准备输入数据（例如，对图像进行预处理，或将用户文本编码成模型需要的格式）。
- 执行推理，获取模型输出。
- 对模型输出进行后处理，并呈现给用户或用于后续逻辑。

> 在本次实验中，我们将会提供完整的推理代码，同学们只需要编译出推理框架Llama.cpp的动态链接库放到项目代码中即可，如果你对推理的具体逻辑感兴趣，也可以阅读源代码来调整应用的逻辑。

附录B: 编写自己的库

第一部分中，我们学习了如何使用一个第三方动态链接库。同学们可能会好奇，我们如何编写自己的库，并发布给别人呢？本节附录就来简单介绍该过程。为了作为简单起见，我们将演示如何编写只包含一个函数 `minus_numbers_in_lib` 的库 `mylib` 。

为了实现该库，我们需要两个文件 `mylib.cpp` 和 `mylib.h` ，前者提供函数的实现，后者定义函数的接口。

1. 创建源代码文件：
   打开终端，创建一个名为 `mylib.cpp` 的文件，在该文件中编写一个简单的减法函数如下：

   ```
   // 使用 extern "C" 可以防止 C++ 编译器的名称修饰 (name mangling),
   // 使得这些函数更容易被 C 语言或其他语言调用, 或者在不同 C++ 编译器间保持一致性。
   // 对于纯 C++ 内部使用, 且主程序也用同一编译器编译时, 这并非总是必需, 但作为库导出是一种良好实践。
   extern "C" {
       double minus_numbers_in_lib(double a,double b) {
           return a - b;
       }
   }
   ```

2. 为库创建头文件(mylib.h):

   在同一文件夹创建 `mylib.h` ，声明库中的函数:

   ```
   #ifndef MYLIB_H
   #define MYLIB_H

   #ifdef __cplusplus
   extern "C" {
   #endif

   // 声明库中导出的函数
   double minus_numbers_in_lib(double a,double b)

   #ifdef __cplusplus
   }
   #endif

   #endif // MYLIB_H
   ```

3. 将源代码交叉编译为动态链接库 (.so 文件):

   使用以下命令即可：

   ```
   g++ -shared -fPIC mylib.cpp -o libmylib.so
   ```

   为了让其他人更方便使用你的库，你可以将该编译过程写入 `Makefile` 。当你的库更复杂，有着更多配置选项时，你可以编写 `CMakeLists.txt` 让 `cmake` 为你生成 `Makefile` 。当然，你还可以让 AI 来帮你编写 `CMakeLists.txt` ..... (套娃中)。

   `Makefile` 、 `CMakeLists.txt` 等的编写超出了讨论范围，你可以自行查询。

附录C: llama-Demo.cpp代码说明

本附录旨在简要介绍实验中提供的 llama-Demo.cpp 示例程序。这个程序是一个基于命令行的C++应用，它调用 Llama.cpp 库（即我们编译的 libllama.so）来加载一个大型语言模型（LLM），并根据用户输入的提示（prompt）进行文本推理生成。

理解这个示例程序的工作流程有助于我们了解如何通过API与第三方库进行交互，以实现复杂的功能。

主要函数说明

以下是对 llama-Demo.cpp 中关键函数功能的简要说明。为简洁起见，部分实现细节将用 ... 表示。

- 头文件导入

```
#include "llama.h" // 引入Llama.cpp库的公共头文件

#include <cstdio>   // 用于标准输入输出，如printf
#include <cstring>  // 用于字符串操作，如strcmp
#include <string>   // 用于std::string类
#include <vector>   // 用于std::vector类
```

- 打印程序使用方法与命令行参数解析

```
// 打印程序使用方法
static void print_usage(int, char ** argv) {
    ...
}

// 解析命令行参数
void ParseArgs(int argc, char** argv,
            std::string& model_path, std::string& prompt, int& n_predict){
                ...
}
```

- 加载GGUF模型文件

```
llama_model* LoadModel(const std::string& model_path) {
    // 设置模型加载参数
    llama_model_params model_params = llama_model_default_params();
    // model_params.n_gpu_layers = 0; // 可设置GPU层数，0表示纯CPU。代码中是99，表示尝试全GPU。
    model_params.n_gpu_layers = 99;
    // 加载模型
    llama_model* model = llama_model_load_from_file(model_path.c_str(), model_params);
    if (model == nullptr) {
        fprintf(stderr, "%s: error: unable to load model from '%s'\n", __func__,
model_path.c_str());
        exit(1); // 加载失败则退出
    }
    printf("Model loaded successfully: %s\n", model_path.c_str());
    return model;
}
```

- 对用户提示进行分词 (Tokenization)。

```
std::vector<llama_token> TokenizePrompt(const llama_vocab* vocab, const std::string& prompt) {
    // 先估算最大token数量，然后实际进行分词
    // llama_tokenize的参数: vocab, 输入字符串, 长度, 输出buffer, buffer大小, 是否加BOS, 是否特殊处理
    // ... (如代码中所示，先调用一次获取长度，再调用一次填充) ...
    const int n_tokens_estimated = -llama_tokenize(vocab, prompt.c_str(), prompt.size(),
nullptr, 0, true, true);
    std::vector<llama_token> prompt_tokens(n_tokens_estimated);
    int n_tokens_actual = llama_tokenize(vocab, prompt.c_str(), prompt.size(),
prompt_tokens.data(), prompt_tokens.size(), true, true);

    if (n_tokens_actual < 0 || n_tokens_actual > n_tokens_estimated ) { // 出错或数量不对
        fprintf(stderr, "%s: error: failed to tokenize the prompt or size mismatch\n",
__func__);
        exit(1);
    }
    prompt_tokens.resize(n_tokens_actual); // 调整为实际token数
    return prompt_tokens;
}
```

- 初始化上下文与采样器

```
// 初始化推理上下文(context)
llama_context* InitializeContext(llama_model* model, int n_prompt_tokens, int n_predict) {
    llama_context_params ctx_params = llama_context_default_params();
    // 设置上下文窗口大小，必须能容纳提示和生成的最大token数
    ctx_params.n_ctx = n_prompt_tokens + n_predict;
    ctx_params.n_batch = n_prompt_tokens; // 初始批处理大小可以设为提示的长度
    // ...
    ctx_params.no_perf = false; // 开启性能计数

    llama_context* ctx = llama_init_from_model(model, ctx_params);
    if (ctx == nullptr) {
        fprintf(stderr, "%s: error: unable to create Llama context\n", __func__);
        exit(1);
    }
    return ctx;
}

// 初始化采样器(sampler)
llama_sampler* InitializeSampler() {
    auto sparams = llama_sampler_chain_default_params();
    sparams.no_perf = false; // 开启性能计数
    // ... (可以修改sparams来调整采样策略，例如温度、top_k, top_p等) ...
    llama_sampler* smpl = llama_sampler_chain_init(sparams);
    if (smpl == nullptr) {
        fprintf(stderr, "%s: error: Failed to create sampler chain\n", __func__);
        exit(1);
    }
    // 向采样链中添加一个贪心采样器 (总是选择概率最高的token)
    // 也可以添加其他采样器，如 llama_sampler_init_top_k(), llama_sampler_init_top_p() 等
    llama_sampler_chain_add(smpl, llama_sampler_init_greedy());
    return smpl;
}
```

- 文本生成函数

```
void GenerateTokens(std::vector<llama_token>& prompt_tokens, llama_context* ctx,
                    const llama_vocab* vocab, llama_sampler* smpl,
                    int n_prompt /*初始提示token数*/, int n_predict /*要生成的最大token数*/) {

    // 1. 准备包含提示的初始批处理 (batch)
    llama_batch batch = llama_batch_get_one(prompt_tokens.data(), n_prompt, 0, 0);
    int n_current_pos = 0; // 当前序列中已处理（解码）的token数量

    // 2. 主生成循环
    // 循环直到达到预测长度，或者遇到序列结束符(EOG)，或者解码失败
    // 代码中循环条件是: n_pos + batch.n_tokens < n_prompt + n_predict
    // 这里的n_pos在代码中是外部循环变量，与batch.n_tokens结合控制总长度。
    for (int n_pos = 0; n_pos + batch.n_tokens < n_prompt + n_predict; ) {
        // 2a. 解码当前批次 (处理tokens)
        // 第一次循环时，batch中是完整的prompt；后续循环时，batch中是上一步生成的单个token。
        if (llama_decode(ctx, batch) != 0) { // 返回0代表成功
            fprintf(stderr, "%s : failed to eval tokens, return code %d\n", __func__, 1);
            break;
        }
        n_pos += batch.n_tokens; // 更新已处理的token总数

        // 2b. 从模型输出中采样下一个token
        llama_token new_token_id = llama_sampler_sample(smpl, ctx, -1); // -1表示从当前上下文最后
一个有效token的logits采样

        // 2c. 检查是否是生成结束标记 (End Of Generation)
        if (llama_vocab_is_eog(vocab, new_token_id)) {
            // printf("\n[EOG]"); // 遇到结束符，停止生成
            break;
        }

        // 2d. 将新生成的token转换为文本并打印
        char buf[128];
        int n = llama_token_to_piece(vocab, new_token_id, buf, sizeof(buf), 0, true);
        if (n < 0) {
            fprintf(stderr, "%s: error: failed to convert token to piece\n", __func__);
        }
        std::string s(buf, n);
        printf("%s", s.c_str());
        fflush(stdout);
        // 2e. 准备下一个批处理，只包含新生成的token，用于下一次迭代
        batch = llama_batch_get_one(&new_token_id, 1);
    }
}
```

附录D: Native C++ 应用中 C++ 函数定义

我们在正文中调用了 add 函数，号称其会调用实现在 `entry/src/main/cpp` 里的加法函数。可具体怎么实现的呢?

如正文所说，项目C++相关代码，存放在`entry/src/main/cpp`目录中，查看该目录，发现目录下有名叫`napi_init.cpp`的C++文件。打开文件，内容如下（省略部分内容）：

```
#include "napi/native_api.h"

static napi_value Add(napi_env env, napi_callback_info info)
{
    // ...
    return sum;
}

EXTERN_C_START
static napi_value Init(napi_env env, napi_value exports)
{
    napi_property_descriptor desc[] = {
        { "add", nullptr, Add, nullptr, nullptr, nullptr, napi_default, nullptr }
    };
    napi_define_properties(env, exports, sizeof(desc) / sizeof(desc[0]), desc);
    return exports;
}
EXTERN_C_END

// ...
```

文件里有两个奇怪的函数，一个叫`Add`，看名字是个加法，但内容很复杂，参数和返回值都看不明白。一个叫`Init`只看得出里面有一行`"add"`什么的。

实际上，这里的`Add`函数，就是我们实际调用的 C++ 函数。而`Init`函数中的下面这行，就是告诉应用，`"add"`这个`ArkTS`函数，对应到了`Add`这个C++函数。

```
// 这个数组是 napi_property_descriptor 结构体数组
// 每一行代表一个ArkTS函数和C++函数的映射。其它地方我们不管，第一个"add"代表ArkTS函数的名字（字符串），第三个"Add"代表C++函数（不是字符串，而是函数名，或者相当于函数指针。）
    napi_property_descriptor desc[] = {
        { "add", nullptr, Add, nullptr, nullptr, nullptr, napi_default, nullptr }
    };
```

再仔细看`Add`函数的内部，调用了很多`napi_`开头的函数。实际上，这些函数就是负责把`add`函数的ArkTS格式的参数，转换成C++能够读取的参数。例如：

```
    double value0;
    napi_get_value_double(env, args[0], &value0);
```

这两行，将`add`函数的第一个参数（`args[0]`），转换为C++中的`double`，并存放在`value0`变量中。

而下面两行相反：

```
    napi_value sum;
    napi_create_double(env, value0 + value1, &sum);
```

创建一个类型为`double`的ArkTS变量`sum`，将`value0 + value1`的值，存入`sum`中。（后续该值被返回，作为`add`函数调用的结果。）

> 如果你还好奇，这些转换是怎么进行的。你可以发现，这些函数都是 `napi_` 开头的，是不是和我们用的 `llama_` 系列函数有点像？对的，这个C++文件引用了 `#include "napi/native_api.h"` 这个头文件。这实际上是OpenHarmony提供的Native API库的一部分。这个C++文件，实际上就是调用了OpenHarmony SDK中的 Native API 库中的函数，实现了 C++ 变量和ArkTS变量间的转换。
>
> 至于具体转换的原理，作为库的使用者，我们是不需要掌握的。这就是调用库的好处。

实际上，只在C++中定义这个函数还不行，我们还需要在ArkTS中声明这个函数存在，这部分代码在 `entry/src/main/cpp/types/libentry/Index.d.ts` 文件中，只有一行：

```
export const add: (a: number, b: number) => number;
```

这里是ArkTS的函数定义的语法，说明我们“导出”了一个 `add` 函数，有两个参数 `a` 和 `b` ，它们都是数字，函数的返回值也是数字。简单说，这行类似下面的C代码：

```
extern const int add(int a, int b);
```

是不是理解了呢？就是定义了一个函数罢了。

这样两边都定义之后，通过NAPI库在后台的一系列操作，就能实现ArkTS应用调用C++啦！

附录E: llama.cpp的使用

我们在实验中为了实验的体验，使用了极小的小故事文本生成模型，名字叫做 TinyStory ,其由于模型太小，所以产生文本的效果有点差。

那么同学就要问了，有没有表现更好一点的模型呢？答案是：有的兄弟，有的。

我们推荐qwen2的0.5B模型（当然其他的GGUF模型也可以，不过要考虑开发板只有2G运存）。

E.0 创建交换分区

在运行大型应用程序，尤其是像Llama.cpp这样的大语言模型时，我们可能会遇到开发板物理内存（RAM）不足的情况。为了让系统在这种情况下依然能够运行（尽管性能可能会有所下降），我们可以配置“交换分区”。

什么是交换分区?

交换分区 (Swap Space)，常被称为“虚拟内存”的一部分，是操作系统在硬盘或其他持久性存储设备（在DAYU200上通常是eMMC内部存储）上划分出来的一块区域。

- 工作原理： 当系统的物理内存 (RAM) 即将耗尽时，操作系统会将RAM中一些暂时不活跃的数据（称为“内存页”）临时移动到交换空间中，从而释放出物理内存给当前更需要的活动进程使用。当那些被移到交换空间的数据再次需要被访问时，操作系统会再将它们从交换空间读回到RAM中。
- 目的：
  - 扩展可用内存： 使得系统能够运行比实际物理内存更大的程序或同时运行更多的程序。
  - 防止内存溢出 (Out-Of-Memory, OOM)： 在物理内存完全用尽时，避免系统因无法分配内存而直接崩溃或杀死重要进程。

如何创建交换分区?

```
# hdc shell 进入开发板执行
cd data/llama_file

# 在2GB的dayu200上加swap交换空间
```

```
# 新建一个空的ram_ohos文件
touch ram_ohos
# 创建一个用于交换空间的文件（8GB大小的交换文件）
dd if=/dev/zero of=/data/ram_ohos bs=1G count=8
# 设置文件权限，以确保所有用户可以读写该文件：
chmod 777 /data/ram_ohos
# 将文件设置为交换空间：
mkswap /data/ram_ohos
# 启用交换空间：
swapon /data/ram_ohos
# 查看内存空间：
free -m
```

E.1 使用qwen2的0.5B模型

> 如果你只是想要尝试一下，并不想自己构建，直接下载睿客网盘中第二阶段素材中的Demo即可（**https://rec.ustc.edu.cn/share/dfbc3380-2b3c-11f0-aee2-27696db61006** 中的 `qwen-Demo.hap` ）

1. 下载qwen2的0.5B模型，下载地址： **https://huggingface.co/Qwen/Qwen2-0.5B-Instruct-GGUF/tree/main** ，下载其中qwen2-0_5b-instruct-q4_0.gguf即可
2. 将其放入 `entry/src/main/resources/resfile` 文件夹下
3. 修改 `entry/src/main/ets/pages/Index.ets` ，修改其中的 `modelName` 变量为:

   ```
   @State modelName: string = 'qwen2-0_5b-instruct-q4_0.gguf';
   ```

4. 重新构建应用运行即可（由于加载模型时间过长，可能会存在appfreeze导致应用闪退，这是正常现象，真正感兴趣联系助教）效果如下所示：