中软国际
基于昇腾云的模型迁移与调优
专业服务解决方案
使用指南

文档版本 01

发布日期 **2025-06-25**

目 录

1 方案概述 3

2 资源和成本规划 6

3 实施步骤 8

3.1 阶段一：场景调研评估 9

3.2 阶段二：迁移方案设计 10

3.3 阶段三：环境搭建准备 10

3.4 阶段四：模型训练适配 18

3.5 阶段五：模型训练调优 28

3.6 阶段六：推理模型迁移 40

3.7 阶段七：推理推理优化 47

修订记录 56

1 方案概述

应用场景

随着人工智能技术的快速发展，市场对算力的需求呈现出爆发式增长的态势。昇腾计算凭借其强大的性能和完善的生态，成为众多企业的重要选择。昇腾 AI 芯片在各行业大模型训练、推理场景中都发挥了重要作用，包括智慧政务、智慧医疗、智能制造、智能矿上等行业领域，昇腾凭借其卓越的性能成为各个产业用户的重要选择。

昇腾计算凭借其强大的性能、完善的生态和灵活的服务，已成为企业数字化转型的重要选择。但在企业依托昇腾算力实现智能化升级的过程中，常因技术栈差异面临软硬件兼容性和使用困难，由于缺乏对华为昇腾 AI 平台的深入了解，遇到技术问题时难以解决，无法充分利用有效的利用算力资源，影响了企业数智化循转型的进程。

企业基于昇腾云的模型迁移和调优服务的业务需求如下：

- 模型技术多样，适配难度大：

  用户在采购昇腾资源或昇腾云服务后，由于各版本已有模型面临的计算资源架构差异出现的兼容性和性能问题，涉及算子层、框架层、模型层等多技术体系，导致问题层出不穷，客户技术难以解决，影响开发进展。

- 缺乏专业技术团队：

  目前针对昇腾 CANN 与相关 AI 框架的人才资源较少，在昇腾环境下模型迁移和调优过程中，一般 AI 算法人员不了解昇腾工具链与框架技术，难以准确定位问题与解决底层问题疑难，缺乏经验与技术手段，造成项目停滞不前。

- 缺少全面的昇腾迁移调优方法和经验

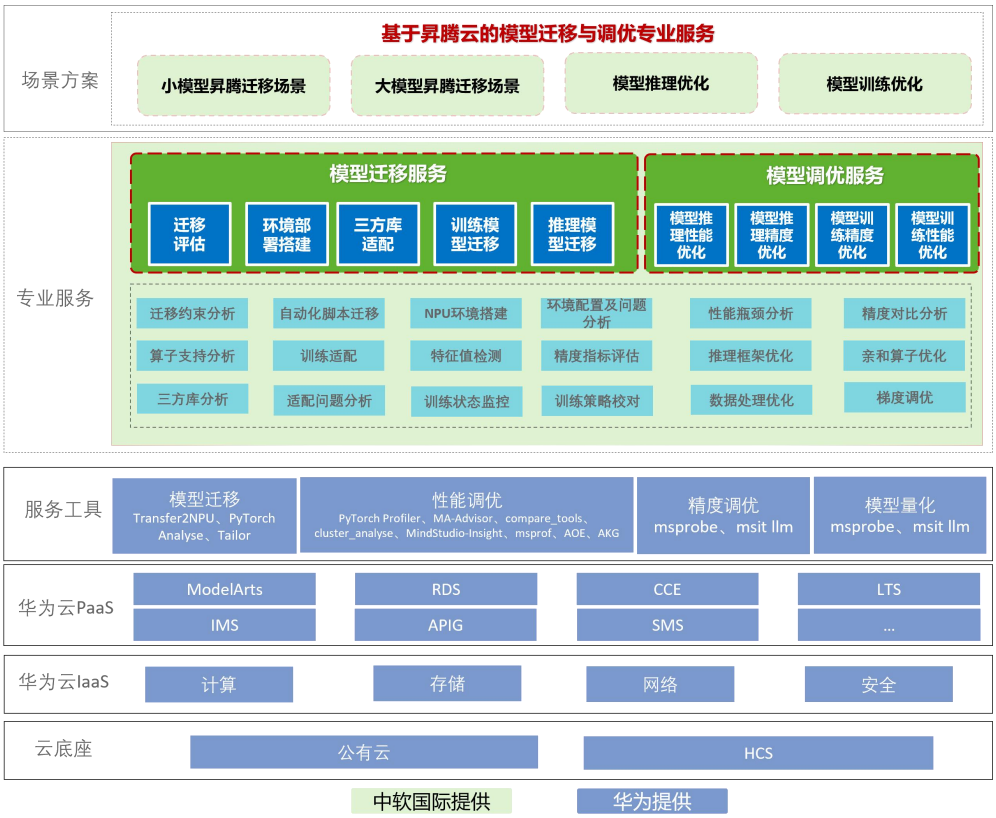
  企业在利用昇腾云资源对现有模型进行适配和调优的过程中，往往缺乏成熟的实践经验和科学的实施方法体系，导致各类、各版本模型无法在昇腾算力下达到最佳的性能与精度，不能发挥出昇腾算力高性价比的优势。

方案架构

中软国际基于昇腾云的模型迁移与调优专业服务解决方案架构如下图所示，本解决方案提供模型从训练到推理部署过程中，所需的方案评估设计、环境搭建、代码迁

移适配、性能精度优化及测试验证等全流程技术支持，提供端到端的实施交付服务，加速企业 AI 应用昇腾上云，实现智能化转型第一步。

图 1 业务架构



中软国际基于昇腾云的模型迁移和调优专业服务，依托于华为云昇腾生态，是由中软国际昇腾专业服务团队打造，解决大、小模型从英伟达向昇腾云环境迁移适配的场景技术难题，提供基于昇腾环境的模型推理优化、模型训练优化等场景技术服务解决方案。

昇腾迁移实现从 GPU 平台到 NPU 平台的模型迁移，通过自动化迁移工具和云服务，解决客户传统模型迁移过程中面临的复杂技术难题。

昇腾调优服务通过多种性能、精度调优手段，借助于华为的专业调式工具，帮助客户解决模型在 NPU 环境下运行的精度和性能问题。

基于昇腾云的迁移和调优服务，结合华为云主机迁移、数据迁移及数据库迁移等高阶服务，有效保障客户数据安全、运行环境稳定，实现高效适配和优化的目标。

主要提供以下服务：

- 基于昇腾云的模型迁移服务：为客户提供迁移评估分析、环境部署搭建、三方库适配、训练模型迁移与推理模型迁移的整体解决方案。
- 基于昇腾云的模型调优服务：基于昇腾云环境，通过华为昇腾工具链与各种技术手段，为用户提供模型训练和模型推理任务中性能、精度分析对比、优化测评等实施服务，保障用户智能应用的高效运行。

方案优势

- 一站式全面昇腾迁移与调优服务：

熟练掌握昇腾云迁移核心技术，通过昇腾迁移“六步法”标准实施流程，实现从调研规划、方案设计、模型训练适配、模型调优、推理部署、集成运维全流程迁移能力及实施方案，满足企业国产化算力的昇腾迁移适配建设需求。

- 降低企业 AI 开发成本：

通过昇腾专业云服务能力和高性价比特性，有效降低 AI 模型训练推理成本，帮助企业在实现智能化转型过程中，既满足了数据安全、可信与 AI 模型自主可控的需求，又优化了企业算力资源的优化配置，实现高效管理与运营。

- 成熟工具链保障：

服务深度绑定华为昇腾全栈工具链，团队全员精通昇腾训练迁移 Transfer2NPU、性能调优（AOE、msprof）、精度调试 msprobe、模型量化 modelslim、压缩 AMCT 及转换 ATC 等工具使用，并参与部分工具开发。；

- 云专业服务团队：

中软国际拥有 150+人员 AI 技术团队，其中专职昇腾算法服务专家 40 人，全员持有 HCCDP-AI/HCCDP-AI Ascend/Ascend C 微认证等证书，15 人 HCIE 证书，13 人 PMP 证书。曾获 2024 昇腾 AI 创新大赛陕西赛区企业赛道铜奖，2024 年度昇腾金种子开发者等荣誉。

约束与限制

无

2 资源和成本规划

以某智能电力公司为例，该公司在智能电力领域提供物联终端和数字化解决方案。该项目对变电站智能巡视业务系统的站内人员安全风险识别模块、设备状态检测模块、缺陷检测模块等 10 多个模型，进行相应的模型适配、精度校验、代码适配、测试验证等服务。

根据客户现有模型规模、数据量，以及考虑到业务的扩展、以及客户对平台性能、业务高可靠的要求，设计了以下的资源与成本清单。实际收费应以账单为准：

资源和成本规划：

资源和成本规划

云资源	规格	数量	每月费用（元）
VPC	网段选择 172.16.0.0/16，其他采用默认配置	1	00.00
Subnet	网段选择 172.16.0.0/24，其他采用默认配置	1	00.00
安全组	按需开放，最小化原则 ，如数据库根据需要开通入方向 3306 等端口	1	00.00
ECS	规格：8vCPUs | 16GB | c7.2xlarge.2 操作系统：Linux 区域：华北-北京四 VPC 名称：VPC-BJ4	5	4500.00
BMS	规格：8*Ascend 910|CPU: 192 核 768GB 操作系统：鲲鹏 区域：华北-北京四 VPC 名称：VPC-BJ4	1	45000
EVS	通用型 SSD | 500GB	10	3500
EIP	按需计费 | 加入共享带宽	5	72
ELB	独享型 | 中型 I | 8,000 HTTP / 800 HTTPS 新建连接数 | 800,000 并发连接数 | 16,000 每秒查询 | 200 Mbit/s 带宽 | 40 LCUs	1	1715
OBS	标准存储单 AZ 存储包 | 5TB	1	456.00

云资源	规格	数量	每月费用（元）
DMS	rocketmq.4u8g.cluster.small | 代理个数：1 超高 IO | 300GB	1	4080
DCS	基础版 | 5.0 | 主备 | X86 | DRAM | 2 | 512 MB	1	33.75
SMS	主机迁移服务	1	0
OMS	对象存储迁移服务	1	0
DRS	数据库迁移服务	1	0
RDS	MySQL 5.7 | 单机通用 | 2 vCPUs | 4GB | 40GB	1	196.00
共享带宽	按带宽计费 | 200Mbit/s 带宽大小 (Mbit/s)	1	16000
DNS	域名解析	1	0
DDoS 防护	保底防护带宽：10Gbit/s | 业务带宽：100Mbit/s | 防护域名数：50 个	1	8820
企业主机安全	专业版	1	90
Web 应用防火墙	标准版	1	3880
总计：92414.45 (每月)			

专业服务清单

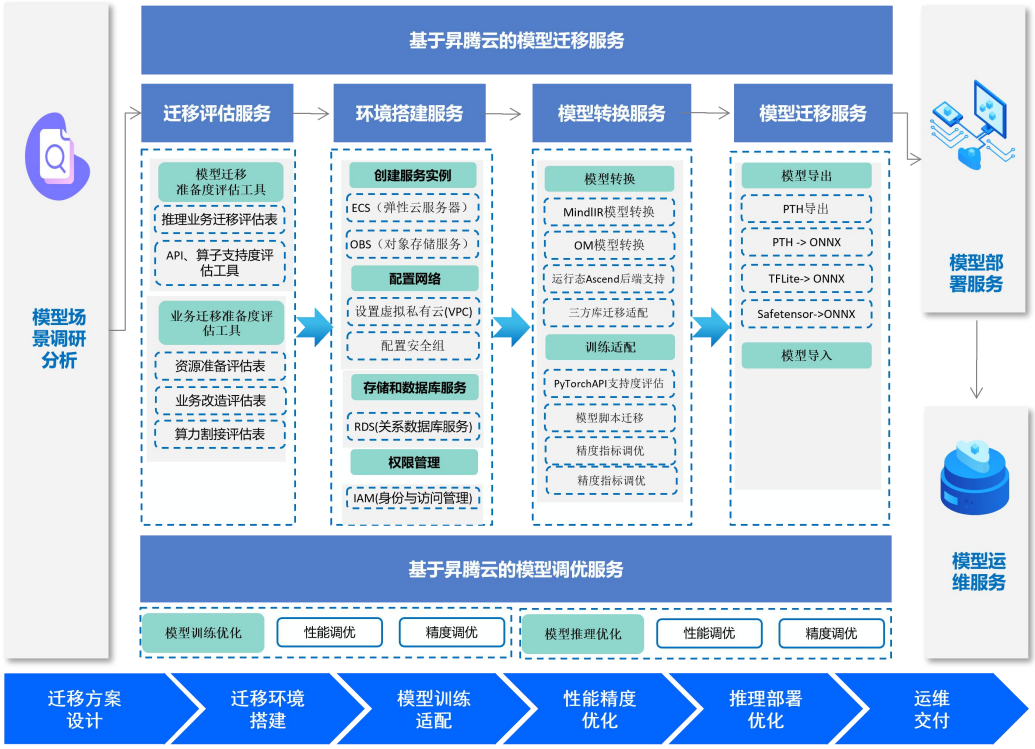
本案例所涉及的上云专业服务报价项如下，实际以收费账单为准：

类别	报价项	量纲
昇腾模型迁移实施服务	模型迁移调研评估	元/人天
	环境部署搭建	元/人天
	三方库适配	元/人天
	推理迁移	元/人天
昇腾模型调优实施服务	性能评估分析	元/人天
	精度对比分析	元/人天
	推理框架优化	元/人天
	数据处理优化	元/人天
合计：		

3 实施步骤

中软国际昇腾服务技术团队，通过熟练掌握昇腾云迁移核心技术，打造昇腾迁移“六步法”标准实施流程，实现从调研规划、方案设计、模型训练适配、模型调优、推理部署、集成运维全流程迁移能力及实施方案，满足企业国产化算力的昇腾迁移适配建设需求，企业上云全流程，深刻赋能企业数字化转型。

企业基于昇腾云模型迁移与调优服务整体的流程图，如下：



3.1 阶段一：场景调研评估

3.1.1 迁移咨询评估

昇腾迁移咨询内容包含不仅限于：算力资源选型、云产品功能特性、问题/需求解决方案以及迁移服务流程等。

现网业务调研：通过收集、梳理客户现网信息，能够对模型迁移准备度进行评估（包括算子覆盖度、三方加速的依赖库、性能基线等）。

迁移评估：具备对模型迁移的技术可行性、迁移难度、资源需求、风险、昇腾云收益进行评估的能力。

3.1.2 信息/需求梳理和收集

对迁移目标模型/系统现状信息进行全方面收集。包含（但不限于）：算力服务器信息、模型版本、应用架构、部署拓扑等。以及借助迁移期望达成的其他目标。

3.1.3 整体分析

对梳理收集上来的信息进行整体分析评估，包含（但不限于）：业务分析、模型分析、技术分析、数据分析、项目分析、调用关系、指标关系、技术提升改造需求、成本分析等。

3.1.4 风险问题评估

对迁移可能碰到的问题和风险进行评估。包含（但不限于）：模型版本适配、模型环境依赖、数据集、AI 推理框架、性能精度指标、网络环境、迁移时间成本及人工成本等。

3.1.5 迁移策略评估

根据各平台/服务/组件特点和实际需要综合评估分析选择昇腾迁移与调优策略。

3.2 阶段二：迁移方案设计

3.2.1 迁移规划

基于调研报告和可行性分析，能够独立完成迁移方案设计，制定迁移计划，明确迁移所需的资源及模型验收标准。

整体训练模型迁移优化设计需考虑模型框架、数据集、源代码、硬件配置、软件版本、模型收敛情况（精度、性能）、评测方案；

推理模型优化设计需考虑框架、模型（ONNX）、硬件配置、软件版本、精度性能指标，评测方案等因素。

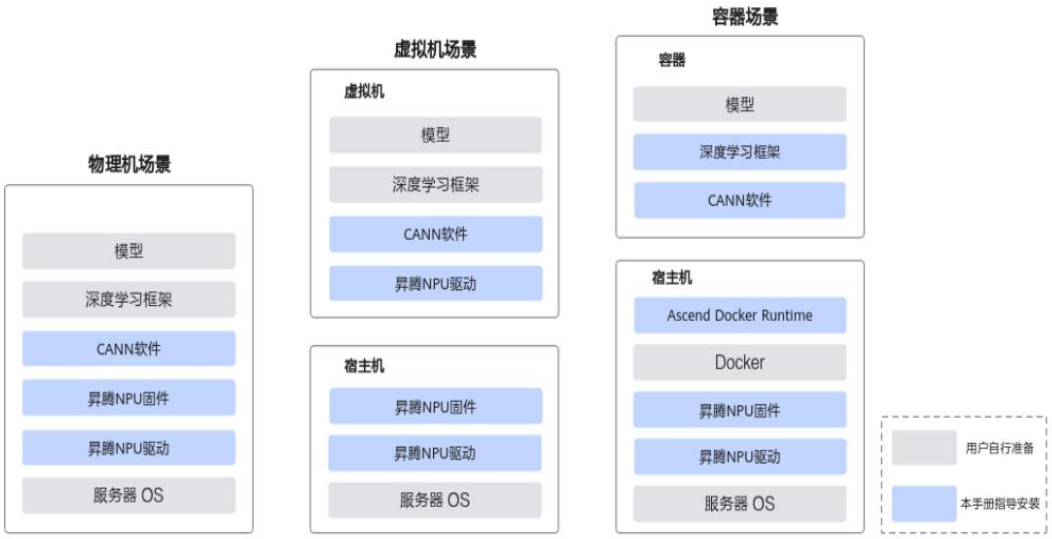
3.2.2 方案设计

提供实际客户案例的需求梳理、需求分析、方案设计等过程文档或案例交付件。

3.3 阶段三：环境搭建准备

ModelArts 开发环境针对推理昇腾迁移的场景提供了云上可以直接访问的开发环境，首先，利用云服务的资源使用便利性，直接使用到不同规格的昇腾设备；其次，通过指定对应的运行镜像，可以直接使用预置的，在迁移过程中所需的工具集，且已经适配到最新的版本可以直接使用；第三，开发者可以通过浏览器入口一 Notebook 方式访问，也可以通过 VSCode 远程开发的模式直接接入到云上环境中完成迁移开发与调测，最终生成适配昇腾的推理应用。

昇腾迁移环境包含物理机、容器、虚拟机场景下，安装驱动、固件和 CANN 软件的方案，部署架构如图所示：



3.3.1 安装软件包清单

软件类型	软件包说明	软件包名称
昇腾NPU驱动	部署在昇腾AI处理器，用于管理查询昇腾AI处理器，同时为上层CANN软件提供处理器控制、资源分配等接口。	Ascend-hdk-<*chip_type*>-npu-driver_<*version*>_linux-<*arch*>.run
昇腾NPU固件	固件包含昇腾AI处理器自带的OS 、电源器件和功耗管理器件控制软件，分别用于后续加载到AI处理器的模型计算、处理器启动控制和功耗控制。	Ascend-hdk-<*chip_type*>-npu-firmware_<*version*>.run
Toolkit	**CANN开发套件包，在训练&推理&开发调试场景**下安装，主要用于训练和推理业务、模型转换、算子/应用/模型的开发和编译。	Ascend-cann-toolkit_<*version*>_linux-<*arch*>.run
Kernels	**CANN二进制算子包**，包括单算子API执行（例如aclnn类API）动态库/静态库文件，以及kernel二进制文件。 **使用场景：** • 单算子API执行（例如aclnn类API）场景下必须安装该软件包。 • 图模式动态shape场景下，建议安装该软件包，安装后可提升编译性能。 • 图模式静态shape场景下，若安装该软件包，安装后可以提升编译性能。若不安装该软件包，按照确定shape编译新的kernel，可以提升算子执行性能。 安装时需已安装Toolkit软件包，请选择运行设备对应产品类型和架构的Kernels软件包。	Ascend-cann-kernels-<*chip_type*>_<*version*>_linux-<*arch*>.run 针对Atlas A3 训练系列产品[*]和Atlas A3 推理系列产品[*]，请获取：Atlas-<*type*>-cann-kernels_<*version*>_linux-<*arch*>.run
NNAL （Ascend Neural Network Acceleration Library）	**CANN神经网络加速库**，包含面向大模型领域的ATB（Ascend Transformer Boost）加速库，可以提升大模型训练和推理性能。 安装时需已安装Toolkit软件包。	Ascend-cann-nnal_<*version*>_linux-<*arch*>.run

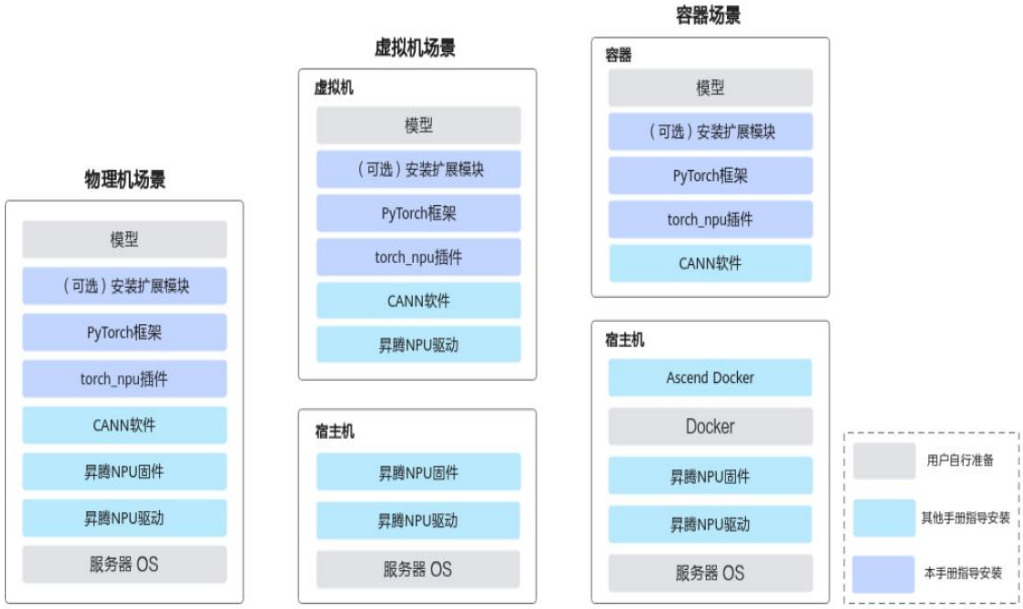
3.3.2 深度学习框架安装

1. MindSpore 系统环境和第三方依赖安装清单

软件名称	版本	作用
Ubuntu 18.04 / CentOS 7.6 / EulerOS 2.8 / openEuler 20.03 / KylinV10 SP1	-	编译和运行MindSpore的操作系统
Python	3.9-3.11	MindSpore的使用依赖Python环境
昇腾AI处理器配套软件包	-	MindSpore使用的Ascend平台AI计算库
GCC	7.3.0	用于编译MindSpore的C++编译器

2. Pytorch 框架安装

安装驱动、固件、CANN 软件、PyTorch 框架和 torch_npu 插件的方案，部署架构如图所示



3. TenserFlow 框架安装

需要安装 TensorFlow 才可以进行算子开发验证、训练业务开发。

- 对于 x86 架构：直接从 pip 源下载即可，系统要求等具体请参考 TensorFlow 官网。需要注意 TensorFlow 官网提供的指导描述有误，从 pip 源下载 CPU 版本需要显式指定 tensorflow-cpu，如果不指定 CPU，默认下载的是 GPU 版本。安装命令参考如下：

  如下命令如果使用非 root 用户安装，需要在安装命令后加上--user，例如：**pip3 install tensorflow-cpu --user**

```
pip3 install tensorflow-cpu==2.6.5
```

- 对于 aarch64 架构：由于 pip 源未提供对应的版本，所以需要用户使用官网要求的 linux_gcc7.3.0 编译器编译 TensorFlow，编译步骤参考官网 TensorFlow 官网。特别注意点如下。

在下载完 tensorflow tag 源码后需要执行如下步骤：

1) 下载“nsync-1.22.0.tar.gz”源码包。

a. 进入源码目录，打开“tensorflow/workspace2.bzl”文件，找到其中 name 为 nsync 的“tf_http_archive”定义。

```
b.    tf_http_archive(
c.        name = "nsync",
d.        sha256 =
"caf32e6b3d478b78cff6c2ba009c3400f8251f646804bcb65465666a9cea93c4",
e.        strip_prefix = "nsync-1.22.0",
f.        system_build_file =
clean_dep("//third_party/systemlibs:nsync.BUILD"),
g.        urls =
[          "https://storage.googleapis.com/mirror.tensorflow.org/github.com/google/nsync/archive/1.22.0.tar.gz",
h.            "https://github.com/google/nsync/archive/1.22.0.tar.gz",
i.        ],
```



```
    )
```

j. 从 urls 中的任一路径下载 nsync-1.22.0.tar.gz 的源码包，保存到任意路径。

2) 修改“nsync-1.22.0.tar.gz”源码包。

a. 切换到 nsync-1.22.0.tar.gz 所在路径，解压缩该源码包。解压缩后存在“nsync-1.22.0”文件夹和“pax_global_header”文件。

b. 编辑“nsync-1.22.0/platform/c++11/atomic.h”。

在 NSYNC_CPP_START_内容后添加如下加粗字体内容。

```
    #include "nsync_cpp.h"

    #include "nsync_atomic.h"

    NSYNC_CPP_START_

    #define ATM_CB_() __sync_synchronize()

    static INLINE int atm_cas_nomb_u32_ (nsync_atomic_uint32_ *p, uint32_t
o, uint32_t n) {

        int result = (std::atomic_compare_exchange_strong_explicit
(NSYNC_ATOMIC_UINT32_PTR_ (p), &o, n, std::memory_order_relaxed,
std::memory_order_relaxed));

        ATM_CB_();

        return result;

    }

    static INLINE int atm_cas_acq_u32_ (nsync_atomic_uint32_ *p, uint32_t o,
uint32_t n) {

        int result = (std::atomic_compare_exchange_strong_explicit
(NSYNC_ATOMIC_UINT32_PTR_ (p), &o, n, std::memory_order_acquire,
std::memory_order_relaxed));

        ATM_CB_();

        return result;

    }

    static INLINE int atm_cas_rel_u32_ (nsync_atomic_uint32_ *p, uint32_t o,
uint32_t n) {

        int result = (std::atomic_compare_exchange_strong_explicit
(NSYNC_ATOMIC_UINT32_PTR_ (p), &o, n, std::memory_order_release,
std::memory_order_relaxed));

        ATM_CB_();

        return result;

    }

    static INLINE int atm_cas_relacq_u32_ (nsync_atomic_uint32_ *p,
uint32_t o, uint32_t n) {
```



```
        int result = (std::atomic_compare_exchange_strong_explicit
(NSYNC_ATOMIC_UINT32_PTR_ (p), &o, n, std::memory_order_acq_rel,
std::memory_order_relaxed));
        ATM_CB_();
        return result;
    }
```

3) 重新压缩“nsync-1.22.0.tar.gz”源码包。
将上个步骤中解压出的内容压缩为一个新的“nsync-1.22.0.tar.gz”源码包，保存（比如，保存在“/tmp/nsync-1.22.0.tar.gz”）。
4) 重新生成“nsync-1.22.0.tar.gz”源码包的 sha256sum 校验码。

执行如下命令后得到 sha256sum 校验码（一串数字和字母的组合）。

```
sha256sum /tmp/nsync-1.22.0.tar.gz
```

5) 修改 sha256sum 校验码和 urls。

进入 tensorflow tag 源码目录，打开“tensorflow/workspace2.bzl”文件，找到其中 name 为 nsync 的“tf_http_archive”定义，其中“sha256=”后面的数字填写 4 得到的校验码，“urls=”后面的列表第二行，填写存放“nsync-1.22.0.tar.gz”的 file://索引。

```
tf_http_archive(
    name = "nsync",
    sha256 =
"caf32e6b3d478b78cff6c2ba009c3400f8251f646804bcb65465666a9cea93c4",
    strip_prefix = "nsync-1.22.0",
    system_build_file = clean_dep("//third_party/systemlibs:nsync.BUILD"),
    urls = [

"https://storage.googleapis.com/mirror.tensorflow.org/github.com/google/nsy
nc/archive/1.22.0.tar.gz",
        "file:///tmp/nsync-1.22.0.tar.gz ",
        "https://github.com/google/nsync/archive/1.22.0.tar.gz",
    ],
)
```

6) 参考官方文档（https://www.tensorflow.org/install/source）完成编译。

说明

ABI 的配置在 TensorFlow 和 FwkPlugin 中需要保持一致（即配置为 0），否则会导致导入 TFA 失败。

7) 安装编译好的 TensorFlow。

以上步骤执行完后会打包 TensorFlow 到指定目录，进入指定目录后执行如下命令安装：

如下命令如果使用非 root 用户安装，需要在安装命令后加上--user，例如：pip3 install tensorflow-*.whl --user

```
pip3 install tensorflow-*.whl
```

8) 执行如下命令验证安装效果。

```
python3 -c "import tensorflow as tf;
print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

如果返回了张量则表示安装成功。安装 TensorFlow 时会自动重装 numpy，导入时如果提示 numpy 版本不兼容，请参考安装 TensorFlow 后，执行 import tensorflow 时报错重装配套版本的 numpy。

4. MindSpeed 安装

MindSpeed 是专为华为昇腾设备设计的大模型加速解决方案。在当前 AI 领域，大模型训练因其复杂性高、技术挑战多而备受关注。特别是在内存资源受限的情境下，如何高效地进行大模型训练成为业界关注的焦点。针对这一需求，MindSpeed 应运而生，以其卓越的性能表现和深度优化的算法体系，助力用户在昇腾设备上高效实现大模型训练。

1) 下载 MindSpeed 源码 2.0.0_core_r0.8.0 分支。

```
git clone -b 2.0.0_core_r0.8.0 https://gitee.com/ascend/MindSpeed.git
```

2) 安装 MindSpeed。

```
pip3 install -e MindSpeed
```

说明

如有旧版本 MindSpeed，请先卸载，再进行安装操作。

3) 获取 Megatron-LM 并指定 core_r0.8.0 分支。

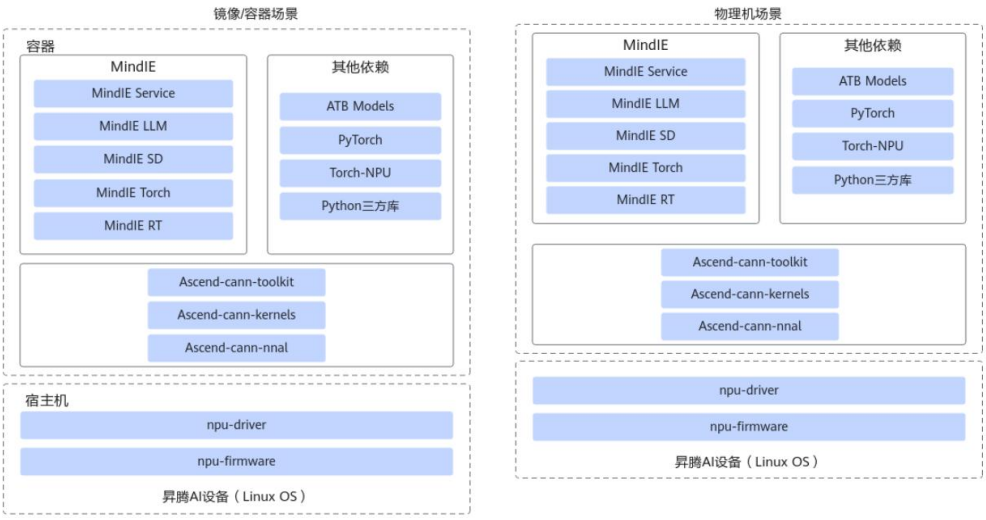
```
git clone https://github.com/NVIDIA/Megatron-LM.git

cd Megatron-LM

git checkout core_r0.8.0
```

3.3.3 MindIE 推理引擎安装

MindIE（Mind Inference Engine，昇腾推理引擎）软件的安装，安装方案图如下：



1、 获取 MindIE 容器镜像

1） 单击 MindIE 容器镜像链接，进入到社区版资源下载页面。

2） 单击 ATB Models 后方的“下载”按钮，弹出 MindIE 镜像获取页面。

3） 在 MindIE 镜像页面的“镜像版本”页签申请权限（使用华为账号登录，如果没有请先注册），填入如图 1 所示信息，单击“提交申请”按钮。

4） 待申请审批后，根据用户环境单击对应镜像后方的“立即下载”按钮。

5） 根据弹出的下载页面提示，下载镜像，操作指导如图所示。

2、 使用镜像

1) 执行以下命令启动容器。

```
docker run -it -d --net=host --shm-size=1g \
    --name <container-name> \
    --device=/dev/davinci_manager \
    --device=/dev/hisi_hdc \
    --device=/dev/devmm_svm \
    -v /usr/local/Ascend/driver:/usr/local/Ascend/driver:ro \
    -v /usr/local/sbin:/usr/local/sbin:ro \
    -v /path-to-weights:/path-to-weights:ro \
    mindie:2.0.RC2-800I-A2-py311-openeuler24.03-lts bash
```

说明

容器启动命令仅供参考，可根据需求自行修改。

2) 执行以下命令进入容器。

```
docker exec -it <container-name> bash
```

3、 使用模型进行推理。

以 LLaMA3 系列模型为例，具体可参考容器中“$ATB_SPEED_HOME_PATH/examples/models/llama3/README.md”中的说明。

执行以下命令进行推理：

```
cd $ATB_SPEED_HOME_PATH
python examples/run_pa.py --model_path /path-to-weights  # 请修改权重路径
```


显示默认问题“Question”和推理结果“Answer”，如下所示：

2024-11-18 11:08:13,291 [INFO] [pid: 389497] logging.py-180: Question[0]: What's deep learning?

2024-11-18 11:08:13,291 [INFO] [pid: 389497] logging.py-180: Answer[0]: Deep learning is a subset of machine learning that uses neural networks to learn from data. Neural networks are

2024-11-18 11:08:13,291 [INFO] [pid: 389497] logging.py-180: Generate[0] token num: (0, 20)

若用户想要自定义输入问题，可使用“--input_texts”参数设置，如：

python examples/run_pa.py --model_path /path-to-weights --input_texts "What is deep learning?" # 请修改权重路径

说明

“$ATB_SPEED_HOME_PATH”已在“.bashrc”中设置好，无需自行设置。

4、 启动服务

MindIE Service 是面向通用模型场景的推理服务化框架，通过开放、可扩展的推理服务化平台架构提供推理服务化能力，支持对接业界主流推理框架接口，满足大语言模型的高性能推理需求。

以下为简易的启动方法：

1. 修改“$MIES_INSTALL_PATH/conf/config.json”

2. 使用后台进程方式启动服务：

cd $MIES_INSTALL_PATH

nohup ./bin/mindieservice_daemon > output.log 2>&1 &

3. 在标准输出流捕获到的文件中，打印如下信息说明启动成功：

Daemon start success!

3.4 阶段四：模型训练适配

3.4.1 Pytorch 模型迁移

通用模型迁移适配方法，可以分为四个阶段：迁移分析、模型迁移、精度调试与性能调优，总体流程如下图所示。

1、 迁移分析

（1）模型选取与约束说明

1) 调研业务需求场景，参考模型选取选取主流仓库模型。

2) 保证选取的模型在第三方平台（如 GPU）可成功运行。

3) 明确迁移前模型运行的硬件型号、精度、性能基线。从权威网站或数据平台获取源模型的性能值基线，或在三方平台实测性能基线。

（2）迁移支持度分析

1） 准备 NPU 环境，获取模型的源码、权重和数据集等文件。

2） 使用迁移分析工具采集目标网络中的模型/算子清单，识别第三方库及目标网络中算子支持情况，分析模型迁移的可行性。

3）模型迁移需要符合模型选取与约束说明。

4）算子开发与适配：在迁移支持度分析中如果存在平台未支持的算子，需进行算子替换或者开发适配。

2、 模型迁移

1）模型脚本迁移

通过模型脚本迁移，实现 GPU -> NPU 的接口替换、NPU 分布式框架改造。

2）环境变量和脚本配置

训练前环境变量配置，配置训练相关环境变量，以保证模型训练可以在 NPU 上正常运行。

模型脚本与启动脚本配置，根据实际场景选择相应操作完成模型脚本和启动脚本配置。

3）关键特性适配

数据类型为 **BF16 或 FP32** 的模型在训练过程中出现的收敛异常，可开启特征值检测，用于检测在训练过程中的梯度特征值是否存在异常，具体可参考特征值检测。

在训练时如需混合使用单精度（float32）与半精度（float16）数据类型，可参考混合精度适配。

4）模型调试

训练过程中，如果遇到问题，可以通过模型调试定位问题发生的代码位置。

常见问题发生场景包括环境配置、脚本配置、硬件配置与集群配置，可从以上场景角度排查问题。

5）模型保存与导出

模型保存与导出用于在线或离线推理。

保存模型文件用于在线推理。

使用模型文件导出 ONNX 模型，通过 ATC 工具将其转换为适配昇腾 AI 处理器的.om 文件，用于离线推理。

3.4.2 TenserFlow 模型迁移

昇腾 AI 处理器进行模型迁移之前，建议用户事先准备好基于 TensorFlow 开发的训练模型以及配套的数据集，并要求在 GPU 或 CPU 上跑通，精度收敛，且达到预期精度和性能要求。同时记录相关精度和性能指标，用于后续在昇腾 AI 处理器进行精度和性能对比。

1、 安装依赖

```
pip3 install pandas==1.3.5
```



```
pip3 install openpyxl
pip3 install google_pasta
```

2、 训练脚本扫描和自动迁移。

进入迁移工具所在目录“${TFPLUGIN_INSTALL_PATH}/npu_device/convert_tf2npu/”，执行命令可同时完成脚本扫描和自动迁移，例如：

```
python3 main.py -i /root/models/examples/test -m
/root/models/example/test/test.py
```

- 迁移过程中，打印如下信息，表明正在扫描相关文件进行脚本迁移。

迁移过程信息

- 迁移结束后，生成迁移后的脚本，以及迁移报告。

迁移结束信息

- 如果没有生成 failed_report.txt，一般迁移后的模型即可直接在昇腾 AI 处理器执行训练，用户可尝试执行训练，如果训练失败，可详细分析迁移报告，同时酌情修改训练脚本再次训练，如果仍然训练失败，请单击 Link 联系技术支持。
- 如果生成了 failed_report.txt，请优先根据报错修改训练脚本，再次执行训练。

3、 迁移报告说明

- success_report.txt：记录工具对脚本的全部修改点，例如：

```
# 表示 adain.py 第 3 行新增头文件引用
/root/models/examples/adain/adain.py:3 import npu_device as npu
# 表示 adain.py 第 4 行新增 npu 虚拟设备初始化
/root/models/examples/adain/adain.py:4 npu.open().as_default()
```

- failed_report.txt：记录迁移过程中的报错信息以及不支持的 api，例如：

```
Finish conver file: /root/ast_test/hvd/model_lib.py
/root/ast_test/hvd/test.py:3, NPU Unsupport API: hvd.allreduce
```

- api_analysis_report.xlsx：为 API 支持度分析报告，用户可以筛选**“不支持”**API 单独分析，并根据修改建议修改训练脚本。

API 支持度分析报告举例

序号	脚本文件名	代码行	模块名	API名	API支持度
1	adain.py	138	tf.io	tf.io.read_file	分析中
2	adain.py	139	tf.image	tf.image.decode_jpeg	支持
3	adain.py	140	tf.image	tf.image.convert_image_dtype	支持
4	adain.py	141	tf.image	tf.image.resize	支持
5	adain.py	156	tf.image	tf.image.convert_image_dtype	支持
6	adain.py	157	tf.image	tf.image.resize	支持
7	adain.py	316	tf.nn	tf.nn.moments	分析中
8	adain.py	317	tf	tf.sqrt	支持
9	adain.py	474	tf	tf.GradientTape	支持
10	adain.py	586	tf.keras	tf.keras.preprocessing.image	不涉及
11	adain.py	589	tf.keras	tf.keras.preprocessing.image	不涉及
12	adain.py	593	tf.keras	tf.keras.preprocessing.image	不涉及

表 2 工具迁移 API 支持度说明

工具迁移 API 支持度	说明
支持（无需迁移）	此类 API 在昇腾 AI 处理器上绝对支持，无需适配修改。 例如"tf.abs"等接口，在昇腾 AI 处理器上能够完全支持，不需要迁移。
工具迁移后 API 功能支持	工具迁移后，该 API 在昇腾 AI 处理器上可以支持。
工具迁移后训练功能打通	工具迁移后，能够保证在昇腾 AI 处理器训练执行成功，但原有 API 功能可能不完全支持。 例如 "tf.compat.v1.config.experimental.set_memory_growth"接口，它的作用一般是将所有 GPU 设置为仅在需要时申请显存空间，在昇腾 AI 处理器上训练时，该接口实际并不生效。 因此，迁移工具会直接 return 返回 None，从而保证在昇腾 AI 处理器上训练正常执行。 "tf.compat.v1.config.experimental.set_memory_growth" --> 直接 return 返回 None。
不支持（不影响迁移，无需干预）	此类 API 在昇腾 AI 处理器上不支持，但不影响脚本执行，无需用户干预。 例如 "tf.compat.v1.config.experimental.get_memory_growth"接口，由于工具会将 "tf.compat.v1.config.experimental.set_memory_growth"直接 return 返回 None，因此对应的 get 接口也不会影响脚本在昇腾 AI 处理器上的执行，即便出现这个接口，用户也无需干预。
不支持（无迁	此类 API 在昇腾 AI 处理器上不支持，且当前暂无具体迁移方

工具迁移API支持度	说明
移方案，建议不使用）	案，建议您不要使用，否则会引起训练失败。 例如"tf.distribute.TPUStrategy"等TPU相关接口，需通过Google TPU设备执行，昇腾AI处理器上不支持，建议用户不要使用。
废弃类	此类API在TensorFlow 2.6版本已经废弃，建议用户使用TensorFlow官网推荐的API，否则可能会引起训练失败。 例如"tf.compat.v1.layers.conv3d "等接口。此类API在TensorFlow 2.6版本已经废弃，建议用户使用TensorFlow官网推荐的API，否则可能会引起训练失败。
分析中	此类API在昇腾AI处理器中的支持度未知，正在分析中。

- api_brief_report.txt：汇总脚本中API支持度统计结果，例如：

```
# 未去重的统计结果，分类和API支持度表中的一致
1.In brief: Total API: 231, in which Support: 222, Unsupport: 2,No operator is involved: 0, Analysing: 0
# 去重后的统计结果，分类和API支持度表中的一致
2.After eliminate duplicate: Total API: 98, in which Support: 92, Unsupport or recommended: 1,No operator is involved: 0, , Analysing: 0
```

3.4.3 MindSpeed 大模型迁移

按照如下步骤操作，即可实现Megatron-LM在昇腾设备上的高效运行，且无缝集成并充分发挥MindSpeed所提供的丰富加速与优化技术。

1. 在“Megatron-LM”目录下修改**pretrain_gpt.py**文件，在“import torch”下新增一行“import mindspeed.megatron_adaptor”代码，如下黑体加粗部分。

```
import os
import torch
import mindspeed.megatron_adaptor
from functools import partial
from typing import Union
```

2. 数据准备，参考Megatron-LM官方文档准备训练数据。
   a. 下载Tokenizer。

新建“Megatron-LM/gpt-tokenizer”目录，并将 vocab.json 和 merges.txt 文件下载至该目录。

b. 下载数据集。

以 Alpaca 数据集为例，可单击 Link 获取。

3. 配置环境变量，当前以 root 用户安装后的默认路径为例，请用户根据 set_env.sh 的实际路径执行如下命令。

```
source /usr/local/Ascend/ascend-toolkit/set_env.sh
```

4. 数据处理，详情可单击对应分支 core_r0.8.0 进行参考。

a. 语料格式转换。

数据处理依赖于多个第三方库，请确保已正确安装如下依赖：

```
pip3 install nltk pyarrow pandas
```

以下代码段展示了如何读取 Parquet 格式的原始语料，并将其转换为 JSON 格式，以便后续处理。

```
import json

import pandas as pd

data_df = pd.read_parquet("train-00000-of-00001-a09b74b3ef9c3b56.parquet")

data_df['text'] = data_df['text'].apply(lambda v: json.dumps({"text": v}))

with open("alpaca_json.json", encoding='utf-8', mode='w') as f:

    for i, row in data_df.iterrows():

        f.write(row['text'])

        f.write('\n')
```

b. 预训练数据集生成。

若在昇腾设备上使用 **preprocess_data.py** 脚本处理数据，须在“Megatron-LM”目录下修改“tools/preprocess_data.py”脚本，在“import torch”下新增一行“import mindspeed.megatron_adaptor”代码，如下黑体加粗部分。

```
import torch

import mindspeed.megatron_adaptor

import numpy as np
```


新建“Megatron-LM/gpt_pretrain_data”目录，通过运行 preprocess_data.py 脚本，可以将转换后的 JSON 文件进一步处理为适合 Megatron-LM 预训练的二进制格式。

```
python tools/preprocess_data.py \
    --input alpaca_json.json \
    --output-prefix ./gpt_pretrain_data/alpaca \
    --tokenizer-type GPT2BPETokenizer \
    --vocab-file ./gpt-tokenizer/vocab.json \
    --merge-file ./gpt-tokenizer/merges.txt \
    --append-eod \
    --log-interval 1000 \
    --workers 8
```

执行成功后，将在 gpt_pretrain_data 目录下生成两个文件：alpaca_text_document.bin 和 alpaca_text_document.idx，代表预处理完成的预训练数据集。

5. 在“Megatron-LM”目录下准备预训练脚本 train_distributed.sh，脚本示例如下：

```
#!/bin/bash
export CUDA_DEVICE_MAX_CONNECTIONS=1
NPUS_PER_NODE=8
MASTER_ADDR=localhost
MASTER_PORT=6001
NNODES=1
NODE_RANK=0
WORLD_SIZE=$(($NPUS_PER_NODE*$NNODES))
CKPT_DIR=./ckpt
VOCAB_FILE=<Specify path to file>/vocab.json
MERGE_FILE=<Specify path to file>/merges.txt
DATA_PATH=<Specify path and file prefix>_text_document
```



```
TP=2

PP=2

CP=1

EP=1

DISTRIBUTED_ARGS="

    --nproc_per_node $NPUS_PER_NODE \

    --nnodes $NNODES \

    --node_rank $NODE_RANK \

    --master_addr $MASTER_ADDR \

    --master_port $MASTER_PORT

"

GPT_ARGS="

    --tensor-model-parallel-size ${TP} \

    --pipeline-model-parallel-size ${PP} \

    --num-layers-per-virtual-pipeline-stage 1 \

    --num-layers 8 \

    --hidden-size 4096 \

    --ffn-hidden-size 14336 \

    --num-attention-heads 64 \

    --seq-length 4096 \

    --max-position-embeddings 4096 \

    --micro-batch-size 1 \

    --global-batch-size 16 \

    --make-vocab-size-divisible-by 1 \

    --lr 1.0e-6 \

    --train-iters 1000 \

    --init-method-std 0.01 \
```



```
    --no-masked-softmax-fusion \
    --attention-softmax-in-fp32 \
    --min-lr 1.0e-7 \
    --weight-decay 0.1 \
    --clip-grad 1.0 \
    --initial-loss-scale 4096.0 \
    --disable-bias-linear \
    --lr-warmup-fraction 0.01 \
    --fp16
"
DATA_ARGS="
    --split 990,5,5
    --data-path $DATA_PATH \
    --vocab-file $VOCAB_FILE \
    --merge-file $MERGE_FILE \
"
OUTPUT_ARGS="
    --log-throughput \
    --log-interval 1 \
    --save-interval 10000 \
    --eval-interval 10000 \
    --eval-iters 10 \
"
torchrun $DISTRIBUTED_ARGS pretrain_gpt.py \
    $GPT_ARGS \
    $DATA_ARGS \
    $OUTPUT_ARGS \
```



```
    --distributed-backend  nccl  \

set +x
```

6. 配置路径。

   请编辑示例脚本 train_distributed.sh，并设置如下环境变量以指定必要的路径：

```
CKPT_DIR=./ckpt

VOCAB_FILE=./gpt-tokenizer/vocab.json

MERGE_FILE=./gpt-tokenizer/merges.txt

DATA_PATH=./gpt_pretrain_data/alpaca_text_document
```

   注意，上述路径需根据您的实际情况进行适当调整。

7. 执行如下命令启动预训练。

```
bash ./train_distributed.sh
```

3.5 阶段五：模型训练调优

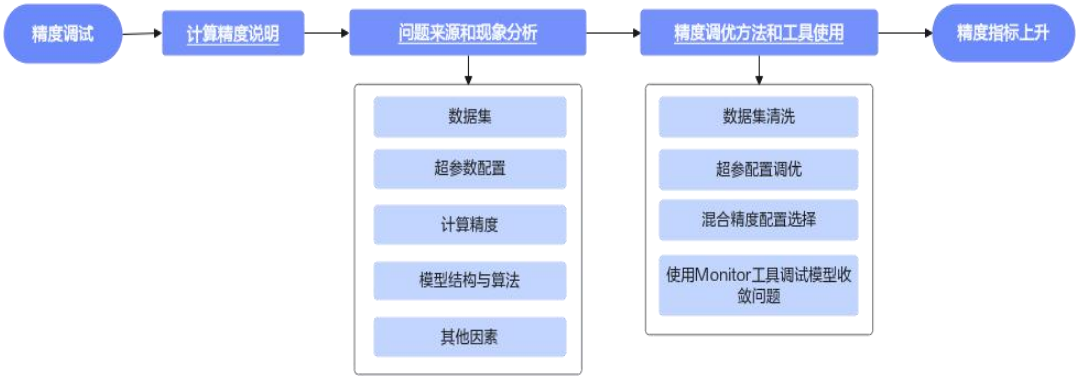
3.5.1 Pytorch 调优

1、 精度调优

训练一个大模型，往往涉及到高额的成本投入与复杂的技术难题。例如，MetaAI 公开了 OPT-175B 大模型的训练日志，披露了从 2021-10-20 到 2022-01-06 之间 75 天内训练大模型遇到的各种问题以及应对方案。这些问题非常有代表性：例如大量硬件失效，需要移除出错节点、排查软件栈错误后重启训练；再比如出现 Loss 上扬、梯度范数异常等棘手的问题，需要调整学习率、batch 尺寸，或者跳过异常训练语料来解决或规避。从上述示例中可以看出，即使在最先进、最主流的 AI 加速器上，训练大模型仍需克服诸多难关。除了 OPT-175B，AI2 发布了 OLMo 系列模型的技术报告、数据、权重、超参数，甚至公开了处理数据、评估模型的代码以及训练日志。OLMo 即便只是 7B 参数的语言模型，涉及到的细节仍然非常繁杂，例如混合精度配置、数据处理、大量训练内部状态监控等。

训练本身是一种研发行为，受数据集、模型结构、并行策略、超参等影响，模型训练过程中可能出现 NaN、溢出、Loss 发散等情况，这时需要对模型进行精度调试。在昇腾处理器上训练模型时，一般而言 Loss 总体呈现下降收敛趋势，即使出现偶尔的尖刺现象也可以通过跳过数据集、断点续训等方式规避。最终，当使用训练后得到的权重，采用常规数据集评估模型分数是否符合社区实践评分预期，即可视为精度调试已完成。

大模型训练过程中常见的问题及其调试方法，可帮助用户将问题消除在训练开始之前，以及缩短模型精度问题定位的时间。精度调试流程如下图所示。



1) 数据集清洗

在构建高质量机器学习模型的过程中，数据集清洗是一项至关重要的前期任务，旨在提升模型训练效果，防止过拟合及生成内容的冗余。

- **非目标语言与低质量样本剔除**：剔除非目标任务语言、丢弃低 perplexity 数据、删去标点/符号过多或过长过短的句子、删除具有某些特定词汇（如 html 标签、链接、脏话、敏感词）的句子。
- **去重**：删除包含大量重复词汇或短语的句子；删除重复率（词/n-grams 共现）过高的段落；删除训练集中可能与测试集相关度过高的内容。这样可以提高训练集质量，缓解语言模型生成内容重复的问题，避免测试集泄露带来的过拟合问题。

当遇到 Loss 或者 perplexity 尖刺，如果确定是数据批次引起的问题，可以跳过尖刺期间看到的一些数据批次，从以前正常的检查点重新开始训练。

2) 超参配置调优

- **batch size 配置**：在模型训练过程中，batch size 的设定通常较大，旨在充分利用大规模训练数据，增强模型训练过程的稳定性。例如，设置 batch size 为 8196，在样本序列长度为 2k 的情况下，可使每个批次处理大约 16M 个 token 输入。大样本的梯度平均值使得参数更新更精确，从而缓解不精确参数更新导致的 loss 跳变。GPT-3 使用动态调整 batch size 的方式，使其每步处理的 token 数从 32K 逐渐增大到 3.2M，这是提高训练稳定性的有效手段。
- **学习率**：学习率一般较小，且包含 warm up 设置，以确保训练平稳。比如在前 0.1%~0.5%的训练步骤中，设置一个线性的学习率递增。峰值学习率一般在 10^{-4} 以下，比如 GPT-3 的学习率是 6×10^{-5}。之后，会采用 cosine 衰减或者线性衰减学习率调度策略，逐渐减小学习率，在收敛前将学习率下降到峰值学习率的 10%左右。
- **优化器**：优化器一般采用 Adam、AdamW 以及 Adafactor。其中，Adafactor 是 Adam 的一个节约显存的变体。也有使用 SGD 作为优化器的例子，但并不常见。
- **权重初始化**：正确初始化权重可以帮助模型更快地收敛并提高性能。例如，通常使用小的高斯噪声或者使用 T-fixup 初始化。
- **其他稳定训练技术**：
  - 梯度裁剪（Gradient Clipping）：作为一种常用的稳定训练手段，通常设定裁剪阈值为 1.0，防止梯度过大引发训练不稳定。
  - Weight Decay（L2 正则化）：设置合理的权重衰减率，如 0.1，有助于防止过拟合，增强模型泛化能力。

- 特殊层的调整：GLM 研究发现，embedding 层往往存在较大的梯度异常情况，故需根据实际情况适度调整相关参数。

3) 混合精度配置选择

大语言模型的成功证明了增大神经网络的参数规模能够提升模型性能，但同时也增大了对加速器内存、算力及通信传输带宽的要求。为了减少内存占用加快收敛速度，大模型训练往往采用 16 位半精度浮点格式，例如 float16 或者 bfloat16。

大量实验已经证明可以使用半精度浮点数训练大模型，也不会对模型性能带来显著影响，相反低精度计算作为正则化的一部分，反而能够给模型泛化能力带来好处。但目前低精度训练对模型的统计学影响也并不那么清晰，所以整个训练过程单纯使用低精度浮点运算非常具有挑战性。

在此情况下，依赖实践经验的混合精度技术成了一个现实的选择。混合精度训练在训练任务中组合地使用不同的数字格式，降低了对内存、算力和通信带宽的需求，极大提高了训练速度。

1) 大规模并行训练的混合精度选择

大规模分布式并行训练中使用半精度浮点数，一般有两种选择：float16 或者 bfloat16。这两种格式的区别见半精度浮点数。float16 相对于 bfloat16 有更高的精度，但是表示范围更小。混合精度训练首先需要把模型中适合的参数转移到半精度浮点类型。

如果混合精度训练中选择 float16，为了避免表示范围小引起的浮点上溢和下溢，混合精度要结合动态 Loss 缩放机制，以下是采用 float16 的混合精度训练典型流程概述：

1) 保留一份 FP32 格式的权重主备份，同时优化器状态也应以 FP32 格式存储。
2) 将 Loss 缩放因子 S 初始化为一个较大的值。
3) 对每一个训练 step：
   a. 将权重复制一份到 FP16 格式。
   b. 使 FP16 格式的权重和激活值进行前向传播。
   c. 将最终的 Loss 乘以缩放因子 S。
   d. 使用 FP16 格式进行后向传播，包括权重、激活值及其对应的梯度。
   e. 若检测到权重梯度中出现 Inf 或 NaN：
      - 减小 S 值。
      - 跳过当前权重更新步骤，重新开始下一个训练步骤。
   f. 将权重梯度乘以 1/S。
   g. 梯度累积或者梯度累积足够步后使用 FP32 更新主权重。
   h. 如果之前 N 步都没有看到 Inf 或者 NaN，增加 S 值。
   i. 在上述过程中，有几处计算必须要以 FP32 完成。比如主权重的更新，因为累加能够导致精度误差积累，所以必须要以 FP32 计算。缩放因子必须是 FP32 类型，甚至 1/S 的计算要将 S 转成双精度数求倒数再转回到 FP32。

使用 BF16 格式的半精度数时，因为 BF16 有更大的表示范围，所以一般无需使用 Loss 缩放机制。但是 BF16 数值精度比 FP16 更差，所以在步骤三的第七点做梯度累积的时候需要使用 FP32，否则有可能会因为梯度累积误差导致模型不收敛。另外 BF16 比 FP16 多 15%的运行时内存，主要原因在于梯度累积时需要转 FP32。

PyTorch 提供了自动混合精度（AMP）的机制，AMP 按需自动调整张量的数据类型（dtype）。例如在 AMP autocast 上下文时，矩阵乘法 matmul 的输入张量会被自动转

化为半精度浮点类型。AMP 也提供了 GradScaler，通过自动调整 Loss 的缩放来防止梯度的下溢和上溢。PyTorch 的 AMP 优化级别使用 apex.amp 的 O1 级，这意味着 PyTorch AMP 使用黑白名单自动决定使用 FP16、BF16 还是 FP32 进行计算，但还有一些特定模型相关的精度敏感的运算并不在 AMP 的自动 upcast 名单中，需要开发者手动干预。

2) 特定计算操作对计算精度的要求

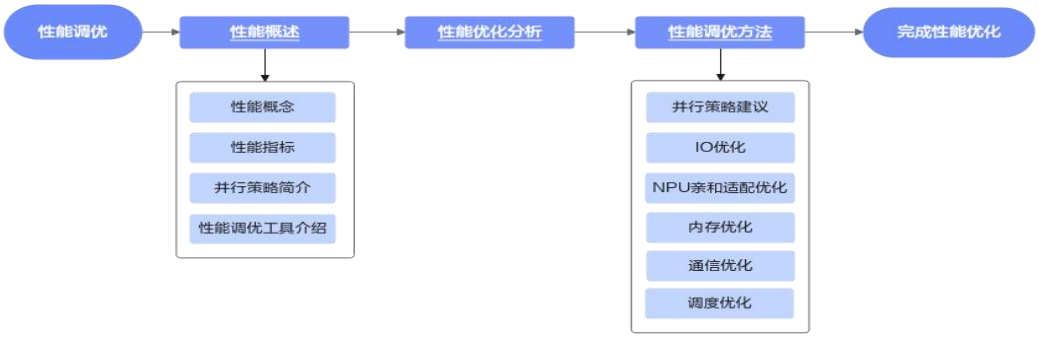
在大规模深度神经网络的设计与训练过程中，针对不同计算操作选择适宜的精度类型至关重要，这一决策既要基于深厚的数学原理分析，又需丰富的实践经验支持。本节内容总结自大量论文中的实践经验，介绍影响模型性能的关键计算操作对精度的具体要求。

- **Softmax 函数**：softmax 函数中有指数运算，容易发生上溢出和下溢出，并且分母有累加操作，目前主流的算子库只使用 FP32 来实现 softmax。
- **Cross Entropy 损失计算**：cross_entropy 用来计算交叉熵损失，和很多损失函数一样，需要使用 FP32 计算以保持数值稳定性，所以算子实现只接受 FP32 输入。
- **矩阵乘法（Matmul）**：matmul 矩阵乘法一般来说可以使用半精度数，但在大语言模型的 attention 层，attention 分数与 V 张量的矩阵乘法需要用 FP32，以便维持数值稳定性，避免上溢，减小累积错误。
- **归一化函数**：归一化函数如 layer_norm、Batch_norm、RMSNorm 需要用 FP32 计算。layer_norm 要计算方差，有累加操作，如果用 FP16，则计算过程中容易发生上溢和比较大的误差，从而影响最后的收敛。Batch_norm 与 layer_norm 类似，也需要将输入张量 upcast 到 FP32 计算。RMSNorm 是 layer_norm 的扩展，在开源 LLaMA 模型中使用，PyTorch 目前并没有提供 RMSNorm 的实现。如果开发者自己实现，需要注意将输入张量强制 upcast 到 FP32 计算，结果可以转回半精度数，主要原因也是它需要计算 L2 均值，涉及累加操作，低精度运算容易累积误差。
- **卷积操作与池化**：卷积操作属于逐单元矩阵乘法，对精度不敏感，可以使用 BF16，不会对模型收敛性产生影响。和卷积类似的还有池化，同样可以使用 BF16。ReLU 和 GeLU 属于逐点运算，同样对精度不敏感。
- **循环神经网络（RNNs）**：RNN 一般对精度比较敏感，LSTM-cell 对精度不敏感，可以用 BF16。
- **模型起始与结束层**：通常大语言模型的第一层和最后一层对精度敏感，比如 GPT 的 embedding 层（词嵌入和位置嵌入层）需要使用 FP32。一般的输出层即便不是 softmax，计算和输出结果也需要是 FP32。
- **通信算子**：涉及到梯度或者激活值累加的通信算子如 all_reduce、reduce_scatter，都需要把输入张量 upcast 到 FP32 以保证数值稳定性。

2、 性能调优

在计算越来越重要的今天，以 GPU（Graphics Processing Unit）和 NPU（Neural Network Processing Unit）为代表的并行计算设备，在人工智能和其他行业，都扮演着重要角色。计算的效率，或者称之为计算的性能，越来越得到广泛关注。

本章节以性能的含义以及性能工具等基础概念介绍为出发点，介绍了训练模型在昇腾设备上的通用性能调优方法。对应的性能调优流程如下图所示。



1) 并行策略通用建议

并行策略的调试与设计需要对具体模型进行详细分析，没有一个通用的万能法则可以适用于所有情况。然而，借鉴以往的调优经验，我们可以总结出一些相对通用的建议。

1. 在面临显存不足、模型过大无法完全加载以及需要进行切分的情况下，优先考虑使用 TP（Tensor Parallelism）进行切分，并确保切分的数量小于等于机器内的计算卡数。例如，在一台服务器上有 8 张计算卡，那么 TP 的最大设置不应超过 8。这样可以充分利用计算资源，减少显存占用。

2. 如果在 TP 切分达到最大显存容量仍然不足的情况下，可以考虑在机器之间使用 PP（Pipeline Parallelism）进行切分。理论上，PP 的数量应该越小越好，以尽可能减少空闲计算资源的浪费。

3. 在机器资源富裕的情况下，可以开启 DP（Data Parallelism）并行，将计算任务分配给多个机器进行并行处理，从而提高处理效率。然而，在机器资源有限的情况下，如果开启 TP+PP 切分后显存仍然不足，可以考虑使用 ZeRO 1 和重计算技术。ZeRO 1 可以将模型优化器状态划分到更多的设备上，并通过集合通信进行同步。同时，重计算技术可以通过选择性重计算来提高显存的使用率，从而提高模型训练效率。

4. 此外，即使在模型能够成功运行的情况下，也可以尝试主动地使用降低内存占用的手段，例如 ZeRO 1 和重计算等，然后增大 batch size。这样有时也会取得令人意外的效果。

综上所述，通过技术能力和合理选择并行策略，可以在资源有限的情况下优化模型训练效率，并充分利用计算资源。然而，对于具体的模型和环境，仍需要进行详细分析和实验，以找到最佳的并行策略和优化方法。

2) IO 优化

在 PyTorch 模型中，数据加载部分的逻辑一般是 DataLoader 及其衍生类，在 DataLoader 加载数据中，要注意以下两个核心点：

- 第一个是数据加载的预处理部分，数据的预处理通常会写在 datasets 里，数据的预处理包括对文本、图片、视频和语音等不同格式数据的处理，数据预处理耗时长是比较容易识别出来的，一般通过打点计时。
- 第二个就是要确定好数据读取的方式，一般而言，每张卡都去读取数据是比较理想的，如果存在 0 卡读取，广播给其他卡的数据读取形式，要关注其性能如何，很多时候都会让模型性能严重劣化。

1、 硬件类

- 检查数据存放的硬盘，最好在 NVME（Non-Volatile Memory Express，一种更快、更可靠的存储设备，适合存储需要高速读写的数据）盘上，当硬盘为共享存储时，需要注意 IO 瓶颈。
- 检查 NUMA（Non-Uniform Memory Access，非一致性内存访问）数量，可以通过 lscpu 命令查看，一般需要 NUMA 数量为 4 或者 8。

2、 参数类

DataLoader 入参：

- num_workers 是一个比较常用的参数，一般而言，对于图像或视频等复杂数据，可以将 num_workers 设置的大一些（如 4 或 8），通过多进程并行提高数据读入和解析的速度。在使用这个参数的时候，需要结合实际情况，如视频解析本身并不存在瓶颈，过多的 num_workers 则会增加进程开销，因此需要酌情考虑。
- 在内存允许的情况下，保持 pin_memory=True，在 PyTorch 中，该参数可以写为 pin_memory_device。
- 一般而言，可以加上 persistent_workers=True，该设置可以减小进程销毁或申请的开销，不过这个方法也可能带来内存瓶颈。

3、 代码技巧类

- 在数据预处理时间较长而影响模型训练时，可以尝试预取数据，即 Data prefetcher。
- Infinite DataLoader 同样可以缓解 epoch 间加载缓慢的问题。
- 提前缓存数据或制作二进制数据，轻量化数据加载，若原始数据是压缩或编码格式，则会引发解压缩或解码，加重 CPU 负载，容易引起 host bound 问题，建议提前处理成二进制格式，快速进行数据读取。
- 对于 NPU 预处理数据比较慢，例如 dlrm 模型，可以考虑将预处理放到 datasets 里用 CPU 多 num_workers 处理数据。
- 对于句子之间长度差距比较大的句子，组成 batch 时可以考虑 resample 一下。对于 NLP 任务，存在输入的 sentence 长短不一，tokenize 之后需要 padding 成相同长度才能组成一个 batch 的情况。对于一些数据集中，所有 sentence 长度差别很大的情况，如果随机取句子，那么需要补的 padding 就很多，造成很多冗余的计算量。此时，可以通过调整 DataSampler，尽量将长度接近的 sentence 组成一个 batch，这样造成很多冗余的计算量就可以减少很多，训练整体的吞吐量会有显著提升。缺点：可能稍微会影响数据的随机性。
- collate_fn 用于将多个 workers 从 datasets 中读取的 batch_list 组合成一个 batch，这个组合过程是可以加速的，例如原来每个 workers 都要对各自的数据 transpose 并且减均值除方差再合并成一个 batch。此时可以先合并成一个 batch，再放到 Device 上，并且用 Device 算力完成 transpose 和除方差操作，达到减轻 CPU 负载的目的。以此原理，timm 仓库设计了 fast_collate 函数，已经是 timm 训练的标配，能够有效提升性能。

3) NPU 亲和适配优化

基于一键迁移的方式可以保证 GPU 的代码迁移到 NPU 上的代码修改轻量化，但站在性能优化的角度上，社区中的模型代码普遍基于 GPU 实现，NPU 和 GPU 在计算

原理和底层架构上存在非常大的区别，因此，有些时候我们需要进行模型层面的修改，亲和适配 NPU，达到最优的性能。性能优化遵循以下大的逻辑：

- 融合算子

  融合算子的优化原理为，通过数学意义上的等价替换，将多个算子融为一个算子的计算，减少冗余计算，同时减少下发次数，从而提高性能。
- 消除多余的 stream 同步

  在很多开源代码中，由于作者在编写时很少考虑性能，因此在代码中可能存在增加很多的 stream 同步操作，这些同步通常是由 h2d（host to device，从 CPU 侧下发到 NPU 侧）、d2h（device to host，从 NPU 侧搬回到 CPU 侧）操作（如 tensor.item、reduce_all、torch.isfinite 等）引入的，原则上，我们需要尽可能减少这些异步流同步的操作，让模型通过异步流实现最佳的并行效率。
- 有时候因为下发或通信等问题，可能导致多卡训练时，卡与卡之间性能不一致，此时由于反向计算后需要对多卡之间的梯度进行同步，产生快卡等慢卡的情况。性能数据分析时，可以明显观察到部分卡上集合通信算子耗时占比高，这种现象的根因并不是集合通信算子耗时久，而是因为性能快的卡在等待性能慢卡，这种问题，要明确找到快卡和慢卡差异产生的第一位置，这个可以通过性能工具来寻找。
- 部分 CPU 上运行的优化方法
  在 mask-rcnn 中，模型的 mask decode 运行 CPU 上进行，且仅用单核实现，性能不够理想。此时，可以将计算部分放到 NPU 上，或者在 CPU 上，通过多进程的方式进行加速。

4) 内存优化

➢ 调整内存参数

- 设置内存因子，限制进程申请的内存上限，取值 0~1。

  例如：设置 0.95，可使用的内存上限为 60G * 0.95 = 57G。

  内存申请超过上限触发内存释放。设置 PyTorch 申请的内存上限，可以避免内存使用极限场景下，PyTorch 耗尽 Device 内存，其他组件申请内存失败导致的进程异常。

```
torch_npu.npu.set_per_process_memory_fraction(0.95)
```

- 垃圾回收阈值，默认不开启，与内存因子配合使用，取值 0~1，为内存上限的百分比。

  当内存申请到 gc 阈值，则触发内存池空闲内存块回收。建议由大到小配置调试。

```
export PYTORCH_NPU_ALLOC_CONF="garbage_collection_threshold:0.95"
```

- 内存块允许切分上限，单位 MB。

  例如：设置 50，大于等于 50M 的内存块不允许切分使用。设置该参数可避免大内存块被切分导致较多的内存碎片影响内存复用。调试时可以先采集内存 profiling，按照算子内存申请降序排列，参数值由大到小尝试。

```
export PYTORCH_NPU_ALLOC_CONF="max_split_size_mb:50"
```


- 使能内存池扩展段功能，由 PyTorch 自己管理虚拟地址与物理地址映射，降低内存碎片。
  对于动态 shape 场景，shape 随 step 增加而增大，从而导致内存块不能复用内存碎片上升，对该场景有较好优化。

```
export PYTORCH_NPU_ALLOC_CONF="expandable_segments:True"
```

说明

若同时设置多个参数可以通过逗号分隔，如下所示：

```
export
PYTORCH_NPU_ALLOC_CONF="garbage_collection_threshold:0.95,
max_split_size_mb:50"
```

注意，expandable_segments:True 不能与上述两个环境变量共用。

➢ 多流复用

流（stream）是 PyTorch 的一个重要机制，每个流代表一个时序上严格的执行逻辑。一般地，PyTorch 在执行时会启动多个流，来并行完成模型的通信和计算任务。每个流在执行过程中会根据自身需要向设备（Device）申请内存，称为该流的内存池。如果一个流申请的内存池需要给另一个流使用，两个流之间需要进行通信，以当前流的内存块（block）对应的数据执行完作为对应流使用这块内存的标志。host 算子下发过快时，计算流的算子来不及复用通信流的算子，特别是通信流上的算子有依赖时，需要全部结束再释放复用，多流复用的逻辑就是让通信流上的内存提前释放，让计算流复用。

多流复用是 PyTorch 侧非常有效的内存优化方法，需要配置环境变量，代码如下：

```
export MULTI_STREAM_MEMORY_REUSE=1
```

➢ 减小 HCCL 通信缓存

HCCL 通信缓存是一种高性能、高可靠的通信缓存技术，用于提高分布式计算系统的通信效率和可靠性。它通过在计算节点上预分配一定数量的内存空间，用于存储通信数据，避免了频繁的内存分配和释放操作，从而提高了通信效率。

可以通过调整以下数值，来控制 HCCL 的缓存大小。默认值为 200MB。

```
export HCCL_BUFFSIZE=200
```

➢ Python GC 优化

GC（Garbage collector）是 Python 提供的可选的垃圾回收器的接口，提供的功能包括关闭收集器、调整收集频率以及设置调试选项。它同时提供对回收器找到但是无法释放的不可达对象的访问。

我们可以通过调用 gc.disable()来关闭 python 的自动垃圾回收机制。

在大模型训练中，我们可能因为大量进程的开销而频繁触发 GC 进行垃圾回收，而垃圾回收在一定程度上会占用 CPU 的性能，如果频繁触发，可能会造成模型训练过程中性能的抖动。

在遇到此类问题时，首先可以主动通过 gc.disable()关闭自动垃圾回收机制；同时，通过如下接口设置回收频率和回收次数，保证内存稳定的同时降低性能抖动。

gc.set_threshold(*threshold0*, *threshold1*, *threshold2*)

其中 threshold0、threshold1 及 threshold2 分别代表 python 的三代对象。

所有新建的对象都是 0 代对象，当某一代对象经历过垃圾回收，依然存活，那么它就被归入下一代对象。垃圾回收启动时，一定会扫描所有 0 代对象，如果 0 代经过一定次数的垃圾回收，那么就会启动对 0 代和 1 代的扫描清理；当 1 代也经历了一定次数的垃圾回收，那么会启动 0、1 和 2，即所有对象的扫描。命令示例如下：

gc.set_threshold(700, 10, 5)

其中 700、10、5 分别代表 0、1、2 三代最大的引用数量，超过引用上限则会触发 GC 进行垃圾回收。

3.5.2 TensorFlow 调优

1、 精度调优

用户迁移后的模型在昇腾 AI 处理器（简称 NPU）上训练，功能已调通，但可能会遇到精度不达标或者收敛效果差的问题，用户模型在昇腾 AI 处理器上执行时，包括但不限于：

- loss 曲线与参考基准差异不符合预期
- 验证准确度与参考基准差异不符合预期

这些精度问题由于具有以下特征而非常难以定位：

- 训练正常结束
- 日志无任何异常
- 仅在与参考基准对比时才发现结果不符合预期

为指引开发者进行精度调优，本节提供了精度调优流程指引。

精度调优思路

精度问题来源可能来自于各个方面：

- 提供的参考基准存在问题
- 进行模型迁移时存在问题
- 网络中算子精度问题等

根据问题来源的不同以及高概率的问题发生点，您可以按照以下流程进行精度问题的定位。

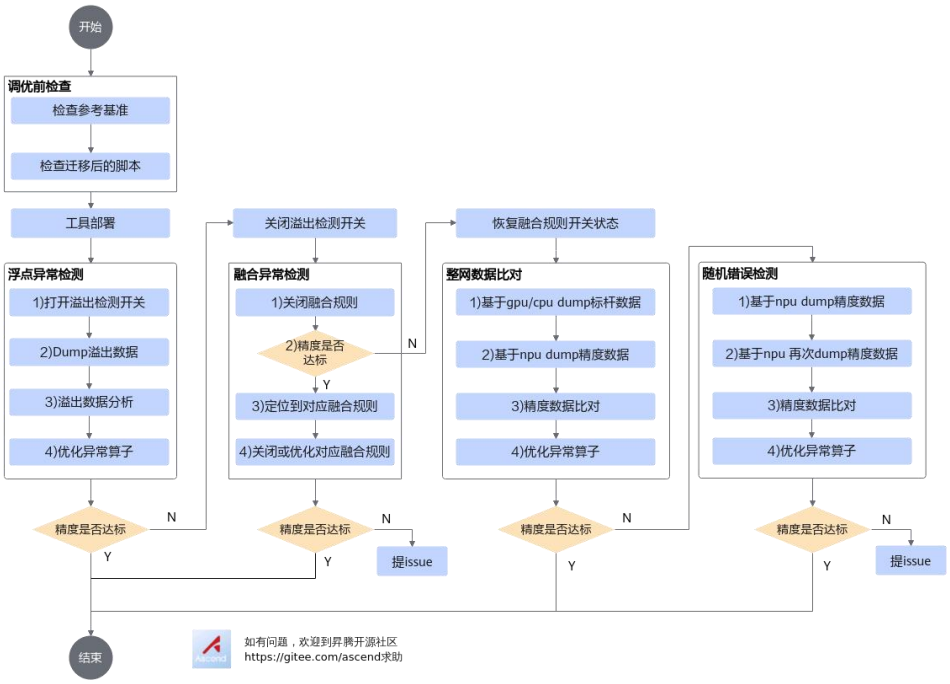
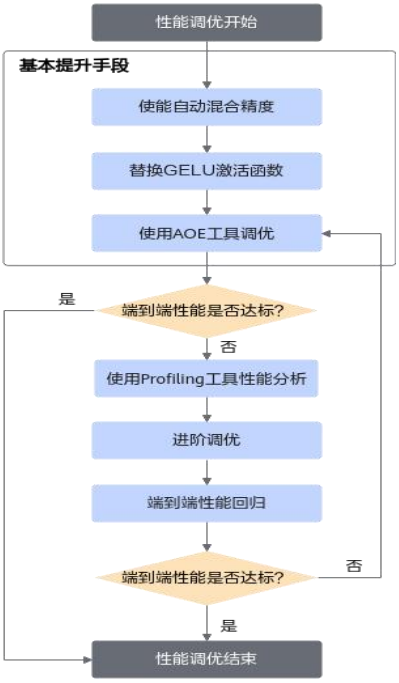


表 1 精度调优流程说明

2、 性能调优

迁移到昇腾 AI 处理器进行训练的网络如果存在性能不达标的问题，可按照如下流程进行性能调优：

图 1 TensorFlow 网络性能调优流程



1) 当性能不达标时，优先推荐进行如下通用的性能提升操作。
    a. 使能自动混合精度训练模式。
    b. 替换 GELU 激活函数。
    c. 使用 AOE 工具进行子图、算子以及梯度切分策略的调优。
2) 再次执行模型训练，并评估训练性能是否达标。
    a. 若性能达标 —> 调优结束。
    b. 若性能不达标 —> 执行 3。
3) 使用 Profiling 工具采集性能数据并分析。

    参考 Profiling 数据采集与分析进行性能数据的采集、解析导出与分析。
4) 根据识别的性能瓶颈参见进阶调优进行进一步的性能提升。
5) 再次执行模型训练，进行回归测试，评估训练性能是否达标。
    a. 若性能达标 —> 调优结束。
    b. 若性能不达标 —> 再次执行 AOE 自动调优。

3.5.3 MindSpeed 调优

MindSpeed 是专为华为昇腾设备设计的大模型加速解决方案。在当前 AI 领域，大模型训练因其复杂性高、技术挑战多而备受关注。特别是在内存资源受限的情境下，如何高效地进行大模型训练成为业界关注的焦点。针对这一需求，MindSpeed 应运而生，以其卓越的性能表现和深度优化的算法体系，助力用户在昇腾设备上高效实现大模型训练。

大模型训练的复杂度与资源密集型特征，对计算平台提出了极高的要求，尤其是内存资源的限制往往成为制约训练效率的瓶颈。为突破这一限制，业界先后发布了

Megatron-LM、DeepSpeed 等大模型训练加速库，通过数据并行、模型并行、优化器并行等策略，实现了多计算卡协同训练，从而对更大规模的模型进行高效训练。

Megatron-LM、DeepSpeed 与 MindSpeed 的主要区别：

计算平台适配性

Megatron-LM 主要针对 GPU 进行了深度优化，利用了 CUDA 和 cuDNN 等技术来加速训练过程。DeepSpeed 生态更为强大，原生支持多种计算设备，包括昇腾计算设备。

MindSpeed 则是专为华为昇腾系列硬件设计的大模型训练加速库，兼容原生 Megatron-LM 框架，它不仅考虑到了硬件层面的亲和性，还结合了昇腾平台特有的软硬件特性进行了优化。

功能特性对比

并行算法优化：三者都支持多种并行策略。此外，MindSpeed 还提供了自动搜索最佳并行配置的能力。

内存资源优化：DeepSpeed 的 ZeRO 技术以其高效的显存管理著称；而 Megatron-LM 则利用混合精度和激活重计算等技术优化内存。相比之下，MindSpeed 不仅提供了类似的内存压缩、复用功能，还引入了差异化的重计算技术，最大限度地减少对额外内存的需求。

通信性能优化：MindSpeed 为昇腾训练设备提供通算融合等策略，减少了通信延迟，提高了整体训练效率。

MindSpeed 在此基础上进行一系列昇腾亲和优化，其核心优势体现在以下几个方面：

表 1 MindSpeed 核心优势说明

优势	说明
并行算法优化	支持模型并行、专家并行、长序列并行等多维并行策略，针对昇腾软硬件架构进行亲和优化，显著提升了集群训练的性能和效率。
内存资源优化	提供内存压缩、复用，以及差异化的重计算技术，最大限度地利用内存资源，有效缓解内存瓶颈，提升训练效率。
通信性能优化	采用通算融合、通算掩盖等策略，配合高效的算网协同机制，大幅提高算力利用率，减少通信延迟，优化整体训练性能。
计算性能优化	集成高性能融合算子库，结合昇腾亲和的计算优化，充分释放昇腾算力，显著提升计算效率。

优势	说明
差异化能力支持	在长序列、权重保存、并行策略自动搜索等场景提供差异化能力。

MindSpeed 作为昇腾设备的专属加速解决方案，凭借其卓越的性能表现与深度优化的算法架构，为客户在 AI 领域实现大模型训练提供了强有力的支持。借助 MindSpeed，用户能够充分挖掘并利用昇腾设备的高性能计算能力，加速大模型训练过程，从而在 AI 领域更快地实现价值。

3.6 阶段六：推理模型迁移

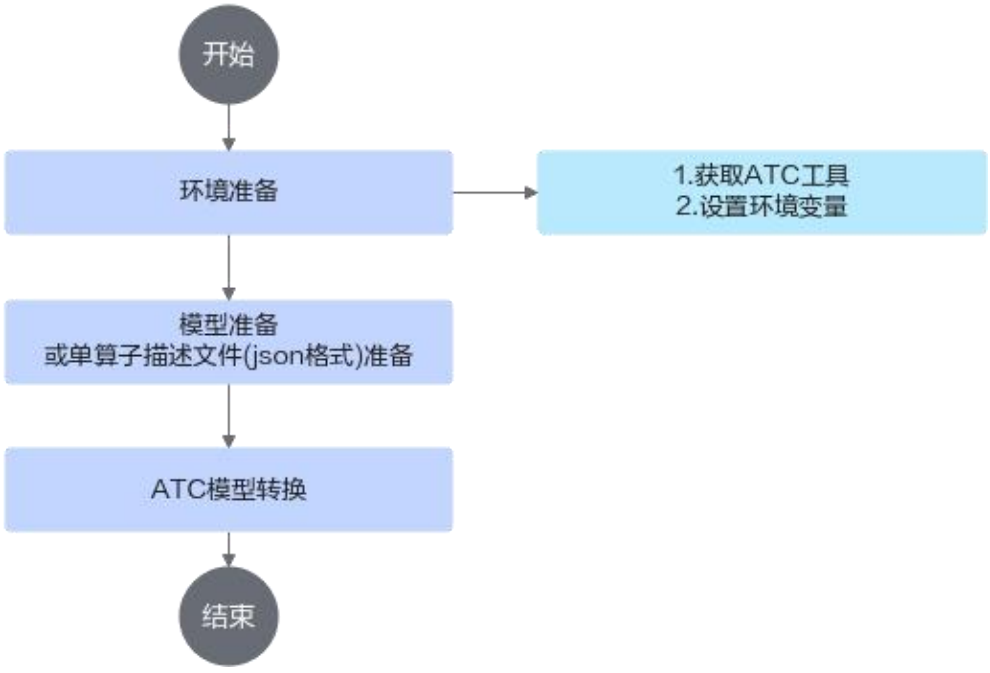
3.6.1 小模型推理迁移

1. ATC 模型转换/MindSpore Lite

ATC 工具运行前需要准备环境和模型，本节给出 ATC 工具的运行流程以及和各组件的交互流程。

1) 运行流程

运行流程如图 1 所示。



- 使用 ATC 工具之前，请先在开发环境安装 CANN 软件包，获取相关路径下的 ATC 工具，然后设置环境变量，详细说明请参见环境搭建。
- 准备要进行转换的模型或单算子描述文件，并上传到开发环境。单算子描述文件相关配置请参见单算子模型转换。
- 使用 ATC 工具进行模型转换，模型转换过程中使用的参数请参见参数说明。

2) 模型转换交互流程

下面以开源框架网络模型转换为 om 离线模型为例，详细介绍模型转换过程中与周边模块的交互流程。

根据网络模型中算子计算单元的不同，分为 AI Core 算子和 AI CPU 算子：AI Core 算子是指在昇腾 AI 处理器的核心计算单元上执行的算子，而 AI CPU 算子则是在 AI CPU 计算单元上执行的算子。

在 AI Core 算子、AI CPU 算子的模型转换交互流程中，虽然都涉及图准备、图拆分、图优化、图编译等节点，但由于两者的计算单元不同，因此涉及交互的内部模块也有所不同，请参见下图。

如果用户使用的网络模型中有自定义算子，也请优先参见上述手册开发部署好自定义算子，模型转换时会优先去查找自定义算子库匹配模型文件中的算子；若匹配失败，则会去查找内置算子库。

模型转换过程中，若遇到 AI CPU 算子不支持某种数据类型导致编译失败的场景，可通过启用 Cast 算子自动插入特性快速将输入转换为算子支持的数据类型，从而实现网络的快速打通。

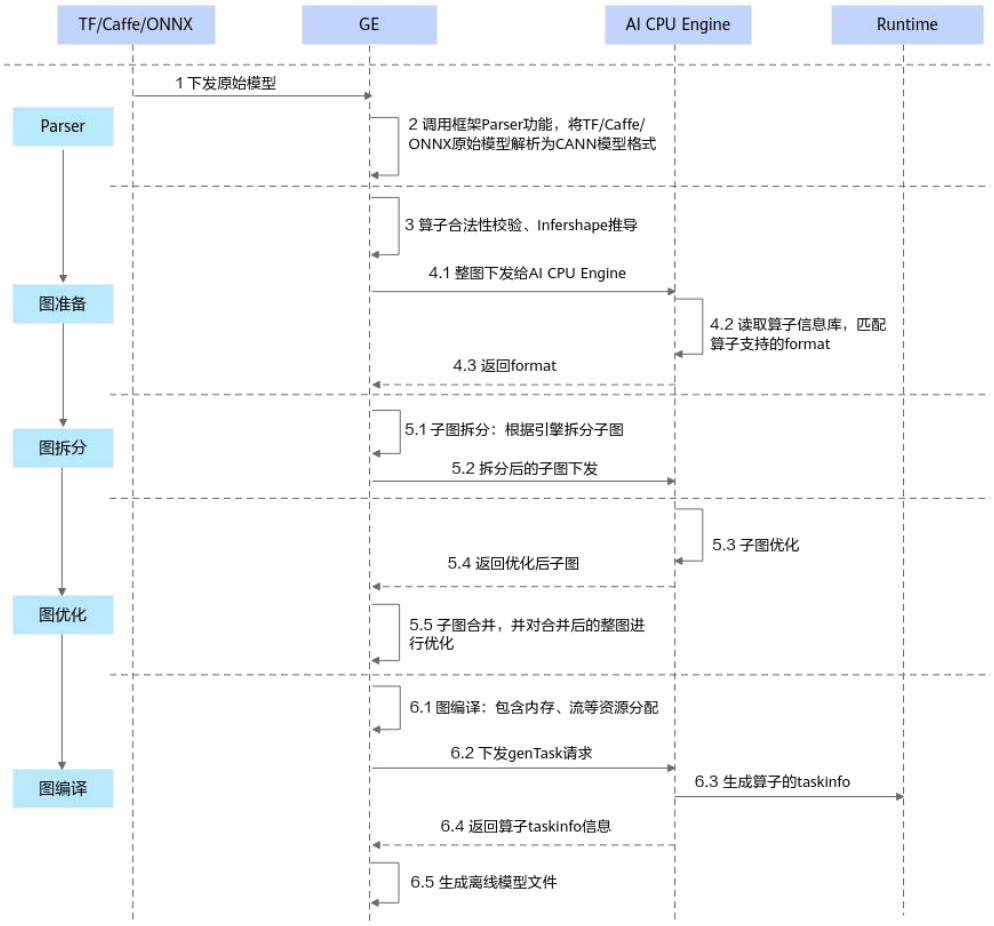
- AI Core 算子模型转换交互流程

1. 调用框架 Parser 功能，将主流框架的模型格式转换成 CANN 模型格式。
2. 图准备阶段：该阶段会完成原图优化以及 Infershape 推导（设置算子输出的 shape 和 dtype）等功能。
3. 图拆分阶段：GE（Graph Engine，图引擎）根据引擎拆分多个子图。
4. 图优化阶段：GE 将拆分后的子图进行优化，优化时按照当前子图流程对 AI Core 算子进行预编译和 UB（Unified Buffer）融合，然后根据算子信息库中算子信息找到算子实现将其编译成算子 kernel（算子的*.o 与*.json），最后将优化后子图返回给 GE。

优化后的子图合并为整图，再进行整图优化。

5. 图编译阶段：GE 进行图编译，包含内存分配、流资源分配等，图编译完成之后生成适配昇腾 AI 处理器的离线模型文件（*.om）。

- AI CPU 算子模型转换交互流程



1. 调用框架 Parser 功能，将主流框架的模型格式转换成 CANN 模型格式。
2. 图准备阶段：该阶段会完成算子基本参数校验以及 Infershape 推导（设置算子输出的 shape 和 dtype）等功能。

   另外，GE 将整图下发给 AI CPU Engine，AI CPU Engine 读取算子信息库，匹配算子支持的 format，并将 format 返回给 GE。

3. 图拆分阶段：GE 根据引擎拆分多个子图。
4. 图优化阶段：GE 将拆分后的子图下发给 AI CPU Engine，AI CPU Engine 进行子图优化，并将优化后子图返回给 GE。

   优化后的子图合并为整图，再进行整图优化。
5. 图编译阶段：GE 进行图编译，包含内存分配、流资源分配等，并向 AI CPU Engine 发送 genTask 请求，AI CPU Engine 返回算子的 taskinfo 信息给 GE，图编译完成之后生成适配昇腾 AI 处理器的离线模型文件（*.om）。

2. 模型转换

MindSpore Lite 提供离线转换模型功能的工具，支持多种类型的模型转换，转换后的模型可用于推理。命令行参数包含多种个性化选项，为用户提供方便的转换途径。

目前支持的输入模型类型有：MindSpore、TensorFlow Lite、Caffe、TensorFlow、ONNX 和 PyTorch。

通过转换工具转换成的 ms 模型，支持转换工具配套及更高版本的 Runtime 推理框架执行推理。

使用 MindSpore Lite 模型转换工具，需要进行如下环境准备工作。

- 编译或下载模型转换工具。
- 将转换工具需要的动态链接库加入环境变量 LD_LIBRARY_PATH。
- export LD_LIBRARY_PATH=${PACKAGE_ROOT_PATH}/tools/converter/lib:${LD_LIBRARY_PATH}
  ${PACKAGE_ROOT_PATH}是编译或下载得到的包解压后的路径。
- 编译 MindSpore Lite 包时若使用的是 Python3.11，则使用转换工具以及推理工具时需要将使用的 Python 动态链接库加入环境变量 LD_LIBRARY_PATH。
- export LD_LIBRARY_PATH=${PATHON_ROOT_PATH}/lib:${LD_LIBRARY_PATH}
  ${PATHON_ROOT_PATH}为使用的 Python 环境所在路径。待解耦 Python 依赖后该环境变量无需设置。

3. 模型推理

通过 MindSpore Lite 模型转换工具转换成.ms 模型后，即可在 Runtime 中执行模型的推理流程。本教程介绍如何使用 C++接口执行推理。

使用 MindSpore Lite 推理框架主要包括以下步骤：

1) 模型读取：从文件系统中读取由模型转换工具转换得到的.ms 模型。

2) 创建配置上下文：创建配置上下文 Context，保存需要的一些基本配置参数，用于指导模型编译和模型执行。

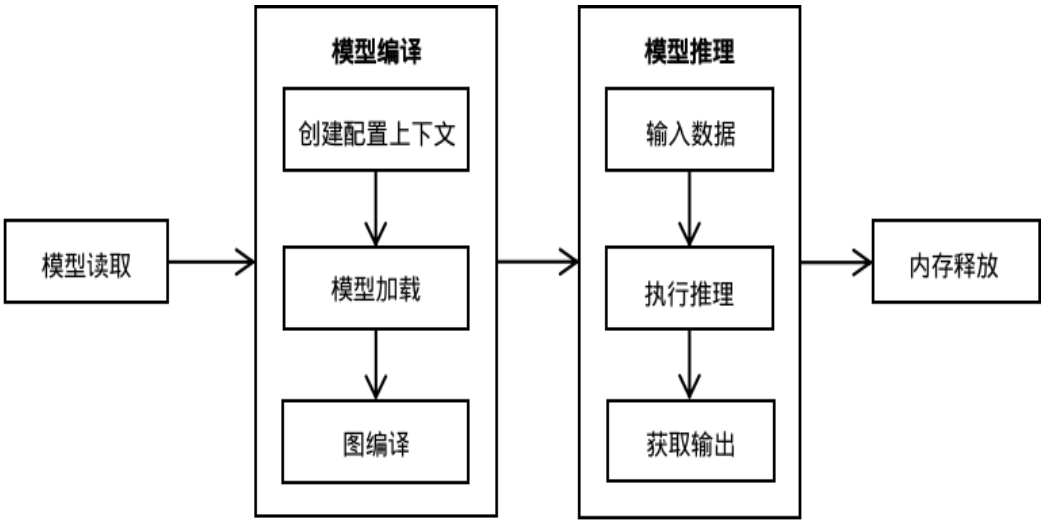
3) 模型创建、加载与编译：执行推理之前，需要调用 Model 的 Build 接口进行模型加载和模型编译。模型加载阶段将文件缓存解析成运行时的模型。模型编译阶段主要进行算子选型调度、子图切分等过程，该阶段会耗费较多时间所以建议 Model 创建一次，编译一次，多次推理。

4) 输入数据：模型执行之前需要向输入 Tensor 中填充数据。

5） 执行推理：使用 Model 的 Predict 进行模型推理。

6） 获得输出：模型执行结束之后，可以通过输出 Tensor 得到推理结果。

7） 释放内存：无需使用 MindSpore Lite 推理框架时，需要释放已创建的 Model。



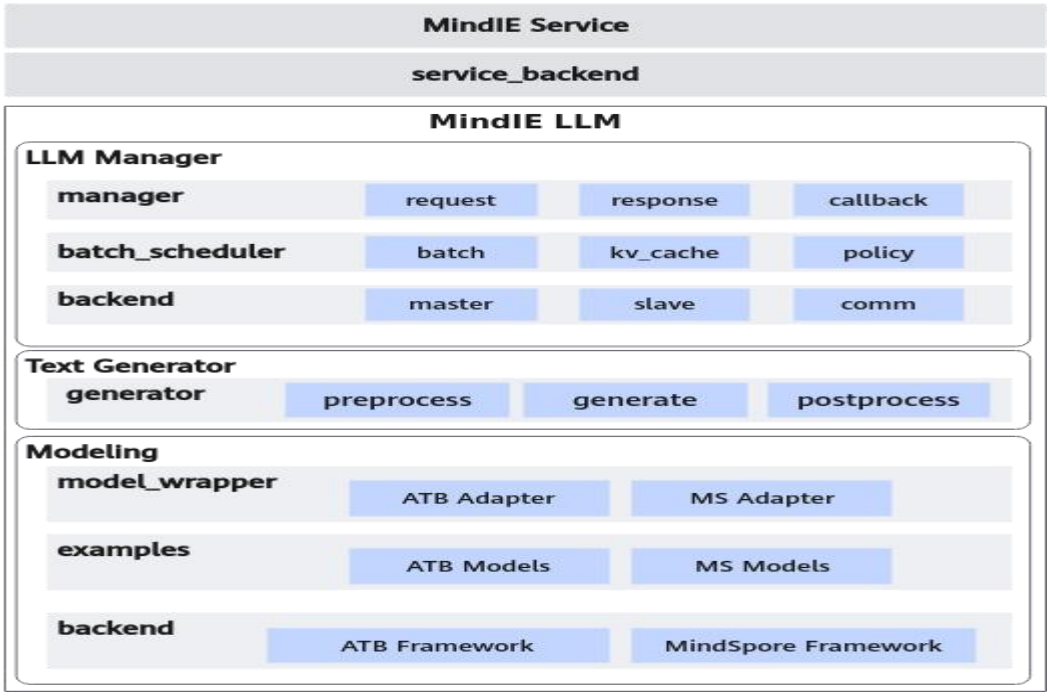
3.6.2 大模型：MindIE

MindIE LLM（Mind Inference Engine Large Language Model，大语言模型）是 MindIE 下的大语言模型推理组件，基于昇腾硬件提供业界通用大模型推理能力，同时提供多并发请求的调度功能，支持 Continuous Batching、Page Attention、FlashDecoding 等加速特性，使能用户高性能推理需求。

MindIE LLM 主要提供大模型推理 Python API 和大模型调度 C++ API。

本手册有助于用户快速了解 MindIE LLM，完成大模型推理的部署测试。

1、 MindIE LLM 架构



MindIE LLM 总体架构分为三层：LLM Manager、Text Generator 和 Modeling。

- LLM Manager：负责状态管理及任务调度，基于调度策略实现用户请求组 batch，统一内存池管理 kv 缓存，返回推理结果，提供状态监控接口。
- Text Generator：负责模型配置、初始化、加载、自回归推理流程、后处理等，向 LLM Manager 提供统一的自回归推理接口，支持并行解码插件化运行。
    - Modeling：提供深度定制优化的模块和内置模型，支持 ATB Models（Ascend Transformer Boost Models）和 MindSpore Models 两种框架。
        - 内置模块包括 Attention、Embedding、ColumnLinear、RowLinear、MLP（multilayer perceptron），支持 Weight 在线 Tensor 切分加载。
        - 内置模型使用内置模块进行组网拼接，支持 Tensor 切分，支持多种量化方式，用户亦可参照样例通过内置模块组网自定义模型。
        - 组网后的模型经过编译优化后，会生成能在昇腾 NPU 设备上加速推理的可执行图。

1. **基础能力**包括浮点、量化、并行。

浮点特性	浮点能力
float16	√
bfloat16	√

MindIE LLM 主打高性能推理，当前仅支持 float16、bfloat16 浮点格式。可通过配置模型 config.json 中'torch_dtype'字段进行类型修改。

量化特性	per channel	per token	per group
W8A8	√	√	×
W8A16	√	×	√
KV Cache int8	√	×	×
W8A8 稀疏量化	√	×	×

MindIE LLM 提供多种量化选择进行推理加速，用户可根据自己的需要进行选择。

表 4 并行特性

并行特性	并行能力
TP（Tensor Parallelism）	√
DP（Data Parallelism）	×
PP（Pipeline Parallelism）	×
EP（Expert Parallelism）	×

MindIE LLM 提供 TP 并行策略。

2. **模型能力**

MindIE LLM 提供如下所示模型预置能力，用户可根据需要进行使用，也可以对模型进行自定义开发迁移。

- LLaMA
- CodeLLaMA
- Baichuan
- Mixtral
- Qwen
- Bloom
- DeepSeek
- Glm
- CodeGeex

- Starcoder
- Gemma

3.7 阶段七：推理推理优化

3.7.1 基于 AscendCL 应用开发语言的应用优化方案

推理应用的精度、性能调优，由于是调优，因此在调优前，请确保已经完成了整网推理功能调测，功能不阻塞，只是推理精度错误、推理精度与标杆数据存在少量差距、模型推理性能不符合预期或待提升等问题。

- **应用的精度问题**可能由于推理功能与其它功能之间的串接问题、整网中算子本身的精度问题等，可参考本章中的建议排查功能串接时的接口参数配置问题、借助工具获取详细数据定位分析问题。
- **应用的性能问题**可能由于模型在昇腾 AI 处理器上的算子适配或数据读写问题、DVPP 接口使用问题等，可参考本章中的建议排查接口使用问题、借助工具优化模型、借助工具获取详细数据定位分析问题。

1、 **推理精度优化**

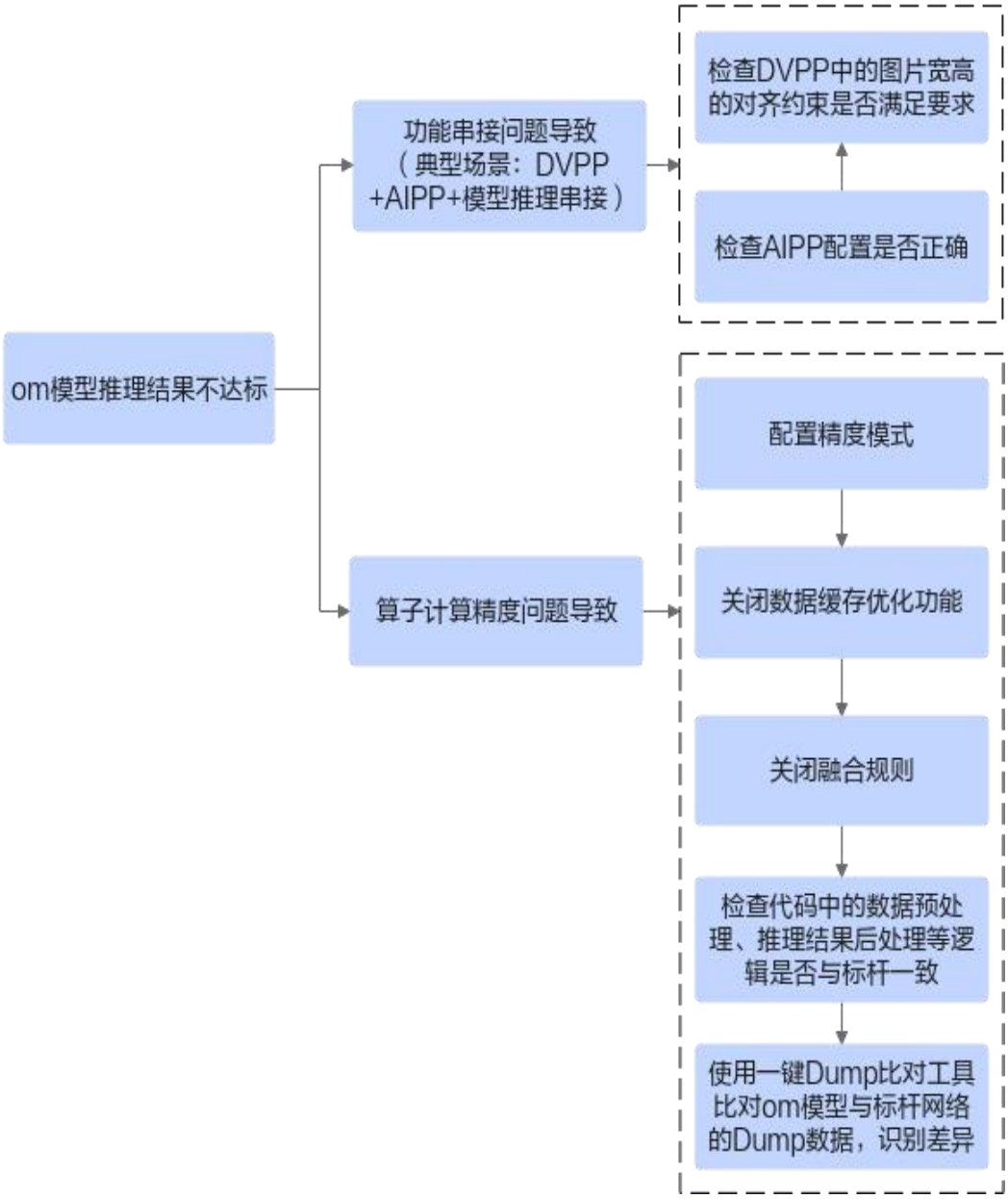
本文介绍整网推理场景下的精度调优流程、相关配置及典型案例等。由于是调优，因此在调优前，请确保已经完成了整网推理功能调测，功能不阻塞，只是推理精度错误，或推理精度与标杆数据存在少量差距。

在整网推理时，可能由于以下原因导致推理精度错误或者推理精度不达标：

- 推理功能与其它功能之间的串接问题，当前比较典型的是 DVPP+AIPP+模型推理串联使用的场景，该场景主要是因为接口中参数的配置问题、模型转换时 AIPP 的配置问题导致。

  如果存在 DVPP、AIPP 功能、模型推理串联使用的场景，建议先排查这部分问题；否则，可以跳过该排查，直接排查算子本身的精度问题。

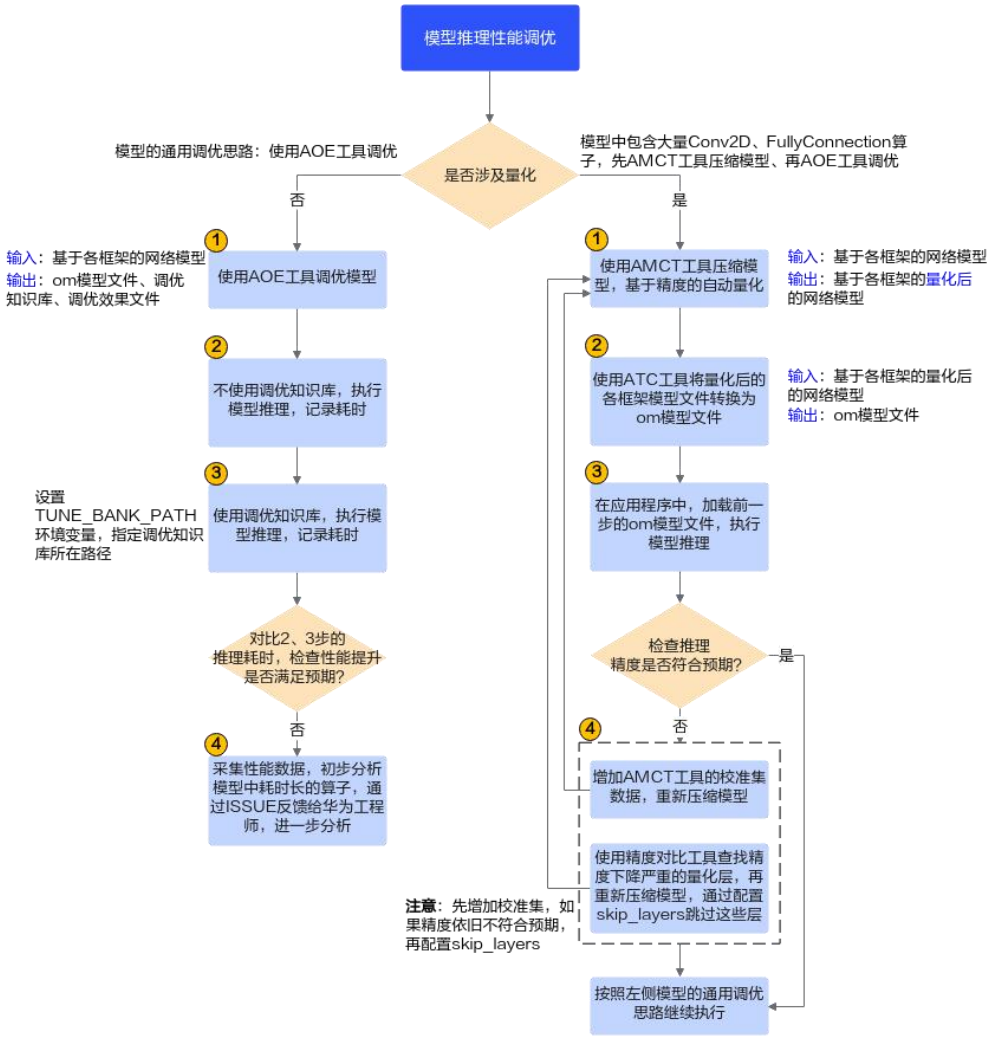
  本文也结合具体的正、反例给出了配置建议，供参考。
- 整网中算子本身的精度问题，该类问题可以借助精度比对工具，根据下文中具体的问题定位流程获取各类数据后，再进行比对、分析，确认是配置问题，还是算子实现问题，再逐一解决问题。本文中会结合具体的比对、分析的案例，介绍如何比对、分析。



2、 推理性能优化

在整网推理时，可能由于模型在昇腾 AI 处理器上的算子适配、数据读写等问题，导致模型推理的性能不符合预期，您可以查阅本节介绍的内容，了解模型推理时的性能调优流程。由于是调优，因此在调优前，请确保已经完成了整网推理功能调测，功能不阻塞，只是模型推理性能不符合预期、待提升。

在下图的性能调优流程中，涉及调优的关键工具为：模型调优工具 AOE（Ascend Optimization Engine）、模型压缩工具 AMCT（Ascend Model Compression Toolkit）。在调优过程中，涉及转换模型、记录模型推理耗时、分析性能瓶颈点等操作时，还会辅助使用模型转换工具 ATC、性能数据采集工具、精度比对工具。



3.7.2 基于 Python 应用开发语言的应用优化方案

1、 内存二次分配管理优化

用户内存管理有两种管理方式：

- 独立内存管理，根据需要单独申请所需的内存，内存不做拆分或者二次分配。
- 内存池管理内存，用户一次性申请一块较大内存，并在使用时从这块较大内存中二次分配所需内存。

在内存二次分配时，请使用如下接口从内存池申请对应内存。由于各接口对申请的内存地址、大小有约束，在内存池管理时，需要对该情况关注处理，否则容易出现内存越界。

计算机视觉领域一般涉及使用媒体数据处理功能，因此会涉及以上多种内存申请接口。内存首地址涉及 64 字节或 128 字节对齐，为方便统一管理，内存首地址**对齐值建议选取较大项**，比如内存首地址 128 字节对齐。

关于媒体数据处理时自行管理内存时的典型场景如下。

图 1 VDEC 场景

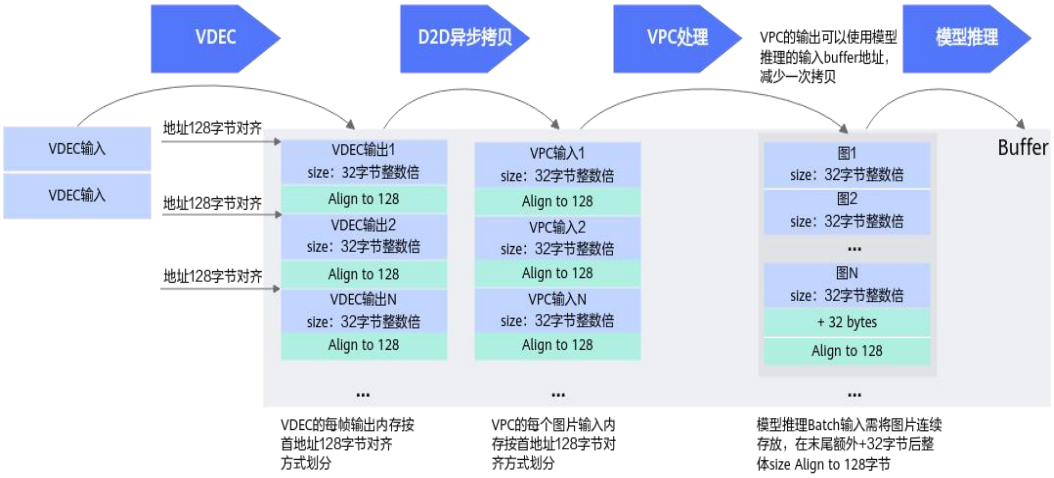
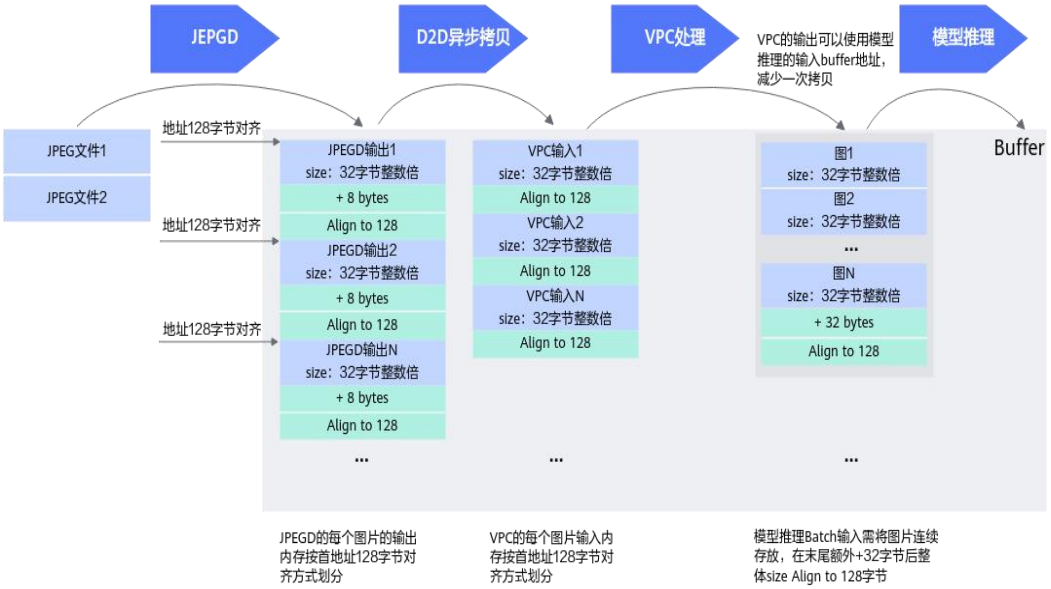


图 2 JPEGD 场景



2、 Profiling性能数据采集

接口用于 Profiling 采集性能数据，实现方式支持以下三种：

- **Profiling pyACL API（通过 Profiling pyACL API 采集并落盘性能数据）**

实现将采集到的Profiling数据写入文件，再使用Profiling工具解析该文件（请参见《性能调优工具用户指南》中的“使用msprof命令解析与导出性能数据”），并展示性能分析数据。

包括以下两种接口调用方式：

- acl.prof.init接口、acl.prof.start接口、acl.prof.stop接口、acl.prof.finalize接口配合使用，实现该方式的性能数据采集。该方式可获取pyACL的接口性能数据、AI Core上算子的执行时间、AI Core性能指标数据等。目前这些接口为进程级控制，表示在进程内任意线程调用该接口，其它线程都会生效。

  一个进程内，可以根据需求多次调用这些接口，基于不同的Profiling采集配置，采集数据。
- 调用acl.init接口，在pyACL初始化阶段，通过*.json 文件传入要采集的Profiling数据。该方式可获取pyACL的接口性能数据、AI Core上算子的执行时间、AI Core性能指标数据等。
  一个进程内，只能调用一次acl.init接口，如果要修改Profiling采集配置，需修改*.json文件中的配置。详细使用说明请参见acl.init接口处的说明，不在本章节描述。

- **Profiling pyACL API for Extension（Profiling pyACL API扩展接口）**
  当用户需要定位应用程序或上层框架程序的性能瓶颈时，可在Profiling采集进程内（acl.prof.start接口、acl.prof.stop接口之间）调用Profiling pyACL API扩展接口（统称为msproftx功能），开启记录应用程序执行期间特定事件发生的时间跨度，并将数据写入Profiling数据文件，再使用Profiling工具解析该文件，并导出展示性能分析数据。
  Profiling工具解析导出操作请参见《性能分析工具使用指南》中的“使用msprof命令解析与导出性能数据”。
  一个进程内，可以根据需求多次调用这些接口。接口调用方式：在acl.prof.start和acl.prof.stop接口之间调用acl.prof.create_stamp、acl.prof.push、acl.prof.pop、acl.prof.range_start、acl.prof.range_stop、acl.prof.destroy_stamp接口。该方式可获取应用程序执行期间特定时间发生的事件并记录事件发生的时间跨度。

  一个进程内，可以根据需求多次调用这些接口。
- **Profiling pyACL API for Subscription（订阅算子信息的Profiling pyACL API）**

  实现将采集到的Profiling数据解析后写入管道，由用户读入内存，再由用户调用pyACL的接口获取性能数据。

  接口调用方式：acl.prof.model_subscribe接口、acl.prof.get*接口、acl.prof.model_unsubscribe接口配合使用，实现该方式的性能数据采集，当前支持获取网络模型中算子的性能数据，包括算子名称、算子类型名称、算子执行时间等。

3、 采集溢出算子信息及分析

在调用acl.init接口初始化pyACL时，在JSON配置文件中增加溢出算子Dump配置。

JSON配置文件中的示例内容如下，示例中的“dump_path”以相对路径为例：


```
{
    "dump":{
        "dump_path":"output",
        "dump_debug":"on"
    }
}
```

当 dump_path 配置为相对路径时，您可以在“应用可执行文件的目录/{dump_path}”下查看导出的数据文件，针对每个溢出算子，会导出两个数据文件：

- 溢出算子的 dump 文件：命名规则如 {op_type}.{op_name}.{taskid}.{stream_id}.{timestamp}，如果 op_type、op_name 出现了“.”、“/”、“\”、空格时，会转换为下划线表示。

  用户可通过该信息知道具体出现溢出错误的算子，并通过解析溢出算子的 dump 文件获取该算子的输入和输出信息。

- 算子溢出数据文件：命名规则如 OpDebug.Node_Opdebug.{taskid}.{stream_id}.{timestamp}，其中 taskid 不是溢出算子的 taskid，用户不需要关注 taskid 的实际含义。

  用户可通过解析算子溢出数据文件获取溢出相关信息，包括溢出算子所在的模型、AICore 的 status 寄存器状态等。

  - 解析溢出算子的 dump 文件

1. 请根据实际情况，将{op_type}.{op_name}.{taskid}.{stream_id}.{timestamp}上传到安装有 Toolkit 软件包的环境。
2. 进入解析脚本所在目录，例如 Toolkit 软件包安装目录为：/home/HwHiAiUser/Ascend/ascend-toolkit/latest。

   ```
   cd /home/HwHiAiUser/Ascend/ascend-toolkit/latest/toolkit/tools/operator_cmp/compare
   ```

3. 执行 msaccucmp.py 脚本，转换 dump 文件为 numpy 文件。举例：

   ```
   python3 msaccucmp.py convert -d /home/HwHiAiUser/dump -out /home/HwHiAiUser/dumptonumpy -v 2
   ```

   说明

   -d 参数支持传入单个文件，对单个 dump 文件进行转换，也支持传入目录，对整个 path 下所有的 dump 文件进行转换。

4. 调用 Python，转换 numpy 文件为 txt 文件。举例：

   ```
   $ python3
   >>> import numpy as np
   >>> a = np.load("/home/HwHiAiUser/dumptonumpy/Pooling.pool1.1147.1589195081588018.output.0.npy")
   >>> b = a.flatten()
   ```



```
>>>
np.savetxt("/home/HwHiAiUser/dumptonumpy/Pooling.pool1.1147.1589195081588018.output.0.txt", b)
```

转换为.txt 格式文件后，维度信息、Dtype 均不存在。详细的使用方法请参考 numpy 官网介绍。

- 解析算子溢出数据文件

由于生成的溢出数据是二进制格式，可读性较差，需要通过工具将 bin 文件解析为用户可读性好的 JSON 文件。

1. 请根据实际情况，将溢出数据文件 OpDebug.Node_Opdebug.{taskid}.{timestamp}上传到安装有 Toolkit 软件包的环境。
2. 进入解析脚本所在路径，例如 Toolkit 软件包安装目录为：/home/HwHiAiUser/Ascend/ascend-toolkit/latest。

```
cd /home/HwHiAiUser/Ascend/ascend-toolkit/latest/toolkit/tools/operator_cmp/compare
```

3. 执行解析命令，例如：

```
python3 msaccucmp.py convert -d /home/HwHiAiUser/opdebug/Opdebug.Node_OpDebug.59.1597922031178434  -out /home/HwHiAiUser/result
```

关键参数：

- -d：溢出数据文件所在目录，包括文件名。
- -out：解析结果待存储目录，如果不指定，默认生成在当前目录下。

4. 解析结果文件内容如下所示。

```
{
    "DHA Atomic Add": {
        "model_id": 0,
        "stream_id": 0,
        "task_id": 0,
        "task_type": 0,
        "pc_start": "0x0",
        "para_base": "0x0",
        "status": 0
    },
    "L2 Atomic Add": {
        "model_id": 0,
        "stream_id": 0,
        "task_id": 0,
        "task_type": 0,
        "pc_start": "0x0",
        "para_base": "0x0",
        "status": 0
    },
    "AI Core": {
        "model_id": 514,
        "stream_id": 563,
```



```
            "task_id": 57,
            "task_type": 0,
            "pc_start": "0x1008005b0000",
            "para_base": "0x100800297000",
            "kernel_code": "0x1008005ae000",
            "block_idx": 1,
            "status": 32
        }
    }
```

参数解释：

- model_id：标识溢出算子所在的模型 id。
- stream_id：标识溢出算子所在的 streamid。
- task_id：标识溢出算子的 taskid。
- task_type：标识溢出算子的 task 类型。
- pc_start：标识溢出算子的代码程序的内存起始地址。
- para_base：标识溢出算子的参数的内存起始地址。
- kernel_code：标识溢出算子的代码程序的内存起始地址，和 pc_start 相同。
- block_idx：标识溢出算子的 blockid 参数。
- status：AICore 的 status 寄存器状态，用户可以从 status 值分析得到具体溢出错误。status 为 10 进制表示，需要转换成 16 进制，然后定位到具体错误。

例如：status 为 272，转换成 16 进制为 0x00000110，则可以判定出可能原因为 0x00000010+0x00000100。

- 0x00000008：符号整数最小负数 NEG 符号位取反溢出。
- 0x00000010：整数加法、减法、乘法或乘加操作计算有溢出。
- 0x00000020：浮点计算有溢出。
- 0x00000080：浮点数转无符号数的输入是负数。
- 0x00000100：FP32 转 FP16 或 32 位符号整数转 FP16 中出现溢出。
- 0x00000400：CUBE 累加出现溢出。

4、 特征向量检索

该部分主要实现了对特征检索的功能验证，生成随机底库，随机生成特征数据进行特征检索（当前支持 **1 ： N**、**M ： N** 两种检索模式，下文的示例代码以 **1 ： N** 为例）。大致可分为初始化、添加特征到底库、底库搜索、精准修改或删除底库特征、去初始化几个主要步骤，具体接口调用方式如下：

- 初始化：调用 acl.init 接口进行初始化、运行管理资源申请，调用 acl.fv.create_init_para 接口创建 aclfvInitPara 类型的数据来指定特征向量检索的初始化参数。
- 添加特征到底库：主要调用 acl.fv.create_feature_info 接口创建 aclfvFeatureInfo 类型数据来表示创建特征的描述信息，然后调用 acl.fv.repo_add 添加底库。
- 底库搜索：调用 acl.fv.search 接口来实现检索。

- 精准修改或删除底库特征：调用 acl.fv.delete 和 acl.fv.modify 接口来实现删除或修改底库中某个特征。下文的代码以删除底库特征为例。
- 去初始化：主要包括释放运行管理资源、调用 acl.fv.destroy_init_para 接口销毁 aclfvInitPara 类型的数据、调用 acl.fv.release 接口特征检索模块去初始化，释放内存空间。

修订记录

发布日期	修订记录
2025-6-25	第一次正式发布。

