CANN
6.3.RC2

TensorFlow 2.6.5 网络模型迁移和训练指南

文档版本 01

发布日期 2023-09-18

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 学习向导

读者对象

本文档适用于AI算法工程师，将基于TensorFlow 2.6.5的Python API开发的训练脚本迁移到昇腾AI处理器上执行训练，并达到训练精度性能最优。

掌握以下经验和技能可以更好地理解本文档：

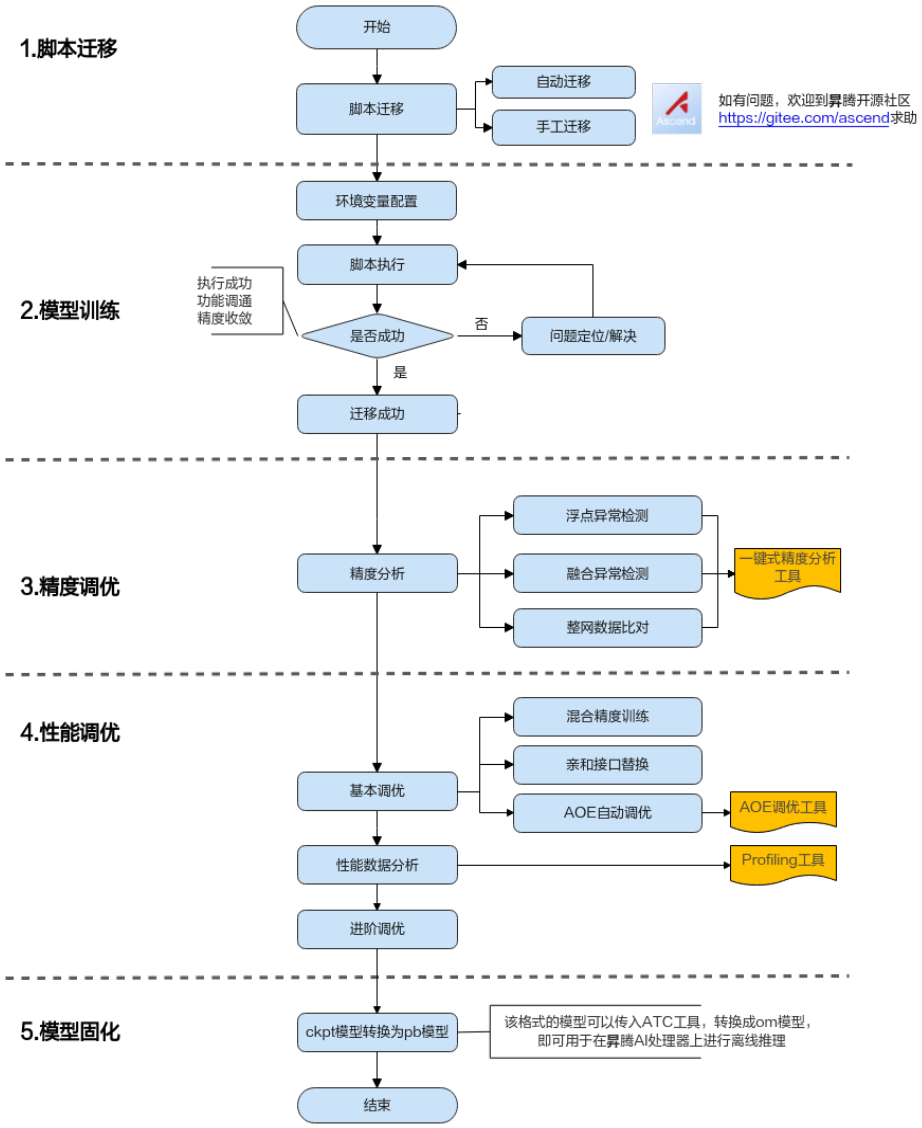
- 熟悉CANN软件基本架构以及特性。
- 熟练的Python语言编程能力。
- 熟悉TensorFlow的API。
- 对机器学习、深度学习有一定的了解，熟悉训练网络的基本知识与流程。

使用前须知

- 在昇腾AI处理器进行模型迁移之前，建议用户事先准备好基于TensorFlow 2.6.5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。
- 本文中的代码片段仅为示例，请用户使用时注意修改适配。

模型开发流程

模型开发的主要工作就是将TensorFlow原始模型迁移到昇腾AI处理器上并执行训练，主要流程如下图所示。

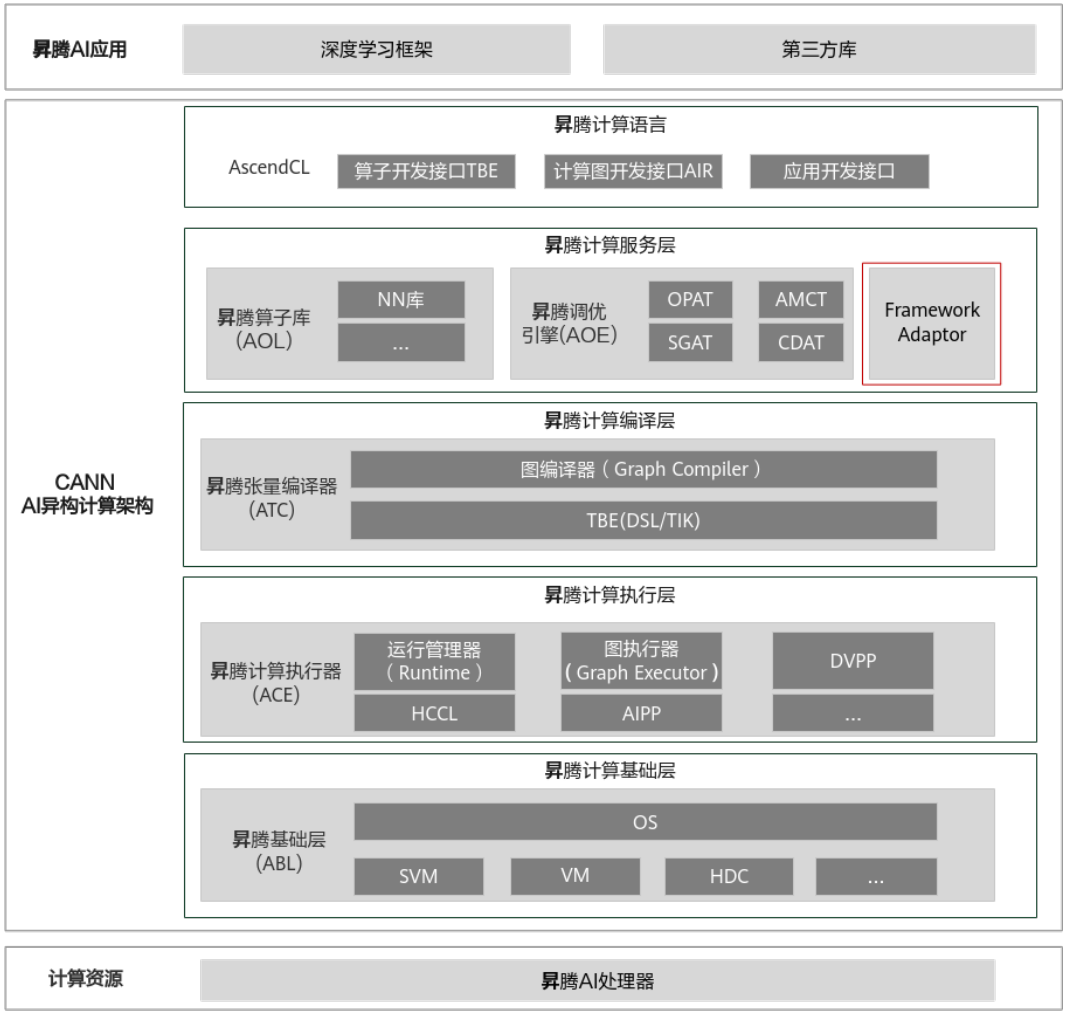


2 TF Adapter 简介

Ascend adapter for TensorFlow 2.x（下称TF Adapter）是TensorFlow 2.x（下称TF2）框架与CANN软件栈间的适配层，用于帮助TF2训练框架的使用者便捷地将训练迁移到昇腾AI处理器（简称NPU）上执行。当前版本的TF Adapter是无侵入的且与TF2有配套关系的Ascend发布件。

TF Adapter在昇腾AI软件栈中的位置如下图所示。

图 2-1 昇腾 AI 软件栈架构图



TF2 关键概念

TF Adapter涉及的TF2关键概念：

- **Eager模式**

  TF2默认执行方式，运算会返回具体的值，而非构建供稍后运行的计算图，更多介绍请参考**链接**。

- **Eager context**

  TF2 Eager模式下的运行上下文，全局唯一，context中持有线程变量成员，满足不同线程内执行上下文的差异化需求。

- **tf.function**

  TF2提供的python函数装饰器，用于将python函数中调用的TF2运算封装成graph执行，获取性能收益，更多介绍请参考**链接**。

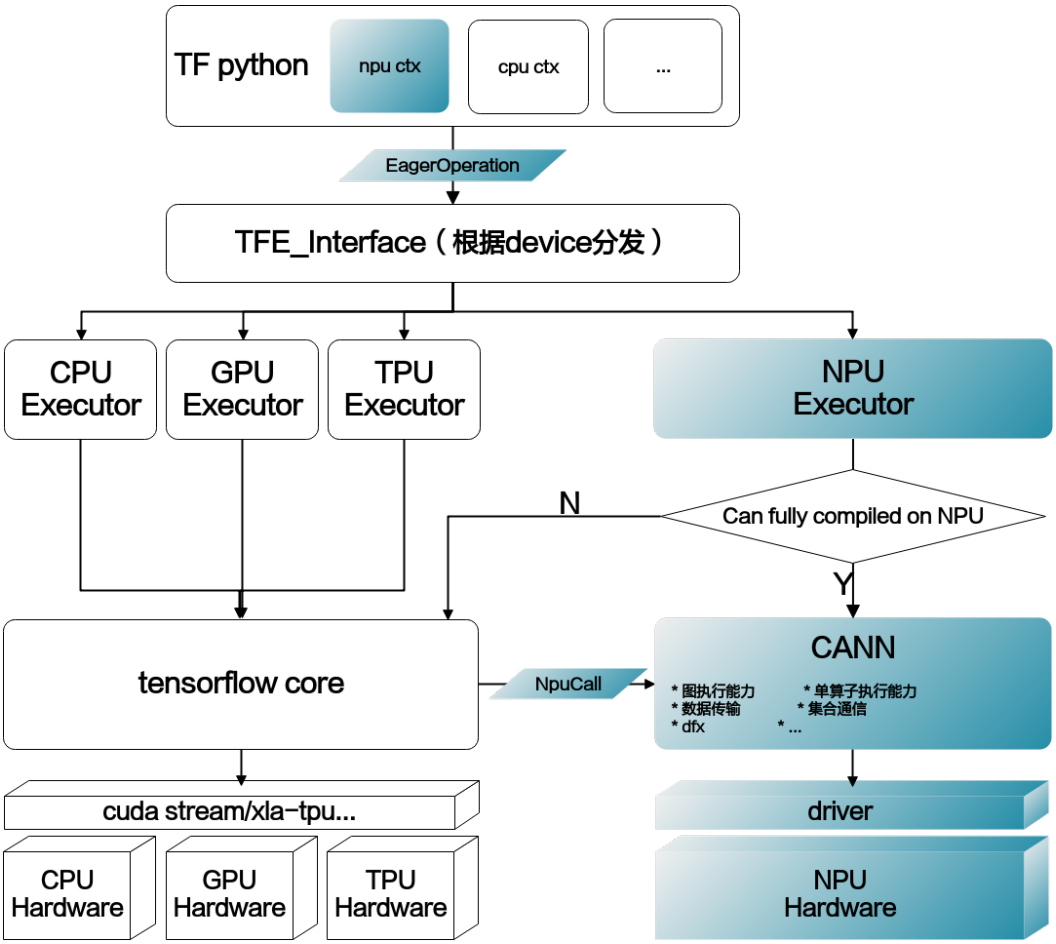
- **TF2自定义设备**

  TF2提供C接口TFE_RegisterCustomDevice提供注册自定义设备的能力，TF Adapter调用该接口将昇腾AI处理器注册成为TF的自定义设备，自定义设备与内置的CPU与GPU地位相当。TF2源码信息请参考**链接**。

TF Adapter 对接原理

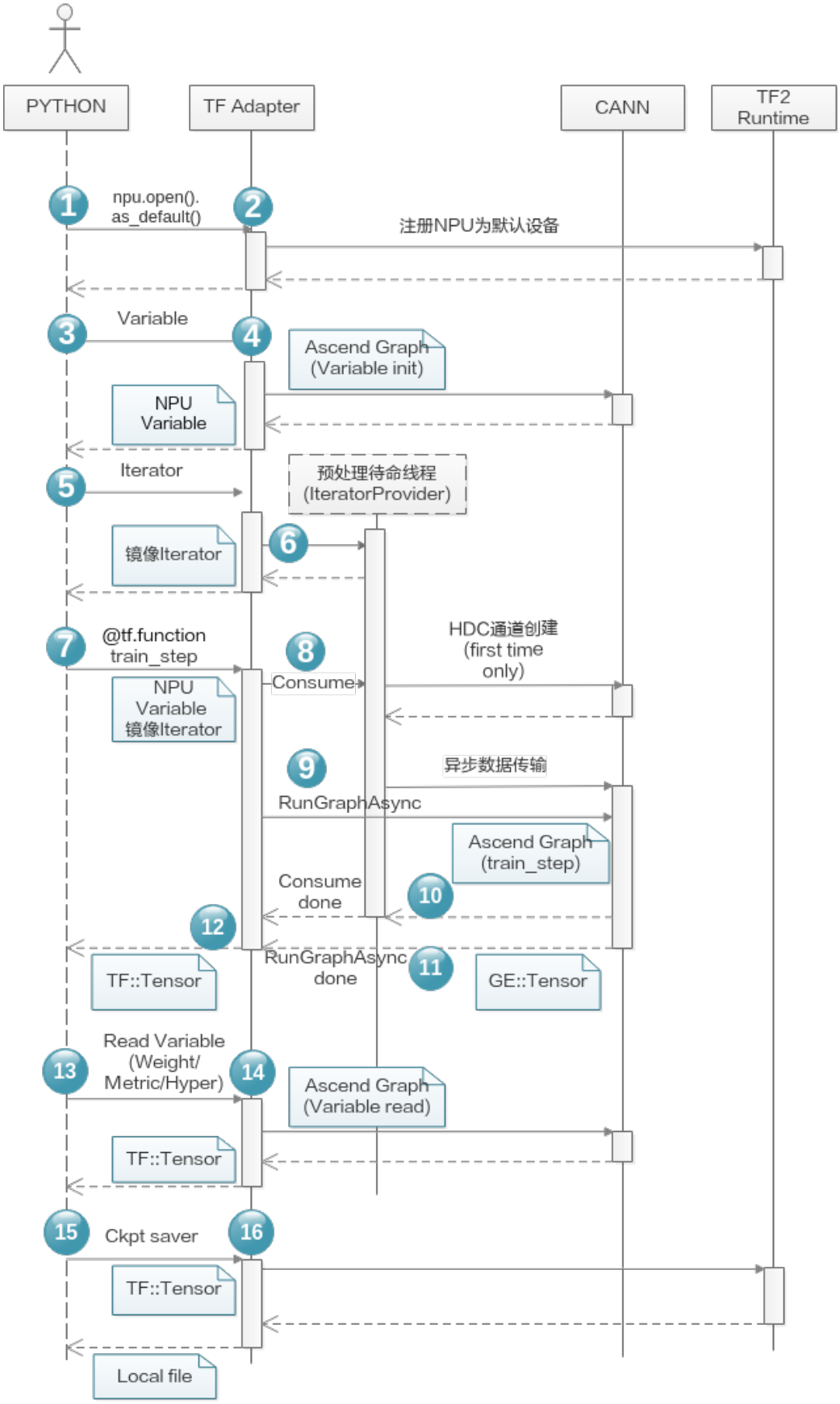
TF Adapter将昇腾AI处理器注册成为TF2自定义设备，并且设置为默认设备，注册成为默认设备后，所有用户指定到昇腾AI处理器或未指定执行设备的运算操作，都将被TF2框架分发到昇腾AI处理器执行，昇腾AI处理器的算子执行接口内部实现时调用CANN的算子/图执行能力，完成昇腾AI处理器上的算子执行。

TF Adapter对接框图，其中有色区域为TF Adapter对接适配部分。



TF Adapter对接时序图：

下图以一次典型的训练流程为例：包括设备初始化，模型（变量）初始化，执行训练，保存Checkpoint几个阶段。



时序图中涉及的概念说明：

- **CANN**

  昇腾AI处理器用户编程接口体系，详情请参考**链接**。
- **TF2 Runtime**

  这里指原生Tensorflow的运行时接口。
- **Iterator**

  Tensorflow数据输入Pipeline的迭代器，通过Iterator访问数据集，是Tensorflow的推荐方式也是昇腾AI处理器上性能亲和的方式，详情请参考**链接**。
- **HDC通道**

  Tensorflow进程到昇腾AI处理器硬件内存的数据传输通道，TF Adapter2.x在Tensorflow进程中通过HDC通道，异步地为昇腾AI处理器上的训练任务供给训练数据。

方案优势

TF Adapter当前对接方案优势：

- NPU成为TF2的自定义设备，从用户视角看来，NPU与GPU/CPU的存在形式一致，且能保持对TF2框架后续演进的兼容性。
- 算子级适配，兼容TF2框架原始特性。特别是函数算子，可充分发挥CANN的图处理优势，加速执行。
- 插件式无侵入对接CANN，无需重新编译部署TF2，快速完成在不同平台上的TF Adapter安装。

3 版本约束

- 该文档仅配套TensorFlow 2.6.5版本使用，当前版本同时支持TensorFlow 1.15版本的模型迁移，具体请参考《**TensorFlow 1.15网络模型迁移和训练指南**》。
- 本文档仅支持在以下昇腾AI处理器型号产品中使用。

  昇腾910 AI处理器

  昇腾910B AI处理器
- 当前版本不支持float64/complex64/complex128/DT_VARIANT数据类型。
- 只支持变量（tf.Variable）资源相关操作在NPU执行。
- 只支持tf.function修饰的函数算子在NPU执行。
- 不支持训练脚本中同时使用tf.compat.v1接口和TF 2.6中eager功能相关的API（如tf.estimator）。
- TensorFlow 2.6.5数据预处理过程默认在Host上执行，而变量需要下沉到Device上初始化，因此当TensorFlow 2.6.5训练脚本的数据预处理使用了变量时，在NPU训练会执行失败，因此需要将该逻辑嵌套在context.device（‘CPU:0’）下，将使用预处理的变量在Host上初始化。
- 集合通信约束：
  - 分布式训练场景下，HCCL会使用Host服务器的部分端口进行集群信息收集，需要操作系统预留该部分端口。默认情况下，HCCL使用60000-60015端口，若通过环境变量**HCCL_IF_BASE_PORT**指定了Host网卡起始端口，则需要预留以该端口起始的16个端口。

    操作系统端口号预留示例：sysctl -w net.ipv4.ip_local_reserved_ports=60000-60015
  - 针对昇腾910 AI处理器：
    - server内只支持1/2/4/8P粒度的分配。
    - allreduce和reduce_scatter仅支持int8、int32、float16和float32数据类型。
  - 针对昇腾910B AI处理器：allreduce和reduce_scatter仅支持int8, int32, float16, float32和bfp16数据类型。

4 环境准备

安装 TensorFlow 2.6

参考《CANN 软件安装指南》的“安装深度学习框架”章节，安装TensorFlow 2.6.5。

安装 TF Adapter 插件

两种安装方式，可任选其一：

- 基于已编译好的软件包安装：参考《CANN 软件安装指南》的“安装框架插件包”章节。
- 基于源码安装，具体操作请参考**链接**。

安装验证

```
import npu_device as npu
npu.open().as_default() # 初始化NPU为默认设备
```

设备初始化成功会打印：

```
Npu device instance /job:localhost/replica:0/task:0/device:NPU:0 created
```

默认执行device 0的初始化，您可以在**11.1.1.1 npu.open**接口中传入NPU设备的ID来初始化其他NPU设备。

5 自动迁移

5.1 工具简介

功能介绍

Ascend平台提供了TensorFlow 2.6.5网络迁移工具，该工具适用于原生的Tensorflow训练脚本迁移场景，AI算法工程师通过该工具分析原生的TensorFlow Python API在昇腾AI处理器上的支持度情况，同时将原生的TensorFlow训练脚本自动迁移成昇腾AI处理器支持的脚本，迁移后的脚本能在昇腾AI处理器上执行训练，功能跑通。对于无法自动迁移的API，您可以参考工具输出的迁移报告，对训练脚本进行相应的适配修改。

获取路径

CANN软件安装完成后，迁移工具在“tfplugin安装目录/tfplugin/latest/python/site-packages/npu_device/convert_tf2npu/”目录下。

使用限制

在使用工具进行模型迁移前，先来了解对原始训练脚本的限制：

1. 要求原始脚本在GPU/CPU上跑通，精度收敛。
2. 要求原始脚本仅使用**TensorFlow 2.6官方API**，或者当脚本中以tf.compat.v1形式调用Tensorflow 1.x API时，支持使用Horovod API。

   若用户脚本使用了其他第三方API，当前工具暂不支持迁移。例如：

   a. 不支持原生Keras API，但由于Tensorflow官方API中包括了Tensorflow的Keras API，因此支持Tensorflow的Keras API。

   b. 不支持CuPy API，即便原始脚本能在GPU上运行成功，但不能保证在昇腾AI处理器运行成功。
3. 原始脚本中的TensorFlow模块最好按照如下方式引用，否则工具迁移后，无法生成准确的迁移报告（但并不影响脚本迁移）。

   ```
   import tensorflow as tf
   import horovod.tensorflow as hvd
   ```


4. 当前版本不支持float64/complex64/complex128/DT_VARIANT数据类型。
5. 关于分布式脚本迁移的限制：
   a. 使用工具迁移前，需要手工添加数据集分片操作，具体请参考**3**。
   b. 当前工具仅支持对使用了Tensorflow Keras优化器（包括SGD/RMSprop/Adam/Ftrl/Adagrad/Adadelta/Adamax/Nadam）的分布式脚本进行自动迁移，其他分布式脚本需要参考**6.7 分布式训练脚本适配（兼容单卡）**进行手工迁移。
   c. 如果用户原始脚本中使用了LossScaleOptimizer，当前工具仅支持将tf.keras.mixed_precision.LossScaleOptimizer迁移为npu.train.optimizer.NpuLossScaleOptimizer，对于其他类型的LossScaleOptimizer，您应当先切换为tf.keras.mixed_precision.LossScaleOptimizer，进行功能精度验证后再手工替换为npu.train.optimizer.NpuLossScaleOptimizer。
6. 迁移工具目前无法自动使能循环下沉功能，如果原始脚本中使用了循环下沉，则需要用户手工使能NPU的循环下沉能力，具体请参考**6.6 训练循环下沉时设置NPU上的循环次数**。

5.2 迁移操作

前提条件

在昇腾AI处理器进行模型迁移之前，建议用户事先准备好基于TensorFlow 2.6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。

迁移操作

步骤1 安装依赖。

pip3 install pandas

pip3 install openpyxl

pip3 install google_pasta

步骤2 训练脚本扫描和自动迁移。

进入迁移工具所在目录，例如“tfplugin安装目录/tfplugin/latest/python/site-packages/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
```

其中main.py为工具入口脚本，参数说明如下所示：

表 5-1 参数说明

参数名	参数说明	可选/必选
-i	被迁移的原始脚本路径，当前该路径仅支持配置为文件夹，不支持单个文件。 **说明** ● 工具仅对-i参数指定的文件夹下的.py文件进行扫描和迁移。 ● 如果用户原始脚本跨目录存放，则建议放到一个目录执行迁移命令，或者在对应目录下依次执行迁移命令。	必选
-o	指定迁移后的脚本路径，该路径不能为原始脚本路径的子目录。 该参数可选，如果不指定，默认生成在当前路径下，例如output_npu_20220517172706/xxx_npu_20220517172706。	可选
-r	指定生成的迁移报告路径，该路径不能为原始脚本路径的子目录。 该参数可选，如果不指定，默认生成在当前路径下，例如report_npu_20220517172706。	可选
-m	Python执行入口文件。 如果原始脚本中没有main函数，由于迁移工具无法识别入口函数，因此无法进行NPU资源初始化，以及NPU训练相关配置。 对于以上场景，需要通过-m参数指定Python执行的入口文件，以便工具可以将用户脚本进行彻底迁移，保证后续训练的顺利执行。 配置示例：-m /root/models/xxx.py	可选
-d	如果原始脚本支持分布式训练，迁移时需要指定原始脚本使用的分布式策略，便于工具对分布式脚本进行自动迁移。取值： ● tf_strategy：表示原始脚本使用tf.distribute.Strategy分布式策略。 ● horovod：表示原始脚本使用horovod分布式模块。	可选
-c	如果在脚本使用了tf.compat.v1 API，控制以Tensorflow 1.x行为执行，需要在执行脚本转换命令时添加-c或者--compat。	可选

说明

通过python3 main.py -h可以获取迁移工具使用帮助。

- 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。

图 5-1 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。

图 5-2 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，请到**昇腾开源社区**求助。
  - 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。

----结束

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。

图 5-3 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 5-2 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API支持度	说明
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（无迁移方案，建议不使用）	此类API在昇腾AI处理器上不支持，且当前暂无具体迁移方案，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
```


6 手工迁移

6.1 简介

本节主要介绍TensorFlow 2.6脚本迁移时涉及的迁移点及其作用或原因，您将在**10.2 手工迁移与训练**章节看到具体的迁移示例以加深理解。如果迁移点中提及的部分API未在上文中出现，可在**11.1.1 TF Adapter 2.6 接口参考**中查阅API详细说明。

通过本节内容，您可以将TensorFlow 2.6脚本迁移到昇腾AI处理器执行训练，功能跑通后，如需进一步进行精度和性能调优，请参考**8 精度调优**和**9 性能调优**章节。

6.2 添加@tf.function 修饰

通常，官方发布或设计良好的脚本都会自带@tf.function修饰或者允许您通过脚本入参控制，您无需额外添加。

特别地，使用keras下Model.fit接口训练的脚本，TF2会在接口内部封装function，同样无需额外添加。

如果脚本中没有@tf.function修饰，您可以先阅读TF2对@tf.function的使用说明，详情见**链接**，然后为您的训练/验证/推理函数添加@tf.function修饰，并保证其在CPU或GPU环境下能正常工作。

6.3 设置 NPU 为默认设备

TF Adapter提供了注册NPU设备的API，将NPU注册为TensroFlow的合法设备。开发者可以在带有@tf.function装饰并在CPU或GPU下可以正常工作的脚本文件开头，添加如下代码，设置NPU为默认设备。

```
import npu_device as npu
# 默认设置Device 0为计算设备
npu.open().as_default()
```

您应当在import其他python包前执行该操作，防止在加载后续包的过程中有未分发到NPU的算子执行行为。

npu.open接口的详细说明可参见**11.1.1.1 npu.open**。

6.4 预处理 batch 动作设置 drop_reminder

当原始网络脚本中使用dataset.batch(batch_size)返回动态形状时，由于数据流中剩余的样本数可能小于batch大小，导致网络中最后一个step的shape与之前的shape不一致，此种场景下会进入动态shape编译流程。为提升网络编译性能，建议将drop_remainder设置为True，丢弃文件中的最后几个样本，确保网络中每个step的shape一致。

```
dataset = dataset.batch(batch_size, drop_remainder=True)
```

关于batch操作的更多细节，请参考**链接**。

6.5 替换 LossScaleOptimizer

如果您的原始脚本中未使用LossScaleOptimizer，可以直接跳过该迁移点。

通常，用户原始脚本在使用混合精度提升性能时，会使用LossScaleOptimizer保证精度，由于NPU上浮点溢出的行为是全局标志置位而非产生Inf或NaN的输出，所以您需要使用NPU提供的**11.1.1.9 npu.train.optimizer.NpuLossScaleOptimizer**以获取正确的溢出检测结果。

npu.train.optimizer.NpuLossScaleOptimizer使用方法与tf.keras.mixed_precision.LossScaleOptimizer完全一致，使用细节可参考**链接**。

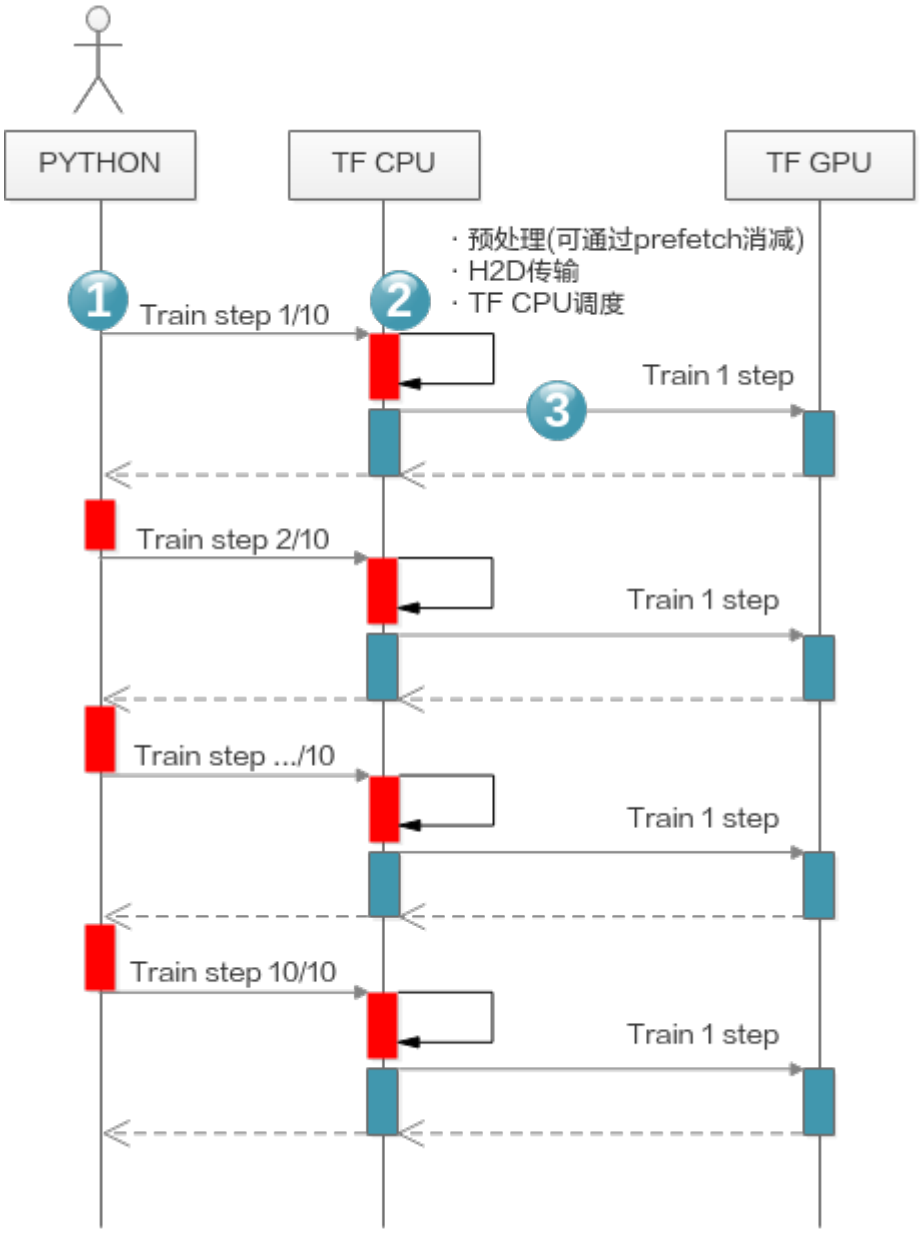
如果您的脚本中使用的是tf.keras.mixed_precision.LossScaleOptimizer，直接替换为npu.train.optimizer.NpuLossScaleOptimizer即可。如果您使用了其他类型的LossScaleOptimizer，您应当先切换为tf.keras.mixed_precision.LossScaleOptimizer，进行功能精度验证后再进行上述替换。

6.6 训练循环下沉时设置 NPU 上的循环次数

该迁移点只要求设置一个**NPU上的循环下沉次数**，我们称之为**npu loop size**，您可以通过**NPU_LOOP_SIZE**的环境变量设置，也可以通过在您的训练脚本中调用**11.1.1.8 npu.set_npu_loop_size**接口进行设置，修改的内容很简单，但是需要您理解**npu loop size**的含义。

npu loop size用于实现NPU训练的极致性能，在介绍**npu loop size**的由来前，先介绍TF2原生流程中的一些性能损耗点。以GPU为例，GPU环境常规训练方式下的工作时

序，如下图所示，脚本侧用户控制执行十次训练，每次在GPU上执行一次训练步，训练步结束后，回到python侧，用户判断当前步数等于10，启动下一次训练，直至训练完十次训练。



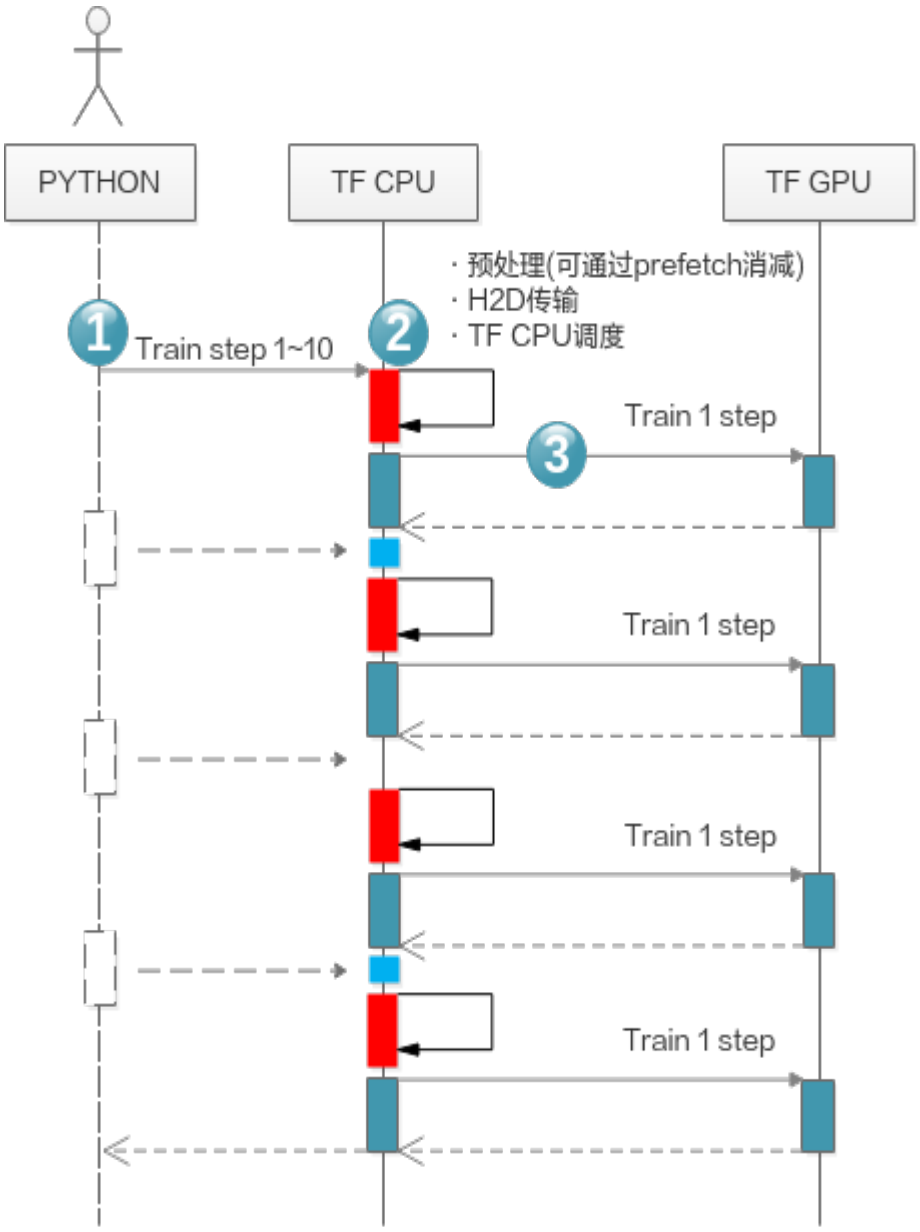
观察时序图上的有色区域我们不难发现，此时不论是CPU还是GPU都是间歇性工作的，该模式下的缺陷：

- Python解释器存在额外开销，且运行耗时不稳定，两个训练步间的间隙造成性能黑洞。
- 数据预处理与GPU训练过程的流水不充分，虽然TF2 Dataset的prefetch功能可以消减预处理过程的耗时影响，但是每次训练时H2D（Host to Device）的数据传输以及CPU调度耗时是无法忽略的。

TF2为了省去Python解释器上的额外开销，推荐用户使用While算子来实现训练循环（也就是所谓的循环下沉，循环下沉并非NPU特有的策略），此时判断训练是否达到指定步数的逻辑不再在Python解释器中进行，而是依赖TF2中的While算子，在编码时，使用者应当这样组织自己的训练（下称“循环下沉的编码方式”）：


```
@tf.function
def loop_train(iterator, steps):
    for i in tf.range(steps):
        train_step(next(iterator))
```

这样的TF2代码，在编译后，会将训练步嵌套在While算子中执行，时序变为下图所示：



可以看出，采用循环下沉的策略后，在Python解释器上的耗时转移到TF CPU上，耗时更短也更稳定，但是在该形式下，仍然有两部分额外开销：

- 预处理H2D数据传输。
- 判定训练到达指定步数的算子计算耗时。

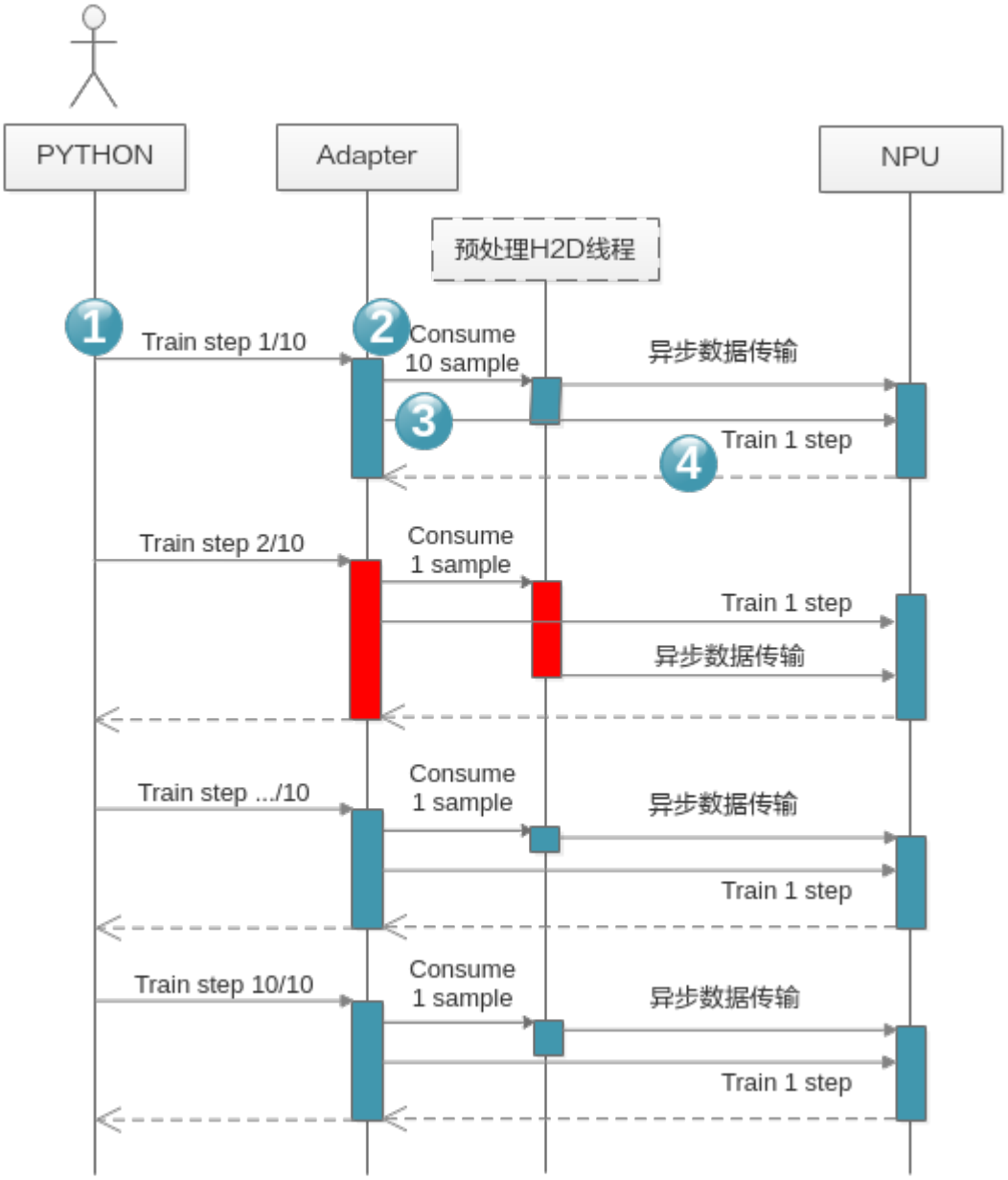
NPU为了达到极致性能，采取了两个策略来消除这两部分额外耗时：

- 异步预处理H2D线程，使得预处理输出传输与NPU训练完全异步，H2D的传输隐藏在NPU训练过程中。

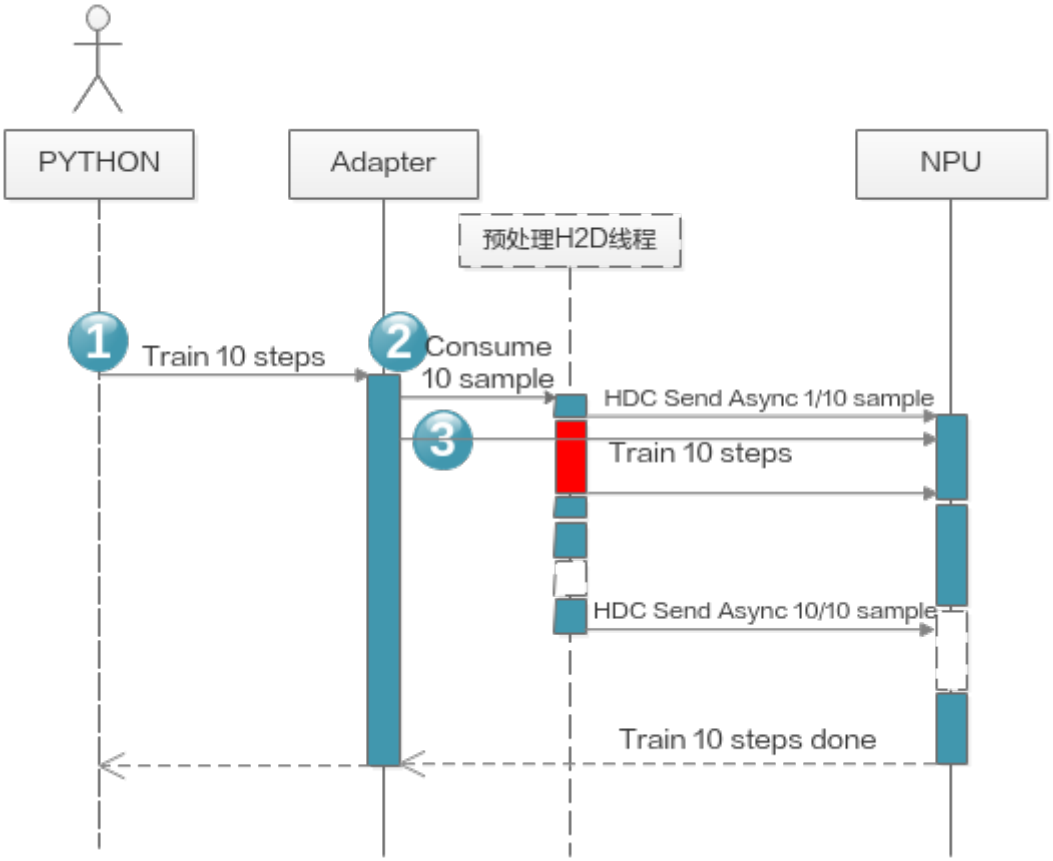
- **需要用户指定训练循环下沉次数**，消除次数判断算子计算耗时（也用于指示预处理数据H2D异步传输次数）。

异步数据传输指TF Adapter的预处理线程主动向NPU发送训练数据，在未使用循环下沉的方式编码时，执行时序如下所示：

此时可以一定程度上消减数据预处理H2D数据传输与CPU调度的耗时（下发训练步的同时，数据传输正在进行）。



当使用了训练循环下沉的编码方式时，NPU上的执行时序图为：



可以看到：

- 脚本发起在NPU上训练十次的请求后，直到训练结束，都不会再与Python解释器交互，而是单纯的NPU运算。
- 预处理的耗时抖动，可以被前面训练步中预处理领先NPU运算的耗时抵消，从而可以抵御更大的数据预处理性能波动。

NPU训练循环下沉与异步预处理数据传输方式可以最大程序地减少训练计算无关的耗时，最大化性能收益，但同时对用户训练有额外约束：

由于预处理线程与NPU训练步异步，在使用循环下沉的编码方式时，需要告诉NPU当前的循环下沉次数，所以NPU要求用户在使用循环下沉的编码方式时，额外设置**npu loop size**，用于指示循环下沉执行的次数。

比如，您使用循环下沉的编码方式组织您的训练：

```
@tf.function
def loop_train(iterator, steps):
    for i in tf.range(steps):
        train_step(next(iterator))
```

当您期望每次loop_train调用会在NPU上训练100个Step时，此时您有两种方式设置**npu loop size**：

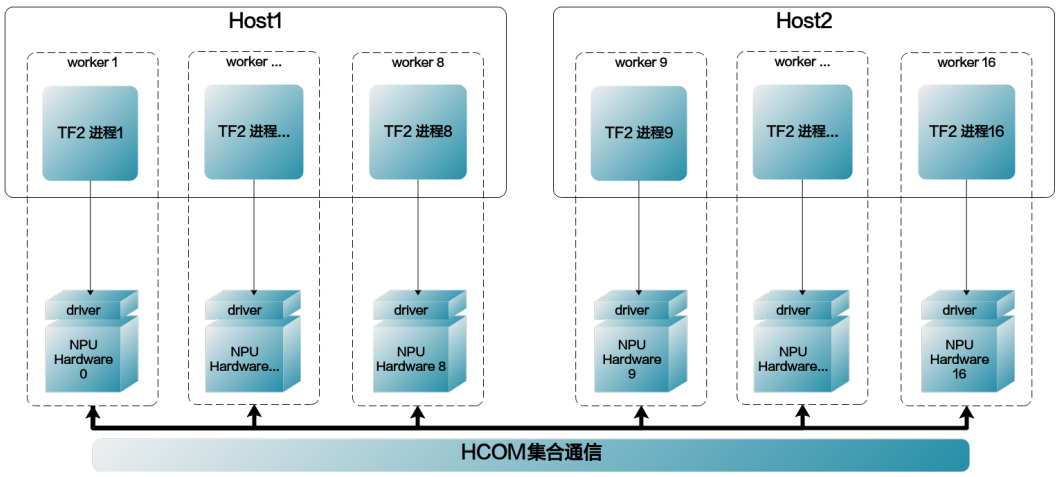
- 在启动训练前通过NPU_LOOP_SIZE的环境变量设置：
  ```
  export NPU_LOOP_SIZE=100
  ```
- 在Python脚本调用loop_train前调用**11.1.1.8 npu.set_npu_loop_size**设置，然后就可以调用loop_train，并传入循环次数100：
  ```
  npu.set_npu_loop_size(100)
  loop_train(train_iter, tf.constant(100))
  ```


您也可以在训练过程中调用npu.set_npu_loop_size来改变NPU上每次下沉执行的步数，比如您总共训练100个Step，希望每次在NPU上循环执行30Step，显然，最后的91~100Step小于**npu loop size**，此时你可以在训练完90Step后调用**11.1.1.8 npu.set_npu_loop_size**来调整**npu loop size**的大小：

```
remaining_steps = 100  # 剩余Steps数
base_loop_size = 30  # 基准npu loop size
npu.set_npu_loop_size(base_loop_size)
while remaining_steps >= base_loop_size:  # 按照基准loop循环下沉训练，直到剩余Step数不足一次loop
    loop_train(train_iterator, tf.constant(base_loop_size))
    remaining_steps -= base_loop_size
if remaining_steps > 0:  # 如果还有未处理的数据，调整为一个较小的npu loop size处理
    npu.set_npu_loop_size(remaining_steps)
    loop_train(train_iterator, tf.constant(remaining_steps))
```

6.7 分布式训练脚本适配（兼容单卡）

NPU上的分布式部署形态如下图所示，每个Tensorflow进程只管理独享的一张NPU训练卡，多个Tensorflow进程间，通过CANN提供的集合通信接口进行集群同步。单独观察某个worker，可以发现其与NPU上的单卡训练，除额外进行了集群内的集合通信外完全一致。



TF Adapter适配时，将单卡NPU视作集群worker数量为1的分布式部署形态，因而NPU的单卡脚本和分布式脚本最终是一致的。

NPU上执行分布式，相较于单卡NPU训练，主要有三部分的额外适配工作：

1. **worker间变量初值同步**

   TF2 Eager模式下，变量在模型生成后即完成初始化，此时需要进行变量初值同步操作，使各个worker上的变量初值一致。

   在模型构建完成后，您应当调用**11.1.1.4 npu.distribute.broadcast**接口完成变量初值同步，该接口要求传入需要进行worker间值同步的变量，通常，您可以通过model.trainable_variables来获取全部需要同步的变量。

2. **worker间梯度聚合**

   执行训练时，不同worker上产生不同的梯度信息grads，通过对多个worker上的梯度进行聚合计算，可以更准确地评估当前训练的误差情况。

   - 当原始脚本中分步骤计算并更新梯度（例如tf.gradient和opt.apply_gradient）时，则需要调用**11.1.1.3 npu.distribute.all_reduce**接口完成梯度聚合运算，该接口要求您传入需要进行worker间聚合计算的梯度以及聚合运算的类型（通常是求平均值）。

- 当原始脚本中计算和更新梯度操作被集成到同一接口中（例如minimize/model.fit）时，则需要调用**11.1.1.5 npu.distribute.npu_distributed_keras_optimizer_wrapper**完成梯度聚合运算。

3. **不同worker上的数据集分片**

   分布式训练时，应当保证每个worker上评估的样本不同，这样才能使得训练结果更符合样本集真实分布，比如您在一个8卡NPU的集群中执行训练，此时一个典型的策略就是第一张NPU卡上训练0-1/8的数据，第二张NPU卡训练1/8-2/8的数据，最后一张卡上训练7/8-8/8的数据。

   - 当数据集为tf.data.Dataset格式时，TF Adapter提供了**11.1.1.6 npu.distribute.shard_and_rebatch_dataset**接口帮您实现上述切分动作，该接口要求您传入需要进行集群切分的dataset（Dataset介绍参考**链接**）以及集群训练时的全局batch大小，例如：

     ```
     # 由于需要使用npu.distribue.shard_and_rebatch接口，在脚本开头import npu
     import npu_device as npu

     if input_context:
       logging.info(
           'Sharding the dataset: input_pipeline_id=%d num_input_pipelines=%d',
           input_context.input_pipeline_id, input_context.num_input_pipelines)
       # 原始的shard逻辑，因为以单机CPU方式启动，所以不会进行实际的shard
       dataset = dataset.shard(input_context.num_input_pipelines, input_context.input_pipeline_id)
     # NPU添加的shard逻辑，会根据集群数量，对数据集和全局batch进行切分
     dataset, batch_size = npu.distribute.shard_and_rebatch_dataset(dataset, batch_size)
     ```

   - 当数据集为Numpy数组时，需要调用numpy方法手工对数据集和全局batch进行切分，例如：

     ```
     (x_train, _), (x_test, _) = keras.datasets.mnist.load_data(os.path.join(args.data_path, 'mnist.npz'))

     # 根据设备数量均分数据集
     x_trains = np.split(x_train, args.rank_size)
     # 按设备编号取对应的数据集分片
     x_train = x_trains[args.device_id]
     x_tests = np.split(x_test, args.rank_size)
     x_test = x_tests[args.device_id]
     # 对全局batch进行切片
     batch_size = args.batch_size // args.rank_size

     mnist_digits = np.concatenate([x_train, x_test], axis=0)
     mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255
     ```

6.8 启动训练参数保持与单卡 CPU 形态一致

该迁移点通常是要求您修改启动脚本时的分布式相关参数。

当前版本我们需要您以单卡CPU的形态启动训练，这句话的含义是指，在昇腾AI处理器上启动训练时，保持与在单卡CPU上启动训练时的参数一致。

设计良好的脚本不会对部署形态做任何假设，配置脚本部署形态为单卡CPU通常只是启动脚本时入参的调整。

这里需要您评估脚本的启动参数，如果脚本支持传入分布式策略，请传入单卡分布式策略（OneDeviceStrategy）。如果支持配置GPU的数量，请配置为0。

我们希望您这么做是因为：

- 单卡CPU的部署形态与NPU的单卡训练形态一致，TF Adapter可以看到较为纯净的训练过程，使得迁移成功率大大提高。
- 可以屏蔽原有脚本默认分布式策略的干扰，使得分布式迁移成功率大大提高。

6.9 compat.v1 脚本手工迁移

6.9.1 迁移背景

TensorFlow 2.6中提供compat模块，用于支持兼容性问题。其中compat.v1模块用于兼容TensorFlow 1.x中的API，即可以在TensorFlow 2.6中使用1.x版本的API，即使该API在TensorFlow 2.6中已经弃用。

在实际场景中，存在部分Tensorflow 2.6脚本以tf.compat.v1形式调用Tensorflow 1.x API的情况，用于以TensorFlow 1.x原有的Session方式（Estimator/Session/Keras）控制脚本的执行行为。对于这类情况的训练脚本，支持迁移到昇腾AI处理器上，且迁移点与TensorFlow 1.x的迁移基本一致。

6.9.2 Estimator 迁移

Estimator 简介

Estimator API属于TensorFlow的高阶API，在2018年发布的TensorFlow 1.10版本中引入，它可极大简化机器学习的编程过程。Estimator有很多优势，例如：对分布式的良好支持、简化了模型的创建工作、有利于模型开发者之间的代码分享等。

TensorFlow 2.6版本中继续支持该高阶API，如果需要沿用在TensorFlow1.x版本中的用法，则可以通过compat.v1模块调用，调用方式如下：

```
tf.compat.v1.estimator.Estimator
```

使用compat.v1的Estimator进行训练脚本开发的流程为：

1. 数据预处理，创建输入函数input_fn。
2. 模型构建，构建模型函数model_fn。
3. 运行配置，实例化Estimator，并传入Runconfig类对象作为运行参数。
4. 执行训练，在Estimator上调用训练方法Estimator.train()，利用指定输入对模型进行固定步数的训练。

下面介绍如何迁移此类Estimator训练脚本，以便在昇腾AI处理器上进行训练。

头文件增加

对于以下步骤中涉及修改的python文件，新增以下头文件引用，用于导入NPU相关库。

```
import npu_device
from npu_device.compat.v1.npu_init import *
npu_device.compat.enable_v1()
```

数据预处理

一般情况下，此部分代码无需改造。如下情况需要进行适配修改：

当原始网络脚本中使用dataset.batch(batch_size)返回动态形状时，由于数据流中剩余的样本数可能小于batch大小，导致网络中最后一个step的shape与之前的shape不一致，此种场景下会进入动态shape编译流程。为提升网络编译性能，建议将

drop_remainder设置为True，丢弃文件中的最后几个样本，确保网络中每个step的shape一致。

```
dataset = dataset.batch(batch_size, drop_remainder=True)
```

但需要注意的是：推理时，当最后一次迭代的推理数据量小于batch size时，需要补齐空白数据到batch size，因为有些脚本最后会加个断言，验证结果的数量要和验证数据的数量一致。

```
assert num_written_lines == num_actual_predict_examples
```

模型构建

一般情况下，此部分代码无需改造。如下情况需要进行适配修改：

- 如果原始网络中使用到了tf.device，需要删除相关代码。
- 对于原始网络中的gelu，建议替换为CANN对应的API实现，以获得更优性能。

  TensorFlow原始代码：

  ```
  def gelu(x):
    cdf = 0.5 * (1.0 + tf.tanh(
       (np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
    return x*cdf
  layers = gelu()
  ```

  迁移后的代码：

  ```
  layers = npu_unary_ops.gelu(x)
  ```

运行配置

TensorFlow通过Runconfig配置运行参数，用户需要按照如下示例，更改config相关配置。

TensorFlow原始代码：

```
session_config=tf.compat.v1.ConfigProto(allow_soft_placement=True,log_device_placement=False))

config=tf.estimator.RunConfig(
  session_config=session_config,
  model_dir=FLAGS.model_dir,
  save_checkpoints_steps=FLAGS.save_checkpoints_steps,
```

迁移后的代码：

```
session_config=tf.compat.v1.ConfigProto(allow_soft_placement=True,log_device_placement=False))
custom_op = sess_config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
sess_config.graph_options.rewrite_options.remapping = rewriter_config_pb2.RewriterConfig.OFF

npu_config=NPURunConfig(
  session_config=sess_config,
  model_dir=FLAGS.model_dir,
  save_checkpoints_steps=FLAGS.save_checkpoints_steps,
```

创建 Estimator

Estimator的迁移只需要更改其config参数为上述npu_config，并将TensorFlow的Estimator迁移为NPUEstimator。

TensorFlow原始代码：

```
mnist_classifier=tf.compat.v1.estimator.Estimator(
  model_fn=cnn_model_fn,
  config=config,
  model_dir="/tmp/mnist_convnet_model")
```


迁移后的代码：

```
mnist_classifier=NPUEstimator(
  model_fn=cnn_model_fn,
  config=npu_config,
  model_dir="/tmp/mnist_convnet_model"
  )
```

执行训练

利用指定输入对模型进行固定步数的训练，此部分代码无需改造。

```
mnist_classifier.train(
  input_fn=train_input_fn,
  steps=20000,
  hooks=[logging_hook])
```

6.9.3 sess.run 迁移

sess.run 简介

sess.run API属于TensorFlow的低阶API，相对于Estimator来讲，灵活性较高，但模型的实现较为复杂。

TensorFlow 2.6版本中已经弃用该API，如果需要在TensorFlow 2.6版本中使用sess.run，需要通过compat.v1模块引用，方式如下：

```
tf.compat.v1.Session.run
```

使用sess.run API进行训练脚本开发的流程为：

1. 数据预处理。
2. 模型搭建/计算Loss/梯度更新。
3. 创建session并初始化资源。
4. 执行训练。

下面介绍如何迁移此类sess.run训练脚本，以便在昇腾AI处理器上进行训练。

头文件增加

对于以下步骤中涉及修改的python文件，新增以下头文件引用，用于导入NPU相关库。

```
import npu_device
from npu_device.compat.v1.npu_init import *
npu_device.compat.enable_v1()
```

数据预处理

一般情况下，此部分代码无需改造。如下情况需要进行适配修改：

当原始网络脚本中使用dataset.batch(batch_size)返回动态形状时，由于数据流中剩余的样本数可能小于batch大小，导致网络中最后一个step的shape与之前的shape不一致，此种场景下会进入动态shape编译流程。为提升网络编译性能，建议将drop_remainder设置为True，丢弃文件中的最后几个样本，确保网络中每个step的shape一致。

```
dataset = dataset.batch(batch_size, drop_remainder=True)
```


但需要注意的是：推理时，当最后一次迭代的推理数据量小于batch size时，需要补齐空白数据到batch size，因为有些脚本最后会加个断言，验证结果的数量要和验证数据的数量一致。

```
assert num_written_lines == num_actual_predict_examples
```

模型搭建/计算 Loss/梯度更新

一般情况下，此部分代码无需改造。如下情况需要进行适配修改：

- 如果原始网络中使用到了tf.device，需要删除相关代码。
- 对于原始网络中的gelu，建议替换为CANN对应的API实现，以获得更优性能。

  TensorFlow原始代码：

  ```
  def gelu(x):
    cdf = 0.5 * (1.0 + tf.tanh(
       (np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
    return x*cdf
  layers = gelu()
  ```

  迁移后的代码：

  ```
  layers = npu_unary_ops.gelu(x)
  ```

创建 session 并初始化资源

在昇腾AI处理器上通过sess.run模式执行训练脚本时，相关配置说明：

- 以下配置默认关闭，请勿开启：
  rewrite_options.disable_model_pruning
- 以下配置默认开启，请勿关闭：
  - rewrite_options.function_optimization
  - rewrite_options.constant_folding
  - rewrite_options.shape_optimization
  - rewrite_options.arithmetic_optimization
  - rewrite_options.loop_optimization
  - rewrite_options.dependency_optimization
  - rewrite_options.layout_optimizer
- 以下配置默认开启，必须显式关闭：
  - rewrite_options.remapping
  - rewrite_options.memory_optimization

TensorFlow原始代码：

```
#构造迭代器
iterator=Iterator.from_structure(train_dataset.output_types,train_dataset.output_shapes)

#取batch数据
next_batch=iterator.get_next()

#迭代器初始化
training_init_op=iterator.make_initializer(train_dataset)

#变量初始化
init=tf.compat.v1.global_variables_initializer()
sess=tf.compat.v1.Session()
sess.run(init)
```



```
#Get the number of training/validation steps per epoch
train_batches_per_epoch=int(np.floor(train_size/batch_size))
```

迁移后的代码：

```
#构造迭代器
iterator=Iterator.from_structure(train_dataset.output_types,train_dataset.output_shapes)

#取batch数据
next_batch=iterator.get_next()

#迭代器初始化
training_init_op=iterator.make_initializer(train_dataset)

#变量初始化
init=tf.compat.v1.global_variables_initializer()

#创建session
config = tf.compat.v1.ConfigProto()
custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF  # 必须显式关闭
config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF  # 必须显式关闭
sess = tf.compat.v1.Session(config=config)
sess.run(init)

#Get the number of training/validation steps per epoch
train_batches_per_epoch=int(np.floor(train_size/batch_size))
```

tf.compat.v1.Session原生功能在Ascend平台上全部支持。

另外，Ascend平台还支持自动混合精度等功能，如果用户需要进行相关使能，可以参考对应接口说明。

执行训练

此部分代码无需改造，例如：

```
#开始循环迭代
for epoch in range(num_epochs):
  ##Initialize iterator with the training dataset
  sess.run(training_init_op)
  for step in range(train_batches_per_epoch):
    #get next batch of data
    img_batch,label_batch=sess.run(next_batch)
    #run the training op
    _,train_loss = sess.run([train_op, loss],feed_dict={x:img_batch,y_:label_batch,is_training:True})
```

但是，如果用户训练脚本中没有使用with创建session，比如将session对象作为自己定义的一个类成员，那么需要在迁移后的脚本中显式调用sess.close()。

```
sess = tf.compat.v1.Session(config=config)
sess.run(...)
sess.close()
```

这是因为，Geop的析构函数在tf.compat.v1.session的close方法中会被调用到，如果是with创建的session，with会调用session的__exit__方法，里面会自动调用close：

```
with tf.compat.v1.Session(config=config) as sess:
    sess.run(...)
```

如果是其他情况，比如是把session对象作为自己定义的一个类成员，那么退出之前需要显式调用sess.close()，这样才可以保证退出的正常。

6.9.4 Keras 迁移

Keras 简介

Keras和Estimator类似，都属于TensorFlow高阶API，提供了方便的构图功能，并对训练、评估、验证、导出提供了方便的接口。

TensorFlow 2.6版本中继续支持Keras API，如果需要沿用在TensorFlow1.x版本中的用法，则可以通过compat.v1模块调用，调用方式如下：

```
tf.compat.v1.Session
```

使用Keras API进行训练脚本开发的一般步骤为：

1. 数据预处理。
2. 模型搭建。
3. 模型编译。
4. 模型训练。

须知

当前仅支持通过Tensorflow的Keras API编写的训练脚本，而不支持原生Keras API。

下面介绍如何迁移此类Keras训练脚本，以便在昇腾AI处理器上进行训练。

头文件增加

对于以下步骤中涉及修改的python文件，新增以下头文件引用，用于导入NPU相关库。

```
import npu_device
from npu_device.compat.v1.npu_init import *
npu_device.compat.enable_v1()
```

迁移点说明

如果您是Keras训练脚本，由于Keras迁移到Ascend平台运行时，某些功能受限支持，例如不支持动态学习率等，因此不建议用户在Ascend平台上迁移Keras开发的网络脚本。如需在Ascend平台运行Keras脚本，您需要对脚本进行如下修改：

创建一个TensorFlow会话并且注册Keras，并增加相关配置项以便在昇腾AI处理器执行训练。同时在训练结束时，关闭会话。

```
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import backend as K
from npu_device.compat.v1.npu_init import *

sess_config = tf.compat.v1.ConfigProto()
custom_op = sess_config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
sess_config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
sess_config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF
sess = tf.compat.v1.Session(config=sess_config)
K.set_session(sess)

#数据预处理...
```



```
#模型搭建...
#模型编译...
#模型训练...

sess.close()
```

6.9.5 Horovod 脚本迁移

Horovod是基于TensorFlow、Keras、PyTorch以及MXNet的分布式训练框架，目的是提升分布式训练的性能。不同于传统的TensorFlow分布式训练采用PS worker架构，Horovod使用Allreduce作为来聚合梯度，能够更好地利用带宽，解决PS worker的瓶颈问题。本节介绍如何迁移基于Horovod开发的分布式训练脚本，用于在昇腾AI处理器进行分布式训练。

Horovod原始代码：

```
import tensorflow as tf
import horovod.tensorflow as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used to process local rank (one GPU per process)
config = tf.compat.v1.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())

# Build model...
loss = ...
opt = tf.compat.v1.train.AdagradOptimizer(0.01 * hvd.size())

# Add Horovod Distributed Optimizer
opt = hvd.DistributedOptimizer(opt)

# Add hook to broadcast variables from rank 0 to all other processes during
# initialization.
hooks = [hvd.BroadcastGlobalVariablesHook(0)]

# Make training operation
train_op = opt.minimize(loss)

# Save checkpoints only on worker 0 to prevent other workers from corrupting them.
checkpoint_dir = '/tmp/train_logs' if hvd.rank() == 0 else None

# The MonitoredTrainingSession takes care of session initialization,
# restoring from a checkpoint, saving to a checkpoint, and closing when done
# or an error occurs.
with tf.compat.v1.train.MonitoredTrainingSession(checkpoint_dir=checkpoint_dir,
                                                 config=config,
                                                 hooks=hooks) as mon_sess:
  while not mon_sess.should_stop():
    # Perform synchronous training.
    mon_sess.run(train_op)
```

迁移后的代码：

```
import tensorflow as tf

# 导入NPU库
import npu_device as npu
from npu_device.compat.v1.npu_init import *

# 本示例调用init_resource接口，另起session进行初始化
(npu_sess, npu_shutdown) = init_resource()
npu.compat.enable_v1()

# Pin GPU to be used to process local rank (one GPU per process)
config = tf.compat.v1.ConfigProto()
config.gpu_options.visible_device_list = str(get_npu_local_rank_id()) # "hvd.local_rank"修改为
```



```
"get_npu_local_rank_id"

# Build model...
loss = ...
opt = tf.compat.v1.train.AdagradOptimizer(0.01 * get_npu_rank_size()) # "hv.size"修改为"get_npu_rank_size"

# NPU allreduce
opt = npu_distributed_optimizer_wrapper(opt) # "hvd.DistributeOptimizer"修改为
"npu_distributed_optimizer_wrapper"

# Add hook to broadcast variables from rank 0 to all other processes during
# initialization.
hooks = [NPUBroadcastGlobalVariablesHook(0, int(os.getenv('RANK_ID', '0')))] #
"hvd.BroadcastGlobalVariablesHook(0)"修改为"NPUBroadcastGlobalVariablesHook(0,
int(os.getenv('RANK_ID', '0')))"

# Make training operation
train_op = opt.minimize(loss)

# Save checkpoints only on worker 0 to prevent other workers from corrupting them.
checkpoint_dir = '/tmp/train_logs' if get_npu_rank_id() == 0 else None # "hvd.rank"修改为"get_npu_rank_id"

# The MonitoredTrainingSession takes care of session initialization,
# restoring from a checkpoint, saving to a checkpoint, and closing when done
# or an error occurs.
with tf.compat.v1.train.MonitoredTrainingSession(checkpoint_dir=checkpoint_dir,
                                         config=config,
                                         hooks=hooks) as mon_sess:
  while not mon_sess.should_stop():
    # Perform synchronous training.
    mon_sess.run(train_op)

# 训练结束后执行shutdown_resource, 同时关闭session
shutdown_resource(npu_sess, npu_shutdown)
close_session(npu_sess)
```


7 模型训练

7.1 执行单 Device 训练

概述

本节介绍如何基于迁移好的TensorFlow训练脚本，在单Device上执行训练。如果没有进行模型迁移，您也可以从**https://gitee.com/ascend/modelzoo**获取已经迁移适配好的训练脚本，直接体验训练过程。

须知

一个Device对应执行一个训练进程，当前不支持多进程在同一个Device上进行训练。

前提条件

- 已根据**环境准备**章节准备好包含1个可用的昇腾AI处理器的基础软硬件环境，或者包含TensorFlow相关模块的Ascend基础镜像。
- 已准备迁移好的TensorFlow训练脚本和对应数据集。
- **如果训练脚本中使用了HCCL集合通信接口**，执行训练前需要配置Device资源信息。可以通过配置文件（ranktable文件）的方式或者环境变量的方式进行配置，**由于是单Device训练，所以仅配置当前一个Device资源即可**，然后启动训练进程，本章节不对此场景的执行步骤展开介绍，详细可参考**7.2 执行分布式训练**。

  需要注意，此种场景下，若通过ranktable文件方式配置资源信息，分布式环境变量“RANK_ID”固定配置为“0”，“RANK_SIZE”固定配置为“1”即可。

在单 Device 上执行训练

步骤1 配置启动训练进程依赖的环境变量。

除此之外，还需进行如下配置：


```
# 请依据实际在下列场景中选择一个进行训练依赖包安装路径的环境变量设置。具体如下（以HwHiAiUser安装用户为例）：
# 场景一：昇腾设备安装部署开发套件包Ascend-cann-toolkit(此时开发环境可进行训练任务)。
. /home/HwHiAiUser/Ascend/ascend-toolkit/set_env.sh
# 场景二：昇腾设备安装部署软件包Ascend-cann-nnae。
. /home/HwHiAiUser/Ascend/nnae/set_env.sh

# tfplugin包依赖。
. /home/HwHiAiUser/Ascend/tfplugin/set_env.sh

# 若运行环境中存在多个python3版本时，需要在环境变量中配置python的安装路径。如下配置以安装python3.7.5为例，可根据实际修改。
export PATH=/usr/local/python3.7.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH

# 添加当前脚本所在路径到PYTHONPATH，例如：
export PYTHONPATH="$PYTHONPATH:/root/models"

export JOB_ID=10086       # 训练任务ID，用户自定义，仅支持大小写字母，数字，中划线，下划线。不建议使用以0开始的纯数字
export ASCEND_DEVICE_ID=0  # 指定昇腾AI处理器的逻辑ID，单卡训练也可不配置，默认为0，在0卡执行训练
```

说明

若训练所在系统环境需要升级gcc（例如CentOS、Debian和BClinux系统），则“LD_LIBRARY_PATH”配置项处动态库查找路径需要添加“${install_path}/lib64”，其中“{install_path}”为gcc升级安装路径。请参见**12.2 安装7.3.0版本gcc**。

步骤2 （可选）为了后续方便定位问题，拉起训练脚本前用户也可以通过环境变量使能Dump计算图。

```
export DUMP_GE_GRAPH = 2                  # 1：全量dump；2：不含有权重等数据的基本版dump；3：只显示节点关系的精简版dump
export DUMP_GRAPH_PATH = /home/dumpgraph  # 默认dump图生成在脚本执行目录，可以通过该环境变量指定dump路径
```

训练任务启动后，会在DUMP_GRAPH_PATH指定的路径下生成若干dump图文件，包括”.pbtxt”和“.txt”dump文件。由于dump的数据文件较多且文件都较大，若非问题定位需要，可以不生成dump图。

步骤3 执行训练脚本拉起训练进程，例如：

python3 /home/xxx.py

----结束

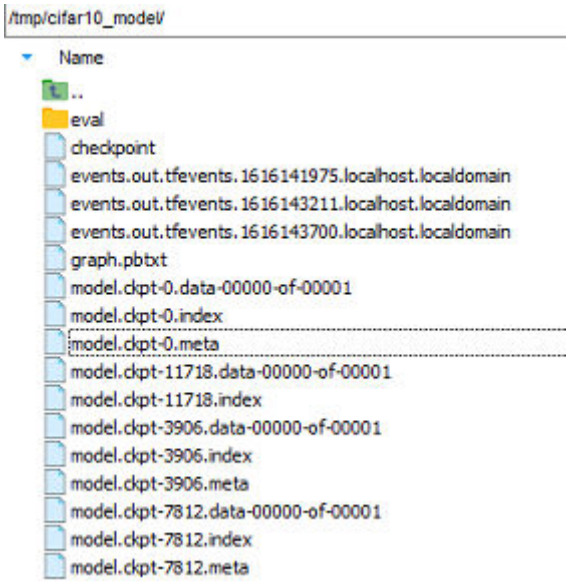
检查执行结果

步骤1 检查训练过程是否正常，Loss是否收敛。

```
I0319 16:55:07.290743 281473582915600 npu_hook.py:114] global_step...11703
INFO:tensorflow:global_step...11704
I0319 16:55:07.311460 281473582915600 npu_hook.py:114] global_step...11704
INFO:tensorflow:global_step...11705
I0319 16:55:07.331395 281473582915600 npu_hook.py:114] global_step...11705
INFO:tensorflow:global_step...11706
I0319 16:55:07.350636 281473582915600 npu_hook.py:114] global_step...11706
INFO:tensorflow:global_step...11707
I0319 16:55:07.370040 281473582915600 npu_hook.py:114] global_step...11707
INFO:tensorflow:global_step...11708
I0319 16:55:07.389427 281473582915600 npu_hook.py:114] global_step...11708
INFO:tensorflow:global_step...11709
I0319 16:55:07.408785 281473582915600 npu_hook.py:114] global_step...11709
INFO:tensorflow:global_step...11710
I0319 16:55:07.428487 281473582915600 npu_hook.py:114] global_step...11710
INFO:tensorflow:global_step...11711
I0319 16:55:07.447768 281473582915600 npu_hook.py:114] global_step...11711
INFO:tensorflow:global_step...11712
I0319 16:55:07.468086 281473582915600 npu_hook.py:114] global_step...11712
INFO:tensorflow:global_step/sec: 50.0385
I0319 16:55:07.490257 281473582915600 basic_session_run_hooks.py:692] global_step/sec: 50.0385
INFO:tensorflow:cross_entropy = 0.35196596, learning_rate = 0.1, train_accuracy = 0.8673828 (1.999 sec)
I0319 16:55:07.490747 281473582915600 basic_session_run_hooks.py:260] cross_entropy = 0.35196596, learning_rate = 0.1, train_accuracy = 0.8673828 (1.999 sec)
INFO:tensorflow:loss = 0.6402068, step = 11713 (1.999 sec)
I0319 16:55:07.490959 281473582915600 basic_session_run_hooks.py:260] loss = 0.6402068, step = 11713 (1.999 sec)
INFO:tensorflow:global_step...11713
```

步骤2 训练结束后，一般会生成如下目录和文件。

- 生成model目录：存放checkpoint文件和模型文件，是否生成该目录和脚本实现有关，若训练脚本中使用saver = tf.train.Saver()和saver.save()保存了模型，则会生成类似如下文件。



- 在脚本执行目录下生成kernel_meta：用于存放算子的.o及.json文件，该文件可用于后续问题定位，默认目录下没有文件，可以修改训练脚本，将运行参数op_debug_level传入3，从而保留.o和.json文件。

----结束

问题定位

如果运行失败，通过日志分析并定位问题。

Host侧日志路径：$HOME/ascend/log/run/plog/plog-pid_*.log，$HOME为Host侧用户根目录。

Device侧日志路径：$HOME/ascend/log/run/device-id/device-pid_*.log。

日志格式：

```
[Level] ModuleName(PID,PName):DateTimeMS [FileName:LineNumber]LogContent
```

说明

了解更多介绍，请参考《**日志参考**》。

一般通过ERROR级别的日志，识别问题产生模块，根据具体日志内容判定问题产生原因。

图 7-1 错误日志样例

表 7-1 问题定位思路

ModuleName	出错流程	解决思路
系统类报错	环境与版本配套错误	系统类报错，优先排查版本配套与系统安装是否正确。
GE	GE图编译或校验问题	校验类报错，通常会给出明确的错误原因，此时需要针对性地修改网络脚本，以满足相关要求。
RUNTIME	环境异常导致初始化问题或图执行问题	对于初始化异常，优先排查当前运行环境配置是否正确，当前环境是否有他人抢占。

7.2 执行分布式训练

7.2.1 分布式训练简介

开发者跨多个进程执行分布式训练时，首先需要配置参与分布式训练的昇腾AI处理器的资源信息，然后再拉起训练进程。

当前有两种配置资源信息的方式，**开发者可以选择其中任一方式，但需要注意两种方式不能混合使用**。

- 通过配置文件的方式，此资源配置文件称为ranktable文件，并配合环境变量RANK_TABLE_FILE、RANK_ID等使用。

  此种方式下配置资源信息、拉起训练进程的详细说明可参见**7.2.2 训练执行（配置文件方式设置资源信息）**。
- 通过环境变量的方式。

  此种方式下配置资源信息、拉起训练进程的详细说明可参见**7.2.3 训练执行（环境变量方式设置资源信息）**。

7.2.2 训练执行（配置文件方式设置资源信息）

7.2.2.1 准备 ranktable 资源配置文件

进行训练之前，需要准备昇腾AI处理器资源配置文件（即Rank table文件），并上传到当前运行环境，该文件用于定义训练的昇腾AI处理器资源信息。

使用前须知

1. 针对昇腾910 AI处理器，如果使用1台训练服务器（Server），要求实际参与集合通信的昇腾AI处理器数目只能为1/2/4/8，且0-3卡和4-7卡各为一个组网。使用2张卡或4张卡训练时，不支持跨组网创建设备集群。
2. 针对昇腾910 AI处理器，Server集群场景下（即由集群管理主节点和一组训练服务器组成训练服务器集群），要求参与集合通信的昇腾AI处理器数目只能为1*n、2*n、4*n、8*n（其中n为参与训练的Server个数，上限为512）。且n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网。

3. 针对昇腾910B AI处理器：如果单Server中昇腾AI处理器的数量是8，Server集群场景（即由集群管理主节点和一组训练服务器组成训练服务器集群）下要求参与集合通信的昇腾AI处理器数目为（1~8）*n（其中n为参与训练的Server个数，上限为1024），其中，n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网，注意每个Sever中参与集合通信的昇腾AI处理器数量保持一致。

准备配置文件

ranktable文件为json格式，以2p场景为例，文件可以命名为rank_table_2p.json。

说明

用户也可以在此处配置全量的昇腾AI处理器资源信息，后续训练进程启动时仅使用其中的指定的几个昇腾AI处理器资源。

针对昇腾910 AI处理器，在ranktable文件中配置参与训练的昇腾AI处理器信息支持两种配置模板，全新场景推荐使用模板一，模板二用于兼容部分已有场景。

- 模板一（推荐使用）

```
{
"server_count":"1"，//参与训练的AI Server数目，此例中，只有一个AI Server
"server_list":
[
  {
      "device":[   // AI Server中的Device列表
                  {
                   "device_id":"0",   // 处理器HDC通道号
                   "device_ip":"192.168.1.8",    // 处理器真实网卡IP
                   "rank_id":"0"     // rank的标识，rankID从0开始
                   },
                   {
                    "device_id":"1",
                    "device_ip":"192.168.1.9",
                    "rank_id":"1"
                    }
              ],
       "server_id":"10.0.0.10"   //AI Server标识，以点分十进制表示IP字符串
   }
],
"status":"completed",  // ranktable可用标识，completed为可用
"version":"1.0"          // ranktable模板版本信息,当前必须为"1.0"
}
```

表 7-2 ranktable 文件说明

配置项	配置说明	可选/必选
server_count	本次参与训练的AI Server个数。	必选
status	Rank table可用标识。 ● completed：表示Rank table可用，可执行训练。 ● initializing：表示Rank table不可用，不可执行训练。	必选
version	Rank table模板版本信息。当前仅支持配置为1.0。	必选

配置项	配置说明	可选/必选
server_list	本次参与训练的AI Server列表。	必选
server_id	AI Server物理IP，以点分十进制表示的字符串，配置示例：10.0.0.10	必选
device_id	昇腾AI处理器的ID，即Device在AI Server上的序列号。 取值范围：[0，实际Device数量-1]	必选
device_ip	昇腾AI处理器集成网卡IP，全局唯一，IPv4格式，以点分十进制表示的字符串，配置示例：192.168.1.8。 可以在当前AI Server执行指令cat /etc/hccn.conf获取网卡IP，例如： address_0=xx.xx.xx.xx netmask_0=xx.xx.xx.xx netdetect_0=xx.xx.xx.xx address_1=xx.xx.xx.xx netmask_1=xx.xx.xx.xx netdetect_1=xx.xx.xx.xx ... 查询到的address_xx即为网卡IP，address后的序号为昇腾AI处理器物理ID，即device_id，后面的ip地址即为需要用户填入的该device对应的网卡IP。 **说明** ● 典型8网口集合通信场景下：ranktable文件中的devices字段会记录每个device可用的网口ip信息，如下所示为device_id为0，网口ip地址为192.168.100.101的devices信息： "devices": [{"device_id": "0", "device_ip": "192.168.100.101"}] ● 针对昇腾910 AI处理器，网口裁剪的场景下，通过ranktable校验得到各Server可用的1/2/4个网口信息，当devices中device_ip字段为空字符串""时，代表该device的网口不在集合通信中使用，如下所示为device_id为0网口不使用时的devices信息： "devices": [{"device_id": "0", "device_ip": ""}]	必选
rank_id	Rank唯一标识，从0开始配置，且全局唯一，取值范围：[0, 总Device数量-1]	必选

- 模板二（兼容部分已有场景）

```
{
"status":"completed",  // Rank table可用标识，completed为可用
"group_count":"1",     // group数量，建议为1
"group_list":          // group列表
 [
  {
   "group_name":"hccl_world_group",//group名称，建议hccl_world_group
   "instance_count":"2",       // instance实例个数，容器场景下可理解为容器个数
   "device_count":"2",         // group中的所有device数目
   "instance_list":[
      {
```



```
        "pod_name":"tf-bae41",     //instance实例名称，一般为容器名称
        "server_id":"10.0.0.10",   //server标识，以IP格式填入
        "devices":[                //instance实例的device列表
         {
           "device_id":"0",           // 昇腾AI处理器HDC通道号
           "device_ip":"192.168.1.8"  // 昇腾AI处理器真实网卡IP
         }
        ]
      },
      {
         "pod_name":"tf-tbdf1",
         "server_id":"10.0.0.10",
         "devices":[
            {
               "device_id":"1",
               "device_ip":"192.168.1.9"
            }
         ]
       }
     ]
  }
 ]
}
```

表 7-3 ranktable 文件说明

配置项	配置说明	可选/必选
status	Rank table可用标识。 ● completed：表示Rank table可用，可执行训练。 ● initializing：表示Rank table不可用，不可执行训练。	必选
group_count	用户申请的Group数量，建议配置为1。	必选
group_list	Group列表。	必选
group_name	Group名称，当group_count为1时，建议配置为hccl_world_group或为空。因为当前版本无论定义为任何值，都会创建名称为hccl_world_group的group。 如果通过该配置文件创建了多个group，则系统会自动将多个group合并为一个名称为“hccl_world_group”的group资源。	可选
instance_count	和instance_list中pod_name个数保持一致，例如：容器场景下为容器实际数量。	必选
device_count	group中设备数量。	必选
instance_list	-	-
pod_name	用户自定义配置，保持instance_list内全局唯一。	必选
server_id	Server物理IP，以点分十进制表示的字符串，配置示例：10.0.0.10	必选

配置项	配置说明	可选/必选
devices	-	-
device_id	昇腾AI处理器物理ID，即Device在Server上的序列号。 取值范围：[0，实际Device数量-1]	必选

device_ip	昇腾AI处理器集成网卡IP，全局唯一，以点分十进制表示的字符串，配置示例：192.168.1.8。
	可以在当前Server执行指令cat /etc/hccn.conf获取网卡IP。
	说明
	● 典型8网口集合通信场景下：ranktable文件中的devices字段会记录每个device可用的网口ip信息，如下所示为device_id为0，网口ip地址为192.168.100.101的devices信息：
	`"devices": [{"device_id": "0", "device_ip": "192.168.100.101"}]`
	● 针对昇腾910 AI处理器，网口裁剪的场景下，通过ranktable校验得到各Server可用的1/2/4个网口信息，当devices中device_ip字段为空字符串""时，代表该device的网口不在集合通信中使用，如下所示为device_id为0网口不使用时的devices信息：
	`"devices": [{"device_id": "0", "device_ip": ""}]`
		必选

针对昇腾910B AI处理器，ranktable文件配置说明如下：

```
{
"server_count":"1"，//参与训练的AI Server数目，此例中，只有一个AI Server
"server_list":
[
  {
      "device":[   // AI Server中的Device列表
                  {
                   "device_id":"0"，  // 处理器HDC通道号
                   "device_ip":"192.168.1.8"，   // 处理器真实网卡IP
                   "rank_id":"0"     // rank的标识，rankID从0开始
                   },
                   {
                    "device_id":"1",
                    "device_ip":"192.168.1.9",
                    "rank_id":"1"
                    }
              ],
       "server_id":"10.0.0.10"   //AI Server标识，以点分十进制表示IP字符串
   }
],
"status":"completed"，  // ranktable可用标识，completed为可用
"version":"1.0"          // ranktable模板版本信息,当前必须为"1.0"
}
```


表 7-4 ranktable 文件说明

配置项	配置说明	可选/必选
server_count	本次参与训练的AI Server个数。	必选
status	Rank table可用标识。 ● completed：表示Rank table可用，可执行训练。 ● initializing：表示Rank table不可用，不可执行训练。	必选
version	Rank table模板版本信息。当前仅支持配置为1.0。	必选
server_list	本次参与训练的AI Server列表。	必选
server_id	AI Server物理IP，以点分十进制表示的字符串，配置示例：10.0.0.10	必选
device_id	昇腾AI处理器的ID，即Device在AI Server上的序列号。 取值范围：[0，实际Device数量-1]	必选
device_ip	昇腾AI处理器集成网卡IP，全局唯一，以点分十进制表示的字符串，配置示例：192.168.1.8。 可以在当前AI Server执行指令cat /etc/hccn.conf获取网卡IP，例如： address_0=xx.xx.xx.xx netmask_0=xx.xx.xx.xx netdetect_0=xx.xx.xx.xx address_1=xx.xx.xx.xx netmask_1=xx.xx.xx.xx netdetect_1=xx.xx.xx.xx ... 查询到的address_xx即为网卡IP，address后的序号为昇腾AI处理器物理ID，即device_id，后面的ip地址即为需要用户填入的该device对应的网卡IP。 **说明** 典型8网口集合通信场景下：ranktable文件中的devices字段会记录每个device可用的网口ip信息，如下所示为device_id为0，网口ip地址为192.168.100.101的devices信息： "devices": [{"device_id": "0", "device_ip": "192.168.100.101"}]	必选
rank_id	Rank唯一标识，从0开始配置，且全局唯一，取值范围：[0, 总Device数量-1]	必选

配置示例

以包含两个Device的资源配置文件为例，假设命名为“rank_table_2p.json”，配置如下：


```
{
"server_count":"1",
"server_list":
[
  {
      "device":[
                  {
                   "device_id":"0",
                   "device_ip":"192.168.1.8",
                   "rank_id":"0"
                   },
                   {
                    "device_id":"1",
                    "device_ip":"192.168.1.9",   // 两个Device需要处于同一网段，0卡和1卡为同一网段
                    "rank_id":"1"
                    }
              ],
       "server_id":"10.0.0.10"
   }
],
"status":"completed",
"version":"1.0"
}
```

7.2.2.2 执行训练

概述

本节介绍如何基于迁移好的TensorFlow训练脚本，执行分布式训练。

使用前须知：

1. 一个Device对应执行一个训练进程，当前不支持多进程在同一个Device上进行训练。
2. 针对昇腾910 AI处理器，如果使用1台训练服务器（Server），要求实际参与集合通信的昇腾AI处理器数目只能为1/2/4/8，且0-3卡和4-7卡各为一个组网。使用2张卡或4张卡训练时，不支持跨组网创建设备集群。
3. 针对昇腾910 AI处理器，Server集群场景下（即由集群管理主节点和一组训练服务器组成训练服务器集群），要求参与集合通信的昇腾AI处理器数目只能为1*n、2*n、4*n、8*n（其中n为参与训练的Server个数，上限为512）。且n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网。
4. 针对昇腾910B AI处理器：如果单Server中昇腾AI处理器的数量是8，Server集群场景（即由集群管理主节点和一组训练服务器组成训练服务器集群）下要求参与集合通信的昇腾AI处理器数目为（1~8）*n（其中n为参与训练的Server个数，上限为1024），其中，n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网，注意每个Sever中参与集合通信的昇腾AI处理器数量保持一致。
5. 分布式训练场景下，HCCL会使用Host服务器的部分端口进行集群信息收集，需要操作系统预留该部分端口。默认情况下，HCCL使用60000-60015端口，若通过环境变量**HCCL_IF_BASE_PORT**指定了Host网卡起始端口，则需要预留以该端口起始的16个端口。

   操作系统端口号预留示例：sysctl -w net.ipv4.ip_local_reserved_ports=60000-60015

前提条件

- 已准备好迁移好的TensorFlow训练脚本和对应数据集。

- 已准备好昇腾AI处理器资源信息配置文件，请参考**7.2.2.1 准备ranktable资源配置文件**。

在多 Device 上执行训练

在多个Device上进行分布式训练时，需要依次拉起所有训练进程（用户可以在不同的shell窗口依次拉起不同的训练进程），下面以单机两个Device的训练场景举例介绍如何拉起各训练进程。

步骤1 拉起训练进程0。

安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行**. ${install_path}/set_env.sh**命令设置环境变量。并进行如下配置：

```
# 请依据实际在下列场景中选择一个进行训练依赖包安装路径的环境变量设置。具体如下（以HwHiAiUser安装用户为例）：
# 场景一：昇腾设备安装部署开发套件包Ascend-cann-toolkit(此时开发环境可进行训练任务)。
. /home/HwHiAiUser/Ascend/ascend-toolkit/set_env.sh
# 场景二：昇腾设备安装部署软件包Ascend-cann-nnae。
. /home/HwHiAiUser/Ascend/nnae/set_env.sh

# tfplugin包依赖。
. /home/HwHiAiUser/Ascend/tfplugin/set_env.sh

# 若运行环境中存在多个python3版本时，需要在环境变量中配置python的安装路径。如下配置以安装python3.7.5为例，可根据实际修改。
export PATH=/usr/local/python3.7.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH

# 当前脚本所在路径，例如：
export PYTHONPATH=/home/test:$PYTHONPATH
export JOB_ID=10086
export ASCEND_DEVICE_ID=0
export RANK_ID=0
export RANK_SIZE=2
export RANK_TABLE_FILE=/home/test/rank_table_2p.json
python3 /home/xxx.py
```

拉起训练进程前，需要对集群参数进行配置，其中集群参数相关的环境变量说明如下：

- RANK_SIZE：参与分布式训练的Device数量。
- RANK_ID：当前Device在集群中的唯一索引，需要与资源配置文件中的索引一致。
- RANK_TABLE_FILE：资源配置文件路径。

其他环境变量说明可参见**11.3 环境变量参考**。

步骤2 拉起训练进程1。

安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行**. ${install_path}/set_env.sh**命令设置环境变量。并进行如下配置：

```
export PYTHONPATH=/home/test:$PYTHONPATH
export JOB_ID=10086
export ASCEND_DEVICE_ID=1
export RANK_ID=1
export RANK_SIZE=2
export RANK_TABLE_FILE=/home/test/rank_table_2p.json
python3 /home/xxx.py
```


说明

- 除了以上方式，您还可以通过自定义启动脚本通过循环方式依次拉起多个训练进程，具体样例请参考**链接**。
- 若训练所在系统环境需要升级gcc（例如CentOS、Debian和BClinux系统），则此处动态库查找路径需要添加“${install_path}/lib64”，其中“{install_path}”为gcc升级安装路径。请参见**12.2 安装7.3.0版本gcc**。

----结束

7.2.3 训练执行（环境变量方式设置资源信息）

7.2.3.1 通过环境变量配置资源信息

进行训练之前，需要配置参与集群训练的昇腾AI处理器的资源信息。开发者可以通过本节所述的环境变量组合的方式配置资源信息，完成集合通信组件的初始化。

使用前须知

1. 针对昇腾910 AI处理器，如果使用1台训练服务器（Server），要求实际参与集合通信的昇腾AI处理器数目只能为1/2/4/8，且0-3卡和4-7卡各为一个组网。使用2张卡或4张卡训练时，不支持跨组网创建设备集群。
2. 针对昇腾910 AI处理器，Server集群场景下（即由集群管理主节点和一组训练服务器组成训练服务器集群），要求参与集合通信的昇腾AI处理器数目只能为1*n、2*n、4*n、8*n（其中n为参与训练的Server个数，上限为512）。且n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网。
3. 针对昇腾910B AI处理器：如果单Server中昇腾AI处理器的数量是8，Server集群场景（即由集群管理主节点和一组训练服务器组成训练服务器集群）下要求参与集合通信的昇腾AI处理器数目为（1~8）*n（其中n为参与训练的Server个数，上限为1024），其中，n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网，注意每个Sever中参与集合通信的昇腾AI处理器数量保持一致。

配置说明

需要在执行训练的每个Server节点上分别配置如下环境变量，进行资源信息的配置，示例如下：

```
export CM_CHIEF_IP = 192.168.1.1
export CM_CHIEF_PORT = 6000
export CM_CHIEF_DEVICE = 0
export CM_WORKER_SIZE = 8
export CM_WORKER_IP = 192.168.0.1
```

- CM_CHIEF_IP、CM_CHIEF_PORT、CM_CHIEF_DEVICE 用于配置Master节点的Host监听IP、监听端口与主Device ID。
  - 监听IP可以通过ifconfig命令进行查询，要求为常规IPv4或IPv6格式。
  - 指定的监听端口号需要确保在训练进程拉起时，无其他业务占用。
- CM_WORKER_SIZE：用于配置参与集群训练的Device数量。
- CM_WORKER_IP：用于配置当前节点与Master进行通信时所用的网卡IP，可通过ifconfig命令进行查询，要求为常规IPv4或IPv6格式。

  需要确保指定的网卡IP能够与Master节点正常通信。

配置示例

假设执行分布式训练的Server节点数量为2，Device数量为16为例，每个Server节点有8个Device。拉起训练进程前，在对应的shell窗口中配置如下环境变量，进行资源信息的配置。

- 节点0，此节点为Master节点。
```
export CM_CHIEF_IP = 192.168.1.1
export CM_CHIEF_PORT = 6000
export CM_CHIEF_DEVICE = 0
export CM_WORKER_SIZE = 16
export CM_WORKER_IP = 192.168.1.1
```
- 节点1
```
export CM_CHIEF_IP = 192.168.1.1
export CM_CHIEF_PORT = 6000
export CM_CHIEF_DEVICE = 0
export CM_WORKER_SIZE = 16
export CM_WORKER_IP = 192.168.2.1
```

7.2.3.2 执行训练

概述

本节介绍如何基于迁移好的TensorFlow训练脚本，执行分布式训练。

使用前须知：

1. 一个Device对应执行一个训练进程，当前不支持多进程在同一个Device上进行训练。
2. 针对昇腾910 AI处理器，如果使用1台训练服务器（Server），要求实际参与集合通信的昇腾AI处理器数目只能为1/2/4/8，且0-3卡和4-7卡各为一个组网。使用2张卡或4张卡训练时，不支持跨组网创建设备集群。
3. 针对昇腾910 AI处理器，Server集群场景下（即由集群管理主节点和一组训练服务器组成训练服务器集群），要求参与集合通信的昇腾AI处理器数目只能为1*n、2*n、4*n、8*n（其中n为参与训练的Server个数，上限为512）。且n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网。
4. 针对昇腾910B AI处理器：如果单Server中昇腾AI处理器的数量是8，Server集群场景（即由集群管理主节点和一组训练服务器组成训练服务器集群）下要求参与集合通信的昇腾AI处理器数目为（1~8）*n（其中n为参与训练的Server个数，上限为1024），其中，n为2的指数倍情况下，集群性能最好，建议用户优先采用此种方式进行集群组网，注意每个Sever中参与集合通信的昇腾AI处理器数量保持一致。
5. 分布式训练场景下，HCCL会使用Host服务器的部分端口进行集群信息收集，需要操作系统预留该部分端口。默认情况下，HCCL使用60000-60015端口，若通过环境变量**HCCL_IF_BASE_PORT**指定了Host网卡起始端口，则需要预留以该端口起始的16个端口。

   操作系统端口号预留示例：sysctl -w net.ipv4.ip_local_reserved_ports=60000-60015

前提条件

- 已准备好迁移好的TensorFlow训练脚本和对应数据集。
- 已在每个参与训练的Device上完成资源信息环境变量的配置，请参考**7.2.3.1 通过环境变量配置资源信息**。

在多 Device 上执行训练

在多个Device上进行分布式训练时，需要依次拉起所有训练进程，下面以单机两个Device的训练场景举例介绍如何拉起各训练进程。用户可以在不同的shell窗口依次拉起不同的训练进程。

步骤1 拉起训练进程0：

1. 配置除资源信息环境变量以外的其他环境变量。

```
# 请依据实际在下列场景中选择一个进行训练依赖包安装路径的环境变量设置。具体如下（以HwHiAiUser安装用户为例）：
# 场景一：昇腾设备安装部署开发套件包Ascend-cann-toolkit(此时开发环境可进行训练任务)。
. /home/HwHiAiUser/Ascend/ascend-toolkit/set_env.sh
# 场景二：昇腾设备安装部署软件包Ascend-cann-nnae。
. /home/HwHiAiUser/Ascend/nnae/set_env.sh

# tfplugin包依赖。
. /home/HwHiAiUser/Ascend/tfplugin/set_env.sh

# 若运行环境中存在多个python3版本时，需要在环境变量中配置python的安装路径。如下配置以安装python3.7.5为例，可根据实际修改。
export PATH=/usr/local/python3.7.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH

# 当前脚本所在路径，例如：
export PYTHONPATH=/home/test:$PYTHONPATH
export JOB_ID=10086
export ASCEND_DEVICE_ID=0
```

2. 拉起训练脚本。

   python3 /home/xxx.py

步骤2 拉起训练进程1：

1. 配置除资源信息环境变量以外的其他环境变量。

```
# 请依据实际在下列场景中选择一个进行训练依赖包安装路径的环境变量设置。具体如下（以HwHiAiUser安装用户为例）：
# 场景一：昇腾设备安装部署开发套件包Ascend-cann-toolkit(此时开发环境可进行训练任务)。
. /home/HwHiAiUser/Ascend/ascend-toolkit/set_env.sh
# 场景二：昇腾设备安装部署软件包Ascend-cann-nnae。
. /home/HwHiAiUser/Ascend/nnae/set_env.sh

# tfplugin包依赖。
. /home/HwHiAiUser/Ascend/tfplugin/set_env.sh

# 若运行环境中存在多个python3版本时，需要在环境变量中配置python的安装路径。如下配置以安装python3.7.5为例，可根据实际修改。
export PATH=/usr/local/python3.7.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH

# 当前脚本所在路径，例如：
export PYTHONPATH=/home/test:$PYTHONPATH
export JOB_ID=10086
export ASCEND_DEVICE_ID=1
```

   说明

   若训练所在系统环境需要升级gcc（例如CentOS、Debian和BClinux系统），则此处动态库查找路径需要添加“${install_path}/lib64”，其中“{install_path}”为gcc升级安装路径。请参见12.2 安装7.3.0版本gcc。

2. 拉起训练脚本。

   python3 /home/xxx.py

----结束

7.2.4 结果说明

分布式训练执行完成后，开发者可以参考本章节检查执行结果、问题定位。

检查执行结果

步骤1 检查运行结果。

不同的训练脚本打印结果不同，若执行分布式训练的每个Device出现类似如下打印信息，说明训练任务已经正常结束。

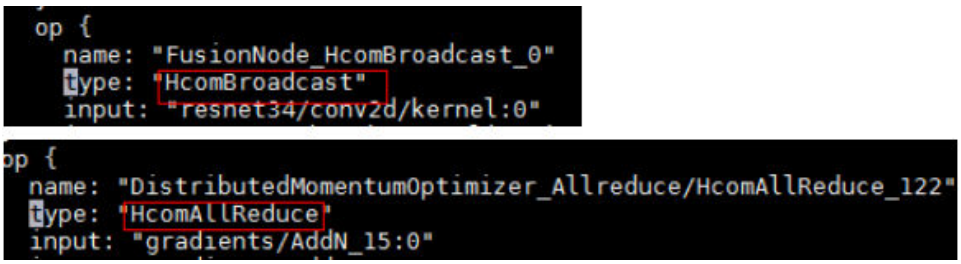
```
INFO:tensorflow:Starting cycle: 1/19
I1108 15:17:36.300130 281473774433312 resnet_run_loop.py:561] Starting cycle: 1/19
INFO:tensorflow:Calling model_fn.
I1108 15:17:36.569131 281473774433312 estimator.py:1148] Calling model_fn.
INFO:tensorflow:Done calling model_fn.
I1108 15:17:46.105373 281473774433312 estimator.py:1150] Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
I1108 15:17:46.453371 281473774433312 basic_session_run_hooks.py:541] Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
I1108 15:17:49.019978 281473774433312 monitored_session.py:240] Graph was finalized.
INFO:tensorflow:Restoring parameters from /home/[redacted]/models/official/resnet/tmp/cifar10_model/model.ckpt-62496
```

当开启环境变量DUMP_GE_GRAPH时，会生成GE的dump图文件。

```
export DUMP_GE_GRAPH=2
```

在dump下来的图文件目录下，搜索到包含HcomBroadcast和HcomAllReduce算子，代表正常插入了NPU间通信的HCCL算子。

图 7-2 GE 的 dump 图

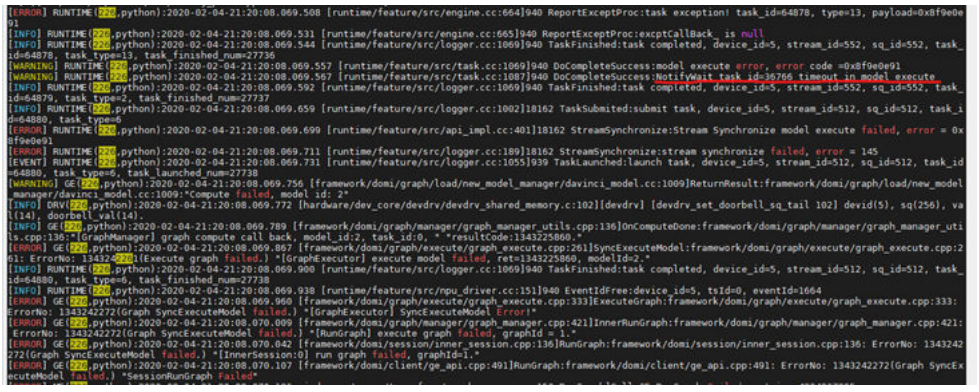


步骤2 如果运行失败，和单Device训练一样，通过日志分析并定位问题。

在$HOME/ascend/log/run/plog下查看Host侧日志plog_*.log，$HOME为Host侧用户根目录。

在单Device执行成功，多单Device执行失败的情况下，一般为集合通信的问题，如图7-3所示。可以参考7.2.5 常见案例进行问题处理。

图 7-3 集合通信问题



----结束

问题定位

如果运行失败，通过日志分析并定位问题。

Host侧日志路径：$HOME/ascend/log/run/plog/plog-pid_*.log，$HOME为Host侧用户根目录。

Device侧日志路径：$HOME/ascend/log/run/device-id/device-pid_*.log。

日志格式：

```
[Level] ModuleName(PID,PName):DateTimeMS LogContent
```

说明

了解更多介绍，请参考《**日志参考**》。

一般通过ERROR级别的日志，识别问题产生模块，具体日志内容判定问题产生原因。

图 7-4 错误日志样例

表 7-5 问题定位思路

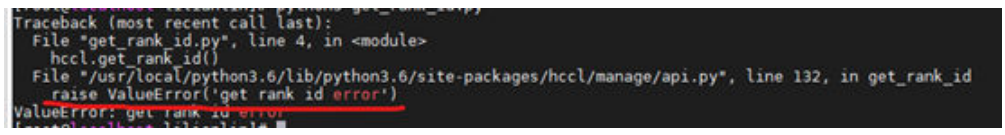
ModuleName	出错流程	解决思路
系统类报错	环境与版本配套错误	系统类报错，优先排查版本配套与系统安装是否正确。
GE	GE图编译或校验问题	校验类报错，通常会给出明确的错误原因，此时需要针对性地修改网络脚本，以满足相关要求。
RUNTIME	环境异常导致初始化问题或图执行问题	对于初始化异常，优先排查当前运行环境配置是否正确，当前环境是否有他人抢占。

7.2.5 常见案例

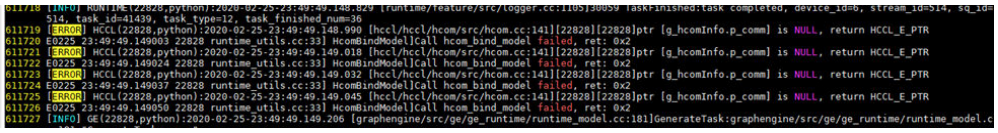
7.2.5.1 管理类接口调用失败

问题现象

屏显信息出现“get rank id error”的错误，如下所示：



查看Host日志，出现“Call hcom_bind_model failed”的错误信息，如下所示：



可能原因

集合通信的管理类python接口需要在完成集合通信初始化之后调用，才能正常执行。

解决方法

训练脚本中，在完成集合通信初始化后再调用集合通信的管理类python接口。

8 精度调优

8.1 精度调优使用场景

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，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

本文档为精度问题定位提供了分析流程指导。

8.2 精度调优流程

精度问题来源可能来自于各个方面：

1. 提供的参考基准存在问题
2. 进行模型迁移时存在问题
3. 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。

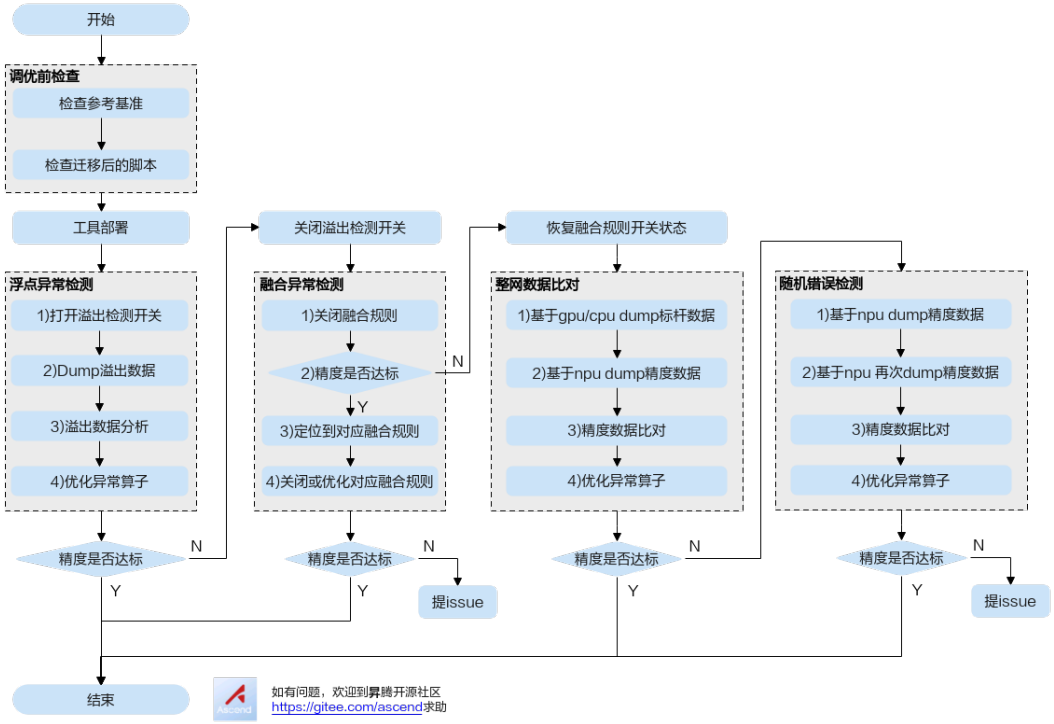


表 8-1 精度调优流程说明

序号	操作	说明
1	**8.3 调优前检查**	训练网络精度调优前，一般进行如下两方面的检查： ● 检查迁移前的脚本，排除参考基准的问题。 ● 检查迁移后的脚本，排除模型迁移时存在的问题。
2	**8.4 工具部署**	训练网络精度调优前，需要在NPU训练环境部署一键式精度分析工具。
3	**8.5 浮点异常检测**	训练网络执行过程中，可能发生频繁的浮点异常情况，即loss scale值下降次数较多或者直接下降为1，此时需要通过分析溢出数据，对频繁的浮点异常问题进行定界定位。
4	**8.6 融合异常检测**	训练网络执行过程中，系统会根据内置的融合规则对网络中算子进行融合，以达到提高网络性能的效果。由于大多数融合是自动识别的，可能存在未考虑到的场景，导致精度问题，因此可以尝试关闭融合规则，定界网络问题是否是由于融合导致。

序号	操作	说明
5	8.7 整网数据比对	排除以上问题后，在训练网络精度仍未达预期时，通过采集训练过程中各算子的运算结果（即Dump数据），然后和业界标准算子（如TensorFlow）运算结果进行数据偏差对比，快速定位到具体算子的精度问题。
6	8.8 随机错误检测	网络执行过程中，可能存在部分计算过程在相同输入的情况下给出了不同的输出的问题。当出现以上随机问题时，可以通过执行两次训练，并分别采集各算子的运算结果（即dump数据），通过比对分析，从而快速定位到导致随机问题的算子层

8.3 调优前检查

8.3.1 检查项汇总

精度调优前，请按照如下表格的要求进行检查，并记录相关检查结果，排除参考基准和模型迁移过程中可能存在的问题。

表 8-2 精度调优前检查项

序号	检查项	简介	检查结果
检查参考基准脚本			
1	8.3.2.1 参考基准：多次训练验证精度一致	基准模型多次训练需要得到相同的结果预期，如果基准模型的多次训练相互对比不满足精度对比要求，那么该参考基准不适合用于精度对比。	通过/不通过/未进行
2	8.3.2.2 参考基准：正确使能混合精度训练	由于昇腾AI处理器（简称NPU）硬件架构仅支持用户模型执行在混合精度训练模式下，故用户模型也需要以混合精度训练。如果用户模型未使能混合精度训练模式、或使能不当，会有可能导致NPU无法训练或训练精度不符合预期的结果。	通过/不通过/未进行
检查迁移后脚本			
3	8.3.3.1 模型迁移：在NPU上正确使能混合精度	在精度调优前，首先需要确保模型迁移成功且在NPU上功能调通。如果涉及分布式训练，确保正确使能。特别需要注意的是，在迁移过程中，需要确保正确使能混合精度训练。	通过/不通过/未进行

序号	检查项	简介	检查结果
4	8.3.3.2 模型迁移：在NPU上正确使能Loss Scale	迁移后的脚本需在NPU上正确使能Loss Scale。且由于NPU计算特性与GPU混合精度计算特性存在差异，LossScaleManager参数也往往需要进行适当的调整以保证精度。	通过/不通过/未进行
6	8.3.3.3 数据处理：数据集与基准一致	训练数据集往往较大，且可能出现不完整情况，需要确保数据集完整性。	通过/不通过/未进行
7	8.3.3.4 数据处理：预处理流程与基准一致	用户模型代码的数据预处理流程可能存在基于资源自动设定的变量，该变量会导致对数据集打乱的随机性不一致。需要从代码级别确认数据预处理的接口调用，尽可能减小该差异。	通过/不通过/未进行
8	8.3.3.5 数据处理：多节点分片方式与基准一致	用户模型代码的多节点数据预处理流程可能存在基于文件名、文件个数等进行分片的模式。该模式下由于文件读取接口在不同节点上对文件名排序的不同，会导致分片差异较大甚至文件重复分片到不同节点等不期望的结果。需要增加调试代码排除类似问题，确认分片规则与基准一致。	通过/不通过/未进行
9	8.3.3.6 训练流程：与基准一致	用户训练过程中，常常会出现例如未清空中间数据等流程错误，然后基于该不期望的中间数据可能会引起精度不一致的问题。用户需要对训练流程有基本的知识积累，并自行检查整个训练、验证过程的正确性。	通过/不通过/未进行
10	8.3.3.7 模型超参：与基准一致	进行脚本迁移的用户可能对超参的一致性不熟悉，导致迁移完成后，但一些超参的计算事实上与参考基准不一致。此时需要检查实际执行超参与参考基准的一致性。	通过/不通过/未进行

8.3.2 检查参考基准脚本

8.3.2.1 参考基准：多次训练验证精度一致

目标

基准模型多次训练需要得到相同的结果预期。

说明

以下章节中如未特别注明，多次训练均需满足“未固定随机性”的前提。

思路

如果基准模型的多次训练相互对比不满足精度对比要求，那么该参考基准不适合用于精度对比，相应分析思路为：

- 模型算法稳定，不引起多次训练较大结果偏差。
- 数据集质量较高，不引起多次训练较大结果偏差。
- 超参稳定，不引起多次训练较大结果偏差。

参考步骤

对于成熟模型（即hyperparameter borrowing）：

1. 检查执行时使用的超参，确保与给定基准一致。
2. 对于集群训练，检查集群训练模式，确保与给定基准一致。
3. 检查数据集文件，确保与给定基准一致。
4. 检查模型代码与执行参数，确保构造的计算逻辑与给定基准一致。
5. 进一步检查计算图，确保计算流程与算子维度与给定基准一致。
6. 重新训练模型并验证最终模型准确度。如果仍旧未达到基准，重复以上检查直到与基准一致。
7. 重复三次以上的训练，确保每次训练的验证准确度均能够与基准一致。如果有不一致现象，重复以上检查直到满足各步骤需求。

对于成熟模型自定义超参：

1. 如果数据集文件为自定义样本，需要确保标记信息没有错误，且格式符合模型的设计预期。
2. 如果无法确认数据集文件的准确性，请直接使用成熟模型提供的数据集（可进行适当裁剪）。
3. 以成熟模型的原始超参为基准，基于实际训练数据集与集群规模进行修改，得到一组待测超参。
4. 重复三次以上的训练，确保每次训练的验证准确度均一致。如果有不一致的现象，重复以上检查直到满足各步骤需求。

对于自定义模型：

- 确保数据集标记没有错误。
- 选定一组调试时稳定的待测超参。
- 重复三次以上的训练，确保每次训练的验证准确度均一致。如果有不一致的现象，调整超参、模型结构并重复以上检查，直到满足各步骤需求。

8.3.2.2 参考基准：正确使能混合精度训练

目标

基准模型为gpu-fp32/cpu-fp32训练时，需要调整为混合精度（gpu-fp16）训练并且能够得到相同的准确度结果。

基准模型为混合精度（gpu-fp16）训练时，建议使用高精度（gpu-fp32/cpu-fp32）模式训练并能够得到相同的准确度结果。

说明

如果由于内存、硬件等限制无法获得高精度模式训练的结果，那么至少需要确保多次重复训练能够得到相同的准确度结果。

思路

昇腾AI处理器（简称NPU）硬件架构仅支持用户模型执行在混合精度训练模式下，故用户模型也需要在GPU上以混合精度训练确保模型收敛后作为参考基准。如果用户模型未在GPU的混合精度模式下验证收敛，使用该参考基准迁移到NPU后可能会导致NPU训练不收敛。

如果混合精度训练和高精度训练相互对比不满足精度对比要求，那么该参考基准不适合用于精度对比，原因为：混合精度的fp16数据类型在该训练的场景下对精度引起了明显的差异，导致gpu混合精度以及npu混合精度训练均存在精度风险，此时需要调整模型结构以确保混合精度训练能够收敛。

参考步骤

步骤1 确保基准模型使能混合精度训练模式。

步骤2 确保使能动态loss_scale。

也可使能静态loss_scale，该方法较为复杂，不建议使用。原因是：静态loss_scale场景下，使用GPU的loss_scale值在NPU上训练时，可能会导致不期望的频繁溢出，影响收敛，此种场景下需要调整NPU的loss_scale值。

步骤3 观察混合精度训练过程中的浮点异常发生次数，确保发生次数较少（异常次数建议在0.5%以下，global batch size较大时建议在0.1%以下）。

步骤4 如果发生次数较多，修改loss_scale的初始值以及放大、缩小的步进阈值，直到训练发生的浮点异常次数非常少。

步骤5 重复三次以上的训练，确保每次训练的验证准确度均一致。如果有不一致的现象，调整超参、模型结构并重复以上检查，直到满足各步骤需求。

----结束

8.3.3 检查迁移后脚本

8.3.3.1 模型迁移：在 NPU 上正确使能混合精度

目标

在精度调优前，首先需要确保模型迁移成功且在NPU上功能调通。如果涉及分布式训练，确保正确使能。

特别需要注意的是，在迁移过程中，需要确保正确使能混合精度训练。

参考步骤

混合精度在NPU上使能可以使用三种主要的方法，以达到性能、精度的最优权衡：

1. 如果用户参考基准脚本使用了GPU的手动混合精度模式（**参考链接**），在模型中已经明确定义了所有的算子数据类型。那么迁移到NPU上时也保持同样的方式，并且请确保NPU的precision_mode需要设置当前版本默认值。

   针对昇腾910 AI处理器，默认配置项为“allow_fp32_to_fp16”。

   针对昇腾910B AI处理器，默认配置项为“must_keep_origin_dtype”。

2. 如果用户参考基准脚本使用了GPU的自动混合精度模式（**参考链接**），利用TensorFlow框架或者第三方接口(例如apex)能力，定义模型的算子数据类型。那么迁移到NPU上时也保持同样的方式，并且确保NPU的precision_mode需要设置当前版本默认值。

   针对昇腾910 AI处理器，默认配置项为“allow_fp32_to_fp16”。

   针对昇腾910B AI处理器，默认配置项为“must_keep_origin_dtype”。

3. 如果用户参考基准脚本使用fp32高精度模式定义模型，那么迁移到NPU上时，需要使能NPU框架的自动混合精度模式（即precision_mode为allow_mix_precision）。

   ```
   import npu_device as npu
   npu.global_options().precision_mode = 'allow_mix_precision'
   npu.open().as_default()
   ```

说明

为尽可能减少重复改图带来不期望的问题，以上方法需要确保仅使能一种。

8.3.3.2 模型迁移：在 NPU 上正确使能 Loss Scale

目标

在混合精度计算中使用float16数据格式数据动态范围降低，造成梯度计算出现浮点溢出，会导致部分参数更新失败。为了保证部分模型训练在混合精度训练过程中收敛，需要配置Loss Scale的方法。

Loss Scale方法通过在前向计算所得的loss乘以loss scale系数S，起到在反向梯度计算过程中达到放大梯度的作用，从而最大程度规避浮点计算中较小梯度值无法用FP16表达而出现的溢出问题。在参数梯度聚合之后以及优化器更新参数之前，将聚合后的参数梯度值除以loss scale系数S还原。

动态Loss Scale通过在训练过程中检查梯度中浮点计算异常状态，自动动态选取loss scale系数S以适应训练过程中梯度变化，从而解决人工选取loss scale系数S和训练过程中自适应调整的问题。

在具体实现中，昇腾AI处理器由于浮点计算特性不同，在计算过程中的浮点异常检查等部分与GPU存在差异，因此需要注意在NPU上正确使能Loss Scale。

参考步骤

6.5 替换LossScaleOptimizer

8.3.3.3 数据处理：数据集与基准一致

目标

数据集与参考数据集一致。

思路

数据集往往较大，复制容易出现不完整，通过校验码确定数据集完整性。

参考步骤

步骤1 获取参考训练与实际环境训练的数据集文件列表，确保两个文件列表一致。

步骤2 获取参考数据集文件与实际环境数据集文件的md5校验码，确保两组校验码一致。

----结束

8.3.3.4 数据处理：预处理流程与基准一致

目标

数据预处理流程与参考模型一致。

思路

用户模型代码的数据预处理流程可能存在基于资源自动设定的变量，该变量会导致对数据集打乱的随机性不一致。需要从代码级别确认数据预处理的接口调用，尽可能减小该差异。典型的例子为：

数据集shuffle时自动检测主机内存大小后设定buffer size，如果NPU主机与参考基准主机内存差异过大，那么数据集打乱的随机性也会产生明显差异，从而导致精度产生不期望的差异。

参考步骤

步骤1 检查文件读取接口，确保读取数量与参考基准一致。

步骤2 检查源数据格式到数据输入样本的转换过程，确保与基准一致。

步骤3 检查对输入样本进行补齐等操作时，确保补齐方式与基准一致。

步骤4 检查对数据输入样本进行乱序的方式一致，例如乱序片段的样本个数、乱序时的并行度均需一致。

----结束

8.3.3.5 数据处理：多节点分片方式与基准一致

目标

多节点对数据的分片方式需要与参考模型一致。

思路

用户模型代码的多节点数据预处理流程可能存在基于文件名、文件个数等进行分片的模式。

该模式下由于文件读取接口在不同节点上对文件名排序的不同，会导致分片差异较大甚至文件重复分片到不同节点等不期望的结果。

需要增加调试代码排除类似问题，确认分片规则与基准一致。

参考步骤

步骤1 分别在参考基准和迁移后的模型上增加对输入文件列表的打印。

步骤2 检查文件在节点间的分片策略与参考基准一致。

----结束

8.3.3.6 训练流程：与基准一致

目标

训练的起始状态、中间过程、结果状态需要与参考基准一致，验证的样本、流程也需要与参考基准一致。

思路

用户训练过程中，常常会出现例如未清空中间数据等流程错误，基于该不期望的中间数据可能会引起精度不一致的问题。

用户需要对训练流程有基本的知识积累，并自行检查整个训练、验证过程的正确性。

参考步骤

步骤1 检查权重的初始化方式，当初始权重为随机初始化时，确保随机特性与基准一致。

步骤2 检查权重的初始化方式，当初始权重为加载预训练权重文件初始化时，确保权重文件与基准一致。

步骤3 检查启动脚本，确保使用了正确的脚本和参数。

步骤4 对于集群训练，确保正确增加了集群训练的参数、或配置符合预期。尤其需要避免常见的各节点仅进行独立训练无任何信息同步的问题。

----结束

8.3.3.7 模型超参：与基准一致

目标

模型超参与参考基准一致。

思路

进行脚本迁移的用户可能对超参的一致性不熟悉，导致迁移完成后，但一些超参的计算事实上与参考基准不一致。

此时需要检查实际执行超参与参考基准的一致性。

出现较频繁的问题包括：

- 分布式训练的迁移过程中，全局batch size与单Device batch size的换算出错，导致NPU的全局batch size与参考基准全局batch size不一致。
- 分布式训练的迁移过程中，全局学习率与单Device学习率的换算出错，导致NPU的全局学习率与参考基准全局学习率不一致。

参考步骤

步骤1 对比检查迁移后的脚本与参考基准脚本中的超参设定参数，确保实际设定值一致。

步骤2 对比检查与超参相关的配置文件，确保实际使用的配置文件一致。

步骤3 运行调试（调试器或打印）参考基准脚本和迁移后的脚本，确认打印的超参数值一致。

步骤4 对比训练过程中的学习率打印，确保参考基准和迁移后训练的变化一致。

----结束

8.4 工具部署

工具简介

一键式精度分析工具，提供了对训练网络进行精度分析的常用功能，主要包括：

- **8.5 浮点异常检测**
- **8.7 整网数据比对**
- **8.6 融合异常检测**

该工具封装了TF Adapter的运行参数，帮助用户更方便的使能相关业务特性，同时对CANN Toolkit包中的精度分析工具进行了封装和功能扩展，从而帮助开发人员快速分析精度问题。

使用约束

- 当前该工具仅支持TensorFlow 1.15和TensorFlow 2.6训练场景，TensorFlow 1.15下的精度调优请参考《**TensorFlow 1.15网络模型迁移和训练指南**》。
- 不能同时采集溢出数据和精度数据。
- 采集溢出数据或精度数据时，都可能会产生较多结果文件，导致磁盘空间不足，请适当控制迭代次数。

工具部署

从**https://gitee.com/ascend/tools**下载**precision_tool**文件夹，上传到训练工作目录下，无需安装。目录结构示例：

```
├── resnet                          // 训练工作目录
│   ├── __init__.py
│   ├── imagenet_main.py              // 基于Imagenet数据集训练网络模型
│   ├── imagenet_preprocessing.py     // Imagenet数据集数据预处理模块
│   ├── resnet_model.py               // resnet模型文件
│   ├── resnet_run_loop.py            // 数据输入处理与运行循环（训练、验证、测试）
│   ├── cifar10_main                  // 训练执行入口文件
│   ├── ...
│   ├── precision_tool          // 一键式精度分析工具目录
│   │   ├── cli.py
│   │   ├── ...
```

- 如果CANN开发/运行环境合一部署，只需将**precision_tool**文件夹，上传到训练工作目录下即可。
- 如果CANN开发/运行环境独立部署，需要将**precision_tool**文件夹上传到CANN运行环境的训练工作目录下，同时将**precision_tool**上传到CANN开发环境的任意目录下。

说明

CANN运行环境（包含昇腾AI处理器，即启动NPU训练的环境），在训练精度调优工作中，主要用于训练时精度数据采集；

CANN开发环境（即安装CANN Toolkit软件的环境），在训练精度调优工作中，主要用于精度数据分析。

典型使用流程

图 8-1 CANN 开发/运行环境合一部署

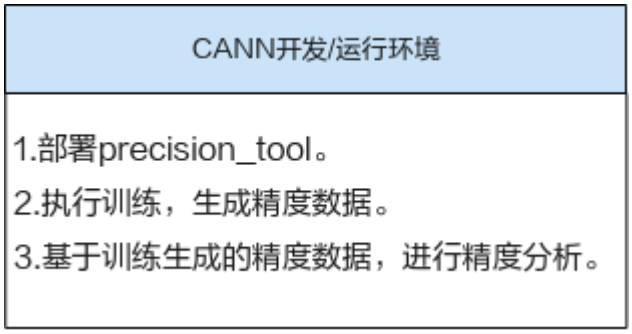
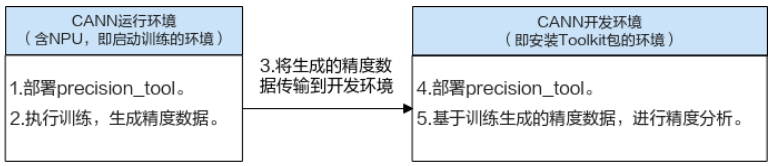


图 8-2 CANN 开发/运行环境独立部署



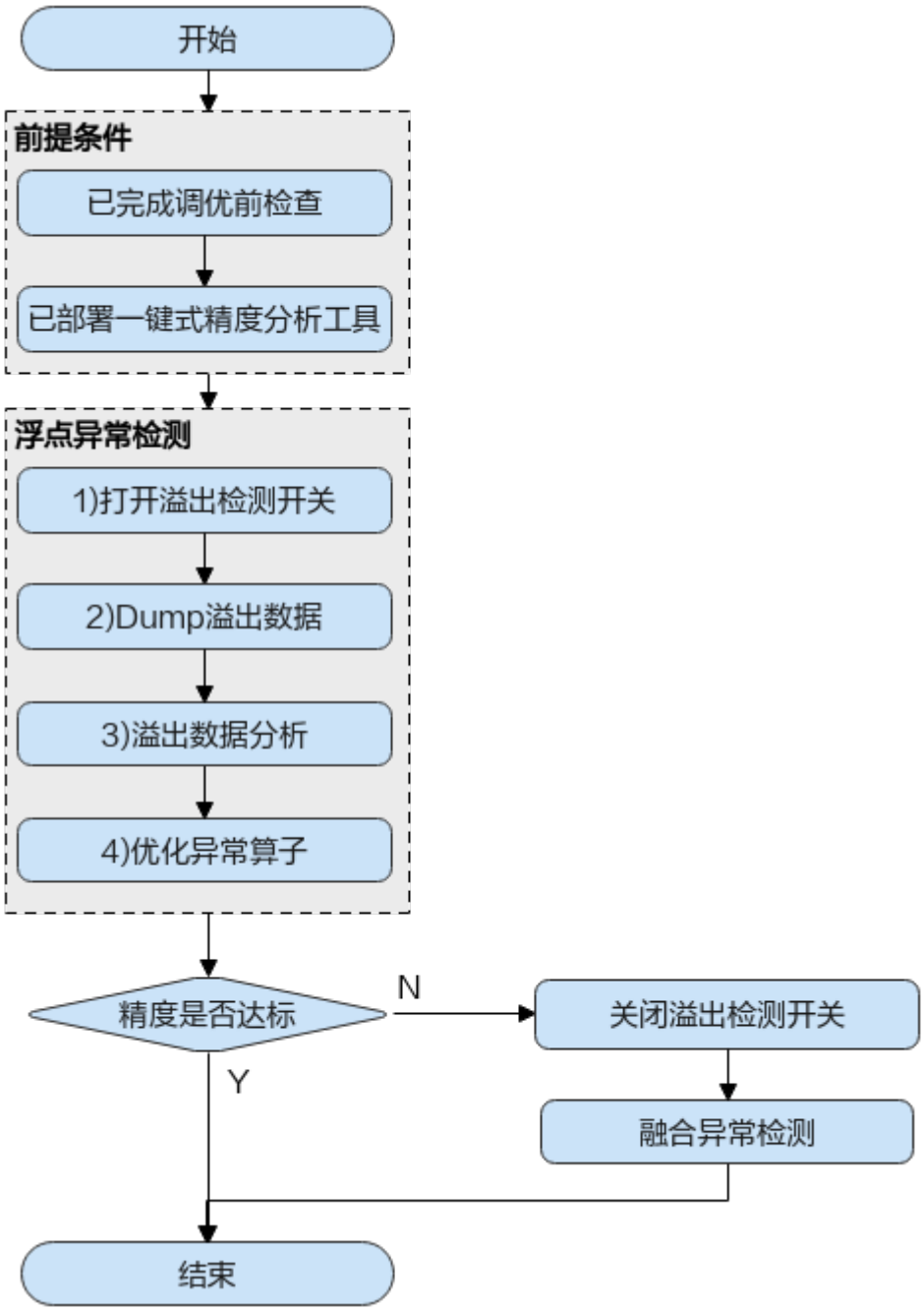
8.5 浮点异常检测

使用场景

训练网络执行过程中，可能发生频繁的浮点异常情况，即loss scale值下降次数较多或者直接下降为1，此时需要通过分析溢出数据，对频繁的浮点异常问题进行定界定位。

但在训练多个step的场景下，如果只是某个step出现了溢出，则可能是正常的偶发溢出，一般在开启loss scale的情况下，会自动跳过该step的训练结果，梯度不更新。对于这种偶发溢出场景一般可以不用关注。

溢出数据检测的主要过程为：



前提条件

1. 已完成**8.3 调优前检查**。
2. 已完成**8.4 工具部署**。
3. 进行溢出数据分析时依赖CANN Toolkit软件包中的工具，因此需要准备CANN开发环境，即Toolkit安装环境。

Dump 溢出数据

以下操作在NPU训练环境执行。

步骤1 修改训练脚本，使能算子溢出数据采集。

```
import precision_tool.tf_config as npu_tf_config
npu_tf_config.npu_device_dump_config(npu_device, action='overflow')
```

步骤2 执行训练，如果网络存在溢出，则在precision_data/overflow/dump下会生成溢出信息文件。

----结束

溢出数据分析

溢出数据分析依赖CANN Toolkit软件包中的atc工具和msaccucmp.py工具，以下操作需要在CANN开发环境，即Toolkit安装环境进行。

步骤1 将precision_tool和precision_data文件夹上传到Toolkit安装环境的任意目录下，目录结构示例：

```
├── precision_tool
│   ├── cli.py
│   ├── ...
├── precision_data
│   ├── overflow
│   │   ├── dump
```

步骤2 安装Python依赖。

```
pip3 install rich
```

步骤3 修改工具precision_tool/lib/config目录下的config.py。

```
# 依赖Toolkit包中的atc和msaccucmp.py工具，一般在run包安装目录，配置到父目录即可
# 默认Toolkit包安装在/usr/local/Ascend，可以不用修改，指定目录安装则需要修改
CMD_ROOT_PATH = '/usr/local/Ascend'
```

步骤4 启动PrecisionTool交互命令行。

python3 ./precision_tool/cli.py

进入交互命令行界面：

PrecisionTool >

步骤5 执行如下命令进行溢出数据分析，详细命令说明可参考**8.10.1 precision_tool命令参考**。

PrecisionTool > ac

根据数据量大小，分析过程需要时间不同，当执行过程中出现算子溢出，则会输出如下结果。

```
[FusedMulAdd][272] bert_encoder_layer_10_intermediate_dense_mul_FusedMulAdd
 - [AI Core][Status:32][TaskId:272] ['浮点计算有溢出']
 - First overflow file timestamp [1620369909487436] -
 |- FusedMulAdd.bert_encoder_layer_10_intermediate_dense_mul_FusedMulAdd.272.7.1620369909496889.input.0.npy
  |- [Shape: (4608, 3072)] [Dtype: float16] [Max: 5.676] [Min: -1.652] [Mean: -0.0671]
 |- FusedMulAdd.bert_encoder_layer_10_intermediate_dense_mul_FusedMulAdd.272.7.1620369909496889.input.2.npy
  |- [Shape: (4608, 3072)] [Dtype: float16] [Max: 126.94] [Min: -36.97] [Mean: -1.5]
 |- FusedMulAdd.bert_encoder_layer_10_intermediate_dense_mul_FusedMulAdd.272.7.1620369909496889.input.1.npy
  |- [Shape: (4608, 3072)] [Dtype: float16] [Max: 16110.0] [Min: 0.0] [Mean: 3.2]
 |- FusedMulAdd.bert_encoder_layer_10_intermediate_dense_mul_FusedMulAdd.272.7.1620369909496889.output.0.npy
  |- [Shape: (4608, 3072)] [Dtype: float16] [Max: 65500.0] [Min: -2294.0] [Mean: -1.586]
```

从上图可以看到：

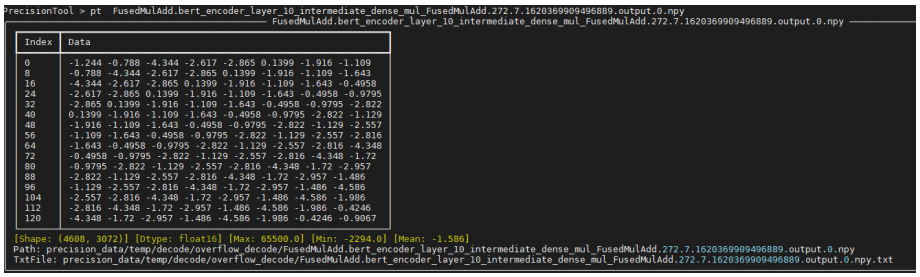
- 算子名为：bert_encoder_layer_10_intermediate_dense_mul_FusedMulAdd
- 算子类型为：FusedMulAdd
- 溢出status信息为：32，表示浮点计算有溢出。

- 溢出类型为：AI Core算子溢出，另外还可能会有其他类型的算子溢出（例如DHA Atomic Add或L2 Atomic Add），建议用户优先考虑并解决AI Core算子溢出问题。
- 算子的输入输出信息，包括shape、dtype、输入输出数据的最大值最小值。

说明

当出现多个算子溢出时，会出现N个溢出算子信息，默认按照算子执行顺序排序，由于后面算子溢出可能是因为前一个算子溢出导致，建议用户优先分析第一个异常算子。

步骤6 执行**pt (-n) [*.npy]**命令，可以查看对应dump数据块的数据信息。



----结束

分析思路参考

进行溢出数据分析的大致思路为：

1. 查看输入输出数据值。
   - 如果输入值中没有Nan和65504，输出数据中存在溢出值（65504/Nan等），则计算存在溢出；
   - 如果输入值中存在Nan或者65504，则需要继续分析前向算子或者用户模型的常量输入是否存在异常；
   - 否则可能在计算过程中存在溢出。
2. 查看溢出算子类型。
   - 对于自定义开发的算子，可以尝试自行进行算子溢出分析（结合算子公式和溢出值进行分析），排查自定义算子是否存在问题。
   - 对于CANN内置算子，也可以先尝试初步分析，如下为常用的分析方向：
     - 如果算子输出类型为float16，解析出的输出数据中出现65504/65500，则可以切换输出算子类型至float32计算，用户可以尝试以下两种方法：
       1) （推荐）方法一：通过**精度调优**的modify_mixlist修改混合精度模式算子黑白灰名单，调整算子精度模式。
       2) 方法二：通过**11.1.1.7 npu.keep_dtype_scope**接口，指定哪些算子保持原有精度。

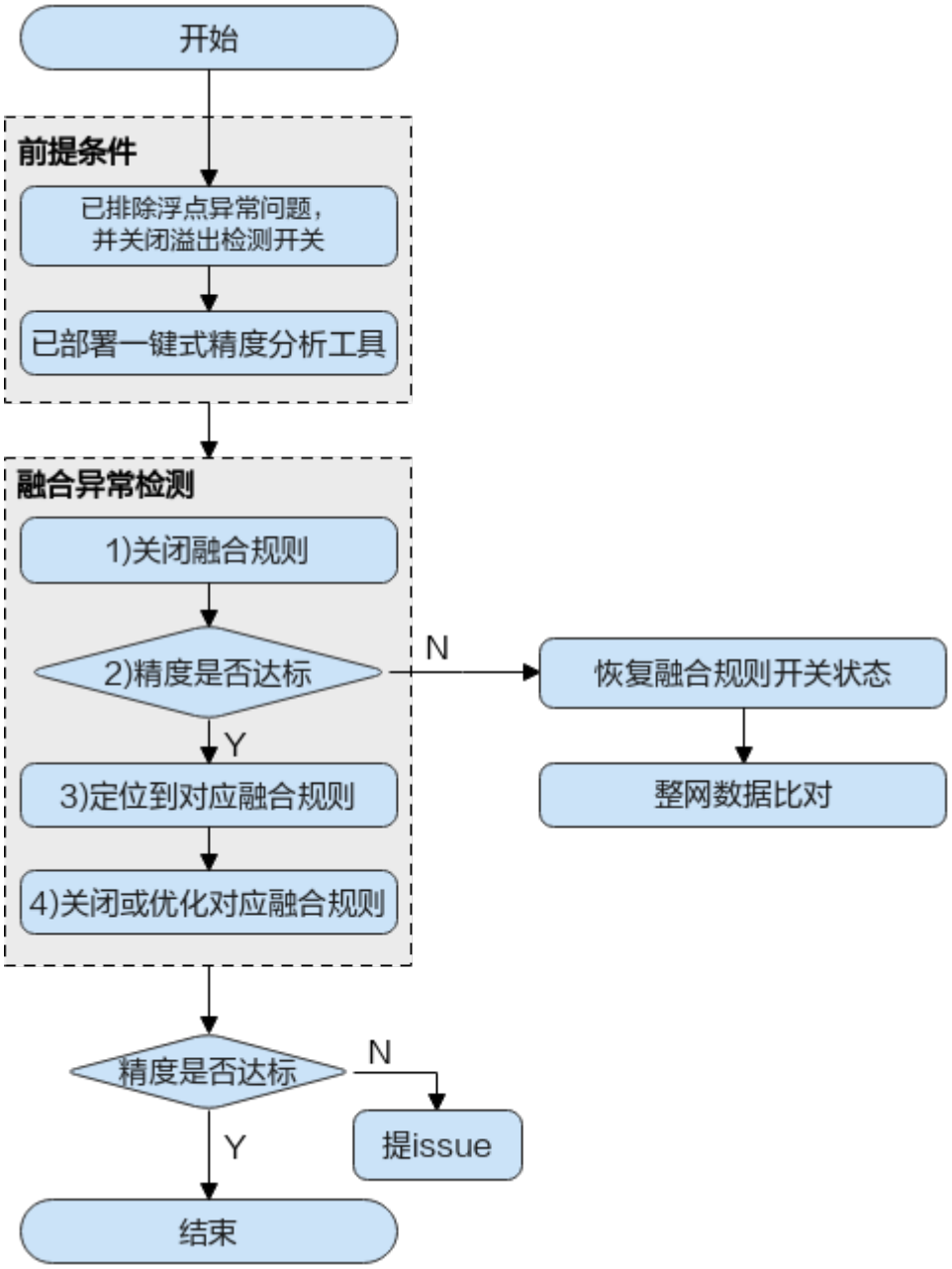
       ```
       import npu_device as npu
       with npu.keep_dtype_scope():
         v = tf.add(1, 1)
       ```
     - 如果输入输出数据中均未出现溢出值，则需要结合算子公式，分析数据计算过程中是否可能出现溢出。例如AvgPool先求和再平均，求和过程可能溢出，但平均后并未溢出。
3. 如果依旧无法解决，欢迎到**昇腾社区**提issue求助。

8.6 融合异常检测

使用场景

训练网络执行过程中，系统会根据内置的融合规则对网络中算子进行融合，以达到提高网络性能的效果。由于大多数融合是自动识别的，但可能存在未考虑到的场景，导致精度问题，因此可以尝试关闭相应融合规则，定界网络问题是否是由于融合导致。

融合异常检测的主要过程为：



前提条件

1. 已完成**8.4 工具部署**。
2. 已排除浮点异常问题，并关闭溢出检测开关。

操作步骤

步骤1 修改训练脚本，关闭全部融合规则。

```
import precision_tool.tf_config as npu_tf_config
npu_tf_config.npu_device_dump_config(npu_device, action='fusion_off')
```

步骤2 执行训练，检查网络精度是否有明显提高。

- 如果网络精度有明显提高，表明是融合问题导致，接下来需要参考**步骤3**定位是哪个融合规则的哪一层算子融合出现了问题。
- 如果网络精度无明显提高，需要恢复融合规则状态（将**步骤1**中关闭全部融合规则的代码注释掉），并进行**8.7 整网数据比对**。

步骤3 定位异常融合规则。

定位融合异常时依赖CANN Toolkit软件包中的atc工具和msaccucmp.py工具，以下操作需要在CANN开发环境，即Toolkit安装环境进行。

1. 开启融合规则，生成dump数据和图结构文件。

   1）恢复融合规则状态（将**步骤1**中关闭全部融合规则的代码注释掉），参考**基于NPU Dump精度数据**，在NPU环境执行训练，采集dump数据，该数据默认保存在precision_data/npu/debug_0目录下。

   2）将以上数据转存到precision_data/npu/debug_1目录下。

   mv precision_data/npu/debug_0/ precision_data/npu/debug_1

   3）执行atc命令，生成包含图结构信息的json文件。

   atc --mode=5 --om=precision_data/npu/debug_1/graph/ge_proto_00005_Build.txt --json=precision_data/npu/debug_1/test_on.json

   说明

   该命令行中ge_proto_00005_Build.txt文件名为举例，实际执行时，需要根据实际图文件名替换。

   在precision_data/npu/debug_1会存在多个类似文件名的图文件，需要找到计算图文件。一般情况选取方法为：将TensorFlow模型保存为pb文件，然后查看该模型，选取其中一个计算类算子的名字作为关键字，找包含该关键字的计算图文件；或者尝试选择文件大小最大的文件，计算图名称取计算图文件graph下的name字段值。

2. 关闭融合规则，生成dump数据和图结构文件。

   1）关闭全部融合规则（参考**步骤1**相关操作），再次在NPU环境执行训练，采集dump数据，该数据默认保存在precision_data/npu/debug_0目录下。

   ```
   npu_tf_config.npu_device_dump_config(npu_device, action='fusion_off|dump')
   ```

   2）执行atc命令，生成包含图结构信息的json文件。

   atc --mode=5 --om=precision_data/npu/debug_0/graph/ge_proto_00006_Build.txt --json=precision_data/npu/debug_0/test_off.json

3. 将融合规则关闭前后生成的dump数据进行比对。

   进入/home/HwHiAiUser/Ascend/ascend-toolkit/latest/toolkit/tools/operator_cmp/compare目录，执行如下命令：

   python3 msaccucmp.py compare -m precision_data/npu/debug_0/dump/20211016180613/1/ge_default_20211016180613_1/1/0 -g

precision_data/npu/debug_1/dump/20211016164504/1/ge_default_20211016164504_1/1/0 -f precision_data/npu/debug_1/test_on.json -cf precision_data/npu/debug_0/test_off.json -out out_dir

在out_dir目录生成精度比对结果。

4. 根据比对结果，找到精度异常的融合算子。
5. 根据该异常算子，匹配对应计算图txt文件，找到相应融合规则名称。如果定位有困难，欢迎到**昇腾社区**提issue求助。

步骤4 定位到具体融合规则后，先恢复融合规则状态（将**步骤1**中关闭全部融合规则的代码注释掉），然后仅关闭指定的融合规则。

```
import precision_tool.tf_config as npu_tf_config
npu_tf_config.npu_device_dump_config(npu_device, action='fusion_switch')
```

同时修改precision_tool下的融合规则配置文件fusion_switch.cfg，配置样例如下，on表示开启，off表示关闭。

```
{
    "Switch":{
        "GraphFusion":{
            "ConvToFullyConnectionFusionPass":"off",
        },
        "UBFusion":{
            "TbePool2dQuantFusionPass":"off"
        }
    }
}
```

说明

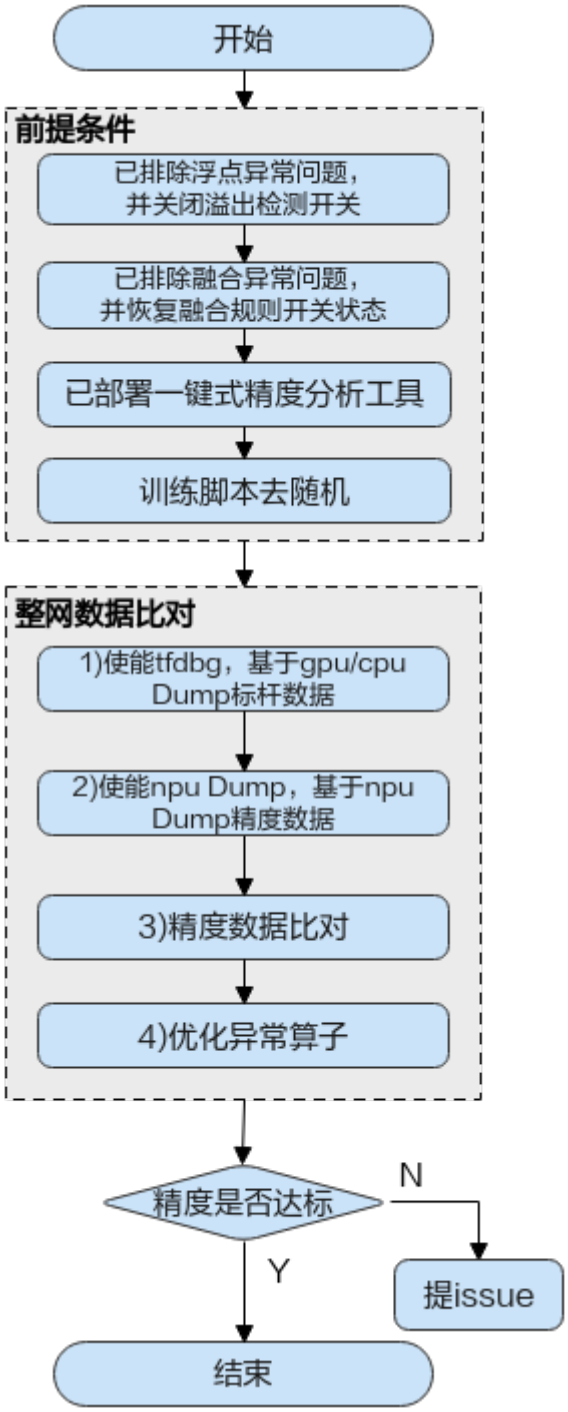
具体融合规则说明请参考《**图融合和UB融合规则参考**》。

----结束

8.7 整网数据比对

使用场景

排除以上问题后，在训练网络精度仍未达预期时，通过采集训练过程中各算子的运算结果（即Dump数据），然后和业界标准算子（如TensorFlow）运算结果进行数据偏差对比，快速定位到具体算子的精度问题。主要过程为：



前提条件

1. 已排除浮点异常问题，并关闭溢出检测开关。
2. 已排除融合异常问题，并恢复融合规则开关状态。
3. 已完成**8.4 工具部署**。
4. 整网数据比对前，需要先检查并去除训练脚本内部使用到的随机处理，避免由于输入数据不一致导致数据比对结果不可用。具体请参考**8.10.3 训练脚本去随机处理**。

5. 整网数据比对时，依赖CANN Toolkit软件包中的工具，因此需要准备CANN开发环境，即Toolkit安装环境。

基于 GPU/CPU Dump 标杆数据

- 在进行TensorFlow 2.x原始训练网络生成npy或dump数据前，要求有一套完整、可执行的标准TensorFlow模型训练工程。GPU训练环境准备可以参考**在ECS上快速创建GPU训练环境**，链接内容仅供参考，请以实际训练场景为准。
- 参见**tfdbg_ascend工具的readme文档**安装TensorFlow 2.x的debug工具tfdbg_ascend。
- 首先要把脚本中所有的随机全部关闭，包括但不限于对数据集的shuffle，参数的随机初始化，以及某些算子的隐形随机初始化（比如dense算子），确认自己脚本内所有参数均非随机初始化。

利用TensorFlow的debug工具tfdbg_ascend生成npy文件。详细的操作方法如下：

步骤1 修改tf训练脚本，在调起模型部分的训练脚本.py文件中修改配置。示例代码如下。

样例一：

1. 导入debug插件。
```
import tfdbg_ascend as dbg
```
2. 在每个step训练启动代码前配置如下代码，例如dump第5个step的数据。
```
    tfdbg.disable()
    if current_step == 5:
      tfdbg.enable()
      tfdbg.set_dump_path('home/test/gpu_dump')
```

样例二：

1. 导入debug插件。
```
import tfdbg_ascend as dbg
```
2. 例如dump第4个step的数据。dbg.enable不配置时，dump功能默认开启；dump路径不指定时，dump文件默认保存在训练脚本所在路径下。
```
class DumpConfig(tf.keras.callbacks.Callback):
    def __init__(self):
        super().__init__()
    def on_batch_begin(self, batch, logs={}):
        if batch == 4:
            dbg.enable()
            dbg.set_dump_path("/user/name1/pip_pkg/dump4")
        else:
            dbg.disable()
```
3. 注册回调函数（define callbacks ）。
```
# define callbacks
        callbacks = [
            ModelCheckpoint(
                f'models/model_epochs-{epochs}_batch-{batch_size}_loss-
{loss_function}_{Mask2FaceModel.get_datetime_string()}.h5'),
            LossHistory(batch_size),
            DumpConfig()
        ]

# fit the model 调起模型的代码位置
history = self.model.fit(train_dataset, validation_data=valid_dataset, epochs=1, callbacks=callbacks,
verbose=2)
```

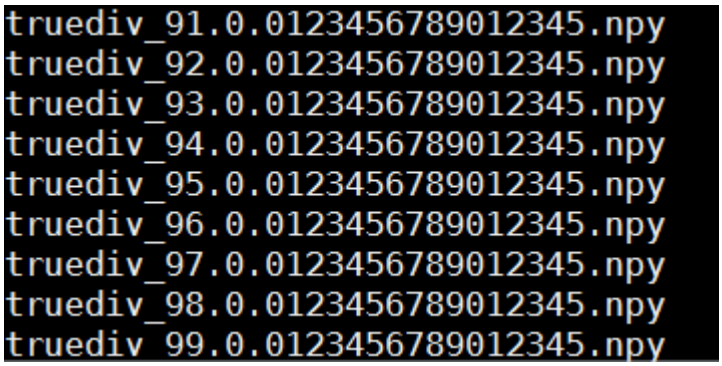
步骤2 执行训练脚本，训练任务停止后，在指定目录下生成*.npy文件。

步骤3 检查生成的npy文件命名是否符合规则，如**图8-3**所示。

说明

- npy文件命名规则：*{op_name}.{output_index}.{timestamp}*.npy，其中op_name字段需满足“A-Za-z0-9_-”正则表达式规则，timestamp需满足[0-9]{1,255}正则表达式，output_index为0~9数字组成。
- 如果因算子名较长，造成按命名规则生成的npy文件名超过255字符而产生文件名异常，这类算子不支持精度比对。

图 8-3 查询.npy 文件



----结束

基于 NPU Dump 精度数据

以下操作在NPU训练环境执行。Dump数据前，需要注意的是：

一般情况下，dump首个step的数据用作后续比对分析即可，为了避免随机权重导致比对不准确的问题，可以在训练开始前保存ckpt，并在训练时加载。如果确定是某个step的精度问题，则建议加载最靠近异常步的ckpt文件。

步骤1 修改工具precision_tool/lib/config目录下的config.py。

```
# dump特定step的数据，一般对比分析dump首层即可，即保持默认值，如需指定特定step可以修改，例如 '0|5|10'
TF_DUMP_STEP = '0'
```

步骤2 修改训练脚本，使能Dump数据采集。

以下修改会同时生成Dump数据和Dump图，用于精度数据比对。

```
import precision_tool.tf_config as npu_tf_config
npu_tf_config.npu_device_dump_config(npu_device, action='dump')
```

说明

除了此种方式，您也可以参考《**精度比对工具使用指南**》的方法修改训练脚本，采集Dump数据，但配置较为复杂，且采集到数据之后，需要手工提取并放在相应目录下，用于后续数据分析。注意两种方式不能重复配置。

步骤3 执行训练，会在precision_data/npu/debug_0目录下分别保存GE的Dump图和Dump数据文件。

----结束

精度数据比对

精度数据分析依赖CANN Toolkit软件包中的atc工具和msaccucmp.py工具，以下操作需要在CANN开发环境，即Toolkit安装环境进行。

步骤1 将precision_tool和precision_data（包括标杆数据和NPU的精度数据）文件夹上传到Toolkit安装环境的任意目录下，目录结构示例：

```
├── precision_tool
│   ├── cli.py
│   ├── ...
├── precision_data
│   ├── npu
│   │   ├── debug_0  // 存放npu dump数据
│   ├── tf
│   │   ├── dump     // 存放标杆dump数据
```

步骤2 安装Python依赖。

```
# graphviz为可选依赖，只有当需要绘制算子子图时才需要安装
pip3 install rich graphviz
# ubuntu/Debian
sudo apt-get install graphviz
# fedora/CentOS
sudo yum install graphviz
```

步骤3 修改工具precision_tool/lib/config目录下的config.py。

```
# 依赖Toolkit包中的atc和msaccucmp.py工具，配置为Toolkit包安装目录
# 默认Toolkit包安装在/usr/local/Ascend，可以不用修改，指定目录安装则需要修改
CMD_ROOT_PATH = '/usr/local/Ascend'
```

步骤4 启动PrecisionTool交互命令行。

python3 ./precision_tool/cli.py

进入交互命令行界面：

PrecisionTool >

步骤5 执行**ac -l [limit_num] (-c)**命令进行整网精度比对。

PrecisionTool > ac -c

根据数据量大小，比对过程需要时间不同。

对比结果会以csv的格式存放在precision_data/temp/vector_compare目录中：

```
precision_data/temp/vector_compare/
├──── 1621512034
│     ├──── result_20210520200038.csv
│     ├──── result_20210520200056.csv
│     └──── result_20210520200113.csv
└──── 1621514375
   ├──── result_20210520203938.csv
   ├──── result_20210520203957.csv
   └──── result_20210520204014.csv
```

您可以直接打开csv文件进行分析，具体请参考**8.10.2 整网精度比对结果文件说明**。

步骤6 除了直接打开csv文件进行精度分析外，您也可以使用**vcs -f [file_name] -c [cos_sim_threshold] -l [limit]**命令筛选比对结果。

vcs命令默认筛选余弦相似度小于0.98的结果，您也可以通过-c参数自定义阈值：

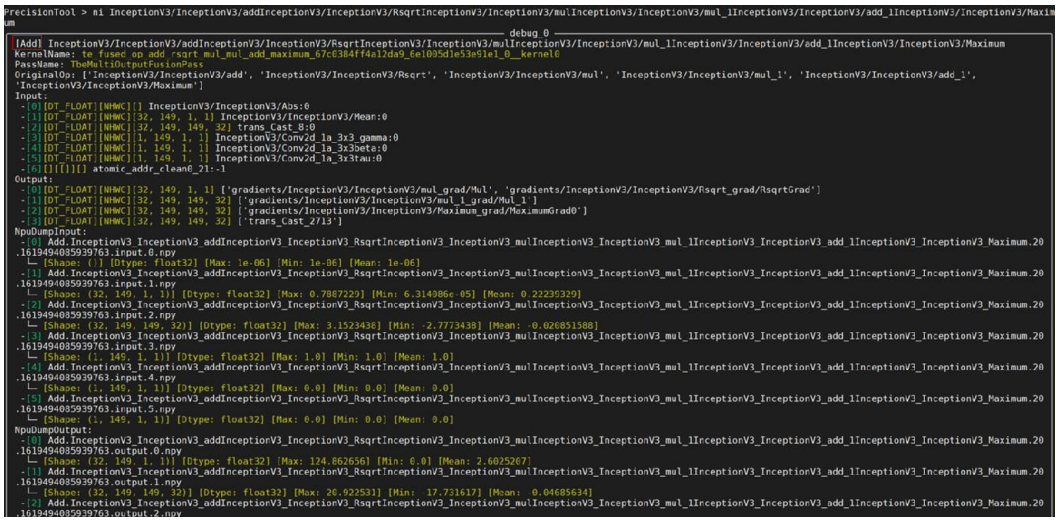
- Left：表示基于NPU运行生成的dump数据的算子名。
- Right：表示基于GPU/CPU运行生成的npy或dump数据的算子名。
- Input和Output：表示该算子各输入输出的余弦相似度算法比对结果，范围是[-1,1]，比对的结果如果越接近1，表示两者的值越相近，越接近-1意味着两者的值越相反。

从上图的比对结果可以看到，算子的输入基本一致，但第一个输出与标杆存在明显差异（余弦相似度为0.806927，小于0.98），说明该算子可能存在精度问题。

说明

当出现多个算子精度问题时，会出现N个异常算子信息，默认按照算子执行顺序排序，由于后面算子精度问题可能是因为前一个算子精度问题导致，建议用户优先分析第一个异常算子。

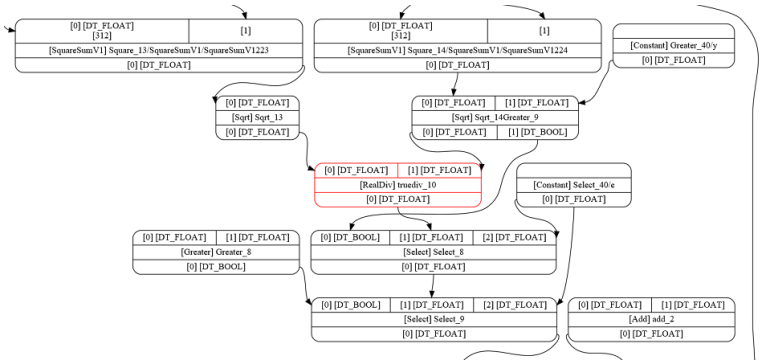
步骤7 执行**ni (-n) [op_name] -s [save sub graph deep]**命令，可以查询异常算子的节点信息。



ni命令可以根据传入的算子名称，得到如下关键信息：

1. 算子类型，以上图为例，算子类型为Add。

   另外，PassName表示该算子为融合算子，对应值表示融合规则名称，OriginOp为融合前的算子，表明是由于算子融合导致精度问题。正常情况下，融合问题应该在**8.5 浮点异常检测**阶段解决。
2. 自动解析Dump数据，打印Dump数据的基础信息（max/min/mean）。
3. 如果传入-s，则会保存一个以当前算子为中心，指定深度的子图结构，例如：



----结束

分析思路参考

整网数据比对提供了一个全网Dump数据与TF标杆数据的逐层累计比对报表，由于整网数据由于硬件差异本身是存在一定给误差的，且误差会随着层数增多而累计，即便精度正常的网络数值上也会存在细微误差，一般采用余弦相似度做初步的可疑算子筛选（注意：余弦相似度较高也不一定说明没有问题，但较低一般代表可能存在问题），精度对比结果可以给出一个大致的分析方向。

1. 根据算子类型，可以判断该算子是否为用户自定义算子：
   - 对于自定义算子，一般由用户自行分析算子的实现逻辑是否与标杆一致，可以ni命令提供的算子参数信息，以及dump数据进行单算子分析。
   - 对于CANN内置算子，如果算子输入或输出类型为float16，则可以切换算子类型至float32计算，用户可以尝试以下两种方法：
     i. （推荐）方法一：通过**性能调优**的modify_mixlist修改混合精度模式算子黑白灰名单，调整算子精度模式。
     ii. 方法二：通过**11.1.1.7 npu.keep_dtype_scope**接口，指定哪些算子保持原有精度。

     ```
     import npu_device as npu
     with npu.keep_dtype_scope():
       v = tf.add(1, 1)
     ```
2. 如果依旧无法解决，欢迎到**昇腾社区**提issue求助。

8.8 随机错误检测

使用场景

网络执行过程中，可能存在部分计算过程在相同输入的情况下给出了不同的输出的问题。

当出现以上随机问题时，可以通过执行两次训练，并分别采集各算子的运算结果（即dump数据），通过比对分析，从而快速定位到导致随机问题的算子层。

操作步骤

步骤1 参考**基于NPU Dump精度数据**，在NPU环境执行训练，采集dump数据，该数据默认保存在precision_data/npu/debug_0目录下。

步骤2 将以上数据转存到precision_data/npu/debug_1目录下。

mv precision_data/npu/debug_0/ precision_data/npu/debug_1

步骤3 再次在NPU环境执行训练，采集dump数据，该数据默认保存在precision_data/npu/debug_0目录下。

步骤4 启动PrecisionTool交互命令行。

python3 ./precision_tool/cli.py

进入交互命令行界面：

PrecisionTool >

说明

如需退出，可执行ctrl + c。

步骤5 使用**vc -lt [left_path] -rt [right_path] -g [graph]**命令进行整网数据对比。

vc -lt -rt precision_data/npu/debug_1/dump/20211016164504/1/ge_default_20211016164504_1/1/0 precision_data/npu/debug_0/dump/20211016180613/1/ge_default_20211016180613_1/1/0

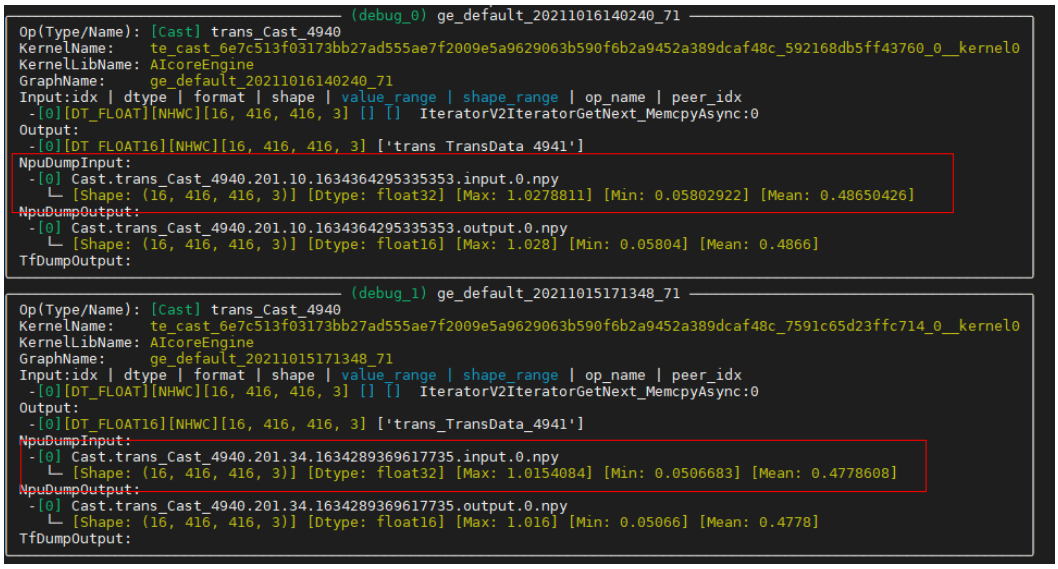
在out_dir目录生成精度比对结果，可参考**8.10.2 整网精度比对结果文件说明**进行数据分析。

步骤6 针对以上结果，还可以使用precision_tool的**ni (-n) [op_name] -s [save sub graph deep]**命令进行单层数据比对分析。

python3 precision_tool/cli.py

PrecisionTool > ni xxx

当precision_data/npu/目录下同时存在debug_0和debug_1的时候，ni命令会同时解析两个文件夹下相同算子名的dump文件，从该解析结果中，可以比较直观的看出数据差异。



----结束

分析思路参考

基于整网比对结果，一般采用余弦相似度做初步的可疑算子筛选（注意：余弦相似度较高也不一定说明没有问题，但较低一般代表可能存在问题），精度对比结果可以给出一个大致的分析方向。

1. 根据算子类型，可以判断该算子是否为用户自定义算子：
   - 对于自定义算子，一般由用户自行分析算子的实现逻辑是否与标杆一致，可以ni命令提供的算子参数信息，以及dump数据进行单算子分析。
   - 对于CANN内置算子，如果算子输入或输出类型为float16，则可以切换算子类型至float32计算，用户可以尝试以下两种方法：
     i. （推荐）方法一：通过**性能调优**的modify_mixlist修改混合精度模式算子黑白灰名单，调整算子精度模式。
     ii. 方法二：通过**11.1.1.7 npu.keep_dtype_scope**接口，指定哪些算子保持原有精度。
     ```
     import npu_device as npu
     with npu.keep_dtype_scope():
         v = tf.add(1, 1)
     ```
2. 如果依旧无法解决，欢迎到**昇腾社区**提issue求助。

8.9 跨版本精度问题检测

由于升级CANN版本导致的精度问题，需要基于前后版本，依次dump这两个版本的精度数据，进行数据比对，方法与**8.8 随机错误检测**一致。

8.10 附录

8.10.1 precision_tool 命令参考

ac -l [limit_num] (-c)

- 命令说明：

  自动化检测命令，列出Fusion信息;解析算子溢出信息。-c：可选，进行全网比对；-l：可选，限制输出结果的条数（overflow解析的条数等）
- 命令示例：

  PrecisionTool > ac -c
- 执行结果：

  ```
  ╭─────────────────────────────────────────────────────────────────────────────╮
  │ [TransData][327] trans_TransData_1170                                       │
  │  - [AI Core][Status:32][TaskId:327] ['浮点计算有溢出']                        │
  │  - First overflow file timestamp [1619347786532995] -                       │
  │  |- TransData.trans_TransData_1170.327.1619347786532995.input.0.npy         │
  │   |- [Shape: (32, 8, 8, 320)] [Dtype: bool] [Max: True] [Min: False] [Mean: 0.11950836181640626] │
  │  |- TransData.trans_TransData_1170.327.1619347786532995.output.0.npy        │
  │   |- [Shape: (32, 20, 8, 8, 16)] [Dtype: bool] [Max: True] [Min: False] [Mean: 0.07781982421875] │
  ╰─────────────────────────────────────────────────────────────────────────────╯
  ```

run [command]

- 命令说明：

不退出交互命令环境执行shell命令，与内置命令不冲突的可以直接执行，否则需要加run前缀。

- 命令示例：

  PrecisionTool > run vim cli.py

  PrecisionTool > vim cli.py

ls -n [op_name] -t [op_type] -f [fusion_pass] -k [kernel_name]

- 命令说明：

  通过[算子名]/[算子类型]查询网络里的算子，模糊匹配。-n：算子节点名称；-t：算子类型；-f：融合类型；-k：kernel name。

- 命令示例：

  PrecisionTool > ls -t Mul -n mul_3 -f TbeMulti

- 执行结果

```
[Mul][TbeMultiOutputFusionPass] InceptionV3/InceptionV3/Mixed_5b/Branch_1/mul_3
[Mul][TbeMultiOutputFusionPass] InceptionV3/InceptionV3/Mixed_5c/Branch_1/mul_3
[Mul][TbeMultiOutputFusionPass] InceptionV3/InceptionV3/Mixed_5d/Branch_1/mul_3
[Mul][TbeMultiOutputFusionPass] InceptionV3/InceptionV3/Mixed_6b/Branch_1/mul_3
```

ni (-n) [op_name] -s [save sub graph deep]

- 命令说明：

  通过[算子名]查询算子节点信息，-n：指定节点名称；-g：graph名称；-d：显示attr信息；-s：保存一个以当前算子节点为根，深度为参数值的子图。

- 命令示例：

  PrecisionTool > ni gradients/InceptionV3/InceptionV3/Mixed_7a/Branch_0/Maximum_1_grad/GreaterEqual -s 3

- 执行结果：

```
╭──────[GreaterEqual]gradients/InceptionV3/InceptionV3/Mixed_7a/Branch_0/Maximum_1_grad/GreaterEqual──────╮
│ [GreaterEqual] gradients/InceptionV3/InceptionV3/Mixed_7a/Branch_0/Maximum_1_grad/GreaterEqual           │
│ Input:                                                                                                   │
│  -[0][DT_FLOAT][NHWC][32, 8, 8, 320] InceptionV3/InceptionV3/Mixed_7a/Branch_0/add_3:0                   │
│  -[1][DT_FLOAT][NHWC][1, 8, 1, 1] InceptionV3/Mixed_7a/Branch_0/Conv2d_1a_3x3tau:0                       │
│  -[2][][[]][]                                                                                            │
│ atomic_addr_clean0_21:-1                                                                                 │
│ Output:                                                                                                  │
│  -[0][DT_BOOL][NHWC][32, 8, 8, 320]                                                                      │
│ ['trans_TransData_1170']                                                                                 │
│ NpuDumpInput:                                                                                            │
│  -[0] GreaterEqual.gradients_InceptionV3_InceptionV3_Mixed_7a_Branch_0_Maximum_1_grad_GreaterEqual.325.1619494134722860.input.0.npy │
│    |- [Shape: (32, 8, 8, 320)] [Dtype: float32] [Max: 5.846897] [Min: -8.368301] [Mean: -0.72565556]    │
│  -[1] GreaterEqual.gradients_InceptionV3_InceptionV3_Mixed_7a_Branch_0_Maximum_1_grad_GreaterEqual.325.1619494134722860.input.1.npy │
│    |- [Shape: (1, 8, 1, 1)] [Dtype: float32] [Max: 0.0] [Min: 0.0] [Mean: 0.0]                           │
│ NpuDumpOutput:                                                                                           │
│                                                                                                          │
│  -[0] GreaterEqual.gradients_InceptionV3_InceptionV3_Mixed_7a_Branch_0_Maximum_1_grad_GreaterEqual.325.1619494134722860.output.0.npy │
```



```
│  |- [Shape: (32, 8, 8, 320)] [Dtype: bool] [Max: True] [Min: False] [Mean:
0.1176300048828125]                                │
│ CpuDumpOutput:                                                                    │
│  -[0]
gradients_InceptionV3_InceptionV3_Mixed_7a_Branch_0_Maximum_1_grad_GreaterEqual.0.1619492699
305998.npy                              │
│  |- [Shape: (32, 8, 8, 320)] [Dtype: bool] [Max: True] [Min: False] [Mean:
0.11764373779296874]                               │
╰──────────────────────────────────────────────────────────────────────────────
────╯
2021-04-27 14:39:55 (15178) -[DEBUG]write 14953 bytes to './precision_data/dump/temp/op_graph/
GreaterEqual.gradients_InceptionV3_InceptionV3_Mixed_7a_Branch_0_Maximum_1_grad_GreaterEqual.
3.gv'
2021-04-27 14:39:55 (15178) -[INFO]Sub graph saved to /root/sym/inception/precision_data/dump/
temp/op_graph
```

pt (-n) [*.npy]

- 命令说明

  查看某个dump数据块的数据信息，并保存到txt文件。-n：可选，待查看的数据文件名。

- 命令示例：

  PrecisionTool > pt
  TransData.trans_TransData_1170.327.1619347786532995.input.0.npy

- 执行结果：

```
╭──────────────────────────────────────────────────────────────╮
│ Shape: (32, 8, 8, 320)                                        │
│ Dtype: bool                                                  │
│ Max: True                                                    │
│ Min: False                                                   │
│ Mean: 0.11950836181640626                                            │
│ Path: ./precision_data/dump/temp/overflow_decode/
TransData.trans_TransData_1170.327.1619347786532995.input.0.npy        │
│ TxtFile: ./precision_data/dump/temp/overflow_decode/
TransData.trans_TransData_1170.327.1619347786532995.input.0.npy.txt │
╰──────────────────────────────────────────────────────────────╯
```

cp (-n) [left *.npy] [right *.npy] -p [print num] -al [atol] -rl [rtol]

- 命令说明：

  对比两个tensor的数据。

  -n：指定需要对比的两个numpy名；

  -p：指定输出的错误数据的个数及前多少个数据；

  -al/rl：指定相对误差的参数，在两个场景中用到：

  1. np.allclose(left, right, atol=al, rtol=rl)

  2. err_cnt += 1 if abs(data_left[i] - data_right[i]) > (al + rl * abs(data_right[i]))

  -s：保存成txt文件，默认打开

- 命令示例：

  PrecisionTool > cp
  Add.InceptionV3_InceptionV3_Mixed_7a_Branch_0_add_3.323.1619494134703053.output.0.npy
  InceptionV3_InceptionV3_Mixed_7a_Branch_0_add_3.0.1619492699305998.npy
  -p 10 -s -al 0.002 -rl 0.005

- 执行结果：

```
            Error Item Table                                    Top Item Table
┌───────────┬──────────────────────┬──────────────────┬────────────────────
───────────┐ ┌──────────┬──────────────────┬──────────────────┐
─────────────────────────┐
│ Index │ Left          │ Right          │ Diff           │ │ Index │ Left          │ Right          │ Diff          │
├───────────┼──────────────────────┼──────────────────┼────────────────────
───────────┤ ├──────────┼──────────────────┼──────────────────┤
─────────────────────────┤
│ 155  │ 0.024600908   │ 0.022271132  │ 0.002329776  │ │ 0    │ -0.9206961  │ -0.9222216  │
0.0015255213  │
│ 247  │ 0.015752593   │ 0.017937578  │ 0.0021849852 │ │ 1    │ -0.6416973  │ -0.64051837 │
0.0011789203  │
│ 282  │ -0.0101207765 │ -0.007852031 │ 0.0022687456 │ │ 2    │ -0.35383835 │ -0.35433492 │
0.0004965663  │
│ 292  │ 0.019581757   │ 0.02240482   │ 0.0028230622 │ │ 3    │ -0.18851271 │ -0.18883198 │
0.00031927228 │
│ 640  │ -0.06593232   │ -0.06874806  │ 0.0028157383 │ │ 4    │ -0.43508735 │ -0.43534422 │
0.00025686622 │
│ 1420 │ 0.09293677    │ 0.09586689   │ 0.0029301196 │ │ 5    │ 1.4447614   │ 1.4466647   │
0.0019032955  │
│ 1462 │ -0.085207745  │ -0.088047795 │ 0.0028400496 │ │ 6    │ -0.3455438  │ -0.3444429  │
0.0011008978  │
│ 1891 │ -0.03433288   │ -0.036525503 │ 0.002192624  │ │ 7    │ -0.6560242  │ -0.6564579  │
0.0004336834  │
│ 2033 │ 0.06828873    │ 0.07139922   │ 0.0031104907 │ │ 8    │ -2.6964858  │ -2.6975214  │
0.0010356903  │
│ 2246 │ -0.06376442   │ -0.06121233  │ 0.002552092  │ │ 9    │ -0.73746175 │ -0.73650354 │
0.00095820427 │
└────────┴──────────────────────┴──────────────────┴──────────────────┘ └──────────┴──
─────────────┴──────────────────┴──────────────────────┘
╭───────────────────────────────────────────────────────────────────────
───────────────────────────────────────────────────────────────────────
──────────╮
│ Left:                                                                                     │
│  |- NpyFile: ./precision_data/dump/temp/decode/
Add.InceptionV3_InceptionV3_Mixed_7a_Branch_0_add_3.323.1619494134703053.output.0.npy      │
│  |- TxtFile: ./precision_data/dump/temp/decode/
Add.InceptionV3_InceptionV3_Mixed_7a_Branch_0_add_3.323.1619494134703053.output.0.npy.txt │
│  |- NpySpec: [Shape: (32, 8, 8, 320)] [Dtype: float32] [Max: 5.846897] [Min: -8.368301] [Mean:
-0.72565556]                 │
│ DstFile:                                                                                  │
│  |- NpyFile: ./precision_data/dump/cpu/
InceptionV3_InceptionV3_Mixed_7a_Branch_0_add_3.0.1619492699305998.npy                     │
│  |- TxtFile: ./precision_data/dump/cpu/
InceptionV3_InceptionV3_Mixed_7a_Branch_0_add_3.0.1619492699305998.npy.txt                 │
│  |- NpySpec: [Shape: (32, 8, 8, 320)] [Dtype: float32] [Max: 5.8425903] [Min: -8.374472] [Mean:
-0.7256237]                  │
│ NumCnt:
655360                                                                            │
│ AllClose: False                                                                           │
│ CosSim:
0.99999493                                                                        │
│ ErrorPer: 0.023504638671875  (rl= 0.005, al=
0.002)                                                 │
╰───────────────────────────────────────────────────────────────────────
───────────────────────────────────────────────────────────────────────
──────────╯
```

vcs -f [file_name] -c [cos_sim_threshold] -l [limit]

- 命令说明：

  查看精度比对结果的概要信息，可以根据余弦相似度阈值过滤出低于阈值的算子/信息

  -f (--file) 可选，指定csv文件，不设置则默认遍历precision_data/temp/vector_compare/目录下最近产生的对比目录内的所有csv

  -c (--cos_sim) 可选，指定筛选所使用的余弦相似度阈值，默认0.98

-l (--limit) 可选，指定输出前多少个结果，默认值3

- 命令示例：

  PrecisionTool > vcs -c 0.98 -l 2

- 执行结果：

```
 2021-05-31 14:48:56 (2344298) -[INFO]Sub path num:[1]. Dirs[['20210529145750']],
choose[20210529145750]
 2021-05-31 14:48:56 (2344298) -[DEBUG]Find ['result_20210529145751.csv',
'result_20210529145836.csv', 'result_20210529145837.csv', 'result_20210529145849.csv',
'result_20210529150404.csv', 'result_20210529151102.csv'] result files in dir precision_data/temp/
vector_compare/20210529145750
 2021-05-31 14:48:56 (2344298) -[INFO]Find 0 ops less then 0.98 in precision_data/temp/
vector_compare/20210529145750/result_20210529145751.csv
 2021-05-31 14:48:56 (2344298) -[INFO]Find 0 ops less then 0.98 in precision_data/temp/
vector_compare/20210529145750/result_20210529145836.csv
 2021-05-31 14:48:56 (2344298) -[INFO]Find 1 ops less then 0.98 in precision_data/temp/
vector_compare/20210529145750/result_20210529145837.csv
 2021-05-31 14:48:56 (2344298) -[INFO]Find 2 ops less then 0.98 in precision_data/temp/
vector_compare/20210529145750/result_20210529145849.csv
 2021-05-31 14:48:56 (2344298) -[INFO]Find 2 ops less then 0.98 in precision_data/temp/
vector_compare/20210529145750/result_20210529150404.csv
 2021-05-31 14:48:56 (2344298) -[INFO]Find 0 ops less then 0.98 in precision_data/temp/
vector_compare/20210529145750/result_20210529151102.csv
┌── [578] pixel_cls_loss/cond_1/TopKV2 ──┐
│ Left:  ['pixel_cls_loss/cond_1/TopKV2']  │
│ Right: ['pixel_cls_loss/cond_1/TopKV2']  │
│ Input:                                   │
│  - [0]1.0        - [1]nan                │
│ Output:                                  │
│  - [0]0.999999   - [1]0.978459           │
└──────────────────────────────────────────┘
┌── [490] gradients/AddN_5 ──┐
│ Left:  ['gradients/AddN_5']  │
│ Right: ['gradients/AddN_5']  │
│ Input:                       │
│  - [0]nan        - [1]1.0    │
│ Output:                      │
│  - [0]0.05469                │
└──────────────────────────────┘
```

vc -lt [left_path] -rt [right_path] -g [graph]

- 命令说明：

  手动指定两个目录，进行整网精度比对。

  -lt 必选，其中一个文件目录

  -rt 必选，另一个目录，一般指标杆目录

  说明

  需要指定到dump数据所在的目录层级，例如：precision_data/npu/debug_0/dump/20220217095546/3/ge_default_20220217095547_1/1/0/

  -g 可选，指定-g将尝试解析graph内的映射关系比对（一般用于NPU和TF之间的数据比对，NPU与NPU之间比对不需要，直接按照算子name对比）

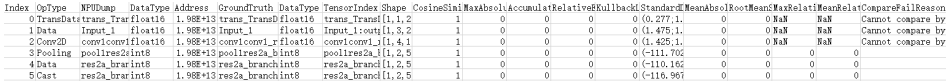
- 命令示例：

  PrecisionTool > vc -lt /path/left -rt /path/right

8.10.2 整网精度比对结果文件说明

整网精度比对会生成比对结果文件result_*.csv，文件内容如下图所示：

图 8-4 全算法维度比对结果



Index	OpType	NPUDump	DataType	Address	GroundTruth	DataType	TensorIndex	Shape	CosineSimi	MaxAbsolu	AccumulatRelativeE	Kullback	StandardE	MeanAbsol	RootMean	MaxRelati	MeanRelat	CompareFailReason
0	TransDat	trans_Trar	float16	1.98E+13	trans_TransD	float16	trans_Transf	[1,1,2	1	0	0	0	0	[0.277,1.	0	0	NaN	NaN	Cannot compare by
1	Data	Input_1	float16	1.98E+13	Input_1	float16	Input_1:out	[1,3,2	1	0	0	0	0	[1.475,1.	0	0	NaN	NaN	Cannot compare by
2	Conv2D	conv1conv1	float16	1.98E+13	conv1conv1_r	float16	conv1conv1_i	[1,4,1	1	0	0	0	0	[1.425,1.	0	0	NaN	NaN	Cannot compare by
3	Pooling	poollres2a	int8	1.98E+13	poollres2a_b	int8	poollres2a_i	[1,2,5	1	0	0	0	0	[-111.702	0	0	0	0	
4	Data	res2a_brar	int8	1.98E+13	res2a_branch	int8	res2a_branc	[1,2,5	1	0	0	0	0	[-110.162	0	0	0	0	
5	Cast	res2a_brar	int8	1.98E+13	res2a_branch	int8	res2a_branc	[1,2,5	1	0	0	0	0	[-116.967	0	0	0	0	

文件参数说明如下表所示：

表 8-3 模型比对结果参数说明

参数	说明
Index	网络模型中算子的ID。
OpSequence	部分算子比对时算子运行的序列。即-f参数指定的全网层信息文件中算子的ID。仅配置-r或-s参数时展示。
OpType	算子类型。指定-f参数时获取算子类型。
NPUDump	表示My Output模型的算子名。
DataType	表示NPU Dump侧数据算子的数据类型。
Address	dump tensor的内存内存地址。用于判断算子的内存问题。仅基于昇腾AI处理器运行生成的dump数据文件在整网比对时可提取该数据。
GroundTruth	表示Ground Truth模型的算子名。
DataType	表示Ground Truth侧数据算子的数据类型。
TensorIndex	表示基于昇腾AI处理器运行生成的dump数据的算子的input ID和output ID。
Shape	比对的Tensor的Shape。
OverFlow	溢出算子。显示YES表示该算子存在溢出；显示NO表示算子无溢出；显示NaN表示不做溢出检测。配置-overflow_detection参数时展示。
CosineSimilarity	进行余弦相似度算法比对出来的结果，取值范围为[-1,1]，比对的结果如果越接近1，表示两者的值越相近，越接近-1意味着两者的值越相反。
MaxAbsoluteError	进行最大绝对误差算法比对出来的结果，取值范围为0到无穷大，值越接近于0，表明越相近，值越大，表明差距越大。
AccumulatedRelativeError	进行累积相对误差算法比对出来的结果，取值范围为0到无穷大，值越接近于0，表明越相近，值越大，表明差距越大。
RelativeEuclideanDistance	进行欧氏相对距离算法比对出来的结果，取值范围为0到无穷大，值越接近于0，表明越相近，值越大，表明差距越大。
KullbackLeiblerDivergence	进行KL散度算法比对出来的结果，取值范围为0到无穷大。KL散度越小，真实分布与近似分布之间的匹配越好。

参数	说明
StandardDeviation	进行标准差算法比对出来的结果，取值范围为0到无穷大。标准差越小，离散度越小，表明越接近平均值。该列显示两组数据的均值和标准差，第一组展示基于昇腾AI处理器运行生成的dump数据的数值(均值;标准差)，第二组展示基于GPU/CPU运行生成的dump数据的数值(均值;标准差)。
MeanAbsoluteError	表示平均绝对误差。取值范围为0到无穷大，MeanAbsoluteError趋于0，RootMeanSquareError趋于0，说明测量值与真实值越近似；MeanAbsoluteError趋于0，RootMeanSquareError越大，说明存在局部过大的异常值；MeanAbsoluteError越大，RootMeanSquareError等于或近似MeanAbsoluteError，说明整体偏差越集中；MeanAbsoluteError越大，RootMeanSquareError越大于MeanAbsoluteError，说明存在整体偏差，且整体偏差分布分散；不存在以上情况的例外情况，因为RMSE ≥ MAE恒成立。
RootMeanSquareError	表示均方根误差。取值范围为0到无穷大，MeanAbsoluteError趋于0，RootMeanSquareError趋于0，说明测量值与真实值越近似；MeanAbsoluteError趋于0，RootMeanSquareError越大，说明存在局部过大的异常值；MeanAbsoluteError越大，RootMeanSquareError等于或近似MeanAbsoluteError，说明整体偏差越集中；MeanAbsoluteError越大，RootMeanSquareError越大于MeanAbsoluteError，说明存在整体偏差，且整体偏差分布分散；不存在以上情况的例外情况，因为RMSE ≥ MAE恒成立。
MaxRelativeError	表示最大相对误差。取值范围为0到无穷大，值越接近于0，表明越相近，值越大，表明差距越大。
MeanRelativeError	表示平均相对误差。取值范围为0到无穷大，值越接近于0，表明越相近，值越大，表明差距越大。
CompareFailReason	算子无法比对的原因。 若余弦相似度为1，则查看该算子的输入或输出shape是否为空或全部为1，若为空或全部为1则算子的输入或输出为标量，提示：this tensor is scalar。
备注： ● 显示“*”，表示在NPU侧新增的算子无对应的标准算子；“NaN”表示无比对结果。 ● 余弦相似度和KL散度比较结果为NaN，其他算法有比较数据，则表明左侧或右侧数据为0；KL散度比较结果为Inf，表明右侧数据有一个为0。	

8.10.3 训练脚本去随机处理

背景介绍

整网数据比对前，需要先检查并去除训练脚本内部使用到的随机处理，避免由于输入数据不一致导致数据比对结果不可用。

操作方法

修改训练脚本，去除使用到的随机处理：

```
# 此处给出一些典型示例，需要根据自己的脚本进行排查
# 1. 对输入数据做shuffle操作
dataset = tf.data.TFRecordDataset(tf_data)
dataset = dataset.shuffle(batch_size*10)    # 直接注释掉该行

# 2. 使用dropout
net = slim.dropout(net, keep_prob=dropout_keep_prob, scope='Dropout_1b') # 建议注释该行

# 3. 图像预处理使用随机的操作(根据实际情况注释，或者替换成其他固定的预处理操作)
# Random rotate
random_angle = tf.random_uniform([], - self.degree * 3.141592 / 180, self.degree * 3.141592 / 180)
image = tf.contrib.image.rotate(image, random_angle, interpolation='BILINEAR')
depth_gt = tf.contrib.image.rotate(depth_gt, random_angle, interpolation='NEAREST')

# Random flipping
do_flip = tf.random_uniform([], 0, 1)
image = tf.cond(do_flip > 0.5, lambda: tf.image.flip_left_right(image), lambda: image)
depth_gt = tf.cond(do_flip > 0.5, lambda: tf.image.flip_left_right(depth_gt), lambda: depth_gt)

# Random crop
mage_depth = tf.concat([image, depth_gt], 2)
image_depth_cropped = tf.random_crop(image_depth, [self.params.height, self.params.width, 4])

# 其他.....
```

验证方法

修改完训练脚本后，有两种检查方法，验证所有的随机处理是否已经规避掉：

1. NPU训练两次，进行整网精度比对， 检查精度数据的余弦相似度是否大于0.98。具体方法为：
   a. **基于NPU Dump精度数据**。
   b. 使用**vc -lt [left_path] -rt [right_path] -g [graph]**命令比较，生成csv文件。

      PrecisionTool > vc -lt /path/left -rt /path/right
   c. 使用**vcs -f [file_name] -c [cos_sim_threshold] -l [limit]**命令，如果余弦相似度大于0.98，表明已经去随机。

      PrecisionTool > vcs
2. GPU或CPU训练两次，根据npy名字比较tensor数据相似度，相似度结果达到一定阈值，表明已经去随机。

9 性能调优

9.1 混合精度训练

混合精度简介

混合精度为业内通用的性能提升方式，通过降低部分计算精度提升数据计算的并行度。混合精度训练方法是通过混合使用float16和float32数据类型来加速深度神经网络训练的过程，并减少内存使用和存取，从而可以训练更大的神经网络，同时又能基本保持使用float32训练所能达到的网络精度。

用户可以在脚本中通过配置“precision_mode”参数或者“precision_mode_v2”参数开启混合精度。例如：

- precision_mode参数配置为allow_mix_precision_fp16/allow_mix_precision。
- precision_mode_v2参数配置为mixed_float16。

说明

“precision_mode”参数与precision_mode_v2参数不能同时使用，建议使用“precision_mode_v2”参数。

“precision_mode”与“precision_mode_v2”参数的详细说明可参见**精度调优**。

开启“自动混合精度”的场景下，推荐使用LossScale优化器（LossScale优化器的迁移请参见**6.5 替换LossScaleOptimizer**），从而补偿降低精度带来的精度损失；若后续进行Profiling数据进行分析时，发现需要手工调整某些算子的精度模式，可以参考**修改混合精度黑白名单**自行指定哪些算子允许降精度，哪些算子不允许降精度。

精度模式设置

下面以将“precision_mode_v2”参数配置为“mixed_float16”为例，说明如何设置混合精度模式。

修改训练脚本，在初始化NPU设备前通过添加**精度调优**中的“precision_mode_v2”参数设置精度模式。

```
import npu_device as npu
npu.global_options().precision_mode_v2 = 'mixed_float16'  # 开启自动混合精度功能，表示混合使用float16和float32数据类型来处理神经网络的过程
npu.open().as_default()
```

修改混合精度黑白名单

开启自动混合精度的场景下，系统会自动根据内置的优化策略，对网络中的某些数据类型进行降精度处理，从而在精度损失很小的情况下提升系统性能并减少内存使用。

内置优化策略在“OPP安装目录/opp/built-in/op_impl/ai_core/tbe/config/<soc_version>/aic-<soc_version>-ops-info.json”，例如：

```
"Conv2D":{
    "precision_reduce":{
        "flag":"true"
},
```

- precision_mode_v2配置为mixed_float16，precision_mode配置为allow_mix_precision_fp16/allow_mix_precision的场景：
  - 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到float16。
  - 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到float16，相应算子仍使用float32精度。
  - 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。

用户可以在内置优化策略基础上进行调整，自行指定哪些算子允许降精度，哪些算子不允许降精度。

- （推荐）通过modify_mixlist参数指定混合精度黑白灰算子名单

  修改训练脚本，在初始化NPU设备前通过添加**精度调优**中的“modify_mixlist”参数指定混合精度黑白灰算子名单配置文件，配置示例如下：

  ```
  import npu_device as npu
  npu.global_options().modify_mixlist = "/home/test/ops_info.json"
  npu.open().as_default()
  ```

  其中ops_info.json为混合精度黑白灰算子名单配置文件，多个算子使用英文逗号分隔，样例如下：

  ```
  {
    "black-list": {                 // 黑名单
      "to-remove": [                // 黑名单算子转换为灰名单算子
      "Xlog1py"
      ],
      "to-add": [                   // 白名单或灰名单算子转换为黑名单算子
      "Matmul",
      "Cast"
      ]
    },
    "white-list": {                 // 白名单
      "to-remove": [                // 白名单算子转换为灰名单算子
      "Conv2D"
      ],
      "to-add": [                   // 黑名单或灰名单算子转换为白名单算子
      "Bias"
      ]
    }
  }
  ```


假设算子A默认在白名单中，如果您希望将该算子配置为黑名单算子，可以参考如下方法：

a. （正确示例）用户将该算子添加到黑名单中：

```
{
 "black-list": {
   "to-add": ["A"]
 }
}
```

则系统会将该算子从白名单中删除，并添加到黑名单中，最终该算子在黑名单中。

b. （正确示例）用户将该算子从白名单中删除，同时添加到黑名单中：

```
{
 "black-list": {
   "to-add": ["A"]
 },
 "white-list": {
   "to-remove": ["A"]
 }
}
```

则系统会将该该算子从白名单中删除，并添加到黑名单中，最终该算子在黑名单中。

c. （错误示例）用户将该算子从白名单中删除，此时算子最终是在灰名单中，而不是黑名单。

```
{
 "white-list": {
   "to-remove": ["A"]
 }
}
```

此时，系统会将该算子从白名单中删除，然后添加到灰名单中，最终该算子在灰名单中。

说明

对于只从黑/白名单中删除，而不添加到白/黑名单的情况，系统会将该算子添加到灰名单中。

- 修改算子信息库

须知

对内置算子信息库进行修改，可能会对其他网络造成影响，请谨慎修改。

a. 切换到“OPP安装目录/opp/built-in/op_impl/ai_core/tbe/config/<soc_version>”目录下。

b. 对aic-<soc_version>-ops-info.json文件增加写权限。

```
chmod u+w aic-<soc_version>-ops-info.json
```

当前目录下的所有json文件都会被加载到算子信息库中，如果您需要备份原来的json文件，建议备份到其他目录下。

c. 修改或增加算子信息库aic-<soc_version>-ops-info.json文件中对应算子的precision_reduce字段。

9.2 替换 GELU 激活函数

GELU是神经网络中一种常见的激活函数，全称为“Gaussian Error Linear Unit”，是RELU的一种平滑版本， 具体可以参考**论文解释**。Tensorflow中GELU对应的实现接口

为tf.nn.gelu和tf.keras.activations.gelu, 部分网络如BERT中也会使用自定义的GELU实现.。使用近似的实现（如将tf.nn.gelu的approximate参数设置为True，或者BERT中的自定义GELU实现）在训练时可以得到更好的性能。NPU上也提供了高性能的GELU近似实现接口**11.1.1.10 npu.ops.gelu**，将GELU在下沉执行时获取更好的性能。

如果你要使用NPU上GELU接口，请注意以下两点：

1. 只有在function模式下，下沉到NPU执行时调用npu.ops.gelu才会获取性能收益；在eager模式下，npu.ops.gelu不会得到比TensorFlow原生接口更好的性能。
2. NPU提供的GELU接口是近似实现，并不保证在所有场景下都能替换标准实现的GELU接口并达到收敛，需要针对具体的网络实现进行尝试。

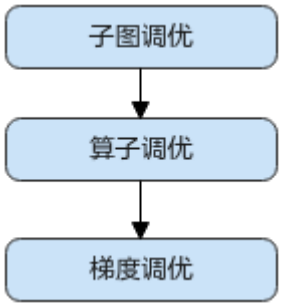
鉴于以上两点，目前自动迁移工具不会自动迁移所有GELU接口，如果你仍希望通过替换GELU激活函数获取性能提升，请参考以下步骤：

1. 引入npu_device模块
   ```
   import npu_device as npu
   ```
2. （function模式下）找到网络中定义或者使用GELU接口的地方，替换为npu_device.ops.gelu：
   ```
   def gelu(x):
    """Gaussian Error Linear Unit
    This is a smoother version of the RELU.
    Original paper: https://arxiv.org/abs/1606.08415
    Args:
    x: float Tensor to perform activation
    Returns:
    `x` with the GELU activation applied
    """
    return npu.ops.gelu(x)
    # return tf.keras.activations.gelu(x, approximate=True)
   ```

9.3 AOE 自动调优

AOE自动调优工具通过生成调优策略、编译、在运行环境上验证的闭环反馈机制，不断迭代出更优的调优策略，最终得到最佳的调优策略，从而可以更充分利用硬件资源，不断提升网络的性能，达到最优的效果。模型训练阶段，分别使能AOE工具进行子图/算子与梯度切分的调优，调优完成后，最优调测策略会固化到知识库，模型再次训练时，无需开启调优，即可以享受知识库带来的性能收益。

建议按照如下调优顺序使用AOE工具进行调优：



训练场景下使能AOE调优有两种方式：

- 设置环境变量
  ```
  # 1：子图调优 2：算子调优 4：梯度调优
  export AOE_MODE=2
  ```
- 修改训练脚本，在初始化NPU设备前通过添加**AOE**中的“aoe_mode”参数指定调优模式：


```
import npu_device as npu
npu.global_options().aoe_config.aoe_mode="1"
npu.open().as_default()
```

关于AOE工具的使用约束及更多功能介绍，请参见《**AOE工具使用指南**》的“TensorFlow训练场景下调优”。

9.4 调整梯度切分策略

背景介绍

分布式训练场景下，各个Device之间计算梯度后执行梯度聚合操作。由于梯度数据会按顺序产生，且产生后不会再变化，为了提高训练性能，我们可以对梯度参数数据进行切分，同一分段中的梯度数据在产生后可以立即开始梯度聚合，使得一部分梯度参数数据聚合和前后向时间并行执行。

系统默认切分策略是：按照梯度数据量切分为两段，第一段梯度数据量为96.54%，第二段梯度数据量为3.46%（部分情况可能出现为一段情况）。因不同网络梯度数据量、梯度计算时间差异，此种切分方式可能不适用于其他的网络，用户可以参考本节内容调整分布式梯度切分策略，从而提升分布式场景下的训练性能。

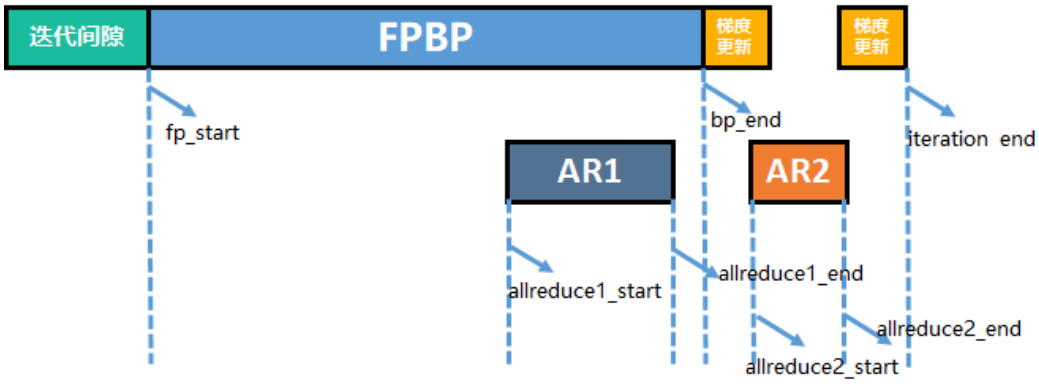
确定梯度切分策略

用户需要结合Profiling工具分析训练过程的迭代轨迹数据（Training Trace），确定梯度切分策略需要调整到什么值以达到分布式场景下训练性能提升的目标。

说明

相关介绍请参考《**性能分析工具使用指南**》。

迭代轨迹数据即训练任务及AI软件栈的软件信息，实现对训练任务的性能分析。以默认的两段式梯度切分为例，通过打印出训练任务中关键节点fp_start/bp_end/allreduce1_start/allreduce1_end/allreduce2_start/allreduce2_end/Iteration_end的时间戳，达到把一个迭代的执行情况描述清楚的目的。



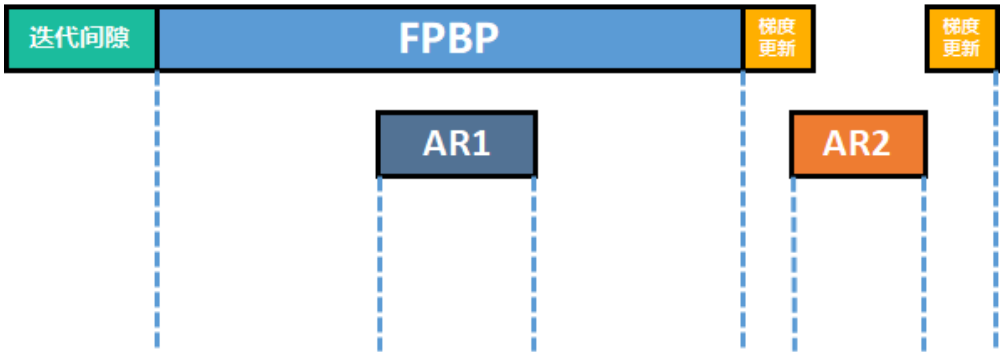
一个较优的梯度数据切分原则为：

- AR1隐藏在FPBP之间。默认情况下Allreduce和前后向串行执行，此时需要配置**hcom_parallel**为True开启Allreduce和前后向并行执行，后面代码示例“**调整梯度切分策略**”中会介绍配置方法。
- AR2的时间尽可能短，从而减少计算后因为集合通信而带来的拖尾时间的消耗。

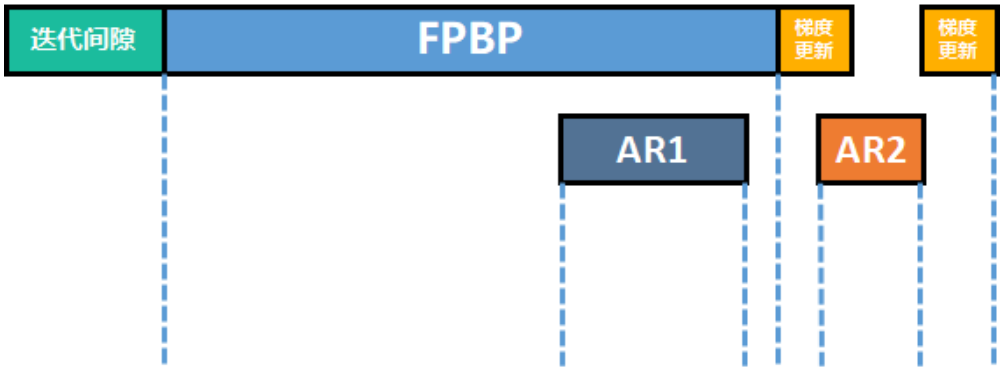
以上述切分原则为依据，用户可以调整梯度切分策略，从而提升分布式场景下的训练性能。下面以两段式梯度切分为例，结合三种优化场景，帮助用户理解如何确定梯度切分策略。

【优化场景1】AR1开始时间较早，AR2时间较长，这种情况下可以将切分点往后设置，从而尽可能缩短AR2的时间。

例如原始设置为第一段梯度数据量为50%，第二段梯度数据量为50%：

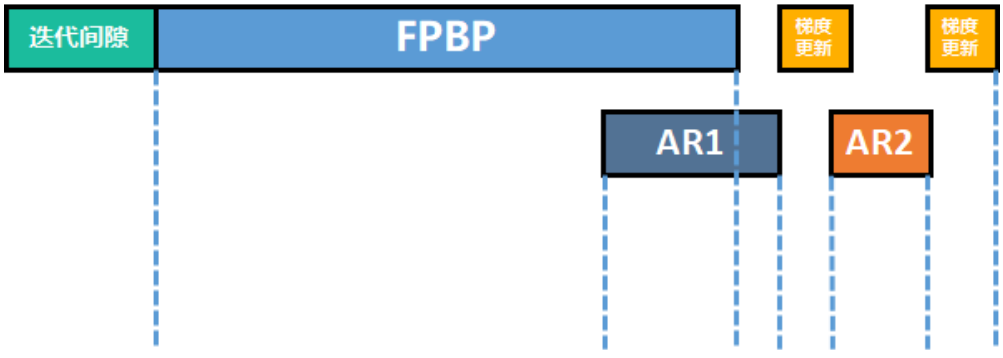


可以修改为第一段梯度数据量为80%，第二段梯度数据量为20%：

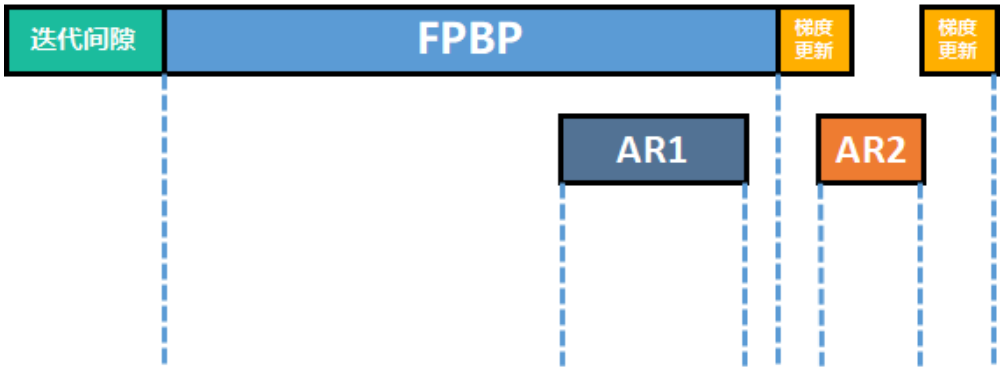


【优化场景2】AR1开始时间较晚，AR1时间超出了FPBP的时间，这种情况下，可以将切分点往前设置，从而将AR1隐藏在FPBP之间。

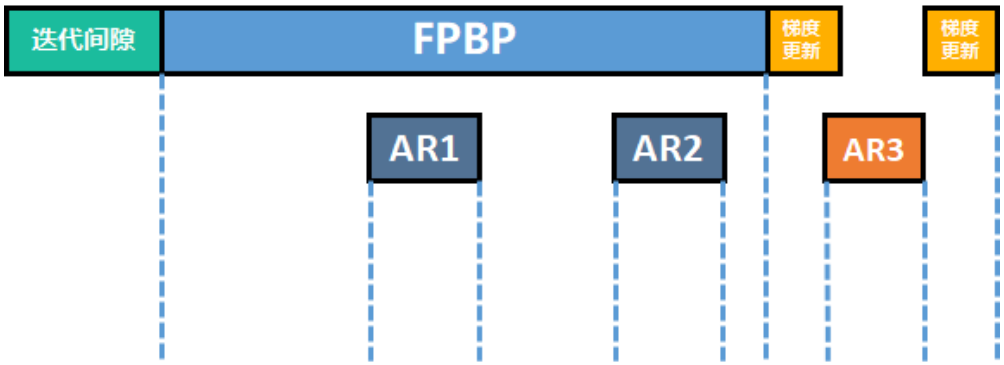
例如原始设置为第一段梯度数据量为90%，第二段梯度数据量为10%：



可以修改为第一段梯度数据量为80%，第二段梯度数据量为20%：



【优化场景3】FPBP数据量较大，计算时间较长，两段式梯度切分的情况下，如果把大部分梯度数据放到AR1中会导致拖尾时间很长，参见优化场景2；如果把大部分的梯度数据放到AR2中会导致AR2时间很长，参见优化场景1。但FPBP中还有较多的时间可以利用，此时可以新增切分段数，使更多的集合通信时间和FPBP并行起来。



调整梯度切分策略

用户可以在训练脚本中调用梯度切分类接口来设置反向计算阶段的allreduce切分融合策略，以下接口二选一使用。

set_split_strategy_by_idx：基于梯度的索引id，在集合通信group内设置反向梯度切分策略。

```
from hccl.split.api import set_split_strategy_by_idx
set_split_strategy_by_idx([20, 100, 159])
```

set_split_strategy_by_size：基于梯度数据量百分比，在集合通信group内设置反向梯度切分策略。

```
from hccl.split.api import set_split_strategy_by_size
set_split_strategy_by_size([60, 20, 20])
```


10 样例参考

10.1 自动迁移与训练

简介

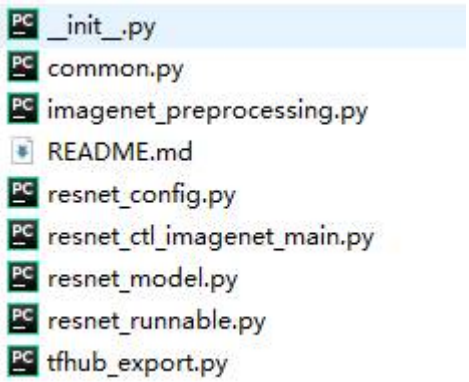
下面介绍如何通过工具迁移ResNet50网络。

下载原始模型和数据集

步骤1 从github下载ResNet50原始模型。

git clone -b r2.6.0 https://github.com/tensorflow/models.git

假设原始代码下载到了/root/models目录下，用户可以在/root/models/official/vision/image_classification/resnet/下查看到下载好的脚本：



步骤2 下载数据集。

参考**LINK**的使用说明，下载imagenet2012数据集并使用imagenet_to_gcs.py脚本转换为TFRecord。

将处理好的数据集放到/root/models/data/imagenet_TF/路径下。

----结束

使用迁移工具进行模型迁移

步骤1 阅读**使用限制**，需要在工具迁移前手工添加数据集分片操作：

```
dataset = tf.data.Dataset.from_tensor_slices(filenames)
import npu_device as npu
# NPU添加的shard逻辑，会根据集群数量，对数据集和全局batch进行切分
dataset, batch_size = npu.distribute.shard_and_rebatch_dataset(dataset, batch_size)
#if input_context:
#  logging.info(
#      'Sharding the dataset: input_pipeline_id=%d num_input_pipelines=%d',
#      input_context.input_pipeline_id, input_context.num_input_pipelines)
#  dataset = dataset.shard(input_context.num_input_pipelines,
#                          input_context.input_pipeline_id)
```

步骤2 在运行环境上安装工具依赖。

pip3 install pandas

pip3 install openpyxl

pip3 install google_pasta

步骤3 执行命令进行工具自动迁移。

1. 进行迁移工具所在目录。

   cd <tfplugin安装目录>/tfplugin/latest/python/site-packages/npu_device/convert_tf2npu/

2. 进行脚本迁移。
   - 若后续执行单Device训练，执行如下命令：

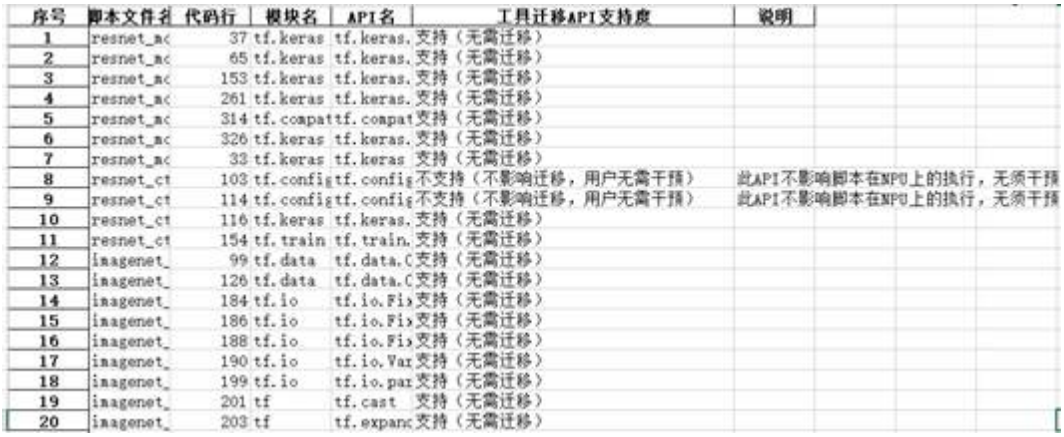
     python3 main.py -i /root/models/official/vision/image_classification/resnet/ -o /root/models/resnet50/ -r /root/models/resnet50/ –m /root/models/official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py
   - 若后续需要执行分布式训练，执行如下命令：

     python3 main.py -i /root/models/official/vision/image_classification/resnet/ -o /root/models/resnet50/ -r /root/models/resnet50/ –m /root/models/official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py -d tf_strategy

     其中“-d”代表原始脚本使用的分布式策略，“tf_strategy”表示原始脚本使用的是tf.distribute.Strategy分布式策略。

步骤4 在**/root/models/resnet50/report_npu_*****下查看迁移报告。

打开api_analysis_report.xlsx，查看ResNet50网络中的API支持度情况：



序号	脚本文件名	代码行	模块名	API名	工具迁移API支持度	说明
1	resnet_mc	37	tf.keras	tf.keras.	支持（无需迁移）	
2	resnet_mc	65	tf.keras	tf.keras.	支持（无需迁移）	
3	resnet_mc	153	tf.keras	tf.keras.	支持（无需迁移）	
4	resnet_mc	261	tf.keras	tf.keras.	支持（无需迁移）	
5	resnet_mc	314	tf.compat	tf.compat	支持（无需迁移）	
6	resnet_mc	326	tf.keras	tf.keras.	支持（无需迁移）	
7	resnet_mc	33	tf.keras	tf.keras	支持（无需迁移）	
8	resnet_ct	103	tf.config	tf.config	不支持（不影响迁移，用户无需干预）	此API不影响脚本在NPU上的执行，无须干预
9	resnet_ct	114	tf.config	tf.config	不支持（不影响迁移，用户无需干预）	此API不影响脚本在NPU上的执行，无须干预
10	resnet_ct	116	tf.keras	tf.keras.	支持（无需迁移）	
11	resnet_ct	154	tf.train	tf.train.	支持（无需迁移）	
12	imagenet_	99	tf.data	tf.data.(	支持（无需迁移）	
13	imagenet_	126	tf.data	tf.data.(	支持（无需迁移）	
14	imagenet_	184	tf.io	tf.io.Fi	支持（无需迁移）	
15	imagenet_	186	tf.io	tf.io.Fi	支持（无需迁移）	
16	imagenet_	188	tf.io	tf.io.Fi	支持（无需迁移）	
17	imagenet_	190	tf.io	tf.io.Var	支持（无需迁移）	
18	imagenet_	199	tf.io	tf.io.par	支持（无需迁移）	
19	imagenet_	201	tf	tf.cast	支持（无需迁移）	
20	imagenet_	203	tf	tf.expanc	支持（无需迁移）	

筛选“API支持度”这一列，发现所有接口分为如下几类：

- 支持：此类API在昇腾AI处理器上绝对支持，无需适配修改。
- 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。
- 不支持（无迁移方案，建议不使用）：此类API在昇腾AI处理器上不支持，且当前暂无具体迁移方案，建议您不要使用，否则会引起训练失败。

步骤5 在**/root/models/resnet50/output_npu_*****下查看迁移后的脚本。

将原始脚本文件夹重命名，例如将/root/models/official/vision/image_classification/resnet重命名为resnet_org

由于导入库文件依赖原始文件夹结构，要将迁移后的/root/models/resnet50/resnet_npu_***重命名为resnet，并拷贝回原始目录

cp -r /root/models/resnet50/ resnet_npu_* /root/models/official/vision/image_classification/resnet**

----结束

执行单 Device 训练

步骤1 由于原始脚本支持分布式训练，迁移后的脚本中使用了HCCL集合通信接口，则需要在单Device上执行训练前准备单Device的资源信息配置文件。否则请跳过此步。（本示例以配置文件的方式设置资源信息，您也可以参见**7.2.3 训练执行（环境变量方式设置资源信息）**通过环境变量的方式设置资源信息。）

单Device的资源信息配置文件中需包含一个Device资源，文件名举例：rank_table_1p.json，配置文件举例：

```
{
"server_count":"1",
"server_list":
[
  {
      "device":[
                  {
                   "device_id":"0",
                   "device_ip":"192.168.1.8",
                   "rank_id":"0"
                   }
              ],
       "server_id":"10.0.0.10"
   }
],
```

```
"status":"completed",
"version":"1.0"
}
```

配置文件的详细介绍请参考**7.2.2.1 准备ranktable资源配置文件**。

步骤2 配置训练进程启动依赖的环境变量。

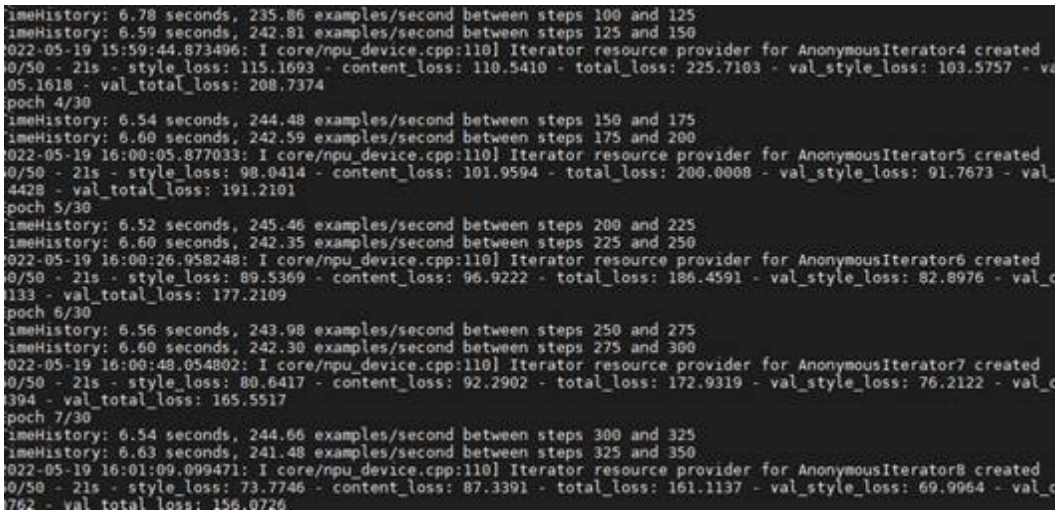
安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行. **${install_path}/set_env.sh**命令设置环境变量。并进行如下配置：

```
# 请依据实际在下列场景中选择一个训练依赖包安装路径的环境变量设置（以HwHiAiUser安装用户为例）。
# 场景一：昇腾设备安装部署开发套件包Ascend-cann-toolkit(此时开发环境可进行训练任务)。
. /home/HwHiAiUser/Ascend/ascend-toolkit/set_env.sh
# 场景二：昇腾设备安装部署软件包Ascend-cann-nnae。
. /home/HwHiAiUser/Ascend/nnae/set_env.sh

# tfplugin包依赖。
. /home/HwHiAiUser/Ascend/tfplugin/set_env.sh

# 若运行环境中存在多个python3版本时，需要在环境变量中配置python的安装路径。如下配置以安装python3.7.5为例，可根据实际修改。
export PATH=/usr/local/python3.7.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH

# 当前脚本所在路径，例如：
export PYTHONPATH="$PYTHONPATH:/root/models"

export JOB_ID=10086       # 训练任务ID，用户自定义，仅支持大小写字母，数字，中划线，下划线。不建议使用以0开始的纯数字
export ASCEND_DEVICE_ID=0  # 指定昇腾AI处理器的逻辑ID，单P训练也可不配置，默认为0，在0卡执行训练
export RANK_ID=0          # 指定训练进程在集合通信进程组中对应的rank标识序号，单P训练固定配置为0
export RANK_SIZE=1        # 指定当前训练进程对应的Device在本集群大小，单P训练固定配置为1
export RANK_TABLE_FILE=/root/rank_table_1p.json # 如果用户原始训练脚本中使用了hvd接口或tf.data.Dataset对象的shard接口，需要配置，否则无需配置。
```

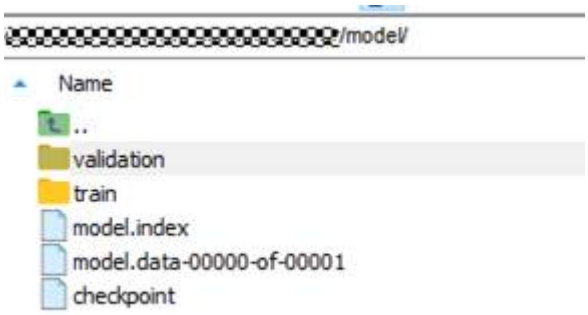
步骤3 执行训练脚本拉起训练进程：

python3 /root/models/adain/adain.py --drop_remainder=True

步骤4 检查训练是否跑通。



步骤5 训练结束后，在/root/models/adain/model下生成checkpoint文件。



----结束

执行分布式训练

下面以两个Device为例，说明如何使用迁移后的脚本在昇腾AI处理器上执行分布式训练。

步骤1 准备包含两个Device的昇腾AI处理器资源信息配置文件，文件名举例：rank_table_2p.json，配置文件举例：

```
{
"server_count":"1",
"server_list":
[
  {
      "device":[
                  {
                   "device_id":"0",
                   "device_ip":"192.168.1.8",
                   "rank_id":"0"
                   },
                   {
                    "device_id":"1",
                    "device_ip":"192.168.1.9",   // 两个Device需要处于同一网段，0卡和1卡为同一网段
                    "rank_id":"1"
                    }
              ],
       "server_id":"10.0.0.10"
   }
],
"status":"completed",
"version":"1.0"
}
```

配置文件的详细介绍请参考**7.2.2.1 准备ranktable资源配置文件**。

步骤2 在不同的shell窗口依次拉起不同的训练进程。

拉起训练进程0：

安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行**. ${install_path}/set_env.sh**命令设置环境变量。并进行如下配置：

```
export PYTHONPATH="$PYTHONPATH:/root/models"
export ASCEND_DEVICE_ID=0
export RANK_ID=0
export RANK_SIZE=2
export RANK_TABLE_FILE=/home/test/rank_table_2p.json
python3 /root/models/official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py
```

拉起训练进程1：

安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行. **${install_path}/set_env.sh**命令设置环境变量。并进行如下配置：

```
export PYTHONPATH="$PYTHONPATH:/root/models"
export ASCEND_DEVICE_ID=1
export RANK_ID=1
export RANK_SIZE=2
export RANK_TABLE_FILE=/home/test/rank_table_2p.json
python3 /root/models/official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py
```

说明

除了以上方式，您还可以通过自定义启动脚本通过循环方式依次拉起多个训练进程，请参考**样例链接**。

----结束

10.2 手工迁移与训练

10.2.1 下载 TF2 官方 Resnet50

首先，我们下载TensorFlow的官方models仓并check out到v2.6.0的tag版本：

git clone https://github.com/tensorflow/models.git

cd models

git checkout v2.6.0

说明

为保证验证结果一致，用户必须切换到v2.6.0的tag版本。切换完成后，我们开始依次分析迁移点并进行迁移，我们假定您对该脚本逻辑及执行参数有基本的了解。

10.2.2 添加@tf.function 装饰

我们分析入口文件official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py，可以看到官方脚本已经默认添加了@tf.function装饰，因而这个迁移点我们直接迁移完成。

```
flags.DEFINE_boolean(name='use_tf_function', default=True,
                     help='Wrap the train and test step inside a '
                     'tf.function.')
```

10.2.3 设置 NPU 为默认设备

我们在入口文件official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py的开头添加注册如下代码，添加完成后该迁移点即完成。

```
import npu_device as npu
npu.open().as_default()
```

10.2.4 替换 LossScaleOptimizer

脚本分析后，我们发现该脚本并未使用LossScaleOptimizer，因而不需要替换，这个迁移点直接完成。

10.2.5 预处理 batch 动作设置 drop_remainder

跟随训练脚本逻辑，找到数据预处理文件：official/vision/image_classification/resnet/imagenet_preprocessing.py。

在数据读取函数“input_fn”内部设置drop_remainder为True，该迁移点完成。

```
def input_fn(is_training,
             data_dir,
             batch_size,
             dtype=tf.float32,
             datasets_num_private_threads=None,
             parse_record_fn=parse_record,
             input_context=None,
             drop_remainder=False,
             tf_data_experimental_slack=False,
             training_dataset_cache=False,
             filenames=None):
......

Returns:
  A dataset that can be used for iteration.
"""
drop_remainder=True

if filenames is None:
  filenames = get_filenames(is_training, data_dir)
dataset = tf.data.Dataset.from_tensor_slices(filenames)
```

10.2.6 设置 NPU 上的循环次数

在确定该是否涉及该适配点前，需要您了解脚本是否采用了**循环下沉的编码方式**，实际上，阅读官方的脚本发现已经开放了开关供用户选择是否循环下沉。

从official/vision/image_classification/resnet/common.py可以看到官方脚本提供了两个入参：

- steps_per_loop传入training loop的大小，可以从注释中看出，在循环中间，只有训练的动作，不会执行任何callback之类的附加操作。
- use_tf_while_loop则决定是否循环下沉，默认为True，即trainning loop默认都会以While算子的形式执行。

```
flags.DEFINE_integer(
    name='steps_per_loop',
    default=None,
    help='Number of steps per training loop. Only training step happens '
    'inside the loop. Callbacks will not be called inside. Will be capped at '
    'steps per epoch.')
flags.DEFINE_boolean(
    name='use_tf_while_loop',
    default=True,
    help='Whether to build a tf.while_loop inside the training loop on the '
    'host. Setting it to True is critical to have peak performance on '
    'TPU.')
```

所以，按照默认值，我们需要设置NPU_LOOP_SIZE环境变量的值与steps_per_loop一致。此环境变量的设置说明可参见**10.2.7 启动单卡训练**。

10.2.7 启动单卡训练

我们暂时略过了**6.7 分布式训练脚本适配（兼容单卡）**的分布式适配，来初步验证下单卡迁移的结果。

首先，我们需要确定启动脚本的参数，为了以CPU单卡形式启动脚本，我们传入distribution_strategy的策略为one_device。

在启动前，我们需要按照official/vision/image_classification/resnet/README.md中的说明，将models路径设置到PYTHONPATH中，例如当前所在目录是/path/to/models，则应当设置环境变量：

```
export PYTHONPATH=$PYTHONPATH:/path/to/models
```

训练中，我们通常每次循环下沉执行一个epoch数据的训练，steps_per_loop值则应当设置为样本总数除以batch大小的结果，为了快速验证功能，我们假定样本总数为64，训练的batch大小为2，同时跳过eval过程。所以此时steps_per_loop的大小为64/2=32，因此我们需要设置环境变量export NPU_LOOP_SIZE=32。最终我们的启动参数如下（其中**/path/to/imagenet_TF/需要替换为您的数据集路径**），需要注意的是，通常您应当以epoch为单位组织训练，这里入参中写入train_steps是为了使训练尽快结束进行基本的功能验证。

```
cd official/vision/image_classification/resnet/
export PYTHONPATH=$PYTHONPATH:/path/to/models
export NPU_LOOP_SIZE=32
python3 resnet_ctl_imagenet_main.py \
--data_dir=/path/to/imagenet_TF/ \
--train_steps=128 \
--distribution_strategy=one_device \
--use_tf_while_loop=true \
--steps_per_loop=32 \
--batch_size=2 \
--epochs_between_evals=1 \
--skip_eval
```

执行该命令后，脚本在NPU上训练完成。

10.2.8 关键日志说明

NPU上的单卡训练将很快完成，下面对执行日志的关键位置进行说明。

关键日志 1：NPU 及初始化配置项及初始化成功打印

这部分日志可以看到NPU初始化时的配置项（如果您修改了npu.global_options，详细参数可参见11.1.1.2.1 简介，这里也会有所体现）以及初始化成功信息，由于我们调用11.1.1.1 npu.open时没有传入任何参数，所以默认在NPU:0上进行了初始化。

```
I core/npu_wrapper.cpp:120] Start graph engine with options:
I core/npu_wrapper.cpp:122]   ge.exec.deviceId:0
I core/npu_wrapper.cpp:122]   ge.exec.dumpMode:output
I core/npu_wrapper.cpp:122]   ge.exec.enableDump:0
I core/npu_wrapper.cpp:122]   ge.exec.enableDumpDebug:0
I core/npu_wrapper.cpp:122]   ge.exec.enable_exception_dump:0
I core/npu_wrapper.cpp:122]   ge.exec.isTailingOptimization:0
I core/npu_wrapper.cpp:122]   ge.exec.precision_mode:allow_fp32_to_fp16
I core/npu_wrapper.cpp:122]   ge.exec.profilingMode:0
I core/npu_wrapper.cpp:122]   ge.exec.variable_acc:1
I core/npu_wrapper.cpp:122]   ge.graphRunMode:1
I core/npu_wrapper.cpp:122]   ge.hcomParallel:0
I core/npu_wrapper.cpp:122]   ge.opDebugLevel:0
I core/npu_wrapper.cpp:122]   ge.opSelectImplmode:high_performance
I core/npu_wrapper.cpp:122]   ge.op_compiler_cache_mode:disable
I core/npu_wrapper.cpp:128] Start graph engine succeed
I core/npu_wrapper.cpp:133] Start tensorflow model parser succeed
I core/npu_wrapper.cpp:154] Create device instance /job:localhost/replica:0/task:0/device:NPU:0 with options:
I core/npu_device_register.cpp:86] Npu device instance /job:localhost/replica:0/task:0/device:NPU:0 created
```

关键日志 2：数据预处理 H2D 线程开启及 HDC 通道创建

这部分日志只有在您使用了Dataset作为预处理Pipeline，并且function的入参是Iterator时才会打印。

从这两条日志上可以看出，TF Adapter启动了预处理H2D线程，同时创建了名为AnonymousIterator0的HDC数据传输通道（该名称与TF2中Iterator的shared_name一致）。

```
I core/npu_device.cpp:204] Iterator resource provider for AnonymousMultiDeviceIterator0 created
I core/npu_device.cpp:204] Iterator resource provider for AnonymousIterator0 created

I kernels/iterator_h2d.cpp:49] Hdc channel for iterator resource AnonymousIterator0 to device [0] created
```

关键日志 3：NPU 检测到循环下沉逻辑，以循环下沉方式执行训练

这个日志的打印内容取决于您是否采用了**循环下沉的编码方式**，本次迁移中，我们采用了该编码方式，所以Graph xxx can loop的判定结果是true，同时设置了NPU上的训练循环大小为32。后两行日志可以看出，开启了总共32次的异步数据传输，然后向NPU设备下发了执行32次训练的请求。

```
I core/npu_device.cpp:1058] Graph __inference_loop_fn_14234 can loop: true
I core/npu_device.cpp:1427] Set npu loop size to 32
I core/npu_device.cpp:1534] Start consume iterator resource AnonymousIterator0 32 times
I core/npu_device.cpp:1554] Start run ge graph 453 pin to cpu, loop size 32
I kernels/iterator_h2d.cpp:49] Hdc channel for iterator resource AnonymousIterator0 to device [0] created
```

关键日志 4：训练执行过程

这部分日志表示训练过程正在执行。

```
I core/npu_device.cpp:1543] Iterator resource AnonymousIterator0 consume 32 times done with status OK
3133748096 controller.py:427] train | step:     32 | steps/sec:    0.5 | output: {'train_accuracy': 0.0, 'train_loss': 12.313879}
s/sec:    0.5 | output: {'train_accuracy': 0.0, 'train_loss': 12.313879}
I core/npu_device.cpp:1534] Start consume iterator resource AnonymousIterator0 32 times
I core/npu_device.cpp:1554] Start run ge graph 453 pin to cpu, loop size 32
I core/npu_device.cpp:1543] Iterator resource AnonymousIterator0 consume 32 times done with status OK
3133748096 controller.py:427] train | step:     64 | steps/sec:   29.6 | output: {'train_accuracy': 0.0, 'train_loss': 12.308612}
s/sec:   29.6 | output: {'train_accuracy': 0.0, 'train_loss': 12.308612}
I core/npu_device.cpp:1534] Start consume iterator resource AnonymousIterator0 32 times
I core/npu_device.cpp:1554] Start run ge graph 453 pin to cpu, loop size 32
I core/npu_device.cpp:1543] Iterator resource AnonymousIterator0 consume 32 times done with status OK
3133748096 controller.py:427] train | step:     96 | steps/sec:    5.1 | output: {'train_accuracy': 0.0, 'train_loss': 12.417062}
s/sec:    5.1 | output: {'train_accuracy': 0.0, 'train_loss': 12.417062}
I core/npu_device.cpp:1534] Start consume iterator resource AnonymousIterator0 32 times
I core/npu_device.cpp:1554] Start run ge graph 453 pin to cpu, loop size 32
I core/npu_device.cpp:1543] Iterator resource AnonymousIterator0 consume 32 times done with status OK
3133748096 keras_utils.py:152] TimeHistory: 68.30 seconds, 3.75 examples/second between steps 0 and 128
3133748096 controller.py:427] train | step:    128 | steps/sec:   30.5 | output: {'train_accuracy': 0.0, 'train_loss': 12.413293}
s/sec:   30.5 | output: {'train_accuracy': 0.0, 'train_loss': 12.413293}
```

其中，下面的日志表示32次异步的数据传输成功完成，如果数据传输过程中出现错误，您也将看到相关打印。

```
I core/npu_device.cpp:1543] Iterator resource AnonymousIterator0 consume 32 times done with status OK
```

关键日志 5：训练进程退出

训练结束后，进程退出，会依次销毁HDC数据传输线程（如果存在），关闭Graph Engine引擎。

```
I core/npu_device.cpp:199] Stopping iterator resource provider for AnonymousIterator0
I core/npu_device.cpp:199] Stopping iterator resource provider for AnonymousMultiDeviceIterator0
I core/npu_hdc.cpp:279] Hdc channel AnonymousIterator0 closed
I core/npu_wrapper.cpp:171] Stop tensorflow model parser succeed
I core/npu_wrapper.cpp:180] Stop graph engine succeed
```

10.2.9 分布式适配

在单卡模式下已经成功完成迁移，我们开始分布式模式的迁移，分布式迁移并不影响单卡功能，您的分布式脚本和单卡脚本最终是同一份脚本，可以同时以单卡或分布式方式执行。我们依次按照**分布式迁移**的过程开始迁移。

1. **worker间变量初值同步**

根据脚本逻辑找到模型创建完成的位置official/vision/image_classification/resnet/resnet_ctl_imagenet_main.py，添加可训练变量的同步操作，这里需要使用**11.1.1.4 npu.distribute.broadcast**接口。

```
with distribute_utils.get_strategy_scope(strategy):
  # 模型创建
  runnable = resnet_runnable.ResnetRunnable(flags_obj, time_callback, per_epoch_steps)
# 变量同步
npu.distribute.broadcast(runnable.model.trainable_variables)
```

2. **worker间梯度聚合**

   根据脚本逻辑，我们找到训练过程中梯度更新的部分official/vision/image_classification/resnet/resnet_runnable.py

```
def train_step(self, iterator):
  """See base class."""

  def step_fn(inputs):
    """Function to run on the device."""
    images, labels = inputs
    with tf.GradientTape() as tape:
      logits = self.model(images, training=True)

      prediction_loss = tf.keras.losses.sparse_categorical_crossentropy(
          labels, logits)
      loss = tf.reduce_sum(prediction_loss) * (1.0 /
                                               self.flags_obj.batch_size)
      num_replicas = self.strategy.num_replicas_in_sync
      l2_weight_decay = 1e-4
      if self.flags_obj.single_l2_loss_op:
        l2_loss = l2_weight_decay * 2 * tf.add_n([
            tf.nn.l2_loss(v)
            for v in self.model.trainable_variables
            if 'bn' not in v.name
        ])

        loss += (l2_loss / num_replicas)
      else:
        loss += (tf.reduce_sum(self.model.losses) / num_replicas)

    grad_utils.minimize_using_explicit_allreduce(
        tape, self.optimizer, loss, self.model.trainable_variables)
    self.train_loss.update_state(loss)
    self.train_accuracy.update_state(labels, logits)
```

这里可以看出，TF2原始脚本中，使用了函数minimize_using_explicit_allreduce来屏蔽部署形态，进入函数内部，可以找到实际执行梯度聚合的函数在：official/staging/training/grad_utils.py。

```
def _filter_and_allreduce_gradients(grads_and_vars,
                                    allreduce_precision="float32",
                                    bytes_per_pack=0):
```

这里需要注意，我们要求以**单卡CPU的形式**启动训练，所以，此时代码中的原始梯度聚合行为不生效（或者说因为是单卡，所以不需要聚合），在这个函数内部，我们需要添加NPU上的梯度聚合操作，这里需要用到**11.1.1.3 npu.distribute.all_reduce**接口。我们在official/staging/training/grad_utils.py添加如下信息：

```
# 由于需要使用npu.distribue.all_reduce接口，在脚本开头import npu
import npu_device as npu

def _filter_and_allreduce_gradients(grads_and_vars,
                                    allreduce_precision="float32",
                                    bytes_per_pack=0):
... ...

  # 原始脚本采用SUM策略
  allreduced_grads = tf.distribute.get_strategy(  # pylint: disable=protected-access
```

```
).extended._replica_ctx_all_reduce(tf.distribute.ReduceOp.SUM, grads, hints)
if allreduce_precision == "float16":
  allreduced_grads = [tf.cast(grad, "float32") for grad in allreduced_grads]

# 由于NPU适配添加的梯度聚合操作，聚合类型保持与原始脚本一致，此处选择“sum”聚合策略
allreduced_grads = npu.distribute.all_reduce(allreduced_grads,reduction="sum")

return allreduced_grads, variables
```

3. **不同worker上的数据集分片**

   根据脚本逻辑找到预处理函数official/vision/image_classification/resnet/resnet_runnable.py：

   ```
   # 假数据，忽略该分支
   if self.flags_obj.use_synthetic_data:
     self.input_fn = common.get_synth_input_fn(
         height=imagenet_preprocessing.DEFAULT_IMAGE_SIZE,
         width=imagenet_preprocessing.DEFAULT_IMAGE_SIZE,
         num_channels=imagenet_preprocessing.NUM_CHANNELS,
         num_classes=imagenet_preprocessing.NUM_CLASSES,
         dtype=self.dtype,
         drop_remainder=True)
   else:
   # 真实的预处理方法
     self.input_fn = imagenet_preprocessing.input_fn
   ```

   找到official/vision/image_classification/resnet/imagenet_preprocessing.py，添加如下信息：

   ```
   # 由于需要使用npu.distribue.shard_and_rebatch接口，在脚本开头import npu
   import npu_device as npu

   if input_context:
     logging.info(
         'Sharding the dataset: input_pipeline_id=%d num_input_pipelines=%d',
         input_context.input_pipeline_id, input_context.num_input_pipelines)
     # 原始的shard逻辑，因为以单机CPU方式启动，所以不会进行实际的shard
     dataset = dataset.shard(input_context.num_input_pipelines, input_context.input_pipeline_id)
   # NPU添加的shard逻辑，会根据集群数量，对数据集和全局batch进行切分
   dataset, batch_size = npu.distribute.shard_and_rebatch_dataset(dataset, batch_size)
   ```

执行完上述步骤后，分布式迁移完成。

说明

新增的调用对单卡流程没有任何影响，这些接口内部会根据是否设置了NPU分布式执行的环境变量来决定是否生效。如果是单卡模式执行，这些接口不会执行任何操作。

10.2.10 启动分布式训练

我们已经完成了分布式的迁移工作，不过此时您仍然可以使用上面的单卡参数进行单卡训练。

如果要进行分布式训练，还需要一些额外的启动参数调整，需要注意的是，这些调整并非NPU特有的：当您使用多卡训练时，您可以对全局batch size进行等比例的放大，比如您在单卡上执行32 batch size大小的训练，在执行集群大小为8的分布式训练时，batch size大小可以调整为32*8以加速训练。

我们以上面启动单卡的参数为例，单卡batch大小为2，所以8卡训练时将batch大小调整为2*8=16，batch的调整会直接影响每个epoch的训练总步数，您应当清楚这些关联关系，并且在batch大小发生变化时作出调整。假定一个epoch样本总数为64，我们每次次循环下沉处理一个epoch，那么，当batch大小为2时，steps_per_loop设置为64/2=32，表示单卡上训练32步即完成了一个epoch训练。但是当我们使用8卡训练，batch大小调整为16后，steps_per_loop应当设置为64/16=4，此时，单卡上训练4步即完成了一个epoch训练，单纯从步数上可以看出有8倍的性能提升。

清楚这些关系后，我们将单卡的参数进行转换，变为分布式的启动参数，由于要启动多个训练程，可以将启动命令行写入脚本，下面的8卡训练脚本仅供参考，比如我们在train.sh的脚本中写入如下内容：

```
export RANK_TABLE_FILE=/path/to/rank_table.json
export RANK_SIZE=8
export RANK_ID=$1
export ASCEND_DEVICE_ID=$2
export NPU_LOOP_SIZE=4
python3 resnet_ctl_imagenet_main.py \
--data_dir=/path/to/imagenet_TF/ \
--train_steps=16 \
--distribution_strategy=one_device \
--use_tf_while_loop=true \
--steps_per_loop=4 \
--batch_size=16 \
--epochs_between_evals=1 \
--skip_eval
```

说明

- /path/to/rank_table.json替换为符合您部署形态的NPU分布式配置文件。
- /path/to/imagenet_TF/替换为您实际的数据集路径。
- 此样例，我们以通过配置文件（即ranktable文件）的方式配置昇腾AI处理器的资源信息，详细的配置文件说明可参见**7.2.2.1 准备ranktable资源配置文件**。当然，开发者也可以通过环境变量的方式指定昇腾AI处理器的资源信息，可参见**7.2.3 训练执行（环境变量方式设置资源信息）**。

在启动前，我们同样需要按照official/vision/image_classification/resnet/README.md中的说明，将models路径设置到PYTHONPATH中，例如当前的目录是/path/to/models，环境变量示例如下：

```
export PYTHONPATH=$PYTHONPATH:/path/to/models
```

之后，您可以执行如下命令启动8卡NPU训练：

```
nohup bash train.sh 0 0 &
nohup bash train.sh 1 1 &
nohup bash train.sh 2 2 &
nohup bash train.sh 3 3 &
nohup bash train.sh 4 4 &
nohup bash train.sh 5 5 &
nohup bash train.sh 6 6 &
nohup bash train.sh 7 7 &
```


11 接口参考

11.1 TF Adapter 2.6 接口参考

11.1.1 TF Adapter 2.6 接口参考

11.1.1.1 npu.open

函数原型

npu.open(device_id=None).as_default()

功能说明

用于注册NPU设备，当前必须连续调用as_default设置NPU为默认设备

参数说明

参数名	输入/输出	描述
device_id	输入	需要初始化的NPU设备ID，如果不传入，默认读取环境变量11.3.1.2 ASCEND_DEVICE_ID的值，如果ASCEND_DEVICE_ID环境变量未设置，则取值为0。

返回值

返回npu_device的实例。

调用示例

```
import npu_device as npu
npu.open().as_default()
```

11.1.1.2 npu.global_options

11.1.1.2.1 简介

函数原型

npu.global_options()

功能说明

返回NPU设备初始化**全局单例**配置对象，通过修改该全局单例对象的属性值，可以控制NPU设备初始化时的选项。该接口必须在调用npu.open接口初始化NPU设备前完成设置。

本节描述NPU提供的全局单例相关配置。

调用示例

如果您需要修改默认配置项，应当在初始化NPU设备前设置全局配置项，比如，您希望将精度模式由allow_fp32_to_fp16修改为allow_mix_precision，您应当这样调用：

```
import npu_device as npu
npu.global_options().precision_mode = 'allow_mix_precision'
npu.open().as_default()
```

11.1.1.2.2 配置参数说明

基础功能

参数名	描述
graph_run_mode	图执行模式，取值： ● 0：在线推理场景下，请配置为0。 ● 1：训练场景下，请配置为1，默认为1。 配置示例： `npu.global_options().graph_run_mode=1`

参数名	描述
deterministic	是否开启确定性计算，开启确定性开关后，算子在相同的硬件和输入下，多次执行将产生相同的输出。 此配置项有以下两种取值： ● 0：默认值，不开启确定性计算。 ● 1：开启确定性计算。 默认情况下，无需开启确定性计算。因为开启确定性计算后，算子执行时间会变慢，导致性能下降。在不开启确定性计算的场景下，多次执行的结果可能不同。这个差异的来源，一般是因为在算子实现中，存在异步的多线程执行，会导致浮点数累加的顺序变化。 但当发现模型执行多次结果不同，或者精度调优时，可以通过此配置开启确定性计算辅助进行调试调优。需要注意，如果希望有完全确定的结果，在训练脚本中需要设置确定的随机数种子，保证程序中产生的随机数也都是确定的。 配置示例： `npu.global_options().deterministic=1`

内存管理

参数名	描述
atomic_clean_policy	是否集中清理网络中所有atomic算子占用的内存，取值包括： ● 0：集中清理，默认为0。 ● 1：不集中清理，对网络每一个atomic算子进行单独清零。当网络中内存超限时，可尝试此种清理方式，但可能会导致一定的性能损耗。 配置示例： `npu.global_options().memory_config.atomic_clean_policy=1`
external_weight	是否将网络中Const/Constant节点的权重外置，取值包括： ● False：权重不外置，保存在图中，默认为False。 ● True：权重外置，将网络中所有Const/Constant节点的权重落盘至临时目录“tmp_weight”下，并将其类型转换为FileConstant；模型卸载时，自动卸载“tmp_weight”目录下的权重文件。 **说明：**一般场景下不需要配置此参数，针对模型加载环境有内存限制的场景，可以将权重外置。 配置示例： `npu.global_options().external_weight=True`

参数名	描述
static_memory_policy	网络运行时使用的内存分配方式。 ● 0：动态分配内存，即按照实际大小动态分配。 ● 2：动态扩展内存。训练与在线推理场景下，可以通过此取值实现同一session中多张图之间的内存复用，即以最大图所需内存进行分配。例如，假设当前执行图所需内存超过前一张图的内存时，直接释放前一张图的内存，按照当前图所需内存重新分配。 默认值是0，配置示例： `npu.global_options().memory_config.static_memory_policy=0` **说明** 为兼容历史版本配置，配置为“1”的场景下，系统会按照“2”动态扩展内存的方式进行处理。

功能调试

参数名	描述

op_debug_config	Global Memory内存检测功能开关。
	取值为.cfg配置文件路径，配置文件内多个选项用英文逗号分隔：
	● oom：在算子执行过程中，检测Global Memory是否内存越界
	● dump_bin：算子编译时，在当前执行路径下的kernel_meta文件夹中保留.o和.json文件
	● dump_cce：算子编译时，在当前执行路径下的kernel_meta文件夹中保留算子cce文件*.cce
	● dump_loc：算子编译时，在当前执行路径下的kernel_meta文件夹中保留python-cce映射文件*_loc.json
	● ccec_O0：算子编译时，开启ccec编译器的默认编译选项-O0，此编译选项针对调试信息不会执行任何优化操作
	● ccec_g ： 算子编译时，开启ccec编译器的编译选项-g，此编译选项相对于-O0，会生成优化调试信息
	配置示例：
	`npu.global_options().op_debug_config="/root/test0.cfg"`
	其中，test0.cfg文件信息为：
	`op_debug_config = ccec_O0,ccec_g,oom`
	说明
	● 开启ccec编译选项的场景下（即ccec_O0、ccec_g选项），会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要开启ccec编译选项。
	由于算子kernel文件过大而无法编译的日志显示如下：
	`message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx)`
	● 此参数取值为“dump_bin”、“dump_cce”、“dump_loc”时，可通过“debug_dir”参数指定调试相关过程文件的存放路径。

enable_exception_dump	是否Dump异常算子的输入和输出信息，dump信息生成在当前脚本执行目录。 ● 0：关闭，默认为0。 ● 1：开启。 配置示例： `npu.global_options().enable_exception_dump=1`

参数名	描述
debug_dir	用于配置保存算子编译生成的调试相关的过程文件的路径，包括算子.o/.json/.cce等文件。 默认生成在当前脚本执行路径下。 配置示例： npu.global_options().debug_dir="/home/test"

精度调优

参数名	描述
precision_mode	算子精度模式，配置要求为string类型。 ● allow_fp32_to_fp16：对于矩阵类算子，使用float16；对于矢量类算子，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网络模型中算子不支持float32，则直接降低精度到float16。 ● force_fp16：算子既支持float16又支持float32数据类型时，强制选择float16。 ● cube_fp16in_fp32out/force_fp32：算子既支持float16又支持float32数据类型时，系统内部根据算子类型的不同，选择合适的处理方式。配置为force_fp32或cube_fp16in_fp32out，效果等同，cube_fp16in_fp32out为新版本中新增配置，对于矩阵计算类算子，该选项语义更清晰。 – 对于矩阵计算类算子，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32。 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32。 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16。 4. 如果以上场景都不支持，则报错。 – 对于矢量计算类算子，如果网络模型中算子同时支持float16和float32，强制选择float32，若原图精度为float16，也会强制转为float32。如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了混合精度黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 ● must_keep_origin_dtype：保持原图精度。如果原图中某算子精度为float16，但NPU中该算子实现不支持float16、仅支持float32，则系统内部自动采用高精度float32；如果原图中某算子精度为float32，但NPU中该算子实现不支持float32、仅支持float16，此场景下不能使用此参数值，系统不支持使用低精度。 ● allow_mix_precision_fp16/allow_mix_precision：开启自动混合精度功能，表示混合使用float16和float32数据类型来处理神经网络的过程。 配置为allow_mix_precision或allow_mix_precision_fp16，效果等同，其中allow_mix_precision_fp16为新版本中新增配置，

参数名	描述
	语义更清晰，便于理解。针对全网中float32数据类型的算子，系统会按照内置优化策略自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。开启该功能开关后，用户可以同时使能Loss Scaling，从而补偿降低精度带来的精度损失。 针对昇腾910 AI处理器，默认配置项为“allow_fp32_to_fp16”。 针对昇腾910B AI处理器，默认配置项为“must_keep_origin_dtype”。 开启自动混合精度的场景下，用户可参考**修改混合精度黑白名单**修改网络中某算子的精度转换规则。 配置示例： `npu.global_options().precision_mode="allow_mix_precision"` **说明** • 该参数不能与“precision_mode_v2”参数同时使用，建议使用“precision_mode_v2”参数。 • 在使用此参数设置整个网络的精度模式时，可能会存在个别算子存在精度问题，此种场景下，建议通过**11.1.1.7 npu.keep_dtype_scope**接口设置某些算子保持原图精度。

参数名	描述
precision_mode_v2	算子精度模式，配置要求为string类型。 • fp16：算子既支持float16又支持float32数据类型时，强制选择float16。 • origin：保持原图精度。如果原图中某算子精度为float16，但NPU中该算子实现不支持float16、仅支持float32，则系统内部自动采用高精度float32；如果原图中某算子精度为float32，但NPU中该算子实现不支持float32、仅支持float16，此场景下不能使用此参数值，系统不支持使用低精度。 • cube_fp16in_fp32out：算子既支持float16又支持float32数据类型时，系统内部根据算子类型的不同，选择合适的处理方式。 – 对于矩阵计算类算子，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32。 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32。 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16。 4. 如果以上场景都不支持，则报错。 – 对于矢量计算类算子，如果网络模型中算子同时支持float16和float32，强制选择float32，若原图精度为float16，也会强制转为float32。如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了混合精度黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 • mixed_float16：开启自动混合精度功能，表示混合使用float16和float32数据类型来处理神经网络的过程。 针对网络中float32数据类型的算子，系统会按照内置优化策略自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。开启该功能开关后，用户可以同时使能Loss Scaling，从而补偿降低精度带来的精度损失。 开启自动混合精度的场景下，用户可参考**修改混合精度黑白名单**修改网络中某算子的精度转换规则。 配置示例： `npu.global_options().precision_mode_v2="origin"`

参数名	描述
	说明 • 该参数不能与“precision_mode”参数同时使用，建议使用“precision_mode_v2”参数。 • 在使用此参数设置整个网络的精度模式时，可能会存在个别算子存在精度问题，此种场景下，建议通过**11.1.1.7 npu.keep_dtype_scope**接口设置某些算子保持原图精度。

参数名	描述

modify_mixlist	开启混合精度的场景下，开发者可通过此参数指定混合精度黑白灰名单的路径以及文件名，自行指定哪些算子允许降精度，哪些算子不允许降精度。
	用户可以在脚本中通过配置“precision_mode”参数或者“precision_mode_v2”参数开启混合精度。例如：
	● precision_mode参数配置为allow_mix_precision_fp16/allow_mix_precision。
	● precision_mode_v2参数配置为mixed_float16。
	说明
	“precision_mode”参数与precision_mode_v2参数不能同时使用，建议使用“precision_mode_v2”参数。
	黑白灰名单存储文件为json格式，配置示例：
	`npu.global_options().modify_mixlist="/home/test/ops_info.json"`
	ops_info.json中可以指定算子类型，多个算子使用英文逗号分隔，样例如下：
	`{`
	`  "black-list": {  // 黑名单`
	`    "to-remove": [  // 黑名单算子转换为灰名单算子`
	`    "Xlog1py"`
	`    ],`
	`    "to-add": [  // 白名单或灰名单算子转换为黑名单算子`
	`    "Matmul",`
	`    "Cast"`
	`    ]`
	`  },`
	`  "white-list": {  // 白名单`
	`    "to-remove": [  // 白名单算子转换为灰名单算子`
	`    "Conv2D"`
	`    ],`
	`    "to-add": [  // 黑名单或灰名单算子转换为白名单算子`
	`    "Bias"`
	`    ]`
	`  }`
	`}`
	如何查询混合精度场景下算子的内置优化策略：内置优化策略在“OPP安装目录/opp/built-in/ai_core/tbe/config/<soc_version>/aic-<soc_version>-ops-info.json”，例如：
	`"Conv2D":{`
	`  "precision_reduce":{`
	`  "flag":"true"`
	`},`
	● true：（白名单）允许将当前float32类型的算子，降低精度到float16。
	● false：（黑名单）不允许将当前float32类型的算子，降低精度到float16。
	● 不配置：（灰名单）当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。

参数名	描述
customize_dtypes	使用precision_mode参数设置整个网络的精度模式时，可能会存在个别算子存在精度问题，此种场景下，可以使用customize_dtypes参数配置个别算子的精度模式，而模型中的其他算子仍以precision_mode指定的精度模式进行编译。需要注意，当precision_mode取值为“must_keep_origin_dtype”时，customize_dtypes参数不生效。 该参数需要配置为配置文件路径及文件名，例如：/home/test/customize_dtypes.cfg。 配置示例： npu.global_options().customize_dtypes = "/home/test/customize_dtypes.cfg" 配置文件中列举需要自定义计算精度的算子名称或算子类型，每个算子单独一行，且算子类型必须为基于Ascend IR定义的算子的类型。对于同一个算子，如果同时配置了算子名称和算子类型，编译时以算子名称为准。 配置文件格式要求： # 按照算子名称配置 *Opname1*::InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* *Opname2*::InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* # 按照算子类型配置 OpType::*TypeName1*:InputDtype:*dtype1,dtype2,…* OutputDtype:*dtype1,…* OpType::*TypeName2*:InputDtype:*dtype1,dtype2,…* OutputDtype:*dtype1,…* 配置文件配置示例： # 按照算子名称配置 resnet_v1_50/block1/unit_3/bottleneck_v1/Relu::InputDtype:float16,int8,OutputDtype:float16,int8 # 按照算子类型配置 OpType::Relu:InputDtype:float16,int8,OutputDtype:float16,int8 **说明** • 算子具体支持的计算精度可以从算子信息库中查看，默认存储路径为CANN软件安装后文件存储路径的：opp/built-in/op_impl/ai_core/tbe/config/*${soc_version}*/aic-*${soc_version}*-ops-info.json。 • 通过该参数指定的优先级高，因此可能会导致精度/性能的下降；如果指定的dtype不支持，会导致编译失败。 • 若通过算子名称进行配置，由于模型编译过程中会进行融合、拆分等优化操作，可能会导致算子名称发生变化，进而导致配置不生效，未达到精度提升的目的。此种场景下，可进一步通过获取日志进行问题定位，关于日志的详细说明请参见《**日志参考**》。

精度比对

参数名	描述
fusion_switch_file	融合开关配置文件路径以及文件名。 格式要求：支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。 系统内置了一些图融合和UB融合规则，均为默认开启，可以根据需要关闭指定的融合规则。 配置示例： `npu.global_options().fusion_switch_file="/home/test/fusion_switch.cfg"` 配置文件fusion_switch.cfg样例如下，on表示开启，off表示关闭。 `{ "Switch":{ "GraphFusion":{ "RequantFusionPass":"on", "ConvToFullyConnectionFusionPass":"off", "SoftmaxFusionPass":"on", "NotRequantFusionPass":"on", "ConvConcatFusionPass":"on", "MatMulBiasAddFusionPass":"on", "PoolingFusionPass":"on", "ZConcatv2dFusionPass":"on", "ZConcatExt2FusionPass":"on", "TfMergeSubFusionPass":"on" }, "UBFusion":{ "TbePool2dQuantFusionPass":"on" } } }` 同时支持用户一键关闭融合规则： `{ "Switch":{ "GraphFusion":{ "ALL":"off" }, "UBFusion":{ "ALL":"off" } } }` 需要注意的是： 1. 关闭某些融合规则可能会导致功能问题，因此此处的一键式关闭仅关闭系统部分融合规则，而不是全部融合规则。 2. 一键式关闭融合规则时，可以同时开启部分融合规则： `{ "Switch":{ "GraphFusion":{ "ALL":"off", "SoftmaxFusionPass":"on" }, "UBFusion":{ "ALL":"off", "TbePool2dQuantFusionPass":"on"`

参数名	描述
	} } }
dump_config.enable_dump	是否开启Data Dump功能，默认值：False。 ● True：开启Data Dump功能，从dump_path读取Dump文件保存路径。 ● False：关闭Data Dump功能。 **说明** 不能同时使能Data Dump和溢出数据检测能力。 配置示例： npu.global_options().dump_config.enable_dump=True
dump_config.dump_path	Dump文件保存路径。enable_dump或enable_dump_debug为true时，该参数必须配置。 该参数指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。 ● 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。 ● 相对路径配置直接以目录名开始，例如：output。 配置示例： npu.global_options().dump_config.dump_path = "/home/HwHiAiUser/output"
dump_config.dump_step	指定采集哪些迭代的Data Dump数据。默认值：None，表示所有迭代都会产生dump数据。 多个迭代用“|”分割，例如：0|5|10；也可以用"-"指定迭代范围，例如：0|3-5|10。 配置示例： npu.global_options().dump_config.dump_step="0|5"
dump_config.dump_mode	Data Dump模式，用于指定dump算子输入还是输出数据，默认为output。取值如下： ● input：仅dump算子输入数据 ● output：仅dump算子输出数据 ● all：dump算子输入和输出数据 配置示例： npu.global_options().dump_config.dump_mode="all"

参数名	描述
dump_config.dump_data	指定算子dump内容类型，取值： ● tensor: dump算子数据，默认为tensor。 ● stats: dump算子统计数据，结果文件为csv格式。 大规模训练场景下，通常dump数据量太大并且耗时长，可以先dump所有算子的统计数据，根据统计数据识别可能异常的算子，然后再指定dump异常算子的input或output数据。 配置示例： `npu.global_options().dump_config.dump_data = "stats"`
dump_config.dump_layer	指定需要dump的算子。取值为算子名，多个算子名之间使用空格分隔。若不配置此字段，默认dump全部算子。 配置示例： `npu.global_options().dump_config.dump_layer = "nodename1 nodename2 nodename3"`
dump_config.enable_dump_debug	溢出检测场景下，是否开启溢出数据采集功能，默认值：False。 ● True：开启采集溢出数据的功能，从dump_path读取Dump文件保存路径，dump_path为None时会产生异常。 ● False：关闭采集溢出数据的功能。 **说明** 不能同时开启Data Dump与溢出数据采集功能，即不同时将dump_config.enable_dump和dump_config.enable_dump_debug参数配置为“True”。 配置示例： `npu.global_options().dump_config.enable_dump_debug=True`
dump_config.dump_debug_mode	溢出检测模式，默认为all，取值如下： ● aicore_overflow：AI Core算子溢出检测，检测在算子输入数据正常的情况下，输出是否不正常的极大值（如float16下65500,38400,51200这些值）。一旦检测出这类问题，需要根据网络实际需求和算子逻辑来分析溢出原因并修改算子实现 ● atomic_overflow：Atomic Add溢出检测，即除了AICore之外，还有其他涉及浮点计算的模块，比如SDMA，检测这些部分出现的溢出问题。 ● all：同时进行AI Core算子溢出检测和Atomic Add溢出检测。 配置示例： `npu.global_options().dump_config.dump_debug_mode="aicore_overflow"`

性能调优

参数名	描述
hcom_parallel	是否启用Allreduce梯度更新和前后向并行执行。 ● True：开启Allreduce并行。 ● False：关闭Allreduce并行。 默认值为“True”，配置示例： npu.global_options().hcom_parallel=True
enable_small_channel	是否使能small channel的优化，使能后在channel<=4的卷积层会有性能收益。 ● 0：关闭。训练（graph_run_mode为1）场景下默认关闭，且训练场景下不建议用户开启。 ● 1：使能。在线推理（graph_run_mode为0）场景下默认开启。 **说明** 该参数使能后，当前只在Resnet50、Resnet101、Resnet152、GoogleNet网络模型能获得性能收益。其他网络模型性能可能会下降，用户根据实际情况决定是否使能该参数。 配置示例： npu.global_options().enable_small_channel=1

参数名	描述
op_precision_mode	设置具体某个算子的高精度或高性能模式，通过该参数传入自定义的模式配置文件op_precision.ini，可以为不同的算子设置不同的模式。 支持按照算子类型或者按照节点名称设置，按节点名称设置的优先级高于算子类型，样例如下： [ByOpType] optype1=high_precision optype2=high_performance optype3=support_out_of_bound_index [ByNodeName] nodename1=high_precision nodename2=high_performance nodename3=support_out_of_bound_index ● high_precision：表示高精度。 ● high_performance：表示高性能。 ● support_out_of_bound_index：表示对gather、scatter和segment类算子的indices输入进行越界校验， 校验会降低算子的执行性能。 具体某个算子支持配置的精度/性能模式取值，可通过CANN软件安装后文件存储路径的opp/built-in/op_impl/ai_core/tbe/impl_mode/all_ops_impl_mode.ini 文件查看。 该参数不能与op_select_implmode、optypelist_for_implmode参数同时使用，若三个参数同时配置，则只有op_precision_mode参数指定的模式生效。 一般场景下该参数无需配置。若使用高性能或者高精度模式，网络性能或者精度不是最优，则可以使用该参数，通过配置ini文件调整某个具体算子的精度模式。 配置示例： npu.global_options().op_precision_mode="/home/test/op_precision.ini"
stream_max_parallel_num	此参数仅适用于NMT网络。 指定AICPU/AICORE引擎的并行度，从而实现AICPU/AICORE算子间的并行执行。 DNN_VM_AICPU为AICPU引擎名称，本示例指定了AICPU引擎的并发数为10； AIcoreEngine为AICORE引擎名称，本示例指定了AICORE引擎的并发数为1。 AICPU/AICORE引擎的并行度默认为1，取值范围为：[1,13]。 配置示例： npu.global_options().stream_max_parallel_num="DNN_VM_AICPU:10,AIcoreEngine:1"

参数名	描述
is_tailing_optimization	此参数仅适用于Bert网络。 分布式训练场景下，是否开启通信拖尾优化，用于提升训练性能。通信拖尾优化即，通过计算依赖关系的改变，将不依赖于最后一个AR（梯度聚合分片）的计算操作调度到和最后一个AR并行进行，以达到优化通信拖尾时间的目的。取值： ● True。 ● False：默认为False。 配置示例： npu.global_options().is_tailing_optimization=True
enable_scope_fusion_passes	指定编译时需要生效的融合规则列表。此处传入注册的融合规则名称，允许传入多个，用“,”隔开。 无论是内置还是用户自定义的Scope融合规则，都分为如下两类： ● 通用融合规则（General）：各网络通用的Scope融合规则；默认生效，不支持用户指定失效。 ● 定制化融合规则（Non-General）：特定网络适用的Scope融合规则；默认不生效，用户可以通过enable_scope_fusion_passes指定生效的融合规则列表。 配置示例： npu.global_options().enable_scope_fusion_passes="ScopeLayerNormPass,ScopeClipBoxesPass"

Profiling

参数名	描述
profiling_config.enable_profiling	是否开启Profiling功能，默认关闭。 ● True：开启Profiling功能，从profiling_options读取Profiling的采集选项。 ● False：关闭Profiling功能。 配置示例： npu.global_options().profiling_config.enable_profiling=True

参数名	描述
profiling_config.profiling_options	Profiling配置选项。 ● output：Profiling采集结果文件保存路径。该参数指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。 – 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。 – 相对路径配置直接以目录名开始，例如：output。 ● storage_limit：指定落盘目录允许存放的最大文件容量。当Profiling数据文件在磁盘中即将占满本参数设置的最大存储空间（剩余空间<=20MB）或剩余磁盘总空间即将被占满时（总空间剩余<=20MB），则将磁盘内最早的文件进行老化删除处理。 单位为MB，有效取值范围为[200, 4294967296]，默认未配置本参数。 参数值配置格式为数值+单位，例如"storage_limit": "200MB"。 未配置本参数时，默认取值为Profiling数据文件存放目录所在磁盘可用空间的90%。 ● training_trace：采集迭代轨迹数据，即训练任务及AI软件栈的软件信息，实现对训练任务的性能分析，重点关注数据增强、前后向计算、梯度聚合更新等相关数据。非必选项配置，用户可不配置，当用户配置时必须设为“on”。 ● task_trace：采集任务轨迹数据，即昇腾AI处理器HWTS，分析任务开始、结束等信息。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。 ● hccl：控制hccl数据采集开关，可选on或off，默认为off。 ● aicpu：采集aicpu数据增强的Profiling数据。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。 ● fp_point：training_trace为on时需要配置。指定训练网络迭代轨迹正向算子的开始位置，用于记录前向计算开始时间戳。配置值为指定的正向第一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入。 ● bp_point：training_trace为on时需要配置。指定训练网络迭代轨迹反向算子的结束位置，记录后向计算结束时间戳，fp_point和bp_point可以计算出正反向时间。配置值为指定的反向最后一个算子名

参数名	描述
	字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入。 • aic_metrics：AI Core和AI Vector Core的硬件信息，取值如下： ArithmeticUtilization：各种计算类指标占比统计 PipeUtilization： 计算单元和搬运单元耗时占比，该项为默认值 Memory：外部内存读写类指令占比 MemoryL0：内部内存读写类指令占比 ResourceConflictRatio：流水线队列类指令占比 L2Cache：读写cache命中次数和缺失后重新分配次数 **说明** 支持自定义需要采集的寄存器，例如："aic_metrics":"**Custom:***0x49,0x8,0x15,0x1b,0x64,0x10*"。 • Custom字段表示自定义类型，配置为具体的寄存器值，取值范围为[0x1, 0x6E]。 • 配置的寄存器数最多不能超过8个，寄存器通过“,”区分开。 • 寄存器的值支持十六进制或十进制。 • l2：控制L2采样数据的开关，可选on或off，默认为off。 • msproftx：控制msproftx用户和上层框架程序输出性能数据的开关，可选on或off，默认值为off。Profiling开启msproftx功能之前，需要在程序内调用msproftx相关接口来开启程序的Profiling数据流的输出，详细操作请参见《**应用软件开发指南（C&C++）**》“高级功能>Profiling性能数据采集”章节。 • task_time：控制任务调度耗时以及算子耗时的开关。涉及在task_time、op_summary、op_statistic文件中输出相关耗时数据。可选on或off，默认为on。 • runtime_api：控制runtime api性能数据采集开关，可选on或off，默认为off。可采集runtime-api性能数据，Host与Device之间、Device间的同步异步内存复制时延runtime-api性能数据。 • sys_hardware_mem_freq：DDR、HBM（昇腾910 AI处理器支持该参数）带宽及内存信息采集频率、LLC的读写带宽数据采集频率以及acc_pmu数据和SOC传输带宽信息采集频率。取值范围为[1,100]，默认值50，单位hz。 • llc_profiling：LLC Profiling采集事件，可以设置为：

参数名	描述
	– 昇腾310 AI处理器，可选capacity（采集AI CPU和Control CPU的LLC capacity数据）或bandwidth（采集LLC bandwidth），默认为capacity。 – 昇腾310P AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。 – 昇腾910 AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。 – 可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。 ● sys_io_sampling_freq：NIC（昇腾310 AI处理器）（昇腾910 AI处理器）、ROCE（昇腾910 AI处理器）采集频率。取值范围为[1,100]，默认值100，单位hz。 ● sys_interconnection_freq：集合通信带宽数据（HCCS，昇腾910 AI处理器）、PCIe数据（昇腾310P AI处理器）（昇腾910 AI处理器）采集频率以及片间传输带宽信息采集频率。取值范围为[1,50]，默认值50，单位hz。 ● dvpp_freq：DVPP采集频率。取值范围为[1,100]，默认值50，单位hz。 ● instr_profiling_freq：AI Core和AI Vector的带宽和延时采集频率。取值范围[300,30000]，默认值1000，单位hz。 ● host_sys：Host侧性能数据采集开关。取值包括cpu和mem，可选其中的一项或多项，选多项时用英文逗号隔开，例如"host_sys": "cpu,mem"。 ● host_sys_usage：采集Host侧系统及所有进程的CPU和内存数据。取值包括cpu和mem，可选其中的一项或多项，选多项时用英文逗号隔开。 ● host_sys_usage_freq：配置Host侧系统和所有进程CPU、内存数据的采集频率。取值范围为[1,50]，默认值50，单位hz。 **说明** instr_profiling_freq开关与training_trace、task_trace、hccl、aicpu、fp_point、bp_point、aic_metrics、l2互斥，无法同时执行。 配置示例： `npu.global_options().profiling_config.profiling_options = '{"output":"/tmp/profiling","training_trace":"on","fp_point":"resnet_model/conv2d/Conv2Dresnet_model/batch_normalization/FusedBatchNormV3_Reduce","bp_point":"gradients/AddN_70"}'`

AOE

参数名	描述
aoe_config.aoe_mode	通过AOE工具进行调优的调优模式。 ● 1：子图调优。 ● 2：算子调优。 ● 4：梯度切分调优。 在数据并行的场景下，使用allreduce对梯度进行聚合，梯度的切分方式与分布式训练性能强相关，切分不合理会导致反向计算结束后存在较长的通信拖尾时间，影响集群训练的性能和线性度。用户可以通过集合通信的梯度切分接口（set_split_strategy_by_idx或set_split_strategy_by_size）进行人工调优，但难度较高。因此，可以通过工具实现自动化搜索切分策略，通过在实际环境预跑采集性能数据，搜索不同的切分策略，理论评估出最优策略输出给用户，用户拿到最优策略后通过set_split_strategy_by_idx接口设置到该网络中。 **说明** 通过修改训练脚本和AOE_MODE环境变量都可配置调优模式，同时配置的情况下，通过修改训练脚本方式优先生效。 配置示例： `npu.global_options().aoe_config.aoe_mode="1"`
aoe_config.work_path	AOE工具调优工作目录，存放调优配置文件和调优结果文件，默认生成在训练当前目录下。 该参数类型为字符串，指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。 ● 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。 ● 相对路径配置直接以目录名开始，例如：output。 配置示例： `npu.global_options().aoe_config.work_path = "/home/HwHiAiUser/output"`

参数名	描述

aoe_config.aoe_config_file	通过AOE工具进行调优时，若仅针对网络中某些性能较低的算子进行调优，可通过此参数进行设置。该参数配置为包含算子信息的配置文件路径及文件名，例如：/home/test/cfg/tuning_config.cfg。
	配置示例：
	`npu.global_options().aoe_config.aoe_config_file="/home/test/cfg/tuning_config.cfg"`
	配置文件中配置的是需要进行调优的算子信息，文件内容格式如下：
	`{ "tune_ops_name":["bert/embeddings/addbert/embeddings/add_1","loss/MatMul"], "tune_ops_type":["Add", "Mul"] "tune_optimization_level":"O1", "feature":["deeper_opat"] }`
	● tune_ops_name：指定的算子名称，当前实现是支持全字匹配，可以指定一个，也可以指定多个，指定多个时需要用英文逗号分隔。此处配置的算子名称需要为经过图编译器处理过的网络模型的节点名称，可从Profiling调优数据中获取，详细可参见《**性能分析工具使用指南**》。
	● tune_ops_type：指定的算子类型，当前实现是支持全字匹配，可以指定一个，也可以指定多个，指定多个时需要用英文逗号分隔。如果有融合算子包括了该算子类型，则该融合算子也会被调优。
	● tune_optimization_level：调优模式，取值为O1表示高性能调优模式，取值为O2表示正常模式。默认值为O2。
	● feature：调优功能特性开关，可以取值为deeper_opat或者nonhomo_split，取值为deeper_opat时，表示开启算子深度调优，aoe_mode需要配置为2；取值为nonhomo_split时，表示开启子图非均匀切分，aoe_mode需要配置为1。
	说明
	如上配置文件中，tune_ops_type和tune_ops_name可以同时存在，同时存在时取并集，也可以只存在某一个。

算子编译

参数名	描述
op_compiler_cache_mode	用于配置算子编译磁盘缓存模式。默认值为enable。 ● enable：启用算子编译缓存功能。启用后，算子编译信息缓存至磁盘，相同编译参数的算子无需重复编译，直接使用缓存内容，从而提升编译速度。 ● disable：禁用算子编译缓存功能。 ● force：启用算子编译缓存功能，区别于enable模式，force模式下会强制刷新缓存，即先删除已有缓存，再重新编译并加入缓存。比如当用户的python或者依赖库等发生变化时，需要指定为force用于清理已有的缓存。 **说明** 配置为force模式完成编译后，建议后续编译修改为enable模式，以避免每次编译时都强制刷新缓存。 使用说明： ● 该参数和op_compiler_cache_dir配合使用。 ● 由于force选项会先删除已有缓存，所以不建议在程序并行编译时设置，否则可能会导致其他模型因使用的缓存内容被清除而编译失败。 ● 建议模型最终发布时设置编译缓存选项为disable或者force。 ● 如果算子调优后知识库变更，则需要通过设置为force来刷新缓存，否则无法应用新的调优知识库，从而导致调优应用执行失败。 ● 注意，调试开关打开的场景下，即op_debug_level非0值或者op_debug_config配置非空时，会忽略算子编译磁盘缓存模式的配置，不启用算子编译缓存。主要基于以下两点考虑： – 启用算子编译缓存功能（enable或force模式）后，相同编译参数的算子无需重复编译，编译过程日志无法完整记录。 – 受限于缓存空间大小，对调试场景的编译结果不做缓存。 ● 启用算子编译缓存功能时，可以通过以下方式来配置来设置缓存文件夹的磁盘空间大小： 1. 通过配置文件op_cache.ini设置。 算子编译完成后，会在op_compiler_cache_dir指定的目录下自动生成op_cache.ini文件，开发者可通过该配置文件进行缓存磁盘空间大小的配置。若op_cache.ini文件不存在，可手动创建。 在“op_cache.ini”文件中，增加如下信息： #配置文件格式，必须包含，自动生成的文件中默认包括如下信息，手动创建时，需要输入 **[op_compiler_cache]** #限制某个芯片下缓存文件夹的磁盘空间的大小，单位为MB

参数名	描述
	ascend_max_op_cache_size=500 #设置需要保留缓存的空间大小比例，取值范围：[1,100]，单位为百分比；例如80表示缓存空间不足时，删除缓存，保留80% **ascend_remain_cache_size_ratio=80** – 上述文件中的ascend_max_op_cache_size和ascend_remain_cache_size_ratio参数取值都有效时，op_cache.ini文件才会生效。 – 若多个使用者使用相同的缓存路径，该配置文件会影响所有使用者。 2. 通过环境变量**11.3.4.2 ASCEND_MAX_OP_CACHE_SIZE**设置。开发者可以通过环境变量ASCEND_MAX_OP_CACHE_SIZE来限制某个芯片下缓存文件夹的磁盘空间的大小，当编译缓存空间大小达到ASCEND_MAX_OP_CACHE_SIZE设置的取值，且需要删除旧的kernel文件时，可以通过环境变量**11.3.4.3 ASCEND_REMAIN_CACHE_SIZE_RATIO**设置需要保留缓存的空间大小比例。 **若同时配置了op_cache.ini文件和环境变量，则优先读取op_cache.ini文件中的配置项，若op_cache.ini文件和环境变量都未设置，则读取系统默认值：默认磁盘空间大小500M，默认保留缓存的空间50%。** 配置示例： npu.global_options().op_compiler_cache_mode="enable"
op_compiler_cache_dir	用于配置算子编译磁盘缓存的目录。 路径支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。 如果参数指定的路径存在且有效，则在指定的路径下自动创建子目录kernel_cache；如果指定的路径不存在但路径有效，则先自动创建目录，然后在该路径下自动创建子目录kernel_cache。 默认值：$HOME/atc_data 配置示例： npu.global_options().op_compiler_cache_dir="/home/test/kernel_cache"

异常补救

参数名	描述
stream_sync_timeout	图执行时，stream同步等待超时时间，超过配置时间时报同步失败。单位：ms 默认值-1，表示无等待时间，出现同步失败不报错。 **说明：集群训练场景下，此配置的值（即stream同步等待超时时间）需要大于集合通信超时时间，即环境变量11.3.5.16 HCCL_EXEC_TIMEOUT的值。** 配置示例： `npu.global_options().stream_sync_timeout=600000`
event_sync_timeout	图执行时，event同步等待超时时间，超过配置时间时报同步失败。单位：ms 默认值-1，表示无等待时间，出现同步失败不报错。 配置示例： `npu.global_options().event_sync_timeout=600000`

实验参数

参数名	描述
jit_compile	模型编译时是否优先在线编译。 • auto：针对静态shape网络，在线编译算子；针对动态shape网络，优先查找系统中已编译好的算子二进制，如果查找不到对应的二进制，再编译算子。 • true：在线编译算子，系统根据得到的图信息进行融合及优化，从而编译出运行性能更优的算子。 • false：优先查找系统中的已编译好的算子二进制文件，如果能查找到，则不再编译算子，编译性能更优；如果查找不到，则再编译算子。 默认值：auto。 配置示例： `npu.global_options().jit_compile = "auto"`

后续版本废弃配置

参数名	描述
op_select_implmode	昇腾AI处理器部分内置算子有高精度和高性能实现方式，用户可以通过该参数配置模型编译时选择哪种算子。取值包括： ● high_precision：表示算子选择高精度实现。高精度实现算子是指在fp16输入的情况下，通过泰勒展开/牛顿迭代等手段进一步提升算子的精度。 ● high_performance：表示算子选择高性能实现。高性能实现算子是指在fp16输入的情况下，不影响网络精度前提的最优性能实现。 默认值为None，代表不使能此配置。 配置示例： npu.global_options().op_select_implmode="high_precision"
optypelist_for_implmode	列举算子optype的列表，该列表中的算子使用op_select_implmode参数指定的模式，当前支持的算子为Pooling、SoftmaxV2、LRN、ROIAlign，多个算子以“,”分隔。 该参数需要与op_select_implmode参数配合使用，配置示例： npu.global_options().op_select_implmode="high_precision" npu.global_options().optypelist_for_implmode="Pooling,SoftmaxV2" 默认值为None，代表不使能此配置。
variable_format_optimize	是否开启变量格式优化。 ● True：开启。 ● False：关闭。 为了提高训练效率，在网络执行的变量初始化过程中，将变量转换成更适合在昇腾AI处理器上运行的数据格式。但在用户特殊要求场景下，可以选择关闭该功能开关。 默认值为None，代表不使能此配置。 配置示例： npu.global_options().variable_format_optimize=True

参数名	描述
op_debug_level	算子debug功能开关，取值： ● 0：不开启算子debug功能。 ● 1：开启算子debug功能，在训练脚本执行目录下的kernel_meta文件夹中生成TBE指令映射文件（算子cce文件*.cce、python-cce映射文件*_loc.json、.o和.json文件），用于后续工具进行AICore Error问题定位。 ● 2：开启算子debug功能，在训练脚本执行目录下的kernel_meta文件夹中生成TBE指令映射文件（算子cce文件*.cce、python-cce映射文件*_loc.json、.o和.json文件），并关闭ccec编译器的编译优化开关且打开ccec调试功能（ccec编译器选项设置为-O0-g），用于后续工具进行AICore Error问题定位。 ● 3：不开启算子debug功能，且在训练脚本执行目录下的kernel_meta文件夹中保留.o和.json文件 ● 4：不开启算子debug功能，在训练脚本执行目录下的kernel_meta文件夹中**保留**.o（算子二进制文件）和.json文件（算子描述文件），生成TBE指令映射文件（算子cce文件*.cce）和UB融合计算描述文件（{$kernel_name}_compute.json） **须知** ● 训练执行时，建议配置为0或3。如果需要进行问题定位，再选择调试开关选项1和2，是因为加入了调试功能会导致网络性能下降。 ● 配置为2，即开启ccec编译选项的场景下，会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要配置为2。 由于算子kernel文件过大而无法编译的日志显示如下： message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx) ● 当该参数取值不为0时，可通过**功能调试**中的“debug_dir”参数指定调试相关过程文件的存放路径。 默认值为None，代表不使能此配置。 配置示例： npu.global_options().op_debug_level=0
graph_memory_max_size	历史版本，该参数用于指定网络静态内存和最大动态内存的大小。 当前版本，该参数不再生效。系统会根据网络使用的实际内存大小动态申请。
variable_memory_max_size	历史版本，该参数用于指定变量内存的大小。 当前版本，该参数不再生效。系统会根据网络使用的实际内存大小动态申请。

11.1.1.3 npu.distribute.all_reduce

函数原型

npu.distribute.all_reduce(values, reduction="mean", fusion=1, fusion_id=-1, group="hccl_world_group")

功能说明

用于NPU分布式部署场景下，worker间的聚合运算。

参数说明

参数名	输入/输出	描述
values	输入	TensorFlow的tensor类型。 tensor支持的数据类型为int8, int32, float16, float32。 tensor支持的数据类型为int8, int16（仅sum支持）, int32, float16, float32。
reduction	输入	String类型。 聚合运算的类型，可以为"mean","max","min","prod"或"sum"。默认为"mean"。
fusion	输入	int类型。 allreduce算子融合标识。 ● 0：不融合，该allreduce算子不和其他allreduce算子融合。 ● 1：按照梯度切分策略进行融合，默认为1。 ● 2：按照相同fusion_id进行融合。
fusion_id	输入	int类型。 allreduce算子的融合id。 对相同fusion_id的allreduce算子进行融合。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

对values进行聚合运算后的结果，类型与values一致，值与values输入一一对应。

调用示例

在多卡上聚合某个值：


```
# rank_id = 0  rank_size = 8
import npu_device as npu
v = tf.constant(1.0)
x = npu.distribute.all_reduce([v], 'sum') # 8.0
y = npu.distribute.all_reduce([v], 'mean') # 1.0
```

11.1.1.4 npu.distribute.broadcast

函数原型

npu.distribute.broadcast(values, root_rank, fusion=2, fusion_id=0, group="hccl_world_group")

功能说明

用于NPU分布式部署场景下，worker间的变量同步。

参数说明

参数名	输入/输出	描述
values	输入	单个TensorFlow的Variable或者Variable的集合。 tensor支持的数据类型为int8, int32, float16, float32, int64, uint64。 tensor支持的数据类型为int8, int16, int32, float16, float32, int64, uint64。
root_rank	输入	int类型。 作为root节点的rank_id，该id是group内的rank id。默认为0。
fusion	输入	int类型。 broadcast算子融合标识。 ● 0：不融合，该broadcast算子不和其他broadcast算子融合。 ● 2：按照相同fusion_id进行融合。默认为2。 ● 其他值非法。
fusion_id	输入	broadcast算子的融合id。 对相同fusion_id的broadcast算子进行融合。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

无。

调用示例

将0卡上的变量广播到其他卡：

```
# rank_id = 0  rank_size = 8
import npu_device as npu
x = tf.Variable(tf.random.normal(shape=()))
print("before broadcast", x)
npu.distribute.broadcast(x, root_rank=0)
print("after_broadcast", x)
```

广播前：

```
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.861866>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-2.8811553>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.60618496>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-1.2442238>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.06573954>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.37332603>
before broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.21582904>
```

广播后：

```
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
after_broadcast <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.91738653>
```

11.1.1.5 npu.distribute.npu_distributed_keras_optimizer_wrapper

函数原型

def npu_distributed_keras_optimizer_wrapper(optimizer, reduce_reduction="mean", fusion=1, fusion_id=-1, group="hccl_world_group")

功能说明

在更新梯度前，添加npu的allreduce操作对梯度进行聚合，然后再更新梯度。该接口仅在分布式场景下使用。

参数说明

参数名	输入/输出	描述
optimizer	输入	TensorFlow梯度训练优化器。
reduction	输入	String类型。 聚合运算的类型，可以为"mean","max","min","prod"或"sum"。默认为"mean"。

参数名	输入/输出	描述
fusion	输入	int类型。 allreduce算子融合标识。 ● 0：不融合，该allreduce算子不和其他allreduce算子融合。 ● 1：按照梯度切分策略进行融合，默认为1。 ● 2：按照相同fusion_id进行融合。
fusion_id	输入	int类型。 allreduce算子的融合id。 对相同fusion_id的allreduce算子进行融合。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

返回输入的optimizer。

调用示例

```
import npu_device as npu
optimizer = tf.keras.optimizers.SGD()
optimizer = npu.distribute.npu_distributed_keras_optimizer_wrapper(optimizer) # 使用NPU分布式计算，更新梯度
model.compile（ optimizer = optimizer ）
```

11.1.1.6 npu.distribute.shard_and_rebatch_dataset

函数原型

npu.distribute.shard_and_rebatch_dataset(dataset, global_bs)

功能说明

用于NPU分布式部署场景下，不同worker上数据集分片及batch大小调整。

参数说明

参数名	输入/输出	描述
dataset	输入	TensorFlow的Dataset类型。 需要进行切分的数据集。
global_bs	输入	全局batch的大小。

返回值

返回一个2个元素的tuple对象，第一个元素为切分后的Dataset，第二个元素为每个worker应当处理的实际batch大小。

调用示例

```
import npu_device as npu
dataset, batch_size = npu.distribute.shard_and_rebatch_dataset(dataset, batch_size)
```

11.1.1.7 npu.keep_dtype_scope

函数原型

npu.keep_dtype_scope()

功能说明

指定哪些算子保持原有精度，如果原始网络模型中的算子精度在昇腾AI处理器上不支持，则系统内部自动采用算子支持的高精度来计算。

参数说明

无。

返回值

Python上下文管理器，在该上下文中的构图算子，会带有NPU识别的特殊属性。

调用示例

```
import npu_device as npu
with npu.keep_dtype_scope():
    v = tf.add(1, 1)
```

11.1.1.8 npu.set_npu_loop_size

函数原型

npu.set_npu_loop_size(loop_size)

功能说明

用于设置NPU循环下沉执行时的循环次数。

参数说明

参数名	输入/输出	描述
loop_size	输入	NPU循环下沉执行时的循环次数，必须为大于0的整数。

返回值

无。

调用示例

```
import npu_device as npu
npu.set_npu_loop_size(100) # 设置循环下沉次数为100
```

11.1.1.9 npu.train.optimizer.NpuLossScaleOptimizer

函数原型

npu.train.optimizer.NpuLossScaleOptimizer(inner_optimizer, dynamic=True, initial_scale=None, dynamic_growth_steps=None)

功能说明

NPU提供的LossScaleOptimizer，由于NPU上的溢出运算不保证输出Inf或者NaN，使用LossScaleOptimizer的脚本应当替换为该优化器，来屏蔽溢出检测的差异。

参数说明

该优化器继承自tf.keras.mixed_precision.LossScaleOptimizer，使用方式完全相同，可以参考**LINK**。

返回值

NpuLossScaleOptimizer类型优化器。

调用示例

```
import npu_device as npu
optimizer = tf.keras.optimizers.Adam(lr=0.1)
optimizer = npu.train.optimizer.NpuLossScaleOptimizer(optimizer)
```

11.1.1.10 npu.ops.gelu

函数原型

npu.ops.gelu (x)

功能说明

计算高斯误差线性单元 (GELU) 激活函数。将输入Tensor乘以1个P(X <= x)，其中P(X) ~ N(0, 1)。

参数说明

参数名	输入/输出	描述
x	输入	输入Tensor，float类型。

返回值

tensor：对输入x执行完GELU操作之后的输出Tensor。数据类型和输入相同。

调用示例

```
import npu_device as npu
output = npu.ops.gelu(x)
```

11.1.1.11 set_device_sat_mod

函数原型

def set_device_sat_mode(mode)

功能说明

设置针对浮点计算的进程级溢出模式，当前可支持两种溢出模式：饱和模式与INF/NAN模式。

- 饱和模式：计算出现溢出时，计算结果会饱和为浮点数极值（+-MAX）。
- Inf/NaN模式：遵循IEEE 754标准，根据定义输出Inf/NaN的计算结果。

使用约束

无。

参数说明

参数名	输入/输出	描述
mode	输入	设置的溢出模式。 ● 0：饱和模式，为默认值。 ● 1：INF-NAN模式，推荐用户使用该模式。 针对昇腾910 AI处理器，仅支持设置为默认值“0”。

返回值

无

调用示例

以下示例仅针对昇腾910B AI处理器，针对昇腾910 AI处理器，开发者无需显式调用此接口。

```
import tensorflow as tf
import npu_device as npu
# 初始化NPU为默认设备
npu.open().as_default()
```



```
# 针对昇腾910B AI处理器，网络执行时调用如下接口进行溢出模式的设置
npu.npu_device.set_device_sat_mode(1)
```

11.1.2 TensorFlow 2.6 API 支持列表

请参考**LINK**中的TF2.6_api_support_list.xlsx文件。

11.2 集合通信接口参考

11.2.1 接口简介

NPUDistributedOptimizer和npu_distributed_optimizer_wrapper可以让用户在不需要感知allreduce的情况下自动完成梯度聚合功能，实现数据并行训练方式。同时为了满足用户灵活的使用方式，集合通信库HCCL提供了常用的rank管理、梯度切分功能、集合通信原型等接口。

HCCL（Huawei Collective Communication Library）是基于昇腾AI处理器的高性能集合通信库，提供单机多卡、多机多卡集合通信原语，在PCIe、HCCS和RoCE高速链路实现集合通信功能，实现分布式训练。

表 11-1 接口列表

分类	接口	简介	定义文件
rank管理	create_group	创建集合通信group。	${install_path}/python/site-packages/hccl/manage/api.py
	destroy_group	销毁集合通信group。	
	get_rank_size	获取group内rank数量（即Device数量）。	
	get_local_rank_size	获取group内device所在服务器内的local rank数量。	
	get_rank_id	获取device在group中对应的rank序号。	
	get_local_rank_id	获取device在group中对应的local rank序号。	
	get_world_rank_from_group_rank	从group rank id，获取该进程对应的world rank id。	
	get_group_rank_from_world_rank	从world rank id，获取该进程在group中的group rank id。	

分类	接口	简介	定义文件
梯度切分	set_split_strategy_by_idx	基于梯度的索引id，在集合通信group内设置反向梯度切分策略。	${install_path}/python/site-packages/hccl/split/api.py
	set_split_strategy_by_size	基于梯度数据量百分比，在集合通信group内设置反向梯度切分策略。	
集合通信算子	allreduce	提供group内的集合通信allreduce功能，对所有节点的同名张量进行约减。	${install_path}/python/site-packages/npu_bridge/hccl/hccl_ops.py
	allgather	提供group内的集合通信allgather功能，将所有节点的输入Tensor合并起来。	
	broadcast	提供group内的集合通信broadcast功能，将root节点的数据广播到其他rank。	
	reduce_scatter	提供group内的集合通信reducescatter功能。	
	send	提供group内点对点通信发送数据的send功能。	
	receive	提供group内点对点通信发送数据的receive功能。	
	reduce	提供group内的集合通信reduce功能，对所有节点的同名张量进行规约，并将数据输出至root节点。	
	alltoallv	集合通信域alltoallv操作接口。向通信域内所有rank发送数据（数据量可以定制），并从所有rank接收数据。	

分类	接口	简介	定义文件
	alltoallvc	集合通信域alltoallvc操作接口。向通信域内所有rank发送数据（数据量可以定制），并从所有rank接收数据。 alltoallvc通过输入参数send_count_matrix传入所有rank的收发参数，与alltoallv相比，性能更优。	

11.2.2 hccl.manage.api

11.2.2.1 create_group

函数原型

def create_group(group, rank_num, rank_ids)

功能说明

创建集合通信用户自定义group。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，集合通信group的标识，不能为hccl_world_group。 hccl_world_group是默认group，由ranktable文件创建，不能通过该接口创建，如果用户传入的group如果是hccl_world_group，创建group失败。
rank_num	输入	int类型。 组成该group的rank数量。 最大值为4096。

参数名	输入/输出	描述
rank_ids	输入	list类型。 组成该group的world_rank_id列表。 在不同单板类型上，有不同的限制。 针对昇腾910 AI处理器： ● 对于Server单机场景，rank_ids需满足如下条件：rank数量必须为1/2/4/8，0-3卡与4-7卡各为一个组网，rank数量为2/4时要求选取的昇腾AI处理器同属一个cluster。 ● 对于Server集群场景，rank_ids满足如下条件： – 各Server要选取相同数量的rank（且数量要求为1/2/4/8）。 – 各Server选取rank数量为2/4时要求选取的昇腾AI处理器同属一个cluster（即rank id按8取模余数都小于4或都大于等于4）。 举例： 假设对三台Server创建group，三台Server的rank id分别为： {0,1,2,3,4,5,6,7} {8,9,10,11,12,13,14,15} {16,17,18,19,20,21,22,23} 则满足要求的rank_ids列表可以是： rank_ids=[1,9,17] rank_ids=[1,2,9,10,17,18] rank_ids=[4,5,6,7,12,13,14,15,20,21,22,23] 针对昇腾910B AI处理器： ● 对于Server单机场景，rank_ids无限制条件。 ● 对于Server集群场景，rank_ids需满足如下条件： – 各Server要选取相同数量的rank（数量大小无要求）。 – 各Server选取的rank对应位置要相等（即rank id按8取模相等）。 举例： 假设对三台Server创建group，三台Server的rank id分别为： {0,1,2,3,4,5,6,7} {8,9,10,11,12,13,14,15} {16,17,18,19,20,21,22,23} 则满足要求的rank_ids列表可以是： rank_ids=[1,9,17] rank_ids=[1,2,9,10,17,18] rank_ids=[4,5,6,7,12,13,14,15,20,21,22,23]

返回值

无。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 重复创建名称相同的group，会创建失败。

11.2.2.2 destroy_group

函数原型

def destroy_group(group)

功能说明

销毁用户自定义group。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，集合通信group的标识。

返回值

无。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
destroy_group(“myGroup”)
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。

- 相同名称的group，11.2.2.2 destroy_group和11.2.2.1 create_group配套使用，必须在create_group完成之后调用；
- 如果用户传入的group恰好是hccl_world_group（默认group），销毁group失败；

11.2.2.3 get_rank_size

函数原型

def get_rank_size(group="hccl_world_group")

功能说明

获取group内rank数量（即Device数量）。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group，如果用户不传，默认为"hccl_world_group"。

返回值

int类型，返回group内rank数量。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
rankSize = get_rank_size(“myGroup”)
#rankSize = 4
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 11.2.2.1 create_group完成之后，调用此API获取本group的rank数量；
- 如果传入"hccl_world_group"，返回world_group的rank数量。

11.2.2.4 get_local_rank_size

函数原型

def get_local_rank_size(group="hccl_world_group")

功能说明

获取group内device所在服务器内的local rank数量。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group，如果用户不传，默认为"hccl_world_group"。

返回值

int类型，返回device所在服务器内的local rank数量。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
lcoalRankSize = get_local_rank_size(“myGroup”)
#localRankSize = 1
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 11.2.2.1 create_group完成之后，调用此API获取本group的local rank数量。
- 如果传入"hccl_world_group"，返回world_group的local rank数量。

11.2.2.5 get_rank_id

函数原型

def get_rank_id(group="hccl_world_group")

功能说明

获取device在group中对应的rank序号。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group，如果用户不传，默认为"hccl_world_group"。

返回值

int类型，返回device所在group的rank id。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
rankId = get_rank_id(“myGroup”)
#rankId = 0/1/2/3
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 11.2.2.1 create_group完成之后，调用此API获取进程在group中的rank id。
- 如果传入"hccl_world_group"，返回进程在world_group的rank id。

11.2.2.6 get_local_rank_id

函数原型

def get_local_rank_id(group="hccl_world_group")

功能说明

获取device在group中对应的local rank序号。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group，如果用户不传，默认为"hccl_world_group"。

返回值

int类型，返回device所在服务器内的local rank id序号。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
localRankId = get_local_rank_id(“myGroup”)
#rankId = 0
```

约束说明

- 必须在集合通信初始化完成之后调用。

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 11.2.2.1 create_group完成之后，调用此API获取进程在group中的local rank id。
- 如果传入"hccl_world_group"，返回进程在world_group的local rank id。

11.2.2.7 get_world_rank_from_group_rank

函数原型

def get_world_rank_from_group_rank(group, group_rank_id)

功能说明

从group rank id，获取该进程对应的world rank id。

参数说明

参数名	输入/输出	描述
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。
group_rank_id	输入	int类型。 进程在group中的rank id。

返回值

int类型，进程在"hccl_world_group"中的rank id。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
worldRankId = get_world_rank_from_group_rank (“myGroup”, 1)
#worldRankId = 8
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 11.2.2.1 create_group完成之后，调用此API转换group rank id到world rank id。

11.2.2.8 get_group_rank_from_world_rank

函数原型

def get_group_rank_from_world_rank(world_rank_id, group)

功能说明

从world rank id，获取该进程在group中的group rank id。

参数说明

参数名	输入/输出	描述
world_rank_id	输入	int类型。 进程在"hccl_world_group"中的rank id。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

int类型，正常返回进程在group中的rank id。

调用示例

```
from npu_bridge.npu_init import *
create_group(“myGroup”, 4, [0, 1, 2, 3])
groupRankId = get_group_rank_from_world_rank (8, “myGroup”)
#groupRankId = 1
```

约束说明

- 必须在集合通信初始化完成之后调用。
- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 11.2.2.1 create_group完成之后，调用此API转换world rank id到group rank id。

11.2.3 hccl.split.api

11.2.3.1 set_split_strategy_by_idx

函数原型

def set_split_strategy_by_idx(idxList, group="hccl_world_group")

功能说明

基于梯度的索引id，在集合通信group内设置反向梯度切分策略，实现allreduce的融合，用于进行集合通信的性能调优。

参数说明

参数名	输入/输出	描述
idxList	输入	list类型。 梯度的索引id列表。 ● 梯度的索引id列表需为非负，升序序列。 ● 梯度的索引id必须基于模型的总梯度参数个数去设置。索引id从0开始，最大值可通过以下方法获得： – 不调用梯度切分接口设置梯度切分策略进行训练，此时脚本会使用11.2.3.2 set_split_strategy_by_size中的默认梯度切分方式进行训练。 – 训练结束后，在INFO级别的host训练日志中搜索"segment result"关键字，可以得到梯度切分的分段的情况如: segment index list: [0,107] [108,159]。此分段序列中最大的数字（例如159）即总梯度参数索引最大的值。 **说明** 完整的训练过程可能出现日志覆盖情况，此时用户可以修改“/var/log/npu/conf/slog/slog.conf”中的配置项**LogAgentMaxFileNum**，提高Host侧保存日志文件的数量。或可以只进行一次迭代训练。 ● 梯度的切分最多支持8段。 ● 比如模型总共有160个参数会产生梯度，需要切分[0,20]、[21,100]和[101,159]三段，则可以设置为idxList=[20,100,159]。
group	输入	String类型。 group名称，可以为"hccl_world_group"或自定义group，默认为"hccl_world_group"。

返回值

无。

调用示例

```
from npu_bridge.npu_init import *
set_split_strategy_by_idx([20, 100, 159], “group”)
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。

- 若用户不调用梯度切分接口设置切分策略，则会按默认反向梯度切分策略切分。
  默认切分策略：按梯度数据量切分为2段，第一段数据量为96.54%，第二段数据量为3.46%（部分情况可能出现为一段情况）。

11.2.3.2 set_split_strategy_by_size

函数原型

def set_split_strategy_by_size(dataSizeList, group="hccl_world_group")

功能说明

基于梯度数据量百分比，在集合通信group内设置反向梯度切分策略，实现allreduce的融合，用于进行集合通信的性能调优。

参数说明

参数名	输入/输出	描述
dataSizeList	输入	list类型。 梯度参数数据量百分比列表。 ● 梯度的索引id列表需为非负，且梯度数据量序列总百分比之和必须为100。 ● 梯度的切分最多支持8段。 ● 比如模型总共有150M梯度数据量，需要切分90M，30M，30M三段，则可以设置dataSizeList =[60,20,20]。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为"hccl_world_group"或自定义group，默认为"hccl_world_group"。

返回值

无。

调用示例

```
from npu_bridge.npu_init import *
set_split_strategy_by_size([60, 20, 20], “group”)
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 在同时基于梯度数据量百分比及梯度的索引id设置反向梯度切分策略时，以基于梯度数据量百分比设置结果优先。
- 若用户不调用梯度切分接口设置切分策略，则会按默认反向梯度切分策略切分。

默认切分策略：ResNet50的最优切分位置，即按梯度数据量切分为2段，第一段数据量为96.54%，第二段数据量为3.46%。

11.2.4 npu_bridge.hccl.hccl_ops

11.2.4.1 allreduce

函数原型

```
def allreduce(tensor, reduction, fusion=1, fusion_id=-1, group =
"hccl_world_group")
```

功能说明

提供group内的集合通信allreduce功能，对所有节点的同名张量进行reduce操作。

参数说明

参数名	输入/输出	描述
tensor	输入	TensorFlow的tensor类型。 针对昇腾910 AI处理器，tensor支持的数据类型为int8, int32, float16, float32。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, int16, int32, float16, float32。
reduction	输入	String类型。 reduce的op类型，可以为”max”,”min”,”prod”和”sum”。 **说明** 针对昇腾910B AI处理器，当前版本“prod”操作不支持int16数据类型。
fusion	输入	int类型。 allreduce算子融合标识。 ● 0：不融合，该allreduce算子不和其他allreduce算子融合。 ● 1：按照梯度切分策略进行融合，默认为1。 ● 2：按照相同fusion_id进行融合。
fusion_id	输入	allreduce算子的融合id。 对相同fusion_id的allreduce算子进行融合。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

tensor：对输入tensor执行完allreduce操作之后的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
result = hccl_ops.allreduce(tensor, "sum")
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- allreduce上游节点暂不支持variable算子。
- 该接口要求输入tensor的数据量不超过8GB。
- allreduce算子融合只支持reduction为sum类型的算子。

11.2.4.2 allgather

函数原型

def allgather(tensor, rank_size, group = "hccl_world_group")

功能说明

提供group内的集合通信allgather功能，将所有节点的输入Tensor合并起来。

参数说明

参数名	输入/输出	描述
tensor	输入	TensorFlow的tensor类型。 针对昇腾910 AI处理器，支持的数据类型为int8, uint8, uint16, int32, uint32, int64, uint64, float16, float32, float64。 针对昇腾910B AI处理器，支持的数据类型为int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64。
rank_size	输入	int类型。 group内device的数量。 最大值为4096。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

tensor：对输入tensor执行完allgather操作之后的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
rank_size = 2
result = hccl_ops.allgather (tensor, rank_size)
```

约束说明

调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。

11.2.4.3 broadcast

函数原型

def broadcast(tensor, root_rank, fusion=2,fusion_id=0,group = "hccl_world_group")

功能说明

提供group内的集合通信broadcast功能，将root节点的数据广播到其他rank。

参数说明

参数名	输入/输出	描述
tensor	输入	TensorFlow的tensor类型，为list。 针对昇腾910 AI处理器，支持的数据类型为int8, uint8,uint16, int32, uint32, int64, uint64, float16, float32, float64。 针对昇腾910B AI处理器，支持的数据类型为int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64。
root_rank	输入	int类型。 作为root节点的rank_id，该id是group内的rank id。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。
fusion	输入	int类型。 broadcast算子融合标识。 ● 0：不融合，该broadcast算子不和其他broadcast算子融合。 ● 2：按照相同fusion_id进行融合。默认为2。 ● 其他值非法。
fusion_id	输入	broadcast算子的融合id。 对相同fusion_id的broadcast算子进行融合。

返回值

tensor：对输入tensor执行完broadcast操作之后的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
root = 0
inputs = [tensor]
result = hccl_ops.broadcast (inputs, root)
```

约束说明

调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。

11.2.4.4 reduce_scatter

函数原型

def reduce_scatter(tensor, reduction, rank_size, group = "hccl_world_group")

功能说明

提供group内的集合通信reducescatter功能，reduce操作由reduction参数指定。

参数说明

参数名	输入/输出	描述
tensor	输入	TensorFlow的tensor类型。 针对昇腾910 AI处理器，tensor支持的数据类型为int8, int32, float16, float32。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, int16, int32, float16, float32。 tensor的第一个维度的元素个数必需是rank size的整数倍。
reduction	输入	String类型。 reduce的op类型，可以为”max”,”min”,”prod”和”sum”。 **说明** 针对昇腾910B AI处理器，当前版本“prod”操作不支持int16数据类型。
rank_size	输入	int类型。 group内device的数量。 最大值：4096。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

tensor：对输入tensor执行完reducescatter操作之后的结果tensor。建议返回tensor的数据量为32Byte对齐，否则会导致性能下降。

调用示例

```
from npu_bridge.npu_init import *
rank_size = 2
result = hccl_ops. reduce_scatter (tensor, "sum", rank_size)
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 该接口要求输入tensor的数据量不超过8GB。

11.2.4.5 reduce

函数原型

def reduce(tensor, reduction, root_rank, fusion=0, fusion_id=-1, group="hccl_world_group")

功能说明

提供group内的集合通信reduce功能，对所有节点的同名张量进行reduce操作，并将结果输出至root_rank。

参数说明

参数名	输入/输出	描述
tensor	输入	TensorFlow的tensor类型。 针对昇腾910 AI处理器，tensor支持的数据类型为int8, int32, float16, float32。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, int16, int32, float16, float32。
reduction	输入	String类型。 reduce的op类型，可以为”max”,”min”,”prod”和”sum”。 **说明** 针对昇腾910B AI处理器，当前版本“prod”操作不支持int16数据类型。
root_rank	输入	int类型。 作为root节点的rank_id，该id是group内的rank id。

参数名	输入/输出	描述
fusion	输入	int类型。 reduce算子融合标识。 ● 0：不融合，该reduce算子不和其他reduce算子融合。 ● 2：按照相同fusion_id进行融合。
fusion_id	输入	reduce算子的融合id。 对相同fusion_id的reduce算子进行融合。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

tensor：对输入tensor执行完reduce操作之后的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
result = hccl_ops.reduce(tensor, "sum", 0)
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 该接口要求输入tensor的数据量不超过8GB。
- reduce算子融合只支持reduction为sum类型的算子。

11.2.4.6 send

函数原型

def send(tensor, sr_tag, dest_rank, group = "hccl_world_group")

功能说明

提供group内点对点通信发送数据的send功能。

参数说明

参数名	输入/输出	描述
tensor	输入	TensorFlow的tensor类型。 针对昇腾910 AI处理器，tensor支持的数据类型为int8, uint8, uint16, int32, uint32, int64, uint64, float16, float32, float64。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64。
sr_tag	输入	int类型。 消息标签，相同sr_tag的send/recv对可以收发数据。
dest_rank	输入	int类型。 数据的目标节点，该rank是group中的rank id。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

无。

调用示例

```
from npu_bridge.npu_init import *
sr_tag = 0
dest_rank = 1
hccl_ops. send (tensor, sr_tag, dest_rank)
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 如果点对点收发数据的rank属于不同的AI Server，要保证昇腾AI处理器的物理id相同
- 要求send和receive必须配对使用，且和图中其他算子要有依赖关系。

11.2.4.7 receive

函数原型

def receive(shape, data_type, sr_tag, src_rank, group = "hccl_world_group")

功能说明

提供group内点对点通信发送数据的receive功能。

参数说明

参数名	输入/输出	描述
shape	输入	接收tensor的shape。
data_type	输入	接收数据的数据类型。 针对昇腾910 AI处理器，tensor支持的数据类型为int8, uint8, uint16, int32, uint32, int64, uint64, float16, float32, float64。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64。
sr_tag	输入	int类型。 消息标签，相同sr_tag的send/recv对可以收发数据。
src_rank	输入	int类型。 接收数据的源节点，该rank是group中的rank id。
group	输入	String类型，最大长度为128字节，含结束符。 group名称，可以为用户自定义group或者"hccl_world_group"。

返回值

tensor：进行receive操作之后接收到对端的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
sr_tag = 0
src_rank = 0
tensor = hccl_ops. receive (tensor.shape, tensor.dtype, sr_tag, src_rank)
```

约束说明

- 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
- 点对点收发数据的rank要属于不同的Server，且昇腾AI处理器物理ID必须相同。
- 要求send和receive必须配对使用，且和图中其他算子要有依赖关系。

11.2.4.8 alltoallv

函数原型

def all_to_all_v(send_data, send_counts, send_displacements, recv_counts, recv_displacements, group="hccl_world_group")

功能说明

集合通信域alltoallv操作接口。向通信域内所有rank发送数据（数据量可以定制），并从所有rank接收数据。

集合通信网络中，每一个集合通信域都会占用**HCCL_BUFFSIZE**（默认值为200M）大小的缓冲区，用户可以根据通信数据量与业务模型数据量的大小，适当调整**HCCL_BUFFSIZE**的大小，提升网络执行性能。

参数说明

参数名	输入/输出	描述
send_data	输入	待发送的数据。 TensorFlow的tensor类型。 针对昇腾910 AI处理器，tensor支持的数据类型为 int8, uint8, uint16, int32, uint32, int64, uint64, float16, float32, float64 。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64。
send_counts	输入	发送的数据量，send_counts[i]表示本rank发给rank i的数据个数，基本单位是send_data数据类型对应的字节数。 例：send_data的数据类型为int32，send_counts[0]=1,send_count[1]=2，表示本rank给rank0发送1个int32类型的数据，给rank1发送2个int32类型的数据。 TensorFlow的tensor类型，tensor支持的数据类型为int64。
send_displacements	输入	发送数据的偏移量，send_displacements[i] 表示本rank发送给rank i的数据块相对于send_data的偏移量，基本单位是send_data数据类型对应字节数。 例： • send_data的数据类型为int32。 • send_counts[0]=1,send_counts[1]=2 • send_displacements[0]=0,send_displacements[1]=1 则表示本rank给rank0发送send_data上的第1个int32类型的数据，给rank1发送send_data上第2个与第3个int32类型的数据。 TensorFlow的tensor类型，tensor支持的数据类型为int64。
recv_counts	输入	接收的数据量，recv_counts[i]表示本rank从rank i收到的数据量。使用方法与send_counts类似。 TensorFlow的tensor类型。tensor支持的数据类型为int64。

参数名	输入/输出	描述
recv_displacements	输入	接收数据的偏移量，recv_displacements[i] 表示本rank发送给rank i数据块相对于recv_data的偏移量，基本单位是recv_data_type的字节数。使用方法与send_displacements类似。 TensorFlow的tensor类型。tensor支持的数据类型为int64。
group	输入	group名称，可以为用户自定义group或者"hccl_world_group"。 String类型，最大长度为128字节，含结束符。

返回值

recv_data：对输入tensor执行完all_to_all_v操作之后的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
result = hccl_ops.all_to_all_v(send_data_tensor, send_counts_tensor, send_displacements_tensor,
recv_counts_tensor, recv_displacements_tensor)
```

约束说明

1. 调用该接口的rank必须在当前接口入参group定义的范围内，不在此范围内的rank调用该接口会失败。
2. alltoallv的通信域需要满足：单server 1p、2p通信域要在同一个cluster内（server内0-3卡和4-7卡各为一个cluster），单server 4p、8p和多server通信域中rank要以cluster为基本单位，并且server间cluster选取要一致。

11.2.4.9 alltoallvc

函数原型

def all_to_all_v_c(send_data, send_count_matrix, rank, fusion=0, fusion_id=-1, group="hccl_world_group")

功能说明

集合通信域alltoallvc操作接口。向通信域内所有rank发送数据（数据量可以定制），并从所有rank接收数据。

alltoallvc通过输入参数send_count_matrix传入所有rank的收发参数，与**11.2.4.8 alltoallv**相比，性能更优。

集合通信网络中，每一个集合通信域都会占用**HCCL_BUFFSIZE**（默认值为200M）大小的缓冲区，用户可以根据通信数据量与业务模型数据量的大小，适当调整**HCCL_BUFFSIZE**的大小，提升网络执行性能。

参数说明

参数名	输入/输出	描述
send_data	输入	待发送的数据。 TensorFlow的tensor类型。 针对昇腾910 AI处理器，tensor支持的数据类型为 int8, uint8, uint16, int32, uint32, int64, uint64, float16, float32, float64 。 针对昇腾910B AI处理器，tensor支持的数据类型为int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64。
send_count_matrix	输入	所有rank的收发参数，send_count_matrix[i][j]表示rank i发给rank j的数据量，基本单位是send_data_type的字节数。 例：send_data_type为int32，send_count_matrix[0][1]=1，表示rank0给rank1发送1个int32。 TensorFlow的tensor类型。tensor支持的数据类型为int64。
rank	输入	int类型。 本节点的rank id，该id是group内的rank id。
fusion	输入	int类型。 alltoallvc算子融合标识，支持以下取值： ● 0：标识网络编译时不会对该算子进行融合，即该alltoallvc算子不和其他alltoallvc算子融合。 ● 2：网络编译时，会对alltoallvc算子按照相同的fusion_id进行融合，即“fusion_id”相同的alltoallv算子之间会进行融合。 **说明：**“fusion_id”相同的alltoallv算子之间融合有一定的前提，算子需要在相同的通信域内，并且算子发送数据的数据类型需要相同。 默认值为“0”。
fusion_id	输入	标识alltoallvc算子的融合id。 int类型。 开启alltoallvc算子融合功能的场景下，需要配置该参数，取值范围[0, 0x7fffffff]。
group	输入	group名称，可以为用户自定义group或者"hccl_world_group"。 String类型，最大长度为128字节，含结束符。

返回值

recv_data：对输入tensor执行完all_to_all_v_c操作之后的结果tensor。

调用示例

```
from npu_bridge.npu_init import *
result = hccl_ops.all_to_all_v_c(send_data_tensor, send_count_matrix_tensor, rank_tensor)
```

约束说明

1. 调用该接口的rank必须在当前接口入参group定义的范围内，输入的rank id有效且不重复，否则调用该接口会失败。
2. alltoallvc的通信域需要满足：单server 1p、2p通信域要在同一个cluster内（server内0-3卡和4-7卡各为一个cluster），单server 4p、8p和多server通信域中rank要以cluster为基本单位，并且server间cluster选取要一致。
3. 各节点输入的send_count_matrix要保持一致。

11.3 环境变量参考

11.3.1 基本信息

11.3.1.1 JOB_ID

功能描述

训练任务ID，用户自定义。仅支持大小写字母，数字，中划线，下划线。不建议使用以0开始的纯数字作为JOB_ID。

配置示例

```
export JOB_ID=10087
```

11.3.1.2 ASCEND_DEVICE_ID

功能描述

指定昇腾AI处理器的逻辑ID。取值范围[0,N-1]，默认为0。其中N为当前物理机/虚拟机/容器内的设备总数。

当用户需要将不同的模型通过同一个训练脚本在不同的Device上执行时，可以通过TF Adapter提供的运行参数session_device_id指定昇腾AI处理器的逻辑ID，该种场景下无需配置ASCEND_DEVICE_ID，如果同时配置，则以session_device_id配置的为准。

对于TensorFlow 2.6训练场景，当npu.open接口调用未传入设备ID时，会读取该环境变量。

配置示例

```
export ASCEND_DEVICE_ID=0
```


11.3.1.3 ENABLE_FORCE_V2_CONTROL

功能描述

如果输入是动态shape，由于tf.case/tf.cond/tf.while_loop这些API对应TensorFlow V1版本的控制流算子（例如Switch、Merge、Enter、LoopCond、NextIteration、Exit、ControlTrigger等）不支持动态shape，仅TensorFlow V2版本的控制流算子（例如If、Case、While、For、PartitionedCall等）支持动态shape，因此，如果用户的训练脚本中使用了这些API，需要将V1版本的控制流算子转换为V2版本，用于支持动态shape功能。另外，如果网络中的分支结构较多，采用V1版本的控制流算子可能导致流数超限，此时也需要将V1版本的控制流算子转换成V2版本算子解决。

配置示例

```
export ENABLE_FORCE_V2_CONTROL=1
```

11.3.1.4 NPU_DEBUG

功能描述

用于开启TF Adapter的Debug级别执行日志，该变量需要在import npu_device前设置。

- "1"或"true"：开启Debug级别执行日志
- "0"或"false"：关闭Debug级别执行日志

配置示例

```
export NPU_DEBUG=1
```

使用约束

此环境变量仅适用于使用Ascend adapter for TensorFlow 2.x的场景。

11.3.1.5 NPU_DUMP_GRAPH

功能描述

用于开启TF Adapter图Dump功能，该变量需要在import npu_device前设置。

- "1"或"true"：开启图Dump功能
- "0"或"false"：关闭图Dump功能

配置示例

```
export NPU_DUMP_GRAPH=1
```

使用约束

此环境变量仅适用于使用Ascend adapter for TensorFlow 2.x的场景。

11.3.1.6 NPU_ENABLE_PERF

功能描述

用于开启TF Adapter图耗时打印功能。

- "1"或"true"：开启图耗时打印功能
- "0"或"false"：关闭图耗时打印功能

配置示例

```
export NPU_ENABLE_PERF=1
```

11.3.1.7 NPU_LOOP_SIZE

功能描述

用于设置NPU上循环下沉的次数，该变量需要在import npu_device前设置。

配置示例

```
export NPU_LOOP_SIZE=32
```

使用约束

此环境变量仅适用于使用Ascend adapter for TensorFlow 2.x的场景。

11.3.2 图编译

11.3.2.1 DUMP_GE_GRAPH

功能描述

把整个流程中各个阶段的图描述信息打印到文件中，此环境变量控制dump图的内容多少。取值：

- 1：全量dump。
- 2：不含有权重等数据的基本版dump。
- 3：只显示节点关系的精简版dump。

配置示例

```
export DUMP_GE_GRAPH=1
```

使用约束

NA

11.3.2.2 DUMP_GRAPH_LEVEL

功能描述

把整个流程中各个阶段的图描述信息打印到文件中，此环境变量可以控制dump图的个数。取值：

- 1：dump所有图。
- 2：dump除子图外的所有图。
- 3：dump最后的生成图。
- 4：dump最早的生成图。

该环境变量只有在DUMP_GE_GRAPH开启时才生效，并且默认为2。

配置示例

```
export DUMP_GRAPH_LEVEL=1
```

11.3.2.3 DUMP_GRAPH_PATH

功能描述

指定DUMP图文件的保存路径，如果不配置默认保存在脚本执行目录下。

可配置为绝对路径或脚本执行目录的相对路径，配置的路径需要为已存在的目录，且执行用户具有读、写、可执行权限。

配置示例

```
export DUMP_GRAPH_PATH=/home/dumpgraph
```

11.3.2.4 ENABLE_NETWORK_ANALYSIS_DEBUG

功能描述

正常训练流程下，在计算图编译失败的时候，会终止流程，不会继续下剩余的图。配置该环境变量时（任意值），在图编译失败的时候，会始终返回成功，从而可以让TF Adapter持续下发计算图。

配置示例

```
export ENABLE_NETWORK_ANALYSIS_DEBUG=1
```

11.3.2.5 GE_USE_STATIC_MEMORY

功能描述

网络运行时使用的内存分配方式，支持以下取值：

- 0：动态分配内存，即按照实际大小动态分配。
- 2：动态扩展内存。训练与在线推理场景下，可以通过此取值实现同一session中多张图之间的内存复用，即以最大图所需内存进行分配。例如，假设当前执行图所

需内存超过前一张图的内存时，直接释放前一张图的内存，按照当前图所需内存重新分配。

默认值是0。为兼容历史版本配置，配置为“1”的场景下，系统会按照“2”动态扩展内存的方式进行处理。

说明

- 该环境变量在后续版本会废弃，建议开发者优先使用TF Adapter的配置参数static_memory_policy进行网络运行时内存分配方式的配置。
- 针对训练与在线推理场景，多张图并发执行时，不支持配置为“2”。
- 此环境变量与配置参数“static_memory_policy”不可同时使用，否则网络运行时会冲突。

配置示例

```
export GE_USE_STATIC_MEMORY=2
```

11.3.2.6 OP_NO_REUSE_MEM

功能描述

在内存复用场景下（默认开启内存复用），支持基于指定算子（节点名称/算子类型）单独分配内存。

通过该环境变量指定要单独分配的一个或多个节点，支持混合配置。配置多个节点时，中间通过逗号(“,”)隔开。

配置示例

- 基于节点名称配置
  ```
  export OP_NO_REUSE_MEM=gradients/logits/semantic/kernel/Regularizer/l2_regularizer_grad/Mul_1,resnet_v1_50/conv1_1/BatchNorm/AssignMovingAvg2
  ```
- 基于算子类型配置
  ```
  export OP_NO_REUSE_MEM=FusedMulAddN,BatchNorm
  ```
- 混合配置
  ```
  export OP_NO_REUSE_MEM=FusedMulAddN, resnet_v1_50/conv1_1/BatchNorm/AssignMovingAvg
  ```

11.3.2.7 SKT_ENABLE

功能描述

用于将多个算子的task任务融合成一个task任务下发，可以减少task调度耗时，缩短网络执行时间。

- 1：打开superkernel功能，执行super task融合流程。仅SoC场景下支持开启superkernel。
- 0：关闭superkernel功能，执行正常task下发流程。默认不支持。

配置示例

```
export SKT_ENABLE=1
```


11.3.2.8 HELP_CLUSTER

功能描述

进行跨多个进程的分布式训练时，需要向Helper配置集群的主调度进程列表。配置方式为环境变量，环境变量名为HELPER_CLUSTER。

- chief：作为Chief角色的主调度进程的IP和端口号；
- worker：所有Worker角色的主调度进程的IP列表（可以不指定端口）。

配置示例

```
export HELPER_CLUSTER={
    "chief": "10.174.28.82:34961",
    "worker": ["10.174.28.83:23581","10.174.28.84"]
}
```

11.3.2.9 MAX_COMPILE_CORE_NUMBER

功能描述

此环境变量用于指定图编译时可用的CPU核数。

该环境变量需要配置为整数，表示图编译时开启多线程，多线程数量与配置的CPU核数相同。

取值范围：1~CPU核数。

配置示例

```
export MAX_COMPILE_CORE_NUMBER=5
```

11.3.3 图执行

11.3.3.1 MAX_RUNTIME_CORE_NUMBER

功能描述

TensorFlow训练场景下，此环境变量用于指定Host侧图执行时可用的CPU核数。

该环境变量需要配置为整数，当取值大于等于3时，表示Host侧调度任务执行时开启多线程，多线程数量与配置的CPU核数相同。

配置示例

```
export MAX_RUNTIME_CORE_NUMBER=5
```

使用约束

1. 此环境变量仅用于TensorFlow训练场景。
2. 配置此环境变量时，Host侧可用于执行的CPU核数需要>=3。

11.3.4 算子编译

11.3.4.1 TE_PARALLEL_COMPILER

功能描述

算子最大并行编译进程数，当大于1时开启并行编译。

网络模型较大时，可通过配置此环境变量开启算子的并行编译功能。最大不超过cpu核数*80%/昇腾AI处理器个数，取值范围1~32，默认值是8。

配置示例

```
export TE_PARALLEL_COMPILER=8
```

11.3.4.2 ASCEND_MAX_OP_CACHE_SIZE

功能描述

启用算子编译缓存功能时，用于限制某个昇腾AI处理器下缓存文件夹的磁盘空间的大小，默认为500，单位为MB。

配置示例

```
export ASCEND_MAX_OP_CACHE_SIZE=500
```

11.3.4.3 ASCEND_REMAIN_CACHE_SIZE_RATIO

功能描述

启用算子编译缓存功能时，当编译缓存空间大小达到ASCEND_MAX_OP_CACHE_SIZE而需要删除旧的kernel文件时，需要保留缓存的空间大小比例，默认为50，单位为百分比。

配置示例

```
export ASCEND_REMAIN_CACHE_SIZE_RATIO=50
```

11.3.5 集合通信与分布式训练

11.3.5.1 RANK_TABLE_FILE

功能描述

分布式训练的昇腾AI处理器资源信息，请填写ranktable文件路径，包含文件路径和文件名。具体请参考**7.2.2.1 准备ranktable资源配置文件**。

配置示例

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```


11.3.5.2 RANK_ID

功能描述

训练进程在集合通信进程组中对应的rank标识序号。

针对昇腾910 AI处理器，当ranktable文件使用模板一时，和rank_id字段保持一致；当ranktable文件使用模板二时，和pod_name字段保持一致。

针对昇腾910B AI处理器，和rank_id字段保持一致。

配置示例

```
export RANK_ID=0
```

11.3.5.3 RANK_SIZE

功能描述

当前训练进程对应的Device在本集群大小，即集群Device的数量。

配置示例

```
export RANK_SIZE=2
```

11.3.5.4 CM_CHIEF_IP

功能描述

进行跨多个进程的分布式训练时，用户可以选择不使用ranktable文件，通过组合使用环境变量**11.3.5.4 CM_CHIEF_IP**、**11.3.5.5 CM_CHIEF_PORT**、**11.3.5.6 CM_CHIEF_DEVICE**、**11.3.5.7 CM_WORKER_SIZE**、**11.3.5.8 CM_WORKER_IP**的方式自动生成资源信息，完成集合通信组件初始化。

本环境变量“CM_CHIEF_IP”用于配置Master节点的监听Host IP。

格式为字符串，要求为常规IPv4或IPv6格式。

配置示例

```
export CM_CHIEF_IP = 192.168.1.1
```

使用约束

此环境变量不能与**11.3.5.1 RANK_TABLE_FILE**、**11.3.5.2 RANK_ID**、**11.3.5.3 RANK_SIZE**混合使用。

11.3.5.5 CM_CHIEF_PORT

功能描述

进行跨多个进程的分布式训练时，用户可以选择不使用ranktable文件，通过组合使用环境变量**11.3.5.4 CM_CHIEF_IP**、**11.3.5.5 CM_CHIEF_PORT**、**11.3.5.6 CM_CHIEF_DEVICE**、**11.3.5.7 CM_WORKER_SIZE**、**11.3.5.8 CM_WORKER_IP**的方式自动生成资源信息，完成集合通信组件初始化。

本环境变量“CM_CHIEF_PORT”用于配置Master节点的监听端口。

支持配置为字符串、数字，取值范围“0~65520”。

配置示例

```
export CM_CHIEF_PORT = 6000
```

使用约束

此环境变量不能与**11.3.5.1 RANK_TABLE_FILE**、**11.3.5.2 RANK_ID**、**11.3.5.3 RANK_SIZE**混合使用。

11.3.5.6 CM_CHIEF_DEVICE

功能描述

进行跨多个进程的分布式训练时，用户可以选择不使用ranktable文件，通过组合使用环境变量**11.3.5.4 CM_CHIEF_IP**、**11.3.5.5 CM_CHIEF_PORT**、**11.3.5.6 CM_CHIEF_DEVICE**、**11.3.5.7 CM_WORKER_SIZE**、**11.3.5.8 CM_WORKER_IP**的方式自动生成资源信息，完成集合通信组件初始化。

本环境变量“CM_CHIEF_DEVICE”用于指定Master节点中统计Server端集群信息的Device逻辑ID。

支持配置为字符串、数字，取值范围“0~AI Server内的最大Device数量”。

配置示例

```
export CM_CHIEF_DEVICE = 0
```

使用约束

此环境变量不能与**11.3.5.1 RANK_TABLE_FILE**、**11.3.5.2 RANK_ID**、**11.3.5.3 RANK_SIZE**混合使用。

11.3.5.7 CM_WORKER_SIZE

功能描述

进行跨多个进程的分布式训练时，用户可以选择不使用ranktable文件，通过组合使用环境变量**11.3.5.4 CM_CHIEF_IP**、**11.3.5.5 CM_CHIEF_PORT**、**11.3.5.6 CM_CHIEF_DEVICE**、**11.3.5.7 CM_WORKER_SIZE**、**11.3.5.8 CM_WORKER_IP**的方式自动生成资源信息，完成集合通信组件初始化。

本环境变量“CM_WORKER_SIZE”用于配置本次业务通信域Device的数量。

支持配置为字符串、数字，取值范围“0~4096”。

配置示例

```
export CM_WORKER_SIZE = 8
```


使用约束

此环境变量不能与11.3.5.1 RANK_TABLE_FILE、11.3.5.2 RANK_ID、11.3.5.3 RANK_SIZE混合使用。

11.3.5.8 CM_WORKER_IP

功能描述

进行跨多个进程的分布式训练时，用户可以选择不使用ranktable文件，通过组合使用环境变量11.3.5.4 CM_CHIEF_IP、11.3.5.5 CM_CHIEF_PORT、11.3.5.6 CM_CHIEF_DEVICE、11.3.5.7 CM_WORKER_SIZE、11.3.5.8 CM_WORKER_IP的方式自动生成资源信息，完成集合通信组件初始化。

本环境变量“CM_WORKER_IP”，用于配置当前Device和Master进行信息交换时所用的网卡IP。

格式为字符串，要求为常规IPv4或IPv6格式。

配置示例

```
export CM_WORKER_IP = 192.168.0.1
```

使用约束

此环境变量不能与11.3.5.1 RANK_TABLE_FILE、11.3.5.2 RANK_ID、11.3.5.3 RANK_SIZE混合使用。

11.3.5.9 HCCL_INTRA_PCIE_ENABLE

功能描述

用于配置Server内是否使用PCIe环路进行多卡间的通信。

需要与环境变量HCCL_INTRA_ROCE_ENABLE配合使用，控制Server内多卡之间的通信方式。下面对HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE的配置组合进行说明。

- HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE不配置或均配置为0，Server内采用PCIe环路进行多卡间的通信。
- HCCL_INTRA_PCIE_ENABLE为1，HCCL_INTRA_ROCE_ENABLE为0，Server内采用PCIe环路进行多卡间的通信，此组合方式为默认配置。
- HCCL_INTRA_PCIE_ENABLE为0，HCCL_INTRA_ROCE_ENABLE为1，Server内采用RoCE环路进行多卡间的通信。

  说明

  不支持HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE均配置为1。

配置示例

```
export HCCL_INTRA_PCIE_ENABLE=1
```


约束说明

此环境变量仅支持昇腾910 AI处理器下的**Atlas 300T Pro 训练卡**。

11.3.5.10 HCCL_INTRA_ROCE_ENABLE

功能描述

用于配置Server内是否使用RoCE环路进行多卡间的通信。

需要与环境变量HCCL_INTRA_PCIE_ENABL配合使用。下面对HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE的配置组合进行说明。

- HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE不配置或均配置为0，Server内采用PCIe环路进行多卡间的通信。
- HCCL_INTRA_PCIE_ENABLE为1，HCCL_INTRA_ROCE_ENABLE为0，Server内采用PCIe环路进行多卡间的通信，此组合方式为默认配置。
- HCCL_INTRA_PCIE_ENABLE为0，HCCL_INTRA_ROCE_ENABLE为1，Server内采用RoCE环路进行多卡间的通信。

  说明

  不支持HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE均配置为1。

配置示例

```
export HCCL_INTRA_ROCE_ENABLE=1
```

约束说明

此环境变量仅支持昇腾910 AI处理器下的**Atlas 300T Pro 训练卡**。

11.3.5.11 HCCL_CONNECT_TIMEOUT

功能描述

用于限制不同设备之间socket建链过程的超时等待时间，默认值120s。取值范围[120,7200]。

不同设备进程在集合通信初始化之前由于其他因素会导致执行不同步。该环境变量控制设备间的建链超时时间，在该配置时间内各设备进程等待其他设备建链同步。

配置示例

```
export HCCL_CONNECT_TIMEOUT=200
```

11.3.5.12 HCCL_WHITELIST_FILE

功能描述

指向HCCL通信白名单配置文件的路径，HCCL通信白名单配置文件格式为：

```
{ "host_ip": ["ip1", "ip2"], "device_ip": ["ip1", "ip2"] }
```

其中：

- device_ip为预留字段，当前版本暂不支持。
- ip格式为点分十进制。

注意

白名单ip需要指定为集群通信使用的有效ip。

配置示例

```
export HCCL_WHITELIST_FILE=/home/test/whitelist
```

11.3.5.13 HCCL_WHITELIST_DISABLE

功能描述

配置在使用HCCL时是否关闭通信白名单。

1：关闭白名单，无需校验HCCL通信白名单。

0：开启白名单，校验HCCL通信白名单。此时需要配置HCCL_WHITELIST_FILE。

缺省值为1，默认关闭白名单。

配置示例

```
export HCCL_WHITELIST_DISABLE=1
```

11.3.5.14 HCCL_IF_IP

功能描述

配置HCCL的初始化root通信网卡IP。

格式为字符串，要求为常规IPv4或IPv6格式，暂只支持Host网卡，且只能配置一个ip地址。

如果不配置，默认按照以下优先序选定Host通信网卡：**HCCL_SOCKET_IFNAME** > docker/lo以外网卡(网卡名字典序升序) > docker网卡 > lo网卡。

配置示例

```
export HCCL_IF_IP=10.10.10.1
```

11.3.5.15 HCCL_IF_BASE_PORT

功能描述

OPBase模式下，使用Host网卡进行HCCL初始化或集合通信计算时，可以通过该环境变量指定Host网卡起始端口号，配置后系统默认占用以该端口起始的16个端口。

默认值为60000，取值范围[0,65520]。

配置示例

```
export HCCL_IF_BASE_PORT = 50000
```

使用约束

分布式训练场景下，HCCL会使用Host服务器的部分端口进行集群信息收集，需要操作系统预留该部分端口。

- 若不通过HCCL_IF_BASE_PORT环境变量指定Host网卡起始端口，默认HCCL使用60000-60015端口，需要执行如下命令预留此范围的操作系统端口：
  ```
  sysctl -w net.ipv4.ip_local_reserved_ports=60000-60015
  ```
- 若通过HCCL_IF_BASE_PORT环境变量指定Host网卡起始端口，例如指定端口为50000，则HCCL使用50000-50015端口，需要执行如下命令预留此范围的操作系统端口：
  ```
  sysctl -w net.ipv4.ip_local_reserved_ports=50000-50015
  ```

11.3.5.16 HCCL_EXEC_TIMEOUT

功能描述

不同设备进程在分布式训练过程中存在卡间执行任务不一致的场景（如仅特定进程会保存checkpoint数据），通过该环境变量可控制设备间执行时同步等待的时间，在该配置时间内各设备进程等待其他设备执行通信同步。

- 针对昇腾910 AI处理器，单位为s，取值范围为：(0, 17340]，默认值为1800。

  需要注意：针对昇腾910 AI处理器，系统实际设置的超时时间 = 环境变量的取值先整除“68”，然后再乘以“68”，单位s。如果环境变量的取值小于68，则默认按照68s进行处理。

  例如，假设HCCL_EXEC_TIMEOUT = 600，则系统实际设置的超时时间为：600整除68乘以68 = 8*68 = 544s。
- 针对昇腾910B AI处理器，单位为s，取值范围为：[0, 2147483647]，默认值为1800，当配置为0时代表永不超时。

配置示例

```
export HCCL_EXEC_TIMEOUT=1800
```

11.3.5.17 HCCL_RDMA_TC

功能描述

用于配置RDMA网卡的traffic class。8bit无符号整型数据，bit2~7配置为DSCP值，bit0~1固定为零；默认值132。

配置示例

```
# DSCP配置为25
export HCCL_RDMA_TC = 100
```


11.3.5.18 HCCL_RDMA_SL

功能描述

用于配置RDMA网卡的service level，该值需要和网卡配置的PFC优先级保持一致。取值范围[0,7]；默认值4。

配置示例

```
# 优先级配置为3
export HCCL_RDMA_SL = 3
```

11.3.5.19 HCCL_ALLTOALL_Z_COPY

功能描述

用于配置AlltoAllv通信时，昇腾AI处理器内进行数据拷贝的方式。

支持以下取值：

- 0：通过多次拷贝、多次收发的方式进行数据通信。默认是0。
- 1：通过一次拷贝、多次收发的方式进行数据通信。

  此方式会根据通信数据量动态申请内存，若分配的内存不足，则会根据实际需求的数据量大小重新申请内存空间。**需要注意，在数据量较大且内存充足的场景下，开启此开关后性能会有所提升，但可能会出现内存耗尽问题，请用户谨慎配置；在数据量较小的场景下，开启此开关无明显性能提升效果。**

配置示例

```
# 环境变量算法配置格式
export HCCL_ALLTOALL_Z_COPY= "1"
```

11.3.5.20 HCCL_RDMA_TIMEOUT

功能描述

用于配置RDMA网卡的超时时间

- 针对昇腾910 AI处理器，取值范围为，取值范围为[5,24]，默认值为19。
- 针对昇腾910B AI处理器，取值范围为，取值范围为[5,20]，默认值为19。

超时时间最小值为：4.096 μs * 2 ^ timeout，其中timeout为该环境变量配置值。

配置示例

```
#超时时间配置为6
export HCCL_RDMA_TIMEOUT=6
```

11.3.5.21 HCCL_RDMA_RETRY_CNT

功能描述

用于配置RDMA网卡的重传次数，取值范围为[1,7]，默认值为7。

配置示例

```
#重传次数配置为5
export HCCL_RDMA_RETRY_CNT=5
```

11.3.5.22 HCCL_SOCKET_IFNAME

功能描述

配置HCCL的初始化root通信网卡名，HCCL可通过该网卡名获取Host IP，完成通信域创建。

支持以下格式配置：

- "eth,enp" ： 使用所有以eth或enp前缀的网卡，比如eth1,eth2,enp1…
- "=eth*,enp**" : 使用指定的eth*或enp**网卡
- "^eth,enp" ： 不使用任何以eth或enp为前缀的网卡
- "^=eth,enp" ： 不使用指定的eth*或enp**网卡

HCCL_SOCKET_IFNAME中可配置多个网卡，取最先匹配到的网卡作为root网卡。

说明

环境变量网**HCCL_IF_IP**的优先级高于HCCL_SOCKET_IFNAME。

如果用户未指定HCCL_IF_IP和HCCL_SOCKET_IFNAME，按照如下优先级选择：

docer/lo以外网卡(网卡名字典序升序) > docker 网卡 > lo网卡

配置示例

```
#使用eth0或endvnic的网卡
export HCCL_SOCKET_IFNAME==eth0,endvnic
```

11.3.5.23 HCCL_SOCKET_FAMILY

功能描述

配置HCCL的初始化root通信网卡使用的IP协议版本，支持配置为IPv4或IPv6版本。

- AF_INET：代表使用IPv4协议
- AF_INET6：代表使用IPv6协议

该环境变量缺省时，优先使用IPv4协议。

该环境变量需要与**HCCL_SOCKET_IFNAME**配合使用，当HCCL通过指定网卡名获取Host IP时，通过该环境变量指定使用网卡的IPv4地址或IPv6地址进行socket通信。

配置示例

```
export HCCL_SOCKET_FAMILY=AF_INET       #IPv4
export HCCL_SOCKET_FAMILY=AF_INET6      #IPv6
```


11.3.5.24 HCCL_BUFFSIZE

功能描述

此环境变量用于控制两个NPU之间共享数据的缓存区大小。单位为M，取值范围：[1，2048]，默认值是200M。

集合通信网络中，每一个HCCL通信域都会占用HCCL_BUFFSIZE大小的缓存区。若集群网络中存在较多的HCCL通信域，此缓存区占用量就会增多，可能存在影响模型数据正常存放的风险，此种场景下，可通过此环境变量减少通信域占用的缓存区大小；若业务的模型数据量较小，但通信数据量较大，则可通过此环境变量增大HCCL通信域占用的缓存区大小，提升数据通信效率。

此环境变量一般用于以下场景：

- 动态shape网络场景。
- 开发人员调用集合通信库HCCL的C语言接口进行框架对接的场景，HCCL库的C语言接口可参见《**集合通信OpBase接口参考**》。

配置示例

```
export HCCL_BUFFSIZE=200
```

11.3.6 性能数据采集

11.3.6.1 PROFILING_MODE

功能描述

是否开启Profiling功能。

- true：开启Profiling功能，从PROFILING_OPTIONS读取Profiling的采集选项。
- false或者不配置：关闭Profiling功能。

配置示例

```
export PROFILING_MODE=true
```

11.3.6.2 PROFILING_OPTIONS

功能描述

Profiling配置选项。

- output：Profiling采集结果文件保存路径。该参数指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。
  - 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。
  - 相对路径配置直接以目录名开始，例如：output。
- storage_limit：指定落盘目录允许存放的最大文件容量。当Profiling数据文件在磁盘中即将占满本参数设置的最大存储空间（剩余空间<=20MB）或剩余磁盘总空间即将被占满时（总空间剩余<=20MB），则将磁盘内最早的文件进行老化删除处理。

取值范围为[200, 4294967296]，单位为MB，例如**--storage-limit**=200MB，默认未配置本参数。

未配置本参数时，默认取值为Profiling数据文件存放目录所在磁盘可用空间的90%。

- training_trace：采集迭代轨迹数据开关，即训练任务及AI软件栈的软件信息，实现对训练任务的性能分析，重点关注前后向计算和梯度聚合更新等相关数据。当采集正向和反向算子数据时该参数必须配置为on。
- task_trace：采集任务信息数据以及Host与Device之间、Device间的同步异步内存复制时延，即昇腾AI处理器HWTS数据，分析任务开始、结束等信息。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。当训练profiling mode开启即采集训练Profiling数据时，配置task_trace为on的同时training_trace也必须配置为on。
- hccl：控制hccl数据采集开关，可选on或off，默认为off。
- aicpu：采集AICPU算子的详细信息，如：算子执行时间、数据拷贝时间等。取值on/off，默认为off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。
- fp_point：指定训练网络迭代轨迹正向算子的开始位置，用于记录前向计算开始时间戳。配置值为指定的正向第一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入；也可直接配置为空，由系统自动识别正向算子的开始位置，例如"fp_point":""。
- bp_point：指定训练网络迭代轨迹反向算子的结束位置，记录后向计算结束时间戳，BP_POINT和FP_POINT可以计算出正反向时间。配置值为指定的反向最后一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入；也可直接配置为空，由系统自动识别反向算子的结束位置，例如"bp_point":""。
- aic_metrics：AI Core和AI Vector Core的硬件信息，取值如下：
  - ArithmeticUtilization：各种计算类指标占比统计；
  - PipeUtilization： 计算单元和搬运单元耗时占比，该项为默认值；
  - Memory：外部内存读写类指令占比；
  - MemoryL0：内部内存读写类指令占比；
  - MemoryUB：内部内存读写指令占比；
  - ResourceConflictRatio：流水线队列类指令占比。
  - L2Cache：读写cache命中次数和缺失后重新分配次数。

  仅昇腾910B AI处理器支持AI Vector Core数据及L2Cache开关。

  说明

  支持自定义需要采集的寄存器，例如：
  "aic_metrics":"**Custom:***0x49,0x8,0x15,0x1b,0x64,0x10*"。
  - Custom字段表示自定义类型，配置为具体的寄存器值，取值范围为[0x1, 0x6E]。
  - 配置的寄存器数最多不能超过8个，寄存器通过“,”区分开。
  - 寄存器的值支持十六进制或十进制。
- l2：控制L2采样数据的开关，可选on或off，默认为off。仅昇腾310P AI处理器、昇腾910 AI处理器、昇腾910B AI处理器支持该参数。
- msproftx：控制msproftx用户和上层框架程序输出性能数据的开关，可选on或off，默认值为off。

Profiling开启msproftx功能之前，需要在程序内调用msproftx相关接口来开启程序的Profiling数据流的输出，详细操作请参见《**应用软件开发指南（C&C++）**》“扩展更多特性>Profiling性能数据采集”章节。

- task_time：控制任务调度耗时以及算子耗时的开关。涉及在ai_stack_time、task_time、op_summary、op_statistic文件中输出相关耗时数据。可选on或off，默认为on。
- runtime_api：控制runtime api性能数据采集开关，可选on或off，默认为off。可采集runtime-api性能数据，包括Host与Device之间、Device间的同步异步内存复制时延等。
- sys_hardware_mem_freq：DDR、HBM带宽采集频率、LLC的读写带宽数据采集频率以及acc_pmu数据和SOC传输带宽信息采集频率，取值范围为[1,100]，默认值50，单位hz。

  说明

  昇腾910B AI处理器不支持DDR采集；仅昇腾910 AI处理器和昇腾910B AI处理器支持HBM采集；仅昇腾910B AI处理器支持acc_pmu数据和SOC传输带宽信息采集。
- llc_profiling：LLC Profiling采集事件，可以设置为：
  - 昇腾310 AI处理器，可选capacity（采集AI CPU和Control CPU的LLC capacity数据）或bandwidth（采集LLC bandwidth），默认为capacity。
  - 昇腾310P AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。
  - 昇腾910 AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。
  - 昇腾910B AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。
- sys_io_sampling_freq：NIC、ROCE采集频率。取值范围为[1,100]，默认值100，单位hz。仅昇腾310 AI处理器、昇腾910 AI处理器、昇腾910B AI处理器支持NIC采集频率；仅昇腾910 AI处理器、昇腾910B AI处理器支持ROCE采集频率。
- sys_interconnection_freq：集合通信带宽数据（HCCS）、PCIe数据采集频率以及片间传输带宽信息采集频率。取值范围为[1,50]，默认值50，单位hz。仅昇腾910 AI处理器、昇腾910B AI处理器支持HCCS采集频率；仅昇腾310P AI处理器、昇腾910 AI处理器、昇腾910B AI处理器支持PCIe采集频率；仅昇腾910B AI处理器支持片间传输带宽信息采集频率。
- dvpp_freq：DVPP采集频率。取值范围为[1,100]，默认值50，单位hz。
- instr_profiling_freq：AI Core和AI Vector的带宽和延时采集频率。取值范围[300,30000]，默认值1000，单位cycle。仅昇腾910B AI处理器支持该参数。

  说明

  instr_profiling_freq开关与training_trace、task_trace、hccl、aicpu、fp_point、bp_point、aic_metrics、l2、task_time、runtime_api

  互斥，无法同时执行。
- host_sys：Host侧性能数据采集开关。取值包括cpu和mem，可选其中的一项或多项，选多项时用英文逗号隔开，例如"host_sys": "cpu,mem"。
- host_sys_usage：采集Host侧系统及所有进程的CPU和内存数据。取值包括cpu和mem，可选其中的一项或多项，选多项时用英文逗号隔开。
- host_sys_usage_freq：配置Host侧系统和所有进程CPU、内存数据的采集频率。取值范围为[1,50]，默认值50，单位hz。

说明

- 除动态shape场景外的其他场景，fp_point、bp_point为自动配置项，无需用户手动配置。动态shape场景不支持自动配置fp/bp，需要用户手动设置。
- 在线推理支持task_trace和aicpu，不支持training_trace。

配置示例

```
export PROFILING_OPTIONS='{"output":"/tmp/
profiling","training_trace":"on","task_trace":"on","fp_point":"","bp_point":"","aic_metrics":"PipeUtilization"}'
```

11.3.7 Log

11.3.7.1 ASCEND_PROCESS_LOG_PATH

功能描述

设置日志落盘路径。

日志存储时如果不存在该目录，会自动创建该目录；如果存在则直接存储。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_PROCESS_LOG_PATH**命令可以查看环境变量设置的路径。
- 若环境变量未配置或配置为空，表示采用日志默认输出方式。

配置示例

```
export ASCEND_PROCESS_LOG_PATH=$HOME/log/
```

可指定日志落盘路径为任意有读写权限的目录。

11.3.7.2 ASCEND_SLOG_PRINT_TO_STDOUT

功能描述

是否开启日志打屏。开启后，日志将不会保存在log文件中，而是将产生的日志直接打屏显示。

取值为：

- 0：关闭日志打屏，即日志采用默认输出方式，将日志保存在log文件中。
- 1：开启日志打屏。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_SLOG_PRINT_TO_STDOUT**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志默认输出方式。

配置示例

```
export ASCEND_SLOG_PRINT_TO_STDOUT=1
```

11.3.7.3 ASCEND_GLOBAL_LOG_LEVEL

功能描述

设置应用类日志的全局日志级别及各模块日志级别。

取值为：

- 0：对应DEBUG级别。
- 1：对应INFO级别。
- 2：对应WARNING级别。
- 3：对应ERROR级别。
- 4：对应NULL级别，不输出日志。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_GLOBAL_LOG_LEVEL**命令可以查看环境变量设置的日志级别。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志配置文件中设置的日志级别。

配置示例

```
export ASCEND_GLOBAL_LOG_LEVEL=1
```

11.3.7.4 ASCEND_MODULE_LOG_LEVEL

功能描述

设置应用类日志的各模块日志级别。

取值为：

- 0：对应DEBUG级别。
- 1：对应INFO级别。
- 2：对应WARNING级别。
- 3：对应ERROR级别。
- 4：对应NULL级别，不输出日志。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_MODULE_LOG_LEVEL**命令可以查看环境变量设置的日志级别。
- 若环境变量未配置或配置为非法值或配置为空，表示采用全局日志级别。

配置示例

```
export ASCEND_MODULE_LOG_LEVEL=TDT=0
```

11.3.7.5 ASCEND_GLOBAL_EVENT_ENABLE

功能描述

设置应用类日志是否开启Event日志。

取值为：

- 0：关闭Event日志。
- 1：开启Event日志。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_GLOBAL_EVENT_ENABLE**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志配置文件中设置的日志级别。

配置示例

```
export ASCEND_GLOBAL_EVENT_ENABLE=0
```

11.3.7.6 ASCEND_LOG_DEVICE_FLUSH_TIMEOUT

功能描述

业务进程退出前，系统有2000ms的默认延时将Device侧应用类日志回传到Host侧，超时后业务进程退出。未回传到Host侧的日志直接在Device侧落盘（路径为/var/log/npu/slog）。

Device侧应用类日志回传到Host侧的延时时间可以通过环境变量**ASCEND_LOG_DEVICE_FLUSH_TIMEOUT**进行设置。

环境变量取值范围为[0, 180000]，单位为ms，默认值为2000。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_LOG_DEVICE_FLUSH_TIMEOUT**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志默认值。
- 如果业务进程不需要等待所有Device侧应用类日志回传到Host侧，可将环境变量设置为0。
- 针对业务进程退出后仍然有Device侧应用类日志未回传到Host侧这种情况，建议设置更大的延时时间，具体调节的大小可以根据device-app-*pid*的日志内容进行判断。

配置示例

```
export ASCEND_LOG_DEVICE_FLUSH_TIMEOUT=5000
```


11.3.7.7 ASCEND_HOST_LOG_FILE_NUM

功能描述

EP场景下，设置应用类日志目录（plog和device-*id*）下存储每个进程日志文件的数量。

环境变量取值范围为[1, 1000]，默认值为10。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_HOST_LOG_FILE_NUM**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志默认值。
- 如果plog和device-*id*日志目录下存储的单个进程的日志文件数量超过设置的值，将会自动删除最早的日志。另外每个plog-*pid*_*.log或device-*pid*_*.log日志文件的大小最大固定为20MB。如果超过该值，会生成新的日志文件。

配置示例

```
export ASCEND_HOST_LOG_FILE_NUM=20
```


12 FAQ

12.1 compat.v1 模式下使用工具迁移 Horovod 脚本后，执行失败

工具在compat.v1模式下迁移Horovod脚本时，会自动替换相关Horovod接口，同时删除原始脚本中的Horovod相关包引用，此时对于原始脚本中的某些特殊写法，可能会导致关联代码报错，建议手工修改适配。例如：

迁移前：

```
from horovod.common.util import env

def main(_):
  tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.INFO)
  with env(HOROVOD_STALL_CHECK_TIME_SECONDS="300"):
      hvd.init()
```

迁移后：

```
None

def main(_):
  tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.INFO)
  with env(HOROVOD_STALL_CHECK_TIME_SECONDS="300"):
      hvd.init()
```

包引用删除后导致找不到env模块，因此可以通过重新添加包引用解决问题：

```
from horovod.common.util import env

def main(_):
  tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.INFO)
  with env(HOROVOD_STALL_CHECK_TIME_SECONDS="300"):
      hvd.init()
```

同时也建议根据实际代码逻辑，判断是否保留这些代码。

12.2 安装 7.3.0 版本 gcc

以下步骤请在root用户下执行。

步骤1 下载gcc-7.3.0.tar.gz，下载地址为**https://mirrors.tuna.tsinghua.edu.cn/gnu/gcc/gcc-7.3.0/gcc-7.3.0.tar.gz**。

步骤2 安装gcc时候会占用大量临时空间，所以先执行下面的命令清空/tmp目录：

```
rm -rf /tmp/*
```

步骤3 安装依赖。

centos/bclinux执行如下命令安装。

```
yum install bzip2
```

ubuntu/debian执行如下命令安装。

```
apt-get install bzip2
```

步骤4 编译安装gcc。

1. 进入gcc-7.3.0.tar.gz源码包所在目录，解压源码包，命令为：
   ```
   tar -zxvf gcc-7.3.0.tar.gz
   ```
2. 进入解压后的文件夹，执行如下命令下载gcc依赖包：
   ```
   cd gcc-7.3.0
   ./contrib/download_prerequisites
   ```
   如果执行上述命令报错，需要执行如下命令在“gcc-7.3.0/”文件夹下下载依赖包：
   ```
   wget http://gcc.gnu.org/pub/gcc/infrastructure/gmp-6.1.0.tar.bz2
   wget http://gcc.gnu.org/pub/gcc/infrastructure/mpfr-3.1.4.tar.bz2
   wget http://gcc.gnu.org/pub/gcc/infrastructure/mpc-1.0.3.tar.gz
   wget http://gcc.gnu.org/pub/gcc/infrastructure/isl-0.16.1.tar.bz2
   ```
   下载好上述依赖包后，重新执行以下命令：
   ```
   ./contrib/download_prerequisites
   ```
   如果上述命令校验失败，需要确保依赖包为一次性下载成功，无重复下载现象。
3. 执行配置、编译和安装命令：
   ```
   ./configure --enable-languages=c,c++ --disable-multilib --with-system-zlib --prefix=/usr/local/linux_gcc7.3.0
   make -j15    # 通过grep -w processor /proc/cpuinfo|wc -l查看cpu数，示例为15，用户可自行设置相应参数。
   make install
   ```

> **注意**
>
> 其中“--prefix”参数用于指定linux_gcc7.3.0安装路径，用户可自行配置，但注意不要配置为“/usr/local”及“/usr”，因为会与系统使用软件源默认安装的gcc相冲突，导致系统原始gcc编译环境被破坏。示例指定为“/usr/local/linux_gcc7.3.0”。

步骤5 配置环境变量。

当用户执行训练时，需要用到gcc升级后的编译环境，因此要在训练脚本中配置环境变量，通过如下命令配置。

```
export LD_LIBRARY_PATH=${install_path}/lib64:${LD_LIBRARY_PATH}
```

其中${install_path}为**3.**中配置的gcc7.3.0安装路径，本示例为“/usr/local/gcc7.3.0/”。

说明

本步骤为用户在需要用到gcc升级后的编译环境时才配置环境变量。

----结束

