CANN
6.3.RC2

TensorFlow 1.15 在线推理指南

文档版本	01
发布日期	2023-08-18

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 概述

简介

在线推理是在AI框架内执行推理的场景，例如在TensorFlow框架上，加载模型后，通过session.run执行推理。相比于离线推理场景，使用在线推理可以方便将原来基于Tensorflow框架做推理的应用快速迁移到昇腾AI处理器，适用于数据中心推理场景。

读者对象

本文档适用于将训练好的TensorFlow网络模型加载到昇腾AI处理器进行在线推理的AI工程师。

掌握以下经验和技能可以更好地理解本文档：

- 熟练的Python语言编程能力
- 熟悉TensorFlow API
- 对机器学习、深度学习有一定的了解

支持的芯片型号

昇腾910 AI处理器

昇腾910B AI处理器

2 环境准备

请参见《**CANN 软件安装指南**》进行环境搭建。

- 请依据实际在下列场景中选择一个进行在线推理依赖包安装路径的环境变量设置。
  - 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推理任务)。
    ```
    # 以root用户安装toolkit包
    . /usr/local/Ascend/ascend-toolkit/set_env.sh
    # 以非root用户安装toolkit包
    . ${HOME}/Ascend/ascend-toolkit/set_env.sh
    ```
  - 场景二：昇腾设备安装部署软件包Ascend-cann-nnae。
    ```
    # 以root用户安装nnae包
    . /usr/local/Ascend/nnae/set_env.sh
    # 以非root用户安装nnae包
    . ${HOME}/Ascend/nnae/set_env.sh
    ```
- 设置tfplugin插件包的环境变量。
  - 以root用户安装tfplugin包。
    ```
    . /usr/local/Ascend/tfplugin/set_env.sh
    ```
  - 以非root用户安装tfplugin包。
    ```
    . ${HOME}/Ascend/tfplugin/set_env.sh
    ```
- 若运行环境中存在多Python3版本时，需要在环境变量中配置Python的安装路径。如下配置以安装Python3.7.5为例，可根据实际修改。
  ```
  export PATH=/usr/local/python3.7.5/bin:$PATH
  export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH

  export JOB_ID=10087
  ```
- 当前脚本所在路径，例如：
  ```
  export PYTHONPATH="$PYTHONPATH:/root/models"
  ```

说明

上述仅列出推理进程启动必配的环境变量，更多介绍请参考**7.4 环境变量参考**。若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**7.5 安装7.3.0版本gcc**。

3 编译配置

在线推理脚本中需要进行必要的编译配置：

```
from npu_bridge.estimator import npu_ops
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig

config = tf.ConfigProto()
custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
# 配置1：选择在昇腾AI处理器上执行推理
custom_op.parameter_map["use_off_line"].b = True

# 配置2：在线推理场景下建议保持默认值force_fp16，使用float16精度推理，以获得较优的性能
custom_op.parameter_map["precision_mode"].s = tf.compat.as_bytes("force_fp16")

# 配置3：图执行模式，推理场景下请配置为0，训练场景下为默认1
custom_op.parameter_map["graph_run_mode"].i = 0

# 配置4：关闭remapping和MemoryOptimizer
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF
```

在线推理几个关键配置项为：

- use_off_line配置为True，表示在昇腾AI处理器上执行推理。
- precision_mode建议保持默认，使用float16精度推理，以获得较优的性能。
- graph_run_mode配置为0。

在线推理当前支持的所有配置项，请参考**7.1 session配置**。

4 样例参考

样例代码

使用在线推理需要充分考虑到sess.run首次执行时需要对模型进行编译和优化，耗时会增多。在编写推理应用时，应尽量保证应用生命周期内不频繁初始化。本例中，我们使用将推理过程封装到Classifier类中，以便应用可以控制Classifier对象的生命周期。

样例代码infer_from_pb.py：

```
# 通过加载已经训练好的pb模型，执行推理
import tensorflow as tf
import os
import argparse
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig
import npu_bridge
import time
import numpy as np

def parse_args():
    '''
    用户自定义模型路径、输入、输出
    '''
    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--batchsize', default=1,
                        help="""batchsize""")
    parser.add_argument('--model_path', default='pb/resnet50HC.pb',
                        help="""pb path""")
    parser.add_argument('--image_path', default='image-50000',
                        help="""the data path""")
    parser.add_argument('--label_file', default='val_label.txt',
                        help="""label file""")
    parser.add_argument('--input_tensor_name', default='input_data:0',
                        help="""input_tensor_name""")
    parser.add_argument('--output_tensor_name', default='resnet_model/final_dense:0',
                        help="""output_tensor_name""")
    args, unknown_args = parser.parse_known_args()
    if len(unknown_args) > 0:
        for bad_arg in unknown_args:
            print("ERROR: Unknown command line arg: %s" % bad_arg)
        raise ValueError("Invalid command line arg(s)")
    return args

def read_file(image_name, path):
    '''
    从标签文件中读取图片的相关信息
    '''
    with open(path, 'r') as cs:
        rs_list = cs.readlines()
        for name in rs_list:
```



```
            if image_name in str(name):
                num = str(name).split(" ")[1]
                break
    return int(num) + 1

def normalize(inputs):
    '''
    图像归一化
    '''
    mean = [121.0, 115.0, 100.0]
    std =  [70.0, 68.0, 71.0]
    mean = tf.expand_dims(tf.expand_dims(mean, 0), 0)
    std = tf.expand_dims(tf.expand_dims(std, 0), 0)
    inputs = inputs - mean
    inputs = inputs * (1.0 / std)
    return inputs

def image_process(image_path, label_file):
    '''
    对输入图像进行一定的预处理
    '''
    imagelist = []
    labellist = []
    images_count = 0
    for file in os.listdir(image_path):
        with tf.Session().as_default():
            image_file = os.path.join(image_path, file)
            image_name = image_file.split('/')[-1].split('.')[0]
            #images preprocessing
            image= tf.gfile.FastGFile(image_file, 'rb').read()
            img = tf.image.decode_jpeg(image, channels=3)
            bbox = tf.constant([0.1,0.1,0.9,0.9])
            img = tf.image.crop_and_resize(img[None, :, :, :], bbox[None, :], [0], [224, 224])[0]
            img = tf.clip_by_value(img, 0., 255.)
            img = normalize(img)
            img = tf.cast(img, tf.float16)
            images_count = images_count + 1
            img = img.eval()
            imagelist.append(img)
            tf.reset_default_graph()

            # read image label from label_file
            label = read_file(image_name, label_file)
            labellist.append(label)

    return np.array(imagelist), np.array(labellist),images_count


class Classifier(object):
    #set batchsize:
    args = parse_args()
    batch_size = int(args.batchsize)

    def __init__(self):

        # 昇腾AI处理器模型编译和优化配置
        config = tf.ConfigProto()
        custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
        custom_op.name = "NpuOptimizer"
        # 配置1： 选择在昇腾AI处理器上执行推理
        custom_op.parameter_map["use_off_line"].b = True
        # 配置2：在线推理场景下建议保持默认值force_fp16，使用float16精度推理，以获得较优的性能
        custom_op.parameter_map["precision_mode"].s = tf.compat.as_bytes("force_fp16")
        # 配置3：图执行模式，推理场景下请配置为0，训练场景下为默认1
        custom_op.parameter_map["graph_run_mode"].i = 0
        # 配置4：关闭remapping和MemoryOptimizer
        config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
        config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF
        # 加载模型，并指定该模型的输入和输出节点
```



```
        args = parse_args()
        self.graph = self.__load_model(args.model_path)
        self.input_tensor = self.graph.get_tensor_by_name(args.input_tensor_name)
        self.output_tensor = self.graph.get_tensor_by_name(args.output_tensor_name)

        # 由于首次执行session run会触发模型编译，耗时较长，可以将session的生命周期和实例绑定
        self.sess = tf.Session(config=config, graph=self.graph)

    def __load_model(self, model_file):
        """
        加载静态图
        """
        with tf.gfile.GFile(model_file, "rb") as gf:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(gf.read())

        with tf.Graph().as_default() as graph:
            tf.import_graph_def(graph_def, name="")

        return graph

    def do_infer(self, batch_data):
        """
        执行推理
        """
        out_list = []
        total_time = 0
        i = 0
        for data in batch_data:
            t = time.time()
            out = self.sess.run(self.output_tensor, feed_dict={self.input_tensor: data})
            if i > 0:
                total_time = total_time + time.time() - t
            i = i + 1
            out_list.append(out)
        return np.array(out_list), total_time

    def batch_process(self, image_data, label_data):
        """
        批处理
        """
        # 获取当前输入数据的批次信息，自动将数据调整为固定批次
        n_dim = image_data.shape[0]
        batch_size = self.batch_size

        # 如果数据不足以用于整个批次，则需要进行数据补齐
        m = n_dim % batch_size
        if m < batch_size and m > 0:
            # 不足部分的数据以0进行填充补齐
            pad = np.zeros((batch_size - m, 224, 224, 3)).astype(np.float32)
            image_data = np.concatenate((image_data, pad), axis=0)

        # 定义可以被分成的最小批次
        mini_batch = []
        mini_label = []
        i = 0
        while i < n_dim:
            # Define the Minis that can be divided into several batches
            mini_batch.append(image_data[i: i + batch_size, :, :, :])
            mini_label.append(label_data[i: i + batch_size])
            i += batch_size

        return mini_batch, mini_label

def main():
    args = parse_args()
    top1_count = 0
    top5_count = 0
    # 数据预处理
```



```
    tf.reset_default_graph()
    print("########NOW Start Preprocess!!!#########")
    images, labels, images_count = image_process(args.image_path, args.label_file)
    # 批处理
    print("########NOW Start Batch!!!#########")
    classifier = Classifier()
    batch_images, batch_labels= classifier.batch_process(images, labels)
    # 开始执行推理
    print("########NOW Start inference!!!#########")
    batch_logits, total_time = classifier.do_infer(batch_images)
    # 计算精度
    batchsize = int(args.batchsize)
    total_step = int(images_count / batchsize)
    print("########NOW Start Compute Accuracy!!!#########")
    for i in range(total_step):
        top1acc = tf.reduce_sum(tf.cast(tf.equal(tf.argmax(batch_logits[i], 1), batch_labels[i]), tf.float32))
        top5acc = tf.reduce_sum(tf.cast(tf.nn.in_top_k(batch_logits[i], batch_labels[i], 5), tf.float32))
        with tf.Session().as_default():
            tf.reset_default_graph()
            top1_count += top1acc.eval()
            top5_count += top5acc.eval()

    print('+---------------------------------------+')
    print('the correct num is {}, total num is {}.'.format(top1_count, total_step * batchsize))
    print('Top1 accuracy:', top1_count / (total_step * batchsize) * 100)
    print('Top5 accuracy:', top5_count / (total_step * batchsize) * 100)
    print('images number = ', total_step * batchsize)
    print('images/sec = ', (total_step * batchsize) / total_time)
    print('+---------------------------------------+')

if __name__ == '__main__':
    main()
```

样例执行

以ResNet50模型为例，执行在线推理样例。

步骤1 下载预训练模型。

1. 在Ascend Gitee仓 > ModelZoo-TensorFlow的**resnet50_for_TensorFlow**路径下，参考README下载已经训练好的原始模型文件“resnet50_tensorflow_1.7.pb”。
2. 准备好样例数据集imagenet2012，您可以从imagenet官方网站**https://www.image-net.org/**获取数据集。

说明

下载的预训练模型文件只支持输入batchsize为1的推理场景，用户也可根据实际，将训练产生的“.ckpt”模型文件转换冻结为自定义batchsize的“.pb”文件。

步骤2 编辑推理脚本。

创建“infer_from_pb.py”模型脚本文件，并参考**样例代码**写入相关代码。

步骤3 执行推理。

参考**2 环境准备**设置环境变量，并执行命令：

python3 infer_from_pb.py --model_path=./resnet50_tensorflow_1.7.pb --image_path=/data/dataset/imagenet2012/val --label_file=/data/dataset/imagenet2012/val_label.txt --input_tensor_name=Placeholder:0 --output_tensor_name=fp32_vars/dense/BiasAdd:0

说明

上述为样例输入，用户可根据实际修改传入参数。用户在不确定pb模型文件节点名时，可参考**7.6 读取pb模型文件的节点名称**获取模型的输入输出节点名。

在计算在线推理性能时，由于首轮推理会进行算子和图编译，耗时较长，因此从第二轮开始计时。

----结束

5 更多特性

5.1 混合精度

概述

混合精度推理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推理的过程，并减少内存使用和存取，从而可以推理更大的神经网络，同时又能基本保持使用float32推理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**。

设置精度模式

下面以sess.run模式下，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。

```
import tensorflow as tf
from npu_bridge.estimator import npu_ops
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig
```



```
config = tf.ConfigProto()

custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name =  "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["precision_mode_v2"].s = tf.compat.as_bytes("mixed_float16")
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF  #关闭remap开关

with tf.Session(config=config) as sess:
  print(sess.run(cost))
```

修改混合精度黑白灰名单

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。下面介绍两种方法：

- （推荐）方法一：通过modify_mixlist指定需要修改的混合精度黑白灰算子名单。

  ```
  from npu_bridge.npu_init import *
  config = tf.ConfigProto()
  custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
  custom_op.name =  "NpuOptimizer"
  custom_op.parameter_map["use_off_line"].b = True
  custom_op.parameter_map["precision_mode_v2"].s = tf.compat.as_bytes("mixed_float16")
  custom_op.parameter_map["modify_mixlist"].s = tf.compat.as_bytes("/home/test/ops_info.json")
  config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
  config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF
  with tf.Session(config=config) as sess:
    print(sess.run(cost))
  ```

  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"
    ]
  }
}
```

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。

- 方法二：直接修改算子信息库。

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/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>0-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。

5.2 混合计算

概述

昇腾AI处理器默认采用计算全下沉模式，即所有的计算类算子全部在Device侧执行。

混合计算模式作为计算全下沉模式的补充，将部分不可离线编译下沉执行的算子留在前端框架中在线执行，用于提升昇腾AI处理器支持Tensorflow的适配灵活性。

使能混合计算

用户可通过配置项mix_compile_mode开启混合计算功能：

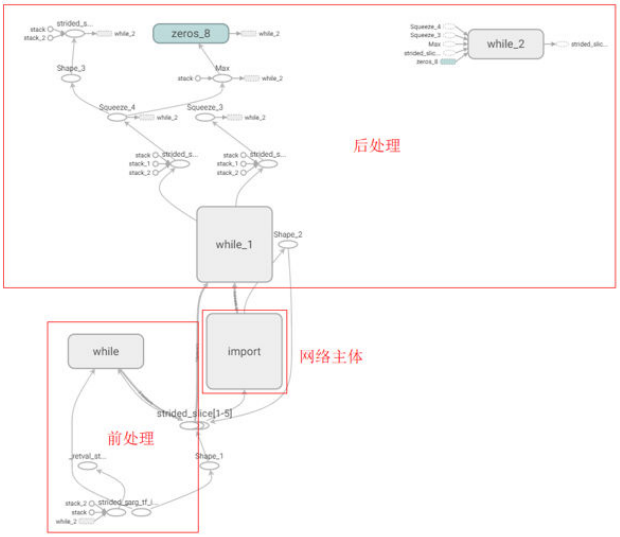
```
import tensorflow as tf
from npu_bridge.estimator import npu_ops
from npu_bridge.estimator.npu import npu_scope
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig

config = tf.ConfigProto()
custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name =  "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["mix_compile_mode"].b =  True   # 开启混合计算
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
```

通过上述配置，系统会将默认不下沉的算子留在前端框架执行，除此之前，还可以自行指定下沉或不下沉的首尾算子，系统会将首尾范围内的算子全部做下沉或不下沉处理。

指定下沉或不下沉的首尾算子

指定下沉或不下沉的首尾算子，系统会将首尾范围内的算子全部做下沉或不下沉处理。例如，对于yolo_v3网络，希望将网络主体部分和后处理部分下沉到昇腾AI处理器执行，前处理部分留在前端框架执行。



可以通过指定首尾算子，将in_nodes, out_nodes范围内的算子全部下沉到昇腾AI处理器执行：

```
import tensorflow as tf
from npu_bridge.estimator import npu_ops
```



```
from npu_bridge.estimator.npu import npu_scope
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig

config = tf.ConfigProto()
custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name =  "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["mix_compile_mode"].b =  True  # 开启混合计算
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF

all_graph_iop = []
in_nodes = []
out_nodes = []

in_nodes.append('import/conv2d_1/con')
in_nodes.append('while_1/strided_sli')
in_nodes.append('while_1/strided_sli')
in_nodes.append('concat')
in_nodes.append('while_1/Const')
in_nodes.append('while_1/Const_4')
in_nodes.append('while_1/Const_2')
in_nodes.append('zeros_7')
in_nodes.append('Const_6')
in_nodes.append('zeros_4')
in_nodes.append('ConstantFolding/whi')
out_nodes.append('strided_slice_13')
all_graph_iop.append([in_nodes, out_nodes])
custom_op.parameter_map['in_out_pair'].s = tf.compat.as_bytes(str(all_graph_iop))
```

也可以反过来实现，将前处理部分留在前端框架执行，即将in_nodes, out_nodes范围内的算子全部留在前端框架执行：

```
import tensorflow as tf
from npu_bridge.estimator import npu_ops
from npu_bridge.estimator.npu import npu_scope
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig

config = tf.ConfigProto()
custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name =  "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["mix_compile_mode"].b =  True  # 开启混合计算
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF

all_graph_iop = []
in_nodes = []
out_nodes = []

in_nodes.append('_arg_tf_image_string_0_0')
in_nodes.append('strided_slice_1/stack')
in_nodes.append('strided_slice_1/stack_1')
in_nodes.append('Const')
in_nodes.append('zeros')
in_nodes.append('zeros_1')
in_nodes.append('Const_2')
out_nodes.append('strided_slice_2')
out_nodes.append('strided_slice_3')
out_nodes.append('strided_slice_4')
all_graph_iop.append([in_nodes, out_nodes])
custom_op.parameter_map['in_out_pair_flag'].b = False
custom_op.parameter_map['in_out_pair'].s = tf.compat.as_bytes(str(all_graph_iop))
```


5.3 性能数据采集与分析

概述

在线推理过程中支持采集Profiling性能数据，然后借助Profiling工具进行数据分析，从而准确定位系统的软、硬件性能瓶颈，提高性能分析的效率，通过针对性的性能优化方法，以最小的代价和成本实现业务场景的极致性能。

当前支持采集的Profiling数据主要包括：

- task_trace：任务轨迹数据，即昇腾AI处理器HWTS/AICore的硬件信息，分析任务开始、结束等信息。
- aicpu_trace：采集aicpu数据增强的Profiling数据。

默认不采集Profiling性能数据，如需采集，请参考本节内容修改脚本。

sess.run 模式下开启 Profiling 数据采集

sess.run模式下，通过session配置项profiling_mode、profiling_options开启Profiling数据采集，具体参数配置要求请参考**Profiling**。

```
custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["profiling_mode"].b = True
custom_op.parameter_map["profiling_options"].s = tf.compat.as_bytes('{"output":"/tmp/profiling","task_trace":"on"}')
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF  #关闭remap开关

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

说明

允许在没有昇腾AI处理器直接连接的服务器上启动训练或推理进程，通过分布式部署能力进行计算任务的远程部署。在该场景下，需要用户指定**output**路径为“/var/log/npu/”，用于保存Profiling采集结果文件。

环境变量方式

除了以上方式，用户还可以修改启动脚本中的环境变量，开启Profiling采集功能。

```
export PROFILING_MODE=true
export PROFILING_OPTIONS='{"output":"/tmp/profiling","task_trace":"on"}'
```

环境变量详细配置要求请参考**7.4 环境变量参考**。

Profiling 数据分析

训练结束后，切换到result_path目录下，可查看到Profiling数据，您可以通过Profiling工具解析数据，具体参考《**性能分析工具使用指南**》。

5.4 Dump

概述

系统支持在线推理过程中采集算子的Dump数据，即算子的输入/输出结果，之后用户可以通过精度比对工具进行算子精度分析。默认推理过程中不采集算子的Dump数据，如需采集，请参考如下方式修改。

sess.run 模式下采集 Dump 数据

sess.run模式下，通过session配置enable_dump、dump_path、dump_mode配置Dump参数：

```
config = tf.ConfigProto()

custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name =  "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
# 是否开启dump功能。
custom_op.parameter_map["enable_dump"].b = True
# dump数据存放路径。
custom_op.parameter_map["dump_path"].s = tf.compat.as_bytes("/tmp/")
# dump模式，默认仅dump算子输出数据，还可以dump算子输入数据，取值：input/output/all。
custom_op.parameter_map["dump_mode"].s = tf.compat.as_bytes("all")
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF

with tf.Session(config=config) as sess:
  print(sess.run(cost))
```

查看 Data Dump 数据

如果采集到了Data Dump数据，则会在*{dump_path}/{time}/{deviceid}/{model_name}/{model_id}/{data_index}*目录下生成dump文件，如举例的“/home/HwHiAiUser/output/20200808163566/0/ge_default_20200808163719_121/11/0”目录。同时在脚本当前目录生成GE图文件，例如ge_proto_xxxxx_Build.txt。

存放路径及文件命名规则：

- dump_path：用户配置的dump数据存放路径。例如“/home/HwHiAiUser/output”。
- time：时间戳，例如20200317020343。
- deviceid：Device设备ID号。
- model_name：子图名称。model_name层可能存在多个文件夹，算子dump数据对应计算图目录下的数据。
- model_id：子图ID。
- data_index：迭代数，用于保存对应迭代的dump数据。如果指定了dump_step，则data_index和dump_step一致；如果不指定dump_step，则data_index序号从0开始计数，每dump一个迭代的数据，序号递增1。
- dump文件：命名规则如*{op_type}.{op_name}.{taskid}.{timestamp}*
- 如果model_name、op_type、op_name出现了“.”、“/”、“\”、空格时，会转换为下划线表示。

Dump 数据分析

请参考《**精度比对工具使用指南**》。

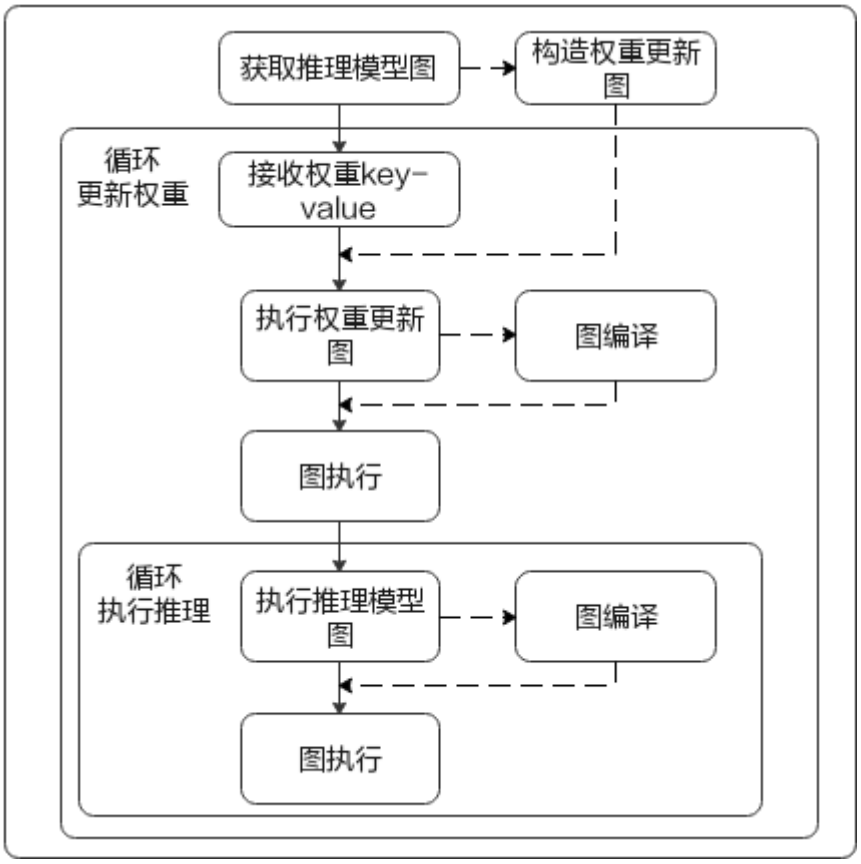
5.5 权重更新

背景

推理过程的同时，训练服务器不停训练得到新的权重，而推理部分希望直接更新最新的权重，不希望再走一遍保存pb、编译成离线模型、然后执行离线模型的流程，这种场景可以采用在线推理模式， 直接更新权重。

整体流程

图 5-1 权重更新流程示意图



如图5-1所示，支持循环地更新权重与执行推理。如果多次执行过程batch_size等不发生变化，理论上虚线部分只执行一次。主要流程：

1. 获取在线推理模型和权重信息，例如从ckpt文件中加载，实际更新用的权重则来自于外部的key-value；
2. 构造权重更新图，包含需要更新的变量和赋值算子，如Assigin；
3. 执行权重更新图， 将第1步获取到的key-value更新到对应权重；
4. 执行在线推理流程。

样例参考

```
import tensorflow as tf
import time
import numpy as np
from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig
from npu_bridge.estimator import npu_ops

class TestPolicy(object):
    def __init__(self, ckpt_path):
        # NPU模型编译和优化配置
        # ------------------------------------------------------------------------------
        config = tf.compat.v1.ConfigProto()
        custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
        custom_op.name = "NpuOptimizer"
        # 配置1： 选择在Ascend NPU上执行推理
        custom_op.parameter_map["use_off_line"].b = True

        # 配置2：在线推理场景下建议保持默认值force_fp16，使用float16精度推理，以获得较优的性能
        custom_op.parameter_map["precision_mode"].s = tf.compat.as_bytes("force_fp16")

        # 配置3：关闭remapping
        config.graph_options.rewrite_options.remapping = RewriterConfig.OFF

        # 配置4：设置graph_run_mode为推理
        custom_op.parameter_map["graph_run_mode"].i = 0

        # 配置5：指定AICORE引擎的并行度为4
        custom_op.parameter_map["stream_max_parallel_num"].s = tf.compat.as_bytes("AIcoreEngine:4")
        # ------------------------------------------------------------------------------

        # 初始化操作
        self.sess = tf.compat.v1.Session(config=config)
        self.ckpt_path = ckpt_path
        # 加载推理模型
        self.load_graph()
        self.graph = self.sess.graph

    def load_graph(self):
        '''
        从ckpt加载推理模型，获取权重的信息，并构造权重更新图
        '''
        saver = tf.compat.v1.train.import_meta_graph(self.ckpt_path + '.meta')
        saver.restore(self.sess, self.ckpt_path)
        self.vars = tf.compat.v1.trainable_variables()
        self.var_placeholder_dict = {}
        self.var_id_to_name = {}
        self.update_op = []
        for id, var in enumerate(self.vars):
            self.var_placeholder_dict[var.name] = tf.compat.v1.placeholder(var.dtype, shape=var.get_shape(),
name=("PlaceHolder_" + str(id)))
            self.var_id_to_name[id] = var.name
            self.update_op.append(tf.compat.v1.assign(var, self.var_placeholder_dict[var.name]))
        self.update_op = tf.group(*self.update_op)

        # 实际场景的权重key-value来自于训练服务器，这里保存一份ckpt中的权重仅用于示例
        self.key_value = self.get_dummy_weights_for_test()

    def unload(self):
        '''
        关闭session，释放资源
        '''
        print("====== start to unload ======")
        self.sess.close()

    def get_dummy_weights_for_test(self):
        '''
        从ckpt中获取权重的信息，并构造权重更新图
        :return: 权重 key-value
        :NOTES: 实际场景的权重key-value来自于训练服务器，这里返回的ckpt中的权重仅用于示例
```



```
        '''
        weights_data = self.sess.run(self.vars)
        weights_key_value = {}
        for id, var in enumerate(weights_data):
            weights_key_value[self.var_id_to_name[id]] = var
        return weights_key_value

    def get_weights_key_value(self):
        '''
        获取权重key-value
        :return: 权重 key-value
        :NOTES: 实际场景的权重key-value来自于训练服务器，这里返回保存的权重仅用于示例
        '''
        return self.key_value

    def update_weights(self):
        '''
        更新权重
        '''
        feed_dict = {}
        weights_key_value = self.get_weights_key_value()
        for key, weight in weights_key_value.items():
            feed_dict[self.var_placeholder_dict[key]] = weight
        self.sess.run(self.update_op, feed_dict=feed_dict)

    def infer(self, input_image):
        '''
        执行推理流程
        :param: input_image 输入的数据，示例中为图像数据
        :return: output 推理结果， 示例中为labels
        '''
        image = self.graph.get_operation_by_name('Placeholder').outputs[0]
        label_output = self.graph.get_operation_by_name('accuracy/ArgMax').outputs[0]
        output = self.sess.run([label_output], feed_dict={image: input_image})
        return output


def prepare_input_data(batch):
    '''
    推理的输入数据
    :param: batch 数据batch_size
    :return: 推理数据
    '''
    image = 255 * np.random.random([batch, 784]).astype('float32')
    return image

if __name__ == "__main__":
    batch_size = 16
    ckpt_path = "./mnist_deep_model/mnist_deep_model"
    policy = TestPolicy(ckpt_path)
    update_count = 10
    for i in range(update_count):
        update_start = time.time()
        policy.update_weights()
        update_consume = time.time() - update_start
        print("Update weight time cost: {} ms".format(update_consume * 1000))
        test_count = 20
        input_data = prepare_input_data(batch_size)
        start_time = time.time()
        for i in range(test_count):
            output = policy.infer(input_data)
            print("result is ", output)
        time_consume = (time.time() - start_time) / (test_count)
        print("Inference average time cost: {} ms \n".format(time_consume * 1000))

    policy.unload()
    print("====== end of test ======")
```


5.6 动态分档

背景介绍

当前系统支持在用户脚本中指定配置动态档位信息，从而支持动态输入的场景。

动态分档支持整图分档和子图分档两种模式，使用方式存在差异。

- 整图分档：使用session参数设置分档信息，输入可以为dataset方式、placeholder方式，或者两种混合方式。对于混合输入，当前仅支持其中一种为动态变化的场景。
- 子图分档：调用scope接口指定分档范围，分档信息需调用指定接口设置到scope内的输入算子上，支持一张图中指定多个分档子图。

使用约束

整图分档和子图分档不可混用，同时配置时仅生效整图分档。

整图分档：

- 用户在脚本中设置的input shape的输入顺序要和实际data节点的name字母序保持一致，比如有三个输入：label、data、mask，则

  input_shape输入顺序应该为data、label、mask：

  ```
  "data:1,1,40,-1;label:1,-1;mask:-1,-1"
  ```

- 对于dataset输入时，get_next接口不允许指定name，否则系统无法识别该输入为dataset输入还是placeholder输入。
- 不支持同一张图中有多个get_next节点。
- 该功能不能和混合计算一起使用。
- 对于dataset和placeholder混合输入，当前仅支持其中一种为动态变化的场景。

子图分档：

- 与整图分档不同，子图场景下用户脚本中设置input shape仅支持index方式指定，不支持name方式。
- 如果分档scope内有多个节点设置了分档信息，每个节点上的档位数必须相等。

sess.run 模式下设置整图动态分档

```
custom_op =  config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name =  "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["input_shape"].s = tf.compat.as_bytes("data:1,1,40,-1;label:
1,-1;mask:-1,-1")
custom_op.parameter_map["dynamic_dims"].s = tf.compat.as_bytes("20,20,1,1;40,40,2,2;80,60,4,4")
custom_op.parameter_map["dynamic_node_type"].i = 0
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF  #关闭remap开关

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

input_shape表示网络的输入shape信息，以上配置表示网络中有三个输入，输入的name分别为data，label，mask，各输入的shape分别为（1,1,40,-1），（1,-1），（-1,-1），其中-1表示该维度上为动态档位，需要通过dynamic_dims设置动态档位参数。

dynamic_dims表示输入的对应维度的档位信息。档位中间使用英文分号分隔，每档中的dim值与input_shape参数中的-1标识的参数依次对应，input_shape参数中有几个-1，则每档必须设置几个维度。结合input_shape信息，dynamic_dims配置为"20,20,1,1;40,40,2,2;80,60,4,4"的含义如下：

有三个“;”，表示输入shape支持三个档位，每个档位中的值对应输入shape中的“-1”的取值：

- 第0档：data(1,1,40,20)，label(1,20)，mask(1,1)
- 第1档：data(1,1,40,40)，label(1,40)，mask(2,2)
- 第2档：data(1,1,40,80)，label(1,60)，mask(4,4)

说明

对于多输入场景，例如有三个输入时，如果只有第二个第三个输入是动态档位，第一个输入为固定输入时，仍需要将固定输入shape填入input_shape字段内，例如：

```
custom_op.parameter_map["input_shape"].s = tf.compat.as_bytes("data:1,1,40,1;label:1,-1;mask:-1,-1")
custom_op.parameter_map["dynamic_dims"].s = tf.compat.as_bytes("20,1,1;40,2,2;60,4,4")
```

dynamic_node_type用于指定动态输入的节点类型。0：dataset输入为动态输入；1：placeholder输入为动态输入。当前不支持dataset和placeholder输入同时为动态输入。

sess.run 模式下设置子图动态分档

```
ori_image = tf.placeholder(dtype=tf.uint8, shape=(None, None, 3), name="ori_image")
resized_img, h_scale, w_scale = npu_cpu_ops.ocr_detection_pre_handle(img=ori_image)

with npu_scope.subgraph_multi_dims_scope(0):
    image_expand = tf.expand_dims(resized_img, axis=0)
    util.set_op_input_tensor_multi_dims(image_expand, "0:-1,-1,3", "480,480;960,960;1920,1920")
    image_tensor_fp32 = tf.cast(image_expand, dtype=tf.float32)
    image_tensor_nchw = tf.transpose(image_tensor_fp32, [0, 3, 1, 2])
    score, kernel = npu_onnx_graph_op([image_tensor_nchw], [tf.float32, tf.uint8],
model_path="text_detection.onnx", name="detection")

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

subgraph_multi_dims_scope的入参0表示scope索引，如果图中出现多个scope，该值需要保证唯一。

set_op_input_tensor_multi_dims有三个输入：

- 第一个输入为scope范围内输入节点的任意输出tensor；
- 第二个输入配置该tensor对应的节点的所有输入shape，每个输入间用";"分隔，input索引与shape间用":"分隔，shape的多个dim间用","分隔；
- 第三个输入为档位信息，每个档位间以";"分隔，档位内dim间用","分隔，dim数与input_shape中"-1"的数量一致。

以上配置表示分档范围的输入节点是expand_dims，该算子有1个输入，为动态shape：(-1,-1,3)，可分为3个档位：

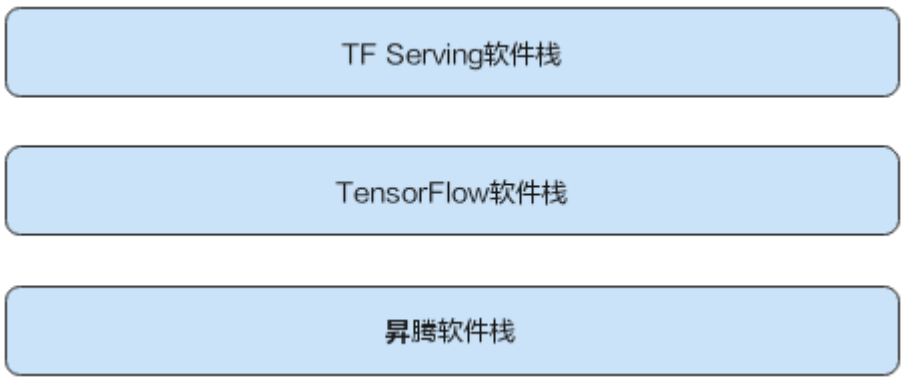
- 第0档：(480,480,3)
- 第1档：(960,960,3)
- 第2档：(1920,1920,3)

6 基于 TF-Serving 部署在线推理业务

6.1 概述

TensorFlow Serving是一款针对机器学习模型的灵活，高性能的服务系统，专为生产环境设计。使用SavedModel格式模型，提供RESTful API + gRPC对外接口，强依赖TensorFlow源码，官网链接：**www.tensorflow.org/tfx/guide/serving?hl=zh-cn**。

使用TensorFlow Serving可以轻松部署新算法和实验，同时保持相同的服务器体系结构和API。TensorFlow Serving实际上是封装了TensorFlow，提供服务化的能力，并且从性能考虑核心代码都采用C++，依赖的TensorFlow也是C++版本。



本文介绍如何基于TF Adapter和TF Serving源码进行编译，以便TF Serving通过Tensorflow可加载TF Adapter插件，最终使用昇腾AI处理器进行在线推理。

以下操作请在昇腾AI处理器所在环境执行。

6.2 环境准备

- 安装开发套件包Ascend-cann-toolkit_*{version}*_linux-*{arch}*.run或深度学习引擎包Ascend-cann-nnae_*{version}*_linux-*{arch}*.run，具体请参见《**CANN 软件安装指南**》。
- 安装框架插件包Ascend-cann-tfplugin_*{version}*_linux-*{arch}*.run，具体请参见《**CANN 软件安装指南**》的“安装开发环境 > 在昇腾设备上安装 > 安装框架插件包”章节。
- 安装如**表6-1**所示相关依赖。

表 6-1 依赖信息

依赖包	版本限制
gcc	8.4及以上版本 用户可使用“cc --version”及“c++ --version” 命令确定当前系统使用的gcc版本。
g++	
zip	无特定版本要求。
unzip	
libtool	
automake	
Python	3.7.5
TensorFlow	1.15.0
tensorflow-serving-api	1.15.0
future	无特定版本要求。
bazel	0.24.1及以上版本
CMake	3.14.0及以上版本
swig	• 若操作系统架构为“aarch64”，软件安装版本需大于或等于3.0.12。 • 若操作系统架构为“x86_64”，软件安装版本需大于或等于4.0.1 。

说明

- gcc和g++的版本一定要保持一致，否则TF Serving源码编译时可能会报错。
- bazel编译安装可参考**6.6.1 源码安装0.24.1版本bazel**。
- CMake编译安装可参考**6.6.2 源码安装3.14.0版本CMake**。
- 如果在安装swig软件包时出现无法安装软件包问题，参考**6.7.1 无法安装swig软件包**解决。
- gcc、g++、zip、unzip、libtool和automake软件包使用apt或yum进行安装，TensorFlow、tensorflow-serving-api和future软件包使用pip3方式安装。

6.3 TF Serving 集成 TF Adapter

以安装用户HwHiAiUser为例，介绍TF Serving集成TF Adapter的操作方法，实际操作时请根据实际路径进行替换。本文中举例路径均需要确保安装用户具有读或读写权限。

步骤1 下载**TF Serving源码**。

TF Serving需与TensorFlow保持版本一致，将源码包上传至服务器任意路径下。

步骤2 进入源码包所在路径，执行如下命令解压并进入TF Serving源码包。

```
unzip 1.15.0.zip
cd serving-1.15.0/
```

步骤3 添加TF Serving第三方依赖。

1. 执行如下命令，在“serving-1.15.0/third_party”目录下创建“tf_adapter”文件夹并进入。
   ```
   cd third_party/
   mkdir tf_adapter
   cd tf_adapter
   ```
2. 执行如下命令，在“tf_adapter”文件夹下拷贝存放“libpython3.7m.so.1.0”文件，并创建软链接。
   ```
   cp /usr/local/python3.7.5/lib/libpython3.7m.so.1.0 .
   ln -s libpython3.7m.so.1.0 libpython3.7m.so
   ```
3. 执行如下命令，在“tf_adapter”文件夹下拷贝存放“_tf_adapter.so”文件，并将“_tf_adapter.so”文件名修改为“lib_tf_adapter.so”。
   ```
   cp /home/HwHiAiUser/Ascend/tfplugin/latest/python/site-packages/npu_bridge/_tf_adapter.so .
   mv _tf_adapter.so lib_tf_adapter.so
   ```

步骤4 编译空的libtensorflow_framework.so、_pywrap_tensorflow_internal.so文件。

1. 在“tf_adapter”文件夹下，执行如下命令。
   ```
   vim CMakeLists.txt
   ```
2. 写入如下内容保存。
   ```
   file(TOUCH ${CMAKE_CURRENT_BINARY_DIR}/stub.c)
   add_library(_pywrap_tensorflow_internal SHARED ${CMAKE_CURRENT_BINARY_DIR}/stub.c)
   add_library(tensorflow_framework SHARED ${CMAKE_CURRENT_BINARY_DIR}/stub.c)
   ```
3. 执行**:wq!**命令保存文件并退出。
4. 执行如下命令，编译出空的.so文件。
   ```
   mkdir temp
   cd temp
   cmake ..
   make
   mv lib_pywrap_tensorflow_internal.so ../_pywrap_tensorflow_internal.so
   mv libtensorflow_framework.so ../libtensorflow_framework.so
   cd ..
   ln -s libtensorflow_framework.so libtensorflow_framework.so.1
   ```


5. 配置环境命令。
```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$(pwd)
```

步骤5 创建BUILD文件并添加内容。

1. 执行以下命令，在“tf_adapter”文件夹下创建BUILD文件。
```
vim BUILD
```
2. 写入如下内容。
```
licenses(["notice"])  # BSD/MIT.

cc_import(
    name = "tf_adapter",
    shared_library = "lib_tf_adapter.so",
    visibility = ["//visibility:public"]
)

cc_import(
    name = "tf_python",
    shared_library = "libpython3.7m.so",
    visibility = ["//visibility:public"]
)
```
3. 执行**:wq!**命令保存文件并退出。

步骤6 修改“serving-1.15.0/tensorflow_serving/model_servers/”路径下的BUILD文件，在“cc_binary”中添加如下加粗内容。
```
cc_binary(
    name = "tensorflow_model_server",
    stamp = 1,
    visibility = [
        ":testing",
        "//tensorflow_serving:internal",
    ],
    deps = [
        ":tensorflow_model_server_main_lib",
        "//third_party/tf_adapter:tf_adapter",
        "//third_party/tf_adapter:tf_python",
        "@org_tensorflow//tensorflow/compiler/jit:xla_cpu_jit",
    ],
)
```

步骤7 编译TF Serving。

在TF Serving安装目录“serving-1.15.0”下执行如下命令，编译TF Serving。
```
bazel --output_user_root=/opt/tf_serving build -c opt --cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0" tensorflow_serving/model_servers:tensorflow_model_server
```

其中“–output_user_root”参数指定了TF Serving的安装路径。请根据实际进行指定。

说明

- 如果编译过程中遇到依赖包下载失败问题，可手动下载，参考**6.6.4 手动下载TF Serving编译依赖包**。
- 如果在TF Serving编译过程中出现“builtins”依赖模块查询失败问题，参考**6.7.2 TF Serving编译时提示缺少builtins**解决。

步骤8 建立软连接。

执行如下命令，创建TF Serving软件包的软链接。
```
ln -s /opt/tf_serving/{tf_serving_ID}/execroot/tf_serving/bazel-out/xxx-opt/bin/tensorflow_serving/model_servers/tensorflow_model_server /usr/local/bin/tensorflow_model_server
```

- *{tf_serving_ID}*为一串如“063944eceea3e72745362a0b6eb12a3c”的无规则字符。请根据实际进行填写。

- *xxx-opt*为工具自动生成文件夹，具体显示请以实际为准。

----结束

6.4 启动 TF Serving

以安装用户HwHiAiUser为例，介绍启动TF Serving的操作方法，实际操作时请根据实际路径进行替换。本文中举例路径均需要确保安装用户具有读或读写权限。

步骤1 在安装用户目录下新建“tf_serving_test”文件夹，并在文件夹中创建配置文件config.cfg并添加如下内容。具体字段可参考**7.1 session配置**。

```
platform_configs {
key: "tensorflow"
  value {
    source_adapter_config {
      [type.googleapis.com/tensorflow.serving.SavedModelBundleSourceAdapterConfig] {
        legacy_config {
          session_config {
            graph_options {
              rewrite_options {
                custom_optimizers {
                  name: "NpuOptimizer"
                  parameter_map: {
                    key: "use_off_line"
                    value: {
                      b: true
                    }
                  }
                  parameter_map: {
                    key: "mix_compile_mode"
                    value: {
                      b: true
                    }
                  }
                  parameter_map: {
                    key: "graph_run_mode"
                    value: {
                      i: 0
                    }
                  }
                  parameter_map: {
                    key: "precision_mode"
                    value: {
                      s: "force_fp16"
                    }
                  }
                }
                remapping: OFF
              }
            }
          }
        }
      }
    }
  }
}
```

步骤2 （可选）加载多个模型时，在“tf_serving_test”文件夹中创建模型导入配置文件models.config并添加如下内容。

此处以inception_v3_flowers、inception_v4和inception_v4_imagenet三个模型为例，请根据实际情况自行替换。

```
model_config_list:{
    config:{
      name:"inception_v3_flowers",      # 模型名称
```



```
        base_path:"/home/HwHiAiUser/tf_serving_test/inception_v3_flowers", # 模型所在路径
        model_platform:"tensorflow"
    },
    config:{
        name:"inception_v4",
        base_path:"/home/HwHiAiUser/tf_serving_test/inception_v4",
        model_platform:"tensorflow"
    },
      config:{
        name:"inception_v4_imagenet",
        base_path:"/home/HwHiAiUser/tf_serving_test/inception_v4_imagenet",
        model_platform:"tensorflow"
      }
}
```

步骤3 在“tf_serving_test”路径下放置训练好的**SavedModel模型**，参见如下目录。

```
squeezenext/
└── 1
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

1为版本号，请参见以上目录结构存放。

说明

如需提升TF Serving部署性能，可将SavedModel格式的模型转换为.om格式的模型，详情请参见**6.6.5 SavedModel模型转换.om模型**。

步骤4 配置环境变量

1. 将“npu_bridge”路径添加至“LD_LIBRARY_PATH”环境变量。
   ```
   export LD_LIBRARY_PATH=/home/HwHiAiUser/Ascend/tfplugin/latest/python/site-packages/npu_bridge:$LD_LIBRARY_PATH
   ```
2. 将“tf_adapter”路径添加至“LD_LIBRARY_PATH”环境变量。
   ```
   export LD_LIBRARY_PATH=/home/HwHiAiUser/xxx/serving-1.15.0/third_party/tf_adapter:$LD_LIBRARY_PATH
   ```

   说明

   “xxx”为TF Serving的安装路径。
3. 请结合选用的CANN软件包，设置环境变量。

- 请依据实际在下列场景中选择一个进行在线推理依赖包安装路径的环境变量设置。
  - 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推理任务）。
    ```
    . /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
    ```
- 设置tfplugin插件包的环境变量。
  ```
  . /home/HwHiAiUser/Ascend/tfplugin/set_env.sh
  ```
- 若运行环境中存在多个Python版本时，需要指定Python3.7.5的安装路径。
  ```
  export PATH=/usr/local/python3.7.5/bin:$PATH
  export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH
  ```

步骤5 启动tensorflow_model_server，传入**步骤1**和**步骤2**中的配置文件，例如：

单个模型时执行如下命令：

```
tensorflow_model_server --port=8500 --rest_api_port=8501 --model_base_path=/home/HwHiAiUser/tf_serving_test/squeezenext --model_name=squeezenext --platform_config_file=/home/HwHiAiUser/tf_serving_test/config.cfg
```

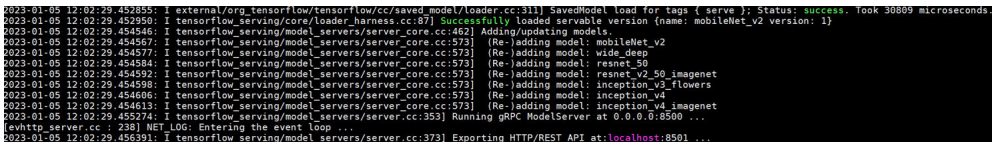

多个模型时执行如下命令：

```
tensorflow_model_server --port=8500 --rest_api_port=8501 --model_config_file=/home/HwHiAiUser/tf_serving_test/models.config --allow_version_labels_for_unavailable_models=true --model_config_file_poll_wait_seconds=60 --platform_config_file=/home/HwHiAiUser/tf_serving_test/config.cfg
```

说明

如果重新安装其他版本CANN软件后，直接启动tensorflow_model_server服务失败，可参考**6.6.6 重新编译TF Serving**解决。

此处需要使用绝对路径。启动成功如下图所示：



通过**tensorflow_model_server --help**命令可查看启动方式及参数，参数解释如下表所示。

表 6-2 参数解释

参数	说明	示例
--port	使用GPRC方式进行通信。	8500
--rest_api_port	使用HTTP/REST API方式进行通信，如果设置为0则不生效，且指定的端口号必须与GPRC不同。	8501
--model_config_file	加载多个模型时，则需要配置此参数文件以导入多个模型，且与模型和--platform_config_file参数配置文件在同一目录下。	/home/HwHiAiUser/tf_serving_test/models.config
--model_config_file_poll_wait_seconds	此参数设置对--model_config_file配置文件刷新时间间隔。当服务开启时，将实时刷新写入--model_config_file配置文件的模型并加载到服务端中。 单位“s”。	60
--model_name	加载一个模型时使用该参数，其值为模型所在版本目录的父目录名。	squeezenext
--model_base_path	加载的模型所在路径，若已配置--model_config_file则可忽略。	/home/HwHiAiUser/tf_serving_test/squeezenext
--platform_config_file	特性配置文件。	/home/HwHiAiUser/tf_serving_test/config.cfg

----结束

6.5 样例参考

样例代码

在安装用户HwHiAiUser目录下，创建Client文件夹，用于存放与服务端通信的脚本deploy.py和推理脚本tf_serving_infer.py，目录如下所示：

```
Client/
  ├─── deploy.py
  └─── tf_serving_infer.py
```

样例代码deploy.py：

```
import tensorflow as tf
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc
import grpc
import numpy as np
import os
import time

class PredictModelGrpc(object):
    def __init__(self, model_name, input_name, output_name, socket='xxx.xxx.xxx.xxx:8500'):# xxx.xxx.xxx.xxx为服务端IP地址
        self.socket = socket
        self.model_name = model_name
        self.input_name = input_name
        self.output_name = output_name
        self.request, self.stub = self.__get_request()
    def __get_request(self):
        channel = grpc.insecure_channel(self.socket, options=[('grpc.max_send_message_length', 1024 * 1024 * 1024),
                                                               ('grpc.max_receive_message_length',
                                                                1024 * 1024 * 1024)])  # 可设置大小
        stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
        request = predict_pb2.PredictRequest()

        request.model_spec.name = self.model_name
        request.model_spec.signature_name = "serving_default"

        return request, stub
    def inference(self, frames):

        t0 = time.time()
        self.request.inputs[self.input_name].CopyFrom(tf.make_tensor_proto(frames, dtype=tf.float32))# 发送请求
        t1 = time.time()
        result = self.stub.Predict.future(self.request, 1000.0) # 执行推理，请求等待时间建议设置为1000.0。
        t2 = time.time()

        res = []
        res.append(tf.make_ndarray(result.result().outputs[self.output_name])[0]) # 获取结果

        t3 = time.time()
        print("Time cost: request.inputs={:.3f} ms, Predict.future={:.3f} ms, get output={:.3f} ms".format((t1 - t0) * 1000, (t2 - t1) * 1000, (t3 - t2) * 1000))# 耗时打印
        return res
```

样例代码tf_serving_infer.py：

```
import numpy as np
from PIL import Image
from scipy import misc
import numpy as np
import scipy
```



```
import imageio
from deploy import PredictModelGrpc
import time
import sys

data_process_time = []
image_path = sys.argv[1]
# 数据预处理
d0 = time.time()
image = misc.imread(image_path)
resized = scipy.misc.imresize(image, (304, 304, 3)) # 原始.pb模型输入节点的Type值，请根据实际值自行修改。
crop_min = abs(304 / 2 - (304 / 2))
crop_max = crop_min + 304
crop_min = int(crop_min)
crop_max = int(crop_max)
image = resized[crop_min:crop_max, crop_min:crop_max, :]

mean_sub = image.astype(np.float32) - np.array([123, 117, 104]).astype(np.float32) # 数据类型从原始.pb模型输入节点的Type中获取，请根据实际的值自行修改。
image = np.expand_dims(np.array(mean_sub), 0)

d1 = time.time()
data_process_time.append(d1 - d0)
model = PredictModelGrpc(model_name='mobileNetv2', input_name='input:0', output_name='MobilenetV2/Logits/output:0') # 根据实际的模型名、模型输入节点名和模型输出节点名自行修改，如果是多输入输出节点时，节点名之间使用;隔开。

# 执行推理
infer_cost_time = []
for i in range(1000):
    t0 = time.time()
    res = model.inference(image)
    t1 = time.time()
    infer_cost_time.append(t1 - t0)
    print("Index= {}, Inference time cost={:.3f} ms".format(i,(t1-t0)*1000))

# 推理耗时打印
print("Batchsize={}, Average inference time cost: {:.3f} ms, Average data process time cost:{:.3f} ms".format(1, (sum(infer_cost_time) - infer_cost_time[0]) / (len(infer_cost_time)) * 1000, (sum(data_process_time) - data_process_time[0]) / (len(data_process_time)) * 1000))
```

说明

用户在不确定原始.pb模型输入输出节点名以及Type值时，可参考**7.6 读取pb模型文件的节点名称**获取。

样例执行

以MobileNetV2模型为例，执行在线推理样例。

步骤1 将.pb模型转换为**SavedModel模型**。

1. 在**gitee代码仓**页面下单击“克隆/下载 > 复制”，复制代码包下载链接。
2. 在开发环境执行命令：git clone URL（其中URL为复制的代码包下载链接），直接将代码包克隆到开发环境。
   ```
   git clone https://gitee.com/ascend/ModelZoo-TensorFlow.git
   ```
3. 将解压后的.pb格式模型（mobileNetv2.pb）移至服务器“/home/HwHiAiUser/tf_serving_test”路径。
4. 在tf_serving_test路径下创建转换脚本pb_to_savedmodel.py，样例代码如下所示：
   ```
   import tensorflow as tf
   from tensorflow.python.saved_model import signature_constants
   from tensorflow.python.saved_model import tag_constants
   from tensorflow.python.framework import convert_to_constants
   from tensorflow.python.framework import tensor_shape
   ```



```
from tensorflow.python.saved_model import save
import sys

def read_pb_model(pb_model_path):
    with tf.gfile.GFile(pb_model_path, "rb") as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        return graph_def
def covert_pb_saved_model(graph_def, export_dir, input_name, output_name):
    builder = tf.saved_model.builder.SavedModelBuilder(export_dir)

    sigs = {}
    with tf.Session(graph=tf.Graph()) as sess:
        tf.import_graph_def(graph_def, name="")
        g = tf.get_default_graph()
        input_name_list = input_name.strip().split(";")
        output_name_list = output_name.strip().split(';')
        input_dict = {}
        output_dict = {}
        for i, s in enumerate(input_name_list):
            ss = s.split(':')[0] + ' : ' + s.split(':')[0] + ','
            print(ss)
            d = g.get_tensor_by_name(s)
            input_dict.update({s:d})
        for i, s in enumerate(output_name_list):
            d = g.get_tensor_by_name(s)
            output_dict.update({s:d})
        out = g.get_tensor_by_name(output_name)
        sigs[signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY] = \
            tf.saved_model.signature_def_utils.predict_signature_def(
                input_dict, output_dict)
        builder.add_meta_graph_and_variables(sess,
                                             [tag_constants.SERVING],
                                             signature_def_map=sigs)
        builder.save()
def covert_pb_to_server_model(pb_model_path, export_dir, input_name='input',
output_name='output'):
    graph_def = read_pb_model(pb_model_path)
    covert_pb_saved_model(graph_def, export_dir, input_name, output_name)


# 转换pb文件为savedmodel
if __name__=="__main__":
    pb_model_path = sys.argv[1]
    export_dir = sys.argv[2]
    input_name = "input:0" # 原始.pb模型输入节点名，请根据实际名称自行修改。如果是多输入节点时，
节点名之间使用;隔开。
    output_name = "MobilenetV2/Logits/output:0" # 原始.pb模型输出节点名，请根据实际名称自行修
改。如果是多输出节点时，节点名之间使用;隔开。
    covert_pb_to_server_model(pb_model_path, export_dir, input_name, output_name)
```

说明

用户在不确定原始.pb模型输入输出节点名时，可参考**7.6 读取pb模型文件的节点名称**。

5. 执行以下命令进行转换。

```
python3 pb_to_savedmodel.py mobileNetv2.pb ./mobileNetv2
```

参数解释：

pb_to_savedmodel.py：转换脚本名称。

mobileNetv2.pb：待转换原始.pb模型名称。

mobileNetv2：saved_model.pb模型文件输出文件路径。

6. 将生成的saved_model.pb模型按照如下目录结构存放。

```
tf_serving_test/
└── mobileNetv2
    └── 1
        ├── saved_model.pb
        └── variables
```


7. 准备数据集。

   在安装用户HwHiAiUser目录下，创建data文件夹，用于存放数据集。

   说明

   此样例以一张格式为.jpg图片数据进行推理，如需使用其他数据集，请用户自行准备。

步骤2 在任意目录下执行以下命令启动tensorflow_model_server，启动成功如**图1 启动成功**所示。

```
tensorflow_model_server --port=8500 --rest_api_port=8501 --model_base_path=/home/HwHiAiUser/
tf_serving_test/mobileNetv2 --model_name=mobileNetv2 --platform_config_file=/home/HwHiAiUser/
tf_serving_test/config.cfg
```

图 6-1 启动成功

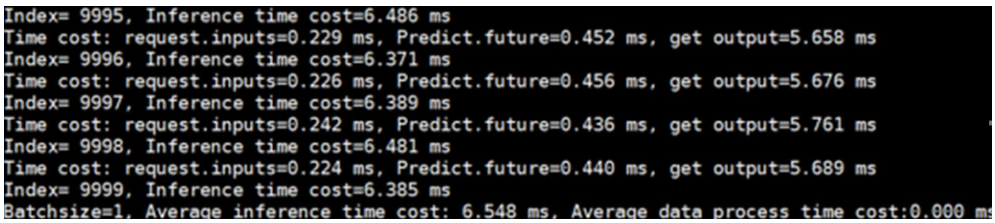
步骤3 使用HwHiAiUser用户重新登录服务器，进入“Client”目录编辑与服务端通信脚本和推理脚本。

创建“deploy.py”通信脚本和“tf_serving_infer.py”推理脚本，并参考**样例代码**写入相关代码。

步骤4 执行如下命令执行推理。

```
python3 tf_serving_infer.py /home/HwHiAiUser/data/cat.jpg
```

步骤5 推理结果如下图所示。



----结束

6.6 常用操作

6.6.1 源码安装 0.24.1 版本 bazel

步骤1 安装系统依赖，此处以Ubuntu与CentOS操作系统为例。

- Ubuntu 18.04 x86_64环境：

  ```
  apt-get install build-essential openjdk-11-jdk python zip unzip
  ```

- Centos 8.3 aarch64环境：

  ```
  yum install java-11-openjdk-devel.aarch64
  yum install java-11-openjdk.aarch64
  yum groupinstall 'Development Tools'
  yum install zip
  ```

若java-11-openjdk安装失败，可进行手动安装，参考**手动安装java-11-openjdk**。

步骤2 配置环境变量。

1. 执行如下命令，打开“.bashrc”文件。


```
vim ~/.bashrc
```

2. 在文件中添加java-11-openjdk的安装路径（以下为示例路径，用户需根据实际路径进行设置）。
```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
```
3. 执行**:wq!**命令保存文件并退出。
4. 执行如下命令使环境变量生效。
```
source ~/.bashrc
```

步骤3 下载**bazel源码压缩包**，将源码包上传至服务器任意路径下。

步骤4 进入源码包所在路径，进行编译安装。

1. 执行如下命令，解压下载的bazel源码压缩包。
```
unzip bazel-0.24.1-dist.zip -d bazel-0.24.1-dist
```
2. 进入解压后的文件夹，执行配置、编译和安装命令。
```
cd bazel-0.24.1-dist/
env EXTRA_BAZEL_ARGS="--host_javabase=@local_jdk//:jdk" ./compile.sh
cp output/bazel /usr/local/bin
```

步骤5 安装验证。

安装完成后重新执行如下命令查看版本号。

```
bazel version
```

----结束

6.6.2 源码安装 3.14.0 版本 CMake

步骤1 下载**CMake源码压缩包**，将源码包上传至服务器任意路径下。

步骤2 进入源码包所在路径，进行编译安装。

1. 执行如下命令，解压下载的CMake源码压缩包。
```
tar -zxvf cmake-3.14.0.tar.gz
```
2. 进入解压后的文件夹，执行配置、编译和安装命令。
```
cd cmake-3.14.0/
./bootstrap --prefix=/usr
make -j4
make install
```

步骤3 安装验证。

安装完成后重新执行如下命令查看版本号。

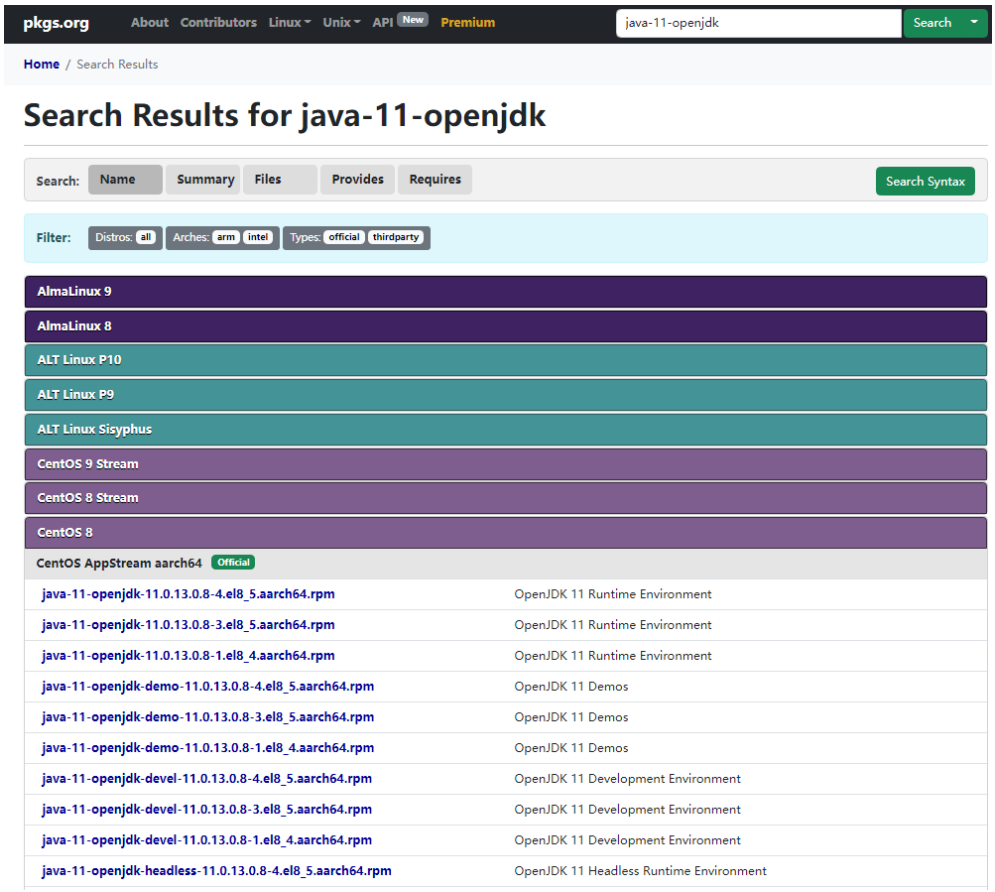
```
cmake --version
```

----结束

6.6.3 手动安装 java-11-openjdk

安装“java-11-openjdk”总共需要安装三个包，用户可通过在**https://centos.pkgs.org**网站右上角搜索“java-11-openjdk”查询以“.rpm”为后缀的软件包的url。网站搜索结果如**图6-2**所示。根据系统版本选择相应软件进行下载，再通过“rpm”命令安装。

图 6-2 网站搜索截图



安装示例

以Centos 8.3 aarch64环境为例，进行“java-11-openjdk”相关软件包安装演示。

表 6-3 软件包下载链接

软件包名	软件包下载路径
java-11-openjdk-headless	链接
java-11-openjdk	链接
java-11-openjdk-devel	链接

步骤1 参考表6-3下载“.rmp”包，将软件包上传至服务器任意路径下。

步骤2 进入软件包所在路径，安装“.rmp”包。

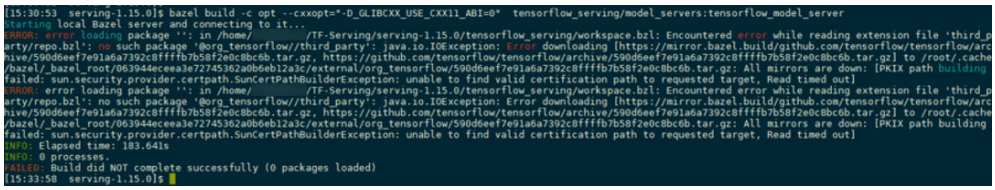
```
rpm -ivh java-11-openjdk-headless-11.0.13.0.8-4.el8_5.aarch64.rpm
rpm -ivh java-11-openjdk-11.0.13.0.8-4.el8_5.aarch64.rpm
rpm -ivh java-11-openjdk-devel-11.0.13.0.8-4.el8_5.aarch64.rpm
```

----结束

6.6.4 手动下载 TF Serving 编译依赖包

TF Serving编译过程中，需要下载依赖包，可能会因为网络问题下载失败，报错如图6-3所示：

图 6-3 TF Serving 编译报错



解决方法如下所示：

步骤1 参考如下链接下载所需依赖包，将依赖包上传至服务器任意路径下（例如：“${HOME}/xxx”）。

tensorflow、**rules_closure**、**bazel-skylib**、**rapidjson**、**abseil-cpp**、**libevent**和**llvm**。

下载的依赖包需要重命名才能使用，如下表所示：

表 6-4 依赖包名称修改

依赖包	修改前	修改后
tensorflow	tensorflow-590d6eef7e91a6a7392c8ffffb7b58f2e0c8bc6b.tar.gz	590d6eef7e91a6a7392c8ffffb7b58f2e0c8bc6b.tar.gz
rules_closure	rules_closure-316e6133888bfc39fb860a4f1a31cfcbae485aef.tar.gz	316e6133888bfc39fb860a4f1a31cfcbae485aef.tar.gz
bazel-skylib	bazel-skylib-0.7.0.tar.gz	0.7.0.tar.gz
rapidjson	rapidjson-1.1.0.zip	v1.1.0.zip
abseil-cpp	abseil-cpp-36d37ab992038f52276ca66b9da80c1cf0f57dc2.tar.gz	36d37ab992038f52276ca66b9da80c1cf0f57dc2.tar.gz
libevent	libevent-release-2.1.8-stable.zip	release-2.1.8-stable.zip
llvm	llvm-7a7e03f906aada0cf4b749b51213fe5784eeff84.tar.gz	7a7e03f906aada0cf4b749b51213fe5784eeff84.tar.gz

步骤2 在编译TF Serving时添加“--distdir”参数，如下所示：

```
bazel --output_user_root=/opt/tf_serving build -c opt --cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0" --distdir=${HOME}/xxx tensorflow_serving/model_servers:tensorflow_model_server
```

----结束

6.6.5 SavedModel 模型转换.om 模型

简介

本章节主要介绍如何使用**saved_model2om.py**工具将训练保存的SavedModel格式的模型重新构图，转换为基于NPU版本用于加载om模型的SavedModel模型，在部署TF Serving时使用转换后的SavedModel模型可以缩短编译时间从而提升TF Serving部署性能。

参数说明

表 6-5 参数说明

参数	参数说明	取值示例
--input_path	• 原始SavedModel模型文件的输入路径。 • 必选。	/home/HwHiAiUser/inputpath/model
--output_path	• 转换成功后生成SavedModel模型文件的输出路径。 • 必选。	/home/HwHiAiUser/outputpath/model
--input_shape	• 输入模型的shape值，格式为"name1:shape;name2:shape;name3:shape"。当设置input_shape时，shape中未明确定义的维度将会被自动设置为1。 • 可选。	input:16,224,224,3
--soc_version	• 输出.om模型的芯片类型。当设置--profiling参数时，无需配置此参数，由当前执行转换的设备决定。 • 必选。	Ascend910
--profiling	• 设置此参数时，则会开启AOE调优。（该参数配置后无需再指定job_type）。 – 取值为1时，启用子图调优； – 取值为2时，启用算子调优。 • 如需进行子图或者算子调优，则该参数必选。	1
--method_name	• 配置TF Serving运行时推理的方法，如果不配置此参数，则会从原始SavedModel模型文件中获取。 • 可选。	/tensorflow/serving/predict
--new_input_nodes	• 重新选择输入节点，格式为：算子:类型:算子名;算子:类型:算子名。 • 可选。	embedding:DT_FLOAT:bert/embedding/word_embeddings:0;add:DT_INT:bert/embedding/add:0

参数	参数说明	取值示例
--new_output_nodes	• 重新选择输出节点，格式为：算子:算子名。 • 可选。	loss:loss/Softmax:0
--output_type	• 指定网络输出数据类型或指定某个输出节点的输出类型，参数的使用方法请参考《AOE工具使用指南》或者《ATC工具使用指南》。 • 可选。	node1:0:FP16
--input_fp16_nodes	• 指定输入数据类型为FP16的输入节点名称，参数的使用方法请参考《AOE工具使用指南》或者《ATC工具使用指南》。 • 可选。	node_name1;node_name2

说明

该工具同时支持ATC和AOE工具中的参数：

- 设置--profiling参数时请参考《AOE工具使用指南》；
- 未设置--profiling参数时请参考《ATC工具使用指南》。

该工具暂不支持ATC和AOE工具中的--out_nodes、--is_input_adjust_hw_layout和--is_output_adjust_hw_layout参数，其中--out_nodes参数可使用表1 参数说明中的--new_output_nodes参数进行代替。

执行转换

步骤1 下载转换工具“saved_model2om.py”至服务器的任一目录，例如上传到*$HOME/tools/*目录下，无需安装。

步骤2 执行以下命令进行转换，具体参数请根据实际修改。

```
python3 saved_model2om.py --input_path "/home/HwHiAiUser/inputpath/model" --output_path "/home/HwHiAiUser/outputpath/model" --input_shape "input:16,224,224,3" --soc_version "Ascend910"
```

如果在转换的过程中需要进行子图或者算子调优，请执行以下命令。

```
python3 saved_model2om.py --input_path "/home/HwHiAiUser/inputpath/model" --output_path "/home/HwHiAiUser/outputpath/model" --input_shape "input:16,224,224,3" --profiling "1"
```

步骤3 转换成功后，会在指定的output_path下生成用于加载om模型的SavedModel模型文件，文件名格式为{om_name}_load_om_saved_model_{timestamp}。

----结束

6.6.6 重新编译 TF Serving

重新安装其他版本CANN软件后，直接启动tensorflow_model_server服务，可能会因为动态链接库链接错误导致服务启动失败，报错如图6-4所示：

图 6-4 动态链接库链接错误

解决方法如下所示：

步骤1 进入“serving-1.15.0/third_party/tf_adapter”目录，执行如下命令。

在“tf_adapter”文件夹下拷贝存放“_tf_adapter.so”文件，并将“_tf_adapter.so”文件名修改为“lib_tf_adapter.so”。

```
cp /home/HwHiAiUser/Ascend/tfplugin/latest/python/site-packages/npu_bridge/_tf_adapter.so .
mv _tf_adapter.so lib_tf_adapter.so
```

步骤2 执行如下命令清理上次编译的缓存，防止增量编译。

```
rm -rvf /opt/tf_serving
bazel clean
```

步骤3 编译TF Serving。

在TF Serving安装目录“serving-1.15.0”下执行如下命令，编译TF Serving。

```
bazel --output_user_root=/opt/tf_serving build -c opt --cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0"
tensorflow_serving/model_servers:tensorflow_model_server
```

其中“–output_user_root”参数指定了TF Serving的安装路径。请根据实际进行指定。

----结束

6.7 FAQ

6.7.1 无法安装 swig 软件包

问题现象

在安装TF Serving编译依赖的软件包时可能会出现无法安装swig软件包问题，并显示如下报错信息。

```
E :The package ascend-cann-toolkit needs to be reinstalled, but I cant`t  find an archive for it.
```

解决方案

可根据以下操作步骤重新安装swig依赖包。

步骤1 备份“/var/lib/dpkg/status”。

```
cp /var/lib/dpkg/status /{newfilepath}/status
```

{newfilepath}表示需要备份的路径，由用户根据实际指定。

步骤2 使用如下命令打开“/var/lib/dpkg/status”文件，定位到出错的软件包记录，最后将该软件包记录删除。如图6-5中标注位置。

```
vim /var/lib/dpkg/status
```


图 6-5 出错的软件包记录

```
 This package contains development headers and static libraries.
Original-Maintainer: Anders Kaseorg <andersk@mit.edu>
Homepage: http://pyyaml.org/wiki/LibYAML

Package: ascend-cann-toolkit
Status: deinstall reinstreq half-configured
Priority: extra
Section: net
Installed-Size: 1113915
Maintainer: Huawei Technologies Co. Ltd <ascend@huawei.com>
Architecture: amd64
Version:
Config-Version:
Description: ascend-cann-toolkit

Package: libpipeline1
Status: install ok installed
Priority: important
Section: libs
Installed-Size: 73
Maintainer: Colin Watson <cjwatson@debian.org>
Architecture: amd64
Multi-Arch: same
```

步骤3 重新安装swig。

```
apt install swig
```

----结束

6.7.2 TF Serving 编译时提示缺少 builtins

问题现象

在TF Serving编译过程中可能会出现builtins依赖模块查询失败问题，报错如图6-6所示。

图 6-6 缺少 builtins 导致的报错信息

解决方案

原因是缺失“future”依赖包，解决方案是安装“future”依赖包，并检查Python指向。

步骤1 执行如下命令，安装future依赖包。

```
pip3.7 install future
```

步骤2 检查“Python”软连接是否指向Python3.7.5。

由于TF Serving编译时需要使用3.7.5版本的Python，TF Serving的相关脚本默认使用的Python解释器关键字“Python”，与系统“Python”软连接默认指向的2.7版本Python不匹配。因此需要将当前“Python”软连接指向“Python3.7.5”。

1. 执行如下命令，检查Python软链接是否指向了Python3.7.5。
   ```
   python --version
   ```
   若Python为3.7.5版本，可直接**步骤3**，反之继续执行之后流程。
2. 执行如下命令创建软连接，将“Python”指向“Python3.7.5”。
   ```
   ln -sf /usr/local/python3.7.5/bin/python3.7  /usr/bin/python
   ```

步骤3 重新编译TF serving。

步骤4 (可选)执行以下命令，复原“Python”软连接指向。

```
ln -sf /usr/bin/python2 /usr/bin/python
```

----结束

6.7.3 TF Adapter 编译结束未生成 tfadapter.tar

问题现象

提示编译成功，但output目录下未生成tfadapter.tar。

解决方案

解决方案为修改“tf_adapter_2.x/CI_Build”文件，并重新编译。

步骤1 用户根据实际在相应路径下执行如下命令，打开需要修改的配置文件。

```
vim tf_adapter_2.x/CI_Build
```

修改内容如下：

```
CONFIGURE_DIR=$(dirname "$0")
cd "${CONFIGURE_DIR}"

############## npu modify begin #############
if [ "$(arch)" != "xxx" ];then
  mkdir -p build/dist/python/dist/
  touch build/dist/python/dist/npu_device-0.1-py3-none-any.whl
  exit 0
fi
############## npu modify end ###############

# 原代码如下：
if [ "$(arch)" != "x86_64" ];then
  mkdir -p build/dist/python/dist/
  touch build/dist/python/dist/npu_device-0.1-py3-none-any.whl
  exit 0
fi
```

步骤2 重新编译TF Adapter。

----结束

7 参考

7.1 session 配置

7.1.1 简介

功能说明

TF Adapter提供了系列session配置用于进行功能调试、性能提升、精度提升等，开发者在昇腾AI处理器上进行模型训练或在线推理时，可以使用这些session配置。

您可以在TensorFlow Adapter软件安装路径下的：python/site-packages/npu_bridge/estimator/npu/npu_estimator.py文件中查看相关配置定义，如果相关参数本章节未列出，表示该参数预留或适用于其他昇腾AI处理器版本，用户无需关注。

调用示例

session配置的通用使用方式如下所示：

```
import tensorflow as tf
from npu_bridge.npu_init import *

config = tf.ConfigProto()
custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF
```



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

7.1.2 session 配置参数说明

基础功能

配置项	说明	使用场景
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： `custom_op.parameter_map["graph_run_mode"].i = 1`	训练/在线推理

session_device_id	当用户需要将不同的模型通过同一个脚本在不同的Device上执行，可以通过该参数指定Device的逻辑ID。
	通常可以为不同的图创建不同的Session，并且传入不同的session_device_id。
	配置示例：
	`config_0 = tf.ConfigProto()`
	`custom_op = config_0.graph_options.rewrite_options.custom_optimizers.add()`
	`custom_op.name = "NpuOptimizer"`
	`custom_op.parameter_map["session_device_id"].i = 0`
	`config_0.graph_options.rewrite_options.remapping = RewriterConfig.OFF`
	`config_0.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF`
	`with tf.Session(config=config_0) as sess_0:`
	`  sess_0.run(...)`
	
	`config_1 = tf.ConfigProto()`
	`custom_op = config_1.graph_options.rewrite_options.custom_optimizers.add()`
	`custom_op.name = "NpuOptimizer"`
	`custom_op.parameter_map["session_device_id"].i = 1`
	`config_1.graph_options.rewrite_options.remapping = RewriterConfig.OFF`
	`config_1.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF`
	`with tf.Session(config=config_1) as sess_1:`
	`  sess_1.run(...)`
	
	`config_7 = tf.ConfigProto()`
	`custom_op = config_7.graph_options.rewrite_options.custom_optimizers.add()`
	`custom_op.name = "NpuOptimizer"`
	`custom_op.parameter_map["session_device_id"].i = 7`
	`config_7.graph_options.rewrite_options.remapping = RewriterConfig.OFF`
	`config_7.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF`
	`with tf.Session(config=config_7) as sess_7:`
	`  sess_7.run(...)`
		训练/在线推理

配置项	说明	使用场景
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： `custom_op.parameter_map["deterministic"].i = 1`	训练/在线推理

内存管理

配置项	说明	使用场景
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： `custom_op.parameter_map["atomic_clean_policy"].i = 1`	训练/在线推理

配置项	说明	使用场景
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： `custom_op.parameter_map["static_memory_policy"].i = 0` **说明** ● 多张图并发执行时，不支持配置为“2”。 ● 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。	训练/在线推理
external_weight	若网络中的weight占用内存较大，且模型加载环境内存受限时，建议通过此配置项将网络中Const/Constant节点的权重外置，防止由于内存不足导致模型编译加载出错。 ● 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： `custom_op.parameter_map["external_weight"].b = True`	训练/在线推理

动态 shape

配置项	说明	使用场景
input_shape	输入的shape信息。 配置示例： `custom_op.parameter_map["input_shape"].s = tf.compat.as_bytes("data:1,1,40,-1;label:1,-1;mask:-1,-1")` 上面示例中表示网络中有三个输入，输入的name分别为data，label，mask，各输入的shape分别为（1,1,40,-1），（1,-1），（-1,-1），name和shape之间以英文冒号分隔。其中-1表示该维度上为动态档位，需要通过dynamic_dims设置动态档位参数。 配置注意事项： ● input_shape中输入的name需要与实际data节点的name的字母顺序保持一致，比如有三个输入：label、data、mask，则input_shape输入顺序应该为data、label、mask ● 如果网络即有dataset输入也有placeholder输入，由于当前仅支持一种输入为动态的场景（例如dataset输入为动态），此时仅需填写dataset所有输入的shape信息。 ● 如果输入中包含标量，则需要填写为0。	在线推理
dynamic_dims	输入的对应维度的档位信息。档位中间使用英文分号分隔，每档中的dim值与input_shape参数中的-1标识的参数依次对应，input_shape参数中有几个-1，则每档必须设置几个维度。并且要求档位信息必须大于1组。 input_shape和dynamic_dims这两个参数的分档信息能够匹配，否则报错退出。 配置示例： `custom_op.parameter_map["dynamic_dims"].s = tf.compat.as_bytes("20,20,1,1;40,40,2,2;80,60,4,4")` 结合上面举例的input_shape信息，表示支持输入的shape为： ● 第0档：data(1,1,40,20)，label(1,20)，mask(1,1) ● 第1档：data(1,1,40,40)，label(1,40)，mask(2,2) ● 第2档：data(1,1,40,80)，label(1,60)，mask(4,4)	在线推理

配置项	说明	使用场景
dynamic_node_type	指定动态输入的节点类型。 ● 0：dataset输入为动态输入。 ● 1：placeholder输入为动态输入。 当前不支持dataset和placeholder输入同时为动态输入。 配置示例： `custom_op.parameter_map["dynamic_node_type"].i = 0`	在线推理

混合计算

配置项	说明	使用场景
mix_compile_mode	是否开启混合计算模式。 ● True：开启。 ● False：关闭，默认关闭。 计算全下沉模式即所有的计算类算子全部在Device侧执行，混合计算模式作为计算全下沉模式的补充，将部分不可离线编译下沉执行的算子留在前端框架中在线执行，提升昇腾AI处理器支持TensorFlow的适配灵活性。 配置示例： `custom_op.parameter_map["mix_compile_mode"].b = True`	训练/在线推理
in_out_pair_flag	混合计算场景下，指定in_out_pair中的算子是否下沉到昇腾AI处理器，取值： ● True：下沉，默认为True。 ● False：不下沉。 配置示例： `custom_op.parameter_map['in_out_pair_flag'].b = False`	在线推理

配置项	说明	使用场景

in_out_pair	混合计算场景下，配置下沉/不下沉部分的首尾算子名。
	配置示例：
	`# 开启混合计算`
	`custom_op.parameter_map["mix_compile_mode"].b = True`
	
	`# 如下配置，将in_nodes, out_nodes范围内的算子全部下沉到昇腾AI处理器执行`
	`in_nodes.append('import/conv2d_1/convolution')`
	`out_nodes.append('import/conv2d_59/BiasAdd')`
	`out_nodes.append('import/conv2d_67/BiasAdd')`
	`out_nodes.append('import/conv2d_75/BiasAdd')`
	`all_graph_iop.append([in_nodes, out_nodes])`
	`custom_op.parameter_map['in_out_pair'].s = tf.compat.as_bytes(str(all_graph_iop))`
	
	`# 或者通过如下配置，将in_nodes, out_nodes范围内的算子不下沉，全部留在前端框架执行`
	`in_nodes.append('import/conv2d_1/convolution')`
	`out_nodes.append('import/conv2d_59/BiasAdd')`
	`out_nodes.append('import/conv2d_67/BiasAdd')`
	`out_nodes.append('import/conv2d_75/BiasAdd')`
	`all_graph_iop.append([in_nodes, out_nodes])`
	`custom_op.parameter_map['in_out_pair_flag'].b = False`
	`custom_op.parameter_map['in_out_pair'].s = tf.compat.as_bytes(str(all_graph_iop))`
		在线推理

功能调试

配置项	说明	使用场景
enable_exception_dump	是否dump异常算子的输入和输出数据，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： `custom_op.parameter_map["enable_exception_dump"].i = 1`	训练

配置项	说明	使用场景

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：
	`custom_op.parameter_map["op_debug_config"].s = tf.compat.as_bytes("/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。
		训练/在线推理

配置项	说明	使用场景
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： `custom_op.parameter_map["debug_dir"].s = tf.compat.as_bytes("/home/test")`	训练/在线推理

精度调优

配置项	说明	使用场景
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	训练/在线推理

配置项	说明	使用场景
	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，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。 训练场景下，针对昇腾910 AI处理器，默认值为“allow_fp32_to_fp16”。 训练场景下，针对昇腾910B AI处理器，默认值为“must_keep_origin_dtype”。 在线推理场景下，默认值为“force_fp16”。 开启自动混合精度的场景下，可参考**修改混合精度黑白灰名单**修改网络中某算子的精度转换规则。 配置示例： `custom_op.parameter_map["precision_mode"].s = tf.compat.as_bytes("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**7.2 keep_tensors_dtypes**接口设置某些算子保持原图精度。	

配置项	说明	使用场景
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	训练/在线推理

配置项	说明	使用场景
	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。 训练场景下：针对昇腾910 AI处理器，该配置项无默认取值，以“precision_mode”参数的默认值为准，即“allow_fp32_to_fp16”。针对昇腾910B AI处理器，该配置项默认值为“origin”。 在线推理场景下：该配置项默认值为“fp16”。 开启自动混合精度的场景下，可参考**修改混合精度黑白灰名单**修改网络中某算子的精度转换规则。 配置示例： `custom_op.parameter_map["precision_mode_v2"].s = tf.compat.as_bytes("origin")` **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**7.2 keep_tensors_dtypes**接口设置某些算子保持原图精度。	

配置项	说明	使用场景

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，配置示例如下：
	`custom_op.parameter_map["modify_mixlist"].s = tf.compat.as_bytes("/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" ] } }`
	说明：上述配置文件样例中展示的算子仅作为参考，请基于实际硬件环境和具体的算子内置优化策略进行配置。
	混合精度场景下算子的内置优化策略可在“OPP安装目录/opp/built-in/op_impl/ai_core/tbe/config/<soc_version>/aic-
		训练/在线推理

配置项	说明	使用场景

	<soc_version>-ops-info.json”文件中查询，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。	

配置项	说明	使用场景
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： custom_op.parameter_map["customize_dtypes"].s = tf.compat.as_bytes("/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	在线推理/训练

配置项	说明	使用场景
	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

配置项	说明	使用场景
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，dump_path为None时会产生异常。 • False：关闭Data Dump功能。 配置示例： `custom_op.parameter_map["enable_dump"].b = True`	训练/在线推理
dump_mode	Data Dump模式，用于指定dump算子输入还是输出数据。取值如下： • input：仅Dump算子输入数据 • output：仅Dump算子输出数据，默认为output。 • all：Dump算子输入和输出数据 配置示例： `custom_op.parameter_map["dump_mode"].s = tf.compat.as_bytes("all")`	训练/在线推理

配置项	说明	使用场景
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enable_dump和enable_dump_debug参数配置为“True”。 配置示例： `custom_op.parameter_map["enable_dump_debug"].b = True`	训练
dump_debug_mode	溢出检测模式，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： `custom_op.parameter_map["dump_debug_mode"].s = tf.compat.as_bytes("all")`	训练
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： `custom_op.parameter_map["dump_path"].s = tf.compat.as_bytes("/home/HwHiAiUser/output")`	训练/在线推理

配置项	说明	使用场景
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： `custom_op.parameter_map["dump_step"].s = tf.compat.as_bytes("0|5|10")`	训练
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： `custom_op.parameter_map["dump_data"].s = tf.compat.as_bytes("stats")`	训练/在线推理
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： `custom_op.parameter_map["dump_layer"].s = tf.compat.as_bytes("nodename1 nodename2 nodename3")`	训练/在线推理

配置项	说明	使用场景

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。
	配置文件样例*fusion_switch.cfg*如下所示，on表示开启，off表示关闭。
	{
	"Switch":{
	"GraphFusion":{
	"RequantFusionPass":"on",
	"ConvToFullyConnectionFusionPass":"off",
	"SoftmaxFusionPass":"on",
	"NotRequantFusionPass":"on",
	"SplitConvConcatFusionPass":"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. 由于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"
	}
		训练/在线推理

配置项	说明	使用场景
	} } 配置示例： custom_op.parameter_map["fusion_switch_file"].s = tf.compat.as_bytes("/home/test/fusion_switch.cfg")	
buffer_optimize	高级开关，是否开启buffer优化。 ● l2_optimize：表示开启buffer优化，默认为l2_optimize。 ● off_optimize：表示关闭buffer优化。 配置示例： custom_op.parameter_map["buffer_optimize"].s = tf.compat.as_bytes("l2_optimize")	在线推理
use_off_line	是否在昇腾AI处理器执行训练。 ● True：在昇腾AI处理器执行训练，默认为True。 ● False：在Host侧的CPU执行训练。 配置示例： custom_op.parameter_map["use_off_line"].b = True	训练/在线推理

性能调优

- 基础配置

配置项	说明	使用场景
iterations_per_loop	sess.run模式下通过set_iteration_per_loop配置小循环次数，即每次sess.run()，在Device侧执行训练迭代的次数。 此处的配置参数和set_iteration_per_loop设置的iterations_per_loop值保持一致，用于功能校验。 配置示例： custom_op.parameter_map["iterations_per_loop"].i = 10	训练

- 高级配置

配置项	说明	使用场景
hcom_parallel	分布式训练场景下，可通过此开关控制是否启用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”，针对小网络（例如：Resnet18），可配置为False。 配置示例： `custom_op.parameter_map["hcom_parallel"].b = True`	训练
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： `custom_op.parameter_map["enable_small_channel"].i = 1`	在线推理/训练

配置项	说明	使用场景
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： custom_op.parameter_map["op_precision_mode"].s = tf.compat.as_bytes("/home/test/op_precision.ini")	训练/在线推理
enable_scope_fusion_passes	指定编译时需要生效的Scope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： custom_op.parameter_map["enable_scope_fusion_passes"].s = tf.compat.as_bytes("ScopeLayerNormPass,ScopeClipBoxesPass")	训练/在线推理

配置项	说明	使用场景
stream_max_parallel_num	此配置项仅适用于NMT网络。 用于指定AI CPU/AI Core引擎的并行度，从而实现AI CPU/AI Core算子间的并行执行。 DNN_VM_AICPU为AI CPU引擎名称，本示例指定了AI CPU引擎的并发数为10； AIcoreEngine为AI Core引擎名称，本示例指定了AI Core引擎的并发数为1。 AI CPU/AI Core引擎的并行度默认为1，取值不能超过AI Core的最大核数。 配置示例： `custom_op.parameter_map["stream_max_parallel_num"].s = tf.compat.as_bytes("DNN_VM_AICPU:10,AIcoreEngine:1")`	训练/在线推理
is_tailing_optimization	此配置项仅适用于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：开启通信拖尾优化。 ● False：不开启通信拖尾优化，默认为False。 必须和NPUOptimizer配合使用，且要求和NPUOptimizer中的is_tailing_optimization值保持一致。 配置示例： `custom_op.parameter_map["is_tailing_optimization"].b = True`	训练

配置项	说明	使用场景

variable_placement	若网络的权重较大，Device侧可能存在内存不足导致网络执行失败的场景，此种情况下可通过此配置将variable的部署位置调整到Host，以降低Device的内存占用。
	● Device：Variable部署在Device。
	● Host：Variable部署在Host。
	默认值为：Device
	约束说明：
	1. 如果此配置项取值为“Host”，需要开启混合计算（即mix_compile_mode取值为“True”）。
	2. 若训练脚本中存在类似tf.case/tf.cond/tf.while_loop等TensorFlow V1版本控制流算子对应的API，此种场景下，如果将“variable_placement”配置为“Host”，可能会导致网络运行失败。为避免此问题，需要在训练脚本中添加如下接口，将TensorFlow V1版本的控制流算子转换为V2版本，并启用资源变量。
	`tf.enable_control_flow_v2()`
	`tf.enable_resource_variables()`
	配置示例：
	`custom_op.parameter_map["variable_placement"].s = tf.compat.as_bytes("Device")`
		训练/在线推理

frozen_variable	当开发者需要将权重保存为checkpoint时，可通过此配置将variable转换为const，以减少Host到Device之间的数据拷贝，从而提升推理性能。 ● True：开启variable到const的转换 ● False：不进行variable到const的转换 默认值为：False 配置示例： `custom_op.parameter_map["frozen_variable"].b = True`	在线推理

Profiling

配置项	说明	使用场景
profiling_mode	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：关闭Profiling功能，默认关闭。 配置示例： `custom_op.parameter_map["profiling_mode"].b = True`	训练/在线推理

配置项	说明	使用场景
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	训练/在线推理

配置项	说明	使用场景
	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：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，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 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,0x64,0x10,0x84,0x85*"。 ● Custom字段表示自定义类型，配置为具体的寄存器值。 ● 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。可选参数。 ● 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	

配置项	说明	使用场景
	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，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： – 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。	

配置项	说明	使用场景
	● 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、task_time、runtime_api互斥，无法同时执行。 ● 在线推理支持task_trace和aicpu，不支持training_trace。 **说明** 在线推理支持task_trace和aicpu，不支持training_trace。 配置示例： `custom_op.parameter_map["profiling_options"].s = tf.compat.as_bytes('{"output":"/tmp/profiling","training_trace":"on","fp_point":"","bp_point":""}')`	

AOE

配置项	说明	使用场景
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： `custom_op.parameter_map["aoe_mode"].s = tf.compat.as_bytes("2")`	训练
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： `custom_op.parameter_map["work_path"].s = tf.compat.as_bytes("/home/HwHiAiUser/output")`	训练

配置项	说明	使用场景

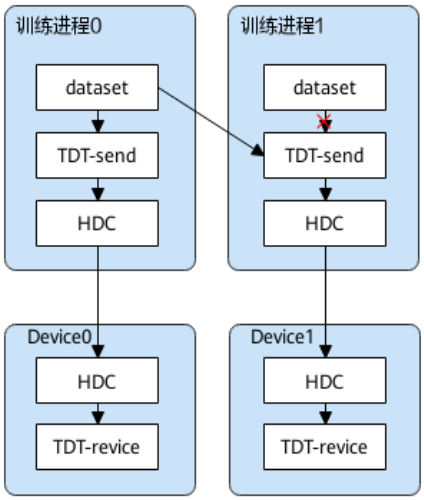
aoe_config_file	通过AOE工具进行调优时，若仅针对网络中某些性能较低的算子进行调优，可通过此参数进行设置。该参数配置为包含算子信息的配置文件路径及文件名，例如：/home/test/cfg/tuning_config.cfg。
	配置示例：
	`custom_op.parameter_map["aoe_config_file"].s=tf.compat.as_bytes("/home/test/cfg/tuning_config.cfg")`
	配置文件中配置的是需要进行调优的算子信息，文件内容格式如下：
	`{ "tune_ops_name":["bert/embeddings/addbert/embeddings/add_1","loss/MatMul"], "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。
		训练

算子编译

配置项	说明	使用场景
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]**	训练/在线推理

配置项	说明	使用场景
	#限制某个芯片下缓存文件夹的磁盘空间的大小，单位为MB **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**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**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： custom_op.parameter_map["op_compiler_cache_mode"].s = tf.compat.as_bytes("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： custom_op.parameter_map["op_compiler_cache_dir"].s = tf.compat.as_bytes("/home/test/kernel_cache")	训练/在线推理

数据增强

配置项	说明	使用场景
local_rank_id	该参数用于推荐网络场景的数据并行场景，在主进程中对于数据进行去重操作，去重之后的数据再分发给其他进程的Device进行前后向计算。  该模式下，一个主机上多Device共用一个进程做数据预处理，但实际还是多进程的场景，在主进程上进行数据预处理，其他进程不在接受本进程上的Dataset，而是接收主进程预处理后的数据。 具体使用方法一般是通过集合通信的get_local_rank_id()接口获取当前进程在其所在Server内的rank编号，用来判断哪个进程是主进程。 配置示例： `custom_op.parameter_map["local_rank_id"].i = 0`	训练/在线推理
local_device_list	该参数配合local_rank_id使用，用来指定主进程给哪些其他进程的Device发送数据。 `custom_op.parameter_map["local_device_list"].s = tf.compat.as_bytes("0,1")`	训练/在线推理

异常补救

配置项	说明	使用场景
hccl_timeout	集合通信超时时间，单位为s，默认值1800s。 当默认时长不满足需求时（例如出现通信失败的错误），可通过此配置项延长超时时间。 ● 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： `custom_op.parameter_map["hccl_timeout"].i = 1800`	训练
op_wait_timeout	算子等待超时时间，单位为s，默认值为120s。 当默认时长不满足需求时，可通过此配置项延长超时时间。 配置示例： `custom_op.parameter_map["op_wait_timeout"].i = 120`	训练
op_execute_timeout	算子执行超时时间，单位为s。 配置示例： `custom_op.parameter_map["op_execute_timeout"].i = 90`	训练
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“hccl_timeout”配置项的值或者环境变量“HCCL_EXEC_TIMEOUT”的值。** 配置示例： `custom_op.parameter_map["stream_sync_timeout"].i = 60000`	训练

配置项	说明	使用场景
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： custom_op.parameter_map["event_sync_timeout"].i = 60000	训练

量化压缩

配置项	说明	使用场景
enable_compress_weight	使能全局weight压缩。 AICore支持Weight压缩功能，通过该参数，可以对Weight进行数据压缩，在进行算子计算时，对Weight进行解压缩，从而达到减少带宽、提高性能的目的。 该参数不能与compress_weight_conf同时使用。 ● True：表示使能。 ● False：表示关闭。默认为False。 配置示例： custom_op.parameter_map["enable_compress_weight"].b = True	在线推理

配置项	说明	使用场景

compress_weight_conf	要压缩的node节点列表配置文件路径以及文件名。node节点主要为conv算子、fc算子。
	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。
	该参数不能与enable_compress_weight参数同时使用。
	weight压缩配置文件中的算子列表由AMCT输出（输出路径为非均匀量化结果路径下记录非均匀量化层名的文件，例如：*module/results/calibration_results/module_nuq_layer_record.txt*），文件内容即为node名称列表，
	配置文件*compress_weight_nodes.cfg*样例如下所示，node名称之间以“;”间隔开。
	`conv1;fc1;conv2_2/x1;fc2;conv5_32/x2;fc6`
	配置示例：
	`custom_op.parameter_map["compress_weight_conf"].s = tf.compat.as_bytes("/home/test/compress_weight_nodes.cfg")`
		在线推理

后续版本废弃配置

以下参数在后续版本将过期，建议开发者不再使用。

配置项	说明	使用场景
op_debug_level	功能调试配置项。 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后，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。 默认值为空，代表不使能此配置。	训练/在线推理

配置项	说明	使用场景
	配置示例： custom_op.parameter_map["op_debug_level"].i = 0	
enable_data_pre_proc	性能调优配置项。 getnext算子是否下沉到昇腾AI处理器侧执行，getnext算子下沉是使能训练迭代循环下沉的必要条件。 ● True：下沉，getnext算子下沉的前提是必须使用TensorFlow Dataset方式读数据。 ● False：不下沉，默认为False。 配置示例： custom_op.parameter_map["enable_data_pre_proc"].b = True	训练
variable_format_optimize	性能调优配置项。 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，例如进行NCHW到NC1HWC0的数据格式转换。但在用户特殊要求场景下，可以选择关闭该功能开关。 默认值为空，代表不使能此配置。 配置示例： custom_op.parameter_map["variable_format_optimize"].b = True	训练
op_select_implmode	性能调优配置项。 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。 默认值为空，代表不使能此配置。 配置示例： custom_op.parameter_map["op_select_implmode"].s = tf.compat.as_bytes("high_precision")	训练/在线推理

配置项	说明	使用场景
optypelist_for_implmode	性能调优配置项。 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，多个算子以英文逗号分隔。 该参数需要与op_select_implmode参数配合使用，例如： op_select_implmode配置为high_precision。 optypelist_for_implmode配置为Pooling。 默认值为空，代表不使能此配置。 配置示例： `custom_op.parameter_map["optypelist_for_implmode"].s = tf.compat.as_bytes("Pooling,SoftmaxV2")`	训练/在线推理
dynamic_input	当前网络的输入是否为动态输入，取值包括： • True：动态输入。 • False：固定输入，默认False。 配置示例： `custom_op.parameter_map["dynamic_input"].b = True` **须知** 当存在不同输入shape的子图时，由于dynamic_inputs_shape_range是针对于单张图的配置属性，因此可能会导致执行异常，建议使用**set_graph_exec_config**以支持动态输入场景。	训练/在线推理
dynamic_graph_execute_mode	对于动态输入场景，需要通过该参数设置执行模式，即dynamic_input为True时该参数生效。取值为： dynamic_execute：动态图编译模式。该模式下获取dynamic_inputs_shape_range中配置的shape范围进行编译。 配置示例： `custom_op.parameter_map["dynamic_graph_execute_mode"].s = tf.compat.as_bytes("dynamic_execute")` **须知** 当存在不同输入shape的子图时，由于dynamic_inputs_shape_range是针对于单张图的配置属性，因此可能会导致执行异常，建议使用**set_graph_exec_config**以支持动态输入场景。	训练/在线推理

配置项	说明	使用场景
dynamic_inputs_shape_range	动态输入的shape范围。例如全图有3个输入，两个为dataset输入，一个为placeholder输入，则配置示例为： custom_op.parameter_map["dynamic_inputs_shape_range"].s = tf.compat.as_bytes("getnext:[128 ,3~5, 2~128, -1],[64 ,3~5, 2~128, -1];data:[128 ,3~5, 2~128, -1]") 使用注意事项： • 使用此参数时，不支持将常量设置为用户输入。 • dataset输入固定标识为“getnext”，placeholder输入固定标识为“data”，不允许用其他表示。 • 动态维度有shape范围的用波浪号“~”表示，固定维度用固定数字表示，无限定范围的用-1表示。 • 对于多输入场景，例如有三个dataset输入时，如果只有第二个第三个输入具有shape范围，第一个输入为固定输入时，仍需要将固定输入shape填入： custom_op.parameter_map["dynamic_inputs_shape_range"].s = tf.compat.as_bytes("getnext:[3,3,4,10],[-1,3,2~1000,-1],[-1,-1,-1,-1]") • 对于标量输入，也需要填入shape范围，表示方法为：[]。 **须知** 当存在不同输入shape的子图时，由于dynamic_inputs_shape_range是针对于单张图的配置属性，因此可能会导致执行异常，建议使用**set_graph_exec_config**以支持动态输入场景。	训练/在线推理
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。	训练/在线推理
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。	训练/在线推理

7.2 keep_tensors_dtypes

函数原型

```
def keep_tensors_dtypes(graph, input_tensors)
```


功能说明

指定哪些算子保持原有精度。

使用约束

该接口仅适用于在线推理场景。

算子精度模式为保持原图精度（即precision_mode指定为must_keep_origin_dtype）时，该接口不生效。

参数说明

参数名	输入/输出	描述
graph	输入	从pb模型导入的图。
input_tensors	输入	需要保持精度的算子名称。

返回值

无。

调用示例

```
from npu_bridge.estimator.npu import util
g=tf.Graph()
util.keep_tensors_dtypes(g,("random_uniform_1/sub:0",))
```

7.3 set_graph_exec_config

函数原型

def set_graph_exec_config(fetch, dynamic_input=False,

dynamic_graph_execute_mode="dynamic_execute",

dynamic_inputs_shape_range=None,

is_train_graph=False,

experimental_config=None)

功能说明

图级别的配置项接口，用于按计算图设置编译和运行选项。通过该接口调用之后，fetch节点会被打上设置的属性。

使用约束

如果同时设置了图级别的参数和session级别的参数，则图级别的参数优先级高。

参数说明

参数名	输入/输出	描述
fetch	输入	图上任意能够执行到的节点，取值包含tensor、operation、list、tuple或者tensor的name。 由于tf.no_op节点会在TensorFlow自身进行图处理时优化掉，因此不能输入该节点
dynamic_input	输入	**说明：该参数后续版本将废弃，建议不要配置此参数。** 当前输入是否为动态输入，取值包括： ● True：动态输入。 ● False：固定输入，默认False。
dynamic_graph_execute_mode	输入	**说明：该参数后续版本将废弃，建议不要配置此参数。** 对于动态输入场景，需要通过该参数设置执行模式，即dynamic_input为True时该参数生效。取值为： dynamic_execute：动态图编译模式。该模式下获取dynamic_inputs_shape_range中配置的shape范围进行编译。

参数名	输入/输出	描述

dynamic_inputs_shape_range	输入	**说明：该参数后续版本将废弃，建议不要配置此参数。**
		动态输入的shape范围。例如全图有3个输入，两个为dataset输入，一个为placeholder输入，则配置示例为：
		`dynamic_inputs_shape_range="getnext:[128 , 3~5, 2~128, -1],[64 ,3~5, 2~128, -1];data:[128 , 3~5, 2~128, -1]"`
		使用注意事项：
		● 使用此参数时，不支持将常量设置为用户输入。
		● dataset输入固定标识为“getnext”，placeholder输入固定标识为“data”，不允许用其他表示。
		● 动态维度有shape范围的用波浪号“~”表示，固定维度用固定数字表示，无限定范围的用-1表示。
		● 对于多输入场景，例如有三个dataset输入时，如果只有第二个第三个输入具有shape范围，第一个输入为固定输入时，仍需要将固定输入shape填入：
		`dynamic_inputs_shape_range="getnext:[3,3,4,10],[-1,3,2~1000,-1],[-1,-1,-1,-1]"`
		● 对于标量输入，也需要填入shape范围，表示方法为：[]。
		● 若网络中有多个getnext输入，或者多个data输入，需要分别保持顺序关系，例如：
		– 若网络中有多个dataset输入：
		`def func(x):`
		`  x = x + 1`
		`  y = x + 2`
		`  return x,y`
		`dataset = tf.data.Dataset.range(min_size, max_size)`
		`dataset = dataset.map(func)`
		网络的第一个输入是x（假设shape range为：[3~5]），第二个输入是y（假设shape range为：[3~6]），配置到dynamic_inputs_shape_range中时，需要保持顺序关系，即
		`dynamic_inputs_shape_range ="getnext:[3~5],[3~6]"`
		– 若网络中有多个placeholder输入：

参数名	输入/输出	描述
		如果不指定placeholder的name，例： x = tf.placeholder(tf.int32) y = tf.placeholder(tf.int32) placeholder的顺序和脚本中定义的位置一致，即网络的第一个输入是x（假设shape range为：[3~5]），第二个输入是y（shape range为：[3~6]），配置到dynamic_inputs_shape_range中时，需要保持顺序关系，即 dynamic_inputs_shape_range= "data:[3~5],[3~6]" 如果指定了placeholder的name，例： x = tf.placeholder(tf.int32, name='b') y = tf.placeholder(tf.int32, name='a') 则网络输入的顺序按name的字母序排序，即 即网络的第一个输入是y（假设shape range为：[3~6]），第二个输入是x（shape range为：[3~5]），配置到dynamic_inputs_shape_range中时，需要保持顺序关系，即 dynamic_inputs_shape_range = "data:[3~6],[3~5]"
is_train_graph	输入	标记该图是否为计算图。 ● True：是计算图 ● False：不是计算图，默认False。
experimental_config	输入	当前版本暂不推荐使用。 。

返回值

fetch

调用示例

一般训练网络中都会执行梯度更新操作，可以将梯度更新操作的返回值作为set_graph_exec_config的fetch入参：

```
from npu_bridge.estimator.npu import util
train_op = util.set_graph_exec_config(train_op,
                                      dynamic_input=True,
                                      dynamic_inputs_shape_range="data:[1~2];getnext:[1~50,1~50],[1~50,1~50]")
```


7.4 环境变量参考

7.4.1 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
```

7.4.2 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
```

7.4.3 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"}'
```

7.4.4 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
```

7.4.5 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
  ```

7.4.6 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

7.4.7 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
```

7.4.8 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
```

7.4.9 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
```

7.4.10 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
```

7.4.11 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
```

7.4.12 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
```


7.4.13 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
```

7.4.14 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
```


7.4.15 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
```

7.4.16 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
```


7.4.17 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
```

7.4.18 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
```

7.5 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

7.6 读取 pb 模型文件的节点名称

当用户无法获取.pb模型文件的节点名称时，可创建readNodeName_form_pb.py脚本并写入如下代码并执行。

```
import tensorflow as tf
from tensorflow.python.platform import gfile

GRAPH_PB_PATH = 'resnet50_tensorflow_1.5.pb'        # .pb模型文件路径，用户可根据实际修改
with tf.Session() as sess:
```



```
print("load graph")
with gfile.FastGFile(GRAPH_PB_PATH, 'rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
    tf.import_graph_def(graph_def, name='')
    for i, node in enumerate(graph_def.node):
        print("Name of the node : %s" % node.name)
```

执行如下命令：

python3.7 readNodeName_form_pb.py

则可以输出模型文件中的节点名称。

此脚本仅支持查看模型中的节点名称，您可以通过开源的模型可视化工具（例如：**Link**）查看网络的拓扑结构。

