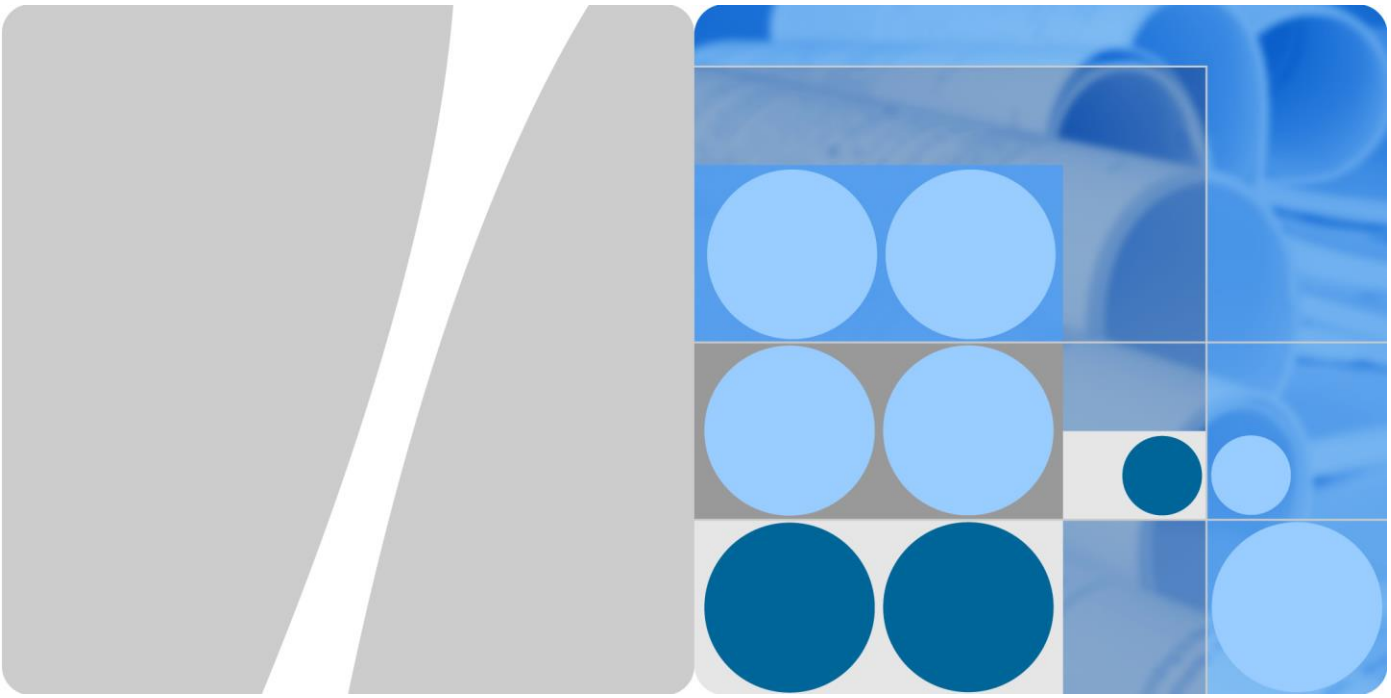


HiAI DDK V320

轻量化工具使用说明书

文档版本 04

发布日期 2020-06-17

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默示的声明或保证。

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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

前言

概述

本文提供对轻量化工具的使用说明。

修改记录

日期	修订版本	修改描述
2020-06-17	04	修改说明书 demo 介绍，添加 QA 章节
2020-03-16	03	新增网络结构搜索内容
2019-12-31	02	新增 V320 版本内容
2019-09-04	01	新增 V310 版本内容

目 录

插图目录

表格目录

1 介 绍

1.1 概述

轻量化工具是一款集多种模型压缩算法和网络结构搜索算法于一体的自动模型轻量化工具，针对 NPU 架构对深度神经网络模型进行深度的模型优化，可以帮助用户自动地完成模型轻量化以及网络结构的生成任务。目前支持无训练模式、重训练模式和网络结构搜索。

- 无训练模式：用户可以直接输入模型，无需重训练，快速的完成模型轻量化。无训练场景下支持 Quant_INT8-8、Quant_Weight_INT8 轻量化算法。适合倾向于快捷方便量化的用户使用。
- 重训练模式：用户能够基于预训练好的全精度基线模型进行重训练，在可接受的精度损失范围内，使模型小型化。重训练场景下支持 Quant_INT8-8 和 Quant_INT8-2 轻量化算法。适合对精度要求较高的用户使用。
- 网络结构搜索：支持分类网络、检测网络、分割网络三种业务类型。用户配置相应的搜索参数和接口函数后，使用网络结构搜索工具进行搜索，在设定的算力约束下，得到优秀的网络模型。适合需要自动生成网络结构的用户。

	支持的框架	支持的策略	支持的设备
无训练模式	Caffe、TensorFlow	Quant_INT8-8、Quant_Weight_INT8	同时支持 CPU 和 GPU 模式，GPU 支持单机单卡
重训练模式	Caffe、TensorFlow	Quant_INT8-8、Quant_INT8-2	支持 GPU，支持单机单卡和单机多卡
网络结构搜索	TensorFlow	HiAIMLEA	支持 GPU，支持单机单卡和单机多卡

通过模型轻量化以及网络结构搜索获得模型在 NPU 上的收益可以参考 6.2 模型收益。

说明

- Quant_INT8-8：权重 8bit 量化，数据 8bit 量化
- Quant_Weight_INT8：权重 8bit 量化，数据不量化
- Quant_INT8-2：权重 2bit 量化，数据 8bit 量化

- HiAIMLEA：基于遗传算法进行网络结构搜索

经过轻量化工具小型化后的模型，可以使用 OMG 转换工具转为 DaVinci 模型，使用方式可参考《华为 HiAI_DDK_V320_OMG 工具使用说明》中的量化章节。

该轻量化工具位于 DDK 包中的 tools/tools_dopt 目录下。

表1-1 tools_dopt 目录

目录文件	描述说明
tools\tools_dopt\caffe	对应 Caffe 框架重训练使用的 so 以及源码
tools\tools_dopt\tensorflow	对应 TensorFlow 框架重训练使用的 so
tools\tools_dopt\dopt_trans_tools	重训练后转换模型的工具
tools\tools_dopt\demo	提供 Caffe 和 TensorFlow 的样例模型
tools\tools_dopt\config	为用户使用的框架信息的配置脚本，例如 Caffe 的路径
tools\tools_dopt\env	轻量化工具 Docker 环境配置文件

1.2 支持范围

- 重训练量化支持 GPU 训练，包括单机单卡和单机多卡训练
- 无训练量化支持 CPU 和 GPU 量化
- 网络结构搜索工具支持单机单卡和单机多卡，进行分类、检测、分割场景的骨架搜索，依赖运行环境参考 4.1 环境准备
- 用户可使用的优化策略在不同的 HiAI DDK 版本下有所不同，优化策略与 HiAI 版本对照表如下：

表1-2 HiAI DDK 版本与优化策略对照表

hiai_version	strategy
HIAI_DDK_V310	Quant_INT8-2
HIAI_DDK_V320	Quant_INT8-2，Quant_INT8-8，Quant_Weight_INT8，HiAIMLEA

1.3 系统要求

使用本工具需同时满足下列环境要求：

- Python 3.6
- Ubuntu 16.04
- GPU 需要使用支持 CUDA®的显卡
- 安装 cuda 9、cudnn 7 (7.6.2) 环境

1.4 用户指引

1.4.1 Caffe 用户和 1.4.2 TensorFlow 用户列举了不同用户场景对应的步骤和工具链功能。用户可以根据下表中列举的不同场景选择适宜的优化方式。

1.4.1 Caffe 用户

表1-3 Caffe 用户指引

用户场景	用户需要准备	工具链提供功能	无法支持的情况	建议使用工具
缺少足量数据集和训练资源 **追求高易用性**	可运行 Caffe 的环境 deploy.prototxt Caffemodel 校准集	INT8-8 量化 权重 INT8 量化 无需重新编译 Caffe，对新手用户友好	无	无训练量化 -> Caffe 模型无训练量化
对精度要求较高 **有足够的数据集和训练资源** **有一定的编程能力**	可运行 Caffe 的环境 Test.prototxt Train.prototxt Caffemodel 训练数据集 测试数据集 (需要重新编译 Caffe)	INT8-8 量化 INT8-2 量化 量化模型精度测试 混合精度量化 模型精度可保障	用户使用 pycaffe 进行而非 prototxt 进行模型定义 可能需要修改 Caffe 源代码	重训练量化 -> Caffe 模型优化训练

1.4.2 TensorFlow 用户

表1-4 TensorFlow 用户指引

用户场景	用户需要准备	工具链提供功能	无法支持的情况	建议使用工具
缺少足量数据集和训练资源	可运行 TensorFlow 的环境	INT8-8 量化 权重 INT8	无	无训练量化 -> TensorFlow 模型无训练量化

用户场景	用户需要准备	工具链提供功能	无法支持的情况	建议使用工具
追求高易用性	模型 Pb 文件 (需要用户自行转换) 校准集	量化 无需 TensorFlow 编写代码，新手用户友好		
对精度要求较高 **有足够的数据集和训练资源** **有一定的编程能力**	可运行 TensorFlow 的环境 模型的 ckpt 文件 训练集 测试集 用户需要根据说明书编写模型接口代码	INT8-8 量化 INT8-2 量化 量化模型精度测试 混合精度量化 模型精度可保障	无	重训练量化-> TensorFlow 模型优化训练
需要自动生成符合需求的网络结构	根据 4.1 环境准备准备环境 训练集 测试集 用户需要根据说明书编写模型接口代码	搜索生成网络结构	无	网络结构搜索

2 无训练量化

2.1 输入准备

无训练量化的输入准备化分为如下步骤：

步骤 1 准备基线模型（2.1.1 准备模型）

步骤 2 准备校准集（2.1.2 准备校准集）

步骤 3 填写 config.prototxt 策略配置（2.1.3 填写 config.prototxt 文件）

----结束

2.1.1 准备模型

- Caffe 用户

  用户需提供需要量化的 prototxt 和 caffemodel 模型。
- TensorFlow 用户

  用户需提供需要量化的 pb 模型。

2.1.2 准备校准集

用户需提供 bin 格式或图片格式的校准集。bin 格式的输入数据需按照以下方式存储，如表 2-1。图片格式的数据为存放测试图片的文件夹，图片格式的输入默认以 BGR 的三通道彩图读取，读出的数据格式为 NCHW，其中 N 为提供图片数量。

说明

轻量化工具支持的图片格式包括 ".bmp", ".dib", ".jpeg", ".jpg", ".jpe", ".png", ".webp",".pbm", ".pgm", ".ppm", ".tiff",".tif",".BMP", ".DIB", ".JPEG", ".JPG", ".JPE", ".PNG",".WEBP",".PBM", ".PGM", ".PPM", ".TIFF",".TIF"。

bin 格式的输入数据支持两种定义方式，分别适用于任意维度的输入数据和 4 维的输入数据。

对于任意维度的输入数据，bin 文件需按照以下方式定义，以维度（50，100，300）的 3 维数据为例，读出的数据形状为（50, 100, 300）。

表2-1 任意维度二进制校准集说明

文件头/数据	地址偏移	Type	Value	Description
文件头（共 20 字节）	0000	32bit int	610	Magic number 当 magic number=610，用来校验文件的合法性
	0004	32bit int	3	Input data rank
	0008	32bit int	50	Input dimension 1
	0012	32bit int	100	Input dimension 2
	0016	32bit int	300	Input dimension 3
数据	…	Float32	...	数据数量等于 50*100*300

对于 4 维的输入数据，bin 文件可以按照表 2-1 定义，也可以按照表 2-2， 以维度为（50，3，28，28）的 4 维数据为例，读出的数据形状为（50, 3, 28, 28）

表2-2 4 维二进制校准集说明

文件头/数据	地址偏移	Type	Value	Description
文件头（共 20 字节）	0000	32bit int	510	Magic number 当 magic number=510，用来校验文件的合法性
	0004	32bit int	50	Input num
	0008	32bit int	3	Input channels
	0012	32bit int	28	Input height
	0016	32bit int	28	Input width
数据	…	Float32	...	数据数量等于 50*3*28*28

当用户提供 IMAGE 格式的校准集时，本工具提供 resize、减均值、除方差的数据预处理方式。其中 resize 方式与 cv2.resize 相同。resize 的算法为 INTER_LINEAR。

2.1.3 填写 config.prototxt 文件

config.prototxt 参数说明如表 2-3 所示。其中 BINARY 模式不进行预处理，IMAGE 模式根据用户给出的均值方差进行预处理。

表2-3 config.prototxt 参数说明表

参数名称	参数描述	是否必填
strategy	优化策略，目前支持两种方式： Quant_INT8-8，Quant_Weight_INT8，默认策略为 Quant_INT8-8	否
device	使用 GPU 还是 CPU 进行量化 USE_GPU: GPU 模式 USE_CPU: CPU 模式	是
exclude_op	支持两种方式。 1. 使用一个 exclude_op，exclude_op 包含多个 op_name，用分号隔开； 2. 使用多个 exclude_op，每个 exclude_op 包含一个 op_name 。 两种方式可混合使用，当所给 op_name 不在模型内时，会报错。	否
preprocess_parameter	指定预处理及输入相关参数。 在 preprocess_parameter 内部包含如下参数： input_type image_format input_file_path mean_value standard_deviation	是

preprocess_parameter 包含的子参数说明如表 2-4 所示，对于模型有多输入的情况，每一个输入都需要配置一份 preprocess_parameter。

表2-4 预处理参数说明表

参数名称	参数描述	是否必填
input_type	使用二进制文件输入还是图片格式输入。 BINARY：使用二进制输入 IMAGE：使用图片格式输入	是
image_format	图片格式。 BGR：使用 BGR 图片格式输入 RGB：使用 RGB 图片格式输入 默认采用 BGR，可不填	否
mean_value	图片预处理的均值参数。	否

参数名称	参数描述	是否必填
	类型为 float 的数值，范围为 0.0-255.0 mean_value 个数必须和输入的 C 维度相等。默认是输入 C 维度个 0.0，可不填。仅 IMAGE 模式下生效。	
standard_deviation	图片与预处理使用的标准差。 类型为 float 的数值，需要>=0.0。仅 IMAGE 模式下生效。	否
input_file_path	输入校准集的绝对路径，bin 文件路径或存有图片的文件夹。 例如/path/to/user/data	是

配置文件的配置示例如下所示。

- BINARY 模式输入

```
strategy: "Quant_INT8-8"
device: USE_GPU
preprocess_parameter:
{
    input_type: BINARY
    input_file_path: "path/to/user/bin/caffe_inception_calibrationset.bin"
}
exclude_op:  "conv1"
exclude_op:  "conv2;conv3"
```

- IMAGE 模式输入

```
strategy: "Quant_INT8-8"
device: USE_GPU
preprocess_parameter:
{
    input_type: IMAGE
    image_format: BGR
    mean_value: 104.0
    mean_value: 113.0
    mean_value: 123.0
    standard_deviation: 0.5
    input_file_path: "path/to/user/images/"
}
exclude_op:  "conv1"
exclude_op:  "conv2;conv3"
```


2.2 Caffe 模型无训练量化

使用 Caffe 进行无训练量化分为如下步骤：

步骤 1 准备 Caffe 环境（2.2.1 环境准备）

步骤 2 执行模型量化（2.2.2 模型量化）

----结束

2.2.1 环境准备

如没有 Caffe 及 pycaffe 环境，需要自行安装编译环境。如已有，则不需要配置环境。

安装 Caffe 环境如下所示：

```
sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev
libopencv-dev libhdf5-serial-dev protobuf-compiler
sudo apt-get install --no-install-recommends libboost-all-dev
sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev
```

进入源码所在路径：

```
cp Makefile.config.example Makefile.config
make -j
make pycaffe
```

2.2.2 模型量化

运行 python ${ROOT}/caffe/dopt/py${PY_VER}/dopt_so.py，其中，${ROOT}为发布包所在路径，py${PY_VER}代表用户所用 python 版本。

运行该脚本的参数如下所示。

说明

路径部分：支持大小写字母、数字，下划线；

文件名部分：支持大小写字母、数字，下划线和点(.)。

表2-5 无训练工具命令行参数说明表

参数名称	参数描述	是否必填
-m, --mode	运行模式 0: 无训练模式 1: 重训练模式	是
--framework	深度学习框架类型 0: Caffe 3: TensorFlow	是

参数名称	参数描述	是否必填
--weight	权值文件路径。当原始模型是 Caffe 时需要指定。	是
--model	Caffe prototxt 文件路径。	是
--cal_conf	校准方式量化配置文件路径。当前支持 Convolution、Full Connection、ConvolutionDepthwise 三种算子的量化，包括权重、偏置、数据量化。 量化配置文件说明请参考“2.1.3 填写 config.prototxt 文件”	是
--output	存放量化完成后的模型文件路径，例如”/path/to/out/resnet18.caffemodel	是
--input_format	输入格式数据, NHWC or NCHW，当用户选择以 IMAGE 格式或文件头为 510 的 bin 文件作为输入数据，并选择输入格式数据为 NHWC 时，工具会自动调整通道顺序；当选择文件头为 610 的 bin 文件作为输入数据时不会调整通道顺序。	是
--input_shape	输入数据的 shape。例如：“input_name1: n1, c1，h1, w1; input_name2: n2, c2, h2,w2”。input_name 必须是转换前的网络模型中的节点名称。多输入 input_shape 之间由';'进行分割。input_shape 中指定各维度输入数据值需与网络模型中指定的输入节点所需形状保持一致。例如：假设转换前网络模型指定输入节点为 input_shape_network: none, 224, 224, 3; input_shape 第 2、3、4 维度输入数值必须为 224,224,3，否则尺寸不匹配。假设转换前网络模型指定输入节点为 input_shape_network: 1, 224, 224, 3；则 input_shape 各维度输入数据均不可变。	否
--out_nodes	指定输出节点。例如：“node_name1; node_name1; node_name2”。node_name 必须是模型转换前的网络模型中的节点名称。	否
--compress_conf	模型文件转为二进制格式文件的路径。例如：“param_file”。该文件为轻量化配置，在使用 OMG 离线模型转换时将被作为参数 compress_conf 的输入	是
--caffe_dir	Caffe 源代码的路径。	是
--device_idx	GPU 或 CPU 设备号。	否

运行量化脚本后，会输出用户--output 传入同名的 prototxt 和 caffemodel， 以及与--compress_conf 传入同名的量化配置文件，例如用户--output 输入 quantmodel.caffemodel，--compress_conf 输入 param，最终会输出 quantmodel.prototxt、quantmodel.caffemodel 和 param。

2.3 TensorFlow 模型无训练量化

使用 TensorFlow 进行无训练量化分为如下步骤：

步骤 1 准备 TensorFlow 环境（2.3.1 环境准备）

步骤 2 执行模型量化（2.3.2 模型量化）

----结束

2.3.1 环境准备

该功能支持 TensorFlow 1.12 CPU 或 GPU 版本。如没有该版本环境，需要自行安装；如已有，则不需要配置环境。

TensorFlow 安装方法如下所示。

pip install tensorflow-gpu==1.12 或 pip install tensorflow==1.12

2.3.2 模型量化

运行 python ${ROOT}/tensorflow/dopt/py${PY_VER}/dopt_so.py，其中，${ROOT}为发布包所在路径，py${PY_VER}代表用户所用 python 版本。

运行该脚本对 TensorFlow 模型进行无训练量化的参数如下所示。

说明

路径部分：支持大小写字母、数字，下划线；

文件名部分：支持大小写字母、数字，下划线和点(.)。

表2-6 无训练命令行说明表

参数名称	参数描述	是否必填
-m, --mode	参考表 2-5	是
--framework	参考表 2-5	是
--model	原始模型文件路径，支持 pb 模型。	是
--cal_conf	参考表 2-5	是
--output	存放量化完成后的模型文件绝对路径，例如”/path/to/out/resnet18.pb”	是
--input_format	参考表 2-5	是

参数名称	参数描述	是否必填
--input_shape	参考表 2-5	是
--out_nodes	参考表 2-5	是
--compress_conf	参考表 2-5	是
--device_idx	参考表 2-5	否

运行量化脚本后，会输出用户--output 传入同名的 pb， 以及--compress_conf 传入同名的量化配置文件，例如用户--output 输入 quantmodel.pb，--compress_conf 输入 param，最终会输出 quantmodel.pb 和 param。

3 重训练量化

3.1 依赖环境准备

重训练量化运行环境主要分两种情况（Docker 环境和 Linux 环境），用户可任选其一。

说明

重训练环境配置完成后，可同时支持无训练和重训练量化。

3.1.1 准备 Docker 环境

镜像制作需要依赖的环境如下：

Nvidia-docker 版本：nvidia-docker2

安装方式参考：https://github.com/NVIDIA/nvidia-docker

步骤 1 解压 ddk 包，并进入 tools 文件夹所在的目录：“cd ./$DDK_PATH/”

步骤 2 重训练量化的 docker 镜像生成，参考如下方式构建：

- 如果用户设置代理：

```
docker build -f tools/tools_dopt/env/docker_tf1.12/Dockerfile . -t
hiai_ddk:v320 --build-arg HTTP_PROXY="http://xxxxxx@xxxxxx.com:8080" --
no-cache
```

- 如果用户没有设置代理：

```
docker build -f tools/tools_dopt/env/docker_tf1.12/Dockerfile . -t
hiai_ddk:v320 --build-arg HTTP_PROXY=" " --no-cache
```

其中选项说明如下：

- -f 指定要使用的 Dockerfile 路径
- -t 镜像的名字及标签，通常是 name:tag 或 name 格式，如默认的 hiai_ddk:v320
- --build-arg 设置镜像创建时的变量
- HTTP_PROXY 代理配置项
- --no-cache 创建镜像的过程不使用缓存

步骤 3 重训练量化的 docker 容器生成，参考如下方式启动：

```
nvidia-docker run -d -it --restart=always --net=host --privileged --name
my_docker -v /user_data:/data hiai_ddk:v320
```

其中选项说明如下：

- -d 以 daemon 形式在后台执行
- -i 以交互模式运行容器，通常与 -t 同时使用；
- -t 为容器重新分配一个伪输入终端，通常与 -i 同时使用
- --restart=always 机器重启时，会自动重启容器
- --net=host 使用宿主机网卡，容器以宿主机 IP 和外部进行交互
- --privileged 当运行多卡搜索时，需要较高权限，因此务必添加此选项
- --name 指定 docker 容器名字
- -v 用于映射宿主机目录到容器内
- hiai_ddk:v320 为镜像名

----结束

3.1.2 准备 Linux 环境

使用轻量化工具重训练功能，用户需要准备如下的环境和依赖：

- 依赖 python 库说明：
  pip install ruamel_yaml
  pip install pathlib
  pip install 'protobuf>=3'
  pip install opencv-python

3.2 Caffe 模型优化训练

Caffe 模型优化训练分为如下步骤：

前提条件：准备模型训练数据集、全精度的基线模型 caffemodel、train.prototxt 和 test.prototxt

步骤 1 安装 Caffe 编译环境（3.2.1 安装编译环境）

步骤 2 配置 res_caffe_standalone.yaml 资源文件（3.2.2 配置资源文件）

步骤 3 优化 scene.yaml 策略配置（3.2.3 优化策略配置）

步骤 4 训练模型（3.2.4 训练模型）

步骤 5 转换模型（3.2.5 转换模型）

----结束

3.2.1 安装编译环境

3.2.1.1 Caffe-1.0 自动化构建

如果用户使用官方提供 Caffe-1.0 release 版本，可以支持量化环境的自动构建，操作如下：

Docker 环境构建

通过 dockefile 构建 image 时，会自动下载 caffe1.0，并完成 caffe-mod 的自动化编译。docker 中 caffe 部分的路径和信息如下：

- 下载的 caffe1.0 地址：/root/ddk/tools/tools_dopt/caffe/caffe-1.0
- 编译好的 caffe 地址：/root/ddk/tools/tools_dopt/caffe/caffe-mod
- docker 内的环境变量已经被默认设置为：

```
export PYTHONPATH=/root/ddk/tools/tools_dopt/caffe/caffe-mod/python:$PYTHONPATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/ddk/tools/tools_dopt/caffe/caffe-mod/libs/
```

Linux 环境构建

步骤 1 下载 Caffe release 版本，链接是：https://github.com/BVLC/caffe/archive/1.0.tar.gz

步骤 2 解压源码文件，放置在 caffe/caffe-1.0 目录

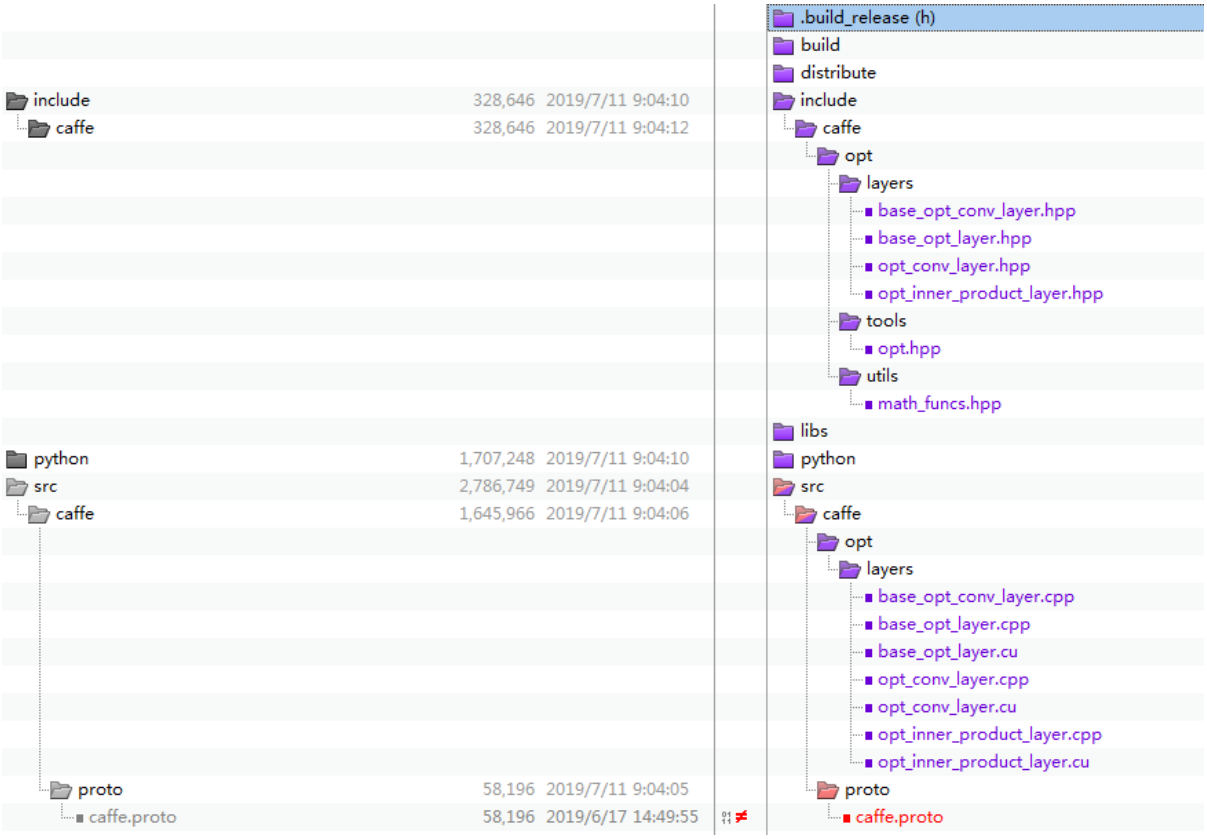
步骤 3 配置原生 Caffe 基础环境

步骤 4 执行 tools/tools_dopt/caffe/build_caffe.sh 脚本，该脚本自动将算法插件合入到 Caffe 源码中，并完成 Caffe 和 py-caffe 的编译

----结束

环境编译完成后，在 caffe/下新增了 caffe-mod/，caffe-1.0 与 caffe-mod 比较，如下图所示，轻量化算法插件已经合入到 Caffe 深度学习框架中了。

图3-1 caffe-mod 目录结构



3.2.1.2 非官方 Caffe 版本手动构建

如果用户使用非官方提供 Caffe release 版本，例如 Caffe-SSD，只能手动构建量化环境，操作如下：

Docker 环境构建

Docker 中的 caffe-mod 路径为：

- 下载的 caffe1.0 地址：/root/ddk/tools/tools_dopt/caffe/caffe-1.0
- 编译好的 caffe 地址：/root/ddk/tools/tools_dopt/caffe/caffe-mod

  用户需要将/root/ddk/tools/tools_dopt/caffe/caffe-mod 替换为用户自定义的 caffe 版本，并根据 3.2.1.2.2 进行手动构建

说明

如果用户手动构建过程中，改变 caffe-mod 路径，需要重新配置环境变化

Linux 环境构建

步骤 1 修改 caffe.proto。该文件的路径为：src\caffe\proto\caffe.protos。

在 message LayerParameter 定义尾部添加一行

```
// LayerParameter next available layer-specific ID: 147 (last added:
```

```
recurrent_param)
message LayerParameter {
  optional string name = 1; // the layer name
  …….
  optional OptParameter opt_param = 155;  // 155是id，需要不能与结构其他id相同
}
```

在整个文件结尾，添加 message OptParameter 结构定义【用户不能自定义同名结构】

```
message OptParameter {
  optional uint32 input_type        = 1 [default = 32];
  optional uint32 weight_type       = 2 [default = 2];

  optional uint32 algo_type         = 3 [default = 1];
  optional float input_scale        = 4 [default = 1.0];
  optional float input_offset       = 5 [default = 0.0];
  optional float weight_scale       = 6 [default = 1.0];
  optional float weight_offset      = 7 [default = 0.0];

  optional FillerParameter weight_q    = 8;    // Reserved Parameter
  optional FillerParameter w_scale     = 9;    // Reserved Parameter
  optional FillerParameter w_offset    = 10;   // Reserved Parameter
  optional FillerParameter i_max       = 11;   // Reserved Parameter
  optional FillerParameter i_min       = 12;   // Reserved Parameter
  optional FillerParameter moving_factor= 13;   // Reserved Parameter
}
```

步骤 2 复制 opt 文件到 src 目录。但是用户不能自定义相同的 layer 文件。

步骤 3 复制 libs 文件夹到 caffe 根目录。

步骤 4 在 Make.config 文件结尾添加

```
USE_NCCL := 1     ## 确认你是否使用了NCCL
USE_CUDNN := 1   ## 确认你是否使用了CUDNN
LIBRARY_DIRS += ./libs/
```

步骤 5 在 Make 文件中添加

```
# NCCL acceleration configuration
LIBRARIES += opt
ifeq ($(USE_NCCL), 1)
NVCCFLAGS += -Wno-deprecated-gpu-targets
      LIBRARIES += nccl
      COMMON_FLAGS += -DUSE_NCCL
endif
```

步骤 6 编译工程


```
make clean
make -j8     #编译caffe
make pycaffe   #编译caffe python接口
```

----结束

完成上述步骤后，Caffe 的量化环境构建完成。

3.2.2 配置资源文件

资源文件，用于配置带算法插件的 Caffe 框架所在位置。资源文件位于 caffe/目录下。

配置 caffe/目录下的 res_caffe_standalone.yaml 文件，用户需填写完整的框架路径，示例：

```
# Resources description
resource:
  name:            res_caffe_standalone
  framework:
    type:               caffe
    version:             "1.0"
    framework_path:       $PATH /caffe-mod/  #用户填写
    #computing type: (1) training (2)inference (3)both training and
inference
    computing_type:        3
```

3.2.3 优化策略配置

选择合适的优化策略，用户需自行完成优化策略配置文件 scen.yaml，如下示例：

```
# Optimization Scenario
scenario:
  strategy:
    name:                Quant_INT8-2
    framework:            caffe
    version:             "1.0"
    accuracy_name:        accuracy
    accuracy_val:         0.91
    skip_layers:
    model:               $PATH/basemodel.caffemodel
    train_prototxt:       $PATH/train.prototxt
    test_prototxt:        $PATH/test.prototxt
    test_iter:           100
    train_one_epoch_iter:   1000

  resource:
    name:                caffe_standalone
```



```
  gpu_id:               0
```

优化策略包含多个方面的参数，参数字典如下：

表3-1 优化策略参数说明表

参数名	类型(python)	取值范围	备注
strategy			
name	string	Quant_INT8-2, Quant_INT8-8	策略名称； Quant_INT8-2：数据 8bit+权重 2bit 量化。v310, v320 支持该策略。 Quant_INT8-8：数据 8bit+权重 8bit 量化。v320 支持该策略。
framework	string	caffe	基线模型训练框架类型
accuracy_name	string	--	accuracy 层的输出名称
accuracy_val	float	0 ~ 1.0	目标精度
skip_layers	string	--	模型不做轻量化的层
model	string	--	基线 caffemodel 地址
train_prototxt	string	--	训练模型配置 prototxt 文件路径
test_prototxt	string	--	测试模型配置 prototxt 文件路径
test_iter	int	--	测试迭代次数
train_one_epoch_iter	int	--	训练迭代次数
resource			
name	string	caffe_standalone	执行重训练量化的资源对象名称； caffe_standalone：Caffe 单机训练资源
gpu_id	string	--	指定使用的 gpuID；

3.2.4 训练模型

模型训练基于已经训练好的全精度基线模型进行训练，需要用户提供如下的文件：

表3-2 用户模型

文件	作用

文件	作用
basemodel.caffemodel	基线 caffemodel
train.prototxt	Caffe 训练模型的 prototxt
test.prototxt	Caffe 测试模型的 prototxt
数据集	训练测试使用

接着执行：python dopt_so.py -c scen.yaml，即开启训练。参数说明见下表。

说明

路径部分：支持大小写字母、数字，下划线；

文件名部分：支持大小写字母、数字，下划线和点(.)。

表3-3 重训练工具命令行参数说明表

参数名称	参数描述	是否必填
-c, --config	重训练量化配置文件路径。 量化配置文件说明请参考 3.2.3 优化策略配置	是

3.2.5 转换模型

进入 tools_dopt/dopt_trans_tools 目录，执行 trans_caffe.sh 完成模型转换。使用方式如下：

```
./trans_caffe.sh $PATH/opt_field/Sub_Task_$INDEX $PATH/caffe-mod
```

其中，第一个参数为模型训练时最后成功的 task 路径，第二个参数为 Caffe 环境路径。生成的新模型和轻量化配置文件位于 $PATH/opt_field/Sub_Task_$INDEX/transedmodel 内。

3.3 TensorFlow 模型优化训练

TensorFlow 模型优化训练分为如下步骤：

前提条件：准备模型训练数据集、全精度的基线 ckpt 文件

步骤 1 准备 TensorFlow 环境（3.3.1 准备 TensorFlow 环境）

步骤 2 配置模型接口定义（3.3.2 配置模型接口）

步骤 3 优化策略文件配置（3.3.3 优化策略配置）

步骤 4 训练模型（3.3.4 训练模型）

步骤 5 转换模型（3.3.5 转换模型）

----结束

3.3.1 准备 TensorFlow 环境

3.3.1.1 准备 Docker 环境

按照 3.1.1 准备 Docker 环境即可。

3.3.1.2 准备 Linux 环境

轻量化工具当前仅支持 tensorflow-gpu 1.12 版本。使用如下的命令安装：pip install tensorflow-gpu==1.12

轻量化工具当前使用 Horovod 进行多卡训练，当前支持单机多卡的训练。

Horovod 的安装，参考 horovod 官方教程：https://github.com/horovod/horovod#install

3.3.2 配置模型接口

轻量化工具针对 TensorFlow 框架的模型优化训练，用户需要按照如下的接口定义配置模型训练文件。接口定义具体如下：

表3-4 get_input_placeholder 接口描述

函数描述	用来创建和返回模型的输入的 placeholder
接口定义	def get_input_placeholder(self):
参数描述	无
返回值	返回一个 list； 包含 image input placeholder 和 labels input placeholder.

表3-5 get_next_batch 接口描述

函数描述	该函数用来读取一个 batch 的输入。
接口定义	def get_next_batch(self, is_train):
参数描述	is_train: 训练时为 True； 测试时为 False
返回值	返回一个 list。包括下一个 batch 的输入 image 和 label

表3-6 forward_fn 接口描述

函数描述	该函数通过读取 placeholder 格式的输入，定义模型的前向推理过程，并返回前向推理的输出结果。

接口定义	def forward_fn(self, inputs, is_train):
参数描述	inputs: get_input_placeholder 返回的 list is_train: 训练时为 True； 测试时为 False
返回值	一个 tensor，为前向推理的输出结果

表3-7 loss_op 接口描述

函数描述	该函数定义并返回模型的损失函数。
接口定义	def loss_op(self, inputs, outputs):
参数描述	inputs: get_input_placeholder 返回的 list outputs: forward_fn 返回的前向推理结果
返回值	一个 tensor，为用户定义的 loss

表3-8 metrics_op 接口描述

函数描述	该函数用来定义模型的评估方法。
接口定义	def metrics_op(self, inputs, outputs):
参数描述	inputs: get_input_placeholder 返回的 list outputs: forward_fn 返回的前向推理结果
返回值	一个 tensor，为用户定义的模型评估方法

表3-9 config_lr_policy 接口描述

函数描述	该函数用来设置模型的学习率
接口定义	def config_lr_policy(self, global_step):
参数描述	global_step: tensorflow 的 global step tensor
返回值	一个 tensor，包含 learning rate

上述接口分别定义了模型输入结构、输入数据、模型结构、accuracy 函数、loss 函数、学习率更新策略函数等六个接口。

下述示例代码，定义了基类 UserModelInterface，并定义了上述的 6 个抽象接口。 UserModel 类中，实现该六个接口的逻辑。 开发者可以根据该代码片段进行模型的定义。

图3-2 UserModelInterface 接口函数定义示例

```
from abc import ABCMeta
from abc import abstractmethod


class UserModelInterface:
    __metaclass__ = ABCMeta

    def __init__(self, dataset_dir, train_batch_size, test_batch_size, train_batch_num, test_batch_num):
        self.dataset_dir = dataset_dir
        self.train_batch_size = train_batch_size
        self.test_batch_size = test_batch_size
        self.train_batch_num = train_batch_num
        self.test_batch_num = test_batch_num

    @abstractmethod
    def get_input_placeholder(self):
        pass

    @abstractmethod
    def get_next_batch(self, is_train):
        pass

    @abstractmethod
    def forward_fn(self, inputs, is_train):
        pass

    @abstractmethod
    def loss_op(self, inputs, outputs):
        pass

    @abstractmethod
    def metrics_op(self, inputs, outputs):
        pass

    @abstractmethod
    def config_lr_policy(self, global_step):
        pass


class UserModel(UserModelInterface):
    def __init__(self, dataset_dir, train_batch_size, test_batch_size, train_batch_num, test_batch_num):
        super(UserModel, self).__init__(dataset_dir, train_batch_size, test_batch_size, train_batch_num, test_batch_num)
        self.data = None

    def get_next_batch(self, is_train):
        pass

    def get_input_placeholder(self):
        pass

    def forward_fn(self, inputs, is_train):
        pass

    def loss_op(self, inputs, outputs):
        pass

    def metrics_op(self, inputs, outputs):
        pass

    def get_batch_num(self, is_train):
        pass

    def config_lr_policy(self, global_step):
        pass
```

以下示例是一个图像分类应用的 mobilenetV1 模型接口的编写例程。

get_input_placeholder interface:

```
def get_input_placeholder(self):
images = tf.placeholder(tf.float32,[None, 224, 224, 3])
labels= tf.placeholder(tf.int32,[None, FLAGS.num_label])
return [images, labels]
```

get_next_batch interface:

以实现了一份读取 ImageNet 数据集的类为例，如下所示：

```
    class Ilsvrc12Dataset():
      """ILSVRC-12 dataset."""

      def __init__(self, data_dir, batch_size, is_train):
        """Constructor function.

        Args:
        * is_train: whether to construct the training subset
        """

      self.is_train = is_train

      # configure file patterns & function handlers
      if is_train:
        self.file_pattern = os.path.join(data_dir, 'train-*-of-*')
        self.batch_size = batch_size
      else:
        self.file_pattern = os.path.join(data_dir, 'validation-*-of-*')
        self.batch_size = batch_size
      self.dataset_fn = tf.data.TFRecordDataset
      self.parse_fn = lambda x: parse_fn(x, is_train=is_train)

    def build(self, enbl_trn_val_split=False):
      """Build iterator(s) for tf.data.Dataset() object.

      Args:
      * enbl_trn_val_split: whether to split into training & validation
subsets

      Returns:
      * iterator_trn: iterator for the training subset
      * iterator_val: iterator for the validation subset
        OR
      * iterator: iterator for the chosen subset (training OR testing)

      Example:
        # build iterator(s)
        dataset = xxxxDataset(is_train=True)  # TF operations are not
created
        iterator = dataset.build()          # TF operations are created
           OR
        iterator_trn, iterator_val =
```



```
dataset.build(enbl_trn_val_split=True)  # for dataset-train only

        # use the iterator to obtain a mini-batch of images & labels
        images, labels = iterator.get_next()
      """

      # obtain list of data files' names
      filenames = tf.data.Dataset.list_files(self.file_pattern, shuffle =
True)
      # if (self.is_train and FLAGS.enbl_multi_gpu):
      #     filenames = filenames.shard(mgw.size(), mgw.rank())

      # create a tf.data.Dataset from list of files
      dataset = filenames.apply(
        tf.contrib.data.parallel_interleave(self.dataset_fn, cycle_length
= FLAGS.cycle_length))

      dataset = dataset.map(self.parse_fn,
num_parallel_calls=FLAGS.nb_threads)
      # create iterators for training & validation subsets separately
      if self.is_train and enbl_trn_val_split:
        iterator_val =
self.__make_iterator(dataset.take(FLAGS.nb_smpls_val))
        iterator_trn =
self.__make_iterator(dataset.skip(FLAGS.nb_smpls_val))
        return iterator_trn, iterator_val

      return self.__make_iterator(dataset)

    def __make_iterator(self, dataset):
      """Make an iterator from tf.data.Dataset.

      Args:
      * dataset: tf.data.Dataset object

      Returns:
      * iterator: iterator for the dataset
      """

      dataset =
dataset.apply(tf.contrib.data.shuffle_and_repeat(buffer_size=FLAGS.buffer
_size))
      dataset = dataset.batch(self.batch_size)
      dataset = dataset.prefetch(FLAGS.prefetch_size)
```



```
      iterator = dataset.make_one_shot_iterator()


      return iterator
```

Ilsvrc12Dataset 类利用 tf.data.Dataset 完成了 ImageNet 数据集的读取。外部可以通过迭代器获取每个 batch 的数据。

基于 Ilsvrc12Dataset，可以完成 get_next_batch interface 的实现：

```
    def get_next_batch(self, is_train):
        sess = tf.get_default_session()
        if is_train == True:
            images, labels = sess.run([self.images_train_iter,
self.labels_train_iter])
        else:
            images, labels = sess.run([self.images_eval_iter,
self.labels_eval_iter])
        return [images, labels]
```

其中，self.images_train_iter 和 self.images_eval_iter 为 Ilsvrc12Dataset 类返回的 iterator，可以通过如下方式创建：

```
    self.dataset_train = DataSet(dataset_dir, train_batch_size,
is_train=True)
    self.iterator_train = self.dataset_train.build()
    self.images_train_iter, self.labels_train_iter =
self.iterator_train.get_next()


    self.dataset_eval = DataSet(dataset_dir, test_batch_size,
is_train=False)
    self.iterator_eval = self.dataset_eval.build()
    self.images_eval_iter, self.labels_eval_iter =
self.iterator_eval.get_next()
```

forward_fn interface:

示例如下：

```
    def forward_fn(self, inputs, is_train, data_format='channels_last'):
        images, labels = inputs
        nb_classes = FLAGS.num_label
        self.model_scope = "model"
        with tf.variable_scope(self.model_scope):
            scope_fn = mobilenet_v1.mobilenet_v1_arg_scope
            with slim.arg_scope(scope_fn(is_training=is_train)): # pylint:
disable=not-context-manager
                outputs, __ = mobilenet_v1.mobilenet_v1(images,
is_training=is_train, num_classes=nb_classes, depth_multiplier=1.0)
```



```
        return outputs
```

其中，mobilenet_v1.mobilenet_v1 为用户定义的模型。

loss_op interface:

使用基于 softmax 的交叉熵作为例子实现该函数：

```
    def loss_op(self, inputs, outputs):
        """Calculate loss (and some extra evaluation metrics)."""
        images, labels = inputs
        loss = tf.losses.softmax_cross_entropy(labels, outputs)
        return loss
```

metrics_op:

以分类模型为例，在 metrics_op 中应当定义为模型的分类精度，实现如下：

```
    def metrics_op(self, inputs, outputs):
        images, labels = inputs
        accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(labels,
axis=1), tf.argmax(outputs, axis=1)), tf.float32))
        return accuracy
```

config_lr_policy:

如下所示：

```
    def config_lr_policy(self, global_step):
         return 1e-4
```

3.3.3 优化策略配置

选择合适的优化策略，用户需自行完成优化策略配置文件 scen.yaml，如下示例：

```
# Optimization Scenario
scenario:
  strategy:
    name:             Quant_INT8-8
    framework:          TensorFlow
    version:            "1.12"
    accuracy_val:        0.98
    skip_layers:
    optimizer:
       type:           adam
    model:             $PATH/user_module.py
    base_model:          $PATH/basemodel.ckpt
    dataset_dir:           $PATH/dataset/
    train_batch_size:      600
    train_data_num:       60000
```



```
  test_batch_size:       100
  test_data_num:        10000
  epoch:                2

resource:
  name:              tensorflow_standalone
  gpu_id:             0
```

优化策略包含多个方面的参数，参数字典如下：

参数名		类型（python）	范围	备注
strategy				
name		string	Quant_INT8-2[1]，Quant_INT8-8	Quant_INT8-2：数据8bit+权重 2bit 量化。v310, v320 支持该策略。 Quant_INT8-8：数据8bit+权重 8bit 量化。v320 支持该策略。
framework		string	TensorFlow	基线模型训练框架类型
version		string	"1.12"	基线模型训练框架版本
accuracy_val		float	0 ~ 1.0	目标精度
skip_layers		string	--	模型不做轻量化的层
optimizer	type	string	adam / momentum	优化器类型； 支持： momentum（默认值） adam
	momentum	float	0 ~ 1.0	动量（仅在 momentum 优化器时有效），默认值 0.9
model		string	--	用户模型.py 接口文件路径
base_model		string	--	基线模型 ckpt 文件路径
dataset_dir		string	--	训练数据集路径
train_batch_size		int	--	训练每迭代 batch size
train_data_num		int	--	训练数据集样本数量
test_batch_size		int	--	测试每迭代 batch Size

参数名	类型（python）	范围	备注
test_data_num	int	--	测试数据集样本数量
epoch	int	--	数据集轮训次数
resource			
name	string	tensorflow_standalone	执行重训练量化的资源对象名称； tensorflow_standalone：tensorflow 单机训练资源
gpu_id	string	--	指定使用的 gpuID； 填写一个 gpu id，如 0，则使用单机单卡模式进行训练； 填写多个 gpu id，如 0,1,2,3，则使用单机多卡模型进行训练

3.3.4 训练模型

模型训练基于已经训练好的全精度基线模型进行训练，需要用户提供如下的文件：

文件	作用
用户模型.py 接口文件	训练测试使用
基线模型 ckpt 文件	基线模型
数据集	训练测试使用

接着执行：python dopt_so.py -c scen.yaml，即开启训练。参数说明见下表。

说明

路径部分：支持大小写字母、数字，下划线；

文件名部分：支持大小写字母、数字，下划线和点(.)。

表3-10 重训练工具命令行参数说明表

参数名称	参数描述	是否必填
-c, --config	重训练量化配置文件路径。 量化配置文件说明请参考 3.3.3 优化策略配置	是

3.3.5 转换模型

进入 tools_dopt/dopt_trans_tools 目录，执行 trans_tensorflow.sh 完成模型转换。使用方式如下：

```
./trans_tensorflow.sh $PATH/opt_field/Sub_Task_$INDEX  output_op
```

其中，第一个参数为模型训练时最后成功的 task 路径，第二个参数为模型的输出节点。生成的新模型和轻量化配置文件位于 $PATH/opt_field/Sub_Task_$INDEX/curmodel/transedmodel 内。

4 网络结构搜索训练

网络结构搜索训练请按照如下步骤进行：

步骤 1 准备环境（4.1 环境准备）

步骤 2 准备数据集（4.2 数据集准备）

步骤 3 配置搜索参数（4.3 搜索参数配置）

步骤 4 配置用户接口（4.4 自定义接口）

步骤 5 搜索和训练网络结构（4.5 搜索训练）

----结束

4.1 环境准备

网络结构搜索工具运行环境主要分两种情况（Docker 环境和 Linux 环境），用户可任选其一。两种环境都可以支持单机单卡和一机多卡运行。

4.1.1 Docker 环境

镜像制作需要依赖的环境如下：

Nvidia-docker 版本：nvidia-docker2

安装方式参考：https://github.com/NVIDIA/nvidia-docker

步骤 1 解压 ddk 包，并进入 tools 文件夹所在的目录：“cd ./$DDK_PATH/”

步骤 2 HiAIML 的 docker 镜像生成，参考如下方式构建：

- 如果用户设置代理：

```
docker build -f tools/tools_dopt/env/docker_tf1.12/Dockerfile -t
hiai_ddk:v320 --build-arg HTTP_PROXY="http://xxxxxx@xxxxxx.com:8080" --
no-cache .
```

其中选项说明如下：

- -f 指定要使用的 Dockerfile 路径

- -t 镜像的名字及标签，通常是 name:tag 或 name 格式，如默认的 hiai_ddk:v320
- --build-arg 设置镜像创建时的变量，HTTP_PROXY 表示代理配置项，如果用户不需设置代理，此字段值为空
- --no-cache 创建镜像的过程不使用缓存
- . 指定镜像构建过程中的上下文环境的目录

  注意：命令最后的“.”不要遗漏。

步骤 3 HiAIML 的 docker 容器生成，参考如下方式启动：

```
nvidia-docker run -d -it --restart=always --net=host --privileged --name my_docker -v /user_data:/data hiai_ddk:v320
```

其中选项说明如下：

- -d 以 daemon 形式在后台执行
- -i 以交互模式运行容器，通常与 -t 同时使用；
- -t 为容器重新分配一个伪输入终端，通常与 -i 同时使用
- --restart=always 机器重启时，会自动重启容器
- --net=host 使用宿主机网卡，容器以宿主机 IP 和外部进行交互
- --privileged 当运行多卡搜索时，需要较高权限，因此务必添加此选项
- --name 指定 docker 容器名字
- -v 用于映射宿主机目录到容器内
- hiai_ddk:v320 为镜像名

----结束

4.1.2 Linux 环境

步骤 1 HiAIMLEA 策略运行环境的依赖在 requirements.txt 文件中，通过如下的方式安装：

```
cd $DDK_PATH/tools/tools_dopt/tensorflow/dopt/py3/sh
hiaimlea_requirements.sh
```

说明

- $DDK_PATH 表示用户解压的 DDK 路径。
- 如果仅需要在单机单卡运行工具，可跳过后续步骤。如需要一机多卡环境运行，请继续完成如下步骤。

步骤 2 参考如下网站进行安装和配置 Horovod 和 Open MPI：

https://github.com/horovod/horovod#install

步骤 3 安装 mpi4py

```
pip3 install mpi4py
```

步骤 4 验证安装

下载 Horovod 官方对应的 Demo 到当前目录:

https://github.com/horovod/horovod/blob/master/examples/tensorflow_mnist.py

在一台服务器上利用 4 张加速卡运行如下命令：

```
horovodrun -np 4 -H localhost:4 python3 tensorflow_mnist.py
```

若正常训练，则 Horovod 环境部署成功。

----结束

4.2 数据集准备

用户根据需要准备数据集或代理数据集。

数据集的接口定义请参见表 4-3。

4.3 搜索参数配置

配置合适的搜索参数，以达到较优的搜索结果。用户需自行完成配置文件（可参考 tools/tools_dopt/demo/hiaiml/ea_cls_imagenet/scen.yaml），如下示例：

```
# Network architecture search scenario
scenario:
  strategy:
    name:              HiAIMLEA
    framework:          TensorFlow
    batch_size:         128
    epochs:            60
    constraint:
      application_type:  "image_classification"
      constraint_type:   "size"
      constraint_value:  11000000
    supernet:
      input_shape:       (224, 224, 3)
      data_format:       "channels_last"
      filters:          [64, 64, 128, 128, 256, 256]
      strides:          [1, 1, 2, 1, 2, 1]
      # feature_choose:    [4, 5]
    optimizer:
      weights_optimizer:
        type:           "Adam"
        betas:          [0.9, 0.999]
        learning_rate:   0.0001
    dataset:
      # pre_train_dir:    "/tmp/tfrecords"
```



```
    train_dir:        "/tmp/ImageNet_tf/"
    val_dir:          "/tmp/ImageNet_tf/"
  searcher:
    generation_num:    100
    pop_size:         40
resource:
  name:              tensorflow_standalone
  gpu_id:            "0,1,2,3,4,5,6,7"
```

搜索参数包含多个方面，参数字典如下：

说明

所有参数区分大小写。

表4-1 搜索参数字典

参数名称	类型	取值范围	参数描述
scenario.strategy			
name	string	HiAIMLEA	**必选项**，策略名
framework	string	TensorFlow	**必选项**，基线模型训练框架类型
batch_size	int	--	**必选项**，数据集的 batch_size
epochs	int	--	**必选项**，数据集轮训次数
scenario.strategy.supernet			
input_shape	tuple	--	可选项，模型输入 shape 的 CHW 或 HWC 的维度，例：(224, 224, 3)
data_format	string	channels_first / channels_last	可选项，数据格式，input_shape 和 data_format 取值需要对应，例："channels_last"
filters	list	--	可选项，搜索骨架每层的 cout，与 strides 对应的列表，格式为 [cout, ..., cout]，例：[64, 64, 128, 128, 256, 256]
strides	list	--	可选项，搜索骨架每层使用的 stride，例：[1, 1, 2, 1, 2, 1]
feature_choose	list	--	检测场景可选项，需要融合的待搜索层，以逗号分隔，从 0 开始计数，例：融合第 4、6 层：[3, 5]
scenario.strategy.constraint			

参数名称	类型	取值范围	参数描述
application_type	string	image_classification / object_detection / semantic_segmentation	**必选项**，应用类型，支持分类、检测和分割场景
constraint_type	string	size / flops	**必选项**，模型约束类型
constraint_value	int	--	**必选项**，模型约束取值，整个网络的大小
scenario.strategy.optimizer			
scenario.strategy.optimizer.weights_optimizer			
type	string	SGD / Momentum / Adam	可选项，优化器[1]，分类检测默认值："Adam"，分割默认值："Momentum"
betas	list	(0, 1)	可选项，衰减因子，Adam 优化器参数，默认值：[0.9, 0.999]
learning_rate	float	(0, 1)	可选项，学习率，多 GPU 时工具会自动翻倍，分类检测默认值：0.0001，分割默认值：0.00001
momentum	float	(0, 1)	可选项，动量，Momentum 优化器参数，默认值：0.9
scenario.strategy.dataset			
pre_train_dir	string	--	分类场景无需此字段，检测分割场景**必选项**，预训练数据集路径或预训练生成的 ckpt 路径[2]。
train_dir	string	--	**必选项**，训练数据集路径
val_dir	string	--	**必选项**，验证数据集路径
scenario.strategy.searcher			
generation_num	int	--	可选项，进化算法的代数，例：100
pop_size	int	--	可选项，进化算法的种群数量，默认值：40
scenario.resource			
name	string	tensorflow_standalone	**必选项**，资源对象名称
gpu_id	string	--	**必选项**，指定使用的 gpuID，多个时以逗号分隔，例：

参数名称	类型	取值范围	参数描述
			"0,1,2,3,4,5,6,7"

注释：

[1] 优化器类型支持 SGD，Momentum 和 Adam 三种优化器类型。

- SGD 优化器，支持 learning_rate 参数。
- Momentum 优化器，支持 momentum、learning_rate 参数。
- Adam 优化器，支持 betas、learning_rate 参数，betas 的形式为 list，list 中索引为 0 的元素为 beta1，索引为 1 的元素为 beta2。

[2] 当没有预训练的 ckpt 文件时，其取值为预训练数据集的目录路径；当有预训练的 ckpt 文件时，其取值为 ckpt 所在的目录路径。

4.4 自定义接口

网络结构搜索基于 TensorFlow 框架进行训练，用户可参考 tools/tools_dopt/demo/hiaiml/ea_cls_imagenet 中的 user_module.py 配置训练接口，函数接口具体定义如下。

说明

仅支持 tf.keras 实现。

UserModule 类

表4-2 UserModule 类

类描述	Class **UserModule** - 定义用户侧接口
函数描述	构造函数
接口定义	def **__init__**(self, epoch, batch_size):

此类用于定义搜索过程中的以下几个函数：

表4-3 数据集读取

函数描述	数据集读取函数
接口定义	def **build_dataset_search**(self，dataset_dir, is_training, is_shuffle):
参数描述	dataset_dir: 数据集路径 is_training: 训练时为 True； 推理时为 False。 is_shuffle: 数据集是否需要 shuffle。提示，在 evalution 阶段

	更新 bn 时，数据集不需要做 shuffle。
返回值	• 训练流程： 分类、分割场景： iterator: TensorFlow 数据集迭代器。 n_data_num: 每次迭代的数据集长度。 检测场景： train_generator:训练集生成器。 train_dataset_size: 训练集数量。 • 推理流程： 分类、分割场景： iterator: TensorFlow 数据集迭代器。 n_data_num: 每次迭代的数据集长度。 检测场景： val_generator:验证集生成器。 val_dataset: 解析 json 文件后的数据集。 val_dataset_size:验证集数量。

表4-4 学习率更新策略

函数描述	学习率更新策略函数
接口定义	def **lr_scheduler**(self, lr_init, global_step):
参数描述	lr_init: 学习率的初始值。 global_step: TensorFlow 的 global step。
返回值	已更新的学习率。

注意：建议使用常数

表4-5 评估函数

函数描述	评估函数
接口定义	def **metrics_op**(self, inputs, outputs):
参数描述	• 分类、分割场景： – inputs: ground truth labels. – outputs: 前向推理的结果。 • 检测场景： – inputs: [valid_dir, model, block_choice]

	valid_dir: 验证集路径。 model: 网络模型。 block_choice: 挑选的网络结构。 – outputs: [data_generator, proxy_val_image_ids, data_size] data_generator: 数据集生成器。 proxy_val_image_ids: 代理数据集图片索引。 data_size: 数据集大小。
返回值	评估结果。

表4-6 Loss 计算函数

函数描述	Loss 计算函数
接口定义	def **loss_op**(self, labels, logits):
参数描述	labels:ground truth labels. logits: 前向推理的结果。
返回值	loss 值，tensor。

PreNet 类

模型中输入层不需要搜索，因此通过固定网络结构的形式定义。

类描述	Class **PreNet** - 模型输入层
函数描述	PreNet 构造函数
接口定义	def **__init__**(self):
参数描述	NA
返回值	NA
函数描述	构建模型的输入结构。
接口定义	def **call**(self, inputs, training=True):
参数描述	inputs: 输入数据。 training: 训练时为 True； 推理时为 False。
返回值	搜索骨架的输入， tensor。

PostNet 类

模型中输出层不需要搜索，因此通过固定网络结构的形式定义。

类描述	Class **PostNet** - 模型输出层
函数描述	PostNet 构造函数
接口定义	def **__init__**(self):
参数描述	NA
返回值	NA
函数描述	构建模型的输出结构
接口定义	def **__call__**(self, inputs, feature_layer=None, training=True):
参数描述	inputs: 搜索骨架的输出。 feature_layer: 需要在 PostNet 中进行融合的待搜索的 layer。分类和分割场景不涉及，仅检测涉及。 training: 训练时为 True； 推理时为 False。
返回值	模型输出，tensor。

4.5 搜索训练

主要从以下三个方面介绍工具的训练入口，维测方式以及搜索结果展示。

4.5.1 训练入口

执行 python3 $DDK_PATH/tools/tools_dopt/tensorflow/dopt/dopt_so.py -c scen.yaml，即开启搜索训练。针对每种场景都有对应的 demo 目录入口，详见 5.5 TensorFlow HiAIMLEA 网络结构搜索 Demo

4.5.2 维测方式

搜索训练过程可以利用 Tensorboard 进行观测中间信息，生成的信息文件保存在 log_* 目录。

图4-1 loss - 模型精度损失 loss 曲线

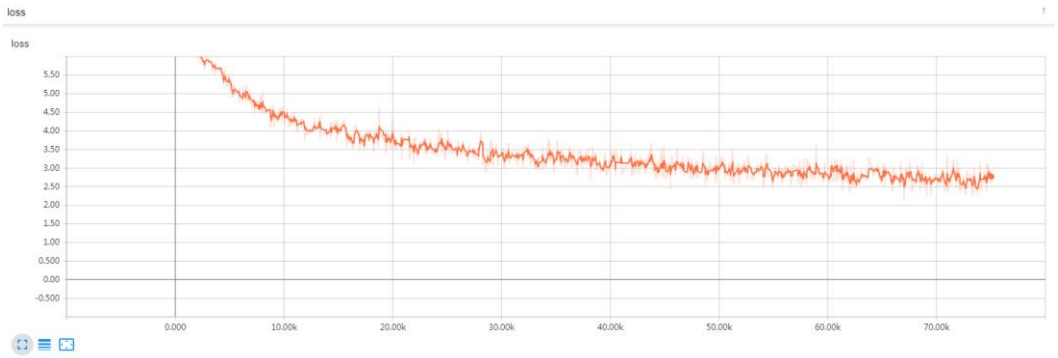


图4-2 lr –学习率变化曲线

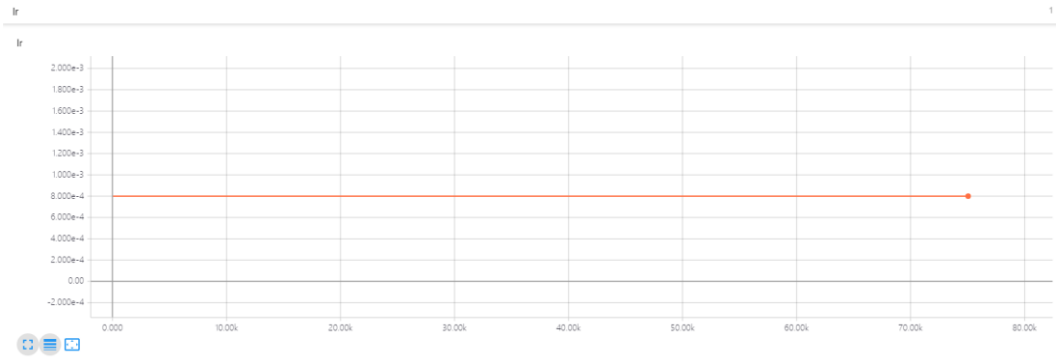
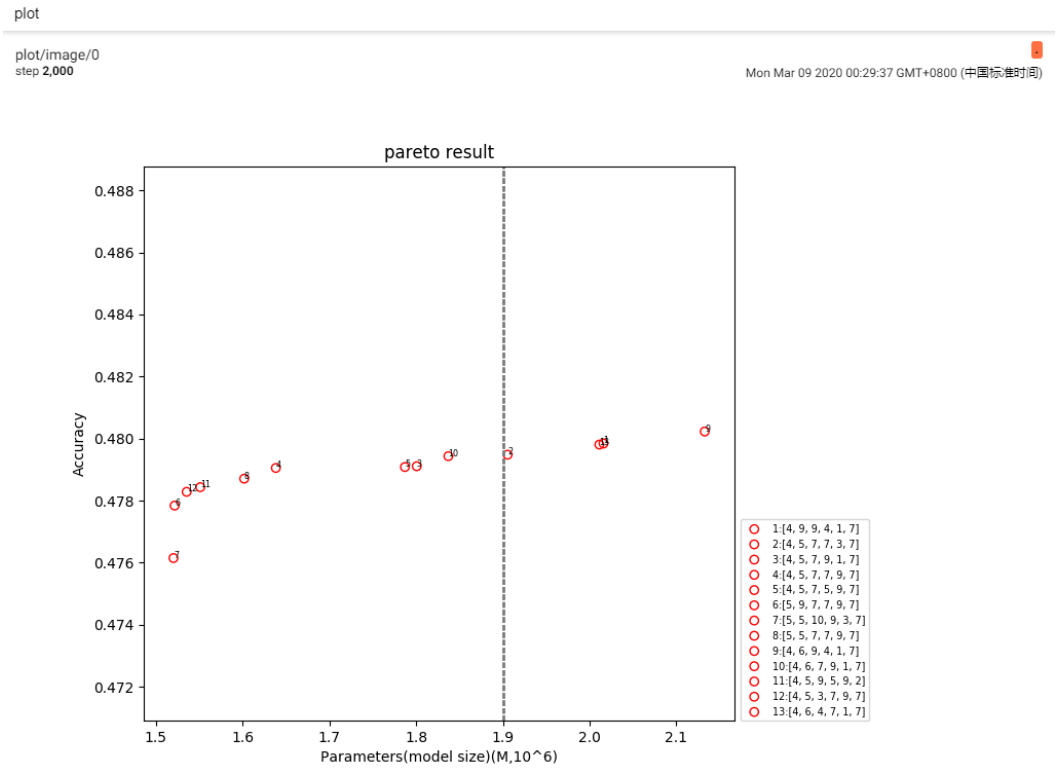


图4-3 pareto-帕累托前沿图



帕累托图横坐标为模型大小或计算量即约束项，纵坐标为结构搜索后的精度。图中的精度为搜索过程的评估结果，如果要获得更好的精度，建议对搜索结构进行充分训练。

搜索模型结构具有不同的精度、参数量/计算量/时延，用户可根据实际需求选择合适的模型。

4.5.3 搜索结果展示

搜索结束后，工具会自动将 pareto front 图中模型结构保存在 results 目录，生成多个 model_arch_result_$NUM.py 文件。其中$NUM 文件编号与 pareto 图上的编号一致，头部有 model_param_size 和 accuracy，用户可根据 log 中的 pareto 图或者这两个参数选择合适的网络结构，如下所示：

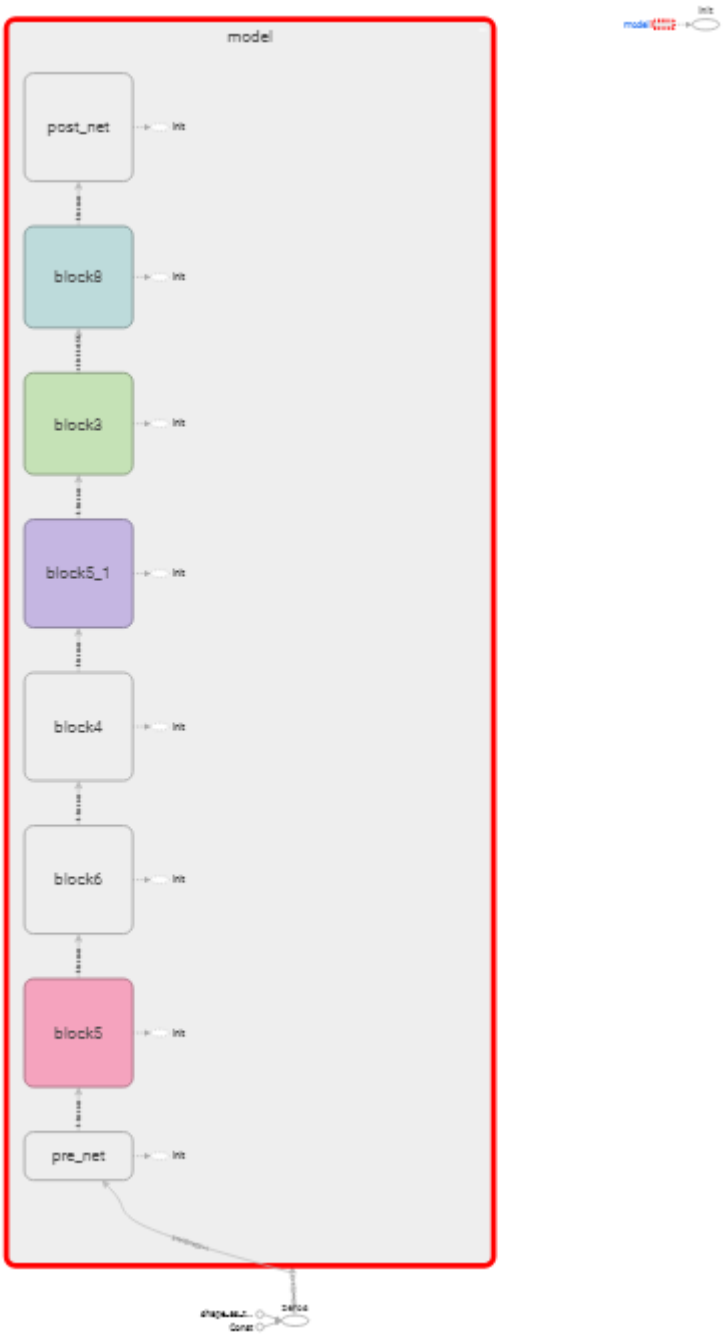
```
# model_param_size:2058496.0
# Accuracy:0.480621

from user_module import PreNet
from user_module import PostNet
from blocks import *


class Model(tf.keras.Model):
    def __init__(self):
        super(Model, self).__init__()
        self.pre_net = PreNet()
        self.post_net = PostNet()
        self.block5_1 = Block5(64, 64, 1, scope='block5_1')
        self.block6_2 = Block6(64, 64, 1, scope='block6_2')
        self.block4_3 = Block4(64, 128, 2, scope='block4_3')
        self.block5_4 = Block5(128, 128, 1, scope='block5_4')
        self.block3_5 = Block3(128, 256, 2, scope='block3_5')
        self.block8_6 = Block8(256, 256, 1, scope='block8_6')

    def call(self, inputs, is_training):
        out = self.pre_net(inputs, is_training)
        out = self.block5_1(out, is_training)
        out = self.block6_2(out, is_training)
        out = self.block4_3(out, is_training)
        out = self.block5_4(out, is_training)
        out = self.block3_5(out, is_training)
        out = self.block8_6(out, is_training)
        out = self.post_net(out, training=is_training)

        return out

if __name__ == '__main__':
    with tf.Graph().as_default():
        fake_input = tf.zeros([1, 224, 224, 3], tf.float32)

        config = tf.ConfigProto()
        config.gpu_options.visible_device_list = str(0)
        sess = tf.Session(config=config)

        model = Model()
        out = model(fake_input, True)

        sess.run(tf.global_variables_initializer())
        summary_writer = tf.summary.FileWriter('.', sess.graph)

        with sess.as_default():
            sess.run(out)
```


用户选定合适的模型结构文件，可以拷贝到 results 的上一级目录，并执行模型结构文件。

```
python3 model_arch_result_$NUM.py
```

执行结束后，当前目录下会生成模型的 pb 文件和 TensorBoard 日志文件，用户可通过 tensorboard 查看模型的图结构。如下：

图4-4 网络结构对应的 graphs



后续训练请参考 5.5 TensorFlow HiAIMLEA 网络结构搜索 Demo 中 readme.md 的指导。

5 示 例

5.1 Caffe Quant_INT8-8 无训练量化 Demo

5.1.1 环境准备

环境准备请参照“2.2.1 环境准备”，安装 caffe 及依赖。

5.1.2 资源配置

- 准备量化模型

  将基线模型的 prototxt 和 caffemodel 文件放入 demo/quant8-8/notrain/caffe_mnist/basemodel/中。该路径下已经放入了 mnist 基线模型，mnist.caffemodel 和 mnist.prototxt。

- 准备量化输入数据

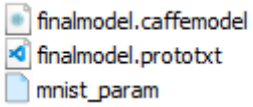
  参照“2.1.2 准备校准集”，将图片或二进制形式的校准集放入 demo/quant8-8/notrain/caffe_mnist/mnist_test/中。该路径下已经放入了图片校准集。

5.1.3 模型量化

将 demo/quant8-8/notrain/caffe_mnist/run_release.sh 中的--caffe_dir 参数改为用户使用的 caffe 源码路径，执行 run_release.sh 即可。

demo/quant8-8/notrain/caffe_mnist/curmodel 中存有量化后的 prototxt、caffemodel 和量化配置文件，如下所示：

图5-1 运行 demo 后生成的文件



5.2 Tensorflow Quant_INT8-8 无训练量化 Demo

5.2.1 环境准备

环境准备请参照“2.3.1 环境准备”，安装 tensorflow 及依赖。

5.2.2 模型配置

- 准备量化模型

  将基线模型的 pb 文件放入 demo/quant8-8/notrain/tensorflow_mnist/basemodel/。该路径下中已经放入了 mnist 基线模型，mnist.pb。

- 准备量化输入数据

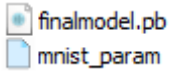
  参照“2.2.2 模型量化”，将图片或二进制形式的校准集放入 demo/quant8-8/notrain/tensorflow_mnist/mnist_test/中。该路径下已经放入了图片校准集。

5.2.3 模型量化

执行 demo/quant8-8/notrain/tensorflow_mnist/下 run_release.sh 即可。

demo/quant8-8/notrain/tensorflow_mnist/curmodel 中存有量化后的 pb 模型和量化配置文件，如下所示：

图5-2 运行 demo 后生成的文件



5.3 Caffe 重训练量化 Demo

5.3.1 数据集准备

步骤 1 从官网（http://yann.lecun.com/exdb/mnist/）下载 mnist 数据集。

步骤 2 通过该脚本（https://github.com/BVLC/caffe/blob/master/examples/mnist/create_mnist.sh），将下载下来的 mnist 数据集转换为 lmdb 格式。

步骤 3 修改 tools_dopt/demo/quant8-8/retrain/caffe_mnist_single_gpu/basemodel 中 prototxt 文件的数据集路径。

----结束

5.3.2 环境准备

请参考“3.2.1 安装编译环境”。安装 caffe 及其依赖

5.3.3 模型配置

步骤 1 修改 config 目录下的 res_caffe_standalone.yaml 文件中的 framework_path 值，请参照“3.2.2 配置资源文件”。

步骤 2 模型文件 caffemodel 以及在 demo 中准备好，并已经放置于 demo/quant8-8/retrain/caffe_mnist_single_gpu/basemodel/中。

----结束

5.3.4 优化策略配置

配置 demo/quant8-8/retrain/caffe_mnist_single_gpu/下的 scen.yaml。本用例中，已经提前填写好了参数，用户只需要根据用户本地环境修改路径既可。

请参照节内容 3.2.3 优化策略配置。

5.3.5 模型训练

执行 demo/quant8-8/retrain/caffe_mnist_single_gpu/run_release.sh，即可以执行重训练量化。

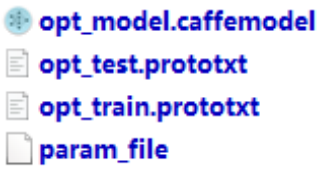
5.3.6 模型转换

查看 demo/quant8-8/retrain/caffe_mnist_single_gpu/opt_field 中生成的 task。

修改 dopt_trans_tools/run_caffe_trans_demo.sh 并执行。

生成的新模型位置： demo/quant8-8/retrain/caffe_mnist_single_gpu/opt_field/Sub_Task_2/transedmodel

图5-3 caffe 模型转换后的文件列表



5.4 TensorFlow 重训练量化 Demo

5.4.1 数据集准备

步骤 1 从官网（http://yann.lecun.com/exdb/mnist/）下载 mnist 数据集。

步骤 2 并修改 tools_dopt/demo/quant8-8/retrain/tensorflow_mnist_single_gpu/scen.yaml 中的 dataset_dir 为正确的数据集路径。

----结束

5.4.2 环境准备

请参考“3.3.1 准备 TensorFlow 环境”。安装 tensorflow-gpu 1.12 版本以及其必要的依赖。

5.4.3 模型配置

步骤 1 并按照“3.3.2 配置模型接口”，完成接口的配置，编写用模型接口文件。本用例中的 basemodel 目录下，已提供了该用户模型接口定义文件 tools_dopt/demo/quant8-8/retrain/tensorflow_mnist_single_gpu/basemodel/mnist_model.py。

步骤 2 提供预训练的 TensorFlow 模型，包括 checkpoint、model.ckpt.data-00000-of-00001、model.ckpt.index、model.ckpt.meta。本用例中的 basemodel 目录下，已经提供了该预训练模型

----结束

5.4.4 优化策略配置

配置 demo/quant8-8/retrain/tensorflow_mnist_single_gpu/下的 scen.yaml 请参照“3.3.3 优化策略配置”。本用例中，已经提前填写好了参数，用户只需要根据用户本地环境修改路径既可。

5.4.5 模型训练

执行 demo/quant8-8/retrain/tensorflow_mnist_single_gpu/run_release.sh 即可。

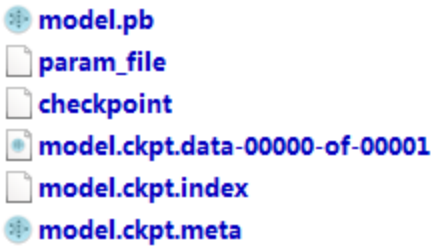
5.4.6 模型转换

查看 demo/quant8-8/retrain/tensorflow_mnist_single_gpu/opt_field 中生成的 task。

修改 dopt_trans_tools/run_tensorflow_trans_demo.sh 并执行。

生成的新模型位置： demo/quant8-8/tensorflow_mnist_single_gpu/opt_field/Sub_Task_1/curmodel/transedmodel

图5-4 tensorflow 模型转换后的文件列表

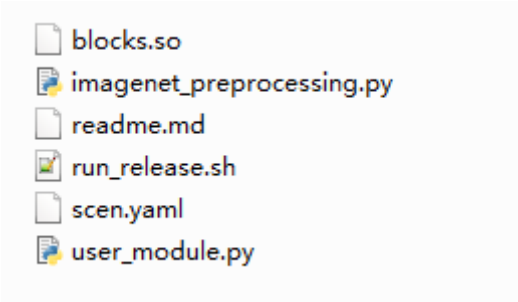


5.5 TensorFlow HiAIMLEA 网络结构搜索 Demo

5.5.1 HiAIML 分类网络

分类网络 Demo 位于 tools/tools_dopt/demo/hiaiml/ea_cls_imagenet，包含 6 个文件，如下：

图5-5 HiAIML 分类网络 demo



- blocks.so 是搜索空间文件
- imagenet_preprocessing.py 是 tensorflow-models 开源代码，处理 ImageNet 数据
- readme.md 是对后续训练给出指导
- run_release.sh 是开始搜索的执行脚本
- scen.yaml 是配置项
- user_module.py 是工具的自定义接口

执行步骤：

步骤 1 准备数据集，并修改 scen.yaml 文件中的数据集路径。

步骤 2 环境准备请参照“4.1 环境准备”。

步骤 3 配置 demo 下的 scen.yaml 文件，请参照“4.3 搜索参数配置”。scen.yaml 中提供了建议参数，用户可根据实际需求修改。

步骤 4 修改 demo 下的 user_module.py 文件，模型接口定义请参照“4.4 自定义接口”。user_module.py 中提供了建议配置，用户可根据实际需求进行修改。

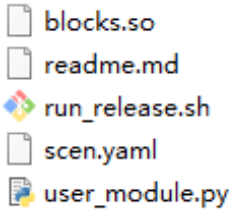
步骤 5 执行脚本 run_release.sh，在 results 下，生成多个 model_arch_result_*.py 文件。用户可根据 log_classifaction 中提供的信息选择合适的网络结构进行训练。后续训练可参考 readme.md 中的指导。

----结束

5.5.2 HiAIML 检测网络

检测网络 Demo 位于 tools/tools_dopt/demo/hiaiml/ea_det_coco，包含 5 个文件，如下：

图5-6 HiAIML 检测网络 demo



- blocks.so 是搜索空间文件
- readme.md 是对后续训练给出指导
- run_release.sh 是开始搜索的执行脚本
- scen.yaml 是配置项
- user_module.py 是工具的自定义接口

执行步骤：

步骤 1 准备数据集，包括预训练数据集 ImageNet 和训练数据集 COCO。若有完成预训练的 ckpt 文件，则不需再准备 ImageNet 数据集。请参照“4.3 搜索参数配置”修改 scen.yaml 文件中的数据集路径。

步骤 2 环境准备请参照“4.1 环境准备”。

步骤 3 加载依赖的开源代码：

1. 进入检测网络 demo 目录：

```
cd tools/tools_dopt/demo/hiaiml/ea_det_coco
```

2. 下载开源代码：

```
git clone https://github.com/pierluigiferrari/ssd_keras.git
```

3. 进入开源代码目录：

```
cd ssd_keras
```


4. 切换到指定版本：

```
git checkout -b v0.9.0
```

5. 按照 readme.md 中的 step1、step2、step3 步骤，修改相关开源文件；
6. 按照 readme.md 中的 step4 步骤，修改 user_module.py；

步骤 4 配置 demo 下的 scen.yaml 文件，请参照“4.3 搜索参数配置”。scen.yaml 中提供了建议参数，用户可根据实际需求修改。

步骤 5 修改 demo 下的 user_module.py 文件，模型接口定义请参照“4.4 自定义接口”。user_module.py 中提供了建议配置，用户可根据实际需求进行修改。

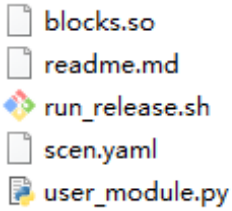
步骤 6 执行脚本 run_release.sh，在 results 下，生成多个 model_arch_result_*.py 文件。用户可根据 log_detection 中提供的信息选择合适的网络结构进行训练。后续训练可参考 readme.md 中的指导。

----结束

5.5.3 HiAIML 分割网络

分割网络 Demo 位于 tools/tools_dopt/demo/hiaiml/ea_seg_voc，包含 5 个文件，如下：

图5-7 HiAIML 分割网络 demo



- blocks.so 是搜索空间文件
- readme.md 是对后续训练给出指导
- run_release.sh 是开始搜索的执行脚本
- scen.yaml 是配置项
- user_module.py 是工具的自定义接口

执行步骤：

步骤 1 准备数据集，包括预训练数据集 ImageNet 和训练数据集 VOC。若有完成预训练的 ckpt 文件，则不需再准备 ImageNet 数据集。请参照“4.3 搜索参数配置”修改 scen.yaml 文件中的数据集路径。

步骤 2 环境准备请参照“4.1 环境准备”。

步骤 3 加载依赖的开源代码：

1. 进入分割网络 demo 目录：

```
cd tools/tools_dopt/demo/hiaiml/ea_seg_voc
```

2. 下载开源代码：

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

3. 进入开源代码目录：

```
cd models
```

4. 切换到指定版本：

```
git checkout v1.13.0
```

5. 返回分割网络 demo 目录：

```
cd ..
```

6. 修改开源实现:

   当前路径为“ea_seg_voc”。

   修改 train_utils.py，该文件的路径为：“.\models\research\deeplab\utils\train_utils.py”。

   - 将第 72 行的“slim.one_hot_encoding”修改为“tf.one_hot”；
   - 在第 74 行的“tf.losses.softmax_cross_entropy”前面增加“return”；

7. 设置 PYTHONPATH 默认路径：

```
export
PYTHONPATH=$PYTHONPATH:`pwd`/models/research:`pwd`/models/research/slim
```

   注意：每次打开终端需要重新执行一次上述命令，或添加到“~/.bashrc”文件，并执行“source ~/.bashrc”。

步骤 4 配置 demo 下的 scen.yaml 文件，请参照“4.3 搜索参数配置”。scen.yaml 中提供了建议参数，用户可根据实际需求修改。

步骤 5 修改 demo 下的 user_module.py 文件，模型接口定义请参照“4.4 自定义接口”。user_module.py 中提供了建议配置，用户可根据实际需求进行修改。

步骤 6 执行脚本 run_release.sh，在 results 下，生成多个 model_arch_result_*.py 文件。用户可根据 log_segmentation 中提供的信息选择合适的网络结构进行训练。后续训练可参考 readme.md 中的指导。

----结束

6 附录

6.1 常见问题 Q&A

Q1：模型有多个输入时如何准备数据和填写配置文件？

A：如果模型定义了多个输入，用户需要为每个输入节点各准备一份 IMAGE 或 BINARY 模式的校准集。如不同节点所需的输入数据存在对应关系，推荐使用 BINARY 模式，防止由于读取图片的顺序不同导致预期之外的行为。在填写量化配置文件时，需要定义与输入节点个数相同的预处理参数，预处理参数的顺序则需要与用户运行工具时中指定的 input_shape 顺序一致。

Q2：出现 Unsupported image format! Unsupported image: xxx 问题应该如何处理？

A：图片校准集中放入了不支持的图片格式的文件，删除该文件即可

Q3：模型重训练优化完成后进行模型转换时，如何选取 Sub_Task 进行转换？

A：选择最后一个成功的 Sub_Task 进行转换，可以得到满足精度约束下最优的模型性能

Q4：模型重训练优化生成的 caffe-mod 是否会影响原生模型的训练和推理？

A：caffe-mod 只补充了必要的模型优化层，没有改变原生 caffe 的层定义，不影响原生模型的训练和推理

Q5：如果只使用单 GPU 进行 tensorflow 模型重训练优化，是否可以不安装 horovod 环境？

A：可以不安装 horovod 环境

6.2 模型收益

以 resnet-18 为例，使用轻量化工具后（Quant_INT8-2 量化）的收益如下：

表6-1 Quant_INT8-2 量化收益

框架	数据集	模型	原始精度	重训练后精度	非量化 DaVinci 模型体积 (MB)	Quant_INT8-2 量化 DaVinci 模型体积 (MB)
Caffe	ImageNet	resnet-18(v1)	66.6%	66.1%	22.405	2.99
TensorFlow		resnet-18(v2)	70.0%	69.0%	22.457	5.15

以 resnet-18 为例，使用轻量化工具后（Quant_INT8-8 量化）的收益如下：

表6-2 Quant_INT8-8 量化收益

框架	数据集	模型	原始精度	重训练后精度	非量化 DaVinci 模型体积 (MB)	Quant_INT8-8 量化 DaVinci 模型体积 (MB)
Caffe	ImageNet	resnet-18(v1)	66.6%	67.3%	22.405	11.3
TensorFlow		resnet-18(v2)	70.0%	69.0%	22.457	11.9

以 resnet-18 为例，使用轻量化工具后（网络结构搜索）的收益如下：

表6-3 网络结构搜索工具分类场景收益对比

网络	模型	参数量（M）	精度
ResNet-18（ImageNet 数据集）	ResNet-18	11.69	70.32%
	HiAIMLEA	10.93	72.13%[1]

表6-4 网络结构搜索工具检测场景收益对比

网络	模型	计算量（G）	mAP@[.5, .95][2]
SSD(backbone：ResNet-18)（COCO 数据集）	ResNet-18	11.6	17.8[3]
	HiAIMLEA	9.25	18.4[3]

表6-5 网络结构搜索工具分割场景收益对比

网络	模型	计算量（G）	mIOU[4]
Deeplab(backbone：ResNet-18)（VOC 数据集）	ResNet-18	40.9	54.1[5]
	HiAIMLEA	28.3	56.1[5]

注释：

[1] 此精度是使用 tensorpack 重训练模型得出。

[2] 此指标的计算方法为：IoU(Intersection over Union)从 0.5~0.95 区间上，以 0.05 为间隔计算 AP 的值，再计算所有 AP 的均值。

[3] 此精度是在 COCO val2017 数据集上测试得出。

[4] 此指标为平均交并比，计算方法为先求每个类别的交并比，再平均。

[5] 此精度是在 VOC val2012 数据集上测试得出。

