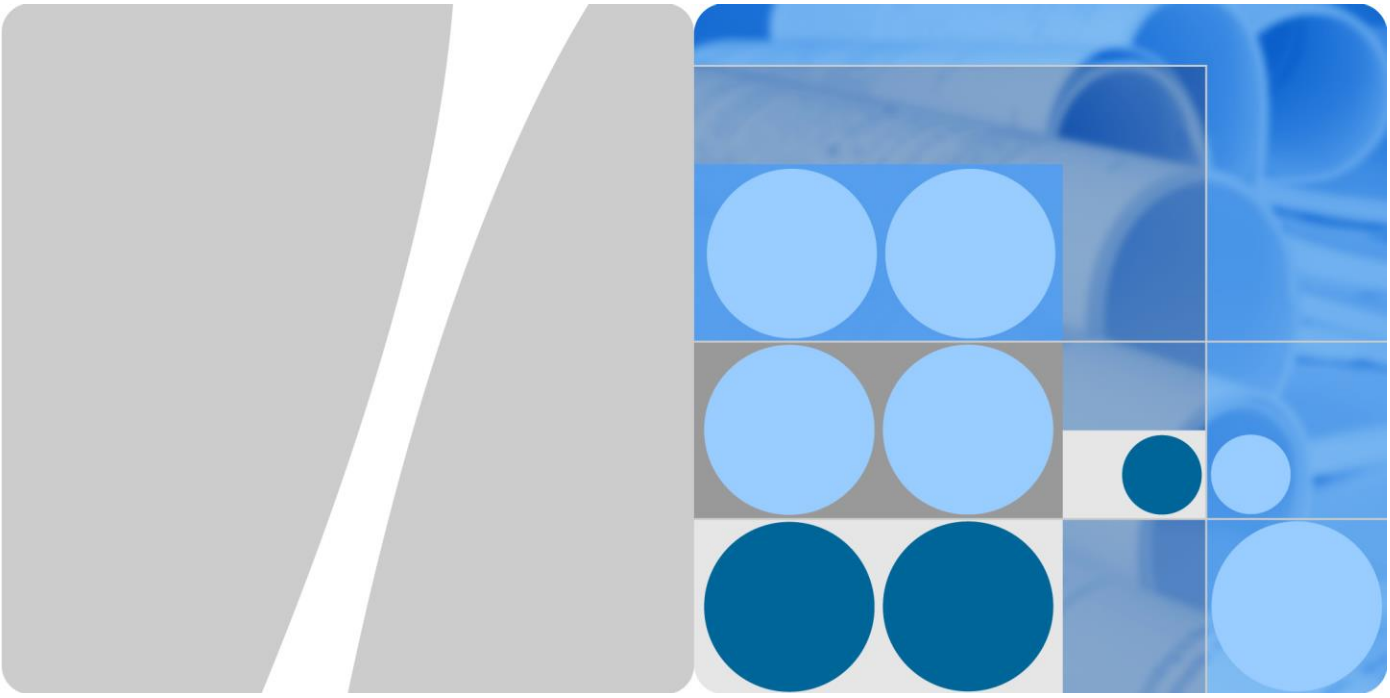


HiAI DDK V320

Lightweight Tool Instructions

Issue 03

Date 2020-03-16

Huawei Technologies Co., Ltd.

Huawei HiAI Application

Send an application email to developer@huawei.com.

Email subject: HUAWEI HiAI + Company name + Product name

Email body: Cooperation company + Contact person + Phone number + Email address

We will reply to you within 5 working days.

Official website: https://developer.huawei.com/consumer/en/

About This Document

Purpose

This document describes how to use the lightweight tool.

Change History

Date	Version	Change Description
2020-03-16	03	Added the description of network architecture search (NAS).
2019-12-31	02	Added the description of HiAI DDK V320.
2019-09-04	01	Added the description of HiAI DDK V310.

Contents

Tables

1 Introduction

1.1 Overview

With multiple model compression algorithms and NAS algorithms, the lightweight tool can optimize deep-learning neural network models according to the neural processing unit (NPU) architecture, helping users automatically build lightweight models and generate network architectures. Currently, the non-training mode, retraining mode, and NAS mode are supported.

- Non-training mode: A user can input a model directly without retraining it to build a lightweight model quickly. In the non-training scenario, the Quant_INT8-8 and Quant_Weight_INT8 lightweight algorithm are supported. This mode is suitable for users who want to use the tool quickly and conveniently.
- Retraining mode: A user can retrain a pre-trained full-precision base model and tailor the model at acceptable precision loss. In the retraining scenario, the Quant_INT8-8 and Quant_INT8-2 lightweight algorithms are supported. This mode is designed for scenarios with high requirements on precision.
- NAS mode: Supports three service types, namely, classification, detection, and segmentation networks. After configuring required search parameters and functions, the user can use the NAS tool to search for network models that meet the computing power restrictions. This mode is applicable to the scenario involving network architecture generation.

	Supported Framework	Supported Strategy	Supported Device
Non-training mode	Caffe and TensorFlow	Quant_INT8-8 and Quant_Weight_INT8	Both CPU and GPU are supported. GPU supports single-machine single-GPU mode.
Retraining mode	Caffe and TensorFlow	Quant_INT8-8 and Quant_INT8-2	GPU is supported. GPU supports single-

	Supported Framework	Supported Strategy	Supported Device
			machine single-GPU mode and single-machine multi-GPU mode.
NAS mode	TensorFlow	HiAIMLEA	GPU is supported. GPU supports single-machine single-GPU mode and single-machine multi-GPU mode.

For details about the model benefits on the NPU through model lightweight and NAS, see section 6.2 "Model Benefits."

NOTE

- Quant_INT8-8: indicates 8-bit weight quantization and 8-bit data quantization.
- Quant_Weight_INT8: indicates 8-bit weight quantization only, without data quantization.
- Quant_INT8-2: indicates 2-bit weight quantization and 8-bit data quantization.
- HiAIMLEA: indicates NAS based on the genetic algorithm.

After tailoring the model by using the lightweight tool, you can convert the model to a Da Vinci model by using the Offline Model Generator (OMG). For details about how to use the OMG, see the *Huawei HiAI DDK V320 OMG Tool Instructions*.

The lightweight tool is stored in the **tools/tools_dopt** directory of the DDK.

Table 1-1 Directory tools_dopt

Directory	Description
tools\tools_dopt\caffe	.so files and source code used for Caffe retraining
tools\tools_dopt\tensorflow	.so files used for TensorFlow retraining
tools\tools_dopt\dopt_trans_tools	Tool for model conversion after model retraining
tools\tools_dopt\demo	Caffe and TensorFlow sample models
tools\tools_dopt\config	Configuration script of the framework information used by the user, for example, the path of Caffe
tools\tools_dopt\Huawei HiAI DDK V320 Lightweight Tool	Lightweight tool instructions

Directory	Description
Instructions	

1.2 Scope

- GPU-based retraining, including single-machine single-GPU mode and single-machine multi-GPU mode
- CPU and GPU quantization in non-training mode
- The NAS tool supports skeleton search of classification, detection, and segmentation networks in single-machine single-GPU and single-machine multi-GPU scenarios. For details about the operating environment, see section 4.1 "Preparing the Environment."
- The optimization strategies vary according to the HiAI DDK version. The following table lists the mapping between HiAI versions and optimization strategies.

Table 1-2 Mapping between HiAI DDK versions and optimization strategies

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

1.3 Requirements

Before using this tool, ensure that the following requirements are met:

- Python 3.6
- Ubuntu 16.04
- GPU(s) supporting CUDA® graphics cards
- CUDA 9 and cuDNN 7.6.2

1.4 Tutorial

1.4.1 Caffe User and 1.4.2 TensorFlow User describe the procedures and toolchain functions in different user scenarios. You can select an appropriate optimization mode based on the scenarios listed in the following table.

1.4.1 Caffe User

Table 1-3 Caffe tutorial

User Scenario	Preparation	Toolchain Functions	Support Failure	Recommended Tool
Insufficient datasets and training resources; High usability requirement	Caffe environment; deploy.prototxt; Caffemodel; Calibration set	INT8-8 quantization; Weight INT8 quantization No need to recompile the Caffe model, friendly to beginners	None	Quantization in non-training mode -> Caffe model quantization in non-training mode
High precision requirement; Sufficient datasets and training resources; Certain programming capability	Caffe environment; Test.prototxt; Train.prototxt; Caffemodel; Training dataset; Test dataset; (The Caffe model needs to be recompiled.)	INT8-8 quantization; INT8-2 quantization; Quantization model precision test; Hybrid-precision quantization; Guaranteed model precision	You can use PyCaffe instead of prototxt to define models. The Caffe source code may need to be modified.	Quantization in retraining mode -> Caffe model training for optimization

1.4.2 TensorFlow User

Table 1-4 TensorFlow tutorial

User Scenario	Preparation	Toolchain Functions	Support Failure	Recommended Tool
Insufficient datasets and	TensorFlow	INT8-8	None	Quantization in

User Scenario	Preparation	Toolchain Functions	Support Failure	Recommended Tool
training resources; High usability requirement	environment; Model file (.pb); (Users need to convert the format.) Calibration set	quantization; Weight INT8 quantization No need to write code by using TensorFlow, friendly to beginners		non-training mode -> TensorFlow model quantization in non-training mode
High precision requirement; Sufficient datasets and training resources; Certain programming capability	TensorFlow environment; Model file (.ckpt); Training dataset; Test dataset; Users need to write the model API code based on the instructions.	INT8-8 quantization; INT8-2 quantization; Quantization model precision test; Hybrid-precision quantization; Guaranteed model precision	None	Quantization in retraining mode -> TensorFlow model training for optimization
Requirement for automatically generating an applicable network architecture	Environment described in section 4.1 "Preparing the Environment"; Training dataset; Test dataset; Users need to write the model API code based on the instructions.	Network architecture search and generation	None	NAS tool

2 Quantization in Non-Training Mode

2.1 Preparation

Perform the following steps:

Step 1 Prepare the base model (2.1.1 Preparing a Model).

Step 2 Prepare the calibration set (2.1.2 Preparing the Calibration Set).

Step 3 Edit the strategy configuration in **config.prototxt** (2.1.3 Editing the config.prototxt File).

----End

2.1.1 Preparing a Model

- Caffe user

  You need to provide the .prototxt and .caffemodel files for quantization.

- TensorFlow user

  You need to provide the PB model to be quantified.

2.1.2 Preparing the Calibration Set

You need to provide the calibration set in the .bin or image format. The input data in .bin format must be stored in the formats, as shown in Table 2-1. The image data is stored in the folder for storing test images. By default, the input data of the image format is read from the BGR color image. The read data format is NCHW, where **N** indicates the number of provided images.

NOTE

The lightweight tool supports images in the following formats: ".bmp", ".dib", ".jpeg", ".jpg", ".jpe", ".png", ".webp",".pbm", ".pgm", ".ppm", ".tiff",".tif",".BMP", ".DIB", ".JPEG", ".JPG", ".JPE", ".PNG",".WEBP",".PBM", ".PGM", ".PPM", ".TIFF", and ".TIF".

The input data in .bin format supports two definition modes, which are applicable to the input data of any dimension and the input data of four dimensions respectively.

For input data of any dimension, the .bin file needs to be defined as follows. For example, for 3-dimensional data (50, 100, 300), the read data shape is (50, 100, 300).

Table 2-1 Description of the binary calibration set of any dimension

File Header/Data	Address Offset	Type	Value	Description
File header (20 bytes in total)	0000	32bit int	610	Magic number When the magic number is **610**, it is used to verify the validity of a file.
	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
Data	...	Float32	...	Data volume = 50 x 100 x 300

For 4-dimensional input data, the .bin file may be defined according to Table 2-1 or Table 2-2. For example, for 4-dimensional data (50, 3, 28, 28), the read data shape is (50, 3, 28, 28).

Table 2-2 Description of the binary calibration set of four dimensions

File Header/Data	Address Offset	Type	Value	Description
File header (20 bytes in total)	0000	32bit int	510	Magic number When the magic number is **510**, it is used to verify the validity of a file.
	0004	32bit int	50	Input num
	0008	32bit int	3	Input channels

File Header/Data	Address Offset	Type	Value	Description
	0012	32bit int	28	Input height
	0016	32bit int	28	Input width
Data	...	Float32	...	Data volume = 50 x 3 x 28 x 28

When the calibration set in image format is provided, this tool provides the data pre-processing modes such as resizing, mean subtraction, and standard deviation division. The resize method is the same as that of cv2.resize. The resize algorithm is INTER_LINEAR.

2.1.3 Editing the config.prototxt File

Table 2-3 describes the config.prototxt parameters. In BINARY mode, no pre-processing is performed. In IMAGE mode, pre-processing is performed based on the mean value and standard deviation provided by the user.

Table 2-3 Description of config.prototxt parameters

Parameter	Description	Mandatory or Not
strategy	Optimization strategy. **Quant_INT8-8** (default) **Quant_Weight_INT8**	No
device	Whether to use GPU or CPU for quantization. USE_GPU: GPU mode USE_CPU: CPU mode	Yes
exclude_op	The following methods are supported: 1. Use one exclude_op, which contains multiple op_name, separated by semicolons (;). 2. Use multiple exclude_op, each of which contains an op_name. The two methods can be used together. If the op_name is unavailable, an error will be reported.	No
preprocess_parameter	Pre-processing and input parameters. **preprocess_parameter** contains the	Yes

Parameter	Description	Mandatory or Not
	following parameters: input_type image_format input_file_path mean_value standard_deviation	

Table 2-4 describes the sub-parameters of **preprocess_parameter**. If a model has multiple inputs, **preprocess_parameter** must be configured for each input.

Table 2-4 Description of the **preprocess_parameter** parameters

Parameter	Description	Mandatory or Not
input_type	Whether to use the binary format or image format for input BINARY: binary format IMAGE: image format	Yes
image_format	Image format for input BGR: BGR image format RGB: RGB image format The default value is **BGR**. This parameter is optional.	No
mean_value	Mean value of image preprocessing The value is of the float type and ranges from 0.0 to 255.0. The number of **mean_value** must be the same as the input C dimension. The default value is **0.0**. This parameter is optional and takes effect only in IMAGE mode.	No
standard_deviation	Standard deviation used by the image and the preprocessing The value is of the float type and must be greater than or equal to 0.0. This parameter takes effect only in IMAGE mode.	No
input_file_path	Absolute path of the input calibration set, that is, the path of the .bin file, or the	Yes

Parameter	Description	Mandatory or Not
	folder that stores images Example: **/path/to/user/data**	

The following is a configuration example:

- BINARY mode:

```
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 mode:

```
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 Model Quantization in Non-Training Mode

Perform the following steps:

Step 1 Prepare the Caffe environment (2.2.1 Preparing the Environment).

Step 2 Quantize the model (2.2.2 Quantizing the Model).

----End

2.2.1 Preparing the Environment

If the Caffe or PyCaffe environment is unavailable, you need to install the compilation environment. Otherwise, you can skip the installation.

Install the Caffe environment as follows:

```
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
```

The source code is located in:

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

2.2.2 Quantizing the Model

Run **python ${ROOT}/caffe/dopt/py${PY_VER}/dopt_so.py**, where **${ROOT}** indicates the path of the release package and **py${PY_VER}** indicates the Python version in use.

The parameters for running the script are as follows:

NOTE

The path can contain uppercase letters, lowercase letters, digits, and underscores (_).

The file name can contain uppercase letters, lowercase letters, digits, underscores (_), and periods (.).

Table 2-5 Parameters in the command lines for the non-training mode

Parameter	Description	Mandatory or Not
-m, --mode	Running mode 0: Non-training mode 1: Retraining mode	Yes
--framework	Deep learning framework 0: Caffe 3: TensorFlow	Yes

Parameter	Description	Mandatory or Not
--weight	Path of the weight file. This parameter needs to be specified when the source model framework is Caffe.	Yes
--model	Path of the Caffe prototxt file	Yes
--cal_conf	Path of the quantization configuration file for the calibration mode. Currently, the Convolution, Full Connection, and ConvolutionDepthwise operators for weight, offset, and data quantization are supported. For details about the quantization configuration file, see 2.1.3 Editing the config.prototxt File.	Yes
--output	Path of the model file after quantization, for example, **/path/to/out/resnet18.caffemodel**	Yes
--input_format	Input data format, which can be NHWC or NCHW. When you select the IMAGE format or the .bin file whose file header is **510** as the input data and select NHWC as the input data format, the tool automatically adjusts the channel sequence. When you select the .bin file whose file header is **610** as the input data, the tool will not adjust the channel sequence.	Yes
--input_shape	Shape of the input data, for example, "input_name1: n1, c1, h1, w1; input_name2: n2, c2, h2, w2". **input_name** must be the node name in the network model before model conversion. If there are multiple inputs, separate their shapes **input_shape** by using semicolons (;). The value of **input_shape** must be consistent with the input node specified in the model before conversion. See the following two examples: If the input node is **input_shape_network: none, 224, 224, 3**, with the last three dimensions specified, the second, third, and fourth dimensions of **input_shape** must be consistent with **input_shape_network**. If the input node is **input_shape_network: 1, 224, 224, 3**, with all the four dimensions	No

Parameter	Description	Mandatory or Not
	specified, **input_shape** can only be (1, 224, 224, 3).	
--out_nodes	Output node, for example, "node_name1; node_name1; node_name2". **node_name** must be the node name in the network model before model conversion.	No
--compress_conf	Path of the binary file converted from the model file, for example, **param_file**. This file is a lightweight configuration file. The file converted using the OMG will be used as the input of the **compress_conf** parameter.	Yes
--caffe_dir	path of the Caffe source code	Yes
--device_idx	GPU or CPU device ID	No

After the quantization script is executed, the .prototxt and .caffemodel files are generated in **--output**, and the quantization configuration file is generated in **--compress_conf**. For example, if you input **quantmodel.caffemodel** for **--output** and **param** for **compress_conf**, the **quantmodel.prototxt**, **quantmodel.caffemodel**, and **param** files will be generated.

2.3 TensorFlow Model Quantization in Non-Training Mode

Perform the following steps:

Step 1 Prepare the TensorFlow environment (2.3.1 Preparing the Environment).

Step 2 Quantize the model (2.3.2 Quantizing the Model).

----End

2.3.1 Preparing the Environment

This function supports TensorFlow 1.12 CPU or GPU. If the required environment is unavailable, you need to install it. Otherwise, you can skip the installation.

To install TensorFlow, run the following command:

pip install tensorflow-gpu==1.12 or **pip install tensorflow==1.12**

2.3.2 Quantizing the Model

Run **python ${ROOT}/tensorflow/dopt/py${PY_VER}/dopt_so.py**, where **${ROOT}** indicates the path of the release package and **py${PY_VER}** indicates the Python version in use.

The parameters for running the script are as follows:

NOTE

The path can contain uppercase letters, lowercase letters, digits, and underscores (_).

The file name can contain uppercase letters, lowercase letters, digits, underscores (_), and periods (.).

Table 2-6 Parameters in the command lines for the non-training mode

Parameter	Description	Mandatory or Not
-m, --mode	See Table 2-5.	Yes
--framework	See Table 2-5.	Yes
--model	Path of the source model file. The PB model is supported.	Yes
--cal_conf	See Table 2-5.	Yes
--output	Absolute path of the model file after quantization, for example, **/path/to/out/resnet18.pb**	Yes
--input_format	See Table 2-5.	Yes
--input_shape	See Table 2-5.	Yes
--out_nodes	See Table 2-5.	Yes
--compress_conf	See Table 2-5.	Yes
--device_idx	See Table 2-5.	No

After the quantization script is executed, the .pb file is generated in **--output**, and the quantization configuration file is generated in **—compress_conf**. For example, if you input **quantmodel.pb** for **--output** and **param** for **--compress_conf**, the **quantmodel.pb** and **param** files will be generated.

3 Quantization in Retraining Mode

3.1 Preparing the Dependency Environment

Both Docker and Linux environments are supported for retraining quantization. You can prepare one of them.

NOTE

After the retraining environment is configured, quantization supports both non-training and retraining modes.

3.1.1 Preparing the Docker Environment

The following environment is required for image creation:

Nvidia-docker packages: **nvidia-docker2**

Install the packages by referring to https://github.com/NVIDIA/nvidia-docker.

Step 1 Decompress the DDK package and run **cd ./$DDK_PATH/** to go to the directory of the **tools** folder.

Step 2 Run the following command to generate a Docker image for retraining quantization:

- If a proxy server is set:

```
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
```

- If no proxy server is set:

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

Where,

- **-f** specifies the Dockerfile path.
- **-t** specifies the image name and tag in the **name:tag** or **name** format. The default value is **hiai_ddk:v320**.

- **--build-arg** sets the variables for creating the image.
- **HTTP_PROXY** sets the proxy.
- **--no-cache** specifies that no cache is used when creating an image.

Step 3 Run the following command to start a Docker container for retraining quantization:

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

Where,

- **-d**: runs the commands in the background in daemon mode.
- **-i**: starts the container in interactive mode. **-i** is often used in conjunction with **-t**.
- **-t**: reallocates a pseudo input terminal to the container. **-t** is often used with **-i**.
- **--restart=always:**: restarts the container automatically upon machine startup.
- **--net=host**: uses the host IP address to interact with external systems when the host NIC is used.
- **--privileged**: grants the multi-GPU NAS with required privilege.
- **--name**: name of the docker container.
- **-v**: maps the host machine directory to the container.
- **hiai_ddk:v320**: image name.

----End

3.1.2 Preparing the Linux Environment

To use the retraining function of the lightweight tool, you need to prepare the following environments and dependencies:

- Description of the dependent Python libraries:

  pip install ruamel_yaml

  pip install pathlib

  pip install 'protobuf>=3'

  pip install opencv-python

3.2 Training the Caffe Model

Perform the following steps:

Prerequisites: The model training datasets and full-precision base models (caffemodel, train.prototxt, and test.prototxt) have been prepared.

Step 1 Install the Caffe compilation environment (3.2.1 Installing the Compilation Environment).

Step 2 Configure the **res_caffe_standalone.yaml** resource file (3.2.2 Configuring Resource Files).

Step 3 Optimize the strategy configuration in **scene.yaml** (3.2.3 Configuring the Optimization Strategy).

Step 4 Train the model (3.2.4 Training the Model).

Step 5 Convert the model (3.2.5 Converting the Model).

----End

3.2.1 Installing the Compilation Environment

3.2.1.1 Automatic Build of Caffe-1.0

If you use the official Caffe-1.0 release, the quantization environment can be automatically built as follows.

Building the Docker Environment

When you build an image using a dockefile, Caffe-1.0 is automatically downloaded and a **caffe-mod** folder is automatically built. The related Caffe paths in Docker are as follows:

- Caffe-1.0 download path: **/root/ddk/tools/tools_dopt/caffe/caffe-1.0**
- Caffe build output path: **/root/ddk/tools/tools_dopt/caffe/caffe-mod**
- The related environment variables in Docker are configured as follows by default:

```
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/
```

Building the Linux Environment

Step 1 Download the Caffe release version from the following link: https://github.com/BVLC/caffe/archive/1.0.tar.gz.

Step 2 Decompress the source code file to the **caffe/caffe-1.0** directory.

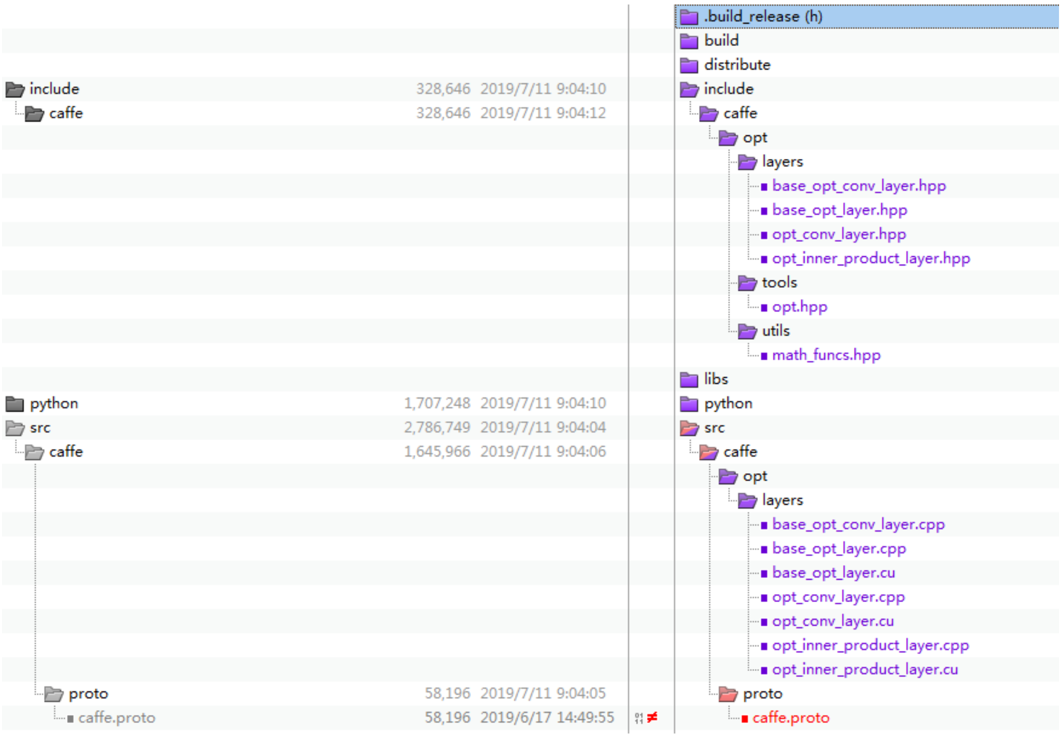
Step 3 Configure the basic environment of native Caffe.

Step 4 Run the **tools/tools_dopt/caffe/build_caffe.sh** script. This script automatically integrates the algorithm plug-in into the Caffe source code and compiles Caffe and PyCaffe.

----End

After the environment compilation is complete, the **caffe-mod/** directory is generated in **caffe/**. The following figure shows the comparison between caffe-1.0 and caffe-mod. The lightweight algorithm plug-in has been integrated into the Caffe deep learning framework.

Figure 3-1 Directory structure of caffe-mod



3.2.1.2 Manual Build of Unofficial Caffe Versions

If you use an unofficial Caffe release version, for example, Caffe-SSD, you can only manually build the quantization environment as follows:

Building the Docker Environment

The related Caffe paths in Docker are as follows:

- Caffe-1.0 download path: **/root/ddk/tools/tools_dopt/caffe/caffe-1.0**
- Caffe build output path: **/root/ddk/tools/tools_dopt/caffe/caffe-mod**

  You need to replace the **caffe-mod** folder in the **/root/ddk/tools/tools_dopt/caffe** directory with the user-defined **caffe-mod** folder and manually build the user-defined **caffe-mode** by referring to section 3.2.1.2 "Manual Build of Unofficial Caffe Versions."

NOTE

If the **caffe-mod** path is changed during manual building, you need to reconfigure the environment.

Building the Linux Environment

Step 1 Modify the caffe.proto file, which is stored in src\caffe\proto\caffe.protos.

Add a line at the end of the **message LayerParameter** definition.

```
// 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 is the ID, which must be different from
other IDs in the structure.
}
```

Add the **message OptParameter** structure definition to the end of the file. (You cannot customize the structure with the same name.)

```
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
}
```

Step 2 Copy the opt file to the **src** directory. However, you cannot customize the same layer file.

Step 3 Copy the **libs** folder to the Caffe root directory.

Step 4 Add the following content to the end of the **Make.config** file:

```
USE_NCCL := 1        ## Check whether you use NCCL.
USE_CUDNN := 1     ## Check whether you use the cuDNN.
```



```
LIBRARY_DIRS += ./libs/
```

Step 5 Add the following content to the **Make** file:

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

Step 6 Build the project.

```
make clean
make –j8        # Build Caffe.
make pycaffe    # Build Caffe Python APIs.
```

----End

Then, the Caffe quantization environment is built.

3.2.2 Configuring Resource Files

Resource files are used to configure the location of the Caffe framework with algorithm plug-ins. The resource files are stored in the **caffe/** directory.

Configure the **res_caffe_standalone.yaml** file in the **caffe/** directory. You need to specify a complete framework path. The following provides an example.

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

3.2.3 Configuring the Optimization Strategy

Select a proper optimization strategy and configure it in **scen.yaml**. The following provides an instance.

```
# 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
```

An optimization strategy contains multiple parameters, as described in Table 3-1.

Table 3-1 Optimization strategy parameters

Parameter	Type (Python)	Value Range	Remarks
strategy			
name	string	Quant_INT8-2, Quant_INT8-8	Strategy name Quant_INT8-2: indicates 8-bit data quantization and 2-bit weight quantization. V310 and V320 support this strategy. Quant_INT8-8: indicates 8-bit data quantization and 8-bit weight quantization. V320 supports this strategy.
framework	string	caffe	Framework type of the trained base model
accuracy_name	string	--	Output name of the accuracy layer
accuracy_val	float	0–1.0	Target accuracy

Parameter	Type (Python)	Value Range	Remarks
skip_layers	string	--	Layers that do not need lightweight implementation by the model
model	string	--	Path of the base model file (.caffemodel)
train_prototxt	string	--	Path of the .prototxt training model configuration file
test_prototxt	string	--	Path of the .prototxt test model configuration file
test_iter	int	--	Number of test iterations
train_one_epoch_iter	int	--	Number of training iterations
resource			
name	string	caffe_standalone	Name of the resource object for quantization in retraining mode **caffe_standalone**: single-machine Caffe training resource
gpu_id	string	--	GPU ID.

3.2.4 Training the Model

Apart from a pre-trained full-precision base model, the following files are also required.

Table 3-2 User model

File	Description
basemodel.caffemodel	Base Caffe model
train.prototxt	prototxt file of the Caffe training model
test.prototxt	prototxt file of the Caffe test model
Dataset	Dataset for training and testing

Run the following command to start training:

python dopt_so.py -c scen.yaml.

The following table describes the argument.

NOTE

The path can contain uppercase letters, lowercase letters, digits, and underscores (_).

The file name can contain uppercase letters, lowercase letters, digits, underscores (_), and periods (.).

Table 1-1 Command line argument for the retraining mode

Argument	Description	Required or Not
-c, --config	Path of the configuration file for retraining quantization For details about the quantization configuration file, see section 3.2.3 "Configuring the Optimization Strategy."	Yes

3.2.5 Converting the Model

Go to the **tools_dopt/dopt_trans_tools** directory and run the **trans_caffe.sh** script to convert the model:

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

The first parameter indicates the path of the successful model training task, and the second parameter indicates the path of the Caffe environment. The converted model and lightweight configuration file are stored in **$PATH/opt_field/Sub_Task_$INDEX/transedmodel**.

3.3 Training the TensorFlow Model

Perform the following steps:

Prerequisites: The model training datasets and full-precision base model (.ckpt file) have been prepared.

Step 1 Prepare the TensorFlow environment (3.3.1 Preparing the TensorFlow Environment).

Step 2 Configure the model APIs (3.3.2 Configuring Model APIs).

Step 3 Configure the optimization strategy configuration file (3.3.3 Configuring the Optimization Strategy).

Step 4 Train the model (3.3.4 Training the Model).

Step 5 Convert the model (3.3.5 Converting the Model).

----End

3.3.1 Preparing the TensorFlow Environment

3.3.1.1 Preparing the Docker Environment

Prepare the Docker environment by referring to section 3.1.1 "Preparing the Docker Environment."

3.3.1.2 Preparing the Linux Environment

Currently, the lightweight tool supports only tensorflow-gpu 1.12. Run the following command:

pip install tensorflow-gpu==1.12

The lightweight tool depends on Horovod for multi-GPU training and currently supports single-machine, multi-GPU training.

For details about how to install Horovod, visit https://github.com/horovod/horovod#install.

3.3.2 Configuring Model APIs

Configure the model training file based on the following API definitions.

Table 3-3 API description of get_input_placeholder

Function Description	Creates and returns the input placeholder of the model.
API Definition	def get_input_placeholder(self):
Description	None
Return Value	A list, including the image input placeholder and labels input placeholder

Table 3-4 API description of get_next_batch

Function Description	Reads the input of a batch.
API Definition	def get_next_batch(self, is_train):
Description	**is_train:** **True** indicates training, and **False** indicates testing.
Return Value	A list, including the input image and label of the next batch

Table 3-5 API description of forward_fn

Function Description	Reads the input in placeholder format, defines the forward inference process of a model, and returns the output of the forward inference.
API Definition	def forward_fn(self, inputs, is_train):
Description	**inputs:** list returned by get_input_placeholder **is_train:** **True** indicates training, and **False** indicates testing.
Return Value	A tensor is returned, which is the output result of the forward inference.

Table 3-6 API description of loss_op

Function Description	Defines and returns the loss function of the model.
API Definition	def loss_op(self, inputs, outputs):
Description	**inputs:** list returned by get_input_placeholder **outputs:** forward inference result returned by **forward_fn**
Return Value	A tensor, that is, the user-defined loss

Table 3-7 API description of metrics_op

Function Description	Defines the model evaluation method.
API Definition	def metrics_op(self, inputs, outputs):
Description	**inputs**: list returned by **get_input_placeholder** **outputs:** forward inference result returned by **forward_fn**
Return Value	A tensor, which is a user-defined model evaluation method

Table 3-8 API description of config_lr_policy

Function Description	Sets the learning rate of the model.
API Definition	def config_lr_policy(self, global_step):
Description	**global_step:** Global step tensor of TensorFlow
Return Value	A tensor, including the learning rate

The six APIs define the model input structure, input data, model structure, accuracy function, loss function, and learning rate update policy function, respectively.

In the following code sample, the preceding six APIs are defined in base class **UserModelInterface**. The logic of the six APIs is defined in class **UserModel**. You can define models based on the following code snippet.

Figure 3-2 Example of defining an API of class UserModelInterface

```
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
```

The following is a programming routine example for the mobilenetV1 model interface of an image classification application.

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:

The following is an example of reading the ImageNet dataset.

```
    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
```

The **tf.data.Dataset** is used in class **Ilsvrc12Dataset** to read the ImageNet dataset. The data of each batch can be obtained through the iterator.

The get_next_batch interface can be implemented based on **Ilsvrc12Dataset**.

```
    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]
```

In the code, **self.images_train_iter** and **self.images_eval_iter** are iterators returned by class **Ilsvrc12Dataset**. You can create them as follows:

```
    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:

For example:

```
    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
```

In the code, **mobilenet_v1.mobilenet_v1** is a user-defined model.

loss_op interface:

Take the Softmax-based cross entropy as an example. This function is implemented as follows:

```
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:

Take the classification model as an example. In **metrics_op**, the classification precision of the model should be defined as follows:

```
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:

For example:

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

3.3.3 Configuring the Optimization Strategy

Select a proper optimization strategy and configure it in **scen.yaml**. The following provides an instance.

```
# 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
```

An optimization strategy contains multiple parameters, as described in Table 3-1.

Parameter		Type (Python)	Scope	Remarks
strategy				
name		string	Quant_INT8-2[1], Quant_INT8-8	Quant_INT8-2: indicates 8-bit data quantization and 2-bit weight quantization. V310 and V320 support this strategy. Quant_INT8-8: indicates 8-bit data quantization and 8-bit weight quantization. V320 supports this strategy.
framework		string	TensorFlow	Framework type of the trained base model
version		string	"1.12"	Framework version of the trained base model
accuracy_val		float	0–1.0	Target accuracy
skip_layers		string	--	Layers that do not need lightweight implementation by the model
optimizer	type	string	adam / momentum	Optimizer type Supported:

Parameter		Type (Python)	Scope	Remarks
				momentum (default) adam
	momentum	float	0–1.0	Momentum (valid only for the momentum optimizer). The default value is **0.9**.
model		string	--	Path of the .py interface file of the user model
base_model		string	--	Path of the .ckpt base model file
dataset_dir		string	--	Path of the training dataset
train_batch_size		int	--	Batch size per training iteration
train_data_num		int	--	Number of training data samples in the dataset
test_batch_size		int	--	Batch size per test iteration
test_data_num		int	--	Number of test data samples in the dataset
epoch		int	--	Dataset epochs
resource				
name		string	tensorflow_standalone	Name of the resource object for quantization in retraining mode **tensorflow_standalone**: single-machine TensorFlow training resource
gpu_id		string	--	GPU ID. If only one GPU ID (such as 0) is entered, the single-machine single-GPU mode is used for training. If multiple GPU IDs (such as 0, 1, 2, and 3) are

Parameter	Type (Python)	Scope	Remarks
			entered, the single-machine multi-GPU mode is used for training.

3.3.4 Training the Model

Apart from a pre-trained full-precision base model, the following files are also required:

File	Description
.py interface file of the user model	Dataset for training and testing
.ckpt base model file	Base model
Dataset	Dataset for training and testing

Run the following command to start training:

python dopt_so.py -c scen.yaml.
The following table describes the argument.

NOTE

The path can contain uppercase letters, lowercase letters, digits, and underscores (_).

The file name can contain uppercase letters, lowercase letters, digits, underscores (_), and periods (.).

Table 1-2 Command line argument for the retraining mode

Argument	Description	Required or Not
-c, --config	Path of the configuration file for retraining quantization For details about the quantization configuration file, see section 3.3.3 "Configuring the Optimization Strategy."	Yes

3.3.5 Converting the Model

Go to the **tools_dopt/dopt_trans_tools** directory and run the **trans_tensorflow.sh** script to convert the model. The method is as follows:

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

The first parameter indicates the path of the successful model training task, and the second parameter indicates the path of the model output node. The converted model and lightweight configuration file are stored in **$PATH/opt_field/Sub_Task_$INDEX/curmodel/transedmodel**.

4 NAS Training

To conduct NAS training, perform the following steps:

Step 1 Prepare an environment. For details, see section 4.1 "Preparing the Environment."

Step 2 Prepare a dataset. For details, see section 4.2 "Preparing a Dataset."

Step 3 Configure search parameters. For details see section 4.3 "Configuring Search Parameters."

Step 4 Configure user interfaces. For details, see section 4.4 "Custom Interfaces."

Step 5 Search and train network architectures. For details, see section 4.5 "Searching and Training."

----End

4.1 Preparing the Environment

You can choose either the Docker or Linux environment to run the NAS tool. Both environments support the single-machine single-GPU mode and the single-machine multi-GPU mode.

4.1.1 Docker Environment

The Docker platform is required for image creation.

Install **nvidia-docker2** by referring to https://github.com/NVIDIA/nvidia-docker.

Step 1 Decompress the DDK package and run **cd ./$DDK_PATH/** to go to the directory of the **tools** folder.

Step 2 Run the following command to build the HiAIML Docker image:

- If a proxy server is set:

```
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 .
```


Where,

- **-f** specifies the Dockerfile path.
- **-t** specifies the image name and tag in the **name:tag** or **name** format. The default value is **hiai_ddk:v320**.
- **--build-arg** sets the variables for creating the image. **HTTP_PROXY** sets the proxy. If proxy setting is not needed, leave this parameter empty.
- **--no-cache** specifies that no cache is used when creating an image.
- The period (.) specifies the context directory for image building.

  Note: Do not omit the period (.) at the end of the command.

Step 3 Run the following command to start the HiAIML Docker container:

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

Where,

- **-d**: runs the commands in the background in daemon mode.
- **-i**: starts the container in interactive mode. **-i** is often used in conjunction with **-t**.
- **-t**: reallocates a pseudo input terminal to the container. **-t** is often used with **-i**.
- **--restart=always:**: restarts the container automatically upon machine startup.
- **--net=host**: uses the host IP address to interact with external systems when the host NIC is used.
- **--privileged**: grants the multi-GPU NAS with required privilege.
- **--name**: name of the docker container.
- **-v**: maps the host directory to the container.
- **hiai_ddk:v320**: image name.

----End

4.1.2 Linux Environment

Step 1 The dependencies of the HiAIMLEA policy running environments are in the **requirements.txt** file. Install them as follows:

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

NOTE

- $DDK_PATH indicates the path of the decompressed DDK.

- If you only need to run the tool in a single-machine single-GPU environment, skip the following steps. To run the tool in a single-machine multi-GPU environment, continue the following steps.

Step 2 Install and configure Horovod and Open MPI by referring to the following website.

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

Step 3 Install mpi4py.

```
pip3 install mpi4py
```

Step 4 Verifying the Installation

Download a demo from the Horovod website.

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

Run the following command on a server using four accelerator cards:

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

If the training can be performed properly, the Horovod environment is successfully deployed.

----End

4.2 Preparing a Dataset

Prepare a dataset or proxy dataset as required.

For details about the dataset API definition, see Table 4-3.

4.3 Configuring Search Parameters

Set proper search parameters to achieve optimal search results. Configure the **tools/tools_dopt/demo/hiaiml/ea_cls_imagenet/scen.yaml** file. The following is an example:

```
# 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"
```

The search parameter dictionary is as follows.

NOTE

All parameters are case sensitive.

Table 4-1 Search parameter dictionary

Name	Category	Value Range	Description
scenario.strategy			
name	string	HiAIMLEA	Required, policy name.
framework	string	TensorFlow	Required, type of the baseline model training framework.
batch_size	int	--	Required, batch size of the dataset.

Name	Category	Value Range	Description
epochs	int	--	Required, number of dataset polling times.
scenario.strategy.supernet			
input_shape	tuple	--	Optional, shape (C,H,W) or (H,W,C) of the model input. For example, **(224,224,3)**.
data_format	string	channels_first/channels_last	Optional, data format. The values of **input_shape** and **data_format** must match. For example, **channels_last**.
filters	list	--	Optional, cout of each layer of the search skeleton. It is a list corresponding to strides in format [cout, ..., cout]. For example, **[64, 64, 128, 128, 256, 256]**.
strides	list	--	Optional, stride used by each layer of the search skeleton. For example, **[1, 1, 2, 1, 2, 1]**.
feature_choose	list	--	Layers to be fused, optional in the object detection scenarios. Separate them by commas (,). The layers are numbered from 0. For example, **[3, 5]** indicates that layers 3 and 5 (the 4th and 6th layers) are to be fused.
scenario.strategy.constraint			
application_type	string	image_classification/object_detection/semantic_segmentation	Required, application type: classification, detection, or segmentation.
constraint_type	string	size/flops	Required, model constraint type.
constraint_value	int	--	Required, model constraint value, that is, the size of the entire network.
scenario.strategy.optimizer			

Name	Category	Value Range	Description
scenario.strategy.optimizer.weights_optimizer			
type	string	SGD/Momentum/Adam	Optional, optimizer[1]. Defaults to **Adam** for classification and detection; defaults to **Momentum** for segmentation.
betas	list	(0, 1)	Optional, attenuation factor, an optional parameter of optimizer Adam. Defaults to **[0.9, 0.999]**.
learning_rate	float	(0, 1)	Optional, learning rate. If multiple GPUs are used, the value will automatically increase by times based on the GPU count. Defaults to **0.0001** for classification and detection; defaults to **0.00001** for segmentation.
momentum	float	(0, 1)	Optional, momentum, a parameter of optimizer Momentum. Defaults to **0.9**.
scenario.strategy.dataset			
pre_train_dir	string	--	Path of the pre-trained dataset or path of the CKPT generated during pre-training[2], a required parameter in detection and segmentation scenarios. This parameter is required in the segmentation scenario but not required in the classification scenario.
train_dir	string	--	Required, path of the training dataset.
val_dir	string	--	Required, path of the validation dataset.
scenario.strategy.searcher			
generation_num	int	--	Optional, algebra of the evolution algorithm. For

Name	Category	Value Range	Description
			example, **100**.
pop_size	int	--	Optional, number of populations in the evolution algorithm. Defaults to **40**.
scenario.resource			
name	string	tensorflow_standalone	Required, resource object name.
gpu_id	string	--	Required, GPU ID. If there are multiple GPUs, separate them by commas (,). For example, **0,1,2,3,4,5,6,7**.

Notes:

[1] Optimizers SGD, Momentum, and Adam are supported.

- SGD supports the **learning_rate** parameter.
- Momentum supports the **momentum** and **learning_rate** parameters.
- Adam supports the **betas** and **learning_rate** parameters. **betas** is a list. Elements in this list are numbered from 0. For example, the element with index 0 is **beta1**, and the element with index 1 is **beta2**.

[2] If there is no pre-trained CKPT file, **pre_train_dir** indicates the path of the pre-trained dataset. If there is a pre-trained CKPT file, **pre_train_dir** indicates the path of the CKPT file.

4.4 Custom Interfaces

NAS is trained based on the TensorFlow framework. You can configure the training interfaces by referring to the **user_module.py** file in **tools/tools_dopt/demo/hiaiml/ea_cls_imagenet**. The interface definition is as follows.

NOTE

Only **tf.keras** is supported.

UserModule Class

Table 4-2 UserModule class

Class Description	Class **UserModule** defines user-side APIs.
Function Description	Constructor
API Definition	def **__init__**(self, epoch, batch_size):

This class implements the following functions in search training.

Table 4-3 Dataset reading

Function Description	Dataset read function
API Definition	def **build_dataset_search**(self, dataset_dir, is_training, is_shuffle):
Description	**dataset_dir**: Dataset path. **is_training**: **True** for training and **False** for inference. **is_shuffle**: Whether the dataset requires shuffle. Note: When BatchNorm is updated in the evaluation phase, the dataset does not need to be shuffled.
Return Value	• For training: Classification and Segmentation scenario: **iterator**: Iterator of the TensorFlow dataset. **n_data_num**: Length of the dataset in each iteration. Detection scenario: • For inference: Classification and Segmentation scenario: **iterator**: Iterator of the TensorFlow dataset. **n_data_num**: Length of the dataset in each iteration. Detection scenario: **val_generator**: Validation set generator. **val_dataset**: Dataset after JSON file parsing. **val_dataset_size**: Number of validation sets.

Table 4-4 Learning rate update policy

Function Description	Learning rate update policy function.
API Definition	def **lr_scheduler**(self, lr_init, global_step):
Parameter Description	**lr_init**: Initial value of the learning rate. **global_step**: Global step of TensorFlow.
Return Value	Updated learning rate.

Note: Constants are recommended.

Table 4-5 Evaluation function

Function Description	Evaluation function
API Definition	def **metrics_op**(self, inputs, outputs):
Parameter Description	• Classification and Segmentation scenario: ♦ **inputs**: ground truth labels. ♦ **outputs**: Result of forward inference. • Detection scenario: ♦ **inputs**: [valid_dir, model, block_choice] **valid_dir**: Validation set path. **model**: Network model. **block_choice**: Selected network architecture. ♦ **outputs**: [data_generator, proxy_val_image_ids, data_size] **data_generator**: Dataset generator. **proxy_val_image_ids**: Image index of the proxy dataset. **data_size**: Size of the dataset.
Return Value	Evaluation result.

Table 4-6 Loss calculation function

Function Description	Loss calculation function
API Definition	def **loss_op**(self, labels, logits):
Parameter Description	**labels**: ground truth labels. **logits**: Result of forward inference.
Return Value	Loss value, a tensor.

PreNet Class

The input layer of the model does not need to be searched. Therefore, it is defined through a fixed network structure.

Class Description	Class **PreNet**, model input layer.
Function Description	**PreNet** constructor function
API Definition	def **__init__**(self):
Parameter Description	N/A
Return Value	N/A
Function Description	Builds the model input structure.
API Definition	def **call**(self, inputs, training=True):
Description	**inputs**: Data is input. **training**: **True** for training and **False** for inference.
Return value	Input of the search skeleton, a tensor.

PostNet Class

The output layer of the model does not need to be searched. Therefore, it is defined through a fixed network structure.

Class Description	Class **PostNet**, model output layer.
Function Description	**PostNet** constructor function
API Definition	def __**init**__(self):
Description	N/A
Return value	N/A
Function Description	Builds the model output structure.
API Definition	def __**call**__(self, inputs, feature_layer=None, training=True):
Description	**inputs**: output of the search skeleton. **feature_layer**: layers to be searched that need to be fused in **PostNet**. This parameter is involved only in classification and segmentation scenarios, not involved in detection scenario. **training**: **True** for training and **False** for inference.
Return value	Model output, a tensor.

4.5 Searching and Training

This tool is introduced in three aspects: training entry, maintenance and test mode, and search result display.

4.5.1 Training Entry

Execute the **python3 $DDK_PATH/tools/tools_dopt/tensorflow/dopt/dopt_so.py -c scen.yaml** file to enable searching and training. The corresponding demo directory is provided for each scenario. For details, see section 5.5 "TensorFlow HiAIMLEA NAS Demo."

4.5.2 Maintenance and Testing Mode

During the searching and training, you can observe the process information using TensorBoard. The generated files are stored in the **log_*** directory.

Figure 4-1 Model precision loss curve (loss)

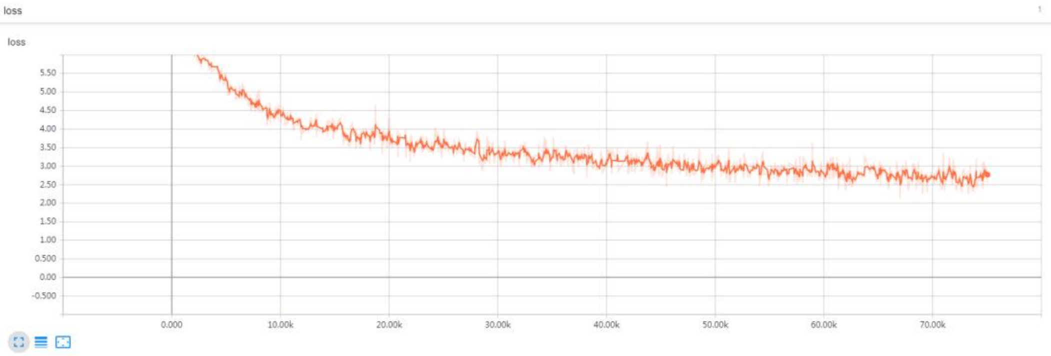


Figure 4-2 Learning rate curve (lr)

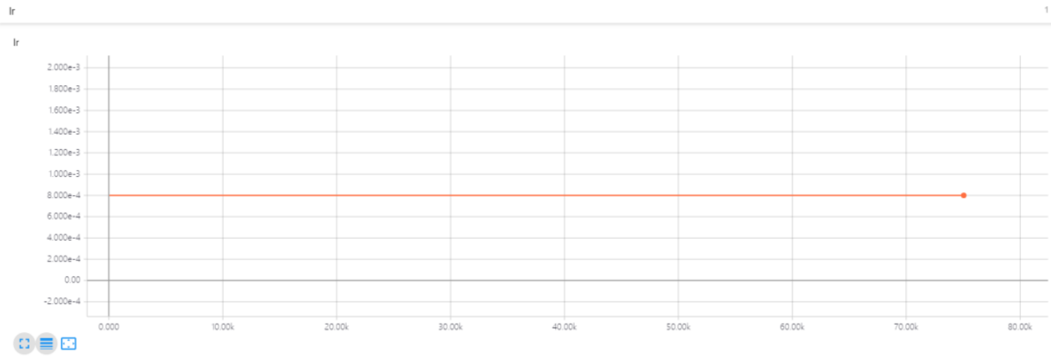
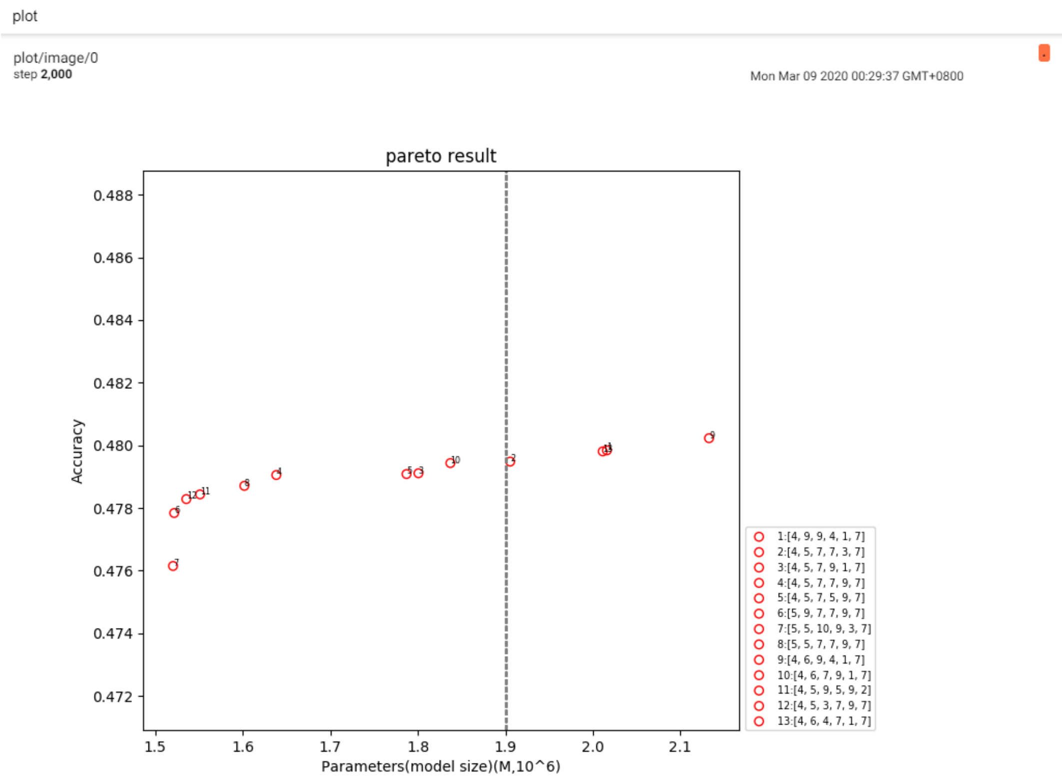


Figure 4-3 Pareto front diagram (lr)



In the Pareto diagram, the horizontal coordinate indicates the model size or computation amount (a constraint type), and the vertical coordinate indicates the precision after architecture search. After training for the search architecture, the precision can be further improved. Search architectures are different in terms of precision, parameter/computation amount, and latency. You can select a proper model based on the actual requirements.

4.5.3 Search Result Display

After the search is complete, the tool automatically saves the model structure in the Pareto front diagram to the **results** directory and generates multiple **model_arch_result_*$NUM*.py** files. The file index **$NUM** is consistent with that in the Pareto diagram. You can select a proper network structure based on the **model_param_size** and **accuracy** parameters and the Pareto diagram in the log, as shown in the following figure.

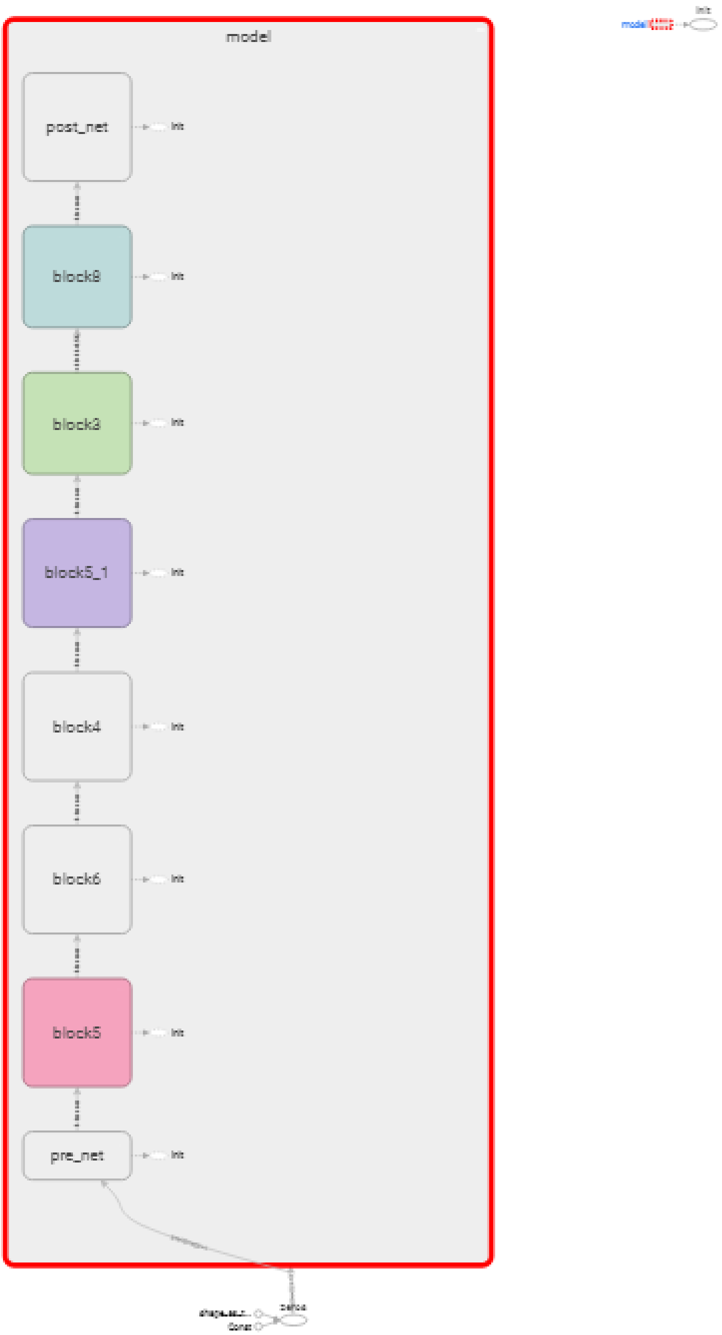
```
# model_param_size:2058496.0
# Accuracy:0.480621

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)
```

Copy the selected model structure file to the upper-level directory of **results**, and run the following command to execute the file.

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

After the execution, a .pb file and a TensorBoard log file of the model are generated in the current directory. You can view the graph of the model using TensorBoard, as shown in the following figure.

Figure 4-4 Graph of the corresponding network architecture



For details about subsequent training, see the **readme.md** file in section 5.5 "TensorFlow HiAIMLEA NAS Demo."

5 Demos

5.1 Caffe Quant_INT8-8 Quantization in Non-Training Mode

5.1.1 Preparing the Environment

Install the Caffe and dependencies. For details, see section 2.2.1 "Preparing the Environment."

5.1.2 Configuring the Resources

- Prepare the quantization model.

  Load the .prototxt file and .caffemodel file of the baseline model to **demo/quant8-8/notrain/caffe_mnist/basemodel/**. This path contains the **mnist.prototxt** and **mnist.caffemodel** files of the MNIST baseline model.
- Prepare the input data for quantization.

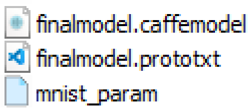
  Load the image or binary calibration dataset to **demo/quant8-8/notrain/caffe_mnist/mnist_test/** by referring to section 2.1.2 "Preparing the Calibration Set." This path contains a preset image calibration dataset.

5.1.3 Quantizing the Model

Change the value of **--caffe_dir** in **demo/quant8-8/notrain/caffe_mnist/run_release.sh** to your Caffe path and execute **run_release.sh**.

The .prototxt, .caffemodel, and quantization configuration files are stored in **demo/quant8-8/notrain/caffe_mnist/curmodel**.

Figure 5-1 Files generated after the demo running



5.2 TensorFlow Quant_INT8-8 Quantization in Non-Training Mode

5.2.1 Preparing the Environment

Install TensorFlow and dependencies. For details, see section 2.3.1 "Preparing the Environment."

5.2.2 Configuring the Model

- Prepare the quantization model.

  Load the .pb file of the baseline model to **demo/quant8-8/notrain/tensorflow_mnist/basemodel/**. This path contains the MNIST baseline model file **mnist.pb**.

- Prepare the input data for quantization.

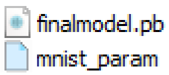
  Load the image or binary calibration dataset to **demo/quant8-8/notrain/caffe_mnist/mnist_test/** by referring to section 2.2.2 "Quantizing the Model." This path contains a preset image calibration dataset.

5.2.3 Quantizing the Model

Run the **run_release.sh** script in **demo/quant8-8/notrain/tensorflow_mnist/**.

The quantized PB model and quantization configuration file are stored in **demo/quant8-8/notrain/tensorflow_mnist/curmodel**.

Figure 5-2 Files generated after the demo running



5.3 Caffe Retraining Quantization

5.3.1 Preparing the Dataset

Step 1 Download the MNIST dataset from http://yann.lecun.com/exdb/mnist/.

Step 2 Convert the downloaded MNIST dataset to the LMDB format using the **https://github.com/BVLC/caffe/blob/master/examples/mnist/create_mnist.sh** script.

Step 3 Change the dataset path of the .prototxt file in **tools_dopt/demo/quant8-8/retrain/caffe_mnist_single_gpu/basemodel**.

----End

5.3.2 Preparing the Environment

Install the Caffe and dependencies. For details, see section 3.2.1 "Installing the Compilation Environment."

5.3.3 Configuring the Model

Step 1 Change the value of **framework_path** in the **res_caffe_standalone.yaml** file in the **config** directory. For details, see section 3.2.2 "Configuring Resource Files.".

Step 2 Prepare the .caffemodel file in the demo and load it to **demo/quant8-8/retrain/caffe_mnist_single_gpu/basemodel/**.

----End

5.3.4 Configuring the Optimization Strategy

Configure the **scen.yaml.tmp** file in the **demo/quant8-8/retrain/caffe_mnist_single_gpu/** directory. In this demo, parameters have been configured. You only need to change the path according to the local environment.

For details, see 3.2.3 Configuring the Optimization Strategy.

5.3.5 Training the Model

Run the **run_release.sh** script in **demo/quant8-8/retrain/caffe_mnist_single_gpu/run_release.sh** to perform retraining quantization.

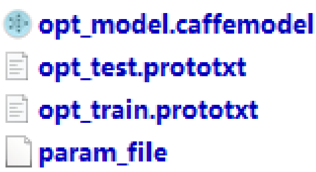
5.3.6 Converting the Model

Check the task generated in **demo/quant8-8/retrain/caffe_mnist_single_gpu/opt_field**.

Modify and run the **dopt_trans_tools/run_caffe_trans_demo.sh** script.

A new model is generated in the following path: **demo/quant8-8/retrain/caffe_mnist_single_gpu/opt_field/Sub_Task_2/transedmodel**

Figure 5-3 Files generated after the Caffe model conversion



5.4 TensorFlow Retraining Quantization

5.4.1 Preparing the Dataset

Step 1 Download the MNIST dataset from http://yann.lecun.com/exdb/mnist/.

Step 2 Change the value of **dataset_dir** in **tools_dopt/demo/quant8-8/retrain/tensorflow_mnist_single_gpu/scen.yaml** to the actual dataset path.

----End

5.4.2 Preparing the Environment

Install tensorflow-gpu 1.12 and its necessary dependencies. For details, see section 3.3.1 "Preparing the TensorFlow Environment."

5.4.3 Configuring the Model

Step 1 Perform API configuration and write the model interface file. For details, see section 3.3.2 "Configuring Model APIs." The user model interface definition file **mnist_model.py** has been provided in the **tools_dopt/demo/quant8-8/retrain/tensorflow_mnist_single_gpu/basemodel** directory.

Step 2 Provide the pre-trained TensorFlow model files, including **checkpoint**, **model.ckpt.data-00000-of-00001**, **model.ckpt.index**, and **model.ckpt.meta**. This pre-trained model has been provided in the **basemodel** directory.

----End

5.4.4 Configuring the Optimization Strategy

Configure the **scen.yaml.tmp** file in **demo/quant8-8/retrain/tensorflow_mnist_single_gpu/**. For details, see 3.3.3 "Configuring the Optimization Strategy." In this demo, parameters have been configured. You only need to change the path based on the local environment.

5.4.5 Training the Model

Run the **demo/quant8-8/retrain/tensorflow_mnist_single_gpu/run_release.sh** script.

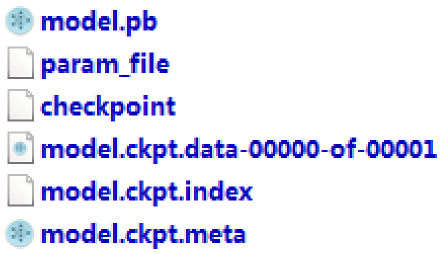
5.4.6 Converting the Model

Check the task generated in **demo/quant8-8/retrain/tensorflow_mnist_single_gpu/opt_field**.

Modify and run the **dopt_trans_tools/run_tensorflow_trans_demo.sh** file.

A new model is generated in the following path: **demo/quant8-8/tensorflow_mnist_single_gpu/opt_field/Sub_Task_1/curmodel/transedmodel**

Figure 5-4 Files generated after the TensorFlow model conversion

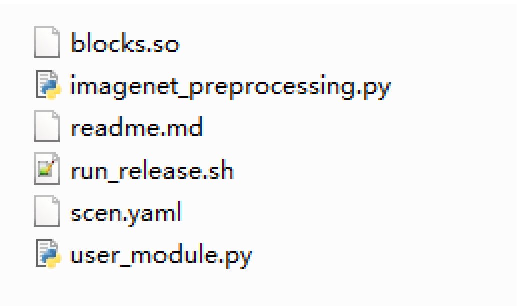


5.5 TensorFlow HiAIMLEA NAS Demo

5.5.1 HiAIML Classification Network

The classification network demo is stored in **tools/tools_dopt/demo/hiaiml/ea_cls_imagenet** and contains the following six files.

Figure 5-5 HiAIML classification network demo



- **blocks.so**: a search space file.
- **imagenet_preprocessing.py**: open-source code of TensorFlow models for processing ImageNet data.
- **readme.md**: guidance for subsequent training.
- **run_release.sh**: used to start the search.
- **scen.yaml**: a configuration item.
- **user_module.py**: customized APIs of the tool.

The procedure is as follows:

Step 1 Prepare a dataset and change the dataset path in **scen.yaml**.

Step 2 For details, see section "4.1 Preparing the Environment."

Step 3 Configure the **scen.yaml** file in the **demo** directory. For details, see section 4.3 "Configuring Search Parameters." This file provides recommended parameters. You can modify them as required.

Step 4 Modify the **user_module.py** file in the **demo** directory. For details about the model API definitions, see section 4.4 "Custom Interfaces." This file provides recommended configurations. You can modify the configurations as required.

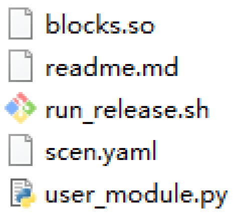
Step 5 Execute the **run_release.sh** script. Multiple **model_arch_result_*.py** files will be generated in **results**. You can select a proper network architecture for training based on the information provided in **log_classification**. For details about subsequent training, see **readme.md**.

----End

5.5.2 HiAIML Detection Network

The detection network demo is located in **tools/tools_dopt/demo/hiaiml/ea_det_coco** and contains the following five files.

Figure 5-6 HiAIML detection network demo



- **blocks.so**: a search space file.
- **readme.md**: guidance for subsequent training.
- **run_release.sh**: used to start the search.
- **scen.yaml**: a configuration item.
- **user_module.py**: customized APIs of the tool.

The procedure is as follows:

Step 1 Prepare datasets, including the pre-training dataset ImageNet and training dataset COCO. If a pre-trained CKPT file is available, you do not need to prepare the ImageNet dataset. Change the dataset paths in the **scen.yaml** file by referring to section 4.3 "Configuring Search Parameters."

Step 2 For details, see section "4.1 Preparing the Environment."

Step 3 Load the dependent open-source code.

- Go to the **demo** directory of the detection network.

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

- Download the open-source code:

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

- Go to the open-source code directory.

```
cd ssd_keras
```

- Switch to the specified version:

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

- Modify related open-source files by referring to Step 1, 2, and 3 in **readme.md**.
- Modify **user_module.py** by referring to Step 4 in **readme.md**.

Step 4 Configure the **scen.yaml** file in the **demo** directory. For details, see section 4.3 "Configuring Search Parameters." This file provides recommended parameters. You can modify them as required.

Step 5 Modify the **user_module.py** file in the **demo** directory. For details about the model API definitions, see section 4.4 "Custom Interfaces." This file provides recommended configurations. You can modify the configurations as required.

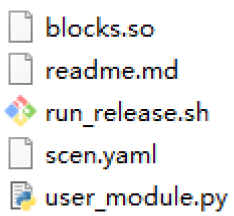
Step 6 Execute the **run_release.sh** script. Multiple **model_arch_result_*.py** files will be generated in **results**. You can select a proper network architecture for training based on the information provided in log_detection. For details about subsequent training, see **readme.md**.

----End

5.5.3 HiAIML Segmentation Network

The segmentation network demo is stored in **tools/tools_dopt/demo/hiaiml/ea_seg_voc** and contains the following five files.

Figure 5-7 HiAIML segmentation network demo



- **blocks.so**: a search space file.
- **readme.md**: guidance for subsequent training.

- **run_release.sh**: used to start the search.
- **scen.yaml**: a configuration item.
- **user_module.py**: customized APIs of the tool.

The procedure is as follows:

Step 1 Prepare datasets, including the pre-training dataset ImageNet and training dataset VOC. If a pre-trained CKPT file is available, you do not need to prepare the ImageNet dataset. Change the dataset paths in the **scen.yaml** file by referring to section 4.3 "Configuring Search Parameters."

Step 2 For details, see section "4.1 Preparing the Environment."

Step 3 Load the dependent open-source code.

- Go to the **demo** directory of the segmentation network.

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

- Download the open-source code:

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

- Go to the open-source code directory.

```
cd models
```

- Switch to the specified version:

```
git checkout v1.13.0
```

- Return to the **demo** directory of the segmentation network.

```
cd ..
```

- Modify the open-source implementation.

  The current path is **ea_seg_voc**.
  Modify the **train_utils.py** file in **.\models\research\deeplab\utils\train_utils.py**.

  Change **slim.one_hot_encoding** in line 72 to **tf.one_hot**.

  Add **return** before **tf.losses.softmax_cross_entropy** in line 74.

- Set the default path of **PYTHONPATH**.

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

  Note: Each time you open the terminal, you need to run the command again. Alternatively, add the command to the **~/.bashrc** file and run the **source ~/.bashrc** command.

Step 4 Configure the **scen.yaml** file in the **demo** directory. For details, see section 4.3 "Configuring Search Parameters." This file provides recommended parameters. You can modify them as required.

Step 5 Modify the **user_module.py** file in the **demo** directory. For details about the model API definitions, see section 4.4 "Custom Interfaces." This file provides recommended configurations. You can modify the configurations as required.

Step 6 Execute the **run_release.sh** script. Multiple **model_arch_result_*.py** files will be generated in **results**. You can select a proper network architecture for training based on the information provided in log_segmentation. For details about subsequent training, see **readme.md**.

----End

6 Appendix

6.1 FAQs

Q1: How do I prepare data and modify the configuration file if a model has multiple inputs?

A: If the model defines multiple inputs, you need to prepare an image or binary calibration dataset for each input node. If the inputs are specific to nodes, the binary format is recommended to prevent unexpected behavior caused by different image reading sequences. When modifying the quantization configuration file, the number of preprocessing parameters to be defined must be the same as that of input nodes, and the sequence of preprocessing parameters must be the same as the **input_shape** sequence specified when running the tool.

Q2: What do I do if the message "Unsupported image format! Unsupported image: xxx" is displayed?

A: This is because the image calibration dataset contains an image of unsupported format. You only need to delete the image.

Q3: How do I select **Sub_Task** for model conversion after model retraining and optimization?

A: Select the last successful **Sub_Task** to achieve the optimal model performance that meets the precision requirement.

Q4: Does the **caffe-mod** folder generated after model retraining and optimization affect the training and inference of the native model?

A: No. This **caffe-mod** folder adds only a necessary model optimization layer and does not change the layer definition of the native Caffe model.

Q5: Is the Horovod environment required if only a single GPU is used for TensorFlow model retraining and optimization?

A: No, you do not need to install the Horovod environment.

6.2 Model Benefits

Take ResNet-18 as an example. Using the lightweight tool for Quant_INT8-2 quantization can yield the following benefits.

Table 6-1 Quantitative benefits of Quant_INT8-2

Framework	Dataset	Model	Initial Precision	Precision After Retraining	Unquantized DaVinci— Model Size (MB)	Quant_INT8-2 DaVinci— Model Size (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

Take ResNet-18 as an example. Using the lightweight tool for Quant_INT8-8 quantization can yield the following benefits.

Table 6-2 Quantitative benefits of Quant_INT8-8

Framework	Dataset	Model	Initial Precision	Precision After Retraining	Unquantized DaVinci— Model Size (MB)	Quant_INT8-8 DaVinci— Model Size (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

Take ResNet-18 as an example. Using the lightweight tool (NAS mode) can yield the following benefits.

Table 6-3 Benefits of classification networks

Network	Model	Parameter Count (M)	Precision
ResNet-18 (ImageNet dataset)	ResNet-18	11.69	70.32%
	HiAIMLEA	10.93	72.13%[1]

Table 6-4 Benefits of detection networks

Network	Model	Parameter Count (G)	mAP@[.5, .95] [2]
SSD (backbone: ResNet-18) COCO dataset	ResNet-18	11.6	17.8[3]
	HiAIMLEA	9.25	18.4[3]

Table 6-5 Benefits of segmentation networks

Network	Model	Parameter Count (G)	mIOU[4]
Deeplab (backbone: ResNet-18) VOC dataset	ResNet-18	40.9	54.1[5]
	HiAIMLEA	28.3	56.1[5]

Notes:

[1] The precision is obtained by using the Tensorpack retraining model.

[2] The indicator is obtained by averaging the APs within the IoU range from 0.5 to 0.95 with a step of 0.05.

[3] The precision is obtained by tests using the COCO val2017 dataset.

[4] This indicator indicates the Mean Intersection over Union (MIoU). The IoU of each category is calculated first, and then the average value is calculated.

[5] The precision is obtained using the VOC val2012 dataset.

