HAILO

Empowering Intelligence

Hailo Dataflow Compiler User Guide

Release 3.20.0

22 September 2022

Table of Contents

Disclaimer and Proprietary Information Notice

Copyright

This version of the document supersedes all previous versions.

General Notice

Hailo, to the fullest extent permitted by law, provides this document "as-is" and disclaims all warranties, either express or implied, statutory or otherwise, including but not limited to the implied warranties of merchantability, non-infringement of third parties' rights, and fitness for particular purpose.

Although Hailo used reasonable efforts to ensure the accuracy of the content of this document, it is possible that this document may contain technical inaccuracies or other errors. Hailo assumes no liability for any error in this document, and for damages, whether direct, indirect, incidental, consequential or otherwise, that may result from such error, including, but not limited to loss of data or profits.

The content in this document is subject to change without prior notice and Hailo reserves the right to make changes to content of this document without providing a notification to its users.

Part I

User Guide

1. Hailo Dataflow Compiler Overview

1.1. Included in this package

This software package includes the following parts:

- Dataflow Compiler Python packages
- HailoRT library to run inference from C/C++ and Python programs
- Hailo's PCIe driver
- Additional files such as an installation script and a configuration file

1.2. Introduction

The Dataflow Compiler API is used for compiling users' models to Hailo binaries. The input of the Dataflow Compiler is a trained Deep Learning model. The output is a binary file which is loaded to the Hailo device.

The HailoRT API is used for deploying the built model on the target device. This library is used by the runtime applications.

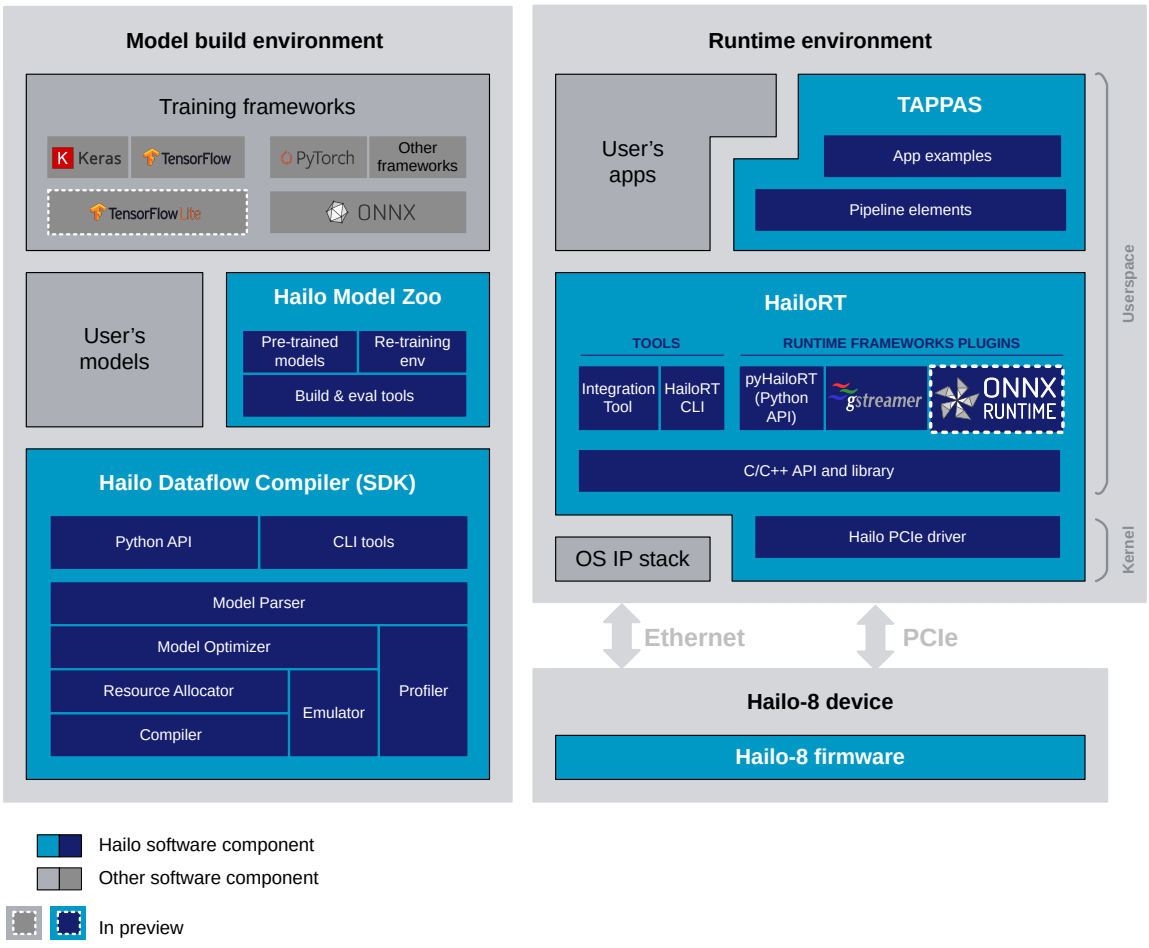

Figure 1. Detailed block diagram of Hailo software packages

1.3. Model build process

The Hailo Dataflow Compiler toolchain enables users to generate a Hailo executable binary file (HEF) based on input from a Tensorflow checkpoint, a Tensorflow frozen graph file or an ONNX file. The build process consists of several steps including translation of the original model to a Hailo model, model parameters optimization, and compilation.

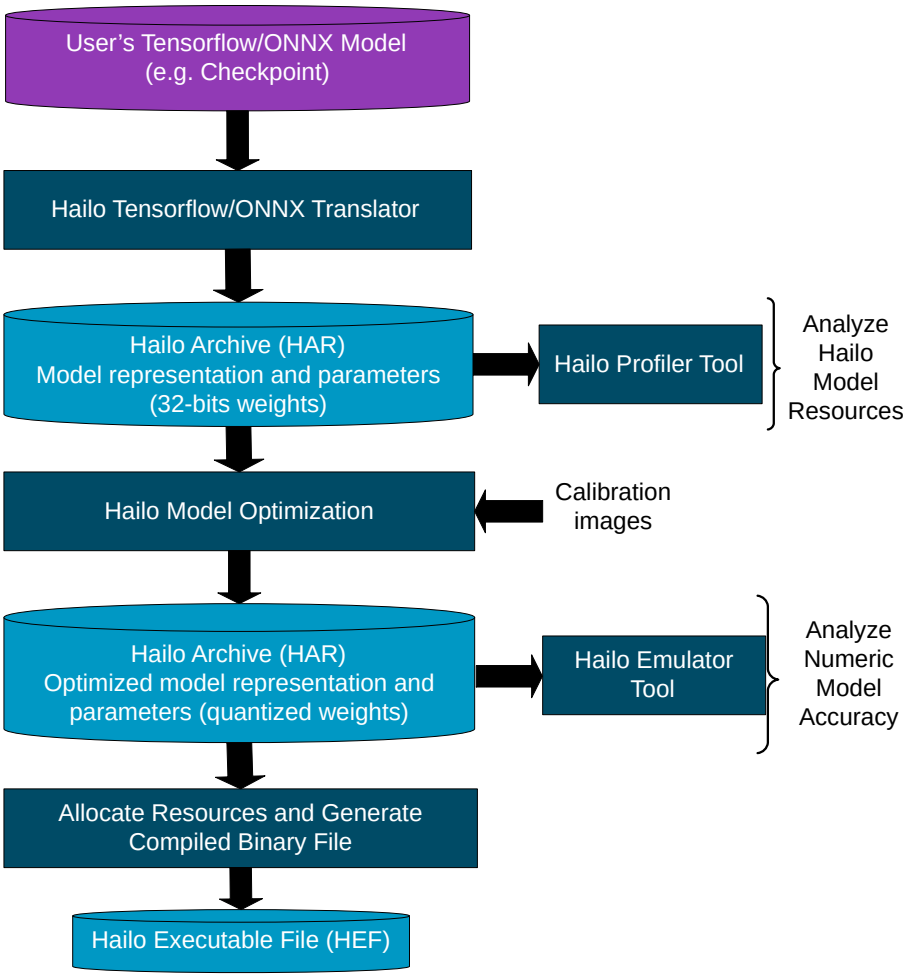

Figure 2. Model build process, starting in a Tensorflow or ONNX model and ending with a Hailo binary (HEF)

1.3.1. Tensorflow and ONNX translation

After the user has prepared the model in its original format, it can be converted into Hailo- compatible representation files. The translation API receives the user's model and generates an internal Hailo representation format (HAR compressed file, which includes HN and NPZ files). The HN model is a textual JSON output file. The weights are also returned as a NumPy NPZ file.

1.3.2. Profiler

The Profiler tool uses the HAR file and profiles the expected performance of the model on hardware. This includes the number of required devices, hardware resources utilization, and throughput (in frames per second). Breakdown of the profiling figures for each of the model's layers is also provided.

1.3.3. Emulator

The Dataflow Compiler Emulator allows users to run inference on their model without actual hardware. The Emulator supports two main modes: *native* mode and *numeric* mode. The native mode runs the original model with float32 parameters, and the numeric mode provides results that are bit-exact to the hardware. The native mode can be used to validate the Tensorflow/ONNX translation process and for calibration (see next section), while the numeric mode can be used to analyze the optimized model's accuracy.

1.3.4. Model Optimization

After the user generates the HAR representation, the next step is to convert the parameters from float32 to int8. To convert the parameters, the user should run the model emulation in native mode on a small set of images and collect activation statistics. Based on these statistics, the calibration module will generate a new network configuration for the 8-bit representation. This includes int8 weights and biases, scaling configuration, and HW configuration.

1.3.5. Compiling the model into a binary image

Now the model can be compiled into a HW compatible binary format with the extension HEF. The Dataflow Compiler Tool allocates hardware resources to reach the highest possible fps within reasonable allocation difficulty. Then the microcode is compiled and the HEF is generated. This whole step is performed internally, so from the user's perspective the compilation is done by calling a single API.

1.4. Deployment process

After the model is compiled, it can be used to run inference on the target device. The HailoRT library provides access to the device in order to load and run the model. This library is accessible from both C/C++ and Python APIs. It also includes command line tools.

In case the device is connected to the host through PCIe, the HailoRT library uses Hailo's PCIe driver to communicate with the device. If Ethernet is used, the library uses the Linux IP stack to communicate.

The HailoRT library can be installed on the same machine as the Dataflow Compiler or on a separate machine. A Yocto layer is provided to allow easy integration of HailoRT to embedded environments.

2. Changelog

Dataflow Compiler v3.20.0 (October 2022)

Model Optimization

- FPS is improved for large models by Quantization to 4-bit for 20% of the model weights is *enabled by default* on large networks to improve FPS

Kernels and Activations

- Added on-chip support for RGBX->RGB conversion using *input conversion command*
- Added support for ONNX operator InstanceNormalization
- Added support for L2 Normalization layer on TensorFlow

Compiler

- Optimized the performance of compiled models

Parser

- Added a recommendation to use onnxsimplifier when parsing fails
- Added a recommendation to use TFLite parser if TF2 parsing fails (see *conversion guide*, on 4.2.5)
- `TensorFlow parser` detects model type automatically

High Level

- **Refactor logger**
  - Cleaned info and warning messages
  - Log files are duplicated into activated_virtualenv/etc/hailo/
  - Log files could be disabled by an *environment variable*
- *HTML Profiler* report includes model optimization information: compression and optimization levels, model modifications, weight and activation value ranges
- Dataflow Compiler is tested on Windows 10 with WSL2 running Ubuntu 20.04

API

- Compiler automatically separates different connected components to multiple *network groups*
  - Mostly relevant for `joined networks` with join_action=JoinAction.NONE
  - HailoRT API should be used to activate/deactivate each network group. It is recommended to use the Scheduler API because it automatically switches between network groups (and .hef files)
  - For more information refer to *network_group model script command*
- Updated *platform_param* model script command to optimize compilation for low PCIe bandwidth hosts
- *Model script command for adding NMS on chip* is simplified (preview)
- Deprecation warning for the legacy –fps argument, use *performance_param* model script command instead
- Removed the already-deprecated APIs
  - integrated_preprocess and ckpt_path arguments from ClientRunner methods
  - Removed har-modifier CLI, and the following related methods: add_nms_postprocess_from_hn, add_nms_postprocess_from_har, dead_channels_removal_from_har, transpose_hn_height_width_from_hn, transpose_hn_height_width_from_har, add_yuv_to_rgb_layers, add_yuv_to_rgb_layers_from_har, add_resize_input_layers, add_resize_input_layers_from_har
  - *npz-csv* (use *params-csv* instead)

- As the parser detects Tensorflow1/2/TFLite automatically, the API for specifying the framework is deprecated
- The argument *onnx_path* of *ClientRunner.translate_onnx_model* was renamed to *model*, and also supports 'bytes' format
- *ClientRunner.load_model_script* can receive either a file object or a string

---

Note: Ubuntu 18.04 will be deprecated in Hailo Dataflow Compiler future version

---

Note: Python 3.6 will be deprecated in Hailo Dataflow Compiler future version

---

Dataflow Compiler v3.19.0 (August 2022)

Parser

- TFLite support to release

Kernels and Activations

- On-chip BGR->RGB color conversion support using *Model Modification Commands*
- Log and Hard Sigmoid *activations* (preview)

Compiler

- Model load time optimizations

High Level and API

- Renamed ClientRunner.profile_hn_model API to `profile()` (to align with the CLI tool)
- Model modification commands support *activation functions replacement* (preview)
- When adding NMS using Model Modification Commands, JSON path should be relative to the model script

Dataflow Compiler v3.18.1 (July 2022)

General

- Fixed a bug that prevented the Dataflow Compiler from running when HailoRT is not installed

Dataflow Compiler v3.18.0 (June 2022)

Parser

- Parser now supports `TFLite models` (preview)
- ONNX models only: Parser now suggests start/end nodes when translation fails (preview)

Model Optimization

- **Introducing *Optimization levels***
  - Using a number between 0 to 3 (default=1), control the complexity of the optimization algorithm
  - The default (1) requires 1024 images for the calibration set as well as GPU, unlike the previous default that used 128 images
- **Introducing *Compression levels***
  - Using a number between 0 to 5 (default=0), control the compression of the model

  - Higher compression corresponds to better performance, but require stronger Optimization algorithms to maintain accuracy
  - Higher compression is achieved with converting more layers to 4-bit
- Optimization and Compression levels are set using a model script command

Kernels and Activations

- New filter sizes for *2D Convolution* kernels (preview)
- New filter sizes for *DepthWise Conv* kernels (preview)
- New filter sizes for *Average Pool* kernels (preview)
- Broader *Resize N.N support* (preview)
- Broader *Resize Bilinear support* (preview)
- Improved degradation and performance of SiLU, Mish, Swish activations

Compiler

- Improved RAM usage during compilation
- Optimized the loading time of compiled models on hardware; Helps pipelines with frequent model switching
- Compilation performs better utilization of the device
- ONNX models only: Option to *export an ONNX model* that contains the compiled model (between the start and end nodes supplied to the parser), as well as the original model's pre & post processing (before the start nodes and after the end nodes). The model could be run using ONNX Runtime (preview)

Documentation and Tutorials

- Suite documentation has a separate user guide
- Updated the parsing tutorial to demonstrate how to convert TF/TF2 models to TFLite
- Added explanation on *how to export PyTorch models to ONNX*
- Less warning messages: Warning are now displayed only if user interaction is suggested

High Level and API

- Profiler accepts runtime_data.json file to create HTML *runtime graph*
- Profiler on pre-placement mode better aligned with compiler
- hailo CLI *–quantization-script* argument is deprecated, please use *–model-script* argument instead
- **Removed the following deprecated interfaces. Use *Model Modification Commands* instead:**
  - integrated_preprocess argument of ClientRunner.translate_tf_model
  - transpose_hn_height_width[_from_hn / _from_har] of hn_modifications class
  - add_yuv_to_rgb_layers[_from_har] of hn_modifications class
  - add_resize_input_layers of hn_modifications class
  - har-modifier CLI tool (`hailo har-modifier`)
- Added the *input_conversion() model script command* as the main API to add input conversion, such as yuv_to_rgb and yuv2_to_yuv

---

Note: Ubuntu 18.04 will be deprecated in Hailo Dataflow Compiler future version

---

Note: Python 3.6 will be deprecated in Hailo Dataflow Compiler future version

---

Dataflow Compiler v3.17.0 (May 2022)

Kernels and Activations

- Supporting *TF operator tf.norm*
- Supporting *Global Average Pooling* with unlimited number of output features (preview)

Compiler

- Bus fixes in profiler report
- Bug fixes and enhancements

High Level and API

- All compilation errors are now being collected and printed together
- Model script commands fixes (NMS, normalization)
- *hailo tutorial* tool now supports user specified IP and port, to allow connection from a remote client
- *quantization-script* argument of 'hailo optimize' has changed to *model-script* to align with other commands
- Removed *layer name truncated* message from output

Dataflow Compiler v3.16.0 (April 2022)

Kernels and Activations

- Mish, Hard-swish, GELU, PReLU, Sqrt *activation support* (preview)
- Expand operator, as broadcast before element-wise operations
- ReduceL2 layer, after rank4 tensors such as Conv
- Conv6x6 stride 2 support

Compiler

- CenterNet BBox decoder + score threshold on-chip support, *more info here*
- YOLOv5 BBox decoder + score threshold on-chip support, *more info here*

High Level and API

- For supported post processing types (such as NMS), adding post processing *using a Model Script file*
- TF version updated to 2.5.2 (CUDA 11.2, Cudnn 8.1)
- *hailo join* CLI for joining networks before compilation

Dataflow Compiler v3.15.0 (February 2022)

High-Level and Documentation

- Installation method now using pip, see: *Installing / Upgrading Hailo Dataflow Compiler*
- Updated the version compatibility table with Hailo Model Zoo and TAPPAS
- New documentation for adding NMS post-processing, see: *Non Maximum Suppression (NMS)*
- Hailo visualizer tool now shows the activation types for activation nodes
- HTML profiler report fixes and new features

Core

- Added support for SiLU and Swish activation types
- Added support for Square operator

- Elementwise Add, Sub, Mul, Div enhanced support for different input types, see: *Elementwise Multiplication and Division*
- 'VALID' padding scheme for depthwise 3x3, 3x5, 5x3, 5x5
- Added depthwise support for two input data tensors (for siamese networks)

Compiler

- Added support for NMS with big models
- Added warning when forcing multi context allocation together with using FPS target (FPS is ignored)
- Optimized the compilation to reach higher utilization of the device

Dataflow Compiler API

- Normalization, Transpose, YUV2RGB, Resize operations are now performed using a model script, see: *Model modification commands*
- Deprecated .hn format support, Hailo Archive (.har) format should be used. Hailo CLI tool commands now output .har by default

Parser

- Parser now shows the original layer name when failing with UnsupportedOperation

Dataflow Compiler v3.14.0 (January 2022)

Core

- Added support for additional Average Pooling cases. See updated *Average Pooling table*
- Added support for *element-wise subtraction* (preview)
- Added support for dilation 16x16 to Conv 3x3, stride 1x1 (preview)
- Depthwise Convolution kernel optimization

Compiler

- Setting fps parameter to None now optimizes the compiled model to maximum FPS See `compile()` for further details.
- In-chip vision pipeline commands (YUV2RGB, reshape, resize) can now be added via model script files (alls)
- **Released features (from preview)**
  - Join wide support - `join()` API that unifies two models to be compiled together
  - Layer merging - resource optimization by merging two layers to use the same controller

Model Optimization

- Added support for 16 bit quantization, applied to the model's last layer. See *precision mode* section for further details

Dataflow Compiler API

- `in-chip NMS` now supports CenterNet (preview)

Parser

- Added support for elementwise subtraction.
- Added support for PyTorch's PixelShuffle as a DepthToSpace in CRD mode (ONNX). For further details take a look at the *Using the ONNX Parser*, *Supported PyTorch APIs*, and *Depth to Space* sections.
- Added support for parsing Tensorflow models in NCHW format.

Dataflow Compiler v3.13.1 (December 2021)

User interface

- Fixed a bug in the *Profiler*.

Dataflow Compiler v3.13.0 (December 2021)

User interface

- The *Profiler* report has a new user interface that contains the model's graph and many other improvements

Parser

- Added HAR Modifier to Hailo CLI API (supported operations: resize input, yuv2rgb, tranpose height/width)
- Added support for Softmax over the features per pixel

Dataflow Compiler v3.12.0 (October 2021)

Note: All past references to this package as "SDK" were modified to "Dataflow Compiler".

Core

- Global *Average Pooling* performance improvement (implemented using *de-fusing*)
- Added "Depthwise Conv 1x1 and Add" support, where depthwise convolution and elementwise addition are fused together
- Added transpose features and width optimization to Depthwise Conv 1x1

Compiler

- *Context switch* enabled by default for large networks (release)

Parser

- The layer names now contain the scope names, e.g., scope_name/conv1
- Added optimization for fusing elementwise add and Depthwise Conv 1x1 (also Normalization and Batchnorm)
- Added support for integrated preprocess for multiple inputs networks (e.g. different normalization for each input, API change)

Examples and packaging

- Added support for Ubuntu 20.04 64 bit
- Added support for Python 3.7/3.8
- Added support for Nvidia Ampere architecture (for model optimization's calibration / Fine Tune)
- HailoRT library installation was separated from *Dataflow Compiler installation* (`insatll.sh`)

Dataflow Compiler v3.11.2 (October 2021)

Parser

- Bug fix in Depthwise convolution support in the *TF Parser*

Dataflow Compiler v3.11.1 (September 2021)

Core

- New *Conv* parameters support: 3x4, 3x6, 3x8, and others

Parser

- Added parsing support for Keras API layers.GlobalMaxPooling2D in the *TF Parser*

Dataflow Compiler v3.11.0 (September 2021)

Core

- *Maxpool 3x3/2* support with VALID padding
- *Softplus activation* support
- *External padding* layer is inserted automatically in additional cases to match the padding requested by the user's model
- The *Reduce Sum* layer is supported in additional cases
- Maxpool layers performance improvement
- *Bilinear Resize and NN Resize* performance improvement (implemented using *de-fusing*)

Models allocation

- *Context switch* support for allocating and compiling multiple models together. Use the `join()` API to merge the models before compiling them (preview)

Parser

- Elementwise addition implementation doesn't use a "dummy convolution" layer by default
- Improved support for addition and multiplication by scalar in the *TF Parser*
- The newly supported layers are supported also in the TF and ONNX parsers

Dataflow Compiler API

- The HAR command line tool supports a verbose info mode
- Model input tensor shapes (input resolutions) can be modified after parsing using the `set_input_tensors_shapes()` API

Model optimization

- New *model scripts (ALLS)* commands related to model optimization and quantization are supported
- *Tiled Squeeze and Excite (TSE)* algorithm support
- The *Equalization algorithm* supports new features and options
- The *IBC algorithm* is supported together with multiple *quantization groups*

Note: The *HEF parameters* `should_use_sequencer` and `params_load_time_compression` are now enabled by default for all models. When enabled, these flags allow faster load of models to the device over a PCIe interface, but prevent Ethernet support. The Ethernet interface is still supported, but a model script (ALLS) that disables these features is now required.

Note: The functions `run_quantization()` and `run_quantization_from_np()`, which have already been deprecated, are not supported from this version.

Dataflow Compiler v3.10.1 (August 2021)

- Upgraded HailoRT and firmware. The *tutorials* are updated to use the newest HailoRT API

Dataflow Compiler v3.10.0 (July 2021)

Core

- *Depthwise conv 2x2/2* support
- *Average pooling 2x2/2* support
- *Nearest neighbor resize* supports any column scale which is an integer power of two
- *"Conv and Add"* and *Group Conv* kernels optimization

Models allocation

- *Context switch* related bug fixes and optimizations (preview)

Parser

- Dataflow Compiler environment was upgraded to Tensorflow v2.4.1, including all components. Translating TF 1.x models is still supported
- In-chip vision pipeline capabilities: Bilinear resize and YUV to RGB conversion can be added to the model after parsing it. See `add_yuv_to_rgb_layers()` and `add_resize_input_layers()`
- The newly supported layers are supported also in the TF and ONNX parsers

Note: Tensorflow *eager execution* is currently not supported together with the Hailo Dataflow Compiler. For example, when combining custom Tensorflow pre- and post-processing nodes together with Hailo's emulator or HW nodes, eager execution has to be turned off.

Note: *(for GPU users)* The requirements of several Nvidia packages have changed due to the Tensorflow upgrade. See the *installation* page for details.

Dataflow Compiler v3.9.1 (July 2021)

- Fixed an inference error that occurred when the device was connected to the Ethernet interface

Dataflow Compiler v3.9.0 (June 2021)

Core

- *Conv* 2x2/2x1 kernel support
- *Average pooling* 3x4/3x4 and other kernel sizes support
- *Deconv* 1x1/1 and 4x4/4 support
- *Spatial broadcasting* is supported in additional cases (using the Nearest Neighbor Resize kernel)
- *Reduce Sum* on features dimension support

- *External padding* layer support

Models allocation

- *Context switch* support (preview). This feature allows to run big models that utilize more than a single device's resources, by splitting them into several contexts and switching between them over PCIe
- Buffering long skip connections in the host's RAM over PCIe

Parser

- Automatic TF model format detection (1.x Checkpoint vs 2.x SavedModel)
- Tensorflow and ONNX *negative* operation support
- The newly supported layers are supported also in the TF and ONNX parsers

Dataflow Compiler API

- HAR command line tool improvements and new CLI syntax
- Log messages cleanup

Quantization

- Improved and extended the quantization analysis *tool* and tutorial
- *Per layer IBC* support

Note: *(for Ethernet users)* Starting at this version, *DDR portals* that buffer intermediate data in the host's RAM over PCIe are used automatically. This behavior optimizes the compilation for PCIe systems, however HEFs that use this feature are not supported when the Hailo device is connected to the host using another interface such as Ethernet. To ensure HEF's compatibility with such systems, this feature can be disabled during compilation using a *model script command*.

Dataflow Compiler v3.8.0 (May 2021)

Core

- *Conv 3x3/1 dilation=3* support
- *Elementwise Addition* on "flat" (rank 2) tensors support

Models allocation

- Improved Bilinear Resize allocation heuristics, so more such layers can be allocated together

Parser

- The Parser CLI tool `hailo parser` now supports an input shape tensor shape option
- The in-chip post processing API (`core_postprocess_api`) now supports custom regression layer predictions order
- The newly supported layers are also supported in the TF and ONNX parsers

Dataflow Compiler API

- The `join()` API supports additional cases, such as a common input tensor to multiple networks, and using the output of one network as the input for the other one (preview)
- The `ClientRunner` class support a new `revert_state()` method that reverts its state, e.g. from after quantization to before quantization
- New Hailo Archive CLI tool named `hailo har`
- Introduced a new quantization method in the `ClientRunner` class, named `quantize()`. The old APIs `run_quantization()` and `run_quantization_from_np()` are now deprecated.

Dataflow Compiler v3.7.1 (April 2021)

- Fixed the bug of missing Jupyter related packages in `requirements.txt`

Dataflow Compiler v3.7.0 (April 2021)

Core

- *Broadcast* support over features from (H,W,1) to (H,W,F)
- The *Maxpool* kernel supports additional cases such as valid padding, odd number of columns, and new kernel (filter) sizes
- *Multiplication by const (scalar)* support
- *Deconv* kernel optimization
- The *Conv* kernel supports additional cases such as 1x1/2x1 and 2x2/2x2

Models allocation

- Control resource optimization by merging two layers to use the same controller (preview)

Parser

- *Tensorflow 2* models parsing
- The newly supported layers are also supported in the TF and ONNX parsers

Quantization

- Fixed a bug where certain quantization algorithms like Equalization affected the native (pre quantization) emulation

Dataflow Compiler API

- Added a new `join()` API that unifies two models to be compiled together (preview)
- The benchmarks moved to their own package
- Added Hailo Archive (HAR) support to the *command line tools*

Note: The syntax of several command line tools such as `hailo compiler` has changed due to the Hailo Archive (HAR) support. See their help message for details.

Dataflow Compiler v3.6.0 (March 2021)

Core

- *Conv 3x3/1x1 dilation=8* support
- Conv 3x1/2x1 support
- *Space to depth* kernel support
- *16x4 mode* support in additional cases

Models allocation

- Improved allocation heuristic for inter-layer (activations) buffer sizes

Parser

- Space to depth support in the *Tensorflow 1.x Parser*.

Dataflow Compiler API

- Updated the *tutorials* to use Hailo Archive (HAR) files

- Weight files (NPZ) and archive files (HAR) size optimization

Note: This version only supports the HEF format for compiled Hailo models. The older JLF format is deprecated.

Note: This version only supports Ubuntu 18.04 and Python 3.6. Ubuntu 16.04 and Python 3.5 are deprecated.

Note: The pre-compiled benchmark models HEF files are temporarily missing in this version. They will be added in future versions. Compiling these HEFs is still supported using the `hailo benchmark build` command.

Dataflow Compiler v3.5.0 (February 2021)

Core

- *Deconv* 16x16/8 support
- *Group Deconv and Depthwise Deconv* support
- New *Maxpool* parameters support: 9x9/1, 13x13/1 (often used by the SPP block) and 2x2/2x1

Models allocation

- *DDR portal* support (preview). This portal buffers long skip connections in the host's RAM over PCIe
- Re-enabled the *Profiler's power estimations*
- *FPS per layer* command support to manually fine tune models' compilation and especially to reduce latency

Parser

- Tensorflow 2 models parsing (preview)
- The newly supported layers are supported also in the TF1.x and ONNX parsers

Dataflow Compiler API

- Changed the Dataflow Compiler architecture so the build server is no longer needed. All operations are now done in the client side.
- *Dead channels removal* tool

Note: The HEF format is the default format in this version. The JLF format is still supported as well, but it is expected to be deprecated.

Dataflow Compiler v3.4.1 (February 2021)

- Upgraded HailoRT and firmware

Dataflow Compiler v3.4.0 (January 2021)

Core

- Squeeze and Excitation building block support
- *Nearest Neighbor Resize* from 1x1xF to HxWxF support
- *Deconv* 8x8/4 support
- *Reduce Max* support

Parser

- *Squeeze and Excitation building block parsing* from TF and ONNX
- ReduceMax operation parsing support from TF and ONNX

Dataflow Compiler client API

- HEF format support when running models on the Hailo device inside a TF graph

---

Note: This version still supports both Ubuntu 16.04 and 18.04, and both Python 3.5 and 3.6. Ubuntu 16.04 and Python 3.5 are expected to be deprecated in future versions, so it's recommended to migrate to Ubuntu 18.04 and Python 3.6.

---

3. Dataflow Compiler Installation

Note: This section is for the installation of the Dataflow Compiler only. For a complete installation of Hailo Suite, which contains all Hailo SW products, see the SW Suite user guide.

3.1. System requirements

The Hailo Dataflow Compiler requires the following minimum hardware and software configuration:

1. Ubuntu 18.04/20.04, 64 bit

Warning: Ubuntu 18.04 will be deprecated in Hailo Dataflow Compiler future version

2. 16+ GB RAM (32+ GB recommended)
3. Python 3.6/3.7/3.8, including `pip` and `virtualenv`

Warning: Python 3.6 will be deprecated in Hailo Dataflow Compiler future version

4. `python3.6-dev`/ `python3.7-dev`/ `python3.8-dev` (accordingly), `python3-tk`, `graphviz`, and `libgraphviz-dev` packages

The following additional requirements are needed for GPU based hardware emulation:

1. Nvidia's Pascal/Turing/Ampere GPU architecture (such as Titan X Pascal, GTX 1080 Ti, RTX 2080 Ti, or RTX A4000)
2. GPU driver version 470
3. CUDA 11.2
4. CUDNN 8.1

Note: The Dataflow Compiler installs and runs Tensorflow, and when Tensorflow is installed from PyPi and runs on the CPU it requires AVX instructions support. Therefore, it is recommended to use a CPU that supports AVX instructions. Another option is to compile Tensorflow from sources without AVX.

Warning: These requirements are for the Dataflow Compiler, **which is used to build models**. Running inference using the HailoRT works on smaller systems as well. In order to run inference and demos on the Hailo-8 device, the latest HailoRT needs to be installed as well. See HailoRT's user guide for more details.

3.2. Installing / Upgrading Hailo Dataflow Compiler

Warning: This installation requires an internet connection (or a local pip server) in order to download Python packages.

Note: If you wish to upgrade both Hailo Dataflow Compiler and HailoRT which are installed in the same virtualenv: update HailoRT first, and then the Dataflow Compiler using the following instructions.

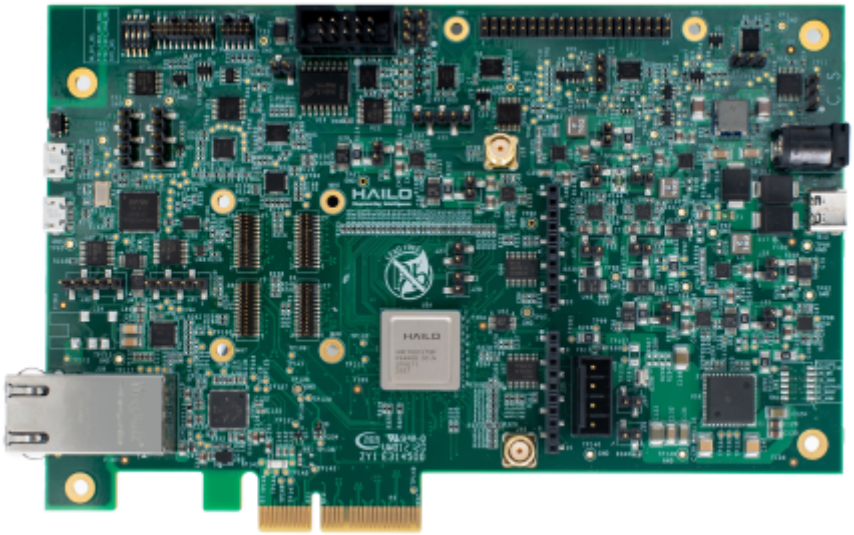

Figure 3. Hailo-8 Evaluation board

Figure 4. Hailo-8 M.2 board

Figure 5. Hailo-8 mPCIe board

Changed in version 3.15: Hailo Dataflow Compiler package is a Wheel file (.whl) that can be downloaded from Hailo's Developer Zone.

For a clean installation, create a virtualenv:

```
virtualenv <VENV_NAME>
```

Enter the virtualenv:

```
. <VENV_NAME>/bin/activate
```

And then, when inside the virtualenv, use (for 64-bit linux):

```
pip install <hailo_dataflow_compiler-X.XX.X-py3-none-linux_x86_64.whl>
```

If you already have an old version (v3.15.0 or newer), enter the virtualenv, and install using the line above. The old version will be updated automatically.

If you already have an old version (<=3.14.0), you have to uninstall it manually from within the existing virtualenv:

```
pip uninstall -y hailo_sdk_common hailo_sdk_client hailo_sdk_server hailo_model_optimization
```

Then install the new package with pip using the method above (the package names were changed from v3.14.0 to v3.15.0).

After installation / upgrade, it is recommended to view Hailo's CLI tool options with:

```
hailo -h
```

Note: You can validate the success of the install/update to latest Hailo packages, by running `pip freeze | grep hailo`.

---

4. Tutorials

The tutorials below go through the model build and inference steps. They are also available as Jupyter notebook files in the directory *VENV/lib/python.../site-packages/tutorials*.

It's recommanded to use the command `hailo tutorial` (when inside the virtualenv) to open a Jupyter server that contains the tutorials.

4.1. Dataflow Compiler tutorials introduction

The tutorials cover the Hailo Dataflow Compiler basic use-cases:

Model compilation:

It is recommended to start with the `Hailo Dataflow Compiler Overview / Model build process` section of the user guide.

The Hailo compilation process consists of three steps:

1. Converting a Tensorflow or ONNX neural-network graph into a Hailo-compatible representation.
2. Quantization of a full precision neural network model into an 8-bit model.
3. Compiling the network to Hailo8 binary files (HEF).

Inference:

1. Blocking inference with the HW-compatible model.
2. Streaming inference with the HW-compatible model.
3. Inference inside a Tensorflow environment.

These use-cases were chosen to show an end-to-end flow, beginning with a Tensorflow / ONNX model and ending with a hardware deployed model.

Throughout this guide we will use the Resnet-v1-18 neural network to demonstrate the capabilities of the Dataflow Compiler. The neural network is defined using Tensorflow checkpoint.

4.1.1. Usage

The HTML and PDF versions are for viewing-only. The best way to use the tutorials is to run them as Jupyter notebooks:

1. The Dataflow Compiler should be installed, either as a standalone Python package, or as part of the Hailo SW Suite.
2. You should activate the Dataflow Compiler virtual environment using `source <virtualenv_path>`
    1. When using the Suite docker, the virtualenv is activated automatically.
3. The tutorial notebooks are located in: `VENV/lib/python.../site-packages/hailo_tutorials`.
4. Running the command `hailo tutorial` will open a Jupyter server that allows viewing the tutorials locally by using the link give at the output of the command.
5. Remote viewing from a machine different then the one used to run the Jupyter server is also possible by running `hailo tutorial --ip=0.0.0.0`

4.2. Parsing tutorial

4.2.1. Hailo parsing example from Tensorflow CKPT to HAR

This tutorial will walk you through parsing Tensorflow checkpoints to the HAR format (Hailo Archive). HAR is a tar.gz archive file that contains the representation of the graph structure and the weights that are deployed to the Hailo hardware.

Note: **Running this code in Jupyter notebook is recommended**, see the Introduction tutorial for more details.

Note: This section demonstrates the Python APIs for Hailo Parser. You could also use the CLI: try `hailo parser {tf, ckpt, tf2, onnx} --help`. More details on Dataflow Compiler User Guide / Building Models / Profiler and other command line tools.

[ ]:
```
%matplotlib inline
import tensorflow as tf

from IPython.display import SVG
from hailo_sdk_client import ClientRunner, NNFramework
```

Choose the checkpoint files to be used throughout the tutorial:

[ ]:
```
model_name = 'resnet_v1_18'
ckpt_path = '../models/resnet_v1_18.ckpt'

start_node = 'resnet_v1_18/conv1/Pad'
end_node = 'resnet_v1_18/predictions/Softmax'
```

The main API of the Dataflow Compiler that the user interacts with is the ClientRunner class (see the API Reference section on the Dataflow Compiler user guide for more information).

First, initialize a ClientRunner and use the translate_tf_model method.

Arguments:

- model_path
- model_name to use
- start_node_names (list of str, optional): Name of the first node to parse.
- end_node_names (list of str, optional): List of nodes, that the parsing can stop after all of them are parsed.

For translating the model, supplying start and end node names might be crucial. You can use the `hailo tb` tool or any other model visualization tool to visualize the model and locate the nodes.

[ ]:
```
runner = ClientRunner(hw_arch='hailo8')
# For Mini PCIe modules or Hailo-8R devices, use hw_arch='hailo8r'
hn, npz = runner.translate_tf_model(ckpt_path, model_name, start_node_names=[start_node], end_node_
↪names=[end_node])
```

4.2.2. Hailo Archive

Hailo Archive is a tar.gz archive file that captures the "state" of the model - the files and attributes used in a given stage from parsing to compilation. You can use the `save_har` method to save the runner's state in any stage and `load_har` method to load a saved state to an uninitialized runner.

The initial HAR file includes: - HN file, which is a JSON-like representation of the graph structure that is deployed to the Hailo hardware. - NPZ file, which includes the weights of the model.

Save the parsed model in a Hailo Archive file:


```
[ ]: hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
runner.save_har(hailo_model_har_name)
```

Visualize the graph with the visualizer tool:

```
[ ]: !hailo visualizer {hailo_model_har_name} --no-browser
SVG('out.svg')
```

Run the profiler tool:

This command will pop-open the HTML report in the browser.

```
[ ]: !hailo profiler {hailo_model_har_name}
```

4.2.3. Parsing example from ONNX to HAR

Parsing of ONNX model to the HAR format is similar to parsing a Tensorflow model.

Choose the ONNX file to be used throughout the example:

```
[ ]: onnx_model_name = 'yolov3'
onnx_path = '../models/yolov3.onnx'
```

Initialize a ClientRunner and use the translate_onnx_model method.

Arguments:

- model_path
- model_name to use
- start_node_names (list of str, optional): Name of the first ONNX node to parse.
- end_node_names (list of str, optional): List of ONNX nodes, that the parsing can stop after all of them are parsed.
- net_input_shapes (dict, optional): A dictionary describing the input shapes for each of the start nodes given in start_node_names, where the keys are the names of the start nodes and the values are their corresponding input shapes. Use only when the original model has dynamic input shapes (described with a wildcard denoting each dynamic axis, e.g. [b, c, h, w]).

You can try translating the ONNX model without supplying the optional arguments.

```
[ ]: runner = ClientRunner(hw_arch='hailo8')
# For Mini PCIe modules or Hailo-8R devices, use hw_arch='hailo8r'
hn, npz = runner.translate_onnx_model(onnx_path, onnx_model_name,
                                      start_node_names=['input_0'],
                                      end_node_names=['890', '825', '760'],
                                      net_input_shapes={'input_0':[1, 3, 640, 640]})
```

4.2.4. Parsing example from Tensorflow 2

Parsing the Tensorflow 2.x SavedModel format is similar to parsing Tensorflow 1.x checkpoints. The Parser identifies the input format automatically, but the user can also specify it explicitly using the `nn_framework` parameter.

The following example shows how to parse a Tensorflow 2 model. It uses a small toy model, which is unrelated to the resnet_v1_18 checkpoint used above.

```
[ ]: model_name = 'dense_example'
model_path = '../models/dense_example_tf2/saved_model.pb'
```

(continues on next page)

(continued from previous page)

```
runner = ClientRunner(hw_arch='hailo8')
hn, npz = runner.translate_tf_model(model_path, model_name, nn_framework=NNFramework.TENSORFLOW2)
```

4.2.5. Common conversion methods from Tensorflow to Tensorflow Lite

The following examples focus on Tensorflow's TFLite converter support for various TF formats, showing how older formats of TF can be converted to TFLite, which can then be used in Hailo's parsing stage.

[ ]:
```
# Building a simple Keras model
def build_small_example_net():
    inputs = tf.keras.Input(shape=(24, 24, 96), name="img")
    x = tf.keras.layers.Conv2D(24, 1, name='conv1')(inputs)
    x = tf.keras.layers.BatchNormalization(momentum=0.9, name='bn1')(x)
    outputs = tf.keras.layers.ReLU(max_value=6.0, name='relu1')(x)
    model = tf.keras.Model(inputs, outputs, name="small_example_net")
    return model

# Converting the model to tflite
model = build_small_example_net()
model_name = 'small_example'
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert() # may cause warnings in jupyter notebook, don't worry.
tflite_model_path = '../models/small_example.tflite'
with tf.io.gfile.GFile(tflite_model_path, 'wb') as f:
    f.write(tflite_model)

# Parsing the model to Hailo format
runner = ClientRunner(hw_arch='hailo8')
hn, npz = runner.translate_tf_model(tflite_model_path, model_name, nn_framework=NNFramework.
↪TENSORFLOW_LITE)
```

[ ]:
```
# Alternatively, convert an already saved SavedModel to tflite
model_path = '../models/dense_example_tf2/'
model_name = 'dense_example_tf2'
converter = tf.lite.TFLiteConverter.from_saved_model(model_path)
tflite_model = converter.convert() # may cause warnings in jupyter notebook, don't worry.
tflite_model_path = '../models/dense_example_tf2.tflite'
with tf.io.gfile.GFile(tflite_model_path, 'wb') as f:
    f.write(tflite_model)

# Parsing the model to Hailo format
runner = ClientRunner(hw_arch='hailo8')
hn, npz = runner.translate_tf_model(tflite_model_path, model_name, nn_framework=NNFramework.
↪TENSORFLOW_LITE)
```

4.3. Model Optimization Tutorial

4.3.1. Hailo optimization example from HAR file

This tutorial will walk you through quantizing your model. The input to this tutorial is a HAR file (before quantization; with native weights) and the output will be an optimized HAR file with quantized weights.

Note: Quantization is only a part of the Model Optimization process performed by Hailo. In this tutorial those words are used interchangeably. For full information see the `Dataflow Compiler user guide / Model optimization` section.

Requirements:

- Run this code in Jupyter notebook, see the Introduction tutorial for more details.
- Verify that you've completed the Parsing Tutorial (or created the HAR file in other way)

Recommendation:

- To obtain best performance run this code with a GPU machine. For full information see the `Dataflow Compiler user guide / Model optimization` section.

[ ]:
```
%matplotlib inline
import os
import json
import numpy as np
import tensorflow as tf
import pandas as pd

from IPython.display import SVG
from IPython.display import display
from matplotlib import pyplot as plt
from PIL import Image
from hailo_sdk_client import ClientRunner
from hailo_sdk_common.targets.inference_targets import SdkNative, SdkNumeric
```

Choose a Hailo Archive file to use throughout the example:

[ ]:
```
model_name = 'resnet_v1_18'
hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
assert os.path.isfile(hailo_model_har_name), 'Please provide valid path for HAR file'
```

Load the network to the ClientRunner from the saved Hailo Archive file:

[ ]:
```
runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)
```

Prepare the calibration data for optimization:

Note that Hailo device is capable of running on-chip pre-processing, such as normalization and resize.

Here we assume the Imagenet dataset is used and the normalization is offloaded to the chip.

[ ]:
```
from tensorflow.python.eager.context import eager_mode

def preproc(image, output_height=224, output_width=224, resize_side=256):
    ''' imagenet-standard: aspect-preserving resize to 256px smaller-side, then central-crop to 224px'
↪''
    with eager_mode():
        h, w = image.shape[0], image.shape[1]
        scale = tf.cond(tf.less(h, w), lambda: resize_side / h, lambda: resize_side / w)
        resized_image = tf.compat.v1.image.resize_bilinear(tf.expand_dims(image, 0), [int(h*scale),␣
↪int(w*scale)])
        cropped_image = tf.compat.v1.image.resize_with_crop_or_pad(resized_image, output_height,␣
↪output_width)
        return tf.squeeze(cropped_image)

images_path = '../data'
images_list = [img_name for img_name in os.listdir(images_path) if
               os.path.splitext(img_name)[1] == '.jpg']
calib_dataset = np.zeros((len(images_list), 224, 224, 3), dtype=np.float32)
for idx, img_name in enumerate(sorted(images_list)):
    img = np.array(Image.open(os.path.join(images_path, img_name)))
    img_preproc = preproc(img)
    calib_dataset[idx,:,:,:] = img_preproc.numpy().astype(np.uint8)

np.save('calib_set.npy', calib_dataset)
plt.imshow(img, interpolation='nearest')
```

(continues on next page)

(continued from previous page)

```
plt.title('Original image')
plt.show()
plt.imshow(np.array(calib_dataset[idx,:,:,:], np.uint8), interpolation='nearest')
plt.title('Preprocessed image')
plt.show()
```

4.3.2. Run the model optimization

- We use Model Script (.alls) file, that includes commands that control the optimization (and later on, compilation) process.
- The Normalization command is used to offload normalization to the device before optimization. It means you don't need to run normalization on-host. Detailed explanation is found in the "Model Modification" section below.
- Here we will run naive optimization, for advanced topics in optimization please refer to the advanced section below.

Note: This section demonstrates the Python APIs for Hailo Model Optimization. You could also use the CLI: try `hailo optimize --help`. More details on Dataflow Compiler User Guide / Building Models / Profiler and other command line tools.

[ ]:
```
csv_path = 'translated_params_resnet_v1_18.csv'
quantized_model_har_path = '{}_quantized_model.har'.format(model_name)

alls_lines = [
    # Add normalization layer with mean [123.675, 116.28, 103.53] and std [58.395, 57.12, 57.375])
    'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.375])\n',
    # For multiple input nodes:
    # {normalization_layer_name_1} = normalization([list of means per channel], [list of stds per␣
↪channel], {input_layer_name_1_from_hn})\n',
    # {normalization_layer_name_2} = normalization([list of means per channel], [list of stds per␣
↪channel], {input_layer_name_2_from_hn})\n',
    # ...
    'model_optimization_config(calibration, batch_size=2)\n',  # Batch size is 8 by default
]

# Save the commands in an .alls file, this is the Model Script
open('simple_script.alls','w').writelines(alls_lines)

# Load the model script to ClientRunner so it will be considered on optimization
runner.load_model_script('simple_script.alls')

# For a single input layer, could use the shorter version - just pass the dataset to the function
# runner.optimize(calib_dataset)
# For multiple input nodes, the calibration dataset could also be a dictionary with the format:
# {input_layer_name_1_from_hn: layer_1_calib_dataset, input_layer_name_2_from_hn: layer_2_calib_
↪dataset}
hn_layers = runner.get_hn_dict()['layers']
print([layer for layer in hn_layers if hn_layers[layer]['type'] == 'input_layer']) # See available␣
↪input layer names
calib_dataset_dict = {'resnet_v1_18/input_layer1': calib_dataset} # In our case there is only one␣
↪input layer
runner.optimize(calib_dataset_dict)


runner.save_har(quantized_model_har_path)
!hailo params-csv {quantized_model_har_path} --csv-path {csv_path}
display(pd.read_csv(csv_path))
```

Evaluate the results by comparing the full precision and the numeric emulation modes' results:


```
[ ]: def _get_imagenet_labels(json_path='../data/imagenet_names.json'):
         imagenet_names = json.load(open(json_path))
         imagenet_names = [imagenet_names[str(i)] for i in range(1001)]
         return imagenet_names[1:]

     imagenet_labels = _get_imagenet_labels()

     def mynorm(data):
         return (data-np.min(data)) / (np.max(data)-np.min(data))

     def net_eval(runner, target, images):
         with tf.Graph().as_default():
             network_input = tf.compat.v1.placeholder(dtype=tf.float32)
             sdk_export = runner.get_tf_graph(target, network_input)
             with sdk_export.session.as_default():
                 sdk_export.session.run(tf.compat.v1.local_variables_initializer())
                 print(sdk_export.output_tensors)
                 probs_batch = sdk_export.session.run(sdk_export.output_tensors, feed_dict={network_input:␣
     ↪images})
         return probs_batch

     def show_results(images, native_res, numeric_res):
         top_inds = []
         top_inds_q = []
         for img, single_nat_res, single_num_res in zip(images, native_res, numeric_res):
             top_ind = np.argmax(single_nat_res)
             top_ind_q = np.argmax(single_num_res)

             plt.figure()
             plt.imshow(mynorm(img))

             plt.title('Full Precision: top-1 class is {0}. Confidence is {1:.2f}%,\n\
                       Quantized: top-1 class is {2}. Confidence is {3:.2f}%'.format(
                       imagenet_labels[top_ind], 100*single_nat_res[top_ind],
                       imagenet_labels[top_ind_q], 100*single_num_res[top_ind_q]))
             top_inds.append(top_ind)
             top_inds_q.append(top_ind_q)
         return (top_inds, top_inds_q)

     images = calib_dataset[:10,:,:,:]
     native_res = net_eval(runner, SdkNative(), images)
     numeric_res = net_eval(runner, SdkNumeric(), images)

     res = show_results(images, native_res[0], numeric_res[0])
```

4.3.3. Model modifications

Adding on-chip input format conversion through model script commands

This block will apply model modification commands using a model script. We will be adding a YUY2->YUV->RGB conversion. Note that the YUY2->YUV conversion will only be part of the compiled model but it won't be part of the optimization process so calibration data should be in YUV format.

- Initialize Client Runner
- Load YUV dataset
- Load model script with the relevant commands
- Using the optimize() API, the commands are applied and the model is quantized
- Usage:

- To create input conversion after a specific layer: yuv_to_rgb_layer = input_conversion(input_layer1, yuv_to_rgb)
- To create input conversion after all input layers: net_scope1/yuv2rgb1, net_scope2/yuv2rgb2 = input_conversion(yuv_to_rgb)

[ ]:
```
model_name = 'resnet_v1_18'
hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
assert os.path.isfile(hailo_model_har_name), 'Please provide valid path for HAR file'
runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)

calib_dataset_yuv = np.load('../model_modifications/calib_dataset_yuv.npz')
# Now we're adding yuy2_to_yuv conversion before the yuv_to_rgb and a normalization layer.
# The order of the layers is determined by the order of the commands in the model script:
# First we add normalization to the original input layer -> the input to the network is now␣
↪normalization1
# Then we add yuv_to_rgb layer, so the order will be: yuv_to_rgb1->normalization1->original_network
# Lastly, we add yuy2_to_yuv layer, so the order will be: yuy2_to_yuv1->yuv_to_rgb->normalization1->
↪original_network
model_script_commands = [
    'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.375])\n',
    'yuv_to_rgb1 = input_conversion(yuv_to_rgb)\n',
    'yuy2_to_yuv1 = input_conversion(yuy2_to_hailo_yuv)\n']
model_script_path = 'model_modifications.alls'
with open(model_script_path, 'w') as model_script:
    model_script.writelines(model_script_commands)
runner.load_model_script(model_script_path)
runner.optimize(calib_dataset_yuv['yuv_dataset'])

modified_model_har_name = '{}_modified.har'.format(model_name)
runner.save_har(modified_model_har_name)
!hailo visualizer {modified_model_har_name} --no-browser
SVG('out.svg')
```

Adding on-chip input resize through model script commands

This block will apply on-chip bilinear image resize at the beginning of the network through model script commands. * Create a bigger (640x480) calibration set out of the Imagenet dataset * Initialize Client Runner * Load the new calibration set * Load the model script with the resize command * Using the optimize() API, the command is applied and the model is quantized

[ ]:
```
images_path = '../data'
images_list = [img_name for img_name in os.listdir(images_path) if
               os.path.splitext(img_name)[1] == '.jpg']
calib_dataset_new = np.zeros((len(images_list), 480, 640, 3), dtype=np.float32)
for idx, img_name in enumerate(images_list):
    img = Image.open(os.path.join(images_path, img_name))
    resized_image = np.array(img.resize((640,480), Image.BILINEAR))
    calib_dataset_new[idx,:,:,:] = resized_image.astype(np.uint8)

np.save('calib_set_480_640.npy', calib_dataset_new)
plt.imshow(img)
plt.title('Original image')
plt.show()
plt.imshow(np.array(calib_dataset_new[idx,:,:,:], np.uint8))
plt.title('Resized image')
plt.show()
```

[ ]:
```
model_name = 'resnet_v1_18'
hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
assert os.path.isfile(hailo_model_har_name), 'Please provide valid path for HAR file'
runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)
```

(continues on next page)

(continued from previous page)

```
calib_dataset_large = np.load('calib_set_480_640.npy')
# Add a bilinear resize from 480x640 to the network's input size - in this case, 224x224.
# The order of the layers is determined by the order of the commands in the model script:
# First we add normalization to the original input layer -> the input to the network is now␣
↪normalization1
# Then we add resize layer, so the order will be: resize_input1->normalization1->original_network
model_script_commands = [
    'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.375])\n',
    'resize_input1 = resize_input([480,640])\n']
model_script_path = 'model_resize.alls'

with open(model_script_path, 'w') as model_script:
    model_script.writelines(model_script_commands)
runner.load_model_script(model_script_path)
calib_dataset_dict = {'resnet_v1_18/input_layer1': calib_dataset_large} # In our case there is only␣
↪one input layer
runner.optimize(calib_dataset_dict)

modified_model_har_name = '{}_resized.har'.format(model_name)
runner.save_har(modified_model_har_name)
!hailo visualizer {modified_model_har_name} --no-browser
SVG('out.svg')
```

4.3.4. Advanced optimization - compression and optimization levels

For aggressive quantization (compress significant amount of weights to 4 bits), we'll need to use higher optimization level to obtain good results. For quick iterations we always recommend starting with the default setting of the model optimizer (optimization_level=1, compression_level=1). However, when moving to production, we recommended to work at the highest complexity level to achieve optimal accuracy. With regards to compression, users should increase it when the overall throughput/latency of the model is not good enough. Note that increasing compression would have negligible effect on power-consumption so the motivation to work with higher compression level is mainly due to FPS considerations.

Here we set the compression level to 4 (which means ~80% of the weights will be quantized into 4 bits) using the compression_level param in a model script and run the model optimization again. Using 4 bit weights might reduce the model's accuracy but will help to reduce the model's memory footprint. In this example, we can see the confidence of some examples decreases after changing several layers to 4bit weights, later the confidence will improve after applying higher optimization_level.

```
[ ]: alls_lines = [
    'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.375])\n',
    'model_optimization_flavor(optimization_level=0, compression_level=4, batch_size=2)\n'  # Batch␣
↪size is 8 by default
]
# -- Reduces weights memory by 80% !

open('simple_4bit.alls','w').writelines(alls_lines)

runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)
runner.load_model_script('simple_4bit.alls')
runner.optimize(calib_dataset)
```

```
[ ]: images = calib_dataset[:10,:,:,:]
native_res = net_eval(runner, SdkNative(), images)
numeric_res = net_eval(runner, SdkNumeric(), images)

res = show_results(images, native_res[0], numeric_res[0])
```


Now, repeating the same process with higher optimization level (For full information see the `Dataflow Compiler user guide / Model optimization` section):

```
[ ]: runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)

images = calib_dataset[:10,:,:,:]

alls_lines = [
    'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.375])\n',
    'model_optimization_flavor(optimization_level=1, compression_level=4, batch_size=2)\n'  # Batch size is 8 by default
]
# -- Reduces weights memory by 80% !

open('highest_optimization_level_and_4bit.alls','w').writelines(alls_lines)

runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)
runner.load_model_script('highest_optimization_level_and_4bit.alls')
runner.optimize(calib_dataset)

native_res = net_eval(runner, SdkNative(), images)
numeric_res = net_eval(runner, SdkNumeric(), images)

top_inds, top_inds_q = show_results(images, native_res[0], numeric_res[0])
```

```
[ ]: print('Full precision predictions:                        {0}\n'\
      'Quantized predictions (with optimization_level=1): {1} ({2}/{3})\n'\
      'Quantized predictions (with optimization_level=0): {4} ({5}/{6})'.format(top_inds,
                                                                                top_inds_q,
                                                                                sum(np.array(top_
inds) == np.array(top_inds_q)),
                                                                                len(top_inds_q),
                                                                                res[1],
                                                                                sum(np.array(top_
inds) == np.array(res[1])),
                                                                                len(top_inds_q)))
```

Finally, save the optimized model to a Hailo Archive file:

```
[ ]: runner.save_har(quantized_model_har_path)
```

4.4. Layer noise analysis tool tutorial

This is an advanced tutorial; You may skip it if your quantization results were satisfying.

This tutorial will guide you through a model quantization analysis, using HailoQuantizationAnalyzer tool that essentially breaks down the quantization noise per layer. The tool's input are the model's HAR file (with full precision weights), as well as .npy images data for calibration and testing. The output is a short noise report summarizing the main quantization noise information per layer.

The tutorial is intended to guide the user in using HailoQuantizationAnalyzer tool, by using it to analyze the classification model Resnet-v1-18, where the layer conv7 is quantized to 4 bits.

The flow is mainly comprised of 3 parts:

- **Input definitions:** Defining the paths to HAR and data for calibration and testing.
- **Model loading and quantization of all model layers:** This step is followed by an analysis step that computes quantization noise at each layer output.
- **Layer by Layer analysis:** This step is the heart of the tool, and computes the quantization noise of each layer output, when the given layer is the **only** quantized layer, while the rest are kept in full precision. This highlights

the quantization sensitivity of the model to that specific layer noise.

Requirements:

- Run this code in Jupyter notebook, see the Introduction tutorial for more details.
- Verify that you've completed the Parsing tutorial (to generate the parsed model Hailo Archive) and the Model Optimization tutorial (to generate the calibration set) or have created them in another way.

[ ]:
```
%matplotlib inline
import os
from hailo_sdk_client.tools.hailo_lat_cli import HailoQuantAnalyzerWrapper
from hailo_model_optimization.algorithms.lat_utils.lat_utils import AnalysisMode
```

4.4.1. Define the input data

The input to the tool is defined by the following parameters that specify all necessary inputs:

- `data_path`: path to preprocessed .npy image files for testing
- `calib_path`: path to preprocessed .npy image files for calibration
- `hailo_model_har_path`: path to the Hailo model (parsed model), a Hailo Archive file

[ ]:
```
data_path = './calib_set.npy'
calib_path = './calib_set.npy'
hailo_model_har_path = './resnet_v1_18_hailo_model.har'
```

You can set your working directory, the data saving path using the `work_dir` variable. This is optional, it can be left as `work_dir=None`, which will cause the data to be saved in a directory named `work_dir` automatically created in the current working path.

[ ]:
```
work_dir = None # if None, a directory name 'work_dir' will be created as a sub-folder in the current␣
↪working path
if work_dir is None:
    work_dir = os.path.join(os.getcwd(), 'work_dir')
if not os.path.isdir(work_dir):
    os.makedirs(work_dir)
```

4.4.2. GPU/CPU settings

[ ]:
```
import os
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"]="5"
```

4.4.3. LAT basic usage

For using the LAT tool with its basic API we only need to suply it the following parameters:

- `model_path`
- `data_path`
- `calib_path`
- `work_dir`

[ ]:
```
batch_size = 8
analyzer = HailoQuantAnalyzerWrapper()
analyzer.run(model_path=hailo_model_har_path, data_path=data_path, calib_path=calib_path, work_
↪dir=work_dir, eval_num=16)
```


4.4.4. The final raw results are saved in the work_dir path as well as the pdf file

```
[ ]: model_name = 'resnet_v1_18'
assert os.path.exists(f'{work_dir}/{model_name}_report.pdf'), "Error! .pdf report file was not␣
↪generated :/"
```

4.4.5. A ~One Liner Default Quantization Analysis

The Analyzer can operate in two type of run modes, controlled via the names in the Enum `AnalysisMode`:

- Quantization analysis run for the globally quantized network (`mode=AnalysisMode.full_quant_net.name` or `'full_quant_net'`)
- A layer by layer quantization analysis run (`mode=AnalysisMode.layer_by_layer.name` or `'layer_by_layer'`))

The user is able to run a default full analysis with the simple command in the previeus code, or he could set the mode of the analyze by passing the argument to analyze_mode. At the end of the analysis, all analysis figures will be plotted.

If a `AnalysisMode.layer_by_layer.name` mode is chosen, the user may also specify which layers to analyze by passing a list of layers to `run_analysis`. If not set, all layers in the model are analyzed.

Note1: Depending on the size of the model and/or the data, this might take a while.

None2: If you are using the CLI and you want to specify analyzed layers, you should pass the list with space seperation as following: –layers resnet_v1_18/conv1 resnet_v1_18/conv4

```
[ ]: analyzer.run(model_path=hailo_model_har_path, data_path=data_path, calib_path=calib_path, work_
↪dir=work_dir,
            analyze_mode=AnalysisMode.layer_by_layer.name, layers=['resnet_v1_18/conv1', 'resnet_v1_
↪18/conv4'])
```

4.4.6. Additional LAT arguments

The LAT can recive the followinf arguments: * `batch_size` - The batch size for the analayze inference (will not affect the batch_size of the calibration) * `eval_num` - The amount of samples to take from the entire data_calib_set. If the eval_num is smaller then the calib_set size, the analayzer will use the full calib_set. * `inverse` - inverse mode is by default False, but if set to True, the analyzer will do act in each of the modes as follow: layer_by_layer - in each layer analyze iteration, the analyzer will quant the entire model except for the analyzed layer. full_quant_mode - the analyzer will quant only the layers that are not chosen for the analyze. Note: if all layers has been chosen so the inverse mode will be just inffering again on the native model. * `show_figures` - Set this argument to True for plotting the results at the end of the run

```
[ ]: analyzer.run(model_path=hailo_model_har_path, data_path=data_path, calib_path=calib_path, work_
↪dir=work_dir, batch_size=4, eval_num=16, inverse=True, show_figures=True)
```

4.5. Compilation tutorial

4.5.1. Hailo compilation example from Hailo Archive quantized model to HEF

This tutorial will walk you through compiling the network to Hailo8 binary files (HEF).

Requirements:

- Run this code in Jupyter notebook, see the Introduction tutorial for more details.
- Have a quantized HAR file.

Note: This section demonstrates the Python APIs for Hailo Compiler. You could also use the CLI: try `hailo compiler --help`. More details on Dataflow Compiler User Guide / Building Models / Profiler and other command line tools.

```
[ ]: from hailo_sdk_client import ClientRunner
     from hailo_sdk_common.targets.inference_targets import ParamsKinds
```

Choose the quantized model Hailo Archive file to use throughout the example:

```
[ ]: model_name = 'resnet_v1_18'
     quantized_model_har_path = '{}_quantized_model.har'.format(model_name)
```

Load the network to the ClientRunner:

```
[ ]: runner = ClientRunner(hw_arch='hailo8', har_path=quantized_model_har_path)
     # For Mini PCIe modules or Hailo-8R devices, use hw_arch='hailo8r'
```

Run compilation (This method can take a couple of minutes):

Note: The `hailo compiler` CLI tool can also be used.

```
[ ]: hef = runner.compile()

     file_name = model_name + '.hef'
     with open(file_name, 'wb') as f:
         f.write(hef)
```

4.5.2. Profiler tool

Run the profiler tool (in post-placement mode, which is accurate to the on-chip allocation):

This command will pop-open the HTML report in the browser.

```
[ ]: har_path = '{}_compiled_model.har'.format(model_name)
     runner.save_har(har_path)
     !hailo profiler {har_path} --mode post_placement
```

Note:

The HTML profiler report could be augmented with runtime statistics, that are saved after the .hef ran on the device using hailortcli.

For more information look under the section: Dataflow Compiler User Guide / Building Models / Profiler and other command line tools / Running the Profiler.

4.6. Inference tutorial

This tutorial will walk you through the inference process.

Requirements:

- HailoRT installed on the same virtual environment, or as part of the Hailo SW Suite.
- Run this code in Jupyter notebook, see the Introduction tutorial for more details.
- Run the *Compilation Tutorial* before running this one.

Note: This section demonstrates PyHailoRT, which is a python library for communication with Hailo devices. For evaluation purposes, you can use `hailortcli run --help` (or the alias `hailo run --help`). More details on HailoRT User Guide / Command Line Tools.

4.6.1. Standalone hardware deployment

The standalone flow allows direct access to the HW, developing applications directly on top of Hailo core HW, using HailoRT.

This way we can use the Hailo hardware without Tensorflow, and even without the Hailo Dataflow Compiler (after the HEF is built).

A HEF is Hailo's binary format for neural networks. The HEF file contains:

- Target HW configuration
- Weights
- Metadata for HailoRT (e.g. input/output scaling)

First create the desired target object. In our example we use the Hailo-8 PCIe interface:

[ ]:
```
from multiprocessing import Process

import numpy as np
from hailo_platform import (HEF, PcieDevice, HailoStreamInterface, InferVStreams, ConfigureParams,
                            InputVStreamParams, OutputVStreamParams, InputVStreams, OutputVStreams,␣
↪FormatType)

# The target can be used as a context manager ("with" statement) to ensure it's released on time.
# Here it's avoided for the sake of simplicity
target = PcieDevice()

# Loading compiled HEFs to device:
model_name = 'resnet_v1_18'
hef_path = f'{model_name}.hef'
hef = HEF(hef_path)

# Get the "network groups" (connectivity groups, aka. "different networks") information from the .hef
configure_params = ConfigureParams.create_from_hef(hef=hef, interface=HailoStreamInterface.PCIe)
network_groups = target.configure(hef, configure_params)
network_group = network_groups[0]
network_group_params = network_group.create_params()

# Create input and output virtual streams params
# Quantized argument signifies whether or not the incoming data is already quantized.
# Data is quantized by HailoRT if and only if quantized == False .
input_vstreams_params = InputVStreamParams.make(network_group, quantized=False,
                                                format_type=FormatType.FLOAT32)
output_vstreams_params = OutputVStreamParams.make(network_group, quantized=True,
                                                  format_type=FormatType.UINT8)

# Define dataset params
input_vstream_info = hef.get_input_vstream_infos()[0]
output_vstream_info = hef.get_output_vstream_infos()[0]
image_height, image_width, channels = input_vstream_info.shape
num_of_images = 10
low, high = 2, 20

# Generate random dataset
dataset = np.random.randint(low, high, (num_of_images, image_height, image_width, channels)).
↪astype(np.float32)
```


Running hardware inference

Infer the model and then display the output shape:

[ ]:
```
input_data = {input_vstream_info.name: dataset}

with InferVStreams(network_group, input_vstreams_params, output_vstreams_params) as infer_pipeline:
    with network_group.activate(network_group_params):
        infer_results = infer_pipeline.infer(input_data)
        # The result output tensor is infer_results[output_vstream_info.name]
        print(f'Stream output shape is {infer_results[output_vstream_info.name].shape}')
```

4.6.2. Streaming inference

This section shows how to run streaming inference using multiple processes in Python.

We will not use infer. Instead we will use a send and receive model. The send function and the receive function will run in different processes.

Define the send and receive functions:

[ ]:
```
def send(configured_network, num_frames):
    vstreams_params = InputVStreamParams.make(configured_network)
    with InputVStreams(configured_network, vstreams_params) as vstreams:
        configured_network.wait_for_activation(1000)
        vstream_to_buffer = {vstream: np.ndarray([1] + list(vstream.shape), dtype=vstream.dtype) for␣
↪vstream in
                             vstreams}
        for _ in range(num_frames):
            for vstream, buff in vstream_to_buffer.items():
                vstream.send(buff)


def recv(configured_network, num_frames):
    vstreams_params = OutputVStreamParams.make(configured_network)
    configured_network.wait_for_activation(1000)
    with OutputVStreams(configured_network, vstreams_params) as vstreams:
        for _ in range(num_frames):
            for vstream in vstreams:
                data = vstream.recv()
```

Define the amount of images to stream and processes, then recreate the target and run the processes:

[ ]:
```
# Define the amount of frames to stream
num_of_frames = 1000

# Start the streaming inference
send_process = Process(target=send, args=(network_group, num_of_frames))
recv_process = Process(target=recv, args=(network_group, num_of_frames))
recv_process.start()
send_process.start()
print(f'Starting streaming (hef=\'{model_name}\', num_of_frames={num_of_frames})')
with network_group.activate(network_group_params):
    send_process.join()
    recv_process.join()
print('Done')
```

4.6.3. Hardware deployment in Tensorflow environment

The `get_tf_graph()` method that was used for emulation in the model optimization tutorial can also be used for running inference on the Hailo device inside a TF environment. When calling this function with an hardware target and the `use_preloaded_compilation` flag turned on, compilation is skipped and the HEF used is the one we already loaded.

First, create the runner and the target:

[ ]:
```
import tensorflow as tf

from hailo_sdk_client import ClientRunner

quantized_model_har_path = f'{model_name}_quantized_model.har'
runner = ClientRunner(hw_arch='hailo8', har_path=quantized_model_har_path)
# For Mini PCIe modules or Hailo-8R devices, use hw_arch='hailo8r'
```

The Tensorflow graph is generated using the preloaded complied HEF.

The `get_tf_graph()` function can re-create the HEF and load it to the device by removing the `use_preloaded_compilation` flag, but that will require loading the quantized parameters to the runner before calling it.

[ ]:
```
graph = tf.Graph()
with graph.as_default():
    network_input = tf.compat.v1.placeholder(dtype=tf.float32)
    preprocess = network_input
    sdk_export = runner.get_tf_graph(
        target=target,
        nodes=preprocess,
        translate_input=True,
        rescale_output=True,
        use_preloaded_compilation=True,
        network_groups=network_groups
    )
    postprocess = tf.argmax(sdk_export.output_tensors[0], axis=1)

    # Running hardware inference:
    with runner.hef_infer_context(sdk_export):
        with sdk_export.session.as_default():
            for i in range(100):
                dataset = np.random.randint(low, high, (num_of_images, image_height, image_width,␣
↪channels)).astype(np.uint8)
                feed_dict = {preprocess: dataset}
                results = sdk_export.session.run(postprocess, feed_dict=feed_dict)
```

4.6.4. Runtime profiler

This will demonstrate the usage of the HTML runtime profiler.

[ ]:
```
model_name = 'resnet_v1_18'
hef_path = f'{model_name}.hef'
compiled_har_path = '{}_compiled_model.har'.format(model_name)

# Run hailortcli (can use `hailo` instead) to run the .hef on the device, and save runtime statistics␣
↪to runtime_data.json)
!hailortcli run {hef_path} collect-runtime-data
# Use post-placement profiler with the runtime statistics to enable the RUNTIME tab
!hailo profiler {compiled_har_path} --mode post_placement --runtime-data runtime_data.json --out-path␣
↪runtime_profiler.html
```


resnet_v1_18 is a small network, which fits in a single device without context-switch (it is called "single context"). Its FPS and Latency are displayed on the "SIMPLE" tab of the pre or post placement reports. In that case, the Runtime Profiler displays the single context's load time (relevant if your application switches between models). After the load, frames are passed through it with the FPS and Latency from the SIMPLE tab.

The Runtime Profiler is most useful with big models, where the FPS and latency cannot be calculated on compile time. In that case, the Runtime profiler displays the load, config and runtime of the contexts, and the sum of all is the latency of a single frame (or batch, if you use hailortcli run with –batch-size argument).

The runtime FPS is also displayed on the hailortcli output.

4.7. Multiple models tutorial

This is an advanced tutorial, that demonstrates the joint compilation of multiple models into a single .hef file.

This is not a must; You can compile the models separately and use the two .hef files in your application.

Why would you want to join the models and compile together?

- Save bandwidth by connecting the same input to multiple networks.
- Chaining networks one after another.
- Keeping the networks separate, side by side (two different connected components), for performance optimizations in some cases.

Requirements:

- HailoRT installed on the same virtual environment, or as part of the Hailo SW Suite.
- Run this code in Jupyter notebook, see the Introduction tutorial for more details.

Throughout this guide we will run two networks, Resnet-v1-18 and MobileNet SSD in parallel.

4.7.1. Parsing

Each network is contained in a Tensorflow checkpoint.

Each Tensorflow checkpoint will be parsed into a HAR format, which is a tar.gz archive file which contain the representation of the graph structure and its weights that are deployed to the Hailo hardware.

To parse a new checkpoint, use the `translate_tf_model` method with `start_nodes` and end_nodes parameters indicating the input and output nodes.

For more details check the "Parsing Tutorial".

[ ]:
```
from hailo_sdk_client import ClientRunner

# Parse Resnet-V1-18 model
model_name_resnet = 'resent_v1_18'
ckpt_path = '../models/resnet_v1_18.ckpt'

start_node = 'resnet_v1_18/conv1/Pad'
end_node = 'resnet_v1_18/predictions/Softmax'

runner1 = ClientRunner(hw_arch='hailo8')
# For Mini PCIe modules or Hailo-8R devices, use hw_arch='hailo8r'
runner1.translate_tf_model(ckpt_path, model_name_resnet, start_node_names=[start_node], end_node_
↪names=[end_node])

# Parse MobileNet SSD model
model_name_mobilenet = 'mobilenet_ssd'
ckpt_path = '../models/mobilenet_ssd.ckpt'
```

(continues on next page)

(continued from previous page)

```
start_node = 'FeatureExtractor/MobilenetV1/MobilenetV1/Conv2d_0/Conv2D'
end_nodes = [
    'BoxPredictor_0/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_0/ClassPredictor/BiasAdd',
    'BoxPredictor_1/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_1/ClassPredictor/BiasAdd',
    'BoxPredictor_2/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_2/ClassPredictor/BiasAdd',
    'BoxPredictor_3/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_3/ClassPredictor/BiasAdd',
    'BoxPredictor_4/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_4/ClassPredictor/BiasAdd',
    'BoxPredictor_5/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_5/ClassPredictor/BiasAdd'
]

runner2 = ClientRunner(hw_arch='hailo8')
runner2.translate_tf_model(ckpt_path, model_name_mobilenet, start_node_names=[start_node], end_node_
↪names=end_nodes)
```

4.7.2. Model Optimization

In this step we will quantize the native weights to numeric (quantized) weights.

To quantize the model use the `quantize` method. The method expects a dictionary of `numpy.ndarray` per input layer name.

We will be using `numpy.ndarray` datasets generated from images that were pre-processed beforehand to match the way the networks were trained and saved as numpy compressed file.

For more details check the “Model Optimization Tutorial”.

[ ]:
```
import numpy as np

calib_set = np.load('../multiple_inputs_and_outputs/data/calib_set.npz')
runner1.quantize(calib_set['resnet_v1_18'], batch_size=2, calib_num_batch=16)
runner2.quantize(calib_set['mobilenet_ssd'], batch_size=2, calib_num_batch=16)
```

4.7.3. Join Runners

In this step we will join the two models so they will be compiled together.

[ ]:
```
runner1.join(runner2, scope1_name='resnet_v1_18', scope2_name='mobilenet_ssd')

har_name = '{}_quantized.har'.format(runner1.model_name)
runner1.save_har(har_name)
```

To view the new network structure you can use TensorBoard:

```
hailo tb -r joined_resnet_v1_18_mobilenet_ssd_quantized.har
```


Advanced Join Flows

In this step, we join two models and merge their inputs, once automatically and then by specifying the names of the input layers to merge using the join_action_info dictionary.

[ ]:
```
from hailo_sdk_client import JoinAction

model_name1 = 'mobilenet_ssd_1'
model_name2 = 'mobilenet_ssd_2'
ckpt_path = '../models/mobilenet_ssd.ckpt'

start_node = 'FeatureExtractor/MobilenetV1/MobilenetV1/Conv2d_0/Conv2D'
end_nodes = [
    'BoxPredictor_0/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_0/ClassPredictor/BiasAdd',
    'BoxPredictor_1/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_1/ClassPredictor/BiasAdd',
    'BoxPredictor_2/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_2/ClassPredictor/BiasAdd',
    'BoxPredictor_3/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_3/ClassPredictor/BiasAdd',
    'BoxPredictor_4/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_4/ClassPredictor/BiasAdd',
    'BoxPredictor_5/BoxEncodingPredictor/BiasAdd',
    'BoxPredictor_5/ClassPredictor/BiasAdd'
]

# Join and merge inputs using the automatic inputs join function
runner3 = ClientRunner(hw_arch='hailo8')
runner4 = ClientRunner(hw_arch='hailo8')
runner3.translate_tf_model(ckpt_path, model_name1, start_node_names=[start_node], end_node_names=end_
↪nodes)
runner4.translate_tf_model(ckpt_path, model_name2, start_node_names=[start_node], end_node_names=end_
↪nodes)

runner3.join(runner4, join_action=JoinAction.AUTO_JOIN_INPUTS)
har_name = '{}.har'.format(runner3.model_name)
runner3.save_har(har_name)


# Join and merge the inputs by using the join_action_info dictionary to specify the input layers to␣
↪merge
# This functionally does the same as AUTO_JOIN_INPUTS, but is used here to demonstrate the different␣
↪syntax
runner3 = ClientRunner(hw_arch='hailo8')
runner4 = ClientRunner(hw_arch='hailo8')
runner3.translate_tf_model(ckpt_path, model_name1, start_node_names=[start_node], end_node_names=end_
↪nodes)
runner4.translate_tf_model(ckpt_path, model_name2, start_node_names=[start_node], end_node_names=end_
↪nodes)

join_info_dict = {model_name1 + '/input_layer1': model_name2 + '/input_layer1'}
runner3.join(runner4, join_action=JoinAction.CUSTOM, join_action_info=join_info_dict)
```


4.7.4. Compilation

In this step we will compile the network to Hailo-8 binary files (HEF).

For more details check the “Compilation Tutorial”.

```
[ ]: hef = runner1.compile()

# Single network group is required for inference without Scheduler
with open("single_network_group.alls", "w") as f:
    f.write("network_group0 = network_group([resnet_v1_18, mobilenet_ssd])")
runner1.load_model_script("single_network_group.alls")
hef = runner1.compile()

hef_path = '{}.hef'.format(runner1.model_name)
with open(hef_path, 'wb') as f:
    f.write(hef)
```

4.7.5. Inference

In this step we will introduce the inference process with different options.

For more details check the “Inference Tutorial” and the HailoRT guide.

```
[ ]: from hailo_platform import (PcieDevice, HEF, InferVStreams, InputVStreamParams, OutputVStreamParams,␣
↪FormatType,
                                 HailoStreamInterface, ConfigureParams, InputVStreams, OutputVStreams)

target = PcieDevice()

# Loading the compiled HEF to the device
hef = HEF(hef_path)

# Configure network groups
configure_params = ConfigureParams.create_from_hef(hef=hef, interface=HailoStreamInterface.PCIe)
network_groups = target.configure(hef, configure_params)
network_group = network_groups[0]
network_group_params = network_group.create_params()

# Create input and output virtual streams params
input_vstreams_params = InputVStreamParams.make(network_group, quantized=False, format_
↪type=FormatType.FLOAT32)
output_vstreams_params = OutputVStreamParams.make(network_group, quantized=True, format_
↪type=FormatType.UINT8)

# Define dataset params
num_of_images = 10
```

Standalone hardware deployment

```
[ ]: input_data = {layer.name: np.random.randn(num_of_images, *layer.shape).astype('float32')
              for layer in hef.get_input_vstream_infos()}

with InferVStreams(network_group, input_vstreams_params, output_vstreams_params) as infer_pipeline:
    with network_group.activate(network_group_params):
        result = infer_pipeline.infer(input_data)
        for output_layer_name, result in result.items():
            # The result output tensor is infer_results[output_vstream_info.name]
            print(f'Stream output shape in output {output_layer_name} is {result.shape}')
```


Standalone hardware deployment - Streaming inference

```
[ ]: from multiprocessing import Process

# Process for sending data
def send(configured_network, num_images):
    vstreams_params = InputVStreamParams.make(configured_network) # network_name argument can be used␣
↪to choose a specific network
    configured_network.wait_for_activation(1000)
    with InputVStreams(configured_network, vstreams_params) as vstreams:
        vstream_to_buffer = {vstream: np.ndarray([1] + list(vstream.shape), dtype=vstream.dtype) for␣
↪vstream in
                             vstreams}
        for _ in range(num_images):
            for vstream, buff in vstream_to_buffer.items():
                vstream.send(buff)

# Process for receiving data
data = None
def recv(configured_network, num_images):
    global data
    vstreams_params = OutputVStreamParams.make(configured_network) # network_name argument can be␣
↪used to choose a specific network
    configured_network.wait_for_activation(1000)
    with OutputVStreams(configured_network, vstreams_params) as vstreams:
        for _ in range(num_images):
            for vstream in vstreams:
                data = vstream.recv()

# Start the processes
send_process = Process(target=send, args=(network_group, num_of_images))
recv_process = Process(target=recv, args=(network_group, num_of_images))
recv_process.start()
send_process.start()
print(f'Starting streaming, num_of_images={num_of_images})')

# Wait for them to be done
with network_group.activate(network_group_params):
    send_process.join()
    recv_process.join()
print('Done')
```

Inference using TensorFlow

```
[ ]: import tensorflow as tf

graph = tf.Graph()
with graph.as_default():
    nodes = {input_layer.name: tf.compat.v1.placeholder(dtype=tf.float32)
             for input_layer in hef.get_input_vstream_infos()}
    sdk_export = runner1.get_tf_graph(target=target, nodes=nodes, translate_input=True, rescale_
↪output=True,
                                      use_preloaded_compilation=True, network_groups=network_groups)
    postprocess = [tf.argmax(output_tensor, axis=1) for output_tensor in sdk_export.output_tensors]

    with runner1.hef_infer_context(sdk_export):
        with sdk_export.session.as_default():
            feed_dict = {nodes[input_layer.name]:
                         np.random.randn(num_of_images, *input_layer.shape).astype('float32')
                         for input_layer in hef.get_input_vstream_infos()}
```

(continues on next page)

(continued from previous page)

```
            results = sdk_export.session.run(postprocess, feed_dict=feed_dict)

    for output_layer_name, result in zip(target.sorted_output_layer_names, results):
        print(f'Stream output shape in output {output_layer_name} is {result.shape}')
```


5. Building Models

5.1. Translating Tensorflow and ONNX models

Note: If you wish to disable the log files of the Dataflow Compiler, you could set the environment variable HAILO_CLIENT_LOGS_ENABLED to false .

5.1.1. Using the Tensorflow Parser

The Dataflow Compiler Tensorflow parser supports the following frameworks:

- Tensorflow v1.15.4, including Keras v2.2.4-tf.
- Tensorflow v2.5.2, including Keras v2.6.0.
- Tensorflow Lite v2.4.0.

The Parser translates the model to Hailo ARchive (.har) format. Hailo ARchive is a tar.gz archive file that captures the "state" of the model - the files and attributes used in a given stage from parsing to compilation.

The basic HAR file includes: - HN file, which is a JSON-like representation of the graph structure that is deployed to the Hailo hardware. - NPZ file, which includes the weights of the model. More files are added when the optimization and compilation stages are done.

Tensorflow models are translated to HAR by calling the `translate_tf_model()` method of the `ClientRunner` object. The nn_`framework` optional parameter tells the Parser whether it's a TF1 or TF2 model. The `start`_node_names and end_node_names optional parameters tell the Parser which parts to include/exclude from parsing. For example, the user may want to exclude certain parts of the post-processing and evaluation, so they won't be compiled to the Hailo device.

See also:

The *parsing tutorial* shows how to use this API.

The supported input formats are:

- TF1 models – checkpoints and frozen graphs (.pb). The Dataflow Compiler automatically distinguishes between them based on the file extension, but this decision can be overridden using the `is_frozen` flag.
- TF2 models – savedmodel format.
- TF Lite models – tflite format.

Supported Tensorflow APIs

Note: APIs that do not create new nodes in the TF graph (such as `tf.name_scope` and `tf.variable_scope`) are not listed because they do not require additional parser support.

Table 1. Supported Tensorflow APIs (layers)

API name	Comments
`tf.nn.conv2d`	
`tf.concat`	
`tf.matmul`	
`tf.avg_pool`	

Continued on next page

Table 1 – continued from previous page

API name	Comments
`tf.nn.maxpool2d`	
`tf.nn.depthwise_conv2d`	
`tf.nn.depthwise_conv2d_native`	
`tf.nn.conv2d_transpose`	• Only `SAME_TENSORFLOW` padding
`tf.reduce_max`	• Only on the features axis and with `keepdims=True`
`tf.reduce_mean`	
`tf.reduce_sum`	• Only with `keepdims=True`
`tf.contrib.layers.batch_norm`	
`tf.image.resize_images`	• See limitations on Supported layers / Resize
`tf.image.resize_bilinear`	• See limitations on Supported layers / Resize
`tf.image.resize_nearest_neighbor`	• See limitations on Supported layers / Resize
`tf.image.crop_to_bounding_box`	• Only static cropping, i.e. the coordinates cannot be data dependent
`tf.image.resize_with_crop_or_pad`	• Only static cropping without padding, i.e. the coordinates cannot be data dependent
`tf.nn.bias_add`	
`tf.add`	• Only one of the following: – Bias add – Elementwise addition layer – Const scalar addition – As a part of input tensors normalization
`tf.multiply`	• Only one of the following: – Elementwise multiplication layer – Const scalar multiplication – As a part of input tensors normalization
`tf.subtract`	• Only one of the following: – Elementwise subtraction layer – Const scalar subtraction – As a part of input tensors normalization
`tf.divide`	• Only one of the following: – Elementwise division layer – Const scalar division – As a part of input tensors normalization
`tf.negative`	
`tf.pad`	
`tf.reshape`	• Only in specific cases, for example: – Features to Columns Reshape layer – Between Conv and Dense layers (in both directions) – As a part of layers such as Feature Shuffle and Depth to Space
`tf.nn.dropout`	
`tf.depth_to_space`	
`tf.nn.softmax`	
`tf.argmax`	
`tf.split`	• Only in the features dimension

Continued on next page

Table 1 – continued from previous page

API name	Comments
`tf.slice`	• Only static cropping, i.e. the coordinates cannot be data dependent
Slicing (`tf.Tensor.__getitem__`)	• Only sequential slices (without skipping) • Only static cropping, i.e. the coordinates cannot be data dependent
`tf.nn.space_to_depth`	
`tf.math.square`	
`tf.math.pow`	• Only in specific case, pow(2) which is square
`tf.norm`	• Reduce L2, keepdims=True, axis = 1,2
`tf.math.l2_normalize`	• L2 Normalization

Table 2. Supported Tensorflow APIs (activations)

API name	Comments
`tf.nn.relu`	
`tf.nn.sigmoid`	
`tf.nn.leaky_relu`	
`tf.nn.elu`	
`tf.nn.gelu`	
`tf.nn.relu6`	
`tf.nn.silu`	
`tf.nn.softplus`	
`tf.nn.swish`	
`tf.exp`	
`tf.tanh`	
`tf.abs`	• Only as a part of the Delta activation parsing
`tf.sign`	• Only as a part of the Delta activation parsing
`tf.sqrt`	
`tf.math.log`	

Table 3. Supported Tensorflow APIs (others)

API name	Comments
`tf.Variable`	
`tf.constant`	
`tf.identity`	

Slim APIs

Note: APIs that do not create new nodes in the TF graph (such as `slim.arg_scope`) are not listed because they do not require additional parser support.

Table 4. Supported Slim APIs

API name	Comments
`slim.conv2d`	
`slim.batch_norm`	
`slim.max_pool2d`	
`slim.avg_pool2d`	
`slim.bias_add`	
`slim.fully_connected`	
`slim.separable_conv2d`	

Keras APIs

Table 5. Supported Keras APIs

API name	Comments
`layers.Conv1D`	
`layers.Conv2D`	
`layers.Conv2DTranspose`	
`layers.Dense`	
`layers.MaxPooling1D`	
`layers.MaxPooling2D`	
`layers.GlobalAveragePooling2D`	
`layers.GlobalMaxPooling2D`	
`layers.Activation`	
`layers.BatchNormalization`	Experimental support
`layers.ZeroPadding2D`	
`layers.Flatten`	Only to reshape Conv output into Dense input
`layers.add`	Only elementwise add after conv
`layers.concatenate`	
`layers.UpSampling2D`	Only `interpolation='nearest'`
`layers.Softmax`	
`layers.Reshape`	Only in specific cases such as Features to Columns Reshape and Dense to Conv Reshape
`layers.ReLU`	
`layers.PReLU`	
`layers.LeakyReLU`	
`activations.elu`	
`activations.exponential`	

Continued on next page

Table 5 – continued from previous page

API name	Comments
activations.gelu	
activations.hard_sigmoid	
activations.relu	
activations.sigmoid	
activations.softplus	
activations.swish	
activations.tanh	

Group conv parsing

Tensorflow v1.15.4 has no group conv operation. The Hailo Dataflow Compiler recognizes the following pattern and automatically converts it to a group conv layer:

- Several (>2) conv ops, which have the same input layer, input dimensions, and kernel dimensions.
- The features are equally sliced from the input layer into the convolutions.
- They should all be followed by the same concat op.
- Bias addition should be before the concat, after each conv op.
- Batch normalization and activation should be after the concat.

Feature shuffle parsing

Tensorflow v1.15.4 has no feature shuffle operation. The Hailo Dataflow Compiler recognizes the following pattern of sequential ops and automatically converts it to a feature shuffle layer:

- `tf.reshape` from 4-dim `[batch, height, width, features]` to 5-dim `[batch, height, width, groups, features in group]`.
- `tf.transpose` where the groups and features in group dimensions are switched. In other words, this op interleaves features from the different groups.
- `tf.reshape` back to the original 4-dim shape.

Code example:

```
reshape0 = tf.reshape(input_tensor, [1, 56, 56, 3, 20])
transpose = tf.transpose(reshape0, [0, 1, 2, 4, 3])
reshape1 = tf.reshape(transpose, [1, 56, 56, 60])
```

More details can be found in the Shufflenet paper (Zhang et al., 2017).

Squeeze and excitation block parsing

Squeeze and excitation block parsing is supported. An example Tensorflow snippet is shown below.

```
out_dim = 32
ratio = 4
conv1 = tf.keras.layers.Conv2D(out_dim, 1)(my_input)
x = tf.keras.layers.GlobalAveragePooling2D()(conv1)
x = tf.keras.layers.Dense(out_dim // ratio, activation='relu')(x)
x = tf.keras.layers.Dense(out_dim, activation='sigmoid')(x)
x = tf.reshape(x, [1, 1, 1, out_dim])
ew_mult = conv1 * x
```

Threshold activation parsing

The threshold activation can be parsed from:

```
tf.keras.activations.relu(input_tensor, threshold=threshold)
```

where `threshold` is the threshold to apply.

Delta activation parsing

The delta activation can be parsed from:

```
val * tf.sign(tf.abs(input_tensor))
```

where `val` can be any constant number.

5.1.2. Using the Tensorflow Lite Parser

Tensorflow Lite models are translated by calling the `translate_tf_model()` method of the `ClientRunner` object. No additional parameters needed.

Note: Hailo supports 32-bit/16-bit TFLite models, since our Model Optimization stage use the high precision weights to optimize the model for Hailo devices. Models that are already quantized to 8-bit are not supported.

See also:

For more info, and some useful examples on converting models from Tensorflow to Tensorflow-lite, refer to the *parsing tutorial*.

Supported Tensorflow Lite operations

Table 6. Supported TFLite operations (layers)

Operator name	Comments
ADD	
AVERAGE_POOL_2D	
CONCATENATION	
CONV_2D	
DEPTHWISE_CONV_2D	

Continued on next page

Table 6 – continued from previous page

Operator name	Comments
DEPTH_TO_SPACE	
DEQUANTIZE	• Only for parsing weight variables that are cast from float32 to float16
FULLY_CONNECTED	
L2_NORMALIZATION	
MAX_POOL_2D	
MUL	
RESHAPE	
RESIZE_BILINEAR	• See limitations on Supported layers / Resize
SOFTMAX	
SPACE_TO_DEPTH	
PAD	
GATHER	
TRANSPOSE	
MEAN	
SUB	
DIV	
SQUEEZE	
STRIDED_SLICE	
SPLIT	
CAST	
MAXIMUM	
ARG_MAX	
MINIMUM	
NEG	
PADV2	
SLICE	
TRANSPOSE_CONV	
EXPAND_DIMS	
SUM	
SHAPE	
POW	
REDUCE_MAX	
PACK	
UNPACK	
REDUCE_MIN	
SQUARE	
RESIZE_NEAREST_NEIGHBOR	• See limitations on Supported layers / Resize

Table 7. Supported TFLite operations (activations)

Operator name	Comments
LOGISTIC	
RELU	
RELU6	
TANH	
EXP	
PRELU	
LESS	
GREATER	• Only as a part of a Threshold activation parsing
EQUAL	
LOG	
SQRT	
LEAKY_RELU	
ELU	
HARD_SWISH	
ABS	• Only as a part of the Delta activation parsing

5.1.3. Using the ONNX Parser

ONNX models are translated by calling the `translate_onnx_model()` method of the `ClientRunner` object. The supported ONNX opsets are 8 and 11.

Supported ONNX operations

Table 8. Supported ONNX operations (layers)

Operator name	Comments
Add	• Only one of the following: – Bias add – Elementwise add – As a part of input tensors normalization – Const scalar addition
ArgMax	
AveragePool	
BatchNormalization	
Concat	
Conv	• Depthwise convolution is also implemented by this ONNX operation
ConvTranspose	
DepthToSpace	• Supported modes: * DCR: the default mode, equivalent to Tensorflow's DepthToSpace operator * CRD: reflects PyTorch's PixelShuffle operator

Continued on next page

Table 8 – continued from previous page

Operator name	Comments
Div	• Only one of the following: – Input tensors normalization – Const scalar division – Elementwise division
Dropout	
Flatten	• Only in specific cases such as between Conv and Dense layers
Gemm	
GlobalAveragePool	
MatMul	
MaxPool	
Mean	
Mul	• Only one of the following: – Elementwise Multiplication layer – Const scalar multiplication – As a part of input tensors normalization – As a prt of several activation functions, see below
Neg	
Pad	
ReduceMax	• Only on the features axis and with `keepdims=True`
ReduceMean	
ReduceSum	• Only with `keepdims=True`, or as a part of a Softmax layer
ReduceL2	• Only in specific cases, after rank4 tensors such as Conv (as oppose to rank2 such as Dense)
Reshape	• Only in specific cases, for example: – Depth to Space layer – Feature Shuffle layer – Features to Columns Reshape layer – Between Conv and Dense layers (in both directions)
Resize	• See limitations on Supported layers / Resize
Slice	
Softmax	
Split	• Only in the features dimension
Sub	• Only one of the following: – Input tensors normalization – Const scalar substraction – Elementwise substraction
Transpose	• Only in specific cases, for example: – Depth to Space layer – Feature Shuffle layer – Features to Columns Reshape layer – Dense like to Conv like Reshape layer
Upsample	• Only Nearest Neightbor resizing
Expand	• Only as broadcast before elementwise operations
InstanceNormalization	

Table 9. Supported ONNX operations (activations)

Operator name	Comments
Abs	• Only as a part of the Delta activation parsing
Clip	• Only as a part of a Relu6 activation parsing
Elu	
Erf	• Only as a part of a GeLU activation parsing
Exp	
Greater	• Only as a part of a Threshold activation parsing
HardSigmoid	
LeakyRelu	
Log	
Mul	• Only as a part of a Threshold or Delta activation parsing (and several non activation layers, see above)
PRelu	
Relu	
Sigmoid	
Sign	• Only as a part of the Delta activation parsing
Softplus	
Sqrt	
Tanh	

Exporting models from PyTorch to ONNX

The following example shows how to export a PyTorch model to ONNX, note the inline comments which explain each parameter in the export function.

Note: Before trying this small example, make sure Pytorch is installed in the environment.

```
# Building a simple PyTorch model
class SmallExample(torch.nn.Module):
   def __init__(self):
      super(SmallExample, self).__init__()
      self.conv1 = torch.nn.Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1))
      self.bn1 = torch.nn.BatchNorm2d(24)
      self.relu1 = torch.nn.ReLU6()

   def forward(self, x):
      x = self.conv1(x)
      x = self.bn1(x)
      x = self.relu1(x)
      return x

# Exporting the model to ONNX
torch_model = SmallExample()
torch_model.eval()
inp = [torch.randn((1, 96, 24, 24), requires_grad=False)]
torch_model(*inp)
onnx_path = 'small_example.onnx'

# Note the used args:
```

(continues on next page)

(continued from previous page)

```
# export_params makes sure the weight variables are part of the exported ONNX,
# training=TrainingMode.PRESERVE preserves layers and variables that get folded into other layers in␣
↪EVAL mode (inference),
# do_constant_folding is a recommendation by pytorch to prevent issues with PRESERVED mode,
# opset_version selects the desired ONNX implementation (currenly Hailo support opset versions from␣
↪11 and newer).
torch.onnx.export(torch_model, tuple(inp), onnx_path,
                  export_params=True,
                  training=torch.onnx.TrainingMode.PRESERVE,
                  do_constant_folding=False,
                  opset_version=13)
```

Supported PyTorch APIs

We have tested support on PyTorch version 1.9.0. Exporting Pytorch models to the ONNX format is done using the `torch.onnx.export` function.

Table 10. Supported PyTorch APIs (layers)

API name	Comments
`torch.nn.AvgPool2d`	
`torch.nn.BatchNorm1d`	
`torch.nn.BatchNorm2d`	
`torch.nn.Conv1d`	
`torch.nn.Conv2d`	
`torch.nn.ConvTranspose2d`	
`torch.nn.Dropout2d`	Does nothing on inference
`torch.nn.Flatten`	
`torch.nn.functional.interpolate`	• See limitations on Supported layers / Resize
`torch.nn.functional.pad`	
`torch.nn.InstanceNorm2d`	
`torch.nn.Linear`	
`torch.nn.MaxPool1d`	
`torch.nn.MaxPool2d`	
`torch.nn.Parameter`	
`torch.nn.PixelShuffle`	
`torch.nn.Softmax`	
`torch.nn.Softmax2d`	
`torch.nn.Upsample`	• See limitations on Supported layers / Resize
`torch.nn.UpsamplingBilinear2d`	• See limitations on Supported layers / Resize
`torch.nn.UpsamplingNearest2d`	• See limitations on Supported layers / Resize
`torch.argmax`	
`torch.cat`	
`torch.max`	See Supported ONNX operations: ReduceMax
`torch.mul`	See Supported ONNX operations: Mul
`torch.reshape`	See Supported ONNX operations: Reshape

Continued on next page

Table 10 – continued from previous page

API name	Comments
torch.split	See Supported ONNX operations: Split
torch.sum	See Supported ONNX operations: ReduceSum
torch.square	
torch.pow	Only in specific case, pow(2) which is square
torch.transpose	See Supported ONNX operations: Transpose

Table 11. Supported PyTorch APIs (activations)

API name	Comments
torch.abs	See Supported ONNX operations: Abs
torch.exp	
torch.gt	See Supported ONNX operations: Greater
torch.log	
torch.sign	See Supported ONNX operations: Sign
torch.sqrt	
torch.nn.ELU	
torch.nn.GELU	
torch.nn.Hardsigmoid	
torch.nn.Hardswish	
torch.nn.LeakyReLU	
torch.nn.Mish	
torch.nn.ReLU	
torch.nn.PReLU	
torch.nn.ReLU6	
torch.nn.Sigmoid	
torch.nn.SiLU	
torch.nn.Softplus	
torch.nn.Tanh	

5.1.4. Layer ordering limitations

This section describes the TF and ONNX parser limitations regarding ordering of layers.

- Bias – only before Conv, before DW Conv, after Conv, after DW Conv, after Deconv, or after Dense.

5.1.5. Supported padding schemes

The following *padding schemes* are supported in Conv, DW Conv, Max Pooling, and Average Pooling layers:

- `VALID`
- `SAME` (*symmetric padding*)
- `SAME_TENSORFLOW`

Other padding schemes are also supported, and will translate into *External Padding* layers.

5.1.6. NMS Post Processing

- **NMS** is a technique that is used to filter the predictions of object detectors, by selecting final entities (e.g., bounding box) out of many overlapping entities.
- The NMS algorithm needs to be fed with bounding boxes, which are calculated out of the network outputs.
- This process is called **"bbox decoding"**, and it consists of mathematically converting the network outputs to box coordinates.
- The bbox decoding calculations can vary greatly from one implementation to another, and include many types of math operations (pow, exp, log, and more).

Hailo supports the following NMS post processing algorithms on-chip:

1. SSD: bbox decoding + NMS
2. CenterNet: bbox decoding + score_threshold filtering
3. YOLOv5: bbox decoding + score_threshold filtering

For implementation on hailo devices:

1. When translating the network using the parser, should supply end_node_names parameter with the layers that come **before** the post-processing (bbox decoding) section. For Tensorflow models for example, it is performed using the API `translate_tf_model()` or the CLI tool: `hailo parser tf --end-node-names [list]`.

   Note: When `hailo` CLI tool is being used, the arguments are separated by spaces: `--end-node-names END_NODE1 END_NODE2 ..` and so on.

   ---

2. The post-processing has to be manually added to the translated (parsed) network using a Model Script command (*nms_postprocess*), which is fed to the `hailo optimize` CLI tool or to the python `optimize()` method. The command adds the relevant postprocess to the Hailo model, according to the architecture (e.g. SSD) and the configuration json file.

The output format of the on-chip post-process can be found on HailoRT guide:

- For Python API, look for *tf_nms_format* and see definitions of *Hailo format* and *TensorFlow format*.
- For CPP API, look for *HAILO_FORMAT_ORDER_HAILO_NMS*. It is similar to the *Hailo format* from the Python API.

SSD

SSD postprocessing consists of bbox decoding and NMS.

We support the specific SSD NMS implementation from TF Object Detection API SSD, tag v1.13.

We assume that the default configurations file is used.

We use the ssd_anchor_generator which uses the center of a pixel as the anchors centers (so anchors centers cannot be changed):

```
anchor_generator {
   ssd_anchor_generator {
   num_layers: 6
   min_scale: 0.2
   max_scale: 0.95
   aspect_ratios: 1.0
   aspect_ratios: 2.0
   aspect_ratios: 0.5
   aspect_ratios: 3.0
   aspect_ratios: 0.3333
   }
}
```

We assume that each branch ("box predictor") has its own anchors repeated on all pixels.

The bbox decoding function we currently support on chip can be found here (see def _decode which contains the mathematical transformation needed for extracting the bboxes).

For this NMS implementation, the end_nodes that come just-before the bbox decoding might be:

```
end_node_names =
[
   "BoxPredictor_0/BoxEncodingPredictor/BiasAdd",
   "BoxPredictor_0/ClassPredictor/BiasAdd",
   "BoxPredictor_1/BoxEncodingPredictor/BiasAdd",
   "BoxPredictor_1/ClassPredictor/BiasAdd",
   "BoxPredictor_2/BoxEncodingPredictor/BiasAdd",
   "BoxPredictor_2/ClassPredictor/BiasAdd",
   "BoxPredictor_3/BoxEncodingPredictor/BiasAdd",
   "BoxPredictor_3/ClassPredictor/BiasAdd",
   "BoxPredictor_4/BoxEncodingPredictor/BiasAdd",
   "BoxPredictor_4/ClassPredictor/BiasAdd",
   "BoxPredictor_5/BoxEncodingPredictor/BiasAdd",
   "BoxPredictor_5/ClassPredictor/BiasAdd"
]
```

An example for the corresponding SSD NMS JSON is found at: `sdk_client/hailo_sdk_client/tools/core_postprocess/nms_ssd_config_example_json_notes.txt`. This example file is not a valid JSON file since it has in-line comments.

CenterNet

CenterNet postprocessing consists of bbox decoding and then choosing the bboxes with the best scores.

Our CenterNet postprocessing corresponds to the `CenterNetDecoder` class on Gluon-CV (link). Therefore we support any CenterNet postprocessing which is equivalent in functionality to the above-mentioned code.

For this implementation, the end_nodes that come just-before the bbox decoding might be:

```
end_node_names =
[
   "threshold_confidence/threshold_activation/threshold_confidence/re_lu/Relu",
```

(continues on next page)

(continued from previous page)

```
    "CenterNet0_conv3/BiasAdd",
    "CenterNet0_conv5/BiasAdd"
]
```

An example for the corresponding CenterNet JSON is found at: `sdk_client/hailo_sdk_client/tools/core_postprocess/CenterNet_example_json_notes.txt`. This example file is not a valid JSON file since it has in-line comments.

YOLOv5

YOLOv5 postprocessing consists of bbox decoding and NMS. The NMS consists of two parts:

1. Filtering bboxes according to their detection score threshold ("low probability" boxes are filtered).
2. Filtering the remaining bboxes with IoU technique: selecting final entities (e.g., bounding box) out of many overlapping entities.

Hailo implemented the bbox decoding in-chip, as well as score threshold filtering. The IoU section needs to be implemented on host, but since score threshold filtering has been performed, the number of bboxes to deal with has decreased by an order of magnitude.

We have tested support for post-processing from the original implementation of YOLOv5, tag v2.0.

The anchors are taken from this file.

The bbox decoding function is described here, on the `Detect` class.

For this implementation, on YOLOv5m, the end_nodes that come just-before the bbox decoding might be:

```
end_node_names =
[
    "Conv_307",
    "Conv_286",
    "Conv_265"
]
```

An example for the corresponding YOLOv5 JSON is found at: `sdk_client/hailo_sdk_client/tools/core_postprocess/nms_yolov5_example_json_notes.txt`. This example file is not a valid JSON file since it has in-line comments.

5.2. Profiler and other command line tools

5.2.1. Using Hailo command line tools

The Hailo Dataflow Compiler offers several command line tools that can be executed from the Linux shell. Before using them, the virtual environment needs to be activated. This is explained in the *tutorials*.

To list the available tools, run:

```
hailo --help
```

The `--help` flag can also be used to display the help message for specific tools. The following example prints the help message of the Profiler:

```
hailo profiler --help
```

The command line tools cover major parts of the Dataflow Compiler's functionality, as an alternative to using the Python API directly:

Model conversion flow

- The `hailo parser` command line tool is used to translate ONNX / TF models into Hailo archive (HAR) files.

Note: Consult *Translating Tensorflow and ONNX models* and `hailo parser {tf, ckpt, tf2, onnx} --help` for further details on the according parser arguments.

- The `hailo optimize` command line tool is used to optimize models' performance.

Note: Consult *Model optimization* and `hailo optimize --help` for further details on quantization arguments.

- The `hailo compiler` command line tool is used to compile the models (in HAR format) into a hardware representation.

Note: Consult *Compilation* and `hailo compiler --help` for further details on compilation arguments.

Analysis and visualization

- The `hailo analyze` command is used to analyze per-layer quantization noise. Consult *Model optimization flow* for further details.
- The `hailo params-csv` command is used to to generate a CSV report with weights statistics, which is useful to analyze the quantization.
- The `hailo tb` command is used to convert HAR or CKPT files to Tensorboard.
- The `hailo visualizer` command is used to visualize HAR files.
- The `hailo har` command is used to extract information from HAR files.
- The `hailo har-modifier` command is used to apply modifications a Hailo archive file. Currently the supported transforms are *transpose*, *YUV2RGB*, and *resize*.

Tutorials

- The `hailo tutorial` command opens Jupyter with the tutorial notebooks folder. Select one of the tutorials to run.

5.2.2. Running the Profiler

The Profiler command line tool analyzes the expected performance of models on the hardware. To run the Profiler, use the following command:

```
hailo profiler --mode pre_placement network.har
```

The user has to set the path of the HAR file to profile. Additional optional parameters may be needed.

The user can also set one of the two running modes using the `--mode` command line option:

- `pre_placement` – Several resources calculation and optimization steps are performed without full allocation or compilation. It runs faster than the `post_placement` mode and does not require the model's weights.
- `post_placement` – The compiled model is analyzed, either by compiling on the fly or by inspecting an existing compilation (HEF). Simulation of the hardware micro-controllers is used to profile the expected performance. This mode is the most accurate, especially in terms of FPS.

Note: The FPS and latency of the whole model is not displayed with big models (which require multi-context), since the actual performance may depend on the host platform (mostly its PCIe generation and number of lanes).

The **Runtime profiler** is created for that purpose: When running with `post_placement` mode, the user can add `--runtime-data runtime_data.json` with a json generated by `hailortcli run <hef-path> collect-runtime-data` command on the target platform. See example at the bottom of Inference Tutorial.

Note: For single-context networks, the profiler report calculates the proposed FPS and latency of the whole model. However, on hosts with low PCIe bandwidth, it might not reflect actual performance. If the performance is worse than the profiler report values, it is recommended to try and *disable DDR buffers*.

5.2.3. Understanding the Profiler report

The Hailo profiler report consists of these parts:

- **Model Details** – The main attributes of the neural network.
- **Profiler Input Settings** – The target device and the required throughput.
- **Performance Summary** – A summary of the performance of the network on target hardware.
- **Device Utilization** – The percentage of the device(s) resources to be used by the target network.
- **Optimization** – Optimization Parameters and Model modifications that relate to the Model Optimization phase
- **Model Description** – The performance for each layer of the network.
- **Layer by Layer Analysis** – The breakdown of model's operations and weights, and power and memory breakdown on device.
- **Model Graph** – The model's graph, available for interactive navigation, and deep-dive into the separate layers.
- **Runtime** – Profiling information driven from an actual run on the target device.

The following sections describe all parts of the report and define the fields in each one.

Model Details

Model Name The model name (for example, Resnet18).

Input Tensors Shapes The resolution of the model's input image (for example, 224x224x3).

Output Tensors Shapes The resolution of the model's output shape (for example, 1x1x1000).

Operations per Input Tensor (OPS) Total operations per input image.

Operations per Input Tensor (MACS) Total multiply-accumulate operations per input image.

Model Parameters The number of model parameters (weights and biases), without any hardware-related overhead.

Profiler Input Settings

Target Device The name of the target device requested by the user.

Number of Devices The number of required devices estimated by the Dataflow Compiler.

Profiler Mode The mode used to run the Profiler. Currently, the supported modes are “pre-placement” and “post-placement”.

Optimization Goal The method used by the Dataflow Compiler to optimize the required hardware resources.

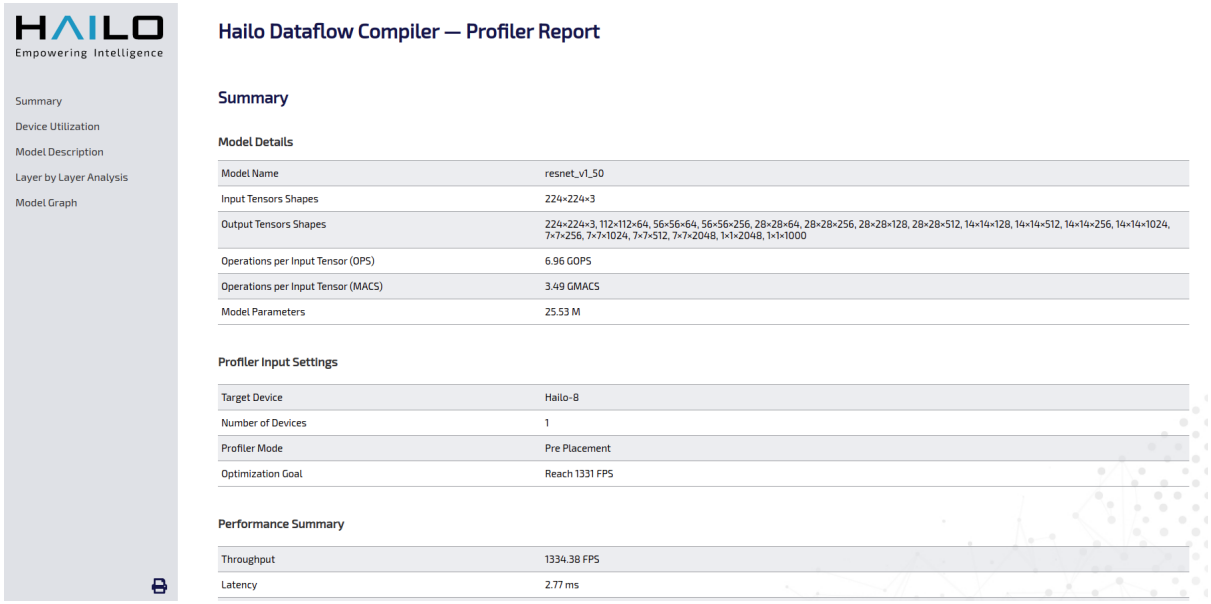

Figure 6. Profiler report screenshot

Performance Summary

Throughput (Bottleneck Layer) The overall network throughput limit as per the resources currently allocated by the Profiler.

Latency The number of milliseconds it takes the network to process an image.

Total NN Core Power Consumption The estimated power consumption of the neural core in Watts as expected at 25° C. This figure excludes power consumed by the chip top and interfaces.

Note: The power estimation is reported with accuracy of +/-20%.

Operations per Second (OP/s) The total operations per second, based on the throughput (FPS) required by the user.

Operations per Second (MAC/s) The total multiply-accumulate operations per second, based on the throughput (FPS) required by the user.

Input Interface Throughput The model’s total input tensor throughput (bytes per second), with (GROSS) and without (NET) any hardware-related overhead, based on the FPS rate required by the user.

Net Output Interface Throughput The model’s total output tensors throughput (bytes per second), with (GROSS) and without (NET) any hardware-related overhead, based on the FPS rate required by the user.

Gross Output Interface Throughput The model’s total output tensors throughput (bytes per second), with hardware-related overhead, based on the FPS rate required by the user.

Device Utilization

Note: This, and the following, sections, use terminology that is related to Hailo-8 devices NN core. A full description of the NN core architecture is not in the scope of this guide.

Compute Usage The percentage of the device(s) compute resources to be used by the target network. Can be expanded to view breakdown to sub-clusters (SCs), input aligners (IAs), and activation/pooling units (APUs), Hailo-8 components.

Memory Usage The percentage of the device(s) memory resources to be used by the target network. This figure includes both weights and intermediate results memory. Can be expanded to view breakdown to L2 (per sub-cluster), L3 (per-cluster), and L4 (device-level) memories.

Control Usage The percentage of the device(s) control resources to be used by the target network.

Note: See the *target device* field for the number of devices considered in calculating the device utilization details.

Optimization

Model Optimization

Optimization Level Complexity of the optimization algorithm that was used to quantize the model

Compression Level Amount of weight compression to 4-bit that was used

Ratio of Weights in 8bit Resulted percentage of 8-bit weights

Ratio of Weights in 4bit Resulted percentage of 4-bit weights

Calibration Set Size Calibration set size that was used to optimize the model

Model Modifications

Input Normalization Input normalization that was added to the model using a model script command

Non-Maximum Suppression NMS post-processing that was added to the model using a model script command

Transpose (H<->W) Was the model transposed on-chip using a model script command

Input Conversion Input color/format conversions that were added to the model using a model script command

Model Description

For each layer, the following fields are defined:

Layer Name The name of the layer, as defined in the HN/HAR.

Layer Type The type of operation performed by this layer (for example, convolution or max pooling).

Input [WxH] The shape of the image processed by this layer.

Kernel [WxH/S] The spatial dimensions of the kernel tensor and the stride (assuming symmetric stride).

Features [In Out] The number of input and output features (channels).

Groups The number of convolution groups. Convolution layers with more than one group are group convolution layers.

Dilation The kernel dilation (assuming symmetric dilation).

Weights [K] The number of layer parameters (weights and biases) in thousands, without any hardware-related overhead.

Ops [GMACs] The total GMAC operations per input image.

Power [mW] The expected power to be consumed by the hardware resources that run this layer.

Throughput [FPS] The maximum FPS for each layer as per the resources currently allocated by the Profiler. The minimum value of this field over all layers determines the overall network throughput limit with current hardware resources.

Note: This figure is expected to change when asking for a different FPS rate. This is because a different amount of hardware resources will be allocated for each layer.

Layer by Layer Analysis

Model Analysis The model's operations and weights breakdown. The tooltip above each layer displays the relative amount of operations or weights required for it, accordingly.

Device Performance Analysis The devices's power and L3 memory utilization breakdown. The tooltip above each layer displays the relative amount of power consumed by it, or L3 memory required for it, accordingly.

Model Graph

Graph representation of the model, allowing for scrolling, zooming in/out, and selection of separate layer to display their information. In case there are multiple contexts required by the model, a dropdown menu on the right allows selection of all, or separate, context/s.

Each layer can be selected to display the parameters related to it in a pop-up panel on the right. The parameters are broken down into four groups:

1. **Layer parameters** – Parameters related to the model's layer itself (regardless of Hailo implementation).
2. **Hailo performance parameters** – The selected layer's performance parameters on Hailo-8.
3. **Advanced** – Deeper dive into Hailo-8 component allocation and utilization parameters.
4. **Optimization** – Information related to optimization of this layer's inputs, weights and activations

The description of all parameters is given below.

Layer parameters

Layer Name The name of the layer, as defined in the HN/HAR file.

Layer Type The type of operation performed by this layer (for example, convolution or max pooling).

Input Size [Height, Width, nFeatures] The shape of the input image/tensor processed by this layer.

Output Size [Height, Width, nFeatures] The shape of the output image/tensor of this layer.

Kernel [Height, Width, Depth] The spatial dimensions of the kernel tensor.

Strides [Height, Width] The 2D stride steps of the kernel.

Dilation [Height, Width] The kernel's dilation.

Number of Parameters How many parameters this layer has

Hailo performance parameters

Num Context [out of X] The selected context, in case the model requires multi-context allocation.

FPS The frames per second processed by the layer.

Power [mWatt] The expected power to be consumed by the hardware resources that run this layer.

Layer Input BW [kB] The layer's input bandwidth in kB.

Layer Output BW [kB] The layer's output bandwidth in kB.

Total number of Cycles The number of clock cycles for processing, required by the layer.

MAC operations The number of multiply-accumulate operations, required by the layer.

Note: In relation to the below memory-related parameters, see *Memory Usage* section above, for overall model's memory utilization breakdown.

Main (L3 - Cluster) Memory Utilization [%] The relative amount of L3 (cluster-level) memory required by the layer.

L2 (Subcluster) Memory Utilization [%] The relative amount of L2 (subcluster level) memory required by the layer.

L4 Memory Utilization [%] The relative amount of L4 (device-level) required by the layer.

Advanced

HailoMAC Computation Utilization [%] The relative amount of Hailo-8's MAC units utilized by the layer.

Subclusters The number of subclusters (SCs) required by the layer.

Subcluster Utilization [%] The relative amount of subclusters utilized by the layer.

Activation Pooling Units The number of Hailo-8's activation and pooling units (APUs) required by the layer.

Activation Pooling Units Activation Utilization [%] The relative amount of APUs utilized by the layer.

Number of inter-layer buffers (after the layer) The number of buffers, after the selected layer, required by it.

Buffer Size The buffer size required by the layer.

Optimization

Ranges [Input, Kernel, Output] Ranges of values

Bits [Input, Weights, Bias, Output] Actual bits used to represent the data

Runtime Data

Latency breakdown Timeline showing the different inference sections, during initial configuration ("Preliminary") and full inference of each context.

Each context consists of four phases:

- Config time – time required to fetch weights and configurations over the PCIe bus
- Load time – some of the fetched data needs to be prepared and loaded into the resources of the device
- Overhead time – initializing / finalizing the resources before / after the inference
- Infer time – the time we wait for all of the neural network layers that are allocated to the chip to complete processing the batch

Percentage The percentage each context takes out of the full network inference, as well as the inference and reconfiguration percentages inside each context.

Bandwidth A graph describing the PCIe bandwidth utilized by each context of the network.

5.3. Model optimization

Translating the models' parameters numerically, from floating point to integer representation, is also known as quantization (or model optimization). This is a mandatory step in order to run models on the Hailo hardware. This step takes place after translating the model from its original framework and before compiling it. For optimized performance, we recommend using a machine with a GPU when running the model optimization and prepare a calibration data with at least 1024 entries.

5.3.1. Model optimization workflow

The model optimization has two main steps: full precision optimization and quantization optimization.

Full precision optimization includes any changes to the model in the floating-point precision domain, for example, Equalization [Meller2019], TSE [Vosco2021] and pruning. Quantization includes compressing the model from floating point to integer representation of the weights and activations (4/8-bits) and algorithms to improve the model's accuracy, such as IBC [Finkelstein2019], AdaRound [Nagel2020] and QFT [McKinstry2019]. Both steps may degrade the model accuracy, therefore, evaluation is needed to verify the model accuracy.

This workflow is depicted in the following diagram.

1. First stage includes full precision validation. This stage is important for making sure parsing was successful and we built the pre/post processing and evaluation of the model correctly. For example, properly using the preprocessing and postprocessing with the model:

```
import tensorflow as tf
from hailo_sdk_client import ClientRunner
from hailo_sdk_common.targets.inference_targets import SdkNative

runner = ClientRunner(har_path=model_path)
with tf.compat.v1.Session() as sess:
    ph = tf.compat.v1.placeholder(tf.float32)
    sdk_graph = runner.get_tf_graph(SdkNative(), ph, custom_session=sess)
    output = sess.run(sdk_graph.output_tensors, feed_dict={ph: input})
```

2. Next, we call the model optimization API to generate an optimized model. To obtain best performance we recommend using a GPU machine and a dataset with at least 1024 entries for calibration. Calibration data is used to gather activation statistics in the inputs/outputs of each layer. This statistic is being used to map the floating-point values into their integer representation, (a.k.a quantization). Use high quality calibration data (that represents well the validation dataset) is crucial to obtain good accuracy. Supported calibration data types are: Numpy array with shape: [BxHxWxC], NPY file of a Numpy array with shape: [BxHxWxC], directory of Numpy files with each shape: [HxWxC] and *tf.data.Dataset* object with expected return value of: [{layer_name: input}, _].

3. Finally, we need to verify the accuracy of the optimized model to validate the process was successful. In case of large degradation (that doesn't meet the accuracy requirement), we can re-try optimization with increased optimization level. **Optimization and Compression levels** allows you to control the model optimization effort and the model memory footprint. For quick iterations we recommend starting with the default setting of the model optimizer (compression_level=1, optimization_level=1). However, when moving to production, we recommended to work at the highest optimization level (optimization_level=3) to achieve optimal accuracy. With regards to compression, users should increase it when the overall throughput/latency of the application is not satisfying. Note that increasing compression would have negligible effect on power-consumption so the motivation to work with higher compression level is mainly due to FPS considerations. To verify the accuracy of the quantized model, we recommend using the Hailo Emulator, for example:


```
import tensorflow as tf
from hailo_sdk_client import ClientRunner
from hailo_sdk_common.targets.inference_targets import SdkPartialNumeric

runner = ClientRunner(har_path=model_path)
with tf.compat.v1.Session() as sess:
    ph = tf.compat.v1.placeholder(tf.float32)
    sdk_graph = runner.get_tf_graph(SdkPartialNumeric(), ph, custom_session=sess)
    output = sess.run(sdk_graph.output_tensors, feed_dict={ph: input})
```

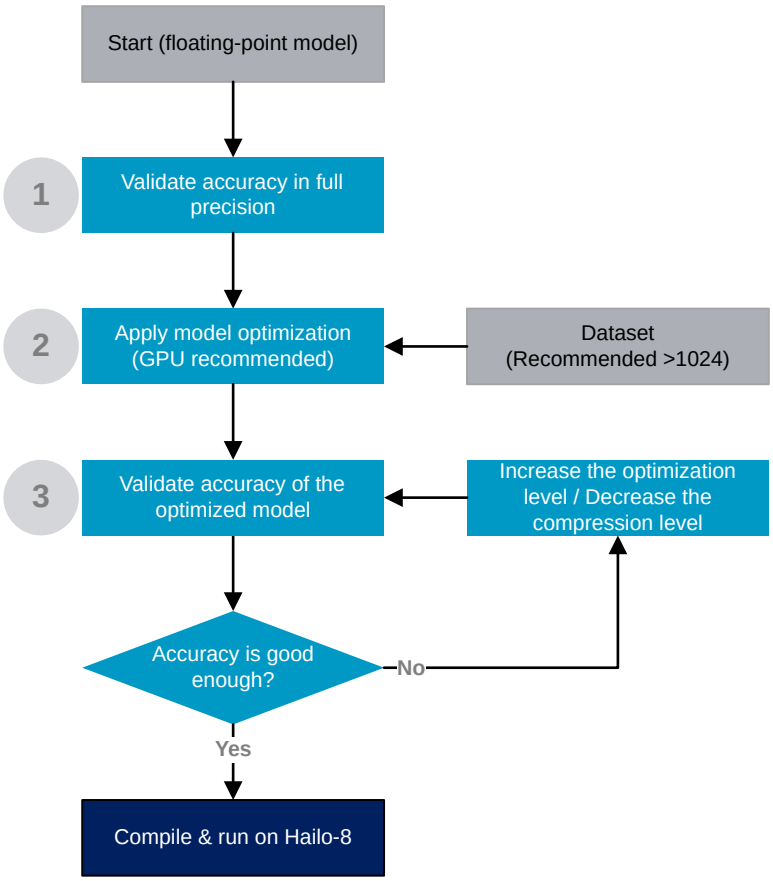

Figure 7. Detailed block diagram of suggested model optimization flow

Usage

The `optimize()` method serves as the model optimization API. This API requires sample dataset (typically >= 1024), which is used to collect statistics. After the statistics are collected, they are used to quantize the weights and activations, that is, map the floating point values into integer representation. Hailo's quantization scheme uses uniformly distributed bins and optimizes for the best trade-off between range and precision.

Before calling the *optimize()* API, you might call `load_model_script()` to load a model script (.alls file) that includes commands that modify the model, affect the basic quantization flow and additional algorithms to improve the accuracy and optimize the running time.

To control the optimization grade, we recommend setting the optimization_level argument through the *model_optimization_flavor* command, which gets values of 0-3 and control which quantization algorithms will be enabled. Using higher optimization level means the model optimization tool will use more advanced algorithms which expected to get better accuracy but will take longer to run. Note that optimization levels 1-3 require at least 1024

images to run. The default setting is Optimization_level=1 unless GPU is not available, or the dataset is not large enough (less than 1024). For refernce, those are the expected running times for optimizing ResNet-v1-50 with compression_level=4 using Nvidia A4000 GPU:

- optimization_level=0: 95s
- optimization_level=1: 496s
- optimization_level=2: 2178s
- optimization_level=3: 6211s

To control the compression degree, we use the compression_level argument through the *model_optimization_flavor* command, which gets values of 0-5 and control the percentage of weights that are quantized to 4-bits (default is using 8-bit precision for weights quantization). Using higher compression level means the compression will be more aggressive and accuracy may be degraded. To recover the accuracy loss, we recommend using higher optimization level as well. High compression rate improves the fps especially for large networks (more than 20M parameters) or when used in a pipeline. The default setting is Compression_level=1.

Note: The algorithms that compose each optimization level are expected to change in future versions.

Next, we present the results of applying different choices of optimization/compression levels on common CV models.

Table 12. Model optimization flavor example showing the RegNetX-800MF model degradation with different configurations of the optimization and compression levels. The degradation reported is the Top-1 accuracy degradation over the 50k validation set of ImageNet-1K.

	Optimization Level - 0	Optimization Level - 1	Optimization Level - 2	Optimization Level - 3
Compression Level - 0	0.41	1.9	-	0.18
Compression Level - 1	1.9	0.47	0.46	0.32
Compression Level - 4	13.87	3.14	1.24	1.24

Table 13. Model optimization flavor example showing the YOLOv5m model degradation with different configurations of the optimization and compression levels. The degradation reported is the mAP accuracy degradation over the 5k validation set of COCO2017.

	Optimization Level - 0	Optimization Level - 1	Optimization Level - 2	Optimization Level - 3
Compression Level - 0	2.23	1.61	-	1.07
Compression Level - 1	2.56	2.03	1.8	0.99
Compression Level - 4	8.48	4.8	2.88	2.1

Table 14. Model optimization flavor example showing the DeepLab-v3-MobileNet-v2 model degradation with different configurations of the optimization and compression levels. The degradation reported is the mIoU accuracy degradation over the validation set of PASCAL-VOC.

	Optimization Level - 0	Optimization Level - 1	Optimization Level - 2
Compression Level - 0	2.95	1.17	-
Compression Level - 1	6.72	1.0	0.73
Compression Level - 4	16.0	2.44	1.5

Debugging

If the quantization accuracy is not satisfying, the following methods should be used (after each step, validate the accuracy of your model):

1. Make sure you are using at least 1024 images in your calibration dataset and machine with a GPU.
2. If you have used **compression_level**, lower its value (the default is the minimum value: 0). For example, use the following command in the model script: *model_optimization_flavor(compression_level=1)*
3. Configure higher **optimization_level** in the model script, for example: *model_optimization_flavor(optimization_level=3)*
4. Configure 16bit output using the following model script command: *quantization_param(output_layer1, precision_mode=a16_w16)*
5. Try to run with different quantization algorithms. Specifically, try the following commands: *post_quantization_optimization(finetune, policy=disabled)*, and *post_quantization_optimization(bias_correction, policy=enabled)*
6. Try to run with activation clipping using the following model script commands: *model_optimization_config(calibration, calibset_size=512)*, and *pre_quantization_optimization(activation_clipping, layers={*}, mode=percentile, clipping_values=[0.01, 99.99])*
7. Use more data and longer optimization process, for example: *post_quantization_optimization(finetune, policy=enabled, learning_rate=0.0001, epochs=8, dataset_size=4000)*
8. Use different loss type, for example: *post_quantization_optimization(finetune, policy=enabled, learning_rate=0.0001, epochs=8, dataset_size=4000, loss_types=[l2, l2, l2, l2])*

See also:

The *model optimization tutorial* shows how to use the optimization API and the optimization/compression levels.

5.3.2. Model scripts

Note: Model scripts can be used to control and configure each of the model processing steps. This section shows the Model script commands that are relevant to the Optimization process. For other commands that can be used, see: *Allocation-related commands*.

Note: Allocation related commands are ignored during model optimization. Model optimization related commands are only validated during allocation. This is to make sure that they do not contradict the existing already quantized model. Model optimization related commands should appear first in the script before any allocation related commands. Note that some commands effect both quantization and allocation, for example, using 4-bit weights.

The model script is loaded before running the model optimization by using the `load_model_script()`.

The model script supports model modification commands, which are processed on `optimize()`:

1. *model_modification_commands*

In addition, the model script supports 5 quantization commands:

1. *model_optimization_flavor*

2. *model_optimization_config*
3. *quantization_param*
4. *pre_quantization_optimization*
5. *post_quantization_optimization*

model_modification_commands

The model script supports 5 model modification commands:

- *input_conversion*
- *resize_input*
- *transpose*
- *normalization*
- *nms_postprocess*
- *change_output_activation*

input_conversion

Adds on-chip conversion of the input tensor.

The conversion could be either a **color conversion**:

- yuv_to_rgb - which is implemented by the following kernel: [[1.164, 1.164, 1.164], [0, -0.392, 2.017], [1.596, -0.813, 0]] and bias [-222.912, 135.616, -276.8] terms. Corresponds to cv::COLOR_YUV2RGB in OpenCV terminology.
- bgr_to_rgb - which transposes between the R and B channels using an inverse identity matrix as kernel, no bias. Corresponds to cv2.cvtColor(src, code) where src is a BGR image, and code is cv2.COLOR_BGR2RGB.

Or a **format conversion**:

- Any of the formats described on *Format conversion*, for example yuy2_to_hailo_yuv.

```
rgb_layer = input_conversion(input_layer1, yuv_to_rgb)
rgb_layer1, rgb_layer2, ... = input_conversion(yuv_to_rgb)  # number of return values should match␣
↪the number of inputs of the network
yuv_layer = input_conversion(input_layer2, yuy2_to_hailo_yuv)
```

resize_input

Performs on-chip resize to the input tensor(s). The resize method used is bilinear interpolation with align_corners=True and half_pixels=False. The input resize limitations are those of resize bilinear *as described here*. When the resize ratio is high, the compilation process will be more difficult, as more on-chip memories and sub-clusters are required.

```
resize_input1 = resize_input([256,256], input1)
resize_input1, resize_input2, ... = resize_input([256,256])  # resize all inputs; return value should␣
↪match the number of inputs of the network
```


transpose

Transposes the whole connected component(s) of the chosen input layer(s), so the network runs transposed on chip (helps performance in some cases). HailoRT is taking care of transposing the inputs and outputs on the host side.

```
transpose(input_layer1)  # transposing the connected components corresponding to the input layers␣
↪specified
transpose()  # transposing all layers and weights
```

normalization

Adds on-chip normalization to the input tensor(s).

```
norm_layer1 = normalization(mean_array, std_array, input_layer)  # adding normalization layer with␣
↪the parameters mean & std after the specified input layer. Multiple commands can be used to apply␣
↪different normalization to each input layer.
norm_layer1, norm_layer2, ... = normalization(mean_array, std_array)  # adding normalization layers␣
↪after all input layers. Return value should match the number of inputs in the network
```

nms_postprocess

For more information about on-chip NMS post-processing, refer to *nms_post_processing*.

```
nms_postprocess('nms_config_file.json', ssd)  # example for adding SSD NMS with config file,␣
↪architecture is written without ''.
```

There are a few options for using this command. Note that in each option, the architecture name must be provided, using meta_arch argument.

1. Specify only the architecture name. In this case a default config file will be used, auto-detecting the layers that come before the post-process.

   For example: nms_postprocess(meta_arch=ssd)

2. Specify the architecture name and some of the config arguments. In this case, a default config file will be used, edited by the arguments provided

   and auto-detecting the layers that come before the post-process. The config arguments that can be set via the command are: nms_scores_th, nms_iou_th, image_dims, classes.

   For example: nms_postprocess(meta_arch=yolo, image_dims=[512, 512], classes=70)

3. Specify the config json path in addition to architecture name. In this case the file provided will be used.

   Please note that when providing the config path, you shouldn't provide any of the config argument using the command, only inside the file.

   For example: nms_postprocess('config_file_path', centernet)

The default config files can be found at `hailo_sdk_client/tools/core_postprocess`:

- `default_nms_config_yolov5.json`
- `default_nms_config_centernet.json`
- `default_nms_config_ssd.json`

For available architectures see `MetaArchitectures`.

change_output_activation

Changes output layer activation. See the *supported activations* section for activation types.

```
change_output_activation(output_layer, activation) # changing activation function of specified output␣
↪layer.
change_output_activation(activation) # changing activation function of all the output layers.
```

model_optimization_flavor

Configure the model optimization effort by setting compression level and optimization level. The flavor's algorithm will behave as default, any algorithm specific configuration will override the flavor's default config

Default values:

- compression_level: 1
- optimization_level: 1 with gpu and 1024 images, 0 otherwise
- batch_size: check default of each algorithm (usually 8 or 32)

Example commands:

```
model_optimization_flavor(optimization_level=3)
model_optimization_flavor(compression_level=2)
model_optimization_flavor(optimization_level=2, compression_level=1)
model_optimization_flavor(optimization_level=1, batch_size=4)
```

Parameters:

Parameter	Values	Default	Required	Description
optimization_level	int; 0<=x<=3	(Read command doc)	False	Optimization level, higher is better but longer, improves accuracy
batch_size	int; 1<=x	(Read command doc)	False	Batch size for the algorithms (adaround, finetune, calibration)
compression_level	int; 0<=x<=5	(Read command doc)	False	Compression level, higher is better but increases degradation, improves fps and latency

model_optimization_config

- *compression_params*
- *calibration*

compression_params

This command automatically reduces some layers' precision mode to a8_w4. The values (between 0 and 1 inclusive) represents how much of the total weights memory usage you want to optimize to 4bit. When the value is 1, all the weights will be set to 4bit, when 0, the weights won't be modified

Example command:

```
# Optimize 30% of the total weights to use 4bit mode
model_optimization_config(compression_params, auto_4bit_weights_ratio=0.3)
```

Note: If you manually set some layers' precision_mode using quantization_param, the optimization will take it into account, and won't set any weight back to 8bit

---

Parameters:

Parameter	Values	Default	Required	Description
auto_4bit_weights_ratio	float; 0<=x<=1	0	False	Set a ratio of the model's weights to reduce to 4bit

calibration

During the quantization process, the model will be inferred with small dataset for calibration purposes. The calibration can be configured here. (This replaces the calib_num_batch and batch_size arguments in `quantize()` API)

Example command:

```
model_optimization_config(calibration, batch_size=4, calibset_size=128)
```

Parameters:

Parameter	Values	Default	Required	Description
batch_size	int; 0<x	8	False	Batch size used during the calibration inference
calibset_size	int; 0<x	64	False	Data items used during the calibration inference

quantization_param

The syntax of each quantization_param command in the script is as follows:

```
quantization_param(<layer>, <parameter>=<value>)
```

For example

```
quantization_param(conv1, bias_mode=double_scale_initialization)
```

Multiple parameters can be assigned at once, by simply adding more parameter-value couples, for example:

```
quantization_param(conv1, bias_mode=double_scale_initialization, precision_mode=a8_w4)
```

Multiple layers can be assigned at once when using a list of layers:


```
quantization_param([conv1, conv2], bias_mode=double_scale_initialization, precision_mode=a8_w4)
```

Glob syntax is also supported to change multiple layers at the same time. For example, to change all layers whose name starts with conv, we could use:

```
quantization_param({conv*}, bias_mode=double_scale_initialization)
```

The available parameters are:

1. *bias_mode*
2. *precision_mode*
3. *null_channels_cutoff_factor*
4. *max_elementwise_feed_repeat*
5. *max_bias_feed_repeat*
6. *quantization_groups*

Additionally, there are some deprecated params detailed here *Deprecated params*

bias_mode

Sets the layer's bias behavior, there are 2 available bias modes. The modes are:

1. `single_scale_decomposition` when set, the bias is represented by 3 values: `UINT8*INT8*UINT4`.
2. `double_scale_initialization` when set, the layer use 16-bit to represent the bias weight of the layer

Some layers are 16-bit by default (for example, Depthwise), while others are not. Switching a layer to 16-bit can have a slightly adverse effect on allocation while improving quantization. If a network exhibits degradation due to quantization, it is strongly recommended to set this parameter for all layers with biases.

All layers that have weights and biases support the `double_scale_initialization` mode.

Example command:

```
quantization_param(conv3, bias_mode=double_scale_initialization)
```

Changed in version 2.8: This parameter was named `use_16bit_bias`. This name is now deprecated.

Changed in version 3.3: `double_scale_initialization` is now the default bias mode for multiple layers.

precision_mode

Precision mode sets the bits available for the layers' weights and activation representation. There are currently 2 available precision modes: a8_w8, a8_w4

- a8_w8 which means 8-bit activations and 8-bit weights. (This is the default)
- a8_w4 which means 8-bit activations and 4-bit weights. Can be used to reduce memory consumption.
- a16_w16 16 bit activations and weights (currently supported only on output layers), might improve results in some models. Supported with the following last layers:
  - Fully connected (dense)
  - Depthwise (dw)
  - Elementwise add (ew_add)
  - Convolution (conv)

Example command:


```
quantization_param(conv3, precision_mode=a8_w4)
```

Note: It is recommended to use *Finetune* when using 4-bit weights

null_channels_cutoff_factor

This command is applicable only for layers with fused batch normalization. The default value is 1e-4.

This is used to zero-out the weights of the so called "dead-channels". These are channels whose variance is below a certain threshold. The low variance is usually a result of the activation function eliminating the results of the layer (for example, a ReLU activation that zeros negative inputs). The weights are zeroed out to avoid outliers that shift the dynamic range of the quantization but do not contribute to the results of the network. The variance threshold is defined by `null channels_cutoff_factor * bn_epsilon`, where `bn_epsilon` is the epsilon from the fused batch normalization of this layer.

Example command:

```
quantization_param(conv4, null_channels_cutoff_factor=1e-2)
```

max_elementwise_feed_repeat

This command is applicable only for conv-and-add layers. The range is 1-4 (integer only) and the default value is 4.

This parameter determines the precision of the elements in the "add" input of the conv-and-add. A lower number will result in higher throughput at the cost of reduced precision. For networks with many conv-and-add operations, it is recommended to switch this parameter to 1 for all conv-and-add layers, in order to see if higher throughput can be achieved. If this results in high quantization degradation, the source of the degradation should be examined and this parameter should be increased for that layer.

Example command:

```
quantization_param(conv5, max_elementwise_feed_repeat=1)
```

max_bias_feed_repeat

The range is 1-32 (integer only) and the default value is 32.

This parameter determines the precision of the biases. A lower number will result in higher throughput at the cost of reduced precision. This parameter can be switched to 1 for all or some layers, in order to see if higher throughput can be achieved. If this results in high quantization degradation, the source of the degradation should be examined and this parameter should be increased for that layer.

This parameter is not applicable for layers that use the `double_scale_initialization` bias mode.

Example command:

```
quantization_param(conv5, max_bias_feed_repeat=1)
```


quantization_groups

The range is 1-4 (integer only) and the default value is 1.

This parameter allows splitting weights of a layer into groups and quantizing each separately for greater accuracy. When using this command, the weights of layers with more than one quantization group are automatically sorted to improve accuracy.

Using more than one group is supported only by Conv and Dense layers (not by Depthwise or Deconv layers). In addition, it is not supported by the last layer of the model (or last layers if there are multiple outputs).

Example command:

```
quantization_param(conv1, quantization_groups=4)
```

pre_quantization_optimization

All the features of this command optimize the model before the quantization process. Some of these commands modify the model structure, and occur before the rest of the commands.

The algorithms are triggered in the following order:

- *dead_channels_removal*
- *se_optimization*
- *equalization*
- *equalization per-layer*
- *weights_clipping*
- *activation_clipping*

dead_channels_removal

Dead channels removal is channel pruning, which removes from the model any layer with null weights and activation output. This might reduce the memory consumption and improve inference time

Example commands:

```
# This will enable the algorithm
pre_quantization_optimization(dead_channels_removal, policy=enabled)
```

Note: This operation will modify the structure of the model's graph

---

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	disabled	True	Enable or disable the dead channels removal algorithm

se_optimization

This feature can modify the Squeeze and Excite block to run more efficiently on the Hailo chip. A more detailed explanation of the TSE algorithm can be found here https://arxiv.org/pdf/2107.02145.pdf

Example commands:

```
# Apply TSE to the first 3 S&E blocks with tile height of 7
pre_quantization_optimization(se_optimization, method=tse, mode=sequential, count=3, tile_height=7)

# Apply TSE to the first 3 S&E blocks with tile height of 9 to the 1st block, 7 to the 2nd and 5 to␣
↪the 3rd
pre_quantization_optimization(se_optimization, method=tse, mode=sequential, count=3, tile_height=[9,␣
↪7, 5])

# Apply TSE to S&E blocks the start with avgpool1 and avgpool2 layers, with tile height of 7, 5␣
↪accordingly
pre_quantization_optimization(se_optimization, method=tse, mode=custom, layers=[avgpool1, avgpool2],␣
↪tile_height=[7, 5])
```

Note: This operation will modify the structure of the model's graph

The tile height has to divide the avgpool height evenly

Note: An in-depth explanation of the TSE algorithm - https://arxiv.org/pdf/2107.02145.pdf

Parameters:

Parameter	Values	Default	Required	Description
method	{tse}	tse	True	Algorithm for Squeeze and Excite block optimization
mode	{sequential, custom, disabled}	disabled	True	How to apply the algorithm on the model
layers	List of {str}	None	False	Required when mode=custom. Set which SE blocks to optimize based on the global avpool of the block
count	int; 0<x	None	False	Required when mode=sequential. Set how many SE blocks to optimize
tile_height	(int; 0<x) or (List of {int; 0<x})	7	False	Set tile height for the TSE. When list is given, it should match the layers count / the count argument. The tile has to devide the height without residue

equalization

This sub-command allows to configure the global equalization behavior during the pre quantization process, this command replaces the old equalize parameter from `quantize()` API

Example command:

```
pre_quantization_optimization(equalization, policy=disabled)
```

Note: An in-depth explanation of the equalization algorithm - https://arxiv.org/pdf/1902.01917.pdf

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	enabled	False	Enable or disable the equalization algorithm

equalization per-layer

This sub-command allows to configure the equalization behavior per layer. Allowed policy mean the behavior derives from the algorithm config

Example commands:

```
# Disable equalization on conv1 and conv2
pre_quantization_optimization(equalization, layers=[conv1, conv2], policy=disabled)

# Disable equalization on all conv layers.
pre_quantization_optimization(equalization, layers={conv*}, policy=disabled)
```

Note:

- Not all layers support equalization
- Layers are related to other
- Disabling 1 layer, disables all related layers
- Enabling 1 layer won't enable the related layers (it has to be done manually)

Parameters:

Parameter	Values	Default	Required	Description
policy	{allowed, enabled, disabled}	None	False	Set equalization behavior to given layer. (default is allowed)

weights_clipping

This command allows changing this behavior for selected layers and applying weights clipping when running the quantization API. This command may be useful in order to decrease quantization related degradation in case of outlier weight values. It is only applicable to the layers that have weights.

- `disabled` mode doesn't take clipping values, and disables any weights clipping mode previously set to the layer
- `manual` mode uses the clipping values as given.
- `percentile` mode calculates layer-wise percentiles (clipping values are percentiles 0 to 100).
- `mmse` mode doesn't take clipping values, and uses *Minimum Mean Square Estimators* to clip the weights of the layer
- `mmse_if4b` similar to mmse, when the layer uses 4bit weights, and disables clipping when it uses 8bit weights. (This is the default)

Example commands:

```
pre_quantization_optimization(weights_clipping, layers=[conv2], mode=manual, clipping_values=[-0.1, 0.
↪8])
pre_quantization_optimization(weights_clipping, layers=[conv3], mode=percentile, clipping_values=[1.0,
↪ 99.0])
pre_quantization_optimization(weights_clipping, layers={conv*}, mode=mmse)
pre_quantization_optimization(weights_clipping, layers=[conv3, conv4], mode=mmse_if4b)
pre_quantization_optimization(weights_clipping, layers={conv*}, mode=disabled)
```

Note: The dynamic range of the weights is symmetric even if the clipping values are not symmetric.

Parameters:

Parameter	Values	Default	Required	Description
mode	{disabled, manual, percentile, mmse, mmse_if4b}	mmse_if4b	True	Mode of operation, described above
clipping_values	[float, float]	None	False	Clip value, required when mode is percentile or manual

activation_clipping

By default, the model optimization does not clip layers' activations during quantization. This command can be used to change this behavior for selected layers and apply activation clipping when running the quantization API. This command may be useful in order to decrease quantization related degradation in case of outlier activation values.

- `disabled` mode doesn't take clipping values, and disables any activation clipping mode previously set to the layer (This is the default)
- `manual` mode uses the clipping values as given.
- `percentile` mode calculates layer-wise percentiles (clipping values are percentiles 0 to 100).

Note: Percentiles based activation clipping requires several iterations of statistics collection, so quantization might take a longer time to finish.

Example commands:


```
pre_quantization_optimization(activation_clipping, layers=[conv1], mode=manual, clipping_values=[0.
↪188, 1.3332])
pre_quantization_optimization(activation_clipping, layers=[conv1, conv2], mode=percentile, clipping_
↪values=[0.5, 99.5])
pre_quantization_optimization(activation_clipping, layers={conv*}, mode=disabled)
```

Parameters:

Parameter	Values	Default	Required	Description
mode	{disabled, manual, percentile, percentile_force}	disabled	True	Mode of operation, described above
clipping_values	[float, float]	None	False	Clip value, required when mode is percentile or manual

post_quantization_optimization

All the features of this command optimize the model after the quantization process.

```
post_quantization_optimization(<feature>, <**kwargs>)
```

Note: Bias correction and Finetune can't be used together

The features of this command are:

- *bias_correction*
- *bias_correction per-layer*
- *finetune*
- *adaround*
- *adaround per-layer*

bias_correction

This sub-command allows to configure the global bias correction behavior during the post quantization process, this command replaces the old ibc parameter from `quantize()` API

Example command:

```
# This will enable the IBC during the post quantization
post_quantization_optimization(bias_correction, policy=enabled)
```

Note: An in-depth explanation of the IBC algorithm - https://arxiv.org/pdf/1906.03193.pdf

Note: Bias correction is recommended when the model contains small kernels or depth-wise layers

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	disabled	False	Enable or disable the bias correction algorithm

bias_correction per-layer

This sub-command allows to enable or disable the Iterative Bias Correction (IBC) algorithm on a per-layer basis. Allowed policy mean the behavior derives from the algorithm config

Example commands:

```
# This will enable IBC for a specific layer
post_quantization_optimization(bias_correction, layers=[conv1], policy=enabled)

# This will disable IBC for conv layers and enable for the other layers
post_quantization_optimization(bias_correction, policy=enabled)
post_quantization_optimization(bias_correction, layers={conv*}, policy=disable)
```

Parameters:

Parameter	Values	Default	Required	Description
policy	{allowed, enabled, disabled}	None	False	Set bias correction behavior to given layer. (default is allowed)

finetune

This sub-command enabled knowledge distillation based fine-tuning of the quantized graph.

Example commands:

```
# enable fine-tune with default configuration
post_quantization_optimization(finetune)

# enable fine-tune with a larger dataset
post_quantization_optimization(finetune, dataset_size=4096)
```

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	disabled	True	Enable or disable finetune training
dataset_size	int; 0<x	1024	False	Number of images used for training; Exception is thrown if the supplied calibration set data stream falls short of that.
batch_size	int; 0<x	None	False	Uses the calibration batch_size by default. Number of images used together in each training step; driven by GPU memory constraints (may need to be reduced to meet them) but also by the algorithmic impact opposite to that of learning_rate.
epochs	int; 0<=x	4	False	Epochs of training
learning_rate	float	None	False	The base learning rate used for the schedule calculation (e.g., starting point for the decay). default value is *0.0002 / 8 * batch_size*.Main parameter to experiment with; start from small values for architectures substantially different from well-performing zoo examples, to ensure convergence.
def_loss_type	{ce, l2, l2rel, cosine}	l2rel	False	The default loss type to use if `loss_types` is not given
loss_layer_names	List of {str}	None	False	Names of layers to be used for teacher-student losses. Names to be given in Hailo HN notation, s.a. *conv20*, *fc1*, etc. Default: the output nodes of the net (the part described by the HN)
loss_types	List of {{ce, l2, l2rel, cosine}}	None	False	(same length as *loss_layer_names*) The teacher-student bivariate loss function types to apply on the native and numeric outputs of the respective loss layers specified byloss_layer_names. For example, ce (standing for 'cross-entropy') is typically used for the classification head(s). Default: the `def_loss_type`
loss_factors	List of {float}	None	False	(same length as *loss_layer_names*) defined bivariate functions on native/numeric tensors produced by respective loss_layer_names , to arrive at the total loss. Default to 1 for all members.
native_layers	List of {str}	[]	False	Don't quantize given layers during training
val_images	int; 0<=x	4096	False	Number of held-up/validation images for evaluation between epochs.
val_batch_size	int; 0<=x	128	False	Batch size for the inter-epoch validation.
Optimizer	{adam, sgd, momentum, rmsprop}	adam	False	set to 'sgd' to use simple Momentum, otherwise Adam will be used.

Advanced parameters:

Parameter	Values	Default	Required	Description
optimizer	{adam, sgd, momentum, rmsprop}	adam	False	set to 'sgd' to use simple Momentum, otherwise Adam will be used.
layers_to_freeze	List of {str}	[]	False	Freeze (don't modify weights&biases for) any layer whose name includes one of this list as substring. As such, this arg can be used to freeze whole layer types/groups (e.g. pass "conv" to freeze all convolutional).
lr_schedule_type	{cosine_restarts, exponential, constant}	cosine_restarts	False	functional form of the learning rate decay within "decay period" - cosine decay to zero (default), exponential smooth or staircase
decay_rate	float	0.5	False	decay factor of the learning rate at a beginning of "decay period", from one to the next one. In default case of cosine restarts, the factor of the rate to which learning rate is restarted next time vs. the previous time.
decay_epochs	int; 0<=x	1	False	duration of the "decay period" in epochs. In the default case of cosine restarts, rate decays to zero (with cosine functional form) across this period, to be then restarted for the next period.
warmup_epochs	int; 0<=x	1	False	duration of warmup period, in epochs, applied before the starting the main schedule (e.g. cosine-restarts).
warmup_lr	float	None	False	constant learning rate to be applied during the warmup period. Defaults to 1/4 the base learning rate.
bias_only	bool	False	False	train only biases (freeze weights).

adaround

Adaround algorithm optimizes layers' quantization by training the rounding of the kernel layer-by-layer

Example commands:

```
post_quantization_optimization(adaround, policy=enabled)
```

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	disabled	False	Enable or disable the adaround algorithm
batch_size	int; 0<x	32	False	batch size of the ada round algorithm
dataset_size	int; 0<x	1024	False	Data samples for adaptive round algorithm
epochs	int; 0<x	320	False	Number of train epochs
warmup	float; 0<=x<=1	0.2	False	Ratio of warmup epochs out of epochs
weight	float; 0<x	0.01	False	Round regulizer weight
train_bias	bool	True	False	Whether to train bias as well or not (will apply bias correction if layer is not trained)
bias_correction_count	int	64	False	Data count for bias correction
mode	{train_4bit, train_all}	train_4bit	False	default train behavior

Advanced parameters:

Parameter	Values	Default	Required	Description
b_range	[float, float]	[20, 2]	False	Max, min for temperature decay
decay_start	float; 0<=x<=1	0	False	Ratio of round train without round regulairzation decay (b)

adaround per-layer

This sub commands allow to toggle layers in the adaround algorithm individually

Example commands:

```
# This will enable IBC for a specific layer
post_quantization_optimization(adaround, layers=[conv1], policy=disabled)
post_quantization_optimization(adaround, layers=[conv17, conv18], policy=enabled)
```

Parameters:

Parameter	Values	Default	Required	Description
policy	{allowed, enabled, disabled}	allowed	False	None

Deprecated params

Some parameters in quantization_param command will be deprecated in the future and trigger deprecation warning when used. Each of the deprecated params have an alternative and it will be detailed below. These params are:

1. *use_16bit_bias*
2. *use_4bit_weights*
3. *equalization*
4. *bias_correction*
5. *activation_clipping_mode* (with `activation_clipping_values`)
6. *weights_clipping_mode* (with `weights_clipping_values`)

use_16bit_bias

The old usage was `quantization_param(conv1, use_16bit_bias=<bool>)` to toggle between True (`double_scale_initialization`) and False (`single_scale_decmoposition`)

The alternative is to use `bias_mode` param as described in *Bias mode*

use_4bit_weights

The old usage was `quantization_param(conv1, use_4bit_weights=<bool>)` to toggle between True (`a8_w4`) and False (`a8_w8`)

The alternative is to use `precision_mode` param as described in *Precision mode*

equalization

The old usage was `quantization_param(conv1, equalization=<policy>)` to toggle between 3 stats `enabled`, `disabled`, and `allowed`

The alternative is to use `pre_quantization_optimization` command as described in *Layer equalization*

bias_correction

The old usage was `quantization_param(conv1, bias_correction=<policy>)` to toggle between 3 stats `enabled`, `disabled`, and `allowed`

The alternative is to use `post_quantization_optimization` command as described in *Layer bias correction*

activation_clipping_mode

The old usage was `quantization_param(conv1, activation_clipping_mode=<mode>, activation_clipping_values=<values>)` to set clipping mode (percentile or manual) with given values

The alternative is to use `pre_quantization_optimization` command as described in *Activation clipping*

weights_clipping_mode

The old usage was `quantization_param(conv1, weights_clipping_mode=<mode>, weights_clipping_values=<values>)` to set clipping mode (percentile or manual) with given values

The alternative is to use `pre_quantization_optimization` command as described in *Weights clipping*

5.4. Models compilation

5.4.1. Basic compilation flow

For inference using TAPPAS or with native HailoRT API

calling `compile()` compiles the model without loading it to the device returning a binary that contains the compiled model, a HEF file.

Note: The default compilation target is Hailo-8. To compile for different architecture (Hailo-8R for example), use `hw_arch='hailo8r'` as a parameter to the translation phase. For example see the tutorial referenced on the next note.

See also:

The *compilation tutorial* shows how to use the `compile()` API.

For inference using ONNX Runtime

After compiling a model, as described in the previous section, that originated from an ONNX model you may choose to extract a new ONNX model that contains the entire network in the original model, with the nodes segmented by the start and end note arguments, replaced by the compiled HEF, by calling `get_hailo_runtime_model()` . This is required if you wish to run inference using OnnxRT with HailoRT.

You can also use the CLI: *hailo har-onnx-rt COMPILED-HAR-FILE* .

This feature is currently in preview, with the following limitations:

- The validated Opsets are 8 and 11.
- The model needs to be dividable to three sections:
  - Pre-processing, which connects only to the Main model
  - Main model, which connects only to the Post-processing
  - Post-processing
- The start_nodes will completely seperate the pre-processing from the Main model No connections (including "Shape" Ops) from the pre-processing is allowed into the main model, unless they are marked as start_nodes
- The end_nodes need to seperate the main model from the post-processing completely (including *Shape* Ops)

`get_hailo_runtime_model()` returns an ONNX model, that you can either pass directly to an ONNXRT session, or first save to a file and then load unto a session.

```
hef = runner.compile() # the returned HEF is not needed when working with ONNXRT
onnx_model = runner.get_hailo_runtime_model() # only possiable on a compiled model
onnx_file = onnx.save(onnx_model , onnx_file_path) # save model to file
```

See also:

The parsing tutorial shows how to load a network from an existing model and setting the start and end note arguments.

More information on using OnnxRT with HailoRT is available here.

Changed in version 3.9: Added *context switch* support using an allocation script command. The context switch mechanism allows to run a big model by automatically switching between several contexts that together constitute the full model.

For inference with Python using TensorFlow

Calling `get_tf_graph()` wraps the process of compiling the model and loading it to the device.

The object that represents the hardware is created first, and then this object is passed to `get_tf_graph()`.

5.4.2. Model scripts

Note: Model scripts can be used to control and configure each of the model processing steps. This section shows the Model script commands that are relevant to the Allocation process. For other commands that can be used, see: *Optimization-related commands*.

Note: Allocation related commands are ignored during model optimization. Model optimization related commands are only validated during allocation. This is to make sure that they don't contradict the existing already optimized model.

The Allocation-related model script commands hint the Dataflow Compiler regarding the model's resources allocation for the Hailo device. They are needed for some advanced cases, especially when trying to reach high throughput or high utilization of the device's resources.

Note: This section uses terminology that is related to Hailo devices NN core. A full description of the NN core architecture is not in the scope of this guide.

Usage

The script is a separate file which can be given to the `get_tf_graph()` and `compile()` methods of the `ClientRunner` class using the `allocator_script_filename` parameter.

For example:

```
compiled_model = client_runner.compile(allocator_script_filename='x.alls')
```

The Allocator script is a text file that should contain one or more of the commands described below.

Context switch parameters

Definition

```
context_switch_param(param=value)
```

Example

```
context_switch_param(mode=enabled, max_utilization=0.3)
```

Description This command modifies context switch policy and sets several parameters related to it:

- mode – Context switch mode. Set to enabled to enable context switch: Automatic partition of the given model to several contexts will be applied. Set to disabled to disable context switch. Set to allowed to let the compiler decide if multi context is required. Defaults to allowed.
- max_utilization – max_utilization is a number between 0.0 and 1.0. It is a threshold for automatic context partition. Model will be partitioned when any of the resources on-chip (control, compute, memory) exceeds the given threshold. Defaults to 0.3.

Allocator parameters

Definition

```
allocator_param(param=value)
```

Example

```
allocator_param(automatic_ddr=False)
```

Description This sets several allocation parameters described below:

- timeout – Compilation timeout for the whole run. By default, the timeout is calculated dynamically based on the model size. The timeout is in seconds by default. Can be given a postfix of 's', 'm', or 'h' for seconds, minutes or hours respectively. e.g. timeout=3h will result to 3 hours.
- automatic_ddr – when enabled, DDR portals that buffer data in the host's RAM over PCIe are added automatically when required. DDR portals are added when the data needed to be buffered on some network edge exceeds a threshold. In addition, DDR portal is added only when there are enough resources on-chip to accommodate it. Defaults to True. Set to False to ensure the HEF compatibility to platforms that don't support it, such as Ethernet based platforms.
- automatic_reshapes – When allowed, Format Conversion (Reshape) layers might be added to networks boundary inputs and outputs. They will be added when supported, and when we have enough resources on-chip to accommodate these functions. When enabled, Format Conversion layers will be added to all supported inputs and outputs without checking that there are enough resources on-chip. When disabled, format conversion layers won't be added to boundary inputs and outputs on chip. Defaults to allowed.
- merge_min_layer_utilization – Threshold of minimum utilization of the 'control' resource, to start the layer auto merger. Auto-merger will try to optimize on-chip implementation by sharing resources between layers, to reach this control threshold. Auto-merger will not fail if target utilization cannot be reached.

Resource Calculation Flow parameters

Definition

```
resources_param(param=value)
```

Example

```
resources_param(strategy=greedy, max_control_utilization=0.9, max_compute_
↪utilization=0.8)
context0.resources_param(max_utilization=0.25)
```

Description This sets several resources calculation flow parameters described below.

- strategy – Resources calculation strategy. When set to greedy, adding more resources to the slowest layers iteratively (Maximum FPS search), to reach the highest possible network FPS (per context). legacy is to be deprecated. Defaults to greedy.
- max_control_utilization – Number between 0.0 and 1.2. Threshold for greedy strategy. Maximum-FPS search will be stopped when the overall control resources on-chip exceeds the given threshold (per context). Defaults to 0.75.

- `max_compute_utilization` – Number between 0.0 and 1.0. Threshold for `greedy` strategy. Maximum-FPS search will be stopped when the overall compute resources on-chip exceeds the given threshold (per context). Defaults to `0.75`.
- `max_memory_utilization` – Number between 0.0 and 1.0. Threshold for `greedy` strategy. Maximum-FPS search will be stopped when the overall weights-memory resources on-chip exceeds the given threshold. Defaults to `0.75`.
- `max_utilization` – Number between 0.0 and 1.0. Threshold for `greedy` strategy. Maximum-FPS search will be stopped when on-chip utilization of any resource (control, compute, memory) exceeds the given threshold. The parameter overrides default thresholds but not the user provided thresholds specified above.

Two formats are supported – the first one affects all contexts, and the second one only affects the chosen context (see example #2).

Place

Definition

```
place(cluster_number, layers)
```

Example

```
place(2, [layer, layer2])
```

Description This points the allocator to place layers in a specific `cluster_number`. Layers which are not included in any `place` command, will be assigned to a cluster by the Allocator automatically.

Shortcut

Definition

```
shortcut(layer_from, layers_to)
```

Examples

```
shortcut1 = shortcut(conv1, conv2)
shortcut2 = shortcut(conv5, [batch_norm2, batch_norm3])
```

Description This command adds a shortcut layer between directly connected layers. The `layers_to` parameter can be a single layer or a list of layers. The shortcut layer copies its input to its output.

Portal

Definition

```
portal(layer_from, layer_to)
```

Example

```
portal1 = portal(conv1, conv2)
```

Description This command adds a portal layer between two directly connected layers. When two layers are connected using a portal, the data from the source layer leaves the cluster before it gets back in and reaches the target layer. The main use case for this command is to solve edge cases when two layers are manually placed in the same cluster. When two layers are in different clusters, there is no need to manually add a portal between them.

L4 Portal

Definition

```
l4_portal(layer_from, layer_to)
```

Example

```
portal1 = l4_portal(conv1, conv2)
```

Description This command adds a L4-portal layer between two directly connected layers. This command is essentially the same as `portal`, with the key difference that the data will be buffered in L4 memory, as opposed to a regular `portal` which buffers the data in L3 memory. The main use case for this command is when a large amount of data needs to be buffered between two endpoints, and we want this data to be buffered in another memory hierarchy.

DDR Portal

Definition

```
ddr(layer_from, layer_to)
```

Example

```
ddr1 = ddr(conv1, conv2)
```

Description This command adds a DDR portal layer between two directly connected layers. This command is essentially the same as `portal`, with the key difference that the data will be buffered in the host, as opposed to a regular `portal` which buffers the data in on-chip memory. Note that this command is supported only in HEF compilations and will work only on supported platforms (i.e. when using the PCIe interface).

Concatenation

Definition

```
concat(layers_from, layer_to)
```

Example

```
concat0 = concat([conv7, conv8], concat1)
```

Description Add a concat layer between several input layers and an output layer. This command is used to split a "large" concat layer into several steps (For example, three concat layers with two inputs instead of a single concat layer with four inputs).

Note: For now this command only supports two input layers (in the argument `layers_from`).

De-fuse

Definition

```
defuse(layer, defuse_number, defuse_type)
```

Examples

```
maxpool1_1, maxpool1_2, maxpool1_c = defuse(maxpool1, 2) # Defuse by output features
conv4a, conv4b, conv4c, conv4concat = defuse(conv4, 3, defuse_type=SPATIAL) # Defuse by output␣
↪columns
dw10_fs, dw10_d0, dw10_d1, dw10_dc = defuse(dw10, 2, defuse_type=INPUT_FEATURES) # Defuse by␣
↪input features
maxpool11_fs, maxpool11_d0, maxpool11_d1, maxpool11_dc = defuse(maxpool11, 2, defuse_type=INPUT_
↪FEATURES) # Defuse by input features
```

Description Defusing splits a logical layer into multiple physical layers in order to increase performance. This command orders the Allocator to defuse the given layer to defuse_number physical layers that share the same job, plus an additional concat layer merges all outputs together (and an input feature splitter in case of feature splitter). Like most mechanisms, the defuse mechanism happens automatically, so no user intervention is required.

Several types of defuse are supported, the most common are:

- Feature defuse: Each physical layer calculates part of the output features. Supported layers: Conv, Deconv, Maxpool, Depthwise conv, Avgpool, Dense, Bilinear resize, NN resize.
- Spatial defuse: Each physical layer calculates part of the output columns. Supported layers: Conv, Deconv, Depthwise conv, Avgpool, Argmax, NN resize.
- Input features defuse: Each physical layer receives a part of the input features. Supported layers: Maxpool, Depthwise conv, Avgpool, NN resize, Bilinear resize.

For Feature defuse, don't use the defuse_type argument (see examples).

Compilation parameters

Definition

```
compilation_param(layer, param=value)
```

Example

```
compilation_param(conv1_d0, resources_allocation_strategy=manual_scs_selection, number_of_
↪subclusters=8, use_16x4_sc=enabled)
```

Description This will update the given layer's compilation param. The command in the example sets the number of subclusters of a specific layer to 8. In addition, it forces 16x4 mode, which means that each subcluster handles 16 columns and 4 output features at once. This is instead of the default of 8 and 8 respectively.

Supported compilation params:

- `resources_allocation_strategy` – defaults to `min_l3_mem_match_fps`, which chooses the the number of subclusters that saves most L3 memory (Conv layers only). Change to `min_scs_match_fps` in order to choose the lowest possible number of subclusters. Change to `manual_scs_selection` to manually choose the number of subclusters (Conv, Dense and DW layers only).
- `use_16x4_sc` – can use 16 pixels multiplication by 4 features – instead of the default 8 pixels by 8 features. This is useful when the number of features is smaller than 8. A table of supported layers is given below (layers that are not mentioned are not supported).
- `no_contexts` – change to True in order to accumulate all the needed inputs for each output row computation in the L3 memory. A table of supported layers is given below (layers that are not mentioned are not supported).

- `balance_output_multisplit` – change to False in order to allow unbalanced output buffers. This can be used to save memory when there are "long" skip connections between layers.
- `number_of_subclusters` – force the usage of a specific number of subclusters. Make sure the `resource allocation strategy` value is set to `manual_scs_selection`. This is only applicable to Conv and Dense layers.
- `fps` – force a layer to reach this throughput, possibly higher than the FPS used for the rest of the model. This parameter is useful to reduce the model's latency, however it is not likely to contribute to the model's throughput which is dominated by the bottleneck layer.

Glob syntax is supported to change many layers at once. For example:

```
compilation_param({conv*}, resources_allocation_strategy=min_scs_match_fps)
```

will change the resources allocation strategy of all the layers that start with conv.

Table 15. 16x4 mode support

Kernel type	Kernel size (HxW)	Stride (HxW)	Dilation (HxW)	Padding
Conv	1x1	1x1	1x1	SAME SAME_TENSORFLOW VALID
Conv	3x3	1x1, 2x1	1x1 2x2 (stride=1x1 only) 3x3 (stride=1x1 only) 4x4 (stride=1x1 only)	SAME SAME_TENSORFLOW
Conv	5x5	1x1, 2x1	1x1	SAME SAME_TENSORFLOW
Conv	7x7	1x1, 1x2, 2x1, 2x2	1x1	SAME SAME_TENSORFLOW
Conv	1x3, 1x5, 1x7	1x1	1x1	SAME SAME_TENSORFLOW
Conv	3x5, 3x7, 5x3, 5x7, 7x3, 7x5	1x1	1x1	SAME SAME_TENSORFLOW
Conv	3x4, 5x4, 7x4, 9x4	1x1	1x1	SAME SAME_TENSORFLOW
Conv	3x6, 5x6, 7x6, 9x6	1x1	1x1	SAME SAME_TENSORFLOW
Conv	3x8, 5x8, 7x8, 9x8	1x1	1x1	SAME SAME_TENSORFLOW
Conv	9x9	1x1	1x1	SAME SAME_TENSORFLOW
DW	3x3	1x1	1x1, 2x2	SAME SAME_TENSORFLOW
DW	5x5	1x1	1x1	SAME SAME_TENSORFLOW

Table 16. No contexts mode support

Kernel type	Kernel size (HxW)	Stride (HxW)	Dilation (HxW)
Conv	3x3	1x1, 1x2, 2x1, 2x2	1x1
Conv	7x7	2x2	1x1

HEF parameters

Definition

```
hef_param(should_use_sequencer=value, params_load_time_compression=value)
```

Example

```
hef_param(should_use_sequencer=True, params_load_time_compression=True)
```

Description This will configure the HEF build. The command in the example enables the use of Sequencer and weights compression for optimized device configuration.

Supported hef parameters:

- `should_use_sequencer` – Using the Sequencer allows faster configurations load to device over PCIe during network activation, but removes Ethernet support for the created HEF. It defaults to `True`.
- `params_load_time_compression` – defaults to `True` and enables compressing layers parameters (weights) in the HEF for allowing faster load to device during network activation. Note that load time compression doesn't reduce the required memory space. This parameter also removes Ethernet support for the created HEF when enabled.

Outputs multiplexing

Definition

```
output_mux(layers)
```

Example

```
output_mux1 = output_mux([conv7, fc1_d3])
```

Description The outputs of the given layers will be multiplexed into a single tensor before sending them back from the device to the host. Contrary to concat layers, output mux inputs do not have to share the same width, height, or numerical scale.

From TF

Definition

```
layer = from_tf(original_name)
```

Example

```
my_conv = from_tf('conv1/BiasAdd')
```

Description This command allows the use of the original (TF/ONNX) layer name in order to make sure that the correct layers are addressed, as the HN layers names and the original layers names differ.

Note: Despite its name, this commands supports original names from both TF and ONNX.

Buffers

Definition

```
buffers(layer_from, layer_to, number_of_rows_to_buffer)
buffers(layer_from, layer_to, number_of_rows_cluster_a, number_of_rows_cluster_b)
```

Example

```
buffers(conv1, conv2, 26)
```

Description This command sets the size of the inter-layer buffer in units of `layer_from`'s output rows. Two variants are supported. The first variant sets the total number of rows to buffer. The second variant sets two such buffer sizes, in case the compiler adds a cluster transition between these layers. The first size sets the number of rows to buffer before the cluster transition, and the second number sets the number of rows after the transition. If there is no cluster transition, only the first number is used. The second variant is mainly used in autogenerated scripts returned by `save_autogen_allocation_script()`.

Feature splitter

Definition

```
feature_splitter(layer_from, layers_to)
```

Example

```
aux_feature_splitter0 = feature_splitter(feature_splitter0, [conv0, conv1])
```

Description Add a feature splitter layer between an existing feature splitter layer and some of its outputs. This command is used to break up a "large" feature splitter layer with many outputs into several steps.

Format conversion

Definition

```
format_conversion(layer_from, layers_to, format_conversion_type)
format_conversion(layer_from, format_conversion_type)
```

Example

```
reshape1 = format_conversion(input_layer1, conv1, tf_rgb_to_hailo_rgb)
reshape_yuy2 = format_conversion(input_layer1, yuy2_to_hailo_yuv)
```

Description Add a format conversion layer. This command is useful to offload host operations to the Hailo device. The supported format conversions are:

- `tf_rgb_to_hailo_rgb` – Converts an NHWC tensor ("TF RGB") to an NHCW tensor ("Hailo RGB"). NHCW is the format used by the Hailo core for most layers, such as Conv and Maxpool. This is useful before the first layer to offload this conversion from the host (in other words, from HailoRT). Despite its name, this conversion can be applied even if the number of features is not 3.
- `hailo_rgb_to_tf_rgb` – The inverse transformation of `tf_rgb_to_hailo_rgb`. This is useful after the last layer to offload this conversion from the host. Despite its name, this conversion can be applied even if the number of features is not 3.
- `yuy2_to_hailo_yuv` – Converts the YUY2 format, which is used by some cameras, to YUV. This is useful together with the YUV to RGB layer (see `add_yuv_to_rgb_layers()`) to create a full vision pipeline YUY2 → YUV → RGB. Corresponds to cv::COLOR_YUV2RGB_YUY2 in OpenCV terminology.
- `tf_rgbx_to_hailo_rgb` - Converts RGBX to Hailo RGB format.

When the command is used without the `layers_to` argument, the new layer is added between `layer_from` and all its successors.

Cascade of portals / buffers

Definition

```
created_layers_list = cascade(layer_from, layer_to, types=[type_0, type_1, ..], extra_arg)
```

Examples

```
some_name_1, some_name_2, some_name_3 = cascade(input_layer, conv1, types=[shortcut, portal,␣
↪shortcut]) # similar to equal_weights
some_name_1, some_name_2, some_name_3 = cascade(input_layer, conv1, types=[shortcut, portal,␣
↪shortcut], equal_weights) # similar to previous
shortcut10, shortcut11, shortcut12, ddr13 = cascade(conv1, conv2, types=[shortcut, shortcut,␣
↪shortcut, ddr], total_buffers=30)
portal20, shortcut21, portal22, shortcut23 = cascade(conv2, conv3, types=[portal, shortcut,␣
↪portal, shortcut], buffers=[10,5,7,2,0])
shortcut30, shortcut31, portal32, portal33 = cascade(shortcut23, conv3, types=[shortcut,␣
↪shortcut, portal, l4_portal], weights=[3,1,5,6,3])
```

Description Add a cascade of N buffers and portals between the source and destination layer. The supported types of the cascade are:

- shortcut
- portal
- l4_portal
- ddr

Supported are four types of arguments:

- buffers=[N+1 comma separated integers]: Append the requested buffers to [layer_from, first_cascade_layer, .., last_cascade_layer]. Possible values are integers, with 0 means no assignment (use the automatically calculated value).
- total_buffers=value: Each layer in [layer_from, first_cascade_layer, .., last_cascade_layer] gets (value / N+1) buffers.
- weights=[N+1 comma separated double values]: Automatically calculates the number of buffers, and tries to divide them between the cascade layers according to the ratio between the weights ([1,1,2] is similar to [0.25,0.25,0.5]). Possible values are doubles.
- equal_weights (also with no extra_arg): Assumes the weights are 1/(N+1) each, continue as "weights" type.

Notes:

- l4_portal must be first in cascade, last in cascade, or surrounded with shortcuts / portals (regular, not l4).
- ddr must be first in cascade, last in cascade, or surrounded with shortcuts / portals (regular, not l4).

Platform Param

Definition

```
platform_param(param=value)
```

Examples

```
platform_param(targets=[ethernet])
platform_param(hints=[low_pcie_bandwidth])
```

Description This sets several parameters regarding the platform hosting Hailo as described below:

- `targets` – a list or a single value of hosting target restrictions such as `Ethernet` which requires shutting a set of features.

  Current supported targets: `Ethernet`, which disables the following features:

  - DDR portals, since the DDR access through PCIe is not available
  - Context Switch (multi contexts), since DDR access is not available
  - Sequencers (a fast PCIe-based model loading)

- `hints` – a list of hints or a single hint about the hosting platform such as `Low PCIE bandwidth` which optimizes performance for specific scenarios.

  Current supported hints: `low_pcie_bandwidth`, adjusts the compiler to reduce the PCIE bandwidth by disabling or changing decision thresholds regarding when PCIE should be used.

Performance Param

Definition

```

```

performance_param(fps=100)

Description Demanded the allocation will find a solution that will give a specific fps. Can be set for the entire net or specific layers, context, or network groups.

- setting `fps=None` means maximal fps.

This command requires:

- Require the compiler to meet the specified FPS.
- The compiler will ignore this command if the model is Multi-Context..

Network Groups

Definition

```
group_name = network_group([scope_1_0, ... , scope_0_n])
```

Example

```
my_group = network_group([net1, net2])
```

Description Define network groups to include the different connected components (“networks” or “scopes”) in the model. When more than one network_group command is used, a multi-network-group .hef file is created. Each network_group should be activated at a time using HailoRT (or, preferably, using HailoRT Scheduler API).

The old default behavior (up to and including v3.19) was to compile all connected components together to a single network group, so that they are allocated side-by-side on chip, along all contexts.

The new default is changed so that each connected component resides on a different network group, each with its own separate contexts (similar to using multiple .hef files). The new behavior could be reverted by explicitly putting all connected components in one network group.

This command requires:

- Using the `join()` API with `join_action=JoinAction.NONE` to create a "multi-network file" that includes both networks (can use multiple times to add more networks).
- Intersection of all network groups should be an empty group - no scope can compile in 2 separate groups.
- Union of all network groups equals all the networks in the hn or given network.

5.5. Supported layers

The following section describes the layers and parameters range that the Dataflow Compiler supports.

Note: Unless otherwise specified, the supported features in this section describe their support across the Dataflow Compiler components (Profiler, Allocator, Compiler, Emulator and Model optimization).

Note: Padding type definitions are:

- `SAME`: *Symmetric padding*.
- `SAME_TENSORFLOW`: Identical to Tensorflow SAME padding.
- `VALID`: No padding, identical to Tensorflow VALID padding.

Note: Up to four successor layers are supported after each layer. Each successor receives the same data, except when using the *Features Split layer*.

5.5.1. Convolution

Table 17. Convolution kernel supported parameters

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding

1x1	1x1, 2x1, 2x2	1x1	`SAME`
			`SAME_TENSORFLOW`
			`VALID`

3x3	1x1, 1x2, 2x1, 2x2	1x1
		2x2 (stride=1x1 only)
		3x3 (stride=1x1 only)
		4x4 (stride=1x1 only)
		6x6 (stride=1x1 only)
		8x8 (stride=1x1 only)
		16x16 (stride=1x1 only)
			`SAME`
			`SAME_TENSORFLOW`
			`VALID`

2x2	2x1	1x1	`SAME`
			`SAME_TENSORFLOW`
			`VALID`

Continued on next page

Table 17 – continued from previous page

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
2x2, 2x3, 2x5,2x7, 3x2, 5x2, 7x2	2x2	1x1	SAME SAME_TENSORFLOW
5x5, 7x7	1x1, 1x2, 2x1, 2x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1 only)
6x6	2x2	1x1	SAME SAME_TENSORFLOW VALID
1x3, 1x5, 1x7	1x1, 1x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1 only)
3x5, 3x7, 5x3, 5x7, 7x3, 7x5	1x1, 1x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1 only)
3x1	1x1, 2x1	1x1	SAME SAME_TENSORFLOW VALID
5x1, 7x1	1x1	1x1	SAME SAME_TENSORFLOW VALID
1x9, 3x9, 5x9, 7x9	1x1	1x1	SAME SAME_TENSORFLOW VALID
9x1, 9x3, 9x5, 9x7, 9x9	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x4, 5x4, 7x4, 9x4	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x6, 5x6, 7x6, 9x6	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x8, 5x8, 7x8, 9x8	1x1	1x1	SAME SAME_TENSORFLOW VALID
1xW	1x1	1x1	SAME SAME_TENSORFLOW VALID
MxN	MxN	1x1	SAME SAME_TENSORFLOW VALID
MxN, where M,N in {1..16}	AxB, where A,B in {1..4}	CxD, where C,D in {1..9}	SAME SAME_TENSORFLOW VALID
Any other (preview)	Any other (preview)	Any other (preview)	SAME SAME_TENSORFLOW VALID

W means the width of the layer's input tensor. In other words, in this case the kernel width equals to the image width.

Table 18. Convolution & add kernel supported parameters

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
1x1, 1x3, 1x5, 1x7	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x1, 3x3, 3x5, 3x7	1x1	1x1	SAME SAME_TENSORFLOW VALID
5x1, 5x3, 5x5, 5x7	1x1	1x1	SAME SAME_TENSORFLOW VALID
7x1, 7x3, 7x5, 7x7	1x1	1x1	SAME SAME_TENSORFLOW VALID

Note: Convolution kernel with elementwise addition supports the addition of two tensors only.

Note: Number of weights per layer <= 8MB (for all Conv layers).

5.5.2. Max Pooling

Table 19. Max pooling kernel supported parameters

Kernel (HxW)	Stride (HxW)	Padding
2x2	1x1, 2x1, 2x2	SAME SAME_TENSORFLOW VALID
1x2	1x2	SAME SAME_TENSORFLOW VALID
3x3	1x1	SAME SAME_TENSORFLOW VALID
3x3	2x2	SAME SAME_TENSORFLOW VALID
5x5, 9x9, 13x13	1x1	SAME SAME_TENSORFLOW VALID
Any other	Any other	SAME SAME_TENSORFLOW VALID

"Any other" means any kernel size or stride between 2 and the tensor's dimensions, for example $2 \leq k_h \leq H$ where k_h is the kernel height and H is the height of the layer's input tensor.

5.5.3. Dense

Dense kernel is supported. It is supported only after a Dense layer, a Conv layer, a Max Pooling layer, a Global Average Pooling layer, or as the first layer of the network.

When a Dense layer is after a Conv or a Max Pooling layer, the data is reshaped to a single vector. The height of the reshaped image in this case is limited to 255 rows.

5.5.4. Average Pooling

Table 20. Average pooling kernel supported parameters

Kernel (HxW)	Stride (HxW)	Padding
2x2	2x2	SAME SAME_TENSORFLOW VALID
3x3	1x1, 2x2	SAME SAME_TENSORFLOW
3x4	3x4	SAME SAME_TENSORFLOW VALID
5x5	1x1, 2x2	SAME SAME_TENSORFLOW
hxW	hxW	VALID
Global	n/a	n/a
Any other (preview)	Any other (preview)	SAME SAME_TENSORFLOW VALID

'W' means the width of the layer's input tensor. In other words, in this case the kernel width equals to the image width. 'h' means any height, from 1 up to the input tensor height.

Note: In Global Average Pooling, F<=8192 cases are supported.

5.5.5. Concat

This layer requires 4-dimensional input tensors (batch, height, width, features), and concatenates them in the features dimension. It supports up to 4 inputs.

5.5.6. Deconvolution

Table 21. Deconvolution kernel supported parameters

Kernel (HxW)	Rate (HxW)	Padding
16x16	8x8	SAME_TENSORFLOW
8x8	4x4	SAME_TENSORFLOW
4x4	4x4	SAME_TENSORFLOW
4x4	2x2	SAME_TENSORFLOW

Continued on next page

Table 21 – continued from previous page

Kernel (HxW)	Rate (HxW)	Padding
2x2	2x2	SAME_TENSORFLOW
1x1	1x1	SAME_TENSORFLOW

5.5.7. Depthwise Convolution

Table 22. Depthwise convolution kernel supported parameters

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
1x1	1x1	1x1	SAME SAME_TENSORFLOW VALID
2x2	2x2	1x1	SAME SAME_TENSORFLOW VALID
3x3	1x1, 2x2	1x1 2x2 (stride=1x1 only) 4x4 (stride=1x1 only)	SAME SAME_TENSORFLOW VALID (stride=1x1, dilation=1x1 only)
3x5, 5x3	1x1	1x1	SAME SAME_TENSORFLOW VALID
5x5	1x1, 2x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1, dilation=1x1 only)
9x9	1x1	1x1	SAME SAME_TENSORFLOW
Any other (preview)	Any other (preview)	Any other (preview)	SAME SAME_TENSORFLOW VALID

Note: In Depthwise Convolution 9x9 layers, only $W\%8 = 0$ and $1 \leq F\%8 \leq 4$ input dimensions are supported, where W is the width and F is the number of features.

5.5.8. Group Convolution

Group Convolution is supported with all supported Convolution kernels.

For Conv 1x1/1, 1x1/2, 3x3/1, and 7x7/2, any number of output features is supported. For all other supported Conv kernels, only OF%8=0 or OF<8 is supported, where OF is the number of output features in each group.

5.5.9. Group Deconvolution and Depthwise Deconvolution

Group Deconvolution is supported with all supported Deconvolution kernels. Only OF%8=0 or OF<8 is supported, where OF is the number of output features in each group.

Depthwise Deconvolution is a sub case of Group Deconvolution.

5.5.10. Elementwise Multiplication and Division

Elementwise operations require:

1. Two input tensors with the same shape.

   Example: [N, H, W, F], [N, H, W, F]

2. Two tensors with the same batch and spatial dimensions, one tensor has features dimension 1.

   Example: [N, H, W, F], [N, H, W, 1]

3. Two tensors with the same batch and feature dimensions, one of them has spatial dimension [1, 1].

   Example: [N, H, W, F], [N, 1, 1, F].

4. Two tensors with the same batch dimension, one of them has feature and spatial dimension [1, 1, 1].

   Example: [N, H, W, F], [N, 1, 1, 1].

Note: The *resize layer* can broadcast a tensor from (batch, 1, 1, F) to (batch, height, width, F), where F is the number of features. This may be useful before the Elementwise Multiplication layer.

5.5.11. Add and Subtract

Add and subtract operations are supported in several cases:

1. Bias addition after Conv, Deconv, Depthwise Conv and Dense layers. Bias addition is always fused into another layer.
2. Elementwise addition and subtraction: When possible, elementwise add / sub is fused into a Conv layer as detailed above. Elementwise add / sub is supported on both “Conv like” and “Dense like” tensors, with shapes in the format shown on *Elementwise Multiplication and Division*
3. Addition of a constant scalar to the input tensor.

5.5.12. Input Normalization

Input normalization is supported as the first layer of the network. It normalizes the data by subtracting the given mean of each feature and dividing by the given standard deviation.

5.5.13. Multiplication by Scalar

This layer multiplies its input tensor by a given constant scalar.

5.5.14. Batch Normalization

Batch normalization layer is supported. When possible, it is fused into another layer such as Conv or Dense. Otherwise, it is a standalone layer.

Calculating Batch Normalization statistics in runtime using the Hailo-8 device is not supported.

5.5.15. Resize

Two methods are supported: Nearest Neighbor (NN) and Bilinear. In both methods, the scaling of rows and columns can be different.

Nearest Neighbor Resize

This method is supported in three cases:

1. When the columns and rows scale is a float >= 1 (for rows also <= 4096), the new sizes are integers.
2. When the input shape is (batch, H, 1, F) and the output shape is (batch, rH, W, F). The number of features F stays the same and the height ratio r is integer. This case is also known as "broadcasting".
3. When the input shape is (batch, H, W, 1) and the output shape is (batch, H, W, F). The height H and the width W stay the same. This case is also known as "features broadcasting".

Bilinear Resize

1. With align_corners=True & half_pixels=False: float values from 2x up to 16x, upscale or downscale (1/16x) 2. With align_corners=False & half_pixels=True: integer values from 2x up to 32x , only upscale, for the columns dimension 3. With align_corners=False & half_pixels=True: float values from 2x up to 16x, upscale or downscale (1/16x), for the rows dimension (preview)

Note: align_corners: If True, the centers of the 4 corner pixels of the input and output tensors are aligned, preserving the values at the corner pixels. See definition here (PyTorch) and here (TensorFlow).

half_pixel: Relevant for Pytorch / ONNX, as defined on the ONNX Operators page, under *coordinate_transformation_mode*.

5.5.16. Depth to Space

Depth to space rearranges data from depth (features) into blocks of spatial data.

Two modes are supported (check out ONNX operators spec for more info - https://github.com/onnx/onnx/blob/main/docs/Operators.md#depthtospace):

1. "DCR" mode – the default mode, where elements along the depth dimension from the input tensor are rearranged in the following order: depth, column, and then row.
2. "CRD" mode – elements along the depth dimension from the input tensor are rearranged in the following order: column, row, and the depth.

Table 23. Depth to space kernel supported parameters

Block size (HxW)
1x2
2x1
2x2

Depth to space is only supported when $IF\%(B_W \cdot B_H) = 0$, where IF is the number of input features, B_W is the width of the depth to space block and B_H is the height of the block.

5.5.17. Space to Depth

Space to depth rearranges blocks of spatial data into the depth (features) dimension. The supported block size is 2x2. Two variants are supported:

1. "Classic" variant – The inverse of the Depth to Space kernel. It is identical to Tensorflow's `space_to_depth` operation.
2. "Focus" variant – Used by models such as YOLOv5. It is defined by the following Tensorflow code:

```
op = tf.concat([inp[:, ::block_size, ::block_size, :], inp[:, 1::block_size, ::block_size, :],
                inp[:, ::block_size, 1::block_size, :], inp[:, 1::block_size, 1::block_size, :]],␣
↪axis=3)
```

where `inp` is the input tensor.

5.5.18. Softmax

Softmax layer is supported in three cases:

1. After a "Dense like" layer with output shape (batch, features). In this case, Softmax is applied to the whole tensor.
2. After another layer, if the input tensor of the Softmax layer has a single column (but multiple features). In this case, Softmax is applied row by row.
3. After another layer, even if it has multiple columns. In this case Softmax is applied pixel by pixel on the feature dimension. This case is implemented by breaking the softmax layer to other layers.

5.5.19. Argmax

Argmax kernel is supported if it is the last layer of the network, and the layer before it is has a 4-dimensional output shape (batch, height, width, features).

Note: Currently argmax supports up to 64 features.

5.5.20. Reduce Max

Reduce Max is supported along the features dimension, and if the layer before it is has a 4-dimensional output shape (batch, height, width, features).

5.5.21. Reduce Sum

If the layer before it is has a 4-dimensional output shape (batch, height, width, features), the Reduce Sum layer is supported along the features and width dimension. If the layer before it has a 2-dimensional output shape (batch, features), the Reduce Sum layer is supported along the features dimension.

5.5.22. Feature Shuffle

Feature shuffle kernel is supported if F%G = 0, where G is the number of feature groups.

5.5.23. Features Split

This layer requires 4-dimensional input tensors (batch, height, width, features), and splits the feature dimension into sequential parts. Only static splitting is supported, i.e. the coordinates cannot be data dependent.

5.5.24. Slice

This layer requires 4-dimensional input tensors (batch, height, width, features), and crops a sequential part in each coordinate in the height, width, and features dimensions. Only static cropping is supported, i.e., the coordinates cannot be data dependent.

5.5.25. Reshape

Reshape is supported in the following cases:

"Conv like" to "Dense like" Reshape Reshaping from a Conv or Max Pooling output with shape (batch, height, W', F') to a Dense layer input with shape (batch, F), where $F = W' \cdot F'$.

"Dense like" to "Conv like" Reshape Reshaping a tensor from (batch, F) to (batch, 1, W', F'), where $F = W' \cdot F'$ and $F'\%8 = 0$.

Features to Columns Reshape Reshaping a tensor from (batch, height, 1, F) to (batch, height, W', F'), where $F = W' \cdot F'$.

5.5.26. External padding

This layer implements zeros padding as a separate layer, to support custom padding schemes that are not one of three schemes that are supported as a part of other layers (`VALID`, `SAME` and `SAME_TENSORFLOW`).

5.5.27. Activations

The following activations are supported:

- Linear
- Relu
- Leaky Relu
- Relu 6
- Elu
- Sigmoid
- Exp
- Tanh
- Softplus
- Threshold, defined by `x if x >= threshold else 0`.
- Delta, defined by `0 if x == 0 else const`.
- SiLU

- Swish
- Mish
- Hard-swish
- Gelu
- PRelu
- Sqrt
- Log (preview)
- Hard-sigmoid (preview)

Activations are usually fused into the layer before them, however they are also supported as standalone layers when they can't be fused.

5.5.28. Square, Pow

- Square operator (x*x) is supported.
- Pow operator is currently supported only with exponent=2 (x^2).

5.5.29. L2 Operators

- ReduceL2 is supported.
- L2Normalization is supported.

5.5.30. Note about symmetric padding

The Hailo Dataflow Compiler supports symmetric padding as supported by other frameworks such as Caffe. As the SAME padding in Tensorflow is not symmetric, the only way to achieve this sort of padding is by explicitly using `tf.pad` followed by a convolution operation with `padding='VALID'`. The following code snippet shows how this would be done in Tensorflow (the padding generated by this code is supported by the Dataflow Compiler):

```
pad_total_h = kernel_h - 1
if strides_h == 1:
    pad_beg_h = int(ceil(pad_total_h / 2.0))
else:
    pad_beg_h = pad_total_h // 2
pad_end_h = pad_total_h - pad_beg_h

# skipping the same code for pad_total_w

inputs = tf.pad(
    inputs,
    [[0, 0], [pad_beg_h, pad_end_h], [pad_beg_w, pad_end_w], [0, 0]])
```


Part II

API Reference

6. Model Build API Reference

6.1. hailo_sdk_client.runner.client_runner

Hailo SDK API client.

class hailo_sdk_client.runner.client_runner.**ClientRunner**(*hn=None*, *hw_arch=None*, ...)
Bases: object

Hailo SDK API client.

__init__(*hn=None*, *hw_arch=None*, *hw_version=None*, *har_path=None*, *har=None*)
SDK client constructor

Parameters

- **hn** – Hailo network description (HN), as a file-like object, string, dict, or HailoNN. Use None if you intend to parse the network description from Tensorflow later.
- **hw_arch** (str, optional) – Hardware architecture to be used. Defaults to hailo8.
- **hw_version** (str, optional) – Version of hardware architecture to be used. Defaults to None, which means the SDK uses the default version.
- **har_path** (str, optional) – Hailo Archive file path to initialize the runner from.
- **har** (str or HailoArchive, optional) – Hailo Archive file path or Hailo Archive object to initialize the runner from.

force_weightless_model(*weightless=True*)
SDK API to force the model to work in weightless mode.

When this mode is enabled, the software emulation graph can be received from get_tf_graph() even when the parameters are not loaded.

Note: This graph cannot be used for running inference, unless the model does not require weights.

Parameters **weightless** (bool) – Set to True to enable weightless mode. Defaults to True.

static get_results_by_layer(*calibration_stats_tensors*, *inference_results*, *prev_result_by_layer=None*)
Prepare model statistics for translate_params() and equalize_params().

Parameters

- **calibration_stats_tensors** (list of tf.Tensor) – List of tensors requested by the SDK for statistics gathering. This list can be obtained by calling get_tf_graph() and accessing the required tensor list via the calibration_stats property.
- **inference_results** (list of numpy.ndarray) – List of inference results corresponding to the calibration_stats_tensors given.
- **prev_result_by_layer** (dict) – A previous return value of this function. If used, the statistics will be based on both previous and current input batches.

Returns A dict where the keys are layer names and the values are the results.

Return type dict

load_model_script(*model_script=None*, *model_script_filename=None*)
SDK API for manipulation of the model build params. This method loads a script and applies it to the existing HN, i.e., modifies the specific params in each layer, and sets the model build script for later use.

Parameters **model_script** (str) – Model script allowing the modification of the current model, prior to quantization / native emulation / profiling, etc. The SDK parses the script, and applies the commands as follows:

1. Post translation related commands – These commands are executed during optimization.
2. Quantization related commands – Some of these commands modify the HN, so after the modification each layer (possibly) has new quantization params. Other commands are executed during optimization.
3. Allocation and compilation related commands – These commands are executed during compilation.

Returns A copy of the new modified HN (JSON dictionary).

Return type dict

translate_params(*inference_results*, *previous_statistics=None*, *use_old_params=False*, ...)
SDK API for parameters translation (quantization) to 8 bit.

Note: This method both loads the translated params and returns them, meaning there is no need to call `load_params()` after this method.

Parameters

- **inference_results** (dict) – Statistics computed using native emulation. Returned by `get_results_by_layer()`.
- **previous_statistics** (ModelParams) – Translated network params returned by a previous call to this function.
- **use_old_params** (bool) – Instructs the SDK to start from the translated params it already has.
- **max_elementwise_feed_repeat** (int, optional) – Max value of elementwise feed repeat, used for calculating the quantized representation of biases and elementwise-add.

Returns Translated (quantized) model parameters.

Return type ModelParams

equalize_params(*inference_results*, *mode='min_based'*)
SDK API for parameters Equalization.

Note: This method both loads the equalized params and returns them, meaning there is no need to call `load_params()` after this method.

Parameters

- **inference_results** (dict) – Statistics computed using native emulation. Returned by `get_results_by_layer()`.
- **()** (mode) – the algo type

Returns Equalized model parameters.

Return type ModelParams

References

Meller, E., Finkelstein, A., Almog, U. and Grobman, M., 2019. Same, Same But Different – Recovering Neural Network Quantization Error Through Weight Factorization. https://arxiv.org/abs/1902.01917

update_params_layer_bias(*bias_diff*, *layer*)
SDK API for updating a layer bias. Used by quantization algorithms such as the Iterative Bias Correction algorithm.

Parameters

- **bias_diff** (numpy.ndarray) – Bias difference in each feature.
- **layer** (tf.Tensor) – Emulation graph node of the layer to which bias_diff is added.

Returns Translated (quantized) model parameters, after the bias change.

Return type ModelParams

load_params(*params*, *params_kind=None*)
Load network params (weights).

Parameters

- **params** – If a string, this is treated as the path of the npz file to load. If a dict, this is treated as the params themselves, where the keys are strings and the values are numpy arrays.
- **params_kind** (str, optional) – Indicates whether the params to be loaded are native, native after BN fusion, or quantized.

Returns Kind of params that were actually loaded.

Return type str

save_params(*path*, *params_kind='native'*)
Save all model params to a npz file.

Parameters

- **path** (str) – Path of the npz file to save.
- **params_kind** (str, optional) – Indicates whether the params to be saved are native, native after BN fusion, or quantized.

get_previous_hailo_export()
Get the last Hailo export returned to the user.

get_tf_graph(*target*, *nodes=None*, *translate_input=None*, *rescale_output=None*, *custom_session=None*, ...)
SDK API for getting Tensorflow graph of current model.

Parameters

- **target** – One of the hardware targets (HailoHWObject) or one of the emulation targets (EmulationObject).
- **nodes** (dict) – Input layer names mapped to last Tensorflow nodes of the pre-processing stage. These nodes represent the inputs of the SDK. Layers are asserted to be the HN input layers' names. If there is only one input to the graph, nodes also accepts a Tensor.
- **translate_input** (bool, optional) – Set to True if the input is in native scale and has to be translated to uint8. Usually, True in numeric and hardware targets. Defaults to None, which sets it to True for numeric and hardware targets and False for native targets.
- **rescale_output** (bool, optional) – Set to True to rescale the results from uint8 to their native scale. Typically, True in numeric and hardware targets. Defaults to None, which sets it to True for numeric and hardware targets and False for native targets.
- **custom_session** (tf.Session, optional) – Tensorflow session in which the returned graph will be loaded. Defaults to None, which means the SDK will create a new session.

- **twin_mode** (bool, optional) – When you want to emulate the same model twice inside the same Tensorflow graph, set it to True in the second call to avoid tensor name conflicts. For instance, this is useful to emulate both native and numeric targets together. Defaults to False.
- **native_layers** (list of str, optional) – When using SdkMixed target, use this param to specify the names of all layers to keep native. All other layers will be emulated in numeric mode. Defaults to None.
- **fps** (float, optional) – Allocation FPS. If None, the compilation process will automatically try to reach max throughput (max FPS). Defaults to None.
- **use_preloaded_compilation** (bool, optional) – Use the HEF loaded to the HailoHWObject. If no HEFs are loaded or this flag is set to False, the HEF will be compiled. Relevant if the requested inference target is a HailoHWObject. Defaults to False.
- **mapping_timeout** (int, optional) – Compilation timeout for the whole run. By default, the timeout is calculated dynamically based on the model size.
- **allocator_script_filename** (str, optional) – Model script allowing fine-tuning of allocation. If present, the Allocator parses it command-by-command and executes.
- **network_groups** (list, optional) – A list of network groups received form configure().

Returns An object that holds the new tensors that have been added to the graph, and serialized HEF in case the target is hardware and the HEF is not previously loaded.

Return type HailoGraphExport

Example

```
>>> runner = get_example_runner()
>>> hailo_export = runner.get_tf_graph(
...     target=SdkNative(),
...     nodes={'input_layer_0': tf.compat.v1.placeholder(dtype=tf.float32, shape=[None,␣
↪10])},
...     translate_input=False,
...     rescale_output=False
... )
```

get_hw_representation(*fps=None*, *mapping_timeout=None*, *allocator_script_filename=None*)
SDK API for compiling current model to Hailo hardware.

Parameters

- **fps** (float, optional) – Allocation FPS. If None, the compilation process will automatically try to reach max throughput (max FPS). Defaults to None.
- **mapping_timeout** (int, optional) – Compilation timeout for the whole run. By default, the timeout is calculated dynamically based on the model size.
- **allocator_script_filename** (str, optional) – Model script allowing fine-tuning of allocation. If present, the allocator parses it command-by-command and executes.

compile()
SDK API for compiling current model to Hailo hardware.

Returns Data of the HEF that contains the hardware representation of this model.

Return type bytes

Example

```
>>> runner = get_example_runner()
>>> compiled_model = runner.compile()
```

hef_infer_context(*hailo_export*)

translate_onnx_model(*model=None*, *net_name='model'*, *start_node_names=None*, *end_node_names=None*, ...)

SDK API for parsing an ONNX model. This creates a runner with loaded HN (model) and parameters.

Parameters

- **model** (str or bytes or pathlib.Path) – Path or bytes of the ONNX model file to parse. Use None if original_model_path is already set.
- **net_name** (str) – Name of the new HN to generate.
- **start_node_names** (list of str, optional) – List of ONNX nodes that parsing will start from.
- **end_node_names** (list of str, optional) – List of ONNX nodes, that the parsing can stop after all of them are parsed.
- **compilation_params** (dict, optional) – Compilation params to add to all layers in the parsed model. Defaults to None.
- **quantization_params** (dict, optional) – Quantization params to add to all layers in the parsed model. Defaults to None.
- **ew_add_policy** (EWAddPolicy, optional) – Parser policy regarding elementwise-add op. Default behavior is to fuse the elementwise-add layer to existing convolution layer.
- **net_input_shapes** (dict or list, optional) – A dictionary describing the input shapes for each of the start nodes given in start_node_names, where the keys are the names of the start nodes and the values are their corresponding input shapes. Use only when the original model has dynamic input shapes (described with a wildcard denoting each dynamic axis, e.g. [b, c, h, w]). Can be list (e.g. [b, c, h, w]) for single input network.
- **augmented_path** – Path to save a modified model, augmented with tensors names (where applicable).
- **disable_shape_inference** – When set to True, shape inference with onnx runtime will be disabled.

Note: Using a non-default start_node_names requires the model to be shape inference compatible, meaning either it has a real input shape, or, in case of a dynamic input shape, the net_input_shapes field is provided to specify the input shapes of the given start nodes.

Returns The first item is the HN JSON as a string. The second item is the params dict.

Return type tuple

translate_tf_model(*model_path=None*, *net_name='model'*, *start_node_names=None*, ...)

SDK API for parsing a TF model given by a checkpoint/pb/savedmodel/tflite file. This creates a runner with loaded HN (model) and parameters.

Parameters

- **model_path** (str) – Path of the file to parse, use None if original_model_path is already set. Supported formats: Chekpoint (TF1): Model name with .ckpt suffix (without the final .meta). Frozen (TF1): Frozen graph, model name with .pb suffix. SavedModel (TF2): Saved model export from keras, file named saved_model.pb|pbtxt from the model dir. TFLite: Tensorflow lite model, converted from ckpt/frozen/keras to file with .tflite suffix.
- **net_name** (str) – Name of the new HN to generate.

- **start_node_names** (list of str, optional) – List of tensorflow nodes that parsing will start from. If this parameter is specified, start_node_name should remain empty.
- **end_node_names** (list of str, optional) – List of Tensorflow nodes, which the parsing can stop after all of them are parsed.
- **compilation_params** (dict, optional) – Compilation params to add to all layers in the parsed model. Defaults to None.
- **quantization_params** (dict, optional) – Quantization params to add to all layers in the parsed model. Defaults to None.
- **ew_add_policy** (EWAddPolicy, optional) – Parser policy regarding elementwise-add op. Default behavior is to fuse the elementwise-add layer to existing convolution layer.
- **tensor_shapes** (dict, optional) – A dictionary containing names of tensors and shapes to set in the Tensorflow graph. Use only for placeholder with a wildcard shape.

Returns The first item is the HN JSON, as a string. The second item is the params dict.

Return type tuple

Example

```
>>> inputs = tf.compat.v1.placeholder(tf.float32, [None, 32, 32, 1])
>>> conv = tf.layers.conv2d(inputs, 16, 3, activation=tf.nn.relu, name='my_conv')
>>> sess = tf.Session()
>>> with sess.as_default():
...     _ = sess.run([tf.compat.v1.global_variables_initializer()])
...     _ = tf.train.Saver().save(sess, './example.ckpt')
>>> runner = ClientRunner(hw_arch='hailo8')
>>> hn, params = runner.translate_tf_model(
...     'example.ckpt', 'MyCoolModel', ['my_conv/Conv2D'], ['my_conv/Relu'])
```

join(*runner*, *scope1_name=None*, *scope2_name=None*, *join_action=JoinAction.NONE*, *join_action_info=None*)

SDK API to join two models, so they will be compiled together.

Parameters

- **runner** (ClientRunner) – The client runner to join to this one.
- **scope1_name** (dict or str, optional) – In case dict is given, mapping between exisiting scope names to new scope names for the layers of this model (see example below). In case str is given, scope name to use for all layers of this model. String can be used only when there is a single scope name.
- **scope2_name** (dict or str, optional) – Same as *scope1_name* for the runner to join.

  Example:

  ```
  >>> net1_scope_names = {'net1_scope1': 'net_scope1',
  ...                     'net1_scope2': 'net_scope2'}
  >>> net2_scope_names = {'net2': 'net_scope3'}
  >>> runner1.join(runner2, scope1_name=net1_scope_names,
  ...              scope2_name=net2_scope_names)
  ```

- **join_action** (JoinAction, optional) – Type of action to run in addition to joining the models:
  - NONE: Join the graphs without any connection between them.
  - AUTO_JOIN_INPUTS: Automatically detect inputs for both graphs, and join them into one. Only works when both networks have a single input of the same shape.
  - AUTO_CHAIN_NETWORKS: Automatically detect output of this model, and input of the other model, and connect them. Only works when this model has a single output, and the other model has a single input, of the same shape.

- CUSTOM: Supply a custom dictionary `join_action_info`, which specifies which nodes from this model need to be connected to which of the nodes in the other graph. If keys and values are inputs, the inputs are joined. If keys are outputs, and values are inputs, the networks are chained as described in the dictionary.

- **join_action_info** (`dict, optional`) – Join information to be given when `join_action` is NONE, as explained above.

Example

```
>>> info = {"net1/output_layer1": "net2/input_layer2",
...         "net1/output_layer2": "net2/input_layer1"}
>>> runner1.join(runner2, join_action=JoinAction.CUSTOM, join_action_info=info)
```

profile(*profiling_mode=None*, *should_use_logical_layers=True*, *hef_filename=None*, *runtime_data=None*)
SDK API of the Profiler.

Parameters

- **profiling_mode** (ProfilerModes, optional) – The mode the profiler is executed in. Defaults to None, which sets the profiling mode to PRE_PLACEMENT.
- **hef_filename** (`str, optional`) – HEF file path. If given, the HEF file is used. If not given and the HEF from the previous compilation is cached, the cached HEF is used; Otherwise the automatic mapping tool is used. Use `compile()` to generate and set the HEF. Only in post-placement mode. Defaults to None.
- **should_use_logical_layers** (`bool, optional`) – Indicates whether the Profiler should combine all physical layers into their original logical layer in the report. Defaults to True.
- **runtime_data** (`str, optional`) – runtime_data.json file path produced by hailortcli collect-runtime-data.

Returns The first item is a JSON with the profiling result summary. The second item is a CSV table with detailed profiling information about all model layers. The third item is the latency data.

Return type tuple

Example

```
>>> runner = get_example_runner()
>>> summary, details, latency = runner.profile()
```

profile_hn_model(*fps=None*, *profiling_mode=None*, *should_use_logical_layers=True*, *allocator_script=None*, ...)
SDK API of the Profiler.

Parameters

- **fps** (`float`) – Target frames per second rate.
- **profiling_mode** (ProfilerModes, optional) – The mode the profiler is executed in. Defaults to None, which sets the profiling mode to PRE_PLACEMENT.
- **hef_filename** (`str, optional`) – HEF file path. If given, the HEF file is used. If not given and the HEF from the previous compilation is cached, the cached HEF is used; Otherwise the automatic mapping tool is used. Use `compile()` to generate and set the HEF. Only in post-placement mode. Defaults to None.
- **should_use_logical_layers** (`bool, optional`) – Indicates whether the Profiler should combine all physical layers into their original logical layer in the report. Defaults to True.
- **allocator_script** (`str, optional`) – allocator_script file path. If given, the allocation will consider the script's instructions. If the script contains quantization params, they will be verified against the current HN.

- **runtime_data** (str, optional) – runtime_data.json file path produced by hailortcli collect-runtime-data.

Returns The first item is a JSON with the profiling result summary. The second item is a CSV table with detailed profiling information about all model layers. The third item is the latency data.

Return type tuple

get_mapped_graph()
SDK API for retrieving model mapping.

save_autogen_allocation_script(*path*)
SDK API for retrieving listed operations of last allocation in .alls format.

Parameters **path** (str) – Path where the script is saved.

Returns False if autogenerated script was not created; otherwise it returns True.

Return type bool

property model_name
Get the current model (network) name.

property model_optimization_commands

property hw_arch

property state
Get the current model state.

property hef
Get the latest HEF compilation.

property model_script

get_params(*keys=None*)
Get the native (non quantized) params the runner uses.

Parameters **keys** (list of str, optional) – List of params to retrieve. If not specified, all params are retrieved.

get_params_after_bn(*keys=None*)
Get the native (non quantized) params after batch normalization fusing.

Parameters **keys** (list of str, optional) – List of params to retrieve. If not specified, all params are retrieved.

get_params_translated(*keys=None*)
Get the quantized params the SDK uses.

Parameters **keys** (list of str, optional) – List of params to retrieve. If not specified, all params are retrieved.

get_params_fp_optimized(*keys=None*)
Get the fp optimized params.

Parameters **keys** (list of str, optional) – List of params to retrieve. If not specified, all params are retrieved.

get_hn_str()
Get the HN JSON after serialization to a formatted string.

get_hn_dict()
Get the HN of the current model as a dictionary.

get_hn()
Get the HN of the current model as a dictionary.

get_hn_model()
Get the HailoNN object of the current model.

get_native_hn_str()
Get the HN JSON after serialization to a formatted string.

get_native_hn_dict()
Get the HN of the current model as a dictionary.

get_native_hn()
Get the HN of the current model as a dictionary.

get_native_hn_model()
Get the HailoNN object of the current model.

set_hn(*hn*)
Set the HN of the current model.

Parameters **hn** – Hailo network description (HN), as file-like object, string, dict or HailoNN.

save_hn(*path*)
Save the HN of the current model.

Parameters **path** (str) – Path where the hn file is saved.

save_native_hn(*path*)
Save the HN of the current model.

Parameters **path** (str) – Path where the hn file is saved.

set_original_model(*original_model_path*)
Set the original model of the current model.

Parameters **original_model_path** (str) – Path to the original model file.

save_har(*har_path*, *compressed=False*, *save_original_model=False*)
Save the current model serialized as Hailo Archive file.

Parameters

- **har_path** – Path for the created Hailo archive directory.
- **compressed** – Indicates whether to compress the archive file. Defaults to False.
- **save_original_model** – Indicates whether to save the original model (TF/ONNX) in the archive file. Defaults to False.

load_har(*har_path=None*, *har=None*)
Set the current model properties using a given Hailo Archive file.

Parameters

- **har_path** (str) – Path to the Hailo archive to restore.
- **har** (str or HailoArchive) – Path to the Hailo Archive file or initialized HailoArchive object to restore.

quantize(*calib_data*, *data_type=CalibrationDataType.auto*, *work_dir=None*, *model_script=None*, ...)
This function will be deprecated, please use optimize instead

revert_state(*state*)
Revert the runner's state to the given state.

Parameters **state** (States) – The state to update. Hailo Model and Quantized Model are allowed.

model_summary()
Prints summary of the model layers.

apply_model_modification_commands()
Apply optimizations to the model: Modify the network layers.

optimize(*calib_data*, *data_type=CalibrationDataType.auto*, *work_dir=None*)
Apply optimizations to the model:

- Modify the network layers.

- Quantize model's params, using optional pre-process and post-process algorithm.

  Parameters

  - **calib_data** – Calibration data for Equalization and quantization process. . Type depends on the `data_type` parameter.
  - **data_type** (`CalibrationDataType`) – calib_data's data type, based on enum values:
    - `auto` – Automatically detection.
    - `np_array` – `numpy.ndarray`, or dictionary with input layer names as keys, and values types of `numpy.ndarray`.
    - `dataset` – `tensorflow.data.Dataset` object with valid signature. signature should be either ((h, w, c), image_info) or ({'input_layer1': (h1, w1, c1), 'input_layer2': (h2, w2, c2)}, image_info) image_info can be an empty dict for the quantization
    - `npy_file` – path to a npy or npz file
    - `npy_dir` – path to a npy or npz dir, assumes same shape to all the items
  - **work_dir** (`optional, str`) – If not None, dump quantization debug outputs to this directory.

get_hailo_runtime_model()
This API is preview.

Generate model allowing to run the full ONNX graph using ONNX runtime including the parts that are offloaded to the Hailo-8 (between the start and end nodes) and the parts that are not.

6.2. hailo_sdk_client.exposed_definitions

This module contains enums used by several SDK APIs.

class hailo_sdk_client.exposed_definitions.**JoinAction**(*value*)
Bases: enum.Enum

Special actions to perform when joining models.

See also:

The `join()` API uses this enum.

NONE = 'none'
join the graphs without any connection between them.

AUTO_JOIN_INPUTS = 'auto_join_inputs'
Automatically detect inputs for both graphs, and join them into one. Only works when both networks have a single input of the same shape.

AUTO_CHAIN_NETWORKS = 'auto_chain_networks'
Automatically detect output of this model, and input of the other model, and connect them. Only works when this model has a single output, and the other model has a single input, of the same shape.

CUSTOM = 'custom'
Supply a custom dictionary `join_action_info`, which specifies which nodes from this model need to be connected to which of the nodes in the other graph. If keys and values are inputs, we join the inputs. If keys are outputs, and values are inputs, we chain the networks as described in the dictionary.

class hailo_sdk_client.exposed_definitions.**JoinOutputLayersOrder**(*value*)
Bases: enum.Enum

Enum-like class to determine the output order of a model after joining with another model.

NEW_OUTPUTS_LAST = 'new_outputs_last'
First are the outputs of this model who remained outputs, then outputs of the other model. The order in each sub-list is equal to the original order.

NEW_OUTPUTS_FIRST = 'new_outputs_first'
First are the outputs of the other model, then outputs of this model who remained outputs. The order in each sub-list is equal to the original order.

NEW_OUTPUTS_IN_PLACE = 'new_outputs_in_place'
If the models are chained, the outputs of the other model are inserted, in their original order, to the output list of this model instead of the first output which is no longer an output. If the models are joined by inputs, the other model's outputs are added last.

class hailo_sdk_client.exposed_definitions.**NNFramework**(*value*)
Bases: enum.Enum

Enum-like class for different supported neural network frameworks.

TENSORFLOW = 'tf'
Tensorflow 1.x

TENSORFLOW2 = 'tf2'
Tensorflow 2.x

TENSORFLOW_LITE = 'tflite'
Tensorflow Lite

ONNX = 'onnx'
ONNX

class hailo_sdk_client.exposed_definitions.**CalibrationDataType**(*value*)
Bases: enum.Enum

Types of data used for calibration during quantization.

np_array = 'np_array'
numpy.ndarray or dict of numpy.ndarray

data_feed = 'data_feed'
Tensorflow 1.x initialisable dataset (tf.data.Iterator)

dataset = 'dataset'
Tensorflow 2.x batched dataset object

npy_file = 'npy_file'

npy_dir = 'npy_dir'

auto = 'auto'
Auto detect calibration data type

class hailo_sdk_client.exposed_definitions.**MetaArchitectures**(*value*)
Bases: enum.Enum

Network meta architectures to which in-chip post-processing can be added.

SSD = 'ssd'
Single Shot Detection meta architecture.

CENTERNET = 'centernet'
Centernet meta architecture (future support).

YOLO = 'yolo'
Yolo meta architecture (future support).

class hailo_sdk_client.exposed_definitions.**States**(*value*)
Bases: enum.Enum

Enum-like class with all client runner states.

UNINITIALIZED = 'uninitialized'


```
ORIGINAL_MODEL = 'original_model'

HAILO_MODEL = 'hailo_model'

QUANTIZED_MODEL = 'quantized_model'

COMPILED_MODEL = 'compiled_model'
```

6.3. hailo_sdk_client.hailo_archive.hailo_archive

class hailo_sdk_client.hailo_archive.hailo_archive.**HailoArchive**(*state*, *original_model_path=None*, ...)
Bases: object

Hailo Archive representation.

6.4. hailo_sdk_client.tools.hn_modifications

6.5. hailo_sdk_client.tools.core_postprocess.core_postprocess_api

hailo_sdk_client.tools.core_postprocess.core_postprocess_api.**add_nms_postprocess**(*runner*, ...)
Adds in-chip NMS post-processing to the given model and runner, changing the state of an initialized runner.

Parameters

- **runner** (ClientRunner) – An initialized runner with a model and its weights.
- **output_hn_path** (str) – Path to write the output model as HN with the newly added post-process. Default is None, in which case the HN is not created
- **output_npz_path** (str) – Path to write the corresponding model params for the newly added post-process. Default is None, in which case the NPZ is not created
- **config_json_path** (str) – Path to configuration file, containing parameters for the SSD NMS post-process.
- **meta_arch** (MetaArchitectures) – Meta architecture for the NMS post-process. Defaults to SSD.
- **params_kind** (ParamsKinds) – Kind of params to be saved. Defaults to NATIVE_FUSED_BN.
- **output_har_path** (str) – Path to write the output model as HAR with the newly added post- process. Default is None, in which case the HAR is not created.
- **enforce_iou_threshold** (bool) – if set to true, use iou in config json. Otherwise, use iou = 1 wether the division factor in config json different than 1.

Example

```
>>> from hailo_sdk_client import ClientRunner
>>> from hailo_sdk_client.tools.core_postprocess.core_postprocess_api import add_nms_postprocess
>>> from hailo_sdk_common.targets.inference_targets import ParamsKinds
>>>
>>> start_nodes = ["FeatureExtractor/MobilenetV2/MobilenetV2/input"]
>>>
>>> end_nodes = ["BoxPredictor_0/BoxEncodingPredictor/BiasAdd", "BoxPredictor_0/ClassPredictor/
↪BiasAdd",
...              "BoxPredictor_1/BoxEncodingPredictor/BiasAdd", "BoxPredictor_1/ClassPredictor/
↪BiasAdd",
...              "BoxPredictor_2/BoxEncodingPredictor/BiasAdd", "BoxPredictor_2/ClassPredictor/
↪BiasAdd",
...              "BoxPredictor_3/BoxEncodingPredictor/BiasAdd", "BoxPredictor_3/ClassPredictor/
↪BiasAdd",
```

(continues on next page)

(continued from previous page)

```
...                "BoxPredictor_4/BoxEncodingPredictor/BiasAdd", "BoxPredictor_4/ClassPredictor/
↪BiasAdd",
...                "BoxPredictor_5/BoxEncodingPredictor/BiasAdd", "BoxPredictor_5/ClassPredictor/
↪BiasAdd"]
>>>
>>> runner = ClientRunner()
>>> hn, npz = runner.translate_tf_model('./CKPT/model.ckpt', name='Mobilenet-v2-ssd',
...                                     start_node_names=start_nodes,
...                                     end_node_names=end_nodes)
>>>
>>> add_nms_postprocess(runner,
...                     config_json_path='nms_ssd_config.json',
...                     output_hn_path='./Mobilenet-v2-ssd-nms.hn',
...                     output_npz_path='./Mobilenet-v2-ssd-nms.npz',
...                     params_kind=ParamsKinds.NATIVE)
Note:
    Basic assumptions:
        - Proposal generator layers are using hard-coded activations:
            * SSD - sigmoid
            * Centernet - relu
```

See also:

add_nms_postprocess_from_har() for additional details.

6.6. hailo_sdk_client.tools.layer_noise_analysis

exception hailo_sdk_client.tools.layer_noise_analysis.**MissingPandocException**
Bases: Exception

exception hailo_sdk_client.tools.layer_noise_analysis.**FinetuneAnalysisModeException**
Bases: Exception

exception hailo_sdk_client.tools.layer_noise_analysis.**FinetuneWithoutInverseException**
Bases: Exception

exception hailo_sdk_client.tools.layer_noise_analysis.**FinetuneQuantModeException**
Bases: Exception

exception hailo_sdk_client.tools.layer_noise_analysis.**IllegalQuantModeAndTestTarget**
Bases: Exception

Raised on unsupported Analyzer operations..

exception hailo_sdk_client.tools.layer_noise_analysis.**IllegalAnalyzerAction**
Bases: Exception

Raised on unsupported Analyzer operations.

exception hailo_sdk_client.tools.layer_noise_analysis.**IllegalAnalysisMode**
Bases: Exception

Raised on unsupported Analyzer operations.

exception hailo_sdk_client.tools.layer_noise_analysis.**QuantAnalyzerException**
Bases: hailo_sdk_client.tools.cmd_utils.base_utils.CmdUtilsBaseUtilError

class hailo_sdk_client.tools.layer_noise_analysis.**QuantAnalyzer**(*runner*, *data_path=None*, ...)
Bases: object

This class analyzes quantized models vs the original floating point models.

__init__(*runner*, *data_path=None*, *calib_path=None*, *batch_size=8*, *work_dir=None*, *tensors_to_analyze=None*, ...)
Initialize the object.

Parameters

- **runner** (ClientRunner) – SDK client runner to work with.
- **data_path** (str) – Path to validation set data.
- **batch_size** (int) – Size of each batch of images to be processed.
- **work_dir** (str) – Path to working directory where the data will be saved.

property nn

set_data_feed_cb(*data_feed_cb*)

set_calib_feed_cb(*calib_feed_cb*)

set_postprocess_cb(*postprocess_cb*)

set_eval_cb(*eval_cb*)

analyze_full_quant_model(*ref_target='sdk_fp_optimized'*, *test_target='sdk_partial_numeric'*, *layers=None*, ...)
Perform noise analysis of the globally quantized model (e.g., as defined in the .alls quantization script, if present).

Parameters

- **ref_target** – (HailoHWObject or EmulationObject): One of the hardware targets or one of the emulation targets (e.g., SdkNumeric()). Used as refernce point for noise calculations.
- **test_target** – (HailoHWObject or EmulationObject): One of the hardware targets or one of the emulation targets (e.g., SdkNumeric()). Used as the test target for noise measeurments.
- **inverse** (bool, optional) – Indicates whether to treat layers as a list of native or numeric layers. Defaults to False, which means to treat them as numeric layers.
- **layers** (list of str, optional) – List of layer names to emulate in numeric mode. If inverse is set to True, the layers will be emulated in native mode. Defaults to None.
- **quant_act** (bool, optional) – Indicates whether to run with quantized activations. Defaults to True.
- **quant_weights** (bool, optional) – Indicates whether to run with quantized weights. Defaults to True.

Returns namedtuple where each element is a NoiseType namedtuple whose elements are a dict with keys being the model layers and values being the quantization noise measurement (e.g., layer_noises.logits.snr['conv2'] gives the logits SNR quantization noise when only conv2 is emulated in numeric mode).

Return type TensorNoises

analyze_layer_by_layer(*ref_target='sdk_fp_optimized'*, *test_target='sdk_mixed'*, *eval_num=1000000000.0*, ...)
Perform layer by layer noise analysis of the model, where at each iteration a different layer is emulated in numeric mode.

Parameters

- **ref_target** – (HailoHWObject or EmulationObject): One of the hardware targets or one of the emulation targets (e.g., SdkNumeric()). Used as reference point for noise calculations.
- **test_target** – (HailoHWObject or EmulationObject): One of the hardware targets or one of the emulation targets (e.g., SdkNumeric()). Used as the test target for noise measurements.
- **layers** (list of str, optional) – List of layers to perform the layer by layer noise analysis. Defaults to None, in which case all network layers are iteratively emulated in numeric mode.

- **inverse** (bool, optional) – If set to True, a layer by layer analysis will be performed, in which all layers are quantized except one in a given iteration. Defaults to False.
- **quant_act** (bool, optional) – Indicates whether to run with quantized activations. Defaults to True.
- **quant_weights** (bool, optional) – Indicates whether to run with quantized weights. Defaults to True.

Returns namedtuple where each element is a NoiseType namedtuple whose elements are a dict with keys being the model layers and values being the quantization noise measurement. For example, `noises.logits.snr['conv2']` gives the logits SNR quantization noise when only conv2 is emulated in numeric mode).

Return type TensorNoises

create_pdf(*file_name='report'*)
Gather all generated figures stored in this object and produce a .pdf report.

Parameters **file_name** (str) – Name of the .pdf file to be compiled. Defaults to `report.pdf`.

run_analysis(*ref_target='sdk_fp_optimized'*, *test_target='sdk_mixed'*, *mode='layer_by_layer'*, ...)

class hailo_sdk_client.tools.layer_noise_analysis.**QuantizationAnalyzer**(*runner*, ...)
Bases: hailo_sdk_client.tools.layer_noise_analysis.QuantAnalyzer

This class quantizes models, and then analyzes the quantized model vs the original floating point model.

__init__(*runner*, *data_path=None*, *calib_path=None*, *calib_feed_cb=None*, *calib_set_size=None*, ...)
Initialize the object.

Parameters

- **runner** (ClientRunner) – SDK client runner to work with.
- **data_path** (str) – Path to validation set data.
- **batch_size** (int) – Size of each batch of images to be processed.
- **work_dir** (str) – Path to working directory where the data will be saved.

quantize_runner(*equalize=True*, *ibc=False*, *run_lbc=False*, *quantization_script_filename=None*, *layers=None*, ...)
Quantize a model. See the documentation of `run_quantization_from_np()` for details.

hailo_sdk_client.tools.layer_noise_analysis.**analyze_net**(*model_path*, *data_path*, *calib_path*, ...)
Run a complete flow of quantizing a model and analyzing the resulting quantization.

6.7. hailo_sdk_client.tools.dead_channels_removal

hailo_sdk_client.tools.dead_channels_removal_api.**dead_channels_removal_from_files**(*input_hn_path*, ...)
Remove dead channels from a model given in HN and NPZ files.

Parameters

- **input_hn_path** (str) – Path to the input model's HN file.
- **input_npz_path** (str) – Path to the corresponding model params.
- **output_hn_path** (str) – Path to write the output model after the dead channels removal.
- **output_npz_path** (str) – Path to write the corresponding model params after the dead channels removal.
- **params_kind** (ParamsKinds) – The input model params kind, NATIVE or NATIVE_FUSED_BN are supported.

See also:

dead_channels_removal_from_runner() for additional details about the dead channels removal algorithm.

hailo_sdk_client.tools.dead_channels_removal_api.**dead_channels_removal_from_runner**(*runner*, ...)

Remove dead channels. This function gets a runner, removes the dead channels from the model it holds, and loads the modified model back to the runner.

In some cases, training causes dead channels in layers, e.g., the output of specific channels of a layer is always zero. In particular, this happens when all the weights of specific channels are zero, and the results of running the activation on the bias of the channels are zero.

In this case, when there are dead channels, and because of the properties of convolution, if the successor layer also has a kernel, the summation of the feature will not change the result, and hence it can be eliminated.

Parameters

- **runner** (ClientRunner) – An initialized runner with a model and its weights. This runner will be modified.
- **output_hn_path** (str, optional) – Path to write the output model.
- **output_npz_path** (str, optional) – Path to write the corresponding output model params.
- **output_har_path** (str, optional) – Path to write the corresponding output model as Hailo Archive.

7. Common API Reference

7.1. hailo_sdk_common.export.hailo_graph_export

Represents how Hailo models are returned to the user for Tensorflow integration.

exception hailo_sdk_common.export.hailo_graph_export.**InputTensorException**
Bases: Exception

Raised when there is a mismatch between the expected inputs and the inputs in practice.

class hailo_sdk_common.export.hailo_graph_export.**ExportLevel**(*value*)
Bases: enum.IntEnum

Enum that describes the granularity of a model export – which layers' tensors are included.

OUTPUT_LAYERS = 0
Exports a list of tensors from the output layers only.

OUTPUT_LAYERS_RESCALED = 1
Exports a list of tensors from output layers only, plus their rescale operations.

ALL_LAYERS = 2
Exports a list of tensors from all layers of the model – output layers and inner layers.

ALL_LAYERS_RESCALED = 3
Exports a list of tensors at all_layers level, plus their rescale operations.

ALL_LAYERS_PRE_ACT_OPS = 4
Exports a list of tensors before the activation of each layer.

ALL_LAYERS_ALL_OPS = 5
Exports a list of tensors at all_layers level, plus all inner operations including bias and pre-activation.

CALIBRATION_STATS = 6
Exports a list of all calibration statistics tensors from all layers of the model.

ACTIVATIONS_HISTOGRAMS = 7
Exports a list of histograms tensors, used in activation clipping prior to quantization.

FT_KERNEL_RANGE = 8

FT_ALPHA = 9

FT_FINAL_KERNEL = 10

FT_KERNEL_FRAC_PART = 11

FT_TRAIN_OUTPUTS = 12

class hailo_sdk_common.export.hailo_graph_export.**VariableExportLevel**(*value*)
Bases: enum.IntEnum

An enumeration.

BIASES = 0
Exports a list of biases from all layers of the model.

BIASES_DELTA = 1
Exports a list of fine-tuned biases used for BFT algorithm.

KERNELS = 2
Exports a list of kernel variables from all layers of the model.

KERNELS_DELTA = 3
Exports a list of fine-tuned kernel variables from all layers of the model.

UNSET_VARIABLES = 4
Exports a list of newly created variables that need to be initialized when the model is returned to client.

class hailo_sdk_common.export.hailo_graph_export.**HailoGraphExport**(*session*, *graph*, *input_tensors*, ...)
Bases: object

Hailo Model export object.

__init__(*session*, *graph*, *input_tensors*, *init_output_exports=None*, *init_variables_exports=None*, *hef=None*, ...)
Constructor for Hailo graph export.

Parameters

- **session** (tf.Session) – Tensorflow session of the returned graph.
- **graph** (tf.Graph) – Tensorflow graph containing the model.
- **input_tensors** (dict) – A dictionary mapping the model's input layers' names in the HN to the names of their input tensors.
- **init_output_exports** (dict) – A dictionary of exports, where the keys are of type ExportLevel and the values are exports of type OutputTensorsExport.
- **init_variables_exports** (dict) – A dictionary of exports, where the keys are of type VariableExportLevel and the values are exports of type VariablesExport.
- **hef** (bytes) – HEF file data.
- **network_groups** (list) – A list of network groups returned from target.configure.

property hef
HEF Binary compiled model files that are loaded to the device.

property input_tensor
tf.Tensor: The input tensor of Hailo emulator/hardware graph.

property network_groups

property hef_infer_wrapper

property input_tensors
list of tf.Tensor: The input tensors of Hailo emulator/hardware graph.

property output_tensors
list of tf.Tensor: Output tensors export at the OUTPUT_LAYERS export level. If rescale_output is enabled, this function returns the rescaled version of the same tensor list.

property ft_train_output_tensors

property all_layers
list of tf.Tensor: All layers (outputs and inner layers) tensors list. If rescale_output is enabled, this function returns the rescaled version of the same tensor list.

property all_layers_pre_act_ops
list of tf.Tensor: Full graph tensors list – including all inner ops of each layer prior to activation.

property all_layers_all_ops
list of tf.Tensor: Full graph tensors list – including all inner ops of each layer.

property calibration_stats
list of tf.Tensor: Full graph stats tensors list – used for model calibration.

property activations_histograms
list of tf.Tensor: Activations histograms tensors list – used for model calibration.

property activations_histograms_layers_names
list of str: Corresponding layers' names list of all activations histograms tensors.

property biases
list of tf.Variable: Full graph bias variables list.

property biases_layers_names
list of str: Corresponding layers' names list of all bias variables (output and inner layers).

property biases_delta
list of `tf.Variable`: Full graph fine tune bias variables list – used for BTF algorithm.

property biases_delta_layers_names
list of str: Corresponding layers' names list of all BFT fine tune bias variables (output and inner layers).

property kernels
list of `tf.Variable`: Full graph kernel variables list.

property kernels_layers_names
list of str: Corresponding layers' names list of kernel variables list (output and inner layers).

property kernels_delta
list of `tf.Variable`: Full graph fine-tuned kernel variables list.

property kernels_delta_layers_names
list of str: Corresponding layers' names list of fine-tuned kernel variables list (output and inner layers).

property ft_kernel_range_tensors
dict of `tf.Tensor`: kernel range by layer name.

property ft_alpha_tensors
dict of `tf.Tensor`: alpha-blend coefficient by layer name.

property ft_final_kernel_tensors
dict of `tf.Tensor`: eventual kernels as used in `SdkFineTune` mode by layer name.

property ft_kern_frac_part_tensors
dict of `tf.Tensor`: fractional part of kernels as used in `SdkFineTune` mode by layer name.

property unset_variables
list of `tf.Variable`: un-initialized variables list.

property unset_variables_layers_names
list of str: un-initialized variables list of layers' names.

property output_tensors_original_names
list of list of str: Corresponding original layers' names list of the basic output tensors list.

property all_layers_original_names
list of list of str: Corresponding original layers' names list of all layers tensors list (output and inner layers).

property output_tensors_layers_names
list of str: Corresponding layers' names list of the basic output tensors list.

property all_layers_names
list of str: Corresponding layers' names list of all layers tensors list (output and inner layers).

property session
`tf.Session`: Tensorflow session to which the new nodes were appended.

property graph
`tf.Graph`: Tensorflow graph to which the new nodes were appended.

property rescale_output
bool: A flag for `rescale_output`. If enabled, `output_tensors` and `all_layers` properties will return the rescaled versions of their levels' tensors lists, respectively.

property variables_exports
dict: A dictionary of exports where the keys are of type `VariableExportLevel` and the values are exports of type `VariablesExport`. Each export holds a list of Tensorflow variables, and a list of Hailo layer names corresponding to each output tensor.

property output_tensors_exports
dict: A dictionary of exports where the keys are of type `ExportLevel` and the values are exports of type `OutputTensorsExport`. Each export holds a list of output Tensorflow tensors, and a list of Hailo layer names corresponding to each output tensor.

get_export_by_level(*export_level=ExportLevel.OUTPUT_LAYERS*)
Retrieve an export entry from the dictionary, according to the given export level.

Parameters **export_level** (ExportLevel) – Which export to get from the dictionary.

Returns Selected export, or None if the level isn't in the dictionary.

Return type OutputTensorsExport

get_variable_export_by_layer(*export_level=VariableExportLevel.BIASES*)
Retrieve a variable export entry from the dictionary, according to the given export level.

Parameters **export_level** (VariableExportLevel) – Which variable export to get from the dictionary.

Returns Selected export, or None if the level isn't in the dictionary.

Return type VariablesExport

get_layers_names_by_level(*export_level=ExportLevel.OUTPUT_LAYERS*)
Retrieve a list of layers' names from an export in the dictionary, according to the given export level.

Parameters **export_level** (ExportLevel) – Which export to get from the dictionary.

Returns Selected export's layer names list, or None if the level isn't in the dictionary.

Return type list of str

get_variable_layers_names_by_level(*export_level=VariableExportLevel.BIASES*)
Retrieves a list of variable layer names from an export in the dictionary, according to the given export level.

Parameters **export_level** (VariableExportLevel) – Which variable export to get from the dictionary.

Returns Selected export layer names list, or None if the level isn't in the dictionary.

Return type list of str

get_original_names_by_level(*export_level=ExportLevel.OUTPUT_LAYERS*)
Retrieve a list of tensors' original names from an export in the dictionary, according to the given export level.

Parameters **export_level** (ExportLevel) – Which export to get from the dictionary.

Returns Selected export's original names list, or None if the level isn't in the dictionary.

Return type list of list of str

get_variable_original_names_by_level(*export_level=VariableExportLevel.BIASES*)
Retrieve a list of variables original names from an export in the dictionary, according to the given export level.

Parameters **export_level** (VariableExportLevel) – Which variable export to get from the dictionary.

Returns Selected variable export original names list, or None if the level isn't in the dictionary.

Return type list of list of str

add_output_tensors_export(*new_export*)
Adds an export output_tensor entry to the dictionary, according to the given export level.

Parameters **new_export** (OutputTensorsExport) – The new export to be added.

add_variables_export(*new_export*)
Adds an export variable entry to the dictionary, according to the given export level.

Parameters **new_export** (VariablesExport) – The new export to be added.

update_original_names(*hailo_nn*)
Updates original names lists for all export entries according to their layer names lists and the given HN.

Parameters **hailo_nn** (str) – Hailo NN JSON string, used for creating a HN object for layers lookup by name.

class hailo_sdk_common.export.hailo_graph_export.**OutputTensorsExport**(*export_level*, *tensors*, ...)
Bases: object

A single output tensors export as part of the HailoGraphExport object which holds one of this for each export level.

__init__(*export_level*, *tensors*, *layers_names*)
Constructor for a single export. Initializes original_names as an empty list, it's calculated post serialization to client.

Parameters

- **export_level** (ExportLevel) – Export level to which the tensors belong.
- **tensors** (list of tf.Tensor) – List of tensors, appended to the graph by the SDK.
- **layers_names** (list of str) – List of layers' names, with respective layers' names for each tensor in tensors.

property export_level
ExportLevel: Export level to which the tensors belong.

property tensors
list of tf.Tensor: List of tensors appended to the graph by the SDK.

property layers_names
list of str: List of layers' names, with respective layers' names for each tensor.

property original_names
list of list of str: List of layers' names in the original user's model, with respective names for each tensor.

class hailo_sdk_common.export.hailo_graph_export.**VariablesExport**(*export_level*, *variables*, ...)
Bases: object

A single variables export as part of the HailoGraphExport object which holds one of this for each export level.

__init__(*export_level*, *variables*, *layers_names*)
Constructor for a single export. Initializes original_names as an empty list, it's calculated post serialization to client.

Parameters

- **export_level** (VariableExportLevel) – Export level to which the variables belong.
- **variables** (list of tf.Variable) – List of variables, appended to the graph by the SDK.
- **layers_names** (list of str) – List of layers' names, with respective layers' names for each variable in variables.

property export_level
ExportLevel: Export level to which the variables belong.

property variables
list of tf.Variable: List of variables appended to the graph by the SDK.

property layers_names
list of str: List of layers' names, with respective layers' names for each variable.

property original_names
list of list of str: List of layers' names in the original user's model, with respective names for each variable.

7.2. hailo_sdk_common.model_params.model_params

class hailo_sdk_common.model_params.model_params.**ModelParams**(*params*, ...)
Bases: object

Dict-like class that contains all parameters used by a model such as weights, biases, etc.

7.3. hailo_sdk_common.profiler.profiler_common

class hailo_sdk_common.profiler.profiler_common.**ProfilerModes**(*value*)
Bases: enum.Enum

Enum-like class for different execution modes of the profiler.

PRE_PLACEMENT = 'pre_placement'
Profiling before placement is made.

POST_PLACEMENT = 'post_placement'
Profiling after placement is made.

class hailo_sdk_common.profiler.profiler_common.**ProfilerDataTypes**(*value*)
Bases: enum.Enum

Enum-like class for different execution modes of the profiler.

NONE = 'none'
No data is passed.

CONFIG_JLF = 'config_jlf'
Passed data config JLF.

HEF = 'hef'
Passed data hef

7.4. hailo_sdk_common.preprocessing.base

7.5. hailo_sdk_common.preprocessing.normalization

7.6. hailo_sdk_common.hailo_nn.hailo_nn

class hailo_sdk_common.hailo_nn.hailo_nn.**HailoNN**(*network_name=None*, *stage=None*, ***kwargs*)
Bases: networkx.classes.digraph.DiGraph

Hailo NN representation. This is the Python class that corresponds to HN files.

stable_toposort(*key=None*)
Get a generator over the model's layers, topologically sorted.

Example

```
>>> example_hn = '''{
...     "name": "Example",
...     "layers": {
...         "in": {"type": "input_layer", "input": [], "output": ["out"], "input_shape": [-
↪1, 10]},
...         "out": {"type": "output_layer", "input": ["in"], "output": [], "input_shape":␣
↪[-1, 10]}
...     }
... }'''
```

(continues on next page)

(continued from previous page)

```
>>> hailo_nn = HailoNN.from_hn(example_hn)
>>> for layer in hailo_nn.stable_toposort():
...     print('The layer name is "{}"'.format(layer.name))
The layer name is "in"
The layer name is "out"
```

to_hn(*network_name*, *npz_path=None*, *json_dump=True*, *should_get_default_params=False*)
Export Hailo model to JSON format (HN) and params NPZ file. The NPZ is saved to a file.

Parameters

- **network_name** (str) – Name of the network.
- **npz_path** (str, optional) – Path to save the parameters in NPZ format. If it is None, no file is saved. Defaults to None.
- **json_dump** (bool, optional) – Indicates whether to dump the HN to a formatted JSON, or leave it as a dictionary. Defaults to True, which means to dump.
- **should_get_default_params** (bool, optional) – Indicates whether the HN should include fields with default values. Defaults to False, which means they will not be included.

Returns The HN, as a string or a dictionary, depending on the json_dump argument.

to_hn_npz(*network_name*, *json_dump=True*, *should_get_default_params=False*)
Export Hailo model to JSON format (HN) and params NPZ file. The NPZ is returned to the caller.

Parameters

- **network_name** (str) – Name of the network.
- **json_dump** (bool, optional) – Indicates whether to dump the HN into a formatted JSON, or leave it as a dictionary. Defaults to True, which means to dump.
- **should_get_default_params** (bool, optional) – Indicates whether the HN should include fields with default values. Defaults to False, which means they will not be included.

Returns The first item is the HN, as a string or a dictionary, depending on the json_dump argument. The second item contains the model's parameters as a dictionary.

Return type tuple

set_input_tensors_shapes(*inputs_shapes*)
Set the tensor shape (resolution) for each input layer.

Parameters **inputs_shapes** (dict) – Each key is a name of an input layer, and each value is the new shape to assign to it. Currently doesn't support changing number of features.

static from_fp(*fp*)
Get Hailo model from a file.

static from_hn(*hn_json*)
Get Hailo model from HN raw JSON data.

static from_parsed_hn(*hn_json*, *validate=True*)
Get Hailo model from HN dictionary.

7.7. hailo_sdk_common.hailo_nn.hn_definitions

class hailo_sdk_common.hailo_nn.hn_definitions.**EWAddPolicy**(*value*)
Bases: enum.Enum

Enum-like class for determining the Fuser policy regarding elementwise-add implementation.

standalone_add = 'standalone_add'
Standalone elementwise addition layer.

fused_conv_and_add = 'fused_conv_and_add'
Convolution layer fused with elementwise addition. In cases fusing isn't supported the fallback is a standard standalone elementwise add layer.

fused_conv_and_add_identity_fallback = 'fused_conv_and_add_identity_fallback'
Convolution layer fused with elementwise addition. In cases fusing isn't supported the fallback is an identity convolution elementwise add layer.

7.8. hailo_sdk_common.targets.inference_targets

exception hailo_sdk_common.targets.inference_targets.**InferenceTargetException**(*message*, ...)
Bases: hailo_sdk_common.exceptions.exceptions.CommonModelException

Raised when an error related to the inference target has occurred.

exception hailo_sdk_common.targets.inference_targets.**NumericTargetException**
Bases: Exception

class hailo_sdk_common.targets.inference_targets.**InferenceTargets**
Bases: object

Enum-like class with all inference targets supported by the Hailo SDK. See the classes themselves for details about each target.

UNINITIALIZED = 'uninitialized'

SDK_NATIVE = 'sdk_native'

SDK_FP_OPTIMIZED = 'sdk_fp_optimized'

SDK_NUMERIC = 'sdk_numeric'

SDK_DEBUG_PRECISE_NUMERIC = 'sdk_debug_precise_numeric'

SDK_PARTIAL_NUMERIC = 'sdk_partial_numeric'

SDK_FINE_TUNE = 'sdk_fine_tune'

SDK_MIXED = 'sdk_mixed'

HW_SIMULATION = 'hw_sim'

HW_SIMULATION_MULTI_CLUSTER = 'hw_sim_mc'

FPGA = 'fpga'

UDP_CONTROLLER = 'udp'

PCIE_CONTROLLER = 'pcie'

HW_DRY = 'hw_dry'

HW_DRY_UPLOAD = 'hw_dry_upload'

UV_WORKER = 'uv'

DANNOX = 'dannox'

ONNXRT = 'ONNXRT'

class hailo_sdk_common.targets.inference_targets.**InferenceDebugTargets**(*value*)
Bases: enum.Enum

Enum-like class with all debugging options supported by the Hailo SDK.

Note: Only the NO_DEBUG option is currently available for users.

NO_DEBUG = 'no_debug'

CLIENT_DEBUGGER = 'client_debugger'

TRACE = 'trace'

FULL = 'full'

class hailo_sdk_common.targets.inference_targets.**EmulationInferenceTargets**
Bases: object

Enum-like class with all emulation inference targets supported by the Hailo SDK. See the classes themselves for details about each target.

UNINITIALIZED = 'uninitialized'

SDK_NATIVE = 'sdk_native'

SDK_FP_OPTIMIZED = 'sdk_fp_optimized'

SDK_NUMERIC = 'sdk_numeric'

SDK_DEBUG_PRECISE_NUMERIC = 'sdk_debug_precise_numeric'

SDK_PARTIAL_NUMERIC = 'sdk_partial_numeric'

SDK_FINE_TUNE = 'sdk_fine_tune'

SDK_MIXED = 'sdk_mixed'

class hailo_sdk_common.targets.inference_targets.**ParamsKinds**
Bases: object

Enum-like class for kinds of model parameters.

NATIVE = 'native'
Original model parameters, usually floating point 32 bit.

NATIVE_FUSED_BN = 'native_fused_bn'
Model parameters after batch normalization fusing into the layers' weights, but before quantization. When loading native parameters, the SDK automatically generates this kind of parameters.

TRANSLATED = 'translated'
Translated model parameters (quantized to 8 bit integer).

FP_OPTIMIZED = 'fp_optimized'
Native after model optimization

class hailo_sdk_common.targets.inference_targets.**EmulationObject**(**args*, ***kwargs*)
Bases: object

A software based inference target.

Note: This class should not be used directly. Use only its inherited classes.

NAME = 'uninitialized'

IS_NUMERIC = False

IS_HARDWARE = False

IS_SIMULATION = False

__init__(*hw_arch=None*)
Inference object constructor.

Parameters **hw_arch**(str, optional) – Name of the hardware architecture. Defaults to None.

use_device(**args*, ***kwargs*)
A context manager that should wrap any usage of the target.

property name
str: The name of this target. Valid values are defined by InferenceObject.

property is_numeric
bool: Determines whether this target is working in numeric mode.

property is_hardware
bool: Determines whether this target runs on a physical hardware device.

property is_simulation
bool: Determines whether this target is used for hardware simulation.

to_json()
Get a JSON representation of this object.

Returns A JSON dump.

Return type str

class hailo_sdk_common.targets.inference_targets.**SdkNative**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

Native emulation inference target. It runs the model as is without any hardware related changes. You can use it to make sure your model has been converted properly into the Hailo representation (HN). In addition, this target useful when you extract statistics for weights quantization.

NAME = 'sdk_native'

class hailo_sdk_common.targets.inference_targets.**SdkFPOptimized**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

Native emulation inference target. It runs the model as SdkNative, but includes all optimizations on the params(weights clipping, equalization, sorting) if layers had one of them set via a quantization script. This target is used during the optimization and quantization process of a model, and can also be used to analyze the optimization.

NAME = 'sdk_fp_optimized'

__init__(*hw_arch=None*, *enable_clipping=True*)
Inference object constructor.

Parameters **hw_arch**(str, optional) – Name of the hardware architecture. Defaults to None.

property enable_clipping

class hailo_sdk_common.targets.inference_targets.**SdkNumeric**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

Numeric emulation inference target. Use this target when you want to get results that are bit exact to the Hailo hardware output without running on an actual device.

NAME = 'sdk_numeric'

IS_NUMERIC = True

class hailo_sdk_common.targets.inference_targets.**SdkDebugPreciseNumeric**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

This target runs the numeric emulation in a special debug mode.

Note: This target is currently unavailable for users.

NAME = 'sdk_debug_precise_numeric'

class hailo_sdk_common.targets.inference_targets.**SdkPartialNumeric**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

Fast numeric emulation target. This target is not hardware bit exact, but it's *hardware like* and it runs much faster by using some of the original Tensorflow CPU/GPU layers' implementations. This target is useful when researching differences between the original model and the quantized model over large datasets without the actual Hailo hardware device.

NAME = 'sdk_partial_numeric'

IS_NUMERIC = True

class hailo_sdk_common.targets.inference_targets.**SdkFineTune**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

Fine tuning target. You can use this mode to train (or fine tune) the model's weights and biases in a quantization aware manner. Fake quantization is used to allow back propagation.

NAME = 'sdk_fine_tune'

__init__()
Fine tune inference object constructor.

property fine_tune_params
FineTuneParams: The current params of this inference object.

set_fine_tune_params(*should_quantize_weights=False*, *should_relax_weights=False*, ...)
Set fine tune params.

See also:

The documentation of FineTuneParams contains additional details.

class hailo_sdk_common.targets.inference_targets.**SdkMixed**(**args*, ***kwargs*)
Bases: hailo_sdk_common.targets.inference_targets.EmulationObject

Mixed emulation target. Some layers will be emulated in *native* mode, while the rest will be emulated in *numeric* mode. This target is useful when researching the effect that quantization of specific layers has on the accuracy of the whole model.

NAME = 'sdk_mixed'

__init__()
SdkMixed inference object constructor.

property mixed_params
SdkMixedParams: The current params of this inference object.

set_mixed_params(*numeric_target='sdk_numeric'*)
Set mixed params.

See also:

The documentation of SdkMixedParams contains additional details.

class hailo_sdk_common.targets.inference_targets.**SdkMixedParams**(*numeric_target='sdk_numeric'*)
Bases: object

Parameters for SdkMixed target.

DEFAULT_NUMERIC_TARGET = 'sdk_numeric'

__init__(*numeric_target='sdk_numeric'*)
SdkMixed params constructor. :param numeric_target: :type numeric_target: EmulationObject, optional :param Which numeric emulation target to set for non-native layers.:

classmethod from_json(*json_str*)
Construct this class from previously exported JSON data.

Parameters json_str (str) – The input JSON data.

Returns The object constructed from the JSON data.

Return type SdkMixedParams

to_json()
Get a JSON representation of this object.

Returns A JSON dump.

Return type str

class hailo_sdk_common.targets.inference_targets.**FineTuneParams**(*should_quantize_weights=False*, ...)
Bases: object

Parameters for SdkFineTune target.

DEFAULT_QUANTIZE_WEIGHTS = False

DEFAULT_QUANTIZE_ACTIVATIONS = True

DEFAULT_RELAX_WEIGHTS = False

__init__(*should_quantize_weights=False*, *should_relax_weights=False*, *should_quantize_activations=True*)
Fine tune params constructor.

Parameters

- **should_quantize_weights** (bool, optional) – Indicates whether the weights should be quantized using fake quantization. A new trainable variable named kernel_delta is added to the graph when this option is turned on.
- **should_relax_weights** (bool, optional) – EXPERIMENTAL. If True, use gradual ("relaxed") quantization for weights fine-tuning instead of STE/fake-quant, exporting the weights' "distance from grid" tensor so that the client can penalize it in the loss function, slowly driving weights towards grid. Note that should_quantize_weights should still be True to use this mode.
- **should_quantize_activations** (bool, optional) – Indicates whether the activation should be quantized using fake quantization. A new trainable variable named fine_tune_bias is added to the graph when this option is turned on.

classmethod from_json(*json_str*)
Construct this class from previously exported JSON data.

Parameters **json_str** (str) – The input JSON data.

Returns The object constructed from the JSON data.

Return type FineTuneParams

to_json()
Get a JSON representation of this object.

Returns A JSON dump.

Return type str

Bibliography

[Meller2019] Eldad Meller, Alexander Finkelstein, Uri Almog and Mark Grobman. "Same, same but different: Recovering neural network quantization error through weight factorization." International Conference on Machine Learning, 2019. http://proceedings.mlr.press/v97/meller19a/meller19a.pdf

[Finkelstein2019] Alexander Finkelstein, Uri Almog and Mark Grobman. "Fighting quantization bias with bias." Conference on Computer Vision and Pattern Recognition Workshops, 2019. https://arxiv.org/pdf/1906.03193.pdf

[McKinstry2019] Jeffrey McKinstry, Steven Esser, Rathinakumar Appuswamy, Deepika Bablani, John Arthur, Izzet Yildiz and Dharmendra Modha. "Discovering Low-Precision Networks Close to Full-Precision Networks for Efficient Embedded Inference." Conference on Neural Information Processing Systems, 2019. https://www.emc2-ai.org/assets/docs/neurips-19/emc2-neurips19-paper-11.pdf

[Nagel2020] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos and Tijmen Blankevoort. "Up or Down? Adaptive Rounding for Post-Training Quantization." International Conference on Machine Learning, 2020. https://arxiv.org/pdf/2004.10568.pdf

[Vosco2021] Niv Vosco, Alon Shenkler and Mark Grobman. "Tiled Squeeze-and-Excite: Channel Attention With Local Spatial Context." International Conference on Computer Vision Workshops, 2021. https://openaccess.thecvf.com/content/ICCV2021W/NeurArch/papers/Vosco_Tiled_Squeeze-and-Excite_Channel_Attention_With_Local_Spatial_Context_ICCVW_2021_paper.pdf

Python Module Index

h

