



# Hailo SDK User Guide

Release 3.11.0

9 September 2021

# Table of Contents

<b>I</b>	<b>User Guide</b>	<b>2</b>
<b>1</b>	<b>Hailo SDK Overview</b>	<b>3</b>
1.1	Included in this package . . . . .	3
1.2	Introduction . . . . .	3
1.3	Model build process . . . . .	4
1.4	Deployment process . . . . .	5
<b>2</b>	<b>Changelog</b>	<b>6</b>
<b>3</b>	<b>Installation</b>	<b>20</b>
3.1	System requirements (SDK) . . . . .	20
3.2	Running SDK installation . . . . .	20
3.3	Updating the SDK . . . . .	22
3.4	SDK configuration file . . . . .	23
3.5	SDK, HailoRT and firmware versions compatibility . . . . .	23
<b>4</b>	<b>Tutorials</b>	<b>24</b>
4.1	SDK tutorials introduction . . . . .	24
4.2	Parsing tutorial . . . . .	24
4.3	Quantization tutorial . . . . .	26
4.4	Layer noise analysis tool tutorial . . . . .	29
4.5	Compilation tutorial . . . . .	35
4.6	Inference tutorial . . . . .	36
4.7	Multiple inputs and outputs tutorial . . . . .	39
<b>5</b>	<b>Building Models</b>	<b>45</b>
5.1	Translating Tensorflow and ONNX models . . . . .	45
5.2	Profiler and other command line tools . . . . .	53
5.3	Numeric translation . . . . .	56
5.4	Models compilation . . . . .	62
5.5	Supported layers . . . . .	69
<b>II</b>	<b>API Reference</b>	<b>77</b>
<b>6</b>	<b>Model Build API Reference</b>	<b>78</b>
6.1	hailo_sdk_client.runner.client_runner . . . . .	78
6.2	hailo_sdk_client.exposed_definitions . . . . .	87
6.3	hailo_sdk_client.hailo_archive.hailo_archive . . . . .	89
6.4	hailo_sdk_client.tools.hn_modifications . . . . .	89
6.5	hailo_sdk_client.quantization.tools.quant_aware_fine_tune . . . . .	89
6.6	hailo_sdk_client.tools.core_postprocess.core_postprocess_api . . . . .	93
6.7	hailo_sdk_client.tools.layer_noise_analysis . . . . .	95
6.8	hailo_sdk_client.tools.dead_channels_removal . . . . .	98
<b>7</b>	<b>Common API Reference</b>	<b>99</b>
7.1	hailo_sdk_common.export.hailo_graph_export . . . . .	99
7.2	hailo_sdk_common.model_params.model_params . . . . .	104
7.3	hailo_sdk_common.profiler.profiler_common . . . . .	104
7.4	hailo_sdk_common.preprocessing.base . . . . .	104
7.5	hailo_sdk_common.preprocessing.normalization . . . . .	104
7.6	hailo_sdk_common.hailo_nn.hailo_nn . . . . .	105
7.7	hailo_sdk_common.hailo_nn.hn_definitions . . . . .	106

7.8	<code>hailo_sdk_common.targets.inference_targets</code> . . . . .	106
<b>Python Module Index</b>		<b>112</b>

## Disclaimer and Proprietary Information Notice

### Copyright

© 2021 Hailo Technologies Ltd ("Hailo"). All Rights Reserved.

No part of this document may be reproduced or transmitted in any form without the expressed, written permission of Hailo. Nothing contained in this document should be construed as granting any license or right to use proprietary information for that matter, without the written permission of Hailo.

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 SDK Overview

## 1.1. Included in this package

This software package includes the following parts:

- SDK 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 SDK API is used for compiling users' models to Hailo binaries. The input of the SDK 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.

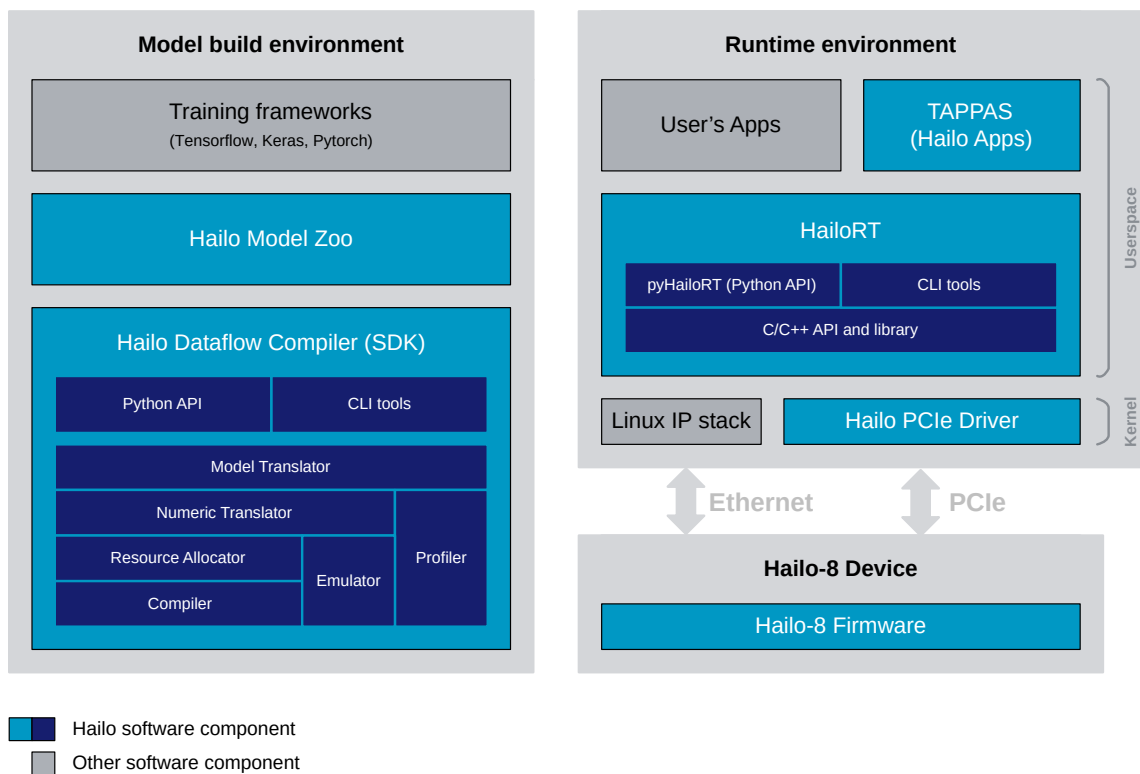


Figure 1: Detailed block diagram of Hailo software packages

## 1.3. Model build process

The Hailo SDK 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, numeric translation of the model's parameters, and compilation.

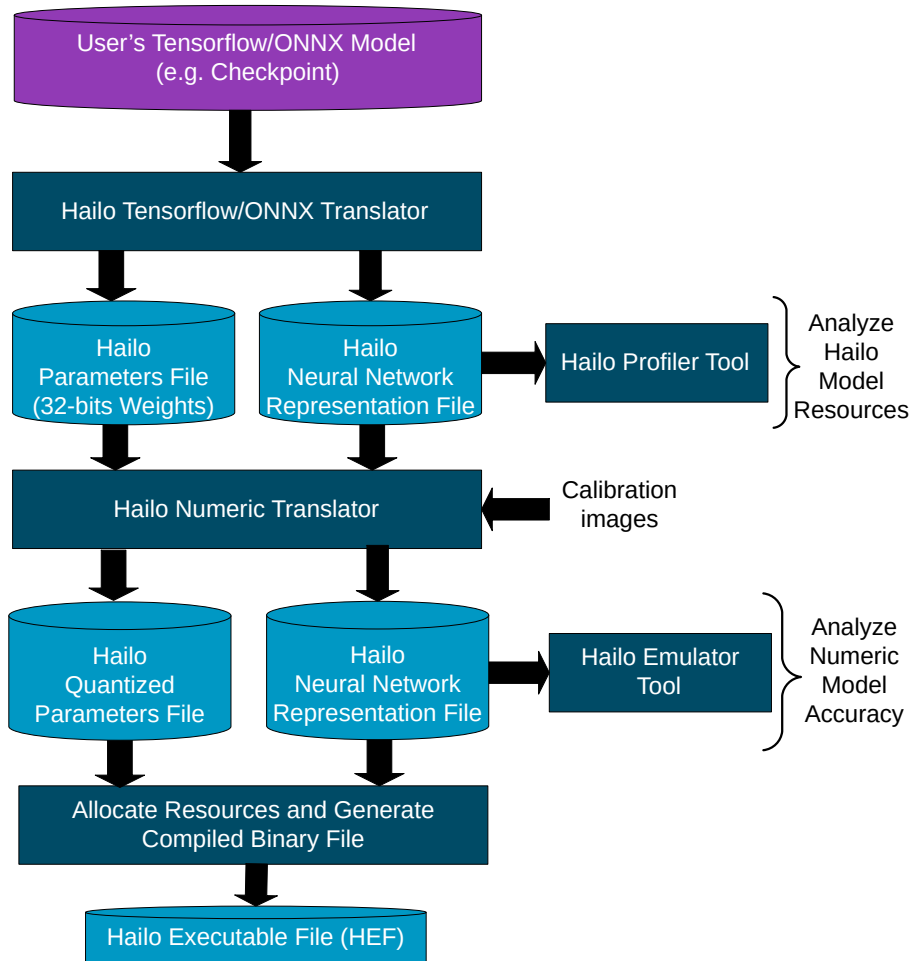


Figure 2: Model build process, starting in a Tensorflow or ONNX model and ending with a Hailo binary (HEF)

### 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 (HN format). The resulting 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 HN 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 SDK 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 quantized model's accuracy.

### 1.3.4. Numeric translation

After the user generates the HN 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 SDK Tool allocates hardware resources that match the FPS rate required by the user. 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 SDK or on a separate machine. A Yocto layer is provided to allow easy integration of HailoRT to embedded environments.



## 2. Changelog

### SDK 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

#### SDK 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.

---

## SDK v3.10.1 (August 2021)

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

## SDK 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

- SDK 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 SDK. 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.

---

## SDK v3.9.1 (July 2021)

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

## SDK 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

#### SDK 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 to such systems, this feature can be disabled during compilation using a [model script command](#).

---

### SDK 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 supported also in the TF and ONNX parsers

#### SDK 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.

## SDK v3.7.1 (April 2021)

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

## SDK 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 supported also in the TF and ONNX parsers

### Quantization

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

### SDK 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.

---

## SDK 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*.

### SDK 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.

---

### SDK 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

#### SDK API

- Changed the SDK 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.

---

### SDK v3.4.1 (February 2021)

- *Upgraded HailoRT and firmware*

## SDK 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

### SDK 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.

---

## SDK v3.3.0 (December 2020)

### Core

- *Elementwise Multiplication* support
- *"Dense like" to "Conv like" Reshape* support

### Models allocation

- New automatic allocation heuristic that creates *portals* without manual script commands

### HailoRT (HailoRT version 2.3 and firmware version 2.3)

- New HEF format support and new inference API that works with HEF binaries
- HailoRT moved to its own user guide. See this guide for the detailed changelog

### Parser

- Elementwise Multiplication support from both *Tensorflow* and *ONNX*
- *"Dense like" to "Conv like" Reshape* support, from both Tensorflow and ONNX
- Input normalization parsing support, from both Tensorflow and ONNX

### Quantization

- *4 bit quantization* and Quantization aware Fine Tuning (QFT) support
- *16 bit bias* is now the default mode for many layers
- Added quantization analysis *tool* and *tutorial*

### SDK client API

- HEF format support. This is the new compiled model binary file format
- Hailo archive format support. This new format encapsulates the HN, NPZ, allocation script and other relevant information in a single file

---

**Note:** The HEF format is currently supported only via the PCIe interface. It is expected to replace the JLF format in the following versions. The JLF format is still supported in this version, through both PCIe and Ethernet, but it is expected to be deprecated.

---

## SDK v3.2.0 (November 2020)

### Models allocation

- Improved allocation heuristics

### HailoRT (HailoRT version 2.2 and firmware version 2.2)

- Re-enabled the firmware configuration CLI tool

### Parser

- Upgraded to Tensorflow v1.15.4. This change affects the whole SDK, including the Tensorflow parser and emulator
- Improved error handling
- Multiple inputs support across the whole tool-chain

### Quantization

- Improved *quantization groups* support

## SDK v3.1.0 (October 2020)

### Core

- *Depthwise conv 3x3/1 dilation=4* support
- *Argmax De-fusing* support
- Extended the *bi-linear resizing* support

### Models allocation

- Added latency support to the *Profiler report*
- Several allocation optimizations

### HailoRT (HailoRT version 2.1 and firmware version 2.1)

- Re-enabled the Ethernet interface
- Production M.2 board support
- Several bug fixes and optimizations

### Quantization

- *Quantization groups* support
- Quantization accuracy improvement in some activation functions such as exp

### Parser

- Added an API to add *in-chip NMS* to models that belong to the SSD (Single Shot Detection) meta architecture
- Maxpool layers with VALID padding support in some cases (e.g. when the kernel is 2x2/2 and the image dimensions are even)
- Threshold activation and biased delta activation support in the ONNX parser
- Several bug fixes

**SDK v3.0.0 (August 2020)**

This is the first release that supports the Hailo-8 production device and the PCIe interface.

**Core**

- Hailo-8 production device support
- *16 bit bias support* in all kernels
- *Depthwise Conv 9x9/1* support
- *Conv 9x9/1* (and similar parameters) support
- Conv 3x3/1 dilation=4 support
- *Deconv spatial De-fusing*

**HailoRT (HailoRT version 2.0 and firmware version 2.0)**

- Hailo-8 production device support
- PCIe interface support in both C/C++ and Python APIs
- Added Hailo's PCIe driver

---

**Note:** SDK v2.x releases only supported the Hailo-8 sample device. v3.x releases will only support the Hailo-8 production device.

---

---

**Note:** The Ethernet interface and power related features are temporarily disabled in this version. They are expected to be re-enabled in future versions.

---

**SDK v2.12.1 (July 2020)****Firmware and Platform API (firmware version 1.13.0)**

- HailoRT now supports sensor related data formats such as Bayer and RGB888
- Argmax output format support in HailoRT
- Ethernet sync support in HailoRT
- Changed the Python inference API to use the HailoRT library in most cases

---

**Note:** The legacy inference implementation that does not use HailoRT is still available in this version. However, it is expected to be deprecated in future versions.

---

**SDK v2.12.0 (July 2020)****Core**

- Conv kernel optimization (skipping padding pixels when possible)
- *Delta activation* support

**Models allocation**

- Added more details to the Profiler report
- Improved de-fusing heuristics

**Quantization**



- Custom *bias feed repeat* support

#### Parser

- Delta activation support in the *TF Parser*
- New layers are supported in the *ONNX Parser*: Row by row Softmax, features to columns reshape, 1D Conv and Maxpool, and Nearest neighbor resize 1x2
- ONNX parser bug fixes

#### Examples and packaging

- Added Resnext-50 and Resnet-18 benchmarks
- Replaced the non standard Resnet-22-FCN-FHD benchmark with the standard Resnet-18-FCN-FHD benchmark
- The benchmarks now validate the UDP limit
- Added the new tutorials into the user guide
- New *command line tool* that runs the server

### SDK v2.11.0 (June 2020)

#### Core

- *"Full row" Conv* support, where the kernel width equals to the image width
- Extended "Conv and Add" support, where convolution and elementwise addition are fused together
- *Asymmetric Maxpool 1x2/2* support
- *Row by row Softmax* support
- *Features to columns reshape* support
- *Threshold activation* support
- *Nearest neighbor resize 1x2* support

#### Quantization

- `run_quantization_from_np()` memory optimization

#### Parser

- *ONNX Parser* Slice layer support
- ONNX Parser bug fixes
- *TF Parser* support for all new kernels
- TF Parser support for *1D Keras models* (Conv1D and Maxpooling1D)

#### Firmware and Platform API (firmware version 1.12.0)

- NMS output format support in HailoRT
- FCR (feature, column, row) format support in HailoRT, also known as NHWC

#### Examples and packaging

- Added new tutorials Jupyter notebooks
- Added the benchmarks JLF files (compiled models) to allow running them without having to compile first

**SDK v2.10.1 (May 2020)****Examples and packaging**

- Added Ubuntu 18.04 and Python 3.6 support
- Fixed the benchmarks' FPS calculation in full streaming mode

**SDK v2.10.0 (May 2020)****Core**

- *Conv with asymmetric stride* 1x2 and 2x1 support
- *Asymmetric Depth to Space* 1x2 and 2x1 support
- Standalone activation support (when it can't be fused to any layer before it)
- Deconv 2x2/2 support
- 16 bit bias support in *Deconv layers*

**Models allocation**

- New *script command* to break a big feature splitter layers into smaller steps

**Parser**

- *ONNX parsing support*
- TF Parser support for all new kernels (including Conv with asymmetric stride, asymmetric Depth to Space and standalone activation)
- Supporting TF placeholder with multiple successors as the Parser's start node

**Firmware and Platform API (firmware version 1.11.0)**

- New data formats support in HailoRT: flat (dense), Argmax and outputs multiplexing
- HailoRT data quantization support (from float32 and uint8 to uint8 for input data, and the reverse conversion for output data)
- New HailoRT APIs to extract input and output streams information from JLFs
- The UDP rate limiter now covers all UDP ports used by HailoRT
- Ethernet related bug fixes and stabilization in the firmware

**Examples and packaging**

- New benchmarks suite that replaces the examples
- New command line tools that wrap major parts of the SDK functionality, for example: quantization, compilation and inference
- Command line auto completion support

---

**Note:** This version includes several API changes:

- High level quantization API functions like `run_quantization()` moved to the `hailo_sdk_client.quantization.quantize` module.
- Emulation inference targets like `SdkNative` moved to the `hailo_sdk_common.targets.inference_targets` module.

Backwards compatibility support is still kept in this version, with deprecation warnings.

---

**SDK v2.9.0 (April 2020)****Core**

- Slice layer support (with static coordinates)
- Tanh activation support

**Models allocation**

- Added more details to the Profiler report

**Quantization**

- Iterative Bias Correction (IBC) algorithm running time and memory optimization

**Parser**

- Slice layer support
- Tanh activation support
- Tensorflow frozen models (frozen PB) support

**Firmware and Platform API (firmware version 1.10.0)**

- *HailoRT* (C API for inference) improvements
- Mini PCIe board support

**SDK v2.8.0 (March 2020)****Core**

- Features split support

**Models allocation**

- Automatic generation of allocation scripts

**Quantization**

- Activation clipping support
- 16 bit bias can be automatically set to all supported layers using a single quantization script command

**Parser**

- Features split support
- Several bug fixes

**Firmware and Platform API (firmware version 1.9.1)**

- First release of the *HailoRT* inference library with C API
- Power measurement enhancements

**SDK v2.7.0 (February 2020)****Core**

- Depthwise Conv 3x3/1 dilation=2 support
- 16x4 mode support in additional parameters of Conv and Depthwise Conv kernels (see the [full list of layers that support 16x4 mode](#))
- Conv 3x3/1 *no contexts* mode support
- Bug fix in case of global average pooling after non Relu activation

**Models allocation**

- Improved the accuracy of pre-placement profiling

**Quantization**

- Weights clipping support

**Examples and packaging**

- Added a new super resolution example

---

**Note:** The implementation of post-placement profiling has changed in this version. This might make post-placement profiling somewhat inaccurate for JLFs that had been compiled in older SDK versions.

---

**SDK v2.6.0 (December 2019)****Core**

- Max pooling 5x5/1 support
- Conv 3x3/1 dilation=2 support
- Conv 3x1, 5x1 and 7x1 support
- Conv kernel optimizations

**Firmware and Platform API**

- Added support for averaging power samples on the DVM chip (increasing power measurements precision)
- Host side dataflow optimizations

**Models allocation**

- Improved terminal output and logging

**Parser**

- Upgraded to Tensorflow v1.13.1. This change affects the whole SDK, including Tensorflow parser and emulator
- Fixed several bugs

**Quantization**

- Quantization aware fine tuning support for both kernels and biases
- Fixed several implementation bugs in the Equalization and IBC algorithms

**Emulator**

- Nvidia Turing GPUs support

**SDK client API**

- HailoNN objects import and export support from the client runner

**Examples and packaging**

- Added new examples: Yolact (instance segmentation), Mobilenet SSD FPN and two variants of EfficientNet

**SDK v2.5.1 (November 2019)****Core**

- Depthwise convolution 5x5/1 and 5x5/2 kernel support

**Firmware and Platform API**

- M.2 board support
- The firmware and platform software now support keeping platform (board) details in the firmware configuration to distinguish the evaluation board from the M.2 board

**Models allocation**

- Printing minimum and maximum inter-layer buffers sizes for each layer to assist manual allocation

**Models compilation**

- Minor bug fixes in the compiler

**Examples and packaging**

- Resnet\_v1\_18\_FCN16 example now runs bilinear resizing and argmax on the device (instead of the host)

**SDK v2.5.0 (November 2019)****Core**

- Conv 3x3 stride 2 with VALID padding support
- Bilinear resizing improvements
- Depthwise convolution kernel optimization

**Firmware and Platform API**

- New fw-control command line tools (identify, reset, scan)
- Better handling of control packet-loss
- Support higher rates by optimizing the “multi-rows” dataflow option
- Platform software package for ARM targets, tested on IMX8

**Models allocation**

- New algorithm for splitting layers between clusters
- Fixed inter-layer buffers sizes calculation
- Improved De-fusing heuristics
- Improved resources estimation accuracy

**Models compilation**

- Major compiler refactoring, which shortens model compilation time

**Quantization**

- Added “quantization scripts” to modify quantization parameters without the need to manually modify HN files
- New quantization parameters: maximum number of element-wise addition feed repeats and null channels elimination threshold
- Improved the quantization algorithm to deal with negative slope errors

**Examples and packaging**

- New examples: Mobilenet-v2, Resnet-v1-50 based face verification, OpenPose based multi-person pose estimation

**SDK v2.4.0 (September 2019)****Core**

- Feature shuffle kernel support
- Softmax support
- Exp activation support

**Firmware and Platform API**

- Improved handling of control drops and latency
- Updated power measurement API to return the number of measured samples
- Improved power measurement averaging calculation

**Models allocation**

- Major improvements in clusters splitting algorithm
- Automatic heuristic for spatial defusing
- Allowing manual overriding of inter-layer buffer sizes using allocation scripts

**Parser**

- Argmax support
- Feature shuffle support
- Softmax support
- Exp activation support
- Various bug fixes

**Quantization**

- High level quantization API refactoring
- Fixed several quantization bugs and improved the error messages

**Examples and packaging**

- Upgrading existing Python virtual environments to newer SDK versions is supported
- Adding new example networks: Resnet22-FCN16-FullHD, Simple-Pose, Resnext26-32x4d and Shufflenet-g8-w1
- Adding profiling and compiling examples

## 3. Installation

### 3.1. System requirements (SDK)

The Hailo SDK requires the following minimum hardware and software configuration:

- Ubuntu 18.04, 64 bit
- 16+ GB RAM (32+ GB recommended)
- Physical connection, depending on the board:
  - PCIe interface or Gigabit Ethernet (802.3) interface for the evaluation board
  - M.2 connector for the M.2 board
  - mPCIe connector for the mPCIe board
- Python 3.6, including pip and virtualenv
- python3.6-dev, python3-tk, graphviz, and libgraphviz-dev packages

The following additional requirements are optional:

- build-essential package (needed to compile the PCIe driver)
- bison, flex, libelf-dev and dkms packages (needed to register the PCIe driver using DKMS)
- cmake (needed to compile the HailoRT example)
- USB web camera (used by several demos)

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

- Nvidia Pascal or Turing GPU (such as Titan X Pascal, GTX 1080 Ti or RTX 2080 Ti)
- GPU driver version 465
- CUDA 11.0
- CUDNN 8.0.4

---

**Note:** The SDK 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.

---

---

**Note:** These requirements are for the SDK, which is used to build models. Running inference using the HailoRT runtime library works on smaller systems as well. See the HailoRT user guide for more details.

---

### 3.2. Running SDK installation

The SDK package includes 3 files:

- hailo\_sdk.tar.gz – Archive of the SDK binaries and Python modules.
- install.sh – Installation script.
- md5s.txt – MD5 hashes of the SDK files.

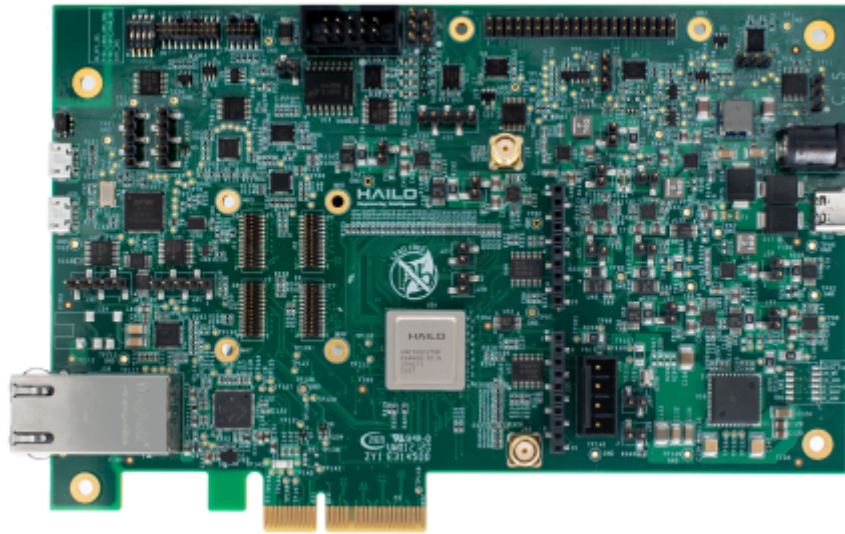


Figure 3: Hailo-8 Evaluation board



Figure 4: Hailo-8 M.2 board



Figure 5: Hailo-8 mPCIe board



### 3.2.1. Installation with the PCIe driver

Run the following command to install the SDK including the PCIe driver:

```
chmod u+x install.sh
./install.sh
```

This script extracts the tar.gz archive and installs the SDK in a new Python virtual environment. It will also compile the PCIe driver for the machine's kernel version and install the driver. The command will ask for root permissions (sudo) during the installation in order to install the driver.

Reboot the machine after the installation is done. The driver will be loaded automatically after reboot. Run the following commands in order to verify it:

```
source hailo_virtualenv/bin/activate
hailo fw-control scan
```

The scan command should find the device.

---

**Note:** The PCIe driver is not signed. In some systems it means that secure boot has to be disabled to load the driver.

---



---

**Note:** By default, the installation creates a new Python virtual environment, which requires an Internet connection (or a local pip server) in order to download Python packages.

---

### 3.2.2. Installation without the PCIe driver

Alternatively, run the following command to install the SDK without the PCIe driver:

```
chmod u+x install.sh
./install.sh --skip-pcie-driver
```

### 3.2.3. Additional installation flags

- The `--help` flag is used to display the all of the usage options of the installation script.
- Use the `--skip-dkms` flag to skip the DKMS installation.

## 3.3. Updating the SDK

When receiving a new version of the SDK, the update can be installed onto:

- A new virtualenv.
- An existing virtualenv created by a previous installation of the Hailo SDK.
- An existing virtualenv created by the user.

In order to update the SDK in an existing virtualenv, follow these steps:

1. Change the install script permission (as described in the [Running SDK installation](#) section)
2. Run the `install.sh` script with the following parameters:
  - `--update-virtualenv` (**Mandatory**): Causes the installation script to update onto an existing virtualenv. This parameter requires the path to the target virtualenv you wish to install on.
  - `--skip-pcie-driver` (**Optional**): Skips the PCIe driver installation.

3. The success of the update is validated by running `pip freeze`. Hailo's packages should be of the latest version.

---

**Note:** The path provided to the `--update-virtualenv` parameter should be the virtualenv directory (the one containing *bin*, *include*, *lib*), and not the project's root directory.

---

---

**Note:** If the firmware is loaded from flash rather than via the PCIe driver, the device's firmware should be upgraded just before upgrading the SDK. See the table below to know which SDK version is compatible with which firmware version.

---

### 3.4. SDK configuration file

The SDK's high-level configuration is defined in a file named `config`.

A default configuration file is included in the SDK package. It is copied to `~/.hailo` during installation. If the config file already exists at `~/.hailo`, the user will be prompted. The Python package searches for the configuration using the following conditions: If a file named `config` exists in the current working directory, it is used. Otherwise, they look at `~/.hailo/config`.

### 3.5. SDK, HailoRT and firmware versions compatibility

Each SDK version matches a specific HailoRT and Hailo-8 firmware version. The details are listed in the following table. When upgrading the SDK, the firmware should also be upgraded accordingly.

Table 1: SDK, HailoRT and Firmware versions compatibility

SDK version	HailoRT version	Firmware version
v3.0.0	v2.0.0	v2.0.0
v3.1.0	v2.1.0	v2.1.0
v3.2.0	v2.2.0	v2.2.0
v3.3.0	v2.3.0	v2.3.0
v3.4.0	v2.4.0	v2.4.0
v3.4.1	v2.5.0	v2.5.0
v3.5.0	v2.5.1	v2.5.1
v3.6.0	v2.6.0	v2.6.0
v3.7.0	v2.7.0	v2.7.0
v3.7.1	v2.7.0	v2.7.0
v3.8.0	v2.8.0	v2.8.0
v3.9.0	v2.9.0	v2.9.0
v3.9.1	v2.9.0	v2.9.0
v3.10.0	v2.9.0	v2.9.0
v3.10.1	v2.10.0	v2.10.0

## 4. Tutorials

The tutorials below go through the model build and inference steps. They are also available as Jupyter notebook files in the directory *tutorials/notebooks/*.

### 4.1. SDK tutorials introduction

The following tutorials cover the basic use-cases of the Hailo SDK:

**Model compilation:**

1. Converting Tensorflow 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 mode.
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 model and ending with an hardware deployed model.

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

#### 4.1.1. Installation

The SDK should be installed before running the code in this guide.

The installation script creates a virtual environment which the SDK depends on, so this notebook should run after activating this virtual environment by running: `source hailo_virtualenv/bin/activate`.

## 4.2. Parsing tutorial

### 4.2.1. Hailo parsing example from Tensorflow CKPT to HN

This tutorial will walk you through parsing Tensorflow checkpoints to the HN format. HN is a JSON-like representation of the graph structure that is deployed to the Hailo hardware.

**Requirements:**

- Run the notebook inside the SDK virtual environment: `source hailo_virtualenv/bin/activate`

```
[ ]: %matplotlib inline
from IPython.display import SVG
from hailo_sdk_client import ClientRunner, NNFramework
from hailo_sdk_common.preprocessing import Normalization
```

Choose the checkpoint files to be used throughout the example:

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

Open Tensorboard visualization to verify the start and end nodes of the graph (first and last nodes that are deployed to hardware):

This command is blocking so interrupt the kernel when you are done.

```
[ ]: # !hailo tb {ckpt_path}
# press Kernel -> Interrupt to proceed
start_node = 'resnet_v1_18/conv1/Pad'
end_node = 'resnet_v1_18/predictions/Softmax'
```

The main API of the SDK that the user interacts with is the ClientRunner class. To parse a new checkpoint, use the translate\_tf\_model method:

```
[ ]: runner = ClientRunner(hw_arch='hailo8')
hn, npz = runner.translate_tf_model(ckpt_path, model_name, start_node_name=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.

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 HN 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} --fps 1000
```

## 4.2.3. Normalization on chip

The following example shows how to fuse a preprocessing stage of normalization to the Hailo-8 device:

```
[ ]: mean_list = [123.675, 116.28, 103.53]
std_list = [58.395, 57.12, 57.375]
normalization_obj = Normalization(mean=mean_list, std=std_list)
hn, npz = runner.translate_tf_model(ckpt_path, model_name, start_node_name=start_node, end_node_
↪ names=[end_node],
                                integrated_preprocess=normalization_obj)
runner.save_har(hailo_model_har_name)
!hailo visualizer {hailo_model_har_name} --no-browser
SVG('out.svg')
```

## 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 = '../ckpt/dense_example_tf2/saved_model.pb'

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

## 4.3. Quantization tutorial

### 4.3.1. Hailo quantization example from HN and NPZ files

This tutorial will walk you through quantizing your model. The input to this tutorial are HN and NPZ files (with native weights) and the output will be NPZ file with quantized weights.

#### Requirements:

- Run the notebook inside the SDK virtual environment: `source hailo_virtualenv/bin/activate`
- Verify that you've run through the Parsing Tutorial (or created the HN+NPZ in other way)

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

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 ParamsKinds, SdkNative, SdkNumeric
```

Choose a Hailo Archive file in a "Hailo Model" (parsed model) state 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 quantization:

Note that the pre-processing is model and dataset dependent. It also depends if the input tensors normalization is offloaded to the Hailo-8 device. Here we assume the Imagenet dataset and that the normalization is offloaded.

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

def preproc(image, output_height=224, output_width=224, resize_side=256):
```

(continues on next page)

(continued from previous page)

```
''' 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)])#, align_corners=False)
        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(os.path.join(images_path, img_name))[1] == '.jpg']
calib_dataset = np.zeros((len(images_list), 224, 224, 3), dtype=np.float32)
for idx, img_name in enumerate(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')
plt.title('Original image')
plt.show()
plt.imshow(np.array(calib_dataset[idx,:,:,:], np.uint8), interpolation='nearest')
plt.title('Preprocessed image')
plt.show()
```

Run the quantization process:

Note that this will run naive quantization, for advanced topics in quantization please refer to the advanced section below.

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

runner.quantize(calib_dataset, batch_size=2, calib_num_batch=16)

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

Evaluate the results by comparing the results of the native and the numeric emulation modes:

```
[ ]: 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):
    results = []
    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())
```

(continues on next page)

(continued from previous page)

```

        print(sdk_export.output_tensors[0])
        probs_batch = sdk_export.session.run(sdk_export.output_tensors[0], 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('Native: 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[:5,:,:,:]
native_res = net_eval(runner, SdkNative(), images)
numeric_res = net_eval(runner, SdkNumeric(), images)

res = show_results(images, native_res, numeric_res)

```

### 4.3.2. Advanced quantization - 4-bit weights and Finetuning

If the network is challenging and/or we want to compress weights below 8 bits, we'll need the Quantization Aware Finetuning (QFT) to get decent results.

Here we set some layers to be 4 bit weights using the a8\_w4 param in Quantization Script, quantize again and show that results are poor without QFT:

```

[ ]: alls_lines = ['quantization_param(conv14, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n',
                  'quantization_param(conv15, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n',
                  'quantization_param(conv16, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n',
                  'quantization_param(conv17, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n',
                  'quantization_param(conv18, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n',
                  'quantization_param(conv19, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n',
                  'quantization_param(conv20, precision_mode=a8_w4, bias_mode=double_scale_
↪initialization)\n']
# -- Reduces weights memory by ~40% !

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

runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)
runner.quantize(calib_dataset, calib_num_batch=4,
                model_script='quant_script.alls')

```

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

      res = show_results(images, native_res, numeric_res)
```

Now, repeating the same process with QFT employed in the course of quantization:

```
[ ]: from hailo_sdk_client.quantization.tools.quant_aware_fine_tune import FineTuneConfigurator

      runner = ClientRunner(hw_arch='hailo8', har_path=hailo_model_har_name)

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

      runner.quantize(calib_dataset, calib_num_batch=6, batch_size=16,
                      model_script='quant_script.all', ft_batch_size=16,
                      finetune=True, ft_cfg=FineTuneConfigurator({'dataset_size': 96}))

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

      top_inds, top_inds_q = show_results(images, native_res, numeric_res)
```

```
[ ]: print(top_inds, top_inds_q)
      sum(np.array(top_inds)==np.array(top_inds_q)) / len(top_inds_q)
```

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

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

## 4.4. Layer noise analysis tool tutorial

This tutorial will guide you through a model quantization analysis, using Hailo's QuantizationAnalyzer tool that essentially breaks down the quantization noise per layer. The tool's input are the model's HN and NPZ files (with Native 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 as to how to use Hailo's QuantizationAnalyzer 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 HN, NPZ 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 Native. This highlights the quantization sensitivity of the model to that specific layer.

### Requirements:

- Run the notebook inside the SDK virtual environment: `source hailo_virtualenv/bin/activate`
- Verify that you've run through the Parsing tutorial (to generate the parsed model Hailo Archive) and the Quantization tutorial (to generate the calibration set) or create them in another way.

```
[ ]: %matplotlib inline
      import os
      import numpy as np
```

(continues on next page)



(continued from previous page)

```
import matplotlib as mpl
from matplotlib import pyplot as plt
# from past.utils import old_div
from hailo_sdk_common.targets.inference_targets import ParamsKinds
from hailo_sdk_common.targets.inference_targets import SdkNumeric
from hailo_sdk_client.tools.layer_noise_analysis import QuantizationAnalyzer
from hailo_sdk_client.tools.layer_noise_analysis_utils.layer_noise_analysis_utils import ResultsMode
from hailo_sdk_client.tools.layer_noise_analysis_utils.figure_generator import _
↳ QunatAnalyzerFigureGenerator
from hailo_sdk_client import ClientRunner
```

#### 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 Hailo model (parsed model) Hailo Archive file

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

If you wish, set your working directory (via the work\_dir variable), to specify the path to data saving. This is optional, and can be left as work\_dir=None, in which case data will be saved in a directory named work\_dir that will be 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. Model Loading and Quantization

Now, create a ClientRunner and load it with the saved Hailo Archive file.

```
[ ]: runner = ClientRunner(har_path=hailo_model_har_path)
```

Once the runner has been set, we are ready for some analysis. First, we create an instance of QuantizationAnalyzer and pass it the created runner, the data set path and (optionally) the working directory path. Also, set its batch size for quantization and inference.

```
[ ]: batch_size = 8
calib_set_size = 64
analyzer = QuantizationAnalyzer(runner, data_path=data_path, calib_path=calib_path, calib_set_
↳ size=calib_set_size, batch_size=batch_size, work_dir=work_dir)
```

Now we can perform quantization of the entire model, with the provided calibration set defined above. This can be done by evoking Analyzer's quantize\_runner() method.

**Note:** In this tutorial, we quantize the Resnet-v1-18 model such that one layer (conv7) is quantized at 4 bits. This is done using the helper function create\_resnet\_example\_all() defined below to create the wanted quantization script, which is then fed into quantize\_runner(). Also, note that for the purposes of this tutorial the calibration path and data path are identical, and for the calibration set we take the first 64 images present in this dataset.

**Note:** The Analyzer supports the Equalization and IBC algorithms when the quantization is done via the analyzer. quantize\_runner() method. Currently the Analyzer does not support the the Fine Tune algorithm.

```
[ ]: def create_resnet_example_all(s_path):
    """Creates a quantization script with conv7 quantized at 4 bits for the resnet-v1-18 example."""
    with open(s_path, 'w') as alls:
        alls.write("quantization_param({conv7}, precision_mode=a8_w4)\n")
    #=====

    alls_path = './resnet_v1_18_quantization_script.all(s)'
    create_resnet_example_all(s_path)

    # Perform Model Quantization
    equalize = False
    ibc = False

    analyzer.quantize_runner(equalize=equalize, ibc=ibc, quantization_script_filename=all(s_path))
```

#### 4.4.3. Optional: Setting post processing and evaluation callbacks to get an additional per degradation analysis

In addition to the noise analysis, The Analyzer is able to perform a per layer degradation analysis. This can be done by setting the Analyzer with pre-defined callbacks for the post-processing and evaluation functions, that should meet certain criteria. Note that evaluation requires the ground truth information, which should be constructed in the data feed. Below is an example for the classification model analyzer here, showing the main important ingredients each callback should have.

**Note:** As conv7 is quantized to 4bit in this tutorial, note the sharp peak in the per layer degradation plots when running the analysis below

```
[ ]: from collections import OrderedDict, namedtuple
    from functools import partial
    import tensorflow as tf

    # Define a named tuple holding information about the evaluation results

    EvalResult = namedtuple('EvalResult', ['value', 'name', 'is_percentage', 'is_bigger_better'])

    # Helper function to contrsuct the data feed iterator form the data .npy and label_index infomration
    def _get_iterator(np_array, label_index=None, batch_size=8):
        dataset = tf.data.Dataset.from_tensor_slices((np_array, label_index))
        dataset = dataset.batch(batch_size)
        return tf.compat.v1.data.make_initializable_iterator(dataset)

    ##### Evaluation Callback Setting #####
    # Defining the class for evaluation according to the following structure

    class ClassificationEval:
        def __init__(self, **kwargs):
            self.metric_names = ['top1', 'top5']
            self.metrics_vals = [0, 0]
            self.reset()

        def update_op(self, net_output, img_info):
            label_index = img_info['label_index']
            self.top1 += \
                list(np.argmax(net_output, 1) == label_index)
            top5 = np.argsort(net_output, -5, axis=1)[: , -5:]
            self.top5 += \
                [label_index[batch_idx] in top5[batch_idx, :]]
```

(continues on next page)

(continued from previous page)

```

        for batch_idx in range(len(label_index))]

    def evaluate(self):
        self.metrics_vals[0] = np.mean(self.top1)
        self.metrics_vals[1] = np.mean(self.top5)

    def get_accuracy_dict(self):
        return OrderedDict([(self.metric_names[0], self.metrics_vals[0]),
                             (self.metric_names[1], self.metrics_vals[1])])

    def get_accuracy(self):
        result_dict = self.get_accuracy_dict()
        return [EvalResult(value=value,
                            name=key,
                            is_percentage=True,
                            is_bigger_better=True)
                for key, value in result_dict.items()]

    def reset(self):
        self.top1 = []
        self.top5 = []

eval_cb = ClassificationEval

#####          Post Processing Callback Setting          #####

# Defining the postprocessing function
def _classification_postprocessing(endnodes):
    ''' Assumes here endnodes are post SoftMax values '''
    probs = endnodes
    return probs

post_proc_cb = _classification_postprocessing

#####   Data Feed with Annotation Information Callback Setting   #####

# Building the data feed with ground truth annotation information
calib_set = np.load(calib_path) # Load the dataset
calib_set_labels_path = '../data/calib_set_labels.npz'
calib_set_labels = np.load(calib_set_labels_path) # load annotations NPZ file

# calib_set_labels is a dict with a 'label_index' key being the class index label,
# and a 'label_class' key being the name of the class.
# The former is picked and fed to construct the data feed iterator callback

label_index = [int(label) for label in calib_set_labels['label_index']]
labels = {'label_index': label_index}
data_feed_cb = partial(_get_iterator, calib_set, label_index=labels, batch_size=batch_size)

#####          Setting the callbacks to the Analyzer          #####

analyzer.set_data_feed_cb(data_feed_cb)
analyzer.set_postprocess_cb(post_proc_cb)
analyzer.set_eval_cb(eval_cb)

```

#### 4.4.4. A ~One Liner Default Quantization Analysis

The Analyzer can operate in two type of run modes, controlled via the names in the Enum ResultsMode:  
 \* Quantization analysis run for the globally quantized network (mode=ResultsMode.full\_quant\_net.name or 'full\_quant\_net') \* A layer by layer quantization analysis run (mode=ResultsMode.layer\_by\_layer.name or 'layer\_by\_layer'))

The user is able to run a default full analysis with a simple command, invoking Analyzer's run\_analysis method, by passing it either mode=ResultsMode.full\_quant\_net.name for global quantization analysis or mode=ResultsMode.layer\_by\_layer.name or a layer by layer quantization analysis. At the end of the analysis, all analysis figures will be plotted.

If a ResultsMode.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.

**Following this cell is a breakdown of the behind the stages processes that are carried out, to assist the user to gain more flexibility and control over the analysis**

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

**Note 2:** This run will produce quite a few plots. A deatiled explanation of each as well as how to produce those separately is given later in this notebook.

```
[ ]: %%time

analyzer.show_figures = True
%matplotlib inline

mode = ResultsMode.full_quant_net.name
analyzer.run_analysis(mode=mode)

mode = ResultsMode.layer_by_layer.name
analyzer.run_analysis(mode=mode)
```

#### 4.4.5. Analysis of the Globally Quantized Model

Here show how one can obatin the actual noise results data and perform a more customized ananlysis. To get hold of the raw data results from each run mode, one can invoke the specific Analyzer method. Here, we perform just the "fully quantized" run and calculate the quantization noise of the activations at each of the model's layers. The Native vs. Numeric quantization noise is calculated by several metrics:

- Cosine Distance
- Signal to Noise Ratio (SNR)
- Mean noise
- Maximum L1 noise
- Mean of per channel L1 noise

The output produced here is a mean over the images in the test set, for each noise measuring metric. To get the analysis results, we invoke QuantizationAnalyzer's analyze\_full\_quant\_model().

```
[ ]: full_quant_net_noise_results, full_quant_net_sampled_activations, full_quant_net_results = analyzer.
      ↪ analyze_full_quant_model()
      print("Done calculating noise for a 'full_quant_net' run")
```

After all images are processed (126 in this tutorial), we can plot the results using a plotter object (a QunatAnalyzer-FigureGenerator instance) that takes as arguments the results produced from analysis run and ResultsMode.full\_quant\_net.name as the run mode. To show the mean per layer noise porfiles, we use plotter draw\_noise\_results function. The plots depict the mean activation noise (according to the metric) at each layer, for the fully quantized model.

**Note:** In this tutorial, the Resnet-v1-18 model has one layer (conv7) whose weights are quantized at 4 bits. This can be appreciated by the plots produced below, especially the Cosine Distance, SNR and Mean per channel L1 noises, depicting a steep quantization noise rise (e.g. note the sharp sudden SNR drop at this layer)

```
[ ]: analyzer.show_figures = True
      %matplotlib inline

      mode = ResultsMode.full_quant_net.name
      plotter = QunatAnalyzerFigureGenerator(analyzer, full_quant_net_results, full_quant_net_noise_results,
                                              full_quant_net_sampled_activations, mode)
      plotter.draw_noise_results()
```

#### 4.4.6. Layer by Layer Noise Analysis

Here, we explore the quantization sensitivity of each layer separately, by calculating the quantization noise of a given layer's output when that layer is the **only** one quantized, while all other layers are kept in Native. This flow repeats for all model layers, quantizing a different layer at each iteration, with the quantization noise metrics as previously done above. To perform this analysis, we invoke QuantizationAnalyzer's analyze\_layer\_by\_layer method.

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

When the layer sensitivity run is done, we plot the results using another plotter instance, with mode=ResultsMode.layer\_by\_layer.name, and call its draw\_noise\_results function again. As before, the plots depict the mean activation noises at each layer. Also plotted is the quantization noise in the model outputs (a.k.a logits) for each single quantized layer.

```
[ ]: %%time
      %matplotlib inline
      layer_by_layer_noise_results, layer_by_layer_sample_activations, layer_by_layer_results = analyzer.
      ↪analyze_layer_by_layer()

      mode = ResultsMode.layer_by_layer.name
      plotter = QunatAnalyzerFigureGenerator(analyzer, layer_by_layer_results, layer_by_layer_noise_results,
                                              layer_by_layer_sample_activations, mode)
      plotter.draw_noise_results()
```

#### 4.4.7. Native vs. Numeric Scatter plots

Here, we generate scatter plots for the data in the sampled activations per channel with the plotter draw\_scatter\_results method. The layers for which the data would be plotted can be specified (or left as None/not passed to draw\_scatter\_results), as well as a list of channels for each layer. We produce such plots here for layers conv1, conv2 and conv3 and the first 16 channels of each layer, producing subplots of nrows=1 X ncols=2 dimensions.

**Note:** If the number of layers analyzed is larger than nrows \* ncols, several plots will be generated.

```
[ ]: channel_list = np.arange(16) # If None, All channles are analyzed
      layers = ['conv1', 'conv2', 'conv3'] # If None, scatter plots for all layer activations
      # requested to be sampled/analyzed are generated
      # layers = None # deaful value

      plotter.draw_scatter_results(layers=layers, channels=channel_list, nrows=1, ncols=2)
```

#### 4.4.8. Activation distribution

Let's have a look at the activations distribution (based on the sampled data) by using the plotter `draw_hist` method. We can create a Probability Density Plot (histogram) or a Cumulative Probability Density plot, by setting `cumulative=True`. Another option is to break down the distribution by channels (by setting `channel_wise=True`). Similarly to the above `draw_scatter_results`, we can set which layers to depict and the subplots format (i.e. `nrows`, `ncols`) at each produced figure.

```
[ ]: cumulative = True
channel_wise = True
layers = ['conv1', 'conv2', 'conv3'] # If None, distribution plots for all layer activations
# requested to be sampled/analyzed are generated
# layers = None # default value

plotter.draw_hist(layers=layers, cumulative=cumulative, channel_wise=channel_wise, nrows=3, ncols=1)
```

#### 4.4.9. Output results to .pdf report file

Finally, we can output the results depicted in the generated figures into a single .pdf file, by using `QuantizationAnalyzer`'s `create_pdf` method.

```
[ ]: pdf_report_name = '_' .join([analyzer._network_name, 'noise_report'])
analyzer.create_pdf(file_name=pdf_report_name)

assert os.path.exists('_' .join([pdf_report_name, 'pdf'])), "Error! .pdf report file was not generated."
↪:/"
```

### 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 the notebook inside the SDK virtual environment: `source hailo_virtualenv/bin/activate`

```
[ ]: 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)
fps = 1200
```

Load the network to the `ClientRunner`:

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

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

```
[ ]: hef = runner.get_hw_representation(fps=fps)

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

Run the profiler tool (in post-placement mode):

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 --hef {model_name}.hef
```

## 4.6. Inference tutorial

This tutorial will walk you through the inference process.

### Requirements:

- Run the notebook inside the Python virtual environment: `source hailo_virtualenv/bin/activate`
- Run the [Compilation Tutorial](#) before running this one.

### 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 the Hailo platform SW. This way we can use the Hailo hardware without Tensorflow, and even without the Hailo SDK (after the HEF is built).

A HEF is Hailo's binary format for neural networks. The HEF files contain:

- Target HW configuration
- Weights
- Metadata for the platform SW (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(hw_arch='hailo8')

# Loading compiled HEFs to device:
model_name = 'resnet_v1_18'
hef_path = f'{model_name}.hef'
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
# 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_from_network_group(network_group, quantized=False,
                                                                    format_type=FormatType.FLOAT32)
output_vstreams_params = OutputVStreamParams.make_from_network_group(network_group, quantized=True,
```

(continues on next page)

(continued from previous page)

format\_type=FormatType.UINT8)

```
# Define dataset params
input_layer = hef.get_input_layers_info()[0]
output_layer = hef.get_output_layers_info()[0]
image_height, image_width, channels = input_layer.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_layer.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)
        print(f'Stream output shape is {infer_results[output_layer.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_from_network_group(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_from_network_group(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, define the processes, recreate the target and run processes:

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

(continues on next page)



(continued from previous page)

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

# The target used as a context manager
with PcieDevice() as target:
    configure_params = ConfigureParams.create_from_hef(hef, interface=HailoStreamInterface.PCIe)
    network_group = target.configure(hef, configure_params)[0]
    network_group_params = network_group.create_params()
    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 quantization 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
import numpy as np

from hailo_sdk_client import ClientRunner
from hailo_platform import PcieDevice, HEF

model_name = 'resnet_v1_18'

quantized_model_har_path = f'{model_name}_quantized_model.har'
runner = ClientRunner(hw_arch='hailo8', har_path=quantized_model_har_path)

hef_path = f'{model_name}.hef'
hef = HEF(hef_path)

# Define dataset params
input_layer = hef.get_input_layers_info()[0]
image_height, image_width, channels = input_layer.shape
num_of_images = 10
low, high = 2, 20
```

The Tensorflow graph is generated using the preloaded compiled 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.

```
[ ]: with PcieDevice(hw_arch='hailo8') as target:
    network_groups = target.configure(hef)

    graph = tf.Graph()
    with graph.as_default():
        network_input = tf.compat.v1.placeholder(dtype=tf.float32)
```

(continues on next page)

(continued from previous page)

```

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)

```

## 4.7. Multiple inputs and outputs tutorial

The following tutorial covers the use-case of multiple inputs and outputs in a single device. Throughout this guide we will run two networks, Resnet-v1-18 and MobileNet SSD in parallel.

### Requirements:

- Run the notebook inside the SDK virtual environment: `source hailo_virtualenv/bin/activate`

### 4.7.1. Parsing

Each network is contained in a Tensorflow checkpoint

Each Tensorflow checkpoint will be parsed into a HN format, which is a JSON-like representation of the graph structure that is deployed to the Hailo hardware. The HAR, which is saved later in this tutorial, contains the HN.

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 = 'resent_v1_18'
ckpt_path = '../ckpt/resnet_v1_18.ckpt'

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

runner1 = ClientRunner(hw_arch='hailo8')
hn, npz = runner1.translate_tf_model(ckpt_path, model_name, start_node_name=start_node, end_node_
names=[end_node])

# Parse MobileNet SSD model
model_name = 'mobilenet_ssd'
ckpt_path = '../ckpt/mobilenet_ssd.ckpt'

start_node = 'FeatureExtractor/MobilenetV1/MobilenetV1/Conv2d_0/Conv2D'

```

(continues on next page)

(continued from previous page)

```
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')
hn, npz = runner2.translate_tf_model(ckpt_path, model_name, start_node_name=start_node, end_node_
↪ names=end_nodes)
```

## 4.7.2. Quantization

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 “Quantization 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 = '../ckpt/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_name=start_node, end_node_names=end_
↪nodes)
runner4.translate_tf_model(ckpt_path, model_name2, start_node_name=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_name=start_node, end_node_names=end_
↪nodes)
runner4.translate_tf_model(ckpt_path, model_name2, start_node_name=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.get_hw_representation(fps=100)

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(hw_arch='hailo8')

# 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_from_network_group(network_group, quantized=False,
    format_type=FormatType.FLOAT32)
output_vstreams_params = OutputVStreamParams.make_from_network_group(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_layers_info()}

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():
            print(f'Stream output shape in output {output_layer_name} is {result.shape}')
```

## Standalone hardware deployment - Streaming inference

```
[ ]: from multiprocessing import Process

input_data = {layer.name: np.ndarray([1] + list(layer.shape), dtype=np.float32)
               for layer in hef.get_input_layers_info()}

def send(configured_network, num_images):
    vstreams_params = InputVStreamParams.make_from_network_group(configured_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)

def recv(configured_network, num_images):
    vstreams_params = OutputVStreamParams.make_from_network_group(configured_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()

with PcieDevice() as target:
    configure_params = ConfigureParams.create_from_hef(hef, interface=HailoStreamInterface.PCIe)
    network_group = target.configure(hef, configure_params)[0]
    network_group_params = network_group.create_params()
    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 (hef=\'{model_name}\', num_of_images={num_of_images})')

    with network_group.activate(network_group_params):
        send_process.join()
        recv_process.join()
    print('Done')
```

## Inference using TensorFlow

```
[ ]: import tensorflow as tf

with PcieDevice(hw_arch='hailo8') as target:
    network_groups = target.configure(hef)

    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_layers_info()}
        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]
```

(continues on next page)

(continued from previous page)

```
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_layers_info():
            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

#### 5.1.1. Using the Tensorflow Parser

The SDK Tensorflow parser supports Tensorflow v1.15.4, which includes Keras v2.2.4-tf, and Tensorflow v2.4.1, which includes Keras v2.4.0.

Translating Tensorflow models is done 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 SDK distinguishes between them automatically based on the file extension, but this decision can be overridden using the `is_frozen` flag.
- TF2 models – savedmodel 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 since they do not require additional parser support.

Table 2: Supported Tensorflow APIs (layers)

API name	Comments
<code>tf.nn.conv2d</code>	
<code>tf.concat</code>	
<code>tf.matmul</code>	
<code>tf.avg_pool</code>	
<code>tf.nn.maxpool2d</code>	
<code>tf.nn.depthwise_conv2d</code>	
<code>tf.nn.depthwise_conv2d_native</code>	
<code>tf.nn.conv2d_transpose</code>	<ul style="list-style-type: none"><li>• Only SAME_TENSORFLOW padding</li></ul>
<code>tf.reduce_max</code>	<ul style="list-style-type: none"><li>• Only on the features axis and with <code>keepdims=True</code></li></ul>
<code>tf.reduce_mean</code>	
<code>tf.reduce_sum</code>	<ul style="list-style-type: none"><li>• Only with <code>keepdims=True</code></li></ul>
<code>tf.contrib.layers.batch_norm</code>	
<code>tf.image.resize_images</code>	<ul style="list-style-type: none"><li>• The new size has to be const (not a tensor)</li><li>• Only one of the following methods:<ul style="list-style-type: none"><li>- Nearest neighbor (integer scale only)</li><li>- Bi-linear (<code>align_corners</code> must be True)</li></ul></li></ul>

Continued on next page



Table 2 – continued from previous page

API name	Comments
<code>tf.image.resize_bilinear</code>	<ul style="list-style-type: none"> <li>The new size has to be const (not a tensor)</li> <li><code>align_corners</code> must be True</li> </ul>
<code>tf.image.resize_nearest_neighbor</code>	<ul style="list-style-type: none"> <li>The new size has to be const (not a tensor)</li> <li>Integer scale only</li> </ul>
<code>tf.image.crop_to_bounding_box</code>	<ul style="list-style-type: none"> <li>Only static cropping, i.e. the coordinates cannot be data dependent</li> </ul>
<code>tf.image.resize_with_crop_or_pad</code>	<ul style="list-style-type: none"> <li>Only static cropping without padding, i.e. the coordinates cannot be data dependent</li> </ul>
<code>tf.nn.bias_add</code>	
<code>tf.add</code>	<ul style="list-style-type: none"> <li>Only one of the following: <ul style="list-style-type: none"> <li>Bias add</li> <li>Elementwise add</li> <li>As a part of input tensors normalization</li> <li>Const scalar addition</li> </ul> </li> </ul>
<code>tf.multiply</code>	<ul style="list-style-type: none"> <li>Only one of the following: <ul style="list-style-type: none"> <li>Elementwise Multiplication layer</li> <li>Const scalar multiplication</li> </ul> </li> </ul>
• As a part of input tensors normalization	
<code>tf.subtract</code>	<ul style="list-style-type: none"> <li>Only for input tensors normalization, or const scalar subtraction</li> </ul>
<code>tf.divide</code>	<ul style="list-style-type: none"> <li>Only for input tensors normalization, or const scalar division</li> </ul>
<code>tf.negative</code>	
<code>tf.pad</code>	
<code>tf.reshape</code>	<ul style="list-style-type: none"> <li>Only in specific cases, for example: <ul style="list-style-type: none"> <li>Features to Columns Reshape layer</li> <li>Between Conv and Dense layers (in both directions)</li> <li>As a part of layers such as Feature Shuffle and Depth to Space</li> </ul> </li> </ul>
<code>tf.nn.dropout</code>	
<code>tf.depth_to_space</code>	
<code>tf.nn.softmax</code>	
<code>tf.argmax</code>	
<code>tf.split</code>	<ul style="list-style-type: none"> <li>Only in the features dimension</li> </ul>
<code>tf.slice</code>	<ul style="list-style-type: none"> <li>Only static cropping, i.e. the coordinates cannot be data dependent</li> </ul>
Slicing ( <code>tf.Tensor.__getitem__</code> )	<ul style="list-style-type: none"> <li>Only sequential slices (without skipping)</li> <li>Only static cropping, i.e. the coordinates cannot be data dependent</li> </ul>
<code>tf.nn.space_to_depth</code>	

Table 3: Supported Tensorflow APIs (activations)

API name	Comments
<code>tf.nn.relu</code>	
<code>tf.nn.sigmoid</code>	

Continued on next page

Table 3 – continued from previous page

API name	Comments
tf.nn.leaky_relu	
tf.nn.elu	
tf.nn.relu6	
tf.nn.softplus	
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

Table 4: 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 since they do not require additional parser support.

Table 5: 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 6: Supported Keras APIs

API name	Comments
layers.Conv1D	
layers.Conv2D	
layers.Conv2DTranspose	
layers.Dense	

Continued on next page

Table 6 – continued from previous page

API name	Comments
layers.MaxPooling1D	
layers.MaxPooling2D	
layers.GlobalAveragePooling2D	
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

### Group conv parsing

Tensorflow v1.15.4 has no group conv operation. The Hailo SDK recognizes the following pattern and converts it automatically 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 SDK recognizes the following pattern of sequential ops and converts it automatically 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 given 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 ONNX Parser

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

## Supported ONNX operations

Table 7: Supported ONNX operations (layers)

Operator name	Comments
Add	<ul style="list-style-type: none"> <li>Only one of the following: <ul style="list-style-type: none"> <li>Bias add</li> <li>Elementwise add</li> <li>As a part of input tensors normalization</li> <li>Const scalar addition</li> </ul> </li> </ul>
ArgMax	
AveragePool	
BatchNormalization	
Concat	
Conv	<ul style="list-style-type: none"> <li>Depthwise convolution is also implemented by this ONNX operation</li> </ul>
ConvTranspose	

Continued on next page

Table 7 – continued from previous page

Operator name	Comments
DepthToSpace	<ul style="list-style-type: none"> <li>• DCR mode only</li> </ul>
Div	<ul style="list-style-type: none"> <li>• Only for input tensors normalization, or const scalar division</li> </ul>
Dropout	
Flatten	<ul style="list-style-type: none"> <li>• Only in specific cases such as between Conv and Dense layers</li> </ul>
Gemm	
GlobalAveragePool	
MatMul	
MaxPool	
Mean	
Mul	<ul style="list-style-type: none"> <li>• Only one of the following: <ul style="list-style-type: none"> <li>– Elementwise Multiplication layer</li> <li>– Const scalar multiplication</li> <li>– As a part of input tensors normalization</li> <li>– As a prt of several activation functions, see below</li> </ul> </li> </ul>
Neg	
Pad	
ReduceMax	<ul style="list-style-type: none"> <li>• Only on the features axis and with keepdims=True</li> </ul>
ReduceMean	
ReduceSum	<ul style="list-style-type: none"> <li>• Only with keepdims=True, or as a part of a Softmax layer</li> </ul>
Reshape	<ul style="list-style-type: none"> <li>• Only in specific cases, for example: <ul style="list-style-type: none"> <li>– Depth to Space layer</li> <li>– Feature Shuffle layer</li> <li>– Features to Columns Reshape layer</li> <li>– Between Conv and Dense layers (in both directions)</li> </ul> </li> </ul>
Resize	<ul style="list-style-type: none"> <li>• Only Nearest Neighbor and Bi-linear resizing</li> <li>• Only align_corners is supported as a coordinate_transformation_mode</li> <li>• Opset 11 only</li> </ul>
Slice	
Softmax	
Split	<ul style="list-style-type: none"> <li>• Only in the features dimension</li> </ul>
Sub	<ul style="list-style-type: none"> <li>• Only for input tensors normalization, or const scalar subtraction</li> </ul>
Transpose	<ul style="list-style-type: none"> <li>• Only in specific cases, for example: <ul style="list-style-type: none"> <li>– Depth to Space layer</li> <li>– Feature Shuffle layer</li> <li>– Features to Columns Reshape layer</li> <li>– Dense like to Conv like Reshape layer</li> </ul> </li> </ul>
Upsample	<ul style="list-style-type: none"> <li>• Only Nearest Neighbor resizing</li> </ul>

Table 8: 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
Elu	
Exp	
Greater	• Only as a part of a Threshold activation parsing
LeakyRelu	
Mul	• Only as a part of a Threshold or Delta activation parsing (and several non activation layers, see above)
Relu	
Sigmoid	
Sign	• Only as a part of the Delta activation parsing
Softplus	
Tanh	

## 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 9: Supported PyTorch APIs (layers)

API name	Comments
<code>torch.nn.AvgPool2d</code>	
<code>torch.nn.BatchNorm1d</code>	
<code>torch.nn.BatchNorm2d</code>	
<code>torch.nn.Conv1d</code>	
<code>torch.nn.Conv2d</code>	
<code>torch.nn.ConvTranspose2d</code>	
<code>torch.nn.Dropout2d</code>	Does nothing on inference
<code>torch.nn.Flatten</code>	
<code>torch.nn.functional.interpolate</code>	Only <code>align_corners</code> is supported when <code>mode='bilinear'</code>
<code>torch.nn.functional.pad</code>	
<code>torch.nn.Linear</code>	
<code>torch.nn.MaxPool1d</code>	
<code>torch.nn.MaxPool2d</code>	
<code>torch.nn.Parameter</code>	
<code>torch.nn.Softmax</code>	
<code>torch.nn.Softmax2d</code>	
<code>torch.nn.Upsample</code>	• Only one of the following methods: - Nearest neighbor (integer scale only) - Bi-linear ( <code>align_corners</code> must be True)
<code>torch.nn.UpsamplingBilinear2d</code>	
<code>torch.nn.UpsamplingNearest2d</code>	

Continued on next page

Table 9 – continued from previous page

API name	Comments
<code>torch.argmax</code>	
<code>torch.cat</code>	
<code>torch.max</code>	See Supported ONNX operations: ReduceMax
<code>torch.mul</code>	See Supported ONNX operations: Mul
<code>torch.reshape</code>	See Supported ONNX operations: Reshape
<code>torch.split</code>	See Supported ONNX operations: Split
<code>torch.sum</code>	See Supported ONNX operations: ReduceSum
<code>torch.transpose</code>	See Supported ONNX operations: Transpose

Table 10: Supported PyTorch APIs (activations)

API name	Comments
<code>torch.abs</code>	See Supported ONNX operations: Abs
<code>torch.gt</code>	See Supported ONNX operations: Greater
<code>torch.sign</code>	See Supported ONNX operations: Sign
<code>torch.nn.LeakyReLU</code>	
<code>torch.nn.ReLU</code>	
<code>torch.nn.PReLU</code>	
<code>torch.nn.ReLU6</code>	
<code>torch.nn.Sigmoid</code>	
<code>torch.nn.Softplus</code>	
<code>torch.nn.Tanh</code>	

### 5.1.3. Layers order 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.

### 5.1.4. 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.

## 5.2. Profiler and other command line tools

### 5.2.1. Using Hailo command line tools

The Hailo SDK 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 will print the help message of the Profiler:

```
hailo profiler --help
```

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

- The `hailo parser` command line tool is used to [translate Tensorflow and ONNX models](#).
- The `hailo quantize` command line tool is used to [quantize models](#).
- The `hailo compiler` command line tool is used to [compile models](#) into hardware representation.

### 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 --fps 30 network.har
```

The user has to set the path of the HAR file to profile. There may be additional optional parameters needed such as the target frames per second (FPS) rate.

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

- **pre\_placement** – Several steps of resources calculation and optimization 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 inspecting an existing compilation (HEF). Simulation of the hardware microcontrollers is used to profile the expected performance. This mode is the most accurate especially in terms of FPS.

### 5.2.3. Understanding the Profiler report

The Hailo profiler report consists of five parts:

- **Model Details** – Presents the major attributes of the neural network.
- **Profiler Input Settings** – Presents the target device and the required throughput.
- **Performance Summary** – Presents the performance figures of the network on target hardware.
- **Device Utilization** – Presents the percentage of the device(s) resources to be used by the target network.
- **Model Description** – Presents the performance for each layer of the network.

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** Total operations per input image.

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

## Profiler Input Settings

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

**Optimization Goal** The method used by the SDK to profile the required hardware resources.

**Profiler Mode** The mode used to run the Profiler. Currently the supported modes are "pre-placement" and "post-placement".

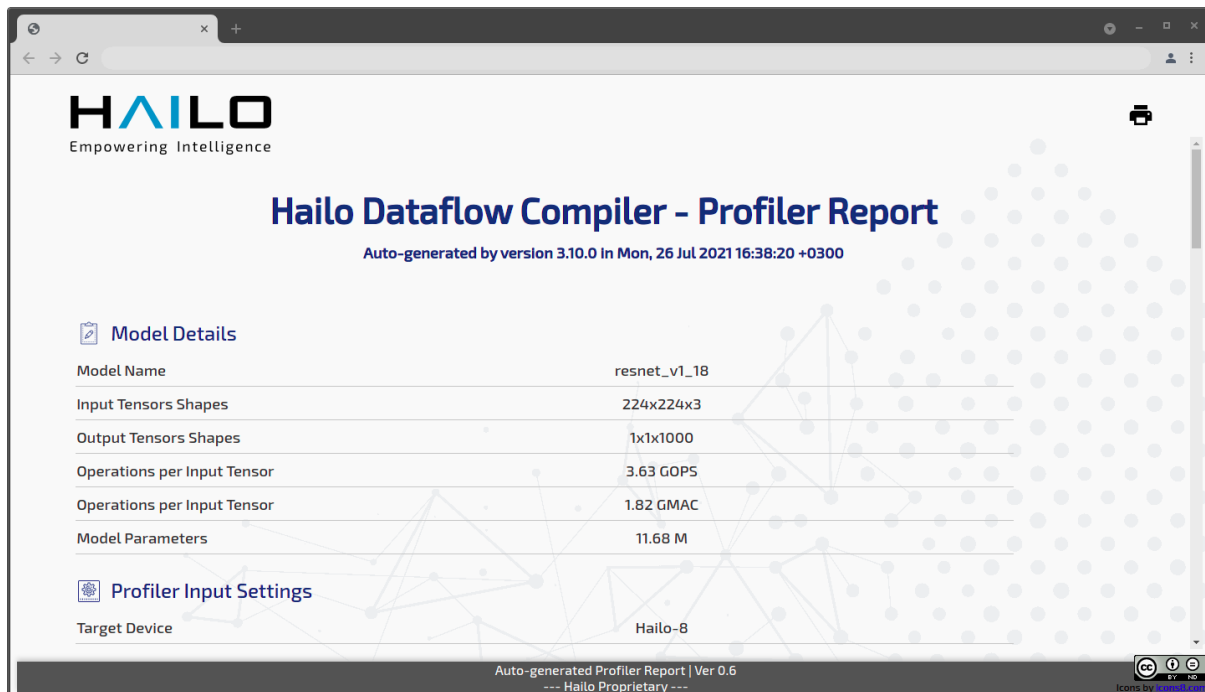


Figure 6: Profiler report screenshot

## Performance Summary

**Number of Devices** The number of required devices estimated by the SDK.

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

**Latency** The number of milliseconds that 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** The total operations per second based on the throughput (FPS) required by the user.

**Net Input Interface Throughput** The model's total input tensor throughput (bytes per second), without any hardware related overhead, based on the FPS rate required by the user.

**Gross Input Interface Throughput** The model's total input tensor throughput (bytes per second), with 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), without 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.

**Current Bottleneck Layer** The layer that limits the overall network throughput, per the resources currently allocated by the Profiler.

---

**Note:** The figures are expected to change when asking for a different FPS rate. This is due to a different amount of hardware resources that will be allocated for each layer.

---

## Device Utilization

**Compute Usage** The percentage of the device(s) compute resources to be used by the target network.

**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.

**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 by the last three figures.

---

## Model Description

For each layer, the following fields are defined:

**Layer Name** The name of this layer, as defined in the HN

**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 [GMAC]** 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 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 due to a different amount of hardware resources that will be allocated for each layer.

---

## 5.3. Numeric translation

Translating the models' parameters numerically, typically from 32 bit floating point to 8 bit integer, is also known as quantization. This is a mandatory step in order to run models on Hailo hardware. This step takes place after translating the model from its original framework and before compiling it.

### 5.3.1. Basic quantization flow

The `quantize()` method serves as the quantization API. This method includes the basic quantization flow and three additional algorithms: Equalization, Iterative Bias Correction (IBC), and Fine Tune.

This API requires sample images (typically 16-64, unless Fine Tune is used). The sample images are used to collect statistics. After the statistics are collected, they are used to quantize the weights.

**See also:**

The [quantization tutorial](#) shows how to use the quantization API.

### 5.3.2. Quantization scripts

The quantization script is a text file that contains per-layer commands that affect their quantization. Use of this file is optional.

Loading the script is done before the quantization by using the `load_model_script()` method or passing the file as an `model_script` argument to `quantize()` method

The model script supports 3 quantization commands

1. `quantization_param`
2. `pre_quantization_optimization`
3. `post_quantization_optimization`

---

**Note:** A single script file may contain both [allocation related](#) commands and quantization related commands. Allocation related commands are ignored during quantization. Quantization related commands are only validated during allocation. This is to make sure that they do not contradict the existing already quantized model. Quantization related commands should appear first in the script, before any allocation related commands.

---

#### `quantization_param`

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

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

For example:

```
quantization_param(conv1_d0, bias_mode=double_scale_initialization)
```

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

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

will change the bias mode of all the layers whose name starts with conv.

The supported parameters are listed below.

### **bias\_mode**

When this parameter is set to `double_scale_initialization`, it says that the layer should use 16 bit to represent the bias weight of the layer, instead of decomposing it as `UINT8*INT8*UINT4`. 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 many layers.

### **precision\_mode**

This command changes the precision of layers' activation and weights. The default mode is `a8_w8`. It means 8 bit activations and 8 bit weights. It can be changed to `a8_w4` in order to set the weights to 4 bit mode in order to reduce memory consumption.

Example command:

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

### **null\_channels\_cutoff\_factor**

This is only applicable to 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 is only applicable 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 better accuracy. When using this commands, weights of layers with more than one quantization group will be 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

The pre\_quantization\_optimization are optimization that are applied to the graph before the precision reduction. They are applied on float32 params in “full precision” mode.

The syntax of each pre\_quantization\_optimization command in the script is as follows:

```
pre_quantization_optimization(feature, param=value)
```

For example:

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

## weights\_clipping

By default, the Hailo SDK does not clip the weights during quantization. This command allows changing this behavior for selected layers and applying weights clipping when running the quantization API. Both manual values and percentiles are supported. In manual mode the values are used as is, while in percentile mode layer-wise percentiles are calculated. 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.

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])
```

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

## activation\_clipping

By default, the Hailo SDK 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. Both manual values and percentiles are supported. In manual mode the values are used as is, while in percentile mode layer-wise percentiles are calculated. This command may be useful in order to decrease quantization related degradation in case of outlier activation values.

**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, mode=manual, clipping_values=[0.188, 1.3332])
pre_quantization_optimization(activation_clipping, mode=percentile, clipping_values=[0.5, 99.5])
```

## 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, and currently supports the following configuration:

Parameter	Supported values	Default	Notes
policy	enabled, disabled	enabled	Configures default behavior for equalization algorithm

Example command:

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

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

## equalization (per layer)

This sub-command allows to configure the equalization behavior per layer, currently supports the following configuration:

Parameter	Supported values	Default	Notes
policy	enabled, allowed, disabled	allowed	Configures equalization behavior for a single layer. when allowed, global equalization policy is used.
layers	List[str] of layer names	None	Layers to configure. If not given, behaves as global equalization configuration

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)

## se\_optimization

This sub-command 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>

This command supports the following parameters:

Parameter	Supported values	Notes
method	tse	Configures default behavior for equalization algorithm
mode	sequential, custom	Layers to configure. If not given, behaves as global equalization configuration
count	int	Has to be field with mode=sequential. Number of S&E blocks to optimize (starting at the beginning of the graph)
layers	List[str] of global average pool layers	Global average pool layers of the S&E to optimize
tile_height	List[int] or int	Tile height for the tse tiling. Default is 7. The length of the list should match count / length of layers.

```
# 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])
```

(continues on next page)

(continued from previous page)

```
# 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 native inference results. 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>

## post\_quantization\_optimization

### 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, and currently supports the following configuration:

Parameter	Supported values	Default	Notes
policy	enabled, disabled	disabled	Configures default behavior for bias correction algorithm

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>

### bias\_correction (per-layer)

This sub-command allows to enable or disable the Iterative Bias Correction (IBC) algorithm on a per-layer basis. The supported parameters of this command are as follows:

Parameter	Supported values	Default	Notes
policy	enabled, disabled, allowed	allowed	Configures bias correction behavior for a single layer. when allowed, global bias correction policy is used.
layers	List[str] of layer names	None	Layers to configure. If not given, behaves as global bias correction configuration

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)
```



## 5.4. Models compilation

### 5.4.1. Basic compilation flow

When working with 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()`.

When working without Tensorflow, calling `get_hw_representation()` compiles the model without loading it to the device. Instead, the binary that contain the compiled model, named HEF, is returned to the user.

Both functions support the `fps` argument to set the required throughput in frames per second.

**See also:**

The [compilation tutorial](#) shows how to use the `get_hw_representation()` API.

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.

### 5.4.2. Allocation scripts

Allocation scripts hint the SDK 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 `get_hw_representation()` methods of the `ClientRunner` class using the `allocator_script_filename` parameter.

For example:

```
compiled_model = client_runner.get_hw_representation(allocator_script_filename='x.all')
```

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

---

**Note:** A single script file may contain both allocation related commands and [quantization related](#) commands. Allocation related commands are ignored during quantization. Quantization related commands are only validated during allocation. This is to make sure that they don't contradict the existing already quantized model.

---

#### Context switch parameters

##### Definition

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

##### Example

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

**Description** This command enables context switch and sets several parameters related to it:

- `mode` – Context switch mode. Set to `automatic` to enable context switch. Automatic partition of the given model to several contexts will be applied.
- `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. It 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 accomodate it. Defaults to `True`. Set to `False` to ensure the HEF compatibility to platforms that don't support it, such as Ethernet based platforms.

## 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

```
a, b, c = defuse(maxpool1, 2)
conv4a, conv4b, conv4concat = defuse(conv4, 2, defuse_type=SPATIAL)
```

**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+1`. The last returned layer (in the first example it is named `c`) is the concat. When defusing a conv layer, if no type is defined, then the default value will be feature defuse. In case a dimension defuse is needed, the `defuse_type` should be `SPATIAL`.

The following layers can be defused:

- Conv – features or spatial
- Deconv – features or spatial
- Maxpool – features
- Depthwise conv – features
- Dense – features
- Argmax – spatial
- Bilinear resize – features
- NN resize – features or spatial

## 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 11: 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	9x9	1x1	1x1	SAME SAME_TENSORFLOW
DW	3x3	1x1	1x1, 2x2	SAME SAME_TENSORFLOW
DW	5x5	1x1	1x1	SAME SAME_TENSORFLOW

Table 12: 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.

When the command is used without the `layers_to` argument, the new layer is added between `layer_from` and all its successors.

## 5.5. Supported layers

The following section describes the layers and parameters range that the SDK supports.

**Note:** Unless specified differently, supported features in this chapter describe their support across the SDK components (Profiler, Allocator, Compiler, Emulator and Quantization).

**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 4 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 13: 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) 8x8 (stride=1x1 only)	SAME SAME_TENSORFLOW VALID
2x2	2x1	1x1	SAME SAME_TENSORFLOW VALID
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)
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

Continued on next page



Table 13 – continued from previous page

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
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
1xW	1x1	1x1	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 14: Convolution &amp; 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 two tensors addition only.

**Note:** Max weights per layer < 8MB (for all Conv layers)

## 5.5.2. Max Pooling

Table 15: Max pooling kernel supported parameters

Kernel (HxW)	Stride (HxW)	Padding
2x2	1x1, 2x1, 2x2	SAME SAME_TENSORFLOW VALID
1x2	1x2	SAME SAME_TENSORFLOW VALID

Continued on next page

Table 15 – continued from previous page

Kernel (HxW)	Stride (HxW)	Padding
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. Kernel sizes and strides that are mentioned elsewhere in the table are excluded from this list.

### 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 will be 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 16: 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
3xW	3xW	SAME SAME_TENSORFLOW VALID
5xW	5xW	SAME SAME_TENSORFLOW VALID
7xW	7xW	SAME SAME_TENSORFLOW VALID

Continued on next page

Table 16 – continued from previous page

Kernel (HxW)	Stride (HxW)	Padding
Global	n/a	n/a

W means the width of the layer's input tensor. In other words, in this case the kernel width equals to the image width.

**Note:** In global average pooling, only  $F < 2048$  cases are supported, and above 1024 features, only  $F \% 128 = 0$  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 17: 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
2x2	2x2	SAME_TENSORFLOW
1x1	1x1	SAME_TENSORFLOW

### 5.5.7. Depthwise Convolution

Table 18: 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
5x5	1x1, 2x2	1x1	SAME SAME_TENSORFLOW
9x9	1x1	1x1	SAME SAME_TENSORFLOW

**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

Elementwise Multiplication requires two input tensors with the same shape.

---

**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

Add operation is 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. When possible, elementwise addition is fused into a Conv layer as detailed above. Elementwise addition is supported on both "Conv like" and "Dense like" tensors.
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

Resize can use the Nearest Neighbor method. This method is supported in three cases:

1. When the columns scale is any integer power of two and the rows scale is any integer between x1 and x4096.
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”.

Resize can also use the Bi-linear method which supports up to x16 upscale or downscale. In both methods the rows and columns scale can be different.

### 5.5.16. Depth to Space

Depth to space rearranges data from depth (features) into blocks of spatial data.

Table 19: 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 Yolo v5. 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 two 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.

### 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).

### 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  $x \geq \text{threshold}$  else 0.
- Delta, defined by 0 if  $x == 0$  else const.

Activations are usually fused into the layer before them, but they are also supported as standalone layers when they can't be fused.

### 5.5.28. Note about symmetric padding

The Hailo SDK supports symmetric padding as supported by other frameworks such as Caffe. Since 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 SDK):

```
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='hailo8', ...)
```

Bases: object

Hailo SDK API client.

```
__init__(hn=None, hw_arch='hailo8', 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_quantization_script(quantization_script_filename)
```

SDK API for manipulation of quantization and allocation params in an existing HN model. This method loads a script and applies it to the existing HN, i.e., modifies the specific params in each layer.

**Warning:** This function is deprecated, use `load_model_script()` instead.

**Parameters** `quantization_script_filename` (str) – Model script allowing the modification of the current model, prior to quantization / native emulation / profiling, etc. The script is parsed and a modified HN is set, where each layer (possibly) has new quantization params.

**Returns** A copy of the modified HN (JSON dictionary).

**Return type** dict

**load\_model\_script**(*model\_script\_filename*)

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_filename` (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. 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 quantization.
2. 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, start\_layer=None, end\_layer=None, 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()`.
- **start\_layer** (str) – Name of the first layer to equalize.
- **end\_layer** (str) – Name of the last layer to equalize.
- **()** (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 saved are native, native after BN fusion, or quantized.

**sort\_params**(*path=None*)

Sort all model params according to quantization groups and save to file.

**Parameters** **path** (str, optional) – If given, the sorted params are saved as a new .npz file with the given path.

**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. 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.
- **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 SDK uses the default value. Defaults to None.
- **use\_preloaded\_compilation** (bool, optional) – Use the HEF loaded to the Hailo-HWObject. 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) – Allocation script allowing fine tuning of allocation. If present, the Allocator parses it command-by-command and executes.
- **node** (tf.Tensor) – Last Tensorflow node of the pre-processing stage. This node represents the input of the SDK (deprecated, see nodes argument above).
- **network\_groups** (list, optional) – A list of network groups received from configure() must be given when use\_preloaded\_compilation is True.

**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=4, 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 SDK uses the default value. 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) – Allocation script allowing fine tuning of allocation. If present, the allocator parses it command-by-command and executes.

**Returns** Data of the HEF that contains the hardware representation of this model.

**Return type** bytes

#### Example

```
>>> runner = get_example_runner()
>>> compiled_model = runner.get_hw_representation(fps=100)
```

**hef\_infer\_context**(*hailo\_export*)

**translate\_onnx\_model**(*onnx\_path, net\_name, start\_node\_name=None, end\_node\_names=None, ...*)

SDK API for parsing an ONNX model. This creates a runner with loaded HN (model) and parameters.

#### Parameters

- **onnx\_path** (str) – Path to 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\_name** (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.
- **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.
- **integrated\_preprocess** ([Preprocess](#), optional) – A preprocessing API object that adds hardware-integrated preprocessing.
- **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\_shape** (list, optional) – The shape of the inputs to the net, used to set in the ONNX graph. Use only when the original model has dynamic input shapes (described with a wildcard denoting each dynamic axis, e.g. [b, c, h, w]).
- **start\_node\_names** (list of str, optional) – List of ONNX nodes that parsing will start from. If this parameter is specified, start\_node\_name should remain empty.
- **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]). If this parameter is specified, net\_input\_shape should remain empty.

**Note:** Using a non-default start\_node\_name or 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**(*ckpt\_path*, *net\_name*, *start\_node\_name*=None, *end\_node\_names*=None, ...)

SDK API for parsing a TF model given by a checkpoint/pb/savedmodel file. This creates a runner with loaded HN (model) and parameters.

#### Parameters

- **ckpt\_path** (str) – Path of the file to parse, either a checkpoint (without the final .meta), frozen graph or TF2 savedmodel. Use None if original\_model\_path is already set.
- **net\_name** (str) – Name of the new HN to generate.
- **start\_node\_name** (str, optional) – Name of the first Tensorflow node to parse. For example, if the model starts with a placeholder followed by a convolution layer, start\_node\_name should contain the name of the convolution operation.
- **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.
- **integrated\_preprocess** ([Preprocess](#), optional) – A preprocessing API object that adds hardware-integrated preprocessing. For example:

```
>>> from hailo_sdk_common.preprocessing import Normalization
>>> runner.translate_tf_model('example.ckpt', #.....
...     integrated_preprocess=Normalization(mean=[0.8, 0.7, 0.6],
...                                             std=[0.1, 0.2, 0.3]))
```

- **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.
- **is\_frozen** (bool, optional) – A flag that indicates that the input network is a frozen graph. By default the decision is done according to the extension of ckpt\_path.
- **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.
- **nn\_framework** ([NNFramework](#), optional) – NN Framework identifier that allows the user to indicate from which framework the model was exported. If not specified, the framework will be auto-detected. The supported values are TENSORFLOW and TENSORFLOW2, here's example with TF2 SavedModel:

```
>>> from hailo_sdk_client import NNFramework
>>> runner.translate_tf_model('tf2_example/saved_model.pb', #.....
...     nn_framework=NNFramework.TENSORFLOW2)
```

**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: '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** (str, optional) – Scope name for the layers of this model.
- **scope2\_name** (str, optional) – Scope name for the layers of the other model.
- **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\_hn\_model**(fps, 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 [get\\_hw\\_representation\(\)](#) 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.

**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.

**Return type** tuple

#### Example

```
>>> runner = get_example_runner()
>>> summary, details = runner.profile_hn_model(fps=100)
```

**get\_mapped\_graph**(*hw\_mapping\_file*)

SDK API for retrieving model mapping as protobuf.

**Parameters** **hw\_mapping\_file** (str) – Path where the mapping is saved.

**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.

**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\_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.



**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.

**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.numpy: 'np\_array'>, work\_dir=None, ...)**

Quantize model's params, using optional pre-process and post-process algorithm. This function replaces `run_quantization()` and `run_quantization_from_np()`.

**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:
  - `data_feed` – Tensorflow 1.x initialisable dataset (`tf.data.Iterator`).
  - `np_array` – `numpy.ndarray` or dict of `numpy.ndarray`.
- **work\_dir** (optional, str) – If not None, dump quantization debug outputs to this directory.
- **model\_script** (str, optional) – Quantization script file path that allows the modification of HN current model, prior to quantization. If present, the script is parsed and a modified HN is set, where each layer has (possibly) new quantization params.
- **calib\_num\_batch** (int, optional) – Number of batches to run calibration on.
- **ft\_cfg** ([FineTuneConfigurator](#), optional) – Quantization-aware fine tune settings and parameters, such as the `learning_rate`, `decay_steps`, `decay_rate`, `epochs`, `optimizer`, and `dataset_size`.
- **equalize** (bool, optional) – Indicates whether to run the Equalization algorithm [1].
- **ibc** (bool, optional) – Indicates whether to run the Iterative Bias Correction algorithm [2].
- **finetune** (bool, optional) – Indicates whether to run the Quantization Aware Fine Tune.

- **start\_layer** (str, optional) – The layer from which to start the Equalization algorithm. If None, the first layer of the net is used.
- **end\_layer** (str, optional) – The layer in which to end the Equalization algorithm. If None, the last layer of the net is used.
- **fast\_ibc** (bool, optional) – Indicates whether to run the Iterative Bias Correction algorithm using Partial Numeric emulation, which is much faster but not bit exact to hardware. Ignored unless ibc is set to True.
- **bft** (bool, optional) – If true, forces run of fine tuning regardless value of finetune, while also restricting training to biases, resulting in Bias Fine Tune algorithm as described in [2].
- **ft\_batch\_size** (int, optional) – Size of each batch to run the Bias Fine Tune algorithm.
- **train\_data\_feed\_callback** (optional) – Callback that returns an initialized iterator, which is a `tf.data.Iterator` object, for the BFT algorithm. A callback is required to ensure that the data feed is generated inside the graphs and sessions used by the BFT. If this parameter is None, the calibration data feed is used.
- **max\_elementwise\_feed\_repeat** (int, optional) – Max value of elementwise feed repeat, used for calculating the quantized representation of biases and elementwise-add.
- **force\_results\_by\_layer** (dict, optional) – If not None, use these data for statistics instead of collecting on calibration set.
- **batch\_size** (int, optional) – Size of each batch when `data_type` is `np_array`.

## References

- [1] 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>
- [2] Finkelstein, A., Almog, U. and Grobman, M., 2019. Fighting Quantization Bias With Bias. <https://arxiv.org/abs/1906.03193>

### **revert\_state**(state)

Revert the runner's state to the given state.

**Parameters** **state** (States) – The state to update. In Quantized Model State, Hailo Model is allowed. In Compiled Model State, both Hailo Model and Quantized Model are allowed.

### **model\_summary**()

Prints summary of the model layers.

## 6.2. hailo\_sdk\_client.exposed\_definitions

This module contains enums used by several SDK APIs.

### **class** hailo\_sdk\_client.exposed\_definitions.JoinAction

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.NNFramework`

Bases: `enum.Enum`

Enum-like class for different supported neural network frameworks.

**TENSORFLOW = 'tf'**

Tensorflow 1.x

**TENSORFLOW2 = 'tf2'**

Tensorflow 2.x

**ONNX = 'onnx'**

ONNX

**class** `hailo_sdk_client.exposed_definitions.CalibrationDataType`

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`)

**class** `hailo_sdk_client.exposed_definitions.MetaArchitectures`

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`

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, ...)  
 Bases: object  
 Hailo Archive representation.

## 6.4. hailo\_sdk\_client.tools.hn\_modifications

hailo\_sdk\_client.tools.hn\_modifications.**add\_yuv\_to\_rgb\_layers**(runner, ...)  
 Add layers translating inputs from YUV to RGB.

### Parameters

- **runner** (**ClientRunner**) – the client runner.
- **input\_names\_to\_translate** (list of str, optional) – List of input layers names to add YUV to RGB translation layers, default to all input layers.
- **output\_har\_path** (str, optional) – path to write the output model with the newly added layers as HAR.

hailo\_sdk\_client.tools.hn\_modifications.**add\_resize\_input\_layers**(runner, input\_shapes, ...)  
 Change the input size by adding resize bi-linear layer.

### Parameters

- **runner** (**ClientRunner**) – the client runner.
- **input\_shapes** (tuple or dict) – The new input shape for each input layer. In case all input layers should have the same input shape use tuple of (h, w) and a resize layer from the given shape will be added to all input layers. Otherwise, use dict of shape tuple (h, w) by input names.
- **output\_har\_path** (str, optional) – path to write the output model with the newly added layers as HAR.

hailo\_sdk\_client.tools.hn\_modifications.**translate\_rgb\_dataset\_to\_yuv**(rgb\_dataset)  
 Translate a given RGB format images dataset to YUV format images. This function is useful when the model expects YUV images, while the calibration images used for quantization are in RGB.

**Parameters** **rgb\_dataset** (numpy.ndarray) – Numpy array of RGB format images with shape (image\_count, h, w, 3) to translate.

## 6.5. hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune

Full Fine Tune algorithm implementation.

**exception** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**FTException**  
 Bases: Exception  
 Fine tune algorithm exception.

**class** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**AlphaBlendConfig**  
 Bases: tuple  
 Alpha Blend configuration.

**property** **a\_decay\_epochs**  
 Alpha Blend decay epochs.

**property** **a\_decay\_power**  
 Alpha Blend decay power.

**class** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**Schedule**(*base\_value*, ...)  
Bases: object

Base class for scheduling definitions. Instances of this (and subclasses) are used to configure time axis behavior of learning rate, and possibly other parameters, such as the loss components importance factors.

**\_\_init\_\_**(*base\_value*, *decay\_rate*=1, *decay\_images*=1000.0, *warmup\_images*=None, *warmup\_value*=None)  
Initialize self. See help(type(self)) for accurate signature.

**get\_val\_by\_step**(*global\_step\_t*)

**class** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**ConstSchedule**(*base\_value*)  
Bases: [hailo\\_sdk\\_client.quantization.tools.quant\\_aware\\_fine\\_tune.Schedule](#)

The trivial (time-independent) schedule.

**\_\_init\_\_**(*base\_value*)  
Initialize self. See help(type(self)) for accurate signature.

**get\_val\_by\_step**(*global\_step\_t*, *batch\_size*)

**class** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**ExpSchedule**(\*args, ...)  
Bases: [hailo\\_sdk\\_client.quantization.tools.quant\\_aware\\_fine\\_tune.Schedule](#)

Exponentially decaying schedule, staircase. Starts from the base value, and reduces by *decay\_factor* each *decay\_images*. It is implemented by wrapping `tf.train.exponential_decay` and adding a warmup.

**\_\_init\_\_**(\*args, \*\*kwargs)  
Initialize self. See help(type(self)) for accurate signature.

**get\_val\_by\_step**(*global\_step\_t*, *batch\_size*)

**class** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**CosineSchedule**(\*args, ...)  
Bases: [hailo\\_sdk\\_client.quantization.tools.quant\\_aware\\_fine\\_tune.Schedule](#)

Repeated Cosine-decay schedule, implemented by wrapping `tf.train.cosine_decay_restarts` and adding a warmup. The *decay\_images* argument is used for the length of one cycle till restart, and the *decay\_rate* argument is used as the factor to reduce on restart (the *m\_mul* argument of `tf.train.cosine_decay_restarts`).

**\_\_init\_\_**(\*args, \*\*kwargs)  
Initialize self. See help(type(self)) for accurate signature.

**get\_val\_by\_step**(*global\_step\_t*, *batch\_size*)

**class** hailo\_sdk\_client.quantization.tools.quant\_aware\_fine\_tune.**FineTuneConfigurator**(*network\_args*=None, ...)  
Bases: object

Use this class to control the settings of the Quantization Aware Fine-Tuning run. Desired configs are passed to the constructor.

**\_\_init\_\_**(*network\_args*=None, *logger*=None)  
Initialize the object.

**Parameters** *network\_args* (dict, optional) – Args to set. If None, default values are used. The list of supported dictionary keys follows, including the default value/behavior in case the key is nonexistent (possibly omitted if trivial, e.g., False, None, [], etc.).

- *bias\_only* (bool): If True, train biases only yielding the baseline BFT procedure from <https://arxiv.org/abs/1906.03193>, giving very similar results to IBC.

Default: **False**.

- *layers\_to\_freeze* (list of str): Don't train kernels and biases for these layers; names to be given in Hailo HN notation, e.g., *conv4*, *dw3*, etc.

Default: **[]**

- *dataset\_size* (int): Training subset size (images per epoch). Should be smaller than total images of the train data source, otherwise exception will be thrown.

Default: `DEFAULT_DATASET_SIZE`

- `epochs` (float): How many times to feed the train subset.

Default: `DEFAULT_EPOCHS`

- `lr_schedule` (subclass of [Schedule](#)): If given, used as is for learning rate schedule, rendering all LR-related args below to be “don’t-care”.

Default: built according to `lr_schedule_type`.

- `lr_schedule_type` (`SCHEDULE_TYPES` enum or str): Specifies which of the built-in [Schedule](#) subclasses are invoked to build a learning rate scheduling object. The parameters for this purpose will be taken from the additional args below, generally used in similar ways across schedules. E.g., if `SCHEDULE_TYPES.EXPONENTIAL` is passed here, an [ExpSchedule](#) instance will be built for `lr_schedule`.

Default: `SCHEDULE_TYPES.COSINE_RESTARTS`

- `decay_epochs` (float): Specifies number of epochs for a “step” of learning rate decay. Translated to units of images and passed to [Schedule](#) subclasses. Exact usage dependent on the scheduler; see docstrings of relevant class.

Default: **1**

- `learning_rate` (float): The base learning rate of any schedule.

Default: `DEFAULT_LEARNING_RATE`

- `decay_rate` (float): Factor for the learning rate decay per decay period. Exact usage dependent on the scheduler.

Default: **0.5**

- `warmup_epochs` (float): Time to spend with a constant (usually small) rate before starting the (usually starting high and decaying) schedule.

Default: **1**

- `warmup_lr` (float): LR to use during the warmup.

Default: `learning_rate / 4`

- `optimizer` (`OPTIMIZERS` enum or str): Type of optimizer, e.g., SGD, Momentum.

Default: **Adam**

- `clip_method` (`PRE_FT_CLIPPING_METHODS` enum or str): The method to use for pre-training clipping of 4-bit kernels.

Default: `DEFAULT_CLIP_METHOD`

- `clip_factor` (float): Clipping factor to use if `clip_method` is `SET_FACTOR`.

Default: `DEFAULT_CLIP_FACTOR`.

- `clip_percentile` (float): Clipping percentile to use if `clip_method` is `SET_PERCENTILE`.

Default: `DEFAULT_CLIP_PERCENTILE`.

- `def_loss_type` (`LOSS_TYPES` enum or str): The default loss type to use if `loss_types` is not given.

Default: `LOSS_TYPES.L2REL`

- `loss_layer_names` (list of str): 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 `LOSS_TYPES`, str, or callable, of 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 by above list. For example, "ce" (standing for "cross-entropy") for the classification head(s). If callable is passed, it will be used when calculating the teacher-student loss given the two tensors of native and quantized nets at the given layer. This a handy generic way to extend the functionality to any bivariate loss.

Default: the `def_loss_type` arg (or its default, "l2rel") for each of the layers.

- **loss\_factors** (list of int, of same length as *loss\_layer\_names*): Weighting factors of the above when summing the total loss.

Default: **1.0** for each one of the layers.

- **post\_batch\_callback** (callable): To be called after each batch, to monitor progress or trigger actions. Functionality normally added by subclassing `BasicFinetuneMonitor`, and passing `process_batch_results` of the instance.
- **native\_layers** (list of str) - Layers to avoid quantizing, to optimize for inference with those layers in double precision. Handy for a workaround, especially for sensitive layers (s.a. output layers), while the others remain in need of QFT.

Default: []

**clip\_kernel\_pre\_ft**(*kernel, lname, bits, num\_groups=1, verbose=True*)

a middle-man to facilitate extensions by subclassing `FineTuneConfigurator`

**classmethod from\_json**(*path, logger=None*)

**info**()

**class** `hailo_sdk_client.quantization.tools.quant_aware_fine_tune.BasicFinetuneMonitor`(*logger=None, ...*)  
Bases: object

Provides a basic, batch-by-batch monitoring callback. It can be subclassed for additional functionality (e.g., for Tensorboard-style real-time plotting).

**\_\_init\_\_**(*logger=None, loss\_log\_period=32, loss\_ma\_coeff=0.98*)

#### Parameters

- **loss\_log\_period** (int) – Print every so and so batches.
- **loss\_ma\_coeff** (float) – How much to smooth the "moving average".

**process\_batch\_results**(*net\_out\_native\_vals, net\_out\_ft\_vals, image\_info\_val, losses\_vals\_d, ...*)

`hailo_sdk_client.quantization.tools.quant_aware_fine_tune.get_loss_factors`(*ft\_cfg, ...*)  
Set the loss factors automatically for the loss layers in case not already configured

`hailo_sdk_client.quantization.tools.quant_aware_fine_tune.fine_tune_from_feed`(*runner, ...*)  
Improve the quantized model's accuracy by Quantization-Aware Finetuning (QFT). This function uses an existing client runner and modifies it. This function is called internally by `run_quantization()`, no use case for user-side invocation yet.

#### Parameters

- **runner** (`ClientRunner`) – SDK client runner to work with. It should already contain the translated parameters to correct. The corrected weights are loaded into this runner.
- **data\_feed\_callback** – Callback that returns an initialized iterator, which is a `tf.data.Iterator` object. A callback is required to ensure that the data feed will be generated inside the graphs and sessions used by the quantization.
- **ft\_cfg** (`FineTuneConfigurator`) – Fine tune configuration.
- **results\_by\_layer** (dict) – Use these data for quantization statistics instead of collecting new calibration set.

## 6.6. hailo\_sdk\_client.tools.core\_postprocess.core\_postprocess\_api

**exception** `hailo_sdk_client.tools.core_postprocess.core_postprocess_api.UnsupportedMetaArchError`  
 Bases: `Exception`

Raised when using an unsupported meta architecture.

`hailo_sdk_client.tools.core_postprocess.core_postprocess_api.add_nms_postprocess_from_hn(input_hn_path, ...)`  
 Adds in-chip NMS post-processing to the given model.

### Parameters

- **input\_hn\_path** (str) – Path to the input model in an HN file.
- **input\_npz\_path** (str) – Path to the corresponding model params, expected in native stage.
- **output\_hn\_path** (str) – Path to write the output model with the newly added post-process.
- **output\_npz\_path** (str) – Path to write the corresponding model params for the newly added post-process.
- **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 loaded and saved. Defaults to `NA-TIVE_FUSED_BN`.

---

**Note:** Additional information about the configuration parameters used in the config JSON file:

- **nms\_scores\_th** (float): Threshold used for filtering out candidates (confidence TF). Any box with score < TH is ignored.
- **nms\_iou\_th** (float): Intersection over union overlap Threshold, used in the NMS iterative elimination process where potential duplicates of detected items are ignored. (not used in Centernet)
- **max\_proposals\_per\_class** (int): Fixed number of outputs allowed in this model, derived from hardware limitations.
- **centers\_scale\_factor** (float): Values used for compensation of rescales done in the training phase (derived from faster\_rcnn architecture). This param rescales anchors centers.
- **bbox\_dimensions\_scale\_factor** (float): Values used for compensation of rescales done in the training phase (derived from faster\_rcnn architecture). This param rescales anchors dimensions.
- **classes** (int): Number of detected classes, e.g., MobilenetV2-SSD trained on COCO has 91 classes.
- **background\_removal** (bool): Toggle background class removal from results, used in inference time.
- **background\_removal\_index** (int): Index of background class for background removal, e.g., in MobilenetV2-SSD it is 0.
- **bbox\_decoders** (list): List of bbox decoders (anchors) for the NMS layer. Each model has its own number of boxes per anchor, and each anchor is described by location, dimensions, and inputs.
- **input\_division\_factor** (int): Division factor of proposals sent to the NMS per class, instead of running NMS on all proposal together. Used to decrease memory usage for NMS (in case of allocation failure).
- **regression\_prediction\_order** (list): Permutation that defines the order of inputs to each bbox decoder layer, from its respective regression layer. The default value is [0, 1, 2, 3], which translates to prediction order of [ty, tx, th, tw] (coordinates description of centroids anchors). Another common prediction order is typical for implementations based on Keras APIs, where the prediction order is usually [tx, ty, tw, th]: Use [1, 0, 3, 2] permutation in the config JSON.



### Example

```
>>> from hailo_sdk_client.tools.core_postprocess.core_postprocess_api import add_nms_postprocess_
↳ from_hn
>>> from hailo_sdk_common.targets.inference_targets import ParamsKinds
>>>
>>> add_nms_postprocess_from_hn(input_hn_path='./Mobilenet-v2-ssd.hn',
...                             input_npz_path='./Mobilenet-v2-ssd.npz',
...                             output_hn_path='./Mobilenet-v2-ssd-nms.hn',
...                             output_npz_path='./Mobilenet-v2-ssd-nms.npz',
...                             config_json_path='nms_ssd_config.json',
...                             params_kind=ParamsKinds.NATIVE)
```

`hailo_sdk_client.tools.core_postprocess.core_postprocess_api.add_nms_postprocess_from_har(input_har_path,...)`  
Adds in-chip NMS post-processing to the given model.

#### Parameters

- **input\_har\_path** (str) – Path to the input model in an HAR file.
- **output\_har\_path** (str) – Path to write the output model as HAR with the newly added post-process.
- **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 loaded and saved. Defaults to `NATIVE_FUSED_BN`.

### Example

```
>>> from hailo_sdk_client.tools.core_postprocess.core_postprocess_api import add_nms_postprocess_
↳ from_har
>>> from hailo_sdk_common.targets.inference_targets import ParamsKinds
>>>
>>> add_nms_postprocess_from_har(input_har_path='./Mobilenet-v2-ssd.har',
...                             output_har_path='./Mobilenet-v2-ssd-nms.har',
...                             config_json_path='nms_ssd_config.json',
...                             params_kind=ParamsKinds.NATIVE)
```

#### See also:

[`add\_nms\_postprocess\_from\_hn\(\)`](#) for additional details.

`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.

### 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_node = "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",
...             "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_name=start_node,
...                                   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
```

`hailo_sdk_client.tools.core_postprocess.core_postprocess_api.create_nms_postprocess(runner,...)`

## 6.7. 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

**class** `hailo_sdk_client.tools.layer_noise_analysis.QuantAnalyzerCLI(parser)`

Bases: `hailo_platform.common.tools.cmd_utils.base_utils.CmdUtilsBaseUtil`

```
__init__(parser)
```

Initialize self. See help(type(self)) for accurate signature.

```
run(args)
```

```
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\_platform.common.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, ...)
```

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.

```
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_native', 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 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.
- **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\_native', 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\_native', 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=64, ...*)

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\_ibc=False, quantization\_script\_filename=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, npz_path, data_path, ...)`  
Run a complete flow of quantizing a model and analyzing the resulting quantization.

## 6.8. hailo\_sdk\_client.tools.dead\_channels\_removal

`hailo_sdk_client.tools.dead_channels_removal.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.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

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

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, ...)
```

Bases: object

Hailo Model export object.

```
__init__(session, graph, input_tensors, init_output_exports=None, init_variables_exports=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: 0>)

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: 0>)

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: 0>)

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: 0>)

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: 0>)

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: 0>)

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, network\_name=None*)  
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**  
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**  
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

**class** hailo\_sdk\_common.preprocessing.base.**Preprocess**  
Bases: object

Base class for preprocessing objects.

---

**Note:** This class should not be used directly. Use only its inherited classes.

---

**to\_json()**  
Represent this pre-processing object as a JSON.

## 7.5. hailo\_sdk\_common.preprocessing.normalization

**class** hailo\_sdk\_common.preprocessing.normalization.**Normalization**(*mean, std*)  
Bases: hailo\_sdk\_common.preprocessing.base.Preprocess

An API class to add hardware-integrated input normalization. It is used by passing it as the `integrated_preprocess` argument to `translate_tf_model()`.

**\_\_init\_\_(mean, std)**  
Hardware-integrated normalization by mean and standard deviation.

**Parameters**

- **mean** (list of float) – 1 dimensional list of mean values for each input feature to be subtracted with.
- **std** (list of float) – 1 dimensional list of standard deviation values for each input feature to be divided by.

**to\_json()**

Represent this pre-processing object as a JSON.

**as\_dict()**

## 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]}
...     }
... }'''
>>> 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.

**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*)

Get Hailo model from HN dictionary.

## 7.7. hailo\_sdk\_common.hailo\_nn.hn\_definitions

**class** `hailo_sdk_common.hailo_nn.hn_definitions.EWAddPolicy`

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\_NATIVE\_CLIPPED** = 'sdk\_native\_clipped'

**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'

```

**class** hailo\_sdk\_common.targets.inference\_targets.**InferenceDebugTargets**

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_NATIVE_CLIPPED = 'sdk_native_clipped'
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).

**class** hailo\_sdk\_common.targets.inference\_targets.**EmulationObject**(hw\_arch=None)

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**(hw\_arch=None)

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.**SdkNativeClipped**(hw\_arch=None)

Bases: [hailo\\_sdk\\_common.targets.inference\\_targets.EmulationObject](#)

Native emulation inference target. It runs the model as [SdkNative](#), but performs weights clipping if layers had weights clipping set via a quantization script. This target is needed for quantization of a model that utilizes weights clipping.

**NAME = 'sdk\_native\_clipped'**

```
class hailo_sdk_common.targets.inference_targets.SdkNumeric(hw_arch=None)
```

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(hw_arch=None)
```

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(hw_arch=None)
```

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
```

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
```

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

## Python Module Index

### h

- `hailo_sdk_client.exposed_definitions`, [87](#)
- `hailo_sdk_client.hailo_archive.hailo_archive`,  
[89](#)
- `hailo_sdk_client.quantization.tools.quant_aware_fine_tune`,  
[89](#)
- `hailo_sdk_client.runner.client_runner`, [78](#)
- `hailo_sdk_client.tools.core_postprocess.core_postprocess_api`,  
[93](#)
- `hailo_sdk_client.tools.dead_channels_removal`,  
[98](#)
- `hailo_sdk_client.tools.hn_modifications`, [89](#)
- `hailo_sdk_client.tools.layer_noise_analysis`,  
[95](#)
- `hailo_sdk_common.export.hailo_graph_export`, [99](#)
- `hailo_sdk_common.hailo_nn.hailo_nn`, [105](#)
- `hailo_sdk_common.hailo_nn.hn_definitions`, [106](#)
- `hailo_sdk_common.model_params.model_params`,  
[104](#)
- `hailo_sdk_common.preprocessing.base`, [104](#)
- `hailo_sdk_common.preprocessing.normalization`,  
[104](#)
- `hailo_sdk_common.profiler.profiler_common`, [104](#)
- `hailo_sdk_common.targets.inference_targets`,  
[106](#)