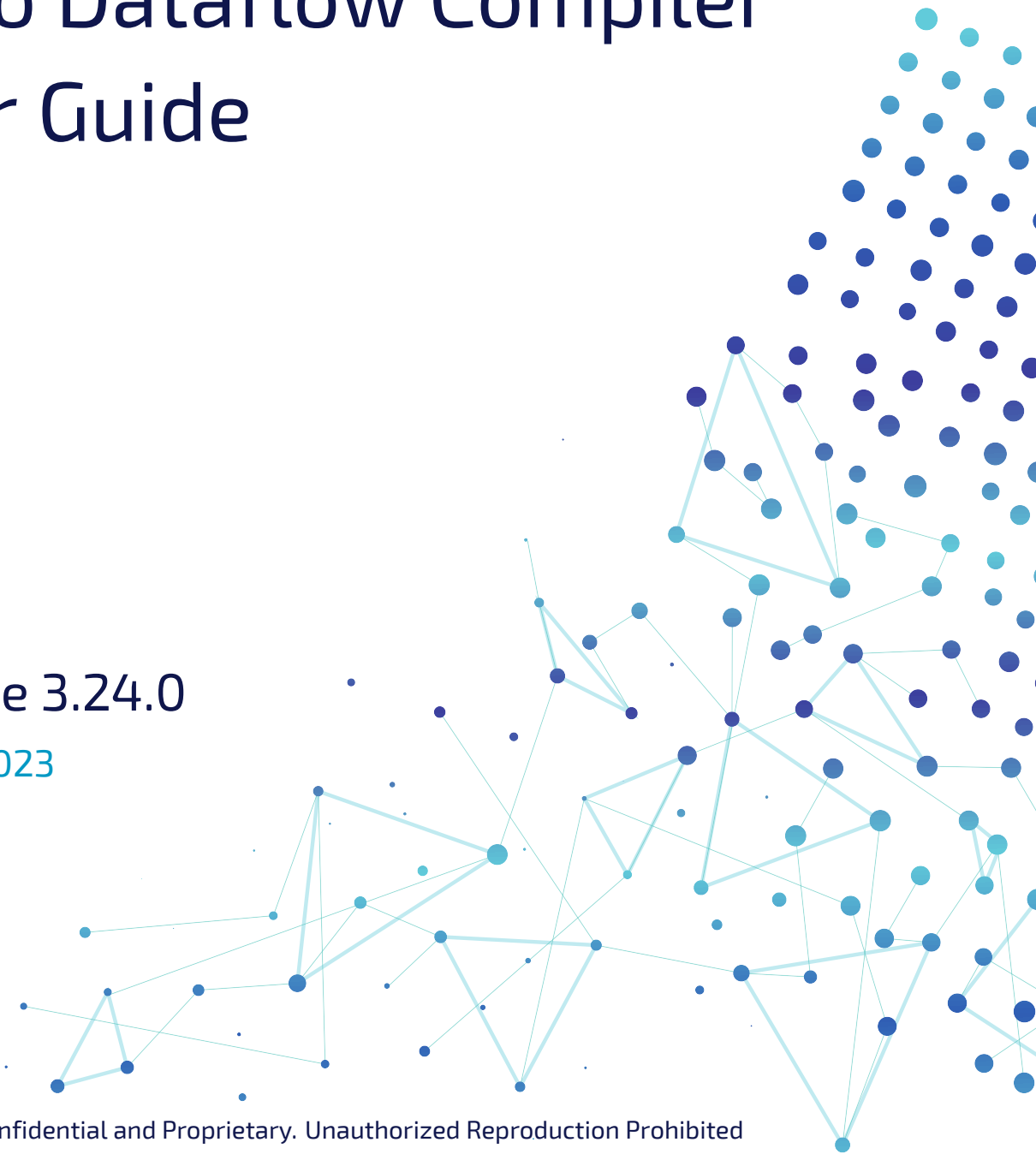
HAILO

Empowering Intelligence

Hailo Dataflow Compiler User Guide



Release 3.24.0

17 June 2023

Table of Contents

Disclaimer and Proprietary Information Notice

Copyright

This version of the document supersedes all previous versions.

General Notice

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

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

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

Part I

User Guide

1. Hailo Dataflow Compiler Overview

1.1. Introduction

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

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

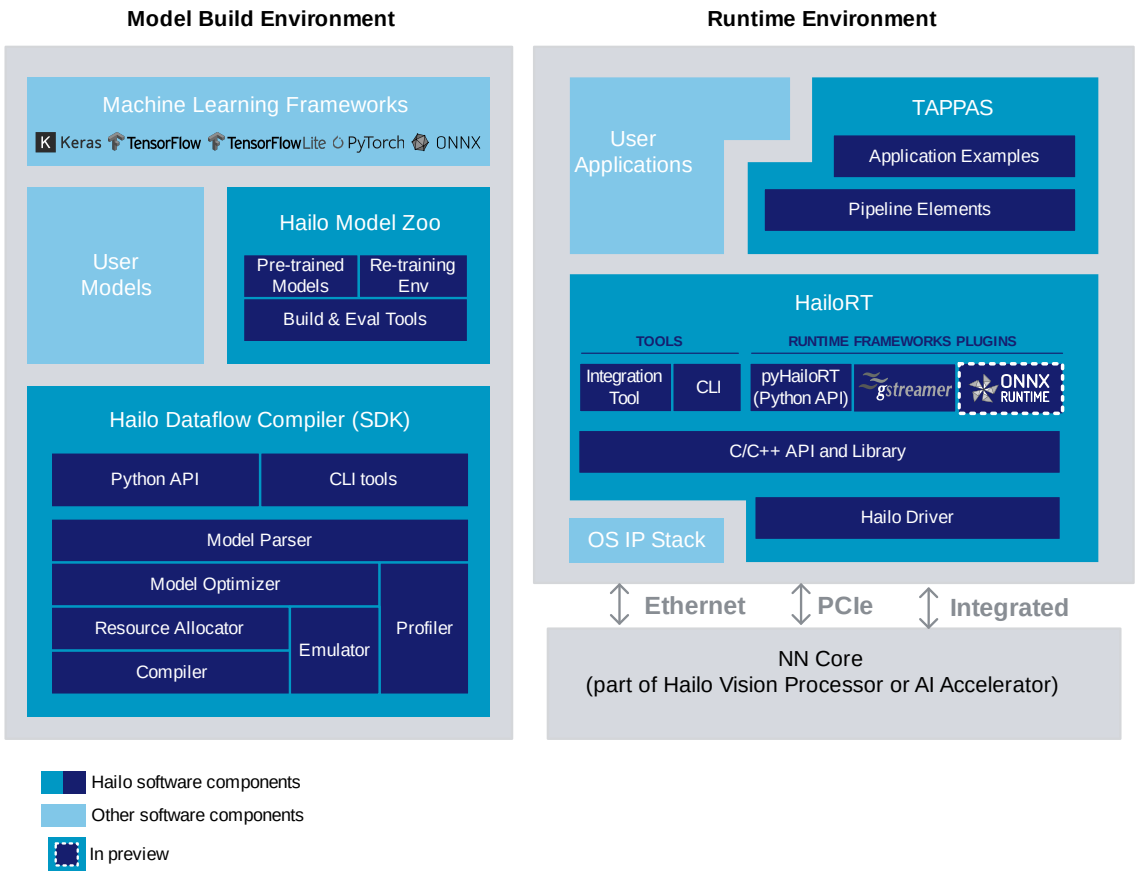


Figure 1. Detailed block diagram of Hailo software packages

1.2. Model Build Process

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

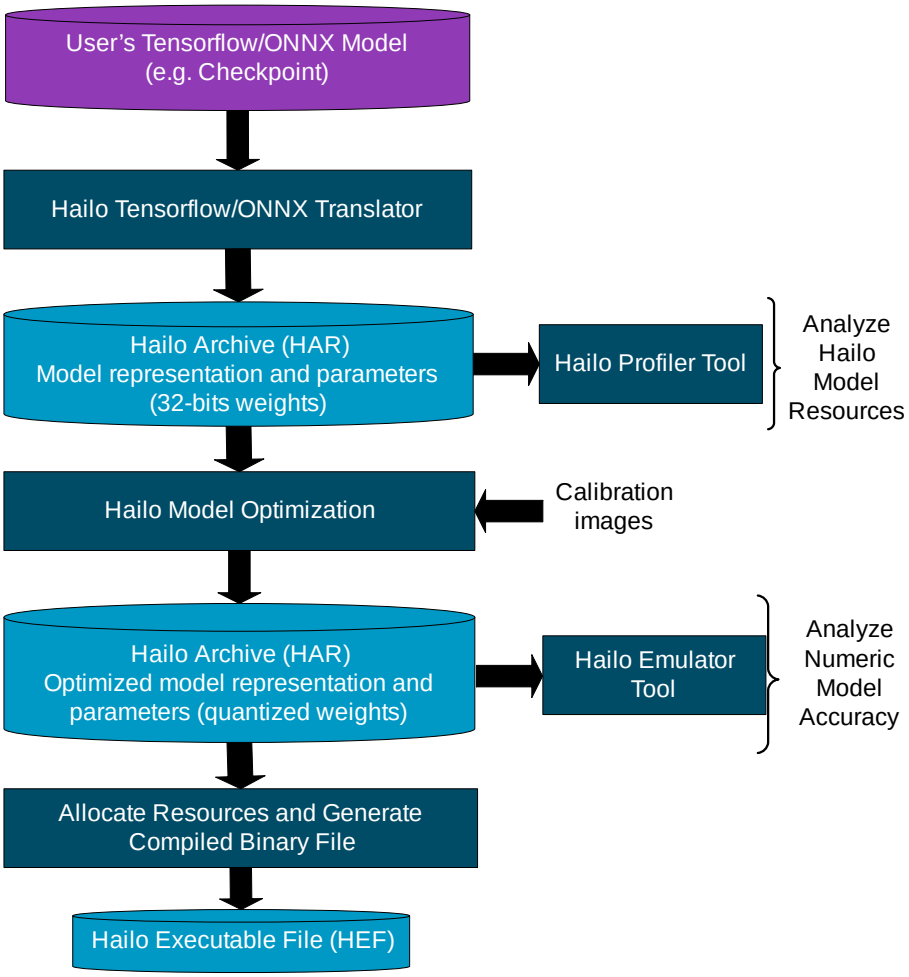


Figure 2. Model build process, starting in a Tensorflow or ONNX model and ending with a Hailo binary (HEF)

1.2.1. Tensorflow and ONNX Translation

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

1.2.2. Profiler

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

1.2.3. Emulator

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

1.2.4. Model Optimization

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

1.2.5. Compiling the Model into a Binary Image

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

1.3. 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.

On Hailo-8, if 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. On Hailo-15, the HailoRT library communicates with the neural code through an internal interface.

The HailoRT library can be installed on the same machine as the Dataflow Compiler (on accelerator modules, such as Hailo-8) or on a separate machine. A Yocto layer is provided to allow easy integration of HailoRT to embedded environments.

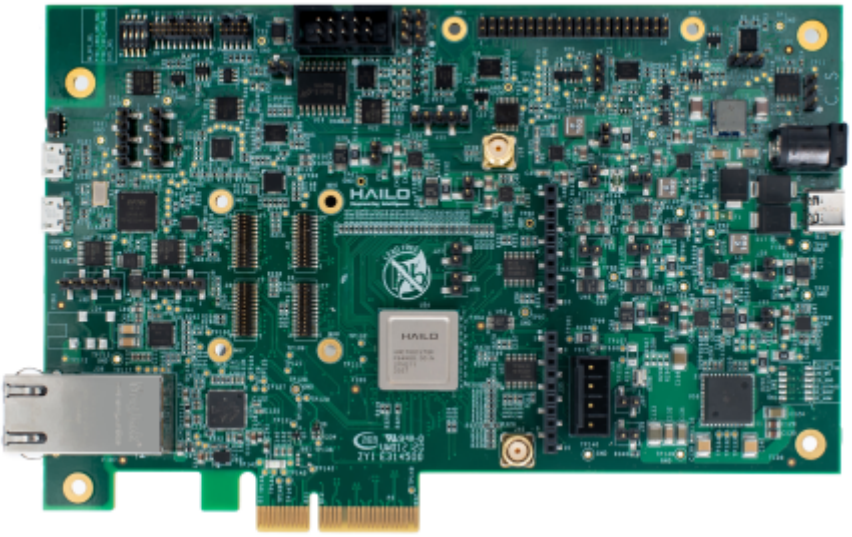


Figure 3. Hailo-8 Boards

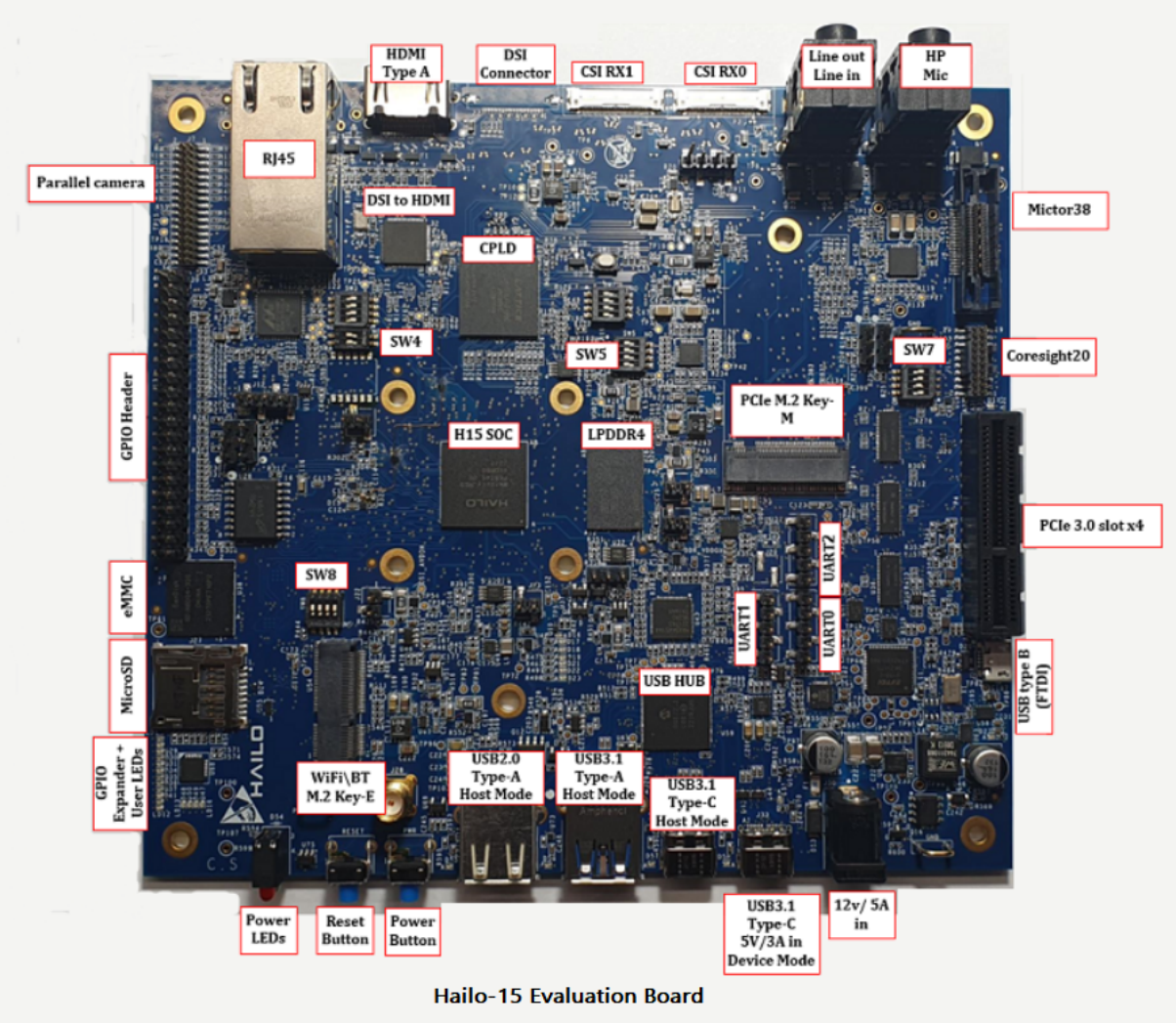


Figure 4. Hailo-15 Boards

2. Changelog

Dataflow Compiler v3.24.0 (July 2023)

General

- Hailo Dataflow Compiler now supports the newly-released Hailo-15H device

Model Optimization

- The automatic 16-bit output layer feature is disabled
- System & GPU memory usage optimizations

Kernels and Activations

- 16bit precision mode can be applied to specific Conv layers inside the model to increase their accuracy

Profiler

- Activation clipping values are showed in the activation histogram plot
- You can use the new profiler HTML design, by appending the `--use-new-report` flag to the CLI command (preview; will be default starting 2023-10)

Parser

- Apply padding correction on Average Pooling layer without external padding
- Start/End Node Name suggestion for models with unsupported ops
- Output layer names order is determined by their order on the parser API

Full Precision Optimization

- Dense layers (fully-connected) input features defuse
  - Automatically performed on big tensors
  - Can be configured manually using a model script command

High-level and Documentation

- NMS auto detection:
  - Detected NMS config saved to native HAR
  - NMS post-process command takes config from auto detection
  - Get auto-detected NMS config using the `get_detected_nms_config()` API
  - If a post-process json configuration files is used (on SSD, for example), the `reg` and `cls` layer names can remain empty, and the auto-detect algorithm will locate them
- Added *set_seed command* for reproducing of quantization results, affects the seed of tensorFlow, numpy, and python.random libraries (preview)
- New API - `get_params_statistics()`
- Apply sigmoid automatically whenever is needed:
  - YOLOX - after the objectness and classes layers before the NMS
  - YOLOv5 - between output convolution layers and the NMS
  - SSD - between classes layers and the NMS

Compiler

- Improved *Performance Mode* algorithm
- Improved FPS on models that are compiled to Hailo-15H

Command Line Tools

- *hailo optimize* using RGB images instead of random data when using –use-random-calib-set
- *hailo analyze-noise* now saves its results inside the model's HAR

Deprecated APIs

- Deprecation warning for *performance_param(optimization_level=max)*, please use *performance_param(compiler_optimization_level)* instead
- Deprecation warning on the *–analysis-data* argument on *hailo profiler*
- Deprecated *get_tf_graph()* API was removed, please use `infer()`

Known issues

- Refer to Hailo AI SW Suite: Known Issues page for an updated list of issues

Dataflow Compiler v3.23.0 (April 2023)

Compiler

- Introducing *Performance Mode*, that gradually increases the utilization to achieve the best FPS (preview)
- The compiler has been optimized for better stability and performance

Model Optimization

- Supporting Quantization-Aware-Training using the `set_keras_model()` API. See the *Quantization-Aware-Training Tutorial* for more details
- *Added support* for 16-bit precision on full networks, in case all layers are supported (preview)
- *Optimization levels* are changed to be between 0 (no optimization) and 4 (best plausible optimization), as opposed to 0-3. Their current description is found in the *model_optimization_flavor* API guide
- The default optimization level is now 2 for GPU and 1024 images, 1 for GPU and less than 1024 images, and 0 for CPU only
- Bias Correction algorithm is used as default (optimization_level=1)
- When importing Hailo python libraries, TF memory allocation mechanism is set to "memory growth" by default, to decrease memory consumption. One can override this with an *environment variable*
- Improved the FineTune algorithm for models with multiple output nodes
- 16-bit output layer is enabled automatically when supported, for small output tensors
- When optimization fails, a better error message is displayed, referring to the failing algorithm

Kernels and Activations

- Transformer building block *Multi head Attention* is now supported (preview)
- Increased support for Conv&Add layers

Profiler

- The HTML profiler now displays a quick version of the layer analysis tool (*Accuracy* tab) automatically
- Added *–stream-fps* flag to *hailo profiler*, to be used with single-context models, to evaluate the performance using an FPS which is lower than the network's FPS
- Added *–collect-runtime-data* flag to *hailo profiler*, to automatically infer using *hailortcli* and display runtime data in the report

Emulator

- Added support for emulating YOLOv5 NMS with *engine=cpu*, as well as for SSD
- Added emulation support for RGBX, NV12, NV21 and i420

Parser

- *nms_postprocess command* supports SSD post-processing also on CPU using the 'engine' flag (preview)
- Automatic anchors extraction for YOLOv5-type NMS models, using a message is displayed during parsing
- Added support for on-chip i420->YUV conversion, using an *input_conversion* command
- Added support for Biased Delta activation on TFLite, that is implemented using ABS->SIGN->MUL
- Added support for SpaceToDepth kernel that is used on YOLOP
- Added support for Spatial Squeeze operator on TFLite
- Added support for new HardSwish structure in ONNX parser
- Added Global MaxPool operator in ONNX parser
- Fixed a bug in the HardSigmoid implementation
- Added *Hailo-ONNX* support for models with *Shape* connections around the HailoOp
- Added *Hailo-ONNX* support for external inputs to the post-processing section
- Added an option to disable hailo-onnx runtime model build, when it hinders model parsing
- Softmax and Argmax can be added to the model using the *logits_layer* model script command
- Whenever NMS is being added (using a *nms_postprocess* command), Sigmoid is now added automatically
- Added *hybrid conversion* commands on the *input_conversion* section: *yuy2_to_rgb*, *nv12_to_rgb*, *nv21_to_rgb*, *i420_to_rgb*

High-level and Documentation

- Log level can be set using the *LOGLEVEL* environment variable (0 [default] to 3)
- *hailo visualizer* shows layers added using model script commands that were folded
- *hailo visualizer* shows input layers conversion type
- Tutorials are now using *runner.infer* API instead of *runner.get_tf_graph*
- Layer Analysis Tool Tutorial has been updated to demonstrate how to increase accuracy
- Model Optimization Tutorial now uses YOLOv5 NMS with *engine=cpu*, and also a bbox visualization code
- Added description of which optimization algorithms are activated with each *optimization level*
- Removed the Multiple Models Tutorial. The `Join API` is still supported

Command Line Tools

- *hailo analyze* command removed, please use *hailo analyze-noise* instead
- New argument *–analyze-mode* added to *hailo analyze-noise*
- New argument *–disable-rt-metadata-extraction* added to *hailo parser onnx*
- New argument *–version* is added to *hailo*

Deprecated APIs

- Deprecation warning for `get_tf_graph()`, please use `infer()`
- `optimize()` is not allowed under `QUANTIZED_MODEL`
- Added *analyze_mode* to argument `analyze_noise()`
- Added *disable_rt_metadata_extraction* argument to `translate_onnx_model()`
- Deprecation warning for *quantization_params* and *compilation_params* arguments from `translate_onnx_model()` and `translate_tf_model()`, please use model script commands *quantization_param* and *compilation_param* instead
- The following ClientRunner APIs are now deprecated: *get_results_by_layer*, *update_params_layer_bias*, *profile_hn_model*, *get_mapped_graph*, *get_params_after_bn*, *set_original_model*, *apply_model_modification_commands*

- Removed deprecated argument *ew_add_policy* from `translate_onnx_model()` and `translate_tf_model()`
- Removed *dead_channels_removal_from_runner* API
- Deprecated *scores_scale_factor* argument to SSD post-process JSON file, use *bbox_dimensions_scale_factor* instead
- Deprecation warning for *context_switch_param* command parameters of type: *goal_network_X*

Known issues

- Some Transformer models are at risk for having a runtime bug when inferring with batch_size > 1, when multi-context allocation is used a workaround is to use the *max_utilization* parameter of *context_switch_param* command to change the failing context partition
- In some cases, using the Fine Tune algorithm when the whole network is quantized to 16-bit might cause a degradation

Dataflow Compiler v3.22.1 (February 2023)

Parser

- Fixed an issue where a model script had to be provided explicitly to *hailo compiler* when an NMS command was used
- Added support for Global Maxpool operator in ONNX parser
- Fixed a parsing issue in Hardswish activation
- Fixed an issue that has prevented YOLOv8 from parsing

Compiler

- Fixed prints to screen during compilation, regarding single/multi context flow and resources utilization
- Removed the warning message of using on-chip NMS with multi context allocation, since the new version of HailoRT fixes the issue

Dataflow Compiler v3.22.0 (January 2023)

Package Updates

- Added support for Ubunbu 22.04, Python 3.9, and Python 3.10
- Ubuntu 18.04, Python 3.6 and Python 3.7 are no longer supported
- Updated Tensorflow requirement to version 2.92
- Updated ONNX requirement to version 1.12.0
- Updated ONNXRuntime requirement to version 1.12.0
- Updated PyTorch requirement to version 1.11

Profiler

- Introducing Accuracy Tab on the *HTML Profiler*, to be used as a tool to analyse and improve accuracy
- Profiler in post-placement mode doesn't require .hef file, when working on a compiled .har file
- Profiler will apply model modifications on pre_placement mode, if a model script was supplied
- `profile()` API will not update the runner state, even if it compiles for the profiling process
- Bug fixes

Model Optimization

- ClientRunner now has a new `SdkFPOptimized` state (see *runner states diagram*), for assessing model accuracy before quantization
- Updated the *Model Optimization workflow* section with simple and advanced optimization flows
- Updated the *Model Optimization Tutorial* with step-by-step instructions for validating accuracy through the optimization and compilation phases
- Updated the *Layer Analysis Tool tutorial* to utilize the new HTML profiler Accuracy tab

Emulator

- Added Emulator support for YUY2 color conversions, using 'emulator_support=True' flag on the *input_conversion* command

Kernels and Activations

- Added support for on-chip NV12->YUV, NV21->YUV and YUV->BGR format conversions, using an *input_conversion*
- Further increased support for *Resize Bilinear layers*
- *Nearest Neighbour Resize* now supports downsampling
- Added support for ReduceSumSquare operator
- Add support for EfficientGCN pooling block

Parser

- *nms_postprocess command* now supports 'engine' flag, that instructs HailoRT to complete YOLOv5 NMS post-processing on the host platform (preview)
- Enhanced the suggestion for end-node names
- Added support for Less operator in both ONNX and Tensorflow parsers
- Add support for dual broadcast in element-wise mult (Hx1xC * 1xWxC -> HxWxC)
- Added support for multiplication by 0 in all frameworks (x*0, x*0+b, (x+b)*0)
- Added support for depthwise with depth multiplier as group convolution in TFLite
- Add support for ADD_N from TFLite models

Compiler

- Optimized the compiler for better stability and performance
- Bug fixes

Known Bugs

- On this version, on-chip YOLOv5 NMS needs to be compiled using the legacy `fps` command.

API

- nms_postprocess model script command now uses relative paths relative to the alls script location. In addition, when working with a HAR file that has model script inside, it uses the json from within the HAR
- On nms_postprocess model script command, changed the 'yolo' meta_arch to be 'yolov5'
- Layer Analysis Tool now exports its data to a json file, that *could be used with the HTML profiler* to unlock the new Accuracy tab
- ClientRunner APIs
  - New
    - analyze_noise
    - optimize_full_precision
  - Argument changes

    - New: analysis_data in profile and profile_hn_model
    - Deprecation warning: fps flag in all APIs that compile (profile_hn_model, get_tf_graph)
    - Deprecation warning: ew_add_policy in translate_onnx_model, translate_tf_model
    - Deprecation warning: apply_model_modifications
    - Removed: model_script_filename in load_model_script
    - Removed: is_frozen, start_node_name, nn_framework in translate_tf_model
    - Removed: start_node_name, net_input_shape, onnx_path in translate_onnx_model
  - Removed
    - quantize
    - equalize_params
    - get_hw_representation
    - revert_state
  - Deprecation warning
    - get_results_by_layer
    - translate_params
    - update_params_layer_bias
    - profile_hn_model
    - get_mapped_graph
    - get_params_after_bn
    - set_original_model
    - apply_model_modification_commands
- High level APIs
  - add_nms_postprocess (not using a model script command) - removed
  - dead_channels_removal_from_runner - deprecation warning
- CLI tools
  - `analyze` was renamed to `analyze-noise`
    - –data_path renamed to –data-path
    - –eval-num renamed to –data-count
    - –calib_path, –alls-path, –quant-mode, –layers, –inverse, –ref-target, –test-target, –analyze-mode removed
    - old flags exist under analyze command
  - `compiler`
    - –alls renamed to –model-script
    - –auto-alls-path renamed to –auto-model-script
  - `har`
    - revert removed
  - `parser`
    - ckpt, tf2 removed (just use hailo parser tf *FILE*)
    - –force-pb and –force-ckpt removed from parser tf

- `profiler`
  * –fps removed
  * –alls renamed to –model-script
  * –analysis-data added

Dataflow Compiler v3.20.1 (November 2022)

Parser

- Added support for custom TFLite operators that implement a biased delta activation
- Added support for rank-2 HardSwish activation
- Optimized HardSwish and Gelu implementation
- Added support for the self operators add(x,x), concat(x,x), and mul(x,x) in the TF and ONNX parsers
- Pinned jsonref package to version 0.3.0 to fix installation error

Dataflow Compiler v3.20.0 (October 2022)

Model Optimization

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

Kernels and Activations

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

Compiler

- Optimized the performance of compiled models

Parser

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

High Level

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

API

- Compiler automatically separates different connected components to multiple *network groups*
  - Mostly relevant for `joined networks` with join_action=JoinAction.NONE
  - HailoRT API can be used to activate/deactivate each network group, although it is recommended to use the Scheduler API because it automatically switches between network groups (and .hef files)

  - For more information refer to *network_group model script command*

- `ClientRunner.compile()` API is introduced (planned to replace runner.get_hw_representation) (preview)
- Updated *platform_param* model script command to optimize compilation for low PCIe bandwidth hosts
- *Model script command for adding NMS on chip* is simplified (preview)
- Deprecation warning for the legacy –fps argument, use *performance_param* model script command instead
- Removed the already-deprecated APIs
  - integrated_preprocess and ckpt_path arguments from ClientRunner methods
  - Removed har-modifier CLI, and the following related methods: add_nms_postprocess_from_hn, add_nms_postprocess_from_har, dead_channels_removal_from_har, transpose_hn_height_width_from_hn, transpose_hn_height_width_from_har, add_yuv_to_rgb_layers, add_yuv_to_rgb_layers_from_har, add_resize_input_layers, add_resize_input_layers_from_har
  - *npz-csv* (use *params-csv* instead)
- As the parser detects Tensorflow1/2/TFLite automatically, the API for specifying the framework is deprecated
- The argument *onnx_path* of *ClientRunner.translate_onnx_model* was renamed to *model*, and also supports 'bytes' format
- *ClientRunner.load_model_script* can receive either a file object or a string

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

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

Dataflow Compiler v3.19.0 (August 2022)

Parser

- TFLite support to release

Kernels and Activations

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

Compiler

- Model load time optimizations

High Level and API

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

Dataflow Compiler v3.18.1 (July 2022)

General

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

Dataflow Compiler v3.18.0 (June 2022)

Parser

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

Model Optimization

- **Introducing *Optimization levels***
  - Using a number between 0 to 3 (default=1), control the complexity of the optimization algorithm
  - The default (1) requires 1024 images for the calibration set as well as GPU, unlike the previous default that used 128 images
- **Introducing *Compression levels***
  - Using a number between 0 to 5 (default=0), control the compression of the model
  - Higher compression corresponds to better performance, but require stronger Optimization algorithms to maintain accuracy
  - Higher compression is achieved with converting more layers to 4-bit
- Optimization and Compression levels are set using a model script command

Kernels and Activations

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

Compiler

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

Documentation and Tutorials

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

High Level and API

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

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

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

Dataflow Compiler v3.17.0 (May 2022)

Kernels and Activations

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

Compiler

- Bus fixes in profiler report
- Bug fixes and enhancements

High Level and API

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

Dataflow Compiler v3.16.0 (April 2022)

Kernels and Activations

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

Compiler

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

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

High Level and API

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

Dataflow Compiler v3.15.0 (February 2022)

High-Level and Documentation

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

Core

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

Compiler

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

Dataflow Compiler API

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

Parser

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

Dataflow Compiler v3.14.0 (January 2022)

Core

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

Compiler

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

Model Optimization

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

Dataflow Compiler API

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

Parser

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

Dataflow Compiler v3.13.1 (December 2021)

User Interface

- Fixed a bug in the *Profiler*.

Dataflow Compiler v3.13.0 (December 2021)

User Interface

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

Parser

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

Dataflow Compiler v3.12.0 (October 2021)

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

Core

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

Compiler

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

Parser

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

Examples and packaging

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

Dataflow Compiler v3.11.2 (October 2021)

Parser

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

Dataflow Compiler v3.11.1 (September 2021)

Core

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

Parser

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

Dataflow Compiler v3.11.0 (September 2021)

Core

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

Models Allocation

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

Parser

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

Dataflow Compiler API

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

Model Optimization

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

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

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

Dataflow Compiler v3.10.1 (August 2021)

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

Dataflow Compiler v3.10.0 (July 2021)

Core

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

Models Allocation

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

Parser

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

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

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

Dataflow Compiler v3.9.1 (July 2021)

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

Dataflow Compiler v3.9.0 (June 2021)

Core

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

Models Allocation

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

Parser

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

Dataflow Compiler API

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

Quantization

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

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

Dataflow Compiler v3.8.0 (May 2021)

Core

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

Models Allocation

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

Parser

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

Dataflow Compiler API

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

Dataflow Compiler v3.7.1 (April 2021)

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

Dataflow Compiler v3.7.0 (April 2021)

Core

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

Models Allocation

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

Parser

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

Quantization

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

Dataflow Compiler API

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

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

Dataflow Compiler v3.6.0 (March 2021)

Core

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

Models Allocation

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

Parser

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

Dataflow Compiler API

- Updated the *tutorials* to use Hailo Archive (HAR) files
- Weight files (NPZ) and archive files (HAR) size optimization

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

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

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

Dataflow Compiler v3.5.0 (February 2021)

Core

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

Models Allocation

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

Parser

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

Dataflow Compiler API

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

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

Dataflow Compiler v3.4.1 (February 2021)

- Upgraded HailoRT and firmware

Dataflow Compiler v3.4.0 (January 2021)

Core

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

Parser

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

Dataflow Compiler client API

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

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

3. Dataflow Compiler Installation

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

3.1. System Requirements

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

1. Ubuntu 20.04/22.04, 64-bit (supported also on Windows, under WSL2)
2. 16+ GB RAM (32+ GB recommended)
3. Python 3.8/3.9/3.10, including `pip` and `virtualenv`
4. `python3.X-dev` and `python3.X-distutils` (according to the Python version), `python3-tk`, `graphviz`, and `libgraphviz-dev` packages. Use the command `sudo apt-get install PACKAGE` for installation.

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

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

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

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

3.2. Installing / Upgrading Hailo Dataflow Compiler

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

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

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

For a clean installation, create a virtualenv:


```
virtualenv <VENV_NAME>
```

Enter the virtualenv:

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

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

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

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

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

```
pip uninstall -y hailo_sdk_common hailo_sdk_client hailo_sdk_server hailo_model_
↪optimization
```

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

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

```
hailo -h
```

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

4. Tutorials

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

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

4.1. Dataflow Compiler Tutorials Introduction

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

Model Compilation:

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

The Hailo compilation process consists of three steps:

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

Inference:

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

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

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

4.1.1. Usage

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

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

4.2. Parsing Tutorial

4.2.1. Hailo Parsing Example from Tensorflow CKPT to HAR

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

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

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

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

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

Choose the checkpoint files to be used throughout the tutorial:

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

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

chosen_hw_arch = 'hailo8'
# For Hailo-15 devices, use 'hailo15h'
# For Mini PCIe modules or Hailo-8R devices, use 'hailo8r'
```

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

First, initialize a ClientRunner and use the translate_tf_model method.

Arguments:

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

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

[ ]:
```
runner = ClientRunner(hw_arch=chosen_hw_arch)
hn, npz = runner.translate_tf_model(ckpt_path, model_name, start_node_names=[start_
↪node], end_node_names=[end_node])
```


4.2.2. Hailo Archive

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

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

Save the parsed model in a Hailo Archive file:

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

Visualize the graph with the visualizer tool:

[ ]:
```
!hailo visualizer {hailo_model_har_name} --no-browser
SVG('resnet_v1_18.svg')
```

Run the profiler tool:

This command will open the HTML report in the browser.

[ ]:
```
!hailo profiler {hailo_model_har_name}
# Profiles the model in pre_placement mode, which yields a fast performance estimation
```

4.2.3. Parsing Example from ONNX to HAR

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

Choose the ONNX file to be used throughout the example:

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

Initialize a ClientRunner and use the translate_onnx_model method.

Arguments:

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

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

[ ]:
```
runner = ClientRunner(hw_arch=chosen_hw_arch)
hn, npz = runner.translate_onnx_model(onnx_path, onnx_model_name,
                    start_node_names=['input_0'],
                    end_node_names=['890', '825', '760'],
                    net_input_shapes={'input_0':[1, 3, 640, 640]})
```


4.2.4. Parsing Example from Tensorflow 2

Parsing the Tensorflow 2.x SavedModel format is similar to parsing Tensorflow 1.x checkpoints. The Parser identifies the input format automatically.

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

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

runner = ClientRunner(hw_arch=chosen_hw_arch)
hn, npz = runner.translate_tf_model(model_path, model_name)
```

4.2.5. Common Conversion Methods from Tensorflow to Tensorflow Lite

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

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

# Converting the Model to tflite
model = build_small_example_net()
model_name = 'small_example'
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.target_spec.supported_ops = [
 tf.lite.OpsSet.TFLITE_BUILTINS, # enable TensorFlow Lite ops.
 tf.lite.OpsSet.SELECT_TF_OPS # enable TensorFlow ops.
]
tflite_model = converter.convert() # may cause warnings in jupyter notebook, don't
↪worry.
tflite_model_path = '../models/small_example.tflite'
with tf.io.gfile.GFile(tflite_model_path, 'wb') as f:
  f.write(tflite_model)

# Parsing the model to Hailo format
runner = ClientRunner(hw_arch=chosen_hw_arch)
hn, npz = runner.translate_tf_model(tflite_model_path, model_name)
```

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

(continues on next page)

(continued from previous page)

```
# Parsing the model to Hailo format
runner = ClientRunner(hw_arch=chosen_hw_arch)
hn, npz = runner.translate_tf_model(tflite_model_path, model_name)
```

[ ]:
```
# Third option, convert h5 file to tflite.
model_path = '../models/ew_sub_v0.h5'
model_name = 'ew_sub_example'
model = tf.keras.models.load_model(model_path)
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.target_spec.supported_ops = [
 tf.lite.OpsSet.TFLITE_BUILTINS, # enable TensorFlow Lite ops.
 tf.lite.OpsSet.SELECT_TF_OPS # enable TensorFlow ops.
]
tflite_model = converter.convert()
tflite_model_path = '../models/ew_sub_example.tflite'
with tf.io.gfile.GFile(tflite_model_path, 'wb') as f:
  f.write(tflite_model)

# Parsing the model to Hailo format
runner = ClientRunner(hw_arch=chosen_hw_arch)
hn, npz = runner.translate_tf_model(tflite_model_path, model_name)
```

4.3. Model Optimization Tutorial

This tutorial will walk you through the process of optimizing your model. The input to this tutorial is a HAR file in Hailo Model state (before optimization; with native weights) and the output will be a quantized HAR file with quantized weights.

Note: For full information about Optimization and Quantization, refer to the `Dataflow Compiler user guide / Model optimization` section.

Requirements:

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

Recommendation:

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

Contents:

- Quick optimization tutorial
- In-depth optimization & evaluation tutorial
- Advanced Model Modifications tutorial
- Compression and Optimization levels

[ ]:
```
# importing everything needed
%matplotlib inline
import os
import json
import numpy as np
import tensorflow as tf
import pandas as pd
import matplotlib.patches as patches

from IPython.display import SVG
```

(continues on next page)

(continued from previous page)

```
from IPython.display import display
from matplotlib import pyplot as plt
from PIL import Image
from hailo_sdk_client import ClientRunner, InferenceContext

IMAGES_TO_VISUALIZE = 5
```

4.3.1. Quick Optimization Tutorial

After you have created the HAR file (or called `runner.translate_tf_model` or `runner.translate_onnx_model`), the next step is to go through the optimization process.

The basic optimization is performed just by calling `runner.optimize(calib_dataset)` (or the CLI `hailo optimize`), as described on the user guide on: Building Models / Model optimization / Model Optimization Workflow.

In order to learn how to deal with common pitfalls, image formats and accuracy, refer to the in-depth section.

[ ]:
```
# First, we will prepare the calibration set. Resize the images to the correct size and crop them.

from tensorflow.python.eager.context import eager_mode

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

    return tf.squeeze(cropped_image)

images_path = '../data'
images_list = [img_name for img_name in os.listdir(images_path) if
       os.path.splitext(img_name)[1] == '.jpg']

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

np.save('calib_set.npy', calib_dataset)
```

[ ]:
```
# Second, we will load our parsed HAR from the Parsing Tutorial

model_name = 'resnet_v1_18'
hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
assert os.path.isfile(hailo_model_har_name), 'Please provide valid path for HAR file'
runner = ClientRunner(har_path=hailo_model_har_name)
# By default it uses the hw_arch that is saved on the HAR. For overriding, use the hw_arch flag.
```

[ ]:
```
# Now we will create a model script, that tells the compiler to add a normalization on the beginning
```

(continues on next page)

(continued from previous page)

```
# of the model (that is why we didn't normalize the calibration set;
# Otherwise we would have to normalize it before using it)
alls = 'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.
↪375])\n' # Batch size is 8 by default

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

# Call Optimize to perform the optimization process
runner.optimize(calib_dataset)

# Save the result state to a Quantized HAR file
quantized_model_har_path = '{}_quantized_model.har'.format(model_name)
runner.save_har(quantized_model_har_path)
```

That concludes the quick tutorial.

4.3.2. In-depth optimization tutorial

The advanced optimization process (see the diagram in the user guide on: Building Models / Model optimization / Model Optimization Workflow), is comprised of the following steps:

1. Test your parsed `Native model` before any changes are made (still on floating point precision), to see that your pre and post processing code works well with the start and end nodes provides. The Native model will match the results of the original model, in between the start_node_names and the end_node_names you have provided during the Parsing stage.
2. Optional: Apply Model Modifications (like input Normalization layer, YUY2 to RGB conversion, changing output activations and others), using a `model script`.
3. Test your `FP Optimized model` (the model after floating point operations and modifications) to see that you achieved the required results.
    - Note: Remember to update your pre and post processing code to match the changes in the model. For example, if you have added normalization to the model, remove the normalization code from your pre-processing code, and feed un-normalized images to the model. If you have added softmax on the outputs, remove the softmax from your post-processing code. Etc.
4. Now perform `Optimization` to the model, using a calibration set you have prepared. The result is a `Quantized model`, that has some degradation compared to the pre-quantized model.
    - Note: The format of calibration set is the same as you have used as inputs for the modified model. For example, if you have added a normalization layer to the model, the calibration set should not be normalized. If you haven't added this layer, you should pre-process and normalize your images.
5. Test your quantized model using the same already-validated code for the pre and post processing.
    - If there is a degradation, you know it's not because of input/output formats, since they were already verified with the pre-quantized model.
6. If you would like to increase the accuracy of the quantized model, you can optimize again using a `model script` to affect the optimization process.
    - Note: The most basic method is to raise the optimization_level, an example model script command is `model_optimization_flavor(optimization_level=4)`. The advanced method is to use the Layer Analysis Tool, presented on the next tutorial.
    - Note: If the accuracy is good, you might want to increase the performance by using 4-bit weights. This is done using compression_level, an example model script command is `model_optimization_flavor(compression_level=2)`.
7. During your next tutorials, compile then run on the on the actual device. Expect the input and output values to be similar to the quantized model's.

The testing (whether on Native, Modified or Quantized model) is performed using our `Emulator` feature, that will be described in this tutorial.

We will now guide you through the steps described above.

Preliminary step: Create testing environment

Hailo offers an `Emulator` for testing the model in its different states. The emulator is implemented as a Tensorflow graph, and its results are the return value of `runner.infer(context, network_input)`. To get inference results, run this API within the context manager `runner.infer_context(inference_context)` where the inference context is one of: `[InferenceContext.SDK_NATIVE, InferenceContext.SDK_FP_OPTIMIZED, InferenceContext.SDK_QUANTIZED]`: - `InferenceContext.SDK_NATIVE`: Testing method of Step 1 on the optimization process steps (`Native model`). Runs the model as is without any changes. You can use it to make sure your model has been converted properly into Hailo's internal representation. Should yield exact results as the original model. - `InferenceContext.SDK_FP_OPTIMIZED`: Testing method of Step 3 on the optimization process steps (`Modified model`). The modified model represents the Hailo model prior to quantization, and is the result of performing model modifications (e.g. normalizing/resizing inputs) and full precision optimizations (e.g. tiled squeeze & excite, equalization). As a result, inference results may vary slightly from the native results. - `InferenceContext.SDK_QUANTIZED`: Testing method of Step 5 on the optimization process steps (`Quantized model`). This inference context emulates the hardware implementation, and is useful for measuring the overall accuracy and degradation of the quantized model. This measurement is done against the original model over large datasets, prior to running inference on the actual Hailo device.

Preliminary step: Create pre and post processing functions

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

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

    if normalize:
      # Default normalization parameters for ImageNet
      cropped_image = (cropped_image - [123.675, 116.28, 103.53]) / [58.395, 57.12,
↪57.375]

    return tf.squeeze(cropped_image)


# ----------------------------------------------------
# Post processing (what to do with the model's outputs)
# ----------------------------------------------------
def _get_imagenet_labels(json_path='../data/imagenet_names.json'):
```

(continues on next page)

(continued from previous page)

```
  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 postproc(results):
  labels = []
  scores = []
  for result in results:
    top_ind = np.argmax(result)
    cur_label = imagenet_labels[top_ind]
    cur_score = 100*result[top_ind]
    labels.append(cur_label)
    scores.append(cur_score)
  return scores, labels


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

def visualize_results(
  images,
  first_scores, first_labels,
  second_scores=None, second_labels=None,
  first_title='Full Precision', second_title='Other'):
  # Deal with input arguments
  assert (second_scores is None and second_labels is None) or (second_scores is not
↪None and second_labels is not None), \
    "second_scores and second_labels must both be supplied, or both not be supplied"
  assert len(images) == len(first_scores) and len(images) == len(first_labels),
↪"lengths of inputs must be equal"

  show_only_first = (second_scores is None)
  if not show_only_first:
    assert len(images) == len(second_scores) and len(images) == len(second_labels),
↪"lengths of inputs must be equal"

  # Display
  for img_idx in range(len(images)):
    plt.figure()
    plt.imshow(mynorm(images[img_idx]))

    if not show_only_first:
      plt.title('{0}: top-1 class is {1}. Confidence is {2:.2f}%,\n\
          {3}: top-1 class is {4}. Confidence is {5:.2f}%'.format(
            first_title,
            first_labels[img_idx], first_scores[img_idx],
            second_title,
            second_labels[img_idx], second_scores[img_idx]))
    else:
      plt.title('{0}: top-1 class is {1}. Confidence is {2:.2f}%'.format(
            first_title,
            first_labels[img_idx], first_scores[img_idx]))
```


Step 1: Test Native Model

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

```
[ ]: model_name = 'resnet_v1_18'
hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
assert os.path.isfile(hailo_model_har_name), 'Please provide valid path for HAR file'
runner = ClientRunner(har_path=hailo_model_har_name)
# By default it uses the hw_arch that is saved on the HAR. For overriding, use the hw_
↪arch flag.
```

Prepare the images to be fed to the model:

```
[ ]: images_path = '../data'
images_list = [img_name for img_name in os.listdir(images_path) if
        os.path.splitext(img_name)[1] == '.jpg']

# Create an un-normalized dataset for visualization
image_dataset = np.zeros((len(images_list), 224, 224, 3), dtype=np.float32)
# Create a normalized dataset to feed into the Native emulator
image_dataset_normalized = np.zeros((len(images_list), 224, 224, 3), dtype=np.
↪float32)
for idx, img_name in enumerate(sorted(images_list)):
  img = np.array(Image.open(os.path.join(images_path, img_name)))
  img_preproc = preproc(img)
  image_dataset[idx,:,:,:] = img_preproc.numpy()
  img_preproc_norm = preproc(img, normalize=True)
  image_dataset_normalized[idx,:,:,:] = img_preproc_norm.numpy()
```

Now call the `Native` emulator:

```
[ ]: %matplotlib inline

# Notice that we use the normalized images, because normalization is not in the model
with runner.infer_context(InferenceContext.SDK_NATIVE) as ctx:
  native_res = runner.infer(ctx, image_dataset_normalized[:IMAGES_TO_VISUALIZE,:,:
↪,:])

native_scores, native_labels = postproc(native_res)
visualize_results(image_dataset[:IMAGES_TO_VISUALIZE,:,:,:], native_scores,
↪native_labels)
```

Steps 2,3: Apply Model Modifications, and Test Modified Model

The `Model Script` is a text file that includes `model script commands`, affecting the stages of the compiler.

In the next steps we will perform the following: - Create a model script for the Optimization process, that also includes the model modifications. - Load the model script (it wont be applied yet) - Call runner.optimize_full_precision() to apply the model modifications (instead, we could call optimize() that also applies the model modifications) - Then we could call the SDK_FP_OPTIMIZED emulation context

```
[ ]: model_script_lines = [
  # Add normalization layer with mean [123.675, 116.28, 103.53] and std [58.395, 57.12,
↪ 57.375])
  'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.
↪375])\n'
  # For multiple input nodes:
  # {normalization_layer_name_1} = normalization([list of means per channel], [list
↪of stds per channel], {input_layer_name_1_from_hn})\n',
  # {normalization_layer_name_2} = normalization([list of means per channel], [list
↪of stds per channel], {input_layer_name_2_from_hn})\n',
```

(continues on next page)

(continued from previous page)

```
  # ...
]

# Load the model script to ClientRunner so it will be considered on optimization
runner.load_model_script(''.join(model_script_lines))
runner.optimize_full_precision()
```

[ ]:
```
%matplotlib inline

# Notice that we use the original images, because normalization is IN the model
with runner.infer_context(InferenceContext.SDK_FP_OPTIMIZED) as ctx:
  modified_res = runner.infer(ctx, image_dataset[:IMAGES_TO_VISUALIZE,:,:,:])

modified_scores, modified_labels = postproc(modified_res)

visualize_results(
  image_dataset[:IMAGES_TO_VISUALIZE,:,:,:],
  native_scores, native_labels,
  modified_scores, modified_labels,
  second_title='FP Modified')
```

Step 4,5: Optimize the model and test its accuracy

1. We will create a calibration dataset (will be the same as the input to the modified model)
2. Then we will call Optimize
3. Then we will test its accuracy vs. the modified model

[ ]:
```
# We are using the original images, just as the input to the SDK_FP_OPTIMIZED emulator
calib_dataset = image_dataset.astype(np.uint8)

# For calling Optimize, you can just use the short version: runner.optimize(calib_
↪dataset)
# We are using here the more general approach, that works also with multiple input
↪nodes.
# The calibration dataset could also be a dictionary with the format:
# {input_layer_name_1_from_hn: layer_1_calib_dataset, input_layer_name_2_from_hn:
↪layer_2_calib_dataset}
hn_layers = runner.get_hn_dict()['layers']
print([layer for layer in hn_layers if hn_layers[layer]['type'] == 'input_layer']) #
↪See available input layer names
calib_dataset_dict = {'resnet_v1_18/input_layer1': calib_dataset} # In our case
↪there is only one input layer
runner.optimize(calib_dataset_dict)
```

[ ]:
```
%matplotlib inline

# Notice that we use the original images, because normalization is in the model
with runner.infer_context(InferenceContext.SDK_QUANTIZED) as ctx:
  quantized_res = runner.infer(ctx, image_dataset[:IMAGES_TO_VISUALIZE,:,:,:])

quantized_scores, quantized_labels = postproc(quantized_res)

visualize_results(
  image_dataset[:IMAGES_TO_VISUALIZE,:,:,:],
  modified_scores, modified_labels,
  quantized_scores, quantized_labels,
  first_title='FP Modified', second_title='Quantized')
```



```
[ ]: # Let's save the runner's state to a Quantized HAR
quantized_model_har_path = '{}_quantized_model.har'.format(model_name)
runner.save_har(quantized_model_har_path)
```

Step 6: How to Raise Accuracy

If you would like to increase the accuracy of the quantized model, you can optimize again using a model script to affect the optimization process.

You have several tools at your disposal.

- Verify that you have a GPU and at least 1024 images in your calibration set
- Raise the optimization_level value using the model_optimization_flavor command. If it fails on high GPU memory, try lowering the batch_size as described on the last example
- Decrease the compression_level value using the model_optimization_flavor command (default is 0, lowest option)
- Set the output layer(s) to use 16-bit accuracy using the command quantization_param(output_layer_name, precision_mode=a16_w16). Note that the DFC will set 16-bit output automatically for small enough outputs.
- Use the Layer Noise Analysis tools to find layers with low SNR, and affect their quantization using weight or activation clipping (see the next tutorial)
- Experiment with the FineTune parameters (refer to the user guide for more details)

For more information refer the user guide in: Building Models / Model optimization / Model Optimization Workflow / Debugging accuracy.

This completes the in-depth optimization tutorial.

4.3.3. Advanced Model Modifications Tutorial

Adding on-chip input format conversion through model script commands

This block will apply model modification commands using a model script. We will be adding a YUY2->'YUV <https://en.wikipedia.org/wiki/YUV>'__->RGB conversion

Unlike the normalization layer, which you could simulate with the SDK_FP_OPTIMIZED and SDK_QUANTIZED emulators, not all format conversions are supported in the emulator (for more information see the `Dataflow Compiler user guide / Model optimization` section). Every conversion that runs in the emulator affects the calibration set, and the user should supply the set accordingly. For example, after adding YUV -> RGB format conversion layer, the calibration set is expected to be in YUV format. However, for some conversions the user may choose to skip the conversion in emulation and to use the original calibration set instead. For instance, in this tutorial we will use YUY2 -> YUV layer without emulation because we want the emulator input and the calibration dataset to remain in YUV format. The format conversion layer would be relevant only when running the compiled .hef file on device.

Note: The NV21 -> YUV conversion is not supported in emulation.

The steps are:

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

- To create input conversion after a specific layer: yuv_to_rgb_layer = input_conversion(input_layer1, yuv_to_rgb)

- To include the conversion in the optimization process: yuv_to_rgb_layer = input_conversion(input_layer1, yuv_to_rgb, emulator_support=True)
- To create input conversion after all input layers: net_scope1/yuv2rgb1, net_scope2/yuv2rgb2 = input_conversion(yuv_to_rgb)

[ ]:
```
# Let's load the original parsed model again
model_name = 'resnet_v1_18'
hailo_model_har_name = '{}_hailo_model.har'.format(model_name)
assert os.path.isfile(hailo_model_har_name), 'Please provide valid path for HAR file'
runner = ClientRunner(har_path=hailo_model_har_name)

# We are using a pre-made YUV calibration set
calib_dataset_yuv = np.load('../model_modifications/calib_dataset_yuv.npz')

# Now we're adding yuy2_to_yuv conversion before the yuv_to_rgb and a normalization
↪layer.
# The order of the layers is determined by the order of the commands in the model script:
# First we add normalization to the original input layer -> the input to the network is
↪now normalization1
# Then we add yuv_to_rgb layer, so the order will be: yuv_to_rgb1->normalization1->
↪original_network
# Lastly, we add yuy2_to_yuv layer, so the order will be: yuy2_to_yuv1->yuv_to_rgb1->
↪normalization1->original_network
model_script_commands = [
  'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.
↪375])\n',
  'yuv_to_rgb1 = input_conversion(yuv_to_rgb)\n',
  'yuy2_to_yuv1 = input_conversion(input_layer1, yuy2_to_hailo_yuv)\n'
]
runner.load_model_script(''.join(model_script_commands))

# Notice that we don't have to call runner.optimize_full_precision(), its only an
↪intermediate step
# to be able to use SdkFPOptimize emulator before Optimization.
runner.optimize(calib_dataset_yuv['yuv_dataset'])

modified_model_har_name = '{}_modified.har'.format(model_name)
runner.save_har(modified_model_har_name)
!hailo visualizer {modified_model_har_name} --no-browser
SVG('resnet_v1_18.svg')
```

Adding on-chip input resize through model script commands

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

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

np.save('calib_set_480_640.npy', calib_dataset_new)
plt.imshow(img)
plt.title('Original image')
```

(continues on next page)

(continued from previous page)

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

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

calib_dataset_large = np.load('calib_set_480_640.npy')

# Add a bilinear resize from 480x640 to the network's input size - in this case, 224x224.
# The order of the layers is determined by the order of the commands in the model script:
# First we add normalization to the original input layer -> the input to the network is
↪now normalization1
# Then we add resize layer, so the order will be: resize_input1->normalization1->
↪original_network
model_script_commands = [
  'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.
↪375])\n',
  'resize_input1 = resize_input([480,640])\n'
]

runner.load_model_script(''.join(model_script_commands))
calib_dataset_dict = {'resnet_v1_18/input_layer1': calib_dataset_large} # In our
↪case there is only one input layer
runner.optimize(calib_dataset_dict)

modified_model_har_name = '{}_resized.har'.format(model_name)
runner.save_har(modified_model_har_name)
!hailo visualizer {modified_model_har_name} --no-browser
SVG('resnet_v1_18.svg')
```

Adding non-maximum suppression (NMS) layer through model script commands

This block will add an NMS layer at the end of the network through the model script command: `nms_postprocess`. The following arguments can be used: * Config json: an external json file that allows you to change the NMS parameters (can be skipped for the default configuration). * Meta architecture: which meta architecture to use (for example, `yolov5`, `ssd`, etc). In this example, we will use `yolov5`. * Engine: defines the inference device for running the nms: `nn_core`, `cpu` or `auto` (this example shows `cpu`).

Usage: * Initialize Client Runner * Translate a YOLOv5 model * Load the model script with the NMS command * Use the `optimize_full_precision()` API to apply the command (Note that `optimize()` API can also be used) * Display inference result

[ ]:
```
model_name = 'yolov5s'
onnx_path = '../models/{}.onnx'.format(model_name)
assert os.path.isfile(onnx_path), 'Please provide valid path for ONNX file'

# Initialize a new client runner
runner = ClientRunner(hw_arch='hailo8')
# You can use any other hw_arch as well.

# Translate YOLO model from ONNX
runner.translate_onnx_model(onnx_path, end_node_names=['Conv_298', 'Conv_248',
↪'Conv_198'])
# Note: NMS will be detected automatically, with a message that contains:
#  - 'original layer name': {'w': [WIDTHS], 'h': [HEIGHTS], 'stride': STRIDE,
↪'encoded_layer': TRANSLATED_LAYER_NAME}
```

(continues on next page)

(continued from previous page)

```
# Then you should use nms_postprocess(meta_arch=yolov5) to add the NMS.

# Add model script with NMS layer at the network's output.
model_script_commands = [
  'normalization1 = normalization([0.0, 0.0, 0.0], [255.0, 255.0, 255.0])\n',
  'resize_input1 = resize_input([480,640])\n',
  'nms_postprocess(meta_arch=yolov5, engine=cpu, nms_scores_th=0.2, nms_iou_th=0.
↪4)\n',
]
# Note: Scores threshold of 0.0 means no filtering, 1.0 means maximal filtering. IoU
↪thresholds are opposite: 1.0 means filtering boxes only if they are equal, and 0.0
↪means filtering with minimal overlap.
runner.load_model_script(''.join(model_script_commands))

# Apply model script changes
runner.optimize_full_precision()

# Infer an image with the Hailo Emulator
with runner.infer_context(InferenceContext.SDK_FP_OPTIMIZED) as ctx:
  nms_output = runner.infer(ctx, calib_dataset_new[:32, ...])
HEIGHT = 480
WIDTH = 640
# For each image
for i in range(32):
  found_any = False
  min_score = None
  max_score = None
  # Go over all classes
  for class_index in range(nms_output.shape[0]):
    score, box = nms_output[i][class_index, 4, :], nms_output[i][class_index, 0:4, :]
    # Go over all detections
    for detection_idx in range(box.shape[1]):
      cur_score = score[detection_idx]
      # Discard null detections (because the output tensor is always padded to MAX_
↪DETECTIONS on the emulator interface.
      # Note: On HailoRT APIs (that are used on the Inference Tutorial, and with C++
↪APIs), the default is a list per class. For more information look for NMS on the
↪HailoRT user guide.
      if cur_score == 0:
        continue

      # Plotting code
      if not found_any:
        found_any = True
        fig, ax = plt.subplots()
        ax.imshow(Image.fromarray(np.array(calib_dataset_new[i], np.uint8)))
      if min_score is None or cur_score < min_score:
        min_score = cur_score
      if max_score is None or cur_score > max_score:
        max_score = cur_score
      y_min, x_min, y_max, x_max = box[0, detection_idx] * HEIGHT, box[1, detection_
↪idx] * WIDTH, box[2, detection_idx] * HEIGHT, box[3, detection_idx] * WIDTH
      center, width, height = (x_min, y_min), x_max - x_min, y_max - y_min
      # draw the box on the input image
      rect = patches.Rectangle(center, width, height, linewidth=1, edgecolor='r',
↪facecolor='none')
      ax.add_patch(rect)

  if found_any:
    plt.title('Plot of high score boxes. Scores between {:.2f} and {:.2f}'.
↪format(min_score, max_score))
```

(continues on next page)

(continued from previous page)

```
    plt.show()
```

4.3.4. Advanced Optimization - Compression and Optimization Levels

For aggressive quantization (compress significant amount of weights to 4-bits), we'll need to use higher optimization level to obtain good results. For quick iterations we always recommend starting with the default setting of the model optimizer (optimization_level=2, compression_level=1). However, when moving to production, we recommended to work at the highest complexity level to achieve optimal accuracy. With regards to compression, users should increase it when the overall throughput/latency of the model is not good enough. Note that increasing compression would have negligible effect on power-consumption so the motivation to work with higher compression level is mainly due to FPS considerations.

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

[ ]:
```
alls_lines = [
  'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.
↪375])\n',
  'model_optimization_flavor(optimization_level=0, compression_level=4, batch_
↪size=2)\n' # Batch size is 8 by default
]
# -- Reduces weights memory by 80% !

runner = ClientRunner(har_path=hailo_model_har_name)

runner.load_model_script(''.join(alls_lines))
runner.optimize(calib_dataset)
```

[ ]:
```
%matplotlib inline

images = calib_dataset[:IMAGES_TO_VISUALIZE,:,:,:]
with runner.infer_context(InferenceContext.SDK_FP_OPTIMIZED) as ctx:
  modified_res = runner.infer(ctx, images)
with runner.infer_context(InferenceContext.SDK_QUANTIZED) as ctx:
  quantized_res = runner.infer(ctx, images)

modified_scores, modified_labels = postproc(modified_res)
quantized_scores, quantized_labels = postproc(quantized_res)

visualize_results(
  image_dataset[:IMAGES_TO_VISUALIZE,:,:,:],
  modified_scores, modified_labels,
  quantized_scores, quantized_labels,
  first_title='FP Modified', second_title='Quantized')
```

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

[ ]:
```
%matplotlib inline

images = calib_dataset[:IMAGES_TO_VISUALIZE,:,:,:]

alls_lines = [
  'normalization1 = normalization([123.675, 116.28, 103.53], [58.395, 57.12, 57.
↪375])\n',
  'model_optimization_flavor(optimization_level=2, compression_level=4, batch_
↪size=2)\n' # Batch size is 8 by default
```

(continues on next page)

(continued from previous page)

```
]
# -- Reduces weights memory by 80% !

runner = ClientRunner(har_path=hailo_model_har_name)
runner.load_model_script(''.join(alls_lines))
runner.optimize(calib_dataset)

modified_scores, modified_labels = postproc(modified_res)
quantized_scores_new, quantized_labels_new = postproc(quantized_res)

visualize_results(
  image_dataset[:IMAGES_TO_VISUALIZE,:,:,:],
  modified_scores, modified_labels,
  quantized_scores_new, quantized_labels_new,
  first_title='FP Modified', second_title='Quantized')
```

```
[ ]: print('Full precision predictions:            {0}\n'\
       'Quantized predictions (with optimization_level=2): {1} ({2}/{3})\n'\
       'Quantized predictions (with optimization_level=0): {4} ({5}/{6})'.
     ↪format(modified_labels,
                                          quantized_labels_new,
                                          sum(np.array(modified_labels) == np.
     ↪array(quantized_labels_new)),
                                          len(modified_labels),
                                          quantized_labels,
                                          sum(np.array(modified_labels) == np.
     ↪array(quantized_labels)),
                                          len(modified_labels)))
```

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

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

4.4. Compilation Tutorial

4.4.1. Hailo Compilation Example from Hailo Archive Quantized Model to HEF

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

Requirements:

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

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

```
[ ]: from hailo_sdk_client import ClientRunner
```

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

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

Load the network to the ClientRunner:


```
[ ]: runner = ClientRunner(har_path=quantized_model_har_path)
     # By default it uses the hw_arch that is saved on the HAR. It is not recommended to
     ↪change the hw_arch after Optimization.
```

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

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

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

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

4.4.2. Profiler tool

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

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

```
[ ]: har_path = '{}_compiled_model.har'.format(model_name)
     runner.save_har(har_path)
     !hailo profiler {har_path} --mode post_placement
     # Now we are using the more accurate post_placement mode.
```

Note:

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

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

4.5. Inference tutorial

This tutorial will walk you through the inference process.

Requirements:

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

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

4.5.1. Standalone Hardware Deployment

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

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

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

- Target HW configuration

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

First create the desired target object.

Note: If you are using the Hailo-15 device, the tutorial and the `resnet_v1_18.hef` file should be copied to and run on the device itself.

[ ]:
```
from multiprocessing import Process

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

# Setting VDevice params to disable the Hailort service feature
params = VDevice.create_params()
params.scheduling_algorithm = HailoSchedulingAlgorithm.NONE

# 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 = VDevice(params=params)

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

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

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

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

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


Running Hardware Inference

Infer the model and then display the output shape:

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

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

4.5.2. Streaming Inference

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

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

Define the send and receive functions:

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


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

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

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

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

# Clean pcie target
target.release()
print('Done')
```


4.5.3. DFC Inference in Tensorflow Environment

Note: This section is not yet supported on the Hailo-15, as it requires the Dataflow Compiler to be installed on the device.

The `runner.infer()` method that was used for emulation in the model optimization tutorial can also be used for running inference on the Hailo device inside the `infer_context` environment. Before calling this function with hardware context, please make sure a HEF file is loaded to your runner, by one of the options: calling `runner.compile()`, loading a complied HAR using `runner.load_har()`, or setting the HEF attribute `runner.hef`.

First, create the runner and load a compiled HAR:

```
[ ]: from hailo_sdk_client import ClientRunner

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

Calling `runner.infer()` within inference HW context to run on the Hailo device (`InferenceContext.SDK_HAILO_HW`):

```
[ ]: import numpy as np
from hailo_sdk_client import InferenceContext
from hailo_platform import HEF

model_name = 'resnet_v1_18'
hef_path = f'{model_name}.hef'
hef = HEF(hef_path)
input_vstream_info = hef.get_input_vstream_infos()[0]
image_height, image_width, channels = input_vstream_info.shape
num_of_images = 10
low, high = 2, 20

with runner.infer_context(InferenceContext.SDK_HAILO_HW) as hw_ctx:
  # Running hardware inference:
  for i in range(10):
    dataset = np.random.randint(low, high, (num_of_images, image_height, image_
↪width, channels)).astype(np.uint8)
    results = runner.infer(hw_ctx, dataset)
```

4.5.4. Runtime Profiler

This will demonstrate the usage of the HTML runtime profiler.

Note: On the Hailo-15 device:

- The `hailortcli run` command should be run on the device itself
- Then the created json file should be copied to the Dataflow Compiler environment
- Then the `hailo profiler` command should be used

```
[ ]: model_name = 'resnet_v1_18'
hef_path = f'{model_name}.hef'
compiled_har_path = f'{model_name}_compiled_model.har'
runtime_data_path = f'runtime_data_{model_name}.json'

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


Notes on the Runtime Profiler

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

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

The runtime FPS is also displayed on the hailortcli output.

4.6. Accuracy Analysis Tool Tutorial

This is an advanced tutorial; You may skip it if your accuracy results were satisfying. Before using it, make sure that your native (pre-quantization) results are satisfying. For more details refer to the Model Optimization Tutorial.

This tutorial will guide you through a model quantization analysis to break down the quantization noise per layer. The tutorial is intended to guide the user in using Hailo analyze noise tool, by using it to analyze the classification model MobileNet-v3-Large-Minimalistic.

The flow is mainly comprised of:

- Input definitions: Defining the paths to the model and data for analysis.
- Preparing the model: Initial Parse and Optimize of the model.
- Accuracy analysis: This step is the heart of the tool, and computes the quantization noise of each layer output, when the given layer is the **only** quantized layer, while the rest are kept in full precision. This highlights the quantization sensitivity of the model to that specific layer noise.
- Visualizing the results: Walk through the results of the accuracy analysis and explain the different graphs and information.
- Re-optimizing the model: After debugging the noise we repeat the optimization process to improve the results.

Requirements:

- Run this code in Jupyter notebook, see the Introduction tutorial for more details.
- Verify that you've completed the Parsing tutorial and the Model Optimization tutorial or generated analysis data in another way.

```
[ ]: import os
     import matplotlib.pyplot as plt
     import numpy as np
     import tensorflow as tf
     from hailo_sdk_client import ClientRunner
```

4.6.1. Input Definitions

- Model path: path to the model to be used in this tutorial
- data_path: path to preprocessed .npy image files for optimization and analysis

```
[ ]: model_name = 'v3-large-minimalistic_224_1.0_float'
     model_path = '../models/' + model_name + '.tflite'
     assert os.path.isfile(model_path), 'Please provide valid path for the model'

     data_path = './calib_set.npy'
```

(continues on next page)

(continued from previous page)

```
assert os.path.isfile(data_path), 'Please provide valid path for a dataset'
har_path = model_name + '.har'
```

It is highly recommended to use GPU when running the analysis tool but if you don't have one in your machine the code will run on the CPU and you can expect long running time.

```
[ ]: if len(tf.config.list_physical_devices('GPU')) == 0:
       print("Warning: you are running the accuracy analysis tool without a GPU, expect
     ↪long running time.")
```

4.6.2. Preparing the Model

In this step, we will parse and optimize the model to prepare it for analysis. For more details checkout the Parsing tutorial and the Model Optimization tutorial.

```
[ ]: runner = ClientRunner(hw_arch='hailo8')
runner.translate_tf_model(model_path, model_name)

model_script = 'normalization1 = normalization([127.5, 127.5, 127.5], [127.5, 127.5,
↪127.5])\n'
runner.load_model_script(model_script)

runner.optimize(data_path)
```

4.6.3. Accuracy Analysis

Though most models work well with our default optimization, some suffer from high quantization noise that induces substantial accuracy degradation. As an example, we choose the MobileNet-v3-Large-Minimalistic neural network model that, due to its structural characteristics, results in a high degradation of 6% for Top-1 accuracy on the ImageNet-1K validation dataset.

To analyze the source of degradation, we will use the Hailo `analyze_noise` API. The analysis tool uses a given dataset to measure noise level in each layer and allows us to pinpoint problematic layers that we should handle. The analysis tool uses the entire dataset by default, to limit the number of images you can use the `data_count` argument. It is recommended to use at least 64 images, preferably not from the calibration set.

The following is equivalent to running the CLI command:

```
hailo analyze-noise quantized_model_har_path --data-path data_path --batch-
size 2 --data-count 64
```

```
[ ]: analysis_data = runner.analyze_noise(data_path, batch_size=2, data_count=64) #
     ↪Batch size is 8 by default
     runner.save_har(har_path)
```

4.6.4. Visualizing the Results

In this section we give a general explanation for the noise analysis report. To visualize the accuracy analysis results and debug our quantization noise we will use the Hailo profiler. The Hailo profiler will generate an HTML report with all the information of your model. In the ACCURACY tab of the report, we will find all the relevant information for this tutorial:

- SNR chart: the signal-to-noise chart on top shows the sensitivity of each layer to quantization (for each output layer). To measure the quantization noise of each layer's output, we iterate over all layers when the given layer is the **only** quantized layer, while the rest are kept in full precision. The graph shows the SNR values in decibels (dB) and any value higher than 10 should be fine (higher is better). In case an output layer is sensitive

across many layers we recommend re-quantize with one of the following global optimization (model script commands):

- Configure the output layer to 16bit output. For example, using the model script command: `quantization_param(output_layer1, precision_mode=a16_w16)`.
- When possible, offload output activation to the accelerator. For example, the following command adds sigmoid activation to the output layer conv51: `change_output_activation(conv51, sigmoid)` and should be used to offload sigmoid from postprocessing code to the accelerator.
- Use massive fine tune which is enabled by default in optimization_level=2 but can be customized. For example, specific fine-tune command: `post_quantization_optimization(finetune, policy=enabled, learning_rate=0.0001, epochs=8, batch_size=4, dataset_size=4000)`. Other useful attributes to this command are: loss_layer_names, loss_factors and loss_types which allows the user to manually edit the loss function of the fine tune training. In case fine tune failed due to GPU memory, you could try to use lower batch_size.
- Increase the optimization level. For example, `model_optimization_flavor(optimization_level=4)` will set the highest optimization level (default is 2).
- Decrease the compression level. For example, `model_optimization_flavor(compression_level=0)` will disable compression (default value is 1).

- Layers information: this section provide per-layer detailed information that would help you debug local quantization errors in your model, for example, specific layer that is very sensitive for quantization. Note that quantization noise may stem from the layers' weights, activations or both.
  - Weight Histogram: this graph shows the weights distribution and can help to identify outliers. If outliers exist in the weight distribution, the following command can be used to clip it, for example, clip the kernel values of conv27: `pre_quantization_optimization(weights_clipping, layers=[conv27], mode=percentile, clipping_values=[0.01, 99.99])`
  - Activation Histogram: this graph shows the activation distribution as collected by the layer noise analysis tool. Wide activation distribution is a major source of degradation source and in general we strongly recommend using a model with batch normalization after each layer to limit the layer's extremum activation values. Another important argument that affects the activation distribution is the calibration size that was used during quantization, to raise it, use the following command: `model_optimization_config(calibration, calibset_size=512)`, the default value for calibration is 64. In case of outliers in the layers' activation distribution, we recommend using the a clipping command, for example: `pre_quantization_optimization(activation_clipping, layers={*}, mode=percentile, clipping_values=[0.01, 99.99])`
  - Scatter Plot: this graph shows a comparison between full precision and quantized values of the layers' activation. The X-axis of each point in this graph is its value in full precision and Y-axis is the value after quantization. Zero quantization noise means the slope would be exactly one. In case of bias noise you expect to find many points above/below the line that represent imperfect quantization, if this is the case, you should use the following commands: `post_quantization_optimization(bias_correction, policy=enabled)` and `post_quantization_optimization(finetune, policy=disabled)`

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

To examine our results, we will first plot the SNR graph for this specfic model. Note that in general the profiler report should be used but here we will use an alternative visualization.

[ ]:
```
output_snr = analysis_data['noise_results']['v3-large-minimalistic_224_1_0_float/
↪output_layer1']
layers, snr = list(output_snr.keys()), list(output_snr.values())
x = [int(x) for x in np.linspace(0, len(layers) - 1, len(layers))]
print('Worst SNR is obtained in the following layers:\n{}'.format([(layers[x].split(
↪'/')[-1], snr[x]) for x in [int(x) for x in np.argsort(snr)[:3]]]))
plt.plot(x, snr)
```

(continues on next page)

(continued from previous page)

```
plt.title('Per-Layer Logits SNR ({}), higher is better.'.format(model_name))
plt.xlabel('Layer')
plt.ylabel('SNR')
plt.grid()
plt.show()
```

4.6.5. Re-optimizing the Model

Next, we will try to improve the model accuracy results by using specific model script commands. Specifically, we will use the `activation_clipping` command on the problematic layers to clip outliers from the output of the layers and `optimization_level=2`. For further information we refer the user to the full Accuracy report in the profiler HTML.

```
[ ]: runner = ClientRunner(hw_arch='hailo8')
runner.translate_tf_model(model_path, model_name)

model_script_commands = [
  'normalization1 = normalization([127.5, 127.5, 127.5], [127.5, 127.5, 127.5])\n',
  'model_optimization_config(calibration, calibset_size=128)\n',
  'pre_quantization_optimization(activation_clipping, layers=[dw1, conv2, conv3],
↪mode=percentile, clipping_values=[0.5, 99.5])\n',
  'pre_quantization_optimization(weights_clipping, layers=[dw1], mode=percentile,
↪clipping_values=[0.0, 99.99])\n',
  'model_optimization_flavor(optimization_level=2, compression_level=0)\n',
]
runner.load_model_script(''.join(model_script_commands))

runner.optimize(data_path)

analysis_data = runner.analyze_noise(data_path, batch_size=2, data_count=64) #
↪Batch size is 8 by default
runner.save_har(har_path)

!hailo profiler {har_path}
```

After fixing the optimization process, we were able to reduce the model degaradtion to 1% (Top-1 accuracy on the ImageNet-1K validation dataset) which is usually our target goal for classification models.

The improvement can also be seen from the new SNR graph:

```
[ ]: output_snr = analysis_data['noise_results']['v3-large-minimalistic_224_1_0_float/
↪output_layer1']
layers, snr = list(output_snr.keys()), list(output_snr.values())
x = [int(x) for x in np.linspace(0, len(layers) - 1, len(layers))]
print('Worst SNR is obtained in the following layers:\n{}'.format([(layers[x].split(
↪'/')[-1], snr[x]) for x in [int(x) for x in np.argsort(snr)[:3]]]))
plt.plot(x, snr)
plt.title('Per-Layer Logits SNR ({}), higher is better.'.format(model_name))
plt.xlabel('Layer')
plt.ylabel('SNR')
plt.grid()
plt.show()
```


4.7. Quantization Aware Training Tutorial

This tutorial is intended for advanced users. If you are satisfied with your previous accuracy results, you may choose to skip it.

This guide will walk you through the steps of performing quantization aware training (QAT) using Hailo's quantized model. It is assumed that you already have a background in training deep neural networks.

Quantization-aware training refers to a set of algorithms that incorporate full network training in a quantized domain. The technique utilizes the straight-through estimator (STE) concept to allow for backpropagation through non-differentiable operations, such as rounding and clipping, during the training process. In deep learning literature, QAT typically refers to an extended training procedure using the full dataset, labels, and multiple GPUs, similar to the original training process. However, it can also be applied in other scenarios.

The main differences between the quantization-aware training method and the optimization method shown in previous tutorials are:

- QAT enables training using labeled data, whereas the FineTune algorithm (*Model Optimization Tutorial*) is limited to training using knowledge distillation from the full precision model.
- QAT supports running on multiple GPUs for faster training.
- QAT allows for the use of a pipeline of networks or the integration of post-processing functions into the training procedure.

In summary, QAT is a useful tool for training quantized models with labeled data and supports multi-GPU training and integration of post-processing functions. Currently, Hailo QAT only supports Keras.

The remainder of this tutorial will cover the following steps:

- Input definitions: In this step, we will prepare the dataset and model for training and testing.
- Full precision training: A short training procedure will be run to initialize the model's weights.
  - In real scenarios, a complete full precision training procedure should take place here. In this notebook, the full precision training has been shorterned to simplify the tutorial.
- Translation of the model: The model will be exported to TFlite, parsed, optimized, and evaluated using the Hailo toolchain.
- Running QAT: Finally, quantization-aware training will be performed on the quantized model to optimize its accuracy.

Requirements:

- Run this code in Jupyter notebook, see the Introduction tutorial for more details.

```
[ ]: # imports
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from hailo_sdk_client import ClientRunner, InferenceContext
```


4.7.1. Input Definitions

The input definitions step of this tutorial involves using the MNIST dataset and a simple Convolutional Neural Network (CNN). The code provided will download the dataset and prepare it for training and evaluation.

```
[ ]: # Model parameters
num_classes = 10
input_shape = (28, 28, 1)

# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Prepare the dataset
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
print("Total number of training samples: {}".format(x_train.shape[0]))
print("Total number of testing samples: {}".format(x_test.shape[0]))
```

```
[ ]: # Define the model
model = tf.keras.Sequential(
  [
    tf.keras.Input(shape=input_shape),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(num_classes, activation="softmax"),
  ]
)
model.summary()
```

4.7.2. Full Precision Training

In this step, we will run a short training procedure to initialize the model's weights. Only 5,000 images from the full training dataset will be used. The accuracy of the model will be measured on the test dataset.

```
[ ]: # Run short training (using only 5k images)
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=[
↪"accuracy"])
model.fit(x_train[:5000], y_train[:5000], batch_size=128, epochs=1)

# Evalute the results
score = model.evaluate(x_train, y_train)
print("Train accuracy: {0:.3f} (Top-1)".format(100 * score[1]))
score = model.evaluate(x_test, y_test)
print("Test accuracy: {0:.3f} (Top-1)".format(100 * score[1]))
```


4.7.3. Translation of the Model

In this step, we will export our trained model into TFlite format to prepare it for use in the Hailo toolchain. After being translated into TFlite, the model can be parsed, optimized, and inferred using the Hailo DFC. The results of the full precision model will be compared to those of the quantized model. It is important to note that the results of the full precision model should be identical to those obtained from the Keras evaluation, while the quantized model may experience some degradation due to quantization noise.

```
[ ]: # Export the model to TFlite
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
tflite_model_path = 'model.tflite'
with tf.io.gfile.GFile(tflite_model_path, 'wb') as f:
  f.write(tflite_model)
```

```
[ ]: # Parse the TFlite model
runner = ClientRunner(hw_arch='hailo8')
runner.translate_tf_model(tflite_model_path)

# Optimize the model: enforce 60% 4bit weights without optimization
model_script_commands = [
  'model_optimization_config(compression_params, auto_4bit_weights_ratio=0.6)\n'
  'model_optimization_flavor(optimization_level=0)\n'
]

runner.load_model_script(''.join(model_script_commands))
runner.optimize(x_train[:1024])
```

```
[ ]: # Evalute the results
with runner.infer_context(InferenceContext.SDK_QUANTIZED) as q_ctx:
  with runner.infer_context(InferenceContext.SDK_FP_OPTIMIZED) as fp_ctx:
    y_infer_fp = runner.infer(fp_ctx, x_test)
    y_infer_q = runner.infer(q_ctx, x_test)

full_precision_result = np.count_nonzero(np.argmax(y_infer_fp, axis=-1) == np.
↪argmax(y_test, axis=-1)) / len(y_test)
quantize_result = np.count_nonzero(np.argmax(y_infer_q, axis=-1) == np.argmax(y_
↪test, axis=-1)) / len(y_test)
print("Test accuracy (floating point): {0:.3f} (Top-1)".format(100 * full_precision_
↪result))
print("Test accuracy (quantized): {0:.3f} (Top-1)".format(100 * quantize_result))
print("Degradation: {0:.3f}".format(100 * (full_precision_result - quantize_
↪result)))
```

4.7.4. Running QAT

In the final step, we will optimize our quantized model to enhance its accuracy. The `runner.get_keras_model` API will be used to obtain a Keras model initialized with the quantized weights. The model can then be trained using straight-through estimator (STE) method.

- The `tf.distribute.MirroredStrategy` API is being used to enable synchronous training across multiple GPUs on the same machine.
- The `runner.get_keras_model` API must be used with `trainable=True` to allow training (usage of `fit`).
- To the Keras model we can add additional layers, postprocessing or other models. For example, here we are adding a new `tf.keras.layers.Softmax` layer.
- For training, we will use the `fit` API provided by Keras. Training can be done with custumized loss functions and different optimizers.

- After traininig is complete, we should update the `ClientRunner` weights with our update model. This is done using the `runner.set_keras_model` API. Only allowed changes to the Keras model includes weight chganges. Once the new weights were updated we can compile the model with our new weights using the `runner.compile` API.

[ ]:
```
with tf.distribute.MultiWorkerMirroredStrategy().scope():
  with runner.infer_context(InferenceContext.SDK_QUANTIZED) as ctx:

    # get the Hailo Keras model for training
    model = runner.get_keras_model(ctx, trainable=True)

    # add external postprocessing
    new_model = tf.keras.Sequential(
      [
        model,
        tf.keras.layers.Softmax()
      ]
    )

    # adding external loss.
    # note that this compile API only compiles the Keras model but doesn't compile the
↪model to the Hailo HW.
    new_model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(learning_rate=1e-6),
              metrics=["accuracy"])

    # move numpy data to tf.data.Dataset to be used by multiple GPUs
    train_data = tf.data.Dataset.from_tensor_slices((x_train, y_train))
    train_data = train_data.batch(128)
    options = tf.data.Options()
    options.experimental_distribute.auto_shard_policy = tf.data.experimental.
↪AutoShardPolicy.OFF
    train_data = train_data.with_options(options)

    # start QAT
    log = new_model.fit(train_data, batch_size=128, epochs=10)

    # set the Keras model after training
    runner.set_keras_model(model)

# plot training curve
plt.plot(log.history['accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Top-1')
plt.xlabel('Epoch')
plt.grid()
plt.show()
```

[ ]:
```
# Evalute the results
with runner.infer_context(InferenceContext.SDK_QUANTIZED) as q_ctx:
  y_infer_qat = runner.infer(q_ctx, x_test)

qat_result = np.count_nonzero(np.argmax(y_infer_qat, axis=-1) == np.argmax(y_test,
↪axis=-1)) / len(y_test)
print("Test accuracy (quantized) before QAT: {0:.3f} (Top-1)".format(100 * quantize_
↪result))
print("Test accuracy (quantized) after QAT: {0:.3f} (Top-1)".format(100 * qat_
↪result))
print("Accuracy improvment: {0:.3f}".format(100 * (qat_result - quantize_result)))
```


4.7.5. Knowledge Distillation and QAT

QAT can gain additional accuracy with training using a teacher (the full precision model) to train the student model (the quantized model) - knowledge distillation. To use the full precision model, we call the `runner.get_keras_model` API with a different context and change the loss accordingly. In the following code, we are generating a new class `Distiller` to distill the full precision and combine with the supervision of the labels.

- Note that, Hailo's FineTune algorithm works in the same way as well (more information can be found in the Model Optimization Tutorial).

```
[ ]: class Distiller(tf.keras.Model):
  def __init__(self, student, teacher):
    super().__init__()
    self._teacher = teacher
    self._student = student

  def compile(self, optimizer, metrics, student_loss_fn, distillation_loss_fn,
↪alpha=0.1, temperature=3):
    super().compile(optimizer=optimizer, metrics=metrics)
    self._student_loss_fn = student_loss_fn
    self._distillation_loss_fn = distillation_loss_fn
    self._alpha = alpha
    self._temperature = temperature

  def train_step(self, data):
    # unpack data (image, label)
    x, y = data

    # forward pass of teacher
    teacher_predictions = self._teacher(x, training=False)

    with tf.GradientTape() as tape:
      # forward pass of student
      student_predictions = self._student(x, training=True)

      # compute supervised loss
      student_loss = self._student_loss_fn(y, student_predictions)

      # compute distillation loss
      distillation_loss = (
        self._distillation_loss_fn(
          tf.nn.softmax(teacher_predictions / self._temperature, axis=1),
          tf.nn.softmax(student_predictions / self._temperature, axis=1)
        )
        * self._temperature**2
      )

      total_loss = self._alpha * student_loss + (1 - self._alpha) * distillation_loss

    # compute gradients
    trainable_vars = self._student.trainable_variables
    gradients = tape.gradient(total_loss, trainable_vars)

    # update weights
    self.optimizer.apply_gradients(zip(gradients, trainable_vars))

    # update the metrics
    results = {m.name: m.result() for m in self.metrics}
    results.update(
      {"total_loss": total_loss, "student_loss": student_loss, "distillation_loss":
↪ distillation_loss}
    )
    return results
```


[ ]:
```
# Parse the TFlite model
runner = ClientRunner(hw_arch='hailo8')
runner.translate_tf_model(tflite_model_path)

# Optimize the model: enforce 40% 4bit weights without optimization
model_script_commands = [
  'model_optimization_config(compression_params, auto_4bit_weights_ratio=0.6)\n'
  'model_optimization_flavor(optimization_level=0)\n'
]

runner.load_model_script(''.join(model_script_commands))
runner.optimize(x_train[:1024])

with runner.infer_context(InferenceContext.SDK_QUANTIZED) as ctx_q:
  with runner.infer_context(InferenceContext.SDK_FP_OPTIMIZED) as ctx_fp:

    # get the Hailo Keras model for training
    student = runner.get_keras_model(ctx_q, trainable=True)

    # geth the full preicison model for kd
    teacher = runner.get_keras_model(ctx_fp, trainable=False)

    # create the kd model
    distiller = Distiller(student=student, teacher=teacher)
    distiller.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-6),
             metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
             student_loss_fn=tf.keras.losses.CategoricalCrossentropy(),
             distillation_loss_fn=tf.keras.losses.KLDivergence(),
             alpha=0.1,
             temperature=10)

    # start QAT
    log = distiller.fit(x_train, y_train, batch_size=128, epochs=10)

    # set the Keras model after training
    runner.set_keras_model(student)
```

[ ]:
```
# Evalute the results
with runner.infer_context(InferenceContext.SDK_QUANTIZED) as q_ctx:
  y_infer_qat = runner.infer(q_ctx, x_test)

qat_with_kd_result = np.count_nonzero(np.argmax(y_infer_qat, axis=-1) == np.
↪argmax(y_test, axis=-1)) / len(y_test)
print("Test accuracy (quantized) with QAT: {0:.3f} (Top-1)".format(100 * qat_result))
print("Test accuracy (quantized) with QAT and KD: {0:.3f} (Top-1)".format(100 * qat_
↪with_kd_result))
print("Accuracy improvment: {0:.3f}".format(100 * (qat_with_kd_result - qat_
↪result)))
```


5. Building Models

This section describes the process of taking ONNX/TF trained model and compiling them to a Hailo executable binary file (HEF). The main API for this process is the `ClientRunner`. The client runner is a stateful object that handles all stages. In each stage, the client runner can be serialized into an Hailo archive file (HAR) that can be loaded in the future to initialize a new client runner. There are three main stages: Translation, Optimization and Compilation.

1. **Translation:** this process takes an ONNX/TF model and translates it into Hailo's internal representation. For that, the `translate_tf_model()` method or the `translate_onnx_model()` method should be used. For examples, see the *parsing tutorial*. At the end of this stage the state of the runner is changed from **Uninitialized** to **Hailo Model** and new functionality is available:

   A. Running inference on `SDK_NATIVE` context. For further details refer to: *Model Optimization Tutorial*.

   B. Profile your model to obtain initial performance analysis. For example, using the command line interface: `hailo profiler --help`.

   Note: The same functionality can be obtained using the command line interface. For example, `hailo parser {tf, onnx} --help`.

2. **Optimization:** in this stage the model is being optimized before compilation using the `optimize()` method. The *optimize* method runs several steps of optimization including quantization which may degrade the model accuracy; therefore, evaluation is needed to verify the model accuracy. For further information see *Model optimization flow* and *Model Optimization tutorial*. You may choose to use the `load_model_script()` method to use advanced configuration before calling optimize. At the end of the optimization stage, the state of the runner is changed from **Hailo Model** to **Quantized Model** and new functionality is available:

   A. Running inference on `SDK_QUANTIZED` context (quantized model emulation). For further details refer to: *Model Optimization Tutorial*. This step allows you to measure the degradation due to quantization of your model without executing on device. We recommend evaluating the quantized model in emulation before proceeding to compilation.

   B. Run the `analyze_noise()` method to execute the layer noise analysis tool and analyze the model's accuracy. This tool is useful to debug quantization issues in case of large degradation in your quantized model. For further details see the *Layer Noise Analysis Tutorial*.

   An alternative option is to use the `optimize_full_precision()` method before calling `optimize()` to run only part of the optimization process. In which case, the runner state will be **FP optimized model** and it will include model modifications, such as normalization or resize, but without the quantization process. Runner in this state can run inference with `SDK_FP_OPTIMIZED` context, see example in: *Model Optimization Tutorial*.

   Note: The same functionality can be obtained using the command line interface. For example, `hailo optimize --help`

3. **Compilation:** this step takes a runner in state **Quantized Model** and compiles it to a Hailo executable binary file (HEF). At the end of this stage the state of the runner is changed from **Quantized Model** to **Compiled Model**, which allows you to export a binary HEF file to run on the Hailo hardware.

   A. Save the HEF file to be used with the HailoRT. For further details refer to the *Compilation Tutorial*.

   B. Running inference on hardware. For further details refer to: *Inference Tutorial*.

   Note: The same functionality can be obtained using the command line interface. For example, `hailo compiler --help`

The following block diagram illustrates how the runner states and the API switch between each other.

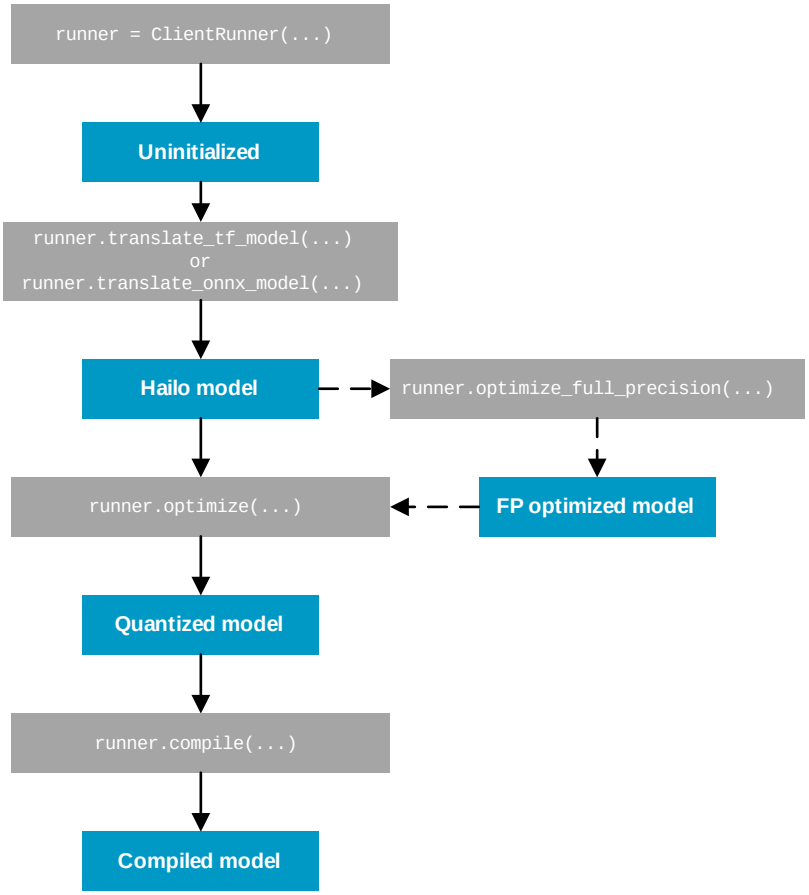


Figure 5. Description of the `ClientRunner` states and its API

5.1. Translating Tensorflow and ONNX Models

5.1.1. Using the Tensorflow Parser

The Dataflow Compiler Tensorflow parser supports the following frameworks:

- Tensorflow v1.15.4, including Keras v2.2.4-tf.
- Tensorflow v2.9.2, including Keras v2.9.0.
- Tensorflow Lite v2.4.0.

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

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

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

See also:

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

The supported input formats are:

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

Supported Tensorflow APIs

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

Table 1. Supported Tensorflow APIs (layers)

API name	Comments
`tf.nn.conv2d`	
`tf.concat`	
`tf.matmul`	
`tf.avg_pool`	
`tf.nn.maxpool2d`	
`tf.nn.depthwise_conv2d`	
`tf.nn.depthwise_conv2d_native`	
`tf.nn.conv2d_transpose`	• Only `SAME_TENSORFLOW` padding
`tf.reduce_max`	• Only on the features axis and with `keepdims=True`
`tf.reduce_mean`	

Continued on next page

Table 1 – continued from previous page

API name	Comments
`tf.reduce_sum`	• Only with `keepdims=True`
`tf.contrib.layers.batch_norm`	
`tf.image.resize_images`	• See limitations on Supported layers / Resize
`tf.image.resize_bilinear`	• See limitations on Supported layers / Resize
`tf.image.resize_nearest_neighbor`	• See limitations on Supported layers / Resize
`tf.image.crop_to_bounding_box`	• Only static cropping, i.e. the coordinates cannot be data dependent
`tf.image.resize_with_crop_or_pad`	• Only static cropping without padding, i.e. the coordinates cannot be data dependent
`tf.nn.bias_add`	
`tf.add`	• Only one of the following: – Bias add – Elementwise addition layer – Const scalar addition – As a part of input tensors normalization
`tf.multiply`	• Only one of the following: – Elementwise multiplication layer – Const scalar multiplication – As a part of input tensors normalization
`tf.subtract`	• Only one of the following: – Elementwise subtraction layer – Const scalar subtraction – As a part of input tensors normalization
`tf.divide`	• Only one of the following: – Elementwise division layer – Const scalar division – As a part of input tensors normalization
`tf.negative`	
`tf.pad`	
`tf.reshape`	• Only in specific cases, for example: – Features to Columns Reshape layer – Between Conv and Dense layers (in both directions) – As a part of layers such as Feature Shuffle and Depth to Space
`tf.nn.dropout`	
`tf.depth_to_space`	
`tf.nn.softmax`	
`tf.argmax`	
`tf.split`	• Only in the features dimension
`tf.slice`	• Only static cropping, i.e. the coordinates cannot be data dependent
Slicing (`tf.Tensor.__getitem__`)	• Only sequential slices (without skipping) • Only static cropping, i.e. the coordinates cannot be data dependent
`tf.nn.space_to_depth`	
`tf.math.square`	
`tf.math.pow`	• Only in specific case, pow(2) which is square

Continued on next page

Table 1 – continued from previous page

API name	Comments
`tf.norm`	• Reduce L2, keepdims=True, axis = 1,2
`tf.math.l2_normalize`	• L2 Normalization
`tf.math.minimum`	
`tf.math.maximum`	

Table 2. Supported Tensorflow APIs (activations)

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

Table 3. Supported Tensorflow APIs (others)

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

Slim APIs

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

Table 4. Supported Slim APIs

API name	Comments
`slim.conv2d`	

Continued on next page

Table 4 – continued from previous page

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

Keras APIs

Table 5. Supported Keras APIs

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

Continued on next page

Table 5 – continued from previous page

API name	Comments
`activations.tanh`	

Group Conv Parsing

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

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

Feature Shuffle Parsing

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

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

Code example:

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

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

Squeeze and Excitation Block Parsing

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

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


Threshold Activation Parsing

The threshold activation can be parsed from:

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

where `threshold` is the threshold to apply.

Delta Activation Parsing

The delta activation can be parsed from:

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

where `val` can be any constant number.

5.1.2. Using the Tensorflow Lite Parser

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

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

See also:

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

Supported Tensorflow Lite Operations

Table 6. Supported TFLite operations (layers)

Operator name	Comments
ADD	
AVERAGE_POOL_2D	
CONCATENATION	
CONV_2D	
DEPTHWISE_CONV_2D	
DEPTH_TO_SPACE	
DEQUANTIZE	• Only for parsing weight variables that are cast from float32 to float16
FULLY_CONNECTED	
L2_NORMALIZATION	
MAX_POOL_2D	
MUL	
RESHAPE	
RESIZE_BILINEAR	• See limitations on Supported layers / Resize
SOFTMAX	

Continued on next page

Table 6 – continued from previous page

Operator name	Comments
SPACE_TO_DEPTH	
PAD	
GATHER	
TRANSPOSE	• Only in specific cases, for example: – Depth to Space layer – Feature Shuffle layer – Features to Columns Reshape layer – Dense like to Conv like Reshape layer
MEAN	
SUB	
DIV	
SQUEEZE	
STRIDED_SLICE	
SPLIT	
CAST	
MAXIMUM	
ARG_MAX	
MINIMUM	
NEG	
PADV2	
SLICE	
TRANSPOSE_CONV	
EXPAND_DIMS	
SUM	
SHAPE	
POW	
REDUCE_MAX	
PACK	
UNPACK	
REDUCE_MIN	
SQUARE	
RESIZE_NEAREST_NEIGHBOR	• See limitations on Supported layers / Resize

Table 7. Supported TFLite operations (activations)

Operator name	Comments
LOGISTIC	
RELU	
RELU6	
TANH	
EXP	
PRELU	

Continued on next page

Table 7 – continued from previous page

Operator name	Comments
LESS	
GREATER	• Only as a part of a Threshold activation parsing
EQUAL	
LOG	
SQRT	
LEAKY_RELU	
ELU	
HARD_SWISH	
ABS	• Only as a part of the Delta activation parsing
ADD_N	
Sign	• Only as a part of the Delta activation parsing
CUSTOM	Only when the operator represents the biased delta activation

5.1.3. Using the ONNX Parser

ONNX models are translated by calling the `translate_onnx_model()` method of the `ClientRunner` object. The supported ONNX opset versions are 8 and 11-15.

Supported ONNX Operations

Table 8. Supported ONNX operations (layers)

Operator name	Comments
Add	• Only one of the following: – Bias add – Elementwise add – As a part of input tensors normalization – Const scalar addition
ArgMax	
AveragePool	
BatchNormalization	
Concat	
Conv	• Depthwise convolution is also implemented by this ONNX operation – 3D convolution (preview)
ConvTranspose	
DepthToSpace	• Supported modes: * DCR: the default mode, equivalent to Tensorflow's DepthToSpace operator * CRD: reflects PyTorch's PixelShuffle operator
Div	• Only one of the following: – Input tensors normalization – Const scalar division – Elementwise division

Continued on next page

Table 8 – continued from previous page

Operator name	Comments
Dropout	
Einsum	• Only specific formula: nkctv,kvw->nctw
Equal	
Flatten	• Only in specific cases such as between Conv and Dense layers
Gemm	
GlobalAveragePool	
GlobalMaxPool	
InstanceNormalization	
Identity	• Only when representing a constant value
MatMul	
MaxPool	
Mean	
Mul	• Only one of the following: – Elementwise Multiplication layer – Const scalar multiplication – As a part of input tensors normalization – As a prt of several activation functions, see below
Neg	
Pad	
ReduceMax	• Only on the features axis and with `keepdims=True`
ReduceMean	
ReduceSum	• Only with `keepdims=True`, or as a part of a Softmax layer
ReduceSumSquare	
ReduceL2	• Only in specific cases, after rank4 tensors such as Conv (as oppose to rank2 such as Dense)
Reshape	• Only in specific cases, for example: – Depth to Space layer – Feature Shuffle layer – Features to Columns Reshape layer – Between Conv and Dense layers (in both directions) – Spatial flatten format conversion: [N, H, W, C] -> [N, 1, H*W, C] (preview) – Spatial unflatten: [N, 1, H*W, C] -> [N, H, W, C] (preview)
Resize	• See limitations on Supported layers / Resize
Slice	
Softmax	
Split	• Only in the features dimension
Squeeze	• Only in specific cases such as between Conv and Dense layers
Sub	• Only one of the following: – Input tensors normalization – Const scalar substraction – Elementwise substraction

Continued on next page

Table 8 – continued from previous page

Operator name	Comments
Transpose	• Only in specific cases, for example: – Depth to Space layer – Feature Shuffle layer – Features to Columns Reshape layer – Dense like to Conv like Reshape layer
Unsqueeze	• Only in specific cases such as between Dense and Conv layers
Upsample	• Only Nearest Neightbor resizing
Expand	• Only as broadcast before elementwise operations

Table 9. Supported ONNX operations (activations)

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

Exporting Models from PyTorch to ONNX

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

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

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

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

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

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

Supported PyTorch APIs

We have tested support on PyTorch version 1.12.0. *Exporting* Pytorch models to the ONNX format is done using the `torch.onnx.export` function.

Table 10. Supported PyTorch APIs (layers)

API name	Comments
`torch.nn.AvgPool2d`	
`torch.nn.BatchNorm1d`	
`torch.nn.BatchNorm2d`	
`torch.nn.Conv1d`	
`torch.nn.Conv2d`	

Continued on next page

Table 10 – continued from previous page

API name	Comments
torch.nn.ConvTranspose2d	
torch.nn.Dropout2d	Does nothing on inference
torch.nn.Flatten	
torch.nn.functional.interpolate	• See limitations on Supported layers / Resize
torch.nn.functional.pad	
torch.nn.InstanceNorm2d	
torch.nn.Linear	
torch.nn.MaxPool1d	
torch.nn.MaxPool2d	
torch.nn.MultiheadAttention	Preview
torch.nn.Parameter	
torch.nn.PixelShuffle	
torch.nn.Softmax	
torch.nn.Softmax2d	
torch.nn.Upsample	• See limitations on Supported layers / Resize
torch.nn.UpsamplingBilinear2d	• See limitations on Supported layers / Resize
torch.nn.UpsamplingNearest2d	• See limitations on Supported layers / Resize
torch.argmax	
torch.cat	
torch.min	
torch.max	
torch.mul	See Supported ONNX operations: Mul
torch.div	See Supported ONNX operations: Div
torch.reshape	See Supported ONNX operations: Reshape
torch.split	See Supported ONNX operations: Split
torch.sum	See Supported ONNX operations: ReduceSum
torch.square	
torch.pow	Only in specific case, pow(2) which is square
torch.transpose	See supported ONNX operations: Transpose
torch.einsum	See supported ONNX operations: Einsum
torch.nn.MultiheadAttention	

Table 11. Supported PyTorch APIs (activations)

API name	Comments
torch.abs	See Supported ONNX operations: Abs
torch.exp	
torch.gt	See Supported ONNX operations: Greater
torch.log	
torch.sign	See Supported ONNX operations: Sign
torch.sqrt	

Continued on next page

Table 11 – continued from previous page

API name	Comments
`torch.nn.ELU`	
`torch.nn.GELU`	
`torch.nn.Hardsigmoid`	
`torch.nn.Hardswish`	
`torch.nn.LeakyReLU`	
`torch.nn.Mish`	
`torch.nn.ReLU`	
`torch.nn.PReLU`	
`torch.nn.ReLU6`	
`torch.nn.Sigmoid`	
`torch.nn.SiLU`	
`torch.nn.Softplus`	
`torch.nn.Tanh`	
`torch.clip`	

5.1.4. Layer Ordering Limitations

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

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

5.1.5. Supported Padding Schemes

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

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

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

5.1.6. NMS Post Processing

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

Hailo supports the following NMS post processing algorithms:

On neural core:

1. SSD/EfficientDet: bbox decoding, score threshold filtering, IoU filtering

2. CenterNet: bbox decoding, score threshold filtering
3. YOLOv5: bbox decoding, score_threshold filtering (supporting also YOLOv7)

On CPU:

1. YOLOv5: bbox decoding, score_threshold filtering, IoU filtering (supporting also YOLOv7)
2. SSD/EfficientDet: bbox decoding, score_threshold filtering, IoU filtering

---

Note: NMS on neural code is only supported in models that are compiled to single context. If the model is compiled with multi-context, undefined runtime behavior might occur. On this case, you are encouraged to either try single context compilation using a model script, or perform the NMS on the host platform.

For implementation on hailo devices:

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

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

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

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

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

3. You can experiment with the output format using the SDK_FP_OPTIMIZED or the SDK_QUANTIZED emulators, before compiling the model. For more information, refer to the *Model optimization workflow* section.

SSD

SSD (which is also used by EfficientDet models) postprocessing consists of bbox decoding and NMS.

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

We assume that the default configurations file is used.

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

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

(continues on next page)

(continued from previous page)

```
  }
}
```

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

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

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

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

An example for the corresponding SSD NMS JSON is found at: `site-packages/hailo_sdk_client/tools/core_postprocess/nms_ssd_config_example_json_notes.txt`, relatively to the virtual environment where the Dataflow Compiler is installed. This example file is not a valid JSON file since it has in-line comments, but a ready-to-use file is on the same folder.

CenterNet

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

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

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

```
end_node_names =
[
  "threshold_confidence/threshold_activation/threshold_confidence/re_lu/Relu",
  "CenterNet0_conv3/BiasAdd",
  "CenterNet0_conv5/BiasAdd"
]
```

An example for the corresponding CenterNet JSON is found at: `site-packages/hailo_sdk_client/tools/core_postprocess/centerNet_example_json_notes.txt`, relatively to the virtual environment where the Dataflow Compiler is installed. This example file is not a valid JSON file since it has in-line comments, but a ready-to-use file is on the same folder.

YOLOv5

YOLOv5 postprocessing (true also for YOLOv7) consists of bbox decoding and NMS. The NMS consists of two parts:

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

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

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

The anchors are taken from this file.

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

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

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

An example for the corresponding YOLOv5 JSON is found at: `site-packages/hailo_sdk_client/tools/core_postprocess/nms_yolov5_example_json_notes.txt`, relatively to the virtual environment where the Dataflow Compiler is installed. This example file is not a valid JSON file since it has in-line comments, but a ready-to-use file is on the same folder.

5.1.7. Reasons and Solutions for differences in the parsed model

On some cases, the translated model might have some differences compared to the original model:

1. BatchNorm layer in training mode. The difference in this case is because the BN params are static in the hailo model (and folded on relevant layers kernel/bias), and in the original model framework, training mode means that the layer would first update moving mean/var and then normalize its output in place. To avoid this case:
   - PyTorch: export your model to ONNX in preserve or eval mode. For more information, check *parsing tutorial*.
   - Keras: set the model's learning phase to 0 (test).
2. Otherwise, please contact our support.

5.2. Profiler and other command line tools

5.2.1. Using Hailo command line tools

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

To list the available tools, run:

```
hailo --help
```

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


```
hailo profiler --help
```

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

Model conversion flow

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

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

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

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

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

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

Analysis and Visualization

The list below describes the Hailo command line interface functions for visualization and analysis:

- The `hailo analyze-noise` command is used to analyze per-layer quantization noise. Consult *Model optimization flow* for further details.
- The `hailo params-csv` command is used to to generate a CSV report with weights statistics, which is useful for analyzing the quantization.
- The `hailo tb` command is used to convert HAR or CKPT files to Tensorboard.
- The `hailo visualizer` command is used to visualize HAR files.
- The `hailo har` command is used to extract information from HAR files.

Tutorials

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

5.2.2. Running the Profiler

The Profiler command line tool analyzes the expected performance of models on the hardware.

To run the Profiler, use the following command:

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

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

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

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

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

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

Note: To analyze the model ACCURACY, the profiler also contains the layer noise analysis tool. The **Accuracy profiler** is created for that purpose: By default, when running after quantization, only partial data is displayed. The user can add the full analysis information by running `hailo analyze-noise <har-path> --data-path <data-path>` command, or by adding `model_optimization_config(checker_cfg, policy=enabled, analyze_mode=advanced)` to the model script before the optimization stage. See example at *Layer Noise Analysis Tutorial*.

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

Note: For single-context networks, `--stream-fps` argument can be used to normalize the power and bandwidth values according to the FPS of the input stream.

Note: You can create the new profiler design HTML report, by appending the flag `--use-new-report` (as of July 2023).

5.2.3. Understanding the Profiler report

The Hailo profiler report consists of the following sections:

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

- **Accuracy section** – Per-layer statistics, both native and quantized, used for gaining insights about degradation factors.

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

Model Details

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

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

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

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

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

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

Profiler Input Settings

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

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

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

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

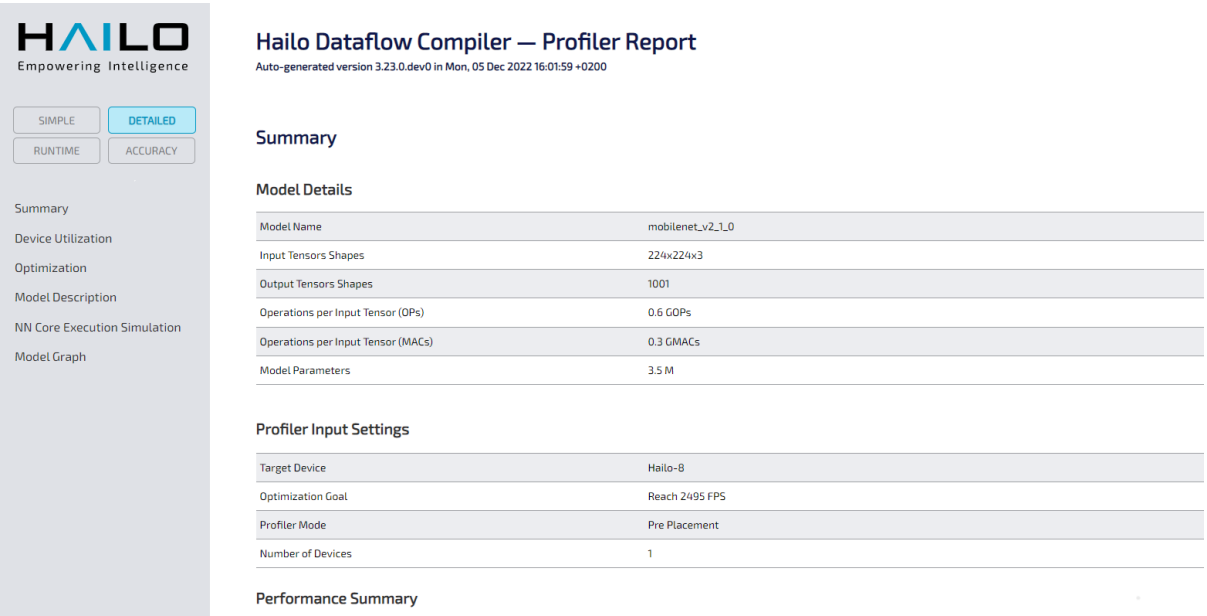


Figure 6. Profiler report screenshot

Performance Summary

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

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

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

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

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

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

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

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

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

Device Utilization

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

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

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

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

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

Optimization

Model Optimization

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

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

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

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

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

Model Modifications

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

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

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

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

Model Description

For each layer, the following fields are defined:

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

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

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

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

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

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

Dilation The kernel dilation (assuming symmetric dilation).

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

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

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

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

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

NN Core Execution Simulation

Displays a simulation of the layers as if they were running on the device. Displays a simulation of three input tensors. Each layer occupies a row, and the rectangles represents the layer working times. The grey areas are times where the layer is idle. Remembering that the Hailo device works row-by-row, you can see the input rows as they are being processed in the layers; When a layer is done, the next one begins processing that row, and so on.

Model Graph

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

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

1. **Layer parameters** – Parameters related to the model's layer itself (regardless of Hailo implementation).
2. **Hailo performance parameters** – The selected layer's performance parameters on the Hailo HW.

3. **Advanced** – Deeper dive into Hailo's HW component allocation and utilization parameters.
4. **Optimization** – Information related to optimization of this layer's inputs, weights and activations

The description of all parameters is given below.

Layer parameters

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

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

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

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

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

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

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

Number of Parameters How many parameters this layer has

Hailo performance parameters

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

FPS The frames per second processed by the layer.

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

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

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

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

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

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

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

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

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

Advanced

HailoMAC Computation Utilization [%] The relative amount of MAC units utilized by the layer.

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

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

Activation Pooling Units The number of activation and pooling units (APUs) required by the layer.

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

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

Buffer Size The buffer size required by the layer.

Optimization

Ranges [Input, Kernel, Output] Ranges of values

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

Runtime section

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

Each context consists of four phases:

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

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

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

Accuracy section

In this section, we describe all the information that exists in the accuracy section, however, not all the data would be available after optimization. To obtain the full accuracy report you should run the `analyze-noise` command.

Model Optimization Displays information about the parameters that affect the Optimization process. The same information as in the Optimization section in the SIMPLE section.

Model Description A plot of signal-to-noise ratio between the full precision and quantized model. The SNR value is measured at the output layer of the model and in each measurement, we only quantize a single layer. This graph shows the sensitivity of each layer to quantization measured in dB.

Layers Information

Displays statistics about each layer, collected by passing the calibration set through the model.

Bits The amount of bits used to represent the [Input, Weights, Bias, Output].

SNR per layer Signal-to-noise ratio between the full precision and quantized model, measured at the layer's output.

Kernel Ranges Minimum and maximum values of the weights of the layer.

Input Ranges Minimum and maximum values at the layer's inputs (before quantization).

Output Ranges Minimum and maximum values at the layer's outputs (before quantization).

Activation Type Specify the activation function Type.

Batch Normalization Specify whether batch normalization was used during training on this layer.

Kernel Shape The shape of the kernel of the specific layer

Input Shapes Shapes of the input to the layer

Output Shapes Shapes of the output from the layer

Weight Histogram This histogram shows the full precision weights distribution. Outliers in the distribution might cause degradation.

Activations Histogram This histogram shows the full precision activations distribution. Outliers in the distribution might cause degradation.

Scatter Plot This graph shows the difference for representative activation values between the full precision and quantized model as measured at the output of the layer. Better quantization means the trend should be closer to a slope of 1 (that represents zero quantization noise). Different channels are represented by different colors.

5.3. Model optimization

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

5.3.1. Model Optimization Workflow

The model optimization has two main steps: Full precision optimization and Quantization optimization.

Full precision optimization includes any changes to the model in the floating-point precision domain, for example, Equalization [Meller2019], TSE [Vosco2021] and pruning. It also includes any model modifications from the model script. Quantization includes compressing the model from floating point to integer representation of the weights (4/8/16-bits) and activations (8/16-bits) and algorithms to improve the model's accuracy, such as IBC [Finkelstein2019], AdaRound [Nagel2020] and QFT [McKinstry2019]. Both steps may degrade the model accuracy, therefore, evaluation is needed to verify the model accuracy.

To perform these steps, one can use the simple optimization flow. Use the `hailo optimize` CLI, or the `load_model_script()` method followed by `optimize()`. Afterwards you can continue to the compilation stage. The simple optimization flow *is depicted in this diagram*.

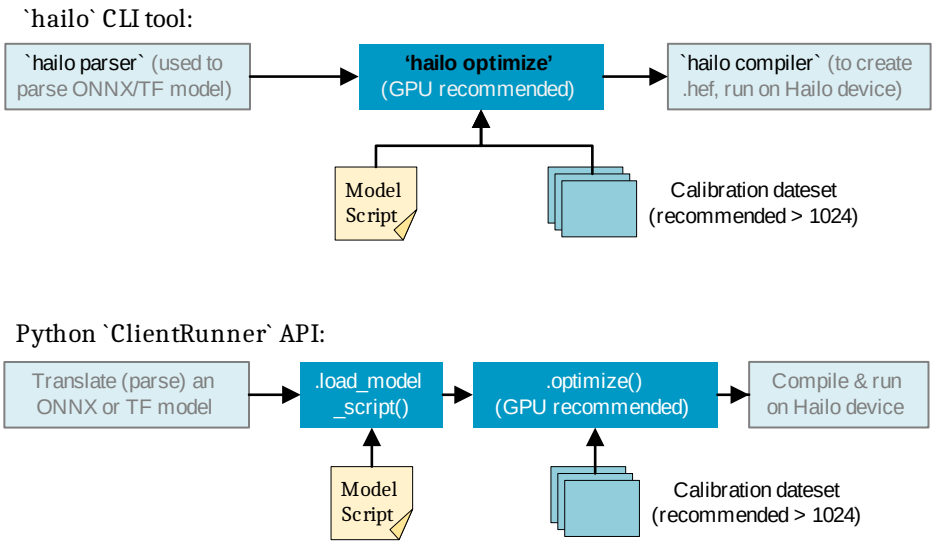


Figure 7. Block diagram of the simple model optimization flow

You can also follow the advanced Python workflow for tracking the accuracy of your model throughout the stages of the optimization. This advanced workflow, as well as the simple flows, are presented on the *Model Optimization Tutorial*.

The advanced workflow consists of number of stages, *which are depicted in the flow chart on the next page*:

1. A preliminary step would be to test the *Native* model before any changes, right after parsing. This stage is important for making sure the parsing was successful, and we built the preprocessing (before the start nodes) and post processing (after the end nodes) correctly. As mentioned, the `SDK_NATIVE` emulator is used for this purpose:

```
import tensorflow as tf
from hailo_sdk_client import ClientRunner, InferenceContext

runner = ClientRunner(har_path=model_path)
with runner.infer_context(InferenceContext.SDK_NATIVE) as ctx:
  output = runner.infer(ctx, input_data)
```

   You can also compare the parsed model to the original model using the command: *hailo parser* with the flag *–compare*. For more information refer to *reasons* section.

2. Load the model script, and use the `optimize_full_precision()` method to apply the model script and the full precision optimizations.

3. Perform full precision validation, when the model is in its final state before the optimization process. This stage is important because it allows you to emulate the input and output formats, taking into account the model modifications (normalization, resize, color conversions, etc.). Getting good accuracy means that we built the pre/post processing functions correctly, and that our infrastructure is ready for testing the quantized model. The `SDK_FP_OPTIMIZED` emulator is used for this purpose:

```
import tensorflow as tf
from hailo_sdk_client import ClientRunner, InferenceContext

runner = ClientRunner(har_path=model_path)
with runner.infer_context(InferenceContext.SDK_FP_OPTIMIZED) as ctx:
  output = runner.infer(ctx, input_data_modified)
```

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

5. Finally, we need to verify the accuracy of the optimized model to validate the process was successful. In case of large degradation (that doesn't meet the accuracy requirement), we can re-try optimization with increased optimization level. **Optimization and Compression levels** allows you to control the model optimization effort and the model memory footprint. For quick iterations we recommend starting with the default setting of the model optimizer (optimization_level=2, compression_level=1). However, when moving to production, we recommended to work at the highest optimization level (optimization_level=4) to achieve optimal accuracy. With regards to compression, users should increase it when the overall throughput/latency of the application is not satisfying. Note that increasing compression would have negligible effect on power-consumption so the motivation to work with higher compression level is mainly due to FPS considerations. To verify the accuracy of the quantized model, we recommend using the `SDK_QUANTIZED` emulator:

```
import tensorflow as tf
from hailo_sdk_client import ClientRunner, InferenceContext

runner = ClientRunner(har_path=model_path)
with runner.infer_context(InferenceContext.SDK_QUANTIZED) as ctx:
  output = runner.infer(ctx, input_data_modified)
```

The advanced optimization flow *is depicted in this diagram*.

Note: Getting familiar with the *runner states diagram* is important for understanding the following diagram.

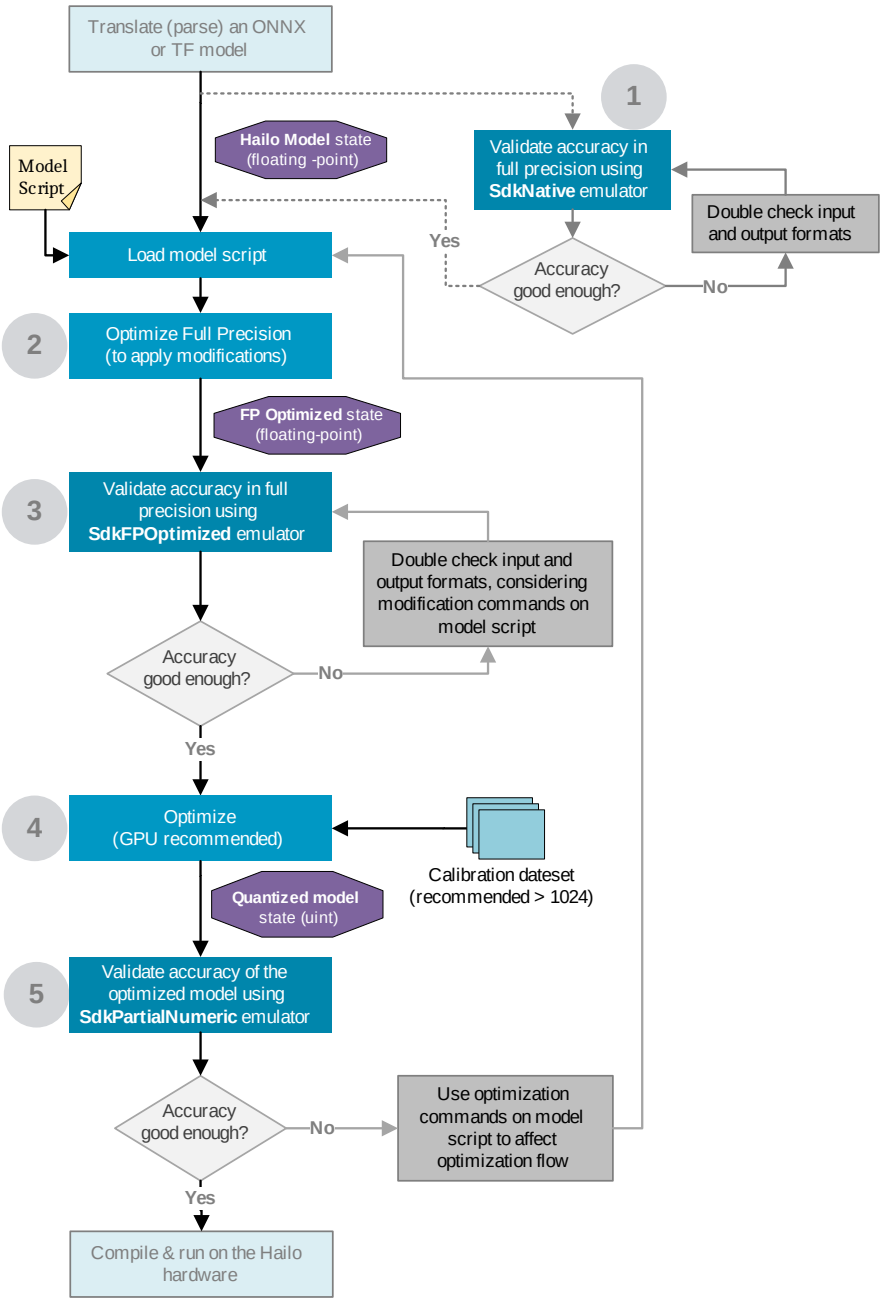


Figure 8. Block diagram of the advanced model optimization flow using Python APIs

Note: If you encounter problems with VRAM allocation during stages other than Adaround, you can attempt to resolve the issue by disabling the memory growth flag. To do this, set the following environment variable:

HAILO_SET_MEMORY_GROWTH=false

By doing so, the default memory allocation method for tensorflow GPU will be modified, and the entire VRAM will be allocated and managed internally.

Additionally, if tensorflow is imported, please make sure the SDK is imported before tensorflow is used.

Model Optimization Flavors

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

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

To control the optimization grade, we recommend setting the optimization_level argument through the *model_optimization_flavor* command, which gets values of 0-4 and control which quantization algorithms will be enabled. Using higher optimization level means the model optimization tool will use more advanced algorithms which expected to get better accuracy but will take longer to run. Note that optimization levels 2, 4 require at least 1024 images to run and optimization level 3 requires 256. The default setting is optimization_level=2 unless GPU is not available, or the dataset is not large enough (less than 1024). For reference, those are the expected running times for optimizing ResNet-v1-50 with compression_level=4 using Nvidia A4000 GPU:

- optimization_level=0: 59s
- optimization_level=1: 206s
- optimization_level=2: 256s
- optimization_level=3: 2828s
- optimization_level=4: 11002s

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

Note: The algorithms that compose each optimization level are expected to change in future versions. To see the current algorithms in use refer to *model_optimization_flavor* command description

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

Table 12. An example of the degradations for the RegNetX-800MF model over various flavor settings. Reported degradations are Top-1 scores over the ImageNet-1K dataset (validation set of 50k images). Note that the RegNetX-800MF model is relatively small (defined as having less than 20M parameters), hence there is only one valid compression level (compression_level=0).

	Optimization Level = 0	Optimization Level = 1	Optimization Level = 2	Optimization Level = 3	Optimization Level = 4
Compression Level = 0	0.41	0.16	0.29	-	0.19

Table 13. An example of the degradations for the YOLOv5m model over various flavor settings. Reported degradations are mAP scores over a validation set of 5k samples from the COCO2017 dataset.

	Optimization Level = 0	Optimization Level = 1	Optimization Level = 2	Optimization Level = 3	Optimization Level = 4
Compression Level - 0	4.12	3.35	1.61	-	0.19
Compression Level = 1	4.12	3.26	2.43	1.91	1.25
Compression Level = 4	8.61	7.67	4.78	2.50	1.58

Table 14. An example of the degradations for the DeepLab-v3-MobileNet-v2 model over various flavor settings. Reported degradations are mIoU scores over the PASCAL-VOC dataset. Note that the DeepLab-v3-MobileNet-v2 model is relatively small (defined as having less than 20M parameters), hence there is only one valid compression level (compression_level=0).

	Optimization Level = 0	Optimization Level = 1	Optimization Level = 2	Optimization Level = 3
Compression Level = 0	0.72	0.61	1.14	-

Debugging Accuracy

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

1. Make sure you are using at least 1024 images in your calibration dataset and machine with a GPU.
2. Validate the accuracy of your model in *~hailo_sdk_common.targets.infer_wrapper.InferenceContext.SDK_FP_OPTIMIZED* emulator to ensure pre and post processing are used correctly. Common pitfalls includes miss handling of preprocessing (for example, data normalization) or usage of the wrong data type for calibration.
3. Usage of BatchNormalization is crusuial to obtain good quantization accuracy because it reducess the activation ranges throughout the network, and therefore it is highly recommended to use it in your training.
4. Run the layer noise analysis tool to identify the source of degradation. For example, using the CLI command:

   hailo analyze-noise har_path –data-path data_path
5. If you have used ***compression_level***, lower its value (the default is 0). For example, use the following command in the model script:

   model_optimization_flavor(compression_level=1)
6. Configure higher ***optimization_level*** in the model script, that activates more optimization algorithms and experiment with different optimization levels. For example:

   model_optimization_flavor(optimization_level=4)
7. Configure 16-bit output. Note that using 16bit output affects the output BW from the Hailo device. For example, using the following model script command:

   quantization_param(output_layer1, precision_mode=a16_w16)
8. Configure 16-bit on specific layers that are sensitive for quantization. Note that using 16bit affects the throughput obtained from the Hailo device. For example, using the following model script command:

   quantization_param(conv1, precision_mode=a16_w16)
9. Try to run with activation clipping using the following model script commands:

   model_optimization_config(calibration, calibset_size=512), and *pre_quantization_optimization(activation_clipping, layers={*}, mode=percentile, clipping_values=[0.01, 99.99])*

10. Use more data and longer optimization process in Finetune, for example:

    post_quantization_optimization(finetune, policy=enabled, learning_rate=0.0001, epochs=8, dataset_size=4000)

11. Use different loss type in Finetune, for example:

    post_quantization_optimization(finetune, policy=enabled, learning_rate=0.0001, epochs=8, dataset_size=4000, loss_types=[l2, l2, l2, l2])

12. Use quantization aware training (QAT). For more information see *QAT Tutorial*.

See also:

The *Model Optimization Tutorial* which explains how to use the optimization API and the optimization/compression levels and the *Layer Noise Analysis Tutorial* which explains how to use the analysis tool.

5.3.2. Model Scripts

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

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

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

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

1. *model_modification_commands*

In addition, the model script supports 5 quantization commands:

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

model_modification_commands

The model script supports 8 model modification commands:

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

input_conversion

Adds on-chip conversion of the input tensor.

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

- yuv_to_rgb - which is implemented by the following kernel: [[1.164, 1.164, 1.164], [0, -0.392, 2.017], [1.596, -0.813, 0]] and bias [-222.912, 135.616, -276.8] terms. Corresponds to cv::COLOR_YUV2RGB in OpenCV terminology.
- yuv_to_bgr - which is implemented by the following kernel: [[1.164, 1.164, 1.164], [2.017, -0.392, 0], [0, -0.813, 1.596]] and bias [-276.8, 135.616, -222.912] terms. Corresponds to cv::COLOR_YUV2BGR in OpenCV terminology.
- bgr_to_rgb - which transposes between the R and B channels using an inverse identity matrix as kernel, no bias. Corresponds to cv2.cvtColor(src, code) where src is a BGR image, and code is cv2.COLOR_BGR2RGB.
- rgb_to_bgr - as the above, transposes between the R and B channels using an inverse identity matrix as kernel, no bias. Corresponds to cv2.cvtColor(src, code) where src is a RGB image, and code is cv2.COLOR_RGB2BGR.

Note: In these commands the input_layer argument is optional. If a layer name is not specified, the conversion will be added after all input layers.

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

Or a **format conversion**:

- Any of the formats described on *Format conversion*, for example yuy2_to_hailo_yuv.
- When converting yuy2_to_yuv, tf_rgbx_to_hailo_rgb or nv12_to_hailo_yuv, by default the conversion will only be part of the compiled model but it won't be part of the optimization process. To include this conversion also in the optimization process, set *emulator_support=True* inside the command. When setting it to True, the calibration set should be given in the source format.
- In these commands, input_layer argument must be provided.

```
yuv_layer = input_conversion(input_layer2, yuy2_to_hailo_yuv) # conversion won't be
↪part of the optimization
yuv_layer = input_conversion(input_layer2, yuy2_to_hailo_yuv, emulator_
↪support=True) # conversion will be part of the optimization
```

Or a **hybrid conversion** :

- yuy2_to_rgb - which is implemented by adding format conversion yuy2_to_yuv and color conversion yuv_to_rgb.
- nv12_to_rgb - which is implemented by adding format conversion nv12_to_yuv and color conversion yuv_to_rgb.
- nv21_to_rgb - which is implemented by adding format conversion nv21_to_yuv and color conversion yuv_to_rgb.
- i420_to_rgb - which is implemented by adding format conversion i420_to_yuv and color conversion yuv_to_rgb.
- When converting yuy2_to_rgb and nv12_to_rgb, by default the conversion between YUY2 to YUV and NV12 to YUV will only be part of the compiled model but it won't be part of the optimization process. To include this conversion also in the optimization process, set emulator_support=True inside the command. When setting it to True, the calibration set should be given in the source format.

```
yuy2_to_yuv_layer, yuv_to_rgb_layer = input_conversion(input_layer1, yuy2_to_rgb)
↪# conversion won't be part of the optimization
yuy2_to_yuv_layer, yuv_to_rgb_layer = input_conversion(input_layer1, yuy2_to_rgb,
↪emulator_support=True) # conversion will be part of the optimization
```


resize_input

Performs on-chip resize to the input tensor(s). The default resize method used is bilinear interpolation with align_corners=True and half_pixels=False, but it can be changed using the *pixels_mode* flag:

- pixels_mode=align_corners
- pixels_mode=half_pixels
- pixels_mode=disabled

The input resize limitations are those of resize bilinear *as described here*. When the resize ratio is high, the compilation process will be more difficult, as more on-chip memories and sub-clusters are required.

```
resize_input1 = resize_input([256,256], input1) # resize a single input
resize_input1 = resize_input([256,256], input1, pixels_mode=half_pixels)
resize_input1, resize_input2, ... = resize_input([256,256]) # resize all inputs;
↪return value should match the number of inputs of the network
resize_input1, resize_input2, ... = resize_input([256,256], pixels_mode=disabled)
```

transpose

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

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

Note: Transposing the network is not supported when the Depth to Space or Space to Depth layers are used.

normalization

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

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

nms_postprocess

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

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

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

1. Specify only the architecture name.
    - If NMS structure was detected during parsing, an autogenerated config file with the values extracted from the original model will be used.

- Otherwise, a default config file will be used.
- Layers that come before the post-process are auto-detected.

For example: nms_postprocess(meta_arch=ssd)

2. Specify the architecture name and some of the config arguments.
   - If NMS structure was detected during parsing, an autogenerated config file with the values extracted from the original model will be used, edited by provided arguments.
   - Otherwise, a default config file will be used, edited by provided arguments.
   - Input layers to post-process will be auto-detected.
   - The config arguments that can be set via the command are: nms_scores_th, nms_iou_th, image_dims, classes.

   For example: nms_postprocess(meta_arch=yolov5, image_dims=[512, 512], classes=70)
3. Specify the config json path in addition to architecture name.
   - The file provided will be used.
   - Please note that when providing the config path, you shouldn't provide any of the config argument using the command, only inside the file.

   For example: nms_postprocess('config_file_path', meta_arch=centernet)

The default config files can be found at `site-packages/hailo_sdk_client/tools/core_postprocess/core_postprocess`, relatively to the virtual environment where the Dataflow Compiler is installed:

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

For available architectures see `NMSMetaArchitectures`.

When the meta-architecture is supported by the neural core, the NMS post-process runs on the neural core by default, otherwise on the host CPU. Use the engine argument in a post_process model script command to configure the inference device manually. There are 3 supported modes: `nn_core`, `cpu`, `auto`:

- `nn_core` which means the NMS post-process will run on the nn-core.

  For example:

  ```
  nms_postprocess(meta_arch=ssd, engine=nn_core)
  ```

  Since nn_core is the default, you can also use:

  ```
  nms_postprocess(meta_arch=ssd)
  ```

- `cpu` which means the NMS post-process will run on the CPU (currently only available for YOLOv5):

  For example:

  ```
  nms_postprocess(meta_arch=yolov5, engine=cpu, image_dims=[512, 512])
  ```

- `auto` currently works only with YOLOv5, performs bbox decoding and score_threshold filtering on the neural core and IoU filtering on CPU.

  For example:

  ```
  nms_postprocess('config_file_path', meta_arch=yolov5, engine=auto)
  ```


change_output_activation

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

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

logits_layer

Adds logits layer after an output layer. The supported logits layers are Softmax and Argmax.

Softmax layer can be added under the following conditions:

1. The output layer has rank 2.
2. Total number of softmax layers is less than three.

Argmax layer can be added under the following conditions:

1. The output layer has rank 4.
2. The operation is only on the channels dimension

```
logits_layer1 = logits_layer(output_layer, softmax, 1) # adding logits layer after
↪the output layer.
logits_layer1, logits_layer2, ..., = logits_layer(argmax, 3) # adding logits layer
↪after all the output layers.
```

set_seed

Sets the global random seed for python random, numpy and tensorflow libraries, and enables operator determism in tensorflow's backend. Setting the seed ensures reproducibility of quantization results.

Note: When running Finetune algorithm on GPU, tensorflow's back-propagation operators can't perform deterministic results.

Note: Using tensorflow's operator determinism comes at the expense of runtime efficiency, it's recommended to use this feature for debugging only. For more details please refer to tensorflow's docs.

```
set_seed(seed=5)
```

model_optimization_flavor

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

Default values:

- compression_level: 1
- optimization_level: 2 for GPU and 1024 images, 1 for GPU and less than 1024 images, and 0 for CPU only.
- batch_size: check default of each algorithm (usually 8 or 32)

Optimization levels: (might change every version)

- -100 nothing is applied - all default algorithms are switched off
- 0 - Equalization
- 1 - Equalization + Iterative bias correction
- 2 - Equalization + Finetune with 4 epochs & 1024 images
- 3 - Equalization + Adaround with 320 epochs & 256 images on all layers
- 4 - Equalization + Adaround with 320 epochs & 1024 images on all layers

Compression levels: (might change every version)

- 0 - nothing is applied
- 1 - auto 4bit is set to 0.2 if network is large enough (20% of the weights)
- 2 - auto 4bit is set to 0.4 if network is large enough (40% of the weights)
- 3 - auto 4bit is set to 0.6 if network is large enough (60% of the weights)
- 4 - auto 4bit is set to 0.8 if network is large enough (80% of the weights)
- 5 - auto 4bit is set to 1.0 if network is large enough (100% of the weights)

Example commands:

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

Parameters:

Parameter	Values	Default	Required	Description
optimization_level	int; -100<=x<=4	(Read command doc)	False	Optimization level, higher is better but longer, improves accuracy
batch_size	int; 1<=x	(Read command doc)	False	Batch size for the algorithms (adaround, finetune, calibration)
compression_level	int; 0<=x<=5	(Read command doc)	False	Compression level, higher is better but increases degradation, improves fps and latency

model_optimization_config

- *compression_params*
- *calibration*
- *checker_cfg*

compression_params

This command controls layers 4-bit and 16-bit quantization. In 4-bit mode, it reduces some layers' precision mode to a8_w4. The values (between 0 and 1 inclusive) represent how much of the total weights memory usage you want to optimize to 4bit. When the value is 1, all the weights will be set to 4bit, when 0, the weights won't be modified. The 16-bit mode is supported only when setting on the entire network (setting 16-bit value of 1) and without using 4-bit (setting 4-bit value to 0).

Example command:

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

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

Note: If you set 16-bit quantization, all layers activations and weights are quantized using 16 bits. In this case, explicit configuration of layer bias mode is not allowed.

Parameters:

Parameter	Values	Default	Required	Description
auto_4bit_weights_ratio	float; 0<=x<=1	0	False	Set a ratio of the model's weights to reduce to 4bit
auto_16bit_weights_ratio	float	0	False	Set a ratio of the model's weights to reduce to 16bit

calibration

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

Example command:

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

Parameters:

Parameter	Values	Default	Required	Description
batch_size	int; 0<x	8	False	Batch size used during the calibration inference
calibset_size	int; 0<x	64	False	Data items used during the calibration inference
force_activation_histogram	bool	False	False	Forces activation histogram on all layers

checker_cfg

Checker Config will generate information about the quantization process using the layer analysis tool.

Example commands:

```
# This will disable the algorithm
model_optimization_config(checker_cfg, policy=disabled)

# When finetune/adaround exist, This will generate information only about logits
model_optimization_config(checker_cfg, policy=enabled, checker_mode=simple)
```

Note: This operation does not modify the structure of the model's graph

Note: Changing the checker_mode from advanced to simple will only have an effect if finetune/adaround exist. When checker_mode is set to simple, the algorithm will generate less information regarding the quantization process, in exchange for quicker optimization time.

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	enabled	False	Enable or disable the checker algorithm during the quantization process.
dataset_size	int; 0<x	16	False	Number of images used for profiling.
batch_size	int; 0<x	None	False	Uses the calibration batch_size by default. Number of images used together in each inference step.
analyze_mode	{simple, advanced}	simple	False	The analysis mode that will be used during the algorithm execution (simple/advanced). simple only execute analysis on the fully quantize net, while advanced also execute layer by layer analysis. Default is simple.
checker_mode	{simple, advanced}	advanced	False	Checker mode described the way the algoritm would be integrated during the quantization process (simple/advanced). When checker mode is set to simple, the algorithm will generate less information regarding the quantization process, in exchange for quicker optimization time. Default is advanced.
batch_norm_checker	bool	True	False	Set whether the algorithm should display a batch normalization warning message when the gathered layer statistics differ from the expected distribution. Default is True.

quantization_param

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

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

For example

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

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

```
quantization_param(conv1, bias_mode=double_scale_initialization, precision_
↪mode=a8_w4)
```

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

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

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

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

The available parameters are:

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

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

Bias_Mode

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

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

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

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

Example command:

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

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

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

precision_mode

Precision mode sets the bits available for the layers' weights and activation representation. There are currently 3 available precision modes: `a8_w8`, `a8_w4`, `a16_w16`.

- `a8_w8` - which means 8-bit activations and 8-bit weights. (This is the default)
- `a8_w4` - which means 8-bit activations and 4-bit weights. Can be used to reduce memory consumption.
- `a16_w16` - set 16 bit activations and weights to improve accuracy results. It's supported only if applied to a supported layer (to achieve better accuracy at the cost of performance degradation). The option to run full model in 16-bit quantization is supported only if all network layers support it. Namely, for entire network quantization, all layers must have 16 bit support. Current supported 16-bit layers are:
  - Fully Connected (dense)
  - Depthwise Convolution (dw)
  - Elementwise Add (ew_add)
  - Convolution (conv)
  - Max Pooling
  - Average Pooling
  - Shortcut
  - Slice
  - External Padding
  - Reshape
  - Resize
  - Feature Split
  - Feature Shuffle
  - Concat
  - Reduce Max
  - Depth to Space
  - Space to Depth
  - Deconvolution
  - Reduce Sum
  - Normalization
  - Activations
  - Output Layer

Example command:

```
quantization_param(conv3, precision_mode=a8_w4) # A specific layer
model_optimization_config(compression_params, auto_16bit_weights_ratio=1) # Full
↪network, in case all layers are supported
```

Note: It is recommended to use *Finetune* when using 4-bit weights. For 16-bit quantizatization, it's recommended to use *model_optimization_config*. For 16-bit quantization, glob syntax is not supported.

null_channels_cutoff_factor

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

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

Example command:

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

max_elementwise_feed_repeat

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

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

Example command:

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

max_bias_feed_repeat

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

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

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

Example command:

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

quantization_groups

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

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

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

Example command:


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

pre_quantization_optimization

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

The algorithms are triggered in the following order:

- *dead_channels_removal*
- *zero_static_channels*
- *se_optimization*
- *equalization*
- *equalization per-layer*
- *weights_clipping*
- *activation_clipping*
- *ew_add_fusing*
- *layer_decomposition*
- *smart_softmax_stats*
- *input_features_defuse*

dead_channels_removal

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

Example commands:

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

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

Parameters:

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

zero_static_channels

Zero static channel will zero out the weights of channels that have zero variance to improve quantization

Example commands:

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

Note: This operation does not modify the structure of the model's graph

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	enabled	True	Enable or disable the zero static channels algorithm
eps	float; 0<=x	1e-07	False	Threshold value to zero channels for the zero static channels algorithm

se_optimization

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

Example commands:

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

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

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

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

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

Parameters:

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

equalization

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

Example command:

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

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

Parameters:

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

equalization per-layer

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

Example commands:

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

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

Note:

- Not all layers support equalization

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

Parameters:

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

weights_clipping

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

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

Example commands:

```
pre_quantization_optimization(weights_clipping, layers=[conv2], mode=manual,
↪clipping_values=[-0.1, 0.8])
pre_quantization_optimization(weights_clipping, layers=[conv3], mode=percentile,
↪clipping_values=[1.0, 99.0])
pre_quantization_optimization(weights_clipping, layers={conv*}, mode=mmse)
pre_quantization_optimization(weights_clipping, layers=[conv3, conv4], mode=mmse_
↪if4b)
pre_quantization_optimization(weights_clipping, layers={conv*}, mode=disabled)
```

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

Parameters:

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

activation_clipping

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

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

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

Example commands:

```
pre_quantization_optimization(activation_clipping, layers=[conv1], mode=manual,
↪clipping_values=[0.188, 1.3332])
pre_quantization_optimization(activation_clipping, layers=[conv1, conv2],
↪mode=percentile, clipping_values=[0.5, 99.5])
pre_quantization_optimization(activation_clipping, layers={conv*}, mode=disabled)
```

Parameters:

Parameter	Values	Default	Required	Description
mode	{disabled, manual, percentile, percentile_force}	disabled	True	Mode of operation, described above
clipping_values	[float, float]	None	False	Clip value, required when mode is percentile or manual
recollect_stats	bool	False	False	Indicates whether stats should be collected after clip

ew_add_fusing

When EW add fusing is enabled ew add layers will be fused into conv and add layers.

Example commands:

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

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

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	enabled	True	Enable or disable the ew add fusing optimization
infusible_ew_add_type	{conv, ew_add}	ew_add	False	Decide whether to create a conv or a standalone ew add layer fusing is not possible

layer_decomposition

This sub commands allow to toggle layers to decomposition mode. Meaning implement 16bits with 8 bits layers

Example commands:

```
# This will decompose a specific layer to increase it precision
pre_quantization_optimization(layer_decomposition, layers=[conv1],
↪policy=disabled)
pre_quantization_optimization(layer_decomposition, layers=[conv17, conv18],
↪policy=enabled)
```

Parameters:

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

smart_softmax_stats

SmartSoftmaxConfig is an algorithm that collect the stats on a softamx block in a smart way Example commands:

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

Parameters:

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

input_features_defuse

This command allows defusing input features for a selected dense layer to a selected number of splits. It can also be used to disable defusing of a layer.

Example commands:

```
pre_quantization_optimization(input_features_defuse, layers=[fc1], num_splits=2)
# this will disable the fusing of fc2
pre_quantization_optimization(input_features_defuse, layers=[fc2], num_splits=1)
```

Note: num_splits might be overwritten by a larger number due to hw limitations.

Parameters:

Parameter	Values	Default	Required	Description
num_splits	int	None	False	number of splits required

post_quantization_optimization

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

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

The features of this command are:

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

bias_correction

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

Example command:

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

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

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

Parameters:

Parameter	Values	Default	Required	Description
policy	{enabled, disabled}	disabled	False	Enable or disable the bias correction algorithm. When Optimization Level >= 1, could be enabled by the default policy.
cache_compression	bool	False	False	Compress layer results when cached to disk

bias_correction per-layer

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

Example commands:

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

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

Parameters:

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

finetune

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

Example commands:

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

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

Parameters:

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

Parameters (cont.):

Parameter	Values	Default	Required	Description
val_images	int; 0<=x	4096	False	Number of held-up/validation images for evaluation between epochs.
val_batch_size	int; 0<=x	128	False	Batch size for the inter-epoch validation.
stop_gradient_at_loss	bool	False	False	Add stop gradient after each loss layer.
Optimizer	{adam, sgd, momentum, rmsprop}	adam	False	set to 'sgd' to use simple Momentum, otherwise Adam will be used.

Advanced parameters:

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

adaround

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

Example commands:

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

Parameters:

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

Advanced parameters:

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

adaround per-layer

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

Example commands:

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

Parameters:

Parameter	Values	Default	Required	Description
policy	{allowed, enabled, disabled}	allowed	False	None
epochs	int	None	False	Amount of train epochs for a specific layer
weight	float; 0<x	None	False	Weight of round regularization
b_range	[float, float]	None	False	Tempreature decay range
decay_start	float; 0<=x<=1	None	False	Ratio of round train without round regularization decay (b)
train_bias	bool	None	False	Toggle bias training
warmup	float; 0<=x<=1	None	False	Ratio of warmup epochs out of epochs
dataset_size	int; 0<x	None	False	Data samples count for the train stage of the specified layer
batch_size	int; 0<x	None	False	Batch size for train / infer of a layer

Deprecated params

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

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

use_16bit_bias

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

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

use_4bit_weights

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

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

equalization

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

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

bias_correction

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

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

activation_clipping_mode

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

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

weights_clipping_mode

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

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

5.4. Models Compilation

5.4.1. Basic Compilation Flow

For Inference Using TAPPAS or With Native HailoRT API

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

Note: The default compilation target is Hailo-8. To compile for different architecture (Hailo-8R for example), use `hw_arch='hailo8r'` as a parameter to the translation phase. For example see the tutorial referenced on the next note. Hailo-15 uses `hw_arch='hailo15h'`.

See also:

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

For Inference using ONNX Runtime

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

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

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

- The validated opset versions are 8 and 11-17.
- The model needs to be dividable to three sections:
  - Pre-processing, which connects only to the Main model
  - Main model, which connects only to the Post-processing
  - Post-processing
- The start_nodes will completely separate the pre-processing from the Main model No connections from the pre-processing is allowed into the main model, unless they are marked as start_nodes
- The end_nodes need to separate the main model from the post-processing completely

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

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

See also:

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

More information on using OnnxRT with HailoRT is available here.

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

For Inference with Python Using TensorFlow

First, to get a runner loaded with compiled model, use one of the options: calling `compile()`, loading a compiled HAR using `load_har()`, or setting the HEF using `hef()`.

To run inference on the model, enter the context manager `infer_context()` and call `infer()` to get the results.

Note: Inference using the TensorFlow inference is not yet supported on the Hailo-15 platform.

5.4.2. Model Scripts

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

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

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

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

Usage

The script is a separate file which can be given to the `load_model_script()` method of the `ClientRunner` class.

For example:

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

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

Context Switch Parameters

Definition

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

Example

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

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

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

Allocator Parameters

Definition

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

Example

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

Description This sets several allocation parameters described below:

- `timeout` – Compilation timeout for the whole run. By default, the timeout is calculated dynamically based on the model size. The timeout is in seconds by default. Can be given a postfix of 's', 'm', or 'h' for seconds, minutes or hours respectively. e.g. timeout=3h will result to 3 hours.
- `automatic_ddr` – when enabled, DDR portals that buffer data in the host's RAM over PCIe are added automatically when required. DDR portals are added when the data needed to be buffered on some network edge exceeds a threshold. In addition, DDR portal is added only when there are enough resources on-chip to accommodate it. Defaults to `True`. Set to `False` to ensure the HEF compatibility to platforms that don't support it, such as Ethernet based platforms.
- `automatic_reshapes` – When enabled, Format Conversion (Reshape) layers might be added to networks boundary inputs and outputs. They will be added when supported, and when we have enough resources on-chip to accommodate these functions. When disabled, format conversion layers won't be added to boundary inputs and outputs. on chip. Defaults to allowed (compiler's decision to enable or disable).
- `merge_min_layer_utilization` – Threshold of minimum utilization of the 'control' resource, to start the layer auto merger. Auto-merger will try to optimize on-chip implementation by sharing resources between layers, to reach this control threshold. Auto-merger will not fail if target utilization cannot be reached.

Resource Calculation Flow Parameters

Definition

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

Example

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

Description This sets several resources calculation flow parameters described below.

- `strategy` – Resources calculation strategy. When set to `greedy`, adding more resources to the slowest layers iteratively (Maximum FPS search), to reach the highest possible network FPS (per context). Defaults to `greedy`.
- `max_control_utilization` – Number between 0.0 and 1.2. Threshold for `greedy` strategy. Maximum-FPS search will be stopped when the overall control resources on-chip exceeds the given threshold (per context). Defaults to `0.75`.
- `max_compute_utilization` – Number between 0.0 and 1.0. Threshold for `greedy` strategy. Maximum-FPS search will be stopped when the overall compute resources on-chip exceeds the given threshold (per context). Defaults to `0.75`.
- `max_memory_utilization` – Number between 0.0 and 1.0. Threshold for `greedy` strategy. Maximum-FPS search will be stopped when the overall weights-memory resources on-chip exceeds the given threshold. Defaults to `0.75`.

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

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

Place

Definition

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

Example

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

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

Shortcut

Definition

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

Examples

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

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

Portal

Definition

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

Example

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

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

L4 Portal

Definition

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

Example

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

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

DDR Portal

Definition

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

Example

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

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

Concatenation

Definition

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

Example

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

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

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

De-fuse

Definition

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

Examples

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

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

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

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

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

Merge

Definition

```
merge(layer1, layer2)
```

Examples

```
merged_layer = merge(conv46, conv47)
merged_layer_conv12_dw5 = merge(conv12, dw5)
merged_layer_conv15_dw6 = merge(conv15, dw6)
merged_layer_conv18_conv19 = merge(conv18, conv19)
```

Description Merging is a mechanism that uses the same hardware resources to compute two layers. The FPS of the layer will be lower than the two original layers, but unless it is a bottleneck layer, it could save resources and result in total higher FPS. It is supported for a subset of layers and connectivity types. Automatic merging of layers is performed on single context when needed, and could be affected with the *allocator_param(merge_min_layer_utilization)* command.

Compilation Parameters

Definition

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

Example

```
compilation_param(conv1_d0, resources_allocation_strategy=manual_scs_
↪selection, number_of_subclusters=8, use_16x4_sc=enabled)
```

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

Supported compilation params:

- `resources_allocation_strategy` - defaults to `min_l3_mem_match_fps`, which chooses the the number of subclusters that saves most L3 memory (Conv layers only). Change to `min_scs_match_fps` in order to choose the lowest possible number of subclusters. Change to `manual_scs_selection` to manually choose the number of subclusters (Conv, Dense and DW layers only).
- `use_16x4_sc` - can use 16 pixels multiplication by 4 features – instead of the default 8 pixels by 8 features. This is useful when the number of features is smaller than 8. A table of supported layers is given below (layers that are not mentioned are not supported).
- `no_contexts` - change to True in order to accumulate all the needed inputs for each output row computation in the L3 memory. A table of supported layers is given below (layers that are not mentioned are not supported).
- `balance_output_multisplit` - change to False in order to allow unbalanced output buffers. This can be used to save memory when there are "long" skip connections between layers.
- `number_of_subclusters` - force the usage of a specific number of subclusters. Make sure the `resource allocation strategy` value is set to `manual_scs_selection`. This is only applicable to Conv and Dense layers.
- `fps` - force a layer to reach this throughput, possibly higher than the FPS used for the rest of the model. This parameter is useful to reduce the model's latency, however it is not likely to contribute to the model's throughput which is dominated by the bottleneck layer.

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

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

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

Table 15. 16x4 mode support

Kernel type	Kernel size (HxW)	Stride (HxW)	Dilation (HxW)	Padding
Conv	1x1	1x1	1x1	SAME SAME_TENSORFLOW VALID
Conv	3x3	1x1, 2x1	1x1 2x2 (stride=1x1 only) 3x3 (stride=1x1 only) 4x4 (stride=1x1 only)	SAME SAME_TENSORFLOW
Conv	5x5	1x1, 2x1	1x1	SAME SAME_TENSORFLOW

Continued on next page

Table 15 – continued from previous page

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

Table 16. No contexts mode support

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

HEF Parameters

Definition

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

Example

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

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

Supported hef parameters:

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

Outputs Multiplexing

Definition

```
output_mux(layers)
```

Example

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

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

From TF

Definition

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

Example

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

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

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

Buffers

Definition

```
buffers(layer_from, layer_to, number_of_rows_to_buffer)
buffers(layer_from, layer_to, number_of_rows_cluster_a, number_of_rows_
↪cluster_b)
```

Example

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

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

Feature Splitter

Definition

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

Example

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

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

Format Conversion

Definition

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

Example

```
reshape1 = format_conversion(input_layer1, conv1, tf_rgb_to_hailo_rgb)
reshape_yuy2 = format_conversion(input_layer1, yuy2_to_hailo_yuv)
reshape_nv12, resize, concat = format_conversion(input_layer1, nv12_to_hailo_
↪yuv)
```

Description Add a format conversion layer. This command is useful to offload host operations to the Hailo device. The supported format conversions are:

- `tf_rgb_to_hailo_rgb` – Converts an NHWC tensor ("TF RGB") to an NHCW tensor ("Hailo RGB"). NHCW is the format used by the Hailo core for most layers, such as Conv and Maxpool. This is useful before the first layer to offload this conversion from the host (in other words, from HailoRT). Despite its name, this conversion can be applied even if the number of features is not 3.
- `hailo_rgb_to_tf_rgb` – The inverse transformation of `tf_rgb_to_hailo_rgb`. This is useful after the last layer to offload this conversion from the host. Despite its name, this conversion can be applied even if the number of features is not 3.
- `yuy2_to_hailo_yuv` – Converts the YUY2 format, which is used by some cameras, to YUV. This is useful together with the YUV to RGB layer (see `add_yuv_to_rgb_layers()`) to create a full vision pipeline YUY2 □ YUV □ RGB. Corresponds to cv::COLOR_YUV2RGB_YUY2 in OpenCV terminology.
- `nv12_to_hailo_yuv` – converts the NV12 format, which is used by a growing number of cameras, to YUV format. This is a useful conversion to be used before the first layer to offload this conversion from the host. Uses a slightly different command format, demonstrated above.
- `nv21_to_hailo_yuv` – Converts the NV21 format, which is used by some cameras, to YUV.
- `i420_to_hailo_yuv` – Converts the i420 format, which is used by some cameras, to YUV.
- `tf_rgbx_to_hailo_rgb` – Converts RGBX to Hailo RGB format.

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

Cascade of Portals / Buffers

Definition

```
created_layers_list = cascade(layer_from, layer_to, types=[type_0, type_1, ..],
↪ extra_arg)
```

Examples

```
some_name_1, some_name_2, some_name_3 = cascade(input_layer, conv1,
↪types=[shortcut, portal, shortcut]) # similar to equal_weights
some_name_1, some_name_2, some_name_3 = cascade(input_layer, conv1,
↪types=[shortcut, portal, shortcut], equal_weights) # similar to previous
shortcut10, shortcut11, shortcut12, ddr13 = cascade(conv1, conv2,
↪types=[shortcut, shortcut, shortcut, ddr], total_buffers=30)
portal20, shortcut21, portal22, shortcut23 = cascade(conv2, conv3,
↪types=[portal, shortcut, portal, shortcut], buffers=[10,5,7,2,0])
shortcut30, shortcut31, portal32, portal33 = cascade(shortcut23, conv3,
↪types=[shortcut, shortcut, portal, l4_portal], weights=[3,1,5,6,3])
```

Description Add a cascade of N buffers and portals between the source and destination layer. The supported types of the cascade are:

- shortcut
- portal
- l4_portal
- ddr

Supported are four types of arguments:

- buffers=[N+1 comma separated integers]: Append the requested buffers to [layer_from, first_cascade_layer, .., last_cascade_layer]. Possible values are integers, with 0 means no assignment (use the automatically calculated value).
- total_buffers=value: Each layer in [layer_from, first_cascade_layer, .., last_cascade_layer] gets (value / N+1) buffers.
- weights=[N+1 comma separated double values]: Automatically calculates the number of buffers, and tries to divide them between the cascade layers according to the ratio between the weights ([1,1,2] is similar to [0.25,0.25,0.5]). Possible values are doubles.
- equal_weights (also with no extra_arg): Assumes the weights are 1/(N+1) each, continue as "weights" type.

Notes:

- l4_portal must be first in cascade, last in cascade, or surrounded with shortcuts / portals (regular, not l4).
- ddr must be first in cascade, last in cascade, or surrounded with shortcuts / portals (regular, not l4).

Platform Param

Definition

```
platform_param(param=value)
```

Examples

```
platform_param(targets=[ethernet])
platform_param(hints=[low_pcie_bandwidth])
```


Description This sets several parameters regarding the platform hosting Hailo as described below:

- `targets` - a list or a single value of hosting target restrictions such as `Ethernet` which requires disabling a set of features.

  Current supported targets: `Ethernet`, which disables the following features:

  - DDR portals, since the DDR access through PCIe is not available
  - Context Switch (multi contexts), since DDR access is not available
  - Sequencers (a fast PCIe-based model loading)

- `hints` - a list of hints or a single hint about the hosting platform such as `Low PCIE bandwidth` which optimizes performance for specific scenarios.

  Current supported hints: `low_pcie_bandwidth`, adjusts the compiler to reduce the PCIE bandwidth by disabling or changing decision thresholds regarding when PCIE should be used.

Performance Param

Definition

```
performance_param(compiler_optimization_level=max)
```

Description Setting this parameter enters *performance mode*, in which the compiler will try as hard as it can to find a solution that will fit in a single context, with the highest performance. This method of compilation will require significantly longer time to complete, because the compiler tries to use very high utilization levels, that might not allocate successfully. If it fails to allocate, it automatically tries lower utilization, until it finds the highest possible utilization.

Network Groups

Definition

```
group_name = network_group([scope_1_0, ... , scope_0_n])
```

Example

```
my_group = network_group([net1, net2])
```

Description Define network groups to include the different connected components ("networks" or "scopes") in the model. When more than one network_group command is used, a multi-network-group .hef file is created. Each network_group should be activated at a time using HailoRT (or, preferably, using HailoRT Scheduler API).

The old default behavior (up to and including v3.19) was to compile all connected components together to a single network group, so that they are allocated side-by-side on chip, along all contexts.

The new default is changed so that each connected component resides on a different network group, each with its own separate contexts (similar to using multiple .hef files). The new behavior could be reverted by explicitly putting all connected components in one network group.

This command requires:

- Using the `join()` API with `join_action=JoinAction.NONE` to create a "multi-network file" that includes both networks (can use multiple times to add more networks).
- Intersection of all network groups should be an empty group - no scope can compile in 2 separate groups.
- Union of all network groups equals all the networks in the hn or given network.

5.5. Supported Layers

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

Note: Unless otherwise specified, the supported features in this section describe their support across the Dataflow Compiler components (Profiler, Allocator, Compiler, Emulator and Model Optimization).

Note: Padding type definitions are:

- `SAME`: *Symmetric padding*.
- `SAME_TENSORFLOW`: Identical to Tensorflow SAME padding.
- `VALID`: No padding, identical to Tensorflow VALID padding.

Note: Up to four successor layers are supported after each layer. Each successor receives the same data, except when using the *Features Split layer*.

5.5.1. Convolution

Convolution layers are supported with any integer values of kernel size, stride, dilation. Padding types supported are: VALID, SAME, and SAME_TENSORFLOW. The following table displays the current optimized params.

Table 17. Convolution kernel optimized parameters

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding

1x1	1x1, 2x1, 2x2	1x1	`SAME`
			`SAME_TENSORFLOW`
			`VALID`

3x3	1x1, 1x2, 2x1, 2x2	1x1
		2x2 (stride=1x1 only)
		3x3 (stride=1x1 only)
		4x4 (stride=1x1 only)
		6x6 (stride=1x1 only)
		8x8 (stride=1x1 only)
		16x16 (stride=1x1 only)
			`SAME`
			`SAME_TENSORFLOW`
			`VALID`

2x2	2x1	1x1	`SAME`
			`SAME_TENSORFLOW`
			`VALID`

2x2, 2x3, 2x5,2x7, 3x2, 5x2, 7x2	2x2	1x1	`SAME` `SAME_TENSORFLOW`

5x5, 7x7	1x1, 1x2, 2x1, 2x2	1x1	`SAME`
			`SAME_TENSORFLOW`
			`VALID` (stride=1x1 only)

6x6	2x2	1x1	`SAME`
			`SAME_TENSORFLOW`
			`VALID`

Continued on next page

Table 17 – continued from previous page

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
1x3, 1x5, 1x7	1x1, 1x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1 only)
3x5, 3x7, 5x3, 5x7, 7x3, 7x5	1x1, 1x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1 only)
3x1	1x1, 2x1	1x1	SAME SAME_TENSORFLOW VALID
5x1, 7x1	1x1	1x1	SAME SAME_TENSORFLOW VALID
1x9, 3x9, 5x9, 7x9	1x1	1x1	SAME SAME_TENSORFLOW VALID
9x1, 9x3, 9x5, 9x7, 9x9	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x4, 5x4, 7x4, 9x4	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x6, 5x6, 7x6, 9x6	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x8, 5x8, 7x8, 9x8	1x1	1x1	SAME SAME_TENSORFLOW VALID
1xW	1x1	1x1	SAME SAME_TENSORFLOW VALID
MxN	MxN	1x1	SAME SAME_TENSORFLOW VALID
MxN, where M,N in {1..16}	AxB, where A,B in {1..4}	CxD, where C,D in {1..9}	SAME SAME_TENSORFLOW VALID
Any other	Any other	Any other	SAME SAME_TENSORFLOW VALID

'W' refers to the width of the layer's input tensor, in this case the kernel width is equal to the image width

Table 18. Convolution & add kernel supported parameters

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
1x1, 1x3, 1x5, 1x7	1x1	1x1	SAME SAME_TENSORFLOW VALID
3x1, 3x3, 3x5, 3x7	1x1	1x1	SAME SAME_TENSORFLOW VALID
5x1, 5x3, 5x5, 5x7	1x1	1x1	SAME SAME_TENSORFLOW VALID
7x1, 7x3, 7x5, 7x7	1x1	1x1	SAME SAME_TENSORFLOW VALID

Note: Convolution kernel with elementwise addition supports the addition of two tensors only.

Note: Number of weights per layer <= 8MB (for all Conv layers).

5.5.2. Max Pooling

Table 19. Max pooling kernel supported parameters

Kernel (HxW)	Stride (HxW)	Padding
2x2	1x1, 2x1, 2x2	SAME SAME_TENSORFLOW VALID
1x2	1x2	SAME SAME_TENSORFLOW VALID
3x3	1x1	SAME SAME_TENSORFLOW VALID
3x3	2x2	SAME SAME_TENSORFLOW VALID
5x5, 9x9, 13x13	1x1	SAME SAME_TENSORFLOW VALID
Any other	Any other	SAME SAME_TENSORFLOW VALID

"Any other" means any kernel size or stride between 2 and the tensor's dimensions, for example $2 \leq k_h \leq H$ where k_h is the kernel height and H is the height of the layer's input tensor.

5.5.3. Dense

Dense kernel is supported. It is supported only after a Dense layer, a Conv layer, a Max Pooling layer, a Global Average Pooling layer, or as the first layer of the network.

When a Dense layer is after a Conv or a Max Pooling layer, the data is reshaped to a single vector. The height of the reshaped image in this case is limited to 255 rows.

5.5.4. Average Pooling

Average Pooling layers are supported with any integer values of kernel size, stride, and dilation. Padding types supported are: VALID, SAME, and SAME_TENSORFLOW. The following table displays the current optimized params.

Table 20. Average pooling kernel optimized parameters

Kernel (HxW)	Stride (HxW)	Padding
2x2	2x2	SAME SAME_TENSORFLOW VALID
3x3	1x1, 2x2	SAME SAME_TENSORFLOW
3x4	3x4	SAME SAME_TENSORFLOW VALID
5x5	1x1, 2x2	SAME SAME_TENSORFLOW
hxW	hxW	VALID
Global	n/a	n/a
SAME SAME_TENSORFLOW VALID		

'W' means the width of the layer's input tensor. In other words, in this case the kernel width equals to the image width. 'h' means any height, from 1 up to the input tensor height.

5.5.5. Concat

This layer requires 4-dimensional input tensors (batch, height, width, features), and concatenates them in the features dimension. It supports up to 4 inputs.

5.5.6. Deconvolution

Table 21. Deconvolution kernel supported parameters

Kernel (HxW)	Rate (HxW)	Padding
16x16	8x8	SAME_TENSORFLOW
8x8	4x4	SAME_TENSORFLOW
4x4	4x4	SAME_TENSORFLOW
4x4	2x2	SAME_TENSORFLOW
2x2	2x2	SAME_TENSORFLOW

Continued on next page

Table 21 – continued from previous page

Kernel (HxW)	Rate (HxW)	Padding
1x1	1x1	SAME_TENSORFLOW

5.5.7. Depthwise Convolution

Depthwise Convolution layers are supported with any integer values of kernel size, stride, and dilation. Padding types supported are: VALID, SAME, and SAME_TENSORFLOW. Utilizing a Depthwise 1x1 stride 1x1 kernel with elementwise addition, supports the addition of two tensors only

Table 22. Depthwise convolution kernel optimized parameters

Kernel (HxW)	Stride (HxW)	Dilation (HxW)	Padding
1x1	1x1	1x1	SAME SAME_TENSORFLOW VALID
2x2	2x2	1x1	SAME SAME_TENSORFLOW VALID
3x3	1x1, 2x2	1x1 2x2 (stride=1x1 only) 4x4 (stride=1x1 only)	SAME SAME_TENSORFLOW VALID (stride=1x1, dilation=1x1 only)
3x5, 5x3	1x1	1x1	SAME SAME_TENSORFLOW VALID
5x5	1x1, 2x2	1x1	SAME SAME_TENSORFLOW VALID (stride=1x1, dilation=1x1 only)
9x9	1x1	1x1	SAME SAME_TENSORFLOW
SAME SAME_TENSORFLOW VALID			

5.5.8. Group Convolution

Group Convolution is supported with all supported Convolution kernels.

For Conv 1x1/1, 1x1/2, 3x3/1, and 7x7/2, any number of output features is supported. For all other supported Conv kernels, only OF%8=0 or OF<8 is supported, where OF is the number of output features in each group.

5.5.9. Group Deconvolution and Depthwise Deconvolution

Group Deconvolution is supported with all supported Deconvolution kernels. Only OF%8=0 or OF<8 is supported, where OF is the number of output features in each group.

Depthwise Deconvolution is a sub case of Group Deconvolution.

5.5.10. Elementwise Multiplication and Division

Elementwise operations require:

1. Two input tensors with the same shape.

   Example: [N, H, W, F], [N, H, W, F]

2. Two tensors with the same batch and spatial dimensions, one tensor has features dimension 1.

   Example: [N, H, W, F], [N, H, W, 1]

3. Two tensors with the same batch and feature dimensions, one of them has spatial dimension [1, 1].

   Example: [N, H, W, F], [N, 1, 1, F].

4. Two tensors with the same batch dimension, one of them has feature and spatial dimension [1, 1, 1].

   Example: [N, H, W, F], [N, 1, 1, 1].

Note: The *resize layer* can broadcast a tensor from (batch, 1, 1, F) to (batch, height, width, F), where F is the number of features. This may be useful before the Elementwise Multiplication layer.

5.5.11. Add and Subtract

Add and subtract operations are supported in several cases:

1. Bias addition after Conv, Deconv, Depthwise Conv and Dense layers. Bias addition is always fused into another layer.
2. Elementwise addition and subtraction: When possible, elementwise add / sub is fused into a Conv layer as detailed above. Elementwise add / sub is supported on both “Conv like” and “Dense like” tensors, with shapes in the format shown on *Elementwise Multiplication and Division*
3. Addition of a constant scalar to the input tensor.

5.5.12. Input Normalization

Input normalization is supported as the first layer of the network. It normalizes the data by subtracting the given mean of each feature and dividing by the given standard deviation.

5.5.13. Multiplication by Scalar

This layer multiplies its input tensor by a given constant scalar.

5.5.14. Batch Normalization

Batch normalization layer is supported. When possible, it is fused into another layer such as Conv or Dense. Otherwise, it is a standalone layer.

Calculating Batch Normalization statistics in runtime using the Hailo device is not supported.

5.5.15. Resize

Two methods are supported: Nearest Neighbor (NN) and Bilinear. In both methods, the scaling of rows and columns can be different.

These methods are supported in three cases:

1. When the columns and rows scale is a float (for rows also <= 4096), the new sizes are integers, where half_pixels and align_corners satisfies one of the following: align_corners=True & half_pixels=False, align_corners=False & half_pixels=True, align_corners=False & half_pixels=False.
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" (NN only).
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" (NN only).

Note: align_corners: If True, the centers of the 4 corner pixels of the input and output tensors are aligned, preserving the values at the corner pixels. See definition here (PyTorch) and here (TensorFlow).

half_pixel: Relevant for Pytorch / ONNX, as defined on the ONNX Operators page, under *coordinate_transformation_mode*.

5.5.16. Depth to Space

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

Two modes are supported (check out ONNX operators spec for more info - https://github.com/onnx/onnx/blob/main/docs/Operators.md#depthtospace):

1. "DCR" mode – the default mode, where elements along the depth dimension from the input tensor are rearranged in the following order: depth, column, and then row.
2. "CRD" mode – elements along the depth dimension from the input tensor are rearranged in the following order: column, row, and the depth.

MxN block size is supported, where M, N are integers, in both modes.

Table 23. Depth to space kernel supported parameters

Block size (HxW)
1x2
2x1
2x2

Depth to space is only supported when $IF\%(B_W \cdot B_H) = 0$, where IF is the number of input features, B_W is the width of the depth to space block and B_H is the height of the block.

5.5.17. Space to Depth

Space to depth rearranges blocks of spatial data into the depth (features) dimension.

1. "Classic" variant – The inverse of the Depth to Space kernel. It is identical to Tensorflow's `space_to_depth` operation. Supports MxN block size, where M, N are integers.
2. "Focus" variant – It supports the 2x2 block size. Used by models such as YOLOv5, YOLOP. It is defined by the following Tensorflow code:

```
op = tf.concat([inp[:, ::block_size, ::block_size, :], inp[:, 1::block_size, ::block_
↪size, :],
        inp[:, ::block_size, 1::block_size, :], inp[:, 1::block_size, 1::block_size,
↪ :]], axis=3)
```

where `inp` is the input tensor.

5.5.18. Softmax

Softmax layer is supported in three cases:

1. After a "Dense like" layer with output shape (batch, features). In this case, Softmax is applied to the whole tensor.
2. After another layer, if the input tensor of the Softmax layer has a single column (but multiple features). In this case, Softmax is applied row by row.
3. After another layer, even if it has multiple columns. In this case Softmax is applied pixel by pixel on the feature dimension. This case is implemented by breaking the softmax layer to other layers.

5.5.19. Argmax

Argmax kernel is supported if it is the last layer of the network, and the layer before it is has a 4-dimensional output shape (batch, height, width, features).

Note: Currently argmax supports up to 64 features.

5.5.20. Reduce Max

Reduce Max is supported along the features dimension, and if the layer before it is has a 4-dimensional output shape (batch, height, width, features).

5.5.21. Reduce Sum

If the layer before it is has a 4-dimensional output shape (batch, height, width, features), the Reduce Sum layer is supported along the features and width dimension. If the layer before it has a 2-dimensional output shape (batch, features), the Reduce Sum layer is supported along the features dimension.

5.5.22. Reduce Sum Square

Reduce Sum Square is supported along the features or the spatial dimensions.

5.5.23. Feature Shuffle

Feature shuffle kernel is supported if F%G = 0, where G is the number of feature groups.

5.5.24. 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.

5.5.25. 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.

5.5.26. 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 $F = W' \cdot F'$.

5.5.27. 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`).

5.5.28. Matmul

This layer implement data driven matrices multiplication X x Y = Z. Input sizes should obey matrices multiplication rules. The layer support the following X, Y output shapes, where B = batch, H = rows, W = columns, F = channels for X and " K' " for Y.

1. Shape(X) = B, H, W, F; shape(Y) = B, F', W', H'; shape(Z) = B, H, W, (H'xW'). Multiplication is executed over 2 dimensional matrices, per batch: ((HxW), F) * (F', H'xW') = (HxW, H'xW')

2. Shape(X) = B, H, W, F; shape(Y) = B, F', W', H'; shape(Z) = B, H, W, F'. Multiplication is executed over 2 dimensional matrices, per batch: (HxW, F) * (H'xW', F') = (HxW, F') Supported only when B output data is positive.

Supported in preview.

5.5.29. Multi Head Attention

This layer is a major building block for Transformer models. It receives (K, Q, V) matrices, and implements the formula:

$$Softmax\left(\frac{Q_i \cdot K_i^T}{\sqrt{d_k}}\right) \cdot V_i$$

When Q_i, K_i, V_i are matrices that result from multiplying the input matrices K, Q, V by W_i^K, W_i^Q, W_i^V respectively (W are learned matrices), i ranges from 0 to #heads - 1. Then concatenating the results after multiplying by a learned weights vector W^0.

5.5.30. Activations

The following activations are supported:

- Linear
- Relu
- Leaky Relu
- Relu 6
- Elu
- Sigmoid
- Exp
- Tanh
- Softplus
- Threshold, defined by `x if x >= threshold else 0`.
- Delta, defined by `0 if x == 0 else const`.
- SiLU
- Swish
- Mish
- Hard-swish (preview)
- Gelu (preview)
- PRelu
- Sqrt
- Log
- Hard-sigmoid
- Min
- Max
- Clip
- AddN (only in TFlite)
- Less

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

5.5.31. Square, Pow

- Square operator (x*x) is supported.
- Pow operator is currently supported only with exponent=2 (x^2).

5.5.32. L2 Operators

- ReduceL2 is supported.
- L2Normalization is supported.

5.5.33. Note about symmetric padding

The Hailo Dataflow Compiler supports symmetric padding as supported by other frameworks such as Caffe. As the `SAME` padding in Tensorflow is not symmetric, the only way to achieve this sort of padding is by explicitly using `tf.pad` followed by a convolution operation with `padding='VALID'`. The following code snippet shows how this would be done in Tensorflow (the padding generated by this code is supported by the Dataflow Compiler):

```
pad_total_h = kernel_h - 1
if strides_h == 1:
  pad_beg_h = int(ceil(pad_total_h / 2.0))
else:
  pad_beg_h = pad_total_h // 2
pad_end_h = pad_total_h - pad_beg_h

# skipping the same code for pad_total_w

inputs = tf.pad(
  inputs,
  [[0, 0], [pad_beg_h, pad_end_h], [pad_beg_w, pad_end_w], [0, 0]])
```


Part II

API Reference

6. Model Build API Reference

6.1. hailo_sdk_client.runner.client_runner

Hailo DFC API client.

class hailo_sdk_client.runner.client_runner.ClientRunner(*hn=None*, ...)
Bases: object

Hailo DFC API client.

__init__(*hn=None*, *hw_arch=None*, *hw_version=None*, *har_path=None*, *har=None*)
DFC 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 DFC 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.

property model_script

force_weightless_model(*weightless=True*)
DFC 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.

set_keras_model(*model: hailo_model_optimization.flows.inference_flow.SimulationTrainingModel*)
Set Keras model after quantization-aware training. This method allows you to set the model after editing it externally. After setting the model new quantized weights are generated.

Parameters model(SimulationTrainingModel) – model to set.

get_keras_model(*context:* hailo_sdk_client.exposed_definitions.InferenceContext, *trainable=False*) ...)
Get Keras model for inference. This method returns a model for inference in either Native, fp optimized, quantized, or HW mode. Editing the keras model won't affect quantization/compilation unless set_keras_model() API is being used.

Parameters

- context (InferenceContext) – inference context generated by infer_context
- trainable(bool, optional) – indicate whether the returned model should be trainable or not. set_keras_model() only supports trainable models.

Example

```
>>> with runner.infer_context(InferenceContext.SDK_NATIVE) as ctx:
>>>   result = runner.get_keras_model(context=ctx)
```

`infer`(*context:* hailo_sdk_client.exposed_definitions.InferenceContext, *dataset*, ...)

DFC API for inference. This method infer the given dataset on the model in either full precision, emulation (quantized), or HW and returns the output.

Parameters

- `context` (`InferenceContext`) – inference context generated by infer_context
- `dataset` – data for Inference. Type depends on the `data_type` parameter.
- `data_type` (`InferenceDataType`) – dataset's data type, based on enum values:
  - `auto` – Automatically detection.
  - `np_array` - `numpy.ndarray`, or dictionary with input layer names as keys, and values types of `numpy.ndarray`.
  - `dataset` - `tensorflow.data.Dataset` object with valid signature. signature should be either ((h, w, c), image_info) or ({'input_layer1': (h1, w1, c1), 'input_layer2': (h2, w2, c2)}, image_info) image_info can be an empty dict for inference
  - `npy_file` – path to a npy or npz file
  - `npy_dir` – path to a npy or npz dir, assumes same shape to all the items
- `data_count` (`int`) – optional argument to limit the number of elements to infer
- `batch_size` (`int`) – batch size for inference

Returns list: list of outputs. Entry i in the list is the output of input i. In case the model contain more than one output, each entry is a list of all the outputs.

Example

```
>>> with runner.infer_context(InferenceContext.SDK_NATIVE) as ctx:
>>>   result = runner.infer(
...     context=ctx,
...     dataset=tf.data.Dataset.from_tensor_slices(np.ones((1, 10))),
...     batch_size=1
...   )
```

`load_model_script`(*model_script=None*)

DFC API for manipulation of the model build params. This method loads a script and applies it to the existing HN, i.e., modifies the specific params in each layer, and sets the model build script for later use.

Parameters `model_script` (`str`) – Model script given as either path to alls file or commands as string 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. Model modification related commands – These commands are executed during optimization.
2. Quantization related commands – Some of these commands modify the HN, so after the modification each layer (possibly) has new quantization params. Other commands are executed during optimization.
3. Allocation and compilation related commands – These commands are executed during compilation.

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

Return type dict

`translate_params` (*inference_results*, *previous_statistics=None*, *use_old_params=False*, ...)
DFC 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`

`load_params` (*params*, *params_kind=None*)
Load network params (weights).

Parameters

- `params` – If a string, this is treated as the path of the npz file to load. If a dict, this is treated as the params themselves, where the keys are strings and the values are numpy arrays.
- `params_kind` (`str, optional`) – Indicates whether the params to be loaded are native, native after BN fusion, or quantized.

Returns Kind of params that were actually loaded.

Return type str

`save_params` (*path*, *params_kind='native'*)
Save all model params to a npz file.

Parameters

- `path` (`str`) – Path of the npz file to save.
- `params_kind` (`str, optional`) – Indicates whether the params to be saved are native, native after BN fusion, or quantized.

`get_previous_hailo_export()`
Get the last Hailo export returned to the user.

`compile()`
DFC API for compiling current model to Hailo hardware.

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

Return type bytes

Example

```
>>> runner = get_example_runner()
>>> compiled_model = runner.compile()
```

`hef_infer_context`(*hailo_export*)

`infer_context`(*inference_context:* hailo_sdk_client.exposed_definitions.InferenceContext, ...)
DFC API for generating context for inference. The context must be used with the *infer* API.

Parameters

- `inference_context` (`InferenceContext`) – Enum to control which inference type to use.
- `device_ids` (`list of str, optional`) – devices ids to create VDevice from, call `Device.scan()` to get list of all available devices. Excludes 'params'.

Raises

- `HailoPlatformMissingException` – In case, HW inference is requested but HailoRT is not installed.
- `InvalidArgumentsException` – In case, InferenceContext is not recognized.

Example

```
>>> with runner.infer_context(InferenceContext.SDK_NATIVE) as ctx:
>>>   result = runner.infer(
...     context=ctx,
...     dataset=tf.data.Dataset.from_tensor_slices(np.ones((1, 10))),
...     batch_size=1
...   )
```

`translate_onnx_model`(*model=None*, *net_name='model'*, *start_node_names=None*, ...)
DFC API for parsing an ONNX model. This creates a runner with loaded HN (model) and parameters.

Parameters

- `model` (`str or bytes or pathlib.Path`) – Path or bytes of the ONNX model file to parse.
- `net_name` (`str`) – Name of the new HN to generate.
- `start_node_names` (`list of str, optional`) – List of ONNX nodes that parsing will start from.
- `end_node_names` (`list of str, optional`) – List of ONNX nodes, that the parsing can stop after all of them are parsed.
- `net_input_shapes` (`dict or list, optional`) – A dictionary describing the input shapes for each of the start nodes given in start_node_names, where the keys are the names of the start nodes and the values are their corresponding input shapes. Use only when the original model has dynamic input shapes (described with a wildcard denoting each dynamic axis, e.g. [b, c, h, w]). Can be list (e.g. [b, c, h, w]) for single input network.
- `augmented_path` – Path to save a modified model, augmented with tensors names (where applicable).
- `disable_shape_inference` – When set to True, shape inference with onnx runtime will be disabled.
- `disable_rt_metadata_extraction` – When set to True, runtime metadata extraction will be disabled. Generating model using `get_hailo_runtime_model()` won't be supported in this case.

Note: Using a non-default `start_node_names` requires the model to be shape inference compatible, meaning either it has a real input shape, or, in case of a dynamic input shape, the `net_input_shapes` field is provided to specify the input shapes of the given start nodes. The order of the output nodes is determined by the order of the end_node_names.

Returns The first item is the HN JSON as a string. The second item is the params dict.

Return type tuple

`translate_tf_model`(*model_path=None*, *net_name='model'*, *start_node_names=None*, ...)

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

Parameters

- `model_path` (`str`) – Path of the file to parse. Supported formats: Chekpoint (TF1): Model name with .ckpt suffix (without the final .meta). Frozen (TF1): Frozen graph, model name with .pb suffix. SavedModel (TF2): Saved model export from keras, file named saved_model.pb|pbtxt from the model dir. TFLite: Tensorflow lite model, converted from ckpt/frozen/keras to file with .tflite suffix.
- `net_name` (`str`) – Name of the new HN to generate.
- `start_node_names` (`list of str, optional`) – List of tensorflow nodes that parsing will start from. If this parameter is specified, start_node_name should remain empty.
- `end_node_names` (`list of str, optional`) – List of Tensorflow nodes, which the parsing can stop after all of them are parsed.
- `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.

Note: The order of the output nodes is determined by the order of the end_node_names.

Returns The first item is the HN JSON, as a string. The second item is the params dict.

Return type tuple

Example

```
>>> inputs = tf.compat.v1.placeholder(tf.float32, [None, 32, 32, 1])
>>> conv = tf.layers.conv2d(inputs, 16, 3, activation=tf.nn.relu, name='my_
↪conv')
>>> sess = tf.Session()
>>> with sess.as_default():
...   _ = sess.run([tf.compat.v1.global_variables_initializer()])
...   _ = tf.train.Saver().save(sess, './example.ckpt')
>>> runner = ClientRunner(hw_arch='hailo8')
>>> hn, params = runner.translate_tf_model(
...   'example.ckpt', 'MyCoolModel', ['my_conv/Conv2D'], ['my_conv/Relu'])
```

`join`(*runner*, *scope1_name=None*, *scope2_name=None*, *join_action=JoinAction.NONE*, *join_action_info=None*)

DFC API to join two models, so they will be compiled together.

Parameters

- `runner` (`ClientRunner`) – The client runner to join to this one.

- `scope1_name` (`dict or str, optional`) – In case dict is given, mapping between exisiting scope names to new scope names for the layers of this model (see example below). In case str is given, scope name to use for all layers of this model. String can be used only when there is a single scope name.
- `scope2_name` (`dict or str, optional`) – Same as *scope1_name* for the runner to join.

  Example:

```
>>> net1_scope_names = {'net1_scope1': 'net_scope1',
...                     'net1_scope2': 'net_scope2'}
>>> net2_scope_names = {'net2': 'net_scope3'}
>>> runner1.join(runner2, scope1_name=net1_scope_names,
...              scope2_name=net2_scope_names)
```

- `join_action` (`JoinAction`, optional) – Type of action to run in addition to joining the models:
  - `NONE`: Join the graphs without any connection between them.
  - `AUTO_JOIN_INPUTS`: Automatically detect inputs for both graphs, and join them into one. Only works when both networks have a single input of the same shape.
  - `AUTO_CHAIN_NETWORKS`: Automatically detect output of this model, and input of the other model, and connect them. Only works when this model has a single output, and the other model has a single input, of the same shape.
  - `CUSTOM`: Supply a custom dictionary `join_action_info`, which specifies which nodes from this model need to be connected to which of the nodes in the other graph. If keys and values are inputs, the inputs are joined. If keys are outputs, and values are inputs, the networks are chained as described in the dictionary.
- `join_action_info` (`dict, optional`) – Join information to be given when `join_action` is `NONE`, as explained above.

Example

```
>>> info = {"net1/output_layer1": "net2/input_layer2",
...         "net1/output_layer2": "net2/input_layer1"}
>>> runner1.join(runner2, join_action=JoinAction.CUSTOM, join_action_
↪info=info)
```

`profile`(*profiling_mode=None*, *should_use_logical_layers=True*, *hef_filename=None*, *runtime_data=None*, ...)

DFC API of the Profiler.

Parameters

- `profiling_mode` (`ProfilerModes`, optional) – The mode the profiler is executed in. Defaults to None, which sets the profiling mode to `PRE_PLACEMENT`.
- `hef_filename` (`str, optional`) – HEF file path. If given, the HEF file is used. If not given and the HEF from the previous compilation is cached, the cached HEF is used; Otherwise the automatic mapping tool is used. Use `compile()` to generate and set the HEF. Only in post-placement mode. Defaults to None.
- `should_use_logical_layers` (`bool, optional`) – Indicates whether the Profiler should combine all physical layers into their original logical layer in the report. Defaults to True.
- `runtime_data` (`str, optional`) – runtime_data.json file path produced by hailortcli collect-runtime-data.
- `analysis_data` (`str, optional`) – analysis_data.json file path produced by the *hailo analyze-noise* command.

- `stream_fps` (`float, optional`) – FPS used for power and bandwidth calculation.

Returns The first item is a JSON with the profiling result summary. The second item is a CSV table with detailed profiling information about all model layers. The third item is the latency data. Fourth is accuracy data.

Return type tuple

Example

```
>>> runner = get_example_runner()
>>> summary, details, latency, accuracy = runner.profile()
```

`save_autogen_allocation_script(`*path*`)`
DFC API for retrieving listed operations of last allocation in .alls format.

Parameters `path` (`str`) – Path where the script is saved.

Returns False if autogenerated script was not created; otherwise it returns True.

Return type bool

`property model_name`
Get the current model (network) name.

`property model_optimization_commands`

`property hw_arch`

`property state`
Get the current model state.

`property hef`
Get the latest HEF compilation.

`property nms_config_file`

`property nms_engine`

`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_translated(`*keys=None*`)`
Get the quantized params the SDK uses.

Parameters `keys` (`list of str, optional`) – List of params to retrieve. If not specified, all params are retrieved.

`get_params_fp_optimized(`*keys=None*`)`
Get the fp optimized params.

Parameters `keys` (`list of str, optional`) – List of params to retrieve. If not specified, all params are retrieved.

`get_params_statistics(`*keys=None*`)`
Get the optimization statistics. During optimization stage we gather statistics about the model and about the optimization algorithms. This method returns this information in a ModelParams structure.

Parameters `keys` (`list of str, optional`) – List of params to retrieve. If not specified, all params are retrieved.

`get_hn_str()`
Get the HN JSON after serialization to a formatted string.

`get_hn_dict()`
Get the HN of the current model as a dictionary.

`get_hn()`
Get the HN of the current model as a dictionary.

`get_hn_model()`
Get the `HailoNN` object of the current model.

`get_native_hn_str()`
Get the HN JSON after serialization to a formatted string.

`get_native_hn_dict()`
Get the HN of the current model as a dictionary.

`get_native_hn()`
Get the HN of the current model as a dictionary.

`get_native_hn_model()`
Get the `HailoNN` object of the current model.

`set_hn(`*hn*`)`
Set the HN of the current model.

Parameters `hn` – Hailo network description (HN), as file-like object, string, dict or `HailoNN`.

`save_hn(`*path*`)`
Save the HN of the current model.

Parameters `path` (`str`) – Path where the hn file is saved.

`save_native_hn(`*path*`)`
Save the HN of the current model.

Parameters `path` (`str`) – Path where the hn file is saved.

`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.

`model_summary()`
Prints summary of the model layers.

`optimize_full_precision()`

Apply model optimizations to the model, keeping full precision:

1. Fusing various layers (e.g. conv and elementwise-add, fold batch_normalization, etc.), including folding of fused layers params.
2. Apply model modification commands from model script (e.g. resize input, transpose, color conversion, etc.)
3. Run structural optimization algorithms (e.g. dead channels removal, tiling squeeze & excite, etc.)

`analyze_noise(`*dataset*, *data_type=CalibrationDataType.auto*, *data_count: int = None*, *batch_size: int = ...*`)`

Run layer noise analysis on quantized model:

- Analyze the model accuracy
- Generate analysis data to be visualized in the Hailo profiler

Parameters

- `dataset` – data for the analysis. Type depends on the `data_type` parameter.
- `data_type` (optional, `InferenceDataType`) – dataset's data type, based on enum values:
  - `auto` – Automatically detection.
  - `np_array` – `numpy.ndarray`, or dictionary with input layer names as keys, and values types of `numpy.ndarray`.
  - `dataset` – `tensorflow.data.Dataset` object with valid signature. signature should be either ((h, w, c), image_info) or ({'input_layer1': (h1, w1, c1), 'input_layer2': (h2, w2, c2)}, image_info) image_info can be an empty dict for inference
  - `npy_file` – path to a npy or npz file.
  - `npy_dir` – path to a npy or npz dir, assumes same shape to all the items.
- `data_count` (`optional, int`) – optional argument to limit the number of elements for analysis
- `batch_size` (`optional, int`) – batch size for analysis
- `work_dir` (`optional, str`) – If not None, dump quantization debug outputs to this directory.
- `analyze_mode` (`optional, str`) – selects the analyzing mode that will run simple or advanced.

`optimize` (*calib_data*, *data_type=CalibrationDataType.auto*, *work_dir=None*)

Apply optimizations to the model:

- Modify the network layers.
- Quantize model's params, using optional pre-process and post-process algorithm.

Parameters

- `calib_data` – Calibration data for Equalization and quantization process. . Type depends on the `data_type` parameter.
- `data_type` (`CalibrationDataType`) – calib_data's data type, based on enum values:
  - `auto` – Automatically detection.
  - `np_array` – `numpy.ndarray`, or dictionary with input layer names as keys, and values types of `numpy.ndarray`.
  - `dataset` – `tensorflow.data.Dataset` object with valid signature. signature should be either ((h, w, c), image_info) or ({'input_layer1': (h1, w1, c1), 'input_layer2': (h2, w2, c2)}, image_info) image_info can be an empty dict for the quantization
  - `npy_file` – path to a npy or npz file
  - `npy_dir` – path to a npy or npz dir, assumes same shape to all the items
- `work_dir` (`optional, str`) – If not None, dump quantization debug outputs to this directory.

`get_hailo_runtime_model()`
This API is preview.

Generate model allowing to run the full ONNX graph using ONNX runtime including the parts that are offloaded to the Hailo-8 (between the start and end nodes) and the parts that are not.

`get_detected_nms_config(`*meta_arch*, *config_path=None*`)`
Get the detected NMS config file: anchors detected automatically from the model's postprocess, and default values corresponding to the meta architecture specified.

Parameters

- `meta_arch` (`NMSMetaArchitectures`) – Meta architecture of the NMS post process.
- `config_path` (`string, optional`) – Path to save the generated config file. Defaults to '{meta_arch}_nms_config.json'.

6.2. hailo_sdk_client.exposed_definitions

This module contains enums used by several SDK APIs.

`class hailo_sdk_client.exposed_definitions.JoinAction(`*value*`)`
Bases: `enum.Enum`

Special actions to perform when joining models.

See also:

The `join()` API uses this enum.

`NONE = 'none'`
join the graphs without any connection between them.

`AUTO_JOIN_INPUTS = 'auto_join_inputs'`
Automatically detect inputs for both graphs, and join them into one. Only works when both networks have a single input of the same shape.

`AUTO_CHAIN_NETWORKS = 'auto_chain_networks'`
Automatically detect output of this model, and input of the other model, and connect them. Only works when this model has a single output, and the other model has a single input, of the same shape.

`CUSTOM = 'custom'`
Supply a custom dictionary `join_action_info`, which specifies which nodes from this model need to be connected to which of the nodes in the other graph. If keys and values are inputs, we join the inputs. If keys are outputs, and values are inputs, we chain the networks as described in the dictionary.

`class hailo_sdk_client.exposed_definitions.JoinOutputLayersOrder(`*value*`)`
Bases: `enum.Enum`

Enum-like class to determine the output order of a model after joining with another model.

`NEW_OUTPUTS_LAST = 'new_outputs_last'`
First are the outputs of this model who remained outputs, then outputs of the other model. The order in each sub-list is equal to the original order.

`NEW_OUTPUTS_FIRST = 'new_outputs_first'`
First are the outputs of the other model, then outputs of this model who remained outputs. The order in each sub-list is equal to the original order.

`NEW_OUTPUTS_IN_PLACE = 'new_outputs_in_place'`
If the models are chained, the outputs of the other model are inserted, in their original order, to the output list of this model instead of the first output which is no longer an output. If the models are joined by inputs, the other model's outputs are added last.

class hailo_sdk_client.exposed_definitions.NNFramework(*value*)
Bases: enum.Enum

Enum-like class for different supported neural network frameworks.

TENSORFLOW = 'tf'
Tensorflow 1.x

TENSORFLOW2 = 'tf2'
Tensorflow 2.x

TENSORFLOW_LITE = 'tflite'
Tensorflow Lite

ONNX = 'onnx'
ONNX

class hailo_sdk_client.exposed_definitions.NMSMetaArchitectures(*value*)
Bases: enum.Enum

Network meta architectures to which on-chip/ on-host post-processing can be added.

SSD = 'ssd'
Single Shot Detection meta architecture.

CENTERNET = 'centernet'
Centernet meta architecture

YOLOV5 = 'yolov5'
Yolov5 meta architecture

YOLOX = 'yolox'
Yolox meta architecture

YOLOV5_SEG = 'yolov5_seg'
Yolov5 seg meta architecture

class hailo_sdk_client.exposed_definitions.States(*value*)
Bases: enum.Enum

Enum-like class with all the ClientRunner states.

UNINITIALIZED = 'uninitialized'
Uninitialized state when generating a new ClientRunner

ORIGINAL_MODEL = 'original_model'
ClientRunner state after setting the original model path (ONNX/TF model)

HAILO_MODEL = 'hailo_model'
ClientRunner state after parsing (calling the translate_onnx_model()/translate_tf_model() API)

FP_OPTIMIZED_MODEL = 'fp_optimized_model'
ClientRunner state after calling the optimize_full_precision() API. This state includes all the full precision optimization such as model modification commands.

QUANTIZED_MODEL = 'quantized_model'
ClientRunner state after calling the optimize() API. This state includes quantized weights.

COMPILED_MODEL = 'compiled_model'
ClientRunner state after compilation (calling the compile() API).

class hailo_sdk_client.exposed_definitions.InferenceContext(*value*)
Bases: enum.Enum

Enum-like class with all the possible inference contexts modes

SDK_NATIVE = 'sdk_native'
SDK_NATIVE context is for inference of the original model (without any modification).

`SDK_FP_OPTIMIZED = 'sdk_fp_optimized'`
SDK_FP_OPTIMIZED context includes all model modification in floating-point (such as normalization, nms, and so on).

`SDK_QUANTIZED = 'sdk_quantized'`
SDK_QUANTIZED context is for inference of the quantized model. Used to measure degradation caused by quantization.

`SDK_HAILO_HW = 'sdk_hailo_hw'`
SDK_HAILO_HW inference context to run on the Hailo-HW.

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.translate_rgb_dataset(`*rgb_dataset*, ...`)`
Translate a given RGB format images dataset to YUV or BGR format images. This function is useful when the model expects YUV or BGR 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.
- `color_type` (`ColorType`) – type of color to translate the data to. Defaults to yuv.

7. Common API Reference

7.1. hailo_sdk_common.model_params.model_params

class hailo_sdk_common.model_params.model_params.ModelParams(*params*, ...)
Bases: object

Dict-like class that contains all parameters used by a model such as weights, biases, etc.

7.2. hailo_sdk_common.profiler.profiler_common

class hailo_sdk_common.profiler.profiler_common.ProfilerModes(*value*)
Bases: enum.Enum

Enum-like class for different execution modes of the profiler.

PRE_PLACEMENT = 'pre_placement'
Profiling before placement is made.

POST_PLACEMENT = 'post_placement'
Profiling after placement is made.

7.3. hailo_sdk_common.hailo_nn.hailo_nn

class hailo_sdk_common.hailo_nn.hailo_nn.HailoNN(*network_name=None*, *stage=None*, ...)
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. Currently doesn't support changing number of features.

`static from_fp` (*fp*)
Get Hailo model from a file.

`static from_hn` (*hn_json*)
Get Hailo model from HN raw JSON data.

`static from_parsed_hn` (*hn_json*, *validate=True*)
Get Hailo model from HN dictionary.

Bibliography

[Meller2019] Eldad Meller, Alexander Finkelstein, Uri Almog and Mark Grobman. "Same, same but different: Recovering neural network quantization error through weight factorization." International Conference on Machine Learning, 2019. http://proceedings.mlr.press/v97/meller19a/meller19a.pdf

[Finkelstein2019] Alexander Finkelstein, Uri Almog and Mark Grobman. "Fighting quantization bias with bias." Conference on Computer Vision and Pattern Recognition Workshops, 2019. https://arxiv.org/pdf/1906.03193.pdf

[McKinstry2019] Jeffrey McKinstry, Steven Esser, Rathinakumar Appuswamy, Deepika Bablani, John Arthur, Izzet Yildiz and Dharmendra Modha. "Discovering Low-Precision Networks Close to Full-Precision Networks for Efficient Embedded Inference." Conference on Neural Information Processing Systems, 2019. https://www.emc2-ai.org/assets/docs/neurips-19/emc2-neurips19-paper-11.pdf

[Nagel2020] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos and Tijmen Blankevoort. "Up or Down? Adaptive Rounding for Post-Training Quantization." International Conference on Machine Learning, 2020. https://arxiv.org/pdf/2004.10568.pdf

[Vosco2021] Niv Vosco, Alon Shenkler and Mark Grobman. "Tiled Squeeze-and-Excite: Channel Attention With Local Spatial Context." International Conference on Computer Vision Workshops, 2021. https://openaccess.thecvf.com/content/ICCV2021W/NeurArch/papers/Vosco_Tiled_Squeeze-and-Excite_Channel_Attention_With_Local_Spatial_Context_ICCVW_2021_paper.pdf

Python Module Index

h

