Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ Neural Network (NN) Library

User Guide

80-VB419-110 Rev. B

September 24, 2018

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Tables

1 Introduction

1.1 Purpose

The Qualcomm® Hexagon™ Neural Network (NN) library is an offloadable NN inference framework. It includes a number of operations that closely match the behavior of operations in popular NN design environments.

This document provides general guidance on how the Hexagon NN library works internally, and how it can be used to run graphs designed in high-level NN frameworks such as TensorFlow and Caffe on the Hexagon DSP.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `cp armcc armcpp`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Overview

The Hexagon NN uses the same general approach as TensorFlow to represent neural networks. In particular, both frameworks share the same concepts of graph, nodes, and tensors. Both frameworks also handle quantization of the datapath in the same way.

If you are a new Hexagon NN user, we recommend that you read about TensorFlow and follow some of the available tutorials so you can familiarize yourself with the basic concepts that govern TensorFlow graphs.

We also recommend that you review the examples included with the Hexagon SDK so you can familiarize yourself with compiling code for the DSP and CPU, and interfacing the DSP from the CPU using FastRPC.

3 Library source and compilation

This section explains how to obtain the Hexagon NN library and build it from source.

3.1 Obtain the Hexagon NN source

The Hexagon NN library is an open source project available from Code Aurora at https://source.codeaurora.org/quic/hexagon_nn/nnlib.

Starting with Hexagon SDK version 3.4, the Hexagon SDK ships with the latest version of the Hexagon NN library available at the time of release.

The Hexagon NN library is located under the `${HEXAGON_SDK_ROOT}/libs/hexagon_nn` folder, which contains two identical copies of the library:

- A `<Hexagon NN version number>` folder to be modified when changes to the library implementation are required,
- A backup in a `<Hexagon NN version number>_backup` folder to be used as a reference in case you made changes to the `<Hexagon NN version number>` folder that you do not want to keep.

3.2 Build the dynamic library

To build the dynamic `.so` library for your device, ensure that you first run `setup_sdk_env.source` from the root of your Hexagon SDK folder, and then go to your Hexagon NN library folder. From there, enter the following command:

```
make tree VERBOSE=1 V65=1 V=hexagon_ReleaseG_dynamic_toolv82_v65
```

NOTE: You might need to adjust the exact build options for your own requirements.

Table 3-1 Compilation build options

Option	Intent
VERBOSE	Set to **1** to display all build commands.
V65 or V66	Set to **1** to use V65- or V66-specific optimizations such as VTCM and scatter-gather. If neither V65 nor V66 are set, HVX instructions are still used, but the code does not leverage optimizations only available in V65 and more recent versions. When no flag is specified, defaults to a V60-compatible implementation.
V	Specify which version of the Hexagon tools to use and which DSP target to compile for. For example, `hexagon_ReleaseG_dynamic_toolv82_v65` uses Hexagon tools 8.2 and builds for V65-based targets.

Option	Intent
tree_clean	Clean the tree for the specified flavor.

For more information and build command examples, see also `${HEXAGON_SDK_ROOT}/libs/README.HOW_TO_BUILD`.

3.3 Define the Android Hexagon NN interfaces

To use Hexagon NN from the Android side, run the following command.

```
make tree V=android_ReleaseG
```

This command ensures that the required Hexagon NN artifacts are placed in the Android ship directory of the Hexagon NN project so that they may be picked up by any project that is dependent upon them.

4 Examples

The `${HEXAGON_SDK_ROOT}/examples/hexagon_nn` folder includes several utilities that demonstrate how to use the Hexagon NN library in your own projects, and how to run the graphs you create on target and on the simulator.

Table 4-1 Hexagon NN directory structure

File or folder name	Intent
`environments/`	Python virtual environment requirements needed to run the scripts
`README.md`	Setup instructions
`scripts/`	Python utility scripts
`setup_hexagon_nn.cmd`	Windows script for setting the environment (Windows SDK install only)
`setup_hexagon_nn.source`	Linux script for setting the environment (Linux SDK install only)
`tutorials/`	Tutorials that show how to build and run graphs with the Hexagon NN

4.1 Environment setup

As mentioned in Section 3.2, you must run `setup_sdk_env.source` from the root of your Hexagon SDK folder before you can use the Hexagon SDK tools to build your projects.

The Hexagon NN example library folder contains Windows and Linux scripts that automate setup operations required to build and use the NN library:

- Set the environment variable, `HEXAGON_NN`, to the non-backup Hexagon NN library source project.

  If the script is invoked with no argument, `HEXAGON_NN` will point to the default SDK Hexagon NN installation.

- If the location specified by `${HEXAGON_NN}` is empty, download the latest version available from Code Aurora (CAF).

  If you maintain your own repository to develop the Hexagon NN library, you can provide this repository as a second argument to the setup script. A third argument allows you to provide a patch to apply to the master branch of the repository from which you retrieved the Hexagon NN library.

A Python virtualenv simplifies the use of the scripts found under the `scripts` folder. Refer to `README.md` for detailed instructions on how to set up your virtual environment for the first time using the list of Python packages included in the `requirements` folder. Activate and deactivate the environment as needed.

NOTE: Some versions of Python installed by anaconda conflict with the use of virtual environments and the instructions provided in the `README.md` file. For example, if using virtual environments fail for python 3, try using python 2 instead. All scripts provided with the Hexagon NN library have been tested with python 2 and 3.

4.2 Tutorials

The `tutorials` directory contains several tutorials. They are ordered so as to introduce you progressively to Hexagon NN and show how to build graphs, leverage existing optimized operations, add new operations, or build full implementations from existing TensorFlow and Caffe graphs.

For more information about any tutorial, refer to the `README.md` files in the parent directory where each tutorial is located, as well as the header of each source file.

Table 4-2 Hexagon NN tutorials

Tutorial	Intent
001-nop	Introduces the most basic Hexagon NN APIs needed to build a graph, and explains how to build and run the graph. The graph contains only a simple NOP operation.
002-add	Introduces the fastRPC protocol required to get the ARM CPU to speak to the DSP. The graph contains a simple float ADD operation.
003-quantized-add	Introduces quantized operations to perform fixed-point arithmetic. The graph contains a simple fixed-point ADD operation surrounded by quantization and dequantization operations.
004-xor-graph	Shows how to debug and validate nodes in a graph. The graph contains a trivial neural network that XORs two input values.
005-addconst	Shows how to create a new operation not already implemented in the Hexagon NN library and how to use it as a node in a graph.
006-threads	Shows how to create a multithreaded Hexagon NN implementation.
007-dumpgraph	Shows how to generate a fixed-point Hexagon NN implementation from a publicly available floating-point TensorFlow model.
008-caffe-to-hexagon-nn	Shows how to generate a fixed-point Hexagon NN implementation from a publicly available floating-point Caffe model.

4.3 Graph_app

The folder `${HEXAGON_NN}/test` includes an application called graph_app. The main program of this application parses command-line arguments, which control execution of a graph, including iteration counts, performance monitoring, and loading inference data from a file. Construction of the graph is performed in an `init_graph()` function, which is incorporated at link-time.

With this approach, the user is only responsible for defining a graph by appending nodes in an `init_graph()` function and linking this code into a graph_app executable. This approach allows the user to focus on the correct and efficient implementation of the graph without needing to write the surrounding control code.

For more information on graph_app, refer to tutorials 007 and 008.

4.4 Scripts

The script directory contains utilities for using the Hexagon NN library more efficiently.

All these scripts are provided *as is*. They are used within Qualcomm to speed up specific tasks but are not thoroughly tested. Despite having many limitations, they are valuable tools to use the Hexagon NN library more efficiently and understand the general workflow of converting a TensorFlow or Caffe implementation into a Hexagon NN graph.

Table 4-3 Hexagon NN scripts

Script	Intent
caffe_to_hexagon_nn	Turns a floating-point Caffe model into a fixed-point Hexagon NN implementation. Thus far, the script has been tested only with VGG 16, and it will probably require additional changes to support other graphs. For details on how to use the script and expand it to support other graphs, see tutorial 008 .
dat2img	Converts a raw RGB data image to a PNG file.
dumpgraph	Converts a TensorFlow model graph to a Hexagon NN implementation with the optional test wrapper. The script also runs TensorFlow automatically to generate reference test data. For more details, see tutorial 007.
dump_for_tensorboard	Uses a TensorFlow model to generate a file that can be fed into the TensorBoard to display the corresponding graph.
imagedump	Converts image data from JPEG to RGB pixel data.
memdbg	Reports memory allocation and deallocation to detect possible memory leaks.

5 APIs

5.1 Lifetime of a graph

A graph passes through several phases:

- Initialization of the empty graph
- Appending nodes to the graph
- Preparing for execution
- Executing one or more times
- Teardown

5.2 Graph initialization and debugging APIs

The Hexagon NN library allows a program to build a graph (or subgraph), and then execute it multiple times on the DSP. The library is implemented as plain C, not C++.

A graph preparation pass is required between graph construction and graph execution, which allows for the graph to be prepared for optimized execution.

5.2.1 hexagon_nn_version()

Sets the memory at *version to an API version.

Failure of this function typically means that the `hexagon_nn` library could not be loaded for some reason.

Prototype

```
int32_t hexagon_nn_version(int *version)
```

Parameters

`in`	version	Pointer to be set to the API version.

Returns

0 on success; otherwise, failure.

5.2.2 hexagon_nn_config()

Call this function before creating a graph. It allows the environment on the DSP to configure some settings.

While this function has no program effect, it might help performance or the ability to allocate large amounts of memory.

This function returns 0.

Prototype

```
int32_t hexagon_nn_config();
```

5.2.3 hexagon_nn_init()

Creates a new graph and returns an identifier to refer to the new graph. After a graph is initialized, nodes can be added to it.

The returned graph is empty and cannot be executed until all nodes have been added and the graph is finalized with hexagon_nn_prepare(). Multiple graphs can be created and can be kept alive in the DSP environment simultaneously.

Prototype

```
nn_id hexagon_nn_init();
```

Returns

NN_ID_INVALID – for an error.

5.2.4 hexagon_nn_snpprint()

Pretty-prints the graph into a specified buffer.

Prototype

```
void hexagon_nn_snpprint(nn_id id, char *buf, uint32_t length);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
out	buf	Pointer to the buffer.
in	length	Output buffer length.

Returns

None.

5.2.5 hexagon_nn_getlog()

Copies log messages into a specified buffer, and resets the internal log buffer.

Prototype

```
void hexagon_nn_getlog(nn_id id, char *buf, uint32_t length);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
out	buf	Pointer to the buffer.
in	length	Output buffer length.

Returns

None.

5.2.6 hexagon_nn_set_debug_level()

Changes the debug verbosity level for messages.

Prototype

```
void hexagon_nn_set_debug_level(nn_id id, int level);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.

Returns

None.

5.3 Node data structure

Nodes connect to each other using data structures with the following features:

- Four dimensions: batches, height, width, and depth
- A pointer to data

Scalar values can be defined as BHWD=1,1,1,1.

Shapes can be defined by setting data to NULL.

The actual BHWD scalar values are computed by each node during execution, and only OP_INPUT and constant nodes have fixed size. During node creation, the max_size of each output is required for reserving output buffers. Node inputs do not need to reserve memory because they merely reference the output data of an earlier node.

The following types are used when specifying node input and output.

5.3.1 hexagon_nn_input

```
typedef struct {
        uint32_t src_id;       // Node identifier
        uint32_t output_idx;   // Which output to use
} hexagon_nn_input;
```

5.3.2 hexagon_nn_output

```
typedef struct {
        uint32_t max_size;
        uint32_t reserved;
} hexagon_nn_output;
```

5.4 Node insertion

Nodes can be added to a graph after `hexagon_nn_init()` and before `hexagon_nn_prepare()`.

Nodes are executed in the order in which they are added, so the creation order is important.

All operations create memory to capture all the information; the data does not need to persist on the caller side.

Two functions are used to append nodes to a graph.

5.4.1 hexagon_nn_append_const_node()

Adds constant nodes to a graph.

Constant nodes produce a single output that can be connected to one graph node input.

Unique node_ids are required for referencing nodes when connecting the graph (for example, specifying which outputs of earlier nodes will be used as inputs to particular subsequent nodes). Node_ids are selected by the caller, but `node_id=0` and `node_id>0xF0000000` are reserved. Node_ids must be unique.

NOTE: On SDM835 and older targets, `hexagon_nn_append_const_node()` will not work properly for arrays larger than 32 MB. Instead, use `hexagon_nn_append_empty_const_node_large_array()`, which expects the same arguments.

Prototype

```
int hexagon_nn_append_const_node(
                nn_id id,
                uint32_t node_id,
                uint32_t batches,
                uint32_t height,
                uint32_t width,
                uint32_t depth,
                const char *data,
                uint32_t data_length);
```

Parameters

in	Id	Unique identifier of the Hexagon NN graph.
in	node_id	Unique identifier of the node.
in	batches	Batch dimension.
in	height	Height dimension.
in	width	Width dimension.
in	depth	Depth dimension.
in	data	Node data pointer.
in	data_length	Length of data to copy.

Returns

0 on success; otherwise, failure.

5.4.2 hexagon_nn_append_node()

Adds an ordinary (non-constant) node to the graph.

Non-constant nodes can have zero or more inputs and zero or more outputs.

An input is described as a source node ID as well as an output index to refer to which one of several outputs a node may have.

An output is described with a maximum size. The true size of an output can be computed dynamically, but the caller must define the maximum amount of data storage required by the output during node creation.

Prototype

```
int hexagon_nn_append_node(
                nn_id id,
                uint32_t node_id,
                uint32_t operation,
                padding_type padding,
                const hexagon_nn_input *inputs,
                uint32_t n_inputs,
                const hexagon_nn_output *outputs,
                uint32_t n_outputs);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
in	node_id	Unique identifier of the node.
in	operation	Node operation type.
in	padding	Padding type.
in	inputs	Input list.
in	n_inputs	Number of input branches.
in	outputs	Output list.
in	n_outputs	Number of output branches.

Returns

0 on success; otherwise, failure.

5.4.3 padding_type

Operations can also have a padding option. The following types of padding are defined:

```
typedef enum {
        NN_PAD_NA,
        NN_PAD_SAME,
        NN_PAD_VALID,
        NN_PAD_MIRROR_REFLECT,
        NN_PAD_MIRROR_SYMMETRIC,
        NN_PAD_SAME_CAFFE
} padding_type;
```

If padding does not apply, choose `NN_PAD_NA`. For nodes that can have padding, the caller can choose `SAME` or `VALID` padding:

- `SAME` padding means the output size should be roughly the same as the input.
- `VALID` padding means that you should only use input values that are valid, which results in a smaller output size for larger filter windows.

5.5 Graph execution

5.5.1 hexagon_nn_prepare()

Prepares a network for execution.

This function is required after all the nodes have been appended and before execution. This call provides a hook where memory can be allocated, data can be rearranged, inputs and outputs can be linked up, and things in the graph can be optimized.

Once a network has been prepared, it can no longer be appended to, but it can be executed.

Prototype

```
int hexagon_nn_prepare(nn_id id);
```

Parameters

in	Id	Unique identifier of the Hexagon NN graph.

Returns

0 on success; otherwise, nonzero.

5.5.2 hexagon_nn_execute()

Executes a network, with provided input data and returning output data.

Execution will fail if the network has not been prepared.

Input is provided to the INPUT node, and output is returned from the OUTPUT node.

Prototype

```
int hexagon_nn_execute(nn_id id,
         uint32_t batches_in,
         uint32_t height_in,
         uint32_t width_in,
         uint32_t depth_in,
         const void *data_in,
         uint32_t data_in_len,
         uint32_t *batches_out,
         uint32_t *height_out,
         uint32_t *width_out,
         uint32_t *depth_out,
         void *data_out,
         uint32_t max_data_out_len,
         uint32_t *data_len_out);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
in	batches_in	Input batch dimension.
in	height_in	Input height dimension.
in	width_in	Input width dimension.
in	depth_in	Input depth dimension.
in	data_in	Input list.
in	data_in_len	Number of input branches.
out	batches_out	Output list.
out	height_out	Output batch dimension.
out	width_out	Output width dimension.
out	depth_out	Output depth dimension.
out	data_out	Output list.
in	max_data_out_len	Maximum length of output buffer.
out	data_out_len	Actual length of output buffer.

Returns

0 on success; otherwise, nonzero.

5.5.3 hexagon_nn_teardown()

Tears down and frees an NN graph. This can be done at any time after `hexagon_nn_init()`. After this function has been invoked, the `nn_id id` is invalid.

Prototype

```
int hexagon_nn_teardown(nn_id id);
```

Parameters

`in`	id	Unique identifier of the Hexagon NN graph.

Returns

0 on success; otherwise, nonzero.

5.6 Graph profiling

5.6.1 hexagon_nn_get_perfinfo()

Gets performance information for the nodes. This function fills the array of performance information structures.

Prototype

```
int hexagon_nn_get_perfinfo(
         nn_id id,
         hexagon_nn_perfinfo *info_out,
         uint32_t info_out_max_len,
         uint32_t *n_items_returned);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
out	Info_out	Pointer to the performance information structure.
in	info_out_max_len	Length of the performance information structure.
out	n_items_returned	Number of nodes in the graph.

Performance information structure

```
struct perfinfo {
        unsigned long node_id;
        unsigned long executions;
        unsigned long counter_lo;
        unsigned long counter_hi;
};
```

Returns

0 on success; otherwise, nonzero.

5.6.2 hexagon_nn_reset_perfinfo()

Resets the per-node performance counters, and changes the performance monitor unit event ID to the specified ID.

- Event ID 0 counts cycles.
- Events ID 1 and 2 are defined by the operation (and typically unused).
- All other event IDs match the hardware PMU event ID.

Prototype

```
int hexagon_nn_reset_perfinfo(
       nn_id id,
       uint32_t event);
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
in	event	ID of the event to be reset.

Returns

0 on success; otherwise, nonzero.

5.6.3 hexagon_nn_last_execution_cycles()

Returns the total number of processor cycles during execution of the most recent `hexagon_nn_execute()` call.

Prototype

```
int hexagon_nn_last_execution_cycles(
         nn_id id,
         uint32_t *cycles_lo_out,
         uint32_t *cycles_hi_out)
```

Parameters

in	id	Unique identifier of the Hexagon NN graph.
out	cycles_lo_out	Lowest 32 bits of the processor cycle counter.
out	cycles_hi_out	Upper 32 bits of the processor cycle counter.

Returns

0 on success; otherwise, nonzero.

5.7 Clock and power setting

Some API calls are available to control the power mode of the device and achieve different trades between power consumption and speed.

5.7.1 set_powersave_level()

Provides a simple parameter between 0 and 255 to control the power saving mode.

A level of 255 indicates that preference should be given to minimizing power consumption. A level of 0 indicates that preference should be given to executing as fast as possible.

Prototype

```
int hexagon_nn_set_powersave_level(unsigned int level)
```

Parameters

in	level	Value of the power level control.

Returns

0 on success; otherwise, nonzero.

5.7.2 set_powersave_details()

Provides an alternate to `set_powersave_level()`, whereby more details can be specified for power saving mode.

Prototype

```
int set_powersave_details(
                corner_type corner,
                dcvs_type dcvs,
                unsigned long latency)
```

Parameters

`in`	corner	Specified voltage corner. `NN_CORNER_RELEASE` – Revokes the power vote that was made.
`in`	dcvs	Specifies whether to use the default DCVS policy for the requested corner, or to override the policy with DCVS enabled/disabled.
`in`	latency	Minimum tolerable µs wakeup latency for the DSP. When set to 0, uses the default latency for the requested corner.

Returns

0 on success; otherwise, nonzero.

6 Data and operations

6.1 Data

As previously mentioned, Hexagon NN uses the same concepts and naming conventions as TensorFlow to describe neural networks.

In particular, inputs and outputs are called *tensors*. Tensors have data and a shape. The shape is 4-D, with lower dimensionality being represented with the size 1 in that dimension. The dimensions are sometimes named *batches*, *height*, *width*, and *depth*.

A scalar value is represented by a tensor of shape `1,1,1,1`.

Sometimes only the shape of a tensor is used. For example, the stride for convolution or filter size for pooling are represented by tensors, although the data of the tensor is ignored and only the shape is used.

6.2 Native operations

Hexagon NN supports a large and growing number of operations.

All operations are documented under `${HEXAGON_NN}/docs/ops.txt`.

If the documentation does not provide sufficient information on what a specific operation does or how it is implemented, you can access the source code directly for each operation under `${HEXAGON_NN}/hexagon/ops/src`.

6.3 Super nodes

During the initialization phase of a graph, the Hexagon NN library optimizes the graph by combining common node patterns into “super nodes”. For example, a node sequence made of a convolution, bias add, range shrink, relu, and pool might turn into one single node so that values are computed on the fly without being written to and read from arrays between each single node.

For more information on what super nodes are created during the graph optimization phase, increase the value of debug_level to return information on which nodes are combined into super nodes.

NOTE: Executing super nodes will bypass the code implementing a specific operation.

If you debug an implementation and do not want a specific node implementation to be bypassed, you can force the execution of the reference version of that operation if it is available.

For example, if a sequence of nodes, including an `OP_QuantizedConv2d_8x8to32` operation, is mapped into a supernode, you can use the `OP_QuantizedConv2d_8x8to32_ref` operation instead. This operation will prevent the supernode from executing.

For the complete list of operation reference implementations available, see file, `${HEXAGON_NN}/interface/ops.def`.

6.4 Custom operations

On occasion, you might need to add a layer with an operation that is not already present in the Hexagon NN. These operations must be added to the Hexagon NN library directly. For step-by-step instructions on how to add a custom operation, see tutorial 005.

If you believe Hexagon NN is missing an operation that is critical for your project, you can contact Qualcomm to inquire whether such an operation can be added to the Hexagon NN, or seek guidance on optimization.

7 Quantization

The Hexagon NN follows the same approach as TensorFlow for supporting fixed-point implementations of graphs. Before you read the rest of this chapter, it might be beneficial if you refer to the online TensorFlow documentation on quantization.

7.1 Coefficient quantization

7.1.1 Data types

The Hexagon NN currently supports two fixed-point formats for representing data: 8-bit and 32-bit. Currently, the 16-bit format is not supported.

Using the 32-bit data type allows for greater accuracy at the expense of large coefficient tables and occasionally longer processing speed.

A graph can use a combination of 8-bit and 32-bit data types. For example, you can use 8-bit coefficients for performing convolution operations (which typically consume very large coefficient arrays), but still use 32-bit coefficients when adding biases (because bias arrays are typically much smaller).

Fixed-point input and output tensors are composed of three sub-tensors. Two of these tensors are floating-point scalars holding the minimum and maximum values that the fixed-point symbols represent. The minimum value should not be strictly positive because zero must be represented correctly. The minimum and maximum values do not need to be symmetric around 0.0, though it is preferable to do so as explained in the next section.

The primary tensor contains the 8-bit or 32-bit integer symbols. The 8-bit symbols are maintained as unsigned numbers (fixed-point symbol 0 represents the minimum value, and fixed-point symbol 255 represents the maximum value) while 32-bit symbols are maintained as signed numbers.

For example, if `min=-1.12345`, `max=+1.2345`, and the tensor is made of 8-bit values, `0xc0=128+64=192` represents value `1.2345*.5=.61725`. If the tensor is made of 32-bit values, then `.61725` corresponds to fixed-point value `0x8000000=1<<30`.

7.1.2 Zero point

Because of the large number of multiply-accumulate values and the large number of very small coefficient values, it is critical to minimize the error made on rounding small values.To accomplish this, no rounding error should be made when representing the value zero, sometimes referred as the *zero point*. Very small positive and negative values should round to perfect-zero and not to some biased-positive or biased-negative value.

In addition, the library performs more efficiently on targets with efficient support for signed multiplication if the coefficients represent a symmetric floating-point range.

Therefore, for all targets, we recommend the following:

- When using the unsigned 8-bit format, represent 0.0 as 128
- When using the signed 32-bit format, represent 0.0 as 0

For example, with 8-bit coefficients, this condition will be met as long as:

```
min/127=-max/128
```

7.2 Activation quantization

To represent data as fixed-point throughout the graph, the following options are available.

7.2.1 Run-time activation quantization

The first approach consists of letting the Hexagon NN implementation automatically track the data range and normalize data after each node accordingly.

This approach might result in the graph occasionally rerunning some of the nodes a second time to adjust the mininum and maximum values when the values previously selected were not large enough to cover the full data range.

Because the Hexagon NN implementation only increases the minimum and maximum absolute values when adjusting them, nodes only need to be rerun for the first few inferences. After a few inferences, the minimum and maximum values of each tensor become stable and are sufficient to represent data without any saturation.

This approach is the simplest approach because it does not require you to characterize the data range throughout the graph.

However, this approach can lead to some inferences running more slowly because individual nodes might need to run twice in a row. This approach can also lead to poorer accuracy because the computed min-max range might be larger than necessary and lead to a large quantization step. Therefore, we recommend that your final implementation uses the method described in Section 7.2.2 for activation quantization.

7.2.2 Compile-time activation quantization

The second approach consists of providing the minimum and maximum values of each tensor when defining a graph. This approach ensures that the Hexagon NN implementation executes each node only once.

To follow that approach, use a `Requantize_32to8` operation instead of a `QuantizeDownAndShrinkRange_32to8` operation when limiting the activation range to 8 bits.

NOTE: You must fully characterize the data flow before defining the graph.

8 Debugging

Debugging a graph can be challenging because data generated out of each node is difficult to interpret. The following are some of the approaches you can use to debug your code.

- First, generate a functional floating-point implementation.

  With large neural networks, even the smallest quantization errors can accumulate often enough to produce an incorrect output. Therefore, first verify that a graph has been correctly defined by ensuring that the floating point implementation of that graph with the Hexagon NN produces the expected output.

  After verifying that the graph has been defined correctly, use quantization and dequantization operations to progressively convert the graph from floating-point to fixed-point data as necessary.

NOTE: Full floating-point implementations can take up to a few minutes to execute.

- Check the nodes

  Use the `op_check` operation to compare two inputs.

  For example, generate a reference implementation and store the output of a given node in a file. When defining the graph, define a constant node to generate the reference data that was captured. Then compare the data to an actual tensor within the graph using an `op_check` operation.

- Print nodes

  Print operations are available if you need to print a tensor. Various formats are supported: quantized 8-bit, quantized 32-bit, and float.

- Use reference operations

  If you suspect a bug in the Hexagon NN implementation, confirm it by using the reference implementation of a given operation.

  This approach eliminates any node optimization or supernode implementations that might introduce bugs that have not been reported and documented yet.

- Use debug messages

  The Hexagon NN will generate progressively more debug information as you increase the `debug_level` value using `hexagon_nn_set_debug_level()`.

  To understand what each debug message means, you must become familiar with the source of each operation. You can also edit the source and manually add your own debugging messages to any operation.

A FAQs

How do I get the latest version of the Hexagon NN source code?

As explained in Section 3.1, Hexagon NN is available both from Code Aurora and when downloading the Hexagon SDK.

Where do I find training material for using Hexagon NN?

The Hexagon SDK comes installed with a set of Hexagon NN tutorials (available under `{HEXAGON_SDK_ROOT}/examples/hexagon_nn`).

Why is my Hexagon NN implementation so slow?

Check the following:

- Profile your application as explained in Section 5.6.
  - Are some layers running much slower than you expect based on the operations they need to perform?
  - Is your implementation running at the highest clock speed available?
- Look for floating-point operations that can be turned into fixed-point operations.
- Look at successions of convolutions, bias-add, and ReLU, and confirm they are turned into one supernode during `prepare()`.

  You can do so by adjusting the debug level as shown in Section 5.2.6 and looking at the logs.
- Look for any dequantize/requantize operations that might not be required.
- Confirm that all nodes are actually required.For example, do not flatten an array that is already flattened.
- Check whether any of the nodes you are using are unoptimized.

  Unoptimized nodes are likely single-threaded, using floating-point operations, or not calling an assembly routine.

Should I run my code on the aDSP or cDSP?

Check the online SDK documentation or the *Qualcomm Hexagon DSP User Guide* (80-VB419-108) to understand which device supports HVX on which DSP.

Why do I get “Failed to disable DSP DCVS”

Occasionally this error is caused by using a device that has not been signed.

What succession of operations turn into supernodes?

Currently, successions of convolution, bias-add, and ReLU are turned into a supernode.

Is my supernode introducing a bug?
Can I prevent supernode conversion?

You can add `print()` or `nop()` nodes between the convolution and bias-add operations to break the pattern and disable the supernode formation.

Can I ensure supernode conversion?

You can explicitly create supernodes just like any other node.

Which operations are fast or slow?

Any operation that uses assembly to perform its work is expected to be fully optimized.

Are operations optimized to work on different tensor and filter sizes?

The most time-critical operations (supernodes) have multiple optimized assembly routines to handle various tensor and filter size combinations.

Which operations and sizes are supported?

Supported operations can be found in the `docs/ops.txt` file.

What should I do if I find a broken operation?

Report any bugs to your CE.

How can I figure out which parts of my graph are slow?
How can I speed up my graph?
What can I do if some important operations are too slow?

As explained in Section 5.6, PMU events allow you to observe the time spent in each node. When you identify nodes that ought to execute faster, look at the source code for the corresponding operation. If you identify some inefficiency in the code, you can then modify the existing operation or create a new operation of your own.

For detailed instructions on creating new operations, see tutorial 005 (available under `{HEXAGON_SDK_ROOT}/examples/hexagon_nn`).