# AI8X Model Training and Quantization

# AI8X Network Loader and RTL Simulation Generator

*April 9, 2020*

*Open the `.md` version of this file in a markdown enabled viewer, for example Typora (http://typora.io). See https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet for a description of Markdown. A PDF copy of this file is available in the repository.*

This software consists of two related projects:

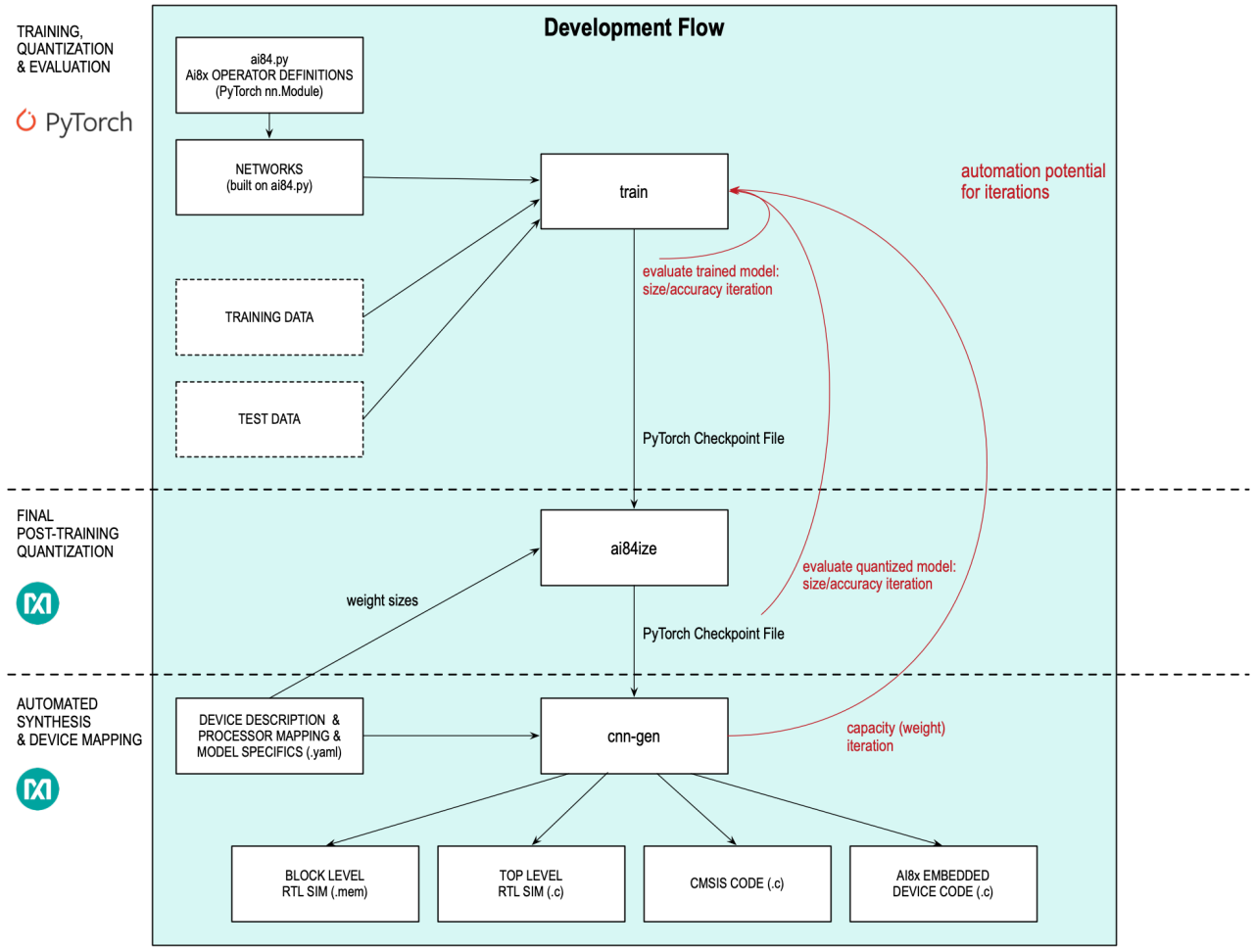
1. AI8X Model Training and Quantization
2. AI8X Network Loader and RTL Simulation Generator

---

## Contents

# Overview

The following graphic shows an overview of the development flow:



# Installation

## File System Layout

Including the SDK from SVN, the expected file system layout will be:

```
..../ai8x-training/
..../ai8x-synthesis/
..../manifold/
..../AI84SDK/
..../AI85SDK/
```

where "...." is the project root.

## Upstream Code

Change to the project root (denoted as `....` above) and run the following commands (substituting your single sign-on name for "first.last"):

```
$ git clone https://first.last@gerrit.maxim-ic.com:8443/ai8x-training
$ git clone https://first.last@gerrit.maxim-ic.com:8443/ai8x-synthesis
```

If the local git environment has not been previously configured, add the following commands:

```
$ git config --global user.email "first.last@maximintegrated.com"
$ git config --global user.name "First Last"
```

## Prerequisites

When going beyond simple tests, model training requires CUDA hardware acceleration (the network loader does not require CUDA).

Install CUDA 10.1 and CUDNN (PyTorch 1.3.1 does not support CUDA 10.2):
https://developer.nvidia.com/cuda-downloads
https://developer.nvidia.com/cudnn

*Note: When using multiple GPUs, the software will automatically use all available GPUs and distribute the workload. To prevent this, either use the `--gpus` command line argument, or set the `CUDA_VISIBLE_DEVICES` environment variable.*

### Shared (Multi-User) and Remote Systems

On a shared (multi-user) system that has previously been set up , only local installation is needed. CUDA and any `apt-get` or `brew` tasks are not necessary.

The `screen` command can be used inside a remote terminal to disconnect a session from the controlling terminal, so that a long running training session doesn't abort due to network issues, or local power saving.

Example:

```
$ ssh targethost
targethost$ screen # or screen -r to resume, screen -list to list
targethost$
Ctrl+A,D to disconnect
```

`man screen` has more information.

### Recommended Software

The following software is optional, and can be replaced with other similar software of the user's choosing.

1. Visual Studio Code (Editor, Free), https://code.visualstudio.com, with the "Remote - SSH" plugin
2. Typora (Markdown Editor, Free during beta), http://typora.io
3. CoolTerm (Serial Terminal, Free), http://freeware.the-meiers.org
   or Serial ($30), https://apps.apple.com/us/app/serial/id877615577?mt=12
4. Git Fork (Graphical Git Client, Free), https://git-fork.com
5. Beyond Compare (Diff and Merge Tool, $60), https://scootersoftware.com

## Project Installation

*The software in this project requires Python 3.6.9 or a later 3.6.x version. Versions 3.7/3.8 are not yet supported.*

It is not necessary to install Python 3.6.9 system-wide, or to rely on the system-provided Python. To manage Python versions, use `pyenv` (https://github.com/pyenv/pyenv).

On macOS (no CUDA support available):

```
$ brew install pyenv pyenv-virtualenv libomp libsndfile
```

On Ubuntu 18.04 LTS:

```
$ sudo apt-get install -y make build-essential libssl-dev zlib1g-dev \
  libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm \
  libncurses5-dev libncursesw5-dev xz-utils tk-dev libffi-dev liblzma-dev \
  libsndfile-dev portaudio19-dev
$ curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-installer
| bash
```

Then, add to ~/.bash_profile or ~/.profile (as shown by the terminal output of the previous step):

```
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

Next, close the Terminal and install Python 3.6.9:

```
$ pyenv install 3.6.9
```

### Windows Systems

Windows is not supported at this time.

### Creating the Virtual Environment

To create the virtual environment and install basic wheels:

```
$ cd ai8x-training
$ git submodule update --init
$ pyenv local 3.6.9
$ python3 -m venv .
$ source bin/activate
(ai8x-training) $ pip3 install -U pip setuptools
```

The next step differs depending on whether the system is Linux with CUDA, or not.

For CUDA 10.1 on Linux:

```
(ai8x-training) $ pip3 install -r requirements-cuda.txt
```

For all other systems:

```
(ai8x-training) $ pip3 install -r requirements-cpu.txt
```

*Note*: On x86 systems, the pre-built TensorFlow wheels require AVX (on macOS, run `sysctl -n machdep.cpu.features` to find out, on Linux use `cat /proc/cpuinfo | grep avx`).
Running a TensorFlow wheel that requires AVX instructions on unsupported CPUs results in `Illegal instruction: 4` on startup.

### Building TensorFlow (for old CPUs)

If the CPU does not support AVX, or to enable support for AVX2, or CUDA, or AMD64, build TensorFlow locally. Otherwise, skip ahead to Nervana Distiller.
*To repeat: Building TensorFlow is not needed when the binary wheels are functioning properly.*

The TensorFlow build requires Java 8 and Bazel, and takes over two hours.
Building TensorFlow 1.14 requires Bazel 0.25.2 (newer or much older versions do not work). See https://docs.bazel.build/versions/master/install-compile-source.html#build-bazel-using-bazel for build instructions.

Once Bazel is installed, compile and install TensorFlow. On Jetson TX1, disable S3. See https://github.com/peterlee0127/tensorflow-nvJetson/blob/master/patch/tensorflow1.12.patch
The removal of `-mfpu=neon` does not seem to be necessary.

```
(ai8x-training) $ pip3 install keras_preprocessing
(ai8x-training) $ git clone https://github.com/tensorflow/tensorflow
(ai8x-training) $ cd tensorflow
(ai8x-training) $ git checkout tags/v1.14.0
(ai8x-training) $ ./configure
(ai8x-training) $ bazel build --config=opt
//tensorflow/tools/pip_package:build_pip_package
(ai8x-training) $ bazel-bin/tensorflow/tools/pip_package/build_pip_package
/tmp/tensorflow_pkg
(ai8x-training) $ pip3 install /tmp/tensorflow_pkg/tensorflow-1.14.0-cp36-cp36m-
macosx_10_13_x86_64.whl
```

## Nervana Distiller

Nirvana Distiller is package for neural network compression and quantization. Network compression can reduce the memory footprint of a neural network, increase its inference speed and save energy. Distiller is automatically installed with the other packages.

On macOS, add the following to `~/.matplotlib/matplotrc`:

```
backend: TkAgg
```

## Uber Manifold

Manifold is a model-agnostic visual debugging tool for machine learning. Manifold can compare models, detects which subset of data a model is inaccurately predicting, and explains the potential cause of poor model performance by surfacing the feature distribution difference between better and worse-performing subsets of data.

There is a hosted version of Manifold at http://manifold.mlvis.io/. To install it locally (for IP reasons and higher speed):

On macOS,

```
brew install yarn npm
```

On Ubuntu 18.04 LTS,

```
$ cd PROJECT_ROOT
$ curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
$ echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee
/etc/apt/sources.list.d/yarn.list
$ curl -sL https://deb.nodesource.com/setup_13.x | sudo -E bash -
$ sudo apt-get update
$ sudo apt-get install nodejs yarn
```

On both Mac and Linux:

```
$ git clone https://github.com/uber/manifold.git
$ cd manifold
$ yarn
# ignore warnings
$ cd examples/manifold
$ yarn
# ignore warnings
```

### Synthesis Project

For `ai8x-synthesis`, some of the installation steps can be simplified. Specifically, CUDA is not necessary.

Start by creating a second virtual environment:

```
$ cd PROJECT_ROOT
$ cd ai8x-synthesis
$ git submodule update --init
$ pyenv local 3.6.9
$ python3 -m venv .
$ source bin/activate
(ai8x-synthesis) $ pip3 install -U pip setuptools
(ai8x-synthesis) $ pip3 install -r requirements.txt
```

---

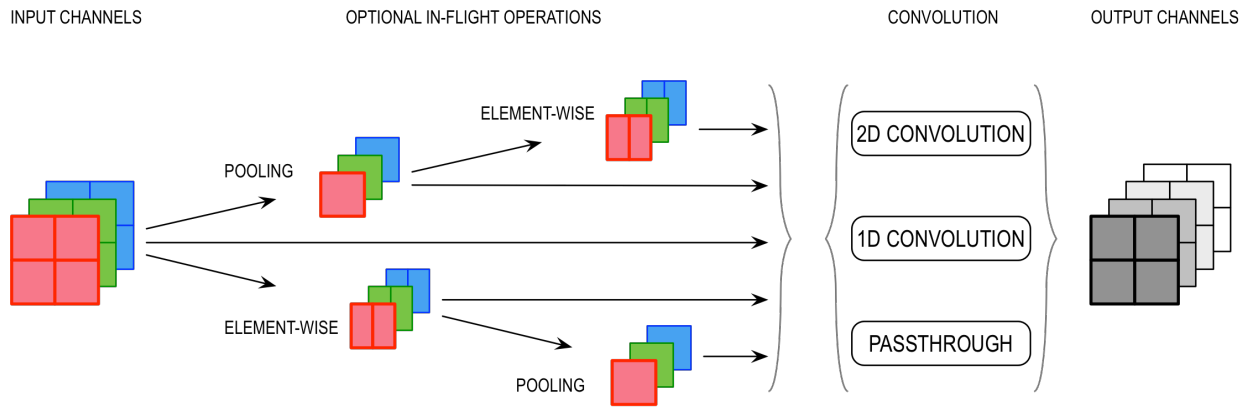
# AI8X Hardware and Resources

AI8X are embedded accelerators. Unlike GPUs, AI8X do not have gigabytes of memory, and cannot support arbitrary data (image) sizes.

## Overview

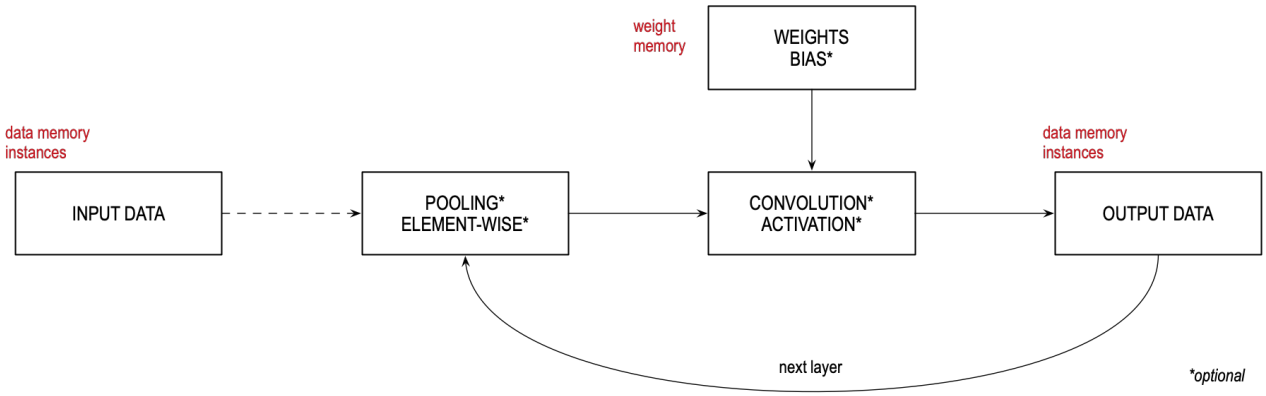
A typical CNN operation consists of pooling followed by a convolution. While these are traditionally expressed as separate layers, pooling can be done “in-flight” on AI8X for greater efficiency.

To minimize data movement, the accelerator is optimized for convolutions with in-flight pooling on a sequence of layers. AI85 and AI86 also support in-flight element-wise operations, pass-through layers and 1D convolutions (without element-wise operations):



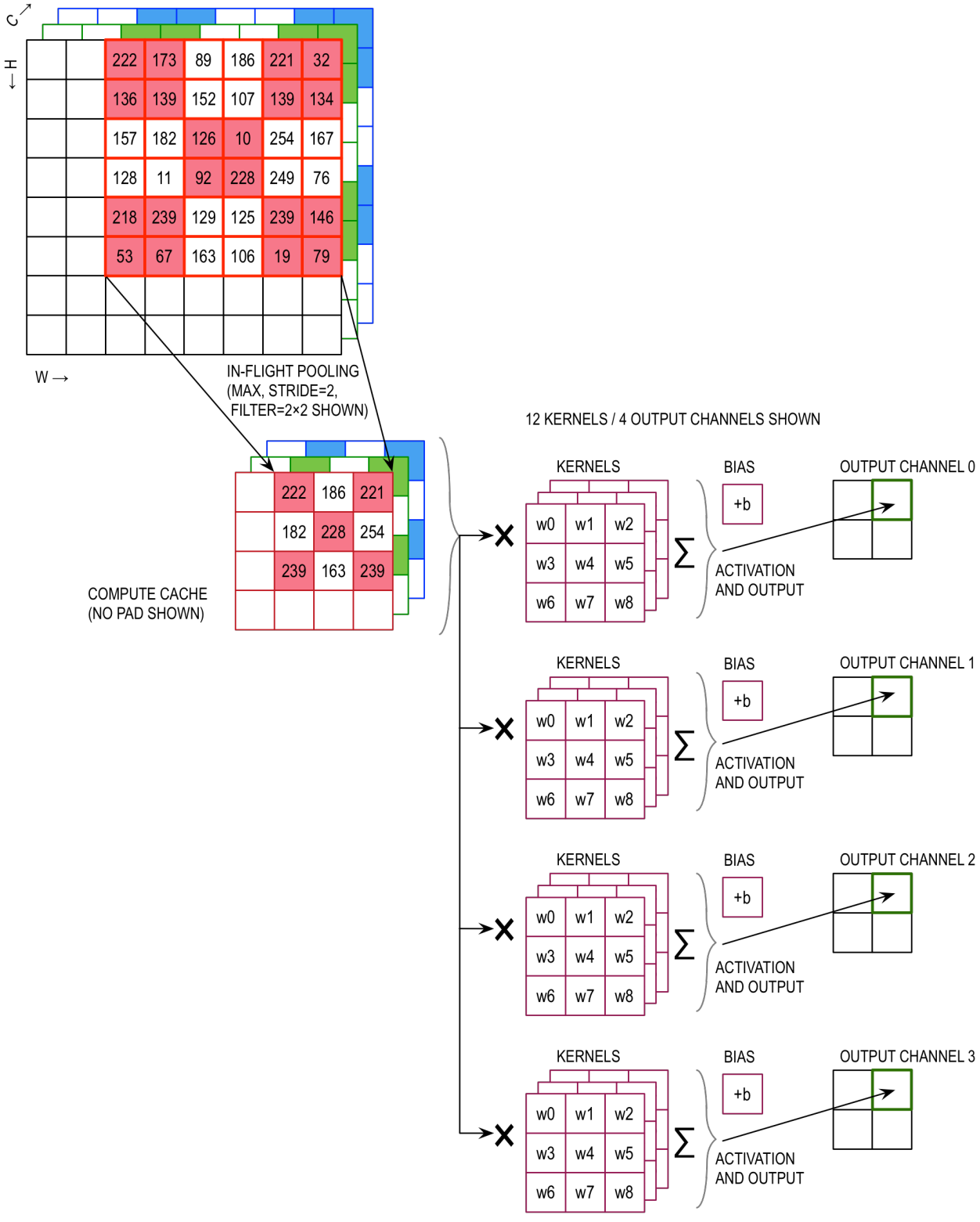

The AI8X accelerator consists of 64 parallel processors. There are four groups that contain 16 processors each.

Each processor includes a pooling unit and a convolutional engine with dedicated weight memory:




Data is read from data memory associated with the processor, and written out to any data memory located within the accelerator. To run a deep convolutional neural network, multiple layers are chained together, where each layer's operation is individually configurable. The output data from one layer is used as the input data for the next layer, for up to 32 layers (where *in-flight* pooling and *in-flight* element-wise operations do not count as layers).

The following picture shows an example view of a 2D convolution with pooling:




## Data, Weights, and Processors

Data memory, weight memory, and processors are interdependent.

In the AI8X accelerator, processors are organized as follows:

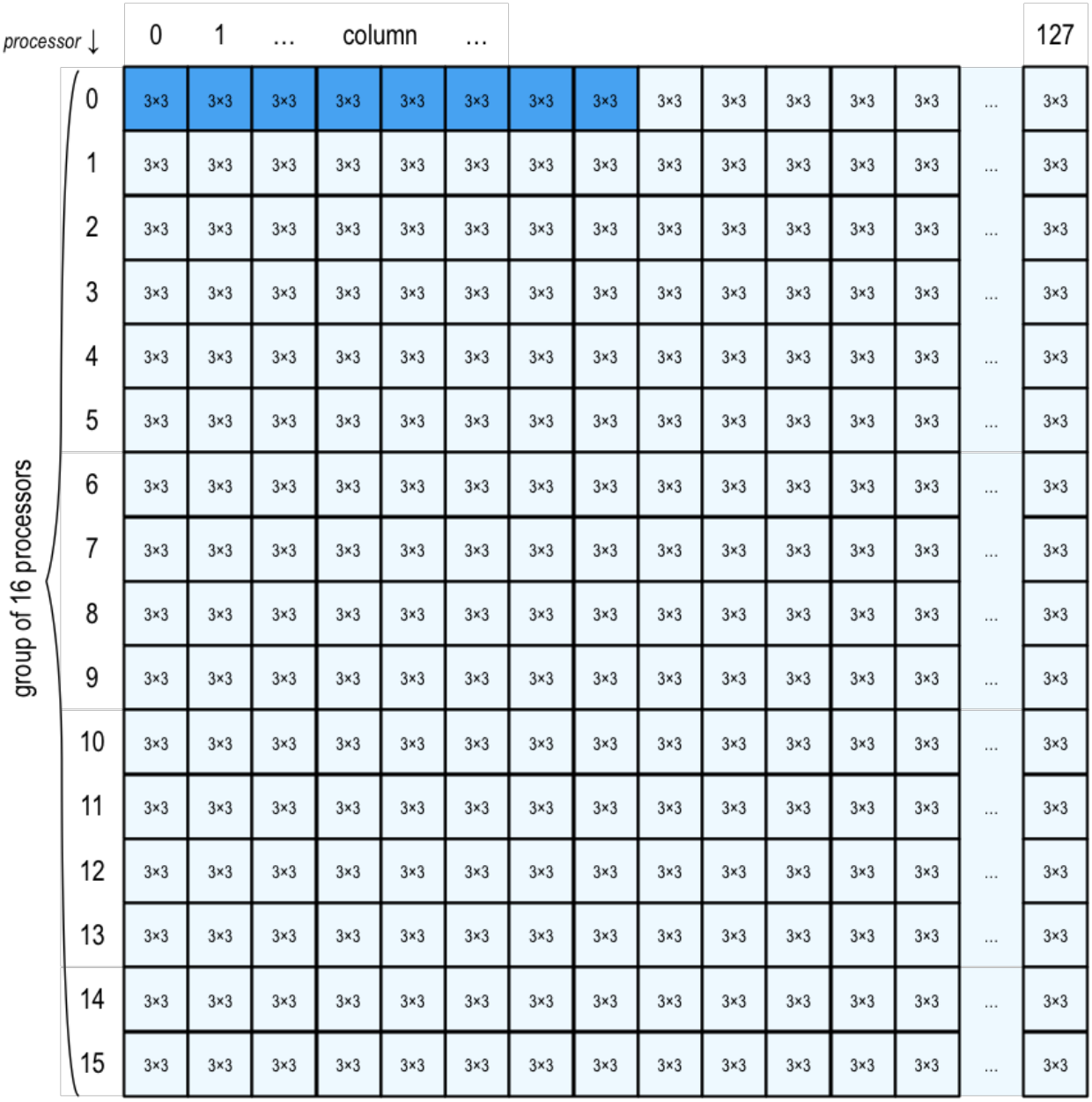
- Each processor is connected to its own dedicated weight memory instance.
- Four processors share one data memory instance.
- A group of sixteen processors shares certain common controls and can be operated as a slave to another group, or independently/separately.

Any given processor has visibility of:

- Its dedicated weight memory, and
- The data memory instance it shares with three other processors.

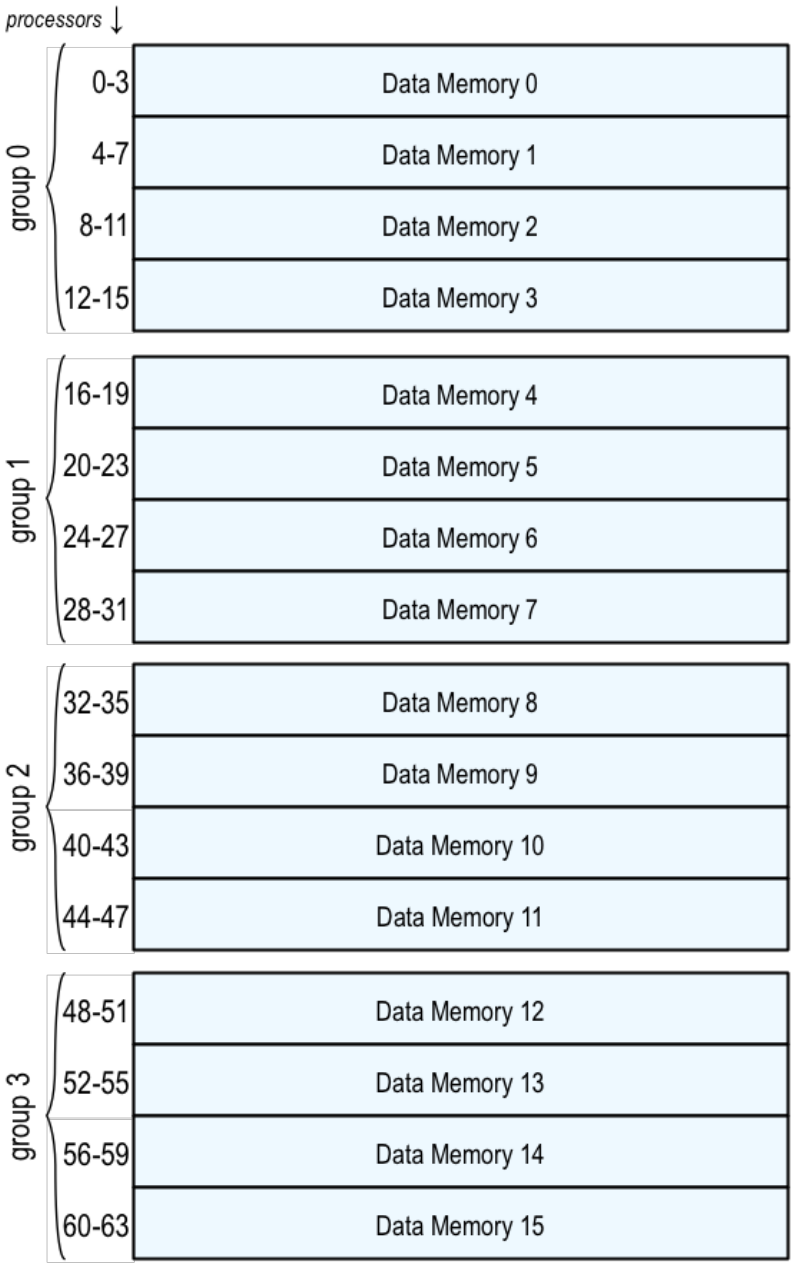
## Weight Memory

For each of the four 16-processor groups, weight memory and processors can be visualized as follows. Assuming one input channel processed by processor 0, and 8 output channels, the 8 shaded kernels will be used:




## Data Memory

Data memory connections can be visualized as follows:



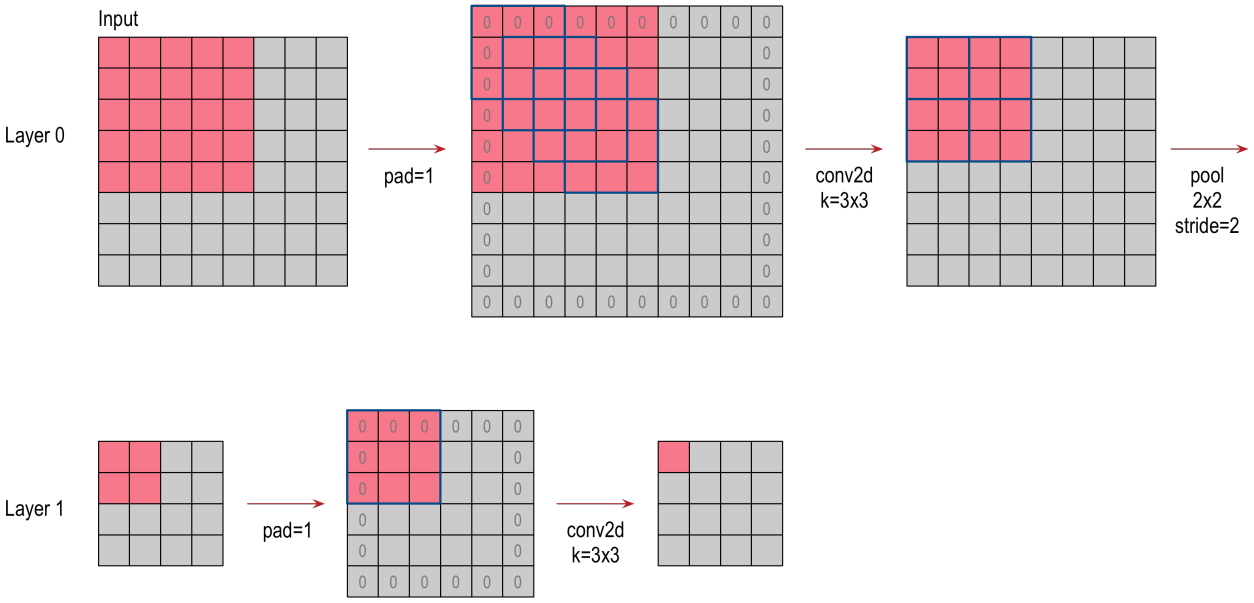

All input data must be located in the data memory instance the processor can access. Conversely, output data can be written to any data memory instance inside the accelerator (but not to general purpose SRAM on the Arm microcontroller bus).

The data memory instances inside the accelerator are single-port memories. This means that only one access operation can happen per clock cycle. When using the HWC data format (see Channel Data Formats), this means that each of the four processors sharing the data memory instance will receive one byte of data per clock cycle (since each 32-bit data word consists of four packed channels).

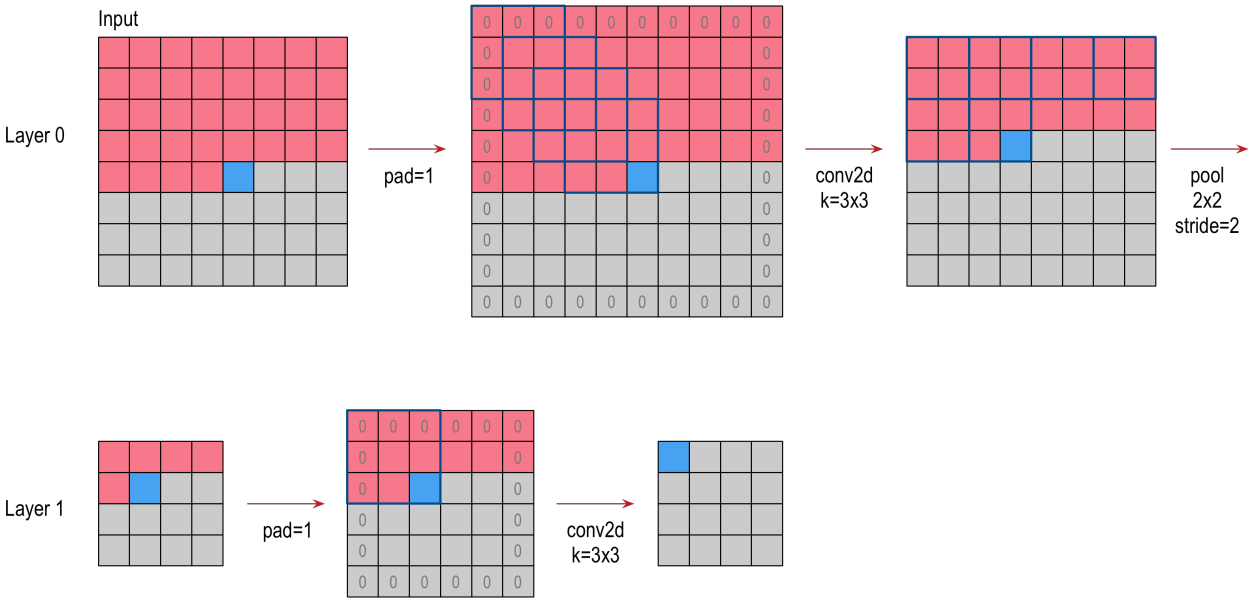
## Streaming Mode

On AI85/AI86, the machine also implements a streaming mode. Streaming allows input data dimensions that exceed the available per-channel data memory in the accelerator.

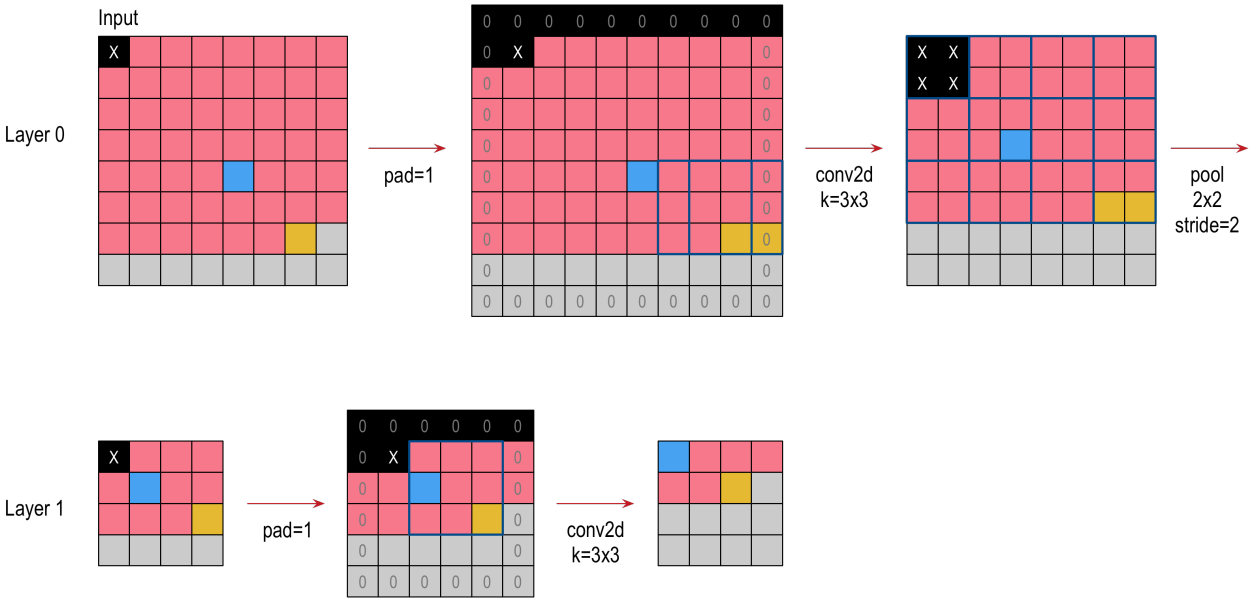
The following illustration shows the basic principle: In order to produce the first output pixel of the second layer, not all data needs to be present at the input. In the example, a 5×5 input needs to be available.




In the accelerator implementation, data is shifted into the Tornado memory in a sequential fashion, so prior  rows will be available as well. In order to produce the *blue* output pixel, input data up to the blue input pixel must be available.




When the *yellow* output pixel is produced, the first (*black*) pixel of the input data is no longer needed and its data can be discarded:




The number of discarded pixels is network specific and dependent on pooling strides and the types of convolution. In general, streaming mode is only useful for networks where the output data dimensions decrease from layer to layer (for example, by using a pooling stride).

*Note: Streaming mode requires the use of FIFOs.*

### FIFOs

Since the data memory instances are single-port memories, software would have to temporarily disable the accelerator in order to feed it new data. Using FIFOs, software can input available data while the accelerator is running. The accelerator will autonomously fetch data from the FIFOs when needed, and stall (pause) when no enough data is available.

The AI85/AI86 accelerator has two types of FIFO:

#### Standard FIFOs

There are four dedicated FIFOs connected to processors 0-3, 16-19, 32-35, and 48-51, supporting up to 16 input channels (in HWC format) or four channels (CHW format). These FIFOs work when used from the ARM Cortex-M4 core and from the RISC-V core.

The standard FIFOs are selected using the `--fifo` argument for `cnn-gen.py`.

#### Fast FIFO

The fast FIFO is only available from the RISC-V core, and runs synchronously with the RISC-V for increased throughput. It supports up to four input channels (HWC format) or a single channel (CHW format). The fast FIFO is connected to processors 0, 1, 2, 3 or 0, 16, 32, 48.

The fast FIFO is selected using the `--fast-fifo` argument for `cnn-gen.py`.

## Accelerator Limits

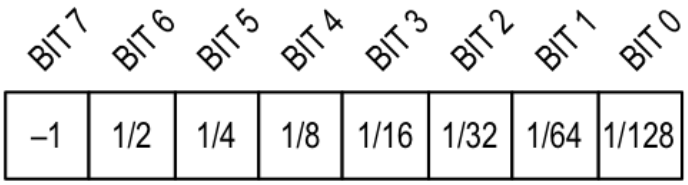
- AI84:
  - The maximum number of layers is 32 (pooling layers do not count when preceding a convolution).
  - The maximum number of input or output channels in any layer is 64 each.
  - The weight memory supports up to 128 * 64 3×3 Q7 kernels (see Number Format). However, weights must be arranged according to specific rules detailed below.
  - There are 16 instances of 16 KiB data memory. Any data channel (input, intermediate, or output) must completely fit into one memory instance. This limits the first-layer input to 128×128 pixels per channel in the CHW format. However, when using more than one input channel, the HWC format may be preferred, and all layer output are in HWC format as well. In those cases, it is required that four channels fit into a single memory instance -- or 64×64 pixels per channel. Note that the first layer commonly creates a wide expansion (i.e., large number of output channels) that needs to fit into data memory, so the input size limit is mostly theoretical.
- AI85:
  - The maximum number of layers is 32 (pooling and element-wise layers do not count when preceding a convolution).
  - The maximum number of input channels in any layer is 1024 each.
  - The maximum number of output channels in any layer is 1024 each.
  - The weight memory supports up to 768 * 64 3×3 Q7 kernels (see Number Format).
    When using 1-, 2- or 4 bit weights, the capacity increases accordingly.
    When using more than 64 input or output channels, weight memory is shared and effective capacity decreases.
    Weights must be arranged according to specific rules detailed below.
  - There are 16 instances of 32 KiB data memory. When not using streaming mode, any data channel (input, intermediate, or output) must completely fit into one memory instance. This limits the first-layer input to 181×181 pixels per channel in the CHW format. However, when using more than one input channel, the HWC format may be preferred, and all layer output are in HWC format as well. In those cases, it is required that four channels fit into a single memory instance -- or 91×90 pixels per channel.
    Note that the first layer commonly creates a wide expansion (i.e., large number of output channels) that needs to fit into data memory, so the input size limit is mostly theoretical.
  - When using streaming, the data sizes are limited to 1023×1023, subject to available TRAM. Streaming is limited to 8 layers or less, and to four FIFOs (up to 4 input channels in CHW and up to 16 channels in HWC format). When using streaming, the product of a layer's input data width, input data height, and input data channels divided by 64 rounded up must not exceed 2^21: $rows * columns * \lceil \frac{channels}{64} \rceil < 2^{21}$.

# Number Format

All weights, bias values and data are stored and computed in Q7 format (signed two's complement 8-bit integers, [-128…+127]). See https://en.wikipedia.org/wiki/Q_%28number_format%29.

The 8-bit value $w$ is defined as:

$$w = (-a_7 2^7 + a_6 2^6 + a_5 2^5 + a_4 2^4 + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0)/128$$




Examples:

| Binary | Value |
|---|---|
| 0000 0000 | 0 |
| 0000 0001 | 1/128 |
| 0000 0010 | 2/128 |
| 0111 1110 | 126/128 |
| 0111 1111 | 127/128 |
| 1000 0000 | −128/128 (–1) |
| 1000 0001 | −127/128 |
| 1000 0010 | −126/128 |
| 1111 1110 | −2/128 |
| 1111 1111 | −1/128 |

On **AI85**, *weights* can be 1, 2, 4, or 8 bits wide (configurable per layer using the `quantization` key). Bias values are always 8 bits wide. Data is 8 bits wide, except for the last layer that can optionally output 32 bits of unclipped data in Q25.7 format when not using activation.

| wt bits | min | max |
|---|---|---|
| 8 | –128 | +127 |
| 4 | –8 | 7 |
| 2 | –2 | 1 |
| 1 | –1 | 0 |

Note that 1-bit weights (and, to a lesser degree, 2-bit weights) require the use of bias to produce useful results. Without bias, all sums of products of activated data from a prior layer would be negative, and activation of that data would always be zero.

## Rounding

AI8X rounding (for the CNN sum of products) uses "round half towards positive infinity", i.e. $y = \lfloor 0.5 + x \rfloor$. This rounding method is not the default method in either Excel or Python/NumPy. The rounding method can be achieved in NumPy using `y = np.floor(0.5 + x)` and in Excel as `=FLOOR.PRECISE(0.5 + X)`.

By way of example:

| Input | Rounded |
|---|---|
| +3.5 | +4 |
| +3.25, +3.0, +2.75, +2.5 | +3 |
| +2.25, +2.0, +1.75, +1.5 | +2 |
| +1.25, +1.0, +0.75, +0.5 | +1 |
| +0.25, 0, –0.25, –0.5 | 0 |
| –0.75, –1.0, –1.25, –1.5 | –1 |
| –1.75, –2.0, –2.25, –2.5 | –2 |
| –2.75, –3.0, –3.25, –3.5 | –3 |

## Addition

Addition works similarly to regular two's-complement arithmetic.

Example:
$w_0 = 1/64 \rightarrow 00000010$
$w_1 = 1/2 \rightarrow 01000000$
$w_0 + w_1 = 33/64 \rightarrow 01000010$

## Saturation and Clipping

Values smaller than $-128 \;/\; 128$ are saturated to $-128 \;/\; 128$ (1000 0000). Values larger than $+127 \;/\; 128$ are saturated to $+127 \;/\; 128$ (0111 1111).

The AI8X CNN sum of products uses full resolution for both products and sums, so the saturation happens only at the very end of the computation.

Example 1:

$w_0 = 127/128 \rightarrow 01111111$
$w_1 = 127/128 \rightarrow 01111111$
$w_0 + w_1 = 254/128 \rightarrow saturate \rightarrow 01111111 (= 127/128)$

Example 2:

$w_0 = -128/128 \rightarrow 10000000$
$w_1 = -128/128 \rightarrow 10000000$
$w_0 + w_1 = -256/128 \rightarrow saturate \rightarrow 10000000 (= -128/128)$

## Multiplication

Since operand values are implicitly divided by 128, the product of two values has to be shifted in order to maintain magnitude when using a standard multiplier (e.g., 8×8):

$w_0 * w_1 = \frac{w'_0}{128} * \frac{w'_1}{128} = \frac{w'_0 * w'_1}{128} \gg 7$

In software,

- Determine the sign bit: $s = sign(w_0) * sign(w_1)$
- Convert operands to absolute values: $w'_0 = abs(w_0); w'_1 = abs(w_1)$
- Multiply using standard multiplier: $w'_0 * w'_1 = w''_0/128 * w''_1/128; r' = w''_0 * w''_1$
- Shift: $r'' = r' \gg 7$
- Round up/down depending on $r'[6]$
- Apply sign: $r = s * r''$

Example 1:

$w_0 = 1/64 \rightarrow 00000010$
$w_1 = 1/2 \rightarrow 01000000$
$w_0 * w_1 = 1/128 \rightarrow shift, truncate \rightarrow 00000001 (= 1/128)$

A “standard” two’s-complement multiplication would return 00000000 10000000. The AI8X data format discards the rightmost bits.

Example 2:

$w_0 = 1/64 \rightarrow 00000010$
$w_1 = 1/4 \rightarrow 00100000$
$w_0 * w_1 = 1/256 \rightarrow shift, truncate \rightarrow 00000000 (= 0)$

“Standard” two’s-complement multiplication would return 00000000 01000000, the AI8X result is truncated to 0 after the shift operation.

## Sign Bit

Operations preserve the sign bit.

Example 1:

$w_0 = -1/64 \rightarrow 11111110$
$w_1 = 1/4 \rightarrow 00100000$
$w_0 * w_1 = -1/256 \rightarrow shift, truncate \rightarrow 00000000 (= 0)$

- Determine the sign bit: $s = sign(-1/64) * sign(1/4) = -1 * 1 = -1$

- Convert operands to absolute values: $w_0' = abs(-1/64); w_1' = abs(1/4)$
- Multiply using standard multiplier: $r' = 1/64 \ll 7 * 1/4 \ll 7 = 2 * 32 = 64$
- Shift: $r'' = r' \gg 7 = 64 \gg 7 = 0$
- Apply sign: $r = s * r'' = -1 * 0 = 0$

Example 2:

$w_0 = -1/64 \rightarrow 11111110$
$w_1 = 1/2 \rightarrow 01000000$
$w_0 * w_1 = -1/128 \rightarrow shift, truncate \rightarrow 11111111 (= -1/128)$

- Determine the sign bit: $s = sign(-1/64) * sign(1/2) = -1 * 1 = -1$
- Convert operands to absolute values: $w_0' = abs(-1/64); w_1' = abs(1/2)$
- Multiply using standard multiplier: $r' = 1/64 \ll 7 * 1/2 \ll 7 = 2 * 64 = 128$
- Shift: $r'' = r' \gg 7 = 128 \gg 7 = 1$
- Apply sign: $r = s * r'' = -1 * 1 \gg 7 = -1/128$

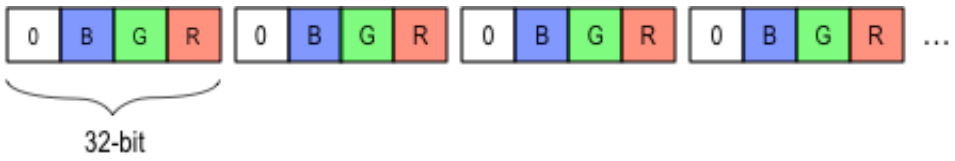
Example 3:

$w_0 = 127/128 \rightarrow 01111111$
$w_1 = 1/128 \rightarrow 00000001$
$w_0 * w_1 = 128/128 \rightarrow saturation \rightarrow 01111111 (= 127/128)$

# Channel Data Formats

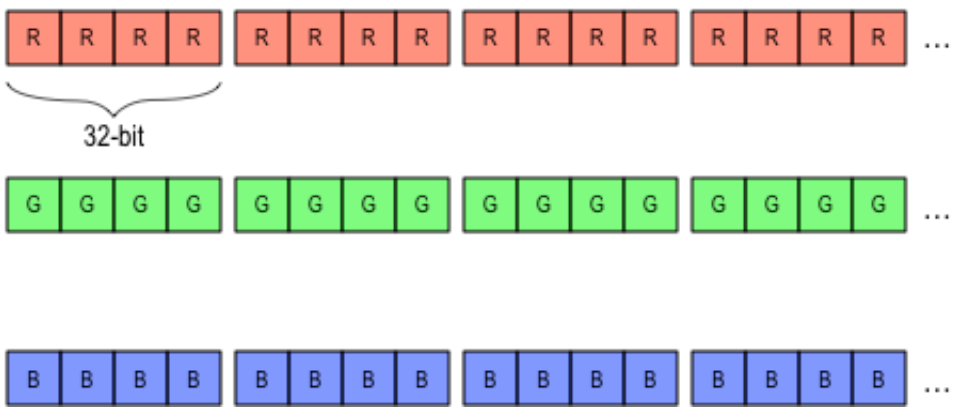
## HWC

All internal data are stored in HWC format, 4 channels per 32-bit word. Assuming 3-color (or 3-channel) input, one byte will be unused. Example:




## CHW

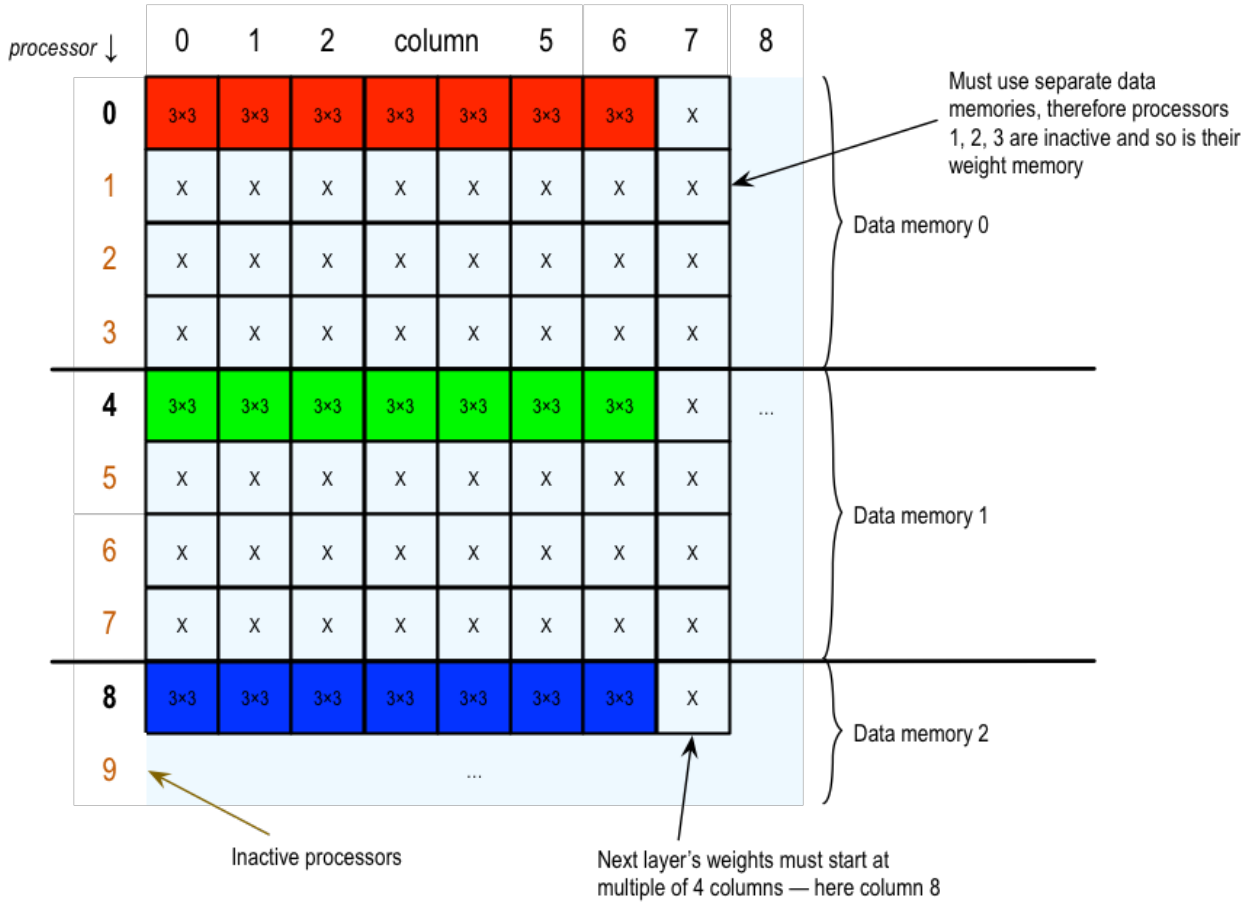
The input layer can also use the CHW format (sequence of channels), for example:



## CHW Data Format and Consequences for Weight Memory Layout

When using the CHW data format, only one of the four processors sharing the data memory instance can be used. The next channel needs to use a processor connected to a different data memory instance, so that the machine can deliver one byte per clock cycle to each enabled processor.

Because of the fact that a processor has its own dedicated weight memory, this will introduce “gaps” in the weight memory map, as shown in the following illustration:




## Active Processors and Layers

For each layer, a set of active processors must be specified. The number of active processors must be the same as the number of input channels for the layer, and the input data for that layer must be located in data memory instances accessible to the selected processors.

It is possible to specify a relative offset into the data memory instance that applies to all processors. *Example:* Assuming HWC data format, specifying the offset as 8192 bytes will cause processors 0-3 to read their input from the second half of data memory 0, processors 4-7 will read from the second half of data memory instance 1, etc.

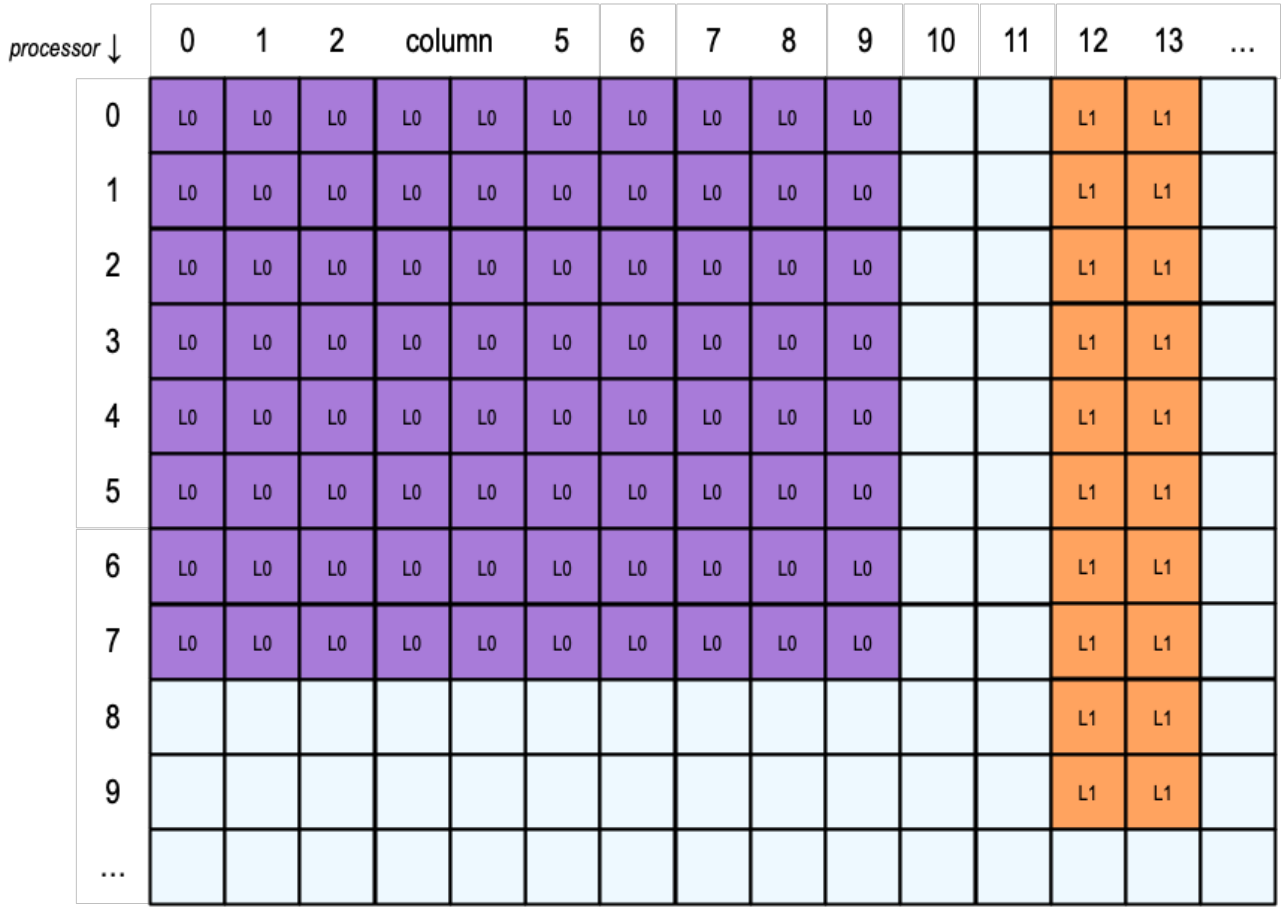
For most simple networks with limited data sizes, it is easiest to ping-pong between the first and second halves of the data memories - specify the data offset as 0 for the first layer, 0x2000 for the second layer, 0 for the third layer, etc. This strategy avoids overlapping inputs and outputs when a given processor is used in two consecutive layers.

Even though it is supported by the accelerator, the Network Generator will not be able to check for inadvertent overwriting of unprocessed input data by newly generated output data when overlapping data or streaming data. Use the `--overlap-data` command line switch to disable these checks, and to allow overlapped data.

## Layers and Weight Memory

For each layer, the weight memory start column is automatically configured by the Network Loader. The start column must be a multiple of 4, and the value applies to all processors.

The following example shows the weight memory layout for two layers. The first layer (L0) has 7 inputs and 9 outputs, and the second layer (L1) has 9 inputs and 2 outputs.




## Weight Storage Example

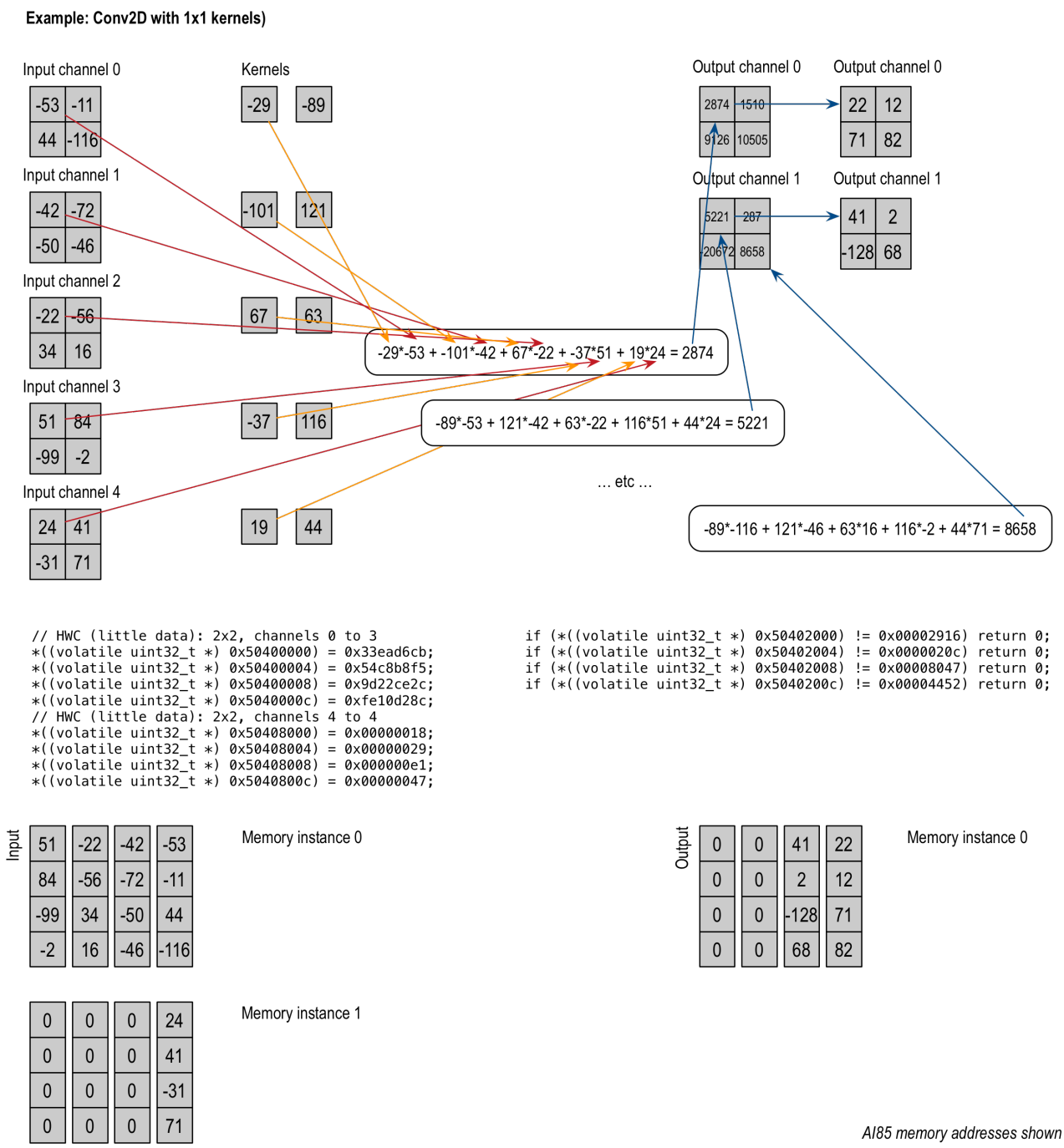
The file `ai84net.xlsx` contains an example for a single-channel CHW input using the `AI85Net5` network (this example also supports up to four channels in HWC).

*Note*: As described above, multiple CHW channels must be loaded into separate memory instances. When using a large number of channels, this can cause "holes" in the processor map, which in turn can cause subsequent layers' kernels to require padding.

The Network Loader prints a kernel map that shows the kernel arrangement based on the provided network description. It will also flag cases where kernel or bias memories are exceeded.

## Example: `Conv2D`

The following picture shows an example of a `Conv2d` with 1×1 kernels, 5 input channels, 2 output channels and data size of 2×2. The inputs are shown on the left, and the outputs on the right, and the kernels are shown lined up with the associated inputs --- the number of kernel rows matches the number of input channels, and the number kernel columns matches the number of output channels. The lower half of the picture shows how the data is arranged in memory when HWC data is used for both input and output.




```
// HWC (little data): 2x2, channels 0 to 3
*((volatile uint32_t *) 0x50400000) = 0x33ead6cb;
*((volatile uint32_t *) 0x50400004) = 0x54c8b8f5;
*((volatile uint32_t *) 0x50400008) = 0x9d22ce2c;
*((volatile uint32_t *) 0x5040000c) = 0xfe10d28c;
// HWC (little data): 2x2, channels 4 to 4
*((volatile uint32_t *) 0x50408000) = 0x00000018;
*((volatile uint32_t *) 0x50408004) = 0x00000029;
*((volatile uint32_t *) 0x50408008) = 0x000000e1;
*((volatile uint32_t *) 0x5040800c) = 0x00000047;
```

```
if (*((volatile uint32_t *) 0x50402000) != 0x00002916) return 0;
if (*((volatile uint32_t *) 0x50402004) != 0x0000020c) return 0;
if (*((volatile uint32_t *) 0x50402008) != 0x00008047) return 0;
if (*((volatile uint32_t *) 0x5040200c) != 0x00004452) return 0;
```




# Limitations of AI84 Networks

The AI84 hardware does not support arbitrary network parameters. Specifically,

- Dilation, groups, element-wise addition, and batch normalization are not supported.
- `Conv2d`:
  - Input data must be square (i.e., rows == columns).
  - Kernel sizes must be 3×3.
  - Padding can be 0, 1, or 2.

    - Stride is fixed to 1. Pooling, including 1×1, can be used to achieve a stride other than 1.
- `Conv1d`:
    - Input data lengths must be a multiple of 3.
    - Kernel size must be 9.
    - Padding can be 0, 3, or 6.
    - Stride is fixed to 3.
- The only supported activation function is `ReLU`.
- Pooling:
    - Pooling is only supported for `Conv2d`.
    - Pooling is always combined with a convolution. Both max pooling and average pooling are available.
    - Pooling does not support padding.
    - Pooling strides can be 1, 2, or 4.
    - On AI84, three pooling sizes are supported:
        - 2×2 with stride 1
        - 2×2 with stride 2
        - and, for the last layer, 4×4 stride 4 using a custom unload function.
    - Pooling must cleanly divide the input data width and height. For example, a 2×2 pool on 17×17 data is not supported.
    - Average pooling does not support more than 2048 bits in the accumulator. This translates to a 4×4 pooling window if activation was used on the prior layer, 3×3 otherwise. Additionally, average pooling is currently implemented as a `floor()` operation. Since there is also a quantization step at the output of the average pooling, it may not perform as intended (for example, a 2×2 `AvgPool2d` of `[[0, 0], [0, 3]]` will return `0`).
    - Pooling window sizes must be even numbers, and have equal H and W dimensions.
- The number of input or output channels must not exceed 64.
- The number of layers must not exceed 32 (where pooling does not add to the count when preceding a convolution).
- The maximum dimension (number of rows or columns) for input or output data is 256.
- Overall weight storage is limited to 64*128 3×3 kernels. However, weights must be arranged in a certain order, see above.
- The hardware supports 1D and 2D convolution layers. For convenience, a single final fully connected layer with 8-bit inputs/weights/bias, and 16-bit output is supported in software, as well as a software `SoftMax` operator.
- Since the internal network format is HWC in groups of four channels, output concatenation only works properly when all components of the concatenation other than the last have multiples of four channels. *Output concatenation is not yet supported in the `ai8x.py` primitives.*

- It is recommended to not use bias on the AI84 convolutions when using post-training quantization as described in this document. The reason is that the bias is not shifted by the same amount as the weights. This could be mitigated using a more involved training procedure, but since bias values do not significantly improve performance in the example networks, and since this will be corrected for AI85, the recommendation is to not use bias values for now.

With the exception of weight storage, and bias use, most of these limitations will be flagged when using the network primitives from `ai8x.py`, see Model Training and Quantization.

## Limitations of AI85 Networks

The AI85 hardware does not support arbitrary network parameters. Specifically,

- Dilation, groups, depth-wise convolutions, and batch normalization are not supported. *Note: Batch normalization should be folded into the weights.*
- `Conv2d`:
  - Kernel sizes must be 1×1 or 3×3.
  - Padding can be 0, 1, or 2.
  - Stride is fixed to 1. Pooling, including 1×1, can be used to achieve a stride other than 1.
- `Conv1d`:
  - Kernel sizes must be 1 through 9.
  - Padding can be 0, 1, or 2.
  - Stride is fixed to 1. Pooling, including 1, can be used to achieve a stride other than 1.
- `ConvTranspose2d`:
  - Kernel sizes must be 3×3.
  - Padding can be 0, 1, or 2.
  - Stride is fixed to 2.
- A programmable layer-specific shift operator is available at the output of a convolution.
- The supported activation functions are `ReLU` and `Abs`, and a limited subset of `Linear`.
- Pooling:
  - Both max pooling and average pooling are available, with or without convolution.
  - Pooling does not support padding.
  - Pooling strides can be 1 through 16. For 2D pooling, the stride is the same for both dimensions.
  - For 2D pooling, supported pooling kernel sizes are 1×1 through 16×16, including non-square kernels. 1D pooling supports kernels from 1 through 16. *Note: 1×1 kernels can be used when a convolution stride other than 1 is desired.*
  - Average pooling is implemented both using `floor()` and using rounding (half towards positive infinity). Use the `--avg-pool-rounding` switch to turn on rounding in the training software and the Network Generator.

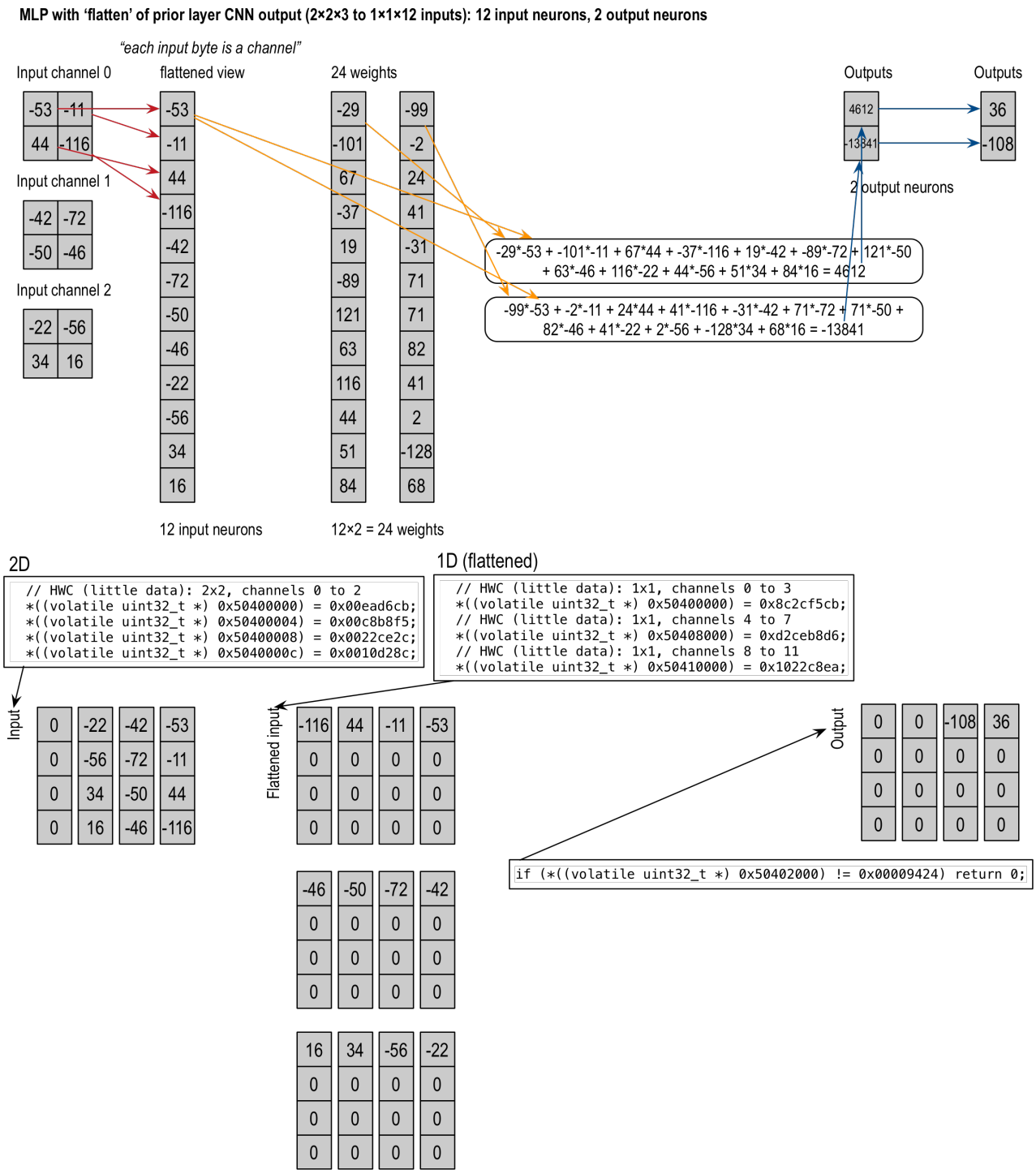
    Example:

    - *floor:* Since there is a quantization step at the output of the average pooling, a 2×2 `AvgPool2d` of `[[0, 0], [0, 3]]` will return $\lfloor \frac{3}{4} \rfloor = 0$.
    - *rounding:* 2×2 `AvgPool2d` of `[[0, 0], [0, 3]]` will return $\lfloor \frac{3}{4} \rceil = 1$.
- The number of input channels must not exceed 1024.
- The number of output channels must not exceed 1024.
- The number of layers must not exceed 32 (where pooling and element-wise operations do not add to the count when preceding a convolution).
- The maximum dimension (number of rows or columns) for input or output data is 1023.
    - When using data greater than 90×91, `streaming` mode must be used.
    - When using `streaming` mode, the product of any layer's input width, input height, and input channels divided by 64 rounded up must not exceed 2^21: $width * height * \lceil \frac{channels}{64} \rceil < 2^{21}$.
- Overall weight storage is limited to 64*768 3×3 8-bit kernels (and proportionally more when using smaller weights, or smaller kernels). However, weights must be arranged in a certain order, see above.
- The hardware supports 1D and 2D convolution layers, 2D transposed convolution layers (upsampling), element-wise addition, subtraction, binary OR, binary XOR as well as fully connected layers (`Linear`) (implemented using 1×1 convolutions on 1×1 data):
    - The maximum number of input neurons is 1024, and the maximum number of output neurons is 1024 (16 each per processor used).
    - `Flatten` functionality is available to convert 2D input data for use by fully connected layers.
    - Element-wise operators support from 2 up to 16 inputs.
    - Element-wise operators can be chained in-flight with pooling and 2D convolution (where the order of pooling and element-wise operations can be swapped).
    - For convenience, a `SoftMax` operator is supported in software.
- Since the internal network format is HWC in groups of four channels, output concatenation only works properly when all components of the concatenation other than the last have multiples of four channels.

## Fully Connected (Linear) Layers

On AI85/AI86, m×n fully connected layers can be realized in hardware by "flattening" 2D input data into m channels of 1×1 input data. The hardware will produce n channels of 1×1 output data. When chaining multiple fully connected layers, the flattening step is omitted. The following picture shows 2D data, the equivalent flattened 1D data, and the output.

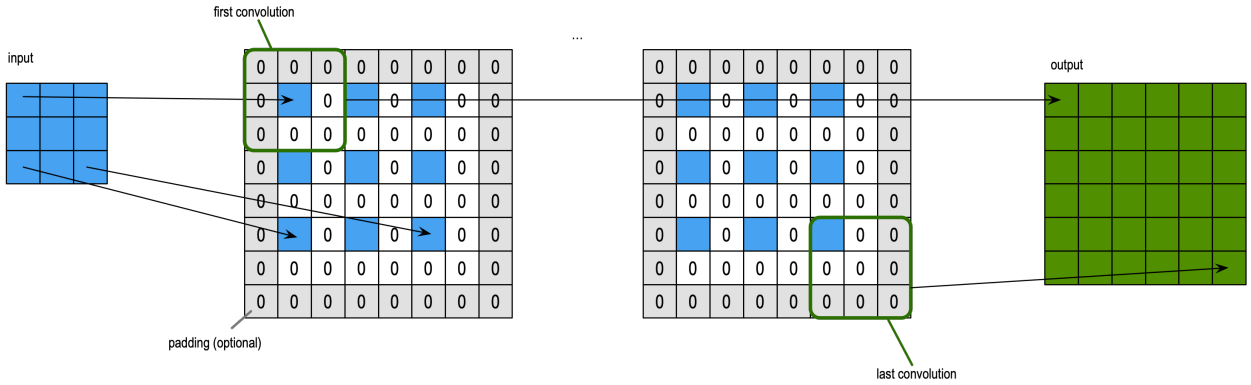
For AI85, both m and n must not be larger than 16.




# Upsampling (Fractionally-Strided 2D Convolutions)

On AI85/AI86, the hardware supports 2D upsampling ("fractionally-strided convolutions", sometimes called "deconvolution" even though this is not strictly mathematically correct). The PyTorch equivalent is `ConvTranspose2D` with a stride of 2.

The example shows a fractionally-strided convolution with a stride of 2, pad of 1, and a 3×3 kernel. This "upsamples" the input dimensions from 3×3 to output dimensions of 6×6.




# Model Training and Quantization

The main training software is `train.py`. It drives the training aspects including model creation, checkpointing, model save, and status display (see `--help` for the many supported options, and the `go_xxx.sh` scripts for example usage).

The `ai84net.py` file contains models that fit into AI84's weight memory. These models rely on the AI84 hardware operators that are defined in `ai8x.py`.

To train the FP32 model for FashionMIST, run `go_fashionmnist.sh` in the `ai8x-training` project. This script will place checkpoint files into the log directory. Training makes use of the Distiller framework, but the `train.py` software has been modified slightly to improve it and add some AI8X specifics.
*Note: `nvidia-smi` can be used in a different terminal during training to examine the GPU resource usage of the training process.*

### Custom nn.Modules

The `ai8x.py` file contains customized PyTorch classes (subclasses of `torch.nn.Module`). Any model that is designed to run on AI8X should use these classes. There are three main changes over the default classes in `torch.nn.Module`:

1. Additional "Fused" operators that model in-flight pooling and activation.
2. Rounding and clipping that matches the hardware.
3. Support for quantized operation (when using the `-8` command line argument).

Note that `torch.nn.Dropout` is not used during inference, and can therefore be used for training without problems.

## Model Comparison and Feature Attribution

Both TensorBoard and Manifold can be used for model comparison and feature attribution.

### TensorBoard

TensorBoard is built into `train.py`. It consists of a local web server that can be started before, during, or after training and it picks up all data that is written to the `logs/` directory. To start the TensorBoard server, use a second terminal window:

```
(ai8x-training) $ tensorboard --logdir='./logs'
TensorBoard 1.14.0 at http://127.0.0.1:6006/ (Press CTRL+C to quit)
```

On a shared system, add the `--port 0` command line option.

Training progress can be observed by starting TensorBoard and pointing a web browser to the port indicated. When using a remote system, use `ssh` in another terminal window to forward the remote port to the local machine:

```
$ ssh -L 6006:127.0.0.1:6006 targethost
```

For classification models, TensorBoard supports the optional `--param-hist` and `--embedding` command line arguments. `--embedding` randomly selects up to 100 data points from the last batch of each verification epoch. These can be viewed in the "projector" tab in TensorBoard.

## Manifold

The quickest way to integrate manifold is by creating CSV files from the training software. *Note that performance will suffer when there are more than about 20,000 records in the CSV file. Subsampling the data is one way to avoid this problem.*

The train.py program can create CSV files using the `--save-csv` command line argument in combination with `--evaluate`:

```
./train.py --model ai84net5 --dataset MNIST --confusion --evaluate --save-csv
mnist --ai84 --exp-load-weights-from ../ai8x-synthesis/trained/ai84-mnist.pth.tar
-8
```

To run the manifold example application:

```
$ cd manifold/examples/manifold
$ npm run start
```

The code will run in JavaScript inside the browser (this may cause warnings that the web page is consuming lots of resources). To run a browser remotely on a development machine, forward X11 using the following command:

```
$ ssh -Yn targethost firefox http://localhost:8080/
```

To forward only the remote web port, use `ssh`:

```
$ ssh -L 8080:127.0.0.1:8080 targethost
```

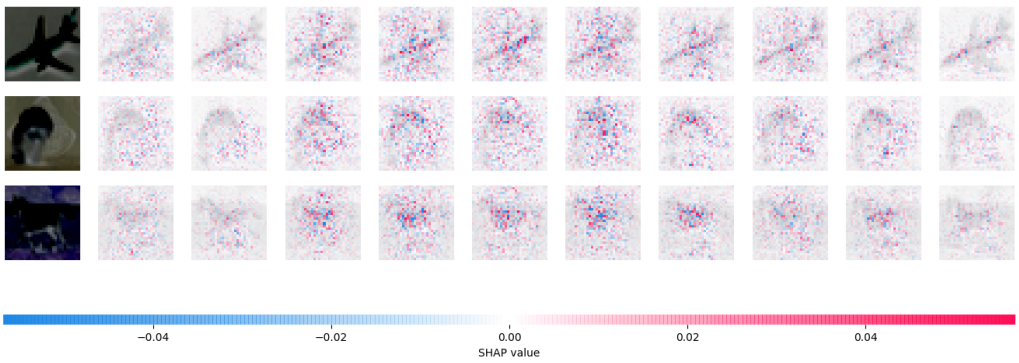
### SHAP — SHapely Additive exPlanations

The training software integrates code to generate SHAP plots (see https://github.com/slundberg/shap). This can help with feature attribution for input images.

The train.py program can create plots using the `--shap` command line argument in combination with `--evaluate`:

```
./train.py --model ai84net5 --dataset CIFAR10 --confusion --evaluate --ai84 --exp-load-weights-from logs/CIFAR-new/best.pth.tar --shap 3
```

This will create a plot with a random selection of 3 test images. The plot shows ten outputs (the ten classes) for the three different input images on the left. Red pixels increase the model's output while blue pixels decrease the output. The sum of the SHAP values equals the difference between the expected model output (averaged over the background dataset) and the current model output.




## Quantization

There are two main approaches to quantization — quantization-aware training and post-training quantization.

Since performance for 8-bit weights is decent enough, *naive post-training quantization* is used in the `ai8xize.py` software. While several approaches are implemented, a simple fixed scale factor is used based on experimental results. The approach requires the clamping operators implemented in `ai8x.py`.

The software quantizes an existing PyTorch checkpoint file and writes out a new PyTorch checkpoint file that can then be used to evaluate the quality of the quantized network, using the same PyTorch framework used for training. The same new checkpoint file will also be used to feed the Network Loader.

Copy the working and tested weight files into the `trained/` folder of the `ai8x-synthesis` project.

Example:

```
(ai8x-synthesis) $ ./ai8xize.py ../ai8x-training/logs/path-to-
checkpoint/checkpoint.pth.tar trained/ai84-fashionmnist.pth.tar -v
```

To evaluate the quantized network:

```
(ai8x-training) $ ./evaluate_fashionmnist.sh
```

## Alternative Quantization Approaches

Post-training quantization can be improved using more sophisticated methods. For example, see https://github.com/pytorch/glow/blob/master/docs/Quantization.md, https://github.com/ARM-software/ML-examples/tree/master/cmsisnn-cifar10, https://github.com/ARM-software/ML-KWS-for-MCU/blob/master/Deployment/Quant_guide.md, or Distiller's approach (installed with this software).

Further, a quantized network can be refined using post-quantization training (see Distiller).

The software also includes an `AI84RangeLinear.py` training quantizer that plugs into the Distiller framework for quantization-aware training. However, it needs work as its performance is not good enough yet and the Distiller source needs to be patched to enable it (add `from range_linear_ai84 import QuantAwareTrainRangeLinearQuantizerAI84` to `distiller/config.py` and remove `False and` from `if False and args.ai84` in `train.py`).

Note that AI84 does not have a configurable per-layer output shift. The addition of this shift value will allow easier quantization on AI85, since fractional bits can be used if weights do not span the full 8-bit range (most read-made quantization approaches require a weight scale or output shift).

In all cases, ensure that the quantizer writes out a checkpoint file that the Network Loader can read.

## Adding New Network Models and New Datasets to the Training Process

The following step is needed to add new network models:

1. Implement a new network model (see `models/ai84net.py` for an example). The file must include the `models` data structure that describes the model (name, minimum number of inputs, and whether it can handle 1D or 2D inputs). `models` can list multiple models in the same file.

The following steps are needed for new data formats and datasets:

1. Develop a data loader in PyTorch, see https://pytorch.org/tutorials/beginner/data_loading_tutorial.html. See `datasets/mnist.py` for an example.
2. Add the new data loader to a new file in the `datasets` directory (for example

`datasets/mnist.py`). The file must include the `datasets` data structure that describes the dataset and points to the new loader. `datasets` can list multiple datasets in the same file. The `regression` key in the structure can be set to `True` to automatically select the `--regression` command line argument.
The `input` key describes the dimensionality of the data, and the first dimension is passed as `num_channels` to the model, whereas the remaining dimensions are passed as `dimension`. For example, `'input': (1, 28, 28)` will be passed to the model as `num_channels=1` and `dimensions=(28,28)`.

3. The training/verification data is located (by default) in `data/DataSetName`, for example `data/CIFAR10`. The location can be overridden with the `--data target_directory` command line argument. The data loader is expected to download and preprocess the datasets as needed and install everything in the specified location.

Train the new network/new dataset. See `go_mnist.sh` for a command line example.

---

# Network Loader

*The network loader currently depends on PyTorch and Nervana's Distiller. This requirement will be removed in the long term.*

The network loader creates C code that programs the AI8X (for embedded execution, or RTL simulation, or CMSIS NN comparison). Additionally, the generated code contains sample input data and the expected output for the sample, as well as code that verifies the expected output.

The `cnn-gen.py` program needs two inputs:

1. A quantized checkpoint file, generated by the AI84 model quantization program `ai8xize.py`.
2. A YAML description of the network.

An example network description for the ai84net5 architecture and FashionMNIST is shown below:

```
# CHW (big data) configuration for FashionMNIST

arch: ai84net5
dataset: FashionMNIST

# Define layer parameters in order of the layer sequence
layers:
- pad: 1
  activate: ReLU
  out_offset: 0x2000
  processors: 0x0000000000000001
  data_format: CHW
- max_pool: 2
  pool_stride: 2
```

```
  pad: 2
  activate: ReLU
  out_offset: 0
  processors: 0xffffffffffffffff
- max_pool: 2
  pool_stride: 2
  pad: 1
  activate: ReLU
  out_offset: 0x2000
  processors: 0xffffffffffffffff
- avg_pool: 2
  pool_stride: 2
  pad: 1
  activate: ReLU
  out_offset: 0
  processors: 0x0fffffffffffffff
```

To generate an embedded AI84 demo in the `demos/FashionMNIST/` folder, use the following command line:

```
$ ./cnn-gen.py --verbose -L --top-level cnn --test-dir demos --prefix
FashionMNIST --checkpoint-file trained/ai84-mnist.pth.tar --config-file
networks/fashionmnist-chw.yaml --fc-layer --embedded-code
```

Running this command will combine the network described above with a fully connected software classification layer. The generated code will include all loading, unloading, and configuration steps.

To generate an RTL simulation for the same network and sample data in the directory `tests/fmnist-....` (where .... is an autogenerated string based on the network topology), use:

```
$ ./cnn-gen.py --verbose --autogen rtlsim --top-level cnn -L --test-dir rtlsim --
prefix fmnist --checkpoint-file trained/ai84-fashionmnist.pth.tar --config-file
networks/fashionmnist-chw.yaml
```

# Network Loader Configuration Language

Network descriptions are written in YAML (see https://en.wikipedia.org/wiki/YAML). There are two sections in each file --- global statements and a sequence of layer descriptions.

## Global Configuration

### `arch` (Mandatory)

`arch` specifies the network architecture, for example `ai84net5`. This key is matched against the architecture embedded in the checkpoint file.

**`bias` (Optional, Test Only)**

The `bias` configuration is only used for test data. *To use bias with trained networks, use the `bias` parameter in PyTorch's `nn.Module.Conv2d()` function. The converter tool will then automatically add bias parameters as needed.*

**`dataset` (Mandatory)**

`dataset` configures the data set for the network. This determines the input data size and dimensions as well as the number of input channels.

Data sets are for example `mnist`, `fashionmnist`, and `cifar-10`.

**`output_map` (Optional)**

The global `output_map`, if specified, overrides the memory instances where the last layer outputs its results. If not specified, this will be either the `output_processors` specified for the last layer, or, if that key does not exist, default to the number of processors needed for the output channels, starting at 0.

Example:
`output_map: 0x0000000000000ff0`

**`layers` (Mandatory)**

`layers` is a list that defines the per-layer description.

## Per-Layer Configuration

Each layer in the `layers` list describes the layer's processors, convolution type, activation, pooling, weight and output sizes, data input format, data memory offsets, and its processing sequence. Several examples are located in the `networks/` and `tests/` folders.

**`sequence` (Optional)**

This key allows overriding of the processing sequence. The default is `0` for the first layer, or the previous layer's sequence + 1 for other layers.

`sequence` numbers may have gaps. The software will sort layers by their numeric value, with the lowest value first.

**`processors` (Mandatory)**

`processors` specifies which processors will handle the input data. The processor map must match the number of input channels, and the input data format. For example, in CHW format, processors must be attached to different data memory instances.

Example:
`processors: 0x0000000000000111`

### `output_processors` (Optional)

`output_processors` specifies which data memory instances and 32-bit word offsets to use for the layer's output data. When not specified, this key defaults to the next layer's `processors`, or, for the last layer, to the global `output_map`.

### `out_offset` (Optional)

`out_offset` specifies the relative offset inside the data memory instance where the output data should be written to. When not specified, `out_offset` defaults to `0`.

Example:

```
out_offset: 0x2000
```

### `in_offset` (Optional)

`in_offset` specifies the offset into the data memory instances where the input data should be loaded from. When not specified, this key defaults to the previous layer's `out_offset`, or `0` for the first layer.

Example:

```
in_offset: 0x2000
```

### `output_width` (Optional)

On AI84, this value (if specified) has to be `8`.

On AI85, when **not** using an `activation`, the last layer can output `32` bits of unclipped data in Q25.7 format. The default is `8` bits.

Example:

```
output_width: 32
```

### `data_format` (Optional)

When specified for the first layer only, `data_format` can be either `chw`/`big` or `hwc`/`little`. The default is `hwc`. Note that the data format interacts with `processors`.

### `operation`

This key (which can also be specified using `op`, `operator`, or `convolution`) selects a layer's main operation after the optional input pooling.
When this key is not specified, a warning is displayed and `Conv2d` is selected. AI84 only supports `Conv1d` and `Conv2d`.

| Operation | Description |
| --- | --- |
| `Conv1d` | 1D convolution over an input composed of several input planes |
| `Conv2d` | 2D convolution over an input composed of several input planes |
| `ConvTranspose2d` | 2D transposed convolution (upsampling) over an input composed of several input planes |
| `None` or `Passthrough` | No operation |
| `Linear` or `FC` or `MLP` | Linear transformation to the incoming data |
| `Add` | Element-wise addition |
| `Sub` | Element-wise subtraction |
| `Xor` | Element-wise binary XOR |
| `Or` | Element-wise binary OR |

Element-wise operations default to two operands. This can be changed using the `operands` key.

**`eltwise` (Optional)**

Element-wise operations can also be added "in-flight" to `Conv2d`. In this case, the element-wise operation is specified using the `eltwise` key.

Example:
`eltwise: add`

**`pool_first` (Optional)**

When using both pooling and element-wise operations, pooling is performed first by default. Optionally, the element-wise operation can be performed before the pooling operation by setting `pool_first` to `False`.

Example:
`pool_first: false`

**`operands` (Optional)**

For any element-wise `operation`, this key configures the number of operands from `2` to `16` inclusive. The default is `2`.

Example:
`operation: add`

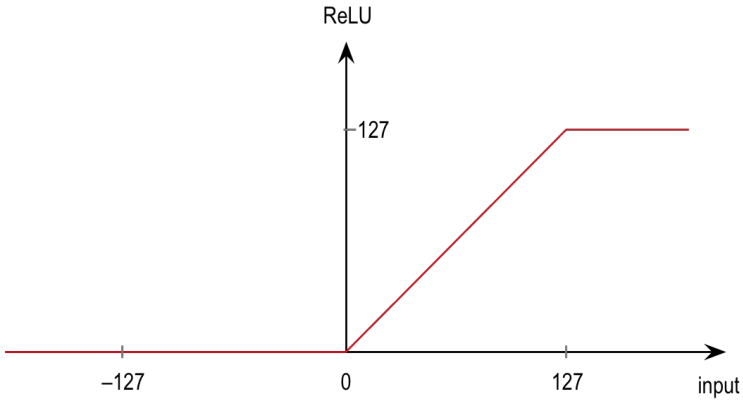
`operands: 4`

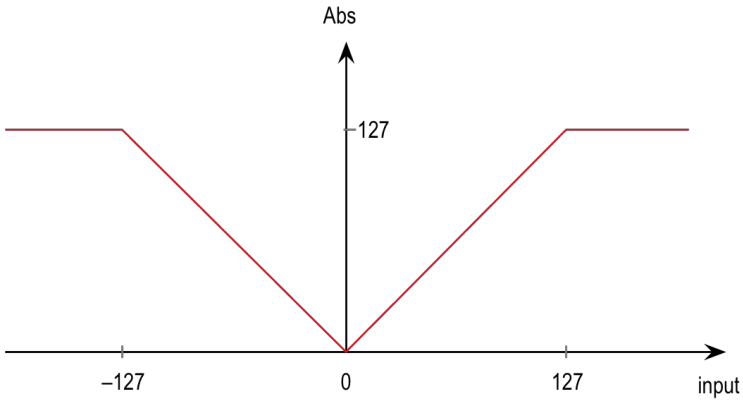
### `activate` (Optional)

This key describes whether to activate the layer output (the default is to not activate). When specified, this key must be `ReLU` or `Abs`. AI84 supports `ReLU` only.

Note that the output values are clipped (saturated) to $[0, +127]$. Because of this, `ReLU` behaves more similar to PyTorch's `nn.Hardtanh(min_value=0, max_value=127)` than to PyTorch's `nn.ReLU()`.

Note that `output_shift` can be used for (limited) linear activation.




### `quantization` (Optional)

On AI84, this key must always be `8` (the default value).

On AI85, this key describes the width of the weight memory in bits and can be `1`, `2`, `4`, or `8` (`8` is the default).

Example:
`quantization: 4`

**`output_shift` (Optional)**

On AI85, when `output_width` is 8, the 32-bit intermediate result can be shifted left or right before reduction to 8-bit. The value specified here is cumulative with the value generated from `quantization`.

The 32-bit intermediate result is multiplied by $2^{totalshift}$, where the total shift count must be within the range $[-8, +8]$, resulting in a factor of $[2^{-8}, 2^{8}]$ or $[0.00390625$ to $256]$.

| quantization | implicit shift | range for `output_shift` |
|---|---|---|
| 8-bit | 0 | $[-8, +8]$ |
| 4-bit | 1 | $[-9, +7]$ |
| 2-bit | 2 | $[-10, +6]$ |
| 1-bit | 4 | $[-11, +5]$ |

Using `output_shift` can help normalize data, particularly when using small weights.

Example:
`output_shift: 2`

**`kernel_size` (Optional)**

2D convolutions:

On AI84, this key must always be `3x3` (the default).
On AI85, `1x1` is also permitted.

1D convolutions:

On AI84, this key must always be `9` (the default).
On AI85, `1` through `9` are permitted.

Example:
`kernel_size: 1x1`

**`stride` (Optional)**

2D convolutions:

This key must always be `1`.

1D convolutions:

On AI84, this key must be set to `3`. On AI85, this key must be `1`.

**`pad` (Optional)**

`pad` sets the padding for the convolution.

- For `Conv2d`, this value can be `0`, `1` (the default), or `2`.
- For `Conv1d`, the value can be `0`, `3` (the default), or `6` on AI84 or `0`, `1`, `2` on AI85.
- For `Passthrough`, this value must be `0` (the default).

**`max_pool` (Optional)**

When specified, performs a `MaxPool` before the convolution. On AI84, `max_pool` is only supported for 2D convolutions. The pooling size can specified as an integer (when the value is identical for both dimensions, or for 1D convolutions), or as two values in order `[H, W]`.

Example:
`max_pool: 2`

**`avg_pool` (Optional)**

When specified, performs an `AvgPool` before the convolution. On AI84, `avg_pool` is only supported for 2D convolutions. The pooling size can specified as an integer (when the value is identical for both dimensions, or for 1D convolutions), or as two values in order `[H, W]`.

Example:
`avg_pool: 2`

**`pool_stride` (Optional)**

When performing a pooling operation, this key describes the pool stride. The pooling stride can be specified as an integer (when the value is identical for both dimensions, or for 1D convolutions), or as two values in order `[H, W]`. The default is `1` or `[1, 1]`.

Example:
`pool_stride: 2`

**`in_channels` (Optional)**

`in_channels` specifies the number of channels of the input data. This is usually automatically computed based on the weights file and the layer sequence. This key allows overriding of the number of channels. See also: `in_dim`.

Example:
`in_channels: 8`

**`in_dim` (Optional)**

`in_dim` specifies the dimensions of the input data. This is usually automatically computed based on the output of the previous layer or the layer(s) referenced by `in_sequences`. This key allows overriding of the automatically calculated dimensions. `in_dim` must be used when changing from 1D to 2D data or vice versa.

See also: `in_channels`.

Example:
```
in_dim: [64, 64]
```

**`in_sequences` (Optional)**

By default, a layer's input is the output of the previous layer. `in_sequences` can be used to point to the output of one or more arbitrary previous layers, for example when processing the same data using two different kernel sizes, or when combining the outputs of several prior layers. `in_sequences` can be specified as an integer (for a single input) or as a list (for multiple inputs). As a special case, `-1` is the input data. The `in_offset` and `out_offset` must be set to match the specified sequence.

Example:
```
in_sequences: [2, 3]
```

See the Fire example for a network that uses `in_sequences`.

**`out_channels` (Optional)**

`out_channels` specifies the number of channels of the output data. This is usually automatically computed based on the weights and layer sequence. This key allows overriding the number of output channels.

Example:
```
out_channels: 8
```

**`streaming` (Optional)**

`streaming` specifies that the layer is using streaming mode. this is necessary when the input data dimensions exceed the available data memory. When enabling `streaming`, all prior layers have to enable `streaming` as well. `streaming` can be enabled for up to 8 layers.

Example:
```
streaming: true
```

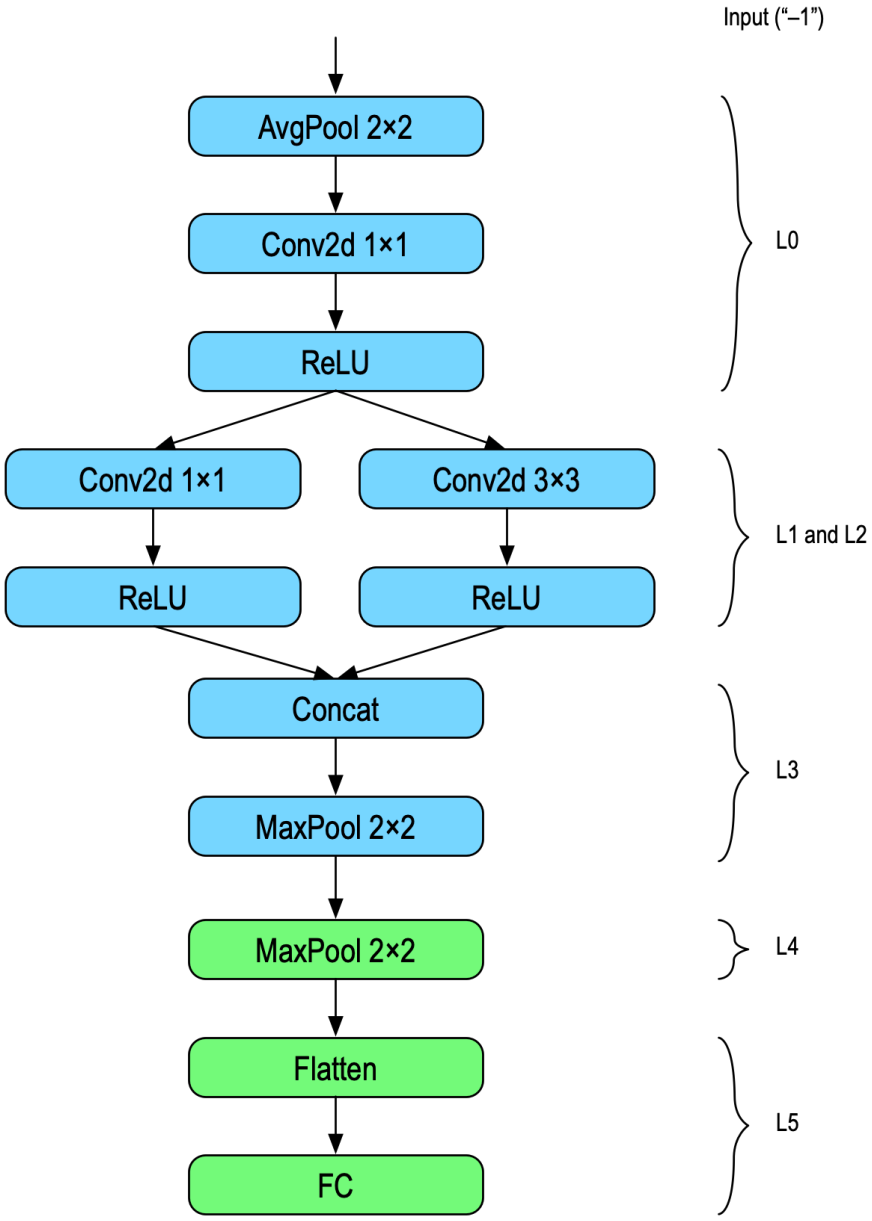
**`flatten` (Optional)**

`flatten` specifies that 2D input data should be transformed to 1D data for use by a `Linear` layer.

Example:
```
flatten: true
```

## Example

The following shows an example for a single “Fire” operation, the AI85/AI86 hardware layer numbers and its YAML description.




```
arch: ai85firetestnet
dataset: CIFAR-10
# Input dimensions are 3x32x32

layers:
### Fire
# Squeeze
```

```yaml
- avg_pool: 2
  pool_stride: 2
  pad: 0
  in_offset: 0x1000
  processors: 0x0000000000000007
  data_format: HWC
  out_offset: 0x0000
  operation: conv2d
  kernel_size: 1x1
  activate: ReLU
# Expand 1x1
- in_offset: 0x0000
  out_offset: 0x1000
  processors: 0x0000000000000030
  output_processors: 0x0000000000000f00
  operation: conv2d
  kernel_size: 1x1
  pad: 0
  activate: ReLU
# Expand 3x3
- in_offset: 0x0000
  out_offset: 0x1000
  processors: 0x0000000000000030
  output_processors: 0x000000000000f000
  operation: conv2d
  kernel_size: 3x3
  activate: ReLU
  in_sequences: 0
# Concatenate
- max_pool: 2
  pool_stride: 2
  in_offset: 0x1000
  out_offset: 0x0000
  processors: 0x000000000000ff00
  operation: none
  in_sequences: [1, 2]
### Additional layers
- max_pool: 2
  pool_stride: 2
  out_offset: 0x1000
  processors: 0x000000000000ff00
  operation: none
- flatten: true
  out_offset: 0x0000
  op: mlp
  processors: 0x000000000000ff00
```

```
  output_width: 32
```

## Adding Datasets to the Network Loader

Adding new datasets to the Network Loader is implemented as follows:

1. Provide network model, its YAML description and quantized weights. Place the YAML file in the `networks` directory, and the quantized weights in the `trained` directory.
2. Provide a sample input. The sample input is used to generate a known-answer test (self test). The sample input is provided as a NumPy "pickle" — add `sample_dset.npy` for the dataset named `dset` to the `tests` directory. This file can be generated by saving a sample in CHW format (no batch dimension) using `numpy.save()`.

For example, the CIFAR-10 3×32×32 image sample would be stored in `tests/sample_cifar-10.npy` in an `np.array` with shape `[3, 32, 32]` and datatype `<i8`. The file can be random, or can be obtained from the dataloader in the `forward()` function of the model.

### Generating a Sample Input

To generate a random sample input, use a short NumPy script. In the CIFAR-10 example:

```
import os
import numpy as np

a = np.random.randint(-128, 127, size=(3, 32, 32), dtype=np.int64)
np.save(os.path.join('tests', 'sample_cifar-10'), a, allow_pickle=False,
fix_imports=False)
```

### Saving a Sample Input from Training Data

It is a good idea to pick a sample where the quantized model computes the correct answer.

1. In the `ai8x-training` project, add the argument `--truncate-testset` to the `evaluate_cifar10.sh` script.
2. Run the modified `evaluate_cifar10.sh` and check that the answer is correct. The output should be similar to the following (`Top1` must be 100.0):

```
...
Dataset sizes:
        training=45000
        validation=5000
        test=1
--- test ---------------------
1 samples (256 per mini-batch)
Test: [    1/    1]    Loss 0.000000    Top1 100.000000    Top5 100.000000
==> Top1: 100.000    Top5: 100.000    Loss: 0.000
```

```
==> Confusion:
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]
 ...
```

3. Edit `models/ai84net.py` and add the following code to the `forward()` function of the `AI84Net5` class:

```
# ...
    def forward(self, x):  # pylint: disable=arguments-differ
        # Save sample
        import numpy as np
        a = x.cpu().numpy().squeeze().astype('int64')
        a = np.clip(a, -128, 127)
        np.save('sample_cifar-10', a, allow_pickle=False, fix_imports=False)

        # Original code
        x = self.conv1(x)
# ...
```

4. Run `evaluate_cifar10.sh` again and copy the saved `sample_cifar-10.npy` file.

### Generating C Code

Run `cnn-gen.py` with the new network and the new sample data. See `gen-demos-ai85.sh` for examples.

## Starting an Inference, Waiting for Completion, Multiple Inferences in Sequence

An inference is started by loading registers and kernels, loading the input, and enabling processing. This code is automatically generated—see the `cnn_load()`, `load_kernels()`, and `load_input()` functions. The sample data can be used as a self-checking feature on device power-up since the output for the sample data is known.

The AI85/AI86 accelerator can generate an interrupt on completion, and it will set a status bit (see `cnn_wait()`). The resulting data can now be unloaded from the accelerator (code for this is also auto-generated in `unload()`).

To run another inference, ensure all groups are disabled (stopping the state machine, as shown in `cnn_load()`). Next, load the new input data and start processing.

---

## CMSIS5 NN Emulation

The Network Loader tool can also create code that executes the same exact network using Arm's CMSISv5 Neural Networking library, optimized for Arm's Microcontroller DSP (reportedly 4.6× faster than executing on the CPU), see https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/converting-a-neural-network-for-arm-cortex-m-with-cmsis-nn and https://github.com/ARM-software/CMSIS_5 for the source code. A version of this repository is automatically checked out as part of the `ai8x-synthesis` project.

The results of the generated code have been verified to match AI84 exactly and may be used to demonstrate the efficacy of the custom CNN accelerator.
Note there are minor rounding errors in the CMSIS average pooling code when enabling SIMD (`-DARM_MATH_DSP`). If there are hard faults on the device, try limiting compiler optimizations to `-O1`.

The `Device/` folder contains a sample Makefile, and a custom fully connected layer in the file `arm_fully_connected_q7_q8p7_opt.c` that returns Q8.7 fixed-point outputs, and a custom `arm_softmax_q8p7_q15.c` which is aware of the fixed-point input (both of these files are also used for the software classification layer on AI84). Additionally, a number of files are provided that provide non-square pooling support, and activation support for more than 64 KiB of data (these files are not needed for AI8X hardware).
The `tornadocnn.h` header file is included which helps both embedded examples as well as CMSIS NN code.

For example, the following command would generate code that performs a CIFAR-10 inference from the `trained/ai84-cifar10.pth.tar` checkpoint file and the `cifar10-hwc.yaml` network description file:

```
(ai8x-synthesis) $ ./cnn-gen.py --top-level cnn --test-dir demos --prefix CIFAR-
10-Arm --checkpoint-file trained/ai84-cifar10.pth.tar --config-file
networks/cifar10-hwc.yaml --fc-layer --embedded-code --cmsis-software-nn
```

When compiling the CMSIS code, it may be necessary to disable compiler optimizations.

*Note*: The CMSIS code generator does not currently support the extended features of AI85 and AI86.

---

# Embedded Software Development Kits (SDKs)

## AI84 SDK

Use SVN to check out the AI84 SDK from https://svn.maxim-ic.com/svn/mcbusw/Hardware/Micro/AI84/Firmware/.

```
$ svn co https://svn.maxim-ic.com/svn/mcbusw/Hardware/Micro/AI84/Firmware/
AI84SDK
```

Additionally, the Arm embedded compiler is required, it is available from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads.

In order for the debugger to work, the OpenOCD `max32xxx` branch from https://github.com/MaximIntegratedMicros/openocd.git must be installed, and an ai84.cfg file must be installed in `/usr/local/share/openocd/scripts/target/` (or `C:\Maxim\Toolchain\share\openocd\scripts\target\` on Windows). A copy of the file is contained in the `hardware` folder of the `ai8x-synthesis` project.

Windows binaries for OpenOCD can be installed via https://www.maximintegrated.com/en/design/software-description.html/swpart=SFW0001500A. This download also includes a version of the Arm compilers.

To compile and install this OpenOCD version on macOS, use:

```
$ git clone https://github.com/MaximIntegratedMicros/openocd.git
$ cd openocd
$ git checkout max32xxx
$ MAKEINFO=true LIBTOOL=/usr/local/bin/glibtool ./configure --disable-dependency-
tracking --enable-dummy --enable-buspirate --enable-jtag_vpi --enable-remote-
bitbang --enable-legacy-ft2232_libftdi
$ make
$ make install
```

Additional SDK instructions can be found in a separate document, https://svn.maxim-ic.com/svn/mcbusw/Hardware/Micro/AI84/docs/trunk/AI84%20Test%20Board%20Setup%20Instructions.docx.

---

## AI85 SDK

Use SVN to check out the preliminary AI85 SDK from https://svn.maxim-ic.com/svn/mcbusw/Hardware/Micro/AI85/SDK/.

```
$ svn co https://svn.maxim-ic.com/svn/mcbusw/Hardware/Micro/AI85/SDK/ AI85SDK
```

The Arm embedded compiler can be downloaded from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads.

The RISC-V embedded compiler can be downloaded from https://github.com/xpack-dev-tools/riscv-none-embed-gcc-xpack/releases/.

In order for the debugger to work, the OpenOCD `max32xxx` branch from https://github.com/MaximIntegratedMicros/openocd.git must be installed (see above for more instructions). Working configuration files are and a `run-openocd-ai85` script are contained in the `hardware` folder of the `ai8x-synthesis` project.

`gen-demos-ai85.sh` will create code that is compatible with the SDK and copy it into the SDK directories (which must exist for each test before calling the generator script).

---

# AI85/AI86 Changes

The `ai8xize.py` quantization tool and the `cnn-gen.py` network generator both have an `--ai85` command line argument that enables code generation for the AI85 and AI86.

The `--ai85` option enables:

- Bias shift << 7.
- Per-layer support for 1, 2 and 4-bit weight sizes in addition to 8-bit weights (this is supported using the `quantization` keyword in the configuration file, and the configuration file can also be read by the quantization tool).
- A programmable shift left/shift right at the output of the convolution that allows for better use of the entire range of weight bits (`output_shift`).
- Support for many more pooling sizes and pooling strides, and larger limits for average pooling.
- Support for pooling without convolution (passthrough mode), and optional rounding for average pooling.
- 1D convolutions.
- 1×1 kernels for 2D convolutions.
- Transposed 2D convolutions (upsampling).
- Data "flattening", allowing the use of 1×1 kernels to emulate fully connected layers.
- In-flight element-wise addition, subtraction, and binary or/xor.
- Support for more weight memory, and more input and output channels.
- Support for non-square data and non-square pooling kernels.
- Support for 32-bit Q25.7 data output for last layer when not using activation.
- Support for streaming mode with FIFOs to allow for larger data sizes.
- Support for absolute value (`Abs`) activation.

---

# AHB Memory Addresses

The following tables show the AHB memory addresses for the AI85 accelerator:

## Data memory

Total: 512 KiB (16 instances of 8192 × 32)

| Group | Instance | Address Range |
|---|---|---|
| 0 | 0 | 0x50400000 - 0x50407FFF |
| 0 | 1 | 0x50408000 - 0x5040FFFF |
| 0 | 2 | 0x50410000 - 0x50417FFF |
| 0 | 3 | 0x50418000 - 0x5041FFFF |
| 0 | 0 | 0x50800000 - 0x50807FFF |
| 1 | 1 | 0x50808000 - 0x5080FFFF |
| 1 | 2 | 0x50810000 - 0x50817FFF |
| 1 | 3 | 0x50818000 - 0x5081FFFF |
| 2 | 0 | 0x50C00000 - 0x50C07FFF |
| 2 | 1 | 0x50C08000 - 0x50C0FFFF |
| 2 | 2 | 0x50C10000 - 0x50C17FFF |
| 2 | 3 | 0x50C18000 - 0x50C1FFFF |
| 3 | 0 | 0x51000000 - 0x51007FFF |
| 3 | 1 | 0x51008000 - 0x5100FFFF |
| 3 | 2 | 0x51010000 - 0x51017FFF |
| 3 | 3 | 0x51018000 - 0x5101FFFF |

## TRAM

Total: 384 KiB (64 instances of 3072 × 16)

| Group | Instance | Address Range* |
|---|---|---|
| 0 | 0 | 0x50110000 - 0x50112FFF |
| 0 | 1 | 0x50114000 - 0x50116FFF |
| | | |

| | | |
|---|---|---|
| 0 | 2 | 0x50118000 - 0x5011AFFF |
| 0 | 3 | 0x5011C000 - 0x5011EFFF |
| 0 | 4 | 0x50120000 - 0x50122FFF |
| 0 | 5 | 0x50124000 - 0x50126FFF |
| 0 | 6 | 0x50128000 - 0x5012AFFF |
| 0 | 7 | 0x5012C000 - 0x5012EFFF |
| 0 | 8 | 0x50130000 - 0x50132FFF |
| 0 | 9 | 0x50134000 - 0x50136FFF |
| 0 | 10 | 0x50138000 - 0x5013AFFF |
| 0 | 11 | 0x5013C000 - 0x5013EFFF |
| 0 | 12 | 0x50140000 - 0x50142FFF |
| 0 | 13 | 0x50144000 - 0x50146FFF |
| 0 | 14 | 0x50148000 - 0x5014AFFF |
| 0 | 15 | 0x5014C000 - 0x5014EFFF |
| 1 | 0 | 0x50510000 - 0x50512FFF |
| 1 | 1 | 0x50514000 - 0x50516FFF |
| 1 | 2 | 0x50518000 - 0x5051AFFF |
| 1 | 3 | 0x5051C000 - 0x5051EFFF |
| 1 | 4 | 0x50520000 - 0x50522FFF |
| 1 | 5 | 0x50524000 - 0x50526FFF |
| 1 | 6 | 0x50528000 - 0x5052AFFF |
| 1 | 7 | 0x5052C000 - 0x5052EFFF |
| 1 | 8 | 0x50530000 - 0x50532FFF |
| 1 | 9 | 0x50534000 - 0x50536FFF |
| 1 | 10 | 0x50538000 - 0x5053AFFF |
| 1 | 11 | 0x5053C000 - 0x5053EFFF |
| 1 | 12 | 0x50540000 - 0x50542FFF |

| | | |
|---|---|---|
| 1 | 13 | 0x50544000 - 0x50546FFF |
| 1 | 14 | 0x50548000 - 0x5054AFFF |
| 1 | 15 | 0x5054C000 - 0x5054EFFF |
| 2 | 0 | 0x50910000 - 0x50912FFF |
| 2 | 1 | 0x50914000 - 0x50916FFF |
| 2 | 2 | 0x50918000 - 0x5091AFFF |
| 2 | 3 | 0x5091C000 - 0x5091EFFF |
| 2 | 4 | 0x50920000 - 0x50922FFF |
| 2 | 5 | 0x50924000 - 0x50926FFF |
| 2 | 6 | 0x50928000 - 0x5092AFFF |
| 2 | 7 | 0x5092C000 - 0x5092EFFF |
| 2 | 8 | 0x50930000 - 0x50932FFF |
| 2 | 9 | 0x50934000 - 0x50936FFF |
| 2 | 10 | 0x50938000 - 0x5093AFFF |
| 2 | 11 | 0x5093C000 - 0x5093EFFF |
| 2 | 12 | 0x50940000 - 0x50942FFF |
| 2 | 13 | 0x50944000 - 0x50946FFF |
| 2 | 14 | 0x50948000 - 0x5094AFFF |
| 2 | 15 | 0x5094C000 - 0x5094EFFF |
| 3 | 0 | 0x50D10000 - 0x50D12FFF |
| 3 | 1 | 0x50D14000 - 0x50D16FFF |
| 3 | 2 | 0x50D18000 - 0x50D1AFFF |
| 3 | 3 | 0x50D1C000 - 0x50D1EFFF |
| 3 | 4 | 0x50D20000 - 0x50D22FFF |
| 3 | 5 | 0x50D24000 - 0x50D26FFF |
| 3 | 6 | 0x50D28000 - 0x50D2AFFF |

| | | |
|---|---|---|
| 3 | 7 | 0x50D2C000 - 0x50D2EFFF |
| 3 | 8 | 0x50D30000 - 0x50D32FFF |
| 3 | 9 | 0x50D34000 - 0x50D36FFF |
| 3 | 10 | 0x50D38000 - 0x50D3AFFF |
| 3 | 11 | 0x50D3C000 - 0x50D3EFFF |
| 3 | 12 | 0x50D40000 - 0x50D42FFF |
| 3 | 13 | 0x50D44000 - 0x50D46FFF |
| 3 | 14 | 0x50D48000 - 0x50D4AFFF |
| 3 | 15 | 0x50D4C000 - 0x50D4EFFF |

**using 32 bits of address space for each 16-bit memory*

## Kernel memory (“MRAM”)

Total: 432 KiB (64 instances of 768 × 72)

| Group | Instance | Address Range* |
|---|---|---|
| 0 | 0 | 0x50180000 - 0x50182FFF |
| 0 | 1 | 0x50184000 - 0x50186FFF |
| 0 | 2 | 0x50188000 - 0x5018AFFF |
| 0 | 3 | 0x5018c000 - 0x5018DFFF |
| 0 | 4 | 0x50190000 - 0x50191FFF |
| 0 | 5 | 0x50194000 - 0x50196FFF |
| 0 | 6 | 0x50198000 - 0x5019AFFF |
| 0 | 7 | 0x5019C000 - 0x5019DFFF |
| 0 | 8 | 0x501A0000 - 0x501A2FFF |
| 0 | 9 | 0x501A4000 - 0x501A6FFF |
| 0 | 10 | 0x501A8000 - 0x501AAFFF |
| 0 | 11 | 0x501AC000 - 0x501ADFFF |
| 0 | 12 | 0x501B0000 - 0x501B2FFF |

| | | |
|---|---|---|
| 0 | 13 | 0x501B4000 - 0x501B6FFF |
| 0 | 14 | 0x501B8000 - 0x501BAFFF |
| 0 | 15 | 0x501BC000 - 0x501BDFFF |
| 1 | 0 | 0x50580000 - 0x50582FFF |
| 1 | 1 | 0x50584000 - 0x50586FFF |
| 1 | 2 | 0x50588000 - 0x5058AFFF |
| 1 | 3 | 0x5058C000 - 0x5058DFFF |
| 1 | 4 | 0x50590000 - 0x50591FFF |
| 1 | 5 | 0x50594000 - 0x50596FFF |
| 1 | 6 | 0x50598000 - 0x5059AFFF |
| 1 | 7 | 0x5059C000 - 0x5059DFFF |
| 1 | 8 | 0x505A0000 - 0x505A2FFF |
| 1 | 9 | 0x505A4000 - 0x505A6FFF |
| 1 | 10 | 0x505A8000 - 0x505AAFFF |
| 1 | 11 | 0x505AC000 - 0x505ADFFF |
| 1 | 12 | 0x505B0000 - 0x505B2FFF |
| 1 | 13 | 0x505B4000 - 0x505B6FFF |
| 1 | 14 | 0x505B8000 - 0x505BAFFF |
| 1 | 15 | 0x505BC000 - 0x505BDFFF |
| 2 | 0 | 0x50980000 - 0x50982FFF |
| 2 | 1 | 0x50984000 - 0x50986FFF |
| 2 | 2 | 0x50988000 - 0x5098AFFF |
| 2 | 3 | 0x5098C000 - 0x5098DFFF |
| 2 | 4 | 0x50990000 - 0x50991FFF |
| 2 | 5 | 0x50994000 - 0x50996FFF |
| 2 | 6 | 0x50998000 - 0x5099AFFF |
| 2 | 7 | 0x5099C000 - 0x5099DFFF |

| | | |
|---|---|---|
| 2 | 8 | 0x509A0000 - 0x509A2FFF |
| 2 | 9 | 0x509A4000 - 0x509A6FFF |
| 2 | 10 | 0x509A8000 - 0x509AAFFF |
| 2 | 11 | 0x509AC000 - 0x509ADFFF |
| 2 | 12 | 0x509B0000 - 0x509B2FFF |
| 2 | 13 | 0x509B4000 - 0x509B6FFF |
| 2 | 14 | 0x509B8000 - 0x509BAFFF |
| 2 | 15 | 0x509BC000 - 0x509BDFFF |
| 3 | 0 | 0x50D80000 - 0x50D82FFF |
| 3 | 1 | 0x50D84000 - 0x50D86FFF |
| 3 | 2 | 0x50D88000 - 0x50D8AFFF |
| 3 | 3 | 0x50D8C000 - 0x50D8DFFF |
| 3 | 4 | 0x50D90000 - 0x50D91FFF |
| 3 | 5 | 0x50D94000 - 0x50D96FFF |
| 3 | 6 | 0x50D98000 - 0x50D9AFFF |
| 3 | 7 | 0x50D9C000 - 0x50D9DFFF |
| 3 | 8 | 0x50DA0000 - 0x50DA2FFF |
| 3 | 9 | 0x50DA4000 - 0x50DA6FFF |
| 3 | 10 | 0x50DA8000 - 0x50DAAFFF |
| 3 | 11 | 0x50DAC000 - 0x50DADFFF |
| 3 | 12 | 0x50DB0000 - 0x50DB2FFF |
| 3 | 13 | 0x50DB4000 - 0x50DB6FFF |
| 3 | 14 | 0x50DB8000 - 0x50DBAFFF |
| 3 | 15 | 0x50DBC000 - 0x50DBDFFF |

**using 128 bits of address space for each 72-bit memory*

## Bias memory

Total: 2 KiB (4 instances of 128 × 32)

| Group | Address Range |
|---|---|
| 0 | 0x50108000 - 0x50109FFF |
| 1 | 0x50508000 - 0x50509FFF |
| 2 | 0x50908000 - 0x50909FFF |
| 3 | 0x50D08000 - 0x50D09FFF |

---

# Updating the Project

To pull the latest code, in either project use:

```
$ git pull
$ git submodule update --init
$ pip3 install -U -r requirements.txt # or requirements-[cpu,cuda].txt
```

# Contributing Code

## Linting

Both projects are set up for `flake8`, `pylint`, and `mypy`. The line width is related to 100 (instead of the default of 80), and the number of lines per module was increased; configuration files are included in the projects.
Code should not generate any warnings in any of the tools (some of the components in the `ai8x-training` project will create warnings as they are based on third-party code).

`flake8`, `pylint` and `mypy` need to be installed into both virtual environments:

```
(ai8x-synthesis) $ pip3 install flake8 pylint mypy
```

## Submitting Changes

Do not try to push any changes into the master branch. Instead, create a local "feature" branch. The easiest way to do this is using a graphical client such as Fork. The command line equivalent is:

```
$ git checkout -b my-new-feature
$ git status
On branch my-new-feature
...
```

Commit changes and create a description of the changes:

```
$ git commit
```

Check the address for the Maxim server:

```
$ git remote -v
origin  https://first.last@gerrit.maxim-ic.com:8443/ai8x-synthesis (fetch)
origin  https://first.last@gerrit.maxim-ic.com:8443/ai8x-synthesis (push)
```

Install a commit hook:

```
$ GITDIR=$(git rev-parse --git-dir)
$ scp -p -P 29418 first.last@gerrit.maxim-ic.com:hooks/commit-msg
${GITDIR}/hooks/
```

And push the changes to Maxim's Gerrit server (do not change "master" to anything else even though the local branch is called "my-new-feature"):

```
$ git push https://first.last@gerrit.maxim-ic.com:8443/ai8x-synthesis
HEAD:refs/for/master
...
remote:
To URL
* [new branch] my-new-feature -> refs/for/master
```

Open the URL in a web browser and request a review for the change list.

---