MAX78000 Model Training and Synthesis

June 16, 2021

The Maxim Integrated AI project is comprised of five repositories:

1. **Start here**:
   Top Level Documentation
2. The software development kit (SDK), which contains drivers and example programs ready to run on the evaluation kits (EVkit and Feather):
   MAX78000_SDK
3. The training repository, which is used for deep learning *model development and training*:
   ai8x-training **(described in this document)**
4. The synthesis repository, which is used to *convert a trained model into C code* using the "izer" tool:
   ai8x-synthesis **(described in this document)**
5. The reference design repository, which contains host applications and sample applications for reference designs:
   refdes

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 this repository. The GitHub rendering of this document does not show the mathematical formulas nor the clickable table of contents.

MAX78000 Model Training and Synthesis
- Part Numbers
- Overview
- Installation
  - File System Layout
  - Prerequisites
    - Shared (Multi-User) and Remote Systems
    - Recommended Software
  - Project Installation
    - System Packages
      - macOS
      - Linux (Ubuntu)
        - RedHat Enterprise Linux / CentOS 8
    - Python 3.8
    - git Environment
    - Project Root
    - Nervana Distiller
    - Manifold
    - Windows Systems
  - Upstream Code

- Creating the Virtual Environment
  - Repository Branches
    - TensorFlow / Keras
- Updating to the Latest Version
  - Updates on Windows
  - Python Version Updates
- Synthesis Project
  - Repository Branches and Updates
- Embedded Software Development Kit (SDK)

MAX78000 Hardware and Resources

- Overview
- Data, Weights, and Processors
  - Weight Memory
  - Data Memory
    - Multi-Pass
- Streaming Mode
  - FIFOs
    - Standard FIFOs
    - Fast FIFO
- Number Format
  - Rounding
  - Addition
  - Saturation and Clipping
  - Multiplication
  - Sign Bit
- Channel Data Formats
  - HWC (Height-Width-Channels)
  - CHW (Channels-Height-Width)
  - Considerations for Choosing an Input Format
  - CHW Input Data Format and Consequences for Weight Memory Layout
- Active Processors and Layers
- Layers and Weight Memory
  - Bias Memories
- Weight Storage Example
- Example: `Conv2D`
- Limitations of MAX78000 Networks
- Fully Connected (Linear) Layers
- Upsampling (Fractionally-Strided 2D Convolutions)

Model Training and Quantization

- Command Line Arguments
  - ONNX Model Export
- Observing GPU Resources
- Custom nn.Modules
  - List of Predefined Modules
  - Dropout
  - view and reshape

- Support for Quantization
  - Data
  - Weights: Quantization-Aware Training (QAT)
- Batch Normalization
- Model Comparison and Feature Attribution
- TensorBoard
  - Remote Access to TensorBoard
- SHAP — SHapely Additive exPlanations
- BatchNorm Fusing
- Command Line Arguments
- Quantization
- Quantization-Aware Training (QAT)
- Post-Training Quantization
- Command Line Arguments
- Example and Evaluation
- Alternative Quantization Approaches
- Adding New Network Models and New Datasets to the Training Process
- Data Loader
  - Normalizing Input Data
- `datasets` Data Structure
- Training and Verification Data
- Training Process
- Netron — Network Visualization
- Neural Architecture Search (NAS)
- Introduction
- Once-for-All
  - Stages and Levels in the MAX78000 Implementation
- Usage
  - Important Considerations for NAS
- NAS Model Definition
- NAS Output

Network Loader (AI8Xize)

- Command Line Arguments
- YAML Network Description
- Network Loader Configuration Language
- Purpose of the YAML Network Description
  - Data Memory Ping-Pong
- Global Configuration
  - `arch` (Mandatory)
  - `bias` (Optional, Test Only)
  - `dataset` (Mandatory)
  - `output_map` (Optional)
  - `layers` (Mandatory)
- Per-Layer Configuration
  - `sequence` (Optional)
  - `processors` (Mandatory)

- `output_processors` (Optional)
- `out_offset` (Optional)
- `in_offset` (Optional)
- `output_width` (Optional)
- `data_format` (Optional)
- `operation`
- `eltwise` (Optional)
- `pool_first` (Optional)
- `operands` (Optional)
- `activate` (Optional)
- `quantization` (Optional)
- `output_shift` (Optional)
- `kernel_size` (Optional)
- `stride` (Optional)
- `pad` (Optional)
- `max_pool` (Optional)
- `avg_pool` (Optional)
- `pool_stride` (Optional)
- `in_channels` (Optional)
- `in_dim` (Optional)
- `in_sequences` (Optional)
- `out_channels` (Optional)
- `streaming` (Optional)
- `flatten` (Optional)
- `write_gap` (Optional)
- `bias_group` (Optional)

Dropout and Batch Normalization

Example

Residual Connections

Adding New Models and New Datasets to the Network Loader

Generating a Random Sample Input

Saving a Sample Input from Training Data

Evaluate the Quantized Weights with the New Dataset and Model

Generating C Code

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

Overview of the Functions in main.c

Overview of the Generated API Functions

Softmax, and Data Unload in C

Generated Files and Upgrading the CNN Model

Contents of the device-all Folder

Determining the Compiled Flash Image Size

Handling Linker Flash Section Overflows

Debugging Techniques

Energy Measurement

Further Information

AHB Memory Addresses

Data memory
TRAM
Kernel memory (“MRAM”)
Bias memory
Contributing Code
Linting
Submitting Changes

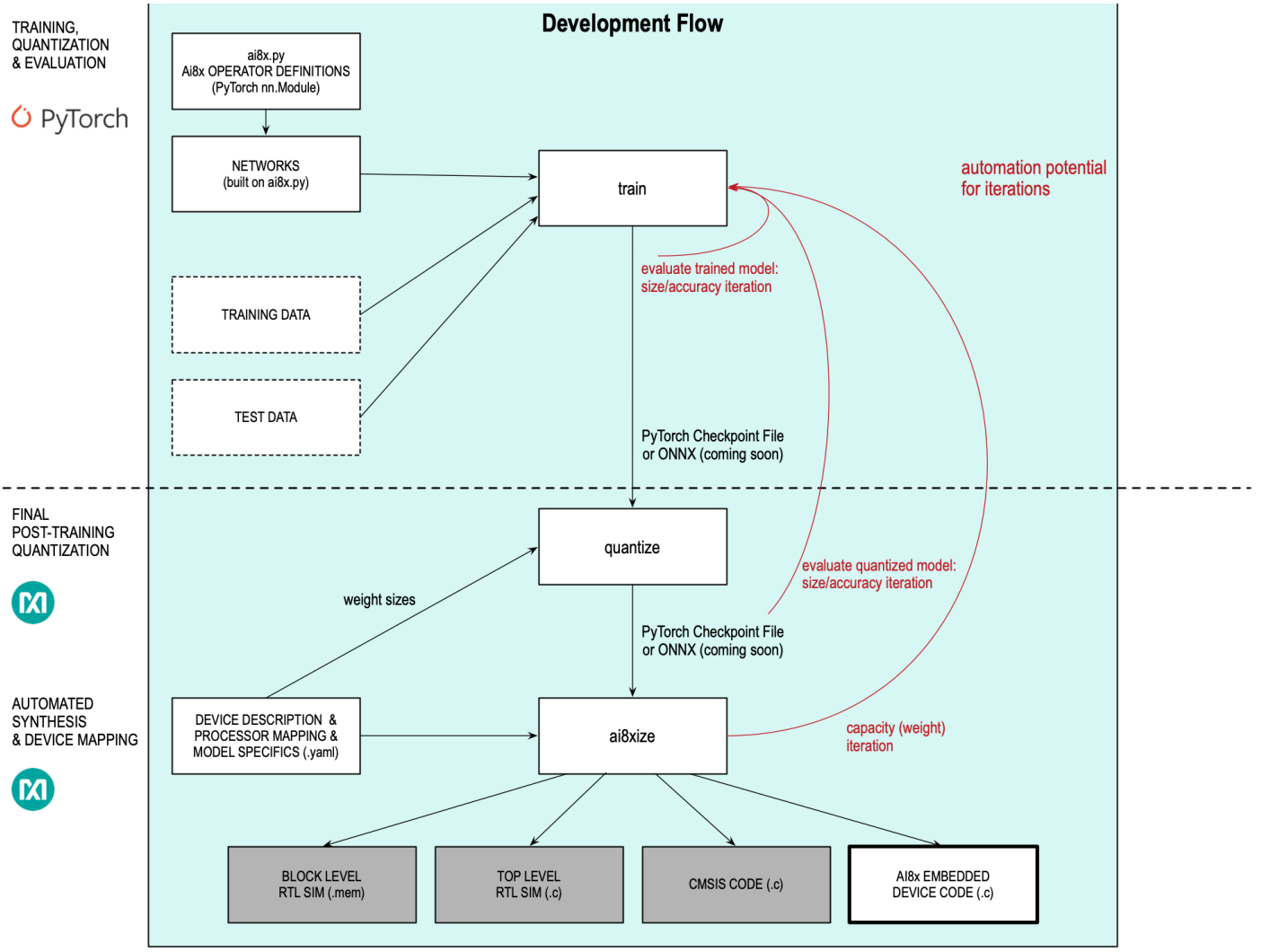
Part Numbers

This document covers several of Maxim’s ultra-low power machine learning accelerator systems. They are sometimes referred to by their die types. The following shows the die types and their corresponding part numbers:

Die Type	Part Number(s)
AI84	*Unreleased test chip*
AI85	**MAX78000**
AI87	MAX78002 (in development)

Overview

The following graphic shows an overview of the development flow:



Installation

File System Layout

Including the SDK, the expected/resulting file system layout will be:

```
1 ..../ai8x-training/
2 ..../ai8x-synthesis/
3 ..../ai8x-synthesis/sdk/
```

where "...." is the project root, for example `~/Documents/Source/AI`.

Prerequisites

This software currently supports Ubuntu Linux 18.04 LTS and 20.04 LTS. The server version is sufficient, see https://ubuntu.com/download/server. *Alternatively, Ubuntu Linux can also be used inside the Windows Subsystem for Linux (WSL2) by following https://docs.nvidia.com/cuda/wsl-user-guide/. However, please note that WSL2 with CUDA is a pre-release and unexpected behavior may occur.*

When going beyond simple models, model training does not work well without CUDA hardware acceleration. The network loader ("izer") does not require CUDA, and very simple models can also be trained on systems without CUDA.

Recommendation: Install the latest version of CUDA 11 on Ubuntu 20.04 LTS. See https://developer.nvidia.com/cuda-toolkit-archive.

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

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 (or alternatively, the more powerful `tmux`) 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. In addition, screen can log all console output to a text file.

Example:

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

`man screen` and `man tmux` describe the software in more detail.

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, $50), https://git-fork.com
   or GitHub Desktop (Graphical Git Client, Free), https://desktop.github.com

5. Beyond Compare (Diff and Merge Tool, $60), https://scootersoftware.com

Project Installation

System Packages

Some additional system packages are required, and installation of these additional packages requires administrator privileges. Note that this is the only time administrator privileges are required.

macOS

On macOS (no CUDA support available) use:

```
$ brew install libomp libsndfile tcl-tk
```

Linux (Ubuntu)

```
$ 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
```

RedHat Enterprise Linux / CentOS 8

While Ubuntu 20.04 LTS is the supported distribution, the MAX78000 software packages run fine on all modern Linux distributions that also support CUDA. The *apt-get install* commands above must be replaced with distribution specific commands and package names. Unfortunately, there is no obvious 1:1 mapping between package names from one distribution to the next. The following example shows the commands needed for RHEL/CentOS 8.

Two of the required packages are not in the base repositories. Enable the EPEL and PowerTools repositories:

```
$ sudo dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-
8.noarch.rpm
$ sudo dnf config-manager --set-enabled powertools
```

Proceed to install the required packages:

```
$ sudo dnf group install "Development Tools"
$ sudo dnf install openssl-devel zlib-devel \
  bzip2-devel readline-devel sqlite-devel wget llvm \
  xz-devel tk tk-devel libffi-devel \
  libsndfile libsndfile-devel portaudio-devel
```

Python 3.8

The software in this project uses Python 3.8.10 or a later 3.8.x version.

It is not necessary to install Python 3.8.10 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
```

On Linux:

```
$ curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-installer | bash  # NOTE: Verify contents of the script before running it!!
```

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

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

If you use zsh as the shell (default on macOS), add these same commands to `~/.zprofile` or `~/.zshrc` in addition to adding them to the bash startup scripts.

Next, close the Terminal, open a new Terminal and install Python 3.8.10.

On macOS:

```
$ env \
  PATH="$(brew --prefix tcl-tk)/bin:$PATH" \
  LDFLAGS="-L$(brew --prefix tcl-tk)/lib" \
  CPPFLAGS="-I$(brew --prefix tcl-tk)/include" \
  PKG_CONFIG_PATH="$(brew --prefix tcl-tk)/lib/pkgconfig" \
  CFLAGS="-I$(brew --prefix tcl-tk)/include" \
  PYTHON_CONFIGURE_OPTS="--with-tcltk-includes='-I$(brew --prefix tcl-tk)/include' --with-tcltk-libs='-L$(brew --prefix tcl-tk)/lib -ltcl8.6 -ltk8.6'" \
  pyenv install 3.8.10
```

On Linux:

```
$ pyenv install 3.8.10
```

git Environment

git Environment

If the local git environment has not been previously configured, add the following commands to configure e-mail and name. The e-mail must match GitHub (including upper/lower case):

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

Project Root

For convenience, define a shell variable named `AI_PROJECT_ROOT`:

```
$ export AI_PROJECT_ROOT="$HOME/Documents/Source/AI"
```

Add this line to `~/.profile` (and on macOS, to `~/.zprofile`).

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 as a git sub-module with the other packages.

Manifold

Manifold is a model-agnostic visual debugging tool for machine learning. The Manifold guide shows how to integrate this optional package into the training software.

Windows Systems

Windows/MS-DOS is not supported for training networks at this time. *This includes the Windows Subsystem for Linux (WSL) since it currently lacks CUDA support.*

Upstream Code

Change to the project root and run the following commands. Use your GitHub credentials if prompted.

```
$ cd $AI_PROJECT_ROOT
$ git clone https://github.com/MaximIntegratedAI/ai8x-training.git
$ git clone https://github.com/MaximIntegratedAI/ai8x-synthesis.git
```

Creating the Virtual Environment

Creating the Virtual Environment

To create the virtual environment and install basic wheels:

```
$ cd ai8x-training
```

If you want to use the “develop” branch, switch to “develop” using this optional step:

```
$ git checkout develop  # optional
```

Then continue with the following:

```
$ git submodule update --init
$ pyenv local 3.8.10
$ python3 -m venv .
$ source bin/activate
(ai8x-training) $ pip3 install -U pip wheel setuptools
```

The next step differs depending on whether the system uses Linux with CUDA 11.x, or any other setup.

For CUDA 11.x on Linux:

```
(ai8x-training) $ pip3 install -r requirements-cu11.txt
```

For all other systems, including CUDA 10.2 on Linux:

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

Repository Branches

By default, the main branch is checked out. This branch has been tested more rigorously than the `develop` branch. `develop`, on the other hand, contains the latest improvements to the project. To switch to `develop`, use the following command:

```
(ai8x-training) $ git checkout develop
```

TensorFlow / Keras

Support for TensorFlow / Keras is currently in the `develop-tf` branch.

Updating to the Latest Version

Updating to the Latest Version

After additional testing, `develop` is merged into the main branch at regular intervals.

After a small delay of typically a day, a "Release" tag is created on GitHub for all non-trivial merges into the main branch. GitHub offers email alerts for all activity in a project, or for new releases only. Subscribing to releases only substantially reduces email traffic.

Note: Each "Release" automatically creates a code archive. It is recommended to use a git client to access (pull from) the main branch of the repository using a git client instead of downloading the archives.

In addition to code updated in the repository itself, submodules and Python libraries may have been updated as well.

Major upgrades (such as updating from PyTorch 1.7 to PyTorch 1.8) are best done by removing all installed wheels. This can be achieved most easily by creating a new folder and starting from scratch at Upstream Code. Starting from scratch is also recommended when upgrading the Python version.

For minor updates, pull the latest code and install the updated wheels:

```
(ai8x-training) $ git pull
(ai8x-training) $ git submodule update --init
(ai8x-training) $ pip3 install -U pip setuptools
(ai8x-training) $ pip3 install -U -r requirements.txt # or requirements-cu11.txt with CUDA 11.x
```

Updates on Windows

On Windows, please *also* use the Maintenance Tool as documented in the Maxim Micro SDK (MaximSDK) Installation and Maintenance User Guide. The Maintenance Tool updates the SDK.

Python Version Updates

Updating Python may require updating `pyenv` first. Should `pyenv install 3.8.10` fail,

```
$ pyenv install 3.8.10
python-build: definition not found: 3.8.10
```

then `pyenv` must be updated. On macOS, use:

```
$ brew update && brew upgrade pyenv
...
$
```

On Linux, use:

```
$ cd $(pyenv root) && git pull && cd -
remote: Enumerating objects: 19021, done.
...
$
```

The update should now succeed:

```
$ pyenv install 3.8.10
Downloading Python-3.8.10.tar.xz...
-> https://www.python.org/ftp/python/3.8.10/Python-3.8.10.tar.xz
Installing Python-3.8.10...
...
$ pyenv local 3.8.10
```

Synthesis Project

The `ai8x-synthesis` project does not require CUDA.

Start by deactivating the `ai8x-training` environment if it is active.

```
(ai8x-training) $ deactivate
```

Then, create a second virtual environment:

```
$ cd $AI_PROJECT_ROOT
$ cd ai8x-synthesis
```

If you want to use the "develop" branch, switch to "develop" using this optional step:

```
$ git checkout develop  # optional
```

Then continue:

```
$ git submodule update --init
$ pyenv local 3.8.10
$ python3 -m venv .
$ source bin/activate
(ai8x-synthesis) $ pip3 install -U pip setuptools
(ai8x-synthesis) $ pip3 install -r requirements.txt
```

Repository Branches and Updates

Repository Branches and Updates

Branches and updates for `ai8x-synthesis` are handled similarly to the `ai8x-training` project.

Installation is now Complete

With the installation of Training and Synthesis projects completed it is important to remember to activate the proper Python virtual environment when switching between projects. If scripts begin failing in a previously working environment, the cause might be that the incorrect virtual environment is active or that no virtual environment has been activated.

Embedded Software Development Kit (SDK)

The MAX78000 SDK is a git submodule of ai8x-synthesis. It is checked out automatically to a version compatible with the project into the folder `sdk`.

If the embedded C compiler is run on Windows instead of Linux or macOS, ignore this section *and install the Maxim SDK executable, see https://github.com/MaximIntegratedAI/MaximAI_Documentation.*

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 SDK has been tested with version 9-2019-q4-major of the embedded Arm compiler. Newer versions may or may not work correctly.*

The RISC-V embedded compiler can be downloaded from https://github.com/xpack-dev-tools/riscv-none-embed-gcc-xpack/releases/. *The SDK has been tested with version 8.3.0-1.1 of the RISC-V embedded compiler. Newer versions may or may not work correctly.*

Add the following to your `~/.profile` (and on macOS, to `~/.zprofile`), adjusting for the actual `PATH` to the compilers:

```
echo $PATH | grep -q -s "/usr/local/gcc-arm-none-eabi-9-2019-q4-major/bin"
if [ $? -eq 1 ] ; then
    PATH=$PATH:/usr/local/gcc-arm-none-eabi-9-2019-q4-major/bin
    export PATH
    ARMGCC_DIR=/usr/local/gcc-arm-none-eabi-9-2019-q4-major
    export ARMGCC_DIR
fi

echo $PATH | grep -q -s "/usr/local/riscv-none-embed-gcc/8.3.0-1.1/bin"
if [ $? -eq 1 ] ; then
    PATH=$PATH:/usr/local/riscv-none-embed-gcc/8.3.0-1.1/bin
    export PATH
    RISCVGCC_DIR=/usr/local/riscv-none-embed-gcc/8.3.0-1.1
    export RISCVGCC_DIR
fi
```

The debugger requires OpenOCD. On Windows, an OpenOCD executable is installed with the SDK. On macOS

and Linux, the OpenOCD fork from https://github.com/MaximIntegratedMicros/openocd.git must be used. An Ubuntu Linux binary is available at https://github.com/MaximIntegratedAI/MAX78000_SDK/blob/master/Tools/OpenOCD/openocd. *Note: A copy of the configuration files and a* `run-openocd-maxdap` *script are contained in the* `hardware` *folder of the* `ai8x-synthesis` *project.*

`gen-demos-max78000.sh` will create code that is compatible with the SDK and copy it into the SDK's Example directories.

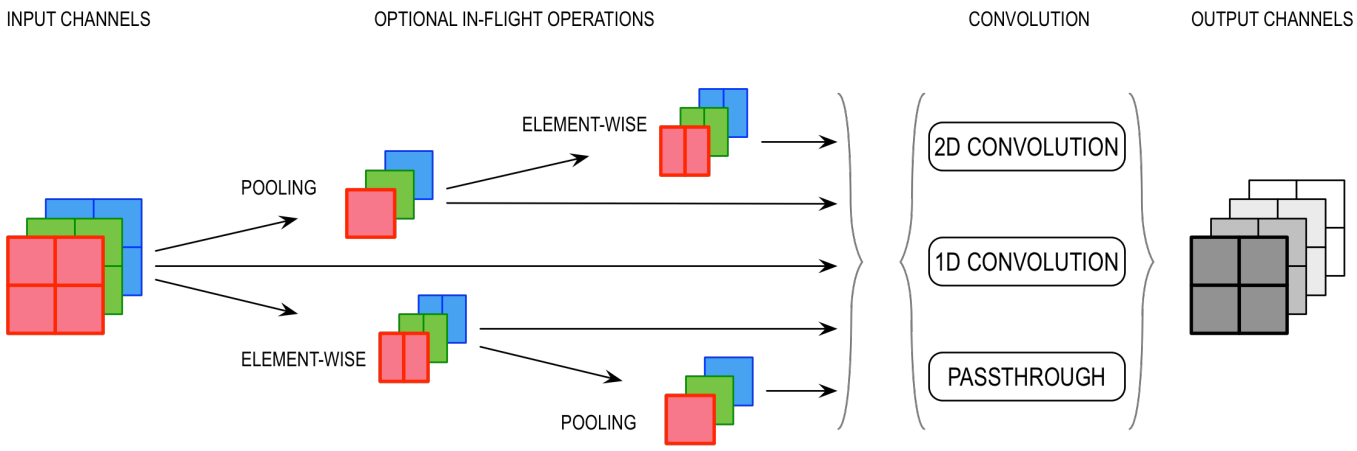
MAX78000 Hardware and Resources

MAX78000/MAX78002 are embedded accelerators. Unlike GPUs, MAX78000/MAX78002 do not have gigabytes of memory, and cannot support arbitrary data (image) sizes.

Overview

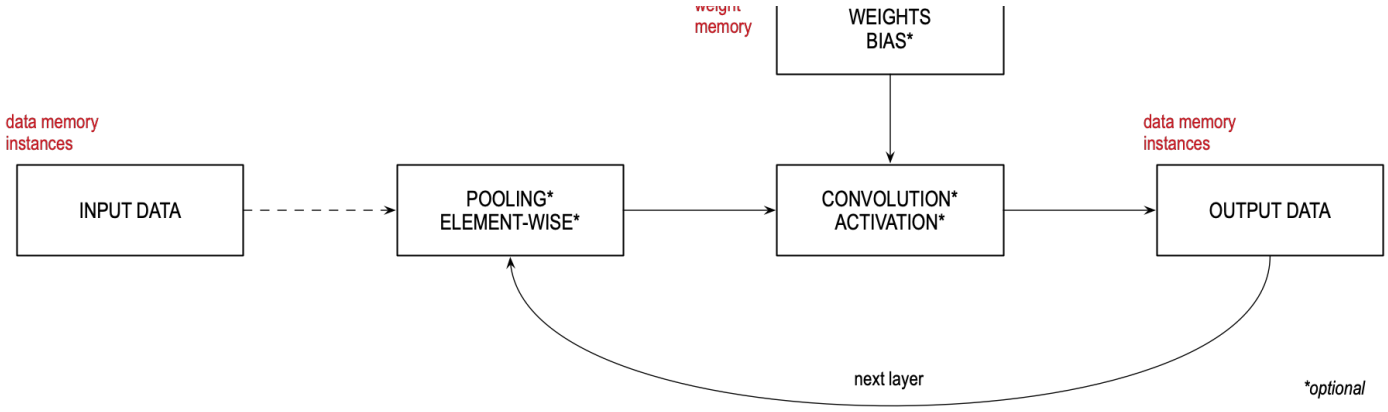
A typical CNN operation consists of pooling followed by a convolution. While these are traditionally expressed as separate layers, pooling can be done "in-flight" on MAX78000/MAX78002 for greater efficiency.

To minimize data movement, the accelerator is optimized for convolutions with in-flight pooling on a sequence of layers. MAX78000 and MAX78002 also support in-flight element-wise operations, pass-through layers and 1D convolutions (without element-wise operations):



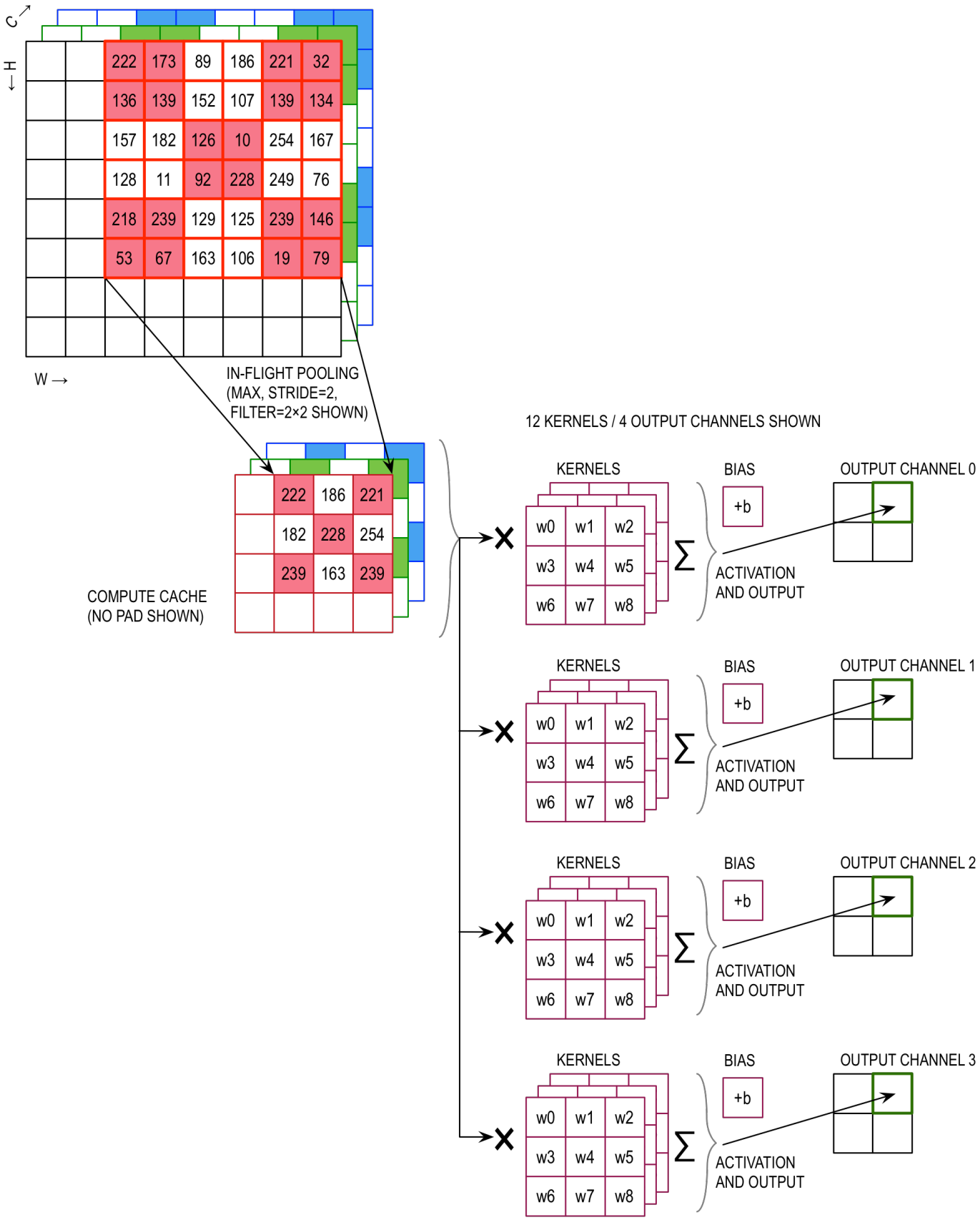
The MAX78000/MAX78002 accelerators contain 64 parallel processors. There are four groups that contain 16 processors each.

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, Weights, and Processors

Data memory, weight memory, and processors are interdependent.

In the MAX78000/MAX78002 accelerator, processors are organized as follows:

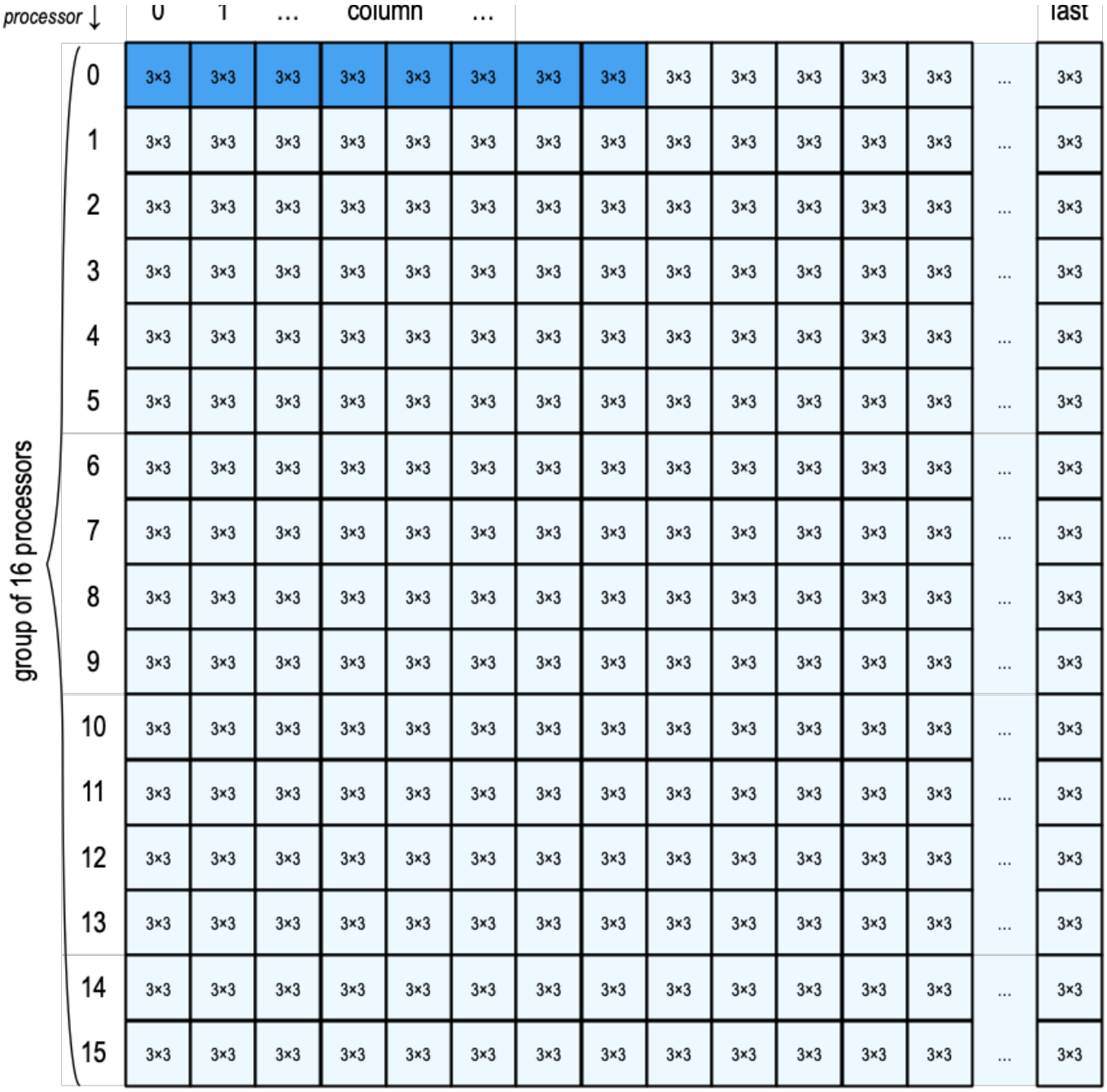
- Each processor is connected to its own dedicated weight memory instance.
- Four processors share one data memory instance.
- A group of sixteen processors shares certain common controls and can be operated as a slave to another group, or independently/separately.

Any given processor has visibility of:

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

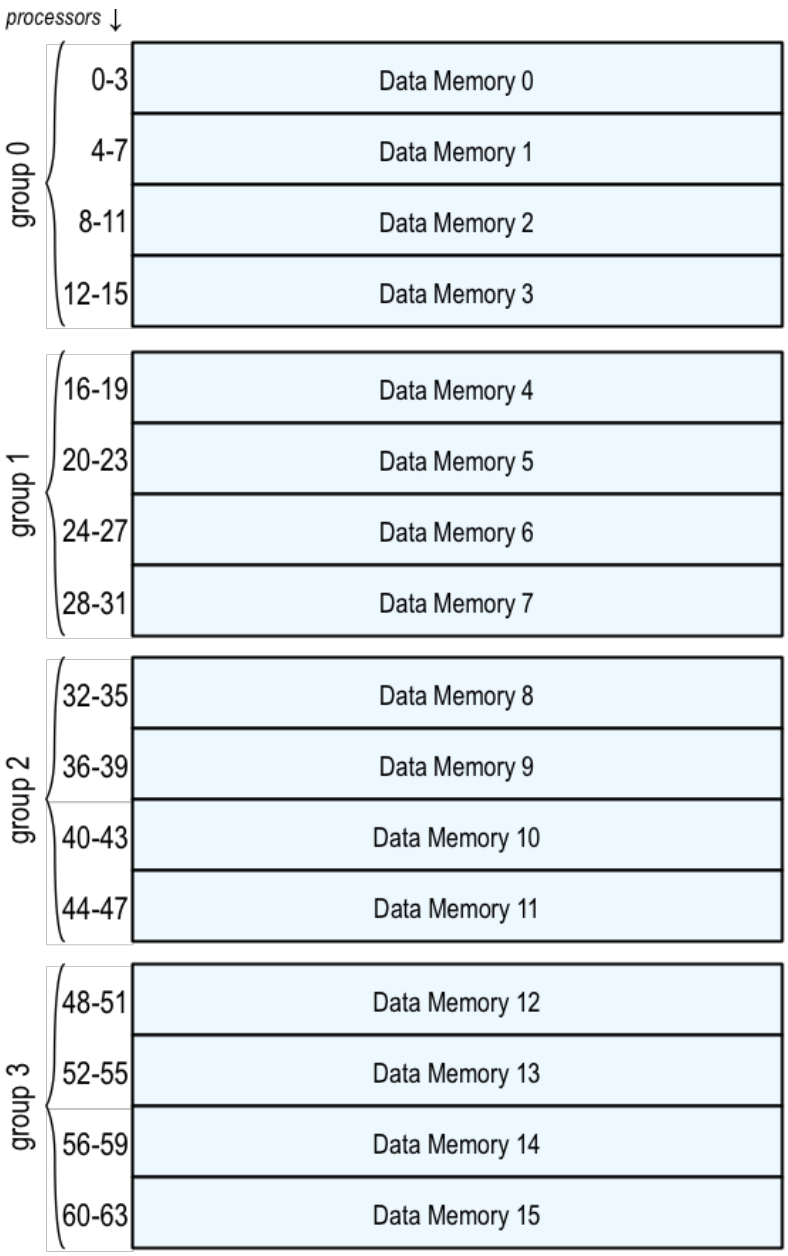
Weight Memory

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).

Multi-Pass

When data has more channels than active processors, “multi-pass” is used. Each processor works on more than one channel, using multiple sequential passes, and each data memory holds more than four channels.

As data is read using multiple passes, and all available processor work in parallel, the first pass reads channels 0 through 63, the second pass reads channels 64 through 127, etc., assuming 64 processors are active.

For example, if 192-channel data is read using 64 active processors, Data Memory 0 stores three 32-bit words: channels 0, 1, 2, 3 in the first word, 64, 65, 66, 67 in the second word, and 128, 129, 130, 131 in the third word. Data Memory 1 stores channels 4, 5, 6, 7 in the first word, 68, 69, 70, 71 in the second word, and 132, 133, 134, 135 in the third word, and so on. The first processor processes channel 0 in the first pass, channel 64 in the second pass, and channel 128 in the third pass.

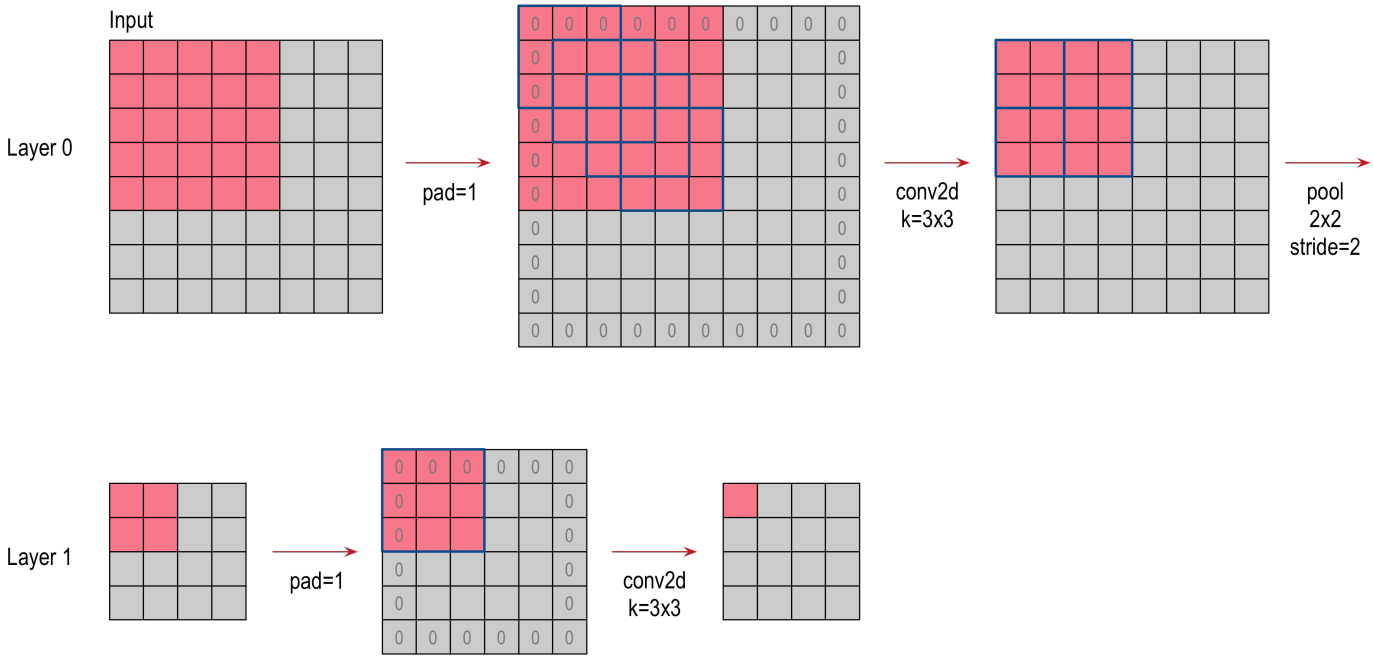
Note: Multi-pass also works with channel counts that are not a multiple of 64, and can be used with fewer than 64 active processors.

Note: For all multi-pass cases, the processor count per pass is rounded up to the next multiple of 4.

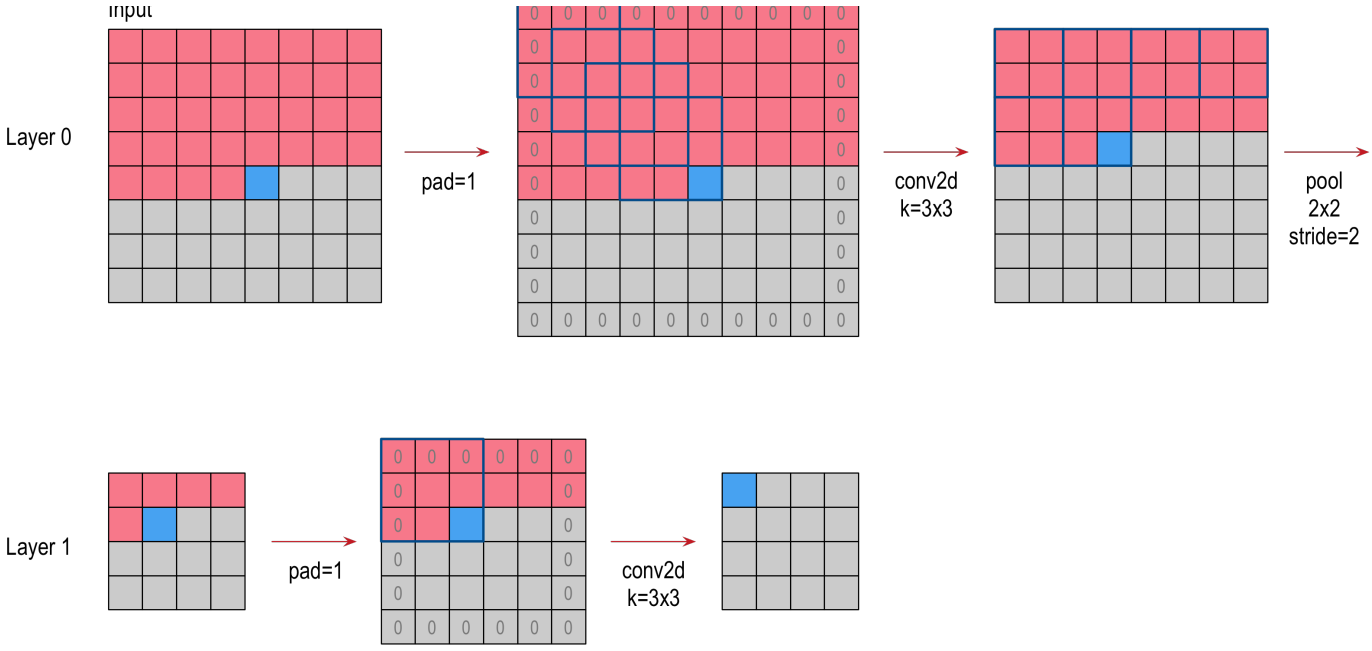
Streaming Mode

The machine also implements a streaming mode. Streaming allows input data dimensions that exceed the available per-channel data memory in the accelerator.

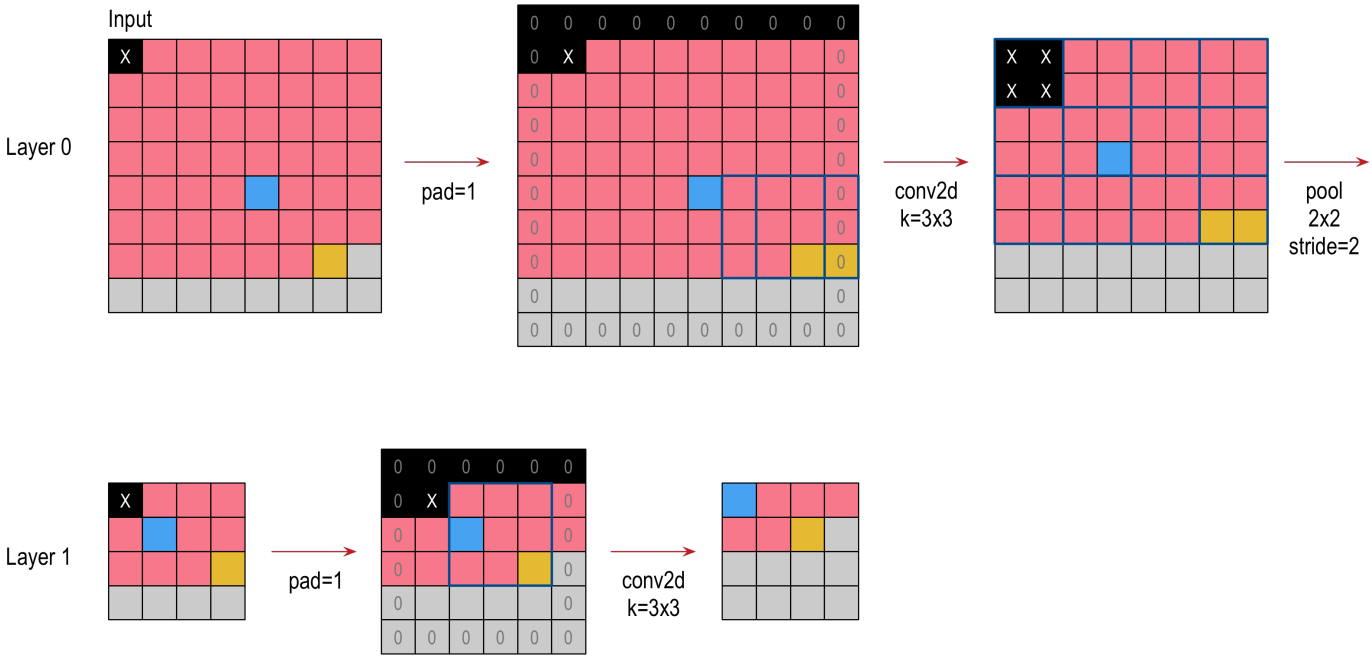
The following illustration shows the basic principle: In order to produce the first output pixel of the second layer, not all data needs to be present at the input. In the example, a 5×5 input needs to be available.



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.

For concrete examples on how to implement streaming mode with a camera, please see the [Camera Streaming

Guide

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 MAX78000/MAX78002 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 `ai8xize.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.

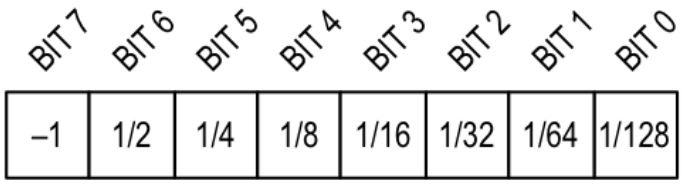
The fast FIFO is selected using the `--fast-fifo` argument for `ai8xize.py`.

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 MAX78000/MAX78002, *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 Q17.14 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.

In other cases, using bias in convolutional layers does not improve inference performance. In particular, Quantization-Aware Training (QAT) optimizes the weight distribution, possibly deteriorating the distribution of the bias values.

Rounding

MAX78000/MAX78002 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 MAX78000/MAX78002 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 MAX78000/MAX78002 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 MAX78000/MAX78002 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:

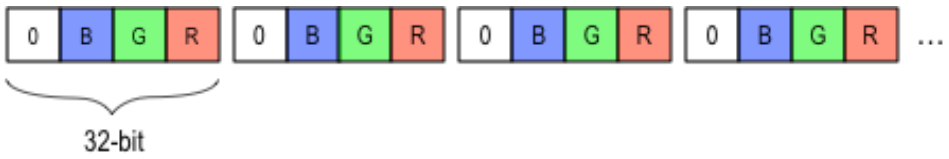
$w_0 = 127/128 \rightarrow 01111111$
$w_1 = 1/128 \rightarrow 00000001$
$w_0 * w_1 = 128/128 \rightarrow saturation \rightarrow 01111111 (= 127/128)$

Channel Data Formats

HWC (Height-Width-Channels)

All internal data are stored in HWC format, four channels per 32-bit word. Assuming 3-color (or 3-channel) input, one byte of the 32-bit word will be unused. The highest frequency in this data format is the channel, so the channels are interleaved.

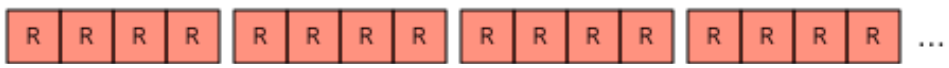
Example:

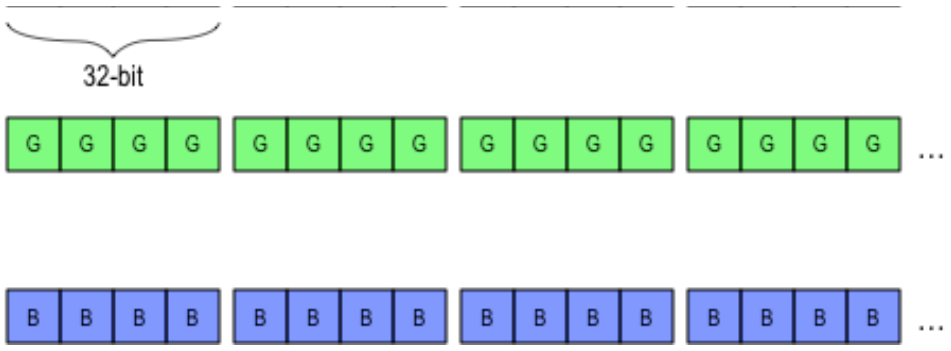


CHW (Channels-Height-Width)

The input layer can alternatively also use the CHW format (a sequence of channels). The highest frequency in this data format is the width or X-axis (W), and the lowest frequency is the channel. Assuming an RGB input, all red pixels are followed by all green pixels, followed by all blue pixels.

Example:





Considerations for Choosing an Input Format

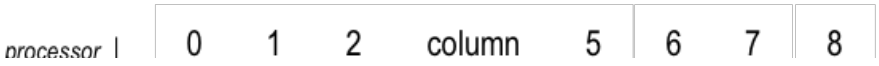
The accelerator supports both HWC and CHW input formats to avoid unnecessary data manipulation. Internal layers always use the HWC format.

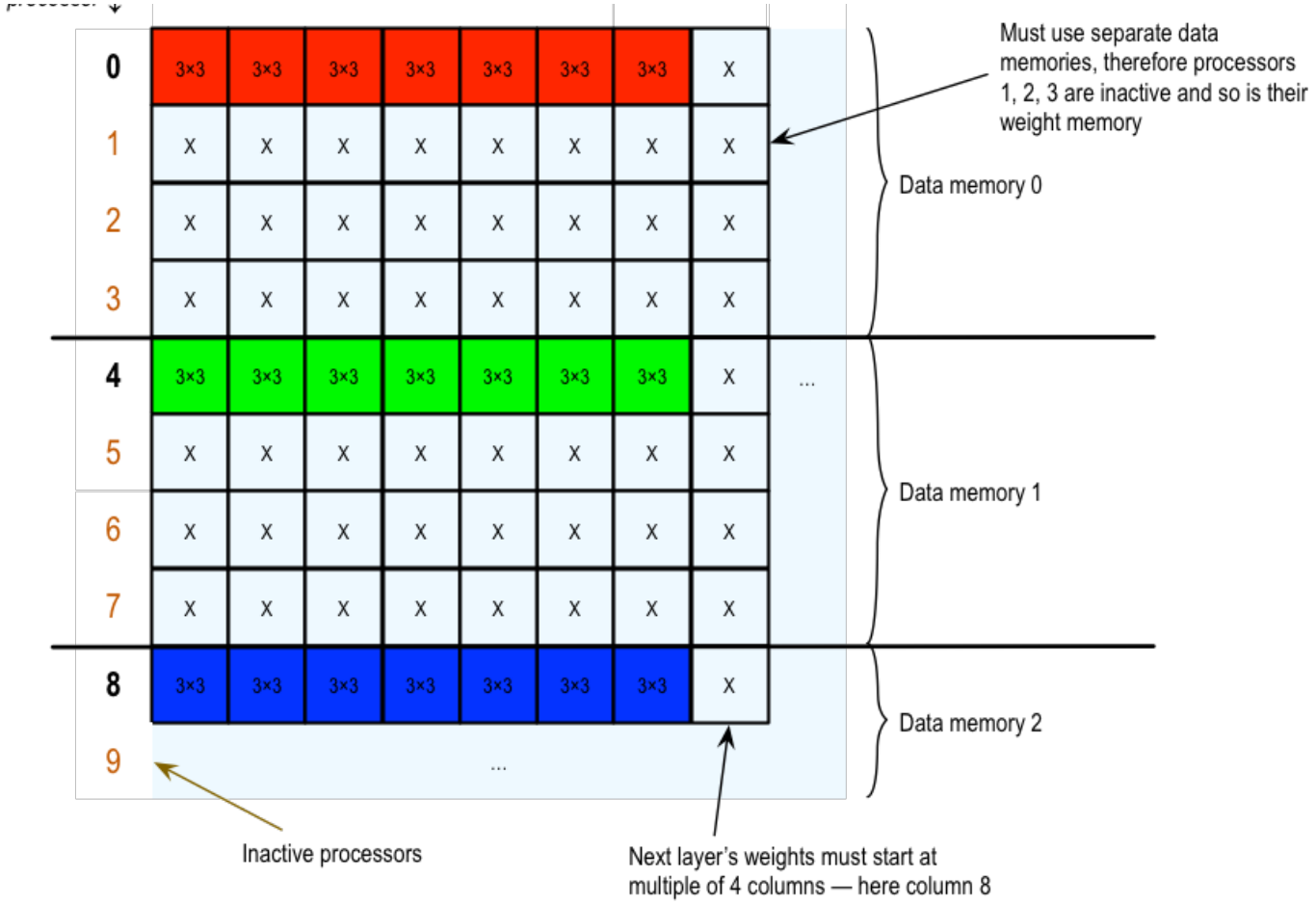
In general, HWC is faster since each memory read can deliver data to up to four processors in parallel. On the other hand, four processors must share one data memory instance, which reduces the maximum allowable dimensions.

CHW Input 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 each 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 input channels for the layer must be equal to, or be a multiple of, the active processors, 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 16384 bytes (or 0x4000) 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.

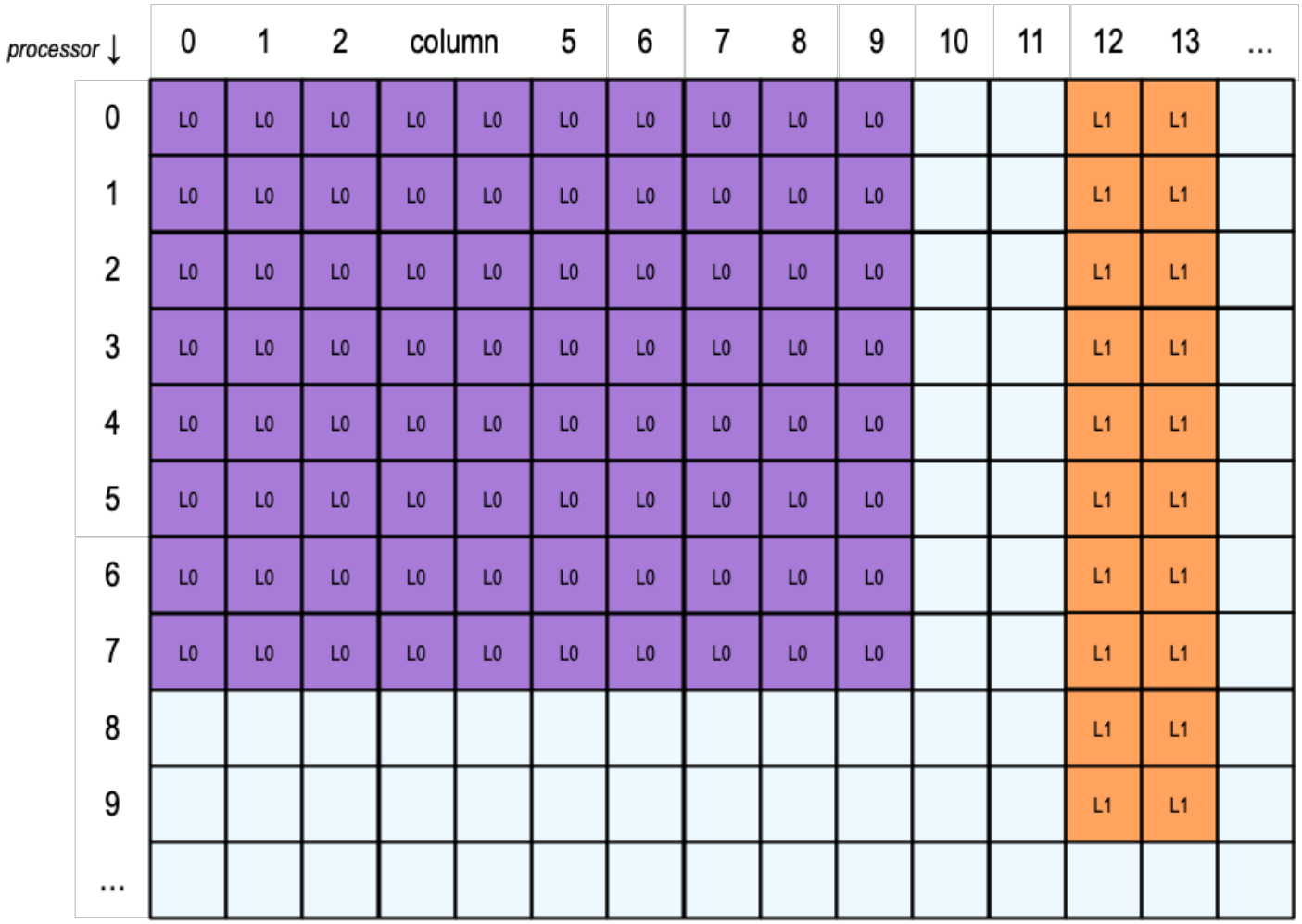
For most simple networks with limited data sizes, it is easiest to ping-pong between the first and second halves of the data memories – specify the data offset as 0 for the first layer, 0x4000 for the second layer, 0 for the third layer, etc. This strategy avoids overlapping inputs and outputs when a 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 8 inputs and 10 outputs, and the second layer (L1) has 10 inputs and 2 outputs.



Bias Memories

Bias values are stored in separate bias memories. There are four bias memory instances available, and a layer can access any bias memory instance where at least one processor is enabled. By default, bias memories are automatically allocated using a modified Fit-First Descending (FFD) algorithm. Before considering the required resource sizes in descending order, and placing values in the bias memory with the most available resources, the algorithm places those bias values that require a single specified bias memory. The bias memory allocation can optionally be controlled using the `bias_group` configuration option.

Weight Storage Example

The file `ai84net.xlsx` contains an example for a single-channel CHW input using the `AI84Net5` network (this example also supports up to four channels in HWC).

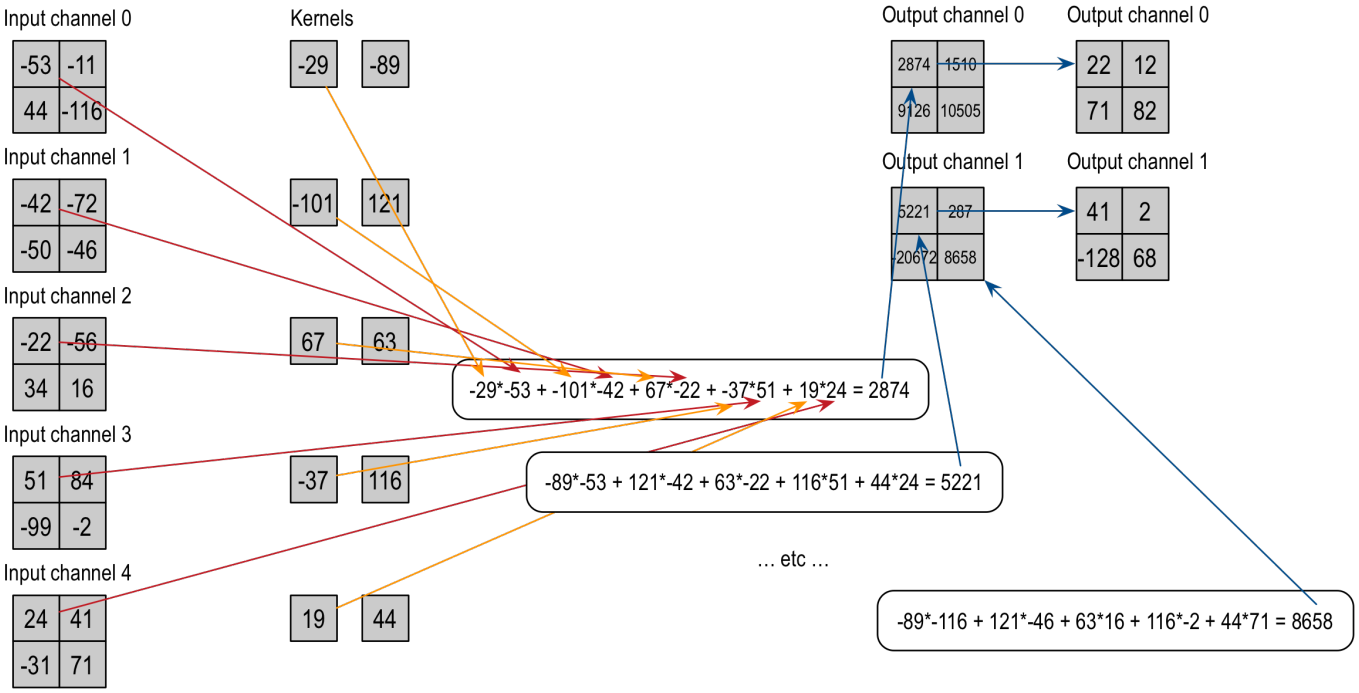
Note: As described above, multiple CHW channels must be loaded into separate memory instances. When using a large number of channels, this can cause "holes" in the processor map, which in turn can cause subsequent layers' kernels to require padding.

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, five input channels, two output channels, and a 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 of 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.

Example: Conv2D with 1x1 kernels)




```
// 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;
```

AI85 memory addresses shown

Limitations of MAX78000 Networks

The MAX78000 hardware does not support arbitrary network parameters. Specifically,

- `Conv2d`:
  - Kernel sizes must be 1×1 or 3×3.
  - Padding can be 0, 1, or 2. Padding always uses zeros.
  - Stride is fixed to [1, 1]

- Stride is fixed to [1, 1].
- `Conv1d`:
  - Kernel sizes must be 1 through 9.
  - Padding can be 0, 1, or 2.
  - Stride is fixed to 1.
- `ConvTranspose2d`:
  - Kernel sizes must be 3×3.
  - Padding can be 0, 1, or 2.
  - Stride is fixed to [2, 2]. Output padding is fixed to 1.
- A programmable layer-specific shift operator is available at the output of a convolution, see `output_shift` (Optional).
- 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 more than 64 channels requires the use of a "fused" convolution in the same layer, unless the pooled dimensions are 1×1.
  - 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 kernel sizes from 1 through 16. *Note: Pooling kernel size values do not have to be the same as the pooling stride.*
  - 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.

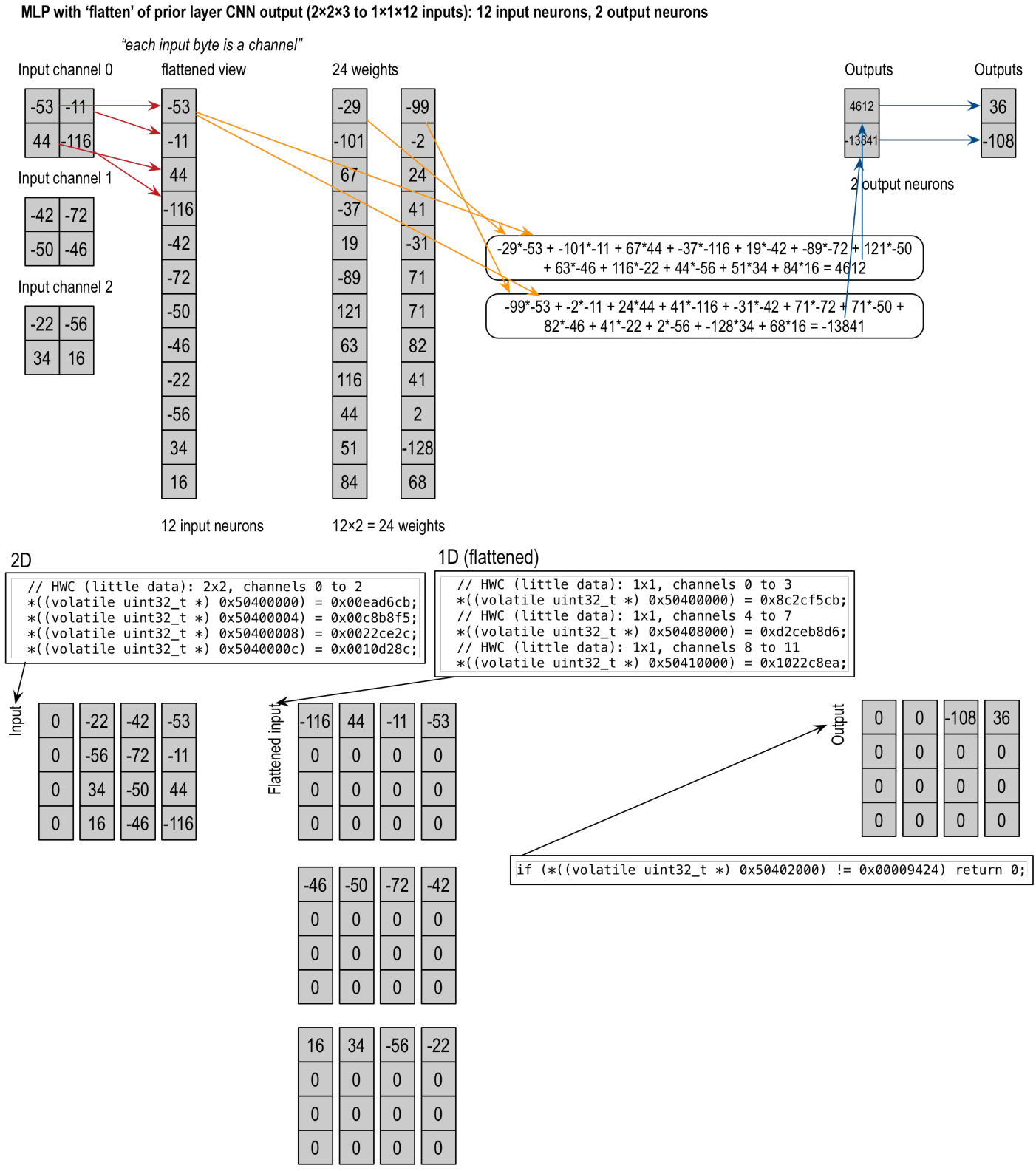
    Example:
    - *floor:* Since there is a quantization step at the output of the average pooling, a 2×2 `AvgPool2d` of `[[0, 0], [0, 3]]` will return $\lfloor \frac{3}{4} \rfloor = 0$.
    - *rounding:* 2×2 `AvgPool2d` of `[[0, 0], [0, 3]]` will return $\lfloor \frac{3}{4} \rceil = 1$.
- The number of input channels must not exceed 1024 per layer.
- The number of output channels must not exceed 1024 per layer.
  - Bias is supported for up to 512 output channels per layer.
- 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.
- Streaming mode:
  - 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: $width * height * \lceil \frac{channels}{64} \rceil < 2^{21}$; *width* and *height* must not exceed 1023.

  - and *height* must not exceed 1023.
  - Streaming is limited to 8 consecutive layers or fewer, and is limited to four FIFOs (up to 4 input channels in CHW and up to 16 channels in HWC format), see FIFOs.
  - For streaming layers, bias values may not be added correctly in all cases.
  - The *final* streaming layer must use padding.
- 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 in Layers and Weight Memory.
- 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 outputs 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., a large number of output channels) that needs to fit into data memory, so the input size limit is mostly theoretical.
- 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`), which are 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, see Fully Connected Layers.
  - When "flattening" two-dimensional data, the input dimensions (C×H×W) must satisfy C×H×W ≤ 16384. Pooling cannot be used at the same time as flattening.
  - 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.
- Dilation, groups, and depth-wise convolutions are not supported. *Note: Batch normalization should be folded into the weights, see Batch Normalization.*

Fully Connected (Linear) Layers

m×n fully connected layers can be realized in hardware by "flattening" 2D input data of dimensions C×H×W into m=C×H×W 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.

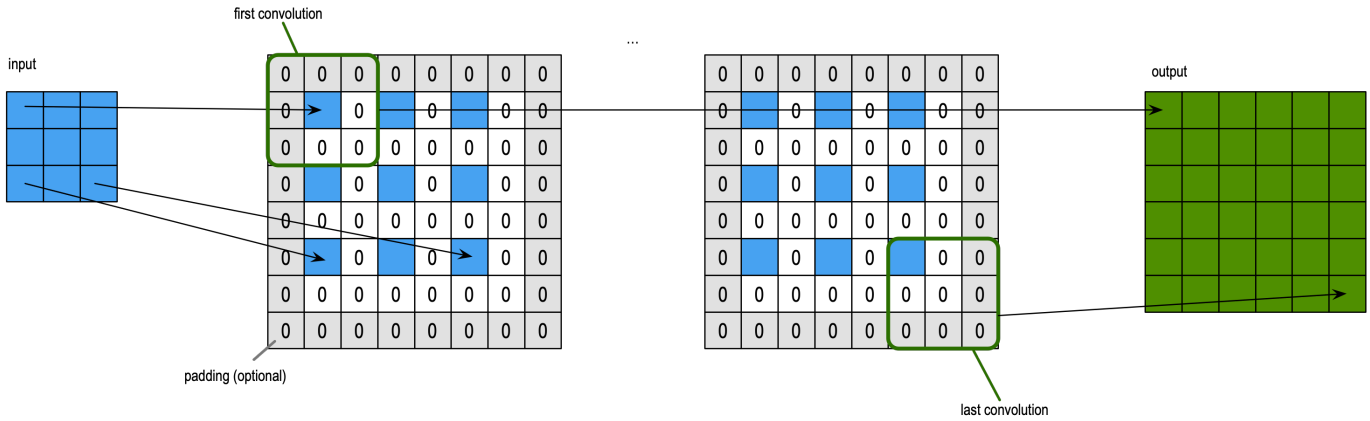
For MAX78000/MAX78002, the product C×H×W must not exceed 16384.



Upsampling (Fractionally-Strided 2D Convolutions)

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, a 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 `scripts/train_*.sh` scripts for example usage).

The `ai84net.py` and `ai85net.py` files contain models that fit into AI84's weight memory. These models rely on the MAX78000/MAX78002 hardware operators that are defined in `ai8x.py`.

To train the FP32 model for MNIST on MAX78000, run `scripts/train_mnist.sh` from 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 MAX78000/MAX78002 specifics.

Since training can take hours or days, the training script does not overwrite any weights previously produced. Results are placed in sub-directories under `logs/` named with the date and time when training began. The latest results are always soft-linked to by `latest-log_dir` and `latest_log_file`.

Command Line Arguments

The following table describes the most important command line arguments for `train.py`. Use `--help` for a complete list.

Argument	Description	Example
`--help`	Complete list of arguments	

Device selection		
`--device`	Set device (default: AI84)	`--device MAX78000`
Model and dataset		
`-a`, `--arch`, `--model`	Set model (collected from models folder)	`--model ai85net5`
`--dataset`	Set dataset (collected from datasets folder)	`--dataset MNIST`
`--data`	Path to dataset (default: data)	`--data /data/ml`
Training		
`--epochs`	Number of epochs to train (default: 90)	`--epochs 100`
`-b`, `--batch-size`	Mini-batch size (default: 256)	`--batch-size 512`
`--compress`	Set compression and learning rate schedule	`--compress schedule.yaml`
`--lr`, `--learning-rate`	Set initial learning rate	`--lr 0.001`
`--deterministic`	Seed random number generators with fixed values	
`--resume-from`	Resume from previous checkpoint	`--resume-from chk.pth.tar`
`--qat-policy`	Define QAT policy in YAML file (default: qat_policy.yaml). Use "None" to disable QAT.	`--qat-policy qat_policy.yaml`
`--nas`	Enable network architecture search	
`--nas-policy`	Define NAS policy in YAML file	`--nas-policy nas/nas_policy.yaml`
Display and statistics		
`--enable-tensorboard`	Enable logging to TensorBoard (default: disabled)	
`--confusion`	Display the confusion matrix	
`--param-hist`	Collect parameter statistics	
`--pr-curves`	Generate precision-recall curves	
`--embedding`	Display embedding (using projector)	

Hardware		
`--use-bias`	The `bias=True` parameter is passed to the model. The effect of this parameter is model-dependent (the parameter does nothing, affects some operations, or all operations).	
`--avg-pool-rounding`	Use rounding for AvgPool	
Evaluation		
`-e`, `--evaluate`	Evaluate previously trained model	
`--8-bit-mode`, `-8`	Simulate quantized operation for hardware device (8-bit data). Used for evaluation only.	
`--exp-load-weights-from`	Load weights from file	
Export		
`--summary onnx`	Export trained model to ONNX (default name: to model.onnx) — *see description below*	
`--summary onnx_simplified`	Export trained model to simplified ONNX file (default name: model.onnx)	
`--summary-filename`	Change the file name for the exported model	`--summary-filename mnist.onnx`
`--save-sample`	Save data[index] from the test set to a NumPy pickle for use as sample data	`--save-sample 10`

ONNX Model Export

The ONNX model export (via `--summary onnx` or `--summary onnx_simplified`) is primarily intended for visualization of the model. ONNX does not support all of the operators that `ai8x.py` uses, and these operators are therefore removed from the export (see function `onnx_export_prep()` in `ai8x.py`). The ONNX file does contain the trained weights and *may* therefore be usable for inference under certain circumstances. However, it is important to note that the ONNX file **will not** be usable for training (for example, the ONNX `floor` operator has a gradient of zero, which is incompatible with quantization-aware training as implemented in `ai8x.py`).

Observing GPU Resources

`nvidia-smi` can be used in a different terminal during training to examine the GPU resource usage of the training process. In the following example, the GPU is using 100% of its compute capabilities, but not all of the available memory. In this particular case, the batch size could be increased to use more memory.

```
$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 430.50       Driver Version: 430.50       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce RTX 208...  Off  | 00000000:01:00.0  On |                  N/A |
| 39%   65C    P2   152W / 250W |   3555MiB / 11016MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
...
```

Custom nn.Modules

The `ai8x.py` file contains customized PyTorch classes (subclasses of `torch.nn.Module`). Any model that is designed to run on MAX78000/MAX78002 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).

List of Predefined Modules

The following modules are predefined:

Name	Description / PyTorch equivalent
Conv2d	Conv2d
FusedConv2dReLU	Conv2d, followed by ReLU
FusedConv2dAbs	Conv2d, followed by Abs
MaxPool2d	MaxPool2d
FusedMaxPoolConv2d	MaxPool2d, followed by Conv2d
FusedMaxPoolConv2dReLU	MaxPool2d, followed by Conv2d, and ReLU
FusedMaxPoolConv2dAbs	MaxPool2d, followed by Conv2d, and Abs
AvgPool2d	AvgPool2d

FusedAvgPoolConv2d	AvgPool2d, followed by Conv2d
FusedAvgPoolConv2dReLU	AvgPool2d, followed by Conv2d, and ReLU
FusedAvgPoolConv2dAbs	AvgPool2d, followed by Conv2d, and Abs
ConvTranspose2d	ConvTranspose2d
FusedConvTranspose2dReLU	ConvTranspose2d, followed by ReLU
FusedConvTranspose2dAbs	ConvTranspose2d, followed by Abs
FusedMaxPoolConvTranspose2d	MaxPool2d, followed by ConvTranspose2d
FusedMaxPoolConvTranspose2dReLU	MaxPool2d, followed by ConvTranspose2d, and ReLU
FusedMaxPoolConvTranspose2dAbs	MaxPool2d, followed by ConvTranspose2d, and Abs
FusedAvgPoolConvTranspose2d	AvgPool2d, followed by ConvTranspose2d
FusedAvgPoolConvTranspose2dReLU	AvgPool2d, followed by ConvTranspose2d, and ReLU
FusedAvgPoolConvTranspose2dAbs	AvgPool2d, followed by ConvTranspose2d, and Abs
Linear	Linear
FusedLinearReLU	Linear, followed by ReLU
FusedLinearAbs	Linear, followed by Abs
Conv1d	Conv1d
FusedConv1dReLU	Conv1d, followed by ReLU
FusedConv1dAbs	Conv1d, followed by Abs
MaxPool1d	MaxPool1d
FusedMaxPoolConv1d	MaxPool1d, followed by Conv1d
FusedMaxPoolConv1dReLU	MaxPool2d, followed by Conv1d, and ReLU
FusedMaxPoolConv1dAbs	MaxPool2d, followed by Conv1d, and Abs
AvgPool1d	AvgPool1d
FusedAvgPoolConv1d	AvgPool1d, followed by Conv1d
FusedAvgPoolConv1dReLU	AvgPool1d, followed by Conv1d, and ReLU
FusedAvgPoolConv1dAbs	AvgPool1d, followed by Conv1d, and Abs
Add	Element-wise Add

Sub	Element-wise Sub
Or	Element-wise bitwise Or
Xor	Element-wise bitwise Xor

Dropout

Dropout modules such as `torch.nn.Dropout()` and `torch.nn.Dropout2d()` are automatically disabled during inference, and can therefore be used for training without affecting inference.

view and reshape

There are two supported cases for `view()` or `reshape()`.

1. Conversion between 1D data and 2D data: Both the batch dimension (first dimension) and the channel dimension (second dimension) must stay the same. The height/width of the 2D data must match the length of the 1D data (i.e., H×W = L).
   Examples:
   ```
   x = x.view(x.size(0), x.size(1), -1)  # 2D to 1D
   x = x.view(x.shape[0], x.shape[1], 16, -1)  # 1D to 2D
   ```
   Note: `x.size()` and `x.shape[]` are equivalent.
   When reshaping data, `in_dim:` must be specified in the model description file.
2. Conversion from 1D and 2D to Fully Connected ("flattening"): The batch dimension (first dimension) must stay the same, and the other dimensions are combined (i.e., M = C×H×W or M = C×L).
   Example:
   ```
   x = x.view(x.size(0), -1)  # Flatten
   ```

Support for Quantization

The hardware always uses signed integers for data and weights. While data is always 8-bit, weights can be configured on a per-layer basis. However, training makes use of floating point values for both data and weights, while also clipping (clamping) values.

Data

When using the `-8` command line switch, all module outputs are quantized to 8-bit in the range [-128…+127] to simulate hardware behavior. The `-8` command line switch is designed for *evaluating quantized weights* against a test set, in order to understand the impact of quantization. *Note that model training always uses floating point values, and therefore `-8` is not compatible with training.*

The last layer can optionally use 32-bit output for increased precision. This is simulated by adding the parameter `wide=True` to the module function call.

Weights: Quantization-Aware Training (QAT)

Quantization-aware training (QAT) is enabled by default. QAT is controlled by a policy file, specified by `qat_

Quantization-aware training (QAT) is enabled by default. QAT is controlled by a policy file, specified by `--qat-policy`.

- After `start_epoch` epochs, training will learn an additional parameter that corresponds to a shift of the final sum of products.
- `weight_bits` describes the number of bits available for weights.
- `overrides` allows specifying the `weight_bits` on a per-layer basis.

By default, weights are quantized to 8-bits after 10 epochs as specified in `qat_policy.yaml`. A more refined example that specifies weight sizes for individual layers can be seen in `qat_policy_cifar100.yaml`.

Quantization-aware training can be disabled by specifying `--qat-policy None`.

For more information, please also see Quantization.

Batch Normalization

Batch normalization after `Conv1d` and `Conv2d` layers is supported using "fusing." The fusing operation merges the effect of batch normalization layers into the parameters of the preceding convolutional layer, by modifying weights and bias values of that preceding layer. For detailed information about batch normalization fusing/fusion/folding, see Section 3.2 of the following paper: https://arxiv.org/pdf/1712.05877.pdf.

After fusing/folding, the network will no longer contain any batchnorm layers. The effects of batch normalization will instead be expressed by modified weights and biases of the preceding convolutional layer.

- When using Quantization-Aware Training (QAT), batchnorm layers are automatically folded during training and no further action is needed.
- When using Post-Training Quantization, the `batchnormfuser.py` script (see BatchNorm Fusing) must be called before `quantize.py` to explicitly fuse the batchnorm layers.

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`. When enabled using `--enable-tensorboard`, it provides 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.

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.

To start the TensorBoard server, use a second terminal window:

```
(ai8x-training) $ tensorboard --logdir='./logs'
TensorBoard 2.2.2 at http://127.0.0.1:6006/ (Press CTRL+C to quit)
```

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

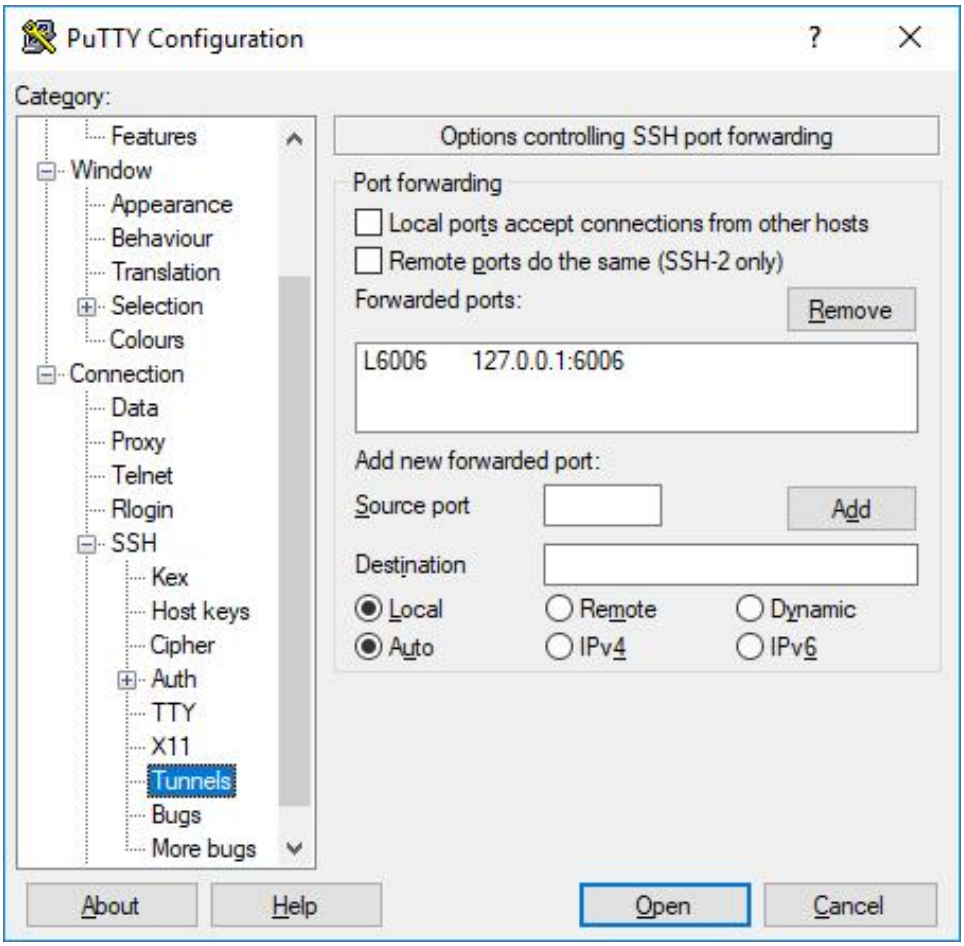
The training progress can be observed by starting TensorBoard and pointing a web browser to the port indicated.

Remote Access to TensorBoard

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
```

When using PuTTY, port forwarding is achieved as follows:



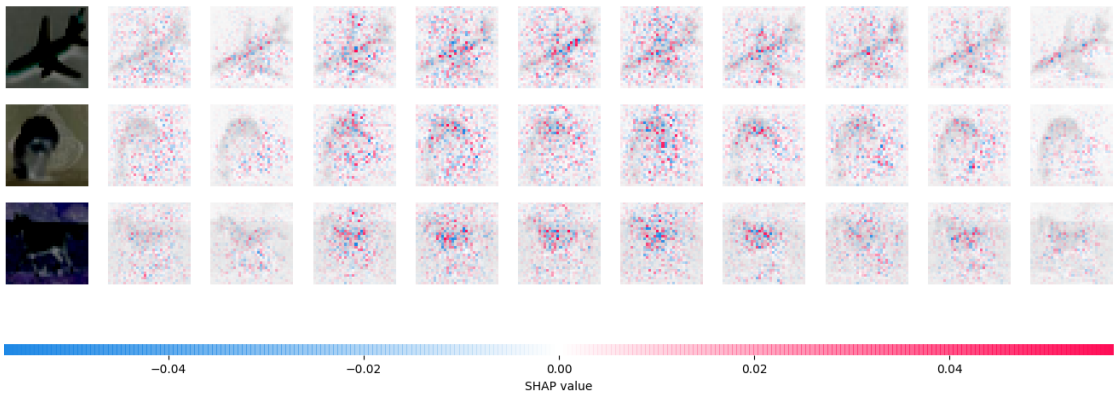
SHAP — SHapely Additive exPlanations

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 ai85net5 --dataset CIFAR10 --confusion --evaluate --device MAX78000
--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.



BatchNorm Fusing

Batchnorm fusing (see Batch Normalization) is needed as a separate step only when both the following are true:

1. Batch Normalization is used in the network and
2. Quantization-Aware Training (QAT) is not used (i.e., when post-training quantization is active).

In order to perform batchnorm fusing, the `batchnormfuser.py` tool must be run *before* the `quantize.py` script.

Note: Most of the examples either don't use batchnorm, so no fusing is needed, or they use QAT, so batchnorm fusing happens automatically.

Command Line Arguments

The following table describes the command line arguments for `batchnormfuser.py`:

The following table describes the command line arguments for `batchnormfuser.py`:

Argument	Description	Example
`-i`, `--inp_path`	Set input checkpoint path	`-i logs/2020.06.05-235316/best.pth.tar`
`-o`, `--out_path`	Set output checkpoint path for saving fused model	`-o best_without_bn.pth.tar`
`-oa`, `--out_arch`	Set output architecture name (architecture without batchnorm layers)	`-oa ai85simplenet`

Quantization

There are two main approaches to quantization — quantization-aware training and post-training quantization. The MAX78000/MAX78002 support both approaches.

For both approaches, the `quantize.py` 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 quantized checkpoint file will also be used to feed the Network Loader.

Quantization-Aware Training (QAT)

Quantization-aware training is the better performing approach. It is enabled by default. QAT learns additional parameters during training that help with quantization (see Weights: Quantization-Aware Training (QAT). No additional arguments are needed for `quantize.py`.

Post-Training Quantization

This approach is also called *"naive quantization"*. It should be used when `--qat-policy None` is specified for training.

While several approaches for clipping are implemented in `quantize.py`, clipping with a simple fixed scale factor performs best, based on experimental results. The approach requires the clamping operators implemented in `ai8x.py`.

Note that the "optimum" scale factor for simple clipping is highly dependent on the model and weight data. For the MNIST example, a `--scale 0.85` works well. For the CIFAR-100 example on the other hand, Top-1 performance is 30 points better with `--scale 1.0`.

Command Line Arguments

The `quantize.py` software has the following important command line arguments:

The `quantize.py` software has the following important command line arguments:

Argument	Description	Example
`--help`	Complete list of options	
Device selection		
`--device`	Set device (default: MAX78000)	`--device MAX78000`
Debug		
`-v`	Verbose output	
Weight quantization		
`-c`, `--config-file`	YAML file with weight quantization information (default: from checkpoint file, or 8-bit for all layers)	`-c mnist.yaml`
`--clip-method`	Non-QAT clipping method — either STDDEV, AVG, AVGMAX or SCALE	`--clip-method SCALE`
`--scale`	Sets scale for the SCALE clipping method	`--scale 0.85`

Note: The syntax for the optional YAML file is described below. The same file can be used for both `quantize.py` and `ai8xize.py`.

Example and Evaluation

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

Example for MNIST:

```
(ai8x-synthesis) $ scripts/quantize_mnist.sh
```

To evaluate the quantized network for MAX78000 (run from the training project):

```
(ai8x-training) $ scripts/evaluate_mnist.sh
```

Alternative Quantization Approaches

If quantization-aware training is not desired, post-training quantization can be improved using more

If quantization-aware training is not desired, 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).

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:

- Implement a new network model based on the constraints described earlier, see Custom nn.Modules (and `models/ai85net.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:

Data Loader

Develop a data loader in PyTorch, see https://pytorch.org/tutorials/beginner/data_loading_tutorial.html. See `datasets/mnist.py` for an example.

The data loader must include a loader function, for example `mnist_get_datasets(data, load_train=True, load_test=True)`. `data` is a tuple of the specified data directory and the program arguments, and the two *bools* specify whether training and/or test data should be loaded.

The data loader is expected to download and preprocess the datasets as needed and install everything in the specified location.

The loader returns a tuple of two PyTorch Datasets for training and test data.

Normalizing Input Data

For training, input data is expected to be in the range $[-\frac{128}{128}, +\frac{127}{128}]$. When evaluating quantized weights, or when running on hardware, input data is instead expected to be in the native MAX7800X range of $[-128, +127]$. Conversely, the majority of PyTorch datasets are PIL images of range $[0, 1]$. The respective data loaders therefore call the `ai8x.normalize()` function, which expects an input of 0 to 1 and normalizes the data to either of these output ranges.

When running inference on MAX7800X hardware, it is important to take the native data format into account, and it is desirable to perform as little preprocessing as possible during inference. For example, an image sensor

may return "signed" data in the range $[-128, +127]$ for each color. No additional preprocessing or mapping is needed for this sensor since the model was trained with this data range.

In many cases, image data is delivered as fewer than 8 bits per channel (for example, RGB565). In these cases, retraining the model with this limited range (0 to 31 for 5-bit color and 0 to 63 for 6-bit color, respectively) can potentially eliminate the need for inference-time preprocessing.

On the other hand, a different sensor may produce unsigned data values in the full 8-bit range $[0, 255]$. This range must be mapped to $[-128, +127]$ to match hardware and the trained model. The mapping can be performed during inference by subtracting 128 from each input byte, but this requires extra processing time during inference.

`datasets` Data Structure

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 `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)`.

The optional `regression` key in the structure can be set to `True` to automatically select the `--regression` command line argument. `regression` defaults to `False`.

The optional `visualize` key can point to a custom visualization function used when creating `--embedding`. The input to the function (format NCHW for 2D data, or NCL for 1D data) is a batch of data (with N ≤ 100). The default handles square RGB or monochrome images. For any other data, a custom function must be supplied.

Training and Verification Data

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.

Training Process

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

Netron — Network Visualization

The Netron tool can visualize networks, similar to what is available within Tensorboard. To use Netron, use `train.py` to export the trained network to ONNX, and upload the ONNX file.

```
(ai8x-training) $ ./train.py --model ai85net5 --dataset MNIST --evaluate --exp-load-weights-from checkpoint.pth.tar --device MAX78000 --summary onnx
```

Neural Architecture Search (NAS)

Neural Architecture Search (NAS)

Introduction

The following chapter describes the neural architecture search (NAS) solution for MAX78000 as implemented in the ai8x-training repository. Details are provided about the NAS method, how to run existing NAS models in the repository, and how to define a new NAS model.

The intention of NAS is to find the best neural architecture for a given set of requirements by automating architecture engineering. NAS explores the search space automatically and returns an architecture that is hard to optimize further using human or "manual" design. Multiple different techniques are proposed in the literature for automated architecture search, including reinforcement-based and evolutionary-based solutions.

Once-for-All

Once-for-All (OFA) is a weight-sharing-based NAS technique, originally proposed by MIT and IBM researchers. The paper introduces a method to deploy a trained model to diverse hardware directly without the need of retraining. This is achieved by training a "supernet," which is named the "Once-for-All" network, and then deploying only part of the supernet, depending on hardware constraints. This requires a training process where all sub-networks are trained sufficiently to be deployed directly. Since training all sub-networks can be computationally prohibitive, sub-networks are sampled during each gradient update step. However, sampling only a small number of networks may cause performance degradation as the sub-networks are interfering with one another. To resolve this issue, a *progressive shrinking* algorithm is proposed by the authors. Rather than optimizing the supernet directly with all interfering sub-networks, they propose to first train a supernet that is the largest network with maximum kernel size, depth and width. Then, smaller sub-networks that share parameters with the supernet are trained progressively. Thus, smaller networks can be initialized with the most important parameters. If the search space consists of different kernel sizes, depths and widths, they are added to sampling space sequentially to minimize the risk of parameter interference. To illustrate, after full model training, the "elastic kernel" stage is performed, where the kernel size is chosen from {1×1, 3×3} while the depth and width are kept at their maximum values. Next, kernel sizes and depths are sampled in the "elastic depth" stage. Finally, all sub-networks are sampled from the whole search space in the "elastic width" stage.

After the supernet is trained using sub-networks, the "architecture search" stage takes place. The paper proposes evolutionary search as the search algorithm. In this stage, the best architecture is searched, given particular hardware constraints. A set of candidate architectures that perform best on the validation set are mutated, and crossovers are performed iteratively in the algorithm.

After the training and search steps, the model is ready to deploy to the target hardware in the OFA method as the parameters are already trained. However, on MAX78000, the model still needs to be quantized for deployment. Therefore, this implementation has an additional step where the model needs to be trained using the quantization-aware training (QAT) module of the MAX78000 training repository.

To summarize, the sequential steps of the Once-for-All supernet training are:

1. Full model training (stage 0): In this step, the supernet with maximum kernel size, depth and width is trained. This network is suggested to be **at least 3× to 5×** larger than the MAX78000 implementation limits, since sub-networks of the supernet are the targets for MAX78000 deployment.
2. Elastic kernel (stage 1): In this step, only sub-networks with different kernel sizes are sampled from the

2. Elastic kernel (stage 1): In this step, only sub-networks with different kernel sizes are sampled from the supernet. For the MAX78000 *Conv2d* layers, the supported sizes are {3×3, 1×1}, and {5, 3, 1} for *Conv1d* layers. Since the sampled sub-network is a part of the supernet, the supernet is updated with gradient updates.
3. Elastic depth (stage 2): In this step, sub-networks with different kernel sizes and depths are sampled from the supernet. In the MAX78000 implementation of OFA, the network is divided into parts called "units." Each unit can consist of a different number of layers and contain an extra pooling layer at its beginning. Depth sampling is performed inside the units. If a sub-network with *N* layers in a specific unit is sampled, the first *D* layers of the unit in the supernet is kept by removing the last *(N-D)* layers. Consequently, the first layers of each unit are shared among multiple sub-networks.
4. Elastic width (stage 3): In addition to kernel size and depth, sub-networks are sampled from different width options in this stage. For width shrinking, the most important channels with the largest L1 norm are selected. This ensures that only the most important channels are shared. To achieve this, the layer output channels are sorted after each gradient update. The input channels of the following layers are sorted similarly to keep the supernet functional.
5. Evolutionary search: For most search space selections, the number of sub-networks is too large to allow for evaluation of each sub-network. During evolutionary search, better architectures are found after each iteration by mutations and crossovers. The processing time required for this stage depends on the candidate pool size and the number of iterations; however, it is generally much shorter than the time spent for the training stages.

In addition to the steps listed above, QAT is performed using the chosen architecture.

For more details and to better understand OFA, please see the original paper.

Stages and Levels in the MAX78000 Implementation

In the NAS module of the *ai8x-training* repository, there are two concepts that are used to indicate the progress of the NAS training process, called "stages" and "levels." *Stage* denotes whether full model training (stage 0), elastic kernel (stage 1), elastic depth (stage 2) or elastic width (stage 3) is being performed. Training is completed after stage 3 has finished.

Levels denote the phases of stages. In the original OFA paper, the authors suggest progressive shrinking to facilitate training of interfering sub-networks. Stages play an important role here. In each stage, a new search parameter is introduced to the training. To further facilitate training, stages can be decomposed into levels. With increasing levels, smaller sub-networks become sampleable since the network is trained well enough to be ready for an increased number of sub-networks. For example, if the deepest unit in the network consists of 4 layers, there are 3 levels in stage 2. The reason for this is that in the level 1 of stage 2 (elastic depth), the last layer is removed with 50% probability in the sub-network sampling. Therefore, possible depths are 3 or 4 for that unit in level 1. In level 2, the possible depths for this unit are [2, 3, 4]. Likewise, the possible depths are [1, 2, 3, 4] in level 3. The first layer in a unit is always present, it is never removed in any sub-network. The same level logic applies to stage 1 and stage 3 as well. In stage 1, kernel sizes are sampled. For 2D convolutions, the possible kernel options are either 1×1 or 3×3, so there is only one level. However, for 1D convolutions, kernel sizes could be 5, 3, or 1; therefore, there are two levels. In stage 3, widths are sampled. The possible widths are

100% of the same layer's width in the supernet, plus 75%, 50%, and 25% of the supernet width. Since there are four options, there are 4–1=3 levels in stage 3. As levels increase, smaller widths become an option in the sampling pool.

sampling pool.

In summary, the architecture of the supernet determines how many levels there will be for training. The deepest unit determines the number of levels in stage 2. Assuming there are three levels in stage 2, then training continues from level 1 of stage 3 just after level 3 of stage 2 has completed. The checkpoint files for each level are saved, so it is possible to resume training from a specific level.

Usage

Network Architecture Search (NAS) can be enabled using the `--nas` command line option. NAS is based on the Once-For-All (OFA) approach described above. NAS is controlled by a policy file, specified by `--nas-policy`. The policy file contains the following fields:

- `start_epoch`: The full model is trained without any elastic search until this epoch is reached.
- `validation_freq` is set to define the frequency in epochs to calculate the model performance on the validation set after full model training. This parameter is used to save training time, especially when the model includes batch normalization.
- The `elastic_kernel`, `elastic_depth` and `elastic_width` fields are used to define the properties of each elastic search stage. These fields include the following two sub-fields:
  - `leveling` enables leveling during elastic search. *See above for an explanation of stages and levels.*
  - `num_epochs` defines the number of epochs for each level of the search stage if `leveling` is `False`.
- `kd_params` is set to enable Knowledge Distillation.
  - `teacher_model` defines the model used as teacher model. Teacher is the model before epoch `start_epoch` if it is set to `full_model`. Teacher is updated with the model just before the stage transition if this field is set to `prev_stage_model`.
  - See here for more information to set `distill_loss`, `student_loss` and `temperature`.
- The `evolution_search` field defines the search algorithm parameters used to find the sub-network of the full network.
  - `population_size` is the number of sub-networks to be considered at each iteration.
  - `ratio_parent` is the ratio of the population to be kept for the next iteration.
  - `ratio_mutation` determines the number of mutations at each iteration, which is calculated by multiplying this ratio by the population size.
  - `prob_mutation` is the ratio of the parameter change of a mutated network.
  - `num_iter` is the number of iterations.
  - `constraints` are used to define the constraints of the samples in the population.
    - `min_num_weights` and `max_num_weights` are used to define the minimum and the maximum number of weights in the network.
    - `width_options` is used to limit the possible number of channels in any of the layers in the selected network. This constraint can be used to effectively use memory on MAX78000.

It is also possible to resume NAS training from a saved checkpoint using the `--resume-from` option. The teacher model can also be loaded using the `--nas-kd-resume-from` option.

Important Considerations for NAS

- Since the sub-networks are intended to be used on MAX78000, ensure that the full model size of OFA is **at least three times** larger than the MAX78000 kernel memory size. Likewise, it is good practice to design it deeper and wider than the final network that may be considered suitable for the given task. If the initial model size is too big, it will slow down the training process, and there is a risk that most of the sub-networks exceed the MAX78000 resources. Therefore, 3× to 5× is recommended as the size multiplier for the full model selection.
- For the width selection, ensure that widths are multiples of 64 as MAX78000 has 64 processors, and each channel is processed in a separate processor. Using multiples of 64, kernel memory is used more efficiently as widths are searched within [100%, 75%, 50%, 25%] of the initial supernet width selection. Note that these are the default percentages, and they can be changed. Rather than sudden decreases, more granularity and a linear decrease are recommended as this is more suitable for progressive shrinking.
- **NAS training takes time**. It will take days or even weeks depending on the number of sub-networks, the full model size and number of epochs at each stage/level, and the available GPU hardware. It is recommended to watch the loss curves during training and to stop training when the loss fully converges. Then, proceed with the next level using the checkpoint saved from the last level.
- The number of batches in each epoch plays an important role in the selection of the number of epochs for each stage/level. If the dataset is ten times bigger and there are ten times more gradient updates, divide the number of epochs by 10 for the same supernet architecture.

NAS Model Definition

The only model architecture implemented in this repository is the sequential model. It is composed of sequential units, which has several sequential *FusedConvNdBNReLU* with an optional *MaxPool* layer at the end, and a *Linear* layer last (see Figure).

All required elastic search strategies are implemented in this model file.

A new model architecture can be implemented by implementing the `OnceForAllModel` interface. The new model class must implement the following:

- `sample_subnet_width`
- `reset_width_sampling`
- `get_max_elastic_width_level`
- `sample_subnet_depth`
- `reset_depth_sampling`
- `get_max_elastic_depth_level`
- `sample_subnet_kernel`
- `reset_kernel_sampling`
- `get_max_elastic_kernel_level`

NAS Output

The NAS trains floating point models and logs the best model architectures with the highest accuracies. When

NAS has completed, a new model file must be created using the new architecture, either by copying the required parameters to post-training quantization, or by initiating quantization-aware training (QAT).

Network Loader (AI8Xize)

The `ai8xize` network loader currently depends on PyTorch and Nervana's Distiller. This requirement will be removed in the future.

The network loader creates C code that programs the MAX78000/MAX78002 (for embedded execution, or RTL simulation). 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 `ai8xize.py` program needs three inputs:

1. A quantized checkpoint file, generated by the MAX78000/MAX78002 model quantization program `quantize.py`, see Quantization.
2. A YAML description of the network, see YAML Network Description.
3. A NumPy "pickle" `.npy` file with sample input data, see Generating a Random Sample Input.

By default, the C code is split into two files: `main.c` contains the wrapper code, and loads a sample input and verifies the output for the sample input. `cnn.c` contains functions that are generated for a specific network to load, configure, and run the accelerator. During development, this split makes it easier to swap out only the network while keeping customized wrapper code intact.

Command Line Arguments

The following table describes the most important command line arguments for `ai8xize.py`. Use `--help` for a complete list.

Argument	Description	Example
`--help`	Complete list of arguments	
Device selection		
`--device`	Set device (default: AI84)	`--device MAX78000`
Hardware features		
`--avg-pool-rounding`	Round average pooling results	

`--simple1b`	Use simple XOR instead of 1-bit multiplication	
Embedded code		
`--config-file`	YAML configuration file containing layer configuration	`--config-file cfg.yaml`
`--checkpoint-file`	Checkpoint file containing quantized weights	`--checkpoint-file chk.pth.tar`
`--display-checkpoint`	Show parsed checkpoint data	
`--prefix`	Set test name prefix	`--prefix mnist`
`--board-name`	Set the target board (default: `EvKit_V1`)	`--board-name FTHR_RevA`
Code generation		
`--overwrite`	Produce output even when the target directory exists (default: abort)	
`--compact-data`	Use *memcpy* to load input data in order to save code space	
`--compact-weights`	Use *memcpy* to load weights in order to save code space	
`--mexpress`	Use faster kernel loading	
`--mlator`	Use hardware to swap output bytes (useful for large multi-channel outputs)	
`--softmax`	Add software Softmax functions to generated code	
`--boost`	Turn on a port pin to boost the CNN supply	`--boost 2.5`
`--timer`	Insert code to time the inference using a timer	`--timer 0`
`--no-wfi`	Do not use WFI instructions when waiting for CNN completion	
File names		

`--c-filename`	Main C file name base (default: main.c)	`--c-filename main.c`
`--api-filename`	API C file name (default: cnn.c)	`--api-filename cnn.c`
`--weight-filename`	Weight header file name (default: weights.h)	`--weight-filename wt.h`
`--sample-filename`	Sample data header file name (default: sampledata.h)	`--sample-filename kat.h`
`--sample-output-filename`	Sample result header file name (default: sampleoutput.h)	`--sample-output-filename katresult.h`
`--sample-input`	Sample data source file name (default: tests/sample_dataset.npy)	`--sample-input kat.npy`
Streaming and FIFOs		
`--fifo`	Use FIFOs to load streaming data	
`--fast-fifo`	Use fast FIFO to load streaming data	
`--fast-fifo-quad`	Use fast FIFO in quad fanout mode (implies --fast-fifo)	
RISC-V		
`--riscv`	Use RISC-V processor	
`--riscv-debug`	Use RISC-V processor and enable the RISC-V JTAG	
`--riscv-exclusive`	Use exclusive SRAM access for RISC-V (implies --riscv)	
Debug and		

logging		
`-v`, `--verbose`	Verbose output	
`--no-log`	Do not redirect stdout to log file (default: enabled)	
`--log-intermediate`	Log data between layers	
`--log-pooling`	Log unpooled and pooled data between layers in CSV format	
`--log-filename`	Log file name (default: log.txt)	`--log-filename run.log`
`-D`, `--debug`	Debug mode	
`--debug-computation`	Debug computation (SLOW)	
`--stop-after`	Stop after layer	`--stop-after 2`
`--one-shot`	Use layer-by-layer one-shot mechanism	
`--ignore-bias-groups`	Do not force `bias_group` to only available x16 groups	
Streaming tweaks		
`--overlap-data`	Allow output to overwrite input	
`--override-start`	Override auto-computed streaming start value (x8 hex)	
`--increase-start`	Add integer to streaming start value (default: 2)	
`--override-rollover`	Override auto-computed streaming rollover value (x8 hex)	
`--override-delta1`	Override auto-computed streaming delta1 value (x8 hex)	

`--increase-delta1`	Add integer to streaming delta1 value (default: 0)	
`--override-delta2`	Override auto-computed streaming delta2 value (x8 hex)	
`--increase-delta2`	Add integer to streaming delta2 value (default: 0)	
`--ignore-streaming`	Ignore all 'streaming' layer directives	
Power saving		
`--powerdown`	Power down unused MRAM instances	
`--deepsleep`	Put Arm core into deep sleep	
Hardware settings		
`--input-offset`	First layer input offset (x8 hex, defaults to 0x0000)	`--input-offset 2000`
`--mlator-noverify`	Do not check both mlator and non-mlator output	
`--write-zero-registers`	Write registers even if the value is zero	
`--init-tram`	Initialize TRAM to 0	
`--zero-sram`	Zero memories	
`--zero-unused`	Zero unused registers	
`--ready-sel`	Specify memory waitstates	
`--ready-sel-fifo`	Specify FIFO waitstates	
`--ready-sel-aon`	Specify AON waitstates	
Various		
`--`	Instead of using large sample input data, use only the first `--`	`--`

`--synthesize-input`	Instead of using large sample input data, use only the first `--synthesize-words` words of the sample input, and add N to each subsequent set of `--synthesize-words` 32-bit words	`--synthesize-input 0x112233`
`--synthesize-words`	When using `–synthesize-input`, specifies how many words to use from the input. The default is 8. This number must be a divisor of the total number of pixels per channel.	`--synthesize-words 64`
`--max-verify-length`	Instead of checking all of the expected output data, verify only the first N words	`--max-verify-length 1024`
`--no-unload`	Do not create the `cnn_unload()` function	
`--no-kat`	Do not generate the `check_output()` function (disable known-answer test)	

YAML Network Description

The quick-start guide provides a short overview of the purpose and structure of the YAML network description file.

The following is a detailed guide into all supported configuration options.

An example network description for the ai85net5 architecture and MNIST is shown below:

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

arch: ai85net5
dataset: MNIST

# Define layer parameters in order of the layer sequence
layers:
- pad: 1
  activate: ReLU
  out_offset: 0x2000
  processors: 0x0000000000000001
  data_format: CHW
  op: conv2d
- max_pool: 2
  pool_stride: 2
  pad: 2
  activate: ReLU

  out_offset: 0
  processors: 0xffffffffffffffff
  op: conv2d
```

```
- max_pool: 2
  pool_stride: 2
  pad: 1
  activate: ReLU
  out_offset: 0x2000
  processors: 0xfffffffffffffff0
  op: conv2d
- avg_pool: 2
  pool_stride: 2
  pad: 1
  activate: ReLU
  out_offset: 0
  processors: 0x0fffffffffffffff0
  op: conv2d
- op: mlp
  flatten: true
  out_offset: 0x1000
  output_width: 32
  processors: 0x0000000000000fff
```

To generate an embedded MAX78000 demo in the `demos/ai85-mnist/` folder, use the following command line:

```
(ai8x-synthesize) $ ./ai8xize.py --verbose --test-dir demos --prefix ai85-mnist --checkpoint-file trained/ai85-mnist.pth.tar --config-file networks/mnist-chw-ai85.yaml --device MAX78000 --compact-data --mexpress --softmax
```

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/ai85-mnist-....` (where .... is an autogenerated string based on the network topology), use:

```
(ai8x-synthesize) $ ./ai8xize.py --rtl --verbose --autogen rtlsim --test-dir rtlsim --prefix ai85-mnist --checkpoint-file trained/ai85-mnist.pth.tar --config-file networks/mnist-chw-ai85.yaml --device MAX78000
```

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.

Note: The network loader automatically checks the configuration file for syntax errors and prints warnings for non-fatal errors. To perform the same checks manually, run: `yamllint configfile.yaml`

Purpose of the YAML Network Description

The network description must match the model that was used for training. In effect, the checkpoint file contains the trained weights, and the YAML file contains a description of the network structure. Additionally, the YAML file (sometimes also called "sidecar file") describes which processors to use (`processors`) and which offsets to read and write data from (`in_offset` and `out_offset`).

Data Memory Ping-Pong

For simple networks with relatively small data dimensions, the easiest way to use the data offsets is by "ping-ponging" between offset 0 and half the memory (offset 0x4000). For example, the input is loaded at offset 0, and the first layer produces its output at offset 0x4000. The second layer reads from 0x4000 and writes to 0. Assuming the dimensions are small enough, this easy method avoids overwriting an input that has not yet been consumed by the accelerator.

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. Please also see `output_processors`.

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.

`processors` is specified as a 64-bit hexadecimal value. Dots ('.') and a leading '0x' are ignored.

Note: When using multi-pass (i.e., using more than 64 channels), the number of processors is an integer division of the channel count, rounded up to the next multiple of 4. For example, 52 processors are required for 100 channels (since 100 div 2 = 50, and 52 is the next multiple of 4). For best efficiency, use channel counts that are multiples of 4.

Example for three processors 0, 4, and 8:

```
processors: 0x0000.0000.0000.0111
```

Example for four processors 0, 1, 2, and 3:

```
processors: 0x0000.0000.0000.000f
```

`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`. `output_processors` is specified as a 64-bit hexadecimal value. Dots ('.') and a leading '0x' are ignored.

`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`. See also Data Memory Ping-Pong.

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)

When **not** using an `activation`, the **last** layer can output `32` bits of unclipped data in Q17.14 format. The default is `8` bits. *Note that the corresponding model's last layer must be trained with* `wide=True` *when* `output_width` *is 32*.

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`, see Channel Data Formats.

`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.

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 *(note: input and output processors must be the same)*
`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.
Note: On MAX78000, this is only supported for 64 channels, or up to 128 channels when only two operands are used. Use a separate layer for the element-wise operation when more operands or channels are needed instead of combining the element-wise operator with a convolution.

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`.

default is `2`.

Example:

`operation: add`

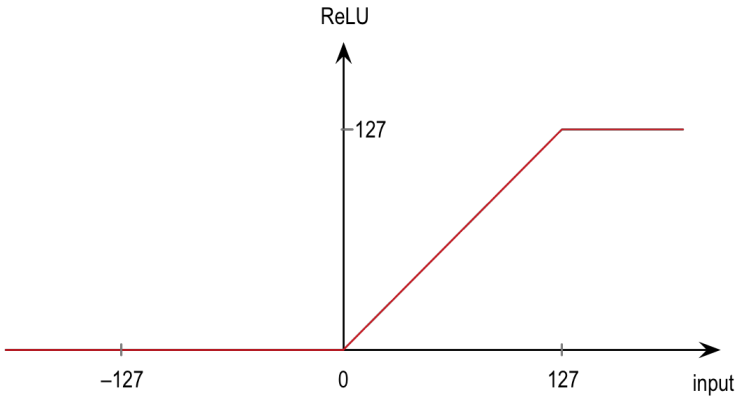
`operands: 4`

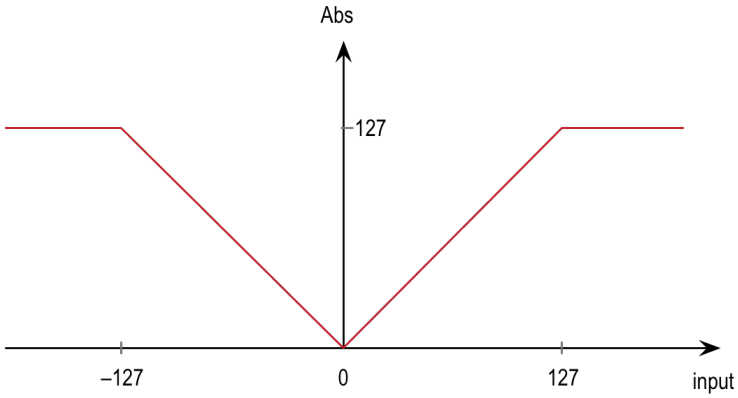
`activate` (Optional)

This key describes whether to activate the layer output (the default is to not activate). When specified, this key must be `ReLU`, `Abs` or `None` (the default). *Please note that there is always an implicit non-linearity when outputting 8-bit data since outputs are clamped to* $[-1, +127/128]$ *during training.*

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)

This key describes the width of the weight memory in bits and can be `1`, `2`, `4`, or `8` (the default is based on the range of the layer's weights). Specifying a `quantization` that is smaller than what the weights require results in an error message. By default, the value is automatically derived from the weights.

Example:

```
quantization: 4
```

`output_shift` (Optional)

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 and used by `quantization`. Note that `output_shift` is not supported for passthrough layers.

The 32-bit intermediate result is multiplied by $2^{totalshift}$, where the total shift count must be within the range $[-15, +15]$, resulting in a factor of $[2^{-15}, 2^{15}]$ or $[0.0000305176$ to $32768]$.

weight quantization	shift used by quantization	available range for `output_shift`
8-bit	0	$[-15, +15]$
4-bit	4	$[-19, +11]$
2-bit	6	$[-21, +9]$
1-bit	7	$[-22, +8]$

Using `output_shift` can help normalize data, particularly when using small weights. By default, `output_shift` is generated by the training software, and it is used for batch normalization as well as quantization-aware training.

Note: When using 32-bit wide outputs in the final layer, no hardware shift is performed and `output_shift` is ignored.

Example:
`output_shift: 2`

`kernel_size` (Optional)

- For `Conv2D`, this key must be `3x3` (the default) or `1x1`.
- For `Conv1D`, this key must be `1` through `9`.
- For `ConvTranspose2D`, this key must be `3x3` (the default).

Example:
`kernel_size: 1x1`

`stride` (Optional)

This key must be `1` or `[1, 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`, `1`, `2`, or `3` (the default).
- For `ConvTranspose2d`, this value can be `0`, `1` (the default), or `2`. *Note that the value follows PyTorch conventions and effectively adds* `(kernel_size - 1) - pad` *amount of zero padding to both sizes of the input, so "0" adds 2 zeros each and "2" adds no padding.*
- For `Passthrough`, this value must be `0` (the default).

`max_pool` (Optional)

When specified, performs a `MaxPool` before the convolution. The pooling size 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]`

(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. The pooling size 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]`.

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]`, where both values must be identical. The default is `1` or `[1, 1]`.

Example:
`pool_stride: [2, 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

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, and the output processors must be adjacent for all sequence arguments.

The order of arguments of `in_sequences` must match the model, and earlier arguments must use lower output processor numbers than later arguments. The following code shows an example `forward` function for a model that concatenates two values:

```
def forward(self, x):
    x = self.conv0(x)  # Layer 0
    y = self.conv1(x)  # Layer 1
    y = torch.cat((y, x), dim=1)
```

In this case, `in_sequences` would be `[1, 0]` since `y` (the output of layer 1) precedes `x` (the output of layer 0) in the `torch.cat()` statement. In the example, layer 0 uses output processors `0xff00` (processors 8-15) and layer 1 uses 0xff (processors 0-7).

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. *Note that flattening cannot be used in the same layer as pooling.*

Example:
`flatten: true`

`write_gap` (Optional)

`write_gap` specifies the number of channels that should be skipped during write operations (this value is multiplied with the output multi-pass, i.e., write every *n*th word where *n* = *write_gap* × *output_multipass*). This creates an interleaved output that can be used as the input for subsequent layers that use an element-wise operation, or to concatenate multiple inputs to form data with more than 64 channels.
Set `write_gap` to `1` to produce output for a subsequent two-input element-wise operation.

Example:
`write_gap: 1`

`bias_group` (Optional)

For layers that use a bias, this key can specify one or more bias memories that should be used. By default, the software uses a "Fit First Descending (FFD)" allocation algorithm that considers the largest bias lengths first, and then the layer number, and places each bias in the available group with the most available space, descending to the smallest bias length.

"Available groups" is the complete list of groups used by the network (in any layer). `bias_group` must reference one or more of these available groups.

`bias_group` can be a list of integers or a single integer.

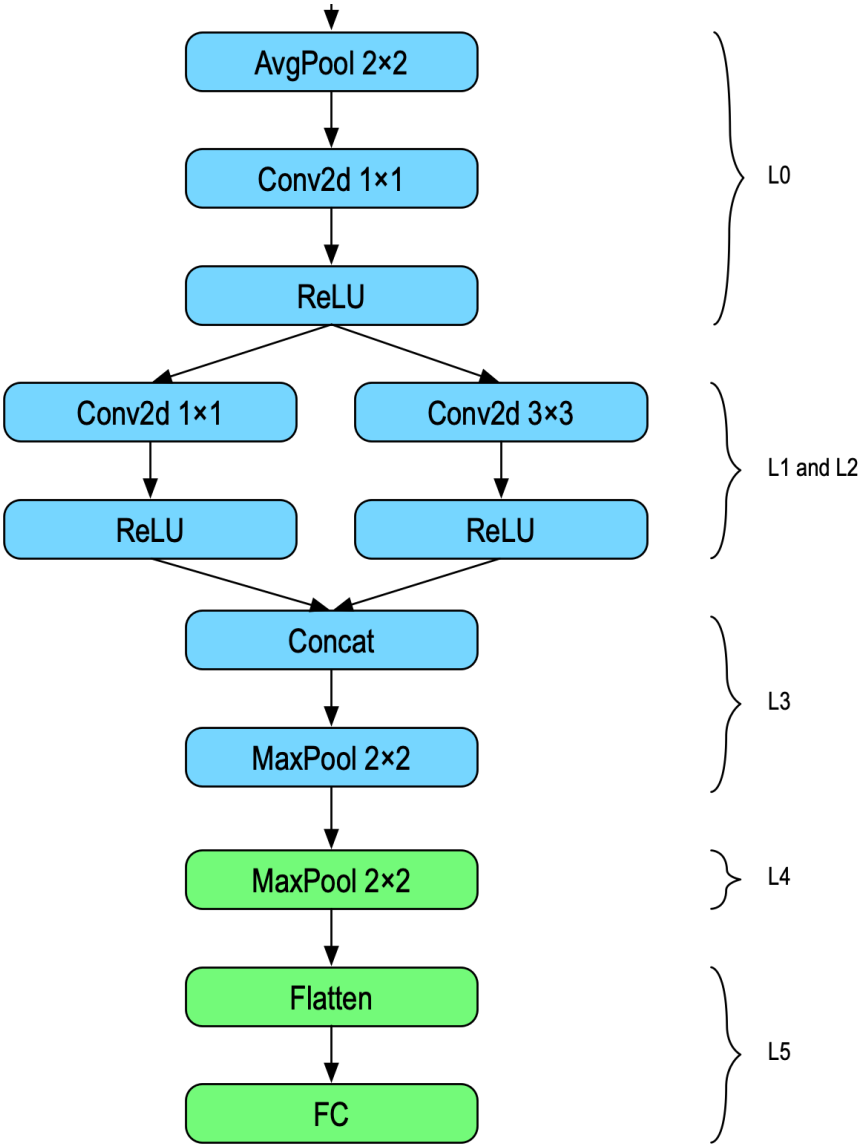
Example:
`bias_group: 0`

Dropout and Batch Normalization

- Dropout is only used during training, and corresponding YAML entries are not needed.
- Batch normalization ("batchnorm") is fused into the preceding layer's weights and bias values (see Batch Normalization), and YAML entries are not needed.

Example

The following shows an example for a single "Fire" operation, the MAX78000/MAX78002 hardware layer numbers and its YAML description.



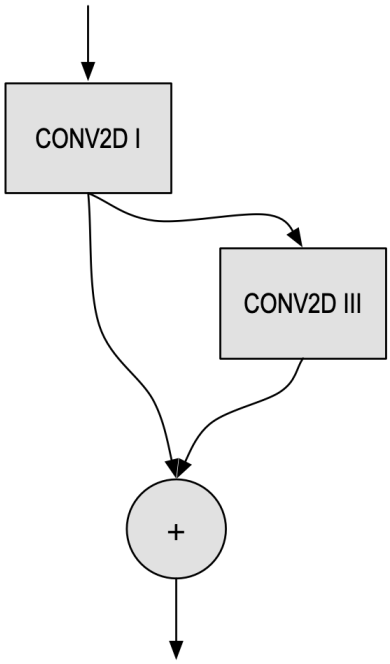

```yaml
arch: ai85firetestnet
dataset: CIFAR-10
# Input dimensions are 3x32x32

layers:
### Fire
# Squeeze
- 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
```

Residual Connections

Many networks use residual connections. In the following example, the convolution on the right works on the output data of the first convolution. However, that same output data also “bypasses” the second convolution

and is added to the output.



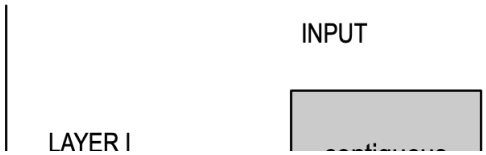
On MAX78000/MAX78002, the element-wise addition works on "interleaved data," i.e., each machine fetch gathers one operand.

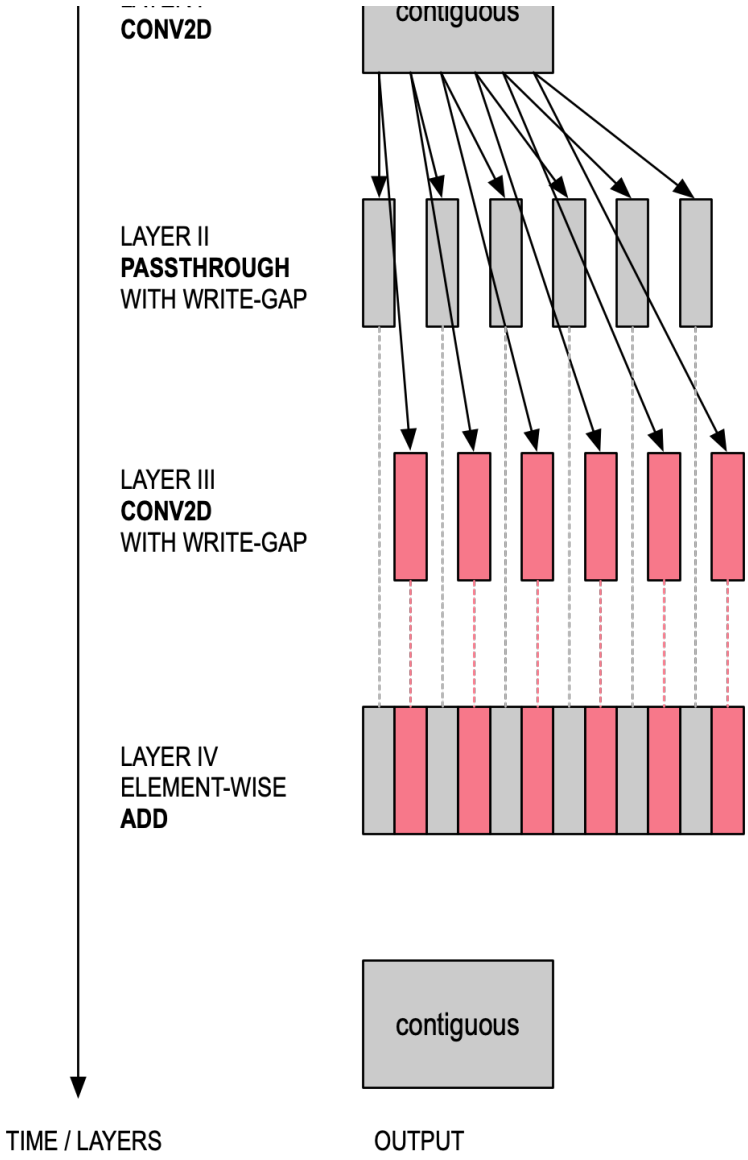
In order to achieve this, a layer must be inserted that does nothing else but reformat the data into interleaved format using the `write_gap` keyword (this operation happens in parallel and is fast).

```
# Layer 1
- out_offset: 0x0000
  processors: 0x0ffff00000000000
  operation: conv2d
  kernel_size: 3x3
  pad: 1
  activate: ReLU

# Layer 2 - re-format data with gap
- out_offset: 0x2000
  processors: 0x000000000000fffff
  output_processors: 0x000000000000fffff
  operation: passthrough
  write_gap: 1

# Layer 3
- in_offset: 0x0000
  out_offset: 0x2004
  processors: 0x000000000000fffff
```

```
  operation: conv2d
  kernel_size: 3x3
  pad: 1
  activate: ReLU
  write_gap: 1

# Layer 4 - Residual
- in_sequences: [2, 3]
  in_offset: 0x2000
  out_offset: 0x0000
  processors: 0x00000000000fffff
  eltwise: add
  ...
```

The same network can also be viewed graphically:





Adding New Models and New Datasets to the Network Loader

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

1. Provide the network model, its YAML description and weights. Place the YAML file (e.g., `new.yaml`) in the `networks` directory, and weights in the `trained` directory.
   The non-quantized weights are obtained from a training checkpoint, for example:
   `(ai8x-synthesis) $ cp ../ai8x-training/logs/2020.06.02-154133/best.pth.tar trained/new-unquantized.pth.tar`

2. When using post-training quantization, the quantized weights are the result of the quantization step. Copy and customize an existing quantization shell script, for example:
   `(ai8x-synthesis) $ cp scripts/quantize_mnist.sh scripts/quantize_new.sh`

Then, *edit this script to point to the new model and dataset* (`vi scripts/quantize_new.sh`), and call the script to generate the quantized weights. Example:

```
(ai8x-synthesis) $ scripts/quantize_new.sh
Configuring device: MAX78000.
Reading networks/new.yaml to configure network...
Converting checkpoint file trained/new-unquantized.pth.tar to trained/new.pth.tar
+----------------------+-------------+----------+
| Key                  | Type        | Value    |
|----------------------+-------------+----------|
| arch                 | str         | ai85net5 |
| compression_sched    | dict        |          |
| epoch                | int         | 165      |
| extras               | dict        |          |
| optimizer_state_dict | dict        |          |
| optimizer_type       | type        | SGD      |
| state_dict           | OrderedDict |          |
+----------------------+-------------+----------+
Model keys (state_dict):
conv1.conv2d.weight, conv2.conv2d.weight, conv3.conv2d.weight,
conv4.conv2d.weight, fc.linear.weight, fc.linear.bias
conv1.conv2d.weight avg_max: tensor(0.3100) max: tensor(0.7568) mean:
tensor(0.0104) factor: 54.4 bits: 8
conv2.conv2d.weight avg_max: tensor(0.1843) max: tensor(0.2897) mean:
tensor(-0.0063) factor: 108.8 bits: 8
conv3.conv2d.weight avg_max: tensor(0.1972) max: tensor(0.3065) mean:
tensor(-0.0053) factor: 108.8 bits: 8
conv4.conv2d.weight avg_max: tensor(0.3880) max: tensor(0.5299) mean:
tensor(0.0036) factor: 108.8 bits: 8
fc.linear.weight avg_max: tensor(0.6916) max: tensor(0.9419) mean: tensor(-0.0554)
factor: 108.8 bits: 8
fc.linear.bias
```

3. 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()`, see below.

   For example, the MNIST 1×28×28 image sample would be stored in `tests/sample_mnist.npy` in an `np.array` with shape `[1, 28, 28]` and datatype `<i8`. The file can be random, or can be obtained from the `train.py` software.

Generating a Random Sample Input

To generate a random sample input, use a short NumPy script. In the grayscale MNIST example:

```
import os
import numpy as np

a = np.random.randint(-128, 127, size=(1, 28, 28), dtype=np.int64)
np.save(os.path.join('tests', 'sample_mnist'), a, allow_pickle=False,
fix_imports=False)
```

For RGB image inputs, there are three channels. For example, a 3×80×60 (C×H×W) input is created using `size=(3, 80, 60)`.

Saving a Sample Input from Training Data

1. In the `ai8x-training` project, add the argument `--save-sample 10` to the `scripts/evaluate_mnist.sh` script. *Note: The index 10 is arbitrary, but it must be smaller than the batch size. If manual visual verification is desired, it is a good idea to pick a sample where the quantized model computes the correct answer.*
2. Run the modified `scripts/evaluate_mnist.sh`. It will produce a file named `sample_mnist.npy`.
3. Save the `sample_mnist.npy` file and copy it to the `ai8x-synthesis` project.

Evaluate the Quantized Weights with the New Dataset and Model

1. Switch to training project directory and activate the environment:

   ```
   (ai8x-synthesis) $ deactivate`
   $ cd ../ai8x-training
   $ source bin/activate
   ```

2. Create an evaluation script and run it:

   ```
   (ai8x-training) $ cp scripts/evaluate_mnist.sh scripts/evaluate_new.sh
   (ai8x-training) $ vim scripts/evaluate_new.sh
   (ai8x-training) $ scripts/evaluate_new.sh
   ```

   Example output:

   ```
   (ai8x-training) $ scripts/evaluate_new.sh
   Configuring device: MAX78000, simulate=True.
   Log file for this run: logs/2020.06.03-125328/2020.06.03-125328.log
   ------------------------------------------------------------
   Logging to TensorBoard - remember to execute the server:
   > tensorboard --logdir='./logs'

   => loading checkpoint ../ai8x-synthesis/trained/new.pth.tar
   => Checkpoint contents:
   +----------------------+-------------+----------+
   ```

```
| Key                  | Type        | Value     |
|----------------------+-------------+-----------|
| arch                 | str         | ai85net5  |
| compression_sched    | dict        |           |
| epoch                | int         | 165       |
| extras               | dict        |           |
| optimizer_state_dict | dict        |           |
| optimizer_type       | type        | SGD       |
| state_dict           | OrderedDict |           |
+----------------------+-------------+-----------+

=> Checkpoint['extras'] contents:
+-----------------+--------+--------------------+
| Key             | Type   | Value              |
|-----------------+--------+--------------------|
| best_epoch      | int    | 165                |
| best_top1       | float  | 99.46666666666667  |
| clipping_method | str    | SCALE              |
| clipping_scale  | float  | 0.85               |
| current_top1    | float  | 99.46666666666667  |
+-----------------+--------+--------------------+

Loaded compression schedule from checkpoint (epoch 165)
=> loaded 'state_dict' from checkpoint '../ai8x-synthesis/trained/new.pth.tar'
Optimizer Type: <class 'torch.optim.sgd.SGD'>
Optimizer Args: {'lr': 0.1, 'momentum': 0.9, 'dampening': 0, 'weight_decay':
0.0001, 'nesterov': False}
Dataset sizes:
  training=54000
  validation=6000
  test=10000
--- test ---------------------
10000 samples (256 per mini-batch)
Test: [   10/   40]    Loss 44.193750    Top1 99.609375    Top5 99.960938
Test: [   20/   40]    Loss 66.567578    Top1 99.433594    Top5 99.980469
Test: [   30/   40]    Loss 51.816276    Top1 99.518229    Top5 99.986979
Test: [   40/   40]    Loss 53.596094    Top1 99.500000    Top5 99.990000
==> Top1: 99.500    Top5: 99.990    Loss: 53.596

==> Confusion:
[[ 979    0    0    0    0    0    0    0    1    0]
 [   0 1132    1    0    0    0    0    2    0    0]
 [   2    0 1026    1    0    0    0    3    0    0]
 [   0    0    0 1009    0    0    0    0    1    0]
 [   0    0    0    0  978    0    0    0    0    4]
 [   1    0    0    3    0  886    1    0    0    1]
```

```
[   5    4    1    0    1    0  946    0    1    0]
[   0    1    3    0    0    0    0 1023    0    1]
[   0    0    0    1    1    1    0    0  970    1]
[   0    0    0    0    4    1    0    3    0 1001]]

Log file for this run: logs/2020.06.03-125328/2020.06.03-125328.log
```

Generating C Code

Run `ai8xize.py` with the new network and the new sample data to generate embedded C code that can be compiled with the Arm and RISC-V compilers. See `gen-demos-max78000.sh` for examples.

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

An inference is started by configuring registers and weights, loading the input, and enabling processing. This code is automatically generated—see the `cnn_init()`, `cnn_load_weights()`, `cnn_load_bias()`, `cnn_configure()`, 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.
To start the accelerator, use `cnn_start()`. The `load_input()` function is called either before `cnn_start()`, or after `cnn_start()`, depending on whether FIFOs are used. To run a second inference with new data, call `cnn_start()` again (after or before loading the new data input using load_input()`).

The MAX78000/MAX78002 accelerator can generate an interrupt on completion, and it will set a status bit (see `cnn.c`). The resulting data can now be unloaded from the accelerator (code for this is also auto-generated in `cnn_unload()`).

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

Overview of the Functions in main.c

The generated code is split between API code (in `cnn.c`) and data-dependent code in `main.c` or `main_riscv.c`. The data-dependent code is based on a known-answer test. The `main()` function shows the proper sequence of steps to load and configure the CNN accelerator, run it, unload it, and verify the result.

`void load_input(void);`
Load the example input. This function can serve as a template for loading data into the CNN accelerator. Note that the clocks and power to the accelerator must be enabled first. If this is skipped, the device may appear to hang and the recovery procedure may have to be used.

`int check_output(void);`
This function verifies that the known-answer test works correctly in hardware (using the example input). This function is typically not needed in the final application.

`int main(void);`
This is the main function and can serve as a template for the user application. It shows the correct sequence of operations to initialize, load, run, and unload the CNN accelerator.

Overview of the Generated API Functions

The API code (in `cnn.c` by default) is auto-generated. It is data-independent, but differs depending on the network. This simplifies replacing the network while keeping the remainder of the code intact.

The functions that do not return `void` return either `CNN_FAIL` or `CNN_OK` as specified in the auto-generated `cnn.h` header file. The header file also contains a definition for the number of outputs of the network (`CNN_NUM_OUTPUTS`). In limited circumstances, this can make the wrapper code somewhat network-independent.

`int cnn_enable(uint32_t clock_source, uint32_t clock_divider);`
Enable clocks (from `clock_source` with `clock_divider`) and power to the accelerator; also enable the accelerator interrupt. By default, on MAX78000, the accelerator runs at 50 MHz (APB clock or PCLK divided by 1).

`int cnn_disable(void);`
Disable clocks and power to the accelerator.

`int cnn_init(void);`
Perform minimum accelerator initialization so it can be configured or restarted.

`int cnn_configure(void);`
Configure the accelerator for the given network.

`int cnn_load_weights(void);`
Load the accelerator weights. Note that `cnn_init()` must be called before loading weights after reset or wake from sleep.

`int cnn_verify_weights(void);`
Verify the accelerator weights (used for debug only).

`int cnn_load_bias(void);`
Load accelerator the bias values (if needed).

`int cnn_start(void);`
Start accelerator processing.

`int cnn_stop(void);`
Force-stop the accelerator regardless of whether it has finished or not.

`int cnn_continue(void);`
Continue accelerator processing after force-stopping it.

`int cnn_unload(uint32_t *out_buf);`

Unload the results from the accelerator. The output buffer must be 32-bit aligned (round up to the next 32-bit size when using 8-bit outputs).

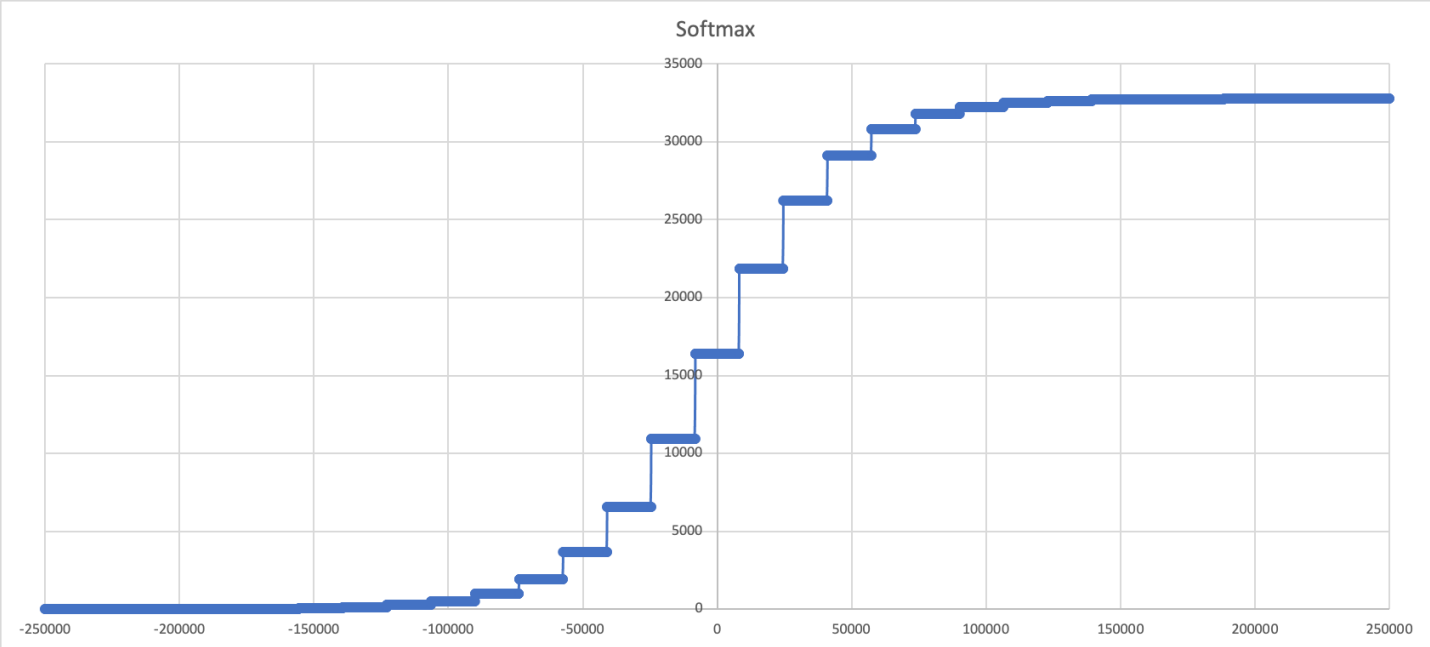
`int cnn_boost_enable(mxc_gpio_regs_t *port, uint32_t pin);`
Turn on the boost circuit on `port`.`pin`. This is only needed for very energy intense networks. Use the `--boost` command line option to insert calls to this function in the wrapper code.

`int cnn_boost_disable(mxc_gpio_regs_t *port, uint32_t pin);`
Turn off the boost circuit connected to `port`.`pin`.

Softmax, and Data Unload in C

`ai8xize.py` can generate a call to a software Softmax function using the command line switch `--softmax`. That function is provided in the `assets/device-all` folder. To use the provided software Softmax on MAX78000/MAX78002, the last layer output should be 32-bit wide (`output_width: 32`).

The software Softmax function is optimized for processing time, and it quantizes the input. When the last layer uses weights that are not 8-bits, the software function used will shift the input values first.



Generated Files and Upgrading the CNN Model

The generated C code comprises the following files. Some of the files are customized based on the project name, and some are custom for a combination of project name and weight/sample data inputs:

File name	Source	Project specific?	Model/weights change?
Makefile	template in assets/embedded-ai87	Yes	No
cnn.c	generated	Yes	**Yes**

cnn.h	template in assets/device-all	Yes	**Yes**
weights.h	generated	Yes	**Yes**
log.txt	generated	Yes	**Yes**
main.c	generated	Yes	No
sampledata.h	generated	Yes	No
sampleoutput.h	generated	Yes	**Yes**
softmax.c	assets/device-all	No	No
model.launch	template in assets/eclipse	Yes	No
.cproject	template in assets/eclipse	Yes	No
.project	template in assets/eclipse	Yes	No

In order to upgrade an embedded project after retraining the model, point the network generator to a new empty directory and regenerate. Then, copy the four files that will have changed to your original project — `cnn.c`, `cnn.h`, `weights.h`, and `log.txt`. This allows for persistent customization of the I/O code and project (for example, in `main.c` and additional files) while allowing easy model upgrades.

The generator also adds all files from the `assets/eclipse`, `assets/device-all`, and `assets/embedded-ai87` folders. These files (when starting with `template` in their name) will be automatically customized to include project-specific information as shown in the following table:

Key	Replaced by
`##__PROJ_NAME__##`	Project name (works on file names as well as the file contents)
`##__ELF_FILE__##`	Output elf (binary) file name
`##__BOARD__##`	Board name (e.g., `EvKit_V1`)
`##__FILE_INSERT__##`	Network statistics and timer

Contents of the device-all Folder

- For MAX78000/MAX78002, the software Softmax is implemented in `softmax.c`.
- A template for the `cnn.h` header file in `templatecnn.h`. The template is customized during code generation using model statistics and timer, but uses common function signatures for all projects.

Determining the Compiled Flash Image Size

The generated `.elf` file (either `max78000.elf` or `max78000-combined.elf`) contains debug and other meta information. To determine the true Flash image size, either examine the `.map` file, or convert the `.elf` to a

binary image and examine the resulting image.

```
% arm-none-eabi-objcopy -I elf32-littlearm build/max78000.elf -O binary temp.bin
% ls -la temp.bin
-rwxr-xr-x  1 user  staff  321968 Jan  1 11:11 temp.bin
```

Handling Linker Flash Section Overflows

When linking the generated C code, the code space might overflow:

```
$ make
  CC    main.c
  CC    cnn.c
  ...
  LD    build/max78000.elf
arm-none-eabi/bin/ld: build/max78000.elf section `.text' will not fit in region `FLASH'
arm-none-eabi/bin/ld: region `FLASH' overflowed by 600176 bytes
collect2: error: ld returned 1 exit status
```

The most likely reason is that the input is too large (from `sampledata.h`), or that the expected output (in `sampleoutput.h`) is too large. It is important to note that this only affects the generated code with the built-in known-answer test (KAT) that will not be part of the user application since normal input and output data are not predefined in Flash memory.

To deal with this issue, there are several options:

- The sample input data can be stored in external memory. This requires modifications to the generated code. Please see the SDK examples to learn how to access external memory.
- The sample input data can be programmatically generated. Typically, this requires manual modification of the generated code, and a corresponding modification of the sample input file.
  The generator also contains a built-in generator (supported *only* when using `–fifo`, and only for HWC inputs); the command line option `--synthesize-input` uses only the first few words of the sample input data, and then adds the specified value N (for example, 0x112233 if three input channels are used) to each subsequent set of M 32-bit words. M can be specified using `--synthesize-words` and defaults to 8. Note that M must be a divisor of the number of pixels per channel.
- The output check can be truncated. The command line option `--max-verify-length` checks only the first N words of output data (for example, 1024). To completely disable the output check, use `--no-kat`.
- For 8-bit output values, `--mlator` typically generates more compact code.
- Change the compiler optimization level in `Makefile`. To change the default optimization levels, modify `MXC_OPTIMIZE_CFLAGS` in `assets/embedded-ai85/templateMakefile` for Arm code and

  `assets/embedded-riscv-ai85/templateMakefile.RISCV` for RISC-V code. Both `-O1` and `-Os` may result in smaller code compared to `-O2`.
- If the last layer has large-dimension, large-channel output, the `cnn_unload()` code in `cnn.c` may cause memory segment overflows not only in Flash, but also in the target buffer in SRAM (`ml_data32[]` or

`ml_data[]` in `main.c`). In this case, manual code edits are required to perform multiple partial unloads in sequence.

Debugging Techniques

There can be many reasons why the known-answer test (KAT) fails for a given network with an error message, or where the KAT does not complete. The following techniques may help in narrowing down where in the network or the YAML description of the network the error occurs:

- For very short and small networks, disable the use of WFI instructions while waiting for completion of the CNN computations by using the command line option `--no-wfi`. *Explanation: In these cases, the network terminates more quickly than the time it takes between testing for completion and executing the WFI instruction, so the WFI instruction is never interrupted and the code may appear to hang.*
- The `--no-wfi` option can also be useful when trying to debug code, since the debugger loses connection when the device enters sleep mode using `__WFI()`.
- By default, there is a two-second delay at the beginning of the code. This time allows the debugger to take control before the device enters any kind of sleep mode. `--no-wfi` disables sleep mode. The time delay can be modified using the `--debugwait` option.
  If the delay is too short or skipped altogether, and the device does not wake at the end of execution, the device may appear to hang and the recovery procedure may have to be used in order to load new code or to debug code.
- For very large and deep networks, enable the boost power supply using the `--boost` command line option. On the EVkit, the boost supply is connected to port pin P2.5, so the command line option is `--boost 2.5`.
- The default compiler optimization level is `-O2`, and incorrect code may be generated under rare circumstances. Lower the optimization level in the generated `Makefile` to `-O1`, clean (`make distclean && make clean`), and rebuild the project (`make`). If this solves the problem, one of the possible reasons is that code is missing the `volatile` keyword for certain variables.
  To permanently adjust the default compiler optimization level, modify `MXC_OPTIMIZE_CFLAGS` in `assets/embedded-ai85/templateMakefile` for Arm code and `assets/embedded-riscv-ai85/templateMakefile.RISCV` for RISC-V code.
- `--stop-after N` where `N` is a layer number may help to find the problematic layer by terminating the network early without having to retrain and without having to change the weight input file. Note that this may also require `--max-verify-length` as described above since intermediate outputs tend to be large, and additionally `--no-unload` to suppress generation of the `cnn_unload()` function.
- `--no-bias LIST` where `LIST` is a comma-separated list of layers (e.g., `0,1,2,3`) can rule out problems due to the bias. This option zeros out the bias for the given layers without having to remove bias values from the weight input file.
- `--ignore-streaming` ignores all `streaming` statements in the YAML file. Note that this typically only works when the sample input is replaced with a different, lower-dimension sample input (for example, use 3×32×32 instead of 3×128×128), and does not support fully connected layers without retraining (use `--stop-after` to remove final layers). Ensure that the network (or partial network when using `--stop-`

`after`) does not produce all-zero intermediate data or final outputs when using reduced-dimension inputs. The log file (`log.txt` by default) will contain the necessary information.

Energy Measurement

The MAX78000 Evaluation Kit (EVKit) revision C and later includes a MAX32625 microcontroller connected to a MAX34417 power accumulator. Since the sample rate of the MAX34417 is slow compared to typical inference times, `ai8xize.py` supports the command line parameter `--energy` that will run 100 iterations of the inference, separating out the input data load time. This allows enough sample time to get meaningful results (recommended minimum: 1 second).

When running C code generated with `--energy`, the power display on the EVKit will display the inference energy.

Note: MAX78000 uses LED1 and LED2 to trigger power measurement via MAX32625 and MAX34417.

See the benchmarking guide for more information about benchmarking.

Further Information

Additional information about the evaluation kits, and the software development kit (SDK) is available on the web at https://github.com/MaximIntegratedAI/MaximAI_Documentation

---

AHB Memory Addresses

The following tables show the AHB memory addresses for the MAX78000 accelerator:

Data memory

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

Group	Instance	Address Range
0	0	0x50400000 - 0x50407FFF
0	1	0x50408000 - 0x5040FFFF

0	1	0x50408000 - 0x5040FFFF
0	2	0x50410000 - 0x50417FFF
0	3	0x50418000 - 0x5041FFFF
1	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	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	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

Contributing Code

Linting

Both projects are set up for `flake8`, `pylint` and `isort` to lint Python code. 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. Shell code is linted by `shellcheck`, and YAML files by `yamllint`.
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` and `pylint` need to be installed into both virtual environments:

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

The GitHub projects use the GitHub Super-Linter to automatically verify push operations and pull requests. The Super-Linter can be installed locally, see installation instructions.
To run locally, create a clean copy of the repository and run the following command from the project directory (i.e., `ai8x-training` or `ai8x-synthesis`):

```
$ docker pull github/super-linter:latest
$ docker run --rm -e RUN_LOCAL=true -e VALIDATE_MARKDOWN=false -e
VALIDATE_PYTHON_BLACK=false -e VALIDATE_ANSIBLE=false -e VALIDATE_EDITORCONFIG=false -e
VALIDATE_JSCPD=false -e FILTER_REGEX_EXCLUDE="attic/.*|inspect_ckpt.py" -v
`pwd`:/tmp/lint github/super-linter
```

Submitting Changes

Do not try to push any changes into the master branch. Instead, create a fork and submit a pull request against the `develop` branch. The easiest way to do this is using a graphical client such as Fork or GitHub Desktop.

Note: After creating the fork, you must re-enable actions in the "Actions" tab of the repository on GitHub.

The following document has more information:
https://github.com/MaximIntegratedAI/MaximAI_Documentation/blob/master/CONTRIBUTING.md