CrossSim Inference Manual (v2.0)

T. Patrick Xiao, Christopher H. Bennett, Ben Feinberg,
Matthew J. Marinella, Sapan Agarwal

Sandia National Laboratories

Last updated: July 25, 2022

Contents

Acronyms

Acronym	Expansion
1T1R	One transistor, one resistor
ADC	Analog-to-digital converter
API	Application programming interface
BGR	Blue green red
CIFAR-10/100	Canadian Institute for Advanced Research image recognition dataset with 10 or 100 classes
CNN	Convolutional neural network
CPU	Central processing unit
CUDA	Compute Unified Device Architecture, developed by Nvidia
DAC	Digital-to-analog converter
FET	Field-effect transistor
FP	Floating-point
GPU	Graphics processing unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
MNIST	Modified National Insitute of Standards and Technology dataset of handwritten digits
MVM	Matrix-vector multiplication
PCM	Phase-change memory
ReLU	Rectified linear unit
ReRAM	Resistive random access memory
SONOS	Silicon-oxide-nitride-oxide-silicon

Chapter 1

Introduction

1.1 Analog in-memory accelerators for neural networks

Neural networks are largely based on matrix computations. During forward inference, the most heavily used compute kernel is the matrix-vector multiplication (MVM): $\mathbf{W}\vec{x}$. Inference is a first frontier for the deployment of next-generation hardware for neural network applications, as it is more readily deployed in edge devices, such as mobile devices or embedded processors with size, weight, and power constraints. Inference is also easier to implement in analog systems than training, which has more stringent device requirements. The main processing kernel used during inference is the MVM.

Analog in-memory computing can potentially provide orders-of-magnitude reductions in the energy consumption of neural network inference, compared to digital accelerators. This comes from the efficiency of computing an MVM using analog electrical signals. This is shown *conceptually* in Fig. 1.1. The memory cell conductances are set proportional to the values of the matrix $\mathbf{W}$ and a vector of input voltages $\vec{V}$, proportional to $\vec{x}$, is applied to the rows. An analog multiplication is computed at each cell, where the current is the product of its conductance G_{ij} and the applied voltage V_i. Kirchoff's law then accumulates these products on the bit line (column) current I_j to form the dot product. The analog dot products are subsequently quantized using an analog-to-digital converter (ADC). The dot products can then be digitally processed and transmitted to the next layer of the neural network.

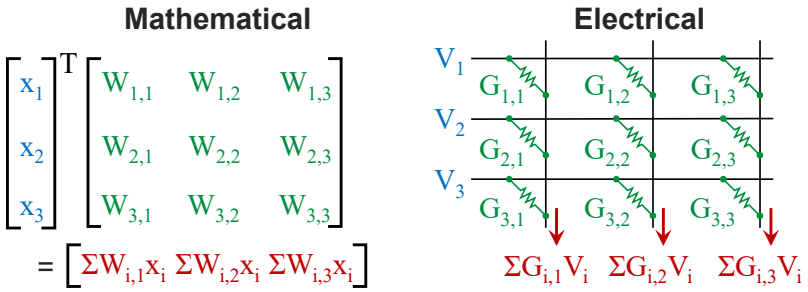

Figure 1.1: (Left) Mathematical representation of an MVM, $\mathbf{W}\vec{x} = \vec{y}$. The two sides of the equation have been transposed for illustration purposes. (Right) Implementation of the MVM in the analog electrical domain using circuit laws within a resistive memory array.

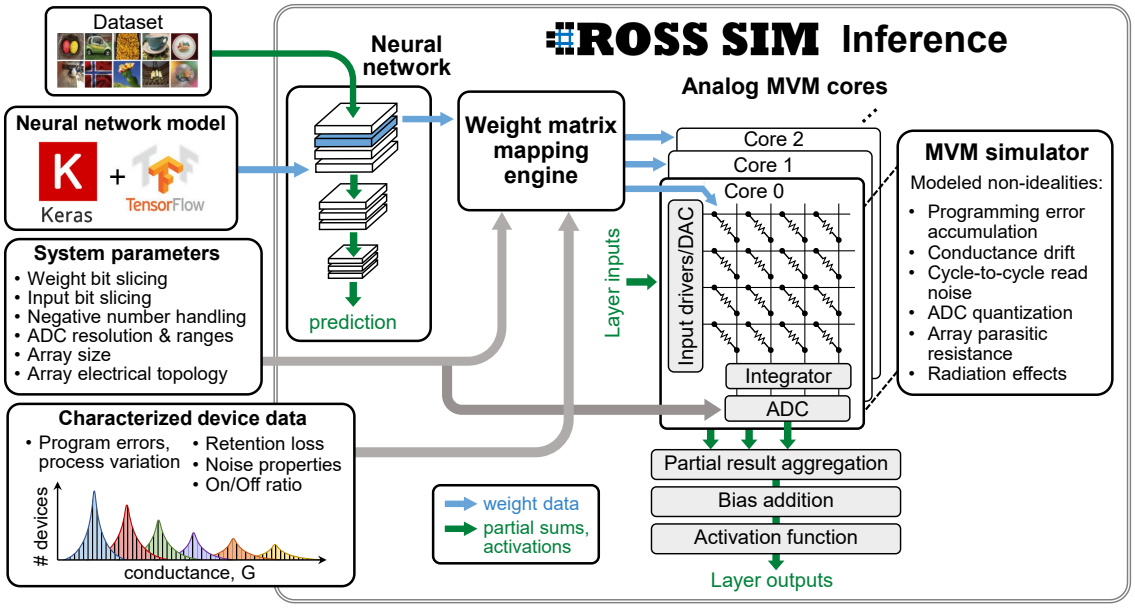

Figure 1.2: Structure of CrossSim inference.

This analog approach has two fundamental advantages for energy efficiency:

- A multi-bit multiplication is conducted with a single device, and summations require no special hardware (just wire crossings).
- The weight matrix does not need to be read out; only the inputs and outputs need to be communicated between processing cores. This dramatically reduces data movement energy.

In-memory MVM has been demonstrated using a wide variety of memory cell technologies. These include two-terminal resistive memories, such as resistive random access memory (ReRAM) and phase change memory (PCM). Transistor-based memories such as floating-gate flash, charge trapping flash, and ferroelectric FET have also be used. Besides the diversity of memory technologies used, there are a variety of practical implementations of the MVM in Fig. 1.1 that differ at the circuit or system level. All of these can be modeled by CrossSim Inference.

CrossSim Inference does *not* explicitly model the energy, area, or speed of analog inference accelerators. Its primary goal is to model the accuracy of neural network inference when implemented in an analog system. Despite its efficiency, analog computing has historically struggled with accuracy and precision, compared to digital computers that can compute at arbitrary precision. Neural network processing, however, presents a new opportunity for analog. Because the purpose of neural networks is to generalize to unseen situations, they must necessarily have some built-in tolerance to error. Whether the error tolerance of an algorithm or application is compatible with the amount of error in an analog system is generally not clear. This is the fundamental question that CrossSim Inference is designed to answer.

1.2 Software overview

1.2.1 CrossSim Inference

Fig. 1.2 shows the software structure of CrossSim Inference, which in this manual will be referred to simply as CrossSim for brevity. The software is written in Python, which interfaces readily with TensorFlow/Keras, a popular machine learning framework [7]. In general terms, the tool takes four inputs to an inference simulation:

1. The neural network model, provided as a Keras H5 file that contains the model's topology and weights.
2. A set of hardware parameters, specified in a Python configuration file, from the device level to the system level, that specifies the analog inference accelerator. This includes design choices for mapping data to physical parameters, the numerical values of the activation and ADC ranges (provided in separate NumPy data files).
3. Characterized data on devices to use for inference, most importantly the error distribution of programmed currents. If no technology is specified, generic memory device error models are also available.
4. The dataset (inputs and labels) on which to simulate inference. CrossSim currently supports several widely-used image classification datasets for benchmarking.

Below, we give an overview of the main processing steps in a CrossSim inference simulation, after a hardware configuration file is provided by the user.

First, CrossSim parses the metadata of the Keras model to build an internal representation of the neural network. Currently, CrossSim supports various layer types and features that are commonly used in state-of-the-art convolutional neural networks (CNNs), including convolutions (standard and depthwise), skip connections, element-wise operations, and concatenations. These can be composed to form, for example, residual blocks in ResNets [11] or inception modules in Inception nets [20]. We follow the conventions in Keras for setting the padding and stride in convolutions and pooling operations. CrossSim also supports quantized models that were trained using Sandia's Whetstone framework [18] or the Larq library [9].

After building the internal representation of the neural network topology, weights from every layer in the Keras model are imported. The weight matrix is first quantized to the desired weight resolution, if they are not already quantized in the Keras model. Then, if specified, batch normalization parameters are folded into the weight matrices of the preceding convolution layer where possible [12]. Since batch normalization is a static linear transformation during inference operations, this folding step reduces the digital processing overhead in an analog inference accelerator.

The resulting weight matrix is then decomposed into arrays of memory cell conductances according to the chosen mapping. CrossSim represents a layer's weight matrix using a collection of analog MVM cores. Each analog core consists of the computational memory array and models of its peripheral circuitry and ADCs. A weight matrix may be broken up into multiple cores, depending on the user-specified system parameters. Partitioning can be done for several reasons:

- A positive core and negative core if using a differential scheme to represent negative numbers.
- Cores for different bit slices of the digital weight values.
- Spatial partitioning of a large matrix into several smaller-sized cores.

For example, if using differential representation with 2 bits/cell and a maximum of 72 rows per array, an 8-bit 4608×512 matrix (the largest matrix in ResNet50) is mapped onto $2 \times 4 \times 64 = 512$

analog cores. Notably, "core" is merely an abstraction in CrossSim: these cores do not necessarily need to be implemented as physically separate arrays in hardware. The provided activation ranges and ADC ranges (if applicable) are loaded into the cores along with the weights. Different cores for the same layer can have different ADC ranges if they belong to different weight bit slices. CrossSim provides a way to collect statistics on the ADC inputs so that the ADC ranges for each core can be calibrated for a given validation dataset and neural network. This is described in Section 9.4.

To model memory device programming errors, CrossSim applies a random conductance error ΔG to every device in the system. The programming error is sampled from a normal distribution with zero mean and standard deviation σ_G based on the provided device data; σ_G can vary as a function of G. If time-dependent data on device errors are provided, CrossSim can also apply a time-dependent drift to the original conductance in addition to a time-dependent random error; analytical expressions can also be used to model drift. After they are applied, programming errors are considered static: they do not change across MVMs or images within an inference simulation.

Input data from the dataset are then loaded into CrossSim and supplied to the first layer of the network. The analog cores are used for the MVM layers (convolutions and fully-connected layers) while other operations such as pooling, activation functions, element-wise additions and inter-layer communication are assumed to be implemented without errors in the digital domain. After the digitization step that occurs separately within each analog core, the digital results from different cores within a layer are aggregated via subtraction (for differential cells), addition (for row-wise spatial partitions), or shift-and-add accumulation (for weight bit slices) to produce the full MVM result. A convolution is unrolled into a sequence of sliding window MVMs as proposed by Shafiee *et al.* [19] and described in Section 6.1.

CrossSim models every analog MAC during inference using the conductance values with errors as described above. Within the analog core, cycle-to-cycle read noise is modeled by adding a random, normally-distributed element-wise perturbation to the conductance matrix (with programming errors already applied) with every MVM. ADC quantization is modeled using a numerical rounding operation, given an ADC resolution and a specified range for the ADC inputs. The effects of array parasitic resistance are modeled using an internal iterative circuit solver. The circuit solver is specialized to a resistive array and makes certain linearizing assumptions about the memory cell devices in order to more rapidly simulate MVMs using fast matrix operations. This circuit solver is described in Section 8.3.

Inference simulations in CrossSim can be accelerated by CUDA GPUs, via the CuPy package for Python. If this option is enabled, all weights and all intermediate data (from the input images to the network outputs) are stored in GPU memory and processed there. We find that the amount of GPU speedup is greatest if a very large amount of computation can be packed into a single matrix arithmetic operation. To this end, CrossSim models multiple analog MVMs within a single matrix operation using a block-diagonal conductance matrix. The number of MVMs to pack together can be selected on a per-layer basis to optimize performance given the available GPU memory.

1.2.2 The CrossSim back end

This manual mainly describes how to use the front-end user interface of CrossSim inference to run simulations of analog neural network inference. These front-end scripts call into low-level methods in the CrossSim backend to perform the simulation. The same back-end supports both inference and training simulations. In general, the user does not need to be familiar with the backend to use CrossSim Inference. Here, we give a very brief overview of the backend software structure, intended

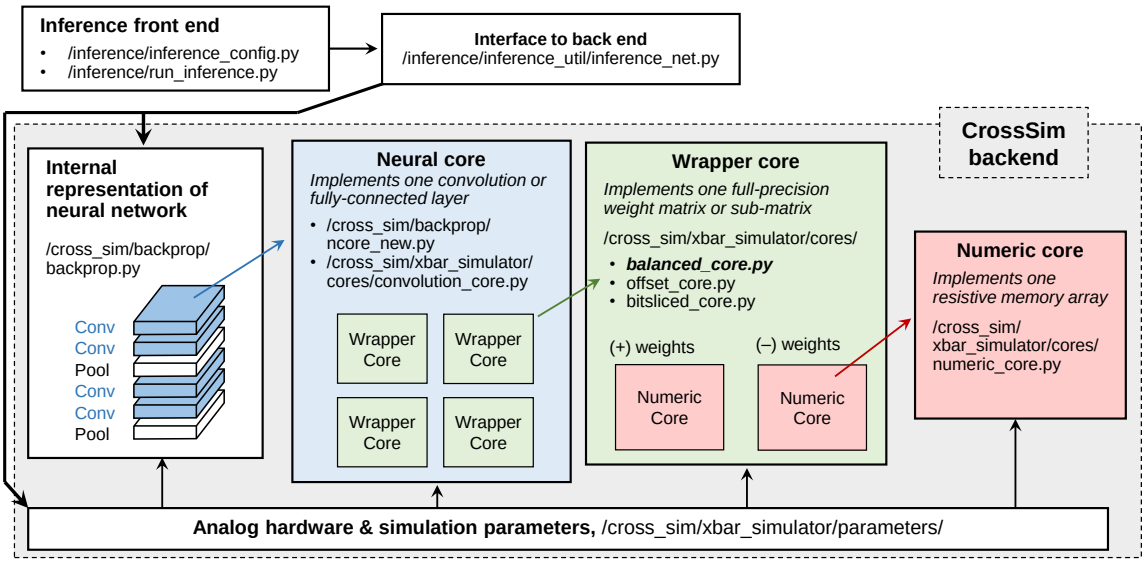

Figure 1.3: Structure of CrossSim's backend, and how it is called by CrossSim Inference.

for readers with a potential interest in adapting or extending the backend.

Fig. 1.3 shows how CrossSim Inference calls into the backend. The user specifies the simulation settings in `inference_config.py` and calls `run_inference.py` to execute the simulation, as described in Chapter 3. The functions in `inference_net.py` then sets the backend simulation parameters and orchestrates the inference simulation: maps the neural network to the analog hardware, loads the weights, loads the dataset, steps through the dataset, and finally returns the accuracy.

The full analog accelerators is represented through several hierarchical abstraction layers, each of which is represented as an object type. From the highest to lowest level:

1. The *full neural network* is implemented in `backprop.py` as a collection of neural cores supplemented by various digital functions. Only layers involving MVMs are allocated to neural cores. The remaining layers (such as pooling and activations) are implemented as error-free digital operations. Despite the name, `backprop.py` is used for both inference and training.
2. A *neural core* implements the kernel for a single MVM layer of the neural network: either a convolutional layer or a fully-connected layer. The neural core contains one or more wrapper core objects representing pieces of the full-sized weight matrix, if the weight matrix is large.
3. A *wrapper core* implements one full-precision piece of the weight matrix. This means that the wrapper core fully contains the positive and negative parts of the weight values, as well as all bits of significance of the weight values (e.g. all weight bit slices), but may not contain all of the elements in the weight matrix. The wrapper core is partitioned into one or more numeric cores, and is organized depending on how the user specifies the weight values to be mapped to conductances (Chapter 6).
4. The *numeric core* implements one resistive memory array with strictly positive device conductances. This is where device- and circuit-level analog models are implemented or called.

Finally, various parameter objects are used to control CrossSim's models at each hierarchical abstraction layer.

1.2.3 Top-level directory structure

The directories at the top level of CrossSim are summarized below:

- `/adc/`: contains scripts to calibrate ADC and activation ranges, and stores these ranges for various neural networks and hardware configurations (See Chapter 9)
- `/cross_sim/`:
  - `/cross_sim/`: contains the CrossSim back-end code
  - `/data/`: contains the provided datasets and lookup tables (for training)
- `/docs/`: documentation, including this manual
- `/examples/`: contains example scripts for directly accessing CrossSim's MVM cores, and lookup table generation from experimental data (for training)
- `/helpers/`: contains various helper functions
- `/inference/`: contains the front-end scripts for CrossSim Inference
- `/pretrained_models/`: contains pre-trained Keras neural networks models to be used with inference
- `/training/`: contains the front-end scripts for CrossSim Training

Chapter 2

Requirements and dependencies

CrossSim Inference is written using Python has been tested on Ubuntu Linux 18.04 and Windows 10. Testing was primarily performed using Python 3.7.6.

With GPU acceleration disabled, CrossSim can run on any CPU that supports the dependencies below. For larger simulations (such as ImageNet), it is strongly recommended to use a CUDA-capable GPU. CrossSim was tested on a NVidia V100 and a Nvidia Quadro RTX 4000, running CUDA 10.2. Based on the amount of available GPU memory, some simulation parameters may need to be adjusted to optimally run larger models (see Section 10.2).

CrossSim requires and has been tested on the following Python packages:

- TensorFlow 2.4.1 (includes Keras 2.4.0)
- NumPy 1.20.3
- CuPy 8.3.0 with CUDA 10.2, CuPy 10.3.1 with CUDA 11.2 (if GPU acceleration is enabled)
- SciPy 1.7.1
- Pandas 1.3.3
- MatPlotLib 3.4.3

Although the version numbers generally need not match exactly, we have found that using a different Keras version can lead to errors during import or differences in the results between Keras and CrossSim due to changes in the Keras API.

2.1 Model-specific requirements

Using some of the specific models provided with CrossSim Inference may require additional packages. To run the ImageNet pre-processing scripts needed for specific neural networks, the following packages may be required:

- OpenCV 4.5.4
- Torchvision 0.11.1

To simulate a neural network model that was quantized using Larq (see Section 5.1.2), the following package is required:

- Larq 0.12.1

2.2 Repository configuration

To reduce the size of the main CrossSim repository, the provided device lookup tables (for training), benchmark datasets (for training and inference), and Keras model files (for inference) are included in the separate `cross_sim_data` and `cross_sim_model` repositories, respectively. To ensure that these directories are placed in the appropriate locations, CrossSim uses git submodules. After cloning the repository, the following commands to configure and clone the submodules into the appropriate places in the directory structure. Note that the combined size of the two repositories is about 1.2GB (as of June 5, 2022) at download, and may take several minutes to clone. After download, the submodule repositories are de-compressed.

```
git submodule init
git submodule update --progress
```

If you would like to use a different data or models repositories, for instance to use CrossSim with a proprietary model, the remote locations of these repositories can be modified by changing the `url` field in the `.gitmodules` file in the main CrossSim directory before running the `git submodule init` and `git submodule update` commands above.

Chapter 3

Using CrossSim Inference

3.1 Running inference simulations

CrossSim Inference is run from the `inference` top-level directory. Inside this directory, there are two types of files:

- `inference_config_XXX.py`: Configuration files containing simulation parameters. Different parameter sets can be saved in different files.
- `run_inference.py`: Simulation scripts that select a configuration file, set up the simulation, and run an inference simulation.

After setting up the configuration file `inference_config_XXX.py`, the file can be selected by using the appropriate import command at the top of `run_inference.py`:

```
import inference_config_XXX as config
```

This loads all of the parameters as fields of the `config` variable. The inference simulation can then be run using the following command inside the `inference` directory:

```
python run_inference.py
```

Before each simulation with a given set of parameters, CrossSim will print to console a summary of the parameters. During the inference simulation, CrossSim will print the cumulative inference accuracy up to that point in the simulation, at a frequency controlled by the parameter `count_interval`. The final accuracy is printed to console when the simulation ends.

3.2 Simulation parameters

The parameters in `inference_config` are listed in Table 3.1. Much of the remainder of this manual will focus on explaining the meaning and use of these parameters.

Table 3.1: Inference simulation parameters

Parameter	Definition	Section
`Nruns`	# of times to run a simulation using identical parameters	–

useGPU	Whether to use GPU to accelerate simulations.	10.1
gpu_num	GPU ID to use for the simulation	10.1
task	Name of dataset to run inference on	4.1
model_name	Name or path of the Keras neural network model	5.2
ntest	# of test examples to run inference on	4.1
ntest_batch	# of test examples to load at a time	4.1
nstart	ID of the first test example to use; defines a subset of the test set	4.1
randomSampling	Whether to randomly sample a subset of the test set	4.1
count_interval	# of test examples between cumulative accuracy updates	3.1
time_interval	Whether to print elapsed time between accuracy updates	–
topk	The top k accuracies to report, e.g. `1` or `(1,5)`	–
show_model_summary	Whether to print a Keras summary of the model	5.2
weight_bits	Bit resolution of the weight values (`0` = do not quantize)	6.2
weight_percentile	Percentile of each layer's weight distribution that defines the quantization range.	6.2
Nslices	# of weight bit slices	6.6
NrowsMax	Maximum # of rows (inputs) to an MVM	8.1
style	Option for mapping negative numbers: `"BALANCED"` or `"OFFSET"`	6.5
balanced_style	Weight mapping option used if `style` is `"BALANCED"`: `"one_sided"` or `"two_sided"`	6.5
digital_offset	If `style` is `"OFFSET"`, whether offset is computed digitally (vs analog)	6.5
input_bitslicing	Whether input bits are applied sequentially via multiple MVMs	8.2
Rp	Unit cell array metal resistance, normalized to minimum cell resistance	8.3
noRowParasitics	Whether to set parasitic voltage drop along the rows to zero	8.3
interleaved_posneg	If `style` is `"BALANCED"`, whether to connect positive and negative cells to the same column	8.3
Icol_max	Maximum allowable magnitude of column current, in Amps (`0` = no limit)	8.4
Icell_max	Maximum possible current through a cell, in Amps	8.4
fold_batchnorm	Whether to fold batch normalization parameters into the convolution weight matrix, where possible	6.2
digital_bias	Whether to add bias digitally (vs analog)	6.2
bias_bits	Bit resolution of bias values, if `digital_bias` is `True` (`0` = do not quantize)	6.2
On_off_ratio	Ratio of maximum to minimum device conductance	7.1
error_model	Option for modeling cell programming errors	7.2
proportional_error	If using a generic error model `error_model` is `"alpha"`, whether error is proportional to conductance (vs independent)	7.2
alpha_error	If using a generic error model `error_model` is `"alpha"`, the amount of error α	7.2

`noise_model`	Option for modeling cell read noise	7.4
`proportional_noise`	Whether to model read noise as proportional to conductance (vs independent)	7.4
`alpha_noise`	The amount of read noise α_{noise}	7.4
`t_drift`	Amount of time elapsed between programming and inference (days)	7.3
`drift_model`	Model used to simulate conductance drift	7.3
`adc_bits`	ADC bit resolution	9.1
`dac_bits`	Input activation bit resolution	9.1
`ADC_per_ibit`	If `input_bitslicing` is `True`, whether to digitize the MVM result after each input bit	9.2
`adc_range_option`	Option for setting the ADC and activation ranges	9.3
`pct`	Percentile option used for selecting a saved set of ADC ranges, if `adc_range_option` is `"calibrated"`	9.4

3.3 Running parameter sweeps

The default `run_inference.py` script runs one or more simulations without sweeping any parameters. The user can run a sweep over one or more of the parameters in `inference_config.py` by manually modifying `run_inference.py` and saving the result as a new script. The parameter sweep overrides the value for the parameter set in `inference_config.py`.

This is done by defining a vector of values for the swept variable in the new script. Then, a `for` loop is created that iterates over this vector, inside or outside the existing `for` loop for identical runs, depending on the desired order. Inside the `for` loop, the appropriate field of the variable `config` is set to the current value of the swept parameter. An example of a parameter sweep can be found in `run_inference_errorloop.py`, which sweeps the amount of conductance programming error, assuming a generic error model. This example also saves the outputs of the parameter sweep to CSV files.

Chapter 4

Loading the dataset

4.1 Available datasets

The dataset used for the inference simulation is specified by the parameter `task`. By default, CrossSim supports the image classification datasets in Table 4.1.

Table 4.1: Provided datasets

Dataset	`task`	Max # test images	Input shape	# Classes
MNIST	`"mnist"`	10,000	(`ntest`, 28, 28, 1)	10
Fashion MNIST	`"fashion"`	10,000	(`ntest`, 28, 28, 1)	10
CIFAR-10	`"cifar10"`	10,000	(`ntest`, 32, 32, 3)	10
CIFAR-100	`"cifar100"`	10,000	(`ntest`, 32, 32, 3)	100
ImageNet	`"imagenet"`	50,000	(`ntest`, 224, 224, 3)	1000

The first four datasets are included with CrossSim inside the directory `cross_sim/data/backprop`. ImageNet is not included for copyright reasons and due its large size, but CrossSim can process the dataset for use in inference simulations if the path to the image files is provided by the user (see Section 4.3).

Some neural networks may require the loaded test images to be pre-processed before running inference. This will be discussed in Section 4.3 for ImageNet, and in Chapter 5 in general.

4.1.1 Selecting test images

Inference can be run on the full test set by setting `ntest` equal to the max # of test images above. The entire dataset is stored in memory during inference.

Inference can also be run on a fixed or random subset of the test set, either to obtain results faster or to accommodate memory limitations (if running a large dataset like ImageNet). The main use cases are described below:

- `ntest`$= N$, `ntest_batch`$= N$, `randomSampling=False`
  Inference is performed on a subset of N images from the test set, which are stored contiguously on disk. All N images are loaded at the same time to memory. See below for instructions on how to choose which N images to use for inference. Setting `ntest_batch` to `0` will default to this setting.

- `ntest`$= N$, `ntest_batch`$= M < N$, `randomSampling=False`
  Inference is performed on a subset of N images from the test set, which are stored contiguously on disk. This subset is further divided into several batches with at most M images each. At most M images are loaded at the same time to memory, and accuracy is separately reported for each batch.
- `ntest`$= N$, `randomSampling=True`
  The full dataset is loaded to memory. N images are randomly sampled from this set, and accuracy is reported for this random subset. `ntest_batch` is ignored.

The test images are stored in a fixed order on disk. In the first two cases above, where inference is run on a subset of the dataset, the parameter `nstart` selects the index of the first image in the subset. For example, one can distribute an inference simulation on the first 10,000 ImageNet images across two GPUs running two instances of CrossSim. In this case, we set `ntest=5000` for both, and `nstart=0` for the first GPU and `nstart=5000` for the second GPU. To perform inference on the next 10,000 images, we set `nstart=10000` for the first GPU and `nstart=15000` for the second GPU.

4.2 Adding a new dataset

A custom dataset, in addition to the ones in Table 4.1, can be added by following the below steps:

1. Place the dataset files in the `/cross_sim/data/datasets/` folder.
2. Add a function to the file `/helpers/dataset_loaders.py` that can be used to load the dataset (partially or in full) into NumPy arrays: `x_test` which contains the inputs, and `y_test` which contains the output labels. This function should support dataset truncation (starting and ending at arbitrary indices). Use the existing functions in this file as a guide.
3. In the function `inference()` in `/inference/inference_utils/inference_net.py`, add a clause that calls the newly defined dataset loader, and ensure that the inputs `x_test` are properly scaled.
4. For any neural network model that is to be used with the new dataset, follow the steps in Section 5.3. Make sure the neural network model includes an input layer (of type `InputLayer`) that matches the dimensions of the input images.

Currently, CrossSim Inference only supports classification datasets with N discrete labels, and the final layer of the network has N outputs, the largest of which is selected as the predicted label. Future releases may support regression tasks.

4.3 ImageNet

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC2012) validation set is not provided with CrossSim due to copyright and the large file size. Instead, to run an ImageNet inference simulation, the user must supply the raw test set JPEG images, then run one of the provided pre-processing scripts to generate images that will be used by CrossSim. To reduce memory and computational overhead, this pre-processing is not performed as part of the inference simulation.

4.3.1 ImageNet pre-processing

The ImageNet pre-processing scripts can be found in `/helpers/imagenet_preprocess`. To use the preprocessing scripts, the user must first supply the path to the validation images in `imagenet_path.py`. The raw ILSVRC2012 images have variable dimensions with side typically being 500 pixels. The files should be named `"ILSVRC2012_val_XXXXXXXX.JPEG"` where the image number varies from 1 to 50,000. The user can also select a subset of the validation set to pre-process, by supplying the number of images to pre-process (`num_images`) and the index of the starting image (`n_start`). The pre-processing step loads the images as BGR, resizes and crops the image typically to 224×224 pixels, then if needed by the neural network, zero-centers or scales the result. The pre-processed images are stored in a NumPy array which has dimensions (`num_images`, 224, 224, 3), which is then saved to a `.npy` file.

At the beginning of an inference simulation, CrossSim selects the appropriate `.npy` file to load based on the chosen image subset and the neural network in `inference_config.py`. The logic for selecting the images is in the file `/helpers/dataset_loaders.py` in the function `load_imagenet()`. The user should fill out the path of the pre-processed `.npy` image files at the top of the function. By default, `load_imagenet()` supports the following pre-processed partitions of the validation set. These are specified inside `/helpers/imagenet_preprocess/preprocess_XXX.py`:

- `num_images=50, n_start=0`
- `num_images=1000, n_start=0`
- `num_images=5000, n_start=0`
- `num_images=25000, n_start=0`
- `num_images=25000, n_start=25000`

If desired, the user can modify the logic in `load_imagenet()` to accept `.npy` files containing different settings than the ones above. Even if the entire dataset fits inside memory, we recommend loading no more than 5000 images per simulation for ImageNet. Leaving more memory available for the CrossSim cores improves the performance of the code (see Section 10). The partitions above containing 25,000 images are loaded 5000 images at a time.

The provided pre-processing scripts are summarized in Table 4.2. Different pre-processing scripts are used to support the different pre-trained models provided with CrossSim, which originally came from several different sources: the `keras.applications` library, MLPerf [17], and Nvidia [1]. A small, fixed set of 50 pre-processed images using each of the five scripts can be found in `/cross_sim/data/backprop/imagenet`. Inference simulations can be run on this small set without having to supply the raw validation images.

4.3.2 MLPerf calibration Set

MLPerf provides a subset of 500 images from the ImageNet validation set to use for calibrating the hardware. CrossSim mainly uses these images to calibrate the ranges of the activations and ADCs for each layer, similar to the process used in Jacob *et al.* [12] for quantizing ImageNet to 8 bits during inference operations. With the raw images available, these images can be selected for pre-processing by using `MLperf_calibration=True` in the pre-processing scripts. In an inference simulation, these images are used in the dedicated ADC range calibration profiling script, `run_inference_profiling.py`. The process for calibrating the ADC ranges will be described in Chapter 9.

Table 4.2: ImageNet pre-processing scripts

Pre-processing script	Output image size	Required packages	Compatible pre-trained models

`preprocess_keras.py`	(224,224,3)	OpenCV 4.5.4	`"ResNet50"`
			`"VGG19"`
			`"MobileNetV1"`
			`"MobileNetV2"`

`preprocess_keras_inception.py`	(299,299,3)	OpenCV 4.5.4	`"InceptionV3"`
`preprocess_torchvision.py`	(224,224,3)	Torchvision 0.11.1	`"ResNet50-int4"`
`preprocess_MLPerfRN50.py`	(224,224,3)	OpenCV 4.5.4	`"ResNet50-v1.5"`
`preprocess_MLPerfMobileNet.py`	(224,224,3)	OpenCV 4.5.4	`"MobileNetV1-int8"`

4.3.3 Labels

In addition to the pre-processed test images, a label file `y_val.npy` is needed that contains the classes corresponding to these images, in the same order. The labels for both the full ImageNet validation set and the MLPerf calibration set can be generated using `generate_labels.py`. The labels file `y_val.npy` used for our tests can be found in `cross_sim/data/backprop/imagenet`.

Chapter 5

Loading the neural network model

The neural network model to use for an inference simulation is specified by a string (`model_name`) in the inference configuration file. This string is used to select and load a known model stored as a keras `.h5` file, match the model to its pre-processed images (for ImageNet), and apply other model-specific simulation parameters.

5.1 Supported neural networks

5.1.1 Supported Keras layer types

CrossSim can run inference on a variety of neural networks supplied as Keras models, and supports the most common layer types used in modern convolutional neural networks. These include the following standard layer types in `keras.layers`:

- Convolution: `Conv2D` and `DepthwiseConv2D` (depth-wise convolutions)
- Fully-connected: `Dense`
- Pooling: `MaxPooling2D`, `AveragePooling2D`, `GlobalMaxPooling2D`, `GlobalAveragePooling2D`
- Element-wise addition: `Add`
- `Concatenate`: along channel dimension only
- `BatchNormalization`
- `ZeroPadding2D`
- Activations: `Activation` (ReLU, sigmoid, and softmax), `ReLU`, and activations inside `Conv2D`, `DepthwiseConv2D`, and `Dense`.
- `Flatten`
- Ignored layers: `Dropout`, `GaussianNoise`, which are enabled only during training time.

Non-sequential models with skip connections (such as ResNet and Inception nets) are supported. For convolution and pooling layers, CrossSim follows the same conventions as Keras to set the amount of input padding based on padding style (”same” or ”valid”), kernel size, and stride. The pooling, element-wise addition, concatenation, and activation operations are assumed to be carried out in digital hardware and are therefore modeled without errors.

If a model contains a Keras layer type other than one of the above *that can be ignored during inference*, the name of the Keras layer type can be added to the list `ignoredLayerTypes` in `keras_parser.py`. Currently, these include `Dropout` and `GaussianNoise`.

5.1.2 Quantized neural networks: Whetstone and Larq

CrossSim also provides some support for models with quantized weights and/or activations trained using Sandia's Whetstone framework [18] as well as the Larq library [9]. The additional supported layers are:

- `Spiking_BRelu` activation type, used by Whetstone models to "sharpen" a linear ReLU activation to a binary one during training. A maximum sharpness of 1 (typical during inference) is a binary activation function with outputs of (0,1) with a transition at an input value of 0.5.
- `QuantConv2D` and `QuantDense`: quantized layers from the Larq library. CrossSim supports weight and/or input binarization using the sign function: outputs of (-1, 1) with a transition at an input value of 0.

Several custom layers are also included to explicitly support the `"ResNet50-int4"` model trained by Nvidia, which uses 4-bit weights and 4-bit activation functions with parameterized scaling steps.

5.2 Available pre-trained models

Table 5.1 shows the list of provided models that can be selected from the inference configuration files using the parameter `model_name`. The Keras H5 files for these models are stored in `/pretrained_models/`, sorted by directory. Some of the models require custom loading methods, model-specific pre-processing methods, and/or model-specific simulation settings, the details of which can be found inside the file `/inference/inference_util/CNN_setup.py`. The detailed topology of the model can be viewed by setting `show_model_summary` to `True`.

All of the non-ImageNet models were trained in-house, while all of the ImageNet models are pre-trained reference models from open-source repositories. The Keras models are all provided as part of the CrossSim models repository, except for the large VGG-19 model. For VGG-19, the weights are downloaded from keras.applications at the beginning of the simulation (the user can store the downloaded weights locally, to speed up subsequent simulations).

5.3 Adding custom models

In addition to the models provided, users can perform inference simulations on their own trained Keras H5 models, provided that it does not contain special layer types not listed in Section 5.1.1. To add a new model, the file `/inference/inference_util/CNN_setup.py` needs to be modified, as described below.

5.3.1 Model path and loading

The first step in adding a new model is to include it as an entry to the function `build_keras_model()`, which takes a `model_name` keyword as input and returns a Keras model object. For many CNN models, this is straightforward: simply add a new `if` clause for the model name and pass the path to the H5 model file into the function `load_keras_model()`. For more complex models, additional steps may be needed (see file for examples).

After the model has been added to `build_keras_model()`, the model name can then be specified in `inference_config.py` together with the matching dataset name.

5.3.2 Model-specific parameters

In order to properly load the dataset and hardware configuration, some additional parameters may need to be set for the added model, inside the function `model_specific_parameters()`. These are listed below:

- `imagenet_preprocess`: For an ImageNet model, this is a string that specifies how the dataset should be pre-processed for this model. This value is used as a keyword in the function `load_imagenet()` in `/helpers/dataset_loaders.py`, to select the appropriate pre-processed dataset in Table 4.2 and (in some cases) to select the appropriate secondary pre-processing function in `keras.applications`. For a new model, `load_imagenet()` may need to be modified to include the correct pre-processing method. For non-ImageNet models, set the value to `None`.
- `subtract_pixel_mean`: For CIFAR-10 and CIFAR-100, this Boolean specifies whether the test set input values should be zero-centered using the mean values for each channel over the training set. This zero-centering is used, for example, in the CIFAR-10 ResNets [11].
- `dataset_normalization`: A string specifying how the loaded dataset should be normalized (`"none"` to do nothing). Available options include `"unsigned_8b"` and `"signed_8b"` which assume uint8 inputs in the range (0,255) and scale them to the range (0,1) and (-1,1), respectively. These options are sufficient for the provided models, but other options can be added as new strings in this function and implemented in `inference_net.py`.
- `positiveInputsOnly`: A list of Boolean values, one for each convolution or fully-connected layer in the network. If `True` for that layer, the inputs are strictly positive or zero; this is the case when the layer directly follows a ReLU activation. This condition removes the need for a sign bit in the input.
  - Note: To determine which layer corresponds to the i^{th} element `positiveInputsOnly[i]`, check the layer's name using `layerParams[i]['name']` in `run_inference.py`.
- `memory_window`: The maximum number of consecutive layers whose activations are stored at one time in memory. Reducing this number can free up some memory in non-sequential CNNs and give a slight improvement in simulation speed. A large value (such as 10) is always safe, but the smallest value that does not throw an error during inference is slightly faster.
- `larq`: Set this to `True` if the model is a Larq quantized model.
- `whetstone`: Set this to `True` if the model is a Whetstone quantized model.

5.3.3 ADC and activation ranges (optional)

If the ADC or activation quantization is enabled (`ADC_bits` or `DAC_bits` not equal to zero), then the ranges of each layer's ADC and activation values need to be specified. To maximize the accuracy at a given resolution, these ranges need to be calibrated, and the calibrated ranges are loaded inside the function `load_adc_activation_ranges()`. The process used for determining and loading these ranges is described in Section 9.4. If ADC or activation quantization is not being modeled, no change is necessary to this function.

Alternatively, ADC/activation quantization can be enabled without calibration using simple design rules for the ranges as described in Section 9.3, though this is likely to yield sub-optimal accuracy.

5.3.4 Performance tuning (optional)

A new `if` clause can optionally be added to the function `get_xy_parallel()` in order to tune CrossSim's performance on the model. The parameters in this function determine how many sliding window MVMs in a convolution can be packed into a single call to the MVM kernel in NumPy or CuPy. In general, optimizing these parameters can improve performance by several fold, especially when running simulations on GPUs with much larger or much smaller amounts of GPU memory. This performance optimization is described in more detail in Section 10.2.

Table 5.1: Provided neural network models (FP = floating point)

Model name	Task	FP accuracy	Notes
`CNN6`	MNIST	98.6%	Four convolutional + two dense layers, 61.6K weights.
`CNN6_v2`	MNIST	98.8%	Same structure as `CNN6` without final pooling layer and uses bounded ReLU (0,1), 119.3K weights.
`cifar10_cnn_brelu`	CIFAR-10	84.5%	Four convolutional + two dense layers with bounded ReLU (0,1), 4.36M weights.
`whetstone_cifar10`	CIFAR-10	82.9%	Whetstone CNN, binary activations, 4.24M weights.
`larq_cifar10`	CIFAR-10	82.6%	Larq CNN, binary weights and binary activations, 10.36M weights.
`ResNet14`	CIFAR-10	90.7%	Follows topology in Ref. [11], 176.6K weights.
`ResNet20`	CIFAR-10	91.3%	Follows topology in Ref. [11], 274.4K weights.
`ResNet32`	CIFAR-10	91.9%	Follows topology in Ref. [11], 470.2K weights.
`ResNet56`	CIFAR-10	92.3%	Follows topology in Ref. [11], 861.8K weights.
`ResNet56_cifar100`	CIFAR-100	73.9%	Follows topology in Ref. [11] with 4× more channels (16× weights) as `ResNet56`.
`Resnet50`	ImageNet	74.9%	From `keras.applications`.
`Resnet50-v1.5`	ImageNet	76.5%	From MLPerf Inference Benchmark [17].
`Resnet50-int4`	ImageNet	76.2%	Trained by Nvidia for the MLPerf Inference Benchmark [1].
`VGG19`	ImageNet	71.3%	From `keras.applications`. Weights will be downloaded.
`InceptionV3`	ImageNet	77.9%	From `keras.applications`.
`MobilenetV1`	ImageNet	70.4%	From `keras.applications`.
`MobilenetV2`	ImageNet	71.3%	From `keras.applications`.
`MobilenetV1-int8`	ImageNet	70.6%	From MLPerf Inference Benchmark [17].

Chapter 6

Weight mapping parameters

This section describes the available options for mapping neural network weight values to memory device conductances G. These parameters can be modified in the inference configuration file, `inference_config.py`. In the following discussion, G_{min} is the minimum conductance of a cell and G_{max} is the maximum conductance; see Section 7.1 on setting their ratio.

6.1 Convolution mapping

CrossSim maps convolutions to MVMs in resistive memory arrays using the scheme proposed by Shafiee *et al.* [19], which is depicted in Fig. 6.1. The full convolution is executed as a sequence of sliding windows, each of which maps to a single array MVM. A single sliding window input (containing all N_{ic} input channels) is unrolled into a vector which drives the rows of the array. The output on the array columns contains the sliding window result in each of the N_{oc} output channels. The weight matrix is fully mapped to a resistive memory array of dimensions $K_{\text{x}}K_{\text{y}}N_{\text{ic}} \times N_{\text{oc}}$. The number of MVMs required is equal to the number of 2D sliding windows, which is also equal to the number of elements along the x and y dimensions of the output feature map.

A fully-connected layer with N_{in} inputs and N_{out} outputs is directly mapped to a matrix with dimensions $N_{\text{inputs}} \times N_{\text{out}}$ and is executed using one MVM per input.

If the weight matrix is large, it may be partitioned spatially over multiple arrays. This is covered in Section 8.1.

6.2 Weight resolution

6.2.1 Weight quantization and range

Weight values in the convolution and fully-connected layers can be quantized before being mapped to device conductances. In order to quantize the weights, it is necessary to specify: the bit resolution N_{W} of the quantized weights, and the numerical range over which the quantization will occur. The former is specified using the parameter `weight_bits`. Setting this value to `0` disables quantization, so floating-point weights (or already quantized weights) are mapped directly to conductances. Other errors can subsequently be applied to these conductances.

CrossSim provides a simple option to set the quantization range, using the `weight_percentile` parameter; denote the value of this parameter as P. Its use is described as follows where:

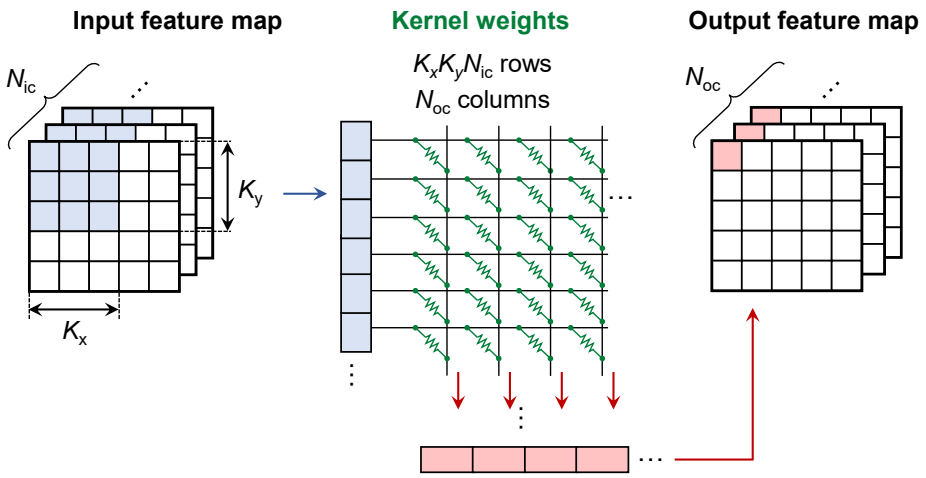

Figure 6.1: Scheme used to map a convolution to a resistive memory array. Each sliding window corresponds to one array MVM.

- $P = 100$: For a given layer's weight matrix $\mathbf{W}$, the maximum absolute weight is found: $W_{\text{absmax}} = \max(|\mathbf{W}|)$. The quantization range is then set to $[-W_{\text{absmax}}, +W_{\text{absmax}}]$. This is the recommended default option.
  - This option ensures that no weight value in $\mathbf{W}$ is clipped. If `weight_bits` is sufficiently high (e.g. $\geq$8 bits for ResNet50, $\geq$11 bits for MobileNetV2), this option leads to only a very small accuracy degradation.
- $P < 100$: The P^{th} percentile value and the $(100-P)^{\text{th}}$ percentile value are found in $\mathbf{W}$. Of these two values, the one with a larger absolute value is selected; call the absolute value of this quantity W_{P}. The quantization range is set to $[-W_{\text{P}}, +W_{\text{P}}]$.
  - Currently, only a single value of P is supported for the full network.
  - This option causes the largest and smallest weight values to be clipped. It assumes that weights are signed. This option is intended for cases where `weight_bits` is small, in which case better accuracy may be possible by using a narrower quantization range, which leads to a smaller separation between weight levels. Modifying P trades off clipping and quantization errors.
- $P > 100$: The quantization range is set to $[-\frac{P}{100} \times W_{\text{absmax}}, +\frac{P}{100} \times W_{\text{absmax}}]$. Note that some part of the conductance range will not be utilized by any of the weights (before applying conductance errors or drift).

In all cases, the quantization range is symmetric about zero. This is a physical necessity if using differential cells. When using offset subtraction, this implies that the offset corresponds to a value of zero (see Section 6.5). Importantly, the above process is done separately for every layer, i.e. the conversion factor between weight and conductance varies from layer to layer depending on the value of W_{absmax}.

Generally speaking, weights are quantized by finding $2^{N_{\text{W}}}$ equally spaced levels in the quantization range, then rounding each weight to the nearest level. The specific implementation of weight value quantization depends on the negative number representation scheme: see Section 6.5. For example, 8-bit quantization with differential cells uses 255 distinct weight levels where the center level is the value zero. With offset subtraction, 256 weight levels are used over the quantization

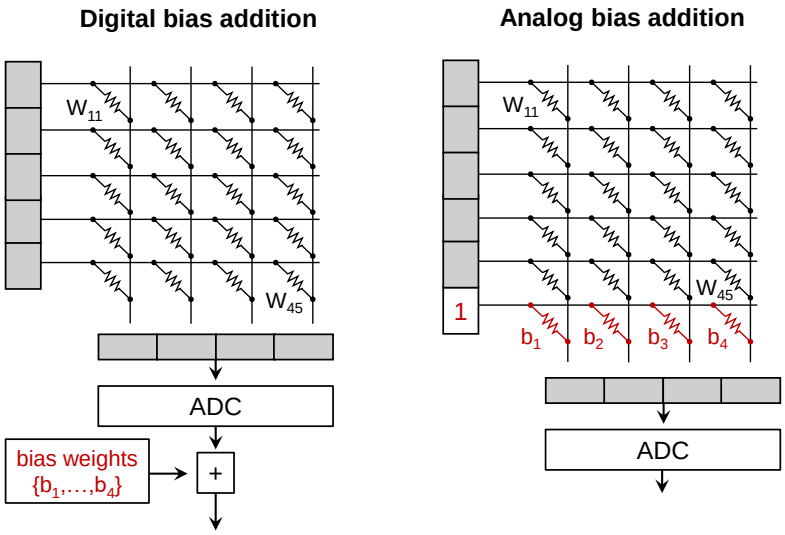

Figure 6.2: Addition of bias weights in the digital and in the analog domain.

range. The difference is small if `weight_bits` is large.

6.3 Bias weights

A convolution or fully-connected layer often includes the addition of a bias vector b to the matrix-vector or matrix-matrix product. As with batch normalization, bias addition can be handled in analog or in digital. This is controlled by the parameter `digital_bias`: `True` for digital, `False` for analog. The two cases are shown in Fig. 6.2.

In the analog case, bias weights are implemented as an additional row in the memory array, which is always driven by an input corresponding to a value of 1; physically, the addition occurs by Kirchoff's circuit law. The bias weights are considered part of the weight matrix $\mathbf{W}$ along with the other weights.

In the digital implementation, the bias weights are not considered part of $\mathbf{W}$ and are stored as digital values and added to the digitized dot products using a digital arithmetic unit; this is assumed to be error-free, though the bias weights can be quantized from floating-point precision.

We have found that `digital_bias=True` often yields higher accuracy, especially if `fold_batchnorm` is set to `True` (see below). This is because the bias weights and the other weights can lie in drastically different numerical ranges. If the difference in the numerical ranges is too large, it may no longer be possible to represent both the weights and bias with adequate precision once weight quantization and device errors are taken into account. However, an analog bias is more energy-efficient since adding the bias digitally incurs a small overhead. Whether this choice affects the accuracy is application dependent and should be evaluated by comparing the results of inference simulations.

If `digital_bias=True`, the number of bits used to quantize the bias weights is controlled by the numerical parameter `bias_bits`. Bias quantization follows the same procedure as weight quantization above, but always assumes $P = 100$, regardless of the value of `weight_percentile`. A value of `0` for `bias_bits` disables bias quantization.

6.4 Batch normalization parameters

During inference time, a batch normalization layer can be "folded" into a preceding convolution layer, since both layers apply linear transformations to the input and the two layers do not need to be kept separate during inference. The folding operation is given by [12]:

$$\mathbf{W}_{\text{folded}} = \gamma \frac{\mathbf{W}}{\sqrt{\sigma^2 + \varepsilon}} \tag{6.1}$$

$$b_{\text{folded}} = \gamma \frac{b - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta \tag{6.2}$$

where the left-hand terms represent the weight matrix and bias vector after folding, $(\mu, \sigma, \gamma, \beta)$ are the batch normalization parameters (unique to each output channel), and ε is a small constant to avoid division by zero.

Folding greatly reduces the complexity of the analog inference accelerator, as no dedicated processing component is needed to perform the batch normalization computation, which involves numerous element-wise multiplications and divisions on the MVM output. CrossSim supports either option, through the parameter `fold_batchnorm` in the configuration file. If `True`, batch normalizations are folded and computed in analog inside the memory array together with the convolution. This folding is applied before weight quantization and all other weight mapping steps. If `False`, the batch normalization is computed as a separate digital step without error.

Batch normalization folding can have an effect on accuracy, because the folding process changes the value distribution of each layer's weights, and the severity of analog errors can depend on these values. Among the ImageNet networks, we have found that the difference is most noticeable with MobileNet, which has fewer total parameters and is more sensitive to weight precision.

6.5 Negative number representation

One of the basic design choices in an analog in-memory inference accelerator is how to map both positive and negative weights onto a memory array, whose cell conductances are strictly positive. There are two main methods to do this: differential (or balanced) cells, and offset subtraction. These are shown in Fig. 6.3. One of these options can be selected using the `style` parameter, using the values `"BALANCED"` or `"OFFSET"`, respectively.

6.5.1 Differential cells

The differential cells scheme uses the difference in conductance of two cells to represent a matrix of signed weights $\mathbf{W}$:

$$\mathbf{W} = \mathbf{W}^+ - \mathbf{W}^- \tag{6.3}$$

where $\mathbf{W}^+$ and $\mathbf{W}^-$ are strictly positive or zero, and thus can be mapped separately to conductances. This definition leaves some ambiguity about how two conductance values are decided from a single weight value. This ambiguity is broken using the parameter `balanced_style`, used only if `style = "BALANCED"`:

- `balanced_style = "one_sided"`: One conductance in the pair encodes the magnitude if the weight is positive, while the other encodes the magnitude if the weight is negative. In every

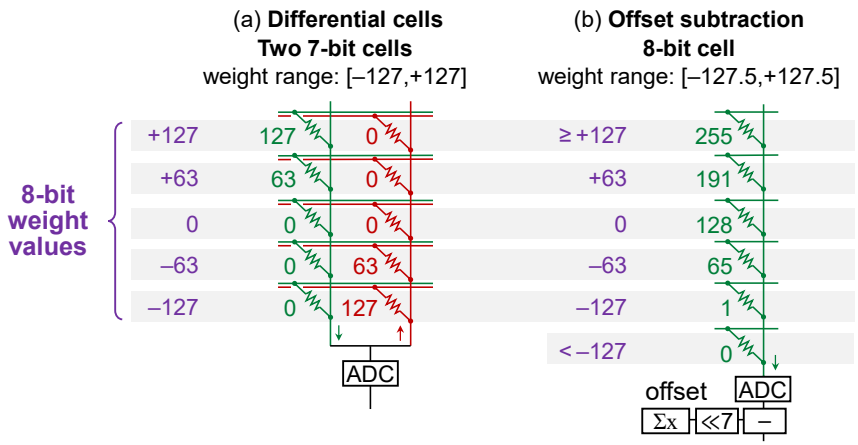

Figure 6.3: Two schemes for representing negative numbers, illustrated for the case where `weight_bits=8` with no bit slicing. For illustration, the weights are rescaled to the ranges shown. (a) Differential cells using a one-sided scheme, (b) Offset subtraction. The conductance is labeled in terms of the state number in ascending order, where 0 corresponds to G_{min} and 127 (255) corresponds to G_{max} for a 7-bit (8-bit) cell.

pair, the conductance that does not encode the magnitude is set to the lowest conductance state G_{min} (encoding zero). A weight value of zero maps to a pair of cells at G_{min}. This is shown in Fig. 6.3(a). **This is the recommended option for inference**, as it maps weight absolute values proportionally to conductance. This has many benefits for error resilience, summarized by Xiao *et al* [22].

- `balanced_style = "two_sided"`: A weight value of zero maps to a pair of cells at the midpoint conductance between G_{min} and G_{max}. For more positive weights, one conductance is increased toward G_{max} while the other is decreased toward G_{min}. For negative weights, this is reversed.

Fig. 6.3(a) shows the one-sided differential cells representation scheme for a an 8-bit weight matrix. Note that since two devices are used per weight, the resolution requirement on each device is one bit less than `weight_bits`. When not using bit slicing, the conductance level G_{max} is used to represent the weight(s) in each layer with the maximum absolute value. The conductance level G_{min} is used for at least one device in every differential pair. A weight value of zero uses G_{min} in both devices. Note that due to a redundancy for the value of zero (either device can be used as the magnitude), the scheme maps 255 unique weight values rather than the full 256.

There are multiple ways to implement the subtraction that is part of the differential scheme. The currents can accumulate separately in the positive and negative columns, then the two column currents can be subtracted using an analog circuit. The currents can also be subtracted via Kirchoff's circuit law if the two devices in the pair conduct current in opposite directions: this is the case shown in Fig. 6.3(a). It may also be possible to subtract the currents within each differential pair before accumulating on a column. CrossSim remains agnostic to how the actual subtraction is performed, but it does assume that the subtraction is performed in the analog domain before the ADC. The ADC then digitizes the difference.

6.5.2 Offset subtraction

Offset subtraction implements a signed weight matrix $\mathbf{W}$ by using an added offset to convert negative weights to positive conductances:

$$\mathbf{W} = \mathbf{W}_{\text{prog}} - W_{\text{offset}} \tag{6.4}$$

where $\mathbf{W}_{\text{prog}}$ is strictly positive or zero, and W_{offset} is a scalar offset that is subtracted element-wise. The offset is chosen such that a zero weight in $\mathbf{W}$ is mapped to a value of W_{offset} in $\mathbf{W}_{\text{prog}}$. This is shown in Fig. 6.3(b). In this case, all 256 conductance levels can be utilized, though in order to keep the symmetry about zero, the highest and lowest levels map a smaller set of weight values.

When computing an MVM between $\mathbf{W}$ and a vector $\vec{x}$, the above becomes:

$$\mathbf{W}\vec{x} = \mathbf{W}_{\text{prog}}\vec{x} - W_{\text{offset}}\,\mathbf{I}\vec{x} \tag{6.5}$$

where $\mathbf{I}$ is the identity matrix. In an analog accelerator, there are two ways to compute the subtraction of the offset (second term):

- Compute the second term using digital hardware, then subtract this offset from each of the digitized dot product values in $\mathbf{W}_{\text{prog}}\vec{x}$, which are computed in analog. This shown in Fig. 6.3(b).
- Compute the second term in analog using an additional column of the array called the "unit column." This was first proposed by Shafiee *et al* [19]. This result is digitized, then subtracted from each of the digitized dot product values.

The first option is selected by setting the parameter `digital_offset=True`, while the second is chosen by setting `digital_offset=False`. This parameter is used only if `style="OFFSET"`. In general, we find that the former yields better error resilience. This is because if the conductances in the unit column have programming errors, then this same error is transferred to all of the dot products within the layer during an MVM [22].

When not using bit slicing, the conductance level G_{max} is used to represent the maximum weight in the quantization range, while the level G_{min} is used to represent the minimum. Since the quantization range is forced to be symmetric about zero while the actual weight distribution may not be, some conductance levels close to these endpoints may be left unused. A weight value of zero maps approximately to the midpoint of the conductance range.

6.6 Weight bit slicing

Bit slicing is a technique that is used to represent weight values with more bits of precision than available in a single device. Bit slicing divides the weight bits into multiple slices, and the results of bit sliced MVMs are combined via shift-and-add reduction. For example, a matrix of 6-bit integers can be divided into two slices of three bits each:

$$\begin{bmatrix} 12 & 58 \\ 29 & 50 \end{bmatrix} = 2^3 \begin{bmatrix} 1 & 7 \\ 3 & 6 \end{bmatrix} + 2^0 \begin{bmatrix} 4 & 2 \\ 5 & 2 \end{bmatrix} \tag{6.6}$$

In CrossSim, the parameter `Nslices` sets the number of bit slices. A value of `1` disables bit slicing. CrossSim then decides the precision of each bit slice based on the weight resolution, `weight_bits`. CrossSim supports any combination of the two parameters, though some pairs will lead to underutilization of the device conductance range, as described below.

The specific way that bit slicing is used to map weights depends on the negative number representation scheme. These are described below.

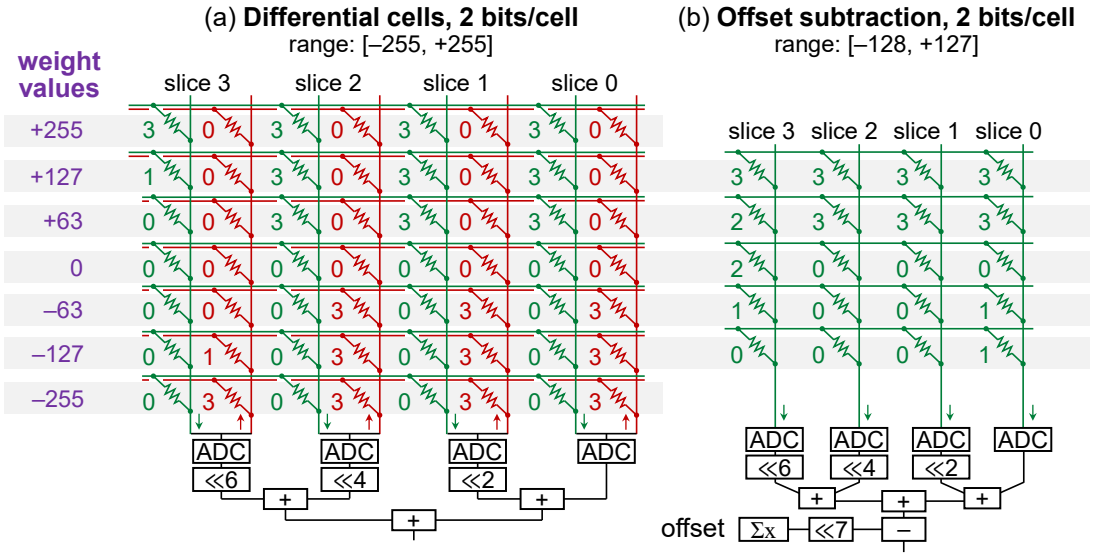

Figure 6.4: Bit slicing configurations with `Nslices=4` that maximally utilize the device conductance range. (a) Differential cells with `weight_bits=9`, (b) Offset subtraction with `weight_bits=8`.

6.6.1 Bit-sliced differential cells

Consider the case where an 8-bit weight matrix $\mathbf{W}$ is split into four 2-bit slices, i.e. `weight_bits=8`, `Nslices=4`. The decomposition of an MVM is done based on the following equation:

$$\mathbf{W}\vec{x} = 2^6\left(\mathbf{W}_3^+\vec{x} - \mathbf{W}_3^-\vec{x}\right) + 2^4\left(\mathbf{W}_2^+\vec{x} - \mathbf{W}_2^-\vec{x}\right) + 2^2\left(\mathbf{W}_1^+\vec{x} - \mathbf{W}_1^-\vec{x}\right) + \left(\mathbf{W}_0^+\vec{x} - \mathbf{W}_0^-\vec{x}\right) \tag{6.7}$$

Suppose that as in Fig. 6.3(a), $\mathbf{W}$ has integer values in the range $[-127, +127]$ and that we use a sign-magnitude bit representation. The seven magnitude bits of the weight values are split across four 2-bit slices. Within each 2-bit slice, the differential cells representation in Equation (6.3) is used. The sign bit is used to decide which of the cells encodes the magnitude in all slices. In this example, each 2-bit matrix $\mathbf{W}_i^{\pm}$ represents one sign of one slice, and is integer-valued in the range [0,3]. As in the differential cells case, an important property of this mapping is that a weight value of zero is mapped entirely onto the G_{min} state in all slices as shown in Fig. 6.4(a). Note that this corresponds to the "one-sided" mapping of differential cells described earlier (with bit slicing, we do not provide a two-sided mapping).

Note that in the above example, there is under-utilization of the device conductance range because seven magnitude bits are sliced across four 2-bit slices, which can represent eight magnitude bits. The top half of the conductance range in the top slice is not utilized. Using the same hardware, it is possible to fully represent 9-bit signed weights in the range $[-255, +255]$. This is the case shown in Fig. 6.4(a). **In general, the device conductance range is fully utilized if (`weight_bits-1`) is divisible by `Nslices`.** If this condition is not met, it may be possible to obtain a higher accuracy with the same hardware by changing the weight resolution so that it is met.

The four quantities inside parentheses in Equation (6.7) are the dot products corresponding to each bit slice. ADC quantization is applied separately to each slice, with the option to use different ADC ranges for each slice; this will be described in Chapter 9. As in the unsliced case, the subtraction within each slice is assumed to occur in analog before the ADC. The shift-and-add

operation over the slices in Equation (6.7) is assumed to be error-free, as it is carried out on the digitized slice-wise dot products.

6.6.2 Bit-sliced offset subtraction

Consider again the case with `weight_bits=8` and `Nslices=4`. Using offset subtraction, the MVM is decomposed as:

$$\mathbf{W}\vec{x} = 2^6\mathbf{W}_3\vec{x} + 2^4\mathbf{W}_2\vec{x} + 2^2\mathbf{W}_1\vec{x} + \mathbf{W}_0\vec{x} - 2^7\mathbf{I}\vec{x} \tag{6.8}$$

This time, $\mathbf{W}$ is an 8-bit matrix with integer values in the range $[-128, +127]$ and $\mathbf{W}_i$ are the 2-bit slices of $\mathbf{W}$ from lowest to highest significance. Each element of $\mathbf{W}_i$ is integer-valued in the range [0, 3] and mapped to the conductance of a single cell. After the results of the slices are aggregated, an offset term is subtracted to represent negative weights. Here, the offset is chosen to be 2^7 to map the range $[-128, +127]$. This case is shown in Fig. 6.4(b).

Like the offset subtraction case in Equation (6.5), the conductances are not proportional to the weight magnitudes. Specifically, a zero weight is mapped to an intermediate conductance state in the top slice.

Slightly different from the bit-sliced differential cells case above, **the device conductance range is fully utilized if** `weight_bits` **is divisible by** `Nslices`. If this condition is not met, it may be possible to obtain a higher accuracy with the same hardware by changing the weight resolution so that it is met.

Chapter 7

Device parameters

This section describes the options for modeling the effects of memory device properties on inference accuracy. These device-level effects are applied at several stages, as shown in Fig. 7.1: during the mapping of weights to conductances, after mapping to conductances, and during the inference simulation.

7.1 Conductance On/Off ratio

The ratio of the maximum programmable conductance G_{max} and the minimum programmable conductance G_{min} can be specified using the parameter `On_off_ratio`. Only the dimensionless ratio $G_{\text{max}}/G_{\text{min}}$ needs to be specified, not the individual conductances. A value of `0` species an infinite On/Off ratio, i.e. $G_{\text{min}} = 0$.

The On/Off ratio sets a lower limit on how much current can be drawn by any given device in the array. The On/Off ratio is critical in cases where weights with low absolute value are mapped to low conductances (i.e. one-sided differential cells). In this case, the On/Off ratio determines how much the accuracy degrades as a result of parasitic array *IR* drops, or programming errors/read noise that is proportional to the conductance [22].

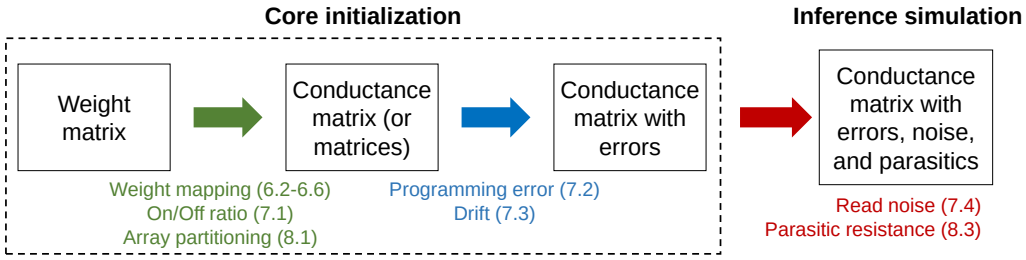

Figure 7.1: Sequential application of device-level effects when representing a neural network weight matrix.

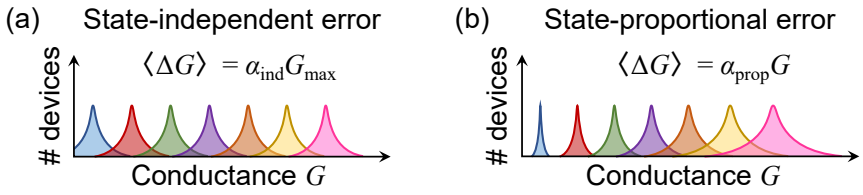

Figure 7.2: Two generic models of conductance programming error. (a) State-independent error, and (b) State-proportional error.

7.2 Conductance programming errors

Random conductance programming errors are sampled and applied once per simulation, during the weight mapping step just prior to inference (during core initialization). These random errors do not change between MVMs or input images. The errors do change between different inference runs; for example, the errors will be different for each call to `run_inference.py` and for each run of a multi-run simulation with `Nruns` runs.

Two methods are available in CrossSim to apply programming errors. The *generic* conductance error model uses a simple model of the error characteristics for a memory device. This option can be useful if an algorithm's resilience to conductance errors in general is being evaluated, and can be used to perform sweeps of the error magnitude. It may also be useful for determining what kind of error characteristics are desirable in a device. The second option allows the user to enter a custom programming error model, informed by experimental data or data from low-level physics simulations. This option is used to determine the accuracy of an algorithm when implemented using a particular memory device technology.

7.2.1 Generic conductance error model

CrossSim includes two types of generic conductance programming error, shown in Fig. 7.2. The generic error option is selected by setting `error_model="alpha"`. From there, state-independent error is selected by setting `proportional_error=False`, and state-proportional error is selected by setting `proportional_error=True`. The amount of each type of error is then set by the parameter `alpha_error` as described below. Any conductance that is perturbed beyond the range $(G_{\text{min}}, G_{\text{max}})$ will be clipped to the endpoints of this range.

In the **state-independent error** model, the amount of conductance error is independent of the target conductance G: see Fig. 7.2(a). The expected amount of programming error $\langle \Delta G \rangle$ is a fixed fraction α_{ind} of the maximum conductance G_{max}. For each conductance, an error ΔG is sampled from a normal distribution with zero mean and a standard deviation of $\alpha_{\text{ind}} G_{\text{max}}$. The value of α_{ind} is set by the parameter `alpha_error` in `inference_config.py`. Note that the definition of α_{ind} differs slightly from the definition used by Xiao *et al.* [22].

In the **state-proportional error** model, the amount of conductance error is proportional to the target conductance G: see Fig. 7.2(b). As in the state-independent case, the programming error for every conductance is sampled from a normal distribution, but the standard deviation of the distribution now depends on the conductance as $\alpha_{\text{prop}} G$. The error increases with conductance, and a conductance of zero has an error of zero (though this is accessible only with an infinite On/Off ratio). The value of α_{prop} is also set by the parameter `alpha_error`.

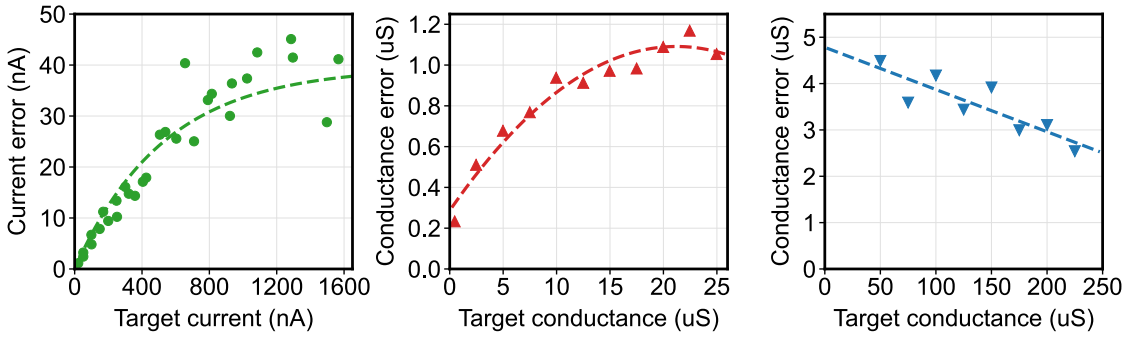

Figure 7.3: Error characteristics of three exemplary devices provided with CrossSim. (a) SONOS device from Xiao *et al.* [21] with a saturating exponential fit to the measured points, (b) PCM device from Joshi *et al.* [13] with a quadratic fit, (c) ReRAM device from Milo *et al.* [15] with a linear fit.

The two simple generic models will not perfectly describe the programming error characteristics of any real device. However, for some cases they are good approximations. For example, the ReRAM devices in Yao *et al.* [25] appear to very closely follow a state-independent error characteristic. By contrast, Xiao *et al.* [21] recently showed that SONOS flash memory devices in the subthreshold regime can have error characteristics that are approximately state-proportional.

Whether or not these generic models fit the properties of any real device, they provide the ability to perform sweeps over the amount of programming error, using the parameter `alpha_error`. This can be used to determine the sensitivity of a neural network's accuracy to varying degrees of error in a typical analog hardware implementation.

In terms of end-to-end accuracy, the key difference between the two error types is data dependence, or its lack thereof. For a given array size, state-independent conductance errors have no dependence on the weight matrix. On the other hand, the overall effect of state-proportional errors depends strongly both on the distribution of weight values used by the neural network, and on the mapping scheme used to map the weights to conductances (see Chapter 6). In practice, we have found that there is generally a large difference in a neural network's sensitivity to the two error types. Since neural networks have weight value distributions that are skewed toward low absolute values, if these low values can be mapped to low conductances (e.g. by using differential cells), then it is highly advantageous to have small programming error at low conductance [22].

7.2.2 Custom conductance error model

The programming error vs. conductance properties of many real devices cannot be modeled as a simple state-independent or state-proportional error model. CrossSim can support arbitrary state-dependent programming error characteristics, provided by the user as a custom function (written in Python). The user can specify a custom error model by modifying the following file:

```
/cross_sim/cross_sim/xbar_simulator/parameters/custom_device/
    weight_error_device_custom.py
```

The function `applyCustomProgrammingError()` in this file will be called each time a resistive memory array is created to map a weight matrix, fully or partially. The argument `input_` to this function contains the conductance values for the array before device errors are applied. These

conductances are in normalized units in the range (1/on_off_ratio, 1). The user can apply an arbitrarily complex error function to this conductance matrix.

Three custom models of device error are provided with CrossSim, based on published literature. These are shown in Fig. 7.3. Consider, for example, the error characteristics of the phase change memory (PCM) device in Joshi *et al.* [13], which is shown in the center plot. The properties of this device can be considered a mixture of state-proportional and state-independent: like state-proportional error, the error increases with conductance, but it does not approach zero error at zero conductance. The effect of this type of error profile cannot be accurately modeled by a generic error model.

To define a custom device programming error model, perform the following steps:

1. In `inference_config.py`, set the parameter `error_model` to a new keyword that describes the device being simulated (e.g. `"HfO2"`)
2. Open `weight_error_device_custom.py`
3. Add a new `if` clause inside the function `applyCustomProgrammingError` that checks that the value of `error_model` matches the keyword entered in the first step.
4. Inside the `if` clause, write a custom error function that modifies the input (unperturbed) conductance matrix `input_` then return the modified matrix.

The devices in Fig. 7.3 are included as examples in the file, and can be used as a template for custom device error profiles added by the user. Each example uses an analytical fit to the data to interpolate the error between the experimentally measured points. In addition to PCM, the file contains two other example programming error profiles, for a SONOS charge trapping memory device using data from Xiao *et al.* [21] and an HfO_2 ReRAM device using data from Milo *et al.* [15].

The conductance matrix passed into `applyCustomProgrammingError` is in normalized conductance units from (On/Off ratio)$^{-1}$ to 1. To apply the error in the PCM case, these values are first de-normalized (scaled and shifted) to lie in the range ($G_{\min}$,$G_{\max}$) of the PCM device. Then the error matrix is computed based on this conductance matrix. The error matrix is used as a matrix of standard deviations for the normally distributed device programming errors.

7.3 Conductance drift

Drift refers to the change in a device's programmed conductance over time, after programming. Physically, this can occur due to structural relaxation within the device (for PCM, ReRAM) or leakage of stored charge from the device (for flash). These time-dependent properties are important for storage-class memory as they determine how long a bit of data can be retained. For analog inference applications these properties are just as (if not more) important since small changes in the device state can lead to large MVM errors when these changes are aggregated over many devices.

Drift can be enabled by setting the parameter `t_drift` to a non-zero value, in units of days. The drift model is selected using the `drift_model` parameter (string). Since drift behavior is highly technology-dependent, there is no generic drift model in CrossSim. Instead, two drift models are provided that correspond to specific device technologies: the 40nm analog SONOS devices in Xiao *et al.* [21] and the PCM devices in Joshi *et al.* [13]. These can be found in the file:

```
/cross_sim/cross_sim/xbar_simulator/parameters/custom_device/
    weight_drift_device_custom.py
```

The function `applyDriftModel()` in this file will be called each time a resistive memory array is created to map a weight matrix, fully or partially. The argument `input_` to this function contains the conductance values for the array before device errors are applied. These conductances are in normalized units in the range (1/`on_off_ratio`, 1). The user can apply an arbitrarily complex function to this conductance matrix that represents the effects of conductance drift and time-dependent random conductance variability.

The SONOS drift model is selected by setting `drift_model="SONOS_interpolate"`. This model includes both (1) the deterministic change over time of the programmed conductance $G(t)$ and (2) the change over time of the conductance error $\Delta G(t)$. The first effect accounts for the average behavior of a device that is programmed to a certain conductance, while the second effect accounts for stochasticity of the drift process as well as device-to-device variations. The model is based on empirically measured conductances up to five days after programming; it does not extrapolate well to significantly longer timescales.

The PCM drift model is selected by setting `drift_model="PCM_Joshi"`. This uses the analytical drift model in Joshi *et al.* [13], where the conductance decays as a power law over time. Note that in their modeled time-dependent accuracy results, Joshi *et al.* applied a drift compensation procedure, which is not included in CrossSim.

To define a custom drift model, perform the following steps, which are similar to the custom programming error model:

1. In `inference_config.py`, set the parameter `drift_model` to a new keyword that describes the device being simulated (e.g. `"HfO2"`)
2. Open `weight_drift_device_custom.py`
3. Add a new `if` clause inside the function `applyDriftModel` that checks that the value of `drift_model` matches the keyword entered in the first step.
4. Inside the `if` clause, write a custom error function that modifies the input (unperturbed) conductance matrix `input_` then return the modified matrix.

The custom model can be interpolation based, like the SONOS model, or can be an analytical expression of conductance and time.

Note that if drift is enabled (`t_drift` is larger than zero), the custom device programming error model will be ignored. This is because a time-dependent error is assumed to be part of the drift model, so disabling the custom programming error model avoids applying the errors twice (once at $t = 0$ and again at $t = t_{\text{drift}}$). However, the generic error model (`error_model = "alpha"`) is not automatically disabled when simulating drift. The programming error is applied prior to applying drift.

7.4 Conductance read noise

Read noise is another type of perturbation applied to the device conductances that represent the weights. Read noise is applied after, and on top of, device programming errors and/or drift as shown in Fig. 7.1. The main distinction between read noise and programming errors and drift is that random read noise is re-sampled on every MVM, rather than only at the core initialization stage. Thus, read noise is dynamic rather than static across different input images as well as different MVMs within a convolution layer. The effect of read noise is not cumulative, i.e. read noise is not re-applied on a conductance matrix that was previously perturbed by read noise.

For a given MVM, read noise is applied in a similar manner to programming errors: for every

conductance, a random number is sampled from a normal distribution and added to the original conductance. Since the noise must be re-sampled for every MVM during inference, enabling read noise can significantly increase the simulation time.

7.4.1 Generic conductance read noise model

Similarly to programming errors, a generic read noise option is avaialable that can apply either state-independent or state-proportional read noise. The generic option is selected by setting `noise_model="alpha"`. To simulate state-independent noise, set `proportional_ noise=False`. To simulate state-proportional noise, set `proportional_noise=True`. Then, set `alpha_noise` to the desired value of α_{ind} or α_{prop}, which are defined in exactly the same way as in Section 7.2.1 for generic programming errors.

Notably, while the effect of read noise on end-to-end accuracy is qualitatively similar to the generic programming errors in Section 7.2.1, the magnitude of the effect is not the same, due to the different re-sampling properties. This is true especially if input bit slicing is enabled (Section 8.2), where read noise is re-sampled for every input bit during an MVM, and there is opportunity for the random noise at different input bits to cancel when the partial results are aggregated.

7.4.2 Custom conductance read noise model

The read noise standard deviation vs. conductance properties of a real device (which does not have purely state-independent or state-proportional noise) can be specified by the user by modifying the following file:

```
/cross_sim/cross_sim/xbar_simulator/parameters/custom_device/
    weight_readnoise_device_custom.py
```

Rather than directly applying a perturbation to the weight matrix, a *matrix of noise standard deviations* is computed based on the values of the conductances in the matrix, after programming errors and drift have been applied. Inside the function `setDeviceReadNoise()`, the user supplies the function that computes this standard deviation matrix. This function will be called during set-up for each resistive array in the system. Then, during the course of the inference simulation, read noise is sampled from normal distributions with the computed conductance-dependent standard deviations on every MVM.

An example of a custom read noise characteristic is provided in the above file, for a fully hypothetical device (`noise_model="parabolic"`). The hypothetical device has a noise standard deviation that varies as a quadratic function of the conductance.

Chapter 8

Array parameters

This section describes the parameterization of the MVM array and the way that it is driven, independent of the memory device technology used.

8.1 Array size and matrix partitioning

It may not always be feasible to fit the full weight matrix onto a single MVM array, due to practical size limitations on the memory array. For example, larger memory arrays may require better select devices to program reliably. Also, in a large memory array, device error accumulation and parasitic resistance have a larger impact on the accuracy of an MVM.

CrossSim supports partitioning the weight matrix onto multiple MVM arrays, *along the column dimension*, i.e. the dimension along which cell currents are summed to form a dot product. The input vector is also partitioned. The maximum number of rows per partition is set by the parameter `NrowsMax`. This partitioning is shown in Fig. 8.1. A single partition of the matrix can still correspond to multiple MVM arrays representing different weight bit slices and/or weight polarities for that matrix partition. Partitioning is not currently supported along the row dimension. Device errors do not accumulate along the rows, though in some systems parasitic voltage drops can accumulate (see Section 8.3).

Partitioning of the N_1 rows of the weight matrix is done according to the following rules. If digital bias addition is disabled, N_1 includes a bias row.

- If N_1 is a multiple of `NrowsMax`, then the number of rows in each array is equal to exactly `NrowsMax`.
- If N_1 is not a multiple of `NrowsMax`, then the N_1 rows of the original matrix will be evenly distributed across the constituent arrays, where each array has at most `NrowsMax` rows.
  - If N_1 is not divisible by the number of partitions, then the partitions may differ slightly in the number of rows.
- If N_1 is less than or equal to `NrowsMax`, the array is assumed to have N_1 rows.

The dot products computed in each partition are separately digitized by an ADC as shown in Fig. 8.1. These digitized results are then added to form the full dot product. Different partitions within the same weight bit slice share the same ADC limits.

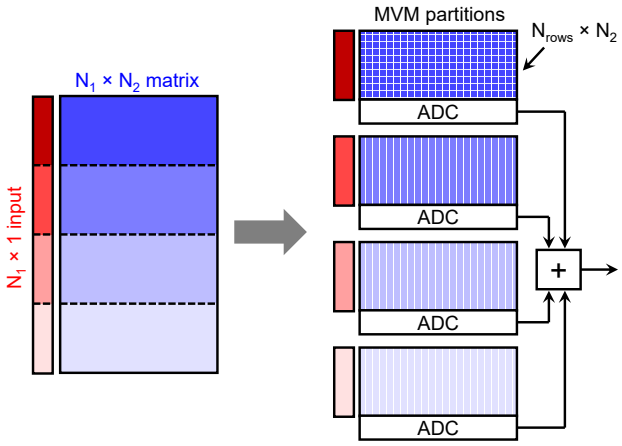

Figure 8.1: Partitioning of a large weight matrix onto multiple MVM arrays.

8.2 Input bit slicing

There are two general methods to drive an MVM array with a vector of multi-bit inputs $\vec{x}$. The first option is to use a digital-to-analog converter (DAC) to represent the value of an element x_i as a voltage. This voltage then drives the row, and cell currents are computed by Ohm's law. This option can be enabled by setting the parameter `input_bitslicing=False`. This method can be energy- and area-inefficient, due to the complexity of having a multi-bit DAC for every row of the array.

The second option is to apply the bits of $\vec{x}$ one at a time to the array, then aggregating the MVM results from the different input bits: this is called *input bit slicing*. For an input with B_{in} bits of resolution, this method involves B_{in} sequential analog MVM operations to compute the full matrix-vector product. This is done for every weight bit slice, and for every matrix partition. Input bit slicing can be enabled by setting `input_bitslicing=True`. Note that this option can significantly increase simulation time, since a separate MVM simulation is required for each input bit.

Fig. 8.2 shows the circuit implementations of the two methods. For input bit slicing, no DAC is needed, and one bit of $\vec{x}$ is selected by a digital multiplexer. There are also hybrid solutions that use an intermediate-resolution DAC [6], but this is not currently supported.

If no device- or array-level non-idealities are included, the two options yield the same result. Otherwise, they can yield different results for a number of reasons. If read noise is enabled, the conductance noise is re-sampled for each input bit. If parasitic resistance is enabled, the parasitic voltage drops will be different in the two cases since the voltage driving the rows will be different. For input bit slicing, the effect of parasitic resistance can vary dramatically among the input bits, since the higher-significance bits tend to be more sparse [16, 23].

The resolution of the input $\vec{x}$ is set by the parameter `dac_bits`, which will be explained in more detail in Chapter 9.

When using input bit slicing, the intermediate MVM result for each input bit is aggregated according to:

$$\vec{y} = \sum_{k=0}^{B_{\text{in}}-1} 2^k \left(\mathbf{W}\vec{x}_k\right) \tag{8.1}$$

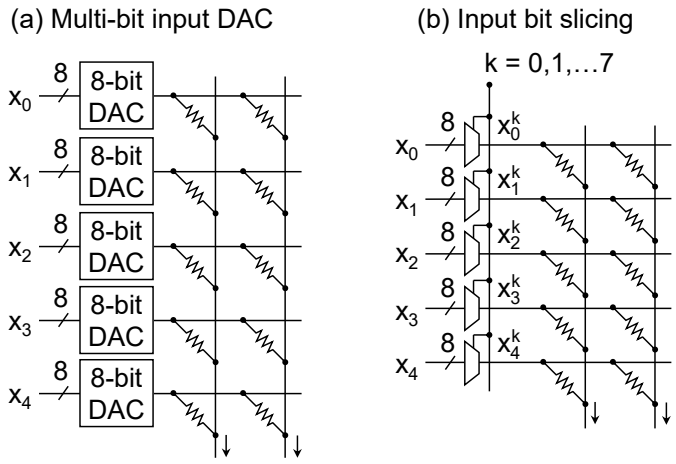

Figure 8.2: Two supported methods for applying a multi-bit input to an array (8-bit input shown, for illustration).

where B_{in} is the number of bits in $\vec{x}$ and we have ignored weight bit slicing and matrix partitioning here for simplicity. This weighted sum can be computed digitally using shift-and-add operations applied to the intermediate MVM results (in parentheses). In this case, the intermediate MVM results are digitized. This option is set with the parameter `ADC_per_ibit=True`.

Alternatively, it is possible to compute this weighted sum in the analog domain using specialized analog capacitive shift-and-add circuits [4]. In this case, only the left-hand side of the equation above is digitized, and only a single ADC operation is needed for all B_{in} input bits. This option is set with the parameter `ADC_per_ibit=False`. Setting the ADC range for these cases will be described in more detail in Chapter 9.

8.3 Parasitic resistance

The resistance of the array's metal interconnects induces parasitic *IR* voltage drops along the columns and/or rows of the array. These voltage drops distort the voltage seen by the memory cells, and induces errors in the cell currents that are spatially non-uniform and highly dependent on both the weight matrix values and the input vector. The effect grows super-linearly with array size, as each new row contributes both a line resistance and a source of current. Parasitic resistance is one of the main factors that limits the size of an MVM array.

CrossSim evaluates the effect of the parasitic voltage drops using a built-in circuit simulator, optimized for array MVM simulations. The parasitic resistance model is incorporated into each analog MVM simulation. It is separately modeled for all input bits, all weight bit slices, and all array partitions.

The effect of parasitic resistance on MVM accuracy depends strongly on the size of the parasitic resistance relative to the device resistance, the weight mapping method (Chapter 6), whether or not input bit slicing is used (Section 8.2), the memory cell design, and the array electrical topology (see below).

Table 8.1: Electrical topology parameters for parasitic resistance simulation

Topology	`noRowParasitics`	`style`	`interleaved_posneg`	`input_bitslicing`
A	`False`	`"balanced"` or `"offset`	Ignored	`True` or `False`
B	`True`	`"balanced"` or `"offset`	`False`	`True`
C	`True`	`"balanced"`	`True`	`True`

8.3.1 Specifying parasitic resistance

The resistance along a metal line is modeled by inserting a resistor R_p between two cells of the array, along a column and/or along a row (see below). This resistance is specified using the parameter `Rp`. If `Rp=0`, the parasitic resistance model is disabled.

The parasitic resistance R_p is specified using normalized units, relative to the *minimum resistance* R_{min} *of a memory device*. For example, if a cell with $R_{\text{min}} = 100$ kΩ is used, a value of $R_p = 10^{-5}$ represents a resistance of 1Ω between every two cells in the array. If parasitic resistance is modeled on both the rows and columns, the simulation assumes a square unit cell, such that R_p has the same value along both the rows and columns.

8.3.2 Array electrical topologies

The effect of parasitic resistance depends on the electrical topology of the memory array. CrossSim currently supports three different electrical topologies, shown in Fig. 8.3(a)-(c). In each topology, a 1T1R memory cell is assumed, where the resistive device is in series with a select transistor. Table 8.1 shows the parameter settings in the inference config file to select each topology. If $R_p > 0$ and the settings do not match one of the options in Table 8.1, an error will be thrown.

CNN inference requires a large number of MVMs: for instance, ResNet50 for ImageNet requires $\sim 10^6$ analog MVMs per input image (with input bit slicing enabled). Thus, to feasibly simulate full neural network inference while accounting for parasitic resistance effects, the three topologies in Fig. 8.3(a)-(c) are simulated using the approximate circuits in Fig. 8.3(d)-(f), respectively. The three topologies and their approximations are described below.

Topology A

In Topology A, shown in Fig. 8.3(a), a common gate bias turns on all the select transistors in the array during an MVM. The select transistor connects the memory device to its corresponding column. The input voltage is directly applied to the memory cell, and thus the cell currents are supplied from the same line that carries the input signal. This means that parasitic voltage drops are present across both the rows and columns of the array. This topology is commonly used with ReRAM or PCM devices [13, 14, 25].

For computational tractability, CrossSim simulates this topology using the approximate circuit in Fig. 8.3(d). This approximation assumes: (1) the memory devices behave as linear resistors under small-signal voltage fluctuations, and (2) the select devices are fully transparent (ideal short circuits) during an MVM. The second assumption is valid if the select transistor has a much lower resistance

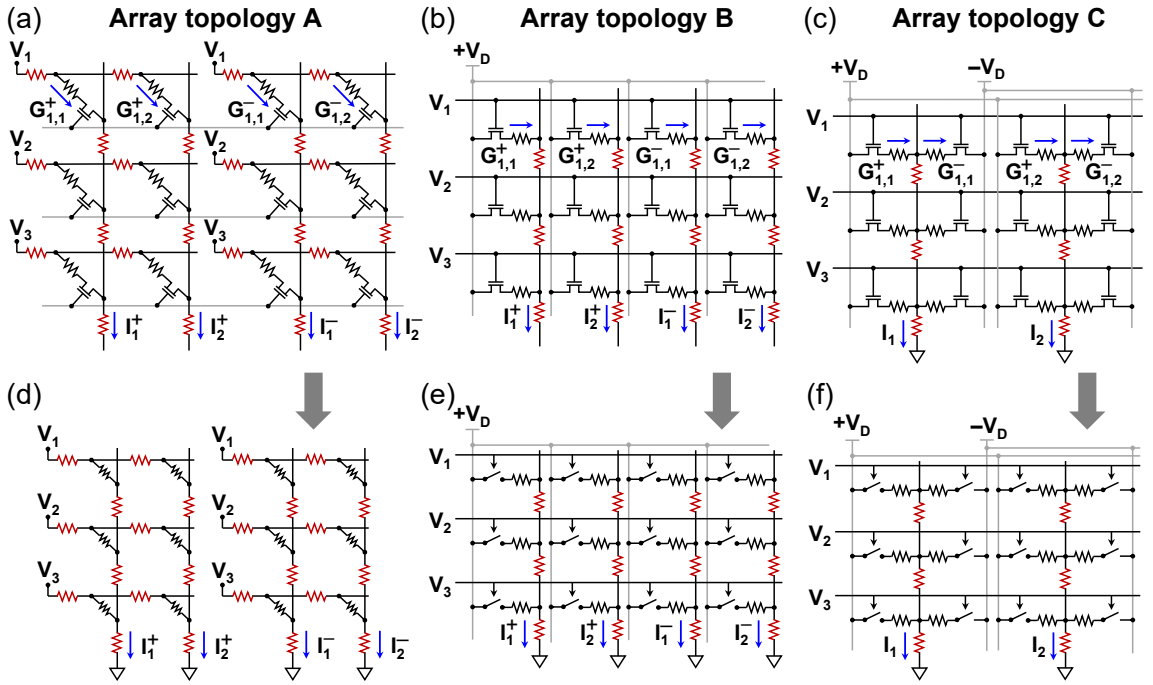

Figure 8.3: (a)–(c) Three array topologies supported by CrossSim, shown for differential cells. Topologies B and C assume input bit slicing is used, such that the input voltages are binary. Black resistors denote memory devices, red resistors R_p are parasitic. For offset subtraction, only the G^+ cells are used. Topology C is only relevant for differential cells. (d)–(f) Approximate circuit models for the three topologies using linear resistors and ideal select devices.

than the memory device when on. This circuit approximation also means topology A can be used to model 1R arrays with no select device.

This setting can be used whether or not input bit slicing is enabled. It is also compatible with both differential cells and offset subtraction.

Topology B

In Topology B, shown in Fig. 8.3(b), the input is applied to the gate of the select transistor, which is used as a switch that connects or disconnects the resistive device to a fixed voltage supply. Since the transistor is used as a switch, the input voltage must be binary: i.e. input bit slicing is assumed. This topology is more commonly used with variable-resistance flash memory devices [2, 8, 10, 21], but can also be used with other resistive memory technologies.

The advantage of this topology is that because the transistor gates draw a negligible current, the input signals are not distorted as they travel along the rows to a given cell. Thus, parasitic resistance along the rows do not need to be modeled. We additionally assume that the voltage supply (V_D), which is shared by all cells, can be sourced from a low-resistance distribution network. Under this assumption, only the parasitic resistance of the columns, where the cell currents are summed, is relevant. Thus, the approximate circuit in Fig. 8.3(e), which includes parasitic resistance only on the columns, is used. The select devices are assumed to behave as ideal short circuits when the input bit is high, and as ideal open circuits when the input bit is low. Thus, for a row with a low input bit,

all cells connected to the corresponding row draw zero current (this is not true for Topology A).

This topology is compatible with both differential cells and offset subtraction, but requires input bit slicing.

Topology C

Topology C, shown in Fig. 8.3(c), is a modification of Topology B specifically for the case where differential cells are used. The cells for the positive and negative weights are interleaved, and each pair is connected to the same column. The positive cell (connected to a positive supply) adds current the column while the negative cell (connected to a negative supply) subtracts current. Thus, the current subtraction needed to implement differential cells occurs within each pair of cells, rather than at the periphery of the array. Currents of different polarity along the bit line can cancel to reduce voltage drops. The sensitivity of this design to parasitic resistance has been shown to be similar to that of Topology B [23]. Fig. 8.3(f) shows the approximate circuit used for simulation.

This topology is compatible only with differential cells, and requires input bit slicing.

8.3.3 Parasitic circuit simulation performance

To simulate the circuits in Fig. 8.3(d)-(f), the currents along the rows and columns are first computed assuming zero wire resistance. The voltage drop across each parasitic resistance is then computed from these currents, and the resulting node voltages along the rows and columns are used to modify the values of the cell currents. These cell currents are subsequently used to re-compute the line parasitic voltage drops iteratively. The simulation is considered to have converged when the worst-case change in node voltage ΔV between iterations falls below $0.0002V_D$. This threshold was determined empirically based on the stability of the end-to-end inference accuracy of ImageNet neural networks. This threshold (0.0002) can be increased to reduce computation time, at a possible cost to simulation fidelity. This can be done by changing the internal simulation parameter `params.numeric_params.Verr_th_mvm` in `inference_util/inference_net.py`.

The linearizing simplifications in the approximate circuits enable the MVMs to be computed using fast matrix operations. A large performance improvement is possible with GPU acceleration. Nonetheless, including the parasitic resistance circuit model ($R_p > 0$) incurs a large performance penalty compared to the case without parasitic resistance. The performance penalty can increase sharply as larger values of R_p are used, which increases the parasitic voltage drops and consequently, increases the number of iterations to convergence in the circuit simulation. The GPU performance of parasitic resistance simulations is discussed in Section 10.3.4.

8.4 Column current limits

CrossSim provides an option to artificially limit the amount of current at the bottom of the column. This is a simple model that first calculates the sum of all the cell currents at the bottom of the column, then clips it to the specified maximum if the sum is too large. This clipping is applied separately to the positive and negative columns for differential cells, before the currents are subtracted.

This is enabled by setting the parameters `Icol_max` and `Icell_max` to non-zero values, representing the current (in Amps) of the maximum allowable column current and the maximum cell current.

Chapter 9

ADC and activation quantization

The analog-to-digital conversion process is an important source of error in analog neural network computation. In CrossSim, an ADC is parameterized by two main quantities: its resolution (bits) and its range. Fig. 9.1(a) shows how these two values are used to convert an analog signal to a digital level. The ADC introduces two types of error: quantization and clipping. Quantization refers to the information loss induced by converting a continuous-valued signal to a discrete level. Clipping error is induced when the signal lies outside the ADC range. Each weight bit slice and matrix partition has a separate ADC, and the aggregation of results from different partitions or weight bit slices is assumed to occur error-free in the digital domain. Aggregation of results from different input bits can be carried out in the analog or digital domain as discussed in Section 8.2.

In addition to ADC quantization, activation values that are the inputs to MVMs can also be quantized, as shown in Fig. 9.1(b). This quantization step is done in the same way as ADC quantization, but is generally not redundant with the ADC, since the results from multiple arrays may be summed and an activation function may be applied before this quantization step. Therefore, the value range used for this step is often different from that used for the preceding ADC. In the code, the activation quantization parameters refer to a DAC (e.g. `dac_bits`, `dac_ranges`) but this quantization step is not necessarily performed by a DAC, and can be done via digital arithmetic with digital inputs and digital outputs.

9.1 Setting the ADC and activation resolution

The ADC and activation resolution can be set by the parameters `adc_bits` and `dac_bits`, respectively, in `inference_config.py`. These are specified as the number of bits B where the number of quantization levels is 2^B. For either case, a value of zero disables the corresponding quantization step.

By default, the resolution set by `adc_bits` and `dac_bits` is applied to all layers in the neural network. It is possible to separately set the quantization resolution for each layer, by using the vectors `adc_bits_vec` and `dac_bits_bits` in `run_inference.py`. These are $1 \times N_{\text{layers}}$ vectors where N_{layers} is the number of MVM layers in the neural network (convolution or fully-connected), and an index of 0 refers to the first MVM layer. As shown in Fig. 9.1(b), the i^{th} element of `dac_bits` specifies the resolution of the i^{th} layer's *input* activations.

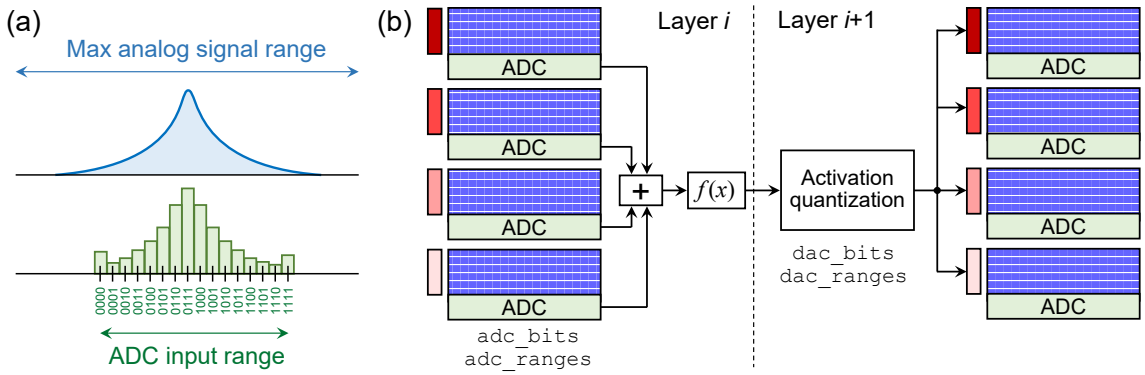

Figure 9.1: (a) Top - distribution of analog values before ADC, Bottom - distribution of digitized ADC outputs, with quantization and clipping errors (4 bits shown as an example). (b) Quantization is applied to each array's analog outputs by the ADCs, and to the digital activation values prior to the next layer. If using differential cells, the subtraction is done in analog before the ADC.

9.2 ADC and activation quantization behavior

In CrossSim, the quantization process for both the ADC and the activations is currently assumed to be deterministic. The quantization levels are always assumed to be equally spaced. Any signal that exceeds the maximum level in the range is clipped to that level, and vice versa for the minimum.

9.2.1 ADC behavior

The following summarizes the behavior of the ADC model under different hardware parameterizations:

- If using differential cells, the subtraction is done in analog before the ADC.
- If a matrix is partitioned spatially over multiple arrays, the results from each partition are digitized by an ADC, then these quantized outputs are summed as shown in Fig. 9.1(b).
- If weight bit slicing is used, results from each weight bit slice are digitized by an ADC, then accumulated via shift-and-add.
- If input bit slicing is used, there are two options for where digitization occurs:
  1. If `ADC_per_ibit` is `False`, the shift-and-add accumulation of results from different input bits is done in analog, then the final result is digitized.
  2. If `ADC_per_ibit` is `True`, the ADC is separately applied to the results from each input bit, then these digitized results are accumulated via shift-and-add.

9.2.2 Activation quantization and input bit slicing

The activation quantization behavior is less dependent on hardware parameters, with the exception of input bit slicing. If input bit slicing is enabled, the `dac_bits` parameter sets the number of bit-wise MVM simulations that are needed per full-precision MVM.

- If the input activations are strictly non-negative (e.g. following a ReLU activation), all of the activation bits are magnitude bits and the number of bit-wise MVMs performed is equal to `dac_bits`.

- If the minimum of the activation range is negative, then one of the activation bits is treated as a sign bit. In this case, it is assumed that three input voltages are available: a positive voltage, a negative voltage of the same magnitude, and zero. This allows both positive and negative inputs to be applied simultaneously for each magnitude bit, and thus only `dac_bits - 1` bit-wise MVMs are needed (the number of magnitude bits). The provided activation range for the layer is converted to a range that is symmetric about zero.

9.3 Setting the ADC ranges

The range of the ADC is specified by the values of the minimum and maximum quantization levels. Any input that falls outside this range are clipped to the limits. These ranges are *static* and do not change during inference. This range is generally, but not always, specified as a (min,max) tuple and is set separately for each of the layers in the neural network. This can be done using one of several global options using the parameter `adc_range_option`, explained in the subsections below. If `adc_bits` is set to zero, the ADC range option is not used.

The scope of the ADC ranges is summarized by the following:

- Different layers use different ADC ranges
- Different weight bit slices use different ADC ranges
- Different matrix partitions within the same weight bit slice use the same ADC ranges
- If input bit slicing is enabled and `ADC_per_ibit=True`, different input bits within the same weight bit slice use the same ADC ranges
- Different columns of the same array (i.e. same layer and same weight bit slice) use the same ADC ranges

9.3.1 Calibrated ADC range

The calibrated ADC range option is enabled by setting `adc_range_option="calibrated"` in the inference configuration file.

In this mode, the ADC ranges for each layer and for each weight bit slice within a layer are loaded from files stored in the `/adc/adc_limits/` folder. These files are NumPy arrays (`.npy`) containing parameters used to specify the range, summarized below. The specific ADC ranges stored in these files are optimized to minimize both quantization and clipping errors, and ideally produce the highest accuracy achievable at a given ADC resolution. If the range is small, the separation between levels is small and the quantization errors will be smaller; however, the clipping errors will be larger. If the range is large, quantization errors will be larger but clipping errors will be smaller. Fig. 9.2(b) illustrates a roughly optimal choice of ADC range at two resolutions. These same considerations have been applied for post-training quantization of neural networks to be run digitally using low-precision computations [12].

Different calibrated ADC ranges are needed whenever there is a significant difference in the ADC input value distributions. This is why different layers and different weight bit slices within a layer have unique ADC ranges[1]. This also means that for a given neural network, a different set of

[1]The only exception to this rule is that different input bits within the same layer and weight bit slice will always use the same ADC limits (when `ADC_per_ibit=True`), even though the ADC input distributions can vary significantly between the input bits. This is due to the practical difficulty of dynamically changing the ADC range with the input bit position.

calibrated ADC ranges is needed for all layers when *any* of the following hardware parameters are changed:

1. Number of weight bit slices (`Nslices`)
2. Maximum number of rows per array (`NrowsMax`)
3. Negative number representation (`style`)
4. Whether the ADC is applied after every input bit with input bit slicing (`ADC_per_ibit`)

Determining the calibrated ADC ranges is not straightforward, and must be done based on the statistical distributions of the input values for each ADC in the entire system. These statistics must first be gathered by running profiling simulations, then the range is found by an optimization procedure. This is described in Section 9.4.

Calibrated ADC ranges are provided for a variety of hardware parameters for several of the provided neural networks, but these provided ranges do not cover all possible values of the hardware parameters above. Please see the function `load_adc_activation_ranges()` in `/inference/inference_util/CNN_setup.py` to see which combinations of neural networks and hardware parameters have provided ADC ranges. Notably the ResNet50v-1.5 network has the largest, though not complete, library of provided ADC ranges with flexibility in all four parameters above. If calibrated ranges are not available for a desired neural network and/or hardware settings, a `ValueError` will be raised inside `load_adc_activation_ranges()`. In this case, the calibrated ADC ranges must first be generated, stored in `.npy` files, then added to this function. CrossSim provides tools to find these calibrated ADC ranges, described in Section 9.4. Alternatively, one of the parameter-less ADC range options (see Sections 9.3.2 and 9.3.3) can be used.

Further details are provided below on how the calibrated range parameters are used.

Unsliced weights

For unsliced weights, the neural network's ADC ranges are provided in a $N_{\text{layers}} \times 2$ array, where for each layer a tuple of floating-point values $(y_{\text{min,ADC}}, y_{\text{max,ADC}})$ specifies the values of the minimum and maximum ADC levels. These are in algorithmic units, i.e. raw activation values, not normalized or converted to hardware units. CrossSim internally converts these to units usable by the ADC model. Here we introduce the notation y to denote array MVM results in analog form, i.e. ADC inputs.

Bit-sliced weights

For bit-sliced weights, the neural network's ADC ranges are provided in a $N_{\text{layers}} \times N_{\text{slices}}$ array. For a given layer and weight bit slice, the ADC range is specified by a single integer C. The ADC limits for the i^{th} bit slice in a given layer are then given by:

$$y_{\text{min,ADC},i} = -y_{\text{max}} \times 2^{-C_i} \text{ or } 0 \tag{9.1}$$

$$y_{\text{max,ADC},i} = y_{\text{max}} \times 2^{-C_i} \tag{9.2}$$

where y_{max} is the maximum possible value of an array MVM result that is mostly a function of array size and not the distributions of weight or activation values, i.e. it is the largest possible value of y assuming every input and weight is the largest possible. If the ADC inputs are strictly non-negative, then the ADC minimum is automatically set to 0. This can occur if (1) the layer only accepts non-negative inputs and (2) offset subtraction is used.

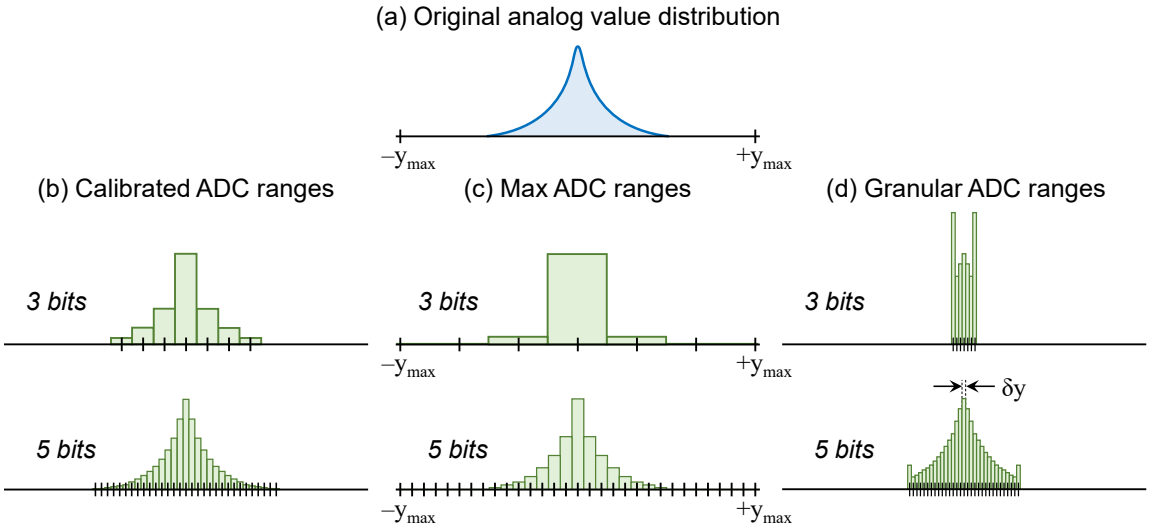

Figure 9.2: (a) Distribution of analog values before the ADC, (b) Distribution after ADC with calibrated ranges, (c) Distribution after ADC with max ranges, (d) Distribution after ADC with granular ranges, for two resolution settings.

The value of $y_{\max}$ is the same for all slices in a layer, but C_i can differ for different bit slices. Since C_i must be an integer, and the ADC minimum and maximum are not controlled separately, the definition above is more restrictive in the possible ADC ranges than for unsliced weights. This is to accommodate the practical implementation of weight bit slicing, which uses shift-and-add operations to combine the results from different bit slices. For this type of accumulation to work, the ADC ranges can only differ by a power of 2; otherwise, results from different bit slices must be scaled by a much more complicated arithmetic unit before being added together. With the definition above, shift-and-add accumulation is still possible; only the number of shift bits between slices may change based on the calibrated ADC ranges. As an example, consider an 8-bit weight divided into two 4-bit slices ($i = 0$ is the low slice, $i = 1$ is the high slice). If $C_0 = C_1$, then the least significant slice result is shifted left 4 bits before being added to the most significant slice results. If $C_0 = C_1 + 2$, then the least significant slice result is shifted left by 2 bits instead.

Range zero-centering

Since many statistical ADC input distributions are symmetric about zero, it is often very important to have an ADC level at exactly the value of zero. This avoids any potential systematic quantization errors. For cases without weight bit slicing, this is done by offsetting the provided calibrated ADC range such that a level in the middle of the range is placed exactly at zero. For cases with weight bit slicing, this is guaranteed by the ADC limits given by Equation (9.1) and (9.2). If $y_{\text{min,ADC}} < 0$, then the number of quantization levels is reduced by one such that it is odd, i.e. there are $2^B - 1$ levels. This ensures a symmetric range about zero with a level at exactly zero.

9.3.2 Max ADC range

The max ADC range option is enabled by setting `adc_range_option="max"` in the inference configuration file.

This setting sets every array's ADC limits to the minimum and maximum possible values of the array's MVM output. It requires no calibration nor any external files to set the ADC ranges. For every array, the range is set according to:

$$y_{\text{min,ADC}} = -y_{\text{max}} \text{ or } 0 \tag{9.3}$$

$$y_{\text{max,ADC}} = y_{\text{max}} \tag{9.4}$$

where y_{max} is the maximum possible value of an array MVM result that is mostly a function of array size and not the distributions of weight or activation values, i.e. it is the largest possible value of y assuming every input and weight is the largest possible. This setting is illustrated in Fig. 9.2(c). If the ADC inputs are strictly non-negative, then the ADC minimum is automatically set to 0. This can occur if (1) the layer only accepts non-negative inputs and (2) offset subtraction is used. The ADC range is independent of the chosen resolution. The separation between ADC levels decreases exponentially with the bits of resolution.

The benefit of the max ADC range setting is that it is simpler to run: it allows an ADC simulation without having to calibrate the ranges to a specific neural network and hardware settings. Another benefit of this setting is that it avoids ADC clipping errors, since the limits are set to the most extreme values accessible.

The disadvantage of this method is that it has high quantization errors, due to the large spacing between levels. It often produces a much lower accuracy than calibrated ADC ranges for a given ADC resolution. This is because in practice, the extreme ADC input values (close to $\pm y_{\text{max}}$) are very rarely seen because both the weights and activations in a neural network have value distributions that are heavily skewed toward zero [22]. This is illustrated by comparing Fig. 9.2(c) to Fig. 9.2(b). The max ADC range setting will generally only produce a high accuracy when the ADC resolution is very high, possibly unrealistically high.

9.3.3 Granular ADC range

The granular ADC range option is enabled by setting `adc_range_option="granular"` in the inference configuration file.

Like the max setting, this option sets the ADC ranges without requiring any calibration, but takes the opposite approach. Rather than using the maximum possible output value y_{max}, the granular option is based on the minimum possible separation between analog outputs δy. This minimum separation is well-defined *under the assumption that the array MVM introduces zero errors or noise*, which will not be true in practice. Nonetheless, it still provides a calibration-free method to set the ADC ranges since δy is only a function of the array size and weight quantization. The separation between ADC levels is set to a fixed value of δy regardless of resolution as shown in Fig. 9.2(d). The range of the ADC increases exponentially with the bits of resolution.

An equivalent way to define the minimum separation δy is the minimum possible non-zero MVM output when the array is free of errors. For this value to be well-defined, CrossSim currently requires:

1. Weight quantization is enabled (`weight_bits > 0`)
2. Activation quantization is enabled (`dac_bits > 0`)

3. Input bit slicing is enabled (`input_bit_slicing = True`)
4. ADC is applied after every input bit (`ADC_per_ibit = True`)

Like the max setting, the benefit of the granular setting is that it allows an ADC simulation to be run without having to calibrate the ranges to a specific neural network and hardware settings. In the theoretical case where the array introduces zero analog errors, this setting also eliminates quantization errors. However, since the range depends directly on resolution, an insufficient resolution leads to large clipping errors, as shown in Fig. 9.2(d).

The ADC resolution that eliminates all clipping errors can be calculated (assuming 1-bit input slices):

$$B_{\mathrm{ADC}} = B_{\mathrm{W}} + \lceil \log_2 N \rceil \tag{9.5}$$

where B_W is the number of weight bits and N is the number of rows activated in an MVM.

Many analog accelerators published in the literature adopt the convention of setting the ADC level separation to δy and setting the ADC resolution to B_{ADC} above [3, 5, 19, 24]. This has been called the *full-precision guarantee* since it eliminates both quantization errors and clipping errors [22]. By using the granular option and setting the ADC resolution to the value given above (with N set to `NrowsMax`, or the otherwise largest array in the system), one can confirm using CrossSim that ideal accuracy can be achieved on a neural network when all other analog errors are disabled [22].

There are, however, two caveats to using the full-precision guarantee. First, "full precision" is only meaningful under the ideal condition that there are *no analog errors*. If the accumulated effect of device programming errors, read noise, drift, or parasitic resistance on a column is larger than the separation between ADC levels, then quantization errors may be small in comparison to the analog errors. Thus, setting the level separation to δy would not be the optimal choice. Secondly, in general near-ideal accuracy can be obtained at a lower ADC resolution than B_{ADC} bits (given above) by using calibrated ranges, even if this choice does not theoretically guarantee zero quantization or clipping errors. This is especially true if differential cells are used, rather than offset subtraction [22].

9.4 Calibrating the ADC ranges

Fig. 9.3 shows the workflow for calibrating the ADC ranges for a given neural network and hardware parameters. A new ADC calibration should be performed each time one or more of the hardware parameters listed in Section 9.3.1 changes, and the ADC ranges corresponding to this case are not provided with CrossSim. The calibration consists of two steps: first, a profiling inference simulation is performed with ADCs disabled and the ADC input values for all arrays are collected. Second, the statistics of the collected ADC input values is used to find the optimal ADC range that minimizes both quantization and clipping errors. After these ranges are found, they are saved to `.npy` files and can be subsequently loaded into inference simulations.

ADC input profiling

The first step to find the calibrated ADC ranges is to collect the statistics on the ADC inputs of every array. This is done using a profiling simulation, which differs from a normal inference simulation in four main ways:

1. The ADC is disabled, i.e. the ADC is an identity function

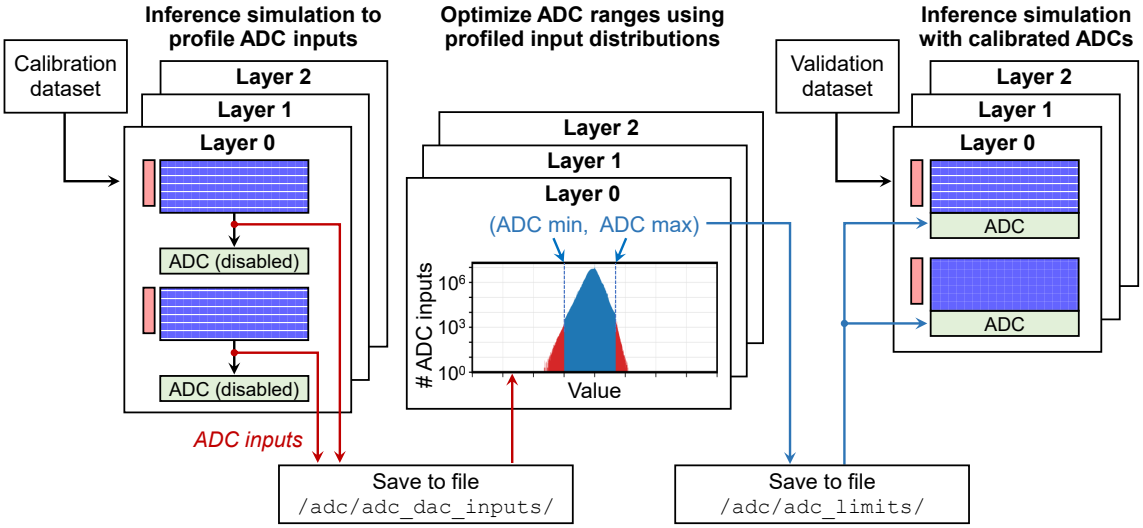

Figure 9.3: Workflow for calibrating the ADC ranges for a given neural network and hardware parameters.

2. During the simulation, all ADC inputs are saved (across all layers, all slices, all input images, and all MVMs)
3. A special dataset, called the calibration set, is used that should not be the same as the validation or test dataset (see Section 9.4.1)
4. The parameters `x_par` and `y_par` are forced to equal 1 to conserve memory (see Section 10.2)

The hardware configuration used for profiling should be specified in `inference_config.py`, as before. The parameter `adc_bits` must be set to `0`, but all other hardware settings should match the hardware settings that would be used after the calibrated ADC ranges are found. To start a profiling simulation, run the following command from the `inference` directory (after modifying the file as described below):

```
python run_inference_profiling.py
```

Currently, all ADC inputs seen during the profiling simulation are stored in memory, then saved to disk at the end of the simulation. Please note that depending on the dataset and neural network, **the output files from an ADC profiling run can consume a large amount of memory and disk space.** If memory runs out before the profiling simulation ends, it may be necessary to break up the calibration set into smaller chunks. The number of ADC inputs depends on the total number of ADC conversions in the system. For a given layer, the number of ADC inputs that must be saved increases proportionally with:

- The size of the calibration set
- The total number of outputs from the layer
- The number of matrix partitions
- The number of weight bit slices
- If `ADC_per_ibit=True`, the number of input activation bits (set by `dac_bits`)

ADC input profiling is enabled by setting `profile_ADC_inputs=True` inside the script `run_inference_profiling.py`. The ADC inputs are saved to a collection of `.npy` files inside the directory specified by `profiling_folder`, which can be set. By default, a new directory

Table 9.1: ADC input profiling outputs

Weight bit slicing?	Input bit slicing?	Number of output `.npy` files
No	No	One file per layer
No	Yes	One file per layer, per input bit[†]
Yes	No	One file per layer, per weight bit slice
Yes	Yes	One file per layer, per weight bit slice

[†] Note that all input bits share the same ADC limits, so there is strictly speaking no need to resolve the ADC inputs for different input bits. However, the input bits are resolved for the user's interest, since different input bits can have significantly different ADC input distributions.

is created inside `/adc/adc_dac_inputs/` with a name that matches the hardware settings in `inference_config.py`. How many files are saved in this directory depends on the hardware settings: see Table 9.1. Each file contains a 1D vector of ADC inputs that all share the same ADC range.

ReLU-aware ADC input profiling

When a ReLU activation is applied directly after the ADC, the lower bound of the ReLU can be used to more optimally set the lower limit of the preceding ADC. Any input to the ReLU function that falls below zero is clipped to zero. Any ADC input that eventually meets that outcome does not need to be profiled, since the information in that ADC input would be lost regardless of the ADC limits. Typically, a ReLU is applied to the sum of the ADC output and a known bias value b. Therefore, any ADC input y that satisfies $y + b < 0$ does not need to be profiled (note that the value of b used is different for different columns).

This selective discarding of ADC inputs can be enabled by setting `profile_adc_biased=True` in `run_inference_profiling.py`. This ultimately helps compress the ADC range in such a way that quantization errors are reduced without increasing clipping errors.

In several situations, the selective discarding of ADC inputs cannot be applied, and the option is automatically disabled on a layer-by-layer basis:

- *The layer's weight matrix is partitioned.* Since the result from the other partitions is unknown, it is impossible to predict whether a given ADC input will be clipped by the ReLU.
- *Weight bit slicing is used.* Since the result from the other slices is unknown, it is impossible to predict whether a given ADC input will be clipped by the ReLU.
- *Input bit slicing is used* and `ADC_per_ibit=True`. Since the result from the other input bits is unknown, it is impossible to predict whether a given ADC input will be clipped by the ReLU.
- The activation is a ReLU, but another neural network operation precedes the ReLU (e.g. element-wise addition of another layer's output).
- The activation is not a ReLU.

In summary, any time there is a digital operation between the ADC and the ReLU other than bias addition, this technique cannot be used.

9.4.1 Calibration dataset

The data used to profile the ADC inputs and calibrate the ADC ranges should ideally not have overlap with the dataset used to validate or test the neural network accuracy. Typically, this calibration set

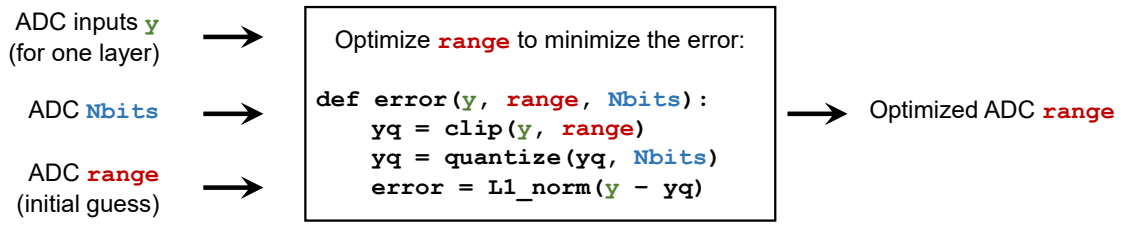

Figure 9.4: Simplified outline of the algorithm used to optimize ADC ranges without weight bit slicing.

can be much smaller than the validation or training sets. For the provided datasets, the calibration option can be enabled by setting `calibration=True` in `run_inference_profiling.py`. The behavior of this parameter depends on the dataset.

- For MNIST, Fashion MNIST, CIFAR-10, and CIFAR-100, a calibration set is created by randomly sampling `ntest` images from the training set, where `ntest` is specified in the configuration file. Note that a different set of calibration images will be selected each time a profiling simulation is started.
- For ImageNet, the pre-processed MLPerf calibration dataset is loaded, which is static and stored in a separate file. See Section 4.3.2.

If desired, the behavior of the `calibration` parameter can be changed by modifying the appropriate function in the file `/helpers/dataset_loaders.py`.

9.4.2 ADC range optimization

After a profiling simulation is completed, the results are saved as `.npy` files in the destination folder in the format summarized by Table 9.1. The next step is to process these ADC input distributions and determine the optimal ADC ranges for each layer and/or weight bit slice. There are many different possible methods to perform this optimization and several example scripts can be found in the `/adc/` directory. The user can also define a custom ADC optimization function.

The calibrated ranges are saved to a `.npy` file, which can be accessed in subsequent inference simulations via the function `load_adc_activation_ranges()` inside `/inference/inference_util/CNN_setup.py`. The user will need to add additional logic to this function to ensure that the correct ADC ranges are loaded.

Range optimization without weight bit slicing

The file `resnet50_adc_limits_1slice.py` is an example script that generates ADC ranges for each layer of a ResNet50 accelerator that does not use weight bit slicing. This file can be adapted for other datasets and neural networks. Profiled ADC inputs are loaded from the directory specified by `profiling_folder` and the ranges are saved to an output at the file specified by `output_name`. By default, the output destination is a file in `/adc/adc_limits/`. The files containing the profiled ADC inputs used by this example script are not included due to their large size, but can be generated using the procedure described in the previous sections.

The file implements the high-level algorithm shown in Fig. 9.4. The value of `Nbits` is a free parameter that specifies the quantization resolution used during the optimization procedure. The

optimal value of this parameter depends on the ADC resolution used during inference, but *the two values do not need to match.* This is because the quantization error computed by the algorithm in Fig. 9.4 is only an approximate proxy for the inference accuracy. In general, a larger value of `Nbits` leads to a larger calibrated ADC range. Empirically, we have found that it is often helpful for `Nbits` to be larger than the actual ADC resolution. In general, we have also found that the inference accuracy is not strongly sensitive to `Nbits` provided that it is not too small.

The range is a tuple specifying the minimum and maximum. In most cases, the minimum and maximum are optimized independently. One exception to this is when ReLU-aware ADC range optimization is used, enabled by setting `reluAwareLimits=True`. In this case, only the maximum is optimized and the minimum is left equal to the smallest value seen in the layer's profiled ADC inputs. This option should only be enabled if the ReLU-aware profiling option was used (see Section 9.4) and some network-specific metadata has to be provided to ensure that this is applied only to the appropriate layers.

The example script `resnet50_adc_limits_1slice_ibit.py` is very similar but processes ADC input files for each input bit, with the assumption that `ADC_per_ibit=True` in the inference simulation. The algorithm is identical since different input bits see the same ADC range.

Range optimization with weight bit slicing

When weight bit slicing is used, the ADC range of each slice is optimized. However, the ADC ranges for different slices are coupled, because they are constrained to differ only by powers of two. This constraint is necessary to enable a practical digital implementation of the shift-and-add aggregation of results from different slices (see Section 9.3.1).

The file `resnet50_adc_limits_bitsliced.py` is an example script that generates ADC ranges for each layer and each weight bit slice of a ResNet50 accelerator that uses weight bit slicing. Rather than directly optimizing the numeric values of the ADC minimum and maximum, this function optimizes the value of C_i for each bit slice as defined by Equations (9.1) and (9.2). The value of C_i is optimized for each layer and each bit slice, then the array containing these values is saved to an `.npy` file. How the optimization is done depends on the negative number representation and whether the layer takes signed inputs as described in Section 9.3.1.

There is a percentile hyperparameter (`pct`) that controls the ADC range optimization process with weight bit slicing. The value of `pct` controls the range of profiled ADC inputs that the ADC limits defined by Equations (9.1)-(9.2) must cover. For example, if `pct=99.999`, then the 99.999^{th} percentile and the 0.001^{th} percentile of the profiled ADC inputs must fall within the ADC limits. A value of `pct` that is closer to 100 will reduce clipping errors, but will increase quantization errors. Several values of this parameter may need to be evaluated to determine the best-performing value (in terms of inference accuracy) at a given ADC resolution. The value of `pct` is included as part of the output file name containing the calibrated ADC ranges. During the inference simulation, the value of `pct` can be specified in `inference_config.py`, and this value is then passed in to the `load_adc_activation_ranges()` function to select the desired set of ADC ranges.

9.5 Setting the activation ranges

As with the ADC, the range of the activations specifies the values of the minimum and maximum quantization levels. These ranges are *static* and do not change during inference. This range is

specified as a (min,max) tuple and is set separately for each of the layers in the neural network. Input activation ranges can differ between layers, but do not differ between weight bit slices or matrix partitions. If `dac_bits` is set to zero, the ranges are not used.

Unlike the ADC ranges, activation ranges do not depend on the hardware parameters such as array size, weight bit slicing, and negative number representation. The same activation ranges can be re-used for many different hardware configurations of the same neural network, with the exception of activation resolution (`dac_bits`). If `dac_bits` is non-zero, calibrated ranges for the input activations must be provided for each layer, similar to the "calibrated" ADC range option.

As mentioned in Section 9.2.2, if input bit slicing is enabled and the activation minimum is negative, then one of the activation bits is treated as a sign bit. This sign-magnitude representation implies that the range is symmetric about zero. In this case, if the provided range is (x_{min}, x_{max}), it is converted to ($-x_{\text{amax}}$,$+x_{\text{amax}}$) where $x_{\text{amax}} = \max(|x_{\text{min}}|, |x_{\text{max}}|)$. Due to the use of the sign bit, there are two redundant representations of zero, and the total number of unique levels is $2^B - 1$.

Calibrated activation ranges are provided for many of the provided neural networks. A full list of these can be found inside the function `load_adc_activation_ranges()` in the file `/inference/inference_util/CNN_setup.py`. In general, these ranges have been optimized for `dac_bits=8` bits, though we have generally found that these same ranges are still very close to optimal for resolutions close to 8 bits. To find calibrated activation ranges for a new neural network, the procedure in Section 9.6 can be used.

9.6 Calibrating the activation ranges

The process for calibrating the activation ranges is similar to the process for calibrating the ADC ranges, described in Section 9.4 and depicted in Fig. 9.3. First, an activation profiling simulation is performed using the calibration dataset and the results are saved to a directory of `.npy` files. Then, the profiled activation inputs are processed to optimize each layer's activation ranges, which are then saved to a `.npy` file. Finally, the calibrated ranges can be selected during an inference simulation as described in Section 9.5. The main difference between ADC and activation range calibration is that as previously described, the optimal activation rages do not directly depend on the negative number representation, number of matrix partitions, weight bit slcing, or input bit slicing. Therefore, for a given neural network, a single calibration step is typically sufficient even if these hardware parameters are changed.

An activation profiling simulation is performed by running `run_inference_profiling.py` with `profile_DAC_inputs=True`. Activation quantization must be disabled, i.e. `dac_bits=0`. The result will be one `.npy` file per neural network layer in the output directory.

After profiling the activations, their range can be optimized. An example script for a ResNet50 accelerator is given in `/adc/resnet50_dac_limits.py`, which uses the same algorithm shown in Fig. 9.4 for the ADC limits. For this particular neural network, because the quantization step always follows a ReLU function, the lower limit of the activation range can always be set to zero and only the upper limit needs to be optimized.

Calibration of the ADC and activation ranges can be performed in either order; in general, we have not found this order to make a large difference, especially at higher ADC and activation resolutions. If the ADC ranges are calibrated second, then the ADC input profiling simulation can be performed with activation quantization enabled, using the previously calibrated activation ranges. The reverse applies if the activation ranges are calibrated second.

Chapter 10

CrossSim Inference GPU performance

CrossSim simulations can be run on a CUDA GPU, if available, to improve performance. To enable this functionality, the CuPy package must be installed, which is used by CrossSim to access the CUDA Toolkit libraries.

10.1 Enabling GPU acceleration

GPU acceleration can be enabled by setting the parameter `useGPU` to True in the configuration file, `inference_config.py`. If multiple GPUs are available on the system, the parameter `gpu_num` selects a GPU based on the ID assigned by CUDA (set to `0` if only one is available). Currently, an individual CrossSim simulation can only be run on one GPU. Multiple independent CrossSim simulations can be simultaneously run on different GPUs.

10.2 Sliding window packing

When simulating CNN inference, the runtime per simulation is typically bottlenecked by the convolutional layers, which must be decomposed into many sliding window MVMs when run on analog arrays (see Section 6.1). CrossSim supports a simulation mode called *sliding window packing* which greatly accelerates the simulation of convolutions. This mode can be enabled with or without GPU acceleration, though the benefits are much greater when using a GPU due to the large matrix operations involved.

Fig. 10.1 illustrates the idea of sliding window packing. A 2×2 grid of sliding windows within the input feature map typically requires four separate MVM simulations to compute. The four MVMs can be packed into a single MVM by constructing a long vector that contains the inputs to all four MVMs (some of which may be repeated). For this packed simulation, the conductance array also needs to be expanded by a factor of four in each dimension. The original conductance array is repeated four times along the diagonal of this expanded array. All other elements that are not part of this block diagonal are set to always have a conductance of exactly zero (actual zero, and not G_{min}). This ensures that the MVM results computed using this expanded array exactly match the results from simulating four sequential sub-MVMs with the original conductance array, i.e. the four sub-MVMs are entirely decoupled. Although a block diagonal matrix is shown in Fig. 10.1, in some cases CrossSim will redistribute the duplicated weights to exploit the fact that many input elements

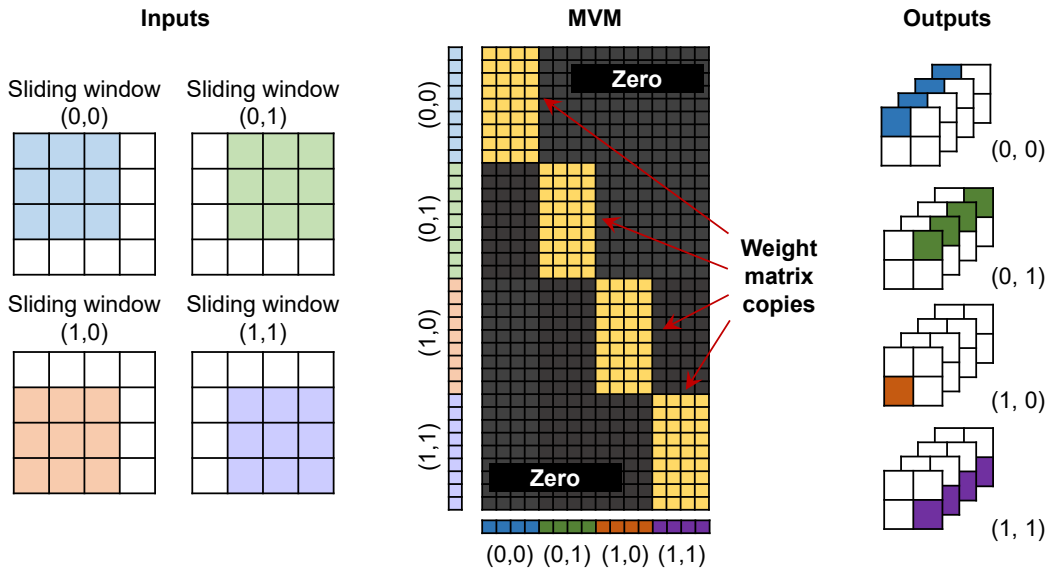

Figure 10.1: Illustration of sliding window packing using an example convolution with a 3×3 sliding window. The input has 1 channel and the output has 4 channels. A 2×2 grid of sliding windows (`x_par=2, y_par=2`) can be computed in parallel by packing them into a single MVM computation. This is done by constructing a block diagonal matrix from the original array conductance matrix (yellow). In some cases, CrossSim will construct a Toeplitz matrix for the convolution, which is more memory-efficient than a block diagonal matrix.

are shared across adjacent sliding windows. This construction allows a smaller matrix to be used with fewer zeros, which saves GPU memory.

More generally, this technique can be used to simulate an $X_{\text{par}} \times Y_{\text{par}}$ grid of sliding windows in parallel using a single packed MVM. Although the individual MVM simulations are bigger, this reduces the number of required MVM simulations by $X_{\text{par}}Y_{\text{par}}$. However, the size of the conductance array in memory might increase by as much as a factor of $\left(X_{\text{par}}Y_{\text{par}}\right)^2$ (for the block diagonal case) even though most of these elements may be zero.

The values of X_{par} and Y_{par} can be set on a layer-by-layer basis for a neural network. These values are set inside the function `get_xy_parallel()` in `/inference/inference_util/CNN_setup.py`. There are two hard constraints on the possible values of X_{par} and Y_{par}:

1. For a given layer, X_{par} must be smaller than the size of the output feature map along x. The same constraint applies to Y_{par}. The shape of the output feature map for each layer can be checked using the `show_model_summary` feature.
2. Across the entire neural network, the values of X_{par} and Y_{par} must be kept small enough that the GPU does not run out of memory when the expanded conductance arrays for all layers (and weight bit slices) are generated.

Additionally, if the product $X_{\text{par}}Y_{\text{par}}$ for a layer is too large, the performance may stop improving or potentially degrade even if the GPU does not run out of memory, because the conductance matrix has become too large to yield a speedup from sliding window packing.

When running an ADC input profiling simulation or an activation profiling simulation (Section 9.4), the values of X_{par} and Y_{par} are automatically set to 1 for all layers, i.e. sliding window packing

is disabled. This is to keep more memory available for the profiled ADC inputs or activations, even if the profiling simulation runs slower.

10.2.1 Parameter tuning for optimal performance

Inside `get_xy_parallel()`, an array of values for X_{par} and Y_{par} is provided for several neural networks and hardware configurations. These values were tuned to give optimal performance for these networks on a Nvidia V100 GPU, while also being able to fit inside 32GB of memory. These may have to be re-tuned when using a GPU with less memory (or more memory). Tuning these parameters without running out of memory may require a trial-and-error procedure.

In general, the early layers of a CNN require the greatest number of sliding windows due to their large input feature maps along x and y. Fortunately, these layers also tend to have the smallest weight matrices due to having fewer input and output channels. This allows X_{par} and Y_{par} to be set to relatively large values for these layers without consuming a large amount of memory, which helps speed up the simulation significantly.

We recommend the following procedure for tuning the X_{par} and Y_{par} parameters for a given neural network model:

- Start by setting $X_{\text{par}} = Y_{\text{par}} = 1$ for all layers, and make sure that this case fits onto the memory.
- Increase X_{par} and Y_{par} gradually for one layer or one group of layers at a time, until either (1) the performance stops improving, (2) X_{par} and Y_{par} reach their maximum values, or (3) the GPU runs out of memory. If an out-of-memory error is encountered, reduce these values.
- Increase X_{par} and Y_{par} for the early layers first. The optimal parameters for performance may call for much larger values for the early layers than for the later layers, as explained above.

When parasitic resistance (Section 8.3) is included in the simulation, the benefits of increasing X_{par} and Y_{par} can diminish much more rapidly. This is because a circuit simulation of an MVM including parasitic resistance is much more computationally complex than a standard MVM simulation, and its complexity increases much more quickly with the size of the conductance array. For this reason, a separate function is use to set X_{par} and Y_{par} when `Rp` is non-zero: `get_xy_parallel_parasitics()`. For any given neural network, the optimal values of X_{par} and Y_{par} are significantly smaller.

10.3 Performance benchmarking

In this section, we benchmark the inference simulation performance of CrossSim for a number of representative use cases. The benchmarking is done using two hardware systems:

- **GPU**: 1× Nvidia V100 GPU, 32GB memory, CUDA 10.2, CuPy v8.3.0
- **CPU**: 2× Intel Xeon 8168, 24 cores each, 2.7 GHz

In all of the following results unless otherwise noted, the performance is based on optimally tuned parameters for sliding window packing.

10.3.1 Performance with GPU acceleration and sliding window packing

Table 10.1 compares the performance of the CPU and GPU on several neural networks. These simulations assumed: differential cells, ADC enabled, `NrowsMax=1152`, no weight bit slicing, no input bit slicing, no read noise, and no parasitic resistance.

When sliding window packing is not used, the performance improves with GPU acceleration if the neural network has sufficiently large MVMs. Optimal sliding window packing enables a

Table 10.1: Comparison of CPU and GPU performance with sliding window packing

Dataset	Neural network	CPU runtime, no sliding window packing	GPU runtime, no sliding window packing	GPU runtime, optimal sliding window packing
ImageNet	ResNet50	20.58s / image	13.59s / image	0.95s / image
ImageNet	VGG-19	82.07s / image	35.50s / image	2.42s / image
CIFAR-10	ResNet-56	1.61s / image	5.68s / image	0.059s / image
MNIST	CNN-6	0.23s / image	0.45s / image	0.0034s / image

significant performance improvement over CPU in all cases, and a $\sim 10\times$ speedup over the GPU baseline without sliding window packing.

We found that sliding window packing generally did not yield any performance improvement on a CPU, even when the values of X_{par} and Y_{par} are re-tuned for the specific neural network. The only exception to this was the smallest network, CNN-6, whose CPU performance improved to 0.05s / image with optimal sliding window packing.

10.3.2 Performance with different analog hardware settings

Table 10.2: Performance on ResNet50 (ImageNet) using different analog hardware settings

ADC enabled	# weight bit slices	# input bit slices	`NrowsMax`	Read noise enabled	GPU runtime
False	1	1	1152	False	0.88s / image
True	1	1	1152	False	0.95s / image
True	1	1	1152	True	4.82s / image
True	1	8	1152	False	5.30s / image
True	4	1	1152	False	3.09s / image
True	1	1	288	False	1.52s / image
True	1	1	144	False	2.28s / image
True	4	8	288	False	32.4s / image
True	4	8	288	True	268.5s / image

Table 10.2 shows the GPU runtime for an inference simulation using ResNet50 on ImageNet, using various settings for the hardware configuration. These simulations assumed: differential cells, 8-bit weight quantization, 8-bit activations, and no parasitic resistance.

Using more weight bit slices, more input bit slices, and more array partitions (smaller `NrowsMax`) all increase the runtime by increasing the number of MVM simulations performed. The effect of `NrowsMax` depends on the neural network. For ResNet50, the largest layers have 4608 inputs while the smallest layers have 64 inputs; thus, for a given value of `NrowsMax`, only a subset of the layers have partitioned weight matrices. Including read noise increases runtime since random numbers must be generated on each MVM simulation. We find that to optimize performance for read noise, it is preferred to have $X_{\text{par}}Y_{\text{par}} > 1$ for as many convolution layers as possible.

10.3.3 Performance vs neural network

Table 10.3: Performance on selected neural networks

Dataset	Neural network	# MVM layers	# weights	GPU runtime
ImageNet	ResNet50	54	25.61M	0.95s / image
ImageNet	VGG-19	19	143.67M	2.42s / image
ImageNet	MobileNetV1	28	4.22M	0.82s / image
ImageNet	InceptionV3	95	23.85M	1.45s / image
CIFAR-100	ResNet56_cifar100	58	13.68M	0.24s / image
CIFAR-10	ResNet56	58	861.8K	0.059s / image
CIFAR-10	ResNet32	34	470.2K	0.029s / image
CIFAR-10	ResNet20	22	274.4K	0.026s / image
CIFAR-10	ResNet14	16	176.6K	0.019s / image
MNIST	CNN-6	6	61.6K	0.0034s / image

Table 10.3 shows the GPU runtime for selected neural network models using the same analog hardware settings. These simulations assumed: differential cells, ADC enabled, `NrowsMax=1152`, no weight bit slicing, no input bit slicing, no read noise, and no parasitic resistance.

10.3.4 Performance with parasitic resistance

Table 10.4 shows the GPU runtime for two CIFAR-10 neural networks using different values of the normalized parasitic resistance `Rp`. These simulations assumed: differential cells, ADC enabled, `NrowsMax=1152`, no weight bit slicing, 8-bit activations, no read noise, `noRowParasitics=True` and `interleaved_posneg=False`.

Inference simulations that include parasitic resistance effects are the most computationally expensive simulations in CrossSim, as described in Section 8.3. The performance is dependent on the value of parasitic resistance; the larger the parasitic resistance, the larger the parasitic voltage drops, and more iterations of the circuit simulation are needed to converge on the solution. For the networks in Table 10.4, the accuracy degradation is relatively small when increasing `Rp` from 10^{-5} to 10^{-4}, but is large when increasing `Rp` from 10^{-4} to 10^{-3} [23].

If `Rp` is large, the runtime per image will typically decrease over the course of an inference simulation as more optimal parameters for controlling the convergence of the circuit simulation are found.

Table 10.4: Performance with parasitic resistance using two CIFAR-10 neural networks

Neural network	**# input bit slices**	`Rp`	**GPU runtime**
ResNet14	8	10^{-5}	14.9s / image
ResNet14	8	10^{-4}	17.2s / image
ResNet14	8	10^{-3}	41.1s / image
ResNet56	8	10^{-5}	67.1s / image
ResNet56	8	10^{-4}	79.5s / image
ResNet56	8	10^{-3}	187.1s / image

Acknowledgments

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Bibliography

[1] "MLPerf Inference Benchmark Suite," https://github.com/mlcommons/inference, 2019.

[2] V. Agrawal, V. Prabhakar, K. Ramkumar, L. Hinh, S. Saha, S. Samanta, and R. Kapre, "In-memory computing array using 40nm multibit SONOS achieving 100 TOPS/W energy efficiency for deep neural network edge inference accelerators," in *Intl. Memory Workshop (IMW)*, May 2020.

[3] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019, p. 715–731.

[4] M. Bavandpour, S. Sahay, M. R. Mahmoodi, and D. Strukov, "Efficient mixed-signal neurocomputing via successive integration and rescaling," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 3, pp. 823–827, 2020.

[5] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, March 2016.

[6] P. Chi, S. Li, S. Li, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

[7] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[8] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, and D. Fick, "Analog in-memory subthreshold deep neural network accelerator," in *Custom Integrated Circuits Conf. (CICC)*, May 2017, pp. 1–4.

[9] L. Geiger and P. Team, "Larq: An open-source library for training binarized neural networks," *Journal of Open Source Software*, vol. 5, no. 45, p. 1746, Jan. 2020. [Online]. Available: https://doi.org/10.21105/joss.01746

[10] X. Guo, F. M. Bayat, M. Bavandpour, M. Klachko, M. R. Mahmoodi, M. Prezioso, K. K. Likharev, and D. B. Strukov, "Fast, energy-efficient, robust, and reproducible mixed-signal neuromorphic classifier based on embedded NOR flash memory technology," in *Intl. Electron Devices Meeting (IEDM)*, Dec. 2017, pp. 6.5.1–6.5.4.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2018, pp. 2704–2713.

[13] V. Joshi, M. L. Gallo, S. Haefeli, I. Boybat, S. Nandakumar, C. Piveteau, M. Dazzi, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Accurate deep neural network inference using computational phase-change memory," *Nature Communications*, vol. 11, 2020.

[14] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James, "Multiscale co-design analysis of energy, latency, area, and accuracy of a ReRAM analog neural training accelerator," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, vol. 8, no. 1, pp. 86–101, 2018.

[15] V. Milo, F. Anzalone, C. Zambelli, E. Perez, M. K. Mahadevaiah, O. G. Ossorio, P. Olivo, C. Wenger, and D. Ielmini, "Optimized programming algorithms for multilevel RRAM in hardware neural networks," in *Intl. Reliability Physics Symp. (IRPS)*, 2021, pp. 1–6.

[16] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[17] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf inference benchmark," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2020.

[18] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, "Training deep neural networks for binary communication with the whetstone method," *Nature Machine Intelligence*, vol. 1, no. 2, pp. 86–94, 2019.

[19] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[21] T. P. Xiao, B. Feinberg, C. H. Bennett, V. Agrawal, P. Saxena, V. Prabhakar, K. Ramkumar, H. Medu, V. Raghavan, R. Chettuvetty, S. Agarwal, and M. J. Marinella, "An accurate, error-tolerant, and energy-efficient neural network inference engine based on SONOS analog memory," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14, 2022.

[22] T. P. Xiao, B. Feinberg, C. H. Bennett, V. Prabhakar, P. Saxena, V. Agrawal, S. Agarwal, and M. J. Marinella, "On the accuracy of analog neural network inference accelerators," *arXiv preprint arXiv:2109.01262*, 2021.

[23] T. P. Xiao, B. Feinberg, J. N. Rohan, C. H. Bennett, S. Agarwal, and M. J. Marinella, "Analysis and mitigation of parasitic resistance effects for analog in-memory neural network acceleration," *Semiconductor Science and Technology*, vol. 36, no. 11, p. 114004, 2021.

[24] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse ReRAM engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th Intl. Symp. on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 236–249. [Online]. Available: https://doi.org/10.1145/3307650.3322271

[25] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, no. 7792, pp. 641–646, 2020.