CrossSim:
Crossbar Simulator

Version 0.3

By Sapan Agarwal, Steven J. Plimpton, Richard L. Schiek, Isaac Richter, Alexander H. Hsia, David R. Hughart, Robin B. Jacobs-Gedrim, Conrad D. James, Matthew J. Marinella

Sandia National Laboratories

1 Contents

2 License

CrossSim is licensed under the BSD-3 License:

3 Overview

CrossSim is a crossbar simulator designed to model resistive memory crossbars for both neuromorphic computing and (in a future release) digital memories. It provides a clean python API so that different algorithms can be built upon crossbars while modelling realistic device properties and variability. The crossbar can be modelled using a fast approximate numerical model as well as a slower, but more accurate circuit simulation of the devices using the parallel spice simulator Xyce (to be documented/debugged in a future release).

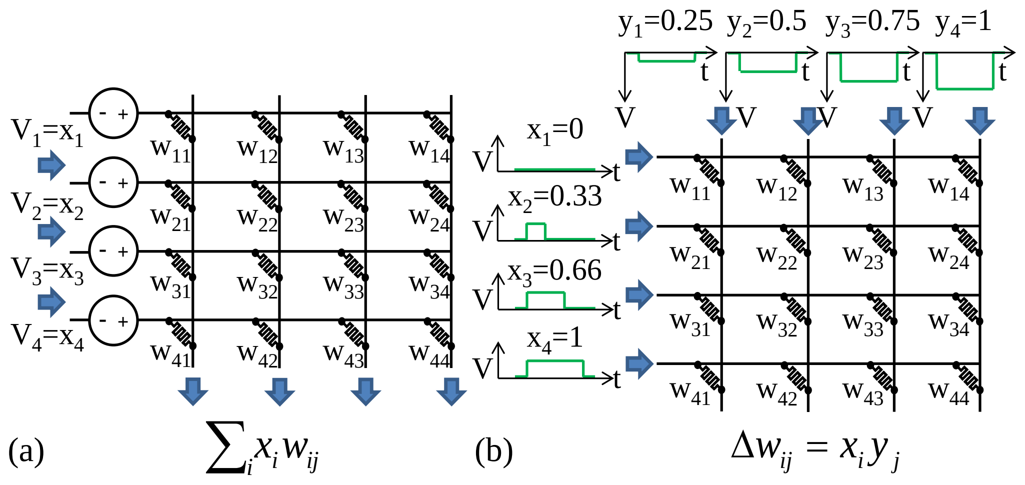
3.1 Neuromorphic Computing Background

Resistive memory crossbars can dramatically reduce the energy required to perform computations in neural algorithms by five orders of magnitude or more when compared to a conventional CPU[1]. For data intensive applications, the computational energy is dominated by moving data between the processor, SRAM (static random access memory), and DRAM (dynamic random access memory)[2]. New approaches based on memristor or resistive memory [3-6] crossbars can enable the processing of large amounts of data by significantly reducing data movement, taking advantage of analog operations [7-11], and fitting more memory on a single chip.

Resistive memories are essentially programmable two terminal resistors. If a write voltage is applied to the device, the resistance will increase or decrease based on the sign of the voltage, allowing the resistance to be programmed. At lower voltages, the state does not change. Consequently, these devices can be used to model a neural synapse wherein the resistance acts like a weight that modulates the voltage applied to it. This has resulted in a large interest in developing neuromorphic systems based on such devices [7-10]. Ideally, the resistive memories would have a perfectly linear and controllable response allowing them to be programmed to any arbitrary analog value. Unfortunately, realistic devices have many non-idealities that need to be modelled in order to model a realistic system.

The key design considerations for an effective neural algorithm accelerator is that it should both reduce the computation energy by orders of magnitude, and it should be flexible enough that it can run many different neural algorithms. In [11] it is shown that a resistive memory crossbar can accelerate two key operations: 1) a parallel read, or vector matrix multiply, and 2) a parallel write or rank one outer product update, as illustrated in Fig. 1. Many neural algorithms such as sparse coding, restricted Boltzmann machines, and backpropagation rely heavily on these two operations. The difference in the implementation of these algorithms is how the inputs and outputs of a crossbar are processed. Consequently our simulator is built around these operations. Our simulator models the crossbar in detail, while providing an API that allows arbitrary "neuron" functions to be written in python that process the inputs and outputs to the crossbar.

An NxN crossbar accelerates O(N^2) operations, while it has O(N) inputs or outputs. This means that the energy to process an input or output can cost O(N) times more than the energy to read or write a single resistive memory element without significantly increasing the system energy. This key insight can be used to optimize the tradeoff between energy efficiency and system flexibility. A crossbar based neural core can be used to perform the parallel vector matrix multiply and outer product update, while a more general purpose digital core can be used to process the inputs and outputs of the crossbar. This is illustrated in Fig. 2. The neural core is illustrated in Fig. 3. The simulator was designed considering this model, but the generalized API can also be used with specialized hardware neuron models.



$V_1=x_1$
$V_2=x_2$
$V_3=x_3$
$V_4=x_4$
w_{11} w_{12} w_{13} w_{14}
w_{21} w_{22} w_{23} w_{24}
w_{31} w_{32} w_{33} w_{34}
w_{41} w_{42} w_{43} w_{44}
(a)
$\sum_i x_i w_{ij}$
$y_1=0.25$ $y_2=0.5$ $y_3=0.75$ $y_4=1$
$x_1=0$
$x_2=0.33$
$x_3=0.66$
$x_4=1$
V
t
(b)
$\Delta w_{ij} = x_i y_j$

Fig. 1. (a) Analog resistive memories can be used to reduce the energy of a vector-matrix multiply. The conductance of each resistive memory device represents a matrix element or weight. Analog input vector values are represented by the input voltages or input pulse lengths, and output vector values are represented by currents. This allows all the read operations, multiplication operations and sum operations to occur in a single step. A conventional architecture must perform these operations sequentially for each weight resulting in a higher energy and delay. A matrix-vector multiply can also be performed by driving the columns and reading the currents on the rows. (b) A parallel write is illustrated. Weight W_{ij} is updated by $x_i \times y_j$. In order to achieve a multiplicative effect the x_i are encoded in time while the y_j are encoded in the height of a voltage pulse. The resistive memory will only train when x_i is nonzero. The height of y_j determines the strength of training when x_i is nonzero. The column inputs y_j can also be encoded in time as in [12].

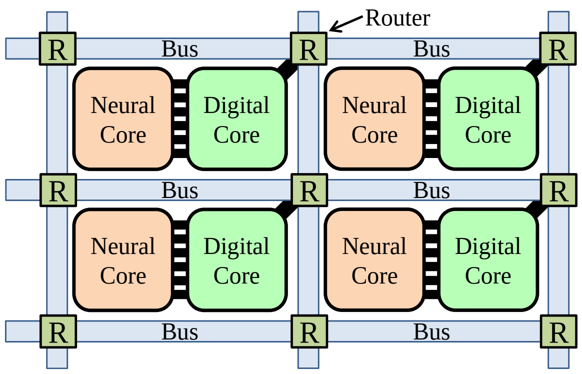
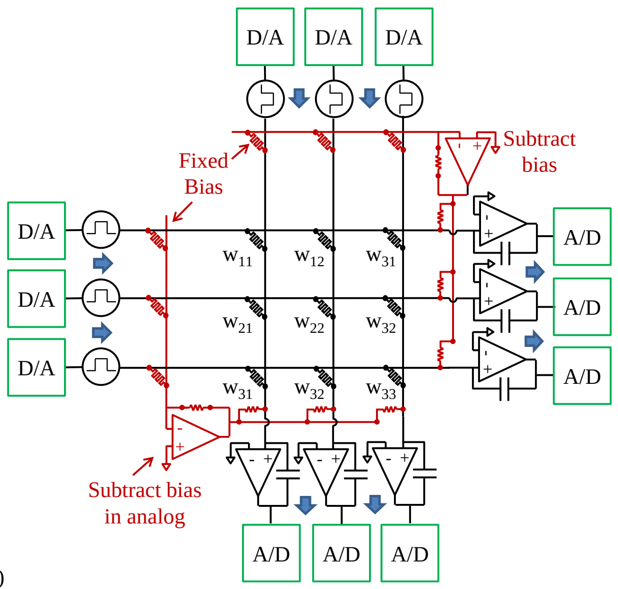
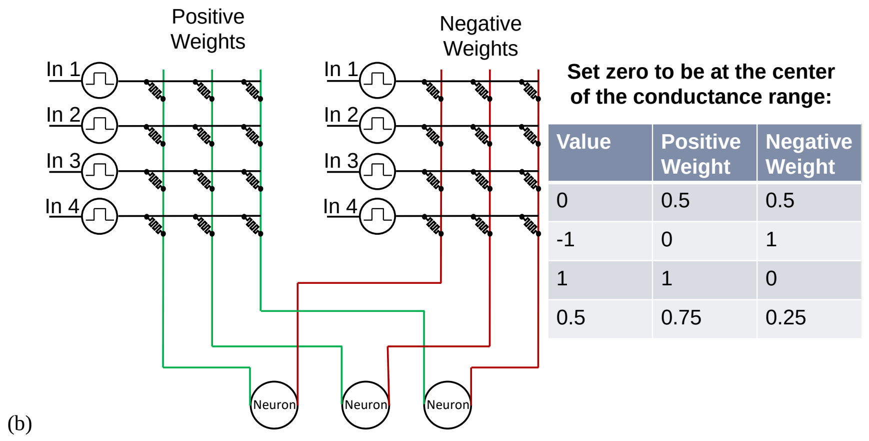


Fig. 2. A general purpose neural architecture is shown. The neural cores only perform parallel vector matrix multiplication, matrix vector multiplication and parallel rank 1 outerproduct updates. For a NxN crossbar the neural core performs $O(N^2)$ operations. The general purpose digital cores process the O(N) inputs/outputs to the neural cores and use the routing network to route data between cores. The flexibility of the digital cores allow for many different algorithms to be implemented, while still taking advantage of the neural cores to accelerate $O(N^2)$ operations. The digital cores can also use digital on-chip resistive memory instruction caches to store slowly changing data while reserving expensive SRAM caches only for the data being processed.





Set zero to be at the center of the conductance range:

Value	Positive Weight	Negative Weight
0	0.5	0.5
-1	0	1
1	1	0
0.5	0.75	0.25

Fig. 3. (a) A neural core is illustrated. A bias row and column are added to the crossbar to allow for negative weights [17]. The rows and columns are driven by either variable length or variable height pulses. The output currents are integrated and then converted to digital using an A/D. (b) Negative weights can also be modelled using a second crossbar to subtract the bias.

4 Installation

CrossSim can be downloaded from http://cross-sim.sandia.gov. It can be installed using pip:

```
>>pip install cross_sim-0.1.0.tar.gz
```

Additionally, examples.tar.gz should be downloaded for useful example scripts on how to use CrossSim.

CrossSim requires Python 3.4 or greater, numpy>1.10, scipy, matplotlib, and mpldatacursor

5 Neural Core API

The crossbar is modelled as a neural core. In order to instantiate a neural core, the parameters defining the neural core need to be set and then the core needs to be instantiated. An example of instantiating a neural core is given in *examples/simple_core_use.py*. To create a neural core the following two objects should be imported from cross_sim:

```
>>from cross_sim import MakeCore
>>from cross_sim import Parameters
```

Next, the default parameters should be loaded:

```
>> params = Parameters()
```

params is the **Parameters** object that specifies all of the simulations settings, noise and non-ideality models that will be used to model the neural core.

The default parameters are stored in *cross_sim/xbar_simulator/parameters/parameter_defaults*. Caution should be used in editing this file as it is not error checked. Parameter settings should be changed as described in Section 6.

After setting parameters in the **params** object as described in Section 6, the neural core should be instantiated:

```
>> neural_core = MakeCore(params=params)
```

Once instantiated, a neural core can be used as the matrix kernel for any neuromorphic algorithm. Each matrix in an algorithm should be instantiated as its own neural core. It stores the weights on a matrix internally and performs noisy updates as well as noisy vector matrix multiplies based on the noise models specified. The functions that can be called are:

```
>>neural_core.set_matrix(weights) # set the initial weights
```

This sets the **weights** to use and creates a crossbar that is the same size as **weights**. The weights are set exactly without noise or non-idealities (subject to clipping limits the limit the weight range)

```
>>neural_core.run_xbar_vmm(vector) # Do a vector matrix multiply
```

This runs a vector matrix multiply using the stored matrix and the input **vector** and returns the result. It applies noise and other non-idealities to the result.

```
>>neural_core.run_xbar_mvm(vector) # Do the transpose, a matrix vector multiplication
```

This runs a matrix vector multiply using the stored matrix and the input **vector** and returns the result. It applies noise and other non-idealities to the result.

```
>>neural_core.update_matrix(row_vector, col_vector, learning_rate=1) # outer product update
```

This runs an outer product update, applying **row_vector** on the rows and **col_vector** on the cols. The updates are clipped and quantized based on the D/A settings in params and the resistive memory noise/non-ideality models are applied to the update. The **learning_rate** scales the size of the update and is applied after the D/A scaling/quantization is applied. The **learning_rate** assumes that the hardware D/A outputs can be efficiently scaled up or down and so D/A scaling & quantization should be applied first and then the learning rate scaling second. If this is not true in a particular system, either row_vector or col_vector should be multiplied by the learning rate.

```
>>neural_core.serial_update(self, matrix, learning_rate =1, by_row=True)
```

This serially adds **matrix** to all the stored weights one row or column at a time. Allows for slow non-neural updates. It assumes the drivers should be maxed out (i.e max out the row driver when applying analog updates to the columns). **by_row**=True applies updates one row at a time (else one column at a time if False). **learning_rate** is defined above. In a neuromorphic algorithm this function should be avoided as it requires serially writing the weights which costs more energy and time.

```
>>neural_core.serial_read(self, by_row=True):
```

Reads the matrix out one row at a time using VMMs (applies the noise/clipping associated with a VMM). **by_row**=True means that the matrix is read out one row at a time using a VMM, while False means the matrix is read out one column at a time using a MVM. In a neuromorphic algorithm this function should be avoided as it requires serially reading the weights which costs more energy and time.

```
>> neural_core._read_matrix()
```

This returns the exact values stored in the matrix and should only be used for debugging. No noise/ non-idealities are applied.

```
>> neural_core.save_weights(filename)
```

Saves the weights in their internal representation to **filename**

```
>> neural_core.load_weights(filename)
```

Loads weights from **filename**

6 Simulation Models & Parameter Settings

All the simulation models can be customized by changing the parameter settings passed to **MakeCore()**. After creating a params object with default settings using the following statement, the simulation models can be customized by changing the defaults.

```
>> params = Parameters()
```

The settings are grouped into 6 different categories:

- params.algorithm_params
  - These settings control how the neural core interacts with external algorithm wrapped around it, i.e., what are the fixed point ranges accepted by the neural core. It also sets the general properties of the neural core.
- params.xbar_params
  - These are the properties of the crossbar A/D and the physical scaling range of the resistive memory
- params.numeric_params
  - These parameters are the properties of the numeric crossbar model
- params.periodic_carry_params
  - These parameters configure periodic carry, if used
- params.xyce_params
  - These are the properties required to configure the xyce crossbar model
- params.memory_params
  - These are the properties required to configure the memory core model
- params.analytics_params

- These are settings for turning on analytics such as saving weights after each update

There is also an entirely internal set of parameters that are derived from the above settings and used internally to help scale from the **algorithm_params** to the **xbar_params**:

- params.wrapper_params (do not use)

6.1 Simulation Type

```
>>params.algorithm_params.weights.crossbar_type = “OFFSET”
```

This specifies the crossbar model to use. Setting it to **“OFFSET”** specifies that negative numbers will be represented by subtracting a bias in the center of the conductance range as illustrated in Fig 3(a). Setting it to **“BALANCED”** will use two crossbars to represent negative numbers. The difference between two weights specifies the final weight. Updates are performed by applying half the update to the positive crossbar and minus half the update to the negative crossbar. Setting it to **“MEMORY”** will use the crossbar as a memory core (to be documented in the future).

```
>>params.algorithm_params.weights.sim_type = “NUMERIC”
```

This specifies the inner model of the simulation and should be set to either **“NUMERIC”** or **“XYCE”** specifying whether the simulation will use a numeric approximation to the circuit or whether an actual circuit simulation in xyce will be run for each neural core operation

6.2 Scaling, A/D and D/A properties

An analog crossbar performs computations in fixed point. The weight range that an external algorithm uses is different from the conductance range stored on the crossbar. Consequently, we need to scale and unscale the weights and vectors for all interactions with an algorithm. To do this we need to specify all the input and output ranges that the algorithm needs. The simulator will then scale to ranges the hardware can use. The scaling ranges need to be specified as follows. In general the min values should be equal and opposite the max values unless otherwise specified. Values beyond the max/min will be clipped to the max/min.

All scaling / clipping can be ignored by setting the following parameter. Ideally, this should only be used for debugging or for comparing to an ideal floating point simulation:

```
>>    params.algorithm_params.disable_clipping = True
```

6.3 Algorithm clipping limits

In general the following limits should be set based on the algorithm being run

```
>>params.algorithm_params.weights.maximum = 10
>>params.algorithm_params.weights.minimum = -10
```

These specify maximum and minimum values that the weights can take. This sets how the weights will be scaled to a physical conductance. When using the offset core the minimum weight can be zero, allowing for only positive weights.

```
>>params.algorithm_params.col_input.maximum = 1.0
>>params.algorithm_params.col_input.minimum = -1.0
```

This specifies the column input range for matrix vector multiplies

```
>>params.algorithm_params.row_input.maximum = 1.0
```

```
>>params.algorithm_params.row_input.minimum = -1.0
```

This specifies the row input range for vector matrix multiplies

```
>>params.algorithm_params. row_output.maximum = 20
>>params.algorithm_params. row_output.minimum = -20
```

This specifies the output range for the result of a matrix vector multiply. The output of a crossbar is fed into an op-amp that saturates at some output value. Consequently, there is a max/min output.

```
>>params.algorithm_params. col_output.maximum = 20
>>params.algorithm_params. col_output.minimum = -20
```

This specifies the column output range for vector matrix multiplies

```
>>params.algorithm_params. col_update.maximum = 5
>>params.algorithm_params. col_update.minimum = -5
>>params.algorithm_params. row_update.maximum = 5
>>params.algorithm_params. row_update.minimum = -5
```

These specify the limits of the row and column update. The max row update × the max col update gives the maximum update size in one cycle.

```
>>params.algorithm_params.serial_read_scaling = 1
```

When doing a serial read, the output of the integrator is scaled by this value before being passed to the A/D and then unscaled after the A/D. We may need a different output range during a parallel read than a serial read, as only one device is being read during a serial read. This option allows for that. Physically, this could represent changing the capacitor used to integrate the output or scaling the length or voltage of a read pulse.

6.4 Crossbar Weight Limits

The resistive memories typically have a finite on/off ratio. Modelling this can affect some noise models. This is modelled by setting the crossbar weight range as follows:

```
>>params.xbar_params.weights.maximum = 1
>>params.xbar_params.weights.minimum =  0.1
```

In the ideal case, max=1 and min=0. Setting the min to a positive non-zero value results in a finite on/off ratio of max/min. If these are changed the following parameters should be changed to match:

```
>>params.xbar_params.weight_clipping.maximum = 1
>>params.xbar_params.weight_clipping.minimum =  0.1
```

The first set of values, **xbar_params.weights**, represents the target range for scaling values between the crossbar and the algorithm. The second set of values, **xbar_params.weight_clipping**, represents a potentially larger clipping window that allows the weights to exceed the target range. This is useful for experimentally characterized devices where the device noise and nonlinearity is characterized over a wide range of conductance, but we may only want to use a narrow range of conductance where the properties are favorable. In this case, **xbar_params.weights** should be set to use the target range while **xbar_params.weight_clipping** should be set to the larger clipping range where the weights are clipped to the max/min.

6.5 A/D and D/A properties

A key part of the neural core is that the inputs and outputs are converted between digital and analog. By default, it is assumed that the conversion is performed ideally, with arbitrary precision and accuracy. For each of the 6 interfaces to the core (**row_input, col_input, row_output, col_output, row_update and col_update**) it is possible to set a finite bit precision as well as different noise models that can be applied. The models are shown for the **row_input**, but can be applied to any of the 6 interfaces. The settings are set under the **xbar_params** as the A/D and D/A properties are properties of the crossbar.

6.5.1 Bit Precision

```
>>params.xbar_params.row_input.bits = 0
```

This sets the number of bits that the input is quantized to. Zero means no quantization / arbitrary precision.

```
>>params.xbar_params.row_input.sign_bit = True
```

This determines whether a sign bit should also be used. The quantization model with and without a sign bit is illustrated in Fig 4. The total number of bits is **bits**+**sign_bit**.

```
>>params.xbar_params.row_input.stochastic_rounding = False
```

If False, the values are quantized by rounding to the nearest value as illustrated in Fig 4. If True, values are stochastically rounded to the nearest value and the exact value determines the probability. For instance, 9.2 will be rounded to nine 80% of the time and rounded to ten 20% of the time. This is useful for training as information from the average value can be learned over many training epochs.

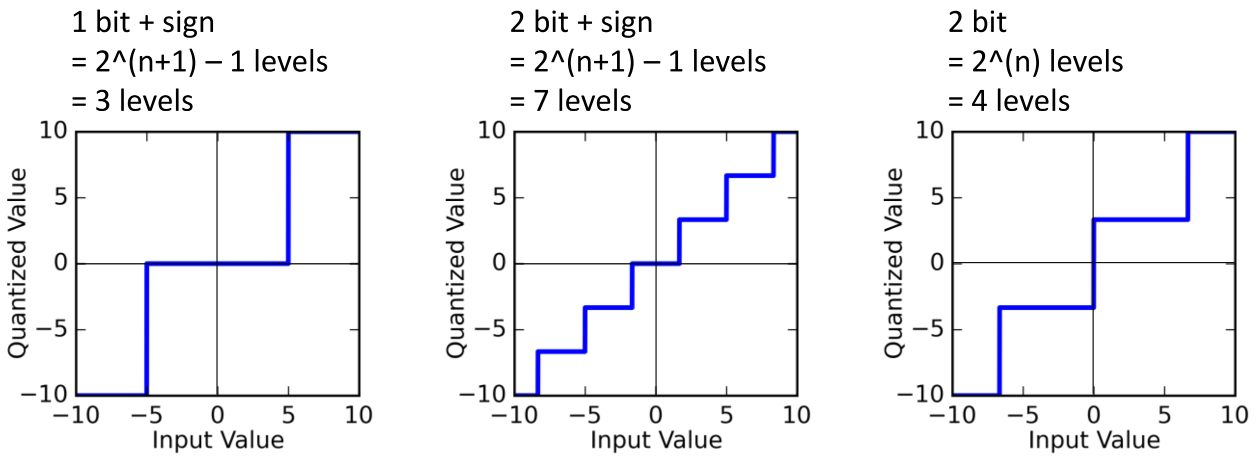


Fig. 4: Quantization models for (a) 1 bit with a sign bit and (b) 2 bits with a sign bit, (c) 2 bits without a sign bit.

6.5.2 Gaussian noise applied after quantization

```
>>params.xbar_params.row_input.normal_error_post.sigma = 0.0
```

This model adds Gaussian noise after the quantization is applied. The noise sigma is defined relative to the full weight range such that:

new value = old value + N(sigma×range)

```
>>params.xbar_params.row_input.normal_error_post.keep_within_range = True
```

Applying noise can cause the values to exceed the clipping ranges set above. Setting this to True (default= True) causes the final value to be clipped to the clipping ranges, False allows the value to exceed the clipping ranges.

```
>>params.xbar_params.row_input.normal_error_post.proportional = False
```

This changes the noise model to a model where the noise is proportional to the value:
False: new value = old value + N(sigma × range)
True: new value = old value × [1+N(sigma)]

6.5.3 Gaussian noise applied before quantization

This noise model, **normal_error_pre**, is the same as above, but is applied before clipping or quantizing the values

```
>>params.xbar_params.row_input.normal_error_pre.keep_within_range = True
>>params.xbar_params.row_input.normal_error_pre.proportional = False
>>params.xbar_params.row_input.normal_error_pre.sigma = 0.0
```

6.5.4 Uniform noise applied after quantization

This is a uniform noise applied after quantization

```
>>params.xbar_params.row_input.uniform_error_post.range = 0.0
```

Setting the range to a nonzero value turns the model on. The range is defined as a fraction of whole range

```
>>params.xbar_params.row_input.uniform_error_post.keep_within_range = True
```

Clips the resulting value to the clipping constraints

6.6 Numeric Matrix Read Noise Model

Each time the matrix is read (VMM or MVM), a Gaussian read noise can be applied to each weight. The noise parameters are defined the same as above for normal_error_post. This model only applies to a numeric crossbar model. The settings are:

```
>>params.numeric_params.read_noise.sigma = 0
>>params.numeric_params.read_noise.proportional = False
>>params.numeric_params.read_noise.keep_within_range = True
```

6.7 Numeric Matrix Update Model

When updating the weights there are two numeric models that can be used: 1) and analytic update model and 2) an experimental lookup table model. The models are selected as follows:

```
>>params.numeric_params.update_model = "ANALYTIC"
```

This model uses the analytic nonlinearity and write noise models in [13] and is described in Section 6.7.2.

```
>>params.numeric_params.update_model = "DG_LOOKUP"
```

This model uses a lookup table based on experimental ΔG vs G plots as described in Section 6.7.1.

6.7.1 Lookup Table Update Model

6.7.1.1 Model Overview

This model takes a experimentally measured ΔG vs G plot as shown if Fig. 5 and uses that to compute the noisy updates. This model is explained in detail in the supplementary information of [14].

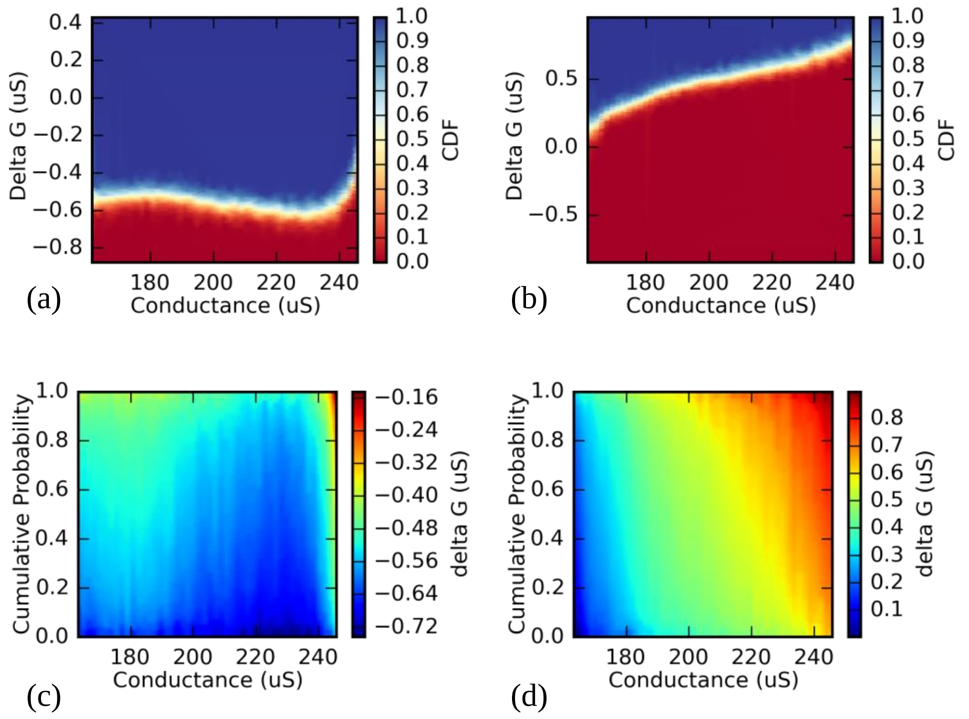


Fig. 5: The ΔG vs G response for (a) decreasing and (b) increasing pulses of a voltage controlled Lithium Ion Synaptic Transistor [14] is shown. In the simulator, the data is represented the cumulative probability vs conductance as shown in (c) and (d) for decreasing and increasing pulses respectively.

Using the experimental ΔG vs G plots, Figs 5(a) and 5(b), we need to determine how much the mean update will be and how much noise to add to that update given the update requested by the algorithm. Consider Fig. 5(b), This plot is generated for only a single pulse size that has a mean update around 0.5 µS, with the exact mean value slightly higher or lower depending on the initial conductance, G_o. For a particular training example, a different update, perhaps 0.1 µS may be desired. To find the update, first we see what state, G_o, the device is in, find the average update at G_o from Fig 5(b) and sample a noise value from the probability distribution at G_o in Fig 5(b). Then we scale the mean value and the noise value according to the desired 0.1 µS update. For instance, if G_o=170 µS, the mean update is 0.3 µS. Sampling from the probability distribution at G_o, we might get an update of 0.35 µS, giving a write noise of 0.05 µS that was added to the mean update of 0.3 µS.

Since all the updates are done in parallel, we can only use a single scaling factor for all the updates. The average update over the conductance range is 0.53 µS, while we wanted an update of 0.1 µS. Consequently, the mean update of 0.3 µS at G_o=170 µS should be scaled by 0.1/0.53 giving a new mean update of 0.057 µS. (The average update of 0.53 µS was only averaged over the center half of the conductance range, 180 µS to 220 µS, as the system was designed to have most of the weights be in the center half of Fig 5b).

The noise cannot simply be scaled by the same scale factor. The noise increases relative to the size of the update as the update gets smaller. This can be understood by thinking of the noise as a random walk. If we use one pulse or two pulses to change the conductance from one state to another, the noise will be the same as long as the average initial and final states are the same. This implies that one longer or higher voltage pulse will cause the same physical change as two sequential pulses that end at the same average conductance. Since the noise is the same for a given ΔG regardless of the number of pulses required to get a given ΔG, ΔG must be proportional to the variance of the noise, σ^2. After multiple pulses, the variance of the noise in each pulse is additive. Therefore, $\sigma \propto \sqrt{\Delta G}$. This means that the noise should be multiplied by the square root of the scaling factor. Thus the 0.05 µS of noise is scaled by $\sqrt{0.1/0.53}$ giving a write noise of 0.022 µS. Thus the final update is the mean update of 0.057 µS + the write noise 0.022 µS, giving an update of 0.78 µS for the requested 0.1 µS updated.

6.7.1.2 Model Usage

The experimentally derived lookup table model is used when **numeric_params.update_model = "DG_LOOKUP"**. In order to use this model, a file containing the experimental lookup table should be supplied as follows:

```
>>params.numeric_params.dG_lookup.file_increasing="TaOx"
>>params.numeric_params.dG_lookup.file_decreasing="TaOx"
```

These statements set the file locations of the lookup tables. The path to a user defined file can be specified or one of the internally stored lookup tables can be specified. The format for a user defined file is described below. The included data sources are listed in the next section.

```
>>params.numeric_params.dG_lookup.Gmin_relative = 0.25
>>params.numeric_params.dG_lookup.Gmax_relative = 0.75
```

The lookup table defines a large conductance range, but using only part of it may give better results. **Gmin_relative** and **Gmax_relative** define the fraction of the lookup table (25% to 75% of the range) to target using. Weights can go outside the targeted range, but the weight scaling will be based on the targeted range. Based on these values, **xbar_params.weights**, and **xbar_params.weight_clipping** will be set when the table is loaded, overwriting any previous settings.

```
>>params.numeric_params.dG_lookup.disable_nonlinearity = False
```

If True, the nonlinearity is ignored and only the write noise is used. This represents a calibrated update where the amount written is based on the current state (This means the nonlinearity still impacts the level of noise as the noise is dependent on the size of the applied update).

```
>>params.numeric_params.dG_lookup.disable_writenoise = False
```

If True, the write noise is ignored and only the nonlinearity is used. This is to investigate the impact of nonlinearity vs noise.

The datafile, which is plotted in Figs 5(c) and 5(d), should be a CSV file formatted as follows:

- 1st row: title text
- 2nd row: conductance bins
- 3rd row: discretized CDF bins
- 4th row onwards: dG Matrix (rows = CDF, Columns = conductance bins)

  If the conductance and CDF bins have a uniform spacing, interpolating from the table will run faster, speeding up the simulation significantly. The example script **create_lookup_table.py** gives an example of how to create a lookup table from raw measurement data.

6.7.1.3 Included Data

CrossSim comes with data from several different sources. These are stored under cross_sim/data/lookup_tables. If the datasets are used, please cite the appropriate data sources listed below. The included datasets are:

LISTA_current

This is data from the Lithium Ion Synaptic Transistor from Ref [14] when measured using current controlled pulses. Current control allows for lower write non-linearity and thus better performance.

LISTA_voltage

This is data from the Lithium Ion Synaptic Transistor from Ref [14]when measured using voltage controlled pulses. The data is plotted in Fig 5.

TaOx

This is data from TaOx based resistive memories from Ref [15]. It was taken using 100ns read pulses.

ENODe

This is data from the electrochemical neuromorphic organic device (ENODe) from [16].

6.7.2 Analytic Update Model

6.7.2.1 Analytic Write Noise Model

A guassian write noise is added to the update. The standard deviation of the noise can follow one of three models, which are compared in [13]. For all three models it is assumed that the noise is proportional to the sqrt(ΔG) so that the variance between multiple updates adds, following a random walk.

```
>>params.numeric_params.write_noise.sigma = 0
```

The normalized standard deviation of the noise is set above. This is converted into an absolute standard deviation based on the model set below. Zero is no noise, disabling the model. The resulting value is automatically clipped to the weight limits.

```
>>params.numeric_params.write_noise.write_noise_model = "G_INDEPENDENT"
```

There are three possible models as defined below (sigma_wn = parameter set above)

"G_INDEPENDENT" # write noise independent of the conductance sigma = sqrt(delta_G × range) × sigma_wn

"G_PROPORTIONAL" # write noise proportional to the current conductance

sigma = sqrt(delta_G × range) × G/range × sigma_wn

"G_INVERSE" # write noise inversely proportional to the current conductance

sigma = sqrt(delta_G × range) × range/G × sigma_wn

6.7.2.2 Analytic Nonlinear Update Model

The size of the update depends on the current state. This dependence can be modelled using a either an asymmetric or a symmetric nonlinearity model. The details of both models are described in detail in [13]. The model can be used by setting the following:

```
>>params.numeric_params.nonlinearity.alpha = 0
```

Alpha is the strength of the nonlinearity. Zero means no nonlinearity, and a larger number is a worse nonlinearity

```
>>params.numeric_params.nonlinearity.symmetric = False
```

True means that a symmetric nonlinearity model will be used and false means an asymmetric nonlinearity model will be used.

6.8 Periodic Carry

6.8.1 Model Overview

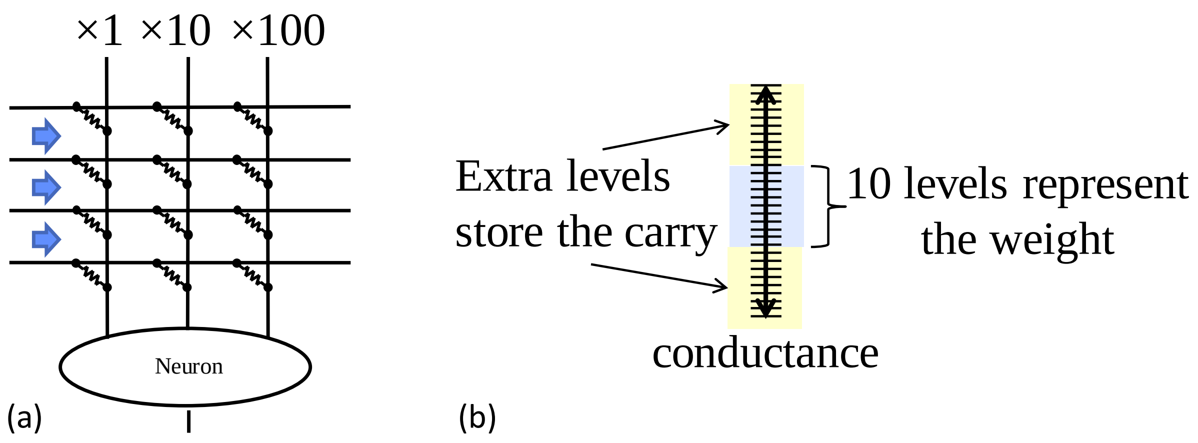


Fig. 6 (a) In a positional number system each column represents a different digit. (b) In order to enable blind updates, extra levels are used to store a carry which is periodically read and used to update the higher order bits.

In order to compensate for noisy nonlinear devices a novel periodic carry method that uses a positional number system can be used to overcome device limitations while maintaining the benefit of parallel analog matrix operations as discussed in [15]. Multiple devices can be used to add additional levels. Adding devices in parallel to a single synapse increases the number of levels linearly with the number of devices.

Positional number systems, such as a base 2 or base 10 can increase the number of levels exponentially with the number of devices as illustrated in Fig 6a, greatly enhancing the information storage capacity of each synapse. Unfortunately, this is not compatible with the parallel outer product update in Fig 1b. When updating a weight, carries need to be performed between digits: i.e. adding 1 to 9 requires resetting the resistive memory that holds 9 to 0 and adding 1 to the next digit. Doing this would require serially reading out and updating the array, eliminating the benefit of the parallel analog computation. We propose a new periodic carry method to solve this problem, illustrated in Fig 6b. Each device uses a portion of its dynamic range to store carry information. Periodically (every 100-1000 updates) the device is read, and if the device is in the carry range, the device is reset to zero and the next bit is updated. This averages out the carry cost, allowing for parallel writes while obtaining significant increase in bits per weight.

6.8.2 Model Usage

>>params.periodic_carry_params.use_periodic_carry = False

Set to True to turn on periodic carry

>> cores_per_weight=3

This specifies the number of cores/digits used to represent the weights

>>number_base = 5

This specifies the number system used between ReRAMs representing a weight (i.e base 5 or base 10 number system). Each weight is number_base times larger than the previous.

>>carry_threshold=0.5

At what fraction of the total weight range (including carry) should a carry be performed (weights less than this fraction are not carried)

>>normalized_output_scale=10

When doing a VMM or MVM on lower order bits, the output scale of the VMM result on lower order bits needs to be different from the overall output scale corresponding to the smaller weight range on a given lower order bit. The scale is set by: input range* weight range* **normalized_output_scale**

>>read_low_order_bits = True

Include lower order bits when reading. When False, only the highest order bit will be used during reads. When true all bits will be used, but there could be errors due to bits saturating depending on how **normalized_output_scale** is set.

>>carry_frequency' : [10,100]

a list of relative carry frequencies to perform the carry on each weight, starting from the second highest order bit to the lowest. The number of carries = **cores_per_weight** - 1

>>exact_carries' : False

If True, carries are computed exactly (no read noise / nonlinearity etc). This implies ideal carries with either write verify or ideal digital carries. If false, read and write noise are applied when computing carriers

>>zero_reset = “CALIBRATED”

How to reset the ReRAMs after a carry. There are three options:

"**EXACT**" - the ReRAMs are reset to zero exactly (implies write verify) after a carry

"**CALIBRATED**" - use calibrated updates based on the current state, i.e. write noise only, no nonlinearity (only implemented w/ lookup table model)

"**BLIND**" - use blind updates based on the average response, does not work well

6.9 Neural Core Analytics

Currently a single analytic is implemented. The neural core can save the weights after each update so that the weight history can be analyzed. The settings for this are given by:

```
>>params.analytics_params.store_weights = False
```

If this is set to True, the neural core will store all the weights after each update

```
>>params.analytics_params.max_storage_cycles = 10000
```

In order to limit memory usage, weights are stored for a finite number of updates, up to **max_storage_cycles**

```
>>params.analytics_params.all_weights = True
```

If True store all the weights in the matrix, if False, only store the weights specified as follows

```
>>params.analytics_params.weight_rows = (0,1,2)
>>params.analytics_params.weight_cols = (0,1,2)
```

If **all_weights** = False, only store the weights with row indicies specified by **weight_rows** and col indicies specified by **weight_cols**

If store_weights =True, the weights are stored in the following list

```
>> neural_core.weights_list
```

And the number of updates is stored here

```
>> neural_core.update_ctr
```

6.10 Xyce Model Settings

To be added in the future

7 Backpropagation

An example file of how to run backpropagation is in examples/train_neural_net.py. This will be documented better in the future.

This example file sets all the neural core parameters based on the simulation model being used (which is set at the bottom of the file under "if __name__ ==’__main__’ ". If an ideal numeric simulation is run, a different simplified neural core model is used. Consequently, the backprop model has two neural core models: ncore.py and ncore_new.py. Only ncore_new.py uses CrossSim described above.

7.1 Training Data

For convenience, several backpropagation training datasets are included in *cross_sim/data/backprop*. If they are used, please cite them. There are:

file_types:

file_types is a Sandia file classification dataset from [17] It is 256 byte-pair statistical attributes to classify 9 file types.

iris:

iris is the iris flower dataset from the UCI machine learning database:

https://archive.ics.uci.edu/ml/datasets/Iris/

mnist

mnist is the MNIST handwritten digits dataset [18]

small_images

small_images is an 8x8 version of MNIST from:

https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

8 Acknowledgements

Sandia National Laboratories is 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.

9 References

[1] T. M. Taha, R. Hasan, C. Yakopcic, and M. R. McLean, "Exploring the design space of specialized multicore neural processors," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, 2013, pp. 1-8.

[2] P. Kogge *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[3] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature,* 10.1038/nature06932 vol. 453, no. 7191, pp. 80-83, 2008.

[4] L. O. Chua, "Memristor-The missing circuit element," *Circuit Theory, IEEE Transactions on,* vol. 18, no. 5, pp. 507-519, 1971.

[5] R. Waser and M. Aono, "Nanoionics-based resistive switching memories," *Nat Mater,* 10.1038/nmat2023 vol. 6, no. 11, pp. 833-840, 2007.

[6] K.-H. Kim *et al.*, "A Functional Hybrid Memristor Crossbar-Array/CMOS System for Data Storage and Neuromorphic Applications," *Nano Letters,* vol. 12, no. 1, pp. 389-395, 2012/01/11 2012.

[7] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale Memristor Device as Synapse in Neuromorphic Systems," *Nano Letters*, vol. 10, no. 4, pp. 1297-1301, 2010/04/14 2010.

[8] C. Ting, Y. Yuchao, and L. Wei, "Building Neuromorphic Circuits with Memristive Devices," *Circuits and Systems Magazine, IEEE,* vol. 13, no. 2, pp. 56-73, 2013.

[9] R. Hasan and T. M. Taha, "Enabling back propagation training of memristor crossbar neuromorphic processors," in *Neural Networks (IJCNN), 2014 International Joint Conference on*, 2014, pp. 21-28.

[10] Y. Kim, Y. Zhang, and P. Li, "A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing," *ACM Journal on Emerging Technologies in Computing Systems (JETC),* vol. 11, no. 4, p. 38, 2015.

[11] S. Agarwal *et al.*, "Energy Scaling Advantages of Resistive Memory Crossbar Based Computation and its Application to Sparse Coding," *Frontiers in Neuroscience,* vol. 9, p. 484, 2016, Art. no. 484.

[12] D. Kadetotad *et al.*, "Parallel Architecture With Resistive Crosspoint Array for Dictionary Learning Acceleration," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on,* vol. 5, no. 2, pp. 194-204, 2015.

[13] S. Agarwal *et al.*, "Resistive memory device requirements for a neural algorithm accelerator," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 929-938.

[14] E. J. Fuller *et al.*, "Li-Ion Synaptic Transistor for Low Power Analog Computing," *Advanced Materials,* vol. 29, no. 4, p. 1604310, 2017.

[15] S. Agarwal *et al.*, "Achieving Ideal Accuracies in Analog Neuromorphic Computing Using Periodic Carry," in *(Accepted) 2017 IEEE Symposium on VLSI Technology* Kyoto, Japan, 2017.

[16] Y. van de Burgt *et al.*, "A non-volatile organic electrochemical device as a low-voltage artificial synapse for neuromorphic computing," *Nat Mater,* Letter vol. 16, no. 4, pp. 414-418, 2017.

[17] J. A. Cox, C. D. James, and J. B. Aimone, "A Signal Processing Approach for Cyber Data Classification with Deep Neural Networks," *Procedia Computer Science,* vol. 61, pp. 349-354, // 2015.

[18] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits [Online]. Available: http://yann.lecun.com/exdb/mnist