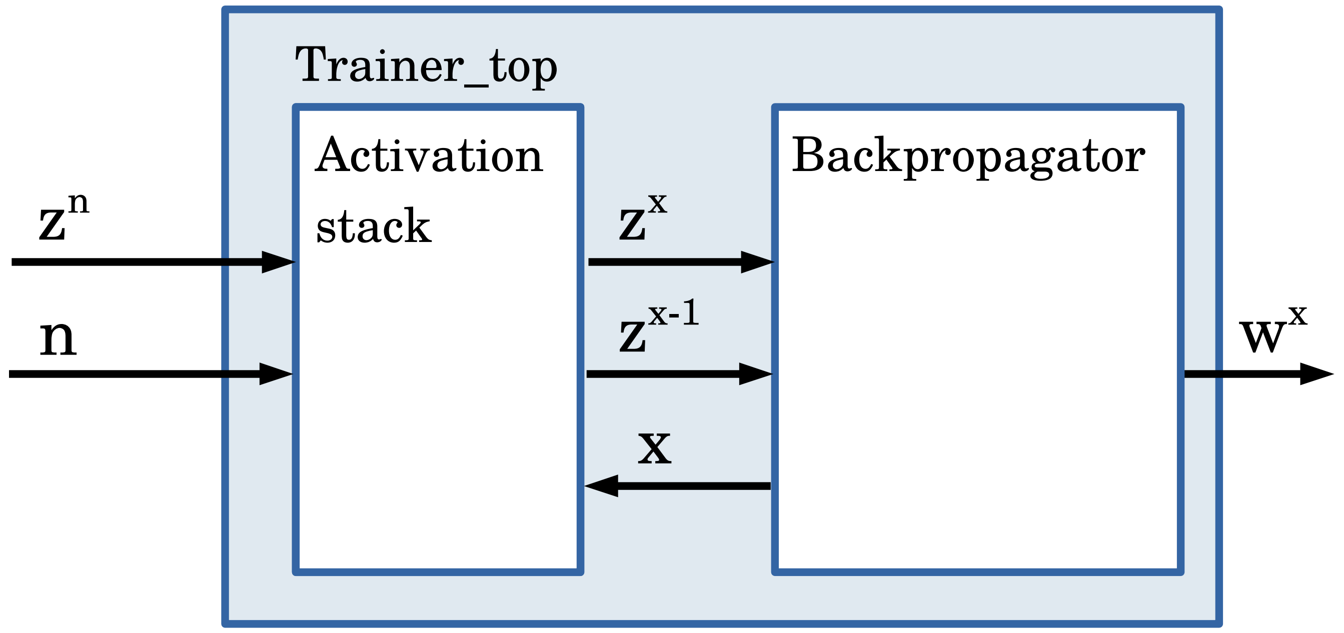
Modules and progress

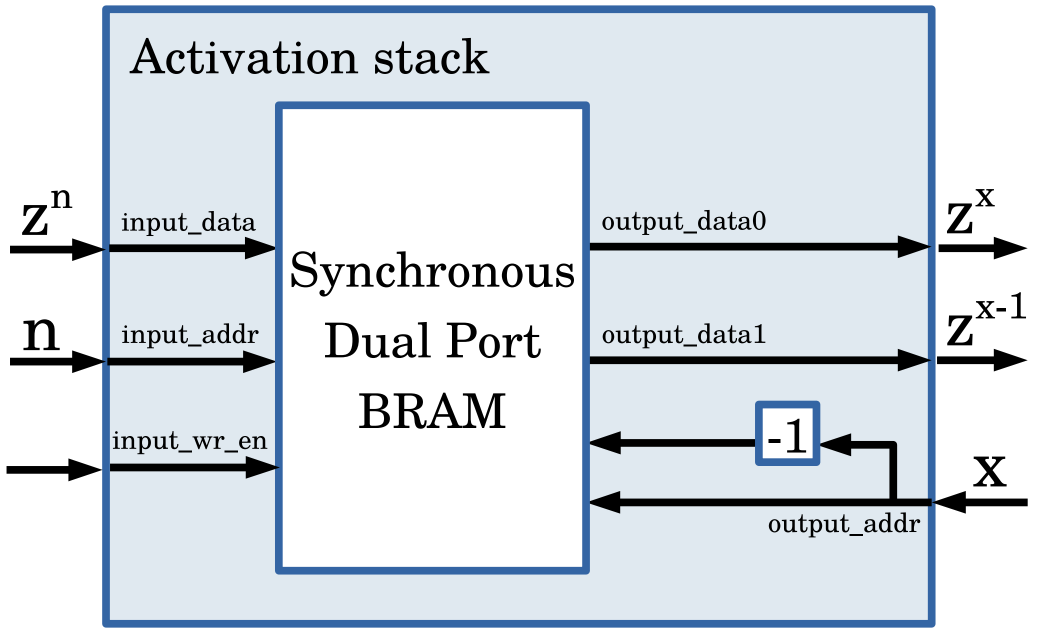
Trainer_top	Refactor
Activation_stack	Done
Dual port BRAM	Done
Backpropagator	Refactor
Weight controller	Implemented in backpropagator
Weight loader	Done
Weight updater	Done
Tiled tensor product	Done
Tiled vector adder	Done
Error controller	Implemented in backpropagator
Error fetcher	Done
Vector dot	Done
Activation LUT	Done
Vector subtract	Done
Error propagator	Done, could use more testing
Matrix vector multiplication	Done
Vector dot	Done
Matrix transpose	Done

Trainer top



The trainer top module receives activations from the forward propagation module, and returns adjusted weight matrices.
The activations are saved in the stack and fed to the backpropagator module which holds the weights.

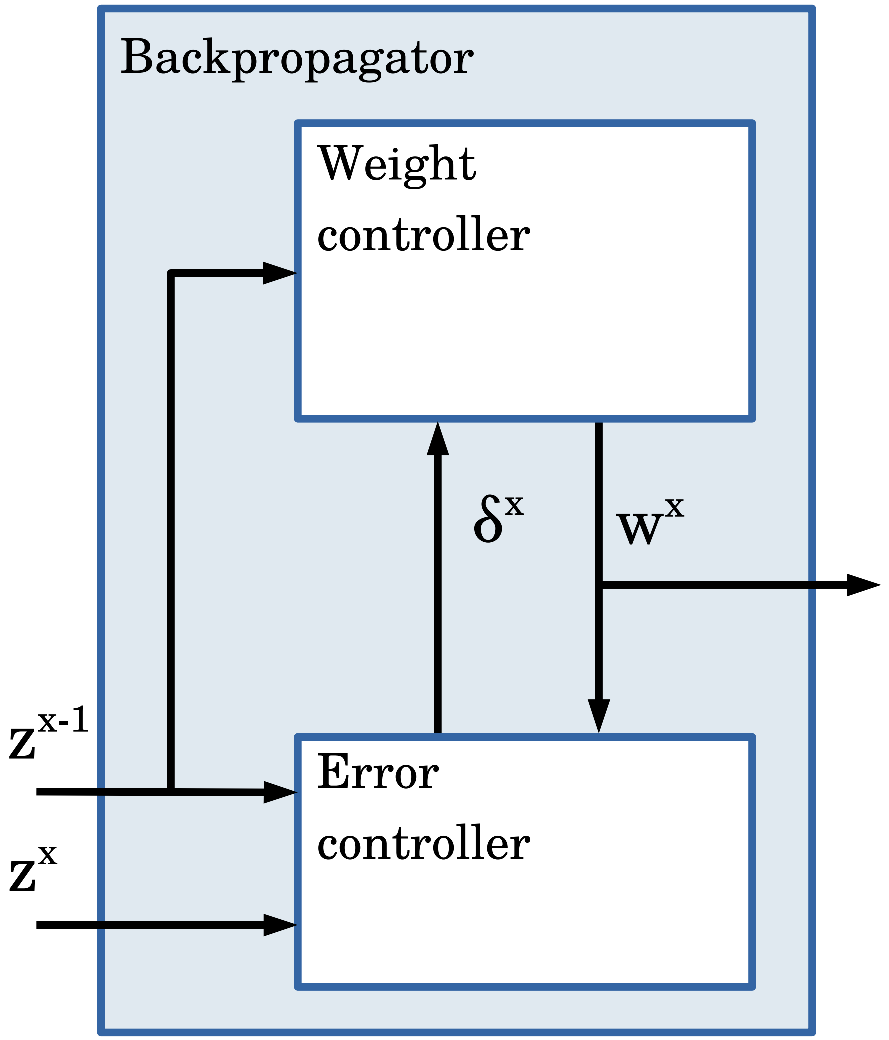
Activation stack



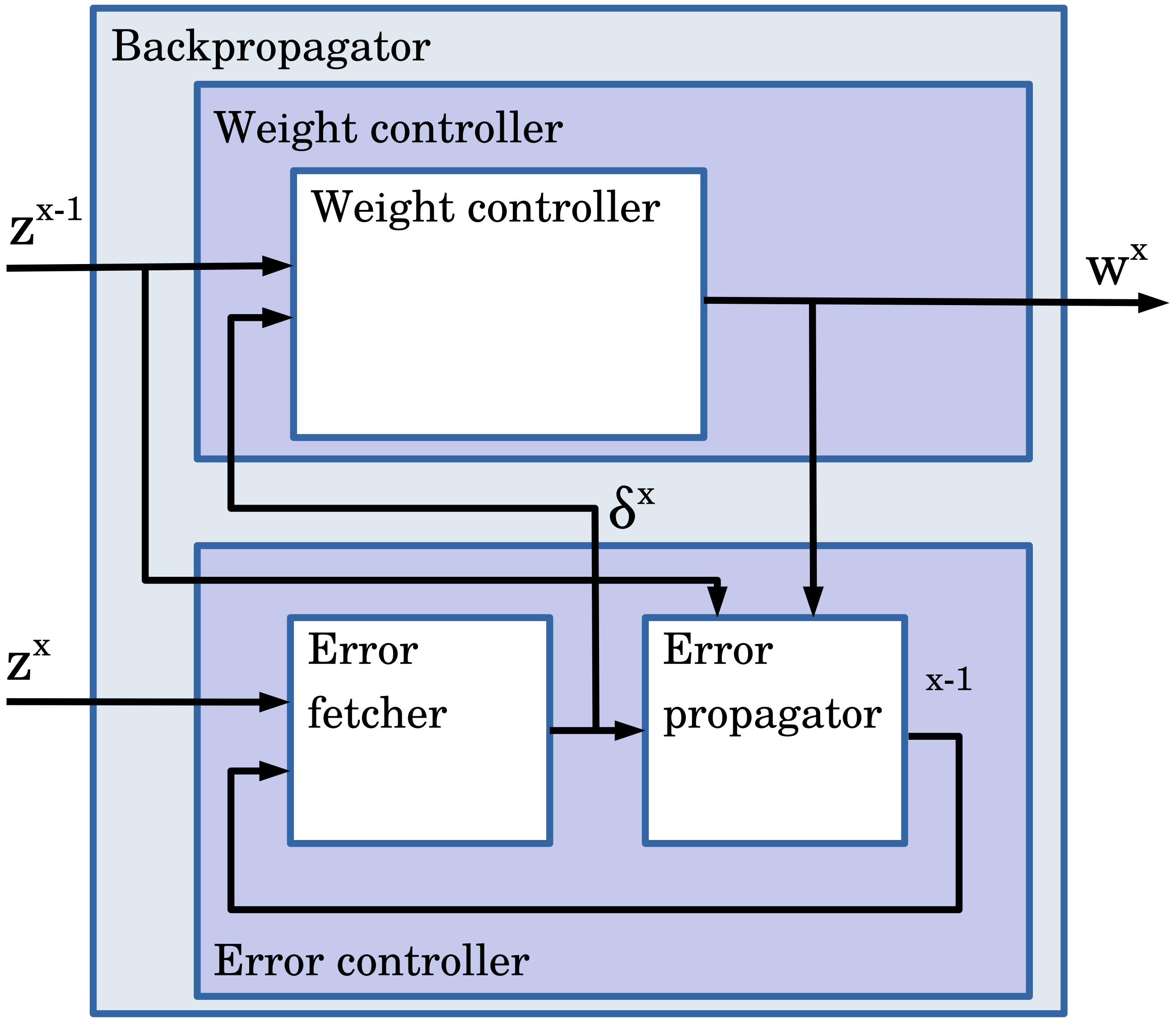
Activation stack is a parametrizable model for storing NN activations with one data and address input, and two outputs.
The data is received from the forward propagation module, and is read by the backpropagation module.
The two read buses read consecutive layer activations starting from the output layer down.

NEURON_NUM parameter is the number of cells of input_data and output_data.
ACTIVATION_WIDTH is the width of each of the NEURON_NUM activations.
STACK_ADDR_WIDTH is the size of the addresses sent to the stack.

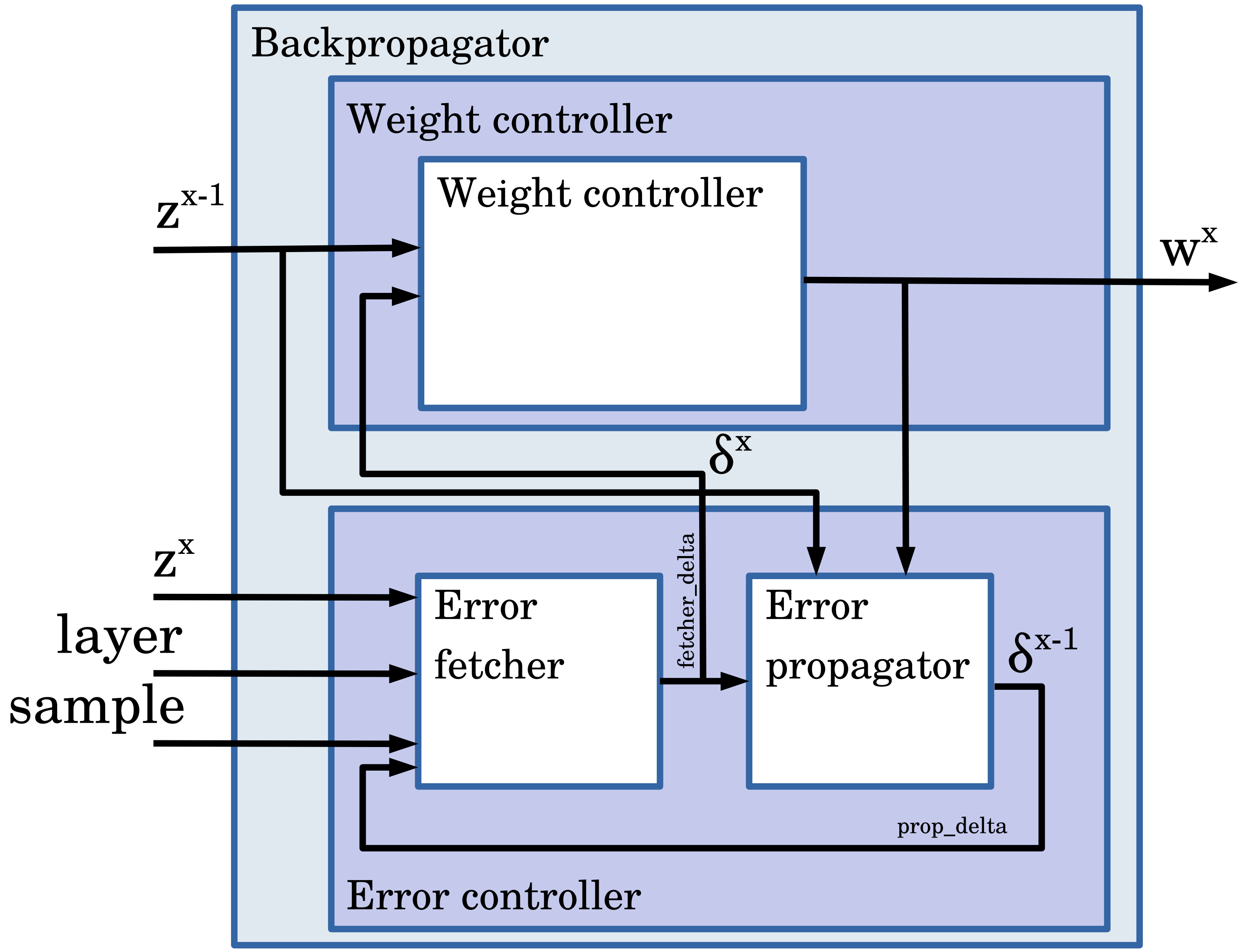
Backpropagator



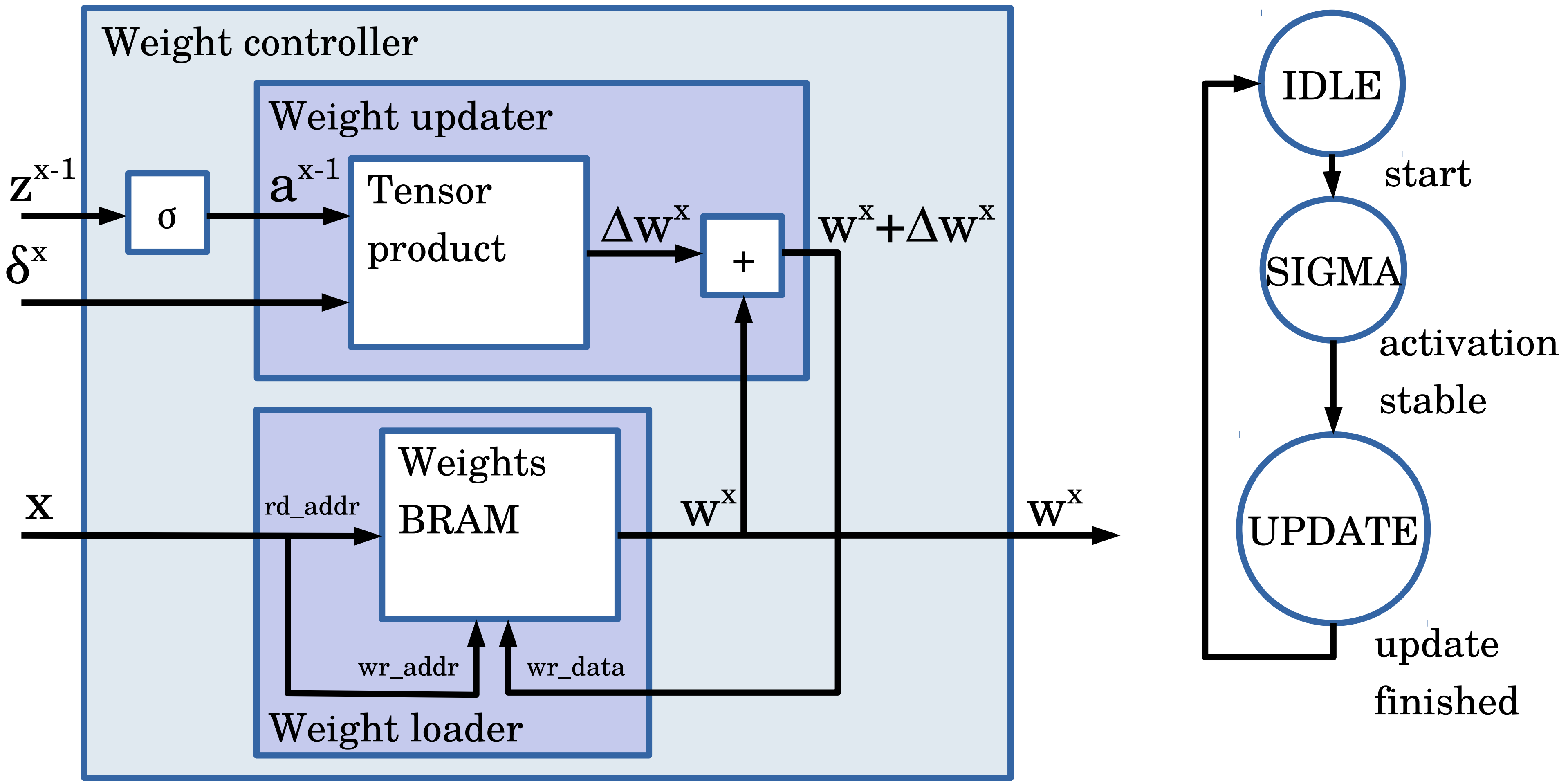
Backpropagator



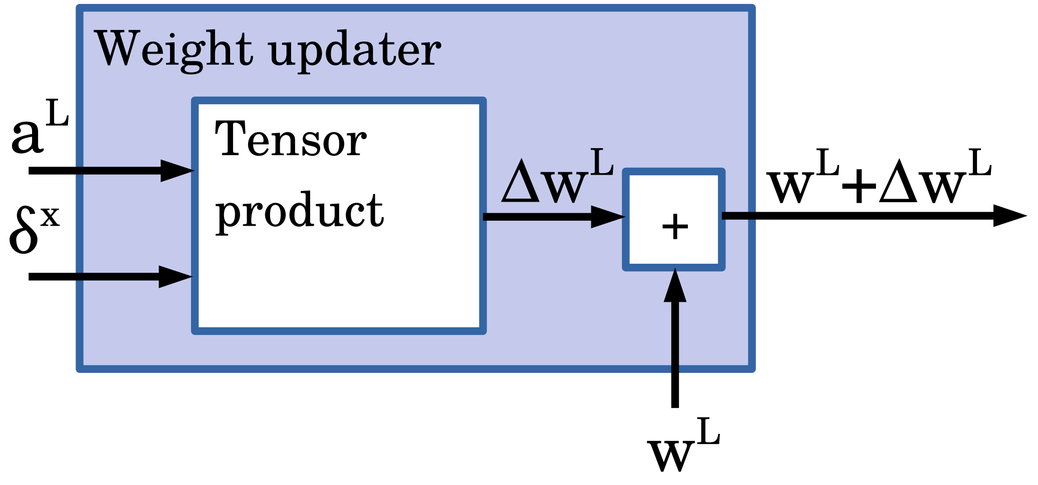
Backpropagator



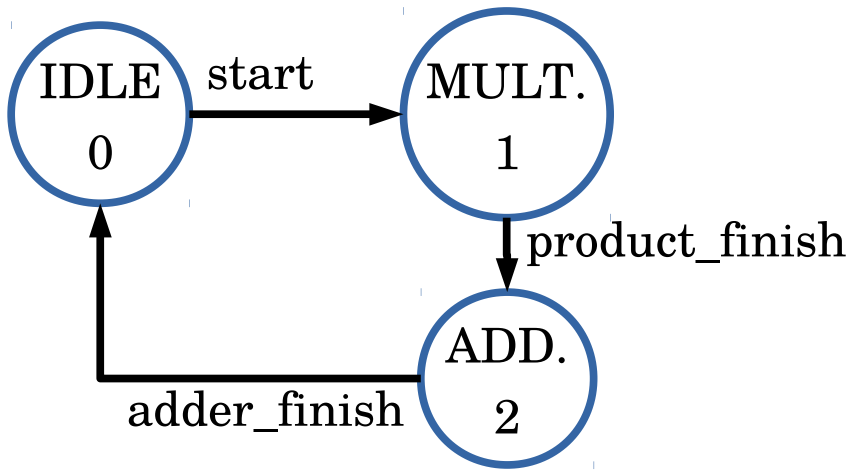
Weight controller



Weight updater



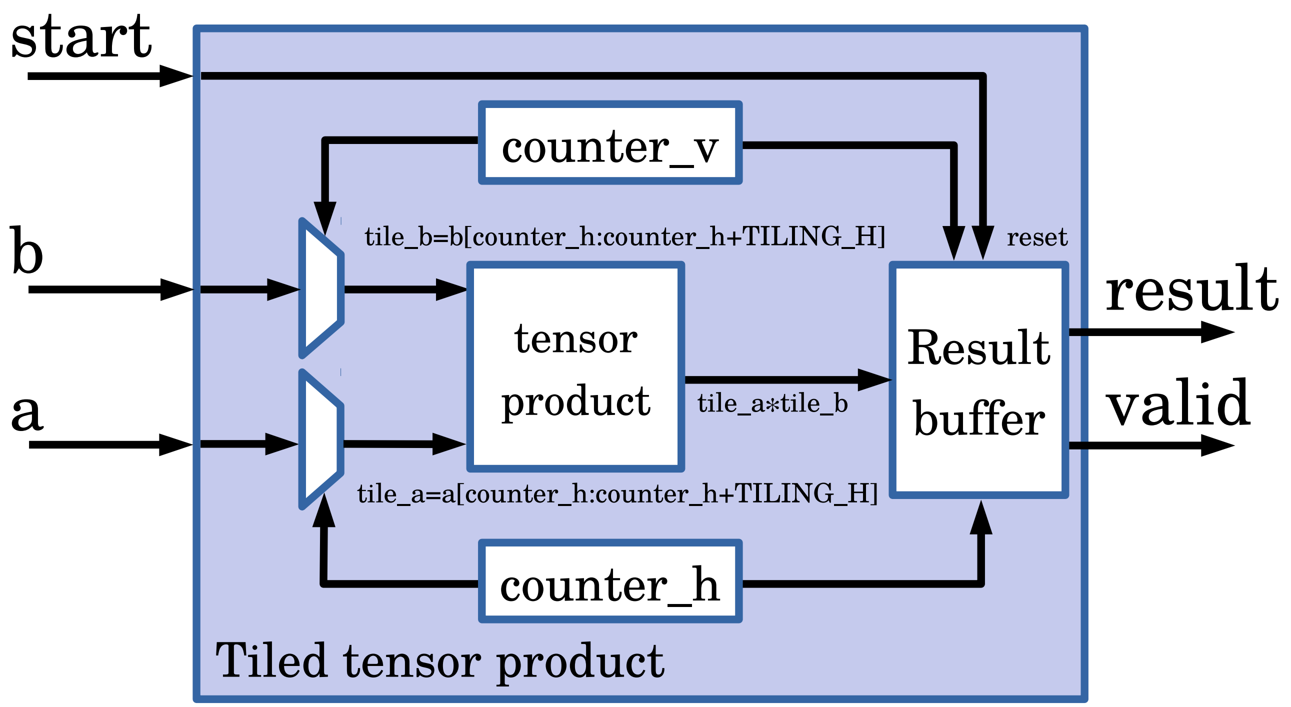
Weight updater receives the neuron activations and their errors, and computes ***ΔW*** as a matrix product of ***a*** and ***δ***. The updates matrix is added with the weights matrix and is sent out.
The product is calculated as a matrix product of a vertical vector ***a*** and a horizontal vector ***δ***.



State diagram

ADD LEARNING RATE

Tiled tensor product



$$\mathbf{g} \otimes \mathbf{e} = \mathbf{g} * \mathbf{e}^T = \begin{pmatrix} g_x \\ g_y \\ g_z \end{pmatrix} \begin{pmatrix} e_x & e_y & e_z \end{pmatrix} = \begin{pmatrix} g_x e_x & g_x e_y & g_x e_z \\ g_y e_x & g_y e_y & g_y e_z \\ g_z e_x & g_z e_y & g_z e_z \end{pmatrix}$$

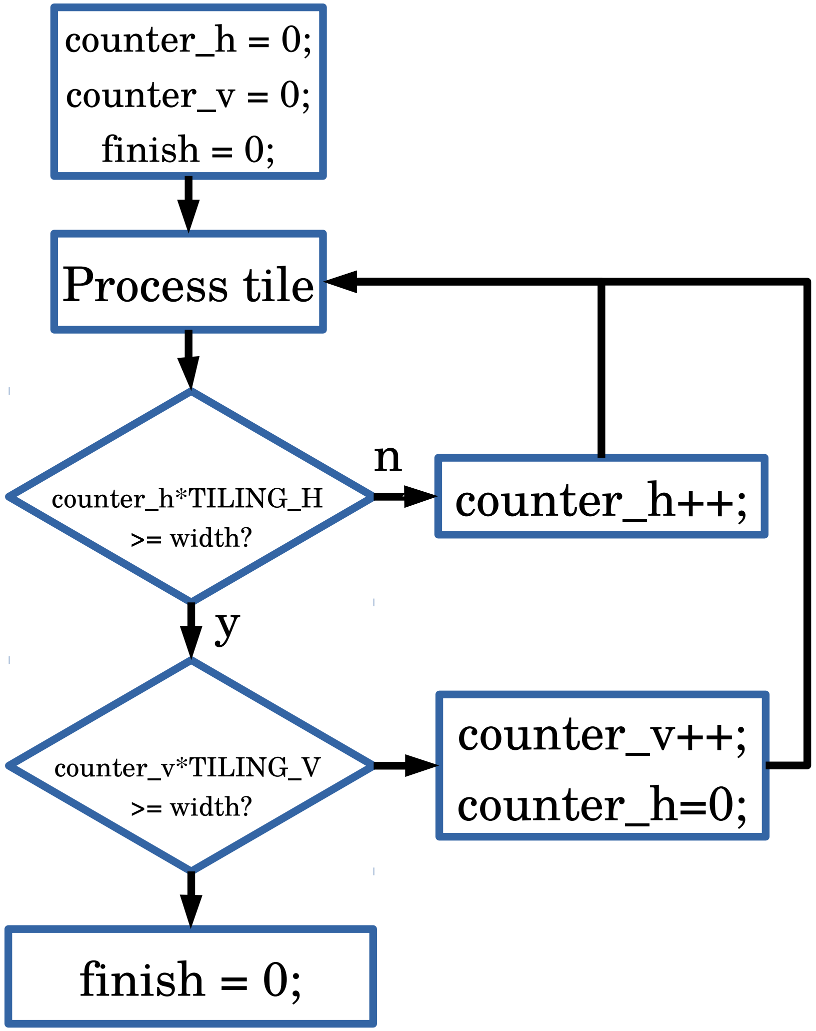
Tensor product is a generalized module that calculates the product of a vertical vector ***a*** and a horizontal vector ***b***, as seen in the picture. On start, the result vector is cleared, and can be used once valid is high.

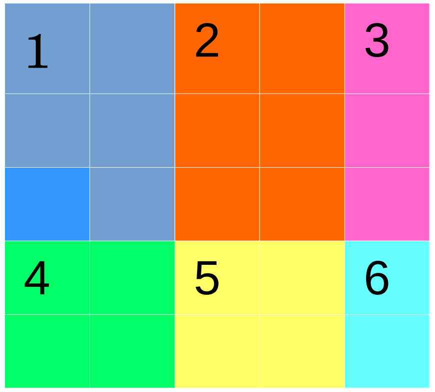
The number of multipliers and cycles can be controlled with the TILING_H and TILING_V parameters, which control the number of rows being processed in parallel and the number of elements processed per row at a time. The number of multipliers inferred can be then calculated as TILING_H*TILING_V.

Vector ***a*** is the vertical one, and ***b*** is the horizontal one, so the TILING_V affects ***a*** and TILING_H affect ***b***.

*The tiled tensor product is meant to be as application agnostic as possible, so it is not tied to the NN concepts.

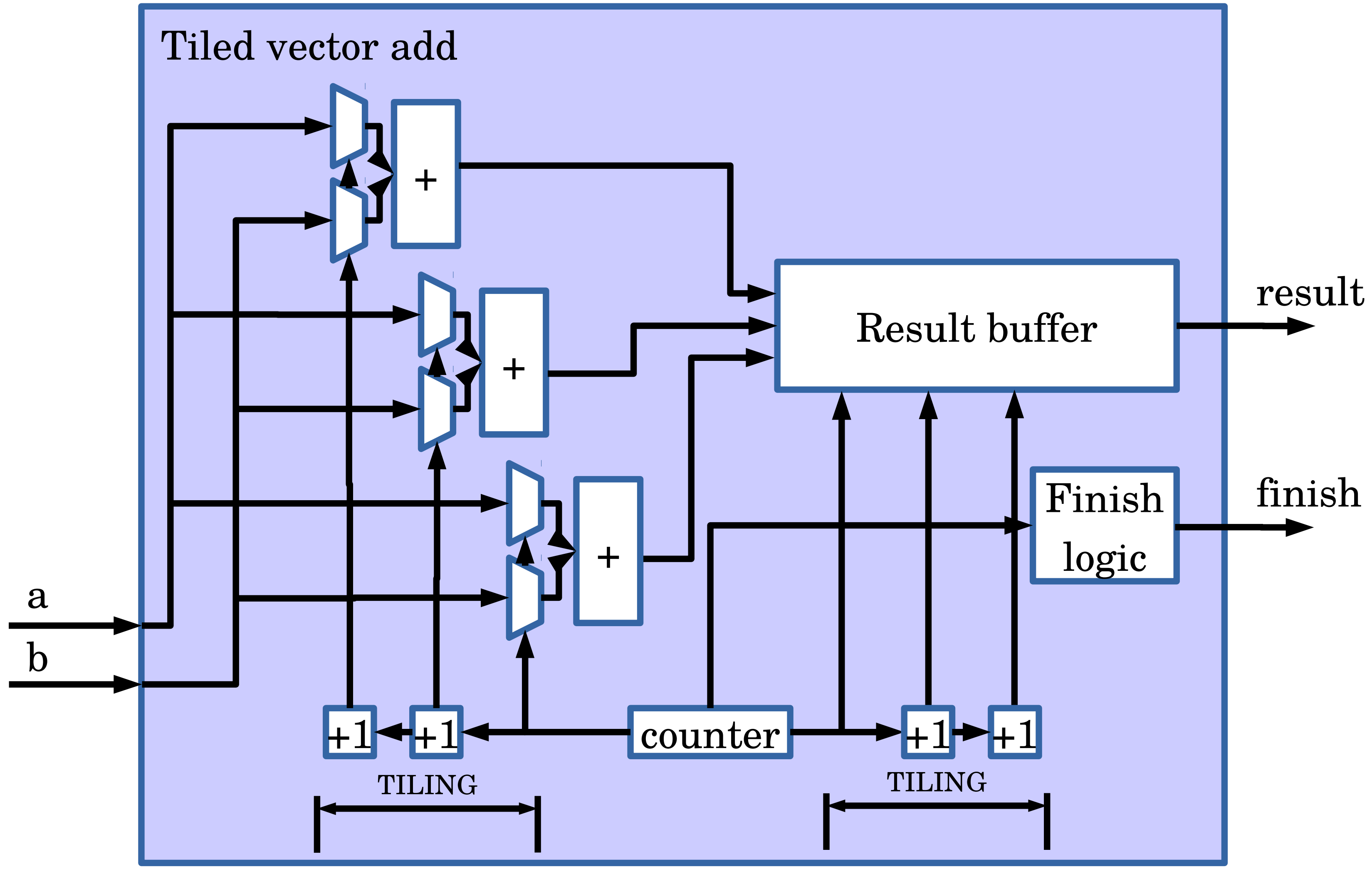
Tiled tensor product logic





Tiling progress of a 5x5 table, tiled with TILING_H=2 and TILING_V=3

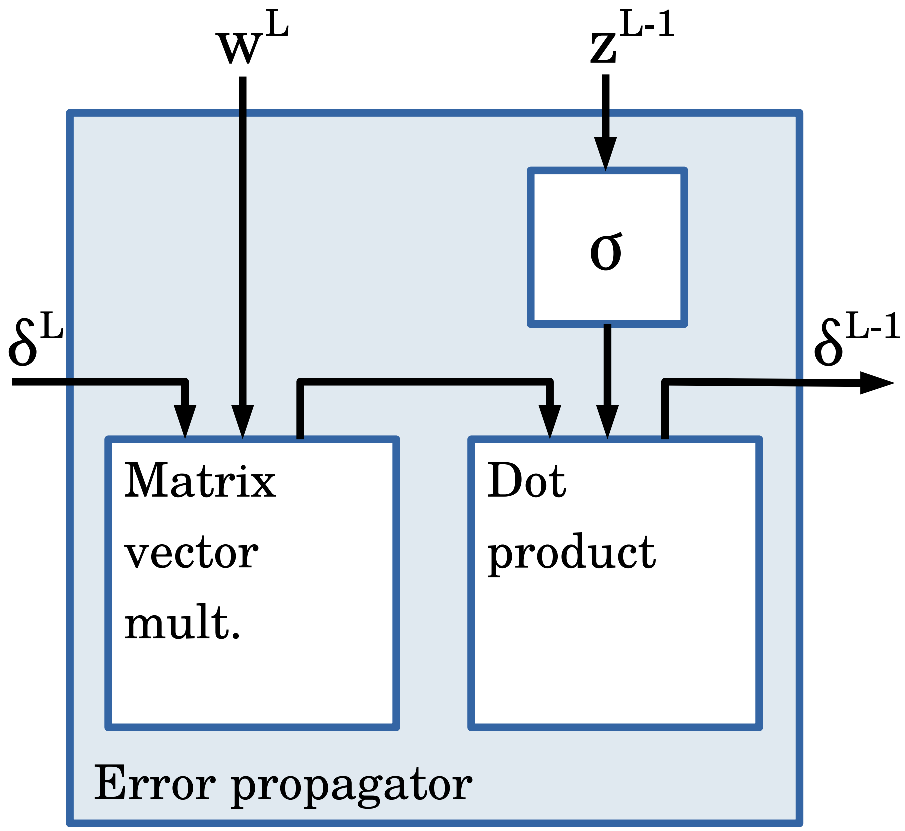
Tiled vector add



Tiled vector add sums two vectors ***a*** and ***b***. The **TILING** parameter controls how many adders are created, affecting the number of cycles needed to add the two vectors.

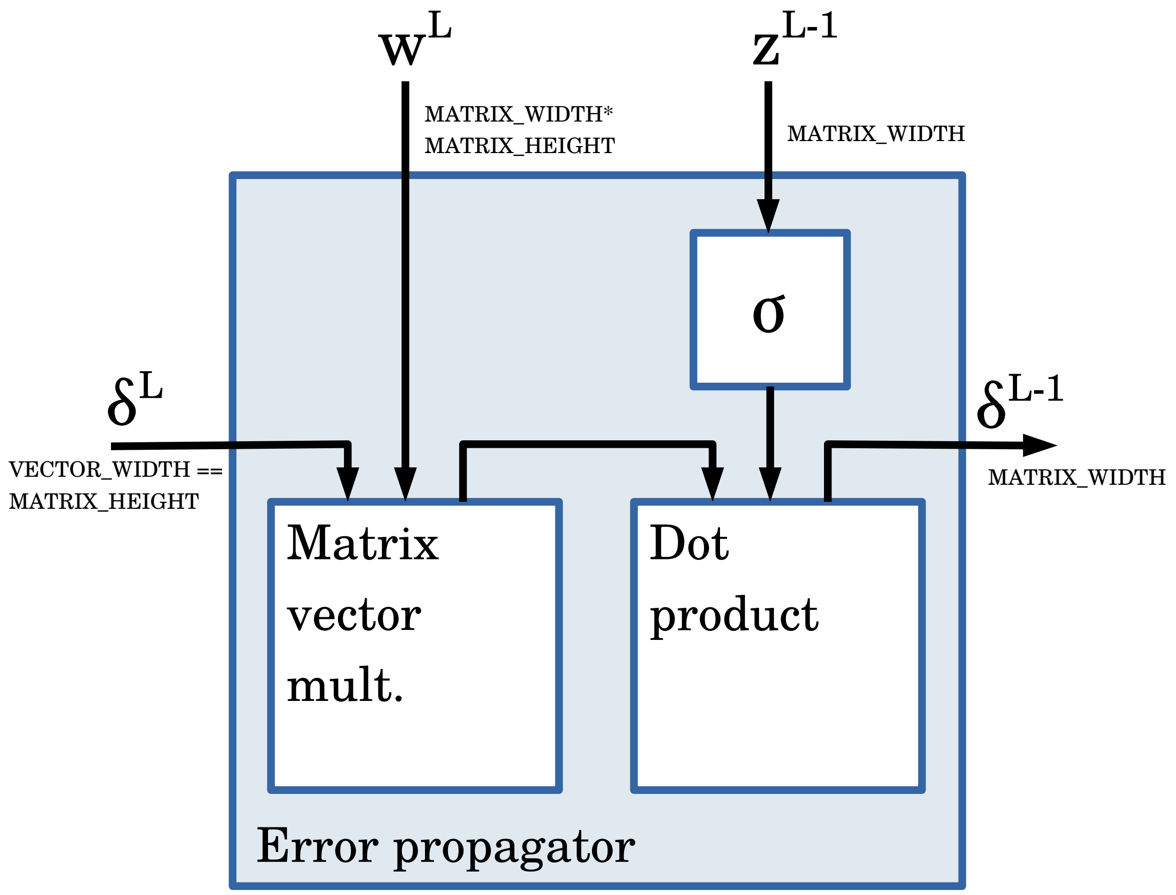
*This document might be *too* descriptive? Abstract the tiling logic away?

Error propagator



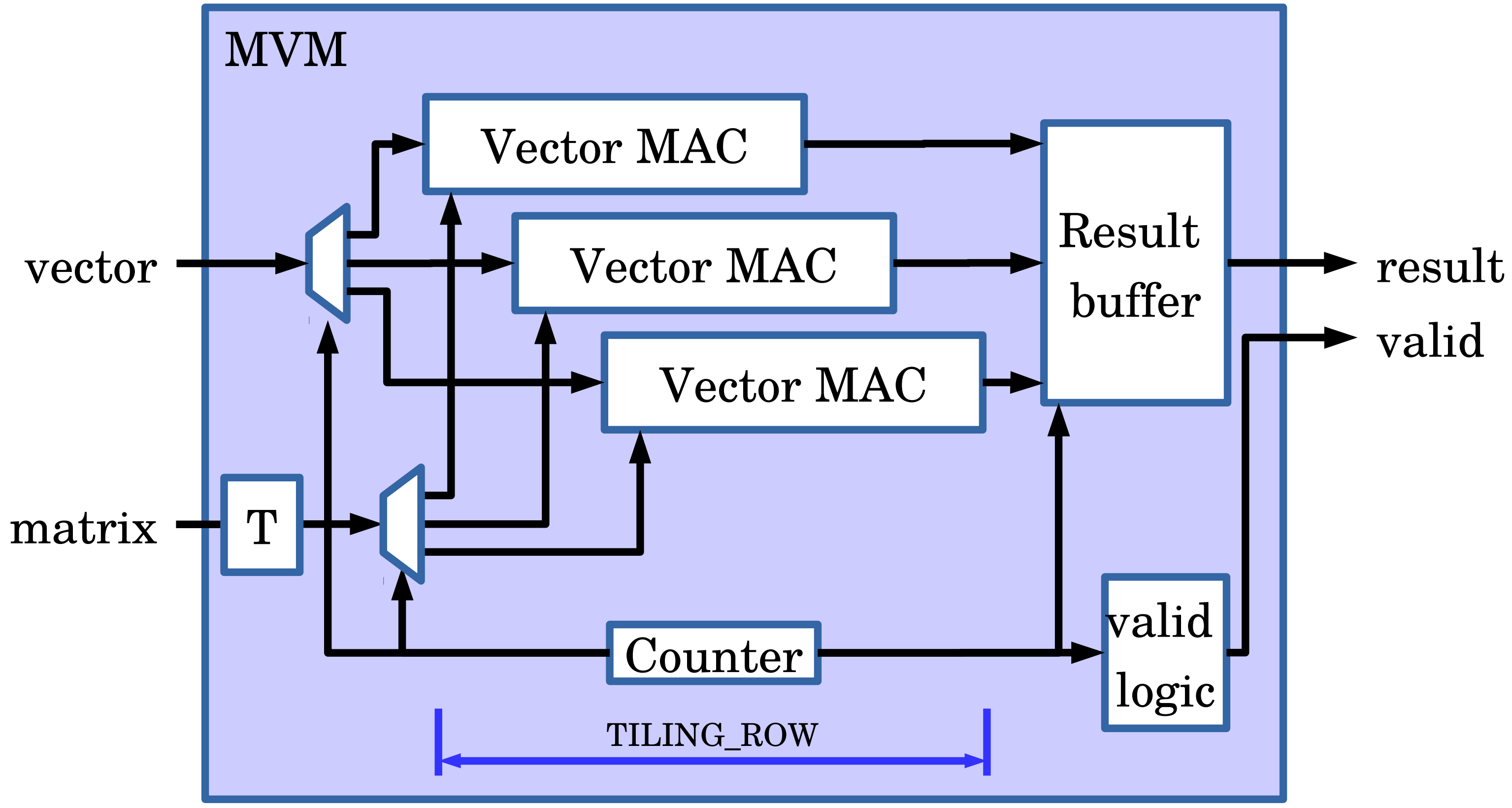
The error propagator passes errors through layers, going from top to bottom.

Error propagator



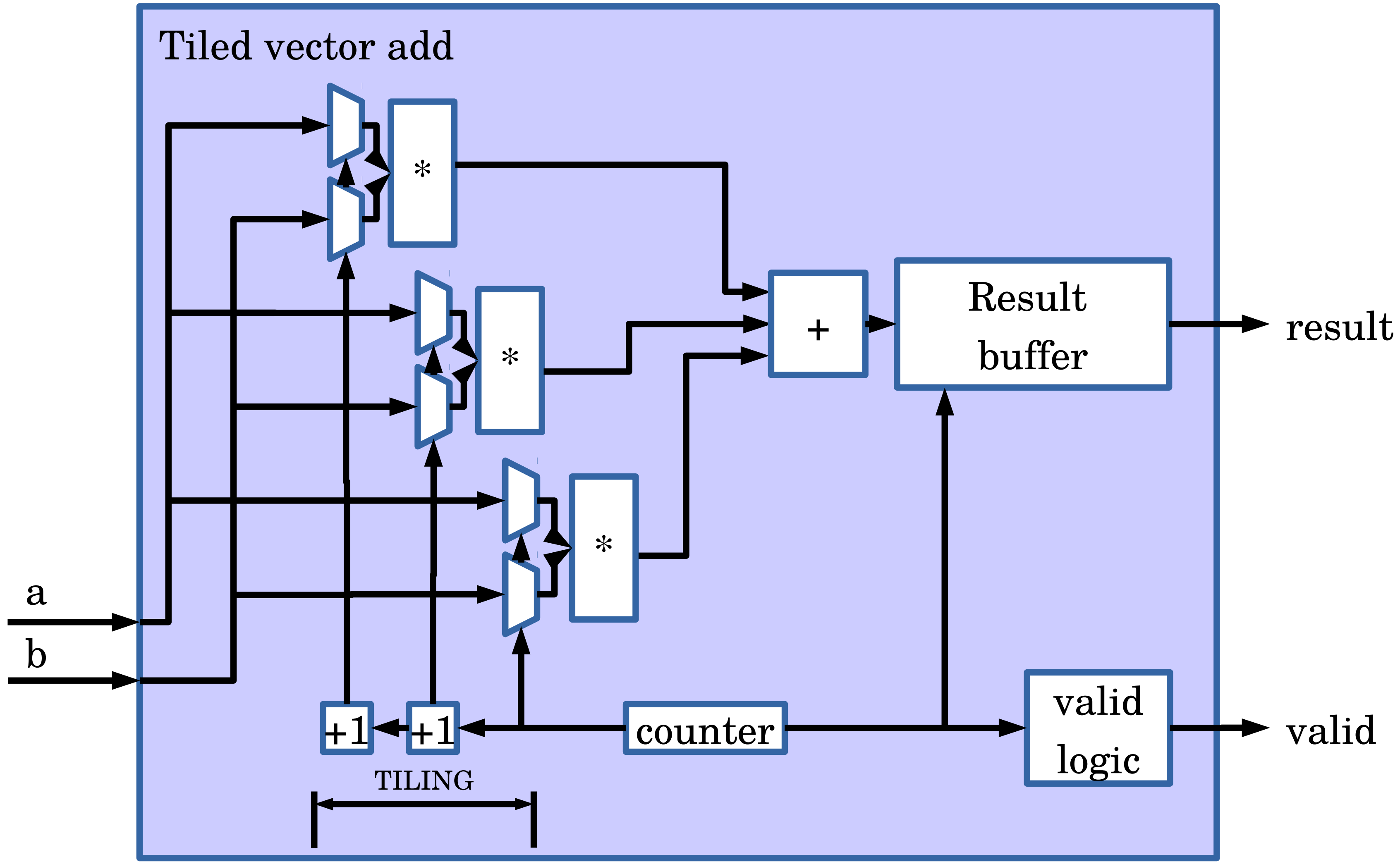
The error propagator passes errors through layers, going from top to bottom.

Matrix vector multiplication



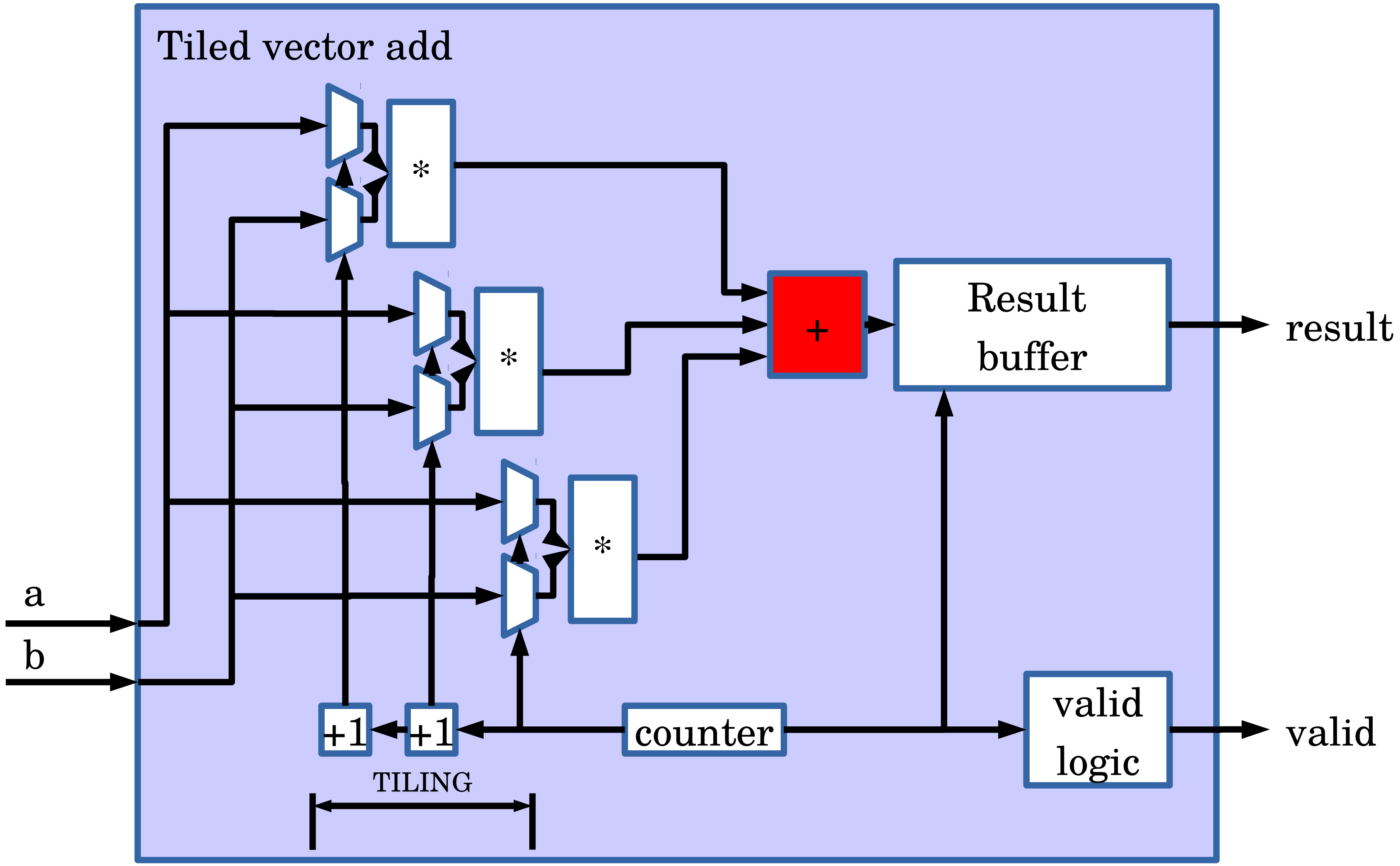
The matrix vector multiplication module receives a vector and a matrix and computes their product. The MVM creates a TILING_ROW vector MAC units that dot multiply and sum two vectors, and feeds them the vector and rows of the matrix. Result buffer stores the outputs of MACs in the appropriate addresses. The TILING_COL parameter controls the number of multipliers in the vector MAC modules. The total number of multipliers is TILING_COL*TILING_ROW.

Vector MAC



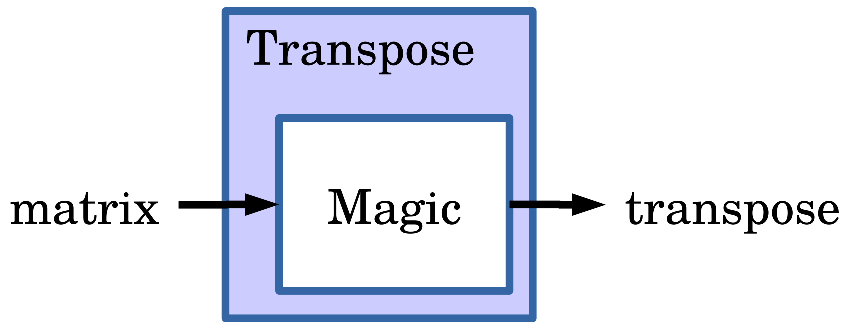
Vector MAC unit computes the sum of a dot product of two vectors.
The TILING parameter controls the number of multipliers in the module

Vector MAC



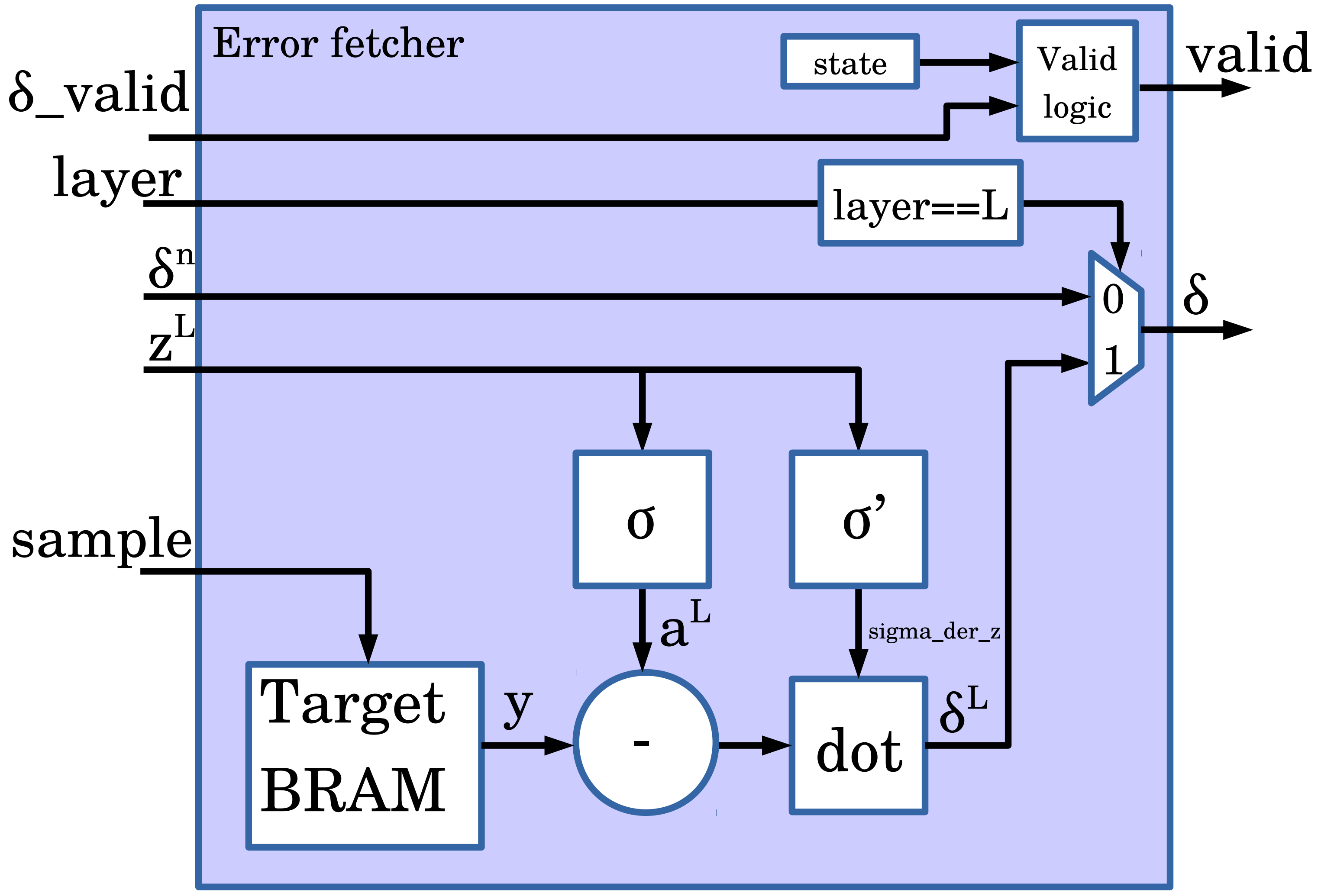
The summing of TILING products is done serially – one vector at a time.
Ideally an adder tree should sum the vector instead of a chain of adders.

Transpose module



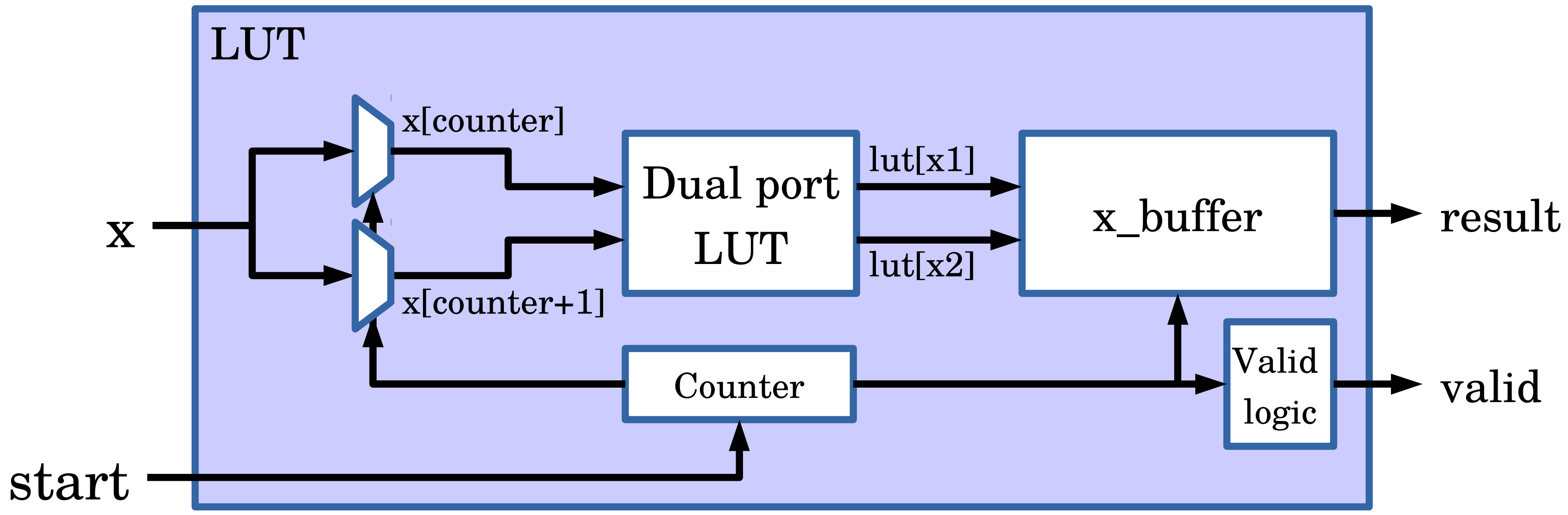
Transpose module takes a WIDTH*HEIGHT matrix of ELEMENT_WIDTH wide cells, and outputs a HEIGHT*WIDTH transposed matrix. The module is completely combinatorial.
Perhaps we should clock this module?

Error fetcher



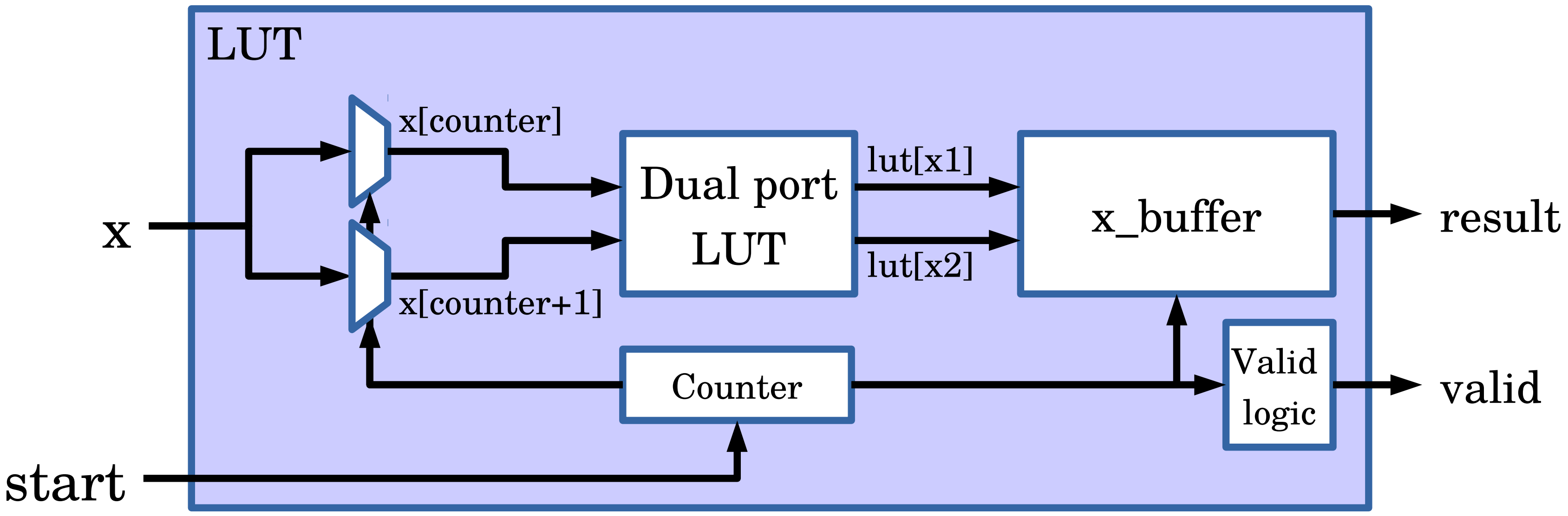
Error fetcher calculates the errors of the output layer, and chooses which error to push depending on the layer currently processed.
The *z* signal is the activation of the highest layer from which the error is calculated using the target stored in BRAM.
The delta signal is the error calculated from the previous layer, which is valid only when the layer processed is not the output layer.
The mux chooses between the delta signal and the calculated layer depending on whether the layer input is equal to the max layer.
The valid signal is high when delta from the input is valid and picked, or when the dot finishes processing the output activations.

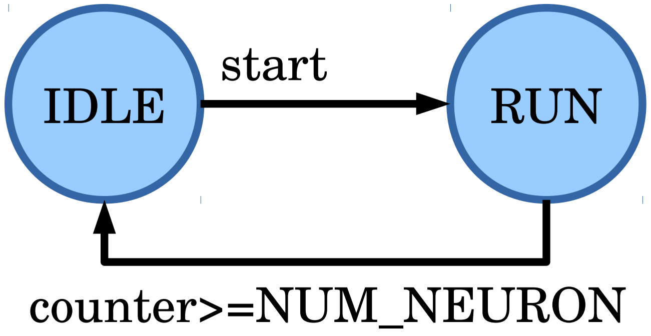
LUT



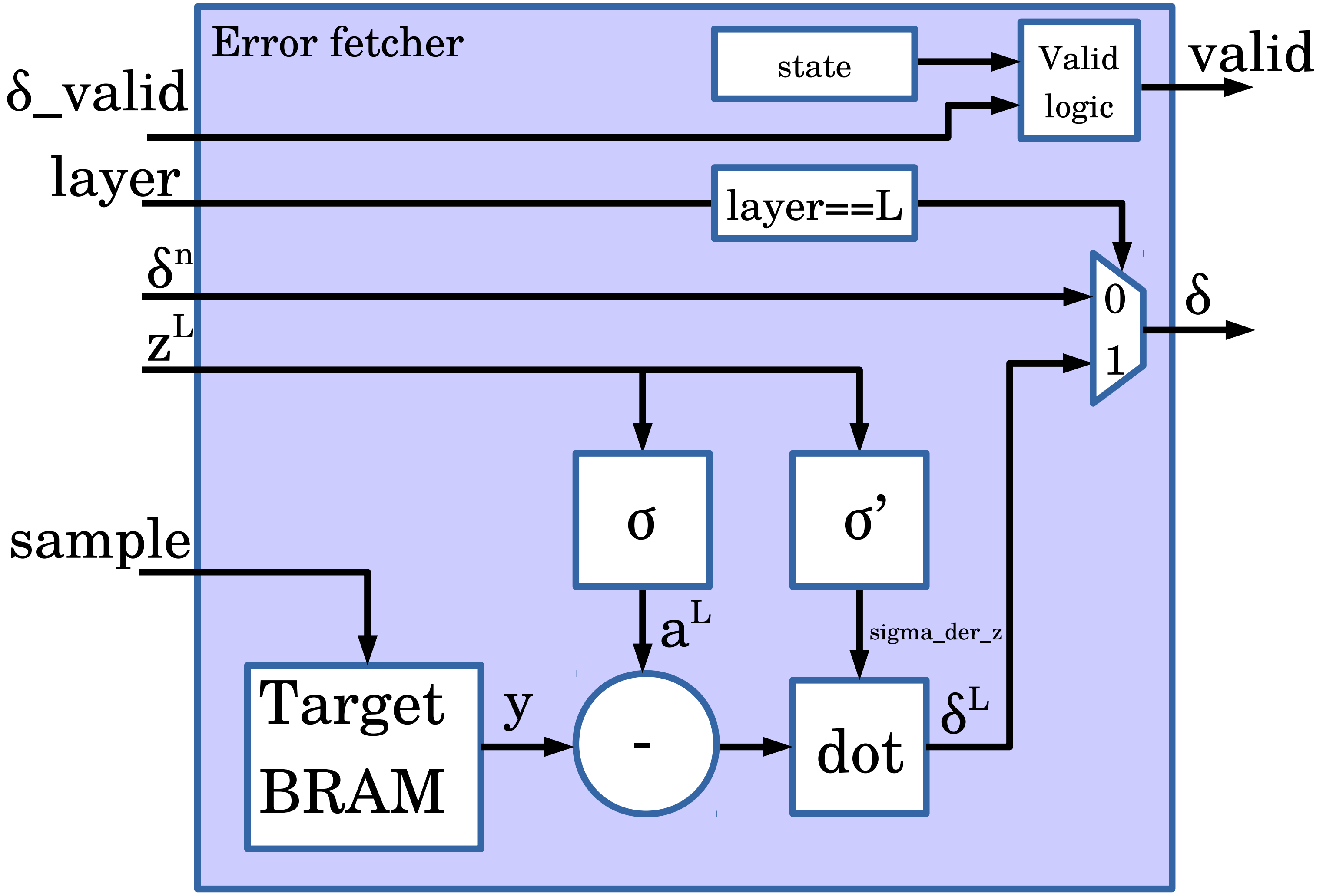
LUT converts an input vector into a vector of values from the ROM. On start, the counter is reset and the input vector is sampled two values at a time, which are ran through a dual port ROM and stored in a buffer. Once the counter value is equal to the vector size, the valid signal is raised.

LUT


LUT
x
x[counter]
x[counter+1]
Dual port LUT
lut[x1]
lut[x2]
x_buffer
result
Counter
Valid logic
valid
start


IDLE
start
RUN
counter>=NUM_NEURON

Error fetcher



Error fetcher calculates the errors of the output layer, and chooses which error to push depending on the layer currently processed.

The ***z*** signal is the activation of the highest layer from which the error is calculated using the target stored in BRAM.

The delta signal is the error calculated from the previous layer, which is valid only when the layer processed is not the output layer.

The mux chooses between the delta signal and the calculated layer depending on whether the layer input is equal to the max layer.

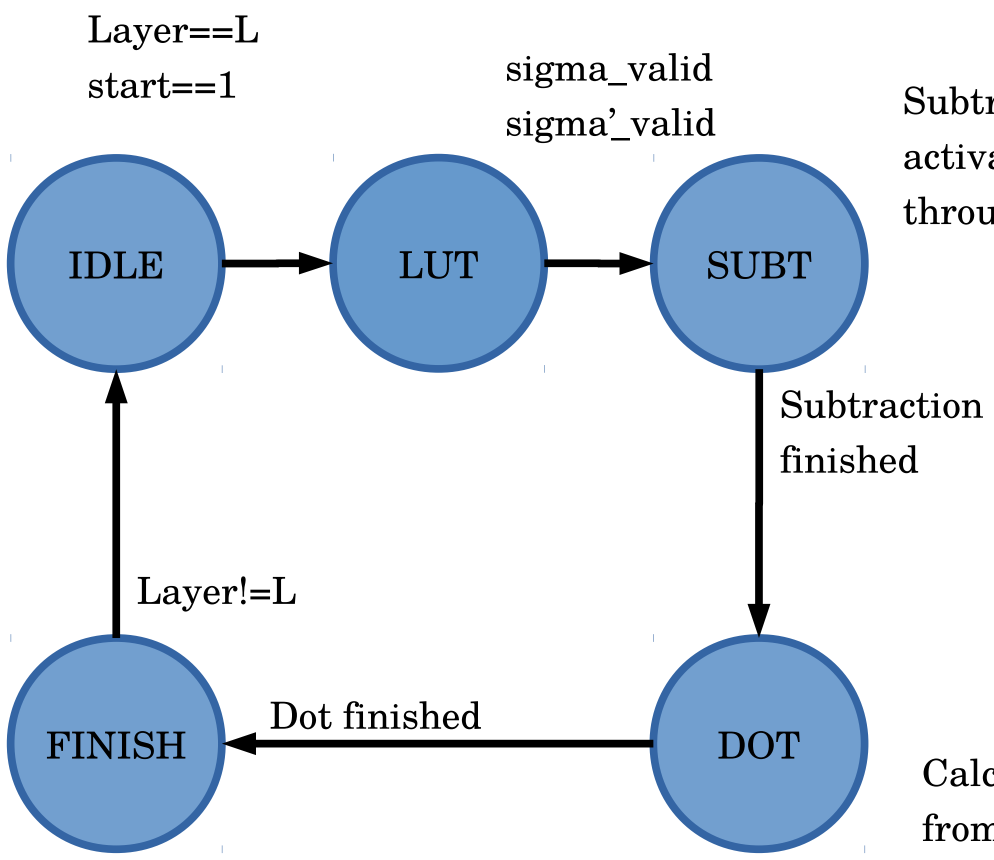
The valid signal is high when delta from the input is valid and picked, or when the dot finishes processing the output activations.

If ***y*** and ***a*** are X bits wide, their result is X+1 bits wide. The output from sigma prime is also X bits wide, so it needs to be converted to X+1. We make the sigma prime return X+1 wide.

Error fetcher

Waits until the layer input is equal to max layer. Passes delta and valid from input.

Subtracts the targets and activations, nothing passes through, valid signal is 0.



Retains the calculated output as long as layer is the max layer. Valid is 1. When the layer changes, goes to IDLE.

Calculates the dot product from the output of the subtractor and sigma derivative. Nothing passes through, valid signal is 0.