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

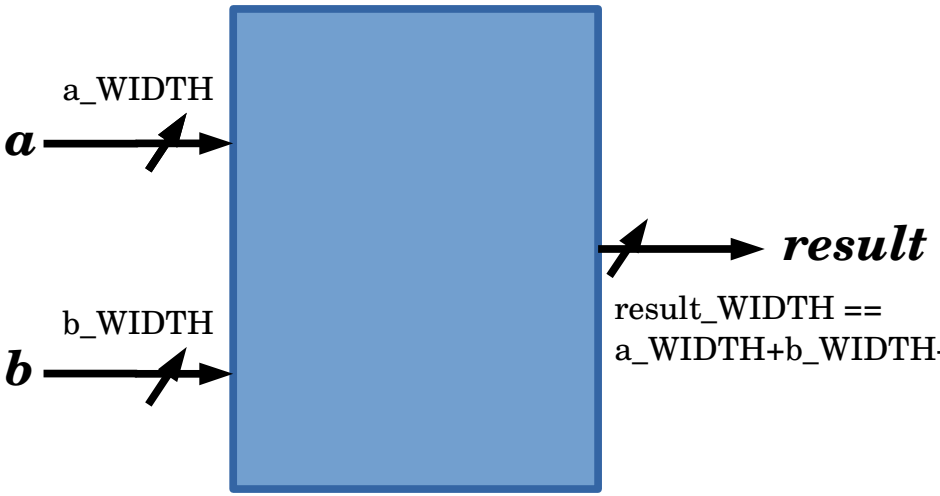
Module name	Can overflow:
Trainer_top	
Activation_stack	
Dual port BRAM	
Backpropagator	
Weight controller	
Weight loader	
Weight updater	
Tiled tensor product	Yes, parametrized out width, has error signal output
Vector add	Yes, parametrized out width, has error signal output
Error controller	
Error fetcher	Contained modules overflow
Vector dot	Yes, parametrized out width, has error signal output
Activation LUT	
Vector subtract	Yes, parametrized out width, has error signal output
Error propagator	Contained modules overflow
Matrix vector multiplication	Contained modules overflow
Vector MAC	Yes, parametrized out width, has error signal output
Vector dot	Yes, parametrized out width, has error signal output
Matrix transpose	

Module name	Uses valid/ready protocol
Trainer_top	
Activation_stack	
Dual port BRAM	
Backpropagator	
Weight controller	yes
Weight loader	no need
Weight updater	yes
Tiled tensor product	yes
Vector add	yes
Error fetcher	yes
Vector dot	yes
Activation LUT	yes
Vector subtract	yes
Error propagator	yes
Matrix vector multiplication	yes
Vector dot	yes

Fixed point issues

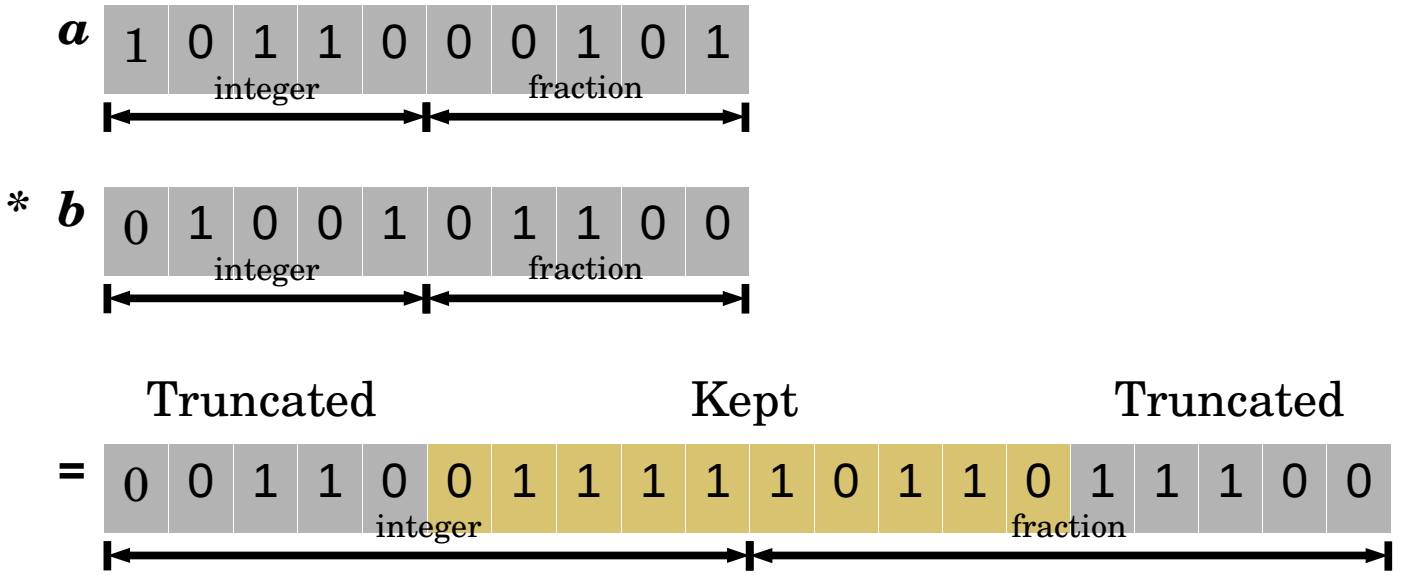
How do we settle the issue of growing fixed point numbers? Every multiplication, the width of a fixed point is the summed width of the inputs, and every subtraction/addition, the width is the incremented width of the larger input.

Since the data is moving in a circle (previous delta reused to calculate the new delta, and weights constantly being updated), we need a large enough size for all of it without sacrificing precision. By picking a large enough value for the signal width, we can truncate signals without too much worry. When multiplying two fixed point numbers, the fraction grows along with the integer part. We can solve the fraction part by truncating it to a predefined amount, and we can avoid issues with integer part by assigning enough width to it for it to not grow out of hand.



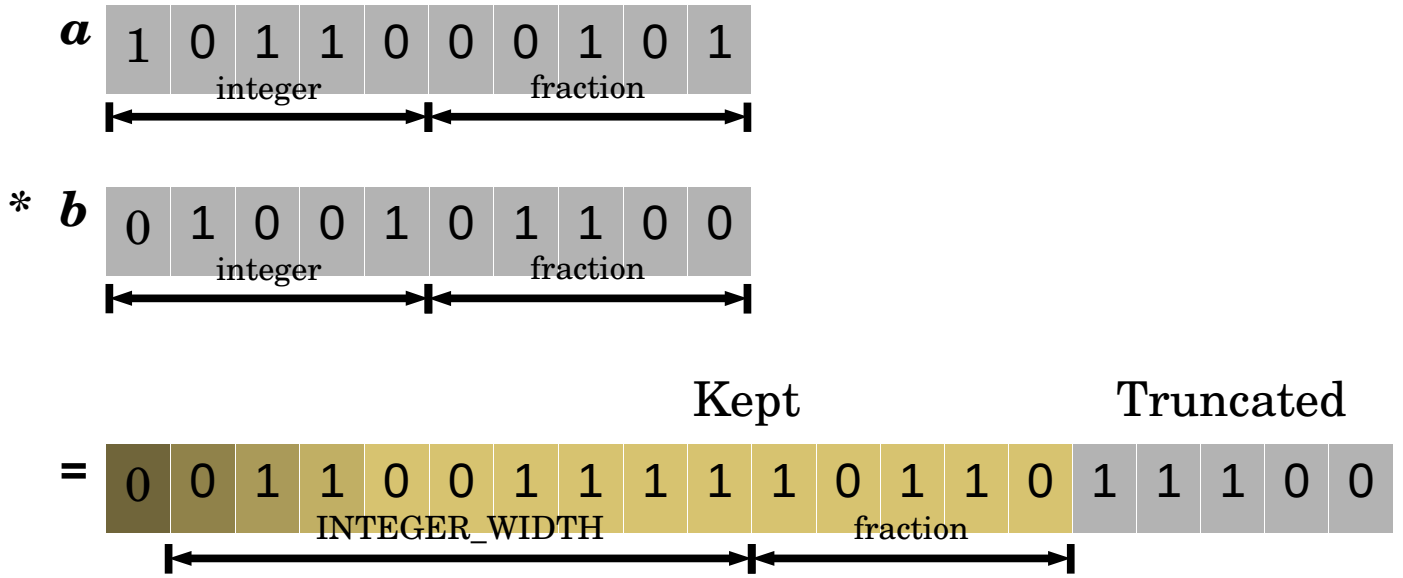
With a global FRACTION parameter setting the number of bits used for the fraction part, and WIDTH-FRACTION used for the integer part, we only need to specify the WIDTH parameter for each fixed point signal.

Option 1



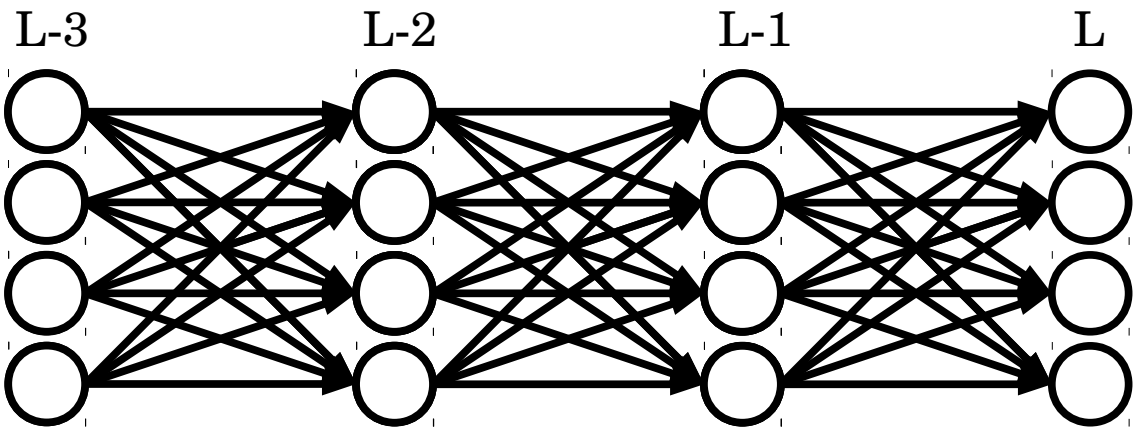
Result width not parametrized: the size of integer and fraction is kept.
The size of the result is the same as that of inputs.

Option 2 - Adopted

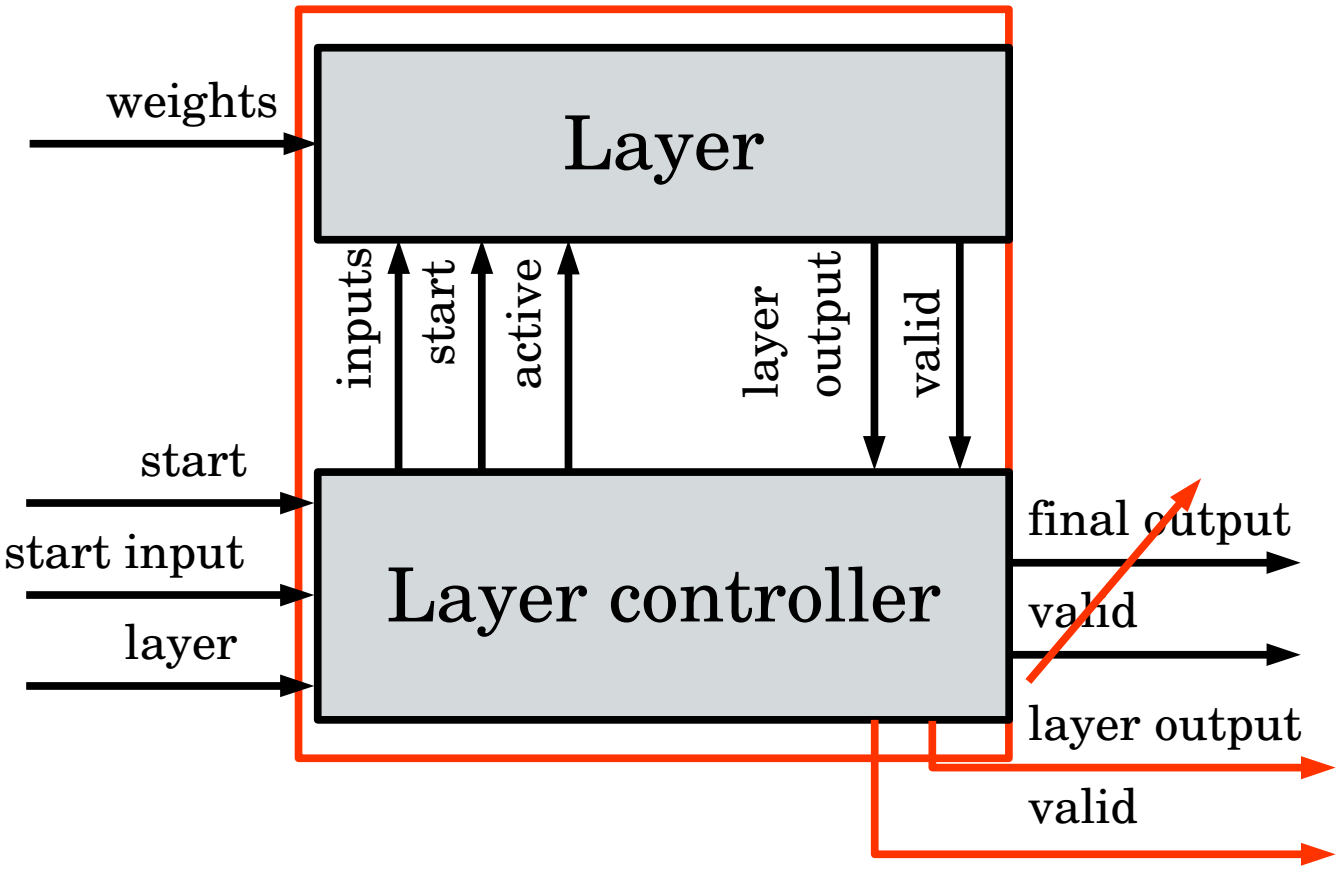


Integer width parametrized: the fraction is truncated to match input, but the size of integer part can be larger than that of inputs.
For some input with size INTEGER_WIDTH + FRACTION, the size of the result is RESULT_INTEGER_WIDTH + FRACTION

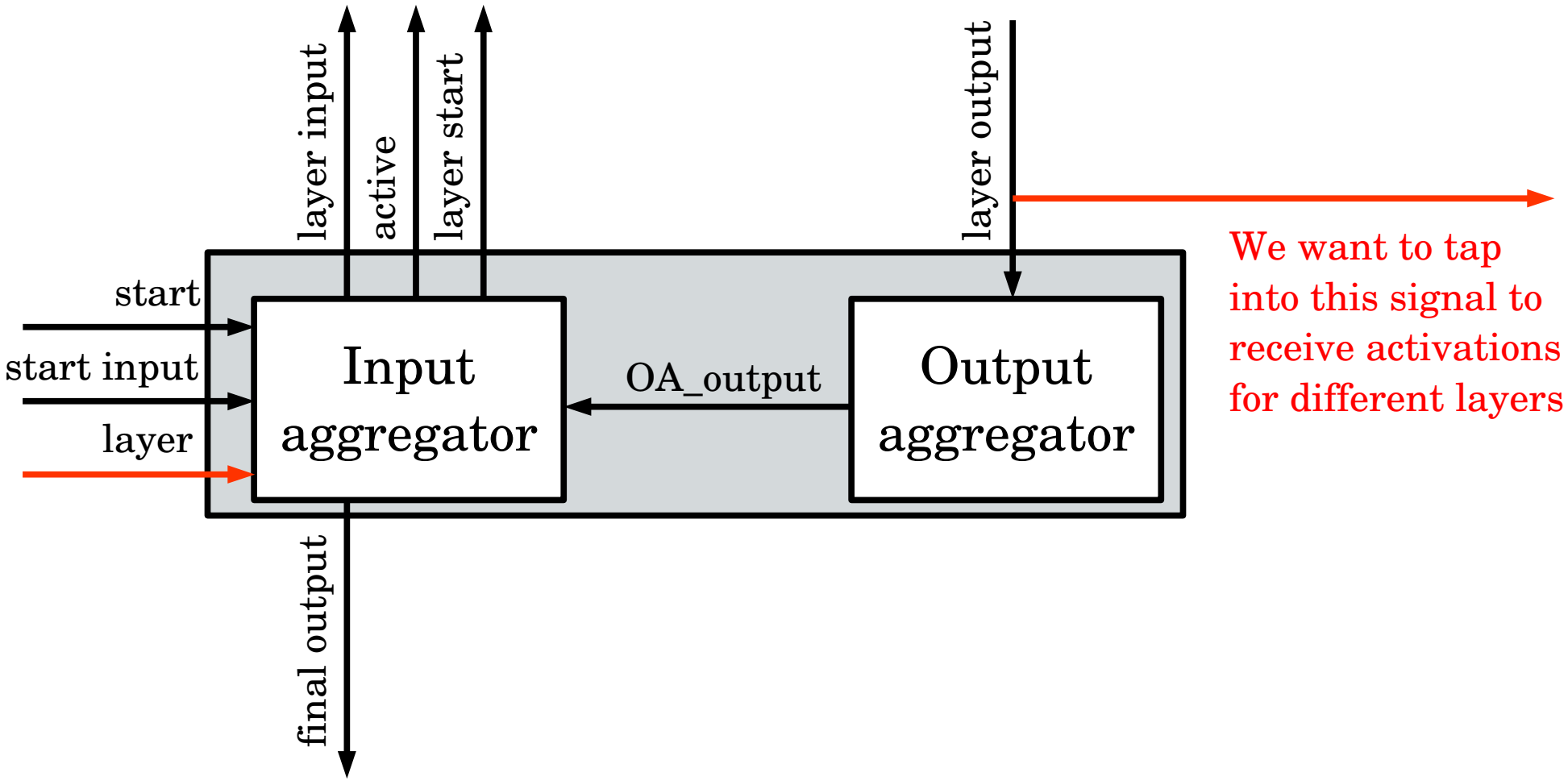
Neural network nomenclature



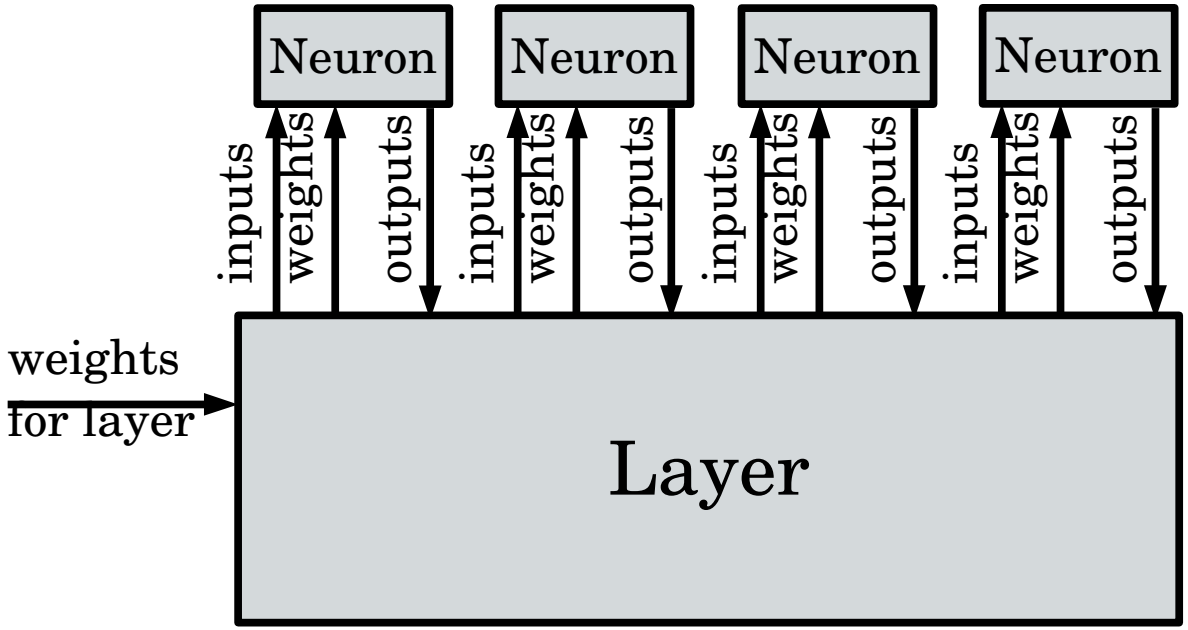
Forward pass



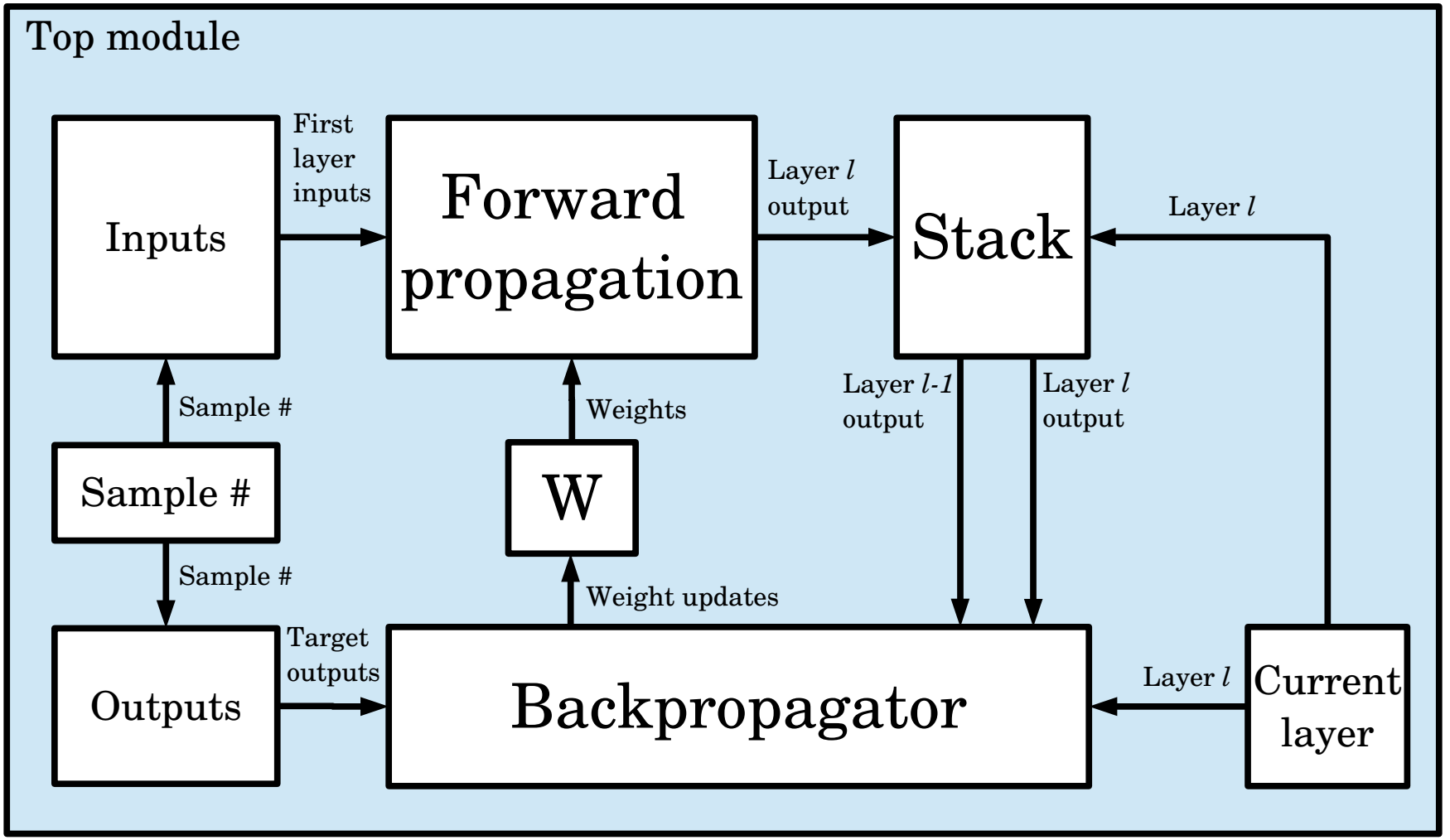
Layer controller



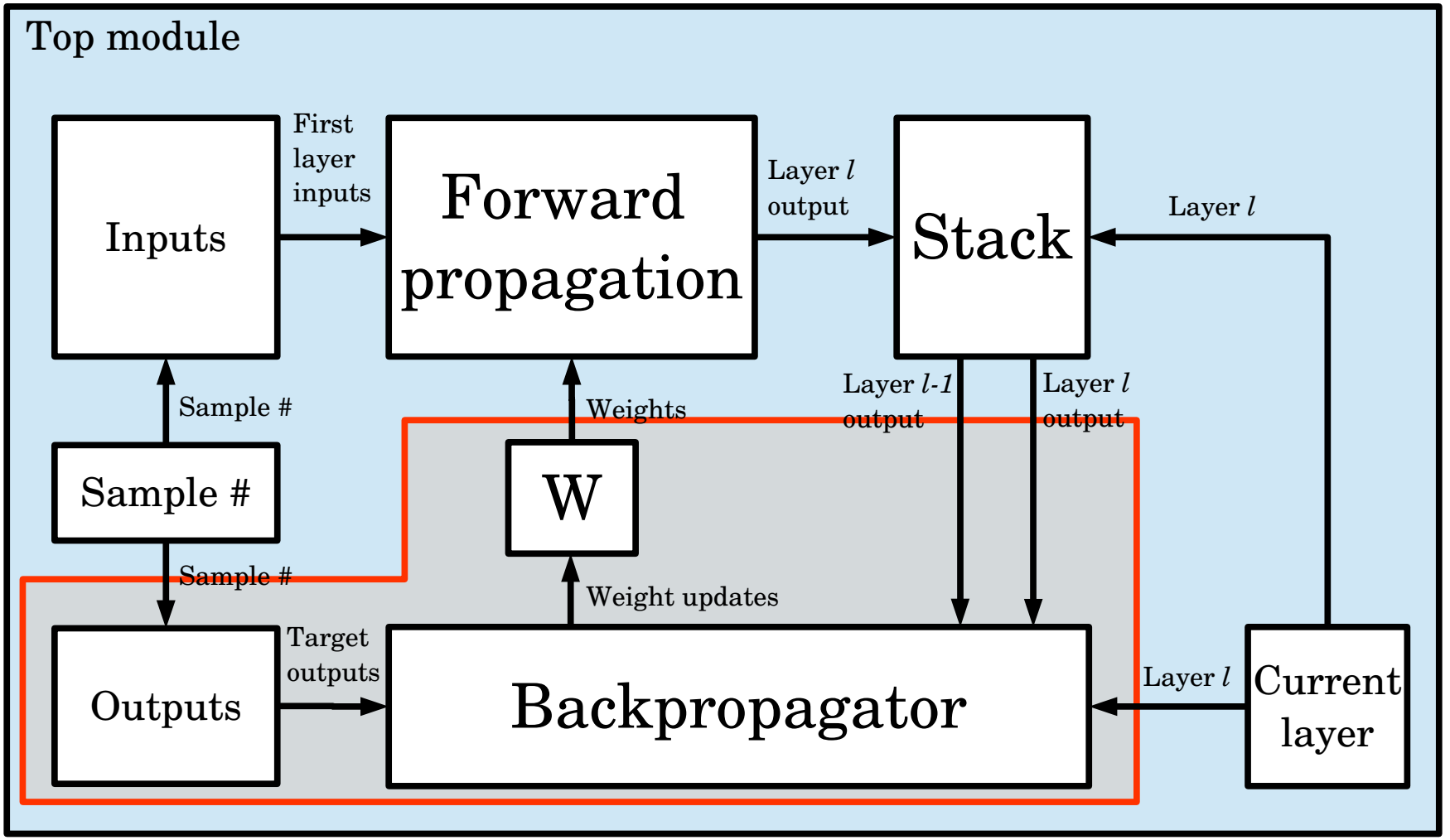
Layer & neurons



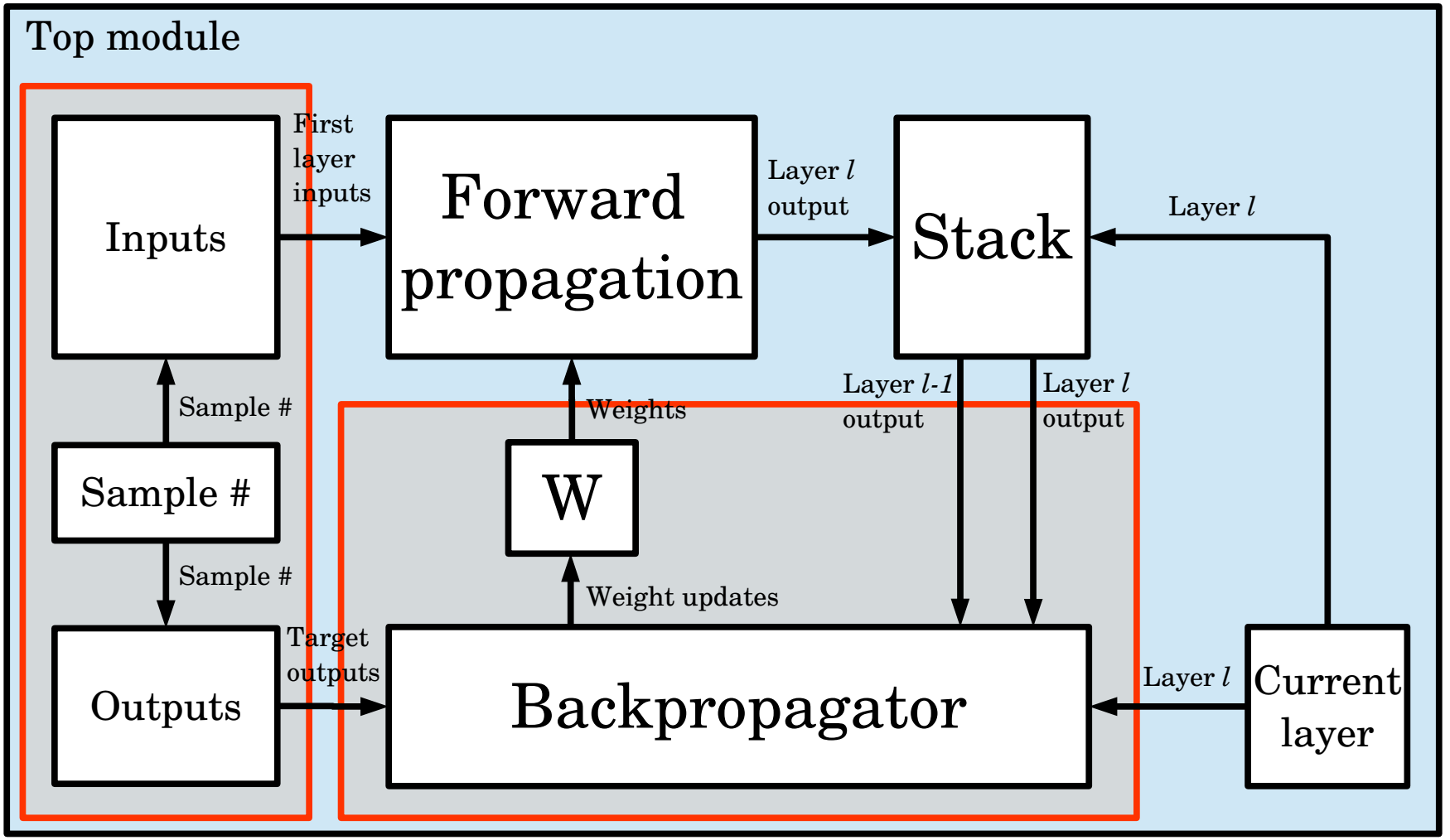
Top module


Top module
Inputs
First layer inputs
Forward propagation
Layer l output
Stack
Layer l
Sample #
Weights
Layer l-1 output
Layer l output
Sample #
W
Sample #
Weight updates
Outputs
Target outputs
Backpropagator
Layer l
Current layer

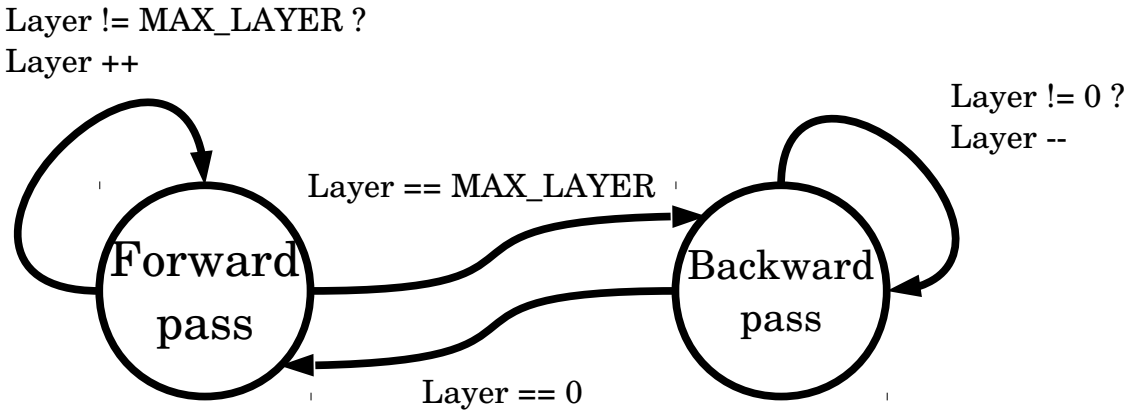
Top module


Top module
Inputs
First layer inputs
Forward propagation
Layer l output
Stack
Layer l
Sample #
Weights
Layer l-1 output
Layer l output
Sample #
W
Sample #
Weight updates
Target outputs
Outputs
Backpropagator
Layer l
Current layer

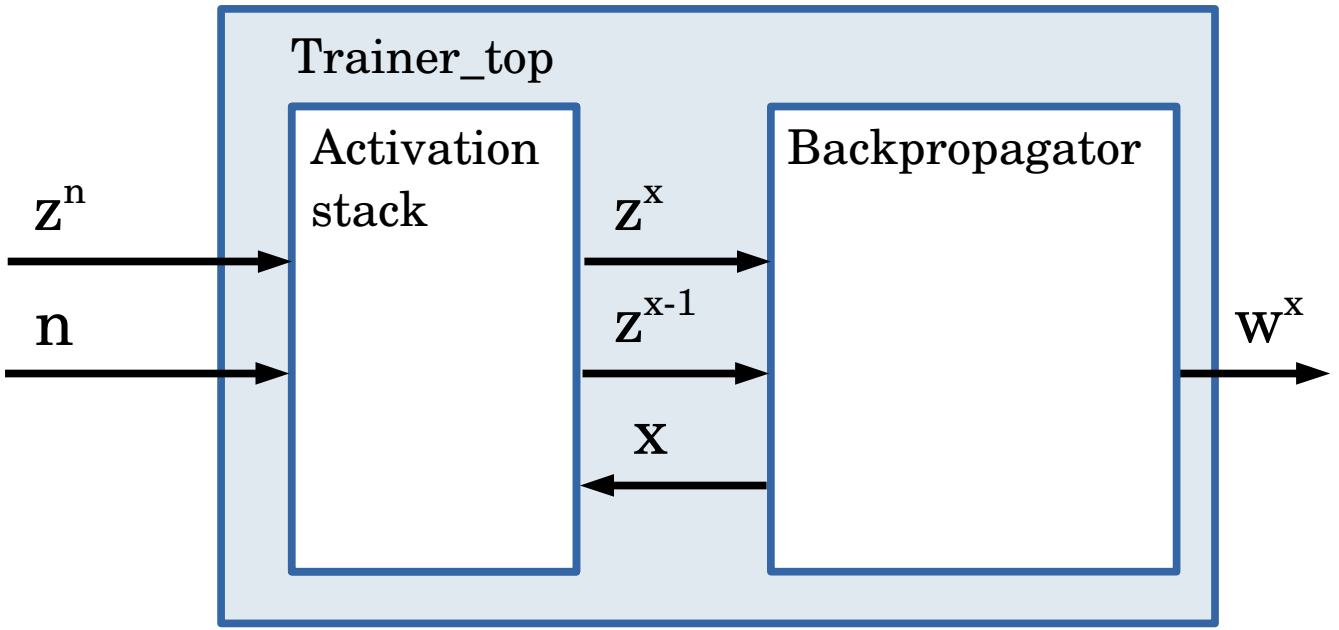
Ideally: Top module



Top module

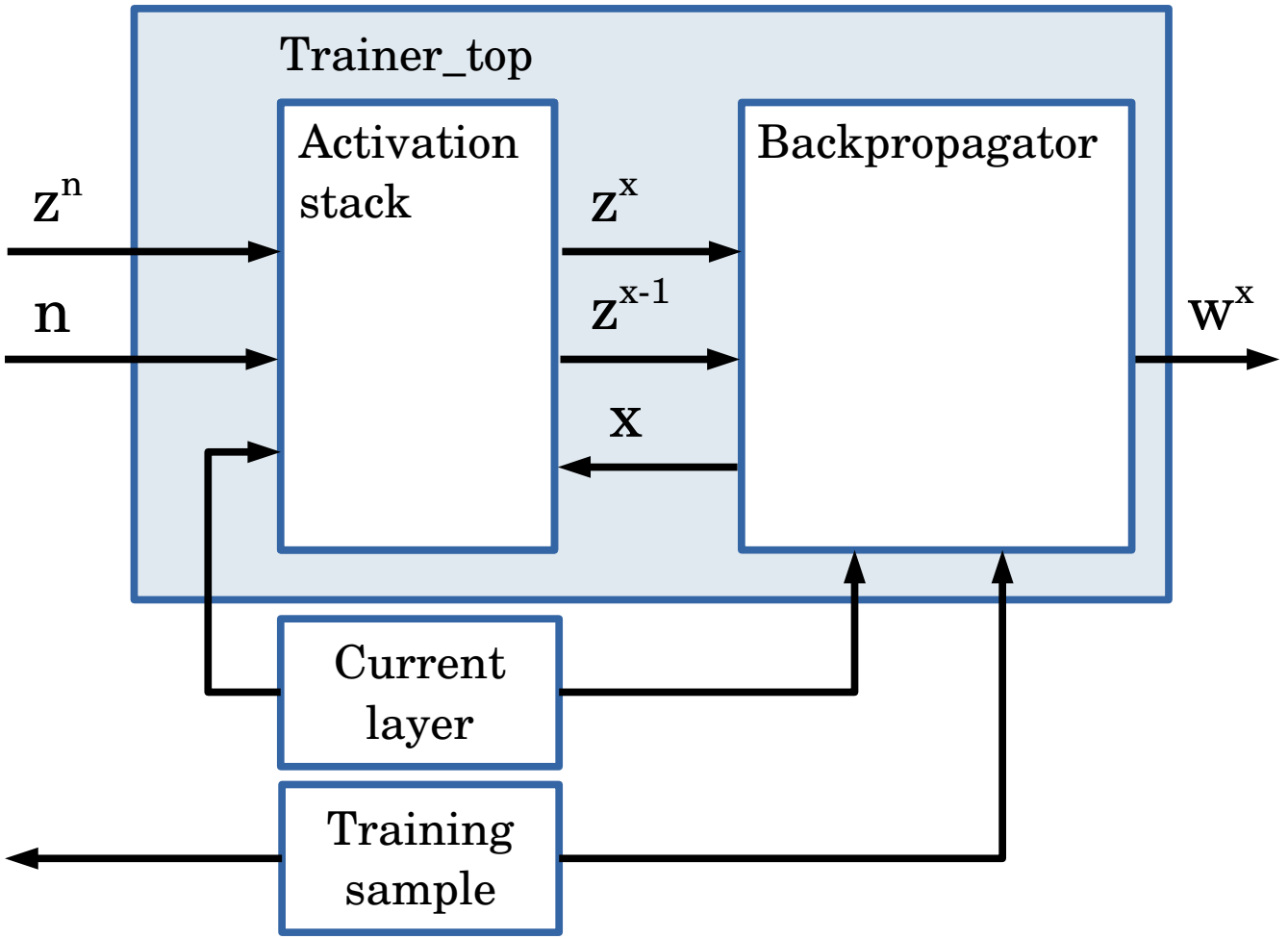

Layer != MAX_LAYER ?
Layer ++
Layer != 0 ?
Layer --
Layer == MAX_LAYER
Forward pass
Backward pass
Layer == 0

Trainer top



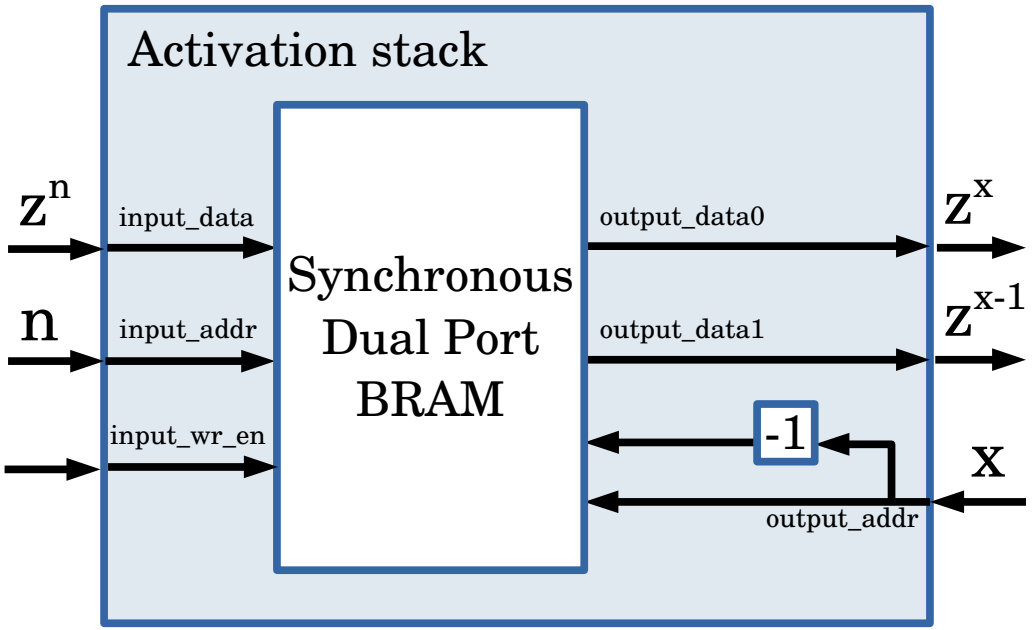
The trainer top module receives activations from the forward propagation module, and returns adjusted weight matrices.
The activations are saved in the stack and fed to the backpropagator module which holds the weights.

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.

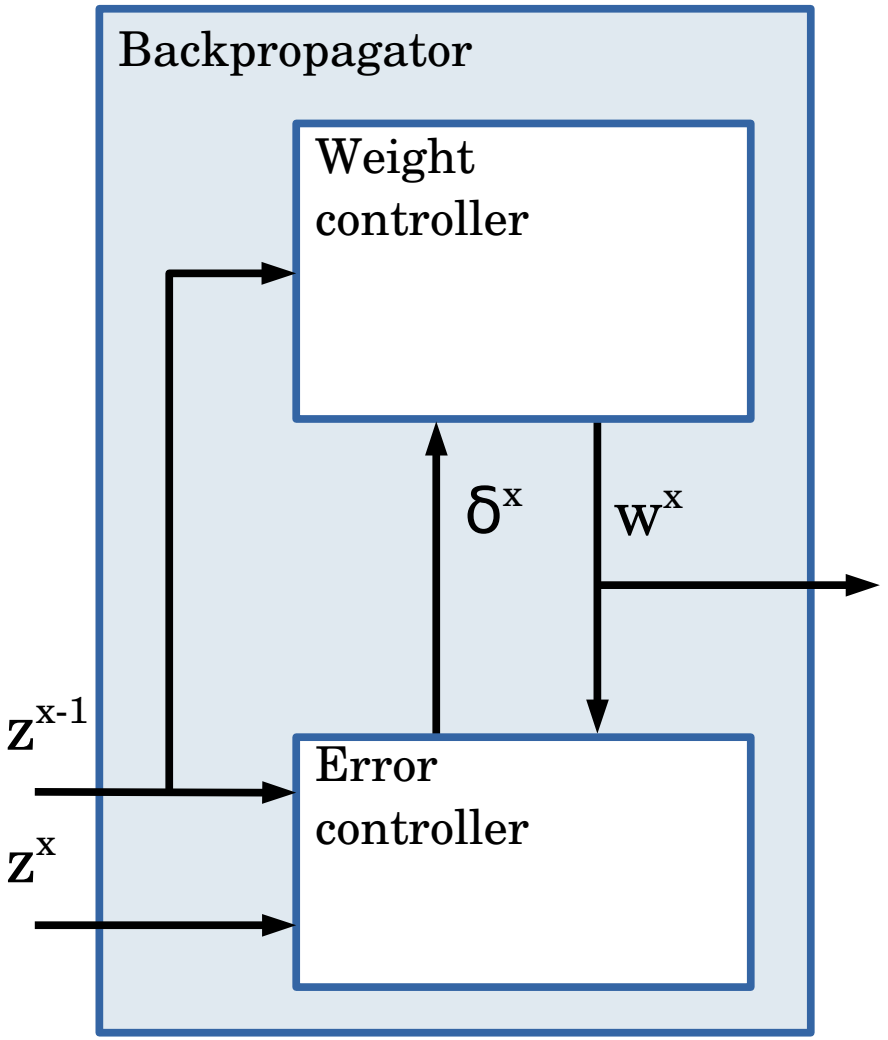
Activation stack



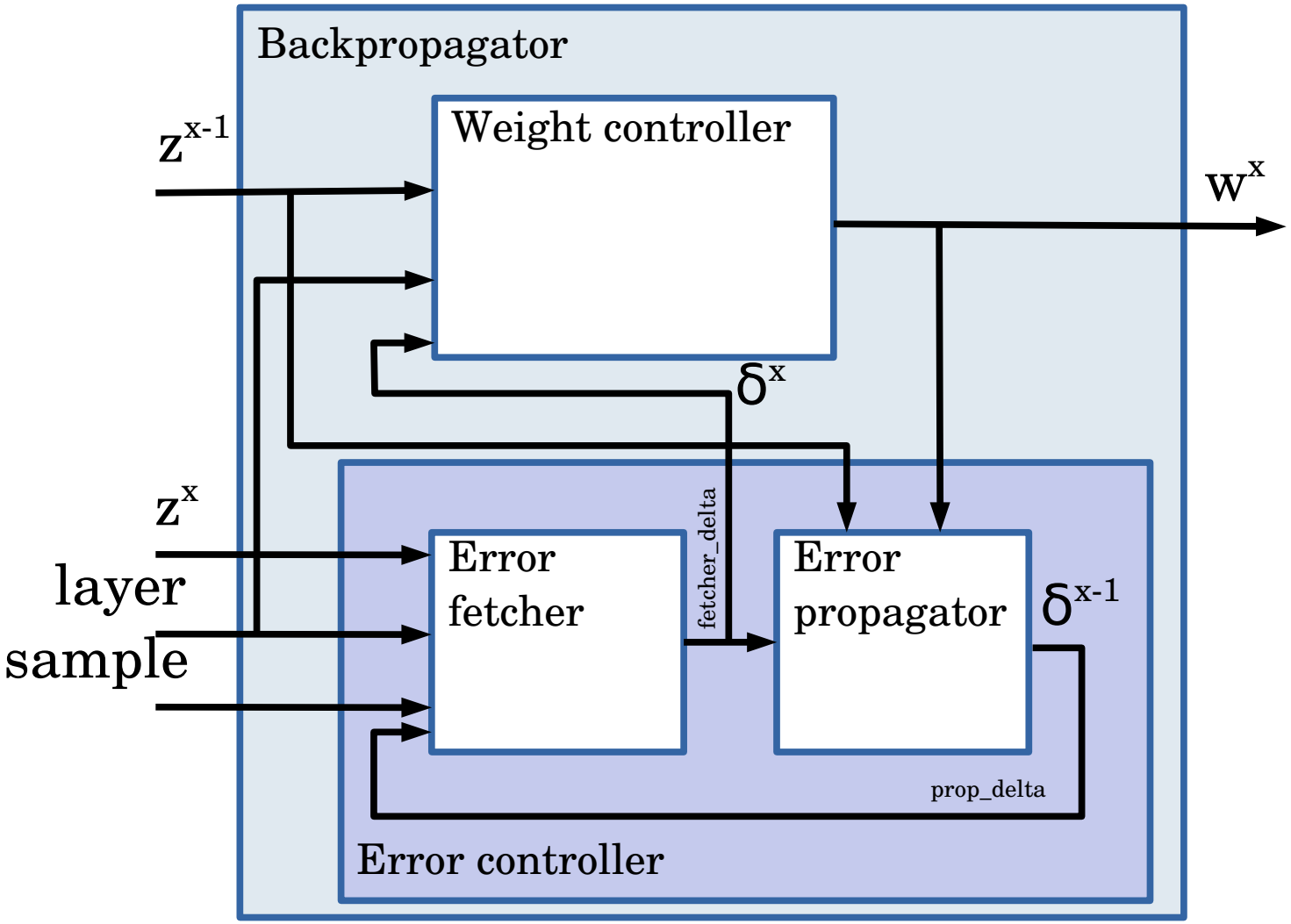
Activation stack is a parametrizable model for storing NN activations with one data and address input, and two outputs.
The data is received from the forward propagation module, and is read by the backpropagation module.
The two read buses read consecutive layer activations starting from the output layer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

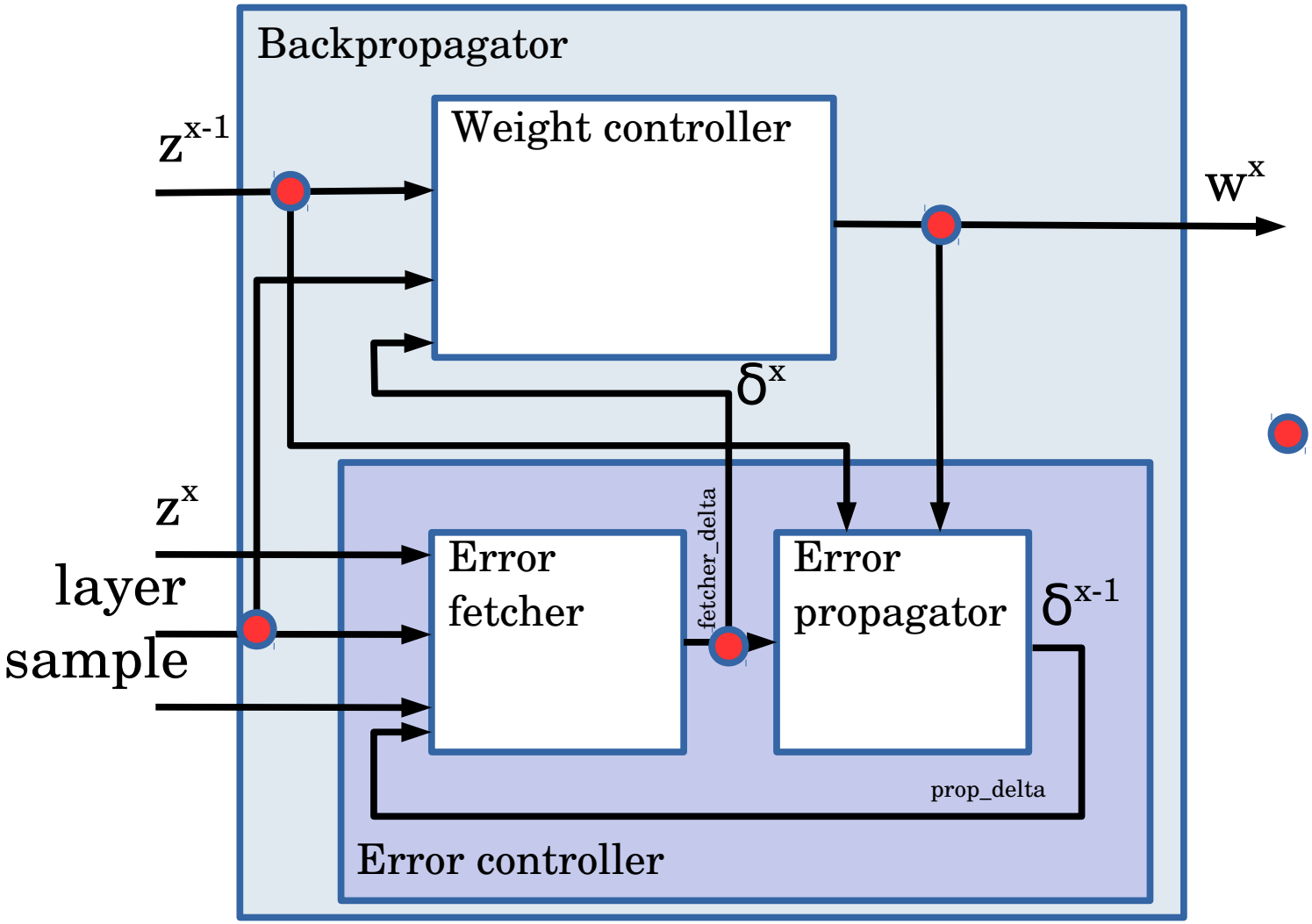
Backpropagator



Backpropagator

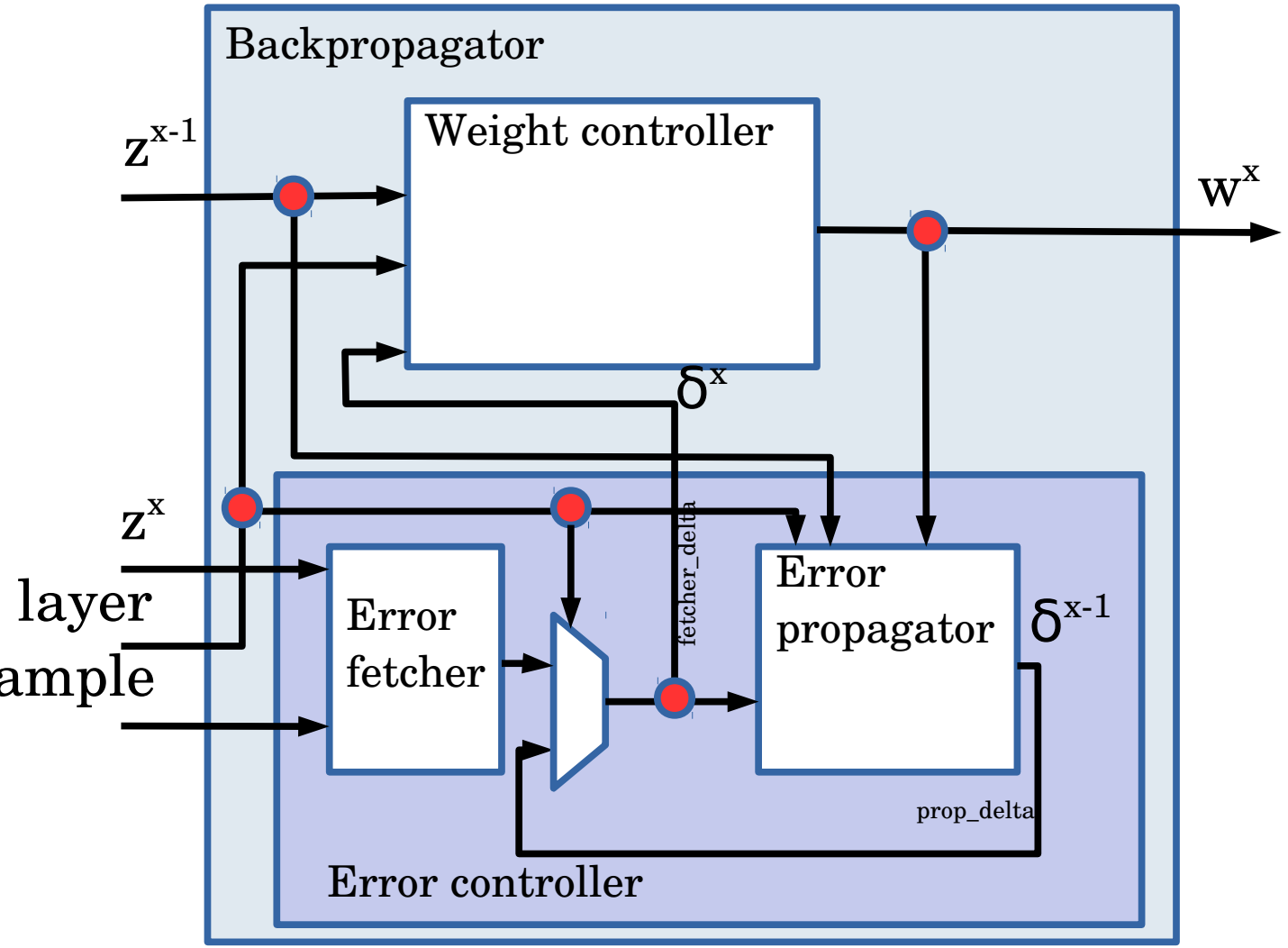


Backpropagator

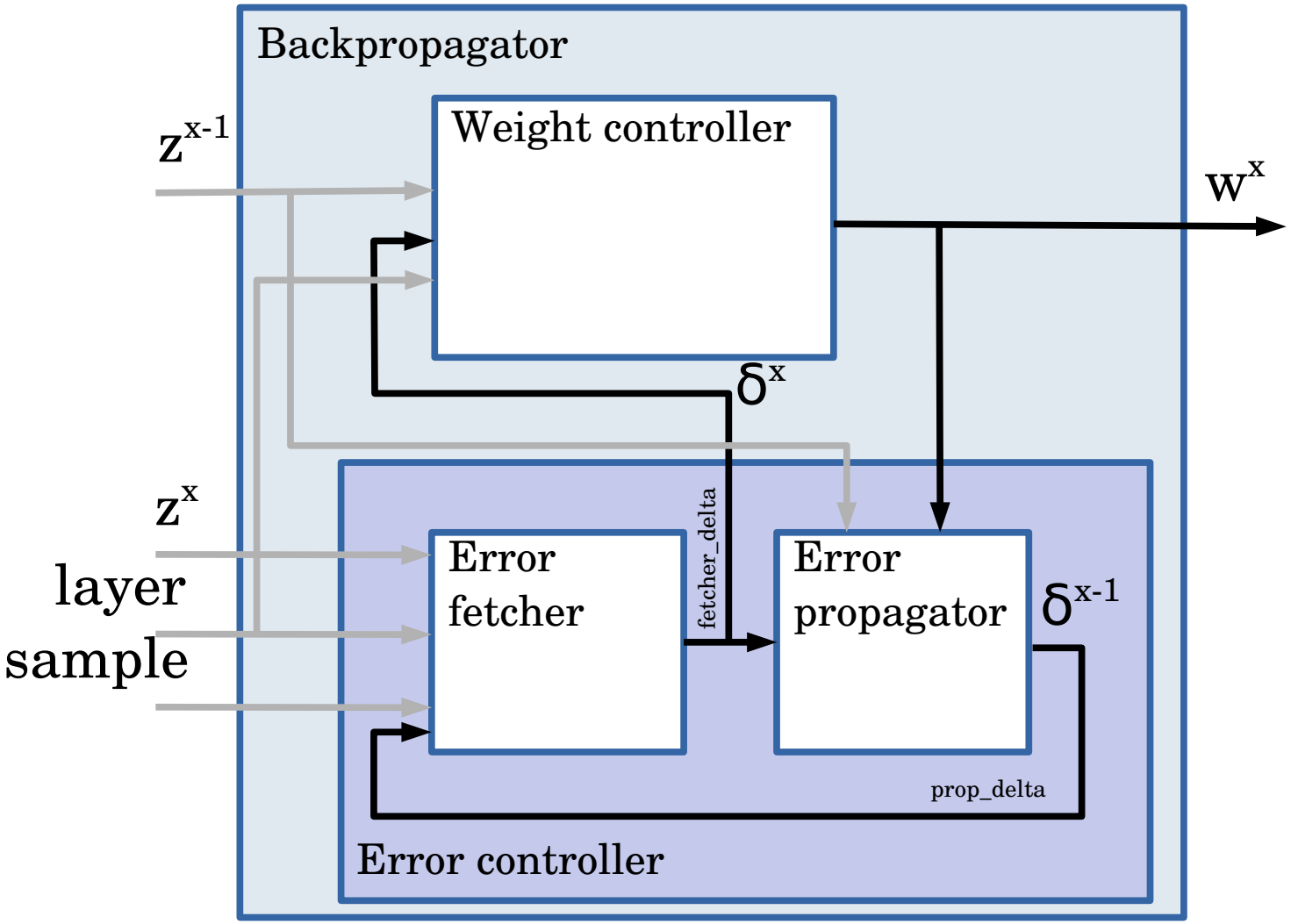


Denotes a split – the FIFO protocol needs to be careful and wait for both ready signals

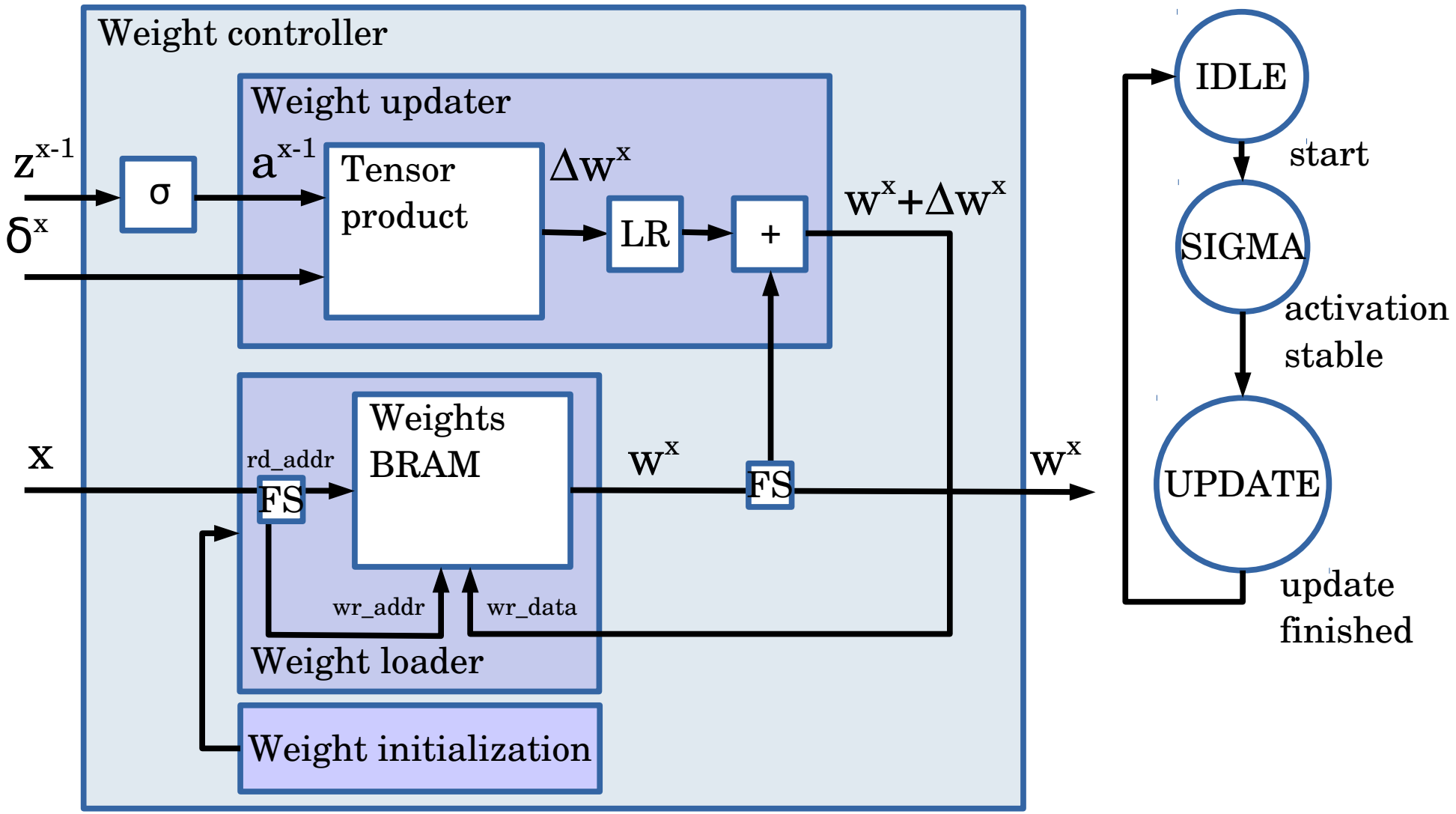
Backpropagator



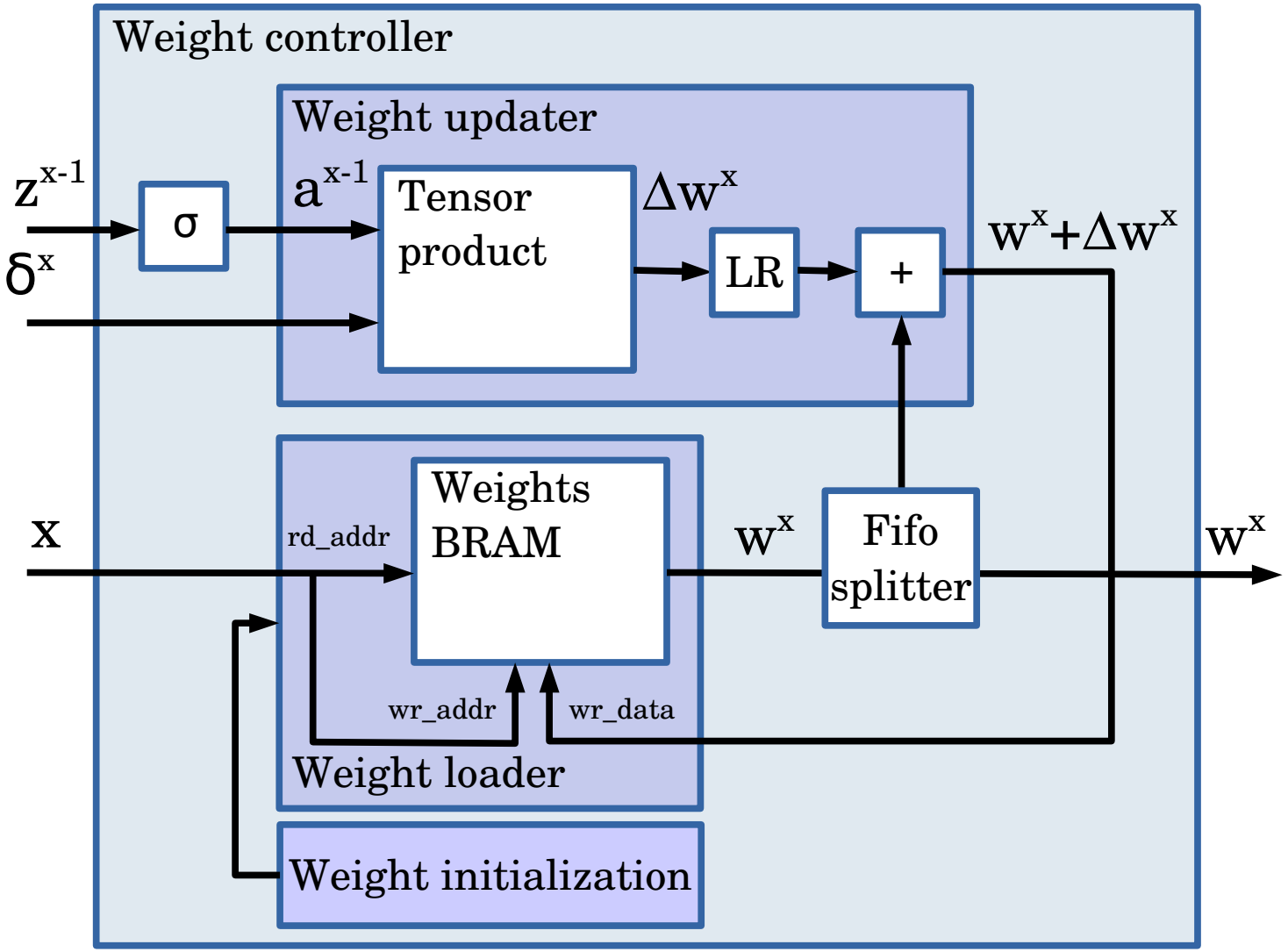
Backpropagator



Weight controller



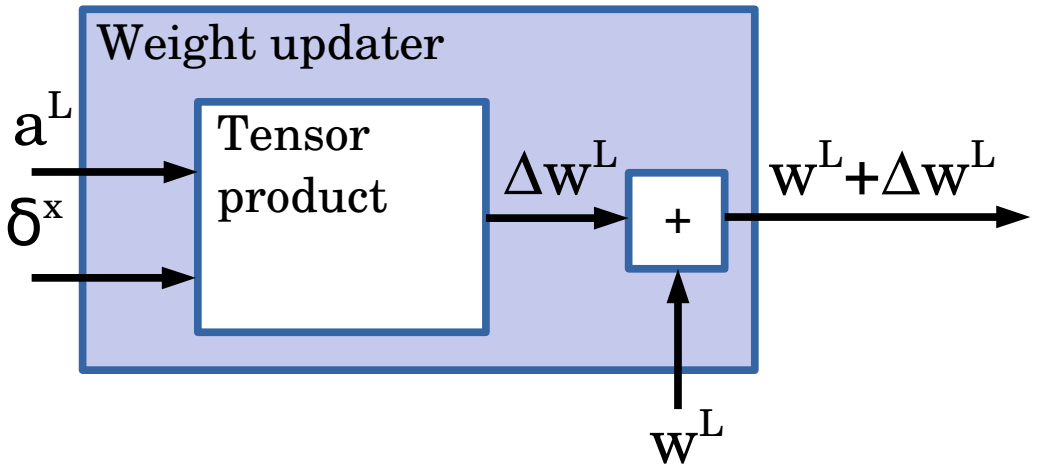
Weight controller

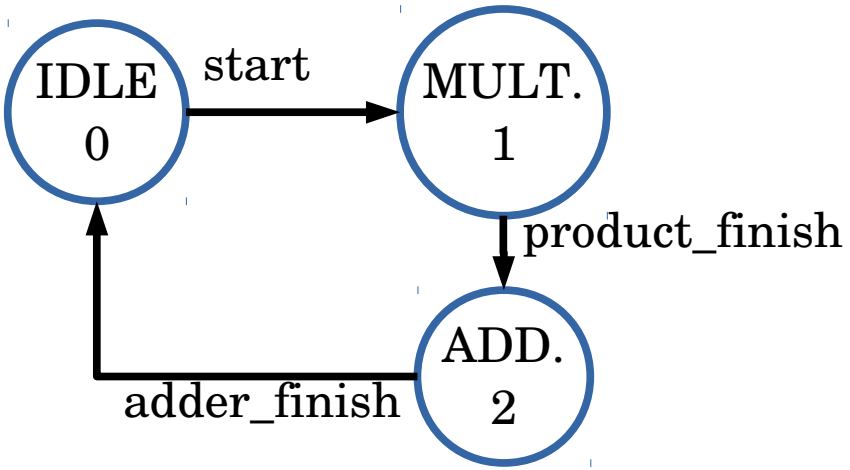


While the weights should be available always, we add a FIFO protocol to be able to distinguish whether we have worked on a new weight matrix, or the old one.

Weights bram only accepts a new input after it sent out an output

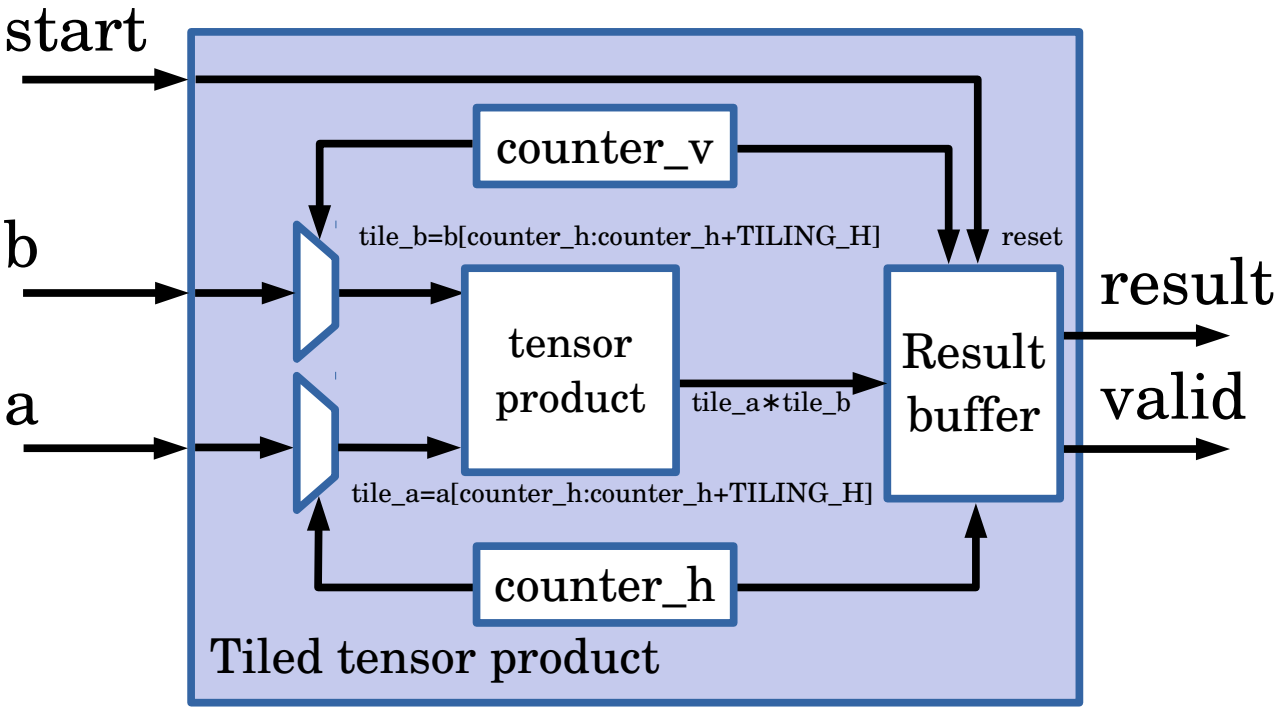
Weight updater





Weight updater receives the neuron activations and their errors, and computes $\Delta \boldsymbol{W}$ as a matrix product of $\boldsymbol{a}$ and $\boldsymbol{\delta}$. 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 $\boldsymbol{a}$ and a horizontal vector $\boldsymbol{\delta}$.

Tiled tensor product



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

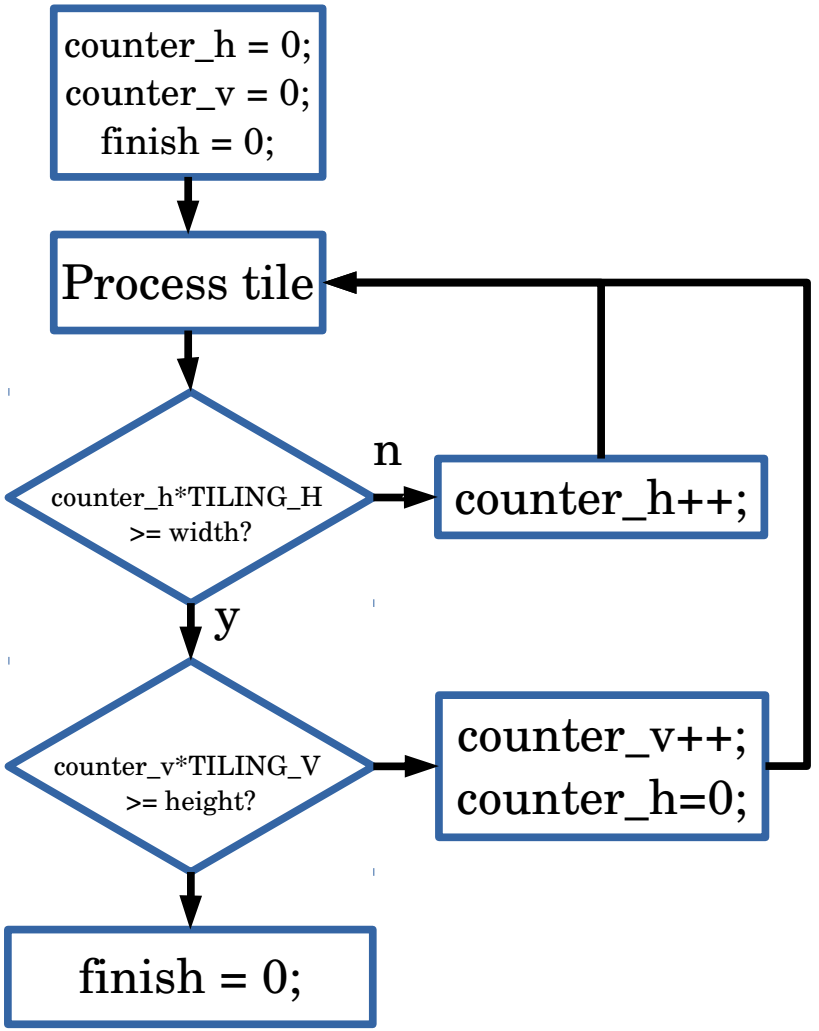
Tensor product is a generalized module that calculates the product of a vertical vector ***a*** and a horizontal vector ***b***, as seen in the picture. On start, the result vector is cleared, and can be used once valid is high.

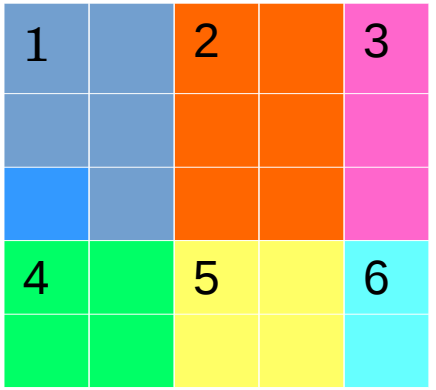
The number of multipliers and cycles can be controlled with the TILING_H and TILING_V parameters, which control the number of rows being processed in parallel and the number of elements processed per row at a time. The number of multipliers inferred can be then calculated as TILING_H*TILING_V.

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.

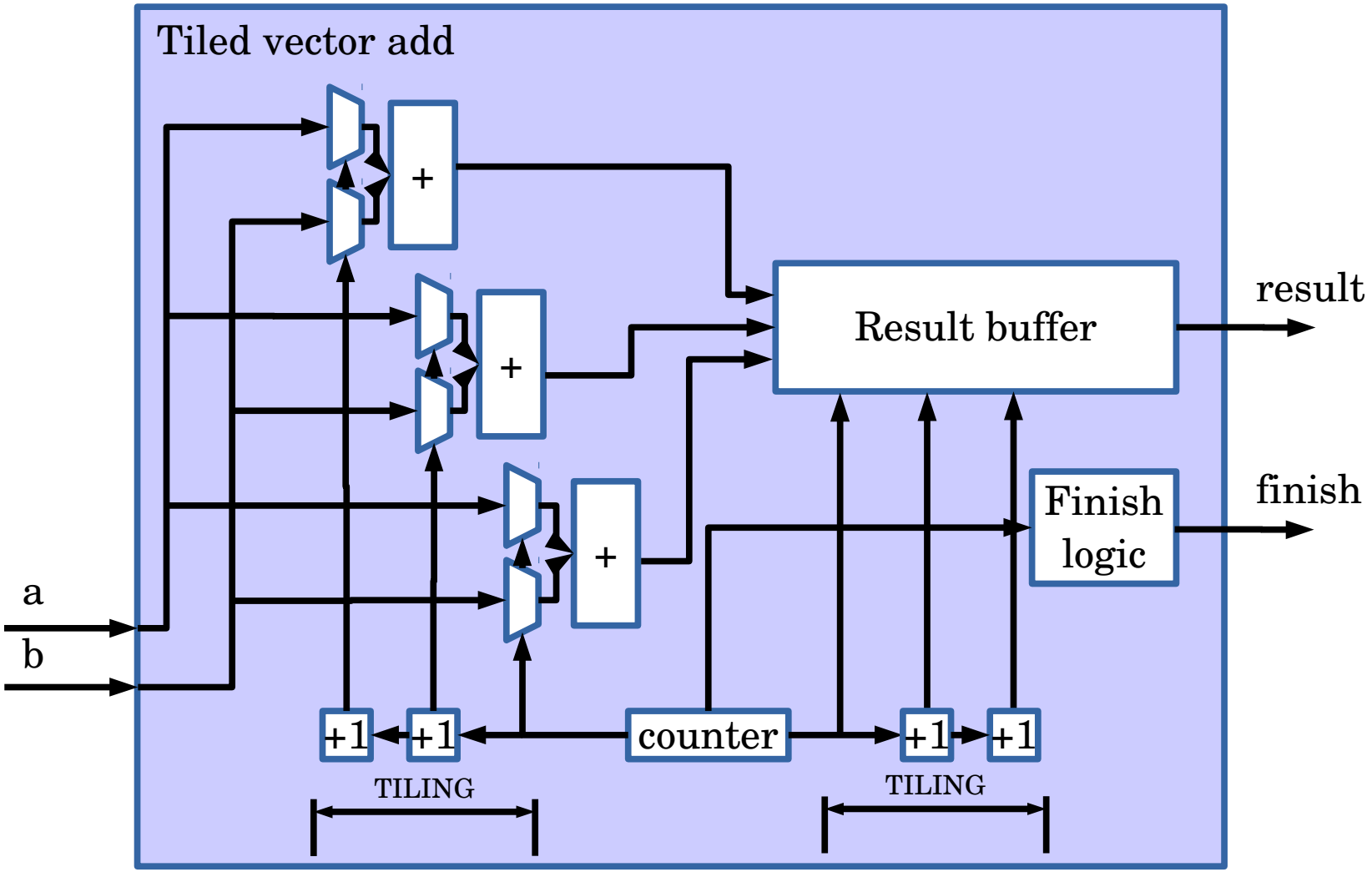
Tiled tensor product logic





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

Tiled vector add

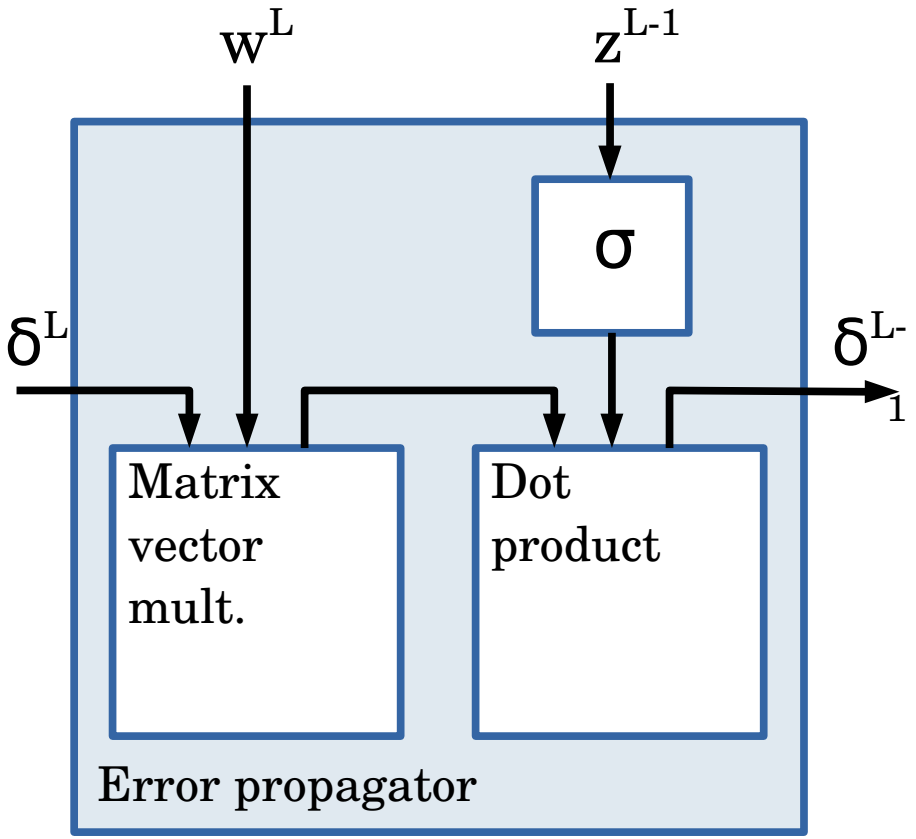


Tiled vector add sums two vectors ***a*** and ***b***. The **TILING** parameter controls how many adders are created, affecting the number of cycles needed to add the two vectors.

The size of the vectors is determined by A_CELL_WIDTH and B_CELL_WIDTH parameters. The fractions are A_

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

Error propagator

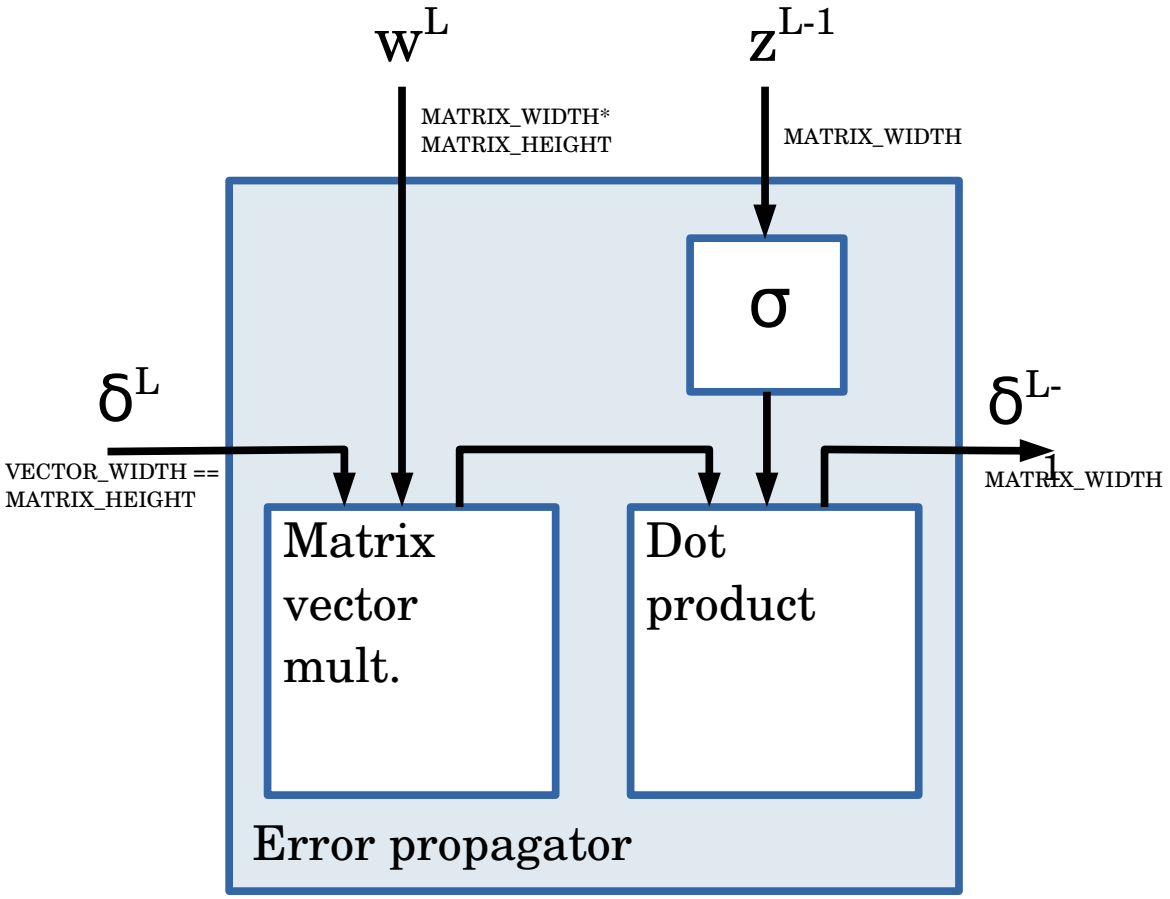


The error propagator passes errors through layers, going from top to bottom.

After the matrix and dot product, the width of the signal grows. How do we set the width of the signal without significantly raising the number of parameters?

By setting the width high enough and

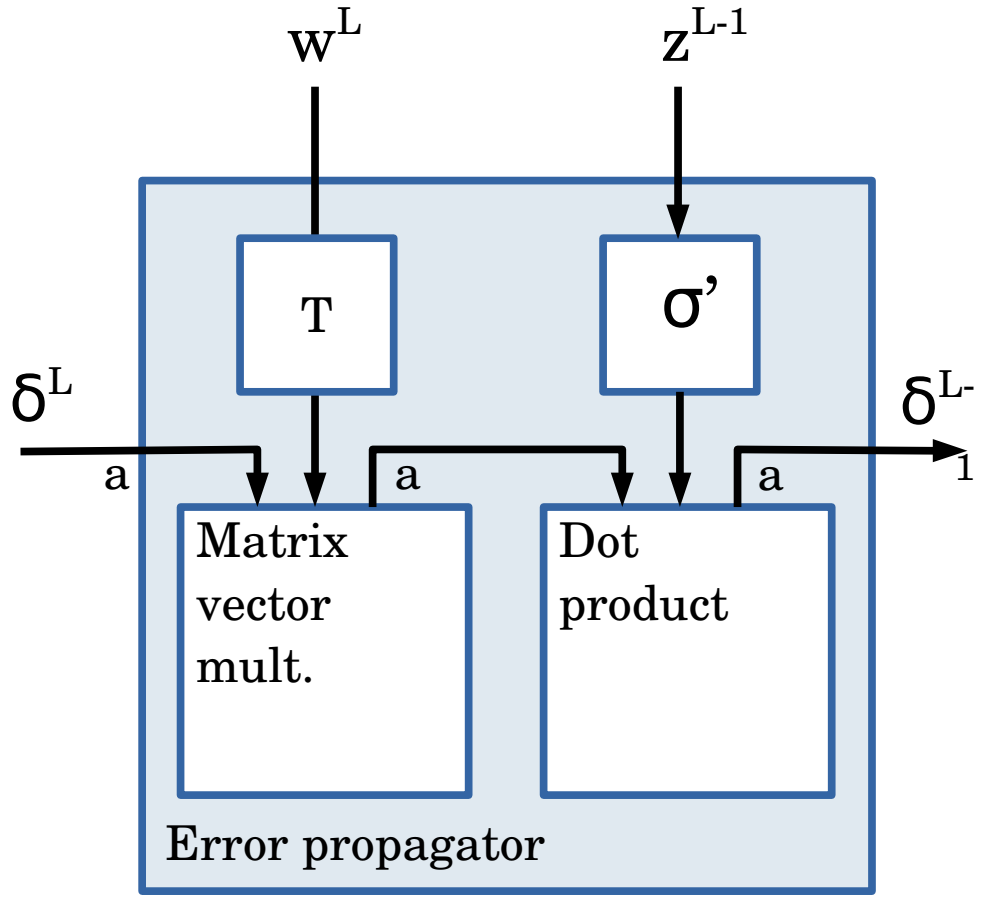
Error propagator



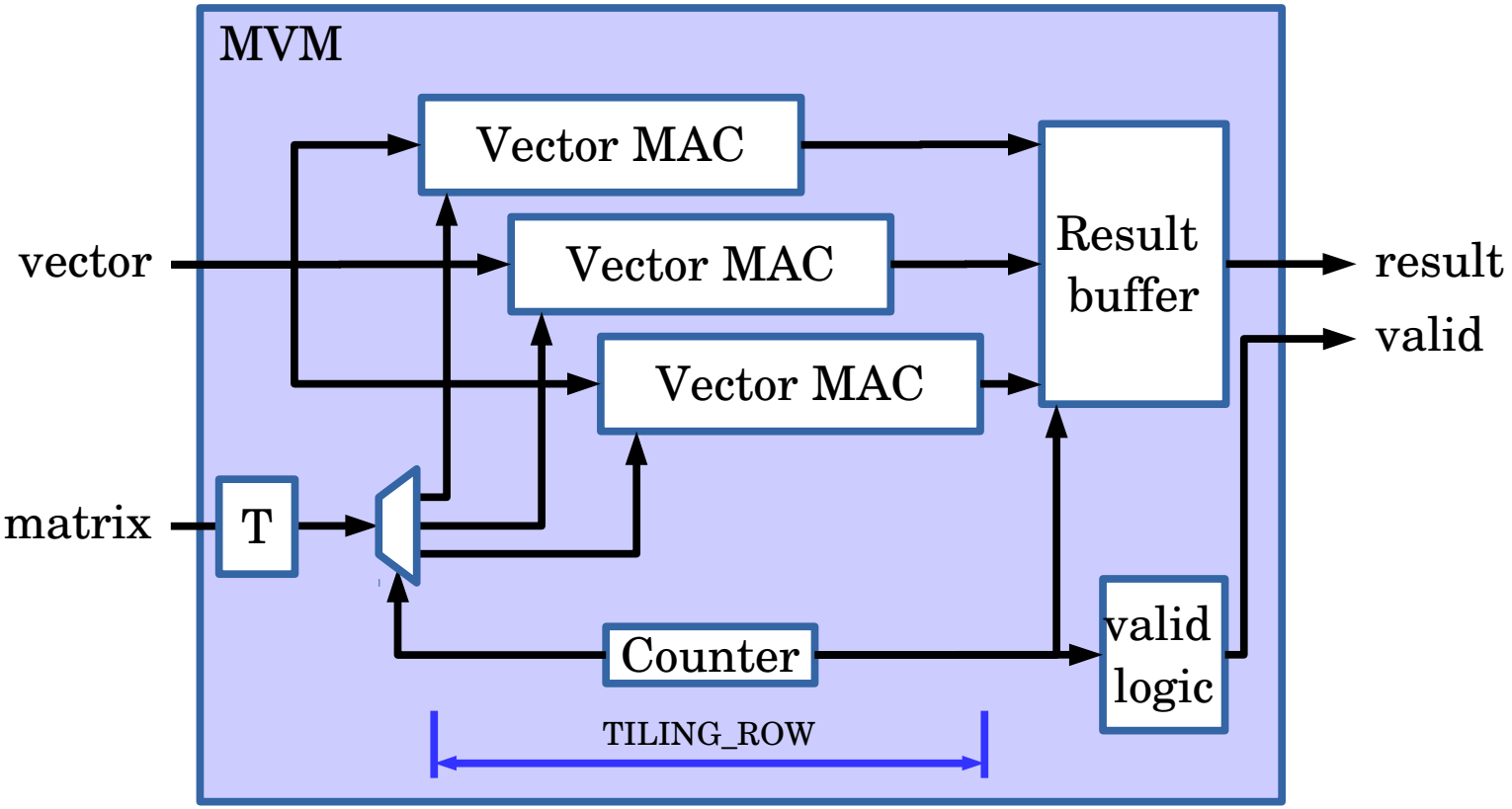
The error propagator passes errors through layers, going from top to bottom.

Since the input and output delta have the same cell width, we choose to prune to the same size after each module. **The output from the matrix vector multiplication and the dot product have the size of delta.**

Error propagator



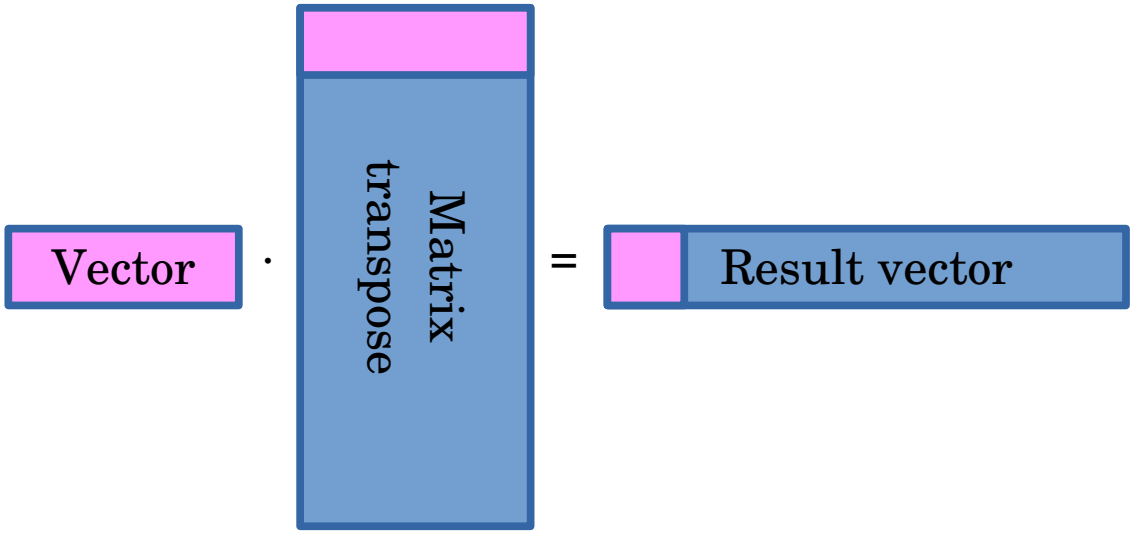
Matrix vector multiplication



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.

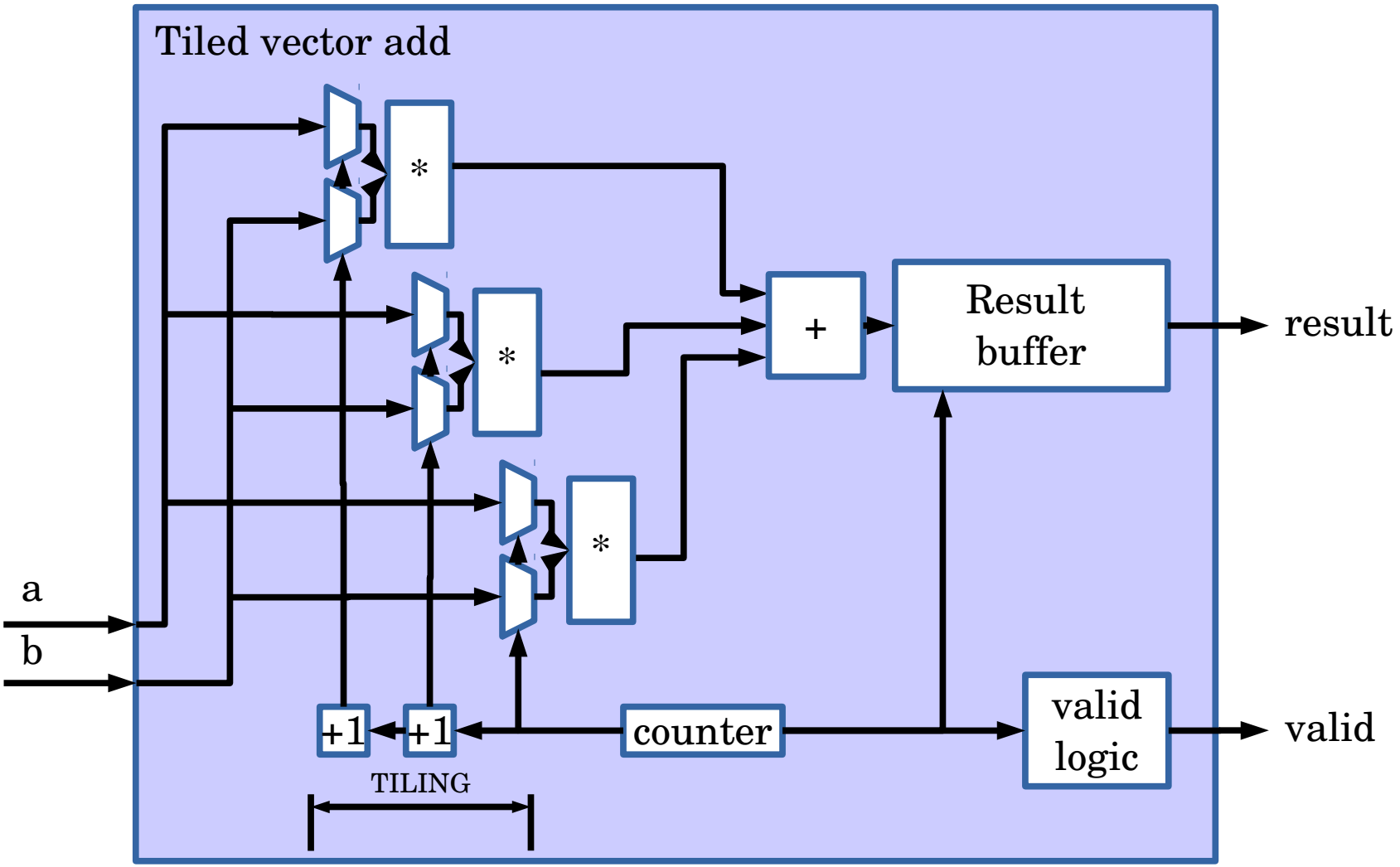
Matrix vector multiplication

<=>



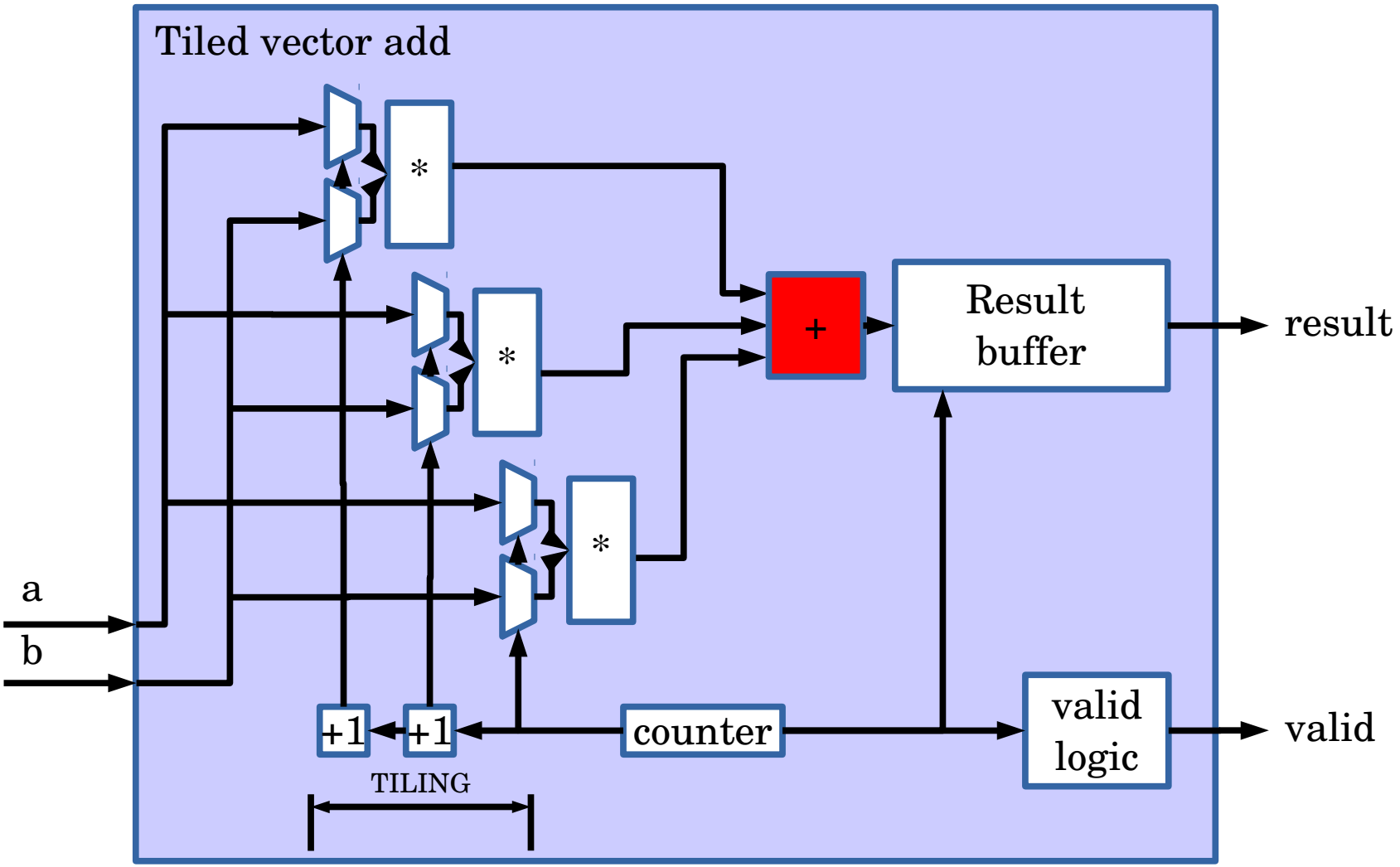
By transposing the matrix, we can perform the matrix vector multiplication by using a vector dot product on the matrix rows. The pink blocks in the image show an example.

Vector MAC



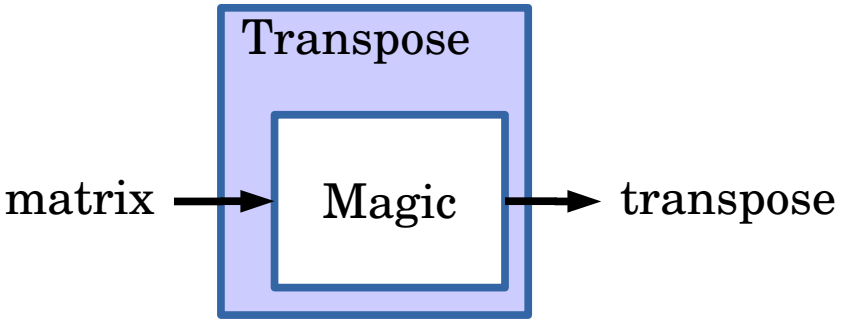
Vector MAC unit computes the sum of a dot product of two vectors.
The TILING parameter controls the number of multipliers in the module

Vector MAC



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

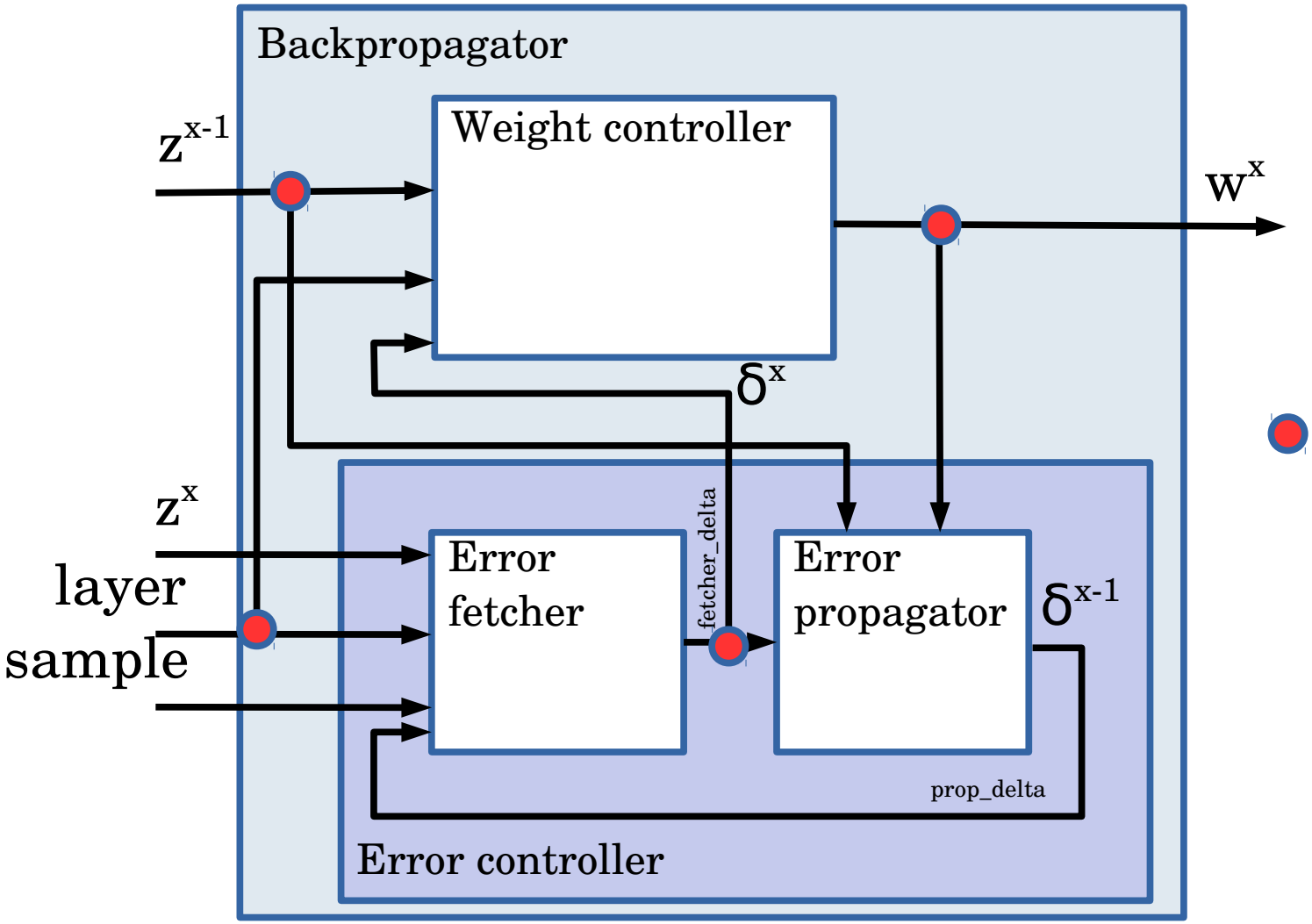
Transpose module



Transpose module takes a WIDTH*HEIGHT matrix of ELEMENT_WIDTH wide cells, and outputs a HEIGHT*WIDTH transposed matrix. The module is completely combinatorial.

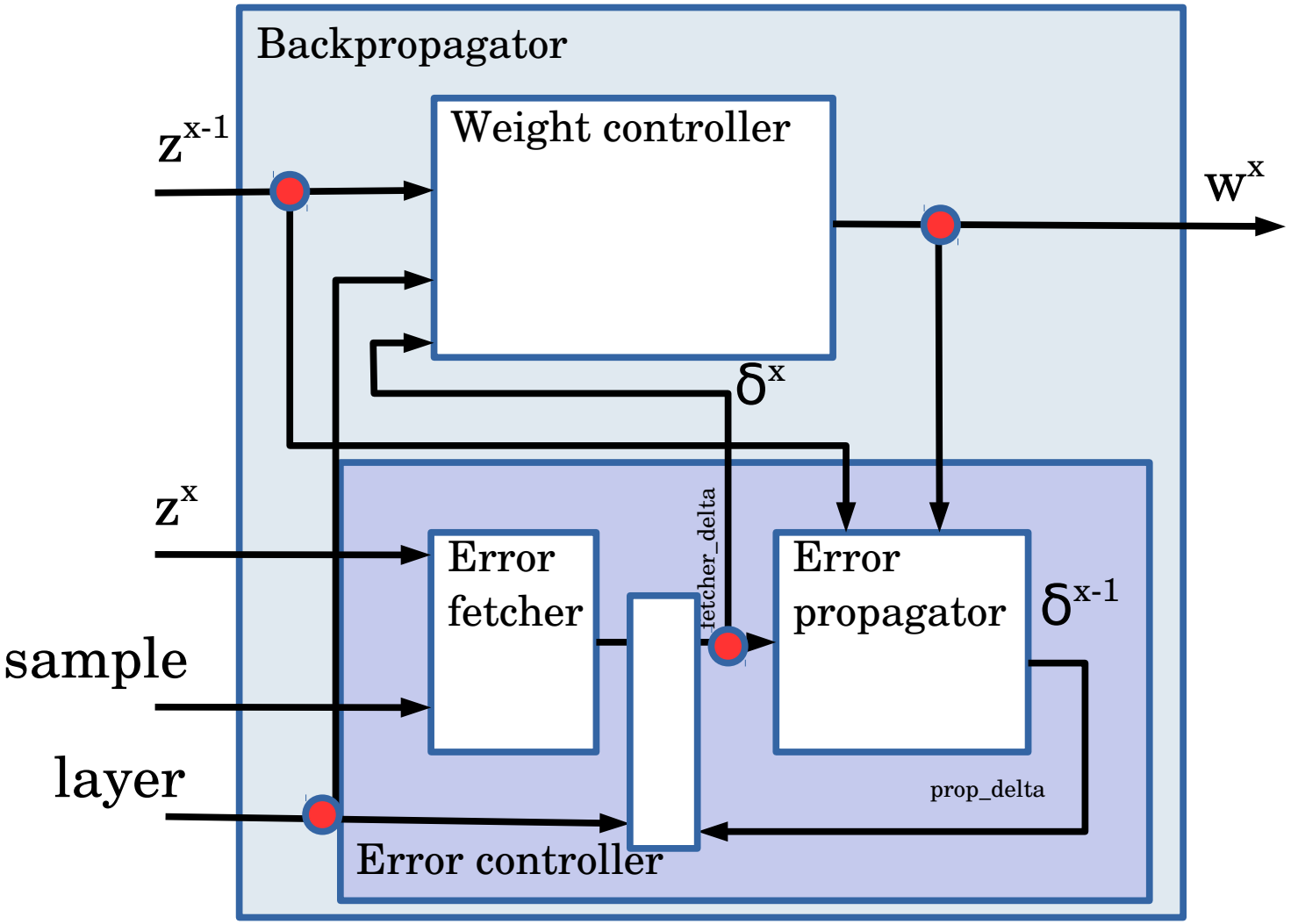
Perhaps we should clock this module?

Backpropagator



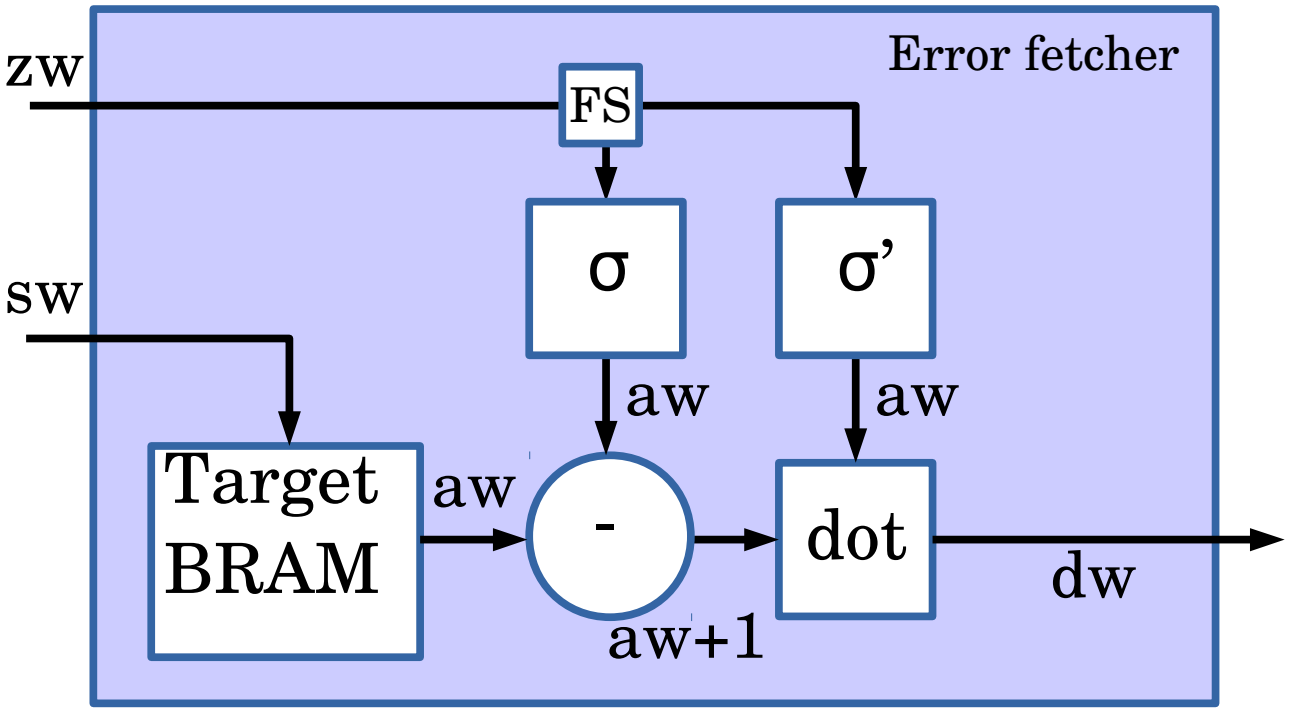
Denotes a split – the FIFO protocol needs to be careful and wait for both ready signals

Backpropagator - revised



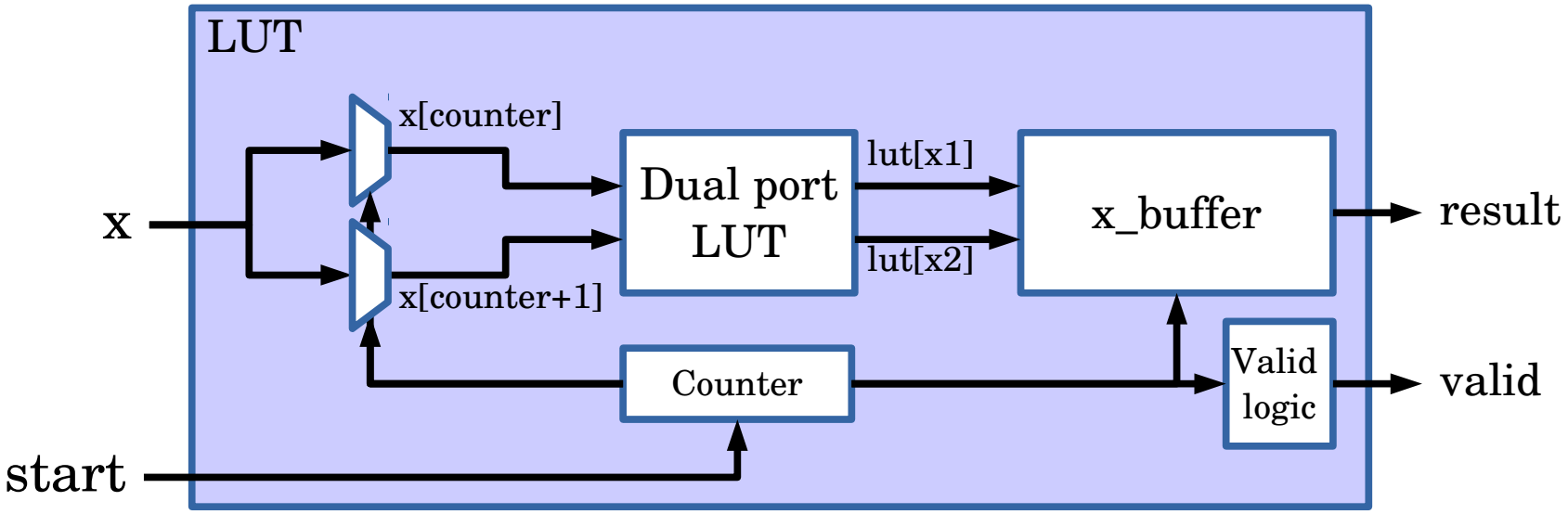
We don't need the last layer's delta, so it should be disregarded. We add the picker module to choose between the deltas from the fetcher and propagator, and disregard the last layer delta.

Error fetcher - widths



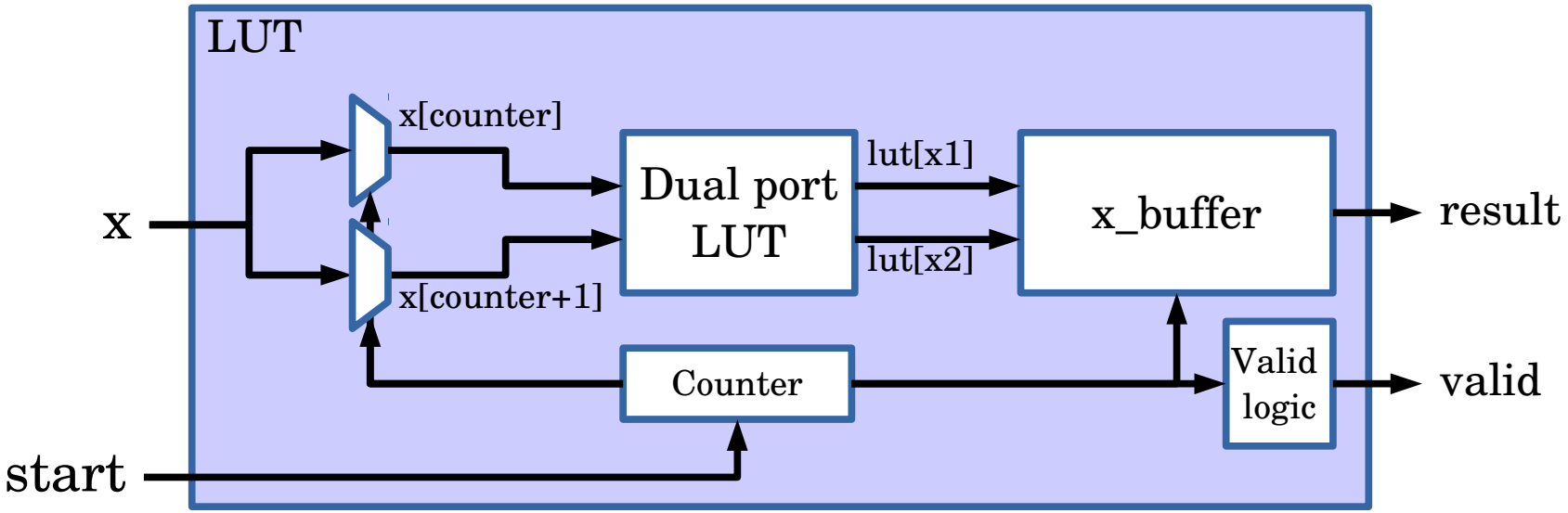
dw – delta width
zw – neuron output width
sw – sample address width
aw – activation width
lw – layer index width

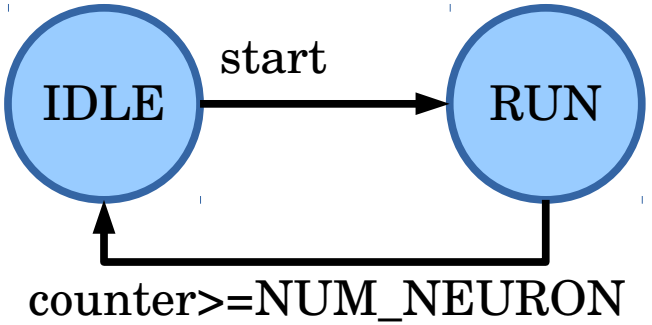
LUT



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.

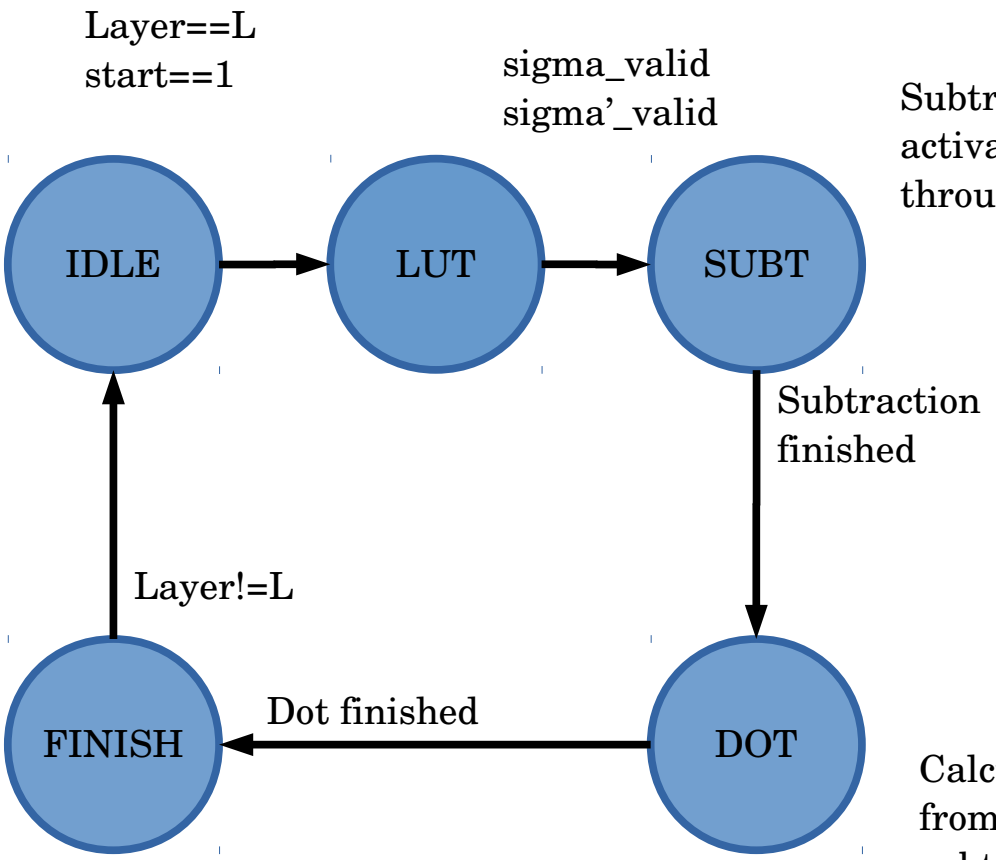
LUT


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


IDLE
start
RUN
counter>=NUM_NEURON

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.