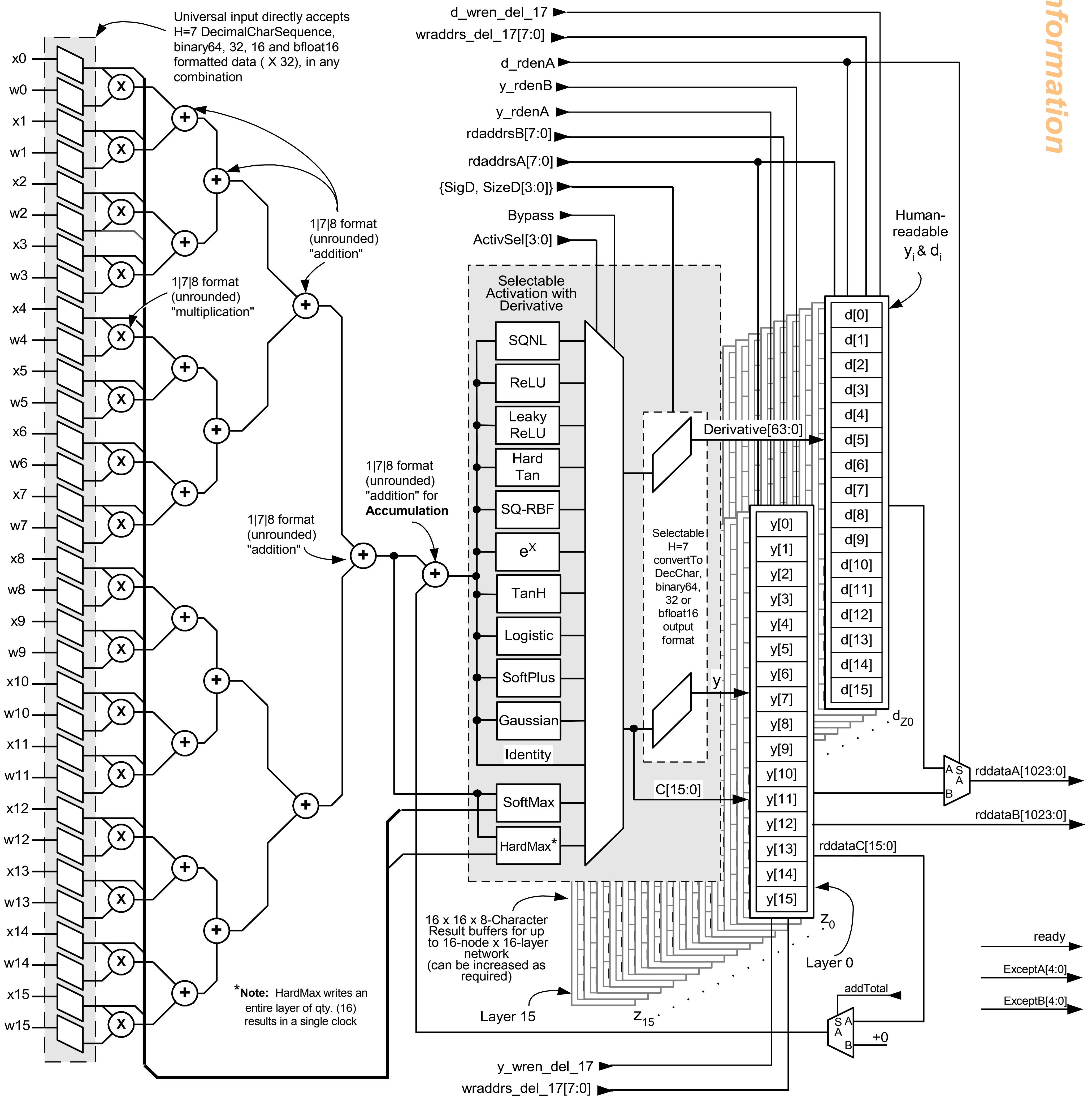
SYMPL Compute Engines

Fused Neural Network (FuNN) eNNgine
With Universal Human-Readable Inputs and Outputs

Created specifically for incorporation into the SYMPL 64-bit Universal Floating-Point Compute Engine, this memory-mapped Fused Neural Network (FuNN) is designed for ready implementation into the latest technology FPGAs. If you are exploring or experimenting with FPGA embeddable ANNs, this FuNN eNNgine is for you because it can compute directly with human-readable decimal character sequence representations, which makes for faster and more convenient debugging of your embedded FPGA design. *See* **Activations** on next page.

Features

- ❑ 16 Input X 16 Output X (up to) 16 Layer Neural Network with selectable activation (*see* reverse side). Bias, if any, consumes one input.
- ❑ Computes directly with human-readable "H=7" decimal character sequence representations, automatically converting them to 1|7|8 format for computations and then automatically converting outputs back to decimal character sequence, binary64, binary32 or bfloat16 before automatically spilling into one of 256 result buffers.
- ❑ Only 1 clock per 32-input node (x[15:0], w[15:0]) computation time (including activation, if any), equating to only 16 clocks per 16-node layer, using REPEAT instruction.
- ❑ Submit data directly from spreadsheets, text editors, etc., without explicit conversion to binary format beforehand and retrieve results in human-readable decimal character format.
- ❑ Fully pipelined fused neural network with up to 16 built-in automatic activation functions and their respective convergence functions. Intermediate results automatically spill into their respective result buffer every clock cycle when using REPEAT instruction.
- ❑ Accumulate mode allows pre or post-accumulation activation, which can be used to increase inputs per layer.
- ❑ Activation, when enabled, automatically computes, in a single clock, not only the selected activation, but also its corresponding derivative, each automatically spilling into their respective result buffers.
- ❑ 1|7|8 formatted intermediate results are automatically converted to either H=7 DecimalCharSequence, binary64, binary32 or bfloat16 format, as specified, immediately prior to spilling into their respective result buffers.
- ❑ Read and store a single or dual 64-bit node or an entire single or dual 1024-bit layer with a single-clock instruction.
- ❑ For intializing weights, the SYMPL FuNN eNNgine includes a novel psuedo-random number generator with selectable output range that can output pseudo-random numbers in "H=7" decimal character, binary64, binary32, or bfloat16 format every clock cycle.
- ❑ Final results, are automatically rounded only once at the final format conversion stage just prior to spilling into their respective result buffers. Accumulated intermediate results are never rounded.



SYMPL *Compute Engines*

Fused Neural Network (FuNN) eNNgine

Selectable Activations with Convergence Functions

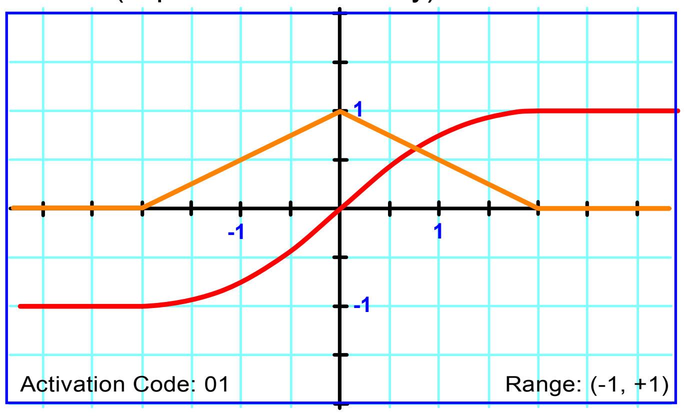
FuNN eNNgine Activations
Shown below are the built-in, single-clock, activations and their respective convergence functions. The SYMPL FuNN compute engine is fully pipelined and can accept a new 2048-bit input (x[63:0], w[63:0]) * 16 every clock cycle. The FuNN eNNgine can be selectively programmed to automatically activate each intermediate result with one of the 10 activations presently available in its repertoire shown below. This activation computation, along with its corresponding convergence function/derivative, consumes only one clock in the pipeline.

The activation for a given push into the neuron is specified with bits 11 thru 14 of the SYMPL compute engine STATUS Register using a single instruction just prior to the push of data into the neuron. Once specified, all subsequent computations will be activated using the specified activation prior to such results automatically spilling into the specified result buffer. The instruction set has a means to bypass the current activation specification on an instruction by instruction basis, providing a means to accumulate results over time before activation, allowing for easy concatenation of vectors.

SQNL (Square Non-Linearity)

$$f(x) = \begin{cases} 1 & : x > 2 \\ x - \frac{x^2}{4} & : 0 <= x <= 2 \\ x + \frac{x^2}{4} & : -2 <= x < 0 \\ -1 & : x < -2 \end{cases}$$

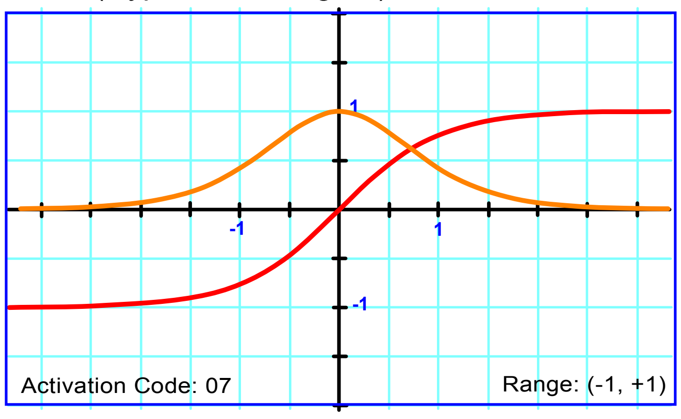
$$f'(x) = \begin{cases} 0 & : x > 2 \\ 1 - \frac{x}{2} & : 0 <= x <= 2 \\ 1 + \frac{x}{2} & : -2 <= x < 0 \\ 0 & : x < -2 \end{cases}$$



TanH (Hyperbolic Tangent)

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

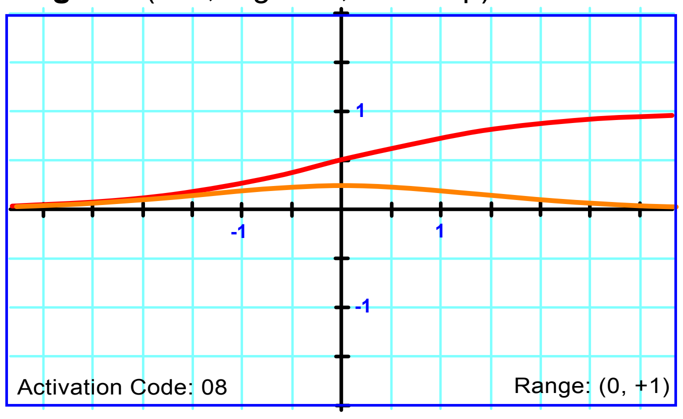
$$f'(x) = 1 - f(x)^2$$



Logistic (aka, Sigmoid, SoftStep)

$$f(x) = \frac{1}{1 + e^{-x}}$$

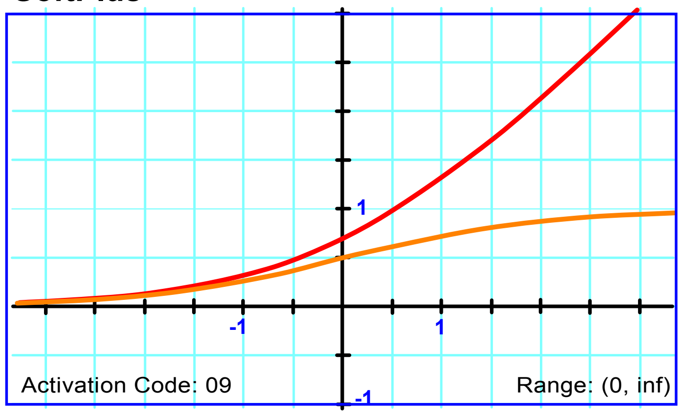
$$f'(x) = f(x)(1-f(x))$$



SoftPlus

$$f(x) = \ln(1 - e^{x})$$

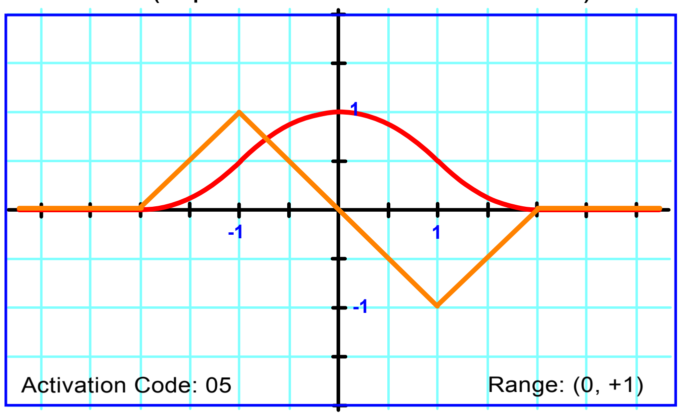
$$f'(x) = \frac{1}{1 + e^{-x}}$$



SQ-RBF (Square Radial Basis Function)

$$f(x) = \begin{cases} 1 - \frac{x^2}{2} & : -1 <= x <= 1 \\ \frac{2 - x^2}{2} & : 1 < |x| < 2 \\ 0 & : \text{otherwise} \end{cases}$$

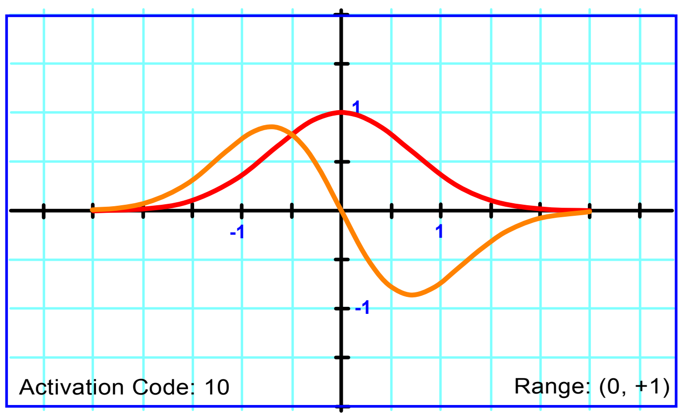
$$f'(x) = \begin{cases} -x & : |x| <= 1 \\ x\text{-}2 & : 1 < x < 2 \\ 2\text{+}x & : -2 < x < -1 \\ 0 & : \text{otherwise} \end{cases}$$



Gaussian

$$f(x) = e^{-x^2}$$

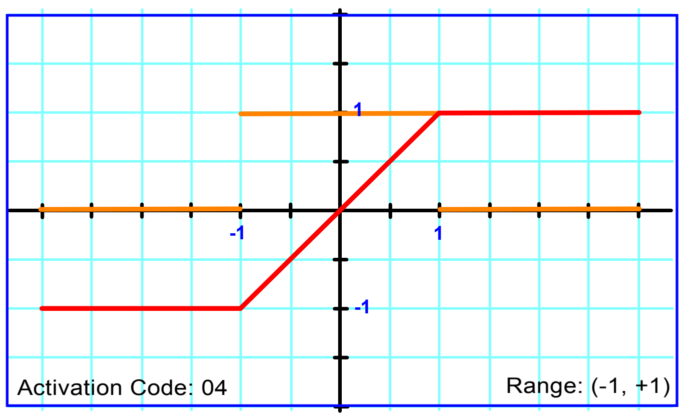
$$f'(x) = -2xe^{-x^2}$$



Hard Tan

$$f(x) = \begin{cases} -1 & : x < -1 \\ x & : -1 <= x <= 1 \\ 1 & : x > 1 \end{cases}$$

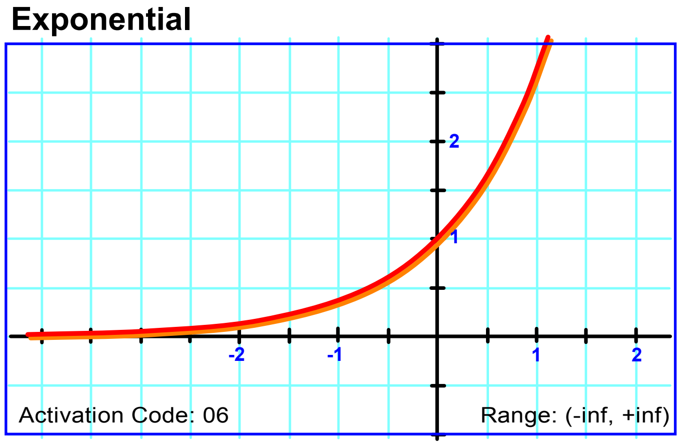
$$f'(x) = \begin{cases} 1 & : -1 <= x <= 1 \\ 0 & : \text{otherwise} \end{cases}$$



Exponential

$$f(x) = e^{x}$$

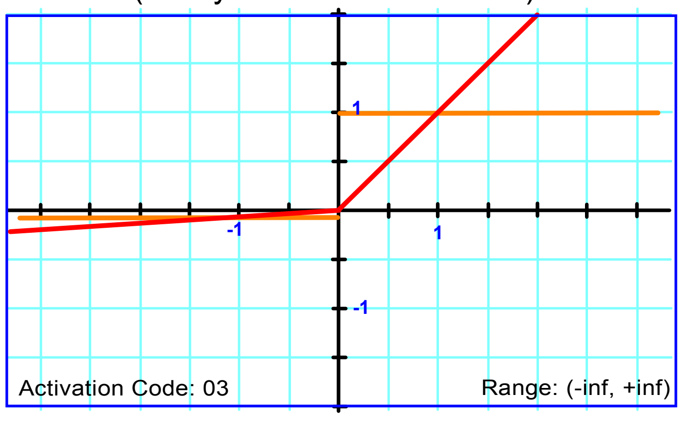
$$f'(x) = e^{x}$$



LReLU (Leaky Rectified Linear Unit)

$$f(x) = \begin{cases} .01x & : x < 0 \\ x & : x >= 0 \end{cases}$$

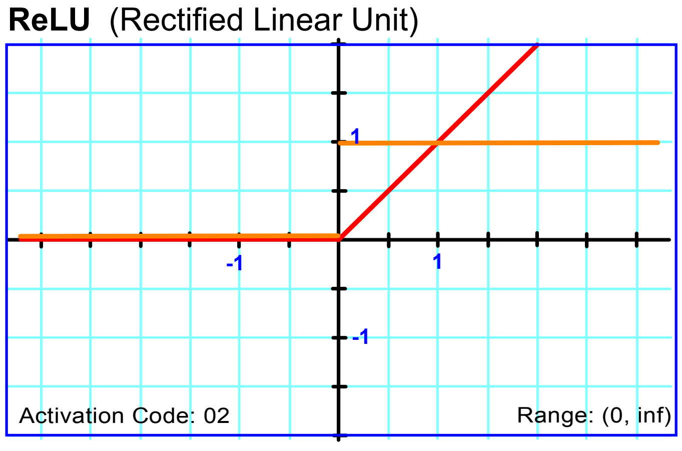
$$f'(x) = \begin{cases} .01 & : x < 0 \\ 1 & : x >= 0 \end{cases}$$



ReLU (Rectified Linear Unit)

$$f(x) = \begin{cases} 0 & : x <= 0 \\ x & : x > 0 \end{cases}$$

$$f'(x) = \begin{cases} 0 & : x <= 0 \\ 1 & : x > 0 \end{cases}$$



SoftMax

$$f(x_i) = \frac{e^{x_i}}{\sum_{i=0}^{15} e^{x_i}}$$

Requiring three steps to determine a result, this activation function is commonly employed in classification neural network models to compute the probabilities with respect all the outputs of a given layer. The first step is to individually compute the exponential (e^x) of each node/cell in the selected layer using the exponential activation function (code: 06) . Once all the nodes in the selected layer have been exponentiated, the entire layer is then summed together by reading the entire layer out and pushing it back into the neural network using no accumulation or activation. As the third and final step, each individual exponentiated node is then divided by the previously computed sum, forming the SoftMax output layer. Note that the “SoftMax” box in the box diagram is actually a general-purpose floating-point “division” operator.

Below is an example source listing that can be used to compute a SoftMax activation of a given layer using only 16 instructions:

```
;--------------------------------------
; SoftMax computation
;--------------------------------------
    _   _4:AR0 = _4:#layer3      ;get pointer to layer 3 (the 4th layer)
    _   _4:AR1 = _4:#_1.0        ;weights for the exponential push are all 1.0
    _   _4:AR2 = _4:#layer4      ;place exponentiated results in layer 4 (the 5th layer)
    _   _1:actMode = _1:#Exp     ;specify exponential as the intermediate activation function
    _   _2:REPEAT = _2:#15
    act   s8:*AR2++[1] = (s8:*AR0++[1], s8:*AR1++[1])  ;compute 16 exponents

    _   _4:AR0 = _4:#layer4      ;point to the intermediate layer containing all exponentials of layer3
    _   _4:AR1 = _4:#_1.0        ;still all weights for this intermediate step are still 1.0
    _   _4:AR2 = _4:#layer5      ;point to the first node of layer5 the summation of all the exponentials
    _   s8:*AR2++[0] = (s128:*AR0++[0], s128:*AR1++[0]) ;do summation without any activation or accumulation

    _   _4:AR0 = _4:#layer5      ;point to the sum of all the exponentials in the first location of layer 5
    _   _4:AR1 = _4:#layer4      ;point to the first exponentiated result in layer 4
    _   _4:AR2 = _4:#layer6      ;point to layer 6  where results of each divide (probability) will be stored
    _   _1:actMode = _1:#SoftMax    ;set intermediate activation mode code to SoftMax (division) e^xi/sum(e^xi)
    _   _2:REPEAT = _2:#15
    act  s8:*AR2++[1] = (s8:*AR1++[1], s8:*AR0++[0]) ;divide each exponential summation
```

HardMax

$$f(x_i) = \max_i\left(\frac{e^{x_i}}{\sum_{i=0}^{15} e^{x_i}}\right) \text{ ? " } \quad 1.0\text{" : " } \quad 0.0\text{"}$$

Using the previously computed SoftMax result vector, with a single push back into neural network, HardMax simultaneously computes a “hard” “1.0” (true) or “0.0” (false) for each node/cell of the SoftMax result intermediate layer and stores the resulting vector in the final output layer. Like the other activation modes, HardMax results can be stored in human-readable decimal character sequence, binary64, binary32 or bfloat16 format, such being specified by the SigD and SizeD of the instruction that pushes the SoftMax result vector back into the neural network with activation code = 12.

Simultaneous with the HardMax results being stored in the layer number specified by the instruction that performs the single push, a 64-bit concatenation of the previously computed exponential with the highest probability in binary32 format, the too-hot representation in 16-bit integer format, and the one-of-16 one-hot in 16-bit integer format is stored in the HardMax derivative memory that shadows the HardMax activation layer. Like all the other activations, the derivative storage layers are offset by 256 (0x100). Thus, after performing the HardMax activation operation, to access the corresponding derivative, simply perform a read of the the HardMax activation layer, but offset by 256 (0x100). For instance, if the HardMax activation results are stored in Layer 7, simply add 256 to the address of layer7 (e.g., use layer7+256) as the effective address.

Once the above SoftMax computation instruction sequence is executed, to perform a HardMax activation of the SoftMax results, use the following HardMax instruction sequence, which is only 4 instructions, effectuating a single push into the neural network:

```
;--------------------------------------
; HardMax computation
;--------------------------------------
    _   _4:AR0 = _4:#layer6     ;point to start of SoftMax result vector
    _   _4:AR1 = _4:#_1.0       ;all weights for this intermediate step are still 1.0
    _   _4:AR2 = _4:#layer7     ;point to layer7 where the results of Hardmax will be stored
    _   _1:actMode = _1:#HardMax

    act s128:*AR2++[0] = (s128:*AR0++[0], s128:*AR1++[0])
```

As mentioned above, the described concatenation is simultaneously written to the corresponding derivative storage layer. For a hypothetical example, suppose the node/cell in the SoftMax intermediate output layer is .8903452 (89%) and it is stored in node 11 of layer6, the described concatenation stored in the corresponding derivative memory location can be retrieved by performing a 64-bit read of location layer7+256+11. The concatenated result will contain the following value when read:

{0x0800, 0x0800, 0x7F618000} ; the positioning is--{too-hot, one-hot, probability}

Note that the “too-hot” representation is included in the concatenation to provide a means for determining which, if any, other nodes have the same value. In such cases, its corresponding bit will also be set to one in the “too-hot” representation.

Example SYMPL FuNN eNNgine Classification Routine

The following is an example of an actual 16-input by 16-object classification routine that uses the FuNN eNNgine. It employs a single input layer with TanH activation. The output layer comprises a SoftMax activation layer followed by a HardMax activation layer. The SoftMax activation comprises three steps: 1) exponentiation of all 16 nodes/cells in the TanH layer, 2) summation of all exponentials, 3) division of each of the 16 exponentials by the summation. As the final step in the classification computation, all 16 results of the SoftMax layer are simultaneously pushed back into the neural network with HardMax activation set. All 16 results of the HardMax are simultaneously computed and stored in the final layer with only one of the results being a literal "1.0" to indicate maximum probability over the remaining 15 other nodes, which will all be "0.0".

In summary, the following routine employs only 30 instructions to classify qty. (16) objects, with each object comprising qty. (16) inputs. The entire process consumes only 2310 clocks to complete. Classification of a single 16-input object consumes only 144 clocks. Every clock cycle there are 18 floating-point multiplies, 37 floating-point additions, 2 floating-point divisions, 5 exponentials and 256 floating-point compares. In addition, there are 32 convertFromDecimalCharacter conversions and two convertToDecimalCharacter conversions—every clock cycle.

```
0000010C 307CDC0000000010       classify:  _   _1:clrDVNCZ = _1:#DoneBit ;clear Done bit
0000010D 327FF72000013880                  _   _4:AR0 = _4:#_1.0         ;get pointer to vector where the "1.0" goes
0000010E 317FEF100000000F                  _   _2:REPEAT = _2:#15
0000010F 2380403000200000                  _   _8:*AR0++[8] = _8:@one    ;generate a vector of qty (16) "1.0"  as constant
00000110 0000000000000000                  _
00000111 327FFD2000010080                  _   _4:AR6 = _4:#outBuffer    ;get pointer to output buffer location
00000112 017FED17FFB00000                  _   _2:LPCNT0 = _2:AR4        ;AR4 already contains # of objects to classify,
                                                                         ;as Host CPU pushed it in from test bench
00000113 327FF72000012880                  _   _4:AR0 = _4:#object0      ;point to first object vector X
                               ;---------------------------------------
                               ; input layer (layer0) computation using TanH activation
                               ;---------------------------------------
00000114 327FF82000013080       loop:      _   _4:AR1 = _4:#obj0Lay0Wt   ;point to first weight vector W
00000115 327FF92000004000                  _   _4:AR2 = _4:#layer00      ;point to first layer results (layer0)
00000116 307FCD0000000007                  _   _1:actMode = _1:#TanH     ;set activation mode
00000117 317FEF100000000F                  _   _2:REPEAT = _2:#15
00000118 8B800AF8000F8401                  act  s8:*AR2++[1] = (s128:*AR0++[0], s128:*AR1++[128])  ;run all the weight W
00000119 0000000000000000                  _                             ;vectors against the current object X vector
0000011A 0200002840000000                  _   _4:0 = _4:*AR0++[128]     ;bump AR1 by 128 to point to next weight
                               ;---------------------------------------
                               ; SoftMax process
                               ;---------------------------------------
                                   ;---------------------------------------
                                   ; second layer (layer1) is intermediate output layer where exponentials are stored
                                   ;---------------------------------------
0000011B 327FF82000004000                  _   _4:AR1 = _4:#layer00      ;point to first layer results (layer0)
0000011C 327FFA2000013880                  _   _4:AR3 = _4:#_1.0         ;point to vector of 1.0s for use as exp weights
0000011D 307FCD0000000006                  _   _1:actMode = _1:#Exp      ;set activation mode to exponential
0000011E 317FEF100000000F                  _   _2:REPEAT = _2:#15        ;AR2 is already pointing to layer1
0000011F 8B800AB8009B8003                  act  s8:*AR2++[1] = (s8:*AR1++[1], s8:*AR3++[0]) ;exponentiate all the results
00000120 0000000000000000                  _                             ; in layer0 and store in layer1
                                   ;---------------------------------------
                                   ; summation of all the exponentials
                                   ;---------------------------------------
                                                                         ;AR1 is already pointing to layer1
                                                                         ;AR3 is already pointing to 1.0 vector
                                                                         ;AR2 is already pointing to layer2
00000121 0B8082F8081F8003                  _   s8:*AR2++[16] = (s128:*AR1++[16], s128:*AR3++[0]) ;store single result in
00000122 0000000000000000                  _                             ;position 0 of layer 2 (the 3rd layer)
                                   ;---------------------------------------
                                   ; divide each exponential by the sum
                                   ;---------------------------------------
                                                                         ;AR1 is already pointing to layer2
00000123 327FFA2000004010                  _   _4:AR3 = _4:#layer01      ;point to the first exponential result in layer 1
                                                                         ;AR2 is already pointing to layer3
00000124 307FCD000000000B                  _   _1:actMode = _1:#SoftMax  ;division e^xi/sum(e^xi)--actually just a
00000125 317FEF100000000F                  _   _2:REPEAT = _2:#15        ;division operation here
00000126 8B800AB800BB8001                  act  s8:*AR2++[1] = (s8:*AR3++[1], s8:*AR1++[0]) ;divide each exponential
00000127 0000000000000000                  _                             ;of layer 1 by the sum of all of the exponentials
                               ;---------------------------------------
                               ; HardMax process
                               ;---------------------------------------
00000128 327FF82000004030                  _   _4:AR1 = _4:#layer03      ;point to start of SoftMax result vector
00000129 327FFA2000013880                  _   _4:AR3 = _4:#_1.0         ;use weight of 1.0 for each node
                                                                         ;AR2 is already pointing to layer4
0000012A 307FCD000000000C                  _   _1:actMode = _1:#HardMax  ;use HardMax activation mode
0000012B 8F8002F8001F8003                  act s128:*AR2++[0] = (s128:*AR1++[0], s128:*AR3++[0])
0000012C 0000000000000000                  _
0000012D 327FFC2000004000                  _   _4:AR5 = _4:#layer00      ;spill the 5 layers of each pass into the
0000012E 317FEF1000000004                  _   _2:REPEAT = _2:#4         ;output buffer for retrieval by the host
0000012F 0784067808500000                  _   _128:*AR6++[128] = _128:*AR5++[16] ;when process is done
00000130 0000000000000000                  _
00000131 127FF417FED43FE3                  _  _4:PCS = (_2:LPCNT0, 16, loop)   ;conditional load of PC with relative
00000132 0000000000000000                  _                                   ; address if specified bit is set
00000133 397FF52000000108                  _  s2:PC = _4:#done              ;go back to done--unconditional load of PC
                                                                         ;with absolute address
```