Low-Power Neural Network Accelerators: Advancements in Custom Floating-Point Techniques

Yarib Nevarez

Universität Bremen

May 22, 2024

Spike-by-Spike Neural Network

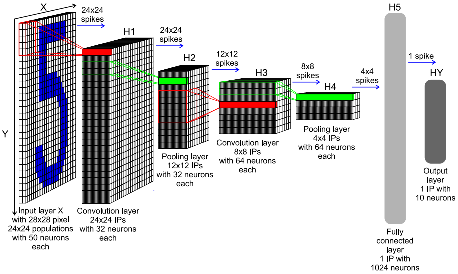


Figure: Spike-by-Spike (SbS) neural network architecture for handwritten digit classification task.

Spike-by-Spike Neural Network

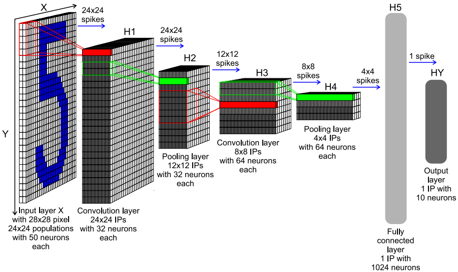


Figure: Spike-by-Spike (SbS) neural network architecture for handwritten digit classification task.

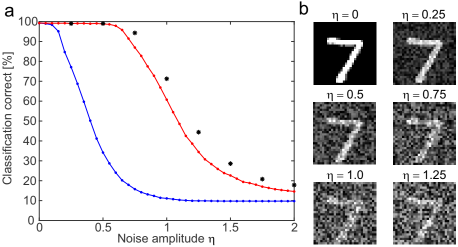


Figure: Performance classification of SbS NN versus equivalent CNN.

Spike-by-Spike Layer Update

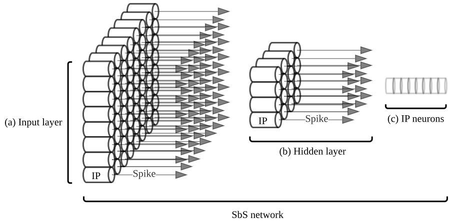


Figure: SbS inference population (IP) as independent computational entities.

$$h_\mu^{new}(i) = \frac{1}{1+\epsilon}\left(h_\mu(i) + \epsilon \frac{h_\mu(i)W(s_t|i)}{\sum_j h_\mu(j)W(s_t|j)}\right)$$

HW/SW Co-Design Framework

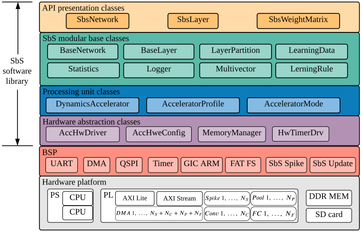


Figure: System-level overview of the embedded software architecture.

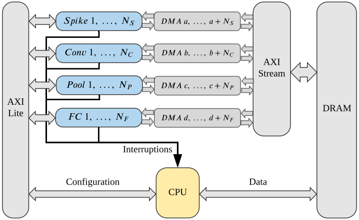


Figure: System-level hardware architecture with scalable number of heterogeneous processing units (PU).

Processing Unit

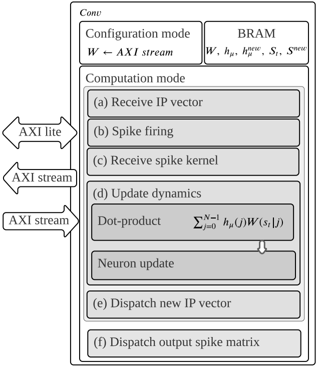


Figure: Conv processing unit.

Processing Unit

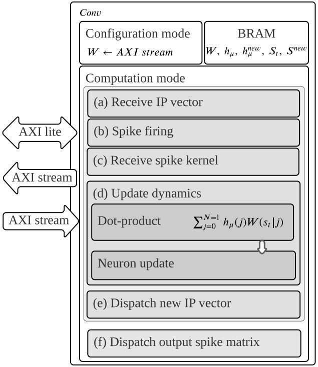


Figure: Conv processing unit.

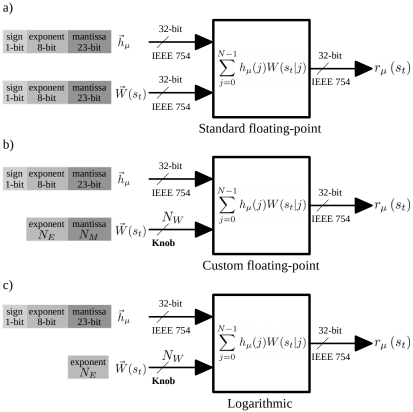


Figure: Dot-product hardware module.

Hybrid Dot-Product Approximation

$$r_\mu\left(s_t\right) = \sum_{j=0}^{N-1} h_\mu(j) W(s_t|j) \tag{1}$$

$$E_{\min} = \log_2(\min_{\forall i}(W(i))) \tag{2}$$

$$N_E = \lceil \log_2(|E_{\min}|) \rceil \tag{3}$$

$$N_W = N_E + N_M \tag{4}$$

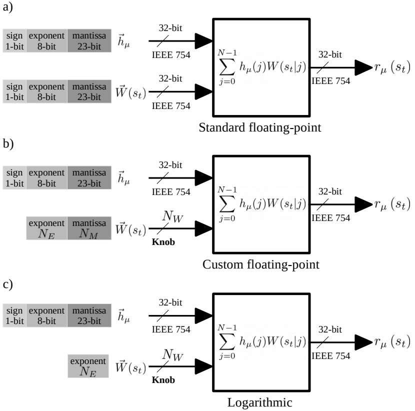


Figure: Dot-product hardware module.

Dot-Product with Standard Floating-Point (IEEE 754)

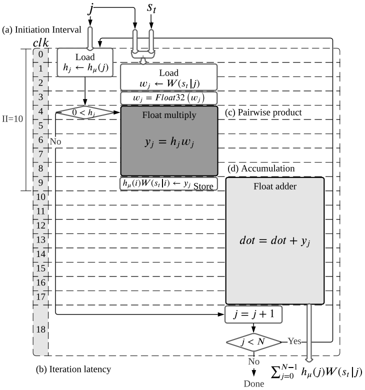


Figure: Dot-product hardware module with standard floating-point computation.

$$L_{f32} = 10N + 9$$

Dot-Product with Hybrid Custom Floating-Point Approximation

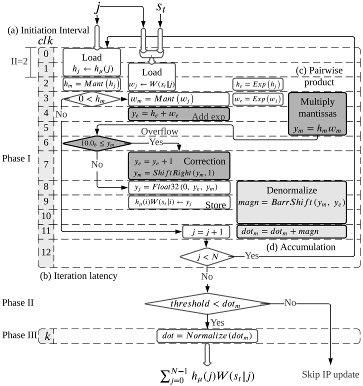


Figure: Dot-product hardware module with hybrid custom floating-point approximation.

$$L_{custom} = 2N + 11$$

Dot-product with Hybrid Logarithmic Approximation

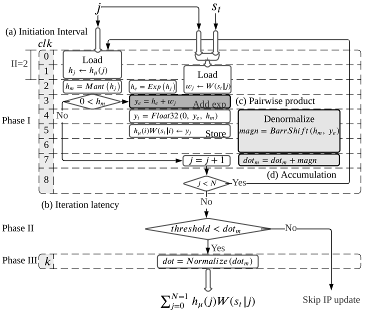


Figure: Dot-product hardware module with hybrid logarithmic approximation.

$$L_{custom} = 2N + 7$$

Acceleration with Standard Floating-Point

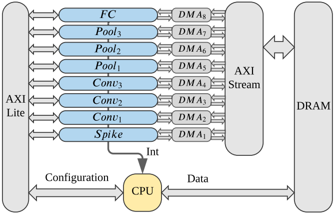


Figure: System overview of the top-level architecture with 8 processing units.

Acceleration with Standard Floating-Point

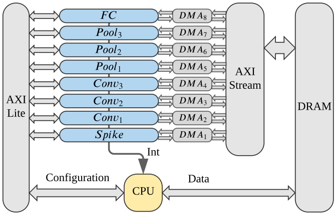


Figure: System overview of the top-level architecture with 8 processing units.

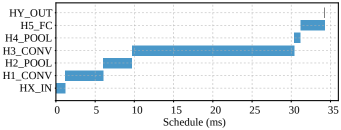


Figure: Computation on embedded CPU.

Acceleration with Standard Floating-Point

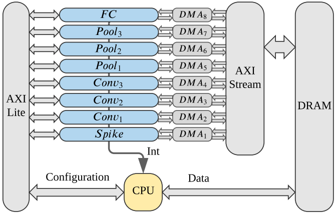


Figure: System overview of the top-level architecture with 8 processing units.

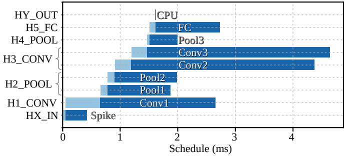


Figure: Performance of processing units with standard floating-point.

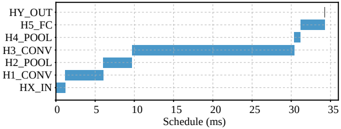


Figure: Computation on embedded CPU.

Acceleration with Standard Floating-Point

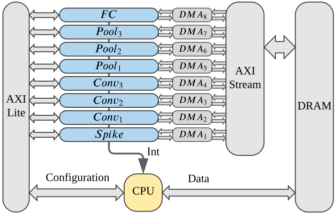


Figure: System overview of the top-level architecture with 8 processing units.

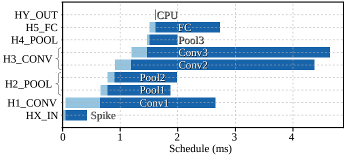


Figure: Performance of processing units with standard floating-point.

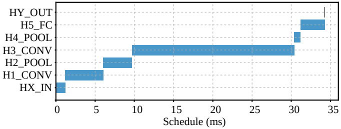


Figure: Computation on embedded CPU.

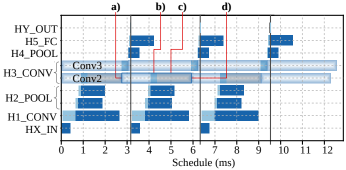


Figure: Performance bottleneck of cyclic computation on processing units with standard floating-point.

Acceleration with Custom Floating-Point

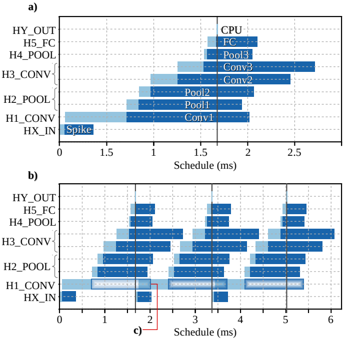


Figure: Performance on processing units with hybrid 8-bit floating-point.

Acceleration with Custom Floating-Point

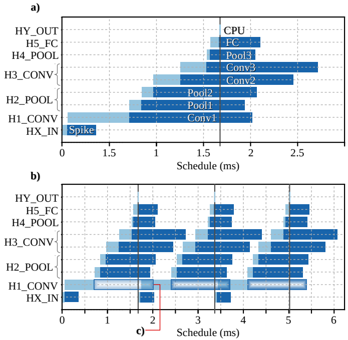


Figure: Performance on processing units with hybrid 8-bit floating-point.

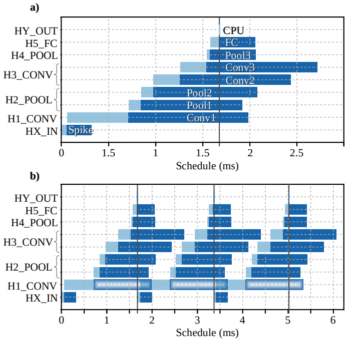


Figure: Performance of processing units with hybrid 4-bit logarithmic approximation.

Noise tolerance

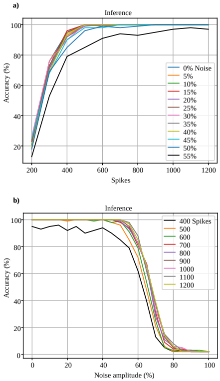


Figure: Noise tolerance with standard floating-point.

Noise tolerance

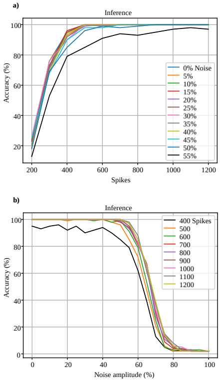


Figure: Noise tolerance with standard floating-point.

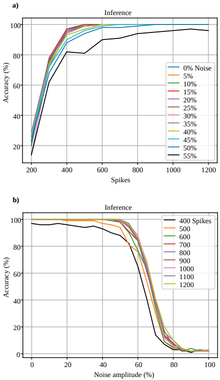


Figure: Noise tolerance with hybrid 8-bit floating-point approximation.

Noise tolerance

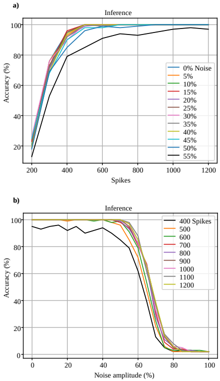


Figure: Noise tolerance with standard floating-point.

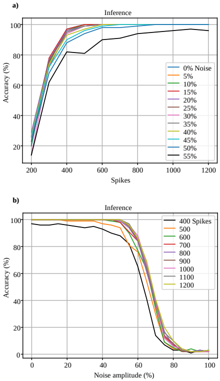


Figure: Noise tolerance with hybrid 8-bit floating-point approximation.

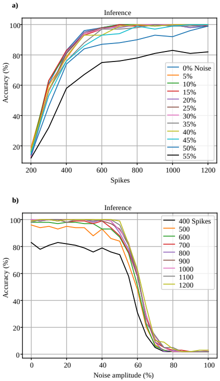


Figure: Noise tolerance with hybrid 4-bit logarithmic approximation.

Accelerator Implementations

Table: Accelerator implementations.

Platform implementation	Power (W)	Clk (MHz)	Latency (ms)	Acceleration	Accuracy (%)
Standard floating-point	2.420	200	3.18	10.7x	98.98
Hybrid floating-point 8-bit	2.369	200	1.67	20.5x	98.97
Hybrid Logarithmic 4-bit	2.324	200	1.67	20.5x	98.84

Convolution Operation

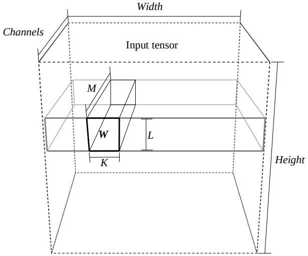


Figure: Two dimensional convolution operation.

$$Conv2D\left(W, b, h\right)_{i,j,o} = \sum_{k,l,m}^{K,L,M} h_{(i+k,j+l,m)} W_{(o,k,l,m)} + b_o$$

HW/SW Co-Design Framework

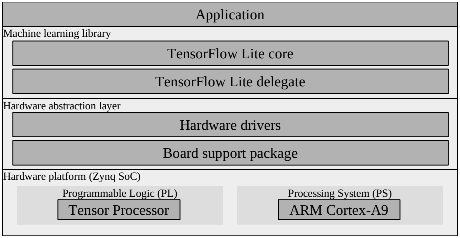


Figure: High level embedded software architecture.

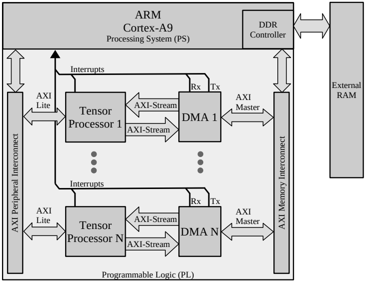


Figure: Base embedded system architecture.

Tensor Processor

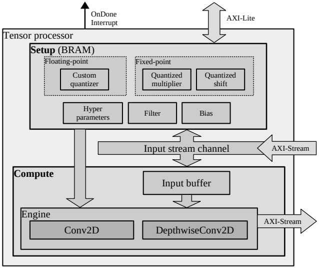


Figure: High level architecture.

Tensor Processor

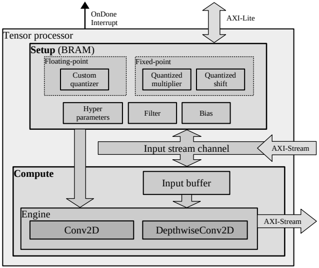


Figure: High level architecture.

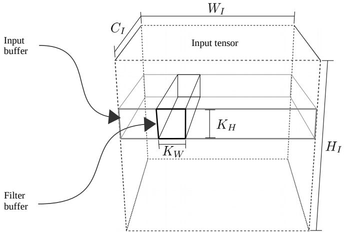


Figure: On-chip memory buffers.

On-Chip Memory

$$TP_M = TP_B + V_M \tag{5}$$

$$TP_B = Input_M + Filter_M + Bias_M \tag{6}$$

$$Input_M = K_H W_I C_I BitSize_I \tag{7}$$

$$Filter_M = C_I K_W K_H C_O BitSize_F \tag{8}$$

$$Bias_M = C_O BitSize_B \tag{9}$$

$$C_O = \frac{TP_M - V_M - K_H W_I C_I BitSize_I}{C_I K_W K_H BitSize_F + BitSize_B} \tag{10}$$

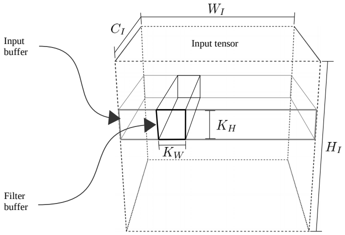


Figure: On-chip memory buffers.

Dot-Product with Hybrid Floating-Point 6-bit

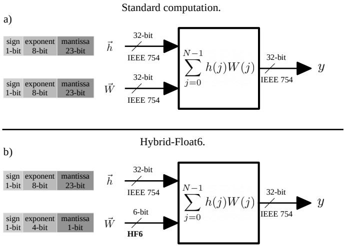


Figure: High level architecture.

Dot-Product with Hybrid Floating-Point 6-bit

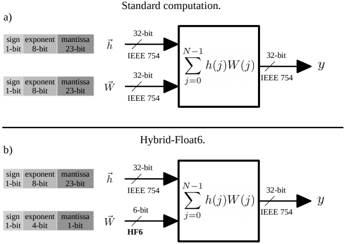


Figure: High level architecture.

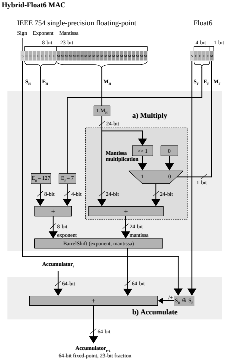


Figure: On-chip memory buffers.

$$L_{hf} = N + 7$$

Custom Floating-Point Quantization

Algorithm 1: Custom floating-point quantization.

Input: $MODEL$ as the CNN.
Input: E_{size} as the target exponent bit size.
Input: M_{size} as the target mantissa bits size.
Input: $STDM_{size}$ as the IEEE 754 mantissa bit size.
Output: $MODEL$ as the quantized CNN.

foreach $layer$ *in* $MODEL$ **do**
  if $layer$ *is* $Conv2D$ *or* $SeparableConv2D$ **then**
    $filter, bias \leftarrow GetWeights(layer)$
    foreach x *in* $filter$ *and* $bias$ **do**
      $sign \leftarrow GetSign(x)$
      $exp \leftarrow GetExponent(x)$
      $fullexp \leftarrow 2^{E_{size}-1} - 1$ `// Get full range value`
      $cman \leftarrow GetCustomMantissa(x, M_{size})$
      $leftman \leftarrow GetLeftoverMantissa(x, M_{size})$
      if $exp < -fullexp$ **then**
        $x \leftarrow 0$
      else
        if $exp > fullexp$ **then**
          $x \leftarrow (-1)^{sign} \cdot 2^{fullexp} \cdot (1 + (1 - 2^{-M_{size}}))$
        else
          if $2^{STDM_{size}-M_{size}-1} - 1 < leftman$ **then**
            $cman \leftarrow cman + 1$ `// Above halfway`
            if $2^{M_{size}} - 1 < cman$ **then**
              $cman \leftarrow 0$ `// Correct mantissa overflow`
              $exp \leftarrow exp + 1$
          $x \leftarrow (-1)^{sign} \cdot 2^{exp} \cdot (1 + cman \cdot 2^{-M_{size}})$
    $SetWeights(layer, filter, bias)$

Acceleration in Sensor Analytics (TinyML)

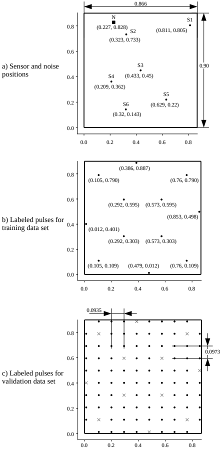


Figure: Structural health monitoring, all lengths are in metters (m).

Acceleration in Sensor Analytics (TinyML)

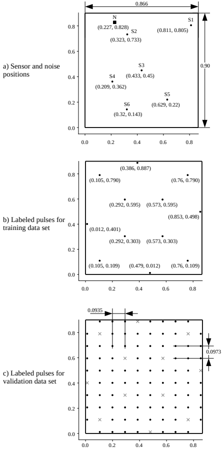


Figure: Structural health monitoring, all lengths are in metters (m).

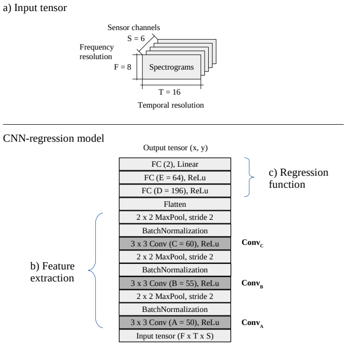


Figure: CNN-regression model for sensor analytics.

Training

Algorithm 2: Training with iterative early stop cycle.

Input: $MODEL$ as the input model
Input: D_{train} as the training data set
Input: D_{val} as the validation data set
Input: N_I as the stop patience for iterative training cycle
Input: N_E as the early stop patience (epochs) for training
Input: B_{size} as the mini-batch size
Output: $MODEL$ as the full-precision output model

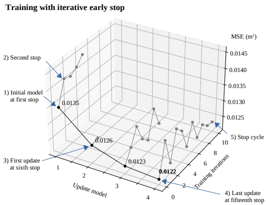
```
// Initial training and evaluation
Train(MODEL, D_train, D_val, N_E, B_size)
mse_i ← Evaluate(MODEL, D_val)
n_I ← 0
while n_I < N_I do
    // Iterative early stop cycle
    Train(MODEL, D_train, D_val, N_E, B_size)
    mse_v ← Evaluate(MODEL, D_val)
    if mse_v < mse_i then
        Update(MODEL)
        mse_i ← mse_v
    end
    else
        MODEL ← LoadPreviousWeights()
        n_I ← n_I + 1
    end
end
```

Training

Algorithm 2: Training with iterative early stop cycle.

Input: $MODEL$ as the input model
Input: D_{train} as the training data set
Input: D_{val} as the validation data set
Input: N_I as the stop patience for iterative training cycle
Input: N_E as the early stop patience (epochs) for training
Input: B_{size} as the mini-batch size
Output: $MODEL$ as the full-precision output model

`// Initial training and evaluation`
$Train(MODEL, D_{train}, D_{val}, N_E, B_{size})$
$mse_i \leftarrow Evaluate(MODEL, D_{val})$
$n_I \leftarrow 0$
while $n_I < N_I$ **do**
  `// Iterative early stop cycle`
  $Train(MODEL, D_{train}, D_{val}, N_E, B_{size})$
  $mse_v \leftarrow Evaluate(MODEL, D_{val})$
  if $mse_v < mse_i$ **then**
    $Update(MODEL)$
    $mse_i \leftarrow mse_v$
  end
  else
    $MODEL \leftarrow LoadPreviousWeights()$
    $n_I \leftarrow n_I + 1$
  end
end

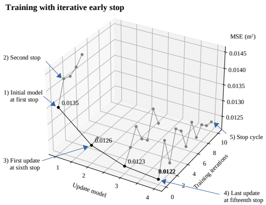


Training

Algorithm 2: Training with iterative early stop cycle.

Input: $MODEL$ as the input model
Input: D_{train} as the training data set
Input: D_{val} as the validation data set
Input: N_I as the stop patience for iterative training cycle
Input: N_E as the early stop patience (epochs) for training
Input: B_{size} as the mini-batch size
Output: $MODEL$ as the full-precision output model

```
// Initial training and evaluation
Train(MODEL, D_train, D_val, N_E, B_size)
mse_i ← Evaluate(MODEL, D_val)
n_I ← 0
while n_I < N_I do
    // Iterative early stop cycle
    Train(MODEL, D_train, D_val, N_E, B_size)
    mse_v ← Evaluate(MODEL, D_val)
    if mse_v < mse_i then
        Update(MODEL)
        mse_i ← mse_v
    end
    else
        MODEL ← LoadPreviousWeights()
        n_I ← n_I + 1
    end
end
```



Algorithm 2: OnMiniBatchUpdate_Callback.

Input: $MODEL$ as the full-precision input model
Input: E_{size} as the target exponent bits size
Input: M_{size} as the target mantissa bits size
Input: D_{train} as the training data set
Input: D_{val} as the validation data set
Input: N_{ep} as the number of epochs
Input: B_{size} as the mini-batch size
Output: $MODEL$ as the quantized output model

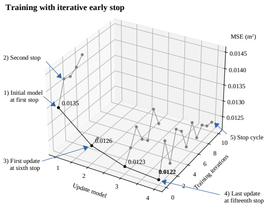
```
// Quantize
MODEL ← QuantizeTraining(MODEL, E_size, M_size)
if 1 < epoch then
    // Update model after first epoch
    mse_v ← Evaluate(MODEL, D_val)
    if mse_v < mse_i then
        Update(MODEL)
        mse_i ← mse_v
    end
end
```

Training

Algorithm 2: Training with iterative early stop cycle.

Input: $MODEL$ as the input model
Input: D_{train} as the training data set
Input: D_{val} as the validation data set
Input: N_I as the stop patience for iterative training cycle
Input: N_E as the early stop patience (epochs) for training
Input: B_{size} as the mini-batch size
Output: $MODEL$ as the full-precision output model

```
// Initial training and evaluation
Train(MODEL, D_train, D_val, N_E, B_size)
mse_i ← Evaluate(MODEL, D_val)
n_I ← 0
while n_I < N_I do
    // Iterative early stop cycle
    Train(MODEL, D_train, D_val, N_E, B_size)
    mse_v ← Evaluate(MODEL, D_val)
    if mse_v < mse_i then
        Update(MODEL)
        mse_i ← mse_v
    end
    else
        MODEL ← LoadPreviousWeights()
        n_I ← n_I + 1
    end
end
```



Algorithm 2: OnMiniBatchUpdate_Callback.

Input: $MODEL$ as the full-precision input model
Input: E_{size} as the target exponent bits size
Input: M_{size} as the target mantissa bits size
Input: D_{train} as the training data set
Input: D_{val} as the validation data set
Input: N_{ep} as the number of epochs
Input: B_{size} as the mini-batch size
Output: $MODEL$ as the quantized output model

```
// Quantize
MODEL ← QuantizeTraining(MODEL, E_size, M_size)
if 1 < epoch then
    // Update model after first epoch
    mse_v ← Evaluate(MODEL, D_val)
    if mse_v < mse_i then
        Update(MODEL)
        mse_i ← mse_v
    end
end
```