algorithms

Article

A Virtual Machine Platform Providing Machine Learning as a Programmable and Distributed Service for IoT and Edge On-Device Computing: Architecture, Transformation, and Evaluation of Integer Discretization

Stefan Bosse [1,2]

[1] Department Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany; sbosse@uni-bremen.de
[2] Department Mechanical Engineering, University of Siegen, 57072 Siegen, Germany

Abstract: Data-driven models used for predictive classification and regression tasks are commonly computed using floating-point arithmetic and powerful computers. We address constraints in distributed sensor networks like the IoT, edge, and material-integrated computing, providing only low-resource embedded computers with sensor data that are acquired and processed locally. Sensor networks are characterized by strong heterogeneous systems. This work introduces and evaluates a virtual machine architecture that provides ML as a service layer (MLaaS) on the node level and addresses very low-resource distributed embedded computers (with less than 20 kB of RAM). The VM provides a unified ML instruction set architecture that can be programmed to implement decision trees, ANN, and CNN model architectures using scaled integer arithmetic only. Models are trained primarily offline using floating-point arithmetic, finally converted by an iterative scaling and transformation process, demonstrated in this work by two tests based on simulated and synthetic data. This paper is an extended version of the FedCSIS 2023 conference paper providing new algorithms and ML applications, including ANN/CNN-based regression and classification tasks studying the effects of discretization on classification and regression accuracy.

Keywords: Tiny ML; distributed sensor networks; discretization; integer arithmetic

check for updates

Citation: Bosse, S. A Virtual Machine Platform Providing Machine Learning as a Programmable and Distributed Service for IoT and Edge On-Device Computing: Architecture, Transformation, and Evaluation of Integer Discretization. *Algorithms* **2024**, *17*, 356. https://doi.org/10.3390/a17080356

Academic Editors: Marek Bolanowski, Andrzej Białas, Janusz Furtak and Andrzej Paszkiewicz

Received: 17 June 2024
Revised: 2 August 2024
Accepted: 5 August 2024
Published: 15 August 2024

1. Introduction

To address ubiquitous computing, edge computing, and distributed sensor networks, as well as a significant increase in device density and sensor deployment towards smart and self-aware sensors, sophisticated and dependable data processing architectures are required. The field of tiny machine learning (ML) is an emerging field posing challenges that are only partially addressed [1]. Floating-point arithmetic, with a high dynamic range and sufficient precision, is frequently used to compute machine learning models. Only integer arithmetic (8–32 bit) is supported by very low-resource tiny embedded systems, e.g., ARM Cortex M0-based systems; hence, training with integer arithmetic must be completed directly on the target device [2] or by model modification and freezing [3]. The computation of complex deep learning (DL) models is further limited by memory and computing power constraints of ultra-low-power devices [4]. To overcome software limitations and limited computability, hardware designs are becoming more popular [5]. We focus on the software processing of ML models on low-resource and low-power devices by model transformation fitting of low-resource devices.

The present work addresses virtualization on the programming level in IoT and sensor networks using very low-resource computers, typically with less than 64 kB of available RAM, with a particular focus on machine learning (ML) provided as a virtualized service. Compared with [6], we provide new algorithms and ML applications, including ANN/CNN-based regression and classification tasks, with a rigorous evaluation of discretization errors. The set of discretized non-linear transfer functions is extended, optimized, and evaluated rigorously. A new unified data set derived from physical simulation is used to demonstrate the capability and accuracy of the ML VM service, which can be deployed on any very low-resource micro-controller providing integer arithmetic only, as well as on desktop computers. We evaluate the VM MLISA and the model transformation process with respect to constraints and its application in Structural Health Monitoring.

A functional prediction or regression model is composed of linear and non-linear functions. The most critical part in the transformation and scaling process of ML models towards integer arithmetic is the non-linear function, e.g., the *tanh* transfer function. A chained composition can introduce high non-linearity, which must be handled carefully to avoid exploding model errors. For this purpose, we will train ANN models with data from highly non-linear analytical functions for the implementation of surrogate models. There is an extended evaluation of computational complexity and requirements for most relevant applications. The advantage of using a textual programming language instead of binary code is demonstrated by extended examples.

This work focuses on a universal VM suitable for implementation on very low-resource embedded systems and providing ML as a programmable service. Although “as a service” is a common cloud-based paradigm, we use this term in terms of virtualization on node and single computer level.

There is ongoing work to implement the computation of ML models with 8 or less bit integer arithmetic (and storage data size) on micro-controllers [7,8], commonly called Tiny ML [9] or ML on commodity devices [1]. We will relax this hard constraint by assuming a 32-bit microprocessor, e.g., the widely used Arm Cortex M series with chip die areas below 0.1 mm^2 and power consumption about 10 mW (active mode). Finally, we implement ML with 16 bit data-size storage (input, intermediate, and output data as well as model parameters). Overflow issues are relaxed by using 32-bit arithmetic internally. In [9], the authors outlined the benefits of Tiny ML in the context of sensor networks. Tiny ML enables the local processing of sensor data without extensive periodic communication to external serves, especially in the context of real-time capable structural health monitoring (SHM) systems. In [9], the computation of complex and deep convolutional neural networks (CNNs) was implemented on the sensor node level, often equipped with digital signal processors (DSPs) with optimized vector operations and sometimes floating-point arithmetic units (FPUs). The authors chose an Arm Cortex M4-based micro-controller, which provides dedicated DSP and FPU operations and 320 kB of RAM. In [3], the authors presented a software framework to run lightweight neural networks on micro-controllers based on both the ARM Cortex-M series and the RISC-V-based parallel ultra-low-power (PULP) platform, especially addressing energy-efficient computing, which is a high constraint on self-powered autonomous sensor nodes. The efficient implementation of ML models using a dedicated model and training library Fast ANN on the Arm Cortex M architecture is also demonstrated in [10]. They claim that the model computation is still possible on FPU-less micro-controllers but do not give evaluations for real use cases. In addition, all these frameworks create static model code, which cannot easily be updated at run-time (no service).

Instead of performing software model transformations to map a model using arithmetic A defined by an accuracy, value, and dynamic range, on a model using arithmetic B with lower accuracy and reduced value and dynamic ranges, the model can be mapped on a dedicated hardware architecture, providing the elementary core operations of ML models, as described by [1]. Although this is the most efficient method, this approach prevents the use of widely used and cheap electronic components and universal and flexible model computations, as well as update services (of the model and the ML services). This compromises the deployment of the proposed ML/VM architecture in embedded systems with limited or no reprogramming capabilities (like material-integrated systems, discussed in Section 2).

This paper is organized as follows. A short introduction to material-integrated sensor nodes for structural health monitoring (SHM) is given to outline the motivation as well as the communication architecture, defining constraints for the VM and its ML service introduced and used in this work. An extended section describes the REXA VM with a focus on ML. The transformation process of continuous ML models to discretized integer-scaled arithmetic models is described in detail, and approximation methodologies and discretization errors are discussed for non-linear transfer functions. Two use cases demonstrate the capability of the transformation process, the REXA VM ML operations, and typical accuracy losses that can be expected in real applications for regression and classification tasks.

We introduce two different model scaling algorithms (static and dynamic) and evaluate the effect of discretization on the model accuracy with the two use-case examples. One use case uses synthetic sensor data created by wave propagation simulation, and the second uses input data from a mathematical model. The first use case combines classification and regression tasks into one model, and the second is a pure regression model. The main focus is on the non-linear activation functions used in neuronal network models and their discretization errors. The two use cases will demonstrate the quality of the proposed scaling approach and the simplicity of the VM programming for classification and regression models, including CNN architectures.

2. Sensor Node Architecture for SHM and Communication Architecture

The virtualization and deployment of VMs onto tiny micro-controllers was inspired by the wireless material-integrated sensor node [11,12], which is embedded between two layers of a fiber–metal laminate plate and developed in the DFG research group 3022 for automated diagnostics of hidden damages in fiber–metal laminates using guided ultrasonic waves (GUW). The sensor node is supplied with power via RFID/NFC communication technology only [13]. The energy harvester is able to deliver up to 15 mW of continuous power, significantly constraining the selection and operation of the electronic parts, as shown in Figure 1. After the sensor node is integrated into the plate, no software updates or maintenance can be applied. The communication with the micro-controller takes place only via the NFC tag. The communication is bidirectional, originally via the NFC tag's EEPROM (code and data). Alternatively, wireless communication can be realized directly with a write-through mode (because the lifetime of the EEPROM is limited to about 1000–10,000 write cycles), writing message data directly to the micro-controller. The sensor node uses an ARM Cortex STM32 L031 device, an NFC tag with a power supply, and a pre-amplifier for piezoelectric sensors. The features are summarized in Table 1. More details and descriptions of the sensor node can be found in [12].

Table 1. Features of the material-integrated wireless sensor node.

Feature	Value
Size	17 × 17 mm (without antenna and sensor)
Sensors	Piezoelectric transducer, MEMS sensor, temperature, radio field strength
Components	ADC (1 MSPS, 8–12 bit resolution), pre-amplifier, power regulator, NFC tag, ARM Cortex M0 STM32L031 micro-controller with 8 kB RAM and 32 kB ROM
Communication	Wireless, NFC (13.56 MHz), up to 100 kb/s
Energy	Energy harvesting by NFC tag, up to 15 mW continuous power
Software	REXA VM (CS = 1024, DS = 512, RS/FS = 32, only single-precision data)

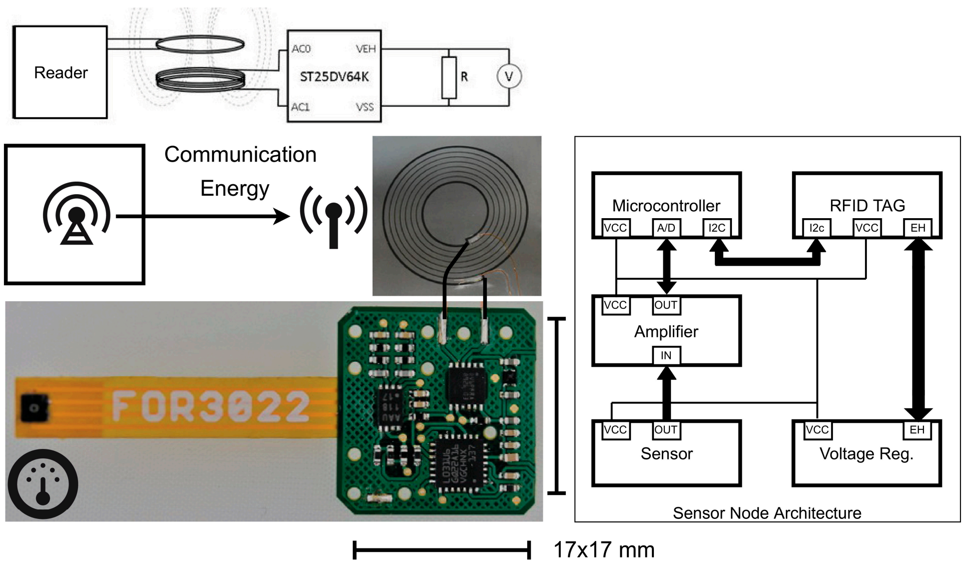


Figure 1. ARM Cortex M0-based sensor node (STM32L031) implementing the REXA VM for material-integrated GUW sensing with NFC for energy transfer and bidirectional communication with only 8 kB of RAM and 32 kB of ROM.

Although this work focuses on the design and implementation of a universal VM suitable for implementation on very low-resource embedded systems and providing ML as a programmable service, communication aspects are relevant and must be considered as constraints, both for the design and operation of the VM and the communication design. Communication with material-integrated sensor nodes is commonly performed by using wireless technologies, but in the presence of metals and dielectric materials, wireless communication is a challenge. Low- and mid-frequency RFID technologies are widely used. In this work, it is assumed that the REXA VM is accessible by RFID/NFC communication, as shown in Figure 2. Sensor nodes communicate wirelessly via RFID/NFC tag circuits with a reader, which is connected short-range to a "remote" VM instance. The reader nodes are connected long-range (wired or wireless) to establish a distributed network. The reader nodes are communication end-points and message routers.

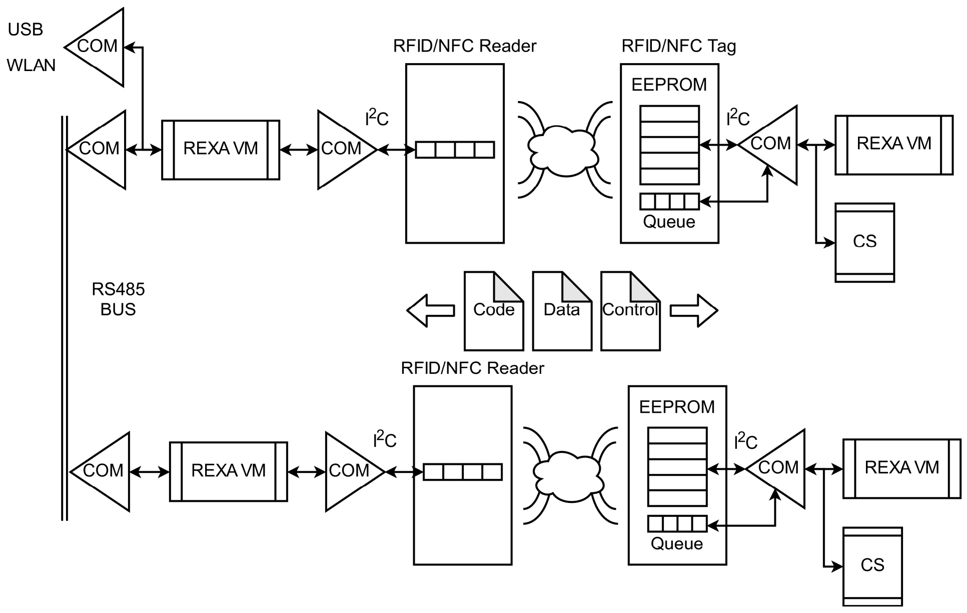


Figure 2. Principle REXA VM network architecture using different wired and wireless communication technologies.

The communication capabilities have an impact on the ML models that can be implemented with respect to the code and data size. It can be expected that the typical message size will not exceed 1k Bytes. The transfer of 1k Byte of payload data requires about one second of transfer time, including packet fragmentation. ML models, including the forward prediction functions, are submitted to the VM via text, including the fixed model parameters. In the use-case section, the text and code + data sizes are measured, showing that small and moderate complex models, including CNNs, can be transferred by such a resource-constraint communication channel.

3. REXA VM Architecture and Programming Language

The REXA VM is the core component for implementing and virtualizing ML on low-resource computers using a programmable approach. The next sub-sections describe the VM architecture briefly to aid in understanding the implementation details of the following use cases.

3.1. Architecture

The real-time capable and extensible architecture (REXA) VM is a full-featured script engine based on a stack processor architecture. A detailed description can be found in [11,14]. Although any programming language can be used, we implemented a modified subset of the stack-based Forth programming language [15]. Any ISA can be implemented by addressing stack-based computation, but Forth is a well-known and long-standing programming language that is firstly a high- and low-level language, secondly can be implemented efficiently on low-resource systems, thirdly is extensible (e.g., by the MLISA introduced in this work), and fourthly requires only simple compilers. In contrast, C/C++ is a compiled language not intended for script execution as intended in this work, and C/C++ compilers are complex, whereas Forth compilers are not.

The full version embeds a text-to-byte-code compiler that translates Forth programs into byte-code. A dialect of the Forth programming language provides high-level constructs like loops and functions (words in the Forth terminology), as well as the compactness of the VM implementation, including a hand-written language parser and an incremental direct compiler producing VM code. One main feature is the binding of data and code in frames without the necessity of a free-used memory block list-driven dynamic memory management. All dynamic run-time data are stored on multiple data stacks. Such binary byte-code frames with embedded data can be exchanged among different processors and sensor nodes. The compilation, as well as the code execution, can be performed under soft real-time constraints. The run-time can be estimated in advance, e.g., offline, by a VM twin with check-pointing, tracing, and monitoring capabilities. The REXA VM is written in plain C and is highly portable. Alternatively, the REXA VM can be implemented in other programming languages like JavaScript to support embedding in UI applications.

The REXA VM was designed especially for deployment on low-resource microcontrollers with less than 64 kB RAM and low clock frequencies below 50 MHz. It utilizes a freely programmable ISA, but the ISA of the VM used in this work is closely related to the Forth programming language [15]. The VM is a pure stack processor, i.e., most operations process data via multiple stack memories with a zero-operand instruction format. There is support for arrays and access to external buffers via the DIOS (see below). The VM instruction loop processes byte-code programs stored in a code segment (CS).

Figure 3 shows the architecture design of the REXA VM and its interoperability with the closely coupled just-in-time (JIT) compiler. The JIT compiler depends on the VM ISA, which can be freely defined, although this work is strongly related to the Forth programing language. The architecture details depend on the configuration (single- or multi-tasking, number of stacks, and customized extensions and accelerators). The principle architecture is equal for software and hardware implementations. Profiling is an optional feature used for predictive real-time scheduling, as well as the energy-aware real-time scheduler.

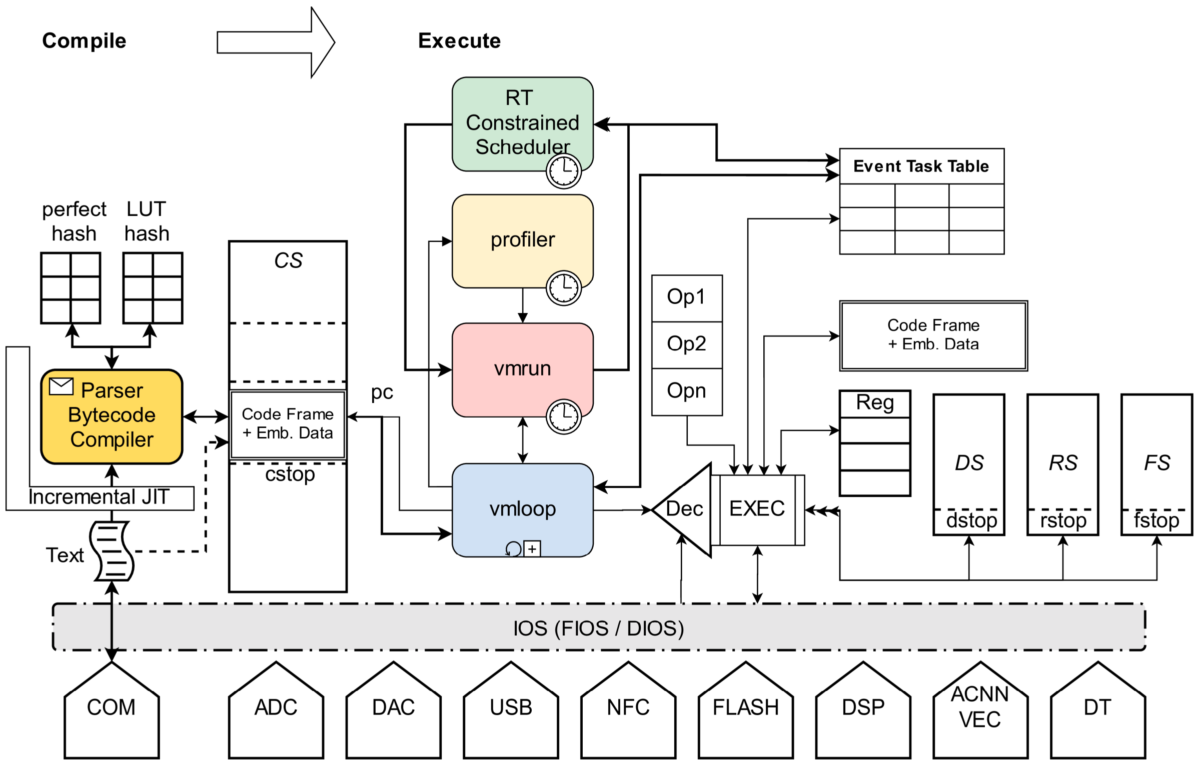


Figure 3. Basic REXA-VM architecture with integrated JIT compiler, stacks, and byte-code processor [11,12].

The code segment (CS) is the central storage for source code, byte-code, and embedded data. The CS is partitioned into dynamically sized code frames, commonly assigned to a task (depending on the scheduling model), as shown in Figure 4. Assuming a 16-bit VM, the CS is limited to 32k Bytes in size. The scheduler controls and monitors the byte-code loop (*vmloop*). Code operations can suspend task execution by waiting for events handled by an event table. The input–output system (IOS, similar to the widely used foreign function interface (FFI), extends the code and data space of the VM) is the central bridge between the core VM and the host application. The VM architecture is optimized for resource sharing, e.g., using an ADC sample buffer for computations from the VM programming level.

Temporary (short lifetime) data are stored and manipulated directly on fixed-size stack memories:

1. The data stack (DS) holds most of the processing data and instruction operands;
2. The return stack (RS) used for function calls (not accessible from the programming level for security reasons);
3. Optional loop stack (FS) used for loop counters and secondary user data (can be merged with RS for memory efficiency).

All non-temporary data are either embedded in the code frames or provided by the host application via the data input–output system layer (DIOS) API or by providing Io functions using the function input–output system layer (FIOS) API. All MLISA operations are attached to the VM using the FIOS; ADC buffers are attached by the DIOS.

The data width of the stack cell is always 16 bit (single word width). The REXA VM also supports double-word operations (as a configurable option). Double words are composed of two single data words (word order depends on the native byte order of the underlying processor). The VM can read and write double words directly from and to stacks (single memory access). The access time of multiplexed single and double word access to the stacks by memory pointer casting is commonly identical (assuming 32-bit microprocessors). The push and pop operations involved in most of the VM instruction code words modify stack pointers (*dstop*, *rstop*, *fstop*). For security reasons, the return stack (which holds code pointers on function calls) should not be accessed directly by program code.

Besides hardcore stacks implemented inside the VM, soft-core stacks can be implemented on the programming level in data arrays (embedded in code frames). Push and pop operations are provided by the core instruction word set:

```
array mystack 100
1 mystack push
mystack pop . cr
3 mystack get ( Gets copy of n-th value from top )
```

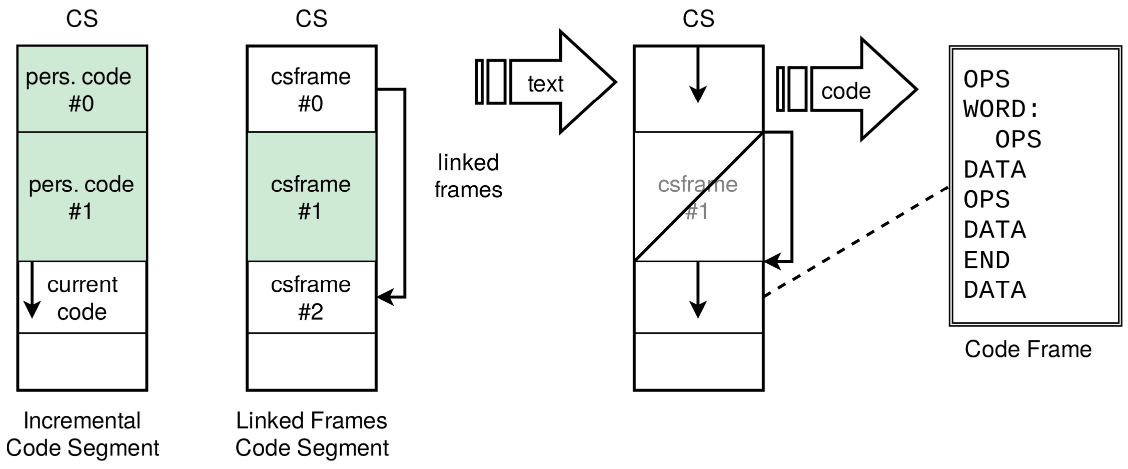


Figure 4. (**Left**) Incremental growing code segment (single-tasking), persistent code cannot be removed. (**Right**) Dynamically partitioned code segments using code frames and linking code frames due to fragmentation.

The compiler translates the source code text into byte-code instructions. It is a just-in-time (JIT) compiler that can compile code incrementally and on demand. Since the ISA of stack processors consists mostly of zero-operand instructions, it supports fine-grained compilation at the token level, including ML models. The source text can be directly stored in the code segment referenced by a code frame (or any other data buffer, alternatively). Most instruction words can be directly mapped to a consecutively numbered operation code. Therefore, the compiler translates the source code into byte-code in place, i.e., by replacing the text with binary byte-code, saving additional target memory buffers. An instruction word consists of at least one character and thus can always be replaced by the op-code (one byte). Although a literal value can consist of only one digit and the data of a single word value occupies two bytes, there is always a space or newline character after a literal value, providing the required data space. Extension of the current code frame at the end is always possible (as long as there is free space in the CS). One exception is a double-word literal value requiring at least two characters and the suffix "l", followed by an obligatory separator character and the space, providing four bytes of data space in total.

Data are either stored on the stacks during run-time or embedded in the code frame during translation. Scalar variables and initialized arrays can always be embedded in place. Non-initialized arrays are appended to the end of the compiled code frame (placing id delayed until the code frame is compiled).

3.2. Programming Language

The programming language consists of the Forth core set, which is mainly zero-operand words. A word is either a numerical or string value storing this value on the data stack or an instruction word like arithmetic or control flow operations. Zero-operand instructions get their operands from the stacks and store results on the stack again. User functions (words) can be defined by using the operator, as shown in Example 1. Because user words, as well as core words, get their operands via the stack and store results on the stack, a comment in the form LHS -- RHS is commonly used to specify the input and output function interface. The left-hand side specifies the input arguments (right is the top of the stack), and the right-hand side specifies the output (if any).

A typical REXA VM program for ML consists of a head section defining initialized and non-initialized arrays, and words computing data, as shown in Example 1 and discussed in detail in the MLISA Section 4.

```
array X 3
array P { 1 2 3 }
array Y 3
( n -- sum )
: productsum
  0 ( sum )
  swap
  ( n ) 0 do
    X i cell+ ! ( X[i] )
    P i cellü ! ( P[i] )
    * ( X[i]*P[i] )
    + ( +sum )
  loop
  ( sum )
;
3 productsum
Y 1 cell+ @ ( Y[1]=sum  )
```

Example 1. *REXA Forth program sketch.*

4. ML Instruction Set Architecture

The extensible REXA VM is a stack-based process that provides ML as a programmable service via its input–output system bridge. To enable and support efficient processing of ML models, a set of basic ML operations are added to the ISA of the VM. This ISA can be extended at any time. The ML instruction set architecture extension MLISA provides universal ML micro-service operations (ML and MLISA as a service, MLaaS). A code frame describes the ML model structure and defines an inference function that evaluates a specific model by applying the following ML core and vector operations to the input data, e.g., a measured time-resolved sensor signal. There are primarily three classes of ML models supported by the MLISA:

1. Decision trees (DT);
2. Fully connected artificial neural networks (FC-ANNs);
3. Convolutional neural networks (CNNs).

An ML task consists of the prior training phase using example data and the post-application inference phase using new unknown data. Actually, we only support the online and on-site inference of already offline-trained models. The following MLISA provides only operations for the applications of mathematical models based on discrete integer arithmetic. The original models were trained with standard numerical methods using floating point arithmetic transformed to integer-scaled models. Training using classical error back-propagation methods is currently not supported due to the requirement of storing a suitable training and test data set on the device, which is not available on very low-resource microcontrollers.

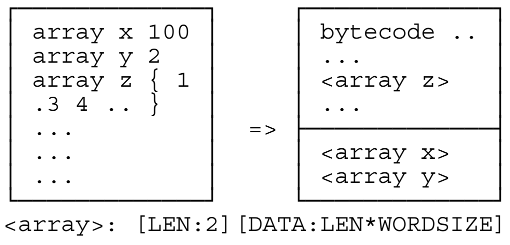
4.1. ML Core Operations

ANN and CNN computations require efficient and generic vector operations crucial to implementing ML on microcontrollers, at least for model inference. The REXA VM provides a unified core set of vector operations that can be used for the iterative computation of ANN and CNN models. It is assumed that the integer data width of the models is *N*-bit and that there is 2*N*-bit arithmetic. In our case, we have 16-bit model data and 32-bit native integer arithmetic. The set of basic operations needed to implement ANN and CNN models and perform forward activation computations consists of the following:

1. Element-wise vector operations, i.e., addition and multiplication, *vecmul*: *op1vec op2vec dstvec scalevec*;
2. Dot-product operations performing a sum of product data fusion (*vecprod*: *veca vecb scale* $\rightarrow$ number);
3. A folding operation for node layer computations (*vecfold*: *invec wgtvec outvec scalevec*);
4. A convolution operation for CNN computations (*vecconv*: *invec wgtvec outvec scale inwidth kwidth stride pad*);
5. A pooling operation for CNN computations reusing the *vecconv* operation. The second argument combines the kernel height and a pooling function index (max, min, and so on) if the kernel width argument is negative (*vecconv*: *invec kheight + poolfun outvec scale inwidth -kwidth stride pad*);
6. A mapping operation applying a function elementwise (*vecmap*: *srcvec dstvec func scalvec*);
7. A reduction operation applying a function to all elements returning an aggregate value (*vecred*: *vec vecoff veclen op*) with the supported functions *min*, *max*, *sum*, and *average*;
8. A vector reshape operation shrinking or expanding a vector (*vecshape*: *srcvec dstvec scale*);
9. A generic scaling operation (*vecscale*: *srcvec dstvec scalevec*).

Vector operations commonly operate on arrays embedded in code frames, as shown in Definition 1. Scaling is typically applied after an aggregation operation, e.g., after computing a vector dot product sum of products (using 2N arithmetic), to avoid overflow. Some operations use one scaling factor for all elements, as discussed in the following section.

Definition 1. *Initialized arrays embedded in place in code frames and non-initialized arrays stored at the end of the compiled code frame.*



```
array x 100              bytecode ..
array y 2                ...
array z { 1              <array z>
.3 4 .. }                ...
...              =>
...                      <array x>
...                      <array y>

<array>: [LEN:2][DATA:LEN*WORDSIZE]
```

4.2. Vector Operations

The dynamic ranges of different integer (fixed point) and floating-point coding are shown in Table 2.

Table 2. Dynamic ranges of different integer (fixed point) and floating-point codings.

Coding	Dynamic Range
Int8	48 dB
Int16	96 dB
Int32	192 dB
Int64	385 dB
float16	180 dB
float32	1529 dB
float64	12,318 dB

The core set of vector operations provided by the REXA VM supporting (16-bit) integer arithmetic ANN and CNN computations are summarized in Tables 3 and 4. These operations are the primary part of the MLISA.

Table 3. Part 1 of the basic vector ANN functions operating on embedded or external array data (e.g., the sample buffer).

Vector Operation
`array <ident> <#cells>` Allocates a data array at the end of the code segment
`array <ident> { v1 v2 .. }` Allocates an initialized data array inside the code segment.

`vecload`
`( srcvec srcoff dstvec --`
Loads a data array into another array buffer. The source can be any external data provided by the IOS or internal embedded data.

`vecscale`
`( srcvec dstvec scalevec -- )`
Scales the source data array with scaling factors from the scale array and stores the result in the destination array. Negative scaling values reduce, and positive values expand the source data values. Can be used for scalar multiplication and division, too.

`vecadd,vecmul`
`( op1vec op2vec dstvec scalevec -- )`
Adds or multiplies two vectors element-wise with an optional result scaling (value 0 disables scaling). Both input and destination vectors must have the same size. Constant down-scaling of all elements is provided by a negative scaling value (instead of vector reference).

`vecsumn`
`( op1vec op2vec .. opnvec dstvec scale n -- )`
Special operation required for sliced convolution: sums over multiple vectors element-wise with an optional result scaling (value 0 disables scaling). Both input and destination vectors must have the same size. Constant down-scaling of all elements is provided by a negative scaling value. Internal double-word arithmetics prevent overflow.

Vector operations always operate on single data words (16 bit), but internally, 32-bit arithmetic is used to avoid over- and underflows. To scale to a signed 16 bit integer, some of the operations use a scale factor or scale factor vector (negative scale values reduce, positive expand the values by the scale factor) to avoid overflows or underflows in following computations, similar to scaled tensors in [7,8]. Vector operations can access arrays stored in code frames or provided externally by the host application (e.g., a signal buffer).

The computation of these operations is defined by the following formulas:

$$\begin{aligned}
&vecmul\left(\vec{a},\vec{b}\right) = (a_1b_1, a_2b_2, .., a_nb_n)^T\\
&dotprod\left(\vec{a},\vec{b}\right) = \vec{a}\cdot\vec{b} = \sum_{i=1}^{n} a_ib_i\\
&fold\left(\vec{a},\hat{c}\right) = \left(\sum_{i=1}^{n} a_ic_{i,1}, \sum_{i=1}^{n} a_ic_{i,2}, .., \sum_{i=1}^{n} a_ic_{i,n}\right)^T\\
&conv\left(\vec{a},\vec{c}\right) = \left(\vec{a}[1:c^n]\cdot\vec{c}, \vec{a}[s:s+c^n]\cdot\vec{c}, \vec{a}[2s:2s+c^n]\cdot\vec{c}, ..\right)\\
&map\left(\vec{a},f\right) = (f(a_1), f(a_2), .., f(a_n))^T\\
&n = \left|\vec{a}\right| = \left|\vec{b}\right|
\end{aligned} \tag{1}$$

with c^n as the kernel size (width multiplied by height) and s as the striding value (default is one).

The *vecconv* operation can be used for convolutional and pooling layers (pooling is used if *wgtwidth* is negative and the *wgtvec* value contains the weight matrix height combined with the pooling function selector). An activation function must be applied separately using the *vecmap* operation, e.g., by applying a *sigmoid* function to all elements of a vector.

The *vecsumn* function is required for sliced convolution. In this case (see architectural description), a partial convolution is performed for one input array and stored in an accumulator array. After all partial convolutions are calculated, the sum of all vectors must be calculated (with final scaling). This can be done in principle with the *vecadd* operation, but due to the accumulator (single word size), overflows can occur before final scaling.

Table 4. Part 2 of the basic vector ANN functions operating on embedded or external array data (e.g., the sample buffer).

Vector Operation

`vecfold`
`( invec wgtvec outvec scalevec -- )`
Performs a folding operation *ivec* × *wgtvec* with a given filter. The weights vector *wgtvec* must have the size |*invec*|*|*outvec*|.

`vecconv`
`( invec wgtvec outvec scale inwidth wgtwidth stride pad -- )`
Performs a two-dimensional kernel-based convolution operation *ivec* ⊗ *wgtvec*. The width of the input and kernel matrix (still a linear array) must be provided, and the width of the output and the heights are computed automatically from the vector lengths. If *wgtwidth* is negative, a pooling operation is performed. The *wgtvec* argument provides then the height of the filter and the operation to be performed.

`vecconv`
`( invec wgtvec outvec scale inwidth wgtwidth stride pad -- )`
Performs a two-dimensional pooling if *wgtwidth* is negative. The *wgtvec* argument provides the height of the filter and the operation to be performed (function index).

`vecmap`
`( srcvec dstvec func scalevec -- )`
Maps all elements from the source array onto the destination array using an external (IOS) or internal (user-defined word) function, e.g., the sigmoid function.

`vecred`
`( vec vecoff veclen op -- index value / valueL valueM )`
Reduces a vector to a scalar value. Supported operations are *min* (1) and *max* (2), returning position and value, *mean* (4) and *average* (8), returning a double-word value.

4.3. Activation Functions

Besides the vector operations that can be simply implemented in closed form with integer arithmetic, transfer functions with non-linear behavior are the most critical part of the integer computation of ML models. There are different transfers (activation) that are used in ANN and CNN models; the most prominent examples are:

- Linear function (*linear*) without x- and y-limits;
- Logistic or sigmoid function (*sigmoid*) with y-limit = [−1, 1];
- Hyperbolic tangent function (*tanh*) with y-limit = [−1, 1];
- Rectifying linear unit (*relu*) with one-side open y-limit = [0, ∞).

$$\begin{aligned} sigmoid(x) &= \tfrac{1}{1+e^{-x}} \\ tanh(x) &= \tfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \end{aligned} \tag{2}$$

The *linear* and *relu* functions can be directly implemented with integer arithmetic without loss of accuracy (except due to the integer discretization). The highly non-linear *sigmoid* and *tanh* functions require an approximation by using a hybrid approach combining a (compacted) look-up table (LUT) and an interpolation function. The *tanh* function can be neglected since it can be replaced, in most cases, by the *sigmoid* function without loss of generalization (of course, prior to training).

Trigonometric functions and functions composed of trigonometric functions are implemented with piecewise linear and non-linear look-up tables. The approximated discretized *sigmoid* function algorithm is shown in Algorithm 1. For example, the error of the discretized sigmoid function is always less than 1% or below 10 digital values while only requiring 30 bytes of LUT space and less than 10 unit operations (in addition to the LUT size of the *fplog10* function). These software functions can be immediately implemented in hardware, too. The LUTs are computed with Algorithm 2. The approximation of the *tanh* function is much more complex and computationally intensive as it involves the computation of two exponential terms, posing exploding behavior for larger negative and positive x values, as shown in Figure 5. The exploding functional behavior is relaxed for the sigmoid function by computing the *sigmoid* function only for positive x-values and using a logarithmic base function, finally mirroring and flipping the result for negative x-values, which does not prevent exploding gradients in the case of the *tanh* function.

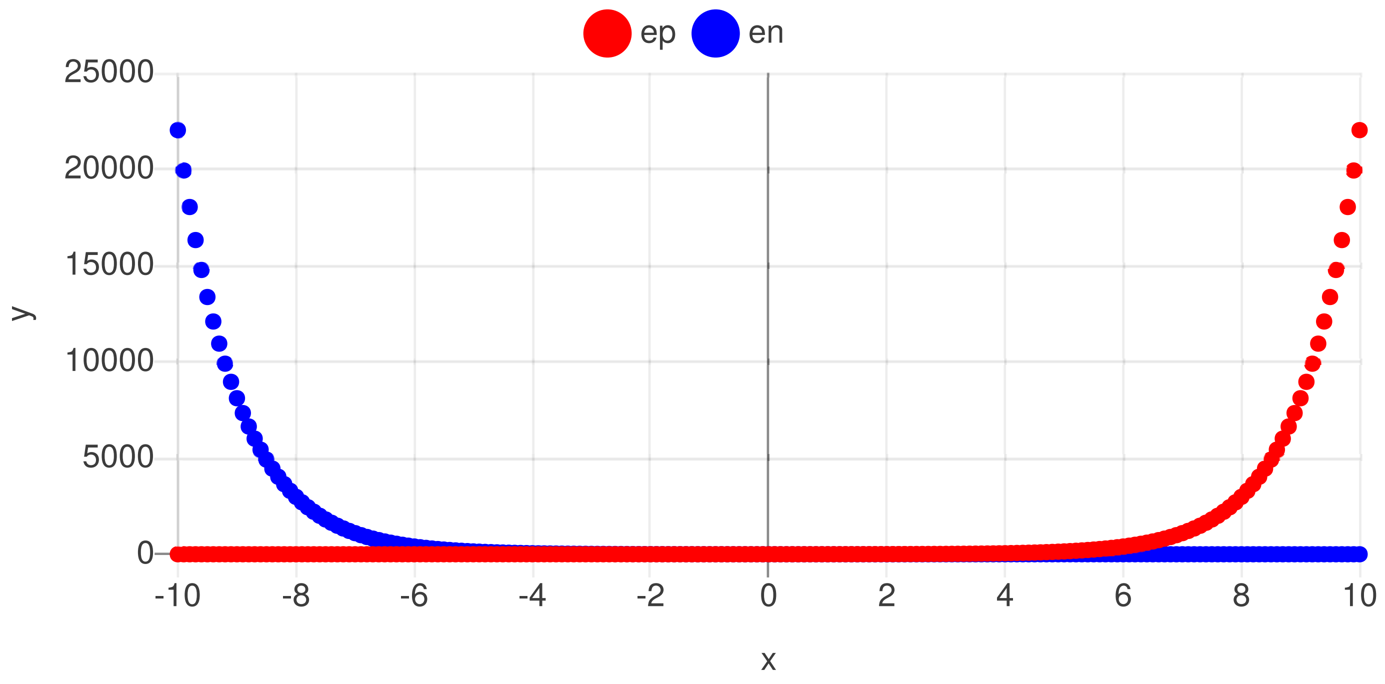


Figure 5. Exploding output values for negative x-values (e^{-x} term) and positive x-values (e^{x} term) of the exponential function.

tanh can be rewritten as shown in the following Eq., computing the discretized *tanh* using the same approach as used for the *sigmoid* function:

$$tanh(x) = sgn(x)\left(1 - \frac{2}{e^{2|x|}+1}\right) \tag{3}$$

The LUT table can be computed with a stretched x distribution as follows, assuming $\Delta x = 1, 2, 3, \ldots$:

$$log10lut = \left\{ int\left(log_{10}\left(\frac{i}{10}\right)100\right) : i \in \mathbb{I}, 10 \le i < 100 \wedge (i-10)\%\Delta x = 0 \right\} \tag{4}$$

with % as the modulo operation that creates an equidistant series of values. The *log10lut* table size is $90/\Delta x$ with an unsigned byte data type. The accuracy (relative error) of the sigmoid approximation is plotted in Figure 6 with an input and output scaling factor of 10 for different LUT sizes. The LUT sizes were 90, 45, and 23, respectively. Using Δx larger than 1 results in a significantly increased approximation error for small x-values (20%), but the average relative error rises only from 1% to 3%.

Algorithm 1. Range-segmented and LUT-based implementation of the sigmoid and hyperbolic tangent functions with less than 1% approximation error for a wide range of x-values (using approximated LUT-based log10 function). Shown is the C program code. The data types are in the format: s = signed or u = unsigned, b = byte, and the number gives the number of bytes.

```
static ub1 log10lut[] = { <90/DX values> }
// x-scale is 1:10 and log10-scale is 1:100
sb2 fplog10(sb2 x) {
  sb2 shift=0;
  while (x>=100) { shift++; x/=10; };
  return shift*100+(sb2)log10lut[x-10];
}
static ub1 sglutA[] = { <24 values> };  // alt. ub2
static ub1 sglutB[] = { <6 elements> };
// y scale 1:1000 [0,1], x scale 1:1000
sb2 fpsigmoid(sb2 x) {
  sb2 y;
  ub1 mirror=x<0?1:0;
  if (mirror) x=-x;
  if (x>=RC1*1000) return mirror?0:1000;
  if (x<=RA1*1000) {
    y = 500+((x*231)/1000);
    return mirror?1000-y:y;
  } else if (x<RB1*1000) {
    ub2 i10 = fplog10((x/5)/2)-OA1;
    y = ((sb2)sglutA[i10])+YA1;
    return mirror?1000-y:y;
  } else {
    ub2 i10 = fplog10((x/10)/10)-OB1;
    y = ((sb2)sglutB[i10])+YB1;
    return mirror?1000-y:y;
  }
  return 0;
}
static ub1 thlutA[] = { <24 values> };  // alt. ub2
static ub1 thlutB[] = { <6 elements> };
// y scale 1:1000 [0,1], x scale 1:1000
sb2 fptanh(sb2 x) {
  sb2 y;
  ub1 mirror=x<0?1:0;
  if (mirror) x=-x;
  if (x>=RC2*1000) return mirror?-1000:1000;
  if (x<=RA2*1000) {
    y = (x*920)/1000;
    return mirror?-y:y;
  } else if (x<RB2*1000) {
    ub2 i10 = fplog10((x/2)/2)-OA2
    y = thlutA[i10]+YA2;
    return mirror?-y:y;
  } else {
    ub2 i10 = fplog10((x/10)/10)-OB2,
    y = thlutB[i10]+YB2;
    return mirror?-y:y;
  }
  return 0;
}
```

The *fpsigmoid* function LUTs are computed iteratively using the *fplog10* function for LUT index stretching, described by the following pseudo-code in Algorithm 2. The symmetry of the sigmoid function is exploited by just computing the positive normalized x-range and applying mirroring and flipping for negative values. The positive function is approximated by four segments. The normalized x-range [0, 1] is handled by a linear function directly computable in the range [0, *RA*], followed by the first highly non-linear segment in the range (*RA*, *RB*), and the converging segment in the range (*RB*, *RC*), and finally a constant segment ($x \geq RC$).

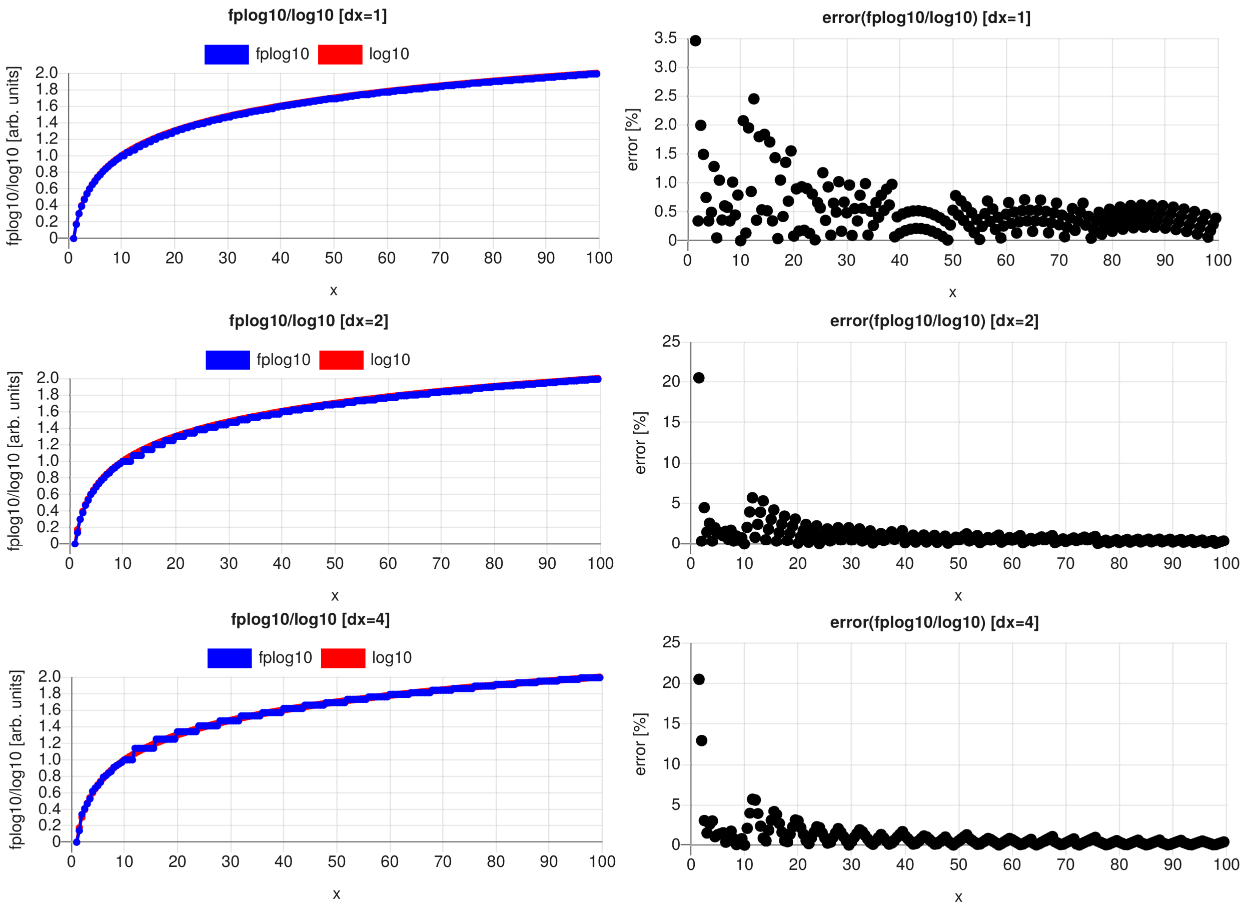


Figure 6. Relative discretization error of integer-scaled LUT-based approximation of the *log10* function for different Δx values (1,2,4) and LUT sizes of 90, 45, and 23, respectively.

Algorithm 2. Computation of the segmented LUTs A/B for the integer-scaled sigmoid and hyperbolic tangent functions. F is the real-valued generator function for sigmoid or tanh LUT computation. The log10 function is used to stretch the x distribution in the LUTs.

```
RA := 1 RB := 3 RC := 10 δ=0.01
OA := int(fplog10(int(x*1000/5))/2)
OB := int(fplog10(int(x*1000/10))/10)
YA := int(F(RA)*1000)
YB := int(F(RB)*1000)
lutA := []
for x=RA to RB step δ do
  i10 := int(fplog10(int(x*1000/5))/2)-OA
  if lutA[i10] = undefined then
    lutA[i10] := int(F(x)*1000)-YA
  endif
done
lutB := []
for x=RB to RC step δ do
  i10 := int(fplog10(int(x*1000/10))/10)-OB
  if lutB[i10] = undefined then
    lutB[i10] := int(F(x)*1000)-YB
  endif
done
```

The accuracy (relative error) of the sigmoid approximation is plotted in Figure 7 with an input and output scaling factor of 10,000 (i.e., 1:10,000). For $x > -3$, the error is below 5% and decreases to 1% on average. For $x < -3$, the relative error increases significantly due to the integer discretization error. The error increases in some x-ranges for lower *fplog10* resolution (smaller LUT sizes, $\Delta x = 4$) but can be improved if the sigmoid interval ranges R are shifted towards larger values (increasing the sigmoid LUT, too). The red and green areas show lowered or increased accuracy. The accuracy of the transfer function itself is not a measure of the accuracy of ML models using this function, especially if post-trained (adapted) using the discretized function instead of the continuous function. For the segment ranges $R = [A = 1, B = 3, C = 10]$, the sizes of the LUTs are $|sglutA| = 24$ and $|sglutB| = 6$, for $R = [1, 7, 15]$, the sizes are $|sglutA| = 43$ and $|sglutB| = 8$, approximately $|sglutA| \approx 6(B - A + 1)$ and $|sglutB| \approx C - B$, in addition to the LUT size of the *fplog10* function $(90/\Delta x)$, requiring in total $6(B - A + 1) + C - B + 90/\Delta x$ Bytes of static storage.

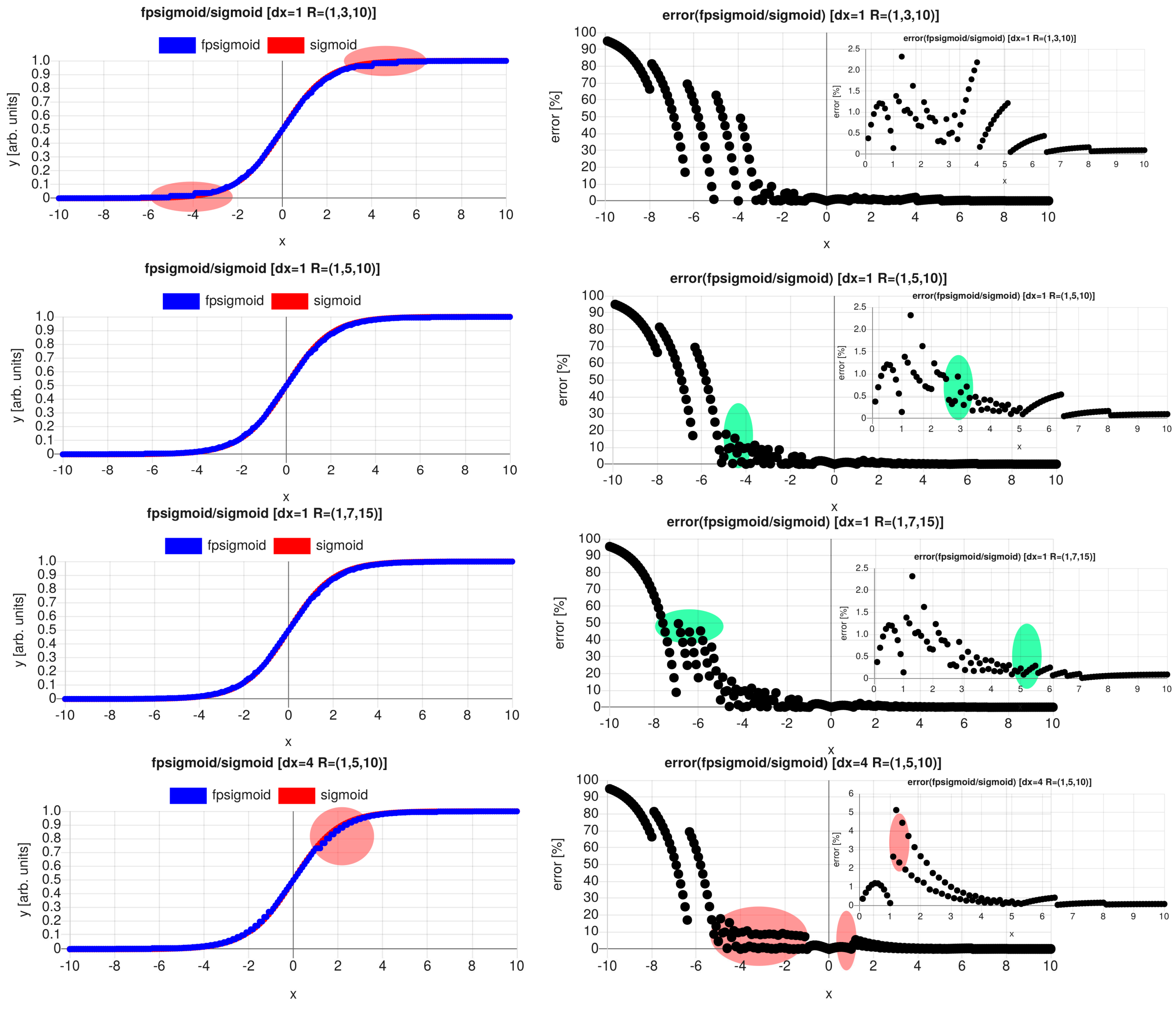


Figure 7. Relative discretization error of integer-scaled LUT-interpolated approximation of the *sigmoid* function using the discretized *log10* LUT-interpolation function for different LUT resolutions and sigmoid segment ranges R. The small error plots show only positive x values.

The implementation of the integer version of the *tanh* function requires extended LUTs due to the higher gradient in the x-range [0, 1], i.e., choosing $RA < 1$. Typical results compared with the real-valued function are shown in Figure 8. Selected error statistics of the *fpsigmoid* and *fptanh* functions are shown in Table 5. The median error is mostly below 1%. Higher errors commonly occur with small y values as a result of integer discretization and not the approximation itself. The LUT sizes vary between 20 and 50 elements and are small enough to be stored in very low-resource micro-controllers, even if a 16-bit data word size is required.

Table 5. Relative discretization error (in %) of the *fpsigmoid* and *fptanh* functions for different parameter settings (only for x > 0). Storage types: B = Byte (8-bit), W = Word (16-bit).

F	Δx	RA	RB	RC	LUTL	LUTA	LUTB	Min	Mean	Median	Max
fpsigmoid	0.5	1	3	10	179B	24B	6B	0.04	0.55	0.34	2.32
fpsigmoid	1	1	3	10	90B	24B	6B	0.04	0.55	0.34	2.32
fpsigmoid	4	1	3	10	23B	24B	6B	0.05	0.78	0.39	5.13
fpsigmoid	1	1	5	10	90B	35W	4B	0.05	0.40	0.20	2.32
fpsigmoid	1	1	5	7	90B	35W	3B	0.004	0.38	0.20	2.32
fpsigmoid	0.5	1	5	7	90B	50W	3B	0.004	0.42	0.21	2.32
fptanh	0.5	0.5	3	10	179B	39W	6B	0.04	0.61	0.10	9.52
fptanh	0.5	1	3	10	179B	24B	6B	0.05	1.00	0.10	17.13
fptanh	0.5	0.5	3	7	179B	39W	5B	0.00	0.59	0.10	9.52
fptanh	1	1	3	10	79B	24B	6B	0.05	0.61	0.10	9.52

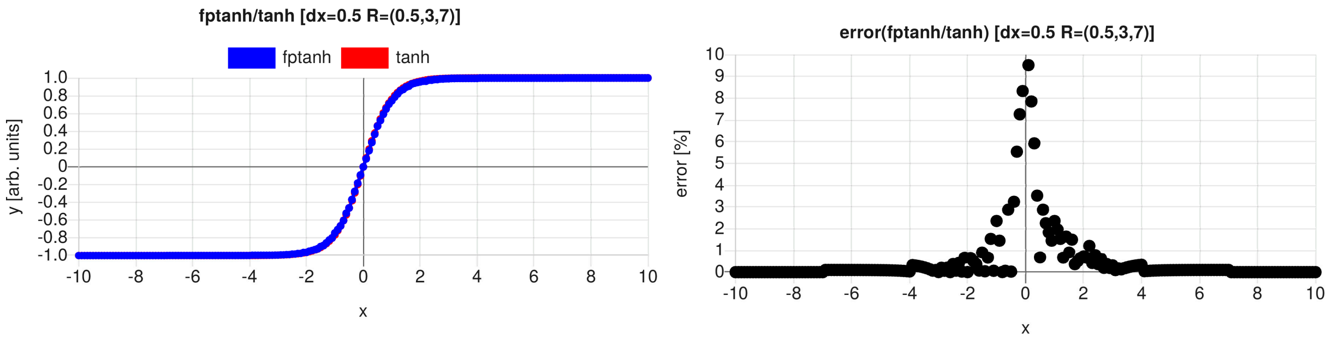


Figure 8. Relative discretization error of integer-scaled LUT-interpolated approximation of the *tanh* function using the discretized *log10* LUT-interpolation function.

To summarize, the accurate approximation of the highly non-linear and widely used *sigmoid* and *tanh* functions is possible with a segmented LUT approach. The computational complexity is low (less than 20 unit operations are required for one function evaluation), and the storage requirement is low (about 200 Bytes for each function). The average relative error is below 1%, except for small integer values limited by the discretization error.

5. Transformation Process

In the following, we introduce the transformation process of a floating-point arithmetic model to a scaled fixed-point (FP) integer arithmetic model. For historical reasons, we call the floating-point model Foo and the fixed-point integer-scaled model FooFP, and the same is true for the activation function, Act and ActFP, respectively. The transformation process can be summarized as follows:

1. Training of model using floating-point arithmetic using a generic ML software frameworks like Neataptic [16] or ConvNetJS [17];
2. Validation of model using test data to estimate model accuracy;
3. Transformation of original model in an annotated unified surrogate Foo model for later analysis and FooFP transformation and model comparison;
4. Restructuring and refactoring of Foo model (layer expansion, i.e., mapping three-dimensional tensors on vectors suitable for vector operations);
5. Statistical analysis (value boundary scans) of data flow in the Foo model using the entire data set;
6. Calculation of scaling factors (static or dynamic scaling);

7. Transformation of the unified Foo model to a surrogate FooFP model using integer-scaled arithmetic;
8. Test of FooFP using discretized and scaled test data and comparison with results from Foo model with respect to overflow (should not occur) and model accuracy deviation;
9. Transforming layers into a sequence of MLISA vector operations;
10. Test MLISA integer model using the (simulated) VM and with the integer data set and validate with Foo/FooFP models.

For the sake of simplicity and modifiability, we use two JavaScript-based ML frameworks:

1. *Neataptic* primarily for pure ANN models [16];
2. *ConvNetJS* for CNN models (including ANN) [17].

Both frameworks provide direct access to the network layers, the forward and backward functions, and enable easy modification to support the model transformation process and perform statistical analysis. Our approach can be used with other frameworks, e.g., PyTorch and Tensorflow, although it is more difficult to implement our algorithms.

There are three phases in the model transformation:

1. Mapping the internal framework model (e.g., ConvNetJS) to an annotated functional unified standard model (USM) aligned to the operational vector semantics of the VM MLISA and refactoring if necessary (see Figure 9);
2. Annotation of the USM with statistics derived from analysis of model parameters, input, intermediate, and output data;
3. Calculation of scale factors based on the statistical analysis and replacement of floating-point vector operations with integer-scaled vector operations (for simulation), arranged in a sequential list of MLISA vector and scale operations.

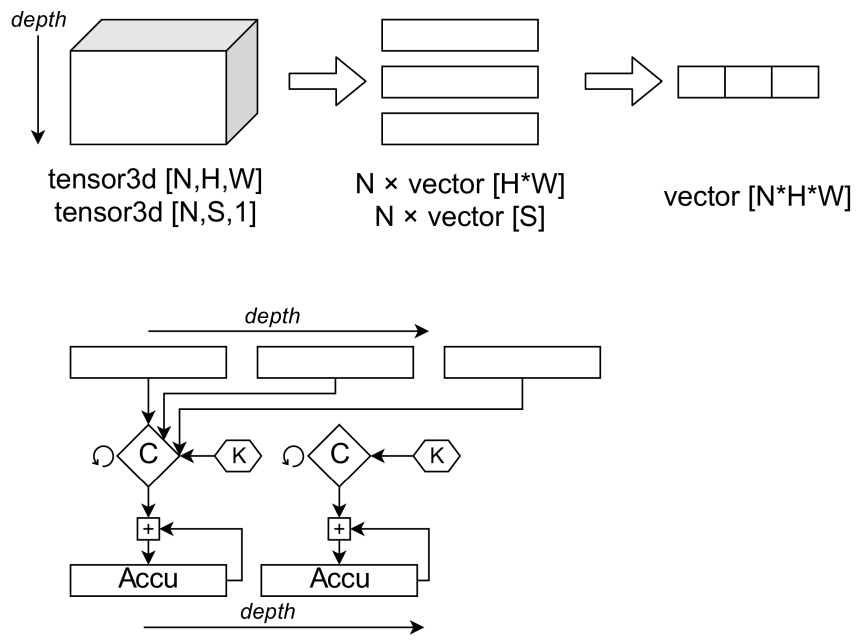


Figure 9. Phase 1 transformation (CNN). (**Top**) Transformation of 3-dim tensors into multiple vectors for convolutional and pooling layers and flattening of multiple vectors from last convolutional or pooling layer into one vector for the input of a fully connected neuronal layer. (**Bottom**) Convolutional and pooling operations factorized into sequential and accumulated vector operations.

The sliced and sequential accumulated convolution, pooling, and product-sum calculation of fully connected neuronal layers is required to match the MLISA vector operations provided by the REXA VM, as shown principally in Algorithm 3. If a previous layer has a depth (z) ordering, i.e., a result of a multi-filter convolution operation, the following layer must process the output for each z-layer independently using slicing and accumulators and, finally, fusion by the *vecsumn* operation. Each slice has its own weight or filter coefficient set.

Algorithm 3. Principle factorization of sliced accumulated operations

```
( Layer i-1=3 )
( Output of layer i-1 )
array yL3N0 62
array yL3N1 62
array yL3N2 62
array yL3N3 62

( Layer i=4 )
( Output of layer i )
array yL4N0 60
array yL4N1 60
array yL4N2 60
array yL4N3 60

( Accumulators of layer i shared by all nodes )
array aL4S0 60
array aL4S1 60
array aL4S2 60
array aL4S3 60

( Filter weights for each slice and node )
array wL4N0S0 { .. }
array wL4N0S1 { .. }
..
: forward
  ( First Conv filter operation N=0 )
  yL3N0 wL4N0S0 aL4S0 -5000 62 3 1 0 vecconv
  yL3N1 wL4N0S1 aL4S1 -5000 62 3 1 0 vecconv
  yL3N2 wL4N0S2 aL4S2 -5000 62 3 1 0 vecconv
  yL3N3 wL4N0S3 aL4S3 -5000 62 3 1 0 vecconv
  aL4S0 aL4S1 aL4S2 aL4S3 yL4N0 0 4 vecsumn

  yL4N0 yL4N0 $ relu 0 vecmap
```

The statistical analysis is split into a static and a dynamic analysis. The dynamic analysis applies the phase-1 transformed Foo ML model (still floating-point arithmetic) to all training and test data samples. The *fivenum* statistics (minimum, maximum, median, and first and third quantiles) are recorded for the input vectors $\boldsymbol{x}$ of a layer, the output vectors $\boldsymbol{y}$, and intermediate values like the sum output of a neuron. Additionally, statistics of static parameters like weights and biases are recorded.

5.1. Scaling

The floating-point numbers must be scaled for the target data type range, e.g., $N = 16$ bit signed integer. The set of values consists of input, output, and intermediate data, convolutional filter coefficients, and weight parameters. The following three significant issues arise:

1. Discretization error (relative and absolute);
2. Underflow (zero);
3. Overflow (modulo N or clipped to max(N)).

To avoid overflows, the scaling factor should be lowered such that the maximum value does not exceed about 0.7 max(N), i.e., introducing a safety margin, e.g., limiting the secure value range of a signed 16-bit integer value to [−10,000, 10,000]. If an ML model is applied to new unknown input data, this secure range can be left without exceeding the real value range of any input, intermediate, and output variables. If the model poses (high) non-linearity, the behavior is unpredictable for unknown data (layer accumulative over- or underflow errors). To avoid increased discretization and underflow errors, the scaling should be lifted, especially between layers, but with some constraints discussed later on.

The idealistic scaling function is:

$$\begin{array}{l} a_x = min(x) \\ k_x = \frac{x_{i,\max}}{max(x)-min(x)} \\ x_i = (x_r - a_x) \cdot k_x \end{array} \tag{5}$$

with x_i as the target scaled (fixed point) integer value and x_r as the original real (floating point) value. This scaling would exploit the full integer value range (including the safety margin), but accumulative (sum) or sign-dependent operations like *relu* would fail due to the origin shift. An improved origin and sign-preserving scaling is then:

$$\begin{array}{l} m_x = \begin{cases} |min(x)| & \text{if} \quad |min(x)| > |max(x)| \\ |max(x)| & \text{if} \quad |max(x)| > |min(x)| \end{cases} \\ k_x = \frac{x_{i,\max}}{m_x} x_i = x_r \cdot k_x \end{array} \tag{6}$$

The scaling is not part of the values. Instead, the scaling factors are static parameters of the model.

Different scaling architectures for functional nodes (neurons) and convolution and pooling nodes are shown in Figure 10. There is symmetric scaling with the same input and output scaling and asymmetric scaling with different scaling factors on the input and output. The activation function expects a specific input and output scaling, therefore requiring intermediate re-scaling to meet this constraint. For instance, the *fpsigmoid* and *fptanh* functions discussed in Section 4.3 expect a static x- and y-scaling of 1000. The weights of neuronal nodes and the kernel coefficients of convolutional nodes are scaled based on the model analysis (minimum and maximum). The bias scaling is the same as for the weights. A convolutional layer applies *n* different filters to the input data, which can be one linear vector or multiple vectors from a previous convolutional layer. The filter dimension can be considered an additional data depth dimension. A neuronal network layer always flattens the depth dimension. The processing of one convolutional operation involves an accumulator that sums the results of the filter application to all input (depth) vectors, as shown in Figure 10 (Bottom).

The dynamic and adaptive re-scaling of intermediate variables and parameters has no effect if a following layer is a discretized LUT-based function (e.g., *tanh*) but can have an effect if there is a non-LUT-based function or if another accumulative (FC/CONV) function is applied.

Things become more difficult if we assume different (optimized) scaling of intermediate values. The finest granularity of dynamic scaling is one vector, e.g., the entire output of a neural node layer or one (depth) output vector of a convolutional or pooling layer gets the same scaling. The following operation processes multiple input vectors sequentially by using an accumulator. If different input vectors have different scaling (fractional to normalized scaling), the scaling must be corrected before summing up the results in the accumulator. Convolutional and neuronal node operations are always product–sum operations, as shown in Figure 11. A scaled product–sum is then given by the following re-scaling:

$$\begin{array}{l} \sigma = \Sigma_i s_x x_i s_w w_i = s_x s_w \Sigma_i x_i w_i \\ \sigma \mid s = \sigma \frac{s_\sigma}{s_w s_x} \end{array} \tag{7}$$

To evaluate the dynamic fine-grained scaling, we compare these models with a globally statically scaled model, i.e., applying a fixed scaling factor to all values, e.g., s_0 = 10,000 for 16-bit signed integer values if the statistical analysis returned a value range within the limits $[-1, 1]$ for all model parameters, input, intermediate, and output values. Therefore, a best guess static scaling is given by *range*/*max*(*M*), where *max*(*M*) is the maximum absolute value of any parameter and any value of the input, intermediate, and output data.

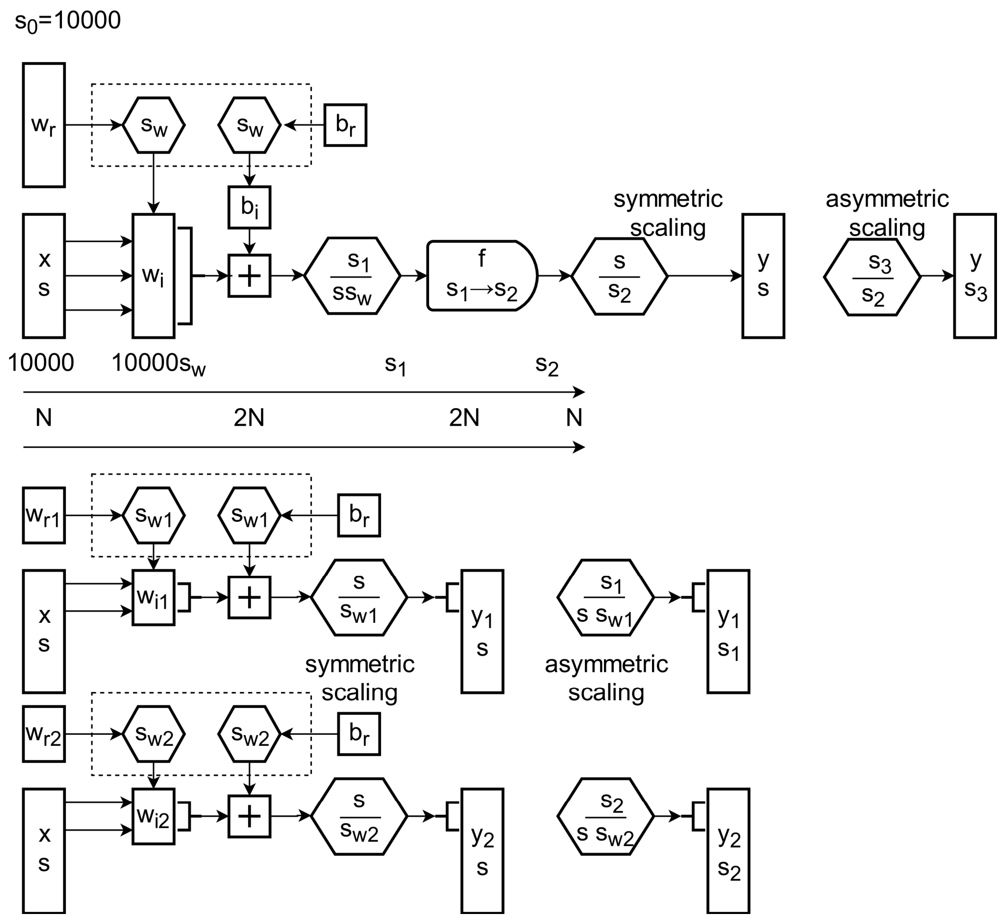


Figure 10. Scaling architectures for (**Top**) functional nodes, i.e., neurons; (**Bottom**) convolution or pooling operation.

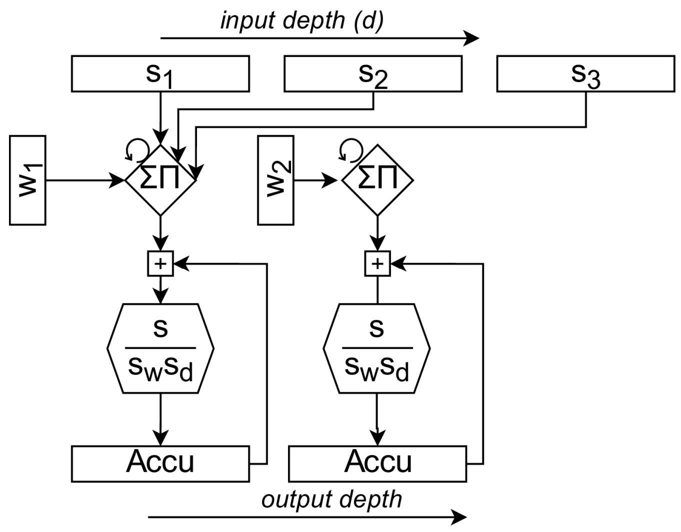


Figure 11. Accumulative scaled convolution or multi-vector input (flattening) neural network operation based on a product–sum calculation. Each accumulative iteration uses a different input scaling s_d normalization with respect to the output scaling s.

Any product–sum calculation with scaled weights (scaling factor s_w) requires a down-scaling of $1/s_w$ afterward, performed directly with the MLISA vector operations, as shown in Algorithm 4.

Algorithm 4. Downscaling of MLISA vector functions assuming a weight scale factor of 5000. The X/Y scaling is not relevant here and must not be adjusted.

```
array X 4
array W { 1000 2000 3000 4000 }
array Y 4
X W Y -5000 vecfold
```

5.2. Workflow

The transformation of the continuous floating-point arithmetic model into scaled discrete integer arithmetic is an iterative process and depends on the specific use case and the training data.

The entire workflow and model processing pipeline is shown in Figure 12. The USM is basically a layer table providing relevant information for each layer L, the layer parameters (weights w and bias b), layer-specific statistical data analysis S, including each layer latent variable output z, a layer surrogate function F using floating-point arithmetic, scaling factors S for each layer and node of a layer (for weight and bias parameters w and b, input and output scaling for each layer, z_{in} and z_{out}, respectively), and finally the integer-scaled and transformed surrogate function FP. The functions F and FP are used for model simulation, under- and overflow analysis, and model error analysis. The MLISA vector operations are derived from the L, P, and S information. Tensor flattening and layer node restructuring are conducted in the first USM transformation phase.

The statistical analysis, as shown in Algorithm 5, must provide value distributions of all model parameters and average statistics of input, intermediate, and output nodes based on the available training (including validation) data. Under- and overflow of integer arithmetic operations must be prevented by choosing the scaling factors with a safety margin. Discretization and rounding errors using integer-scaled arithmetic are accumulative across model layers, requiring a simulation of the scaled model to detect range violations.

The scaled transformation can be static using one fixed model scaling factor s_0 based on the absolute maximum value calculated from all (x,w,b,z,y) values, as shown in Algorithm 6, or dynamically adapting each layer scaling independently to fill the available integer value range optimally (reduced by the safety margin), as shown in Algorithm 7. This is completed by using layer-specific re-scaling factors applied to the global preset factor s_0.

Note that layer parameters are vectors of vectors (i.e., a matrix). One vector is associated with one node of a layer, e.g., a neuron function (vector of weights) or one convolution operation of a convolutional layer (vector of kernel coefficients).

Algorithm 5. Static and dynamic analysis of the pre-transformed continuous USM/Foo model. The layer-specific F function is a surrogate and simulation function using floating-point arithmetic that also performs statistical collection on calling. The *compute* table implements all layer-specific computations, e.g., convolution or application of activation functions. Finally, the global model statistics *stats* = (*min*,*max)* of all values and parameters are computed

```
updateStats = (l,x) => ( l.stats[$x] := l.stats[$x] + fivenum(x) )
for ∀ l ∈ L do:
  if l.parameters.w then updateStats(l,w=l.parameters.w)
  if l.parameters.b then updateStats(l,b=l.parameters.b)
  l.F = (l,x) => ( updateStats(l,x)
                  y=compute[l.type](x,l.parameters)
                  updateStats(l,y)
                  y )
predictFoo = (L,D) =>
  for ∀ d ∈ D do
    x := d
    for ∀ l ∈ L do:
      y := l.F(l,x)
      x := y
  L.stats = □ { ∀ l.stats ∈ L }  // global model statistics
  y
```

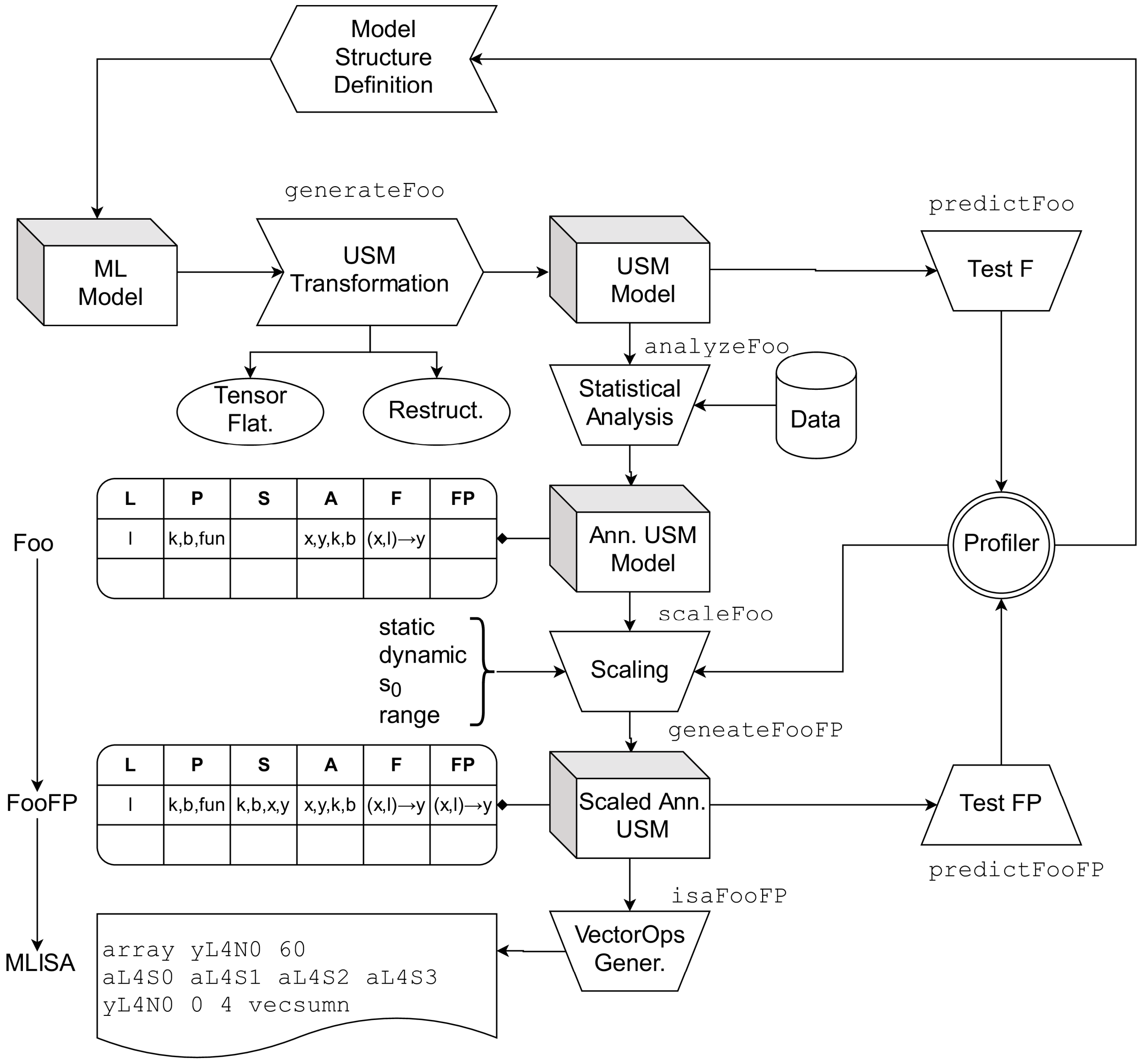


Figure 12. The ML model transformation pipeline creating an intermediate USM and then creating a sequence of MLISA vector operations.

Algorithm 6. Static scaling algorithm transforming a continuous into a discrete model. The layer-specific *FP* function is a surrogate and simulation function using bit-accurate integer-scaled arithmetic. The default scale is s_0, applied to all model parameters and input values. The default range includes a safety margin, e.g., for 16-bit integer, it could be *range* = 10,000 (but maximal about 30,000)

```
s0 = range / max(|L.stats.max|,|L.stats.min|)
for ∀ l ∈ L do
  if l.parameters.w then l.parametersFP.w = scale(l.parameters.w,s0)
  if l.parameters.b then l.parametersFP.b = scale(l.parameters.b,s0)
  l.FP = (l,x) => ( y=computeFP[l.type](x,l.parametersFP,s0)
                    y )
```

Algorithm 7. Simplified dynamic scaling algorithm transforming a continuous into a discrete model. The layer-specific *FP* function is a surrogate and simulation function using bit-accurate integer-scaled arithmetic. The default scale is s_0, applied to all model parameters and input values, but re-scaling factors can modify the default scale, including input and output scaling of layer functions. It is important to keep track of the current layer input and output re-scaling (*reScaleCurrent*).

```
range = 10000
reScaleCurrent := 1
range = 10000 // default ± integer value range
imul = (x,k) => ( if k>0 then int(x*k) else int(x/k) )
idiv = (x,k) => ( if k>0 then int(x/k) else int(x*k) )
adaptScale = (max,scale,range) => ( rescale=range/(max*scale)
                              rescale<0?-int(-1/rescale):
                                       int(rescale))
for ∀ l ∈ L do
  l.scale=s0*reScaleCurrent
  yrescale =
adaptScale(max(|l.stats.y.min|,|l.stats.y.max|),s0*reScaleCurrent,range)
  if not layerHasFixedScale(l) and
      not layerHasFixedScale(next) and
      yrescale>reScaleCurrent then
    l.rescaleY = int(yrescale/reScaleCurrent)
  else if yrescale < 0 then
    if -yrescale > reScaleCurrent or
        (reScaleCurrent%-yrescale) ≠ 0) yrescale = -reScaleCurrent
    l.rescaleY = yrescale
  else
    l.parametersFP.rescaleY  = 1
  if l.parameters.w then
    l.scaleW=s0
    l.rescaleW = adaptScale(max(|l.stats.w.min|,|l.stats.w.max|),s0,range)
  if l.parameters.w then l.parametersFP.w = scale(l.parameters.w,
                                            imul(s0,l.rescaleP))
  if l.parameters.b then l.parametersFP.b = scale(l.parameters.b,
                                            imul(s0,reScaleCurrent))
  if layerHasFixedScale(l) then
        if reScaleCurrent ≠ 1 then
          l.scaleX   = s0*reScaleCurrent
          l.scaleY   = yscale
          l.rescaleX  =-reScaleCurrent
          l.rescaleY  =1
          reScaleCurrent := 1
        else
          l.scaleX=xscale
          l.scaleY=yscale
          l.rescaleX=1
          l.rescaleY=1
        }
  l.FP = (l,x) => ( y=computeFP[l.type](x,l.parametersFP,
                                      l.scaleW,
                                      l.rescaleW,
                                      l.rescaleY,
                                      l.scaleX,
                                      l.scaleY)
                  y )
predictFooFP = (L,D) =>
  for ∀ d ∈ D do
    x := d
    for ∀ l ∈ L do:
      y := l.FP(l,x)
      x := y
  y
```

The final MLISA REXA VM code synthesis creates the necessary data storage (input, intermediate, and output arrays, as well as the parameter arrays). The sharing of arrays is supported for a chain of 1:1 mapping operations, e.g., application of a transfer function.

Sharing of dynamic data storage (array unions) is difficult to implement if the union would contain arrays of different lengths. REXA VM arrays always contain a length header at the beginning, preventing the sharing of different length arrays.

5.3. Unified Model Graph

The previous workflow, consisting of model pre-transformation, analysis, and post-transformation, is merged in only the meta-graph model, as shown in Definition 2.

Definition 2. *Unified model graph merging the original ML model, the USM with its Foo and FooFP surrogate models, and all transformation parameters.*

```
├ type
├ class
├ network : USM network parameters
│ └ layers : layer []
│   ├ layer1
│   │ ├ layer_type : string
│   │ ├ input
│   │ ├ output
│   │ ├ parameters
│   │ │ ├ k
│   │ │ ├ b
│   │ │ ├ n
│   │ │ ├ x
│   │ │ └ y
│   │ ├ stats
│   │ │ ├ x
│   │ │ │ ├ min
│   │ │ │ └ max
│   │ │ ├ y
│   │ │ │ ├ min
│   │ │ │ └ max
│   │ │ ├ k
│   │ │ │ ├ min
│   │ │ │ └ max
│   │ │ └ b
│   │ │   ├ min
│   │ │   └ max
│   │ ├ parametersFP
│   │ │ ├ scale
│   │ │ ├ k
│   │ │ └ b
│   │ ├ f  : function (x,l)
│   │ └ fp : function (x,l)
│   ├ layer2
│   ├ layer3 ..
├ trainer : Information about model trainer
├ input : []
├ output : []
└ model : Original model structure (convnet.js)
```

6. Simulation and Data Set

Sampling of experimental measuring data originating from damaged structures is a difficult task with respect to parameter variance, i.e., damage position, size, sensor positions, and so on, and ground truth labeling. Therefore, for this study, we used simulated GUW time-resolved signal data. The signals were simulated using an extended version of the SimNDT simulator [16] based on an elasto-dynamic finite integration technique [18]. A transmission GUW experiment commonly utilizes two transducers, one generator (pitch signal), and one sensor (catch signal). The generator signal was a sine wave of base frequency 40 kHz and a Gaussian mask window (5 cycles). The simulation was carried out with a time step of 0.06 µs, a total of 5000 steps (300 µs), with each tenth step recorded. In total, 7×6 damage positions were simulated. Circular damage (air, 30 mm diameter) placed at a specific center location (x,y) modifies the GUW signals, as shown in Figure 13. The host material was a plate of 500×500 mm with high absorbing damping material at each plate side (to minimize wave reflections at edges and plate sides).

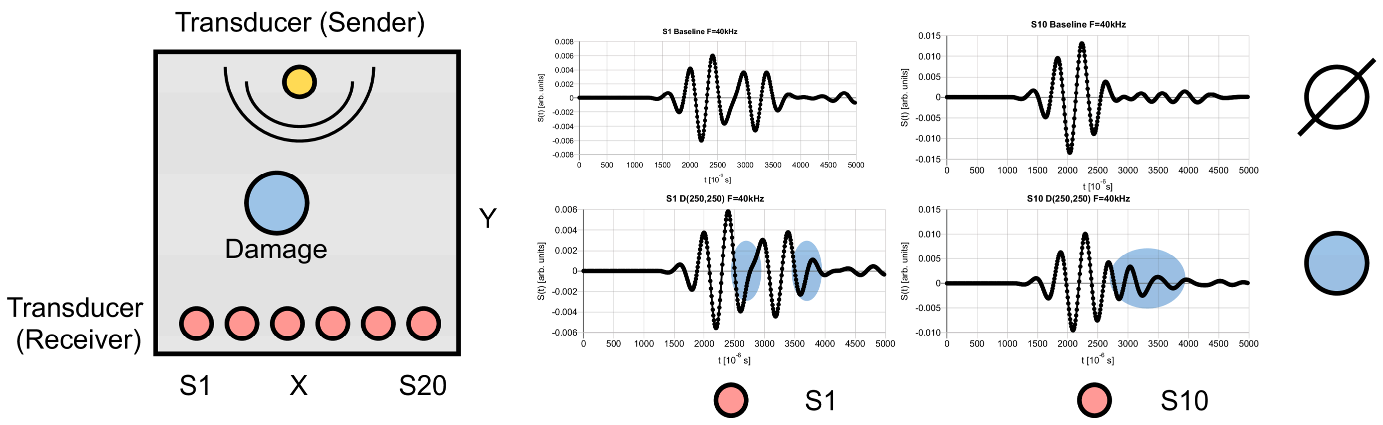


Figure 13. GUW signal simulation using a 2 dim viscoelastic wave propagation model. (**Left**) Simulation set-up. (**Right**) Some example signals with and without damage (blue areas show damage features).

7. Use Case 1: CNN for Damage Location Regression and Classification

The first use case uses the data delivered by the GUW simulation introduced in the previous section. In total, there were 43 different data sub-sets, each related to a specific position of the circular damage in the plate structure, including the baseline measurement without damage. A classical CNN model was chosen to predict the damage positions (x,y) and provide binary damage classification. The CNN input was a down-sampled and low-pass filtered GUW signal (128 data points). The outputs are two continuous variables, p_x and p_y, normalized to the full range of the damage location in the x- and y-direction with a 10% margin, i.e., the minimum location coordinate corresponds to 0.1, and the maximum value corresponds to 0.9. If $p_x < 0.1$ and $p_y < 0.1$, then no damage was detected (i.e., classification output). The model architecture and its parameters are shown in Definition 3. The CNN was implemented with the ConvNetJS framework [17] and trained with the typical 500 epochs at a default learning rate of $\alpha = 0.01$ using the adagrad trainer (batch size was chosen as 1 due to the low sample count).

Definition 3. *Architecture and parameters of the CNN model (ConvNetJS) using 1 dim convolution and pooling operations.*

```
Convolutional Neural Network
============================
Classes:   undefined
Input:     [128,1,1]
Output:    [2]
Layers:    [L8 P13]
[1] input : out=[128,1,1]
[2] conv : in=[128,1,1] out=[124,1,4] k=[5,1] filters=4 stride=1
[3] relu : out=[124,1,4]
[4] pool : in=[124,1,4] out=[62,1,4] k=[2,1] filters=4 stride=2
[5] conv : in=[62,1,4] out=[60,1,4] k=[3,1] filters=4 stride=1
[6] relu : out=[60,1,4]
[7] pool : in=[60,1,4] out=[30,1,4] k=[2,1] filters=4 stride=2
[8] fc : in=[120] out=[16]
[9] tanh : out=[16]
[10] fc : in=[16] out=[8]
[11] tanh : out=[8]
[12] fc : in=[8] out=[2]
[13] regression : in=[2] out=[2]
Predictors: 128
```

A typical GUW signal and its low-pass-filtered and down-sampled version are shown in Figure 14. The low-pass filter was a simple exponential filter with a filter coefficient of $k = 0.2$.

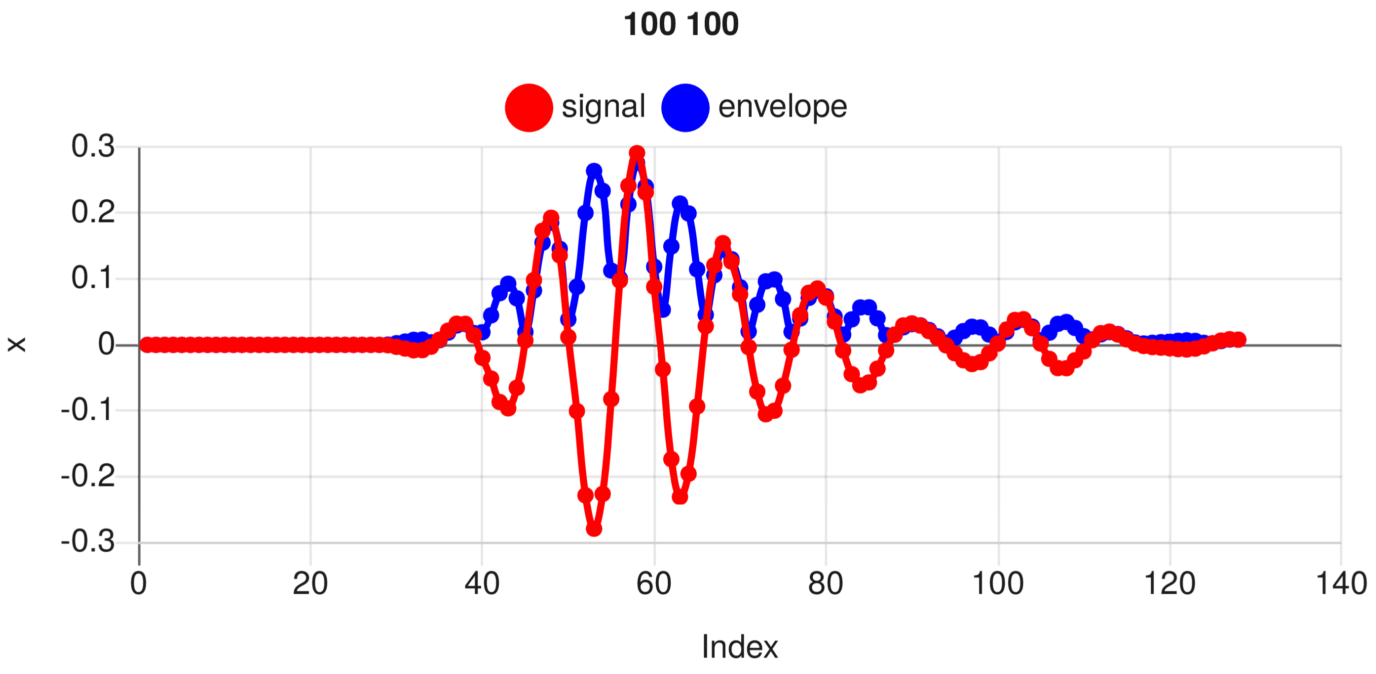


Figure 14. Down-sampled GUW signal from simulation and low-pass-filtered rectified (envelope approximation) signal as input for the CNN (damage at position x = 100, y = 100).

To capture training variations, the original model was trained 100 times, each training starting with a randomly initialized model but always with the same training and test data set. Figure 15 shows the comparison of the prediction results of the continuous Foo and discretized and scaled FooFP model for the regression tasks. The prediction delivers the damage position coordinates, and a non-damage detection is given by a (0,0) value pair (or close to). The RMSE and maximal position prediction errors are computed. Results for static and dynamic scaling were compared. The summary of results is as follows:

1. The total value range of all input, output, intermediate, and parameter values depends on the particular training of the original model but is mostly in the range $[-4, 4]$ (see Figure 15, *V* column). Therefore, a static or preset scaling of s_0 is in the range [3000, 10,000].
2. The discretization error of integer arithmetic is neglectable for all linear operations but depends slightly on the discretization parameters of the non-linear functions (FP1/FP2/FP5 in Figure 15 represents LUT resolution with $1/\Delta X = 1/2/5$).
3. The dynamic scaling, compared with static scaling, shows no significant improvement in the model accuracy (maximal 5%) but is unexpected with a larger variance (see Figure 15).
4. The overall discretization error depends on a particular model parameter set, i.e., with nearly the same floating-point accuracy, the integer model accuracy can differ significantly (RMSE and $E_{\max}$). Multiple trained models should be analyzed with respect to the final discretization error, selecting the best model.
5. The prediction error of the discretized model (RMSE and $E_{\max}$) differs only slightly.
6. There is no increase in the classification error compared with the continuous model, showing an overall stable prediction behavior.

To conclude, the discretization, even with a moderate static scaling, does not degrade the prediction accuracy.

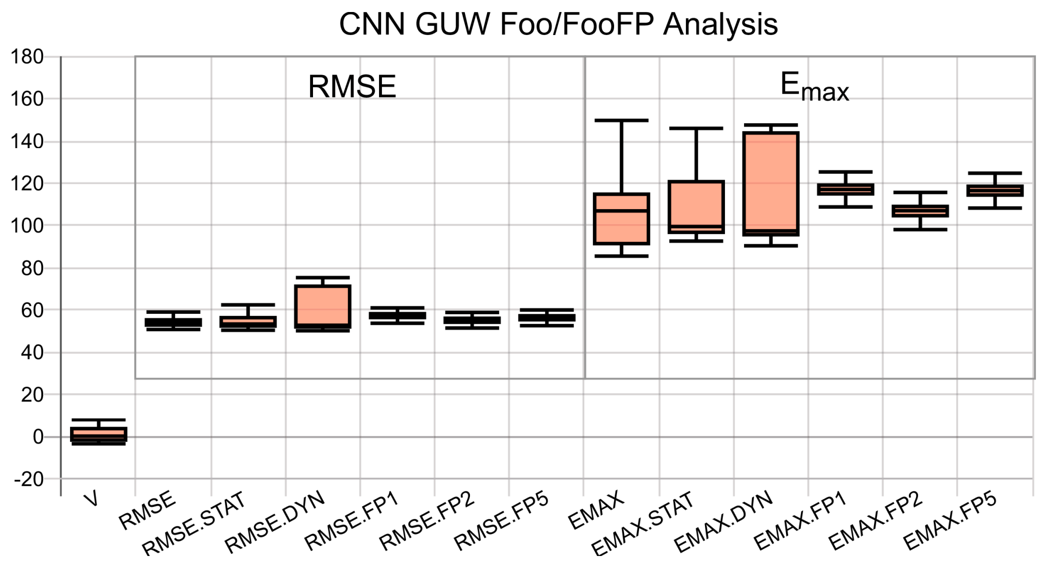


Figure 15. Foo/FooFP model analysis of the GUW regression CNN model. The classification error was always zero.

The MLISA REXA VM Forth program of the discretized model is shown in Appendix A in Algorithm A1. The model code occupies 1746 dynamic and 2166 static words in the CS (i.e., occupying about 8k Bytes of RAM). The entire textural code size is 18,452 Bytes. The forward computation requires the execution of 1280 words, which is equivalent to 85 ms/MHz (assuming 15k/words/s/MHz [11]).

8. Use Case 2: ANN Polynomial Models

Based on the previous use case evaluation indicating non-linear functions as the primary source for approximation errors, we want to force high non-linearity in an ML surrogate model for a polynomial of degree n:

$$F(x) = k_0 + k_1 x + k_2 x^2 + k_3 x^3 + \ldots + k_n x^n \tag{8}$$

The ANN model architecture for the implementation of such a surrogate model is shown in Definition 4. It consists of only 17 neurons. A polynomial of degree 4 was chosen with the following parameters:

$$\begin{aligned} k_0 &= 100 \\ k_1 &= 0.5 \\ k_2 &= 1 \\ k_3 &= -0.5 \\ k_4 &= -0.1 \\ x &= [0, 10] \end{aligned} \tag{9}$$

The model was trained using the polynomial model (500 epochs); 100 independent models were trained using the same training data with 1000 randomly selected function samples. After model training, the Foo transformation process and a model analysis were performed.

Definition 4. *Architecture and parameters of the ANN model (ConvNetJS) as a surrogate regression model for highly non-linear analytical model functions.*

```
Artificial Neural Network
============================
Classes:   undefined
Input:     [1,1,1]
Output:    [1]
Layers:    [L5 P9]
[1] input : out=[1]
[2] fc : in=[1] out=[4]
[3] tanh : out=[4]
[4] fc : in=[4] out=[8]
[5] tanh : out=[8]
[6] fc : in=[8] out=[4]
[7] tanh : out=[4]
[8] fc : in=[4] out=[1]
[9] regression : in=[1] out=[1]
Predictors: 1
```

The following discretization parameters for the activation function were chosen, as shown in Table 6.

Table 6. Different discretization parameters chosen for the activation functions.

Nr.	**1/Δx**	**RA**	**RB**	**RC**
1	1	0.5	3	7
2	3	0.5	3	7
3	5	0.5	3	7
4	7	0.5	3	7
5	10	0.5	3	7
6	10	0.7	3	7

The statistical analysis of the prediction results of the continuous Foo and the discretized FooFP model is shown in Figure 16. The summary of the results is as follows:

1. The prediction errors of the discretized model (RMSE and $E_{\max}$) are significantly higher compared to the continuous model.
2. The discretization error results from the non-linear *tanh* function, which is clearly highlighted if a scaled float-point alternative is used in the discretized model, with errors similar to the continuous model.
3. Choosing the *tanh* and underlying *log10* discretization parameters leads to the prediction error. Modification of the LUT partitioning and interval coefficients can improve the *RMSE* as well as the maximum error $E_{\max}$.
4. Dynamic scaling shows no significant improvement in the model accuracy (maximal 5%).
5. As shown at the bottom of Figure 16, the discretization error is not constant; instead, it introduces discontinuity.

The MLISA REXA VM Forth program of this model is shown in Appendix A in Algorithm A2. The model code occupies 18 dynamic and 89 static words in the CS. The entire textural code size is 921 Bytes. The forward computation requires the execution of 70 words, which is equivalent to 5 ms/MHz.

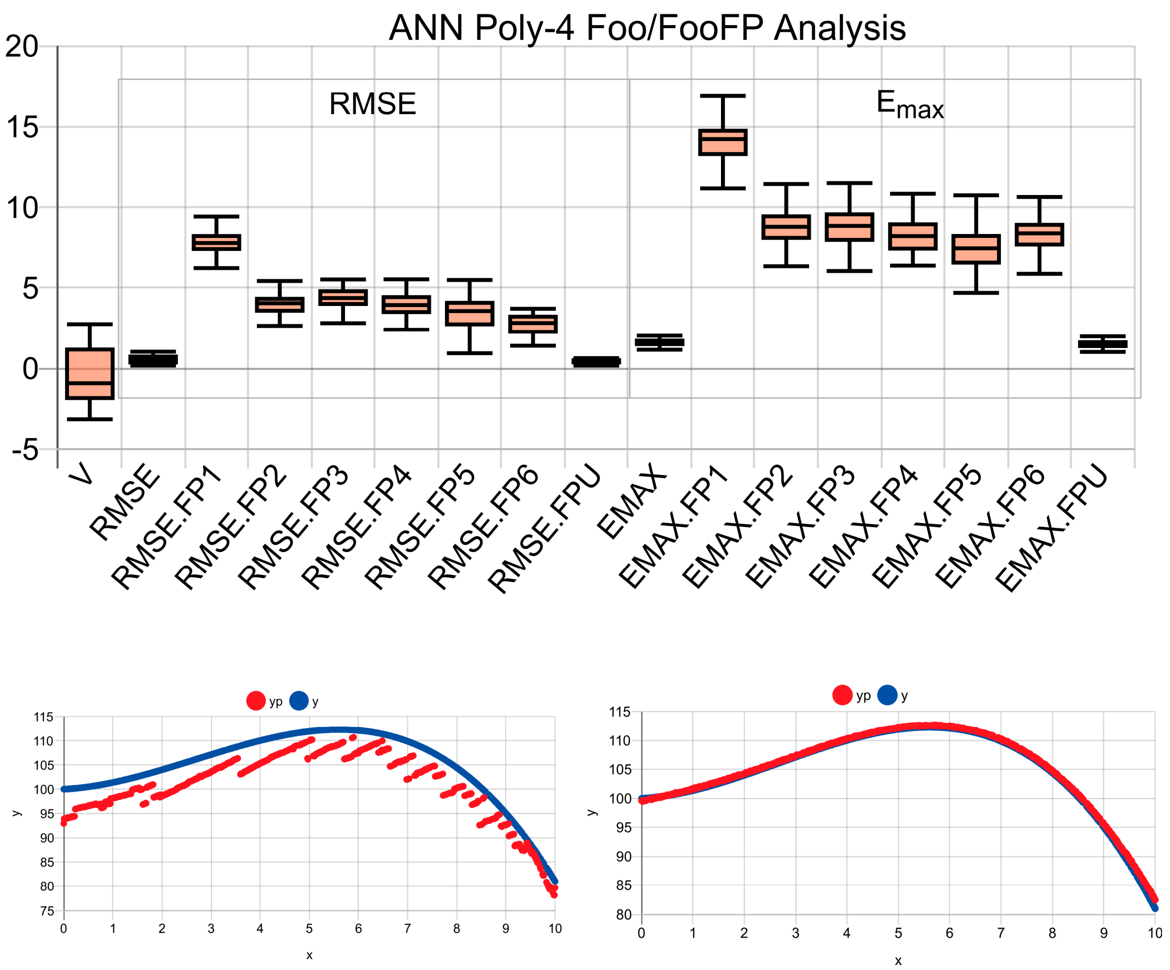


Figure 16. (**Top**) Analysis of the ANN FP and DS models comparing RMSE and E_{max} values for different configurations of the activation function approximation, including an FPU replacement. (**Bottom**) Selected prediction results are shown with discontinuities in the top plot using ActDS configuration 5 and without using the FPU replacement for the tanh function.

9. Discussion

The two use cases clearly showed the benefit of the proposed scaling approach and the simplicity of the VM programming for classification and regression models, including CNN architectures. The results can be summarized as follows:

1. The prediction error of the discretized model compared with the continuous model is comparable if there is no high non-linearity (use case 1) but significantly higher if there is a higher degree of non-linearity (use case 2).
2. Dynamic scaling compared to static scaling shows no significant improvement, only for very low default s_0 scaling factors.
3. Model optimization with respect to the average classification or RMSE and peak regression errors is possible via the optimization of the non-linear piecewise-segmented and LUT-based activity functions (*sigmoid*, *tanh*). The optimization requires the modification of function approximation parameters, but these functions are statically built into the VM (as a service).
4. Even if the micro-controller provides an FPU, the VM should continue using 16-bit integer arithmetic to satisfy the still remaining hard memory limits. Moreover, the JIT run-time compiler translates text-to-byte-code in place. For instance, a constant value “0” can always be replaced by a binary 16 Bit container since tokens are separated by space or newline characters. A 32-bit engine using the FPU would make this

approach impossible, and the memory footprint would increase significantly (doubles at maximum).

5. The REXA VM implementation of the discretized models requires typical code sizes (including data) of about 1–20k Bytes, which can be transferred using (RFID) wireless communication. Even more complex models can be processed by a 16-bit VM with a code segment size limit of 32k Bytes.
6. The average computation times of models with the REXA VM range from 1 to 100 ms, which is fully sufficient even for ad-hoc remotely powered sensor nodes via RFID fields.

10. Conclusions

In previous work [6], we investigated the effect of ML model discretization in only classification tasks. This work investigated the effect of integer-scaled discretization for regression tasks with two use cases as well and presented the model transformation and scaling algorithms in detail. The first use case considers time-dependent ultrasonic signals as an input for a damage location regression model. The second use case uses synthetic data from a highly non-linear polynomial function to investigate the impact of discretization of the non-linear activation functions. Static (one scaling factor for the entire model) was surprisingly fully sufficient, and dynamic fine-grained scaling of different stages of the model does not improve the overall prediction accuracy of the model. The highest impact on the model accuracy is the discretization and step-wise approximation of non-linear (activation) functions. As an outlook, the model activation functions could be generated for a specific model at run-time by the VM based on parameters provided by the transformed model. However, the generation of the approximated functions requires floating-point versions of at least the *log* and *e* functions, optimally by the floating-point version of the activation functions. If floating-point functions are available, the non-linear activation functions could be directly calculated without approximation and discretization errors (at least significantly lower errors). One solution could be generic activation function templates that can use different LUTs transferred separately to the VM. Finally, the impact of integer discretization on the accuracy of recurrent state-based neural networks should be investigated.

Funding: The authors expressly acknowledge the financial support of the research work on this article within the Research Unit 3022 "Ultrasonic Monitoring of Fibre Metal Laminates Using Integrated Sensors" (Project number: 418311604) by the German Research Foundation (Deutsche Forschungsgemeinschaft (DFG)).

Data Availability Statement: There is no experimental data used in this work. The code is published and referenced in the paper.

Conflicts of Interest: There are no conflicts of interest.

Appendix A

Appendix A.1 Abbreviations

Act	Activation and transfer function (using floating point arithmetic)
ActFP	Discretized and scaled activation function for fixed-point integer arithmetic
FooFP	Discretized and scaled Model for integer arithmetic
Foo	Floating point model
Forth	Stack-based programming language with reverse polish notation
GUW	Guided Ultrasonic Waves (Lamb wave)
MLISA	Machine Learning Instruction Set Architecture
REXA	Real-time capable and Extensible Architecture
SHM	Structural Health Monitoring
VM	Virtual Machine

Appendix A.2 MLISA Code

Algorithm A1. Shortened MLISA code of discretized regression GUW CNN model (use case 1)

```
( Layer 1 input )
array xL0 128
( Layer 2 conv )
array yL1N0 124 array yL1N1 124 array yL1N2 124 array yL1N3 124
array bL1 { 1129 ..(3) }
array wL1N0 { 1299 ..(4) } array wL1N1 { -5119 ..(4) }
array wL1N2 { 2334 ..(4) } array wL1N3 { -4736 ..(4) }
( Layer 3 relu )
( Layer 4 pool )
array yL3N0 62  array yL3N1 62
array yL3N2 62  array yL3N3 62
array aL3 124
( Layer 5 conv )
array yL4N0 60  ...
array bL4 { 1159 ..(3) }
array aL4S0 60  array aL4S1 60
array aL4S2 60  array aL4S3 60
array wL4N0S0 { -750 ..(2) } ...
...
( Layer 8 fc )
array yL7 16
array bL7 { 939 ..(15) }
array aL7S0 16 array aL7S1 16
array aL7S2 16 array aL7S3 16
array wL7N0S0 { -407 ..(29) }
...
( Layer 9 tanh )
( Layer 10 fc )
array yL9 8
array bL9 { 103 ..(7) }
array wL9 { 3962 ..(127) }
( Layer 11 tanh )
( Layer 12 fc )
array yL11 2
array bL11 { -355 ..(1) }
array wL11 { -1915 ..(15) }
( Layer 13 regression )
: forward
  ( Layer 1 input )
  ( Layer 2 conv )
  xL0 wL1N0 yL1N0 -5000 128 5 1 0 vecconv
  xL0 wL1N1 yL1N1 -5000 128 5 1 0 vecconv
  xL0 wL1N2 yL1N2 -5000 128 5 1 0 vecconv
  xL0 wL1N3 yL1N3 -5000 128 5 1 0 vecconv
  ( Layer 3 relu )
  yL1N0 yL1N0 $ relu 0 vecmap
  ...
  ( Layer 4 pool )
  yL2N0 1 aL3 -5000 124 -2 2 0 vecconv
  aL3 1 aL3 -5000 124 -2 2 0 vecconv
  aL3 1 aL3 -5000 124 -2 2 0 vecconv
  aL3 1 yL3N0 -5000 124 -2 2 0 vecconv
  yL2N0 1 aL3 -5000 124 -2 2 0 vecconv
  aL3 1 aL3 -5000 124 -2 2 0 vecconv
  aL3 1 aL3 -5000 124 -2 2 0 vecconv
  aL3 1 yL3N1 -5000 124 -2 2 0 vecconv
  ...
  ( Layer 5 conv )
  yL3N0 wL4N0S0 aL4S0 -5000 62 3 1 0 vecconv
  yL3N1 wL4N0S1 aL4S1 -5000 62 3 1 0 vecconv
  yL3N2 wL4N0S2 aL4S2 -5000 62 3 1 0 vecconv
  yL3N3 wL4N0S3 aL4S3 -5000 62 3 1 0 vecconv
```

Algorithm A1. *Cont.*

```
    aL4S0 aL4S1 aL4S2 aL4S3 yL4N0 0 4 vecsumn
    ...
    ( Layer 6 relu )
    yL4N0 yL4N0 $ relu 0 vecmap
    ...
    ( Layer 7 pool )
    yL5N0 1 aL6 -5000 60 -2 2 0 vecconv
    aL6 1 aL6 -5000 60 -2 2 0 vecconv
    aL6 1 aL6 -5000 60 -2 2 0 vecconv
    aL6 1 yL6N0 -5000 60 -2 2 0 vecconv
    ...
    ( Layer 8 fc )
    yL6N0 aL7S0 wL7N0S0 0 vecfold
    yL6N1 aL7S1 wL7N0S1 0 vecfold
    yL6N2 aL7S2 wL7N0S2 0 vecfold
    yL6N3 aL7S3 wL7N0S3 0 vecfold
    aL7S0 aL7S1 aL7S2 aL7S3 yL7N0 -5000 4 vecsumn
    ...
    ( Layer 9 tanh )
    yL7 yL7 $ tanh 0 vecmap
    ( Layer 10 fc )
    yL8 yL9 wL9 -5000 vecfold
    yL9 yL9 bL9 0 vecadd
    ( Layer 11 tanh )
    yL9 yL9 $ tanh 0 vecmap
    ( Layer 12 fc )
    yL10 yL11 wL11 -5000 vecfold
    yL11 yL11 bL11 0 vecadd
    ( Layer 13 regression )
  ;
```

Algroithm A2. MLISA code of discretized Poly4 ANN model

```
array xL0 1  array yL1 4
array bL1 [ -281 ..(3) ]  array wL1 [ -1875 ..(3) ]
array yL3 8
array bL3 [ -1016 ..(7) ] array wL3 [ 1537 ..(31) ]
array yL5 4
array bL5 [ 701 ..(3) ]   array wL5 [ 324 ..(31) ]
array yL7 1
array bL7 [ -655 ..(0) ]  array wL7 [ -906 ..(3) ]
: forward
  xL0 yL1 wL1 -5000 vecfold
  yL1 yL1 bL1 0 vecadd
  yL1 yL1 $ tanh 0 vecmap
  yL2 yL3 wL3 -5000 vecfold
  yL3 yL3 bL3 0 vecadd
  yL3 yL3 $ tanh 0 vecmap
  yL4 yL5 wL5 -5000 vecfold
  yL5 yL5 bL5 0 vecadd
  yL5 yL5 $ tanh 0 vecmap
  yL6 yL7 wL7 -5000 vecfold
  yL7 yL7 bL7 0 vecadd
;
```

References

1. Guo, S.; Zhou, Q. *Machine Learning on Commodity Tiny Devices*; Taylor & Francis: Boca Raton, FL, USA, 2023.
2. Ray, P.P. A review on TinyML: State-of-the-art and prospects. *J. King Saud Univ.-Comput. Inf. Sci.* **2021**, *34*, 1595–1623. [CrossRef]
3. Wang, X.; Magno, M.; Cavigelli, L.; Benini, L. FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things. *arXiv* **2022**, arXiv:1911.03314. [CrossRef]
4. Alajlan, N.N.; Ibrahim, D.M. TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications. *Micromechanics* **2022**, *13*, 851. [CrossRef] [PubMed]
5. Jain, V.; Giraldo, S.; Roose, J.D.; Mei, B.B.L.; Verhelst, M. TinyVers: A Tiny Versatile System-on-chip with State-Retentive eMRAM for ML Inference at the Extreme Edge. *arXiv*, **2023**, arXiv:2301.03537.

6. Bosse, S. IoT and Edge Computing using virtualized low-resource integer Machine Learning with support for CNN, ANN, and Decision Trees, IoT-ECAW. In Proceedings of the 18th Conference on Computer Science and Intelligence Systems FedCSIS 2023 (IEEE #57573), Warsaw, Poland, 17–20 September 2023.
7. Banner, R.; Hubara, I.; Hoffer, E.; Soudry, D. Scalable Methods for 8-bit Training of Neural Networks. *arXiv* **2018**, arXiv:1805.11046.
8. Ghaffari, A.; Tahaei, M.S.; Tayaranian, M.; Asgharian, M.; Vahid, P.N. Is Integer Arithmetic Enough for Deep Learning Training? *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 27402–27413.
9. Donati, G.; Zonzini, F.; Marchi, L.D. Tiny Deep Learning Architectures Enabling Sensor-Near Acoustic Data Processing and Defect Localization. *Computers* **2023**, *12*, 129. [CrossRef]
10. Magno, M.; Cavigelli, L.; Mayer, P.; von Hagen, F.; Benini, L. FANNCortexM: An Open Source Toolkit for Deployment of Multi-layer Neural Networks on ARM Cortex-M Family Microcontrollers. In Proceedings of the IEEE 5th World Forum on Internet of Things (WF-IoT), Limerick, Ireland, 15–18 April 2019.
11. Bosse, S.; Bornemann, S.; Lüssem, B. Virtualization of Tiny Embedded Systems with a robust real-time capable and extensible Stack Virtual Machine REXAVM supporting Material-integrated Intelligent Systems and Tiny Machine Learning. *arXiv* **2023**, arXiv:2302.09002.
12. Bornemann, S.; Lang, W. Considerations and Limits of Embedding Sensor Nodes for Structural Health Monitoring into Fiber Metal Laminates. *Sensors* **2022**, *22*, 4511. [CrossRef] [PubMed]
13. Finkenzeller, K. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication*; John Wiley & Sons: New York, NY, USA, 2010.
14. Bosse, S.; Polle, C. Tiny Machine Learning Virtualization for IoT and Edge Computing using the REXA VM. In Proceedings of the 10th International Conference on Future Internet of Things and Cloud (FiCloud 2023), Marrakesh, Marroco, 14–16 August 2023; IEEE Catalog Number: CFP23FIC-ART. ISBN 979-8-3503-1635-3.
15. Hayes, J.R.; Fraeman, M.E.; Williams, R.L.; Zaremba, T. An architecture for the direct execution of the Forth programming language. *ACM SIGARCH Comput. Archit. News* **1987**, *15*, 42–49. [CrossRef]
16. Neataptic. Available online: https://wagenaartje.github.io/neataptic/ (accessed on 1 July 2024).
17. ConvNetJS. Available online: https://cs.stanford.edu/people/karpathy/convnetjs/ (accessed on 1 July 2024).
18. Molero-Armenta, M.; Iturrarán-Viveros, U.; Aparicio, S.; Hernández, M.G. Optimized OpenCL implementation of the Elastodynamic Finite Integration Technique for viscoelastic media. *Comput. Phys. Comm.* **2014**, *185*, 2683–2696. [CrossRef]