Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 367–376
DOI: 10.15439/2023F7745
ISSN 2300-5963 ACSIS, Vol. 35

IoT and Edge Computing using virtualized low-resource integer Machine Learning with support for CNN, ANN, and Decision Trees

Stefan Bosse
0000-0002-8774-6141
University of Bremen
Dept. Mathematics and Computer Science, Bremen, Germany
Email: sbosse@uni-bremen.de

***Abstract*—Data-driven models used for predictive classification and regression tasks are commonly computed using floating point arithmetic preserving accuracy by automatic scaling even in high non-linear functions. With respect to distributed sensor networks like the IoT, sensor data is acquired on low-resource embedded systems and delivered to data servers characterized by big data volumes. In specific use cases and domains, local predictive modelling on low-power devices is desired or required. But heterogeneity of host platforms and dynamic programming disables machine code deployment. This work addresses Tiny ML on very low-resource devices (microcontrollers, less than 32 kB RAM and ROM) by using a stack-based Tiny Virtual Machine providing core ML operations to implement Decision Trees (DT), Artificial Neural Networks (ANN), and Convolutional Neural Networks (CNN). VM program code is always provided in textual format and compiled just-in-time to Bytecode to ensure portability, servicability, and mobility. Two damage diagnostics use-cases demonstrate the suitability of the VM approach, and even time consuming computational tasks do not compromise the overall responsiveness of the platform by using a real-time approach. This work addresses the underlying integer arithmetic operations required to implement efficient and fast computable ML models on microncontrollers.**

***Index Terms*—Tiny ML, Virtualization, Embedded Systems.**

I. Introduction

TO ADDRESS ubiquitous computing, edge computing, and distributed sensor networks, driven by a significant increase in device density and sensor deployment toward smart and self-contained sensors, advanced and dependable data processing architectures are required. Tiny machine learning is a new and challenging field [1]. In order to calculate ML models, high precision floating point arithmetic is frequently used. Only integer arithmetic (8–32 bits) is offered by low-resource tiny embedded systems, therefore direct training using integer arithmetic [2] or model transformation and freezing [3] are required, ideally on the target device itself [4]. These issues are also addressed in our study. Ultra low-power devices place additional restrictions on the computation of deep learning (DL) models [5] and hardware designs are becoming more popular [6]. An example for such a tiny low-resource embedded system is shown in Fig. 1.

This work was supported by DFG Grant 418311604

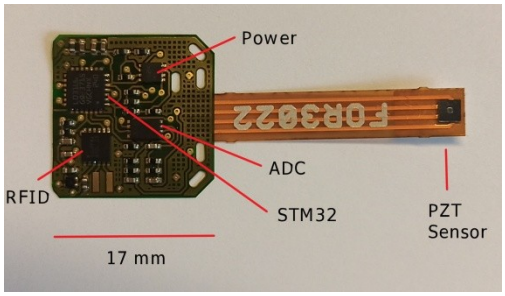


Fig. 1. An example of a highly integrated and miniaturized sensor node with a STM32 ARM Cortex microcontroller supplied entirely by an RFID energy harvester (source with courtesy: IMSAS, B. Lüssem, University of Bremen).

In this study, a real-time capable and extendable application-specific stack virtual machine (REXA-VM) with several distinctive and unique features is introduced and analyzed, specifically addressing ML computations. In contrast to common integer-based ML models using 8 bit scaled arithmetic [2], this VM supports 16 and 32 bit operations. The novelty of this work is the capability of a VM to process common ML models delivered in text format. The program text embeds model parameters as well the forward computation function for a specific already trained model. Virtualization of services and data processing in embedded devices play an important role in heterogeneous network environments [7].

Another problem involves non-continuous energy supply, such as that delivered to the sensor node from external sources utilizing RFID/NFC. This type of non-continuous energy supply introduces severe power restrictions limiting the set of usable microcontrollers (mainly without FPU) and necessitates real-time data processing to the appropriate degree. Running computationally expensive operations without jeopardizing IO event handling (i.e., the device's responsiveness) requires the VM's real-time capability, which is not covered in this work. It is anticipated that a REXA VM node receives remote communications over wireless tech-

nology. Direct transmission of program text code to the VM for processing and compilation is possible.

II Arithmetic

Linear and moreover non-linear functions are commonly computed using at least 32 Bit Floating Point (FP) arithmetic. FP bases on an exponential representation (and approximation) of real numbers. Most ML models are linear or non-linear functions. Deep neuronal networks with non-linear activation functions can approximate highly non-linear models. The dynamic scaling of FP values and operations enables the computation of functions with high gradients and large value interval spans. Although, there is 16 Bit FP arithmetic, this reduced precision arithmetic commonly have no advantages over fixed point arithmetics (XP) except providing a higher dynamic range. The required dynamic range depends on the value range of the input and output variables as well as on latent (hidden) variables of intermediate functions, e.g., hidden nodes of an ANN. XP is often used in hardware implementations of ANNs [8] and rely on integer arithmetic that is the only available arithmetic on low-resource computers, e.g., the STM32 ARM Cortex M microcontroller series. XP arithmetic has the disadvantages of underflow and the requirement of software post-correction (multiplication) when used on microcontrollers, lowering the arithmetic's performance.

In contrast to common numerical approaches, in this work, XP values are replaced by in-advance dynamically scaled arithmetic (SA) using <value,scale> tuples. Scaling is only applied after an aggregating operation was performed, e.g., the computation of a vector product in an ANN layer, as illustrated in Fig. 2. This is relaxed by the fact that N Bit integer arithmetic (e.g., 32) is assumed at the core, but only M=N/2 Bit integer values (e.g., 16 Bits) are used to represent operands and result values. SA is used in this work to approximate complex (nested) functions. The scaling values are computed from data for a particular function, e.g., an ANN classifier. The relative approximation error increases with decreasing (real) values. A scaling factor can be shared by multiple values (e.g., vector elements), reducing memory requirements.

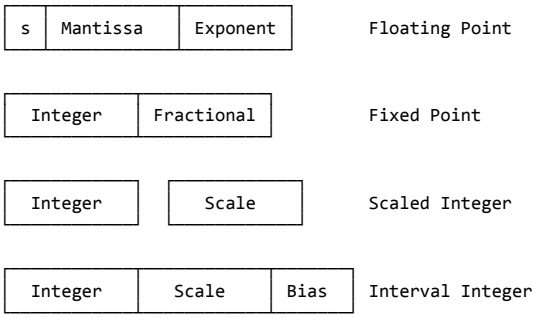


Fig. 2. Comparison of different arithmetic classes

A function F is transformed to a an integer approximation by:

1. Decomposing arithmetic expressions (and functions) in scalable arithmetic functions;
2. Annotating original expressions and functions with value intervals based on a representing test data set (input and output values of the composed function);
3. Calculating the scaling factors based on the interval annotations and pre-defined function value range annotations, e.g., a pre-defined sigmoid function.
4. Calculating the approximation error for the test data (eventually modifying the functional structure or changing scaling factors to reduce the overall function error).

The transformation process is not addressed in this work. Values of function variables (input, output, latent) in a specific data context and application can lie in a small interval, e.g., [0.11,0.12]. Pure scaling, e.g., with M=16 Bits), would use k=250000, but the entire integer range would be only [27500,30000], effectively reducing the resolution to 12 Bits with significantly increased approximation error. To increase the usable range for integer approximations of real numbers, a bias offset can be introduced, approximating a real number by a <bias,scale,value> tuple. But this kind of arithmetic would require post-corrections and dedicated arithmetics, and scaling factors and bias must be specified for each particular value (in contrast, to pure scaling), increasing memory storage.

A data-driven predictive model function is composed of vector operations and transfer functions. The approximation error in such a composed and chained functional system is accumulative. Using linear transfer functions the error is linear accumulative and show no exploding gradients. But using non-linear functions, e.g., based on logarithmic functions, the approximation error is non-linear with exploding gradients and underflows, at consist of approximation based on SA and approximation of non-linear functions itself, as discussed in Sec. IV.F.

III Virtual Machine

Details of the REXA VM architecture, features, capabilities and the compiler can be found in the technical paper [9]. In the following section the ML-relevant features are summarized only. The REXA-VM may be implemented in compact embedded systems with a microcontroller and as little as 8 KB of data RAM and 16 kB of code ROM. In large-scale and heterogeneous networks, virtualization and Machine Learning (ML) are essential for unified sensor and data processing [10]. A scriptable Tiny ML interface and signal analysis numbers utilizing 16-bit scaled arithmetic are two important features. This VM supports 16 and 32 bit operations natively, preventing frequent arithmetic overflow and underflow problems. In contrast to common integer-

based ML models using 8 bit scaled arithmetic [2], this VM supports 16 and 32 bit operations.

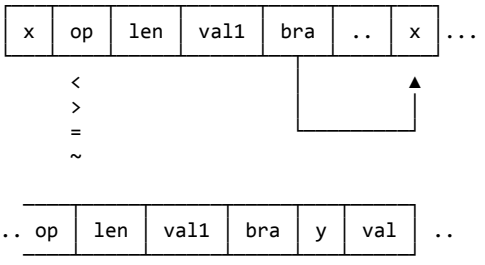
The REXA VM was designed especially for the deployment on low-resource microcontrollers with less than 64 kB RAM and low clock frequencies below 50 MHz. It utilizes a freely programmable ISA, but the ISA of the VM used in this work is closely related to the FORTH programming language [11]. The VM is a pure stack processor, i.e., most operations processing data via multiple stack memories with a zero-operand instruction format. The VM instruction loop processes Bytecode programs stored in a code segment (CS).

Each VM program consists of data and instructions stored in a code fragment in the CS. The main user program memory is the code segment of the VM (CS), which is organized in byte cells and has a static fixed size. An important feature of the CS is the direct embedding of program data besides code instructions. The Bytecode is compiled just-in-time by an integrated compiler. The VM and the compiler operate both incrementally, i.e., the processing time of each of them can be limited and scheduled, a primary feature required in single IO task programs with a main loop processing IO events and performing computations. Since the ISA of stack processors consists mostly of zero-operand instructions, it supports fine-grained compilation at the token level. The source text can be directly stored in the code segment (in-place) referenced by a code frame (or any other data buffer, alternatively). Most instruction words can be directly mapped to a consecutively numbered operation code.

IV ML MODELLS

A Decision Trees

Decision trees, as lightweight predictor models well suited for tiny embedded systems, can be efficiently stored in Linear Search Tables (LST), as introduced earlier for compiler parsing.



Def. 1. Format of a Linear Search Tree (LST) implementing a decision tree

Decision trees consist of nodes associated with input variables x_j or output variables y_k (and specific outcomes of a prediction). Directed edges connecting nodes are functional evaluations of a node variable.

There are three basic operations: Binary relation (</>), equality (=), and nearest value approximation (≈). The data format is shown in Def. 1. Each slide starts with the input variable to be evaluated (or target for output), the operation applied to choices, a field specifying the number of choices, and value-branch pairs. Decision tress can always approximated by integer arithmetic without error accumulation or exploding gradient (and underflow) issues. Therefore, the decision tree is here the gold standard for classifications problems and compared with ANN implementations.

B Artificial Neural Network (ANN)

An ANN consists of two parts:

1. The data, i.e., for parameter, input, and output variables;
2. The structure and functions processing the data.

For the sake of simplicity, fully connected networks are assumed, but any irregular network structure is a sub-set of a fully connected structure and can be used with the following operational architectures, too. In contrast to common ANN software frameworks, REXA VM provides only core vector operations, as discussed later on. The parameter data is embedded in a code frame by using the initialized *array* constructor. Both parameter and input/output data can be stored in the program code frame, shown in the next section.

ANN computations are decomposed in vector operations provided by the VM platform, discussed below. It can be shown that the complexity and memory requirement of this (textual data) approach is low even for complex network structures. Compiled code embedding data require typically less than 1 kBytes of RAM.

The principle structure of an ANN model and its forward computation using the vector operations discussed at next is shown below. There are initialized parameter arrays (weights, biases, and scaling factors) and latent variable arrays (neuron output).

```
array input N
array wghtsL1 { 1 2 3 .. }
array biasL1  { 1 2 .. }
array scaleL1 { 1 2 .. }
array outL1 N
..
: fwd
  .. vecmul
  .. vecadd
  .. vecmap
  ..
;
```

C Convolutional Neural Networks (CNN)

The structure of a CNN consists of different layers. A minimal basic layer architecture set consists of:

1. A convolutional layer applying a kernel filter mask to an input image (linear multiply-summation operation) producing a filtered output image;
2. A pooling layer extracting relevant features from images by applying special filters (e.g., a maximum value selection);
3. An ANN layer (commonly fully connected).

CNN computations are decomposed in vector operations provided by the VM platform, discussed below. The complexity and memory requirements is much higher than compared with ANN implementations. Especially the ANN layer is connected to all elements of the arrays of the pooling layer. Memory requirement is typically more than 4 kBytes, depending on the network structure, input dimension, and layer sizes. More details and evaluations can be found in the use-case sections.

The principle structure of a CNN model and its forward computation using the vector operations discussed at next is shown below (here the first convolution and the second pooling layer are merged to save storage space). There are initialized parameter arrays (kernel weights, biases, and scaling factors) and latent variable arrays (intermediate images, neuron outputs).

```
array input N
array kernL1p1 { 1 2 3 .. }
array kernL1p2 { 1 2 3 .. }
array kernL1p3 { 1 2 3 .. }
array cnvtmpL1 N
array poolL1p1 N
array poolL2p2 N
array poolL2p3 N
...
: fwd
  ( conv & pool )
  .. vecconv
  .. vecmap
  .. vecconv
  .. vecconv
  .. vecmap
  .. vecconv
  ..
;
```

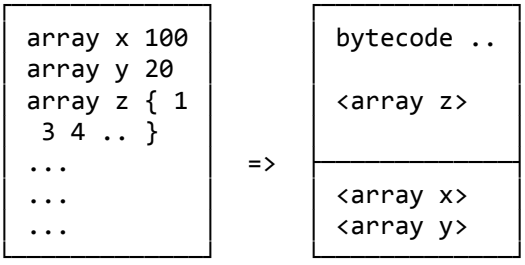
D ML Core Operations

ANN and CNN computations required efficient and generic vector operations crucial to implement ML on microcontrollers. The REXA VM provides a core set of vector operations that can be used for the computation of ANN and CNN models. Training using classical error backpropagation is currently not supported due to requirement of storing a suitable training and test data set.

All the basic operations you need to implement ANNs and perform forward activation computations are:

1. Element-wise vector operations (e.g., *vecmul*: *op1vec op2vec dstvec scalevec*);
2. Dot-product operation performing a sum of product data fusion (*vecprod*: *veca vecb scale* → 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*). This function also serves as a pooling operation;
5. A mapping operation applying a function elementwise (*vecmap*: *srcvec dstvec func scalvec*);
6. A reduction operation applying a function to all elements returning an aggregate (*vecred*: *vec vecoff veclen op*); Supported functions are *min*, *max*, *sum*, and *average*;
7. A vector reshape operation shrinking or expanding a vector (*vecshape*: *srcvec dstvec scale*);
8. A generic scaling operations (*vecscale*: *srcvec dstvec scalevec*).

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




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

Def. 2. Initialized arrays embedded in-place in code frames and non-initialized arrays stored at the end of the compiled code frame

E Vector Operations

The core set of vector operations provided by the REXA VM supporting integer arithmetic ANN computations is summarized in Tab. 1.

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.

Vector Operation

`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, positive values expand the source data values.

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

`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, 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.

`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.

TAB. 1. BASIC VECTOR ANN FUNCTIONS OPERATING ON EMBEDDED OR EXTERNAL ARRAY DATA (E.G., THE SAMPLE BUFFER)

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 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 [4,12]. Vector operations can access arrays stored in code frames or provided externally by the host application (e.g., a signal buffer).

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). The application of an activation function must be done separately using the *vecmap* operation, e.g., by applying a *sigmoid* function to all elements of a vector.

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

$$
\begin{aligned}
&\mathrm{vecmul}\left(\vec{a},\vec{b}\right)=\left(a_1\cdot b_1,a_2\cdot b_2,..,a_n\cdot b_n\right)^T\\
&\mathrm{dotprod}\left(\vec{a},\vec{b}\right)=\sum_{i=1}^{n}a_i\cdot b_i\\
&\mathrm{fold}\left(\vec{a},\hat{c}\right)=\left(\sum_{i=1}^{n}a_i\cdot c_{i,1},\sum_{i=1}^{n}a_i\cdot c_{i,2},..,\sum_{i=1}^{n}a_i\cdot c_{i,n}\right)^T\\
&\mathrm{conv}\left(\vec{a},\vec{c}\right)=\sum_{i=1,+s}^{a_h}\sum_{j=1,+s}^{a_w}\sum_{k=1}^{c_h}\sum_{l=1}^{c_w}a_{i+k,j+l}\cdot c_{k,l}\\
&\mathrm{map}\left(\vec{a},f\right)=\left(f\left(a_1\right),f\left(a_2\right),..,f\left(a_n\right)\right)^T\\
&n=\left|\vec{a}\right|=\left|\vec{b}\right|,\left|c\right|=n\cdot m
\end{aligned}
\tag{1}
$$

F Activation Functions

There are different transfer (activation) that are used in ANN and CNN modells, mosr prominent examples are:

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

The *linear* and *relu* functions can be directly implemented with integer arithmetic without loss of accuracy (except due to integer discretizing). The highly non-linear *sigmoid* and *tanh* functions require an appropriate approximation by using a hybrid approach of the usage of a (compacted) look-up table (LUT) and interpolation. 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 segmented linear and non-linear look-up tables. For example, the error of the discrete sigmoid function is always less than 1%, while only requiring 30 bytes of LUT space and less than 10 unit operations, as shown in Alg. 1. These software functions can be immediately implemented in hardware, too. The LUTs are computed with Alg. 2.

```
static ub1 sglut13[] = { <24 values> };
static ub1 sglut310[] = { <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>=10000) return mirror?0:1000;
  if (x<=1000) {
    y = 500+(((x*231)/1000));
    return mirror?1000-y:y;
  } else if (x<3000) {
    ub2 i10 = ((fplog10((x/5)|0)/2))-65;
    y = ((sb2)sglut13[i10])+731;
    return mirror?1000-y:y;
  } else {
    ub2 i10 = ((fplog10((x/10)|0)/10))-14;
    y = ((sb2)sglut310[i10])+952;
    return mirror?1000-y:y;
  }
  return 0;
}
static ub1 log10lut[] = { <100 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];
}
```

Alg. 1. Range-segmented and LUT-based implementation of the sigmoid function with less than 1% approximation error (using approximated LUT-based log10 function)

The LUT tables can be computed as follows:

$$\text{log10lut} = \left\{ int\left(\log_{10}\left(\frac{i}{10}\right)100 \right) : i \in \mathbb{I}, 0 \le i \le 99 \right\} \quad (2)$$

The *fpsigmoid* function LUTs are computed iteratively using the *fplog10* function, described by the following pseudo code algorithm Alg. 2:

```
sglut13 := []
for x=1 to 2.95 step 0.05 do
  i10 := int(fplog10(int(x*1000/5))/2)-65
  if sglut13[i10] = undefined then
    sglut13[i10] := int(sigmoid(x)*1000)-731
  endif
done
sglut310 := []
for x=3 to 9.9 step 0.1 do
  i10 := int(fplog10(int(x*1000/10))/10)-14
  if sglut310[i10] = undefined then
    sglut310[i10] := int(sigmoid(x)*1000)-952
  endif
done
```

Alg. 2. Computation of the LUTs for the scaled integer sigmoid function

The accuracy (relative error) of the sigmoid approximation is plotted in Fig. 3 with an input and output scaling factor of 10000 (i.e., 1:10000). For $x > -3$ the error is below 5% and decreases to 1% in average. Only for $x < -3$ the relative error increases significantly due to the integer resolution.

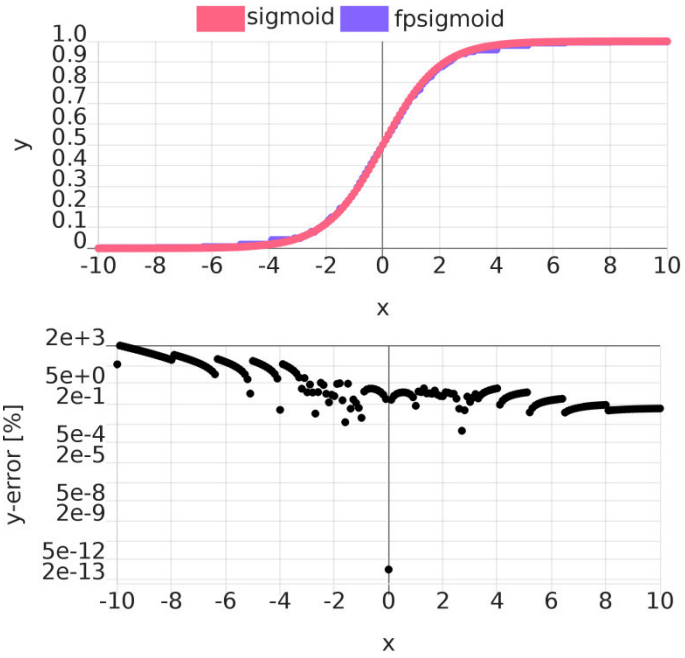


Fig. 3. Relative discretization error of scaled integer LUT-interpolated approximation of the *sigmoid* function

V Evaluation

Computation time results for ANN and CNN models are shown in Fig. 4 and 5. The code size required to store static and dynamic model parameters are shown in Fig. 6 and 6. Two different host platforms were tested: A generic i5 x86 clocked @2900 MHz (during test) and a STM32F103C8 microcontroller clocked @72 MHz with 20 kB RAM. All tests are processed by the operational same REXA VM. The computation time was normalized to the CPU clock frequency to enable comparison between different platforms. The REXA VM provided a code segment with 6k words capacity and a data stack with 256 words. The VM was compiled with GNU CC (gcc version 7), and the ARM-STM32 version was compiled with the Arduino software toolkit. With the configuration described above, 3 kB RAM remains for the VM program stack, which is sufficient. The REXA VM allocates memory only statically on the heap, there is no dynamic memory allocation during run-time.

The computational times were plotted against the number of neurons (ANN) and cells (CNN). The number of cells of a CNN is the sum of the static parameters and dynamic variables.

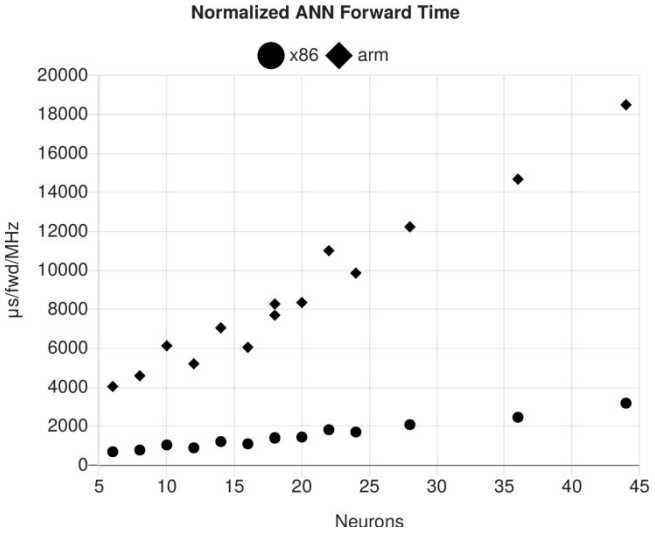


Fig. 4. Normalized computation times for ANNs of different size (with two, three, and four layers) and two different host platforms (Generic i5 x86 @2900 MHz and STM32F103C8 @72MHz) as a function of neurons. The computation time is approximately linear with the number of neurons (independent of network layer structure)

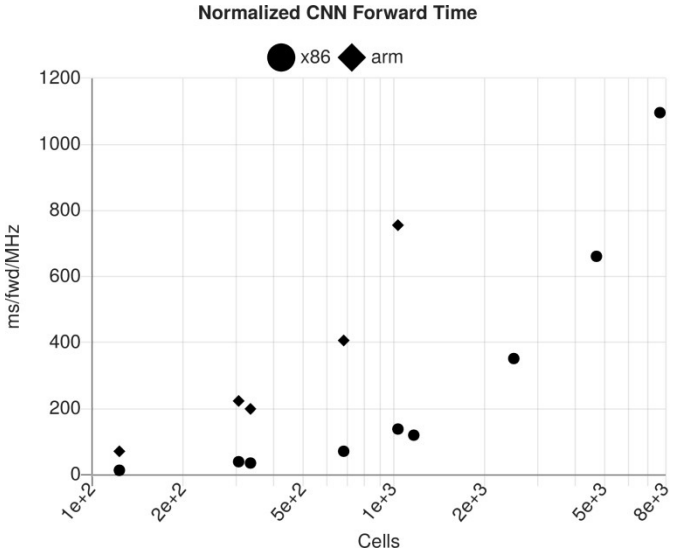


Fig. 5. Normalized computation times for CNNs of different size (with one and two convolution-pooling layer pairs) and two different host platforms (Generic i5 x86 @2900 MHz and STM32F103C8 @72MHz) as a function of cells (product of parameters and variables). The computation time grows about O(n-$log(n)$) with the number of cells n.

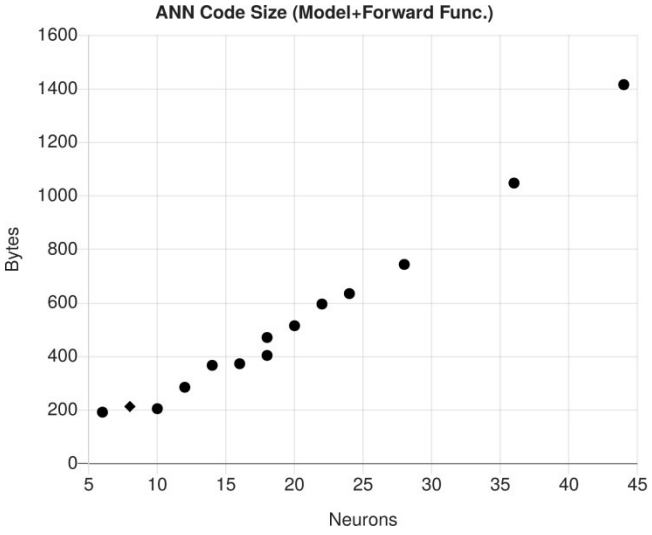


Fig. 6. Code size of ANN as a function of the number of neurons.

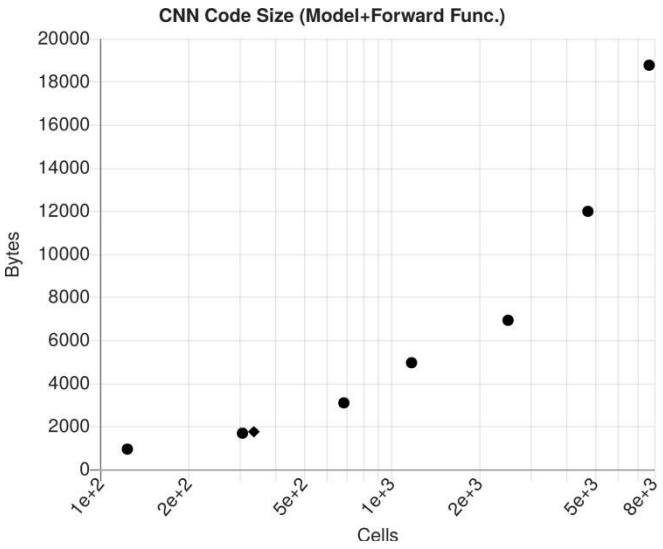


Fig. 7. Code size of CNN as a function of the number of cells.

The performance test shows the suitability of a low-resource microcontroller to store and compute small and medium sized ANN and even smaller CNN models. The forward (inference) computation time is always below one Second, typically about 10-100 ms with 16 MHz clock frequency. The required code space (including model data and code) is below 10 kBytes, typically about 1-2 kBytes. The ARM Cortex M processor under performs by a factor of 5 compared with a x86 processor, which is well known.

VI Use Cases

A Damage detection with an ANN

In this use-case, aggregated feature variables derived from time-dependent Ultrasonic signals (Guided Ultrasonic

Waves, GUW) from multi-path measurements were used to predict a damage in a composite materials and to estimate its location. Details can be found in [13]. The processing architecture is shown in Fig. 8.

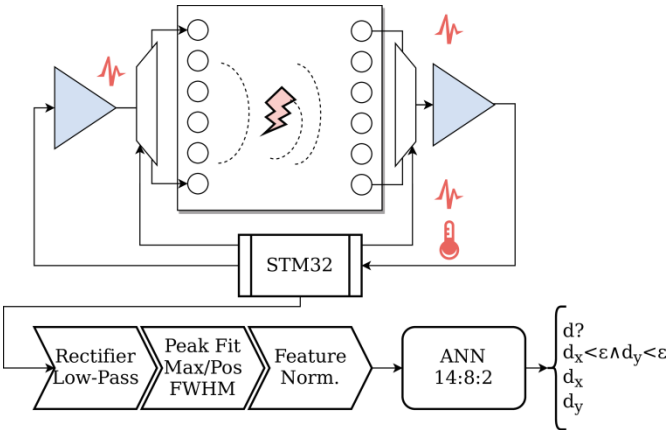


Fig. 8. Multi-path GUW measurement and data processing for damage detection (classification and location regression)

The feature variables were computed from the signal hull, mainly by analyzing the first main maximum (position and width). The hull signal was computed using (1) the analytical signal via the Hilbert transform (using FFT) and (2) by applying a rectifier and low-pass filter. Only the second method can be implemented on the STM32 microcontroller. Assuming six measuring paths and the two most significant feature variables normalized peak position and peak height, additionally using a measure temperature and the base frequency of the pitch signal, the feature vector consists of 14 variables in total. This scaled feature vector is the input for a simple ANN (three layers, one hidden, typical layer structure [14,8,2], sigmoid activation functions). The output of the ANN provided an estimation of the x- and y-coordinates of the damage location (or close to 0 if there is no damage detected). This is a hybrid classification and regression model. If only classification is required, one output neuron is sufficient.

The ANN with a [14,8,1] layer structure providing a binary damage classification was trained and transformed to the proposed integer numerics requiring about 1k Bytes code size, shown in Ex. 1.

```
( Layers: 14,8,2 #neurons:24 )
array input 14
( Layer I )
array wghtsI { 329 -499 ... 10 400 }
array biasI { -764 389 ... -907 -405 }
array scaleI { -3 9 ... 5 9 }
array actI 14
( Layer H1 )
array wghtsH1 { 622 -790 ... 708 248 }
array scaleH1 { 0 5 ... -4 7 }
array actH1 8
( Layer O )
array wghtsO { 869 939 ... 785 910 }
array biasO { 252 -565 }
array scaleO { 4 5 }
array output 2
( Input data is stored in input )
( Output data is stored in output )
: forward
  ( Layer I )
  input wghtsI actI scaleI vecmul
  actI biasI actI 0 vecadd
  actI actI $ sigmoid 0 vecmap
  ( Layer H1 )
  actI wghtsH1 actH1 scaleH1 vecfold
  actH1 biasH1 actH1 0 vecadd
  actH1 actH1 $ sigmoid 0 vecmap
  ( Layer O )
  actH1 wghtsO output scaleO vecfold
  output biasO output 0 vecadd
  output output $ sigmoid 0 vecmap
;
```

Ex. 1. REXA VM program for an ANN classifier for damage prediction from 14 aggregate feature variables and two output variables (parameter values are only for illustration)

The ANN requires only 620 Bytes in the CS memory of the REXA VM. The computation time (prediction) is about 1 ms/MHz (Intel x86 i5, i.e. 0.5µs @2900 MHz) and about 5 ms/MHz (STM32 ARM Cortex).

B Damage detection with a CNN

Similar to the previous use-case, single-path Ultrasonic time-dependent measuring signals are used to predict a damage in a composite material. In contrast to the previous example, no strong aggregate feature variables could be identified. Instead, a discrete wavelet transform using high- and low-pass filters are used to decompose the sensor signal into wavelet coefficients (first 5 levels were chosen). The output of the filters (detail and approximation) are decimated by a factor of two, retaining only the even samples, since each filter output contains half of the frequency content, but an equal amount of samples as the input signal. With increasing level the number of data elements decreases by a factor 2. To provide the output of multiple levels in matrix form, the higher levels are expanded. Here, we shrink the lower levels to the number of elements of the highest level (5). The original signal window contained about 2000 samples, finally providing only 50 data points for the fifth DWT decomposition layer. All DWT vectors are combined into a 50 × 5 elements matrix, treated as a two-dimensional spectogram image. The processing architecture is shown in Fig. 9.

A simple CNN was used to classify signals and predict damages. The CNN consists of one convolution layer with three filters (3 × 3 pixel), striding and padding set to two, output applied to a *relu* function, followed by one max-pooling layer (striding=2, padding=0). Finally, a softmax/fully connected two-neuron layer performs the classification (sigmoid activation function). The REXA VM program is shown in Ex. 2.

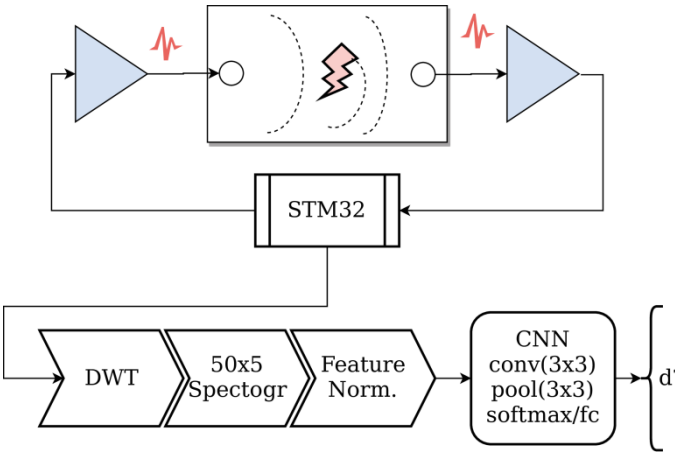


Fig. 9. Single-path GUW measurement and data processing using a CNN for damage detection

```
( Layers: conv,pool,fc )
array input 250
( Layer 1 conv )
array cK0L1 { -612 -692 ... 962 -467 }
array cK1L1 { -214 832 ... -644 -455 }
array cK2L1 { 764 -275 .. 978 600 }
array cSL1 { 817 390 572 }
array cOL1 104
( Layer 2 pool )
array pO0L2 12
array pO1L2 12
array pO2L2 12
( Layer 3 fc )
array fW0L3P0 { -468 -905 ... -632 518 }
array fW0L3P1 { -147 -932 ... -275 872 }
array fW0L3P2 { -327 -798 ... -61 -621 }
array fW1L3P0 { -126 894 ... -818 -870 }
array fW1L3P1 { 488 -408 ... 963 -887 }
array fW1L3P2 { -519 963 .. 895 -170 }
array fAL3 12
array fBL3 { -746 -776 }
array fSL3 { 1 3 }
array fOL3 2
array output 2
( Input data is stored in input )
( Output data is stored in output )
: forward
  ( Layer 1 conv )
  ( merged with Layer 2 pool )
  input cK0L1 cOL1 cSL1 0 cell+ @ 50 3 2 2 vec-
conv
  cOL1 cOL1 $ relu 0 vecmap
  cOL1 256 3 + pO0L2 0 26 -3 2 0 vecconv
  input cK1L1 cOL1 cSL1 1 cell+ @ 50 3 2 2 vec-
conv
  cOL1 cOL1 $ relu 0 vecmap
  cOL1 256 3 + pO1L2 0 26 -3 2 0 vecconv
  input cK2L1 cOL1 cSL1 2 cell+ @ 50 3 2 2 vec-
conv
  cOL1 cOL1 $ relu 0 vecmap
  cOL1 256 3 + pO2L2 0 26 -3 2 0 vecconv
  ( Layer 3 fc )
  pO0L2 fW0L3P0 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO1L2 fW0L3P1 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO2L2 fW0L3P2 fAL3 0 vecmul
  fAL3 0 12 8  vecreduce
  2+ 2+ fSL3 0 cell+ @ 2ext 2/ 2red sigmoid
  fOL3 0 cell+ !
  pO0L2 fW1L3P0 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO1L2 fW1L3P1 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO2L2 fW1L3P2 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  2+ 2+ fSL3 1 cell+ @ 2ext 2/ 2red sigmoid
  fOL3 1 cell+ !
;
```

Ex. 2. REXA VM program for a CNN classifier for damage prediction from 50 × 5 feature variables (DWT spectogram) and two output variables (parameter values are only for illustration)

The CNN requires only 1500 Bytes in the CS memory of the REXA VM, fitting into a STM32F103C8 (20 kBytes RAM). The computation time (prediction) is about 30 ms/MHz (Intel x86 i5, i.e. 10µs @2900 MHz) and about 150 ms/MHz (STM32 ARM Cortex).

Due to the high integration level and the minimization of components the measuring data is characterized by noise (analog and digital sources), drift, and a superposition by environmental signals (main AC line, e.g.). Despite the data quality limitations, a damage prediction accuracy about 95% can be achieved by the CNN. Considering the low complexity of the CNN, the results showing the suitability of even simple data-driven classifier models processed directly by a material-integrated sensor node with a low-resource microcontroller.

VII Conclusion

The stack-based REXA VM was introduced targeting CPUs with integer arithmetic only and providing virtualization and a unique set of vector operations used to compute Artificial and Convolutional Neural Networks under high resource constrains. It could be shown that even with less than 20 kBytes of RAM memory (simple) CNNs can be computed. The VM has a built just-in-time text-to-Bytecode compiler. A ML model is provided on programming level with a mix of data and computational statements. The VM uses a shared code segment for program text and compiled Bytecode with embedded data without necessity to have a dynamic memory management (heap). The computational times for medium sized ANNs and small CNNs are about 1-300 ms/MHz, reasonable for sef-powered sensor networks. The source code of the REXA VM can be downloaded from github [14].

References

[1] Guo, S., Zhou, Q. , Machine Learning on Commodity Tiny Devices, Taylor & Francis, 2023

[2] Ray, P. P., A review on TinyML: State-of-the-art and prospects, Journal of King Saud University-Computer and Information Sciences, 2021, pp.1595-1623, https://doi.org/10.1016/j.jksuci.2021.11.019
[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 Thing, arXiv:1911.03314v3, 2022
[4] Banner, R. , Hubara, I., Hoffer, E., Soudry, D., Scalable Methods for 8-bit Training of Neural Networks, arXiv:1805.11046, 2018
[5] Alajlan, N. N., Ibrahim, D. M., TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications, micromechanics, vol. 13, no. 851, 2022, https://doi.org/10.3390/mi13060851
[6] Jain, V., Giraldo, S., Roose, J. D., Linyan, Mei, B. B., Verhelst, M. , TinyVers: A Tiny Versatile System-on-chip with State-Retentive eMRAM for ML Inference at the Extreme Edge, arXiv:2301.03537, 2023,
[7] Heiser, G., The role of virtualization in embedded systems, In Proceedings of the 1st workshop on Isolation and integration in embedded systems, 11-16 April, 2008, https://doi.org/10.48550/arXiv.2301.03537
[8] Zhang, L., Implementation of fixed-point neuron models with threshold, ramp and sigmoid activation functions, In IOP Conference Series: Materials Science and Engineering (Vol. 224, No. 1, p. 012054). IOP Publishing, 2017
[9] Bosse, S., Bornemann, S., Lüssum, 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:2302.09002 [cs.OS], 2023, https://doi.org/10.48550/arXiv.2302.09002
[10] Bauer, M., IoT Virtualization with ML-based Information Extraction, in IEEE 7th World Forum on Internet of Things 2021, https://doi.org/10.1109/WF-IoT51360.2021.9595119
[11] Hayes, J. R. Fraeman, M. E., Williams, R. L. Zaremba, T., An architecture for the direct execution of the Forth programming language, ACM SIGARCH Computer Architecture News, 15(5), 1987, pp. 42-49. https://doi.org/10.1145/36177.36182
[12] Ghaffari, A.,. Tahaei, M. S., Tayaranian, M., Asgharian, Vahid, M., Nia, P., Is Integer Arithmetic Enough for Deep Learning Training?, Advances in Neural Information Processing Systems 35. 2022: 27402-27413.
[13] Bosse, S., Polle, C., Fast Temperature-Compensated Method for Damage Detection and Structural Health Monitoring with Guided Ultrasonic Waves and Embedded Systems, Eng. Proc. 2021, 10(1), 78; https://doi.org/10.3390/ecsa-8-11283
[14] https://github.com/bsLab/rexavm, REXA VM repository, on-line, accessed 31.7.2023