ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems

Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, Claire Pagetti

▶ To cite this version:

Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, Claire Pagetti. ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems. 24th Euromicro Conference on Real-Time Systems (ECRTS 2022), Jun 2022, Modène, Italy. 10.4230/LIPIcs.ECRTS.2022.3 . hal-03707284

HAL Id: hal-03707284
https://ut3-toulouseinp.hal.science/hal-03707284v1

Submitted on 28 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems

Iryna De Albuquerque Silva

ONERA, Toulouse, France

Thomas Carle

IRIT – Univ Toulouse 3 – CNRS, France

Adrien Gauffriau

Airbus, Toulouse, France

Claire Pagetti

ONERA, Toulouse, France

Abstract

Machine learning applications have been gaining considerable attention in the field of safety-critical systems. Nonetheless, there is up to now no accepted development process that reaches classical safety confidence levels. This is the reason why we have developed a generic programming framework called ACETONE that is compliant with safety objectives (including traceability and WCET computation) for machine learning. More practically, the framework generates C code from a detailed description of off-line trained feed-forward deep neural networks that preserves the semantics of the original trained model and for which the WCET can be assessed with OTAWA. We have compared our results with Keras2c and uTVM with static runtime on a realistic set of benchmarks.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Software notations and tools

Keywords and phrases Real-time safety-critical systems, Worst Case Execution Time analysis, Artificial Neural Networks implementation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.3

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.1.6`

Funding This project received funding from the French "Investing for the Future – PIA3" program within the Artificial and Natural Intelligence Toulouse Institute (ANITI).

1 Introduction

The use of artificial intelligence approaches is already of vital importance in many research areas. In particular, when embedded in aircraft systems, intelligent algorithms could help in tasks such as navigation, predictive maintenance and air traffic control, improving safety and saving environmental resources. Nonetheless, not much progress has been made in embedding machine learning solutions in safety-critical systems as most of those applications do not reach classical safety confidence levels and are not implemented with accepted development process [2, 5]. The scope of this work is the safe real-time implementation of neural networks on embedded platforms.

Context. We focus on safety-critical domains and in particular on aeronautics that is subject to *certification*. The question of how to safely and reliably implement a neural network on an adequate hardware is of vital importance. Indeed, certification requirements, in particular those of the DO 178-C [14][1], impose strong guarantees on the quality of the code and expect the designer to:

[1] Classical guidance for the implementation process of the software items

© Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, and Claire Pagetti;
licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio; Article No. 3; pp. 3:1–3:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- ensure *traceability* between the requirements and the (source) code;
- compute the *WCET (Worst Case Execution Time)* [39] for each piece of code;
- run *intensive testing* to verify the compliance of the implementation to the requirements. This includes unit tests to verify both that the executable provides the intended function and there is no hidden unintended function (by activating all the branches of the code).

The purpose is thus to provide a programming framework compliant with these objectives for machine learning. This work is restricted to *off-line trained feed-forward deep neural networks* (referred to simply as neural networks or DNN subsequently). The off-line design of such neural networks is done by defining the structure (that is the number and the type of layers), choosing the training data set and using a learning framework such as Tensorflow [1] or PyTorch [28]. The result of the design is called the *inference model*: it comprises a neural network with its parameters (e.g. weights, biases, activation functions or kernels). The implementation – the part we focus on – consists in coding the inference model in an adequate programming language and porting the code on the target hardware.

Contributions. The first challenge brought by the implementation is the *semantic preservation*: the reproducibility of the behavior observed when executing the *inference model* within the training tool on the target hardware. Thus our first contribution (see section 2) is to formally define the semantics of DNN (by extending and formalizing existing works of the literature) and explaining the challenges brought by the current frameworks. Indeed, the training tools such as Tensorflow/Keras or PyTorch do not encode the basic operations, such as convolutions (and thus matrices operations) in the same manner.

The second challenge is *predictability*: the capacity to assess the worst-case execution time (WCET) of a sequential code. In the ML literature, most of the implementations are done on GPUs or TPUs with a runtime engine such as Tensorflow that interprets the *computation graph*, i.e., a graph describing the mathematical structure of the neural network. Such an interpreter uses dynamic memory and scheduling allocation and as we focus on safety-critical domain – and more specifically avionics –, such an approach is not practical for two reasons. First, the hardware targets that are compatible with certification are not those mentioned earlier. We thus focus on general purpose multi-core commercial off-the-shelf (COTS) hardware such as the T1042 from NXP, the Coolidge from Kalray [18] or the Keystone from Texas Instrument [37] (used in the experiments). Second, the programs, including the application, the RTOS and the runtime, must be *predictable*. There are some initiatives to make such runtime predictable such as eIQ and KaNN. However, there is still a large amount of work and proof to show the capability to compute a WCET for these tools. This is the reason why we target a more classical static approach which consists in generating an equivalent C code to execute the model (no interpretation) such as proposed in [8]. Our second contribution is the development of ACETONE (Avionics C code generator for Neural NEtworks), a framework that generates a real-time C code *semantically equivalent* to the inference model (see section 3) and that fits the aeronautic requirements. We made a particular effort on the software architecture to make the framework:

- *modular*: it is very simple to add new DNN structures, new types of layers or new refinement of the existing ones.
- *very easy to use*: a person non familiar with our framework can very quickly generate their C code and port them on their target;
- *extremely traceable*: looking at the generated C code, it is humanly possible to trace back to the original exported DNN model, which is an expected property from the DO-178C;

- *predictable*: we used a static WCET analyzer of the literature, OTAWA [4] developed at the University of Toulouse, to assess the WCET of the code. This means that the C code is expected to run sequentially on a single core (no parallelization targeted in this paper), all the memory allocations are static and the schedule (here the sequence of executions) is also static. The compilation of the C code to a binary must also use the flag *-O0* (no compiler optimization). These restrictions are important to keep in mind to understand the philosophy of the code generation.

The last contribution is a thorough evaluation of our framework together with a comparison with state of the art C code generator frameworks, namely Keras2c [10] and uTVM with static C runtime [35]. Section 4 details the methodology: we have selected a set of representative benchmarks and identified a set of criteria to assess the quality of a code (in accordance with the DO-178C objectives listed above). Section 5 gives the results of the experiments. We were able to assess most of our criteria for all the benchmarks and frameworks. In particular, we have ported the binary on an Arm Cortex-A15 of the Keystone [37] to compare the measured and worst-case execution times. Overall, in terms of performance, we are comparable to and even slightly better than the other frameworks. This stems, for uTVM, from the restrictions needed for predictability and the compilation with *-O0*. In that sense, our implementations are optimal with respect to our criteria.

2 Reminder on Deep Neural Networks

We focus on the inference of off-line trained feed-forward Deep Neural Networks (DNN). More precisely, we consider convolutional neural networks (CNN) and multi-perceptron (or fully-connected) neural networks.

2.1 Functions performed by DNN

There are multiple ways to define DNNs: directed graphs, computational graphs or simply the mathematical functions transforming the input into the output. The latter is the way we propose to explain the computations needed to be done by the C code. The input of those functions can be seen as a multi-dimensional vector also called *tensor*. Subsequently, we will only consider 1D-, 2D- and 3D-tensors but to save space, we only provide definitions for 3D. We only consider inference with one input (no batch).

▶ **Definition 1** (Tensor). *A 3D-tensor T is represented by its* size (n_h, n_w, n_c) *where n_h is the height, n_w the width and n_c the number of channels (or feature maps). We denote by T_{x_1,x_2,x_3} the value of T for the indices x_1, x_2, x_3. We denote by $T[s_{11} : s_{21}, ..., s_{1k} : s_{2k}]$ the* slice *of T of all the values $T_{s_{11}+x_1,...,s_{1k}+x_k}$ with $i \in [1, k]$ and $x_i \in [1, s_{2i} - s_{1i}]$.*

▶ **Definition 2** (Feed-forward Deep Neural Network). *A* feed-forward neural network $N =< l_1, \dots, l_n >$ *is a succession of* layers l_i *taking as input the output of the previous layer l_{i-1}. The first layer takes the input tensor. A layer can be of* $\text{type}(l) \in \{$act, bias, padd, conv, pool, batch norm, flat, dense$\}$ *where* act *is an activation,* padd *is a padding,* bias *is a bias adding,* conv *is a convolution,* pool *is a pooling,* batch norm *is normalization,* flat *is flattening and* dense *is a perceptron. A layer comes with a set of parameters (e.g. weights or stride).*

▶ **Definition 3** (Function associated to a DNN). *The function f_N computed by a DNN $N =< l_1, \dots, l_n >$ is the composition of the functions computed by each layer $f_N = f_{l_n} \circ \dots \circ f_{l_1}$.*

The semantics of each function is given in [38]. We give the definition, with mathematical equations, of the main layers used in the use cases depicted in section 4.1.

▶ **Definition 4** (Activation function). *Let $f : \mathbb{R} \longrightarrow \mathbb{R}$ be a function (e.g.* ReLu, sigmoid*). The activation function $\mathcal{A}_f$ applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{A}_f(I)$ of size (n_h, n_w, n_c) defined by $O_{x,y,z} = f(I_{x,y,z})$ for all $x \leq n_h$, $y \leq n_w$ and $z \leq n_c$. We could also write $O_{x,y,z} = map(I, f)$.*

▶ **Definition 5** (Bias layer associated function). *Let B be a 3D-tensor of size (n_h, n_w, n_c). The bias function $\mathcal{B}_B$ applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{B}_B(I)$ of size (n_h, n_w, n_c) defined by $O_{x,y,z} = I_{x,y,z} + B_{x,y,z}$ for all $x \leq n_h$, $y \leq n_w$ and $z \leq n_c$.*

▶ **Definition 6** (Padding layer associated function). *Let $p = (p_t, p_b, p_l, p_r)$ be a 4-tuple of integers representing the padding to be applied on each border of a 3D-tensor. The padding function $\mathcal{P}_p$ applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}_p(I)$ of size (o_h, o_w, o_c) with $o_h = n_h + p_t + p_b$, $o_w = n_w + p_l + p_r$ and $o_c = n_c$ such that*

$$O_{x,y,z} = \begin{cases} 0 & \text{if } x \leq p_t \text{ or } x > n_h + p_t \text{ or } y \leq p_l \text{ or } y > n_w + p_l \\ I_{x-p_t, y-p_l, z} & \textit{otherwise} \end{cases}$$

▶ **Definition 7** (2D-convolution associated function). *Let K be a vector of 3D-tensors $[K^1, K^2, \ldots, K^{\text{nb_kernel}}]$ representing the* kernels *of the convolution. Each kernel K^i is of size (f_h, f_w, f_c). Let $s = (s_h, s_w)$ be the* stride *parameter with s_h and s_w two integers. The 2D-convolution*[2] *$\mathcal{C}_{K,s}$ applied to a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{C}_{K,s}(I)$ of size (o_h, o_w, o_c) with $o_h = \left\lfloor \frac{n_h - f_h}{s_h} + 1 \right\rfloor$, $o_w = \left\lfloor \frac{n_w - f_w}{s_w} + 1 \right\rfloor$ and $o_c = \text{nb_kernel}$. We have $O_{x,y,z} = \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} \sum_{m=1}^{f_c} K^z_{i,j,m} \cdot I_{s_h \cdot (x-1)+i, s_w \cdot (y-1)+j, m}$ for all $x \leq o_h$, $y \leq o_w$ and $z \leq o_c$. Note that also we must have $f_c = n_c$ thus, convolutions are often applied on 3D-tensors on which* padding *has been applied first to fit the sizes. See definition 14.*

▶ **Definition 8** (Pooling layer associated function). *Let $s = (s_h, s_w)$ be the* stride *parameters, let $k = (k_h, k_w)$ be the height and width of the* window *and let $f : \mathbb{R}^{k_h \cdot k_w} \longrightarrow \mathbb{R}$ be a function (e.g.* max *or* average*). The pooling applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}\text{ool}_{k,s,f}(I)$ of size (o_h, o_w, o_c) with $o_h = \left\lfloor \frac{n_h - k_h}{s_h} + 1 \right\rfloor$, $o_w = \left\lfloor \frac{n_w - k_w}{s_w} + 1 \right\rfloor$ and $o_c = n_c$ with $O_{x,y,z} = f(I[s_h \cdot (x-1)+1 : s_h \cdot (x-1)+k_h+1][s_w \cdot (y-1)+1 : s_w \cdot (y-1)+k_w+1][z])$.*

▶ **Definition 9** (Batch norm layer associated function). *Let γ be 1D-tensor of size n_c be the* scale*, let β be 1D-tensor of size n_c be the* offset*, let μ be 1D-tensor of size n_c be the* mean *(on the batch fixed during the training), let V be 1D-tensor of size n_c be the* variance *(on the batch fixed during the training), let ϵ be a float used to ensure no division per* 0*. The batch norm applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{BN}_{\gamma,\beta,\mu,\sigma,\epsilon}(I)$ of size (n_h, n_w, n_c) with $O_{x,y,z} = \frac{\gamma_z}{\sqrt{V_z + \epsilon}} \cdot I_{x,y,z} + \left(\beta_z - \frac{\mu_z}{\sqrt{V_z + \epsilon}}\right)$. We often denote by $\alpha_z = \frac{\gamma_z}{\sqrt{V_z + \epsilon}}$ and $\mathcal{B}_z = \left(\beta_z - \frac{\mu_z \cdot \gamma_z}{\sqrt{V_z + \epsilon}}\right)$, so that $O_{x,y,z} = \alpha_z \cdot I_{x,y,z} + \mathcal{B}_z$.*

▶ **Definition 10** (Flattening layer). *The flattening layer applied to a 3D-tensor I of size (n_h, n_w, n_c) outputs the 1D-tensor $O = \mathcal{F}\text{lat}(I)$ of size $n_o = n_h \times n_w \times n_c$ such that $O_x = I_{x \bmod n_w, \left\lfloor \frac{x \bmod (n_h \cdot n_w)}{nw} \right\rfloor, \left\lfloor \frac{x}{n_h \cdot n_w} \right\rfloor}$.*

▶ **Definition 11** (Dense layer). *Let W be a 2D-tensor of size (n_o, n_i) (for the* weights*) and B be a 1D-tensor of size n_o (for the* biases*). The dense layer applied to a 1D-tensor I of size n_i outputs the 1D-tensor of size n_o $O = \mathcal{D}\text{ense}(I) = W \cdot I + B$, i.e. $O_x = \sum_{k=1}^{n_i} W_{x,k} \cdot I_k + B_x$.*

[2] There may be an additional parameter, that is the *dilatation* supported by the code generation and not detailed here.

Then, we can define easily the function associated to a DNN from those basic functions.

▶ **Example 12** (Multi-perceptron / fully-connected neural network)**.** A *fully-connected neural network* is a succession of *dense and activation layers*. The function associated to the DNN of figure 1 is $N = f_{l_3} \circ f_{l_2} \circ f_{l_1} = \mathcal{A}_3(W_3 \cdot (\mathcal{A}_2(W_2 \cdot (\mathcal{A}_1(W_1 \cdot I + B_1)) + B_2)) + B_3)$. Its structure corresponds to 2 hidden layers with 3 neurons each, 2 inputs and 1 output. Short notation: $(2, 3, 3)$.

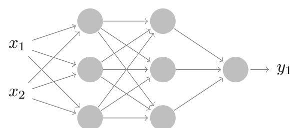

Figure 1 Fully-connected NN.

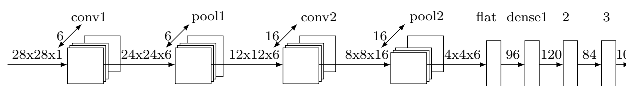

Figure 2 LeNet-5 CNN.

▶ **Example 13** (LeNet-5)**.** The LeNet-5 [23] model is the basic CNN developed for handwritten digits images recognition. We used the pre-trained LeNet-5 from Keras which is shown in figure 2. Such a graphical representation is classical to highlight the layers and the number of feature maps.

The size of the input / output tensors are shown on the figure. The first 2D-convolution *conv1* takes inputs of size $28 \times 28 \times 1$, is composed of 6 kernels K^i of size $5 \times 5 \times 1$ and of a stride $s = (1, 1)$. The activation function *tanh* is applied to the outputs. The first pooling layer *pool1* is an average pooling with stride $s = (2, 2)$ and window $k = (2, 2)$. The second 2D-convolution *conv2* is composed of 16 kernels K^i of size $5 \times 5 \times 6$ and of a stride $s = (1, 1)$. The activation function *tanh* is applied to the outputs. The second pooling layer *pool2* is an average pooling with stride $s = (2, 2)$ and window $k = (2, 2)$. The 3D-tensor of size $6 \times 6 \times 4$ is flattened in a 1D-tensor of size 96. There are three dense layers with respectively $(n_i, n_o) = (96, 120)$, $(n_i, n_o) = (120, 84)$ and $(n_i, n_o) = (84, 10)$. The two first dense layers apply the activation function *tanh* and the last one a *softmax*. Thus the function associated to this LeNet-5 is: $N = \mathcal{A}_{softmax} \circ f_{dense3} \circ \mathcal{A}_{tanh} \circ f_{dense2} \circ \mathcal{A}_{tanh} \circ f_{dense1} \circ f_{flat} \circ f_{pool2} \circ \mathcal{A}_{tanh} \circ f_{conv2} \circ f_{pool1} \circ \mathcal{A}_{tanh} \circ f_{conv1}$.

2.2 Semantics-preserving model transformation

At this stage, it is acceptable to transform the DNN model as long as the semantics is preserved. This can be interesting when it yields an improvement of the implementation. We list here some transformations worth to be made before coding.

▶ **Definition 14** (Extended 2D-convolution layers)**.** *In the literature, convolutions usually integrate other parameters than those listed in definition 7. Indeed, a convolution is often defined together with the padding, the activation function and even in some cases with a bias. We thus denote by* $\mathcal{C}_{p,K,s,B,f} = \mathcal{A}_f \circ \mathcal{B}_B \circ \mathcal{C}_{K,s} \circ \mathcal{P}_p$ *(all combinations by removing a function work). Note that this is common to consider bias B in convolution where* $B_{x_1,y_1,z} = B_{x_2,y_2,z}$ *with* $x_1 \neq x_2$ *and* $y_1 \neq y_2$*.*

▶ **Property 1** (Well-balanced 2D-convolution layers)**.** *It is usual to have the output height and width equal to the input height and width, i.e.* $o_h = n_h$ *and* $o_w = n_w$*. In that case, we must have* $n_h = \lfloor \frac{n_h + p_t + p_b - f_h}{s_h} + 1 \rfloor$ *and* $n_w = \lfloor \frac{n_w + p_l + p_r - f_w}{s_w} + 1 \rfloor$*. The padding should also satisfy* $p_t + p_b - f_h = -1$ *and* $p_l + p_r - f_w = -1$*.*

▶ **Property 2** (Portability issue between training frameworks). *We remark that for a given kernel size, several solutions may exist to the equations of property 1. For instance, with a kernel size of* $(5, 5)$ *and stride of 1, 4 different paddings for each dimension satisfy the equations. Thus classical frameworks like Tensorflow or PyTorch have different strategies (thus imply different semantics) when implementing a convolution that preserves the input size for height and width.*

▶ **Property 3** (Max pooling and ReLu activation layers). *Applying a ReLu activation layer before a max pooling layer is semantically equivalent to applying the ReLu activation layer after the max pooling layer. However, the number of operations is reduced when applying the ReLu activation after if the stride* $s = (s_h, s_w)$ *of the pooling satisfies* $s_h > 1$ *or* $s_w > 1$*.*

Proof. Let us assume that the input tensor I is of size (n_h, n_w, n_c) and that we do the ReLu before the pooling. Then we will do $O_{x,y,z} = max(ReLu(I[s_h.(x-1)+1 : s_h.(x-1)+k_h+1][s_w.(y-1)+1 : s_w.(y-1)+k_w+1][z])))$ thus ReLu will be applied $n_h \times n_w \times n_c$ times. On the contrary, if the ReLu is done after the pooling, we will do $O_{x,y,z} = ReLu(max(I[s_h.(x-1)+1 : s_h.(x-1)+k_h+1][s_w.(y-1)+1 : s_w.(y-1)+k_w+1][z])))$ thus the ReLu will be applied $o_h \times o_w \times o_c$ times. Note also that $max(max(x_i), 0) = max(max((x_i, 0))$, thus the semantics is preserved. ◀

▶ **Property 4** (Merging a batch norm with a convolution). *Applying a batch norm layer after a convolution layer is semantically equivalent to applying a single convolution with modified kernels and bias. This reduces the number of operations and saves memory bandwidth required for storing intermediate tensors.*

Proof. Let suppose that the input tensor I is of size (n_h, n_w, n_c) and that we have a convolution layer $\mathcal{C}_{p,K,s,B,f}$ followed by a batch-norm layer $\mathcal{BN}_{\alpha,\mathcal{B}}$. The output tensor is $O = \mathcal{BN}_{\alpha,\mathcal{B}}(\mathcal{C}_{p,K,s,B,f}(I))$.

$$\begin{aligned} O_{x,y,z} &= f\left(\alpha_z.\left(\sum_{i=1}^{f_h}\sum_{j=1}^{f_w}\sum_{m=1}^{f_c} K^z_{i,j,m}.I_{s_h\cdot(x-1)+i,s_h\cdot(y-1)+j,m} + B_{x,y,z}\right) + \mathcal{B}_z\right) \\ &= f\left(\sum_{i=1}^{f_h}\sum_{j=1}^{f_w}\sum_{m=1}^{f_c} \alpha_z.K^z_{i,j,m}.I_{s_h\cdot(x-1)+i,s_h\cdot(y-1)+j,m} + \alpha.B_{x,y,z} + \mathcal{B}_z\right) \end{aligned}$$

This is the equation of a convolution $\mathcal{C}'_{p,\alpha.K,s,\alpha.B+\mathcal{B},f}$ ◀

2.3 Model description for the code generation

Once a model has been trained, validated and possibly optimized with semantics-preserving transformations, its detailed description can be exported from the learning framework. As we want to generate the inference associated code, we assume the DNN representation to be cleaned from any irrelevant training-related feature (e.g. loss). A first challenge brought by the implementation is the *semantic preservation*: the reproducibility of the behavior observed at the end of the design when executing the *inference model* within the training tool and on the hardware target. Even though the semantics is clear in the literature, the training tools do not encode the (default) operations[3] in the same manner. This is particularly true for convolutions, where some implementations start from the top left and some from the bottom right of the matrix, or compute the padding in a different way. This has been observed in [24] and could be reproduced by experimenting with the frameworks. There are lots of works tackling the interoperability among frameworks, by proposing conversion tools [24] or

[3] when not specifying in detail the parameters, which could be very tricky

defining open source formats such as protobuf [15], Onnx [3] or Nnef [38] (Neural Network Exchange Format). A description must contain all the necessary information to encode the same behaviour: this includes the number of layers, the type of every layer, the parameters of each layer including the activation function specification and anything required to reproduce the behaviour. So far, Nnef is the most adequate format as it contains the necessary elements to reconstruct most of the semantics of a model. We currently use a degraded version of Nnef in Json to allow full text description (and not binary) to help the debugging but as future work we will comply with the Nnef.

3 C back-end

We have developed a Python prototype to generate C code. We do not detail the front-end which first imports the Json description file, focusing instead on the back-end. We reuse the semantics of definition 3 considering every layer as an independent programming function for the code generation. The *forward-pass* for inference then consists in calling each layer function in the correct order with the accurate parameters and inputs.

3.1 Software architecture

The C back-end is composed of a library of functions and other model-dependent files. This library is, to a certain extent, hard-coded as the bodies of functions needed for inference are defined in the Python prototype and the corresponding C files will be generated whenever needed. The model-dependent files refer to the weights, biases and auxiliary parameters that are also written as C files.

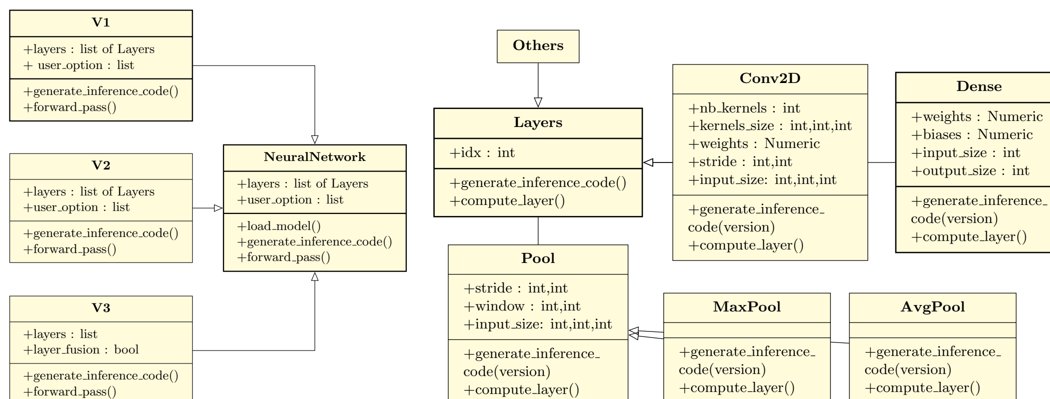

Figure 3 Software architecture – several versions.

Figure 4 Software architecture of *layers*.

Figure 3 shows the software architecture as an Uml diagram. There are several compilation strategies named V1, V2 and V3. We decided to proceed like that in order to allow a design space exploration (DSE): our goal was to understand what is the most suited approach for a given model and hardware. The main class *NeuralNetworks* contains two variables: *layers* that contains the list of *Layers* (another class defined hereafter) and *user_option* that captures the options chosen by the user for the generation, such as applying semantics-preserving transformations or selecting the version. That class defines three methods (in addition to the classical *init*): *load_model* which imports the Json DNN description; *forward_pass*

that concatenates the layers to encode the DNN function as the composition of layers and *generate_inference_code* which generates the C code. All the classes V1, V2 and V3 inherit from the *NeuralNetworks* class.

Figure 4 shows the *Layers* class which is inherited by several sub-classes, one per type of layer. The main class stores the *idx* of the layer and basically defines two abstract methods. The first is *generate_inference_code*, that implements the semantics of the layer in C language, and the second is *compute_layer*, that actually executes the functions of the layer, mainly for debugging and evaluation.

For each type of layer, we define the parameters (e.g. the weights and biases for *dense*) as variables and the methods (*generate_inference_code* and *compute_layer*) are refined. We did not detail all the layers (*others* grouping the missing ones).

The prototype supports all the layers defined in section 2.1 and the *ReLu*, *Hyperbolic Tangent*, *Sigmoid* and *Linear* activation functions.

3.2 Version 1 – generic inference function

A layer is defined as a data type, a *struct* statement, whose fields encode the parameters (e.g., type or input size). Every layer has the same definition and their particular parameters will be defined as constants in a header file. The first hidden layer of the fully-connected neural network of example 12 is a *Dense* layer depicted in listing 1.

Listing 1 A Dense layer – see Definition 11.

```
double biases_Dense_01[3] = {0.07543805986642838, 0.025200579315423965, 0.03704497963190079};
 ...
 struct Layer net[nb_layers]{
   [1] = {
     .layer_type = Dense,
     .layer_size = l1_size, // output_size
     .weights = weights_Dense_01,
     .biases = biases_Dense_01,
     .actv_function =  relu,
     .pad = 0x0,
     ... }, ... };
```

layer_type and *actv_function* are function pointers to the aforementioned functions of the C library. The other fields are pointers, mostly to arrays which are also written to C source files. In the example, the *biases* field points to a static double array of size 3 shown in the listing. Unnecessary fields point to null. The whole network is treated as an array, an indexed linear sequence, of these structures.

Afterwards, an *inference* function is defined (see listing 2). It is a generic function, i.e. identical for every DNN (whether fully-connected or not), responsible for connecting the layers. It simply consists in 2 nested *for* loops (one ranging over the number of layers and the second ranging over the number of operations to be done for the current layer).

Listing 2 Inference – see Definition 3.

```
for (int i=1; i < nb_layers; ++i) {
      net[i].layer_type(i, output_pre, output_cur);
      for (int j = 0; j < net[i].layer_size; ++j){
        output_pre[j] = output_cur[j]; }}
```

This logic of having generic definitions for the layers functions leads to a helpful simplicity in terms of execution and code size. However, using function pointers leads the WCET analysis tool to consider that each call made in the loop is a call to the most expensive function (or to the same function in worst context), which can be very pessimistic.

3.3 Version 2 – inlined inference function

The second version keeps the definition and declaration of layers as was done for Version 1. What changed is the *inference function* which is optimized by in-lining the programming functions for layers and activations, i.e., directly writing their body to the C file. The only parameters stored in a header C file are the weights and biases, since loops bounds are now hard-coded, meaning that the inference function is no longer generic. The Listing 3 gives part of the *inference function* for the first *dense* layer of example 12.

Listing 3 Dense layer in-lined code of the inference function.

```
for (int i = 0; i < 3; ++i) {   // Dense_1
    dotproduct = 0;
    for (int j = 0; j < 2; ++j) {
        dotproduct += output_pre[j] * weights_Dense_01[(i + 3*j)]; }
    dotproduct += biases_Dense_01[i];
    output_cur[i] = dotproduct > 0 ? dotproduct : 0;} ...
```

The straightforward effect of this optimization is improving time performance since we eliminate the function-call and struct parsing overheads, however it comes at the cost of using more instruction space, as we duplicate code, producing larger source files, which can be prohibitive in an embedded environment. Nonetheless, OTAWA produces more precise estimation for the WCET since we are able to provide the correct context in which layers are executed with no overestimation for loop bounds.

3.4 Version 3 – unrolled inference function

The third version is completely different and we reuse a philosophy of full in-lining (with loop unrolling) that can be seen *à la Scade* [9]. In particular, there is no declaration of layers and parameters as was done in listing 1. Listing 4 presents the beginning of the instructions to deal with the first *dense* layer of example 12.

Listing 4 Dense layer code with in-lining and loop-unrolling.

```
dotproduct = 0;        // Dense_1
dotproduct += nn_input[0] * -1.0743303298950195;
dotproduct += nn_input[1] * 0.8140403032302856;
dotproduct += 0.07543805986642838;
output_cur[0] = dotproduct > 0 ? dotproduct : 0;
dotproduct = 0;
dotproduct += nn_input[0] * -0.18220123648643494;
dotproduct += nn_input[1] * 0.7036496996879578;
dotproduct += 0.025200579315423965;
output_cur[1] = dotproduct > 0 ? dotproduct : 0;
```

The main advantages of this optimization are the elimination of computational overhead due to branching on the termination condition and the delay of reading data from memory, since everything needed for the layers operations is self contained in a C source file. Similarly to Version 2, we have the capacity of doing a better instruction pipelining. Additionally, we remove incertitude about the execution path, which is advantageous for the WCET analysis. However, it worsens the drawback already identified in the V2: the instruction space becomes huge and for large DNNs, the approach is not sustainable.

4 Comparative approach for C code generation frameworks

In order to test in practice the advantages and limitations of our framework, as well as its behavior compared to the other frameworks in the literature, we have defined the following methodology. We have selected a set of representative benchmarks (section 4.1) of the literature compliant with our restrictions (e.g. feed-forward DNN with restricted types of layers). The idea was to consider a large test campaign by varying several parameters (number

and type of layers, data type of parameters, type of activation). We then define three criteria to assess the quality of implementation in accordance with the DO-178C requirements (see section 4.2). In particular, not all criteria require the same level of test campaign: computing the WCET needs to be done once whereas the measurements need to be repeated several times. Finally, we introduce the two code generation frameworks selected for comparison (see section 4.3).

4.1 Benchmark description

Fully-connected networks – ACAS-Xu experience. The first models correspond to the classical fully-connected networks as shown in the example 12. We rely in particular on the airborne collision avoidance system for unmanned aircraft (ACAS-Xu) [27]. The ACAS-Xu system takes five input variables, i.e., information from sensors measurements, and computes five action advisories, represented by scores. The original design relies on a set of off-line computed lookup tables (LUT) to make avoidance decisions. Some work [19, 12] proposed to replace those LUT with some surrogate neural networks in order to reduce the memory footprint and thus to improve the execution time. We consider several DNN models with various structures, all with a ReLU activation function in hidden layers, linear activation for output layer and floating-point single precision (FP32) data type:

- regular structures with the same number of neurons per layer. We consider 7 hidden layers with *reg50* (50 neurons per layer), *reg100* (100 neurons per layer) and *reg200* (200 neurons per layer);
- decreasing structures with *decr128* (5 hidden layers of size (128, 128, 64, 32, 16)) and *decr256* (6 hidden layers of size (256, 256, 128, 64, 32, 16));

LeNet-5. The LeNet-5 model [23] refers to the feed-forward convolutional neural network introduced in the example 13. It is one of the earliest models of this type and is known for promoting the development of deep learning with the introduction of the back-propagation algorithm. Although this model is simple, it contains the main basic layers: convolution, pooling and dense layers. All the layers have the same tangent hyperbolic activation function, except for the last one, where a softmax is performed. Thus, it has 44,426 trainable parameters to stock and an inference pass executes 572,504 floating-point operations (FLOPs).

CifarNet. CifarNet was first introduced in [20] and was for a long time the state-of-the-art model used to solve the object classification problem on the Cifar-10 dataset, which consists of 32 x 32 RGB images of 10 classes. CifarNet is composed of three convolutional layers, and its pooling layers, followed by two dense layers (see figure 5). The ReLu activation function is applied to all the layers. The main difference with LeNet-5 is that it has a three-dimensional input and the convolutional layers have additional parameters such as padding and a non equal to 1 stride, which adds some complexity in terms of computation. With this configuration the number of trainable parameters increases to 122,570 alongside with 9,18 million FLOPs for inference.

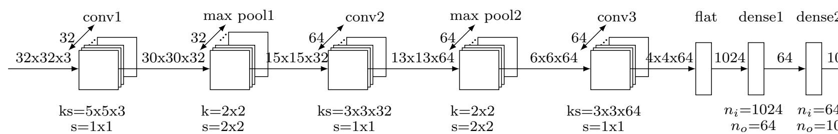

Figure 5 CifarNet CNN.

AlexNet. The AlexNet architecture was first defined in [21] and is considered as one of the most influential works in computer vision. Indeed, thanks to the use of convolution layers and GPUs to accelerate deep learning, it achieved a considerably improved performance over other methods in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of 2012. The ImageNet dataset [13] is composed of 256 x 256 RGB images categorized under 1000 object class categories. AlexNet has five convolution layers, three pooling layers and three dense layers (see figure 6[4]) with approximately 61 million tunable parameters and 1,64 billion FLOPs. Additionally, this model uses the ReLU activation function, which was presented as novelty and proved to be more efficient in learning phase than the, at the time, standard hyperbolic tangent [21].

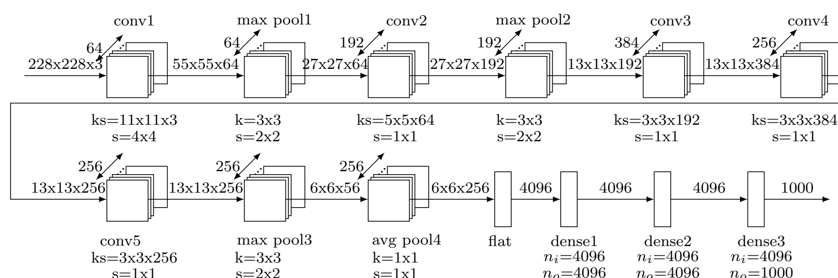

Figure 6 AlexNet CNN.

4.2 Criteria of comparison

We have identified three criteria of comparison that correspond to the most important avionics constraints to be respected.

Semantic preservation. To validate the correctness of the code generation, we need to prove the *semantic preservation*, that is the capacity to reproduce the inference observed in the training tool on the target. To do so, we could have used formal methods (such as Coq [36] as was done for Velus [6]) but instead, we chose to review the code generated and run a large campaign of tests. This technique may be less sound but is indeed an acknowledged way in the certification standard DO178C [14]. The semantic preservation is assessed by comparing the predictions of the C code with those provided by the training framework.

▶ **Definition 15** (Semantic preservation). *Let $x = (x_1, x_2, ..., x_n)$ be a vector representing the training framework outputs for a given set of inputs and $\tilde{x} = (\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n)$ be the vector of outputs of the C code execution. We define the absolute error as*

$$\|\tilde{x} - x\|_\infty = max_{0 \le i \le n} |(\tilde{x} - x)_i|$$

This norm asserts a maximum bound on the error observed for a given testing sample.

Measured and Worst Case Execution Time. For each C code, we want to assess both its performance and its predictability. The performance is evaluated by executing the code an Arm Cortex-A15 of the Keystone [37] and measuring the execution time.

▶ **Definition 16** (Measured execution time). *To obtain the measured execution time, we run a sample (i.e. an input) 50 times and store the average observed time.*

[4] There is a pre-processing that consists of a *scaling* from 256 x 256 to 228 x 228 of the input

In order to assess the predictability, we compile each C code for a *lpc2138* Arm-based target, and compute its WCET with OTAWA [4]. The choice of the hardware (*lpc2138* Arm-based target) was dictated by the libraries available in the OTAWA framework. Even though it is not representative of the Arm Cortex-A15 of the Keystone, the comparison between the WCETs of the various versions still provides valuable insights on how the shape of the generated code impacts the level of precision that can be achieved during the analysis. Despite its limitations OTAWA is open-source and presently maintained, thus up to date with current WCET calculation techniques. We did not experiment with other static timing analysis tools that may or may not have the same limitations.

Memory layout of executable. Because it is important to efficiently use the resources in order to be predictable and efficient, we also analyze the memory layout of the C executable. The memory space is segmented into discrete blocks with specific purposes. We mainly focus on the stack, data, BSS and text segments. The stack segment contains all the data needed by a function call, including the arguments passed to the routine and its local variables. The data segment contains the explicitly initialized global variables and static local variables, its size does not change at runtime. Uninitialized variable data are stored in the BSS segment. Lastly, the text segment contains the executable instructions and constant variable that can not be modified.

4.3 Others C back-end frameworks

We chose two open-source frameworks from the TinyML [31] domain that were developed with nearly the same objectives as ACETONE.

Keras2C. As explained in [10], the Keras2c back-end was developed to address real-time applications and not to optimize the code for speed. Indeed, the generated C code layout is very similar to Version 1 presented in Section 3.2, where the programming functions describing the layers are generic and all the mutable data are passed into and out of each function during the inference execution. Thus, in terms of timing analysis, Keras2c presents the same downside as our first version, which is an overestimated WCET due to the inability of passing the context in which a function is called when there are multiple occurrences of it.

Another drawback observed in this framework is the declaration of the weights of layers as local variables initialized in the core of the function. Thus weight arrays are always allocated on the stack. In case of large networks, heavy arrays are then stored within the stack and such an approach is not at all recommended. Moreover, these arrays shall be initialized in each function with a memcopy from the reference one declared by the compiler in the text segment. This is not ideal for the computation time that is waste to copy weights for each layer and each inference. We preferred a zero copy strategy using static variable declaration for the weights, this saves space in the stack and computation time.

To avoid declaring to heavy array on the stack, Keras2c chooses to use dynamic memory allocation when working with large neural networks, which implies additional certification challenges in terms of verification and is not at all suited for WCET analysis.

MicroTVM with static C runtime. The TVM compiler [7] outputs a *model execution graph* – encoded as Json– and simplified parameters. In order to execute the model, the TVM runtime has to rebuild this graph in memory, load the parameters, and then call the operator implementations in the correct order by parsing the computation graph. This is the principle of a graph interpreter / executor that we also found in Tensorflow.

uTVM [11] is a runtime developed to execute graphs produced by TVM on bare-metal targets. The code generation flow remains mostly the same, specific changes are needed in the runtime in order to avoid the usage of traditional operating-systems abstractions and support standalone model inference. The main parts of uTVM process are:

1. the production of a relay module depending of the training framework;
2. compilation, where TVM implements each operator into tensor intermediate representation followed by code generation;
3. integration of the generated code along with TVM C runtime library, into a user-supplied binary project;
4. and deployment, when a binary is built and inference can be run.

A drawback of this graph executor logic in an avionics context is the amount of memory overhead required in parsing the Json, a dynamic scheduling and a dynamic memory allocation, which we are not able to analyze.

To bypass these limitations, [35] provided a patch to uTVM that relies on a static scheduling and memory allocation. We call that framework *uTVM with static C runtime* or Static uTVM subsequently. It uses the relay module produced by TVM and generates a dedicated C source code that calls the generated operator implementations directly, eliminating the need of a graph Json parsing, and which is able to execute the model statically. By doing minor changes in this Static uTVM, we were able to proceed to a timing analysis of the inference model and could observe that the generated code when analyzed with OTAWA is very similar to our Version 2 (Section 3.3).

5 Experiments

This section summarizes the results when assessing the criteria for the different frameworks and the different benchmarks. We have in addition to the benchmarks identified before, considered VGG-16 [34]. Unfortunately, we only manage to generate the C code for V1 and V2 and analyze the semantic preservation. Other frameworks and analyses were not able to handle such a large network (138.36 million parameters).

Semantic preservation. We use the formula of definition 15 to compute the maximal observed error over 1000 tests when the generated code were executed on a x86 target. The three versions (V1, V2, V3) encode the same semantics, so no need to make them all appear. For our tool and for Keras2c, the reference was Keras and for Static uTVM it was Tensorflow Lite. The results using single-precision FP are shown in table 1. We can note that all the frameworks produce very similar results with errors in the order of 10^{-6}, which is considered acceptable. For the ACAS-Xu regular models, using the learnt parameters (weights and biases) present in the lookup tables led to values larger than 1 (around 10^{5}) in outputs. This had an influence in the floating point precision which in turn affected our semantic preservation assessment, so we proceeded to use random initialized parameters and have normalized outputs instead.

Measured and Worst Case Execution Time. We measured the inference time on the Arm Cortex-A15 (implementing the ARMv7 architecture) of the Keystone. For all experiments, caches were activated and we put data and code sections in the DDR. We used the flag *mfloat-abi=hard* in order to use the *neon floating point unit* of the processor. C codes were compiled without any optimization level (-O0). Table 2 shows the results where the measured execution times (MET) are computed following definition 16.

Table 1 Results for the semantic preservation in FP32 precision.

	Maximum error								
Framework	ACAS-Xu *reg50*	ACAS-Xu *reg100*	ACAS-Xu *reg200*	ACAS-Xu *decr128*	ACAS-Xu *decr256*	LeNet-5	CifarNet	AlexNet	VGG-16
Ours (V1)	2.0265e-06	1.4305e-06	4.7683e-07	1.4305e-06	5.9604e-07	1.7881e-06	6.1988e-06	2.142e-06	4.7087e-06
Keras2C	2.0265e-06	1.4305e-06	4.7683e-07	8.34465e-07	9.5367e-07	2.0265e-06	5.6028e-06	–	–
uTVM static	1.6689e-06	9.5367e-07	1.1921e-07	2.3842e-07	2.3842e-07	1.9073e-06	4.2915e-06	–	–

Table 2 Measured execution times on the Arm with -O0 flag.

	Execution time [cycles]				
Framework	ACAS-Xu *reg50*	ACAS-Xu *decr128*	ACAS-Xu *decr256*	LeNet-5	CifarNet
Ours (V1)	381 439	888 190	3 975 111	23 934 418	464 386 831
Ours (V2)	243 195	533 767	2 339 851	12 186 378	233 450 428
Ours (V3)	357 483	650 895	6 466 297	–	–
Keras2C	499 315	1 104 134	4 977 515	25 786 401	642 390 830
uTVM static	416 796	681 708	2 677 785	10 201 249	193 599 362

Table 3 Measured execution times on the Arm with -O3 flag.

	Execution time [cycles]	
Framework	ACAS-Xu *decr256*	CifarNet
Ours (V2)	441 992	53 773 643
Keras2C	2 117 467	273 594 356
uTVM static	291 609	69 022 625

Among our versions, V2 produces the best MET. V2 has even a better MET than Keras2c and static uTVM for fully-connected networks (ACAS), and is slightly slower that uTVM for CNNs. Indeed, for the latter, static uTVM performs additional optimization (e.g. on tensor operations). On fully-connected networks, the tensor operations are basic matrix multiplications that do not require any optimization techniques. Keras2c has the worst MET for all benchmarks: we attribute that to the strategy to allocate weights tensors on the stack that adds a *memcpy* overhead at each layer (copy the weights from .text to stack).

Outside the avionics world, performance is looked for and thus inference codes are generally compiled with the -O3 option. Calling for this option enables the utilization of *Single Input Multiple Data (SIMD)* instructions on the Keystone. We thus also compiled two benchmarks with this flag to observe the impact. The results are given in table 3. First, for all versions, the MET is greatly reduced, due to the SIMD instructions well adapted to these algorithms. Keras2c has the same drawback due to copy of weights on the stack. Since -O3 only optimizes the computation of tensor operations, the time to copy data remains the same. Thus, the difference between Keras2c and two others remains high. Secondly, V2 has best MET for CNN and worst with fully-connected network. We do not try to optimize the utilization of SIMD instruction (array organization), thus we also believe this is not the case of static uTVM. This would require a dedicated back-end for floating point unit of Arm.

Table 4 WCET given by OTAWA for different benchmarks.

	WCET [cycles]						
Framework	ACAS-Xu *reg50*	ACAS-Xu *reg100*	ACAS-Xu *reg200*	ACAS-Xu *decr128*	ACAS-Xu *decr256*	LeNet-5	CifarNet
Ours (V1)	8 025 404	21 288 195	84 655 395	26 092 073	121 206 406	6 881 827 044	361 743 738 250
Ours (V2)	5 617 830	13 971 737	55 122 437	6 128 253	24 461 227	165 718 749	3 018 534 290
Keras2c	5 033 535	19 692 951	79 383 490	36 838 054	112 237 358	1 160 385 934	97 959 064 345
static uTVM	4 008 298	15 711 232	58 832 502	6 765 413	27 015 092	113 449 651	3 215 754 680

Table 4 shows the WCET of the benchmarks. OTAWA requires flow-fact information, that is information about the control flow: loop bounds and addresses of targets for indirect function calls (function pointers). Obtaining this information for our generated code was

easy (and making this process automatic is part of future work). For Keras2c and uTVM, we had to first modify the generated code to analyze only the inference code (as we did for our code), and to leave the initialization functions out of the WCET. OTAWA was not able to provide a WCET bound for V3 nor for AlexNet and VGG-16 architectures, because those binaries are too large and it runs out of memory during the analysis.

Looking at Table 4, we observe that shape of the C code has a significant impact on the WCET bound. This is not simply a question of performance optimizations, but also of the capacity to provide precise flow-fact information to the analyzer. C codes that employ function pointers (V1 and Keras2c) overall get larger WCETs than the others, because we were unable to provide contextual information about the layers function calls. When all layers perform an equivalent number of operations (the ACAS-Xu regular structures), this impact is reduced. For the other cases, the pessimism appears clearly such as for decr256. Indeed, although decr256 performs less computations than reg200, as attested by the WCET of V2 and static uTVM, the WCETs for V1 and Keras2c are significantly higher than the ones of the reg200.

OTAWA assumes that each call to a layer function is a call to the worst layer. In V2, the layers are implemented as a sequence of separate loops, and in static uTVM as a sequence of separate instructions calling the layer functions. Consequently, OTAWA is able to benefit from the detailed flow-fact information for these versions.

Memory layout of executable. We analyzed the memory layout of the generated codes when compiled to ARM Cortex-A15. For the sake of simplicity, we only present the results obtained for the ACAS-Xu *reg50* model as the same trend is observed for the other models. From Table 5 it is possible to understand how different the memory usage of the different frameworks is.

Table 5 Memory layout of the executable generated for ACAS-Xu *reg50*.

	Size of memory segments [bytes]			
Benchmark	**stack**	**.data**	**.bss**	**.text**
Ours (V1)	240	66 548	708	17 004
Ours (V2)	158	65 860	708	18 556
Ours (V3)	140	2 444	708	1 603 980
Keras2c	129 280	–	2 840	12 744 060
static uTVM	210	–	2 808	12 688 208

In our work, we privileged writing all parameters as constants to statically allocate all memory at compile time and better use the stack, which is also translated in the data segment size. The non-initialized data basically corresponds to the outputs. Additionally, in V3 we observe that the text segment is bigger since all constants are directly written in the C source code. We notice that, V1 and V2 are very efficient in terms of .text and stack size compared to Keras 2c or static uTVM. Because Keras2c allocates all the weights tensors on the stack, the stack size is higher than other versions. Moreover, weights shall also be present in the .text segment. We notice, that the stack size is much higher than the size required for storing weights. In addition Keras 2c allocates work arrays that are not used for computing dense layer. Our stack measurement is coherent with stack information given by *gcc* compiler. For uTVM, we explain the size of the .text by all tensor operations functions embedded in the TVM library. In our version, we only embedded necessary tensor operations function.

6 Related Work

We found plenty of frameworks that provide the possibility to run neural networks. Most of them rely on an inference engine that dynamically explores a computation graph. Without ignoring them, we decided to focus the related work on tools that are more adapted to avionics constraints.

Generic C code generator frameworks. The first work [8] is guided by avionics constraints as well and, in order to provide an efficient implementation of DNN inference models, the authors developed an automatic code generator that allows preserving semantics of the trained machine learning model. However, the code generation tool is not extensively described nor made available.

The second is Keras2c [10]. This method consists in a library to convert Keras models into real-time compatible C code, supporting a wide range of layers and relying only on C standard library functions. In the section 5, we have extensively compared our results with Keras2c. The study of [29] also investigates a predictable implementation of neural networks for safety-critical cyber-physical systems. They embed the Keras2c code on Patmos, a time-predictable processor, which is part of the larger T-CREST [32] project. The software tool-chain of the latter includes a LLVM-based compiler and the Platin tool for WCET analysis.

uTVM [11] is an extension of TVM that provides an implementation of TVM for micro-controllers already presented in section 4.3. The adaptation of uTVM with static C runtime [35] has extensively been compared with our results in the section 5.

N2D2 [33] is an end to end framework from the creation of the model to its implementation including the training. On the code generation, the authors explore how approximation techniques can improve the performance and energy efficiency of hardware accelerators in machine learning applications. We will assess these tools as future work.

Proprietary code generator frameworks. New massively parallel hardware adapted to neural networks need specific programming pattern in order to obtain the best possible computation performance. Hardware manufacturers provide tools that enable clients to generate optimized code for their target. We can cite eIQ [26] from NXP, TensorRT [25] from NVIDIA, KaNN [17] from Kalray or OpenVINO [16] from Intel. eIQn, TensorRT and OpenVino rely on a dynamic graph explorer runtime while KaNN proposes a static scheduling and memory allocation. These tools only generate optimized code for specific targets and do not implement a generic approach. Xilinx Vitis AI is a tool that generates application code for Xilinx targets. Such targets are composed of a host CPU (from the x86 or ARM families) and a hardware accelerator that is composed of programmable logic (FPGA). The tool generates C code for the host, and so-called "kernels" that are called by the host (using an API such as OpenCL) and executed by the accelerator. The data transfers between the host and the accelerator are handled using Xilinx runtime.

CoreAVI[5] claims to develop code generation toolchains for AI models compatible with DO-178C and ISO-26262 requirements. They mainly target GPUs with Intel Tiger Lake or AMD E9171. We were not able to assess their solution and we believe that they are only supporting CUDA or Vulkan code generation (no C generation).

[5] `https://coreavi.com`

The Matlab Coder toolbox allows the generation of the C code for the inference of an already trained network. The generated code requires no external library, which makes it portable. Ansys[6] proposes to use the Scade toolchain to generate C code compatible with DO178C requirements. To our knowledge, this targets at this time traditional processors and relies on the conversion of neural networks models into Lustre nodes. Then, they use the qualified C code generator. The converter AI models into Scade will have to guarantee the semantic preservation.

LLVM front-end frameworks. TVM [7] is a tool capable of compiling machine learning models from different popular frameworks and generating specific low-level optimized code for a diverse set of hardware back-ends.

MLIR (Multi-Level Intermediate Representation Overview) [22] is a LLVM intermediate representation which was developed with the idea to use the same IR for all compiler optimizations (hence the "Multi-Level"). It contains particular features that target machine learning applications, in particular it is possible to represent computation graphs in MLIR. MLIR can be instantiated into dialects that allow to put the focus on particular aspects of the code, to specify constraints or apply specific optimizations. An example of MLIR dialect that is particularly relevant to critical embedded applications such as the ones we target is described in [30]: it enables the semantics of synchronous reactive applications inside an MLIR description.

7 Conclusions

Machine learning applications are proven to be useful and are largely used in many domains, however, most of them are not built with avionics constraints in mind. In this work, we presented our approach to automatically reproduce the inference model of feed-forward neural networks in C code, respecting semantic preservation, predictability and aeronautic requirements. We proposed a framework that is modular and straightforward, capable of generating readable and traceable code. We also compared the present work with the state of the art and proved our approach to be competitive in the evaluated criteria.

As future work, we have already identified along the paper many improvements to be made (e.g. compliance with Nnef, automatic flow-fact generation). We will continue exploring other frameworks to get the best practices. We also plan to target parallel C code execution. The current versions are suitable for pipelining or parallelizing computations.

References

1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

2 Erin Alves, Devesh Bhatt, Brendan Hall, Kevin Driscoll, Anitha Murugesan, and John Rushby. Considerations in assuring safety of increasingly autonomous systems. *NASA*, 2018.

3 Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://onnx.ai/, 2019.

4 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis (regular paper). In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

[6] https://www.ansys.com/fr-fr/products/embedded-software/

5. Siddhartha Bhattacharyya, Darren Cofer, David Musliner, Joseph Mueller, and E. Engstrom. Certification considerations for adaptive systems. *2015 International Conference on Unmanned Aircraft Systems, ICUAS 2015*, pages 270–279, July 2015. doi:10.1109/ICUAS.2015.7152300.
6. Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th Conference on Programming Language Design and Implementation (PLDI)*, pages 586–601, 2017.
7. Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.
8. Sergei Chichin, Dominique Portes, Marc Blunder, and Victor Jegu. Capability to embed deep neural networks: Study on cpu processor in avionics context. In *10th European Congress Embedded Real Time Systems (ERTS 2020)*, 2020.
9. Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. Scade 6: From a kahn semantics to a kahn implementation for multicore. In *2018 Forum on Specification Design Languages (FDL)*, pages 5–16, 2018.
10. Rory Conlin, Keith Erickson, Joseph Abbate, and Egemen Kolemen. Keras2c: A library for converting keras neural networks to real-time compatible C. *Eng. Appl. Artif. Intell.*, 100:104182, 2021.
11. TVM consortium. microTVM: TVM on bare-metal, 2021. URL: https://tvm.apache.org/docs/topic/microtvm/index.html.
12. Mathieu Damour, Florence De Grancey, Christophe Gabreau, Adrien Gauffriau, Jean-Brice Ginestet, Alexandre Hervieu, Thomas Huraux, Claire Pagetti, Ludovic Ponsolle, and Arthur Clavière. Towards certification of a reduced footprint acas-xu system: A hybrid ml-based solution. In *40th International Conference Computer Safety, Reliability, and Security (SAFECOMP)*, pages 34–48, 2021.
13. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 248–255, 2009.
14. EUROCAE / RTCA. Do-178c, software considerations in airborne systems and equipment certification, 2011.
15. Google. Protocol buffers, 2001. URL: https://developers.google.com/protocol-buffers/.
16. Intel. Open vino documentation, 2018.
17. Kalray. Kann platform for high-performance machine learning inference on kalray's mppa® intelligent processor, 2021.
18. Kalray. Mppa® coolidge™ processor - white paper, 2021. URL: https://www.kalrayinc.com/documentation/.
19. Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *29th International Conference Computer Aided Verification (CAV)*, pages 97–117, 2017.
20. Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
21. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *26th Annual Conference on Neural Information Processing Systems*, pages 1106–1114, 2012.
22. Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, et al. MLIR: scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *International Symposium on Code Generation and Optimization, (CGO)*, pages 2–14, 2021.

23 Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, 1989.

24 Y. Liu, C. Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. Enhancing the interoperability between deep learning frameworks by model conversion. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

25 NVIDIA. Tensorrt documentation, 2021.

26 NXP. Eiq™ ml software development environment, 2020. URL: `https://www.nxp.com/design/software/development-software/eiq-ml-development-environment:EIQ`.

27 Michael P. Owen, Adam Panken, Robert Moss, Luis Alvarez, and Charles Leeper. Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2019.

28 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

29 Hammond Pearce, Xin Yang, Partha S. Roop, Marc Katzef, and Torur Biskopsto Strom. Designing neural networks for real-time systems. *IEEE Embedded Systems Letters*, pages 1–1, 2020.

30 Hugo Pompougnac, Ulysse Beaugnon, Albert Cohen, and Dumitru Potop-Butucaru. From SSA to Synchronous Concurrency and Back. Research Report RR-9380, INRIA Sophia Antipolis - Méditerranée (France), December 2020. URL: `https://hal.inria.fr/hal-03043623`.

31 Partha Pratim Ray. A review on tinyml: State-of-the-art and prospects. *Journal of King Saud University - Computer and Information Sciences*, 34(4):1595–1623, 2022.

32 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

33 Olivier Sentieys, Silviu Filip, David Briand, David Novo, Etienne Dupuis, Ian O'Connor, and Alberto Bosio. Adequatedl: Approximating deep learning accelerators. In *24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS 21)*, 2021.

34 Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations (ICLR)*, 2015.

35 Rafael Stahl. µtvm staticrt codegen, 2021. URL: `https://github.com/tum-ei-eda/utvm_staticrt_codegen`.

36 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, version 8.0 edition, 2004. URL: `http://coq.inria.fr/`.

37 Texas Instruments. TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip. Technical Report SPRS893E, Texas Instruments Incorporated, 2013.

38 The Khronos NNEF Working Group. Neural Network Exchange Format, 2018.

39 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.