# Fixed-Point Code Synthesis for Neural Networks[*]

Hanane Benmaghnia[1], Matthieu Martel[1,2], and Yassamine Seladji[3]

[1] University of Perpignan Via Domitia, Perpignan, France
[2] Numalis, Cap Omega, Rond-point Benjamin Franklin 34960 Montpellier, France
[3] University of Tlemcen Aboubekr Belkaid, Tlemcen, Algeria
[1]{first.last}@univ-perp.fr,[3]yassamine.seladji@univ-tlemcen.dz

**Abstract.** Over the last few years, neural networks have started penetrating safety critical systems to take decisions in robots, rockets, autonomous driving car, etc. A problem is that these critical systems often have limited computing resources. Often, they use the fixed-point arithmetic for its many advantages (rapidity, compatibility with small memory devices.) In this article, a new technique is introduced to tune the formats (precision) of already trained neural networks using fixed-point arithmetic, which can be implemented using integer operations only. The new optimized neural network computes the output with fixed-point numbers without modifying the accuracy up to a threshold fixed by the user. A fixed-point code is synthesized for the new optimized neural network ensuring the respect of the threshold for any input vector belonging the range $[x_{min}, x_{max}]$ determined during the analysis. From a technical point of view, we do a preliminary analysis of our floating neural network to determine the worst cases, then we generate a system of linear constraints among integer variables that we can solve by linear programming. The solution of this system is the new fixed-point format of each neuron. The experimental results obtained show the efficiency of our method which can ensure that the new fixed-point neural network has the same behavior as the initial floating-point neural network.

**Keywords:** Computer Arithmetic, Code Synthesis, Formal Methods, Linear Programming, Numerical Accuracy, Static Analysis.

## 1 Introduction

Nowadays, neural networks have become increasingly popular. They have started penetrating safety critical domains and embedded systems, in which they are often taking important decisions such as autonomous driving cars, rockets, robots, etc. These neural networks become larger and larger while embedded systems still have limited resources (memory, CPU, etc.) As a consequence, using and running deep neural networks [26] on embedded systems with limited resources introduces several new challenges [7,9,11,12,15,16,18,19]. The fixed-point arithmetic is more adapted for these embedded systems which often have a working processor with integers only. The approach developed in this article concerns the fixed-point and integer arithmetic applied to trained neural networks (NNs). NNs are trained on computers with a powerful computing unit using most of the time the IEEE754 floating-point arithmetic [13,21]. Exporting NNs using fixed-point arithmetic can perturb or change the answer of the NNs which are in general sensible to the computer arithmetic. A new approach is required to adapt NN computations to the simpler CPUs of embedded systems. This method consists in using fixed-point arithmetic because it is faster and lighter to manipulate for a CPU while it is more complicated to handle for the developer. We consider the problem of tuning the formats (precision) of an already trained floating-point NN, in such a way that, after tuning, the synthesized fixed-point NN behaves almost like the original performing computations. More precisely, if the NN is an interpolator, i.e. NNs computing mathematical functions, the original NN (floating-point) and the new NN (fixed-point) must behave identically if they calculate a given function $f$, such that, the

---

absolute error (Equation (1)) between the numerical results computed by both of them is equal to or less than a threshold set by the user. If the NN is a classifier, the new NN have to classify correctly the outputs in the right category comparing to the original NN. This method is developed in order to synthesize NNs fixed-point codes using integers only. This article contains nine sections and an introductory example in Section 3, where we present our method in a simplified and intuitive way. Some notations are introduced in Section 2. In Section 4, we present the fixed-point arithmetic, where we show how to represent a fixed-point number and the elementary operations. The errors of computations and conversions inside a NN are introduced in Section 5. Section 6 deals with the generation of constraints to compute the optimal format for each neuron using linear programming [30]. Our tool and its features are presented in Section 7. Finally, we demonstrate the experimental results in Section 8 in terms of accuracy and bits saved. Section 9 presents the related work then Section 10 concludes and gives an overview of our future work.

## 2   Notations

In the following sections, we will use these notations:
●$\aleph$: Set of fixed-point numbers. ●$\mathbb{N}$: Set of natural integers. ●$\mathbb{Z}$: Set of relative integers. ●$\mathbb{R}$: Set of real numbers. ●$\mathbb{F}$: Set of IEEE754 floating-point numbers [13]. ●**NN**: Neural Network. ● $< M^{\hat{x}}, L^{\hat{x}} >$: Format of the fixed-point number $\hat{x}$ where $M^{\hat{x}}$ represents the Most significant bit (integer part) and $L^{\hat{x}}$ the Least significant bit (fractional part). ●$\epsilon_{\hat{x}}$: Error on the fixed-point number $\hat{x}$. ●$b$: Bias. ●$W$: Matrix of weights. ●$m$: Number of layers of a neural network. ●$n$: Number of neurons by layer. ●$k$: Index of layer. ●$i$: Index of neuron. ●**ufp**: Unit in the first place [13,21]. ● **ReLU**: Rectified Linear Unit [10, 24,29]. ●$T$: size of data types (8, 16, 32 bits.) ●$\oplus$: Fixed-point addition. ●$\otimes$: Fixed-point multiplication.

## 3   An Introductory Example

In this section, we present a short example of a fully connected neural network [3] containing three layers ($m = 3$) and two neurons by layer ($n = 2$) as shown in Figure 1. The objective is to give an intuition of our approach.

Our main goal is to synthesize a fixed-point code for an input NN with an error threshold between 0 and 1 defined by the user, and respecting the initial NN which uses the floating-point arithmetic [13, 21]. The error threshold is the maximal absolute error accepted between the original floating-point NN and the synthesized fixed-point NN in all the outputs of the output layer (max norm). This absolute error is computed by substracting the fixed-point value to the floating-point value (IEEE754 with single precision [13,21]) as defined in Equation (1). To compute this error, we convert the fixed-point value into a floating-point value.

$$Absolute\,error = |FloatingPoint\,Result - FixedPoint\,Result| \qquad (1)$$

In this example, the threshold is 0.02 and the data type $T = 32$ bits. In other words, the resulting error of all neurons in the output layer ($u_{30}$, $u_{31}$ of Figure 1) must be equal to or less than 0.02 using integers in 32 bits.

Hereafter, we consider the feature layer $X_0$, which corresponds to the input vector. The biases are $b_0$, $b_1$ and $b_2$. The matrices of weights are $W_0$, $W_1$ and $W_2$, such that each bias (respectively matrix of weights) corresponds to one layer.
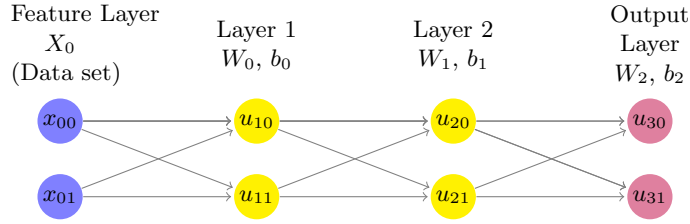
**Fig. 1.** A fully connected NN with 3 layers and 2 neurons per layer.

The affine function for the $k^{th}$ layer is defined in a standard way as

$$f_{k,i}: \quad \mathbb{F}^n \longrightarrow \mathbb{F} \tag{2}$$
$$X_{k-1} \longmapsto u_{k,i} = f_{k,i}(X_{k-1}) = \sum_{j=0}^{n-1} (w_{k-1,i,j} \times x_{k-1,j}) + b_{k-1,i},$$

$\forall 1 \leq k \leq m$, $\forall 0 \leq i < n$, where $X_{k-1} = (x_{k-1,0}, ..., x_{k-1,n-1})^t$ is the input vector (the input of the layer $k$ is the output of the layer $k-1$), $b_{k-1} = (b_{k-1,0}, ..., b_{k-1,n-1})^t \in \mathbb{F}^n$ and $W_{k-1} \in \mathbb{F}^{n \times n}$ ($w_{k-1,i,j}$ is the coefficient in the line $i$ and column $j$ in $W_{k-1}$.)
Informally, a fixed-point number is represented by an *integer value* and a *format* $< M, L >$ which gives us the information about the number $M$ of significant bits before the binary point and $L$ the number of significant bits after the binary point required for each coefficient in $W_{k-1}, b_{k-1}, X_0$, and the output of each neuron $u_{k,i}$. We notice that, at the beginning, we convert the input vector $X_0$ in the size of the data type required $T$. In fixed-point arithmetic, before computing the affine function defined in Equation (2), we need to know the optimal formats $< M, L >$ of all the coefficients. To compute these formats, we generate automatically linear constraints according to the given threshold. These linear constraints formally defined in Section 6 are solved by linear programming [30], and they give us the optimal value of the number of significant bits after the binary point $L$ for each neuron. We show in Equation (3) some constraints generated for the neuron $u_{31}$ of the NN of Figure 1.

$$\begin{cases} L_{31}^u \geq 6 \\ L_{31}^u + M_{31}^u \leq 31 \\ L_{31}^u \leq L_{20}^x, \\ L_{31}^u \leq L_{21}^x, \\ ... \end{cases} \tag{3}$$

We notice that $L_{20}^x$ (respectively $L_{21}^x$) is the length of the fractional part of $u_{20}$ (respectively $u_{21}$.) The first constraint gives a lower bound for $L_{31}^u$, so the output $u_{31}$ in the output layer has to fullfil the threshold fixed by the user and the error done must be equal to or less than this one. In other words, the number of significant bits of each neuron in the output layer must be equal at least to 6 (if it is greater than 6, this means that we are more accurate.) The value 6 is obtained by computing the unit in the first place [13, 21] of the threshold defined as

$$\forall x \in \mathbb{F}, \quad \mathrm{ufp}(x) = \min \{i \in \mathbb{N} : 2^{i+1} > x\} = \lfloor \log_2(x) \rfloor \tag{4}$$

The second constraint avoids overflow and ensures compliance to the data type chosen by the user (integers on 8, 16 or 32 bits.) The third and fourth constraints ensure that the length of the fractional part $L_{31}^u$ computed is less than or equal to the length of the fractional parts of all its inputs ($L_{20}^x$ and $L_{21}^x$.) Using the formats resulting from the solver, firstly, we convert all the coefficients of weights matrices $W_{k-1}$, biases $b_{k-1}$ and inputs $X_0$ from floating-point values to fixed-point values.

Let $b_{k-1}, W_{k-1}$ and $X_0$ used in this example be

$$b_0 = \begin{pmatrix} -2 \\ 4.5 \end{pmatrix}, b_1 = \begin{pmatrix} 1.2 \\ 0.5 \end{pmatrix}, b_2 = \begin{pmatrix} 3 \\ 1 \end{pmatrix}, W_0 = \begin{pmatrix} 3.5 & 0.25 \\ -1.06 & 4.1 \end{pmatrix}, W_1 = \begin{pmatrix} -0.75 & 4.85 \\ 2.1 & 0.48 \end{pmatrix},$$

$$W_2 = \begin{pmatrix} -5 & 12.4 \\ 0.2 & -2 \end{pmatrix}, X_0 = \begin{pmatrix} 2 \\ 0.5 \end{pmatrix}.$$

Table 1 presents the output results for each neuron. The floating-point results are shown in the second column and the fixed-point results in the third one. The last column contains the absolute error defined in Equation (1) for the output layer ($u_{30}$, $u_{31}$) only.

**Table 1.** Comparison between the floating-point and the fixed-point results corresponding to the NN of Figure 1.

| Neuron | Floating-Point Result | Fixed-Point Result | Absolute Error |
|:---:|:---:|:---:|:---:|
| $u_{10}$ | 5.125 | 2624<3,9>= 5.125 | / |
| $u_{11}$ | 4.43 | 4535<2,10>= 4.4287 | / |
| $u_{20}$ | 18.8417 | 9643<5,9>= 18.8339 | / |
| $u_{21}$ | 13.3889 | 6854<5,9>= 13.3867 | / |
| $u_{30}$ | 74.8136 | 76620<9,10>= 74.8247 | $1.06 \times 10^{-2} \approx 2^{-7}$ |
| $u_{31}$ | -22.0094 | -22536 <7,10>= -22.0078 | $1.63 \times 10^{-3} \approx 2^{-10}$ |

The error threshold fixed by the user at the beginning was 0.02 (6 significant bits after the binary point.) As we can see, the absolute error of the output layer in the Table 1 is under the value of the threshold required. This threshold is fulfilled with our method using fixed-point arithmetic. Now, we can synthesize a fixed-point code for this NN respecting the user's threshold, the data type $T$, and ensuring the same behavior and quality as the initial floating-point NN.

Figure 2 shows some lines of code synthesized by our tool for the neurons $u_{10}$ and $u_{11}$ using Equation (2) and the fixed-point arithmetic. The running code gives the results shown in the third column of the Table 1. For example, the line 5 represents the input $x_{00} = 2$ in the fixed-point representation. This value is shifted on the right through 6 bits (line 7) in order to be aligned and used in the multiplication (line 8) by $w_{000} = 3.5$ represented by 112 in the fixed-point arithmetic. The fixed-point output $u_{10}$ (2624) in the Table 1 is returned by the line 16.

## 4    Fixed-Point Arithmetic

In this section, we briefly describe the fixed-point arithmetic as implemented in most digital computers [4,5,32]. Since fixed-point operations rely on integer operations, computing with fixed-point numbers is highly efficient. We start by defining the representation of a fixed-point number in Subsection 4.1, then we present briefly the operations needed (addition, multiplication and activation functions) in this article in Subsection 4.2.

### 4.1    Representation of a Fixed-Point Number

A fixed-point number is represented by an **integer** value and a **format** $< M, L >$ where $M \in \mathbb{Z}$ is the number of significant bits before the binary point and $L \in \mathbb{N}$ is the number of significant bits after the binary point. We write the fixed-point number $\hat{a} = value_{<M^{\hat{a}}, L^{\hat{a}}>}$ and define it in Definition 1.

**Definition 1.** *Let us consider $\hat{a} \in \aleph$, $A_{\hat{a}} \in \mathbb{N}$ such that*
$\hat{a} = (-1)^{s_{\hat{a}}}.A_{\hat{a}}.\beta^{-L_{\hat{a}}}$ *and* $P_{\hat{a}} = M_{\hat{a}} + L_{\hat{a}} + 1$, *where $\beta$ is the basis of representation,*

```
1   int main()
2   { /* That NN has 3 layers and 2 neurons per layer */
3     int mul, u[3][2], x[4][2];
4
5   x[0][0]=1073741824;        // <1,29>
6   x[0][1]=1073741824;        // <-1,31>
7   x[0][0]=x[0][0]>>6;        // <1,23>
8   mul=112*x[0][0];           // <3,9>=<1,5>*<1,23>
9   mul=mul>>19;
10  u[1][0]=mul;               // <3,9>
11  x[0][1]=x[0][1]>>7;        // <-1,24>
12  mul=16*x[0][1];            // <-2,14>=<-2,9>*<-1,24>
13  mul=mul>>19;
14  mul=mul>>5;
15  u[1][0]=u[1][0]+mul;       //<3,9>=<3,9>+<-2,9>
16  u[1][0]=u[1][0]+-1024;     //<3,9>=<3,9>+<1,9>
17  u[1][0]=max(0,u[1][0]);    // ReLU(u[1][0])
18  x[1][0]=u[1][0];
19  x[0][0]=x[0][0]>>3;        // <1,20>
20  mul=-543*x[0][0];          // <2,10>=<0,9>*<1,20>
21  mul=mul>>19;
22  u[1][1]=mul;               // <2,10>
23  x[0][1]=x[0][1]>>5;        // <-1,19>
24  mul=262*x[0][1];           // <2,10>=<2,10>*<-1,19>
25  mul=mul>>19;
26  u[1][1]=u[1][1]+mul;       //<2,10>=<2,10>+<2,10>
27  u[1][1]=u[1][1]+4608;      //<2,10>=<2,10>+<2,10>
28  u[1][1]=max(0,u[1][1]);    // ReLU(u[1][1])
29  ...
30  return 0; }
31
```

**Fig. 2.** Fixed-point code synthesized for the neurons $u_{10}$ and $u_{11}$ of Figure 1 on 32 bits.

$\hat{a}$ *is the fixed-point number with implicit scale factor* $\beta^{-L_{\hat{a}}}$ *(Figure 3),* $A_{\hat{a}}$ *is the integer representation of* $\hat{a}$ *in the basis* $\beta$, $P_{\hat{a}} \in \mathbb{N}$, $P_{\hat{a}} = M_{\hat{a}} + L_{\hat{a}} + 1$ *is the length of* $\hat{a}$ *and* $s_{\hat{a}} \in \{0, 1\}$ *is its sign.*
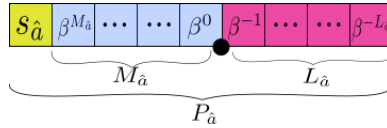


**Fig. 3.** Fixed-point representation of $\hat{a}$ in a format $< M^{\hat{a}}, L^{\hat{a}} >$.

The difficulty of the fixed-point representation is managing the position of the binary point manually against the floating-point representation which manages it automatically.

*Example 1.* : The fixed-point value $3 < 1, 1 >$ corresponds to 1.5 in the floating-point representation. We have first to write 3 in binary then we put the binary point at the right place (given by the format) and finally we convert it again into the decimal representation: $3_{10} < 1, 1 > = 11_2 < 1, 1 > = 1.1_2 = 1.5_{10}$.

## 4.2 Elementary Operations

This subsection defines the elementary operations needed in this article like addition and multiplication which are used later in Equation (11). We also define $Re\hat{L}U$ (respectively $\hat{Linear}$) in fixed-point arithmetic which corresponds to the activation function in some NNs [10, 24, 29].

1. **Fixed-Point Addition**
   Let us consider the two fixed-point numbers $\hat{a}$, $\hat{b} \in \aleph$ and their formats $< M^{\hat{a}}, L^{\hat{a}} >$, $< M^{\hat{b}}, L^{\hat{b}} >$ respectively. Let $\oplus$ be the fixed-point addition given by $\hat{c} \in \aleph$, $\hat{c} = \hat{a} \oplus \hat{b}$.
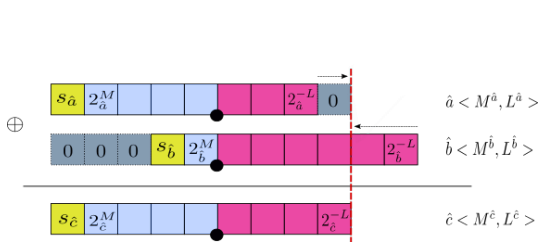


**Fig. 4.** Addition of two fixed-point numbers without a carry.
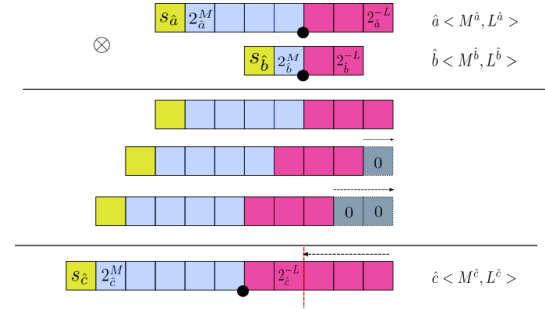


**Fig. 5.** Multiplication of two fixed-point numbers.

Figure 4 shows the fixed-point addition between $\hat{a}$ and $\hat{b}$. The fixed-point format required is $< M^{\hat{c}}, L^{\hat{c}} >$. The objective is to have a result in this format, this is why we start by aligning the length of the fractional parts $L^{\hat{a}}$ and $L^{\hat{b}}$ according to $L^{\hat{c}}$. If $L^{\hat{a}} > L^{\hat{c}}$, we truncate with $L^{\hat{a}} - L^{\hat{c}}$ bits, otherwise, we add $L^{\hat{c}} - L^{\hat{a}}$ zeros on the right hand of $\hat{a}$, then we do the same for $\hat{b}$. The length of the integer part $M^{\hat{c}}$ must be the maximum value between $M^{\hat{a}}$ and $M^{\hat{b}}$. If there is a carry, we add $+1$ to the number of bits of the integer part, otherwise the result is wrong. The algorithm of the fixed-point addition is given in [20, 23].

2. **Fixed-Point Multiplication**
   Let us consider the two fixed-point numbers $\hat{a}$, $\hat{b} \in \aleph$ and their formats $< M^{\hat{a}}, L^{\hat{a}} >$, $< M^{\hat{b}}, L^{\hat{b}} >$ respectively. Let $\otimes$ be the fixed-point multiplication given by $\hat{c} \in \aleph$, $\hat{c} = \hat{a} \otimes \hat{b}$.
   Figure 5 shows the fixed-point multiplication between $\hat{a}$ and $\hat{b}$. The fixed-point format required is $< M^{\hat{c}}, L^{\hat{c}} >$. The objective is to have a result in this format, this is why we start by doing a standard multiplication which is composed by shifts and additions. If $(L^{\hat{a}} + L^{\hat{b}}) > L^{\hat{c}}$, we truncate with $(L^{\hat{a}} + L^{\hat{b}}) - L^{\hat{c}}$ bits, otherwise, we add $L^{\hat{c}} - (L^{\hat{a}} + L^{\hat{b}})$ zeros on the right hand of $\hat{c}$. The length of the integer part $M^{\hat{c}}$ must be the sum of $M^{\hat{a}}$ and $M^{\hat{b}}$, otherwise the result is wrong. The algorithm of the fixed-point multiplication is given in [20, 23].

3. **Fixed-Point $Re\hat{L}U$**
   Definition 2 defines the fixed-point $Re\hat{L}U$ which is a non-linear activation function computing the positive values.

   **Definition 2.** *Let us consider the fixed-point number $\hat{a} = V_{\hat{a}} < M^{\hat{a}}, L^{\hat{a}} > \in \aleph$ and the fixed-point zero written $\hat{0} = 0 < 0, 0 > \in \aleph$. Let $\hat{c} = V_{\hat{c}} < M^{\hat{c}}, L^{\hat{c}} > \in \aleph$ be the result of the fixed-point $Re\hat{L}U$ given by*

$$\hat{c} = Re\hat{L}U(\hat{a}) = m\hat{a}x(\hat{0}, \hat{a}), \tag{5}$$

$$where \ V_{\hat{c}} = max(0, V_{\hat{a}}) \ and \ < M^{\hat{c}}, \ L^{\hat{c}} >= \begin{cases} < M^{\hat{a}}, L^{\hat{a}} > & if \quad V_{\hat{c}} = V_{\hat{a}}, \\ < 0, 0 > & otherwise. \end{cases}$$

4. **Fixed-Point $\hat{Linear}$**

   Definition 3 defines the fixed-point $\hat{Linear}$ which is an activation function returning the identity value.

   **Definition 3.** *Let us consider the fixed-point number $\hat{a} \in \aleph$ with the format $< M^{\hat{a}}, L^{\hat{a}} >$. Let $\hat{c} \in \aleph$ be the result of the fixed-point $\hat{Linear}$ activation function given by*

   $$\hat{c} = \hat{Linear}(\hat{a}) = \hat{a}. \tag{6}$$

## 5 Error Modelling

In this section, we introduce some theoretical results concerning the fixed-point arithmetic errors in Subsection 5.1 and we show the numerical errors done inside a NN in Subsection 5.2. The error on the output of the fixed-point affine transformation function can be decomposed into two parts: the propagation of the input error and the computational error. Hereafter, $\hat{x} \in \aleph$ is used for the fixed-point representation with the format $< M^{\hat{x}}, L^{\hat{x}} >$ and $x \in \mathbb{F}$ for the floating-point representation. $\bar{X} \in \mathbb{F}^n$ is a vector of $n$ floating-point numbers and $\hat{X} \in \aleph^n$ a vector of $n$ fixed-point numbers.

### 5.1 Fixed-Point Arithmetic Error

This subsection defines two important properties about errors made in fixed-point addition and multiplication which are used to compute affine transformations in a NN (substraction and division are useless in our context.) We start by introducing the propositions and then the proofs. Proposition 1 defines the error of the fixed-point addition when we add two fixed-point numbers and Proposition 2 defines the error due to the multiplication of two fixed-point numbers.

**Proposition 1.** *Let $\hat{x}, \hat{y}, \hat{z} \in \aleph$ with a format $< M^{\hat{x}}, L^{\hat{x}} >$ (respectively $< M^{\hat{y}}, L^{\hat{y}} >$, $< M^{\hat{z}}, L^{\hat{z}} >$.) Let $x, y, z \in \mathbb{F}$ be the floating-point representation of $\hat{x}, \hat{y}, \hat{z}$. Let $\epsilon_{\oplus} \in \mathbb{R}$ be the error between the fixed-point addition $\hat{z} = \hat{x} \oplus \hat{y}$ and the floating-point addition $z = x + y$. We have that*

$$\epsilon_{\oplus} \leq 2^{-L^{\hat{x}}} + 2^{-L^{\hat{y}}} + 2^{-L^{\hat{z}}}. \tag{7}$$

*Proof.* *Let us consider $\epsilon_{\hat{x}}, \epsilon_{\hat{y}}, \epsilon_{\hat{z}} \in \mathbb{R}$ errors of truncation in the fixed-point representation of $\hat{x}, \hat{y}$ and $\hat{z}$ respectively. These ones are bounded by $2^{-L^{\hat{x}}}$ ($2^{-L^{\hat{y}}}$, $2^{-L^{\hat{z}}}$ respectively) because $L^{\hat{x}}$ ($L^{\hat{y}}$, $L^{\hat{z}}$ respectively) is the last correct bit in the fixed-point representation of $\hat{x}$ (respectively $\hat{y}, \hat{z}$.)*
*We have that $z = x + y$, $\hat{z} = \hat{x} \oplus \hat{y}$ and $\epsilon_{\oplus} \leq \epsilon_{\hat{x}} + \epsilon_{\hat{y}} + \epsilon_{\hat{z}}$. Then we obtain $\epsilon_{\hat{z}} \leq 2^{-L^{\hat{x}}} + 2^{-L^{\hat{y}}} + 2^{-L^{\hat{z}}}$.* ∎

**Proposition 2.** *Let $\hat{x}, \hat{y}, \hat{z} \in \aleph$ with a format $< M^{\hat{x}}, L^{\hat{x}} >$ (respectively $< M^{\hat{y}}, L^{\hat{y}} >$, $< M^{\hat{z}}, L^{\hat{z}} >$.) Let $x, y, z \in \mathbb{F}$ be the floating-point representation of $\hat{x}, \hat{y}, \hat{z}$ in such a way $x = \hat{x} + \epsilon_{\hat{x}}$ (respectively $y = \hat{y} + \epsilon_{\hat{y}}$, $z = \hat{z} + \epsilon_{\hat{z}}$.)*
*Let $\epsilon_{\otimes} \in \mathbb{R}$ be the resulting error between the fixed-point multiplication $\hat{z} = \hat{x} \otimes \hat{y}$ and the floating-point multiplication $z = x \times y$. We have that*

$$\epsilon_{\otimes} \leq \hat{y} \times 2^{-L^{\hat{x}}} + \hat{x} \times 2^{-L^{\hat{y}}} + 2^{-L^{\hat{z}}}. \tag{8}$$

*Proof.* *Let us consider $\epsilon_{\hat{x}}, \epsilon_{\hat{y}}, \epsilon_{\hat{z}} \in \mathbb{R}$ errors of truncation of $\hat{x}, \hat{y}$ and $\hat{z}$ respectively. These ones are bounded by $2^{-L^{\hat{x}}}$ ($2^{-L^{\hat{y}}}$, $2^{-L^{\hat{z}}}$ respectively) because $L^{\hat{x}}$ ($L^{\hat{y}}$, $L^{\hat{z}}$ respectively) is the last correct bit in the fixed-point representation of $\hat{x}$ (respectively $\hat{y}, \hat{z}$.)*
*We have that $z = x \times y$, $\hat{z} = \hat{x} \otimes \hat{y}$. We compute $(\hat{x} + \epsilon_{\hat{x}}) \times (\hat{y} + \epsilon_{\hat{y}})$ and then we obtain*

$\epsilon_\otimes \leq \hat{y} \times \epsilon_{\hat{x}} + \hat{x} \times \epsilon_{\hat{y}} + \epsilon_{\hat{x}} \times \epsilon_{\hat{y}} + \epsilon_{\hat{z}}$. We get rid of the second order error $\epsilon_{\hat{x}} \times \epsilon_{\hat{y}}$ which is negligible in practice because our method needs to know only the most significant bit of the error which will be used in Equation (28) in Section 6. Now, the error becomes

$$\epsilon_\otimes \leq \hat{y} \times \epsilon_{\hat{x}} + \hat{x} \times \epsilon_{\hat{y}} + \epsilon_{\hat{z}}. \tag{9}$$

Finally, we obtain $\epsilon_\otimes \leq \hat{y} \times 2^{-L^{\hat{x}}} + \hat{x} \times 2^{-L^{\hat{y}}} + 2^{-L^{\hat{z}}}.$      ■

## 5.2   Neural Network Error

Theoretical results about numerical errors inside a fully connected NN using fixed-point arithmetic are shown in this subsection. There are two types of errors: round off errors due to the computation of the affine function in Equation (11) and the propagation of the error of the input vector.

In a NN with fully connected layers [3],$\forall \bar{b} \in \mathbb{F}^n$, $\forall W \in \mathbb{F}^{n \times m}$, an output vector $\bar{u} \in \mathbb{F}^n$ is defined as

$$f : \quad \begin{aligned} \mathbb{F}^m &\longrightarrow \mathbb{F}^n \\ \bar{X} &\longmapsto \bar{u} = f(\bar{X}) = W.\bar{X} + \bar{b} \end{aligned} \tag{10}$$

Proposition 3 shows how to bound the numerical errors of Equation (10) using fixed-point arithmetic.
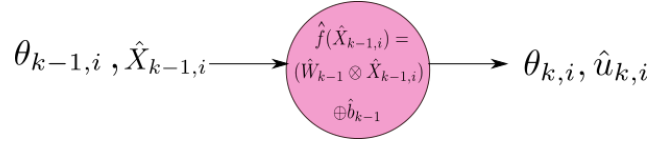


**Fig. 6.** Representation of $\hat{u}_{k,i}$ the $i^{th}$ neuron of the $k^{th}$ layer.

**Proposition 3.** *Let us consider the affine transformation as defined in Equation (11). It represents the fixed-point version of Equation (10). This transformation corresponds to what is computed inside a neuron (see Figure 6.) Let $\hat{u}_{k,i} \in \aleph$ be the fixed-point representation of $u_{k,i}$ and $\theta_{k,i} \in \mathbb{R}$ the error due to computations and conversions of the floating-point coefficients to fixed-point coefficients such that*

$$\hat{f} : \quad \begin{aligned} \aleph^n &\longrightarrow \aleph \\ \hat{X}_{k-1} &\longmapsto \hat{u}_{k,i} = \hat{f}(\hat{X}_{k-1}). \end{aligned} \tag{11}$$

*where $\hat{f}(\hat{X}_{k-1}) = \sum_{j=0}^{n-1}(\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}) \oplus \hat{b}_{k-1,i}$ and $\hat{X}_{k-1} = (\hat{x}_{k-1,0}, ..., \hat{x}_{k-1,n-1})^t$, $1 \leq k \leq m$, $0 \leq i < n$. Then the resulting error $\theta_{k,i}$ for each neuron $\hat{u}_{k,i}$ of each layer is given by*

$$1 \leq k < m+1, \;\; 0 \leq i < n, \quad \theta_{k,i} \leq \sum_{j=0}^{n-1}(2^{M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}}} + 2^{M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}}) + n \times 2^{-L_{k,i}^{\hat{u}}} + 2^{-L_{k,i}^{\hat{u}}+1}. \tag{12}$$

*Proof.* The objective is to bound the resulting error for each neuron (Figure 6) of each layer due to affine transformations by bounding the error of the formula of Equation (11). We compute first the error of multiplication $\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}$ using Proposition 2. Then we bound the fixed-point sum of multiplications $\sum_{j=0}^{n-1}(\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j})$ and finally we use

Proposition 1 to bound the error of addition of $\sum_{j=0}^{n-1}(\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}) \oplus \hat{b}_{k-1,i}$.

Let $\epsilon_\alpha \in \mathbb{R}$ be the error of $\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}$ and $2^{-L_{k,i}^{\hat{u}}}$ the truncation error of the output neuron $\hat{u}$ . Using Proposition 2, we obtain

$$\epsilon_\alpha \leq 2^{M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}}} + 2^{M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}} + 2^{-L_{k,i}^{\hat{u}}}. \tag{13}$$

Now, let us consider $\epsilon_\beta \in \mathbb{R}$ as the error of $\sum_{j=0}^{n-1}(\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j})$. This error is computed by using the result of Equation (13) such that

$$\epsilon_\beta \leq \sum_{j=0}^{n-1}(2^{-L_{k-1,i,j}^{\hat{w}}} \times 2^{M_{k-1,j}^{\hat{x}}} + 2^{-L_{k-1,j}^{\hat{x}}} \times 2^{M_{k-1,i,j}^{\hat{w}}}) + \sum_{j=0}^{n-2} 2^{-L_{k,i}^{\hat{u}}} + 2^{-L_{k,i}^{\hat{u}}}. \tag{14}$$

Consequently,

$$\epsilon_\beta \leq \sum_{j=0}^{n-1}(2^{M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}}} + 2^{M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}}) + n \times 2^{-L_{k,i}^{\hat{u}}}. \tag{15}$$

Finally, let $\epsilon_\gamma \in \mathbb{R}$ be the error of $\sum_{j=0}^{n-1}(\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}) \oplus \hat{b}_{k-1,i}$. Using Equation (15) and Proposition 1 we obtain $\epsilon_\gamma \leq \epsilon_\beta + 2^{-L_{k,i}^{\hat{u}}}$. Finally,

$$\epsilon_\gamma \leq \sum_{j=0}^{n-1}(2^{M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}}} + 2^{M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}}) + n \times 2^{-L_{k,i}^{\hat{u}}} + 2^{-L_{k,i}^{\hat{u}}+1}. \tag{16}$$

If we combine Equations (12) and (16), we obtain

$$\theta_{k,i} = \epsilon_\gamma \leq \sum_{j=0}^{n-1}(2^{M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}}} + 2^{M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}}) + n \times 2^{-L_{k,i}^{\hat{u}}} + 2^{-L_{k,i}^{\hat{u}}+1}. \quad \blacksquare \tag{17}$$

In this section, we have bounded the affine transformation error $\theta_{k,i}$ for each neuron $\hat{u}_{k,i}$ of each layer $k$ of the NN in Equation (17), respecting the equivalent floating-point computations. This resulting error $\theta_{k,i}$ is used in Section 6 to compute the optimal format $< M_{k,i}^{\hat{u}}, L_{k,i}^{\hat{u}} >$ for each neuron $\hat{u}_{k,i}$.

## 6   Constraints Generation

In this section, we demonstrate how to generate the linear constraints automatically for a given NN, in order to optimize the number of significant bits after the binary point $L_{k,i}^{\hat{u}}$ of the format $< M_{k,i}^{\hat{u}}, L_{k,i}^{\hat{u}} >$ corresponding to the output $\hat{u}_{k,i}$. Let us remember that we have a floating-point NN with $m$ layers and $n$ neurons per layer working at some precision, and we want to compute a fixed-point NN with the same behavior than the initial floating-point NN for a given input vector. This new fixed-point NN must respect the threshold error and the data type $T \in \{8, 16, 32\}$ bits for the C synthesized code. The variables of the system of constraints are $L_{k,i}^{\hat{u}}$ and $L_{k-1,i,j}^{\hat{w}}$. They correspond respectively to the length of the fractional part of the output $\hat{u}_{k,i}$ and $\hat{w}_{k-1,i,j}$. We have $M_{k,i}^{\hat{u}}, M_{k-1,i}^{\hat{x}}, M_{k-1,i,j}^{\hat{w}} \in \mathbb{Z}$, and $L_{k,i}^{\hat{u}}, L_{k-1,i}^{\hat{x}}, L_{k-1,i,j}^{\hat{w}} \in \mathbb{N}$, for $1 \leq k < m+1$, and $0 < i, j < n$, such that $M_{k,i}^{\hat{u}}$ (respectively $M_{k-1,i}^{\hat{x}}, M_{k-1,i,j}^{\hat{w}}$) can be negative when the value of the floating-point number is between $-1$ and $1$. We have also, $M_{k-1,i,j}^{\hat{w}}$ (respectively $M_{0,i}^{\hat{x}}$ the number of bits before the binary point of the feature layer) which is obtained by computing the ufp defined in

Equation (4) of the corresponding floating-point coefficient. Finally, the value of $M_{k,i}^{\hat{u}}$ is obtained through the fixed-point arithmetic (addition and multiplication) in Section 4. In Equation (18) of Figure 7 (respectively (19) and (20)), the length $M_{k,i}^{\hat{x}} + L_{k,i}^{\hat{x}}$ (respectively $M_{k,i}^{\hat{u}} + L_{k,i}^{\hat{u}}$ and $M_{k,i,j}^{\hat{w}} + L_{k,i,j}^{\hat{w}}$) of the fixed-point number $\hat{x}$ (respectively $\hat{u}$ and $\hat{w}$) must be less than or equal to $T - 1$ to ensure the data type required. We use $T - 1$ in these three constraints because we keep one bit for the sign. Equation (21) is about the multiplication. It asserts that the total number of bits of $\hat{x}$ and $\hat{w}$ is not exceeding the data type $T - 1$. Equations (22), (23) and (24) assert that the number of significant bits of the fractional parts cannot be negative. The boundary condition for the neurons of the output layer is represented in Equation (25). It gives a lower bound for $L_{m,i}^{\hat{u}}$ and then ensures that the error threshold is satisfied for all the neurons of the output layer.

$$M_{k,i}^{\hat{x}} + L_{k,i}^{\hat{x}} \leq T - 1, \qquad 0 \leq k \leq m,\ 0 \leq i < n \tag{18}$$

$$M_{k,i}^{\hat{u}} + L_{k,i}^{\hat{u}} \leq T - 1, \qquad 1 \leq k < m + 1,\ 0 \leq i < n \tag{19}$$

$$M_{k,i,j}^{\hat{w}} + L_{k,i,j}^{\hat{w}} \leq T - 1, \qquad 0 \leq k < m,\ 0 \leq i, j < n \tag{20}$$

$$M_{k,i,j}^{\hat{w}} + L_{k,i,j}^{\hat{w}} + M_{k,j}^{\hat{x}} + L_{k,j}^{\hat{x}} \leq T - 1, \qquad 0 \leq k < m,\ 0 \leq i, j < n \tag{21}$$

$$L_{k,i}^{\hat{x}} \geq 0, \qquad 0 \leq k \leq m,\ 0 \leq i < n \tag{22}$$

$$L_{k,i}^{\hat{u}} \geq 0, \qquad 1 \leq k < m + 1,\ 0 \leq i < n \tag{23}$$

$$L_{k,i,j}^{\hat{w}} \geq 0, \qquad 0 \leq k < m,\ 0 \leq i, j < n \tag{24}$$

$$L_{m,i}^{\hat{u}} \geq |\mathrm{ufp}(|Threshold|)|, \quad 0 \leq i < n,\ m : \text{last layer of NN} \tag{25}$$

$$\forall j \,:\, L_{k,i}^{\hat{u}} \leq L_{k-1,j}^{\hat{x}}, \quad 1 \leq k < m + 1,\ 0 \leq i, j < n \tag{26}$$

$$L_{k,i}^{\hat{x}} \leq L_{k,i}^{\hat{u}}, \quad 1 \leq k < m + 1,\ 0 \leq i < n \tag{27}$$

$$L_{k,i}^{\hat{u}} \times (\mathrm{ufp}(n) + 1) + \sum_{j=0}^{n-1}(L_{k-1,j}^{\hat{x}} + L_{k-1,i,j}^{\hat{w}}) \geq \sum_{j=0}^{n-1}(M_{k-1,j}^{\hat{x}} + M_{k-1,i,j}^{\hat{w}}) - \mathrm{ufp}(|Threshold|) - 1,$$
$$1 \leq k < m + 1,\ 0 \leq i, j < n \tag{28}$$

**Fig. 7.** Constraints generated for the formats optimization of each neuron of the NN.

In Figure 7, Equation (26) represents the constraint where the propagation is done in a forward way, and Equation (27) represents the constraint where the propagation is done in a backward way. These constraints bound the length of the fractional parts in the worst case. The constraint of Equation (26) aims at giving an upper bound of $L_{k,i}^{\hat{u}}$. It ensures that $L_{k,i}^{\hat{u}}$ of the output of the neuron $i$ of the layer $k$ is less than (or equal to) all its inputs $L_{k-1,j}^{\hat{x}}$, $0 \leq j < n$. The constraint of Equation (27) gives an upper bound for $L_{k,i}^{\hat{x}}$ of the input $\hat{x}$. This constraint ensures that the number of significant bits after the binary point $L_{k,i}^{\hat{x}}$ of the input of the neuron $i$ of the layer $k + 1$ is equal to (or less than) $L_{k,i}^{\hat{u}}$ of

the neuron $i$ of the previous layer $k$. The constraint of Equation (28) in Figure 7 aims at bounding $L_{k,i}^{\hat{u}}$ of the output of the neuron $i$ for the layer $k$ and $L_{k-1,i,j}^{\hat{w}}$ of the coefficients of matrix $\hat{W}_{k-1}$. This constraint corresponds to the error done during the computation of the affine transformation in Equation (11). The Equation (28) is obtained by the linearization of Equation (12) of Proposition 3, in other words, we have to compute the ufp of the error. The ufp of the error, written $\text{ufp}(\theta_{k,i})$, is computed as follow

Using Equation (17) of Proposition 3, we have

$$\text{ufp}(\theta_{k,i}) \leq \text{ufp}(\sum_{j=0}^{n-1}(2^{M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}}} + 2^{M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}}) + n \times 2^{-L_{k,i}^{\hat{u}}} + 2^{-L_{k,i}^{\hat{u}}+1}),$$

then we obtain

$$\text{ufp}(\theta_{k,i}) \leq \sum_{j=0}^{n-1}(M_{k-1,j}^{\hat{x}} - L_{k-1,i,j}^{\hat{w}} + M_{k-1,i,j}^{\hat{w}} - L_{k-1,j}^{\hat{x}}) - L_{k,i}^{\hat{u}} \times (\text{ufp}(n)+1) + 1.$$

We notice that $\text{ufp}(\theta_{k,i}) \leq \text{ufp}(|Threshold|) \leq 0$ because the error is between 0 and 1.

Finally, $L_{k,i}^{\hat{u}} \times (\text{ufp}(n)+1) + \sum_{j=0}^{n-1}(L_{k-1,j}^{\hat{x}} + L_{k-1,i,j}^{\hat{w}}) \geq \sum_{j=0}^{n-1}(M_{k-1,j}^{\hat{x}} + M_{k-1,i,j}^{\hat{w}}) - \text{ufp}(|Threshold|) - 1.$ ∎

All the constraints defined in Figure 7 are linear with integer variables. The optimal solution is found by solving them by linear programming. This solution gives the minimal number of bits for the fractional part required for each neuron $\hat{u}_{k,i}$ of each layer taking into account the data type $T$ and the error threshold tolerated by the user in one hand, and on the other hand the minimal number of bits of the fractional part required for each coefficient $\hat{w}_{k-1,i,j}$.

## 7 Implementation

In this section, we present our tool. Our approach which is computing the optimal formats $< M_{k,i}^{\hat{u}}, L_{k,i}^{\hat{u}} >$ for each neuron $\hat{u}_{k,i}$ of each layer for a given NN, satisfying an error threshold between 0 and 1 and a data type $T$ given by the user is evaluated through this tool.

Our tool is a fixed-point code synthesis tool. It synthesizes a C code for a given NN. This code contains arithmetic operations and activation functions, which use the fixed-point arithmetic (integer arithmetic) only. In this article, we present only the $Re\hat{L}U$ and $Lin\hat{e}ar$ activation functions (defined in Equation (5) and (6) respectively) but we can also deal with $Sig\hat{m}oid$ and $T\hat{a}nh$ activation functions in our current implementation. They are not shown but they are available in our framework. We have chosen to approximate them through piecewise linear approximation [6] using fixed-point arithmetic. We compute the corresponding error like in $Re\hat{L}U$ and $Lin\hat{e}ar$, then we generate the corresponding constraints.

A description of our tool is given in Figure 8. It takes a floating-point NN working at some precision, input vectors and a threshold error chosen by the user. The user also has the possibility to choose the data type $T \in \{8, 16, 32\}$ bits wanted for the code synthesis. First, we do a preliminary analysis of the NN through many input vectors in order to determine the range of the outputs of the neurons for each layer in the worst case. We compute also the most significant bit of each neuron of each layer in the worst case which gives us the range of the inputs of the NN $[x_{min}, x_{max}]$ for which we certify that our synthesized code is valid for any input belong this range respecting the required data type and the threshold. Our tool generates automatically the constraints mentioned in Figure 7 in Section 6 and solves them using the linprog function of scipy library [30] in Python.
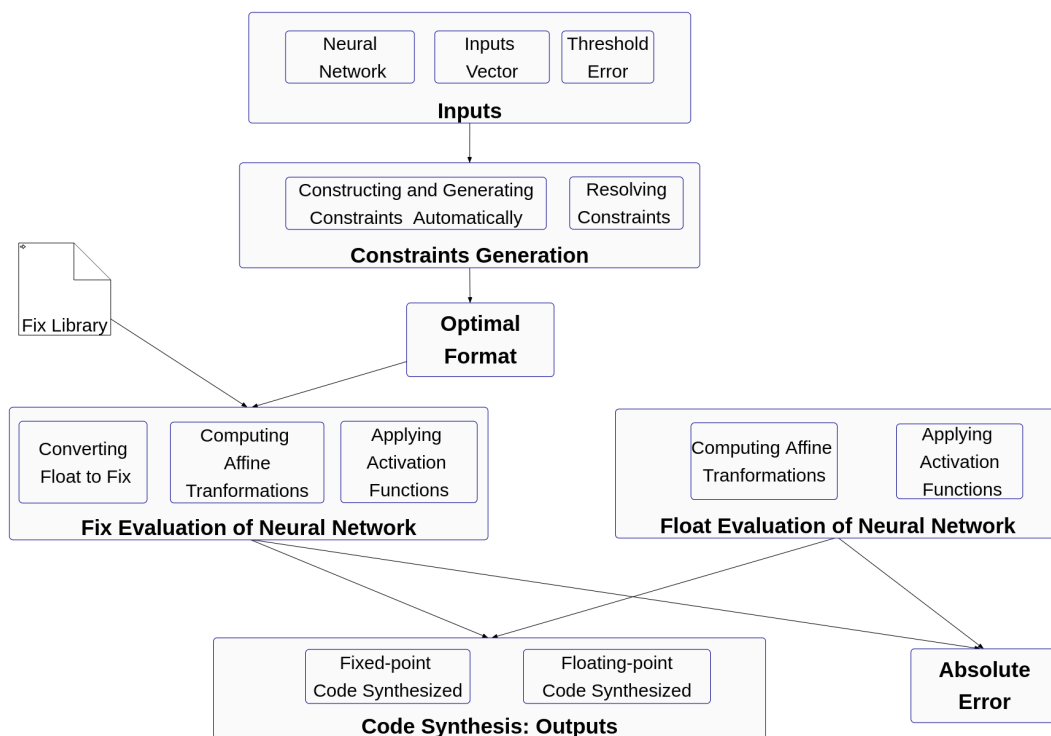
**Fig. 8.** Our tool features.

Then, the optimal formats for each output neuron, input neuron and coefficients of biases and matrices of weights are obtained. The optimal formats are used for the conversion from the floating-point into fixed-point numbers for all the coefficients inside each neuron. To make an evaluation of the NN in fixed-point arithmetic i.e computing the function in Equation (11), a fixed-point library is needed. Our library contains mainly the following functions: fixed-point addition and multiplication, shifts, fixed-point activation functions $\hat{Tanh}$, $\hat{Sigmoid}$, $\hat{ReLU}$ and $\hat{Linear}$. The conversion of a fixed-point number to a floating-point number and the conversion of a floating-point number to a fixed-point number is also available in this library. The last step consists of the fixed-point code synthesis. We also synthesize a floating-point code to make a comparison with the fixed-point synthesized code. We show experiments and results of some NNs in Section 8, and we compare them with floating-point results in terms of memory space (bits saved) and accuracy.

## 8   Experiments

In this section, we show some experimental results done on seven trained NNs. Four of them are interpolators which compute mathematical functions and the three others are classifiers. These NNs are described in Table 2. The first column gives the name of the NNs. The second column gives the number of layers and the third one shows the number of neurons. The number of connections between the neurons is shown in the fourth column and the number of generated constraints for each NN by our approach is given in the last column.

The NN *hyper* in Table 2 is an interpolator network computing the hyperbolic sine of the point $(x, y)$. It is made of four layers, 48 fully connected neurons (576 connections.)

**Table 2.** Description of our experimental NNs.

| Desc.<br>NN | Layers | Neurons | Connections | Constraints |
|:---:|:---:|:---:|:---:|:---:|
| ***hyper*** | 4 | 48 | 576 | 1980 |
| ***bumps*** | 2 | 60 | 1800 | 5010 |
| ***CosFun*** | 4 | 40 | 400 | 1430 |
| ***Iris*** | 3 | 33 | 363 | 1243 |
| ***Wine*** | 2 | 52 | 1352 | 3822 |
| ***Cancer*** | 3 | 150 | 7500 | 21250 |
| ***AFun*** | 2 | 200 | 10000 | 51700 |

The number of constraints generated for this NN in order to compute the optimal formats is 1980. The NN *bumps* is an interpolator network computing the *bump* function. The affine function $f(x) = 4 \times x + \frac{3}{2}$ is computed by the *AFun* NN (interpolator) and the function $f(x, y) = x \times cos(y)$ is computed by the *CosFun* NN (interpolator.) The classifier *Iris* is a NN which classifies the *Iris* plant [27] into three classes: *Iris-Setosa*, *Iris-Versicolour* and *Iris-Virginica*. It takes four numerical attributes as input (sepal length in cm, sepal width in cm, petal length in cm and petal width in cm.) The NN *Wine* is also a classifier. It classifies wine into three classes [2] through thirteen numerical attributes (alcohol, malic acid, ash, etc.) The last one is the *Cancer* NN which classifies the cancer into two categories (malignant and benign) through thirty numerical attributes [31] as input. These NNs originally work in IEEE754 single precision. We have transformed them into fixed-point NNs satisfying a threshold error and a data type $T$ (the size of the fixed-point numbers in the synthesized code) set by the user. Then we apply the $Re\hat{L}U$ activation function defined in Equation (5) or the $Lin\hat{e}ar$ activation function defined in Equation (6) (or $Sig\hat{m}oid$ and $T\hat{a}nh$ also.)

## 8.1 Accuracy and Error Threshold

The first part of experiments is for accuracy and error threshold. It concerns Table 3, Figure 9 and Figure 11 and shows if the concerned NNs satisfy the error threshold set by the user using the data type $T$. If the NN is an interpolator, it means that the output of the mathematical function $f$ has an error less than or equal to the threshold. If the NN is a classifier, it means that the error of classification of the NN is less than or equal to $(threshold \times 100)\%$.

The symbol $\times$ in Table 3 refers to the infeasability of the solution when the linear programming [30] fails to find a solution or when we cannot satisfy the threshold using the data type $T$. The symbol $\sqrt{}$ means that our linear solver has found a solution to the system of constraints (Section 6.) Each line of Table 3 corresponds to a given NN in some precision and the columns correspond to the multiple values of the error thresholds. For example, in the first line of the first column, the NN *hyper_32* requires a data type $T = 32$ bits and satisfies all the values of threshold (till $10^{-6} \approx 2^{-20}$.) In the fifth column, $2^{-4}$ means that we require at least four significant bits in the fractional part of the worst output of the last layer of the NNs. The fixed-point NNs *bumps_32, AFun_32, CosFun_32, Iris_32* and *Cancer_32* fulfill the threshold value $2^{-10}$ which corresponds to ten accurate bits in the fractional part of the worst output. Beyond this value, the linear programming [30] does not find a solution using a data type on 32 bits for these NNs. Using the data type $T = 16$ bits, all the NNs except *AFun_16* have an error less than or equal to $2^{-4}$ and ensure the correctness at least of four bits after the binary point of the worst output in

the last layer. Only the NNs *Wine_8* and *Cancer_8* can ensure one significant bit after the binary point using data type on 8 bits. In the other NNs, only the integer part is correct.

The results can vary depending on several parameters: the input vector, the coefficients of $W$ and $b$, the activation functions, the error threshold and the data type $T$. Generally, when the coefficients are between $-1$ and $1$, the results are more accurate because their ufp (Equation(4)) are negative and we can go far after the binary point. The infeasability of the solutions depends also on the size of the data types $T$, for example if we have a small data type $T$ and a consequent number of bits before the binary point in the coefficients (large value of $M$), we cannot have enough bits after the binary point to satisfy the small error thresholds.

Figure 9 represents the fixed-point outputs of the interpolator *CosFun_32* (lines) and the floating-point outputs (shape) for multiple inputs. All the fixed-point outputs must respect the threshold $2^{-10}$ (ten significant bits in the fractional part) and must use the data type $T = 32$ bits in this case. We can see that the two curves are close. This means that the new NN (fixed-point) has the same behavior and answer comparing to the original NN. The result is correct for this NN for any inputs $x \in [-4, 4]$ and $y \in [-4, 4]$.

**Table 3.** Comparison between the multiple values of error thresholds set by the user and our tool experimental errors using data types on 32, 16 and 8 bits for the fixed-point synthesized code.

| Data Types | Threshold<br>NN | $10^0 = 2^0$ | $0.5 = 2^{-1}$ | $10^{-1} \approx 2^{-4}$ | $10^{-2} \approx 2^{-7}$ | $10^{-3} \approx 2^{-10}$ | $10^{-4} \approx 2^{-14}$ | $10^{-5} \approx 2^{-17}$ | $10^{-6} \approx 2^{-20}$ |
|---|---|---|---|---|---|---|---|---|---|
| **32 bits** | *hyper_32* | √ | √ | √ | √ | √ | √ | √ | √ |
| | *bumps_32* | √ | √ | √ | √ | √ | × | × | × |
| | *AFun_32* | √ | √ | √ | √ | √ | × | × | × |
| | *CosFun_32* | √ | √ | √ | √ | √ | × | × | × |
| | *Iris_32* | √ | √ | √ | √ | √ | × | × | × |
| | *Wine_32* | √ | √ | √ | √ | √ | √ | × | × |
| | *Cancer_32* | √ | √ | √ | √ | √ | × | × | × |
| **16 bits** | *hyper_16* | √ | √ | √ | × | × | × | × | × |
| | *bumps_16* | √ | √ | √ | × | × | × | × | × |
| | *AFun_16* | √ | √ | × | × | × | × | × | × |
| | *CosFun_16* | √ | √ | √ | × | × | × | × | × |
| | *Iris_16* | √ | √ | √ | × | × | × | × | × |
| | *Wine_16* | √ | √ | √ | × | × | × | × | × |
| | *Cancer_16* | √ | √ | √ | × | × | × | × | × |
| **8 bits** | *hyper_8* | × | × | × | × | × | × | × | × |
| | *bumps_8* | √ | × | × | × | × | × | × | × |
| | *AFun_8* | × | × | × | × | × | × | × | × |
| | *CosFun_8* | × | × | × | × | × | × | × | × |
| | *Iris_8* | √ | × | × | × | × | × | × | × |
| | *Wine_8* | √ | √ | × | × | × | × | × | × |
| | *Cancer_8* | √ | √ | × | × | × | × | × | × |

Figure 11 shows the results of the fixed-point classifications for the NNs *Iris_32* (right) and *Wine_32* (left) using a data type $T = 32$ bits and a threshold value $2^{-7}$ for multiple input vectors ($= 8$). For example, the output corresponding to the input vectors 1 and 2 of the *Iris_32* NN is *Iris-Versicolour*, for the input vectors 3, 5 and 6 *Iris-Virginica* and *Iris-Setosa* for the others. The results are interpreted in the same way for the *Wine_32* NN. We notice that we have obtained the same classifications with the floating-point NNs using the same input vectors, so our new NN has the same behavior as the initial NN.
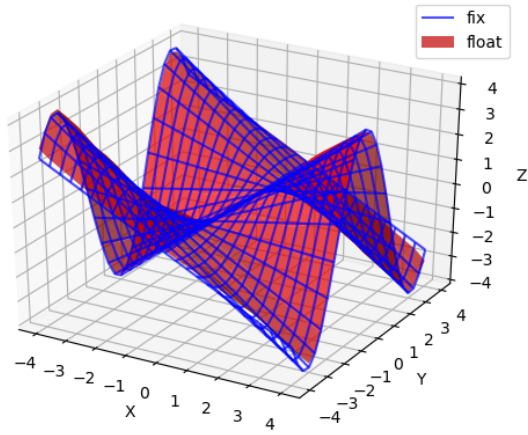
**Fig. 9.** Fixed-point outputs vs floating-point outputs of the *CosFun_32* NN for multiple inputs using a data type on 32 bits and an error threshold $\approx 2^{-10}$.
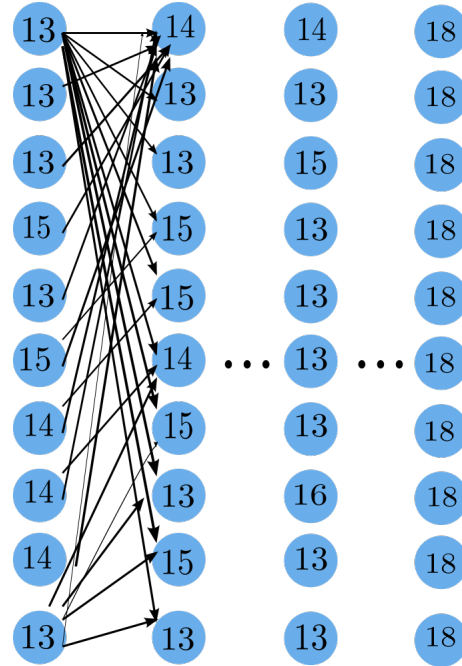


**Fig. 10.** Number of bits of each neuron of the *CosFun_32* NN after formats optimization for a threshold $\approx 2^{-10}$.
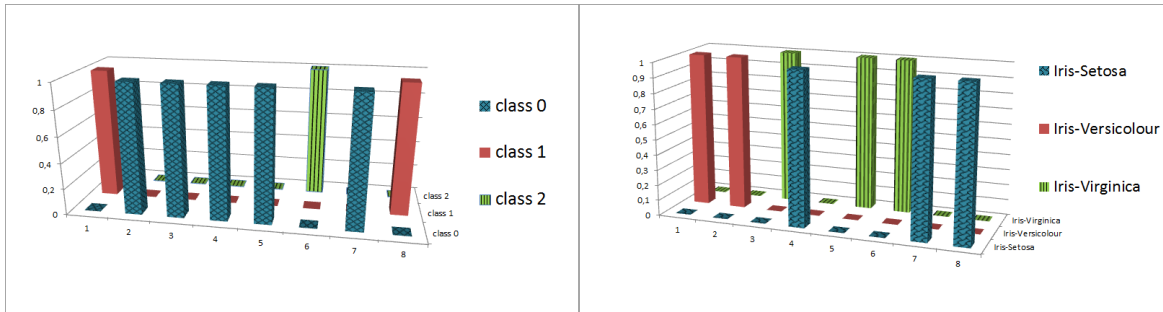


**Fig. 11.** Results of the fixed-point classification of the NNs *Iris_32* (right) and *Wine_32* (left) respecting the threshold value $2^{-7}$ and the data type $T = 32$ bits.

## 8.2   Bits/Bytes Saved

The second part of experiments concerns Figure 10, Figure 12 and Table 4 and aims at showing that our approach saves bytes/bits through the computation of the optimal format for each neuron done in Section 6. At the beginning, all the neurons are represented in $T \in \{8, 16, 32\}$ bits and after the optimization step, we reduce consequently the number of bits for each neuron while respecting the threshold set by the user.

Figure 10 shows the total number of bits of all the neurons for each layer of the *CosFun_32* NN after the optimization of the formats $< M, L >$ in order to satisfy the threshold $2^{-10}$ and the data type $T = 32$ bits for the fixed-point synthesized code in this case. In the synthesized C code, the data types will not change through this optimization because they are defined at the beginning of the program statically, but it is interesting to use these results in FPGA [4,11] for example. We notice that the size of all the neurons was 32 bits at the beginning and after our tool optimization (resolving the constraints of

Section 6), we need only 18 bits for the neurons of the output layer to satisfy the error threshold. We win 14 bits which represents a gain of 43.75%.
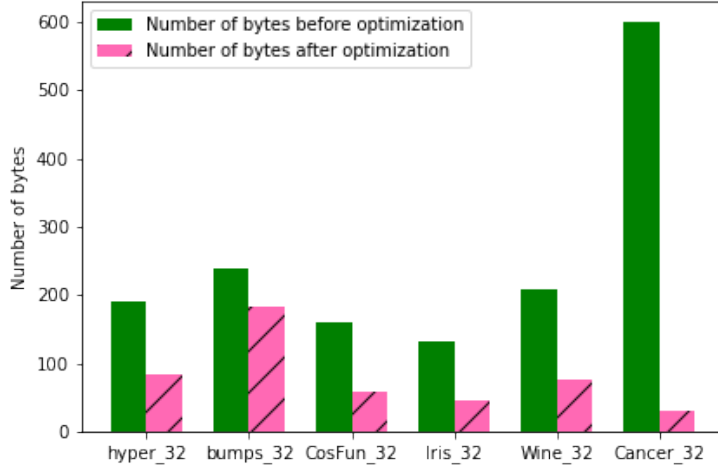


**Fig. 12.** Initial vs saved bytes for our experimental NNs on 32 bits respecting the threshold $2^{-7}$.

In the initial NNs, all the neurons were represented on 32 bits (4 bytes) but after the formats optimization (Section 6) through linear programming [30], we save many bytes for each neuron of each layer and the size of the NN becomes considerably small. It is useful in the case when we use FPGA [4,11], for example. Figure 12 shows the size of each NN (bytes) before and after optimization, and the Table 4 demonstrates the percentage of gain for each NN. For example, in the NN *Cancer_32*, we reduce for $18\times$ the number of bytes comparing to the initial NN (we earn up $94, 66\%$.) Our approach saves bits and takes in consideration the threshold error set by the user (in this example it is $2^{-7}$.)

**Table 4.** Gain of bytes of the experimental NNs after formats optimization for a threshold $\approx 2^{-7}$ and a data type $T = 32$ bits.

| NN \ Desc. | Size before opt. | Size after opt. | Bytes saved | Gain (%) |
|---|---|---|---|---|
| *hyper_32* | 192 | 84 | 108 | **56, 25** |
| *bumps_32* | 240 | 183 | 57 | **23.75** |
| *CosFun_32* | 160 | 59 | 101 | **63.12** |
| *Iris_32* | 132 | 47 | 85 | **64.39** |
| *Wine_32* | 208 | 77 | 131 | **62.98** |
| *Cancer_32* | 600 | 32 | 568 | **94.66** |

### 8.3    Conclusion

These experimental results show the efficiency of our approach in terms of accuracy and bits saved (memory space.) As we can see in Figure 2, the synthesized code contains only assignments, elementary operations $(+, \times, >>, <<)$ and conditions used in the activation functions. The execution time corresponding to the synthesized code for the experimental NNs of the Table 2 is in only few milliseconds (ms).

# 9   Related Work

Recently, a new line of research has emerged on compressing machine learning models [15, 16], using other arithmetics in order to run NNs on devices with small memories, integer CPUs [11, 12] and optimizing data types and computations error [14, 18].

In this section, we give an overview of some recent work. We present the multiple tools and frameworks (SEEDOT [11], DEEPSZ [15], Condensa [16]) more or less related to our approach. There is no approach comparable with our method because none of them respects a threshold error set by the user in order to synthesize a C code using only integers for a given trained NN without modifying its behavior. We can cite also FxpNet [7] and Fix-Net [9] which train neural networks using fixed point arithmetic (low bit-width arithmetic) in both forward pass and backward pass. The articles [9, 19] are about quantization which aims to reduce the complexity of DNNs and facilitate potential deployment on embedded hardware. There is also another line of research who has emerged recently on understanding safety and robustness of NNs [8, 10, 17, 25, 28]. We can mention the frameworks Sherlock [8], AI² [10], DeepPoly [25] and NNV [28].

The SEEDOT framework [11] synthesizes a fixed-point code for machine learning (ML) inference algorithms that can run on constrained hardware. This tool presents a compiling strategy that reduces the search space for some key parameters , especially scale parameters for the fixed-point numbers representation used in the synthesized fixed-point code. Some operations are implemented (multiplication, addition, exponential, argmax, etc.) in this approach. Both of SEEDOT and our tool generate fixed-point code, but our tool fullfills a threshold and a data type required by the user. SEEDOT finds a scale for the fixed-point representation number and our tool solves linear constraints for finding the optimal format for each neuron. The main goal of this compiler is the optimization of the fixed-point arithmetic numbers and operations for an FPGA and micro-controllers.

The key idea of [14] is to reduce the sizes of data types used to compute inside each neuron of the network (one type per neuron) working in IEEE754 floating-point arithmetic [13, 21]. The new NN with smaller data types behaves almost like the original NN with a percentage error tolerated. This approach generates constraints and does a forward and a backward analysis to bound each data type. Our tool has a common step with this approach, which is the generation of constraints for finding the optimal format for each neuron (fixed-point arithmetic) for us and the optimal size (floating-point arithmetic) for each neuron for this method.

In [12], a new data type called Float-Fix is proposed. This new data type is a trade-off between the fixed-point arithmetic [4, 20, 23] and the floating-point arithmetic [13, 21]. This approach analyzes the data distribution and data precision in NNs then applies this new data type in order to fulfill the requirements. The elementary operations are designed for Float-Fix data type and tested in the hardware. The common step with our approach is the analysis of the NN and the range of its output in order to find the optimal format using the fixed-point arithmetic for us and the the optimal precision for this method using Float-Fix data type. Our approach takes a threshold error not to exceed but this approach does not.

DEEPSZ [15] is a lossy compression framework. It compresses sparse weights in deep NNs using the floating-point arithmetic. DEEPSZ involves four key steps: network pruning, error bound assessment, optimization for error bound configuration and compressed model generation. A threshold is set for each fully connected layer, then the weights of this layer are pruned. Every weight below this threshold is removed. This framework determines the best-fit error bound for each layer in the network, maximizing the overall compression ratio with user acceptable loss of inference accuracy.

The idea presented in [16] is about using weight pruning and quantization for the compression of deep NNs [26]. The model size and the inference time are reduced without appreciable loss in accuracy. The tool introduced is Condensa where the reducing memory footprint is by zeroing out individual weights and reducing inference latency is by pruning 2-D blocks of non-zero weights for a language translation network (Transformer).

A framework for semi-automatic floating-point error analysis for the inference phase of deep learning is presented in [18]. It transforms a NN into a C++ code in order to analyze the network need for precision. The affine and interval arithmetics are used in order to compute the relative and absolute errors bounds for deep NN [26]. This article gives some theoretical results which are shown for bounding and interpreting the impact of rounding errors due to the precision choice for inference in generic deep NN [26].

## 10    Conclusion & Future Work

In this article, we introduced a new approach to synthesize a fixed-point code for NNs using the fixed-point arithmetic and to tune the formats of the computations and conversions done inside the neurons of the network. This method ensures that the new fixed-point NN still answers correctly compared to the original network based on IEEE754 floating-point arithmetic [13]. This approach ensures the non overflow (sufficient bits for the integer part) of the fixed-point numbers in one hand and the other hand, it respects the threshold required by the user (sufficient bits in the fractional part.) It takes in consideration the propagation of the round off errors and the error of inputs through a set of linear constraints among integers, which can be solved by linear programming [30]. Experimental results show the efficiency of our approach in terms of accuracy, errors of computations and bits saved. The limit of the current implementation is the large number of constraints. We use linprog in Python [30] to solve them but this method does not support a high number of constraints, this is why our experimental NNs are small.

A first perspective is about using another solver to solve our constraints (Z3 for example [22]) which deals with a large number of constraints. A second perspective is to make a comparison study between Z3 [22] and linprog [30] in term of time execution and memory consumption. A third perspective is to test our method on larger, real-size industrial neural networks. We believe that our method will scale up as long as the linear programming solver will scale up. If this is not enough, a solution would be to assign the same format to a group of neurons in order to reduce the number of equations and variables in the constraints system. A last perspective is to consider the other NN architectures like convolutional NNs [1, 3, 10].

## References

1. Abraham, A.: Artificial neural networks. Handbook of measuring system design (2005)
2. Aeberhard, S., Coomans, D., de Vel, O.: The classification performance of RDA. Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland, Tech. Rep pp. 92–01 (1992)
3. Albawi, S., Mohammed, T.A., Al-Zawi, S.: Understanding of a convolutional neural network. In: 2017 International Conference on Engineering and Technology (ICET). pp. 1–6. IEEE (2017)
4. Bečvář, M., Štukjunger, P.: Fixed-point arithmetic in FPGA. Acta Polytechnica **45**(2) (2005)
5. Catrina, O., de Hoogh, S.: Secure multiparty linear programming using fixed-point arithmetic. In: Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security. vol. 6345, pp. 134–150. Springer (2010)
6. Çetin, O., Temurtaş, F., Gülgönül, Ş.: An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function. Dicle Medical Journal/Dicle Tip Dergisi **42**(2) (2015)

7. Chen, X., Hu, X., Zhou, H., Xu, N.: Fxpnet: Training a deep convolutional neural network in fixed-point representation. In: 2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017. pp. 2494–2501. IEEE (2017)

8. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: NASA Formal Methods - 10th International Symposium, NFM. vol. 10811, pp. 121–138. Springer (2018)

9. Enderich, L., Timm, F., Rosenbaum, L., Burgard, W.: Fix-net: pure fixed-point representation of deep neural networks. ICLR (2019)

10. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy. pp. 3–18. IEEE Computer Society (2018)

11. Gopinath, S., Ghanathe, N., Seshadri, V., Sharma, R.: Compiling KB-sized machine learning models to tiny IoT devices. In: Programming Language Design and Implementation, PLDI 2019. pp. 79–95. ACM (2019)

12. Han, D., Zhou, S., Zhi, T., Wang, Y., Liu, S.: Float-fix: An efficient and hardware-friendly data type for deep neural network. Int. J. Parallel Program. **47**(3), 345–359 (2019)

13. IEEE: IEEE standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70 (2008)

14. Ioualalen, A., Martel, M.: Neural network precision tuning. In: Quantitative Evaluation of Systems, 16th International Conference, QEST. vol. 11785, pp. 129–143. Springer (2019)

15. Jin, S., Di, S., Liang, X., Tian, J., Tao, D., Cappello, F.: Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In: In International Symposium on High-Performance Parallel and Distributed Computing, HPDC. pp. 159–170. ACM (2019)

16. Joseph, V., Gopalakrishnan, G., Muralidharan, S., Garland, M., Garg, A.: A programmable approach to neural network compression. IEEE Micro **40**(5), 17–25 (2020)

17. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Computer Aided Verification, CAV. vol. 10426, pp. 97–117. Springer (2017)

18. Lauter, C.Q., Volkova, A.: A framework for semi-automatic precision and accuracy analysis for fast and rigorous deep learning. In: 27th IEEE Symposium on Computer Arithmetic, ARITH 2020. pp. 103–110. IEEE (2020)

19. Lin, D.D., Talathi, S.S., Annapureddy, V.S.: Fixed point quantization of deep convolutional networks. In: Balcan, M., Weinberger, K.Q. (eds.) Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. JMLR Workshop and Conference Proceedings, vol. 48, pp. 2849–2858. JMLR.org (2016)

20. Lopez, B.: Implémentation optimale de filtres linéaires en arithmétique virgule fixe. (Optimal implementation of linear filters in fixed-point arithmetic). Ph.D. thesis, Pierre and Marie Curie University, Paris, France (2014)

21. Martel, M.: Floating-point format inference in mixed-precision. In: NASA Formal Methods - 9th International Symposium, NFM. vol. 10227, pp. 230–246 (2017)

22. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)

23. Najahi, M.A.: Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks. Ph.D. thesis, University of Perpignan, France (2014)

24. Sharma, S., Sharma, S.: Activation functions in neural networks. Towards Data Science **6**(12), 310–316 (2017)

25. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang., POPL **3**, 41:1–41:30 (2019)

26. Sun, Y., Huang, X., Kroening, D., Sharp, J., Hill, M., Ashmore, R.: Testing deep neural networks. arXiv preprint arXiv:1803.04792 (2018)

27. Swain, M., Dash, S.K., Dash, S., Mohapatra, A.: An approach for iris plant classification using neural network. International Journal on Soft Computing **3**(1),  79 (2012)

28. Tran, H., Yang, X., Lopez, D.M., Musau, P., Nguyen, L.V., Xiang, W., Bak, S., Johnson, T.T.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: Computer Aided Verification - 32nd International Conference, CAV. vol. 12224, pp. 3–17. Springer (2020)

29. Urban, C., Miné, A.: A review of formal methods applied to machine learning. CoRR **abs/2104.02466** (2021)

30. Welke, P., Bauckhage, C.: Ml2r coding nuggets: Solving linear programming problems. Tech. rep., Technical Report. MLAI, University of Bonn (2020)

31. Wolberg, W.H., Street, W.N., Mangasarian, O.L.: Breast cancer wisconsin (diagnostic) data set [http://archive.(ics. uci. edu/ml/] (1992)
32. Yates, R.: Fixed-point arithmetic: An introduction. Digital Signal Labs **81**(83),  198 (2009)

## Authors

**H Benmaghnia** received a Master degree in High Performance Computing and Simulations from the University of Perpignan in France. She did a Bachelor of Informatic Systems at the University of Tlemcen in Algeria. Currently, she is pursuing her PhD in Computer Science at the University of Perpignan in LAboratoire de Modélisation Pluridisciplinaire et Simulations (LAMPS). Her research interests include Computer Arithmetic, Precision Tuning, Numerical Accuracy, Neural Networks and Formal Methods.

**M Martel** is a professor in Computer Science at the University of Perpignan in LAboratoire de Modélisation Pluridisciplinaire et Simulations (LAMPS), France. He is also co-founder and scientific advisor of the start-up Numalis, France. His research interests include Computer Arithmetic, Numerical Accuracy, Abstract Interpretation, Semantics-based Code Transformations & Synthesis, Validation of Embedded Systems, Safety of Neural Networks & Arithmetic Issues, Green & Frugal Computing, Precision Tuning & Scientific Data Compression.

**Y Seladji** received PhD degree in computer science from Ecole Polytechnique, France. Currently, she is associate professor in the University of Tlemcen. Her research interests include Formal Methods, Static Analysis, Embedded Systems.