Neural Network Precision Tuning[⋆]

Arnault Ioualalen[1] and Matthieu Martel[1,2]

[1] Numalis
Cap Omega, Rond-point Benjamin Franklin
34960 Montpellier, France
[2] Laboratoire de Mathématiques et Physique (LAMPS)
Université de Perpignan Via Domitia, France
ioualalen@numalis.com matthieu.martel@univ-perp.fr

Abstract. Minimizing the precision in which the neurons of a neural network compute is a desirable objective to limit the resources needed to execute it. This is specially important for neural networks used in embedded systems. Unfortunately, neural networks are very sensitive to the precision in which they have been trained and changing this precision generally degrades the quality of their answers. In this article, we introduce a new technique to tune the precision of neural networks in such a way that the optimized network computes in a lower precision without modifying the quality of the outputs of more than a percentage chosen by the user. From a technical point of view, we generate a system of linear constraints among integer variables that we can solve by linear programming. The solution to this system is the new precision of the neurons. We present experimental results obtained by using our method.

Keywords: Formal methods, floating-point arithmetic, static analysis, dynamic analysis, linear programming, numerical accuracy.

1 Introduction

Neural networks are more and more used in many domains, including critical embedded systems in aeronautics, space, defense, automotive, etc. These neural networks also become larger and larger while embedded systems still have limited resources, mainly in terms of computing power and memory. As a consequence, running large neural networks on embedded systems with limited resources introduces several new challenges. While recent work has focused on safety [7, 9, 22, 21, 12] and security properties [14, 24], a different problem is addressed in this article which concerns the accuracy of the computations. It is well-known that neural networks are sensitive to the precision of the computations, or, in other terms to the computer arithmetic used during their training and execution. Indeed, a neural network working correctly in some computer arithmetic (e.g.

[⋆] This work is supported by La Region Occitanie under Grant GRAINE - Syfi. https://www.laregion.fr

IEEE754 single precision [1]) may behave poorly if we run it in lower or even in higher precision (e.g. in IEEE754 half or double precision).

We consider the problem of tuning the precision of an already trained neural network, assumed to behave correctly at some precision, in such a way that, after tuning, the network behaves *almost* like the original one while performing its computations in lower precision. In this article, we focus on interpolator networks, i.e. in networks computing mathematical functions. In this case, we will say that the original and optimized networks behave almost identically if they compute functions f and $\hat{f}$ respectively, such that, for any input x, the relative error between the numerical results computed by both networks is less than some user defined constant δ:

$$\left| \frac{f(x) - \hat{f}(x)}{f(x)} \right| \leq \delta \ . \tag{1}$$

This definition should be adapted for classifier networks without impacting the rest of the techniques presented here. More precisely, in this case, we should compare the original and optimized networks with respect to a performance metric (recall, precision, F1-score, etc.)

Recently, a lot of work has been done concerning precision tuning of general programs (without direct connection to neural networks), based on static analysis [2, 5] or dynamic analysis [13, 18, 20]. In this article, we adapt the approach introduced in [15] to neural networks. We consider fully connected networks with `ReLU` or `tanh` functions (see Section 2.1). We always assume that these networks are already trained and work correctly in the sense that they have satisfying performances in terms of interpolation or classification. We assume that each neuron has its own precision for the computations. However, we assume that all the computations performed inside the same neuron (summation and activation function) use the same precision. Finally, we assume that the ranges of the inputs and outputs of each neuron are given. Several techniques have been developed recently to solve precisely this problem [7, 9], which is orthogonal to our. Currently, in our implementation, we compute these ranges by dynamic analysis even if we aim at implementing static analysis techniques in (near) future work. We generate a set of constraints describing the propagation of the errors throughout the neural network. The strength of our approach is that we only generate linear constraints among integers (and only integers). These constraints are easy to solve by standard tools. Optimizing the precision of the network under the correctness constraint of Equation (1) then becomes a linear programming problem. We demonstrate the efficiency of our technique by showing how the size of interpolator neural networks can be reduced in function of the parameter δ of Equation (1).

The rest of this article is organized as follows. Preliminary notions and notations are introduced in Section 2. They concern neural networks and computer arithmetic. The propagation of the roundoff errors throughout a neural network is modeled in Section 3. The generation of constraints is introduced in Section 4 and experimental results are given in Section 5. Section 6 concludes.

$$W_1 = \begin{pmatrix} 0.9 & 0.0 & 2.3 \\ 1.1 & -0.7 & 0.0 \\ 0.1 & -2.1 & 0.4 \end{pmatrix} \quad b_1 = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

$$W_2 = \begin{pmatrix} 0.0 & -0.3 & 1.1 \\ 1.0 & 0.2 & 0.0 \\ -0.4 & 0.4 & 1.1 \end{pmatrix} \quad b_2 = \begin{pmatrix} -0.1 \\ 0.0 \\ -0.1 \end{pmatrix}$$

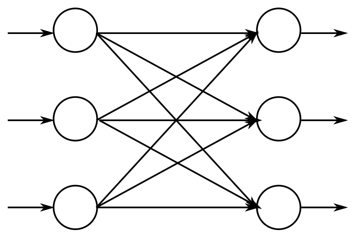

Fig. 1. Example of a fully-connected two-layer network with three neurons by layer.

2 Preliminary Definitions

In this section, we introduce preliminary notions and notations concerning neural networks and computer arithmetics. Section 2.1 is dedicated to neural networks while Section 2.2 focuses on the floating-point arithmetic. Finally, Section 2.3 introduces precision tuning.

2.1 Neural Networks

In this article, a neural network is defined by means of affine transformations defined by the grammar of Equation (2) in function of an input vector $\overline{x} \in \mathbb{R}^m$.

$$f(\overline{x}) ::= \mathtt{ReLU}(W \cdot \overline{x} + \overline{b}) \mid \mathtt{tanh}(W \cdot \overline{x} + \overline{b}) \mid f_1(f_2(\overline{x})) \tag{2}$$

The hyperbolic tangent is denoted `tanh` and a rectified linear unit (`ReLU`) activation function is defined by

$$\mathtt{ReLU}(\overline{x}) = \big(\max(0, x_1), \ldots, \max(0, \overline{x}_m)\big)^T \ . \tag{3}$$

Following Equation (2), an affine function is either an affine map $f \ : \ \mathbb{R}^m \rightarrow \mathbb{R}^n$ composed with a `ReLU` or `tanh` function or the composition of the former elements. In general, an affine function with `ReLU` or `tanh` $f \ : \ \mathbb{R}^m \rightarrow \mathbb{R}^n$ defines a fully connected layer of a neural network. The whole network is a sequence of ℓ layers, which corresponds to the composition of ℓ affine functions $f_1 \circ f_2 \ldots \circ f_\ell$.

An example of neural network is given in Figure 1. This fully connected neural network is made of two layers, each layer containing three neurons. The matrices W_1 and W_2 correspond to the first and second layers respectively and b_1 and b_2 are the second members of each layer. For example, the first neuron of the first layer computes $0.9\overline{x}_1 + 2.3\overline{x}_3 + 0.1$ in function of the entry $\overline{x} \in \mathbb{R}^3$.

Note that other operations, different from affine transformations and usually performed by some layers of other kinds of neural networks, such as convolutional layers or max pooling layers, can be reduced to affine transformations [9]. We may then omit them in our work without loss of generality.

Format	Name	p	e bits	e_{min}	e_{max}
Binary16	Half precision	11	5	-14	$+15$
Binary32	Single precision	24	8	-126	$+127$
Binary64	Double precision	53	11	-1122	$+1223$
Binary128	Quadruple precision	113	15	-16382	$+16383$

Fig. 2. Basic binary IEEE754 formats.

2.2 Computer Arithmetics

We introduce here some elements of floating-point arithmetic [1, 17]. First of all, a *floating-point number* x in base β is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{4}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the *significand*, $0 \le d_i < \beta$, $0 \le i \le p-1$, p is the *precision* and e is the exponent, $e_{min} \le e \le e_{max}$.

A floating-point number x is *normalized* whenever $d_0 \neq 0$. The IEEE754 Standard defines binary formats (with $\beta = 2$) and decimal formats (with $\beta = 10$). In this article, without loss of generality, we only consider normalized numbers and we always assume that $\beta = 2$ (which is the most common case in practice). The IEEE754 Standard also specifies a few values for p, e_{min} and e_{max} which are summarized in Figure 2. Finally, special values also are defined: nan (Not a Number) resulting from an invalid operation, $\pm\infty$ corresponding to overflows, and $+0$ and -0 (signed zeros).

The IEEE754 Standard also defines five rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero, to the nearest ties to even and to the nearest ties to away and we write them $\circ_{-\infty}$, $\circ_{+\infty}$, $\circ_0$, $\circ_{\sim_e}$ and $\circ_{\sim_a}$, respectively. The semantics of the elementary operations $\diamond \in \{+,\ -,\ \times,\ \div\}$ is then defined by

$$f_1 \ \diamond_\circ \ f_2 \ = \ \circ\,(f_1 \ \diamond \ f_2) \tag{5}$$

where $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_{\sim_e}, \circ_{\sim_a}\}$ denotes the rounding mode. Equation (5) states that the result of a floating-point operation $\diamond_\circ$ done with the rounding mode $\circ$ returns what we would obtain by performing the exact operation $\diamond$ and next rounding the result using $\circ$. The IEEE754 Standard also specifies how the square root function must be rounded in a similar way to Equation (5) but does not specify the roundoff of other functions like sin, log, etc.

We introduce hereafter two functions which compute the *u*nit in the *f*irst *p*lace and the *u*nit in the *l*ast *p*lace of a floating-point number. These functions are used further in this article to generate constraints encoding the way roundoff errors are propagated throughout computations. The ufp of a number x is

$$\mathsf{ufp}(x) = \min\left\{ i \in \mathbb{N} \ : \ 2^{i+1} > x \right\} = \lfloor \log_2(x) \rfloor \ . \tag{6}$$

The ulp of a floating-point number which significant has size p is defined by

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - p + 1 \ . \tag{7}$$

The ufp of a floating-point number corresponds to the binary exponent of its most significant digit. Conversely, the ulp of a floating-point number corresponds to the binary exponent of its least significant digit.

2.3 Precision Tuning

The method developed in this article aims at tuning the precision of neural networks. While this subject is new for neural networks, some work has been carried out recently in this domain for usual computer programs and, in this section, we introduce some background material about this domain. Precision tuning consists of finding the least floating-point formats enabling a program to compute some results with an accuracy requirement. Precision tuning allows compilers to select the most appropriate formats (for example IEEE754 [1] half, single, double or quadruple formats [1, 17]) for each variable. It is then possible to save memory, reduce CPU usage and use less bandwidth for communications whenever distributed applications are concerned. So, the choice of the best floating-point formats is an important compile-time optimization in many contexts. Precision tuning is also of great interest for the fixed-point arithmetic [11] for which it is important to determine data formats, for example in FPGAs [8, 16]. In mixed precision, i.e. when every variable or intermediary result may have its own format, possibly different from the format of the other variables, this problem leads to a combinatorial explosion.

Several approaches have been proposed to determine the best floating-point formats as a function of the expected accuracy on the results. Darulova and Kuncak use a forward static analysis to compute the propagation of errors [6]. If the computed bound on the accuracy satisfies the post-conditions then the analysis is run again with a smaller format until the best format is found. Note that in this approach, all the values have the same format (contrarily to our framework where each control-point has its own format). While Darulova and Kuncak develop their own static analysis, other static techniques [10, 23] could be used to infer from the forward error propagation the suitable formats. This approach has also been improved in [5]. Chiang *et al.* [2] have proposed a method to allocate a precision to the terms of an arithmetic expression (only). They use a formal analysis via Symbolic Taylor Expansions and error analysis based on interval functions. In spite of our linear constraints, they solve a quadratically constrained quadratic program to obtain annotations.

Other approaches rely on dynamic analysis. For instance, the Precimonious tool tries to decrease the precision of variables and checks whether the accuracy requirements are still fulfilled [18, 20]. Lam *et al* instrument binary codes in order to modify their precision without modifying the source codes [13]. They also propose a dynamic search method to identify the pieces of code where the precision should be modified.

Another related research direction concerns the compile-time optimization of programs in order to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [4, 19].

3 Roundoff Error Modelling

In this section, we introduce some theoretical results concerning the numerical errors done inside a neural network. The error on the output of an affine transformation function can be decomposed in two parts, the propagation of the errors on the input vector and the roundoff errors arising in the computation of the affine function itself. We show in Proposition 2 that the numerical error on the output of an affine transformation function can be expressed by $\max(p+\mu, q+\nu)+1$ where p is related to the precision of the input vector, q is the precision in which the affine transformation is computed and where μ and ν are constants depending only on the neural networks, i.e. on W and b.

Following Equation (2), a fully connected layer of a neural network computes an output vector $\overline{u} \in \mathbb{R}^n$ in function of an input vector $\overline{x} \in \mathbb{R}^m$ such that

$$\overline{u} = f(\overline{x}) = W \cdot \overline{x} + \overline{b} \ , \tag{8}$$

for some $n \times m$ matrix W and for some vector $\overline{b} \in \mathbb{R}^n$ (the case of `ReLU` and `tanh` functions will be discussed at the end of Section 4.) Proposition 1 states how to bound the numerical errors arising in Equation (8).

Proposition 1 *Let us consider a fully connected layer of a neural network as defined in Equation (8). Let p_i, $1 \le i \le n$, denote the precision of the i^{th} neuron of the layer. Let $\hat{x}$ be an approximated input and $\overline{e}$ some absolute error bound on the input, i.e. the exact input $\overline{x}$ satisfies $|\overline{x} - \hat{x}| \le \overline{e}$. Then the networks computes the output $f(\hat{x})$ and, for all i, $1 \le i \le n$, the absolute error $\overline{err}_i$ on the i^{th} component of the output $\overline{err} = |f(\overline{x}) - f(\hat{x})|$ on this output is bound by*

$$\overline{err}_i \le \sum_{j=1}^{m} W_{ij} \cdot \overline{e}_j + 2^{-p_i} \cdot \left(\overline{b}_i + (m+1) \cdot \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| \right) \ . \tag{9}$$

Proof. Using the notations of Equation (8), we have

$$\overline{u}_i = \sum_{j=1}^{m} W_{ij} \cdot \overline{x}_j + \overline{b}_i, \quad 1 \le i \le n \ . \tag{10}$$

Then the error $\overline{err}$ on the output is

$$\overline{err} = W \cdot \overline{e} + \overline{c} \tag{11}$$

where $W \cdot \overline{e}$ is the propagation of the initial error on the input and $\overline{c}$ the error introduced by the computation of $\overline{u} = f(\hat{x})$ in machine. We need to bound $\overline{c}$. Explicitely,

$$\overline{u}_i = \hat{f}_i(\hat{x}) = W_{i1} \cdot \hat{x}_1 + W_{i2} \cdot \hat{x}_2 + \ldots + W_{im} \cdot \hat{x}_m + \overline{b}_i \ . \tag{12}$$

First, the errors due to products in Equation (12) are bound by

$$\begin{aligned} err_\times(\overline{u}_i) &\le |W_{i1} \cdot \hat{x}_1| \cdot 2^{-p_i} + |W_{i2} \cdot \hat{x}_2| \cdot 2^{-p_i} + \ldots + |W_{im} \cdot \hat{x}_m| \cdot 2^{-p_i} & (13) \\ &= \big(|W_{i1} \cdot \hat{x}_1| + |W_{i2} \cdot \hat{x}_2| + \ldots + |W_{im} \cdot \hat{x}_m|\big) \cdot 2^{-p_i} \ . & (14) \end{aligned}$$

Then the errors due to additions are bound by

$$err_+(\overline{u}_i) \leq 2^{-p_i} \cdot (m-1) \cdot \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| + 2^{-p_i} \cdot (\overline{b}_i + \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j|) \quad (15)$$

and, consequently,

$$err(\overline{u}_i) = err_\times(\overline{u}_i) + err_+(\overline{u}_i) \quad (16)$$

$$\leq 2^{-p_i} \cdot m \cdot \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| + 2^{-p_i} \cdot \left(\hat{x}_i + \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| \right) \quad . \quad (17)$$

Finally, by combining equations (11) and (17), we bound $\overline{err}$ the global error vector on the output $\overline{u}$ by

$$\overline{err}_i \leq \sum_{j=1}^{m} W_{ij} \cdot \overline{e}_j + 2^{-p_i} \cdot \left(\overline{b}_i + (m+1) \cdot \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| \right) \quad . \quad (18)$$

□

The next step consists of linearizing the equations in order to make them easier to solve by a solver.

Proposition 2 *Let us consider a fully connected layer of a neural network as defined in Equation (8). Let $\overline{p}_i$, $1 \leq i \leq n$, denote the precision of the i^{th} neuron of the layer. Let $\hat{x}$ be an approximated input of precision $\overline{q}$, i.e. the absolute error $\overline{e}$ on the input, is bound by $|\overline{x}_i - \hat{x}_i| \leq \overline{e}_i < 2^{-q_i}$, $\forall i,\ 1 \leq i \leq n$. Then the accuracy of the i^{th} output $\overline{u}_i = f_i(\hat{x})$ is $\max(\mu_i + p_i, \nu_i + q_i) + 1$ with μ_i and ν_i two constants defined by*

$$\mu_i = \mathbf{ufp}\left(\overline{b}_i + (m+1) \cdot \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| \right) \quad \nu_i = \mathbf{ufp}\left(\left| \sum_{j=1}^{m} W_{ij} \right| \right) \quad \forall i,\ 1 \leq i \leq n \ .$$

Proof. Let us write

$$\overline{\alpha}_i = \overline{b}_i + (m+1) \cdot \sum_{j=1}^{m} |W_{ij} \cdot \hat{x}_j| \quad (19)$$

Indeed, the vector $\overline{\alpha}$ is constant. Let $\overline{e}$ be the data error vector introduced in Equation (11) and let q be the accuracy of the input, we have

$$q_i = \min\ \left\{ r \in \mathbb{N}\ :\ \overline{e}_i \leq 2^r \right\} \quad (20)$$

Let

$$\overline{\beta}_i = \left| \sum_{j=1}^{m} W_{ij} \right| \quad (21)$$

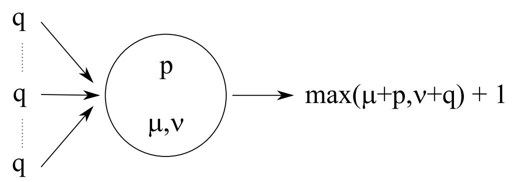

Fig. 3. Accuracy propagation throughout a neuron as defined in Proposition 2.

Again, the $\overline{\beta}_i$, $1 \leq i \leq n$ are constants. Then

$$\left| \sum_{j=1}^{m} W_{ij} \cdot \overline{e}_j \right| \leq \overline{\beta}_i \cdot 2^{q_i} \tag{22}$$

Consequently,

$$\overline{err} \leq \overline{\alpha} \cdot 2^{p} + \overline{\beta} \cdot 2^{q} \tag{23}$$

Let $\overline{\mu} = \mathbf{ufp}(\overline{\alpha})$ and $\overline{\nu} = \mathbf{ufp}(\overline{\beta})$. Equation (23) becomes

$$\forall i,\ 1 \leq i \leq n,\ \overline{err}_i \leq 2^{\overline{\mu}_i} \cdot 2^{\overline{p}_i} + 2^{\overline{\nu}_i} \cdot 2^{\overline{q}_i} = 2^{\overline{\mu}_i + \overline{p}_i} + 2^{\overline{\nu}_i + \overline{q}_i} \tag{24}$$

$$\leq 2^{\max(\overline{\mu}_i + \overline{p}_i, \overline{\nu}_i + \overline{q}_i) + 1} \tag{25}$$

□

The way Proposition 2 defines the propagation of a neuron is summarized in Figure 3.

4 Constraint Generation

In this section, we describe our algorithm to tune the precision of a neural network. We assume that the input network correctly computes a function $f(\overline{x})$. When we decrease this precision, the network computes a new function $\hat{f}(\overline{x})$. Then we aim at finding the smallest precision such that the relative error

$$\left| \frac{f(\overline{x}) - \hat{f}(\overline{x})}{f(\overline{x})} \right| < \delta \quad , \tag{26}$$

for a given tolerance δ specified by the user.

We generate the constraints of Figure 4, explained hereafter. Let us consider a neural network made of ℓ layers of n fully connected neurons. The variables of the constraint system are $\texttt{Prec}(W[k,i])$, for $0 \leq i < n$, $0 \leq k < \ell$ and $\texttt{Prec}(X[k,i])$, for $0 \leq i < n$, $0 \leq k < \ell + 1$. They correspond respectively to the accuracy used to compute inside the neurons and the accuracy of the output of each neuron. Next the constraints depend on values computed *a priori* in

$$\forall 0 \leq i < n,\ \forall 0 \leq k < \ell,\ 0 < \texttt{Prec}(X[k,i]) \leq \texttt{ComputedPrec}(X[k,i]) \quad \text{(PB)}$$

$$\forall 0 \leq i < n,\ \forall 0 \leq k < \ell,\ 0 < \texttt{Prec}(W[k,i]) \leq \texttt{InitialPrec}(W[k,i]) \quad \text{(IP)}$$

$$\forall 0 \leq i < n,\ \texttt{Prec}(\texttt{Output}[i]) \leq \texttt{Prec}(X[\ell,i]) \quad \text{(PC)}$$

$$\forall 0 \leq i < n,\ \forall 0 \leq k < \ell,\ \texttt{Prec}(X[k+1,i]) - \texttt{Prec}(W[k,i]) \leq \mu - 1 \quad \text{(EW)}$$

$$\forall 0 \leq i < n,\ \forall 0 \leq k < \ell,\ \texttt{Prec}(X[k+1,i]) - \texttt{Prec}(X[k,i]) \leq \nu - 1 \quad \text{(EX)}$$

Fig. 4. Constraints generated for precision optimization of neural networks.

function of the network and its input datasets. First, $\forall 0 \leq i < n,\ \forall 0 \leq k < \ell$, $\texttt{InitialPrec}(W[k,i])$ is the initial precision of the i^{th} neuron of the k^{th} layer, i.e. the precision used for this neuron in the original network before optimization. Second, $\forall 0 \leq i < n$, $\texttt{Output}[i]$ is the precision wanted by the user for the i^{th} neuron of the output of the last layer. This precision can be computed from the parameter δ. Finally, $\forall 0 \leq i < n,\ \forall 0 \leq k < \ell + 1,\ \texttt{ComputedPrec}(X[k,i])$ is the precision of the output of the i^{th} neuron of the layer $k-1$. This precision is computed by static or dynamic analysis by applying Proposition 2 to all the neurons of the original network (with its original precision). Note that for all $0 \leq i < n$, $X[0,i]$ corresponds to the input of the network. Our algorithm works as follows.

1. **Compute the forward accuracy of the network.** For each neuron $0 \leq i < n$ of the k^{th} layer, $0 \leq k < \ell$, we compute the precision `precX[k,i]` of the output `X[k,i]` in the worst case, for all input vectors of the considered dataset D. We also compute at the same time, for each neuron, the minimum and maximum values `Xmin[k,i]` and `Xmax[k,i]` of its output for D. In our implementation, these computations are done by dynamic analysis but they can be done by static analysis using the techniques of [7, 9].
2. **Generate constraints for precision bounds.** On one hand, the forward accuracy computed at Step 1 gives an upper bound on the accuracy of the output `X[k,i]` of each neuron such that $0 \leq i < n$, $0 \leq k < \ell$. On the other hand, the precision desired by the user (thanks to the parameter δ) gives a lower bound on the accuracy of the outputs of the last layer. For each neuron, we generate the constraints (PB), (IP) and (PC) of Figure 4.
3. **Generate constraints for backward precision conditions.** Using Proposition 2, we generate the constraints (EW) and (EC) of Figure 4. In function of the precision of the outputs of the neurons of some layer k, these constraints set conditions on the precision of the neurons of layer k and on the precision of the inputs of layer k. Hence, they propagate in a backward way the constraints set by the user on the final outputs of the network by means of the parameter δ.

The constraints of Figure 4 are linear constraints among integers. We find an optimal solution the system by linear programming. This solution gives the accuracy needed for each neuron for the δ parameter chosen by the user.

As mentioned in Section 2.1 at Equation (2), the dot product performed in each neuron can be composed with another mathematical function, typically a `ReLU` or `tanh` function. Indeed, the examples of Section 5 do use `tanh` functions. While a `ReLU` does not impact the accuracy in the worst case (it just keeps the input value or reset it to zero which does not make the accuracy decrease), this is not the case for the `tanh` function. However, $\forall x \in \mathbb{R},\ |\texttt{tanh}(x)| \leq x$ and the function only reduces the errors in absolute value. It is then possible to make the approximation $\texttt{tanh}(x) \approx x$ without under-estimating the roundoff errors done in the computations of the neurons. In other word, we may get rid of the `tanh` function in the error analysis and constraint generation.

For neural networks combining dot products with other mathematical functions, or to improve the error bounds on the results of the `tanh` function, a fine error propagation can be computed by means of Taylor series developments. For example, for a value x approximated by a floating-point number f with an error e, i.e. $x = f + e$, we have

$$\begin{aligned}
\texttt{tanh}(f+e) &\approx (f+e) - \frac{1}{3}(f+e) && (27)\\
&= (f+e) - \frac{1}{3}\left(f^3 + 3f^2e + 3fe^2 + e^3\right) && (28)\\
&= \left(f - \frac{1}{3}f^3\right) + \left(e - f^2e - fe^2 - \frac{1}{3}e^3\right) && (29)\\
&\approx \texttt{tanh}(f) + \left(\texttt{tanh}(e) - f^2e - fe^2\right) && (30)
\end{aligned}$$

Consequently, the error propagated by the `tanh` function can be approximated by $\texttt{tanh}(e) - f^2e - fe^2$. Similar reasonings can be done for other elementary functions.

5 Experimental Results

In this section, we show on two representative neural networks how our precision tuning method may optimize the precision. These networks originally work in IEEE754 double precision. The prototype used for these experiments has been implemented in Python 2.7 using the `linprog` function of the `scipy` library.

5.1 Neural Network Computing the Hyperbolic Sine

The first neural network we consider computes the hyperbolic sine of the point (x, y). This network, displayed in Figure 5, is made of four layers containing 12, 8, 4 and 1 neurons respectively. The curve in the bottom left corner of Figure 5 displays the percentage of bits that we can save by our method in function of the parameter δ which sets the relative error that we accept between the outputs of the original and transformed networks. On this curve, 100% corresponds to the

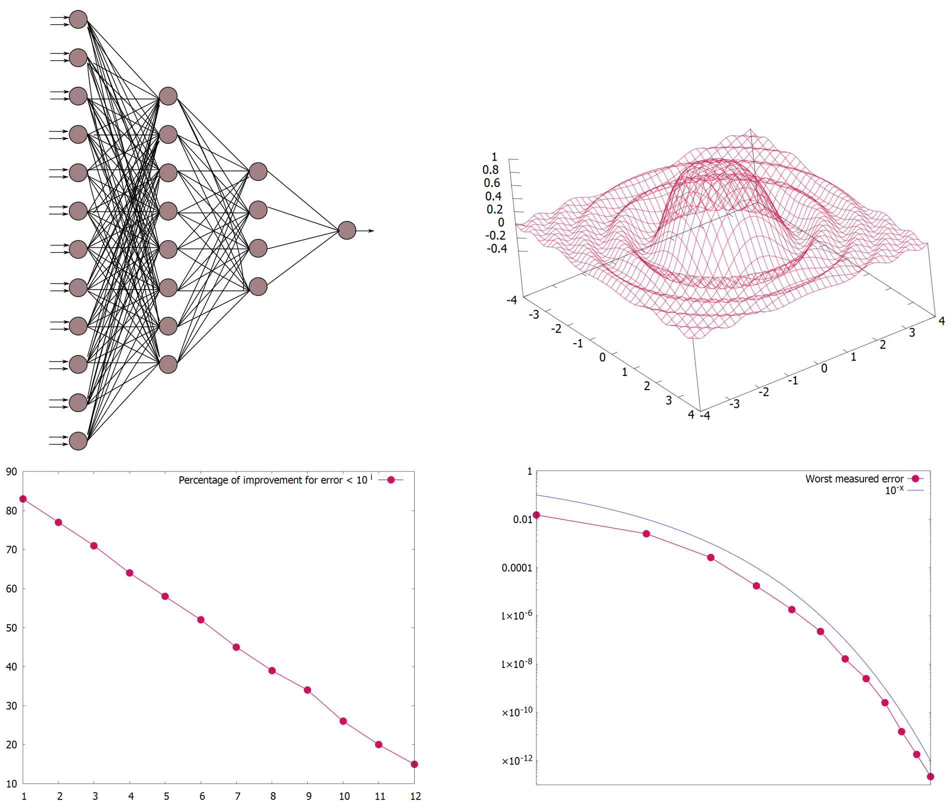

Fig. 5. Top left: Interpolator network used to compute the hyperbolic sine function. Top right: The hyperbolic sine function. Bottom left: Percentage of improvement in bits, compared to the initial network in double precision. Bottom right: Measured error between the original and optimized networks.

case where all the computations are done in double precision. As we can observe, our technique makes it possible to save a significant amount of bits, depending on δ. The curve in the bottom right corner of Figure 5 displays the measured distance between the results of the original and optimized networks. This error is compared to the worst accepted error defined by δ. We can see that the actual error is always less than δ.

In Figure 6, we give more details on the results of our optimization for the case $\delta = 10^{-6}$. The left part of the figure shows the actual number of bits needed for each neuron in this case. Indeed, our method is able to save 54% of bits in this case. In addition, in the right part of Figure 6, we display the best IEEE754 formats that we may choose according to the number of bits needed for each neuron. For this example, 56% of the neurons can be set in single precision while

26				DOUBLE			
24				SINGLE			
23	26			SINGLE	DOUBLE		
22	24			SINGLE	SINGLE		
24	23	26		SINGLE	SINGLE	DOUBLE	
24	23	24	24	SINGLE	SINGLE	SINGLE	SINGLE
24	25	24		SINGLE	DOUBLE	SINGLE	
24	25	24		SINGLE	DOUBLE	SINGLE	
25	25			DOUBLE	DOUBLE		
26	25			DOUBLE	DOUBLE		
26				DOUBLE			
28				DOUBLE			

Fig. 6. Left: Number of bits needed for each neuron for $\delta = 10^{-6}$. Right: Best IEEE754 format which can be used for $\delta = 10^{-6}$.

guaranteeing that the error between the neural network working fully in double precision and the optimized network will be less than $\delta = 10^{-6}$ for any input.

The mean execution time to generate and solve the constraints is 0.9 seconds for $\delta = 10^{-6}$. This time does not change significantly is we take another value for δ.

5.2 Neural Network Computing a Bump Function

In this section, we introduce a second network computing a function of a point (x, y) displayed in Figure 7. Again, we compute the percentage of bits that we can save by our method in function of the parameter δ. Again, our results, displayed in Figure 7 show that our method makes it possible to save an important number of bits (curve at the bottom left corner of Figure 7). As in Section 5.1, we also measure the error between the original and optimized networks and compare it to the theoretical error defined by the parameter δ (curve at the bottom right corner of Figure 7).

The mean execution time for constraint generation and constraint solving is 25 seconds. Note that our implementation is not optimized and consider that all the layers have the same number of neurons (90 in this example.)

6 Conclusion

In this article, we introduced a new method to tune the precision of the computations done inside the neurons of a network in order to save memory while ensuring that the network still answers correctly, compared to the original network. Our method models the propagation of the roundoff errors through a set of linear constraints among integers which can be solved by linear programming. Experimental results show the efficiency of our method.

A first perspective is to test our method on larger, real-size neural networks. This requires to improve our prototype to manage some implementation details.

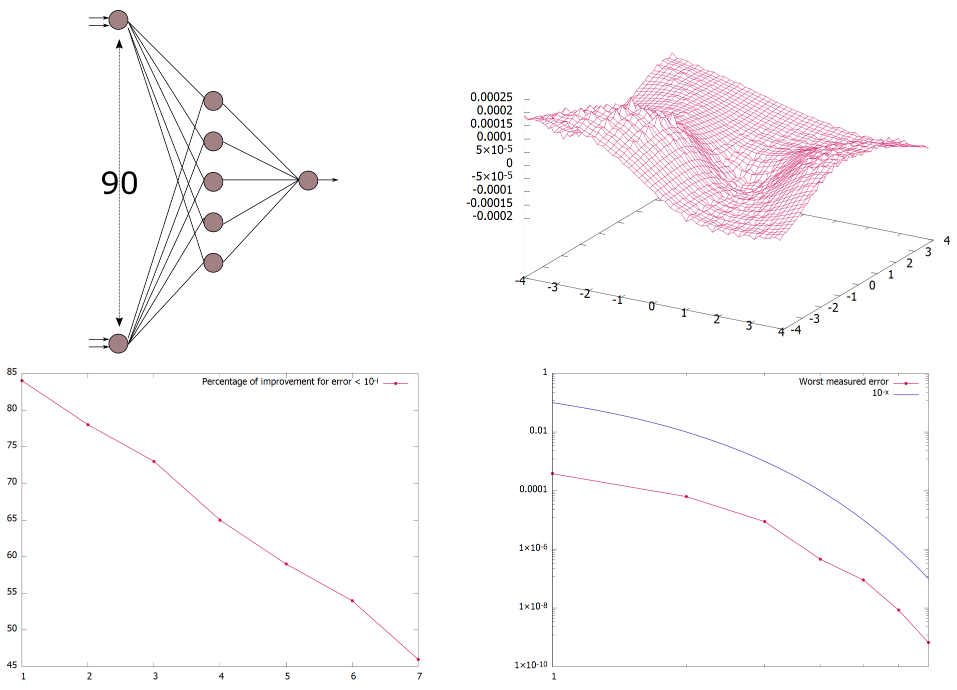

Fig. 7. Top left: An interpolator network used to compute the bump function displayed on the right of the top right corner of the figure. Bottom left: Percentage of improvement in bits, compared to the initial network in double precision. Bottom right: Measured error between the original and optimized 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 precision to a group of neurons in order to reduce the number of equations and variables in the constraint system. The choice of the best partition remains an open question currently and additional work should be carried out in this direction.

A second perspective is to extend our method to classifiers, i.e. to neural networks recognizing patterns given as inputs. While most our our approach can be reused for classifier, it would be necessary to formally define what is an acceptable approximated output for the networks working with less precision. A possibility would be to check that the original and optimized networks almost always classify the inputs in the same way (in $\delta\%$ of the cases, δ being chosen by the user.) In particular, we aim at testing our method on neural networks developed for standard recognition benchmarks such as CIFAR and MNIST.

A third perspective is to generate code for the fixed-point arithmetic [11]. Fixed-point arithmetic (or possibly integer arithmetic) are more and more used to run neural networks, specially in embedded systems. To cope with the fixed-point arithmetic, we need to adapt the error propagations equations of Section 3 without changing the general approach developed in this article.

A last perspective is to improve the method itself, by optimizing the equations of Section 3. For example, some errors are over-estimated. In addition, all the computations done inside the same neuron have the same accuracy and we could improve this point. The way the computations are done inside neurons could also be transformed by re-parsing of the computations in order ot improve their accuracy [3, 4] and, consequently, to allow smaller formats.

References

1. ANSI/IEEE: IEEE Standard for Binary Floating-point Arithmetic (2008)
2. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamaric, Z.: Rigorous floating-point mixed-precision tuning. In: POPL. pp. 300–315. ACM (2017)
3. Damouche, N., Martel, M.: Mixed precision tuning with salsa. In: Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018. pp. 185–194. SciTePress (2018)
4. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. STTT 19(4), 427–448 (2017), https://doi.org/10.1007/s10009-016-0435-0
5. Darulova, E., Horn, E., Sharma, S.: Sound mixed-precision optimization with rewriting. In: Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS. pp. 208–219. IEEE Computer Society / ACM (2018)
6. Darulova, E., Kuncak, V.: Sound compilation of reals. In: POPL'14. pp. 235–248. ACM (2014)
7. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: NASA Formal Methods - 10th International Symposium, NFM. Lecture Notes in Computer Science, vol. 10811, pp. 121–138. Springer (2018)
8. Gao, X., Bayliss, S., Constantinides, G.A.: SOAP: structural optimization of arithmetic expressions for high-level synthesis. In: International Conference on Field-Programmable Technology. pp. 112–119. IEEE (2013)
9. 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, SP. pp. 3–18. IEEE (2018)
10. Goubault, E.: Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In: SAS. LNCS, vol. 7935, pp. 1–3. Springer (2013)
11. Graphics, M.: Algorithmic C Datatypes, software version 2.6 edn. (2011), http://www.mentor.com/esl/catapult/algorithmic
12. 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 - 29th International Conference, CAV. Lecture Notes in Computer Science, vol. 10426, pp. 97–117. Springer (2017)
13. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., LeGendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: Supercomputing, ICS'13. pp. 369–378. ACM (2013)

14. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: 6th International Conference on Learning Representations, ICLR 2018. OpenReview.net (2018)
15. Martel, M.: Floating-point format inference in mixed-precision. In: NFM. LNCS, vol. 10227, pp. 230–246 (2017)
16. Martel, M., Najahi, A., Revy, G.: Code size and accuracy-aware synthesis of fixed-point programs for matrix multiplication. In: Pervasive and Embedded Computing and Communication Systems. pp. 204–214. SciTePress (2014)
17. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser Boston (2010)
18. Nguyen, C., Rubio-Gonzalez, C., Mehne, B., Sen, K., Demmel, J., Kahan, W., Iancu, C., Lavrijsen, W., Bailey, D.H., Hough, D.: Floating-point precision tuning using blame analysis. In: Int. Conf. on Software Engineering (ICSE). ACM (2016)
19. P. Panchekha, A. Sanchez-Stern, J.R.W., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI'15. pp. 1–11. ACM (2015)
20. Rubio-Gonzalez, C., Nguyen, C., Nguyen, H.D., Demmel, J., Kahan, W., Sen, K., Bailey, D.H., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: Int. Conf. for High Performance Computing, Networking, Storage and Analysis. pp. 27:1–27:12. ACM (2013)
21. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018. pp. 10825–10836 (2018)
22. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL 3(POPL), 41:1–41:30 (2019)
23. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In: FM'15. LNCS, vol. 9109, pp. 532–550. Springer (2015)
24. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th USENIX Security Symposium, USENIX Security 2018. pp. 1599–1614. USENIX Association (2018)