Efficient Implementation of Neural Networks Usual Layers on Fixed-Point Architectures

Dorra Ben Khalifa, Matthieu Martel

Efficient Implementation of Neural Networks Usual Layers on Fixed-Point Architectures

Dorra Ben Khalifa
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse
Toulouse, France
dorra.ben-khalifa@enac.fr

Matthieu Martel
LAMPS Laboratory
Université de Perpignan Via Domitia
Perpignan, France
matthieu.martel@univ-perp.fr

Abstract

In this article, we present a new method for implementing a neural network whose weights are floating-point numbers on a fixed-point architecture. The originality of our approach is that fixed-point formats are computed by solving an integer optimization problem derived from the neural network model and concerning the accuracy of computations and results at each point of the network. Therefore, we can bound mathematically the error between the results returned by the floating-point and fixed-point versions of the network. In addition to a formal description of our method, we describe a prototype that implements it. Our tool accepts the most common neural network layers (fully connected, convolutional, max-pooling, etc.), uses an optimizing SMT solver to compute fixed-point formats and synthesizes fixed-point C code from the Tensorflow model of the network. Experimental results show that our tool is able to achieve performance while keeping the relative numerical error below the given tolerance threshold. Furthermore, the results show that our fixed-point synthesized neural networks consume less time and energy when considering a typical embedded platform using an STM32 Nucleo-144 board.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Hardware** → **Power estimation and optimization**; • **Software and its engineering** → **Source code generation**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

Keywords: Neural Networks, Code Synthesis, Constraint Generation, Computer Arithmetic, Embedded Systems

LCTES '24, June 24, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3657578

ACM Reference Format:
Dorra Ben Khalifa and Matthieu Martel. 2024. Efficient Implementation of Neural Networks Usual Layers on Fixed-Point Architectures. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3652032.3657578

1 Introduction

Deep neural networks (DNN) are increasingly used in embedded systems, which poses specific implementation challenges. In order to make DNN usable on smaller devices, it is necessary to reduce the computing, energy and storage requirements of these networks. One can achieve this by a fixed-point translation of the network weights and computations (also called quantization), which usually leads to use 16 or 32 bits word-length integers. However, the main difficulty lies in the fact that these DNN are generally trained on a desktop computer with high computing power before being ported to the target architecture with lower computing power. Moreover, it is not uncommon for the target architecture to use fixed-point arithmetic [22] while the machine used for training uses floating-point arithmetic [1]. It is then necessary to perform this arithmetic change without degrading the performance of the network. It is this problem that we address in this article.

Synthesizing the fixed-point code for a DNN from its floating-point description presents several difficulties since the formats of the numbers must be managed manually and the DNN are very sensitive to the arithmetic used. A change in precision or, even more, in the whole arithmetic, can greatly affect the recognition. To overcome these challenges, we propose in this article a method to synthesize fixed-point code from the floating-point description of a DNN. Our approach consists in generating a set of constraints modeling the accuracy of the computations across the network. The solution of this system, whose unknowns are the fixed number formats, is used to guide the code synthesis [4]. We use a SAT solver [2] and consider the most classical layers of DNN: fully connected, convolutional, max-pooling, upsampling, etc. For example, these layers allow the implementation of a U-Net neural network [19]. Note that, we only generate integer constraints, which greatly simplifies the resolution

compared to real or floating-point constraints. We implement our approach in a prototype tool called Popinns which takes as input a Tensorflow 2.0 model and generates the C code of this model in fixed-point arithmetic. We evaluate Popinns on several neural networks composed of different layers and with thousands of parameters.

Our results show that the analyses carried out by our tool to synthesis fixed-point codes take only a few seconds, even for large neural networks. In terms of accuracy of the codes generated, the relative errors of the fixed-point code compared with the floating-point one for several thresholds remain below the theoretical error, which corresponds to the maximum theoretical error allowed by the user. In addition, the energy consumption results for our codes done by software-defined PowerMeters[1], show that the fixed-point codes generated consume less energy for certain neural networks and, in the worst case, behave like the floating-point codes. As our objective is embedding neural networks onto low-power devices, we have tested our synthesized neural networks on low-power 32-bit micro-controller. The results show that our fixed-point neural networks consume less time and energy on an ARM micro-controller..

The rest of this article is organized as follows. An overview of our technique is presented in Section 2. In Section 3, we introduce background material concerning the fixed-point arithmetic and neural networks. Error propagation through the different layers of a network is described in Section 4. Constraint generation is introduced in Section 5 and the experimental results are given in Section 6. Related work is discussed in Section 7 and Section 8 concludes.

2 Overview of the Tool

In this section, we provide an overview of our method implemented in Popinns to synthesize the fixed-point code of a DNN. Popinns takes as input a Tensorflow 2.0 model and we will use the following simple model as an example.

```
num_classes = 4
input_shape = (8, 8, 1)
model = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(1,kernel_size=(3, 3),
        activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(num_classes,activation="relu")
        ])
```

Once the model has been trained, the following command is all that is needed to generate the code in fixed-point arithmetic.

```
threshold = 6
popinns(model,input_shape,imgs,threshold)
```

[1] https://powerapi.org/

The threshold parameter gives the maximal relative error allowed by the user between the floating-point and fixed-point versions of the model. In our example, 6 means that we want 6 significant bits for the fractional part of each value of the result vector, or, in other words, an absolute error less than 2^{-6}. In addition, imgs is a list of inputs (images in this case) used to perform the dynamic range analysis (typically, these inputs are a subset of the training set).

After a range analysis, Popinns generates a system of constraints from the model. The range analysis is used to determine the weights of the most significant bits of the values at each control point of the DNN. This information is needed to generate the constraints. The constraints themselves are inequalities between linear expressions among integer variables and constants. They are not linear because they also contain implications to encode the min and max operations. The variables are the precision (number of bits) of the inputs of each layer as well as the precision in which each operation is carried out inside each layer. For a dense layer, a different precision is used for the computation of each output. Alternative solutions have been introduced in [5]. Our constraints also depend of constants related to static parameters such as the size of the layer, the weights of a dense layer or of a convolutional kernel, etc. For example, for a dense layer, we generate the set C_k of constraints such that

$$C_k = \left\{ \mathbf{f}^k_{y_i} \leq \max \left(\begin{array}{l} \mathrm{i}_\Psi - \mathbf{f}^k_i, \\ \mathrm{i}_{\Omega_i} - \mathbf{f}^k_{x_i}, \\ 1 - \mathbf{f}^k_i - \mathbf{f}^k_{x_i}, \mathbf{f}^k_i \end{array} \right) + \log_2(n) + 1 : \ 0 \leq i < O_k \right\}$$

where n is the size of the input vector, y_i, $0 \leq i < L$, is one of the L outputs of the layer, $\Omega_i = \max\{|w_{ij}|, \ 0 \leq j < n\}$ where w_{ij} are the synaptic weights of the layer and $\Psi = \max\{|x_j|, \ 0 \leq j < n\}$ where x_j is the input vector (note that the x_j are intervals thanks to the range analysis). The round-off errors on the weights are denoted by ε_{w_i}, where $\varepsilon_{w_i} = \max\{\varepsilon(w_{ij}), \ 0 \leq j < n\}$ and the round-off errors on the inputs are denoted by ε_x, where $\varepsilon_x = \max\{\varepsilon(x_j), \ 0 \leq j < n\}$. We also have $\mathbf{f}^k_i = \lceil \log_2(\varepsilon_{w_i}) \rceil$ and $\mathbf{f}^k_{x_i} = \lceil \log_2(\varepsilon_{x_i}) \rceil$ the precision of the i^{th} neuron of Layer k and of the i^{th} input, respectively.

The solution to the system of constraint, given by an optimizing SMT solver (we use Z3 [7, 18] in practice), provides the formats of the fixed-point numbers (size of the integer and fractional parts). Using this information, Popinns generates a C code in fixed-point precision. For example, the code of the dense layer of our example network is given below.

```
...
fixed_t W_3[36] = { -416, -1392, -2547, 1925,
    ... };
tmp = 0;
for (int j=0;j<9;j++) {
  tmp=fx_addx(tmp,fx_mulx(W_3[0*9+j],x[3][0][j
      ][0],12));
};
x[4][0][0][0] = RELU(fx_xtox(tmp,12,8));
```

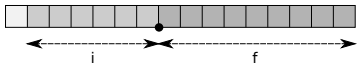

Figure 1. A fixed-point number in format $Q_{\mathrm{i,f}}$, with i = 6 and f = 9. The leftmost bit is used for the sign.

```
tmp = 0;
for (int j=0;j<9;j++) {
  tmp=fx_addx(tmp,fx_mulx(W_3[1*9+j],x[3][0][j
      ][0],12));
};
x[4][0][1][0] = RELU(fx_xtox(tmp,12,8));
...
```

The code synthesized by `Popinns` uses the `fixmath`[2] library for fixed-point arithmetic. For example, the function `fx_mulx(a,b,n)` multiplies `a` and `b` and return a result with `n` bits of fractional part and `fx_xtox` converts the format of a fixed-point number. The fixed-point numbers are stored into 32 bits integers. The first values of `W_3` correspond to the floating-point numbers −0.10160828, −0.33981544, −0.6217438, 0.4698881 stored with 12 bits of fractional part.

For example, evaluations of our network in floating-point and fixed-point arithmetic yields the following results.

$$y_{float} = \big(0.034995, 0.313115, 0.000000, 0.427760\big)^T \quad (1)$$

$$y_{fixed} = \big(0.031250, 0.312500, 0.000000, 0.421875\big)^T \quad (2)$$

The worst relative error between elements of y_{float} and y_{fixed} is 1.3% (less than the threshold) and the cross-entropy between both vectors is 1.36.

3 Background Material

In this section, we introduce the concepts and notations needed to understand this article. Section 3.1 is devoted to fixed-point arithmetic and the roundoff errors generated by this arithmetic. Section 3.2 presents the different types of neural network layers considered in our work.

3.1 Fixed-Point Arithmetic

Fixed-point arithmetic [22] encodes real values inside integers thanks to an implicit scaling factor.

Definition 3.1 (Fixed-point Numbers)**.** Let V be a k-bit signed integer, combined with a factor $\mathbf{f} \in \mathbb{Z}$. Then V represents the real value v defined by

$$v = V \cdot 2^{-\mathbf{f}} \quad . \quad (3)$$

In this article, $Q_{\mathrm{i,f}}$ denotes the format of a given fixed-point number represented using a k-bit integer associated to a scaling factor $\mathbf{f}$, where $k = \mathbf{i} + \mathbf{f}$, as illustrated in Figure 1. Note that unlike the exponent of a floating-point number, the scaling factor of a fixed-point number is static and is not encoded in the program. It is known only by the programmer who is in charge of all the scaling details. For example, when adding two fixed-point numbers, both operand points need to be aligned first, i.e. operands have to be set in the same fixed-point format and this alignment may introduce a round-off error.

In order to bound the round-off errors introduced by the fixed-point arithmetic, let v, v_l , and v_r be three fixed-point variables in the formats $Q_{\mathrm{i,f}}$, $Q_{\mathrm{i}_l,\mathrm{f}_l}$ and $Q_{\mathrm{i}_r,\mathrm{f}_r}$, respectively, and let $\circ \in \{+,-,\times\}$ be some arithmetic operation. We assume that

$$v = v_l \circ v_r \quad (4)$$

For the sake of conciseness, in this article, we do not deal with the determination of the fixed-point formats of the results of elementary operations which can be found in [16]. Instead, let us focus on the error $\varepsilon(v)$ on the result of some operation.

Proposition 3.2 (Errors of Fixed-point Operations)**.** *Let $v = v_l \circ v_r$, $\circ \in \{+,-,\times\}$ be some fixed-point operation.*

i) In absence of overflow, addition and subtraction are error-free. Hence, for $\pm \in \{+,-\}$ we have:

$$\varepsilon(v) = \varepsilon(v_l) \pm \varepsilon(v_r) \quad . \quad (5)$$

ii) For a multiplication, we have:

$$\varepsilon(v) = \varepsilon(v_l) \cdot v_r + \varepsilon(v_r) \cdot v_l + \varepsilon(v_l) \cdot \varepsilon(v_r) + \varepsilon_\times \quad (6)$$

where $\varepsilon_\times$ is the error due to the multiplication itself.

Usually, in fixed-point arithmetic this error is due to the truncation of the exact result of the multiplication to fit in the format $Q_{\mathrm{i,f}}$. Hence we have:

$$\varepsilon_\times \leq 2^{-(\mathrm{f}_l+\mathrm{f}_r)} - 2^{-\mathrm{f}} \quad . \quad (7)$$

A left shift introduces no error but only a possible overflow. Conversely, a right shift of r bits may also be followed by the truncation of the exact result to fit into a smaller format $Q_{\mathrm{i,f}}$.

Proposition 3.3 (Error of Shift Operations)**.** *The evaluation of $v = v_l \gg r$ generates an error defined by*

$$\varepsilon(v) = \varepsilon(v_l) \cdot 2^{-r} + \varepsilon_\gg \quad , \quad \text{with} \quad \varepsilon_\gg \leq 2^{-\mathrm{f}_l} - 2^{-\mathrm{f}} \text{ and } \mathbf{f} = \mathbf{f}_l - r \quad . \quad (8)$$

Note that, in our method to synthesize fixed-point codes for DNNs, no round-off error can be accumulated through the computations since the formats $Q_{\mathrm{i,f}}$ will be chosen in order to maintain the accuracy of the computation greater or equal to the precision of the variables (this is indeed the goal of the system of constraints presented in Section 5.)

3.2 Neural Networks

DNN are made of various kinds of layers which process the input data in many different ways [21]. The layers considered in this article are reviewed hereafter. First, we consider fully connected layers.

[2] https://savannah.nongnu.org/projects/fixmath/

Definition 3.4 (Fully Connected Layer)**.** A fully connected layer (also called dense or FC layer) is made of L neurons, each taking n entries $x_0, \ldots, x_{n-1}$ and computing the output

$$y_i = \sum_{j=0}^{n-1} w_{ij} \cdot x_j, \quad 0 \leq i < L \ . \tag{9}$$

In Equation (9) the w_{ij}, $0 \leq i < L$, $0 \leq j < n$, belong to the matrix W representing the synaptic weights of the neurons. A fully connected layer is generally followed by an activation function layer.

Definition 3.5 (Activation Function Layer)**.** An activation function is usually a non-linear function f applied to an input vector $x = x_0 \ldots x_{n-1}$ component-wise, i.e.

$$f(x) = \big(f(x_0), \ldots, f(x_{n-1})\big)^T \tag{10}$$

Typically $f \in \{\text{ReLU},\ \text{softmax},\ \text{sigmoid}, \ldots\}$ where

$$\text{ReLU}(x_i) = \begin{cases} 0 \text{ if } x_i \leq 0 \\ x_i \text{ otherwise} \end{cases}, \quad x_i \in \mathbb{R},\ 0 \leq i \leq n-1, \tag{11}$$

$$\text{softmax}(x) = y \text{ such that } y_i = \frac{e^{x_i}}{\sum_{k=1}^{L} e^{x_k}}, \quad x, y \in \mathbb{R}^L, \tag{12}$$

$$\text{sigmoid}(x_i) = \frac{e^{x_i}}{e^{x_i}+1} \quad x_i \in \mathbb{R},\ 0 \leq i < n \ . \tag{13}$$

Next, a convolutional layer operates on two-dimensional inputs typically representing an original image or an image processed by former layers [11].

Definition 3.6 (Convolutional Layer)**.** Let M be a matrix of size (s_y, s_x) representing the input and K another matrix of size (s_K, s_K) representing the convolutional kernel. The convolutional layer computes a new matrix M' of size $(s'_y, s'_x) = (s_y - 2 \cdot \lfloor \frac{s_K}{2} \rfloor, s_x - 2 \cdot \lfloor \frac{s_K}{2} \rfloor)$ such that, for $0 \leq i < s'_y$, $0 \leq j < s'_x$ we have

$$M'_{ij} = \sum_{u=0}^{s_K-1} \sum_{v=0}^{s_K-1} k_{uv} \cdot M_{i+u,j+v} \ . \tag{14}$$

We focus now on the max pool layer which replaces each block B of size (p_y, p_x) of an original matrix M by the maximal value of B.

Definition 3.7 (Max Pooling Layer)**.** A two-dimensional max pool layer of factor (p_y, p_x) transforms a (s_y, s_x) matrix M into a $(s'_y, s'_x) = (\lceil \frac{s_y}{p_y} \rceil, \lceil \frac{s_x}{p_x} \rceil)$ matrix M' such that

$$M'_{ij} = \max_{\substack{0 \leq u < p_y \\ 0 \leq v < p_x}} M_{i \cdot p_y + u, j \cdot p_x + v}, \quad 0 \leq i < s'_y,\ 0 \leq j < s'_x \ . \tag{15}$$

We end this description with upsampling layers which multiply the size of their input matrices by some factor s.

Definition 3.8 (Upsampling Layer)**.** Let M be an Upsampling a (s_y, s_x) matrix and s a non-negative integer corresponding to an expansion factor. The upsampling layer expands M into a larger matrix M' of size $(s'_y, s'x) = (s \cdot s_y, s \cdot s_x)$. The new elements are inserting around the elements of M and may be determined by copy (nearest mode) or interpolation (bilinear mode). For example, in nearest mode we have

$$M'_{ij} = M_{\lfloor \frac{i}{s} \rfloor, \lfloor \frac{j}{s} \rfloor},\ 0 \leq i < s'_y,\ 0 \leq j < s'_x \ . \tag{16}$$

The layers described above in this section are these needed to implement, e.g., a U-Net neural network for image segmentation [19].

4 Error Propagation through Layers

In this section, we introduce the equations modelling the propagation of errors through the layers of neural networks introduced in Section 3.2. We start with the fully connected layers.

Proposition 4.1 (Errors in FC Layers)**.** *Let y_i, $0 \leq i < L$, be one of the outputs of a fully connected layer, computed as defined in Equation (9). Let $\Omega_i = \max\{|w_{ij}|,\ 0 \leq j < n\}$, $\Psi = \max\{|x_j|,\ 0 \leq j < n\}$, $\varepsilon_{w_i} = \max\{\varepsilon(w_{ij}),\ 0 \leq j < n\}$ and $\varepsilon_x = \max\{\varepsilon(x_j),\ 0 \leq j < n\}$. Then the error $\varepsilon(y_i)$ is bound by*

$$\varepsilon(y_i) \leq n \cdot \Big[\varepsilon_{w_i} \cdot \Psi + \varepsilon_x \cdot \Omega_i + \varepsilon_{w_i} \cdot \varepsilon_x + \varepsilon_\times\Big] \ . \tag{17}$$

Proof Using equations (5) and (6) of Section 3.1, for a layer of L neurons taking n entries each, and for all $0 \leq i < L$, we have

$$\varepsilon(y_i) = \sum_{j=0}^{n-1} \big(\varepsilon(w_{ij}) \cdot x_j + \varepsilon(x_j) \cdot w_{ij} + \varepsilon(w_{ij}) \cdot \varepsilon(x_j) + \varepsilon_\times\big) \, .$$

Then, using Ω_i, Ψ and ε_{w_i}, we can bound $\varepsilon(y_i)$ by

$$\begin{aligned} \varepsilon(y_i) &\leq \sum_{j=0}^{n-1} \big(\varepsilon_{w_i} \cdot \Psi + \varepsilon_x \cdot \Omega_i + \varepsilon_{w_i} \cdot \varepsilon_x + \varepsilon_\times\big) \ , \\ &\leq n \cdot \Big[\varepsilon_{w_i} \cdot \Psi + \varepsilon_x \cdot \Omega_i + \varepsilon_{w_i} \cdot \varepsilon_x + \varepsilon_\times\Big] \ . \end{aligned} \tag{18}$$

■

Note that the bound given in Equation (17) is conservative. We believe that it could be tighten at the price of a more complex proof.

Let $Q_{\mathbf{i}_x, \mathbf{f}_x}$ and $Q_{\mathbf{i}_i, \mathbf{f}_i}$ be the formats of the input vector x and the working format of the current neuron i, respectively. In Equation (18), Ω_i is known statically and Ψ is computed by range analysis and assumed to be known at constraint generation time. Additionally, $\varepsilon_{w_i} \leq 2^{-\mathbf{f}_i}$ and, according to Equation (7),

$$\varepsilon_\times \leq 2^{-(\mathbf{f}_i + \mathbf{f}_x)} - 2^{-\mathbf{f}_i} \ . \tag{19}$$

Thus, the only terms that are unknown at this stage in Equation (18) are the integer quantifies $\mathbf{f}_x$ and $\mathbf{f}_i$. We will see in Section 5 that the $\mathbf{f}_x$ and $\mathbf{f}_i$ are computed by propagation from one layer to the other assuming that $\mathbf{f}_x$ is given by the user for the first layer, just like $\mathbf{f}_i$ for the last one.

Second, we approximate the activation functions by piecewise linear functions which need to be themselves translated

into fixed-point. Property 4.2 details how we proceed for ReLU.

Proposition 4.2 (Errors in ReLU Layers). *For a ReLU function as defined in Equation (11) and for a fixed-point number x in format $Q_{\mathrm{i,f}}$, we have*

$$\varepsilon\big(ReLU(x)\big) \leq \begin{cases} 2^{-\mathrm{f}} \text{ if } x \geq 0 \\ 0 \text{ otherwise} \end{cases} . \tag{20}$$

Proof Since x is in format $Q_{\mathrm{i,f}}$, we have $-2^{-\mathrm{f}} < \varepsilon(x) < 2^{-\mathrm{f}}$. Let us consider the three following cases.

i) If $x \geq 2^{-\mathrm{f}}$ then $x \geq \varepsilon(x)$ and $x - \varepsilon(x) > 0$. Then no unstable test [20] may occur and $\varepsilon\big(\mathrm{ReLU}(x)\big) = \varepsilon(x) < 2^{-\mathrm{f}}$.

ii) If $x = 0$ then $\mathrm{ReLU}(x) = 0$ and

$$\mathrm{ReLU}(x + \varepsilon(x)) = \begin{cases} \varepsilon(x) \text{ if } \varepsilon(x) > 0 \\ 0 \text{ otherwise} \end{cases} .$$

In any case, $\varepsilon(x) < 2^{-\mathrm{f}}$.

iii) Finally, if $x \leq 2^{-\mathrm{f}}$, then no unstable test may occur, $\mathrm{Relu}(x) = \mathrm{Relu}(x + \varepsilon(x)) = 0$ and $\varepsilon\big(\mathrm{ReLU}(x)\big) = 0$. ■

We omit the piece-wise linearization and the computation of the errors for the other activation functions such as softmax or sigmoid. They are detailed in [5]. Let us now focus on convolutions.

Proposition 4.3 (Errors in Convolutional Layers). *Let M'_{ij} be one of the values computed during a convolution following Equation (14). Let $\Gamma = \max\{|k_{ij}|,\ 0 \leq i,j < s_K\}$, let $\Psi_i = \max\{|M_{ij}|,\ 0 \leq j < s_x\},\ \ 0 \leq i < s_y$ and let $\varepsilon_k = \max\{\varepsilon(K_{ij}),\ 0 \leq i,j < s_K\}$ and $\varepsilon_{x_i} = \max\{\varepsilon(M_{ij}),\ 0 \leq j < s_x\}$ for $0 \leq i < s_y$. The error on M'_{ij} is bounded by*

$$\varepsilon(M'_{ij}) \leq \sum_{u=0}^{s_K-1} s_K \cdot \Big[\varepsilon_k \cdot \Psi_{i+u} + \varepsilon_{x_{i+u}} \cdot \Gamma + \varepsilon_k \cdot \varepsilon_{x_{i+u}} + \varepsilon_\times\Big] . \tag{21}$$

Proof Following the reasoning used for the fully connected layer, we compute only one error per row of M', that is, for all $0 \leq j < s'_x$, $\varepsilon(M'_{ij}) \leq \varepsilon(y_i)$ where

$$\begin{aligned} \varepsilon(y_i) &\leq \sum_{u=0}^{s_K-1} \sum_{j=0}^{s_K-1} \left(\varepsilon_k \cdot \Psi_{i+u} + \varepsilon_{x_{i+u}} \cdot \Gamma + \varepsilon_k \cdot \varepsilon_{x_{i+u}} + \varepsilon_\times\right) \\ &\leq \sum_{u=0}^{s_K-1} s_K \cdot \Big[\varepsilon_k \cdot \Psi_{i+u} + \varepsilon_{x_{i+u}} \cdot \Gamma + \varepsilon_k \cdot \varepsilon_{x_{i+u}} + \varepsilon_\times\Big] . \end{aligned}$$

■

Again, the unknowns in Equation (4) are ε_{x_i}, ε_k and $\varepsilon_\times$ which only depend on $\mathbf{f}_{x_i}$ and on the precision $\mathbf{f}_k$ in which is performed the convolution. Both $\mathbf{f}_{x_i}$ and $\mathbf{f}_k$ will be the variables of our system of constraints in Section 5.

We consider now max pool layers for which we are going to compute only one error per row of M', as stated in Property 4.4.

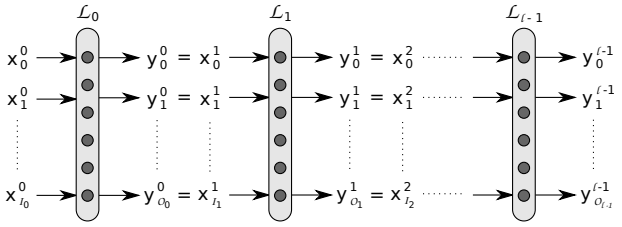

Figure 2. Notations used to represent a DNN at constraint generation time.

Proposition 4.4 (Errors in Max Pooling Layers). *Let us consider a max pool layer of factor (p_y, p_x) and let $\varepsilon_{x_i} = \max\{\varepsilon(M_{ij}),\ 0 \leq j < s_x\}$ for $0 \leq i < s_y$. Then, for $0 \leq i < s'_y$ and $0 \leq j < s'_x$,*

$$\varepsilon(M'_{ij}) \leq 2 \cdot \max_{0 \leq u < p_y} \varepsilon_{x_{i \cdot p_y + u}} . \tag{22}$$

Intuitively, in Equation (22), the factor 2 is added to cover the situations where some unstable test could occur. In this case, the error may be twice greater than $\varepsilon_{x_{i \cdot p_y + u}}$.

Finally, an upsampling layer (in nearest mode) only duplicates the elements of the matrix M and the errors follow as stated in Property 4.5.

Proposition 4.5 (Errors in Upsampling Layers). *For an upsampling layer of factor s as defined in Equation (16) and for $0 \leq i < s'_y$, $0 \leq j < s'_x$, we have*

$$\varepsilon(M'_{ij}) = \varepsilon(M_{\lfloor \frac{i}{s} \rfloor, \lfloor \frac{j}{s} \rfloor}) . \tag{23}$$

5 Constraint Generation

In this section, we introduce the constraints which model the accuracy of the fixed-point computations inside a neural network composed of the layers introduced in Section 3.2. These constraints, summarized in Figure 3, are derived from the error propagation equations introduced in Section 4. We introduce some useful properties in Section 5.1. Next, the constraint generation is described in Section 5.2 and the cost function is discussed in Section 5.3.

5.1 Preliminary Properties

First of all, the following lemma and corollary are used in the proofs of properties 5.3 and 5.4.

Lemma 5.1. *Let $a > 0$ and $b > 0$ be two real numbers, then*

$$\log_2(a+b) \leq \max\big(\log_2(a), \log_2(b)\big) + 1 . \tag{24}$$

Proof Since $a > 0$ and $b > 0$, $a + b \leq 2 \cdot \max(a,b)$ and by monotony of the $\log_2$ function,

$$\log_2(a+b) \leq \log_2\big(2 \cdot \max(a,b)\big) \leq \log_2\big(\max(a,b)\big) + 1 .$$

Using again the monotony of the $\log_2$ function, we know that $\log_2\big(\max(a,b)\big) = \max\big(\log_2(a), \log_2(b)\big)$. Consequently,

$$\log_2(a+b) \leq \max\big(\log_2(a), \log_2(b)\big) + 1 .$$

■

$$C_{init} = \left\{ \mathbf{f}^0_{x_j} \leq \Theta_{in} \ : \ 0 \leq j < \mathcal{I}_0 \right\} \cup \left\{ \Theta_{out} \leq \mathbf{f}^{\ell-1}_{y_i} \ : \ 0 \leq i < O_{\ell-1} \right\} \quad \text{(Init)}$$

$$C_{link} = \left\{ \mathbf{f}^{k+1}_{x_i} \leq \mathbf{f}^k_{y_i} : \ 0 \leq k < \ell - 1, \ 0 \leq i < O_k \right\} \quad \text{(Link)}$$

$$C^k_{FC} = \left\{ \begin{array}{l} \mathbf{f}^k_i \geq \mathbf{f}^k_{y_i} + \mathrm{i}_\Psi + \llcorner \log_2(n) \lrcorner + 1 \\ \mathbf{f}^k_{x_i} \geq \mathbf{f}^k_{y_i} + \mathrm{i}_{\Omega_i} + \llcorner \log_2(n) \lrcorner + 1 \end{array} \ : \ 0 \leq i < O_k \right\} \quad \text{(FC)}$$

$$C^k_{ReLU} = \left\{ \ \mathbf{f}^k_{x_i} \geq \mathbf{f}^k_{y_i} + 1 \quad : \ 0 \leq i < O_k \right\} \quad \text{(ReLU)}$$

$$C^k_{convo} = \left\{ \begin{array}{l} \mathbf{f}^k \geq \mathbf{f}^k_{y_i} + \mathrm{i}_\Psi + \llcorner \log_2(n) \lrcorner + 2 \\ \mathbf{f}^k_{x_{i+u}} \geq \mathbf{f}^k_{y_i} + \mathrm{i}_\Gamma + \llcorner \log_2(n) \lrcorner + 2 \\ \mathbf{f}^k_{x_{i+u}} \geq \mathbf{f}^k_{y_i} - \mathbf{f}^k + \llcorner \log_2(n) \lrcorner + 3 \end{array} \ : \ \begin{array}{l} 0 \leq u < s_K \\ 0 \leq i < O_k \end{array} \right\} \quad \text{(Convo)}$$

$$C^k_{maxpool} = \left\{ \mathbf{f}^k_{y_i} \leq \mathbf{f}^k_{x_{i \times p_y + p}} + 1 \ : \ 0 \leq p \leq p_y, \ 0 \leq i < O_k \right\} \quad \text{(MaxPool)}$$

$$C^k_{upsamp} = \left\{ \mathbf{f}^k_{y_i} \leq \mathbf{f}^k_{x_{\llcorner \frac{i}{p_y} \lrcorner}} \ : \ 0 \leq i < O_k \right\} \quad \text{(UpSamp)}$$

Figure 3. Constraints generated to determine the fixed point formats of a DNN.

Corollary 5.2. *Let $a_1, \ldots, a_n$ be n positive real numbers, then*

$$\log_2(a_1 + \ldots + a_n) \leq \max\left(\log_2(a_1), \ldots, \log_2(a_n)\right) + n - 1 \ . \tag{25}$$

Proof By induction on n, using Lemma 5.1. ∎

We present hereafter two properties that will allow us to replace the inequalities of properties 4.1 and 4.3 by integer constraints. First, let us consider a dot product performed by a fully connected layer $\mathcal{L}_k$. We have the following property.

Proposition 5.3 (Constraints for FC Layers)**.** *Let Ψ, Ω_i and ε_{w_i} be defined as in Property 4.1 and let $n = \mathcal{I}_k$. Let $\mathbf{f}^k_i = \lceil \log_2(\varepsilon_{w_i}) \rceil$ and $\mathbf{f}^k_{x_i} = \lceil \log_2(\varepsilon_{x_i}) \rceil$ be the precision of the i^{th} neuron of $\mathcal{L}_k$ and of the i^{th} input, respectively. Finally, let $i_{\Omega_i} = \lceil \log_2(\Omega_i) \rceil$ and $i_\Psi = \lceil \log_2(\Psi) \rceil$. Then we have*

$$\left| \begin{array}{l} \mathbf{f}^k_i \geq \mathbf{f}^k_{y_i} + i_\Psi + \llcorner \log_2(n) \lrcorner + 1 \\ \mathbf{f}^k_{x_i} \geq \mathbf{f}^k_{y_i} + i_{\Omega_i} + \llcorner \log_2(n) \lrcorner + 1 \end{array} \right. \tag{26}$$

Proof Let $2^{-\mathbf{f}^k_{y_i}}$ be an upper bound on the error introduced by the i^{th} neuron of the layer, following equations (18) and (19), since $\varepsilon_{w_i} \leq 2^{-\mathbf{f}^k_i}$ and $\varepsilon_{x_i} \leq 2^{-\mathbf{f}^k_{x_i}}$, we have

$$n \cdot [2^{-\mathbf{f}^k_i} \cdot \Psi + 2^{-\mathbf{f}^k_{x_i}} \cdot \Omega_i + 2^{-\mathbf{f}^k_i} \cdot 2^{-\mathbf{f}^k_{x_i}} + 2^{-\mathbf{f}^k_i} \cdot 2^{-\mathbf{f}^k_{x_i}} - 2^{-\mathbf{f}^k_i} \ ] \leq 2^{-\mathbf{f}^k_{y_i}}$$

and

$$-\mathbf{f}^k_{y_i} \leq \log_2\left(n \cdot \left[2^{-\mathbf{f}^k_i} \cdot \Psi + 2^{-\mathbf{f}^k_{x_i}} \cdot \Omega_i + 2 \cdot 2^{-\mathbf{f}^k_i} \cdot 2^{-\mathbf{f}^k_{x_i}} - 2^{-\mathbf{f}^k_i}\right]\right) \ . \tag{27}$$

We have the following equations:

i) $\log_2(2^{-\mathbf{f}^k_i} \cdot \Psi) = \mathrm{i}_\Psi - \mathbf{f}^k_i$,

ii) $\log_2(2^{-\mathbf{f}^k_{x_i}} \cdot \Omega_i) = \mathrm{i}_{\Omega_i} - \mathbf{f}^k_{x_i}$,

iii) $\log_2(2 \cdot 2^{-\mathbf{f}^k_i} \cdot 2^{-\mathbf{f}^k_{x_i}}) = 1 - \mathbf{f}^k_i - \mathbf{f}^k_{x_i}$,

iv) $\log_2(-2^{-\mathbf{f}^k_i}) = \mathbf{f}^k_i$.

Using Corollary 5.2, we have to take the maximum of these four quantifies but the last two ones corresponding to points *iii)* and *(iv)* are implied by the ones of points *i)* and *ii)* and may be omitted. Then using simply Lemma 5.1 instead of Corollary 5.2, Equation (27) becomes

$$-\mathbf{f}^k_{y_i} \geq \max\left(\mathrm{i}_\Psi - \mathbf{f}^k_i, \mathrm{i}_{\Omega_i} - \mathbf{f}^k_{x_i}\right) + \llcorner \log_2(n) \lrcorner + 1 \ .$$

It follows that

$$\mathbf{f}^k_{y_i} \leq \min\left(\mathbf{f}^k_i - \mathrm{i}_\Psi, \mathbf{f}^k_{x_i} - \mathrm{i}_{\Omega_i}\right) - \llcorner \log_2(n) \lrcorner - 1 \ . \tag{28}$$

Finally, by decomposing the min of Equation (28), we obtain Equation (26). ∎

Similarly, the following property is used to replace the inequalities of properties 4.3 by integer constraints in convolutions.

Proposition 5.4 (Constraints for Convolutional Layers)**.** *Let Γ and Ψ_i, ε_k and ε_{x_i} be defined as in Property 4.3 and let $n = \mathcal{I}_k$. Let $\mathbf{f}^k = \lceil \log_2(\varepsilon_k) \rceil$ be the precision of the convolution performed at Layer $\mathcal{L}_k$, let $\mathbf{f}_{x_i} = \lceil \log_2(\varepsilon_{x_i}) \rceil$ be the precision of the i^{th} line of inputs and let $i_{\Psi_i} = \lceil \log_2(\Psi_i) \rceil$. Then*

$$\left| \begin{array}{l} \mathbf{f}^k \geq \mathbf{f}^k_{y_i} + i_\Psi + \llcorner \log_2(n) \lrcorner + 2 \ , \\ \mathbf{f}^k_{x_{i+u}} \geq \mathbf{f}^k_{y_i} + i_\Gamma + \llcorner \log_2(n) \lrcorner + 2, \ 0 \leq u < s_K \ , \\ \mathbf{f}^k_{x_{i+u}} \geq \mathbf{f}^k_{y_i} - \mathbf{f}^k + \llcorner \log_2(n) \lrcorner + 3, \ 0 \leq u < s_K \ . \end{array} \right. \tag{29}$$

Proof From Equation (21), using the notations introduced above, and since since $\varepsilon_k \leq 2^{-\mathbf{f}^k}$ and $\varepsilon_{x_i} \leq 2^{-\mathbf{f}_{x_i}}$, we have

$$s_K \cdot \sum_{u=0}^{s_K-1} \left[2^{-\mathbf{f}^k} \cdot \Psi_{i+u} + \Gamma \cdot 2^{-\mathbf{f}_{x_{i+u}}} + 2^{-\mathbf{f}^k} \cdot 2^{-\mathbf{f}_{x_{i+u}}} + 2^{-\mathbf{f}^k} \cdot 2^{-\mathbf{f}_{x_i}} - 2^{-\mathbf{f}_i^k}\right] \leq 2^{\mathbf{f}_{y_i}^k} \quad . \quad (30)$$

As in Property 5.3, we use the following equations.

i) $\log_2(2^{-\mathbf{f}^k} \cdot \Psi_{i+u}) = \mathrm{i}_{\Psi_{i+u}} - \mathbf{f}^k$,
ii) $\log_2(\Gamma \cdot 2^{-\mathbf{f}_{x_{i+u}}}) = \mathrm{i}_\Gamma - \mathbf{f}_{x_{i+u}}$,
iii) $\log_2(2 \cdot 2^{-\mathbf{f}^k} \cdot 2^{-\mathbf{f}_{x_{i+u}}}) = 1 - \mathbf{f}^k - \mathbf{f}_{x_{i+u}}$,
iv) $\log_2(-2^{-\mathbf{f}^k}) = \mathbf{f}^k$.

Using Corollary 5.2, we have to take the maximum of three of these quantities for all $0 \leq u < s_K$ (the last one being implied by the others.) By rewriting Equation (30), we obtain Equation (29). ∎

5.2 Constraint Generation

For constraint generation, we consider a DNN made of ℓ layers $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{\ell-1}$ (see Figure 2). For the sake of simplicity, we consider that the inputs and outputs of each layer have a depth of one. Nevertheless, in our implementation, we accept input shapes of greater depth (three-dimensional shapes instead of the two-dimensional shapes considered here to avoid cumbersome notations.)

Let $\mathcal{I}_k$ and $\mathcal{O}_k$ respectively be the number of inputs and outputs of $\mathcal{L}_k$, $0 \leq k < \ell$. Let x_j^k, $0 \leq j < \mathcal{I}_k$, (resp. y_i^k, $0 \leq i < \mathcal{O}_k$) be the j^{th} input (resp. i^{th} output) of Layer $\mathcal{L}_k$, $0 \leq k < \ell$ and let $\varepsilon(x_j^k)$ and $\varepsilon(y_i^k)$ be the errors associated to these inputs and outputs (note that $\mathcal{O}_k = \mathcal{I}_{k+1}$ for $0 \leq k < \ell - 1$).

The unknowns of our system of constraints are the precisions (fractional sizes f of Figure 1) $\mathbf{f}_{x_j}^k \in \mathbb{Z}$ and $\mathbf{f}_{y_i}^k \in \mathbb{Z}$ on the inputs and outputs of each layer, i.e. $\varepsilon(x_j^k) \leq 2^{-\mathbf{f}_{x_j}^k}$, resp. $\varepsilon(y_i^k) \leq 2^{-\mathbf{f}_{y_i}^k}$, as well as the working precision $\mathbf{f}_i^k \in \mathbb{Z}$ of the neurons, $0 \leq i < \mathcal{O}_k$. In this way, we only have integer constraints. Recall that this greatly simplifies the resolution by Z3 compared to real or floating-point constraints.

In Figure 3, Rule (Init) sets the error thresholds $\Theta_{in} \in \mathbb{Z}$ and $\Theta_{out} \in \mathbb{Z}$ on the inputs and outputs of the network. These thresholds are given by the user. Rule (Link) simply relates the outputs of one layer to the inputs of the next layer. Rules (FC) and (Convo) are for fully connected and convolutional layers. They directly come from properties 5.3 and 5.4 introduced in Section 5.1. Rule (MaxPool) maps the maximal precision of some input block of size (p_y, p_x) to the corresponding element of the output block. Finally, Rule (UpSamp) duplicate the precision of the input matrix into the elements of the larger matrix.

5.3 Cost Function

As mentioned earlier, to solve the system of constraints introduced in Section 5.2, we call the Z3 optimizing SMT solver [7]. To optimize the solution the solver needs a cost function and several relevant functions may be defined for this purpose as presented in the work of [3]. In this work, we minimize the total number of bits needed to represent the fractional parts of the fixed-point numbers. Then our cost function for a DNN N made of ℓ layers $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{\ell-1}$ is

$$\mathrm{cost}(N) = \sum_{0 \leq k < \ell} \left(\sum_{0 \leq j < \mathcal{I}_k} \mathbf{f}_{x_j}^k + \sum_{0 \leq i < \mathcal{O}_k} \mathbf{f}_{x_j}^k + \sum_{0 \leq i < \mathcal{O}_k} \mathbf{f}_i^k \right) \quad (31)$$

However, other cost functions could make sense, for example, to minimize the number of format conversions (e.g. before additions), to reduce the size $\mathbf{i} + \mathbf{f}$ of the largest format $Q_{\mathbf{i},\mathbf{f}}$ of the values arising in the computations (e.g. to let them fit into a predefined format such as 32 bits), to minimize the size of the operators (that is $\sum_{0 \leq k < \ell} \left(\sum_{0 \leq i < \mathcal{O}_k} \mathbf{f}_i^k \right)$), etc. Weighed sums of combinations of these different cost functions could also used.

NN	CV	MP	US	FL	FC	RL	IN	CL	PA
1	1	1	-	2	2	2	64	4	126
2	-	-	-	1	1	1	64	12	780
3	1	-	-	1	1	1	144	10	1 020
4	2	1	1	1	1	3	144	10	390
5	-	-	-	2	2	1	64	8	1 176
6	1	-	-	3	3	2	64	6	2 838
7	-	-	-	4	4	3	100	5	10 405
8	1	-	-	3	3	3	144	10	8 120

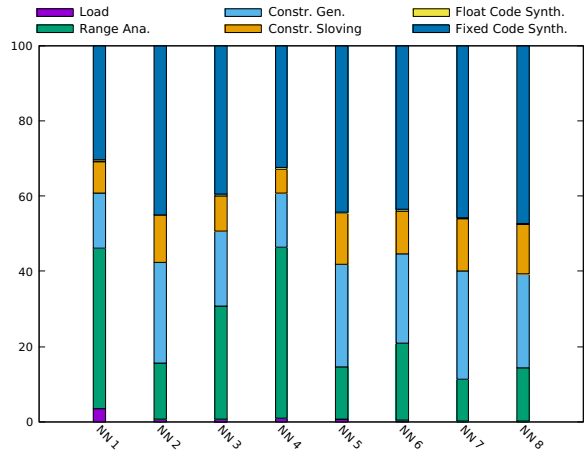

Figure 4. Top: Description of the neural networks used in our experiments. NN: identifier of the network. Next is indicated the number of layer of each kind. CV: convolutional, MP: Max Pool, US: Up Samp, FL: Flatten, FC: Fully connected, RL: ReLu. IN: Size of the input. CL: Number of classes. PA: number of parameters. **Botttom:** Percentage of time spent in each phase of `Popinns` to synthesize the fixed-point code of the network.

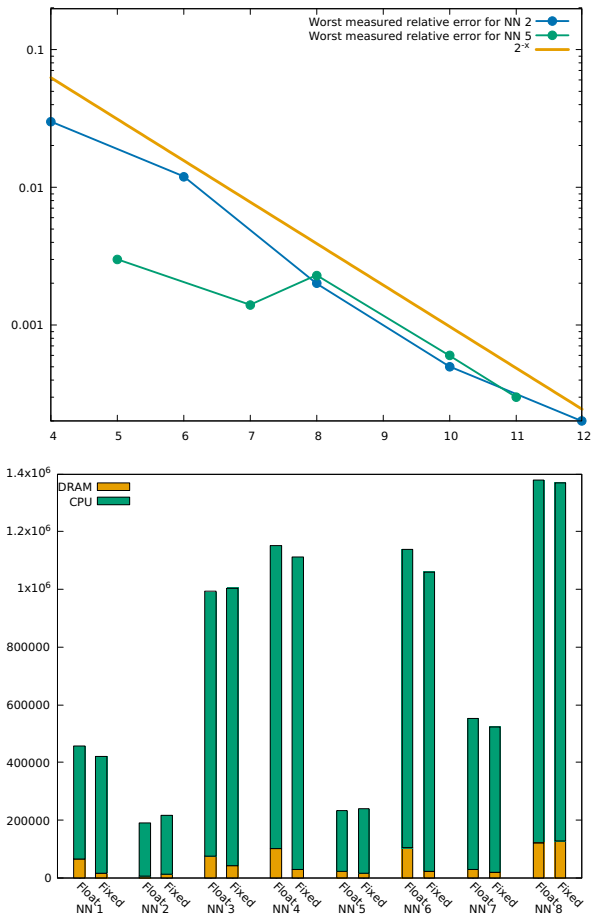

Figure 5. Top: Relative error of the fixed-point version of some neural network (NN2 and NN5) with respect to the original floating-point version. Errors are measured on evaluations of the networks in function of the accuracy threshold set by the user. The 2^{-x} curve indicates the theoretical threshold. **Bottom:** Measurement of the energy consumption (CPU and DRAM) of the floating-point and fixed-point versions of our neural networks.

6 Experimental Results

In this section, we introduce some experimental results assessing the efficiency of our prototype `Popinns` which implements the technique described in Section 5 of this article.

`Popinns` takes as input a `Tensorflow 2.0` model and generates the `C` code of this model in fixed-point arithmetic. `Popinns` processes in 6 stages described hereafter. **Load:** The `Tensorflow` model is read and translated into `Popinns` internal representation. Currently, the layers accepted are `dense`, `conv2d`, `max_pooling2d`, `up_sampling2d` and `flatten`. The `ReLU` activation function is also handled by our tool. **Range Analysis:** Once the model is extracted, a range analysis is performed. In the current version of `Popinns`, this analysis is dynamic but we plan to make it static using affine forms [6]. The dynamic analysis consists of running the DNN with a set of input data and taking, for each output, the join of the values obtained at each run. This gives an under-approximation of the possible values which is acceptable in practice. **Constraint Generation:** The third stage is the generation of the constraints. Our constraints are inequalities between linear expressions among integer variables and constants. They are not linear because they also contain implications to encode the min and max operations. The variables are the precision (number of bits) of the inputs of each layer as well as the precision in which each operation is carried out inside each layer. **Constraint solving:** The solution of the generated constraints is computed by the `z3` optimizing SMT solver [7] that gives the optimal formats of the fixed-point numbers at each point of the DNN. **Floating-point and fixed-point Code Synthesis:** The last steps consist of synthesizing the floating-point and fixed-point code implementing the DNN.

`Popinns` has been evaluated on 8 neural networks, each made of a mix of the layers introduced in Section 3.2 and having up to 10 000 parameters. The characteristics of these networks are given in the left hand side of Figure 4.

The histogram in the right hand side of Figure 4 shows the percentage of time spent by `Popinns` in each stage of the synthesis process. The range analysis and the fixed-point code generation are the longest operations. The duration of the range analysis obviously depends on how many inputs are used in the computation of the under-approximation (100 in our experiments). Next, it is interesting to underline that solving the system of constraint does not require much time. Finally, the fixed-point code is larger (and its generation longer) because some loops are unfolded in order to use different formats at different iterations. Overall, the time needed for the code synthesis remains short, ranging from 2.05s for the smallest network NN1 to 5.7s for the largest network NN7.

Concerning the accuracy of the synthesized codes, left hand side of Figure 5 shows the relative errors of the fixed-point code with respect to the floating-point code for two of our networts, NN2 and NN5. The errors have been computed for several thresholds (x-axis) and must remain under the yellow line which corresponds to the maximal allowed theoretical error (note that the y-axis is in logarithmic scale). These errors have been measured on evaluations of the floating-point and fixed-point networks and fulfill the user's requirements. The right hand side of Figure 5 depicts the energy consumed[3] by the execution of the neural networks. We observe that the energy consumed by the DRAM and the CPU for fixed-point and floating-point code versions is similar for most of our neural networks. One observation on these results is that we do not gain much in terms of energy, given that the fixed-point library used only returns uniform formats in 32 bits.

[3] https://github.com/powerapi-ng/jouleit

Concerning the evaluation on an embedded platform, we tested all the synthesized neural networks on a STM32F207ZG micro- controller with a 32-bit Arm® Cortex®-M3 CPU (120 MHz max) with 1 MByte of Flash program memory and 128 Kbyte of RAM. We used Miosix, an OS kernel designed to run on 32bit micro-controllers[4]. As the Cortex®-M3 ARM core lacks the support for hardware floating-point, to compile all the benchmarks we have used the software floating-point provided by the compiler when the `-msoft-float` command line switch is passed. With the switch enabled, the compiler will not generate any floating-point unit (FPU) instructions, and appropriate function calls to emulate floating-point computation are generated by passing floating-point arguments in integer registers. The time measurements were taken by querying the high-resolution timer provided by the Miosix API, implemented by exploiting one of the micro-controller's internal timers. Concerning our tool setup, we will evaluate NNs with a single user-given threshold requirement. This precision was chosen as the best requirement that gives the lowest relative error with respect to the original floating-point result. For the energy measurement, we employed the X-NUCLEO-LPM01A expansion board which is a 1.8 V to 3.3 V programmable power supply source with advanced power consumption measurement capability.

The results depicted in Figure 6 show that out of 8 neural networks, only NN1, NN2 , NN3 and NN4 can run on the board. This is due to the small amount of flash memory available in this micro-controller, which is not sufficient for the sizes of the remaining neural networks. We observe, on the top hand side of Figure 6, that the fixed-point neural networks generated are 2× to 4× faster than the floating-point networks as for NN1 and NN4 respectively. In addition, the fixed-point versions of NN2 and NN3 manage to run on the board in no more than 0.5 seconds, unlike their floating-point versions, which consume a lot of memory on the board. In terms of power consumption measured with the X-NUCLEO-LPM01A STM32 power shield, the fixed-point neural networks generated by our method consume less energy than their original floating-point versions. For instance, for NN4, the fixed-point model consumes about 0.25 μJ less energy than the floating-point model. Our smallest neural network NN1 consumes only 0.076μJ in its fixed-point version compared to 0.130μJ in its floating-point version.

7 Related Work

While much effort has been devoted to the safety and robustness of deep learning code, here we focus only on studies conducted on the effects of reduced precision for neural networks using fixed-point arithmetic. There are a number of different reduced precision data representations, the more standard floating-point based approaches [8, 10, 14] and customized fixed-point synthesis schemes [5, 17, 23].

[4] https://miosix.org/

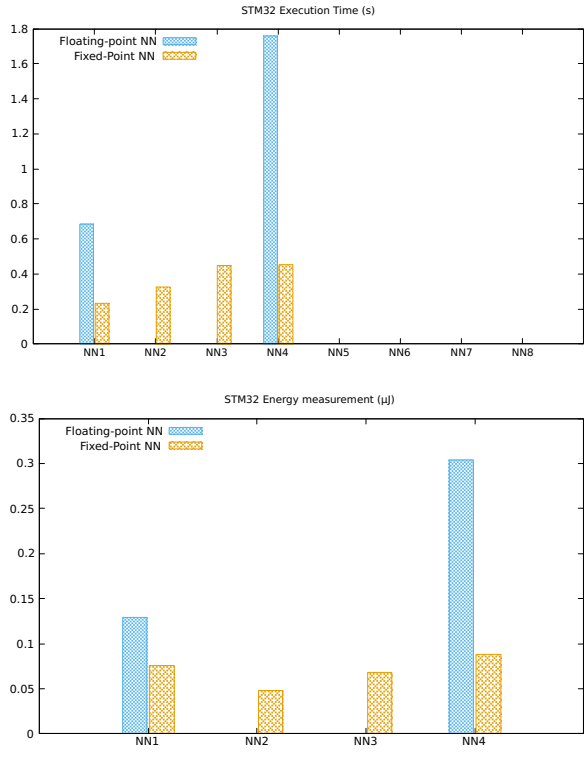

Figure 6. Measurement of execution time in seconds (top) and power consumption in μJ (bottom) of the floating-point and fixed-point synthesized neural networks tested on the STM32 Nucleo-144 board.

Low-Precision Floating-Point Inference. The main approach proposed in [10] consists on 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. Based on the formal modeling of the propagation of rounding errors, a set of linear constraints among integers is generated which can be solved by linear programming. However, the prototype implemented is limited to floating-point arithmetic, and no fixed-point solution has been proposed to run a DNN.

The work proposed in[14] presents a semi-automated framework to bound and interpret the impact of rounding errors due to the precision choice for inference in generic DNNs. The framework can receive any TensorFlow/Keras model in the front-end and computes and then propagates rounding errors through the computations for the back-end by affine and interval arithmetic. However, this work is limited to floating-point precision, whereas ours goes further by converting floating-point formats to fixed-point ones and targeting embedded architectures such as STM32 microcontrollers. Ferro et al. [8] proposed a floating-point autotuning tool on different kinds of neural networks. Their

tuning approach is based on a stochastic arithmetic in order to obtain the lowest precision for each of its parameters. However, this work do not propose a conversion from floating-point to fixed-point neural networks.

Flexpoint [12] is a data format based on tensors with an N-bit mantissa storing an integer value in two's complement form, and an M-bit exponent e, shared across all elements of a tensor. The 16-bit Flexpoint closely matches 32-bit floating-point format in training several deep learning models without modifying the models or their hyper-parameters. However, Flexpoint presents potential limitation in performance and efficiency when compared to more aggressive quantization schemes.

Low-Precision Fixed-Point Inference. The quantization method presented in the recent Aster [15] tool assigns mixed fixed-point precision to the neural networks that solve regression tasks, while guaranteeing a provided error bound. Aster optimizes the number of bits needed to implement a network and generates more efficient fixed-point code for custom hardware such as FPGAs. However, the tool focuses only on neural network controllers that are typically regression models. Another solution to synthesize fixed-point code based on constraint generation is described in [5]. The proposed tool generates a system of constraints with integer variables that can be solved by an SMT solver. Consequently, the solution to this system give the minimal number of bits required for each neuron and each synaptic weight. Unlike our tool which handle several type of neural network layers, this approach is limited to fully connected layers.

The work presented in [9] studied the effect of limited precision data representation and computation on neural network training. their results show that substituting floating-point units with fixed-point arithmetic circuits comes with significant gains in the energy efficiency and computational throughput, while potentially risking performance of the neural network. Shiftry [13] is a compiler from high-level floating-point ML models to fixed-point C programs with 8-bit and 16-bit integers, with lower memory requirements. It uses a data-driven float-to-fixed procedure and a RAM management mechanism.

8 Conclusion and Future Work

In this article, we have introduced a new method to generate fixed-point code for a DNN written in `Tensorflow 2.0` with formal guarantees on error bounds. This technique has been implemented in a prototype tool, `Popinns`. The originality of our method is to rely on a formal semantics describing the propagation of the round-off errors throughout the network. Consequently, we may minimize the size of the fixed-point formats and ensure that an error threshold on the results is satisfied by solving a system of constraints. We have shown that the time needed for code synthesis (including constraint solving) remains small: about 5 seconds for a network with 10 000 weights.

In future work, we aim at adding more kinds of layers to our tool and optimize the execution-time of the code generated. Additionally, we aim to synthesize VHDL code to implement directly the neural networks on FPGA or ASIC circuits. As reducing the energy footprint of programs is a subject of interest to us, our goal is to optimize our fixed-point generated codes by using a fixed-point library that provides mixed precision instead of the one used in this article. We also want to use other high-performance energy measurement tools on embedded architectures. Another perspective is to compare `Popinns` to other tools enabling to translate floating-point DNN into fixed-point [17], even if these tools do not provide formal bounds on the errors introduced by the translation.

Acknowledgments

This work is partly funded by Grant Readynov n°21009519 of Region Occitanie (Project RAHW).

References

[1] ANSI/IEEE 2008. *IEEE Standard for Binary Floating-point Arithmetic*. ANSI/IEEE.

[2] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885.

[3] Dorra Ben Khalifa and Matthieu Martel. 2022. Constrained Precision Tuning. In *8th International Conference on Control, Decision and Information Technologies, CoDIT*. 230–236.

[4] Dorra Ben Khalifa and Matthieu Martel. 2023. On the Functional Properties of Automatically Generated Fixed-Point Controllers. In *9th International Conference on Control, Decision and Information Technologies, CoDIT*. IEEE, 786–791.

[5] Hanane Benmaghnia, Matthieu Martel, and Yassamine Seladji. 2022. Code Generation For Neural Networks Based On Fixed-Point Arithmetic. *ACM Trans. Embed. Comput. Syst.* (sep 2022). https://doi.org/10.1145/3563945

[6] Luiz Henrique de Figueiredo and Jorge Stolfi. 2004. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms* 37, 1-4 (2004), 147–158.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 4963)*. Springer, 337–340.

[8] Quentin Ferro, Stef Graillat, Thibault Hilaire, Fabienne Jézéquel, and Basile Lewandowski. 2022. Neural Network Precision Tuning Using Stochastic Arithmetic. In *Software Verification and Formal Methods for ML-Enabled Autonomous Systems - 5th International Workshop, FoMLAS 2022, and 15th International Workshop, NSV (Lecture Notes in Computer Science, Vol. 13466)*. Springer, 164–186.

[9] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*. JMLR.org, 1737–1746.

[10] Arnault Ioualalen and Matthieu Martel. 2019. Neural Network Precision Tuning. In *Quantitative Evaluation of Systems, 16th International Conference, QEST (Lecture Notes in Computer Science, Vol. 11785)*. Springer, 129–143.

[11] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-Scale Video Classification with Convolutional Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. IEEE Computer Society, 1725–1732.

[12] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 1742–1752.

[13] Aayan Kumar, Vivek Seshadri, and Rahul Sharma. 2020. Shiftry: RNN inference in 2KB of RAM. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 182:1–182:30.

[14] Christoph Quirin Lauter and Anastasia Volkova. 2020. A Framework for Semi-Automatic Precision and Accuracy Analysis for Fast and Rigorous Deep Learning. In *27th IEEE Symposium on Computer Arithmetic, ARITH 2020*. IEEE, 103–110.

[15] Debasmita Lohar, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. 2023. Sound Mixed Fixed-Point Quantization of Neural Networks. *ACM Trans. Embed. Comput. Syst.* 22, 5s (2023), 26 pages. https://doi.org/10.1145/3609118

[16] Benoit Lopez. 2014. *Implémentation optimale de filtres linéaires en arithmétique virgule fixe. (Optimal implementation of linear filters in fixed-point arithmetic)*. Ph. D. Dissertation. Pierre and Marie Curie University, Paris, France.

[17] Norbert Mitschke, Michael Heizmann, Klaus-Henning Noffz, and Ralf Wittmann. 2019. A Fixed-Point Quantization Technique for Convolutional Neural Networks Based on Weight Scaling. In *2019 IEEE Int. Conf. on Image Processing, ICIP 2019*. IEEE, 3836–3840.

[18] David Monniaux. 2016. A Survey of Satisfiability Modulo Theory. In *Computer Algebra in Scientific Computing - 18th International Workshop, CASC (Lecture Notes in Computer Science, Vol. 9890)*. Springer, 401–425.

[19] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI (Lecture Notes in Computer Science, Vol. 9351)*. Springer, 234–241.

[20] Laura Titolo, César A. Muñoz, Marco A. Feliú, and Mariano M. Moscato. 2018. Eliminating Unstable Tests in Floating-Point Programs. In *Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11408)*. Springer, 169–183.

[21] S. Weidman. 2019. *Deep Learning from Scratch: Building with Python from First Principles*. O'Reilly Media.

[22] Randy Yates. 2020. *Fixed-Point Arithmetic: An Introduction*. Technical Report. Digital Signal Labs.

[23] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, and Yunji Chen. 2020. Fixed-Point Back-Propagation Training. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020*. Computer Vision Foundation / IEEE, 2327–2335.

Received 2024-02-29; accepted 2024-04-01