Rigorous Floating-Point to Fixed-Point Quantization of Deep Neural Networks on STM32 Micro-controllers

Dorra Ben Khalifa, Matthieu Martel

Rigorous Floating-Point to Fixed-Point Quantization of Deep Neural Networks on STM32 Micro-controllers

Dorra Ben Khalifa[1] and Matthieu Martel[2,3]

***Abstract*— Embedding artificial intelligence onto low-power devices is a challenging task that has been partially overcome by recent advances in machine learning and hardware design. Currently, deep neural networks can be deployed on embedded targets to perform various tasks such as speech recognition, object detection or human activity recognition. However, it is still possible to optimize deep neural networks on embedded devices. These optimizations mainly concern energy consumption, memory and real-time constraints, but also easier deployment at the edge. In addition, there is still a need for a better understanding of what can be achieved for different use cases. This work focuses on the quantization and deployment of deep neural networks on low-power 32-bit micro-controllers. In this article, the quantization method used is based on solving an integer optimization problem derived from the neural network model and concerning the accuracy of the computations and results at each point of the network. We evaluate the performance of our quantization method on a collection of neural networks measuring the analysis time and time-to-solution improvement between the floating- and fixed-point networks, considering a typical embedded platform employing a STM32 Nucleo-144 micro-controller.**

***Index Terms*— Embedded systems; artificial intelligence; constraint generation; quantization; power consumption; micro-controllers**

I. Introduction

In recent years, Deep Neural Networks (DNN) have demonstrated their ability to solve challenging problems in many fields including audio recognition, image classification, or human activity monitoring [1]. One of the well-known disadvantages of DNNs is that they are time, memory and energy intensive, as the number of operations and parameters increases with the complexity of the model architecture.

Due to their low power consumption and small size, embedded systems have also been widely used in automatic controls, wearable devices, home appliances and many others. However, mapping the resulting DNNs onto fast and compact embedded systems remains a challenging problem that seriously hinders the application and development of neural network technology in embedded platforms [2]. A further difficulty lies in the fact that these DNNs are usually trained on a desktop computer with high computing power, before being ported to the target architecture, which has low computing power. It is then necessary to perform this arithmetic conversion without degrading the performance of the network. Quantization is a technique that significantly reduces the memory footprint. It involves reducing the number of bits used to code each model weight, so that the total memory footprint is reduced by the same factor. Quantization also makes it possible to use fixed-point rather than floating-point coding [3]. Several advantages are possible if operations are carried out using integer rather than floating-point data types. An important advantage is that integer operations require far fewer computations on most processor cores, including micro-controllers especially in cases where there is no Floating-Point Unit (FPU) available, so floating-point instructions have to be emulated in software, resulting in a significant overhead.

In this article, we present a method to synthesize fixed-point code from the floating-point description of a DNN. Our approach consists of generating a system of constraints describing the propagation of the fixed-point arithmetic errors throughout a DNN and of finding an optimal solution. The constraints involve only integer quantities and can be efficiently solved by a Satisfiability Modulo Theories (SMT) optimizing solver (we use `Z3` in practice [4].) A special constraint specifies the error threshold tolerated on the outputs. As a result, we obtain the minimal fixed-point formats with regard to our cost function, which guarantee that the error threshold is respected. Our technique is implemented in a tool, `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. Since our aim is to integrate neural networks into low-power devices, in this article

[1] Dorra Ben Khalifa is with Fédération ENAC ISAE-SUPAERO ONERA, University of Toulouse, France `dorra.ben-khalifa@enac.fr`

[2] Matthieu Martel is with LAMPS Laboratory, University of Perpignan Via Domitia, France `matthieu.martel@univ-perp.fr`

[3] Matthieu Martel is also with the Numalis company, Bureaux du Polygone, 265 Avenue. des États du Languedoc, Montpellier, France.

we evaluate our neural networks synthesized on low-power 32-bit micro-controllers. The results show that our fixed-point neural networks consume less time and energy on an ARM micro-controller than their original floating-point version. The results of the mean relative errors measured during evaluations of the fixed-point versus floating-point versions of the model are also presented.

The rest of this article is organized as follows. Section II discusses the related work. Section III presents an overview of our fixed-point synthesis tool for DNNs. In section IV, we review the results obtained by executing DNNs on a STM32 micro-controller board. Section V concludes this work and discusses future perspectives.

II. Related Work

As there is a large body of work on efficient neural network inference, we focus here on a representative selection of the techniques closest to our method, which convert neural networks from floating-point to fixed-point arithmetic while guaranteeing a certain error threshold on the resulting DNN. We also omit work on fixed-point code generation for numerical programs.

The work proposed in [5] 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 in [6] 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 our tool goes further by converting floating-point formats to fixed-point ones and targeting embedded architectures such as STM32 micro-controllers. Ferro et al. [7] proposed a floating-point auto-tuning 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.

Recently, the quantization method presented in the recent Aster [8] 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 [9]. 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 [10] 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 [11] 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.

III. Neural Network Quantization Method Based Constraint Generation

In this section, we give an overview of the technique implemented in our tool, `Popinns`, and how it proceeds to synthesize the fixed-point program corresponding to a given neural network.

`Popinns` takes as entry a Tensorflow model[1] and translates it into its 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. For example, the code below defines a model made of a convolutional layer followed by a dense layer with ReLU (the flatten layer translates the matrix resulting from the convolution into a vector.) This model takes as inputs images of size `height` $\times$ `width`$= 16 \times 16$ and classify them in different classes (`numclass`$= 6$). As indicated further in Table I, this model has 404 parameters.

[1] `https://www.tensorflow.org/`

```
height = 16 ; width = 16 ;
num_classes = 6
input_shape = (height, width, 1)
model = keras.Sequential(
    [
    keras.Input(shape=input_shape),
    layers.Conv2D(1, kernel_size=(3, 3)),
    layers.Flatten(),
    layers.Dense(num_classes,
                 activation="relu")
    ]
)
```

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

```
threshold = 8
popinns(model,1,height,width,imgs,threshold)
```

In the sequence above `imgs` is an array containing several images of size `height`×`width`. It corresponds typically to a subset of the training set and is used to perform the dynamic range analysis. This 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 the most significant bits of the values arising at each point of the model (this step is crucial to to generate the system of constraints described in the next paragraph)

Finally, `threshold` denotes the accuracy required for the fixed-point model which is set by the user. In our example, an accuracy of 2^{-8} is required, which means that the errors between the outputs of the original model and those of the fixed-point code synthesized by `Popinns` must be less than 2^{-8}.

Our key approach is based on a formal semantics describing the propagation of the errors through the computations performed by the DNN [12]. From these semantics, we deduce a system of constraints made of 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 unknowns are the fixed-point formats of the numbers arising in the computations and a cost function is used to indicate to the underlying optimizing solver (we use the `Z3` SMT solver [4]) to find the minimal sizes for the fractional parts of the fixed-point numbers, while ensuring that the accuracy threshold is respected. This system of constraints is described in [12].

Once the constraint system has been solved (in a few seconds in our experiments), `Popinns` uses the fixed-point formats found by the solver to synthesize a fixed-point `C` code relying on the `Fixmath` library[2] for the fixed-point operations.

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

A fixed-point number is represented by a k-bit signed integer X, combined with a scale factor $f \in \mathcal{Z}$. Then X represents the real value x defined by

$$x = X \cdot 2^{-f} . \tag{1}$$

We denote $Q_{i,f}$ the format of a given fixed-point number represented using a k-bit integer associated to a scaling factor f , where $k = i + f$. In `Popinns`, the formats $Q_{i,f}$ of the fixed-point numbers and variables are determined thanks to the range analysis which yields the sizes i of the integer part and thanks to the solution to the system of constraints which gives the sizes f of the fractional parts.

```
[...]
// -- convolution
x[1][0][0][0] = fx_addx(fx_addx(fx_addx(
    fx_addx(fx_addx(fx_addx(fx_addx(fx_addx(
    fx_addx(0,fx_mulx(147580,x
    [0][0][0][0],18)),fx_mulx(124002,x
    [0][0][0][1],18)),fx_mulx(-147646,x
    [0][0][0][2],18)),fx_mulx(98488,x
    [0][0][1][0],18)),fx_mulx(-76541,x
    [0][0][1][1],18)),fx_mulx(-33375,x
    [0][0][1][2],18)),fx_mulx(-10303,x
    [0][0][2][0],18)),fx_mulx(-104803,x
    [0][0][2][1],18)),fx_mulx(-63949,x
    [0][0][2][2],18)) ;
// -- conversion Q13,18 -> Q18,13
x[1][0][0][0] = fx_xtox(x[1][0][0][0],18,13);
[...]
// -- FC Layer
tmp = 0;
for (int j=0;j<49;j++) {
  tmp = fx_addx(tmp,fx_mulx(W_3[0*49+j], x
      [3][0][j][0],13));};
  // fx_k_d_i=13 f_k_i=15 fy_k_d_i=8
  x[4][0][0][0] = RELU(fx_xtox(tmp,15,8));
[...]
```

An excerpt from this code is given hereafter for our example. The first statements are for the convolution. The values 147580, 124002, ... correspond to the coefficients of the convolutional kernel translated in the adequate fixed-point formats. The last statements correspond to the fully connected layer and ReLU. The comment in the for loop indicate that the inputs `x`, computing precision and output `y` of the first neuron of the layer respectively are $Q_{18,13}$, $Q_{16,15}$ and $Q_{23,8}$ (for the sake of conciseness, we have discarded these comments for the convolutional layer but they are present in `Popinns`'s output).

Evaluating the floating-point and fixed-point version of our example model, for a particular yet representative input, yields the following results:

$$\begin{aligned} y_{\text{float}} = \big(& -0.100635, -0.069120, -0.037741, \\ & -0.109075, -0.039620, 0.071354 \big) \end{aligned}$$

and

$$y_{\text{fix}} = \big(\begin{array}{l} -0.101562, -0.070312, -0.031250, \\ -0.117188, -0.023438, 0.062500 \big) \end{array}$$

In particular, we can observe that the dominant class is clearly preserved.

IV. Experimental Evaluation

Since our goal is to execute the DNNs synthesized by `Popinns` on embedded architectures, we present in this section the results obtained in terms of execution time, energy consumption and average relative error between the fixed-point version generated by our tool and the original floating-point version.

A. Experimental Setup

The device evaluation for our benchmarks is done using a 32-bit Arm® Cortex®-M3 CPU (120 MHz max) with 1 MByte of flash program memory and 128Kbyte of RAM based STM32F2072G micro-controller[3]. We used Miosix[4], an operating system kernel designed to run on 32bit micro-controllers. 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 the `Popinns` setup, we evaluate 10 neural networks with different error thresholds: 6, 8, 10 and 12 bits, given by the user. This allows us to determine the threshold that gives the smallest relative error between the floating-point and fixed-point versions of the neural network. For experiments measuring execution time and energy consumption, we have chosen a single error threshold, which is considered the best requirement for obtaining the lowest relative error. For experiments measuring execution time and energy consumption, we have chosen a single error threshold, which is considered the best requirement and gives the lowest relative error.

For power consumption 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. It performs consumption averaging (static measurement up to 200 mA) as well as real-time analysis (dynamic measurement up to 50 mA with 100 kHz bandwidth). Figure 1 depicts how to power the X-NUCLEO-LPM01A via its micro-USB port and how to use wires to connect it to the target STM32 Nucleo-144 board.

[3] `https://www.st.com`
[4] `https://miosix.org/`

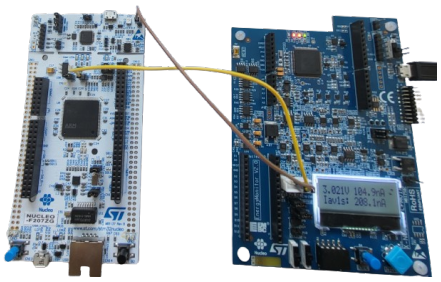

Fig. 1: Connecting the power shield to the target board using basic connectors.

NN	#P	# Cl	Size Input	CV	MP	FL	FC	RL
1	122	6	196	2	1	1	1	2
2	182	8	100	1	1	2	2	2
3	404	6	256	1	-	1	1	-
4	505	5	100	-	-	1	1	-
5	600	4	144	-	-	2	2	-
6	620	10	256	1	-	1	1	1
7	798	4	256	1	-	1	1	1
8	1020	10	144	1	-	1	1	2
9	1318	12	100	-	-	3	3	-
10	1586	8	256	1	-	1	1	1

TABLE I: Description of the neural networks used in our experiments. **NN** is the reference of the network, **#P** indicates its number of parameters, **#Cl** the number of recognized classes, **CV**, **MP**, **FL**, **FC** and **RL** respectively indicate the number of convolutional, maxpool, flatten, fully connected and ReLU layers of the networks.

B. Experimental Results

In our experiments we used 10 different neural networks as shown in Table I. These networks are composed of a mix of the layers and they have about 100 parameters (case of the smallest network, NN1) and up to 1600 parameters (case of the largest network, NN10). Table II focuses on the mean relative errors computed by comparing the outputs of the fixed-point codes with the results of the original floating-point codes. These errors were computed for several thresholds (6, 8,10 and 12 bits) for 6 of our networks, NN1 to NN6. The

NN	Thresh=6	Thresh=8	Thresh=10	Thresh=12
1	0.05	0.018	0.022	0.003
2	0.03	0.011	0.014	0.0086
3	0.08	0.021	0.057	0.021
4	0.038	0.044	0.031	0.020
5	0.05	0.021	0.051	0.035
6	0.068	0.011	0.03	0.0085

TABLE II: Mean relative errors measured on evaluations of the fixed-point versions of the models compared to their floating-point versions, for models NN1 to NN6 described in Table I.

remaining networks, NN7 to NN10, could not be run on the STM32 board due to the small amount of flash memory available in this micro-dcontroller, which is not sufficient for the sizes of the remaining neural networks. For the majority of the benchmarks, our tool is able to constrain the relative error below 0.1%. These results confirm that the relative error measurement depends on the threshold requested by the user and the range of variables computed after a certain number of program executions.

Figure 2 depicts the measurement of execution time and power consumption of the floating-point and fixed-point synthesized neural networks tested on the STM32 Nucleo-144 board. We recall that energy measurements are performed using the X-NUCLEO-LPM01A STM32 power shield.

We observe in the top hand side of Figure 2, that the fixed-point neural networks generated are $2\times$ to $4\times$ faster than the floating-point networks as for NN1 and NN2 respectively. In addition, the fixed-point versions of NN3 to NN6 manage to run on the board in less than 1 second, unlike their floating-point versions, which consume a lot of memory on the board and were therefore unable to run. These results validate the main objective of this work, which is to successfully run optimal versions of DNNs on low-resource architectures, especially since in terms of performance, their behaviors is identical to the original more resource-intensive version. The bottom hand side of Figure 2 shows that the fixed-point neural networks generated by our method consume less energy than their original floating-point versions. For example, for the NN2, the fixed-point model consumes only 0.05 μJ compared with its floating-point version, which consumes $0.2\mu J$. Our biggest neural network NN6 with 620 parameters consumes $0.05\mu J$ in its fixed-point version.

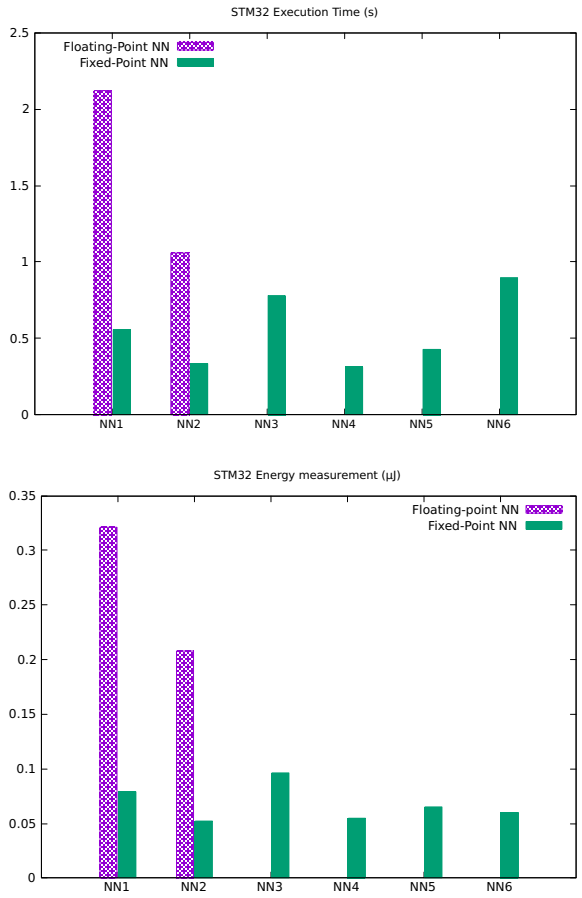

Fig. 2: Measurement of execution time in seconds (top) and power consumption in μJ (bottom) of the floating-point and fixed-point synthesized neural networks tested on the STM32 Nucleo-144 board.

V. Conclusion and Future Work

In this article, we focused on the quantization and deployment of deep neural networks on a low-power STM32 Nucleo-144 micro-controller. Our quantization technique is based on formal semantics describing the propagation of round-off errors through the network. As a result, we can minimize the size of fixed-point formats and guarantee that an error threshold on the results is satisfied. We have also shown that the fixed-point codes generated are less time- and energy-intensive than their original floating-point models.

In future work, we aim at adding more kinds of layers to `Popinns` and optimize the execution time of the code generated. Moreover, we aim at synthesizing VHDL code and implement directly the neural networks on FPGA or ASIC circuits. Another perspective is to

compare `Popinns` to other tools enabling to translate floating-point DNNs into fixed-point while providing formal bounds on the errors introduced by the translation. Finally, we aim at mixing quantization techniques, with pruning techniques [13], [14], which remove the less useful weights of DNNs in order to reduce even more the resources needed for their execution.

REFERENCES

[1] Y. Zhou, S. Moosavi-Dezfooli, N. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, S. A. McIlraith and K. Q. Weinberger, Eds. AAAI Press, 2018, pp. 4596–4604.

[2] P. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," *Sensors*, vol. 21, no. 9, p. 2984, 2021.

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

[4] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[5] A. Ioualalen and M. Martel, "Neural network precision tuning," in *Quantitative Evaluation of Systems, 16th International Conference, QEST*, ser. Lecture Notes in Computer Science, vol. 11785. Springer, 2019, pp. 129–143.

[6] C. Q. Lauter and A. Volkova, "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, 2020, pp. 103–110.

[7] Q. Ferro, S. Graillat, T. Hilaire, F. Jézéquel, and B. Lewandowski, "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*, ser. Lecture Notes in Computer Science, vol. 13466. Springer, 2022, pp. 164–186.

[8] D. Lohar, C. Jeangoudoux, A. Volkova, and E. Darulova, "Sound mixed fixed-point quantization of neural networks," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 5s, 2023.

[9] H. Benmaghnia, M. Martel, and Y. Seladji, "Code generation for neural networks based on fixed-point arithmetic," *ACM Trans. Embed. Comput. Syst.*, sep 2022.

[10] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015*, ser. JMLR Workshop and Conference Proceedings, vol. 37. JMLR.org, 2015, pp. 1737–1746.

[11] A. Kumar, V. Seshadri, and R. Sharma, "Shiftry: RNN inference in 2kb of RAM," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 182:1–182:30, 2020.

[12] D. Ben Khalifa and M. Martel, "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*. ACM, 2024.

[13] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations," *CoRR*, vol. abs/2308.06767, 2023.

[14] A. Vysogorets and J. Kempe, "Connectivity matters: Neural network pruning through the lens of effective sparsity," *J. Mach. Learn. Res.*, vol. 24, pp. 99:1–99:23, 2023.