Neural Network Precision Tuning Using Stochastic Arithmetic

Quentin Ferro, Stef Graillat, Thibault Hilaire, Fabienne Jézéquel, Basile Lewandowski

Neural Network Precision Tuning Using Stochastic Arithmetic

Quentin Ferro[1], Stef Graillat[1], Thibault Hilaire[1], Fabienne Jézéquel[1,2], and Basile Lewandowski[1]

[1] Sorbonne Université, CNRS, LIP6, Paris, F-75005, France
[2] Université Paris-Panthéon-Assas, Paris, F-75005, France
{quentin.ferro, stef.graillat, thibault.hilaire, fabienne.jezequel}@lip6.fr

Abstract. Neural networks can be costly in terms of memory and execution time. Reducing their cost has become an objective, especially when integrated in an embedded system with limited resources. A possible solution consists in reducing the precision of their neurons parameters. In this article, we present how to use auto-tuning on neural networks to lower their precision while keeping an accurate output. To do so, we use a floating-point auto-tuning tool on different kinds of neural networks. We show that, to some extent, we can lower the precision of several neural network parameters without compromising the accuracy requirement.

Keywords: Precision, Neural Networks, Auto-Tuning, Floating-Point, Stochastic Arithmetic

1 Introduction

Neural networks are nowadays massively used and are becoming larger and larger. They often need a lot of resources, which can be a problem, especially when used in a critical embedded system with limited computing power and memory. Therefore it can be very beneficial to optimise the numerical formats used in a neural network. This article describes how to perform precision auto-tuning of neural networks. From a neural network application and an accuracy requirement on its results, it is shown how to obtain a mixed precision version thanks to the PROMISE tool [15]. A particularity of PROMISE is the fact that it uses stochastic arithmetic [38] to control rounding errors in the programs it provides.

Minimizing the format of variables in a numerical simulation can offer advantages in terms of execution time, volume of data exchanged and energy consumption. During the past years several algorithms and tools have been proposed for precision auto-tuning. On the one hand, tools such as FPTuner [8], Salsa [10], Rosa/Daisy [12,11], TAFFO [5], POP [2] rely on a static approach and are not intended to be used on very large codes. On the other hand, dynamic tools such as CRAFT HPC [24], Precimonious [33], HiFPTuner [16], ADAPT [30],

FloatSmith [25], PROMISE [15], have been proposed for precision auto-tuning in large HPC codes. Moreover, tools have been recently developed for precision auto-tuning on GPUs: AMPT-GA [22], GPUMixer [23], GRAM [18]. A specificity of PROMISE lies in the fact that it provides mixed precision programs validated thanks to stochastic arithmetic [38], whereas other dynamic tools rely on a reference result possibly affected by rounding errors. PROMISE has been used in various applications based on linear algebra kernels, but not yet for precision auto-tuning in neep neural networks.

While much effort has been devoted to the safety and robustness of deep learning codes (see for instance [13,34,28,27,32]) a few studies have been carried out on the effects of rounding error propagation on neural networks. Verifiers such as MIPVerify [36] are designed to check properties of neural networks and measure their robustness. However the impact of floating-point arithmetic both on neural networks and on verifiers is pointed out in [43]. Because of rounding errors, the actual robustness and the robustness provided by a verifier may radically differ. In [9,42] it is shown how to control the robustness of different neural networks with greater efficiency using interval arithmetic based algorithms. In [26] a software framework is presented for semi-automatic floating-point error analysis in the inference phase of deep neural networks. This analysis provides absolute and relative error bounds thanks to interval and affine arithmetics.

Precision tuning of neural networks using fixed-point arithmetic has been studied in [3]. Thanks to the solution of a system of linear contraints, the fixed-point precision of each neuron is determined, taking into account a certain error threshold.

In this article, we consider floating-point precision tuning, which is also studied in [20]. Focusing on interpolator networks, i.e. networks computing mathematical functions, the authors propose an algorithm that takes into account a given tolerance δ on the relative error between the assumed correctly computed function and the function computed by the network. The main difference with the present article is the auto-tuning algorithm: in [20] the precision is optimized by solving a linear programming problem, while PROMISE uses a hypothesis-trial-result approach through the Delta-Debug algorithm [41]. Furthermore, the algorithm in [20] relies on a reference result that may be altered by rounding errors, while PROMISE uses stochastic arithmetic for the numerical validation of its results.

Stochastic arithmetic uses for rounding error estimation a random rounding mode: the result of each arithmetic operation is rounded up or down with the same probability. As a remark, another stochastic rounding often used in neural network training and inference uses a probability that depends on the position of the exact result with respect to the rounded ones (see for instance [17,31,39,14,29,35]). This stochastic rounding does not aim at estimating rounding errors, it enables the update of small parameters and avoids stagnations that may be observed with round to nearest.

In this work, we consider tuning the precision of an already trained neural network. One of our contributions is a methodology for tuning the precision of a

neural network using PROMISE in order to obtain the lowest precision for each of its parameters, while keeping a certain accuracy on its results. We present and compare the results obtained for different neural networks: an approximation of the sine function, an image classifier processing the MNIST dataset (2D pictures of handwritten digits), another image classifier using this time convolutional layers and processing the CIFAR10 dataset (3D images of different classes), and a last one introduced in [4] and used in [26] that aims at approximating a Lyapunov function of a nonlinear controller for an inverted pendulum.

After a preliminary reminder on deep neural networks and stochastic arithmetic in Section 2, Section 3 describes our methodology and Section 4 presents our results considering the different neural networks previously mentioned.

2 Preliminary

2.1 Neural Networks

An artificial neural network is a computing system defined by several neurons distributed on L different layers. Generally, we consider dense layers that take as an input a vector and in which the main computation is a matrix-vector product. In this case, from one layer to another, a vector of neurons $x^{(k)} \in \mathbb{R}^{n_k}$ with $k \in \{1, ..., L-1\}$ is transformed into a vector $x^{(k+1)} \in \mathbb{R}^{n_{k+1}}$ by the following equation

$$x^{(k+1)} = g^{(k)}(W^{(k)}x^{(k)} + b^{(k)}) \tag{1}$$

where $W^{(k)} \in \mathbb{R}^{n_{k+1} \times n_k}$ is a weight matrix, $b^{(k)} \in \mathbb{R}^{n_{k+1}}$ a bias vector and $g^{(k)}$ an activation function. The activation function is a non-linear and often monotonous function. Most common activation functions are described below.

- Sigmoid: computes $\sigma(x) = 1/(1 + e^{-x})\ \forall x \in \mathbb{R}$
- Hyperbolic Tangent: simply applies $\tanh(x)\ \forall x \in \mathbb{R}$
- Rectified Linear Unit: $\text{ReLU}(x) = \max(x, 0)\ \forall x \in \mathbb{R}$
- Softmax: normalizes an input vector x into a probability distribution over the output classes. For each element x_i in x, $\text{softmax}(x_i) = e^{x_i} / \sum e^{x_j}$

Figure 1 shows a diagram representation of a neural network with three layers: the input layer, one hidden layer, and the output layer.

Dense layers are sometimes generalized to multidimensional arrays and involve tensor products. Other layers exist such as convolution layers, often used on multi-dimensional arrays to extract features out of the data. To do so, a convolution kernel is applied to the input data to produce the output. Different kinds of layers, for instance pooling layers and flatten layers, do not require weight nor bias. Pooling layers reduce the size of the input data by summarizing it by zones given a function such as the maximum or the average. Flatten layers are intended to change the shape of data, from a 3D tensor to a vector for example and can be used to pass from a convolutional layer to a dense layer.

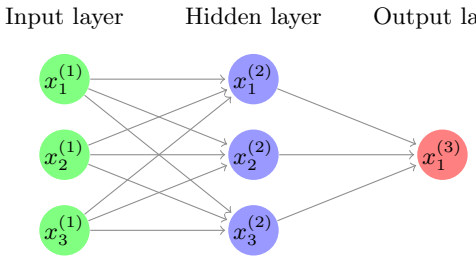

Fig. 1: Neural network with one hidden layer

2.2 Floating-Point Arithmetic

A floating-point number x in base β is defined by :

$$x = (-1)^s \times m \times \beta^e \tag{2}$$

with s its sign being either 1 or 0, m its significand being an integer and e its exponent being also an integer. In this paper, we consider binary floating-point numbers, i.e. numbers in base β=2 that adhere to the IEEE 754 Standard [1]. The IEEE 754 Standard defines different formats with a fixed number of bits for the significand and the exponent. The number of bits for the significand is the precision p, hence the significand can take values ranging from 0 to $\beta^p - 1$. The exponent e ranges from e_{min} to e_{max} with $e_{min} = 1 - e_{max}$ and $e_{max} = 2^{len(e)-1} - 1$, with $len(e)$ the exponent length in bits. The sizes of the three different formats used in this paper, commonly named half, single (or float), and double, are sum up in Table 1. As a remark, another 16-bit format called bfloat16 exists, used for example on ARM NEON CPUs. Thanks to their 8-bit-large exponent, bfloat16 numbers benefit from a wider range, but have a lower significand precision (8 bits instead of 11).

Name	Format	Length	Sign	Significand length[3]	Exponent length
Half	binary16	16 bits	1 bit	11 bits	5 bits
Single	binary32	32 bits	1 bit	24 bits	8 bits
Double	binary64	64 bits	1 bit	53 bits	11 bits

Table 1: Basic binary IEEE 754 formats

2.3 Discrete Stochastic Arithmetic (DSA)

Discrete Stochastic Arithmetic (DSA) is a method for rounding error analysis based on the CESTAC method [37,7]. The CESTAC method allows the estima-

[3] Including the implicit bit (which always equals 1 for normal numbers, and 0 for subnormal numbers). The implicit bit is not stored in memory.

tion of round-off error propagation that occurs when computing with floating-point numbers. Based on a probabilistic approach, it uses a random rounding mode: at each operation, the result is either rounded up or down with the same probability. Using this rounding mode, the same program is run N times giving us N samples $R_1, \ldots, R_N$ of the computed result R. The accuracy of the computed result (i.e. its number of exact significant digits) can be estimated thanks to Student's law. In practice we take $N = 3$ and increasing the sample size does not improve the accuracy estimation. Theoretical elements can be found in [6,37].

The CADNA [21,37,6] (Control of Accuracy and Debugging for Numerical Applications) software[4] implements DSA in codes written in C, C++ or Fortran. It introduces new variable types, the stochastic types. Each stochastic variable contains three floating-point values and one integer being the exact number of correct digits. CADNA can print each computed value with only its exact significant digits. In practice, thanks to operator overloading, the use of CADNA only requires to change declaration of variables and input/output statements.

2.4 The PROMISE software

The PROMISE software[5] aims at reducing the precision of the variables in a given program. From an initial code and a required accuracy, it returns a mixed precision code, lowering the precision of the different variables while keeping a result that satisfies the accuracy constraint. To do so, some variables are declared as custom typed variables that PROMISE recognizes. PROMISE will consider tweaking their precisions. Different variables can be forced to have the same precision by giving them the same custom type. It may be useful to avoid compilation errors or casts of variables.

PROMISE computes a reference result using CADNA and relies on the Delta-Debug algorithm [40] to test different type configurations, until a suitable one lowering the precision while satisfying the accuracy requirement is found. PROMISE provides a transformed program that can mix half, single and double precision. Half precision can be either native on CPUs that support it or emulated thanks to a library developed by C. Rau[6]. PROMISE dataflow is presented in Figure 2. After computing the reference result, PROMISE tries to lower the precision of the variables from double to single, then from single to half, using twice the Delta-Debug algorithm. The accuracy requirement may concern one or several variables (e.g. in an array). PROMISE checks that the number of digits in common between the computed result(s) and the reference result(s) is at least the required accuracy. In the case of several variables, the requirement has to be fulfilled by all of them.

[4] `http://cadna.lip6.fr`
[5] `http://promise.lip6.fr`
[6] `http://half.sourceforge.net`

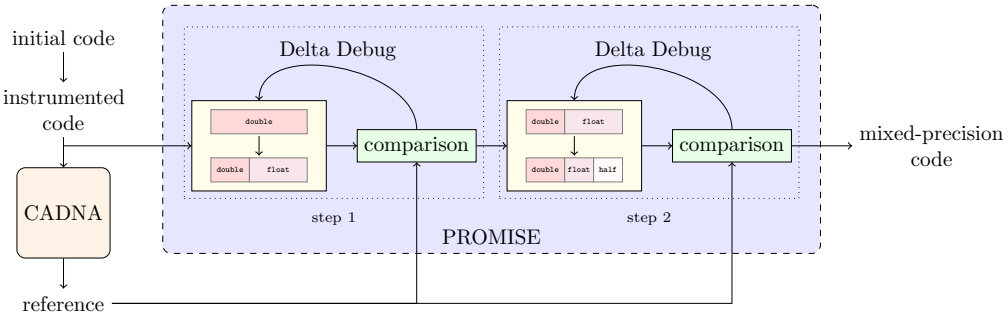

Fig. 2: PROMISE dataflow

3 Methodology

For neural network management, we use Python language with either Keras[7] or PyTorch[8]. Keras and Pytorch are two open-source Python libraries that implement structures and functions to create and train neural network models. Both of them also allow us to save our model in HDF5 (Hierarchical Data Format)[9], a file format designed to store and organize large data. HDF5 uses only two types of objects: datasets that are multidimensional arrays of homogeneous type, and groups, which contain datasets or other groups. HDF5 files can be read by Python programs thanks to the h5py package. The associated data can be manipulated using Pandas[10], a Python library that proposes data structures and operations to manage large amount of data.

Keras is used to develop, train and save our neural network models, except in the case of the inverted pendulum which uses PyTorch. As already mentioned in the introduction, precision tuning is performed on trained models, hence in the inference stage. The process path is summarized in Figure 3. For each neural network, we first convert the HDF file to CSV files using a Python script. The script loads the HDF file, stores the parameters in Pandas DataFrames, and then saves the parameters in CSV files thanks to the Pandas DataFrame *to_csv* function. For each layer that needs it, we create a CSV file with the weights of the layer and a CSV file with the bias of the layer. Indeed, some layers do not need weights nor bias, for example flattening layers that only change the data shape (from 2 dimensions to 1 dimension for example). Secondly, we use the data in the CSV files to create a C++ file, once again thanks to a Python script that reads the CSV files and creates the necessary variables and computation. The translation scripts are based on the work done in the keras2c[11]library. Once the C++ file created, we apply PROMISE on it.

[7] `https://keras.io`
[8] `https://pytorch.org`
[9] `https://www.hdfgroup.org`
[10] `https://pandas.pydata.org`
[11] `https://f0uriest.github.io/keras2c/`

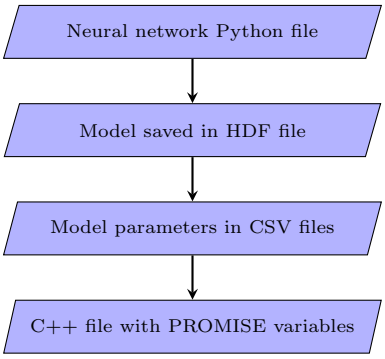

Fig. 3: Flowchart of the translation from a Python neural network to a C++ file with PROMISE variables

4 Experimental results

Results obtained for four different neural networks are presented in this section. For simplicity, in the rest of this article we refer to as a layer all layers except the input one, i.e. the first layer (or layer 1) is the first hidden layer and the last one is the output layer. The total number of layers is then the number of hidden layers +1. For the neural networks using a database, test_data[i] refers to (i+1)th test input provided by the database. PROMISE is applied to each neural network considering one type by neuron (half, single or double), then one type per layer, i.e. all the parameters of a layer have the same precision. In our analysis, the difference between the two approaches lies in the number of different type declarations in the code. However it must be pointed out that, in dense layers, having one type per neuron implies independent dot products, whereas having one type per layer would enable one to compute matrix-vector products that could perform better.

In our experiments, the input is in double precision. In accordance with Figure 2, for any neural network, the reference value is the value computed at the very first step of PROMISE. All the results presented in this section have been obtained on a 2.80 GHz Intel Core i5-8400 CPU having 6 cores with 16GB RAM except indicated otherwise.

4.1 Sine Neural Network

To approximate the sine function, we use a classical densely-connected neural network with 3 layers. It is a toy problem, since using a neural network to compute sine is not necessary. However this simple example validates our approach. The tanh activation function is used in the 3 different dense layers. The layers have respectively 20, 6 and 1 neuron(s) and the input is a scalar value x. Figure 4 presents the computation carried out by the neural network, considering one type per neuron. Colored variables are PROMISE variables, the precision of

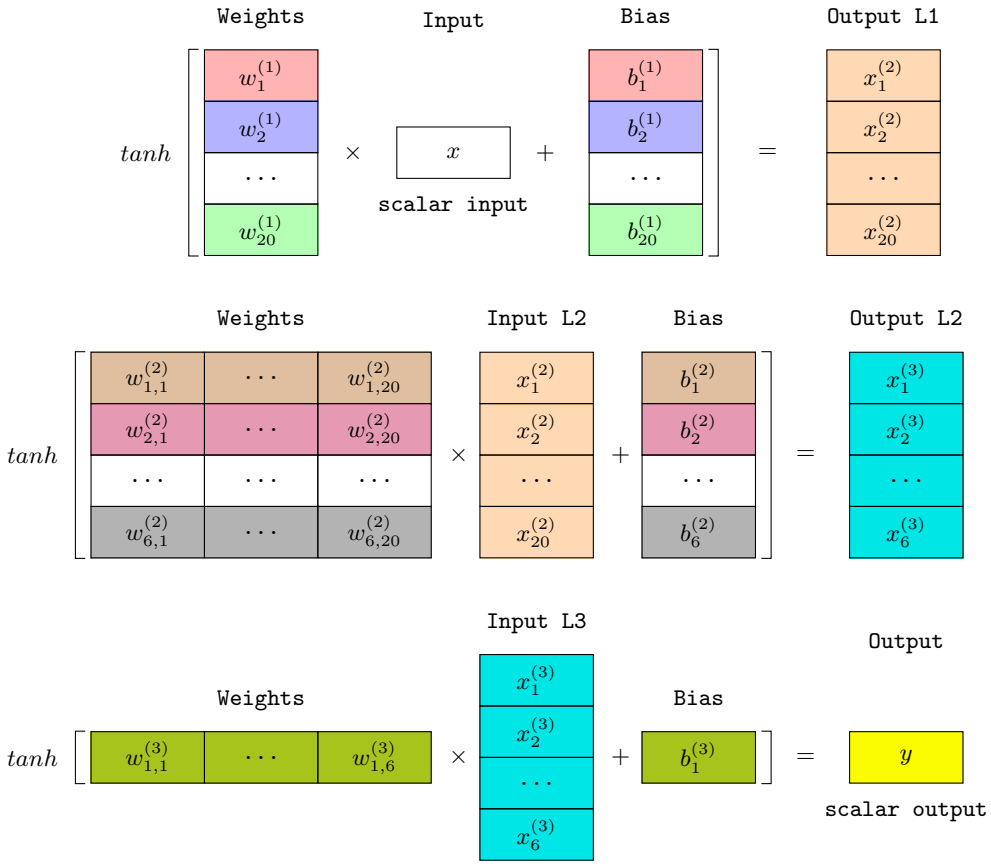

Fig. 4: Computation carried out in the sine neural network

which can be tweaked. Variables with the same color have the same precision. The parameters of a neuron (weight(s) and bias) have the same color, hence the same type. The output type of each layer is also tuned. In this example, 30 different types are set in total.

Figure 5 displays the distribution of the different types with input value 0.5 considering one type per neuron. The x-axis presents the required accuracy on the results, i.e. the number of significant digts in common with the reference result computed using CADNA. We can notice the evolution of the distribution depending on the number of exact significant digits required on the result. As expected, first we only have half precision variables, then some of them start to be in single, then in double precision, until eventually all of them are in double precision. Therefore, requiring the highest accuracy is not compatible with lowering the precision in this neural network. But still, a good compromise can be found, since we only have single and half precision variables for a required accuracy up to 7 digits, and still have 1/3 of single precision variables for a required accuracy up to 9 digits.

Figure 5 also presents the total runtime in seconds of PROMISE for each required accuracy. It consists of the time to compute the reference result, and the time to apply the Delta-Debug algorithm twice (from double to single precision then from single to half precision), compiling and executing the tested distribution each time. It can be noticed that the runtimes (less than 2 minutes) remain reasonable given the 3^{30} possibilities.

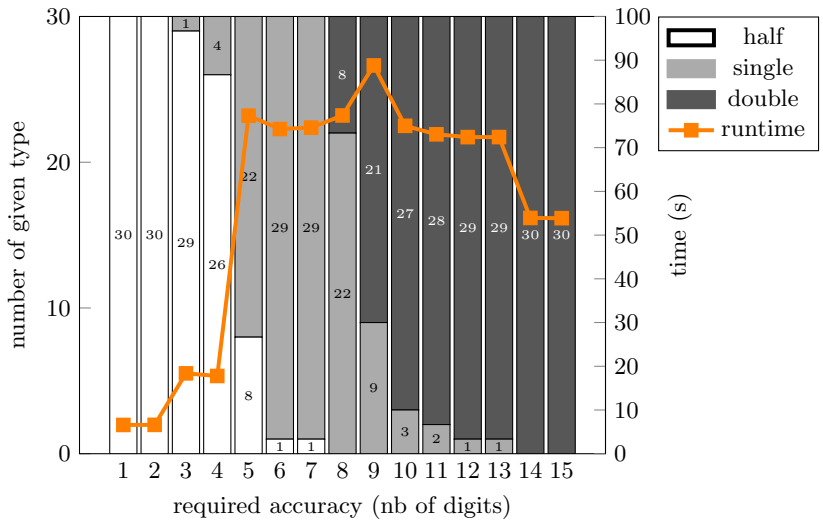

Fig. 5: Number of variables of each type and runtime for the sine neural network with input value 0.5

Figure 6 shows the type distribution considering one precision per layer. This approach per layer enables one to decrease the execution time of PROMISE, but it does not really help lowering the precision of the network parameters. Indeed, each time a parameter in a layer requires a higher precision, all the parameters of the same layer pass in higher precision. But still, it can be noticed that the first layer (that represents 2/3 of the neurons) stays in half precision for a required accuracy up to 4 digits.

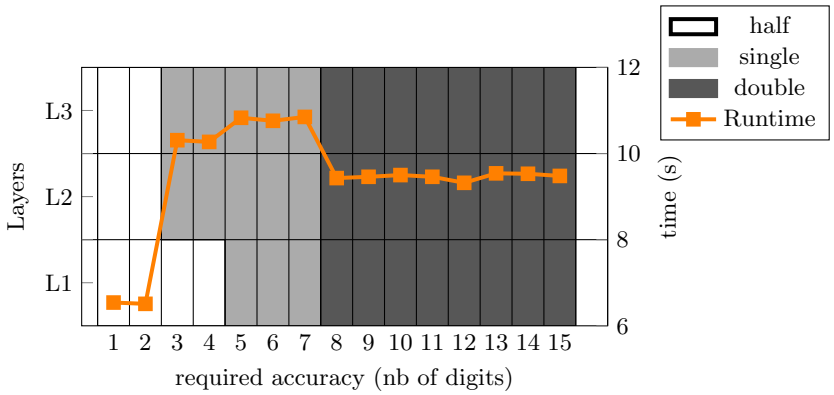

Fig. 6: Precision of each layer for the sine neural network with input value 0.5

Figures in Appendix A.1 and Appendix A.2 show that the input value can have a slight impact on the type distribution. We compare the results for two input values randomly chosen: 0.5 and 2.37. From Figure 5, with input value 0.5, PROMISE provides a type distribution with 8 half precision variables and 22 single precision variables for a required accuracy of 5 digits. From Appendix A.1, with input value 2.37, only 4 half precision variables are obtained for a required accuracy of 5 digits. Actually, if 3, 4, or 5 digits are required, less variables are in half precision than with input 0.5. But if 8 digits are required, one variable remains in half precision with input 2.37, while no variable is in half precision with input 0.5. Nevertheless, the type distribution with respect to the required accuracy remains globally the same, parameters all starting as half precision variables, then passing to higher precision.

4.2 MNIST Neural Network

Experiments have been carried out with an image classification neural network processing MNIST data of handwritten digits[12]. This neural network also uses classical dense layers. The main difference in this case is that the entry is a vector of size 784 (flatten image) and the output is a vector of size 10. This neural

[12] `http://yann.lecun.com/exdb/mnist`

network consists of two layers: the first one with 64 neurons and the activation function ReLU, and the second one with 10 neurons and the activation function softmax which provides the probability distribution for the 10 different classes. Considering, as previously, one type per neuron, plus one type for the output of each layer, 76 different types have to be set either to half, single, or double precision.

Figure 7 shows the type distribution considering one type per neuron with the input image test_data[61]: the 62^{nd} test data out of the 10,000 provided by MNIST. The x-axis represents the required accuracy on the output consisting in a vector of size 10. The maximum possible accuracy on the output is 13 digits, higher expectation could not be matched. Such high accuracy is nonetheless not realistic because not necessary for a classifier, but exhaustive tests have been performed.

The main difference with the sine neural network lies in the fact that a significant number of variables stay in half precision no matters the required accuracy. Depending on the input, around 50% of the variables can stay in half precision and sometimes nearly 60% as shown in Appendix B.2. Thus, applying PROMISE to this neural network, even when requiring the highest accuracy, can help lowering the precision of its parameters. The variables that keep the lowest precision are not the same depending on the input, but they always belong to the first layer. Hence, the first layer seems to have less impact on the output accuracy than the second one.

The runtimes also reported in Figure 7 are much higher than for the sine approximation network. For the majority of the accuracy requirements, more than 15 minutes are necessary to obtain a mixed precision version of the neural network. The MNIST neural network has one layer less than the sine approximation network, but more neurons. Since we consider one precision per neuron, the number of possible type configurations (3^{76}) is much higher, hence the runtime difference. However, the execution time of PROMISE remains reasonable and performing such a tuning by hand would have been much more time consuming.

With both the sine neural network and MNIST neural network, PROMISE execution time tends to increase with the accuracy requirement. This can be explained by the Delta-Debug algorithm in PROMISE. As previously described, PROMISE firstly checks whether the accuracy requirement can be satisfied with double precision. Then, thanks to the Delta-Debug algorithm, PROMISE tries to lower the precision of most variables from double to single, and this can be very fast if single precision is enough to match the required accuracy. Finally, PROMISE tries to transform the single precision declarations into half precision ones, and again this can be fast if half precision is suitable for all these declarations. The number of programs compiled and executed by PROMISE tends to increase with the required accuracy on the results. For instance, in the case of MNIST neural network, if 1 or 2 digits are required, 18 type configurations are tested by PROMISE, whereas if 7 digits are required, 260 configurations are tested.

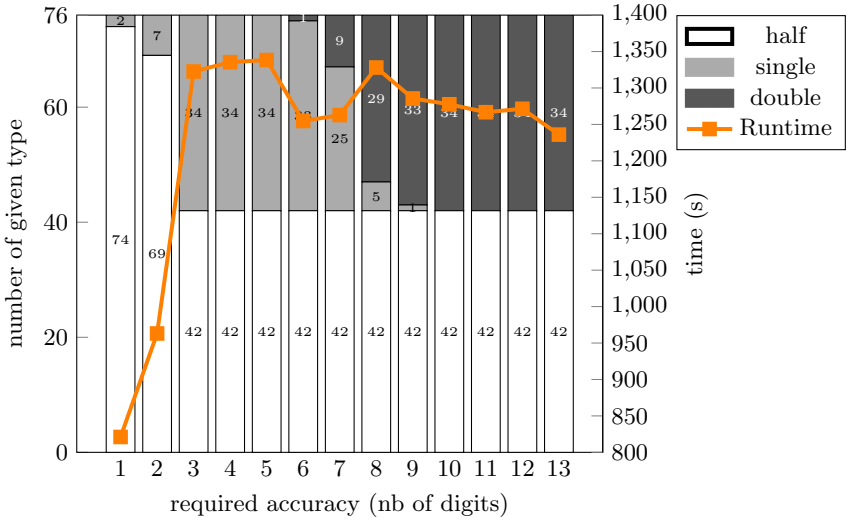

Fig. 7: Number of variables of each type and runtime for MNIST neural network with test_data[61] input

Results obtained considering one precision per layer are presented in Appendix B.1. The analysis is similar to the one previously given for the sine neural network. With the approach per layer, the execution time of PROMISE is lower than with the approach per neuron. But this approach forces some variables to be declared in higher precision. It can be noticed that with the approach per layer, both layers have the same precision. All the network parameters share the same type. Appendix B.2 and Appendix B.3 present the results obtained with another input. Like with the sine neural network, changing the input induces slight changes in the type configurations provided by PROMISE. However the same trend can be observed.

4.3 CIFAR Neural Network

The neural network considered here is also an image classifier, but this time processing the CIFAR10 dataset. CIFAR[13] is a dataset having 100 classes of colored images and the CIFAR10 dataset is reduced to 10 classes. Because images are of size 32x32x3 (32 wide, 32 high, 3 color channels), the network input is a 3D tensor of shape (32, 32, 3). The neural network consists of 5 layers: a convolutional layer having 32 neurons with activation function ReLU, followed by a max pooling of size 2x2, a convolutional layer having 64 neurons with activation function ReLU, a flatten layer, and finally a dense layer of 10 neurons with activation function softmax. Taking into account one type per neuron and one type for each layer output, 111 types can be set.

[13] `https://www.cs.toronto.edu/~kriz/cifar.html`

Results presented here have been obtained on a 2.80 GHz Intel Core i9-10900 CPU having 20 cores and 64GB RAM. PROMISE provides transformed programs taking into account a required accuracy. The maximum possible accuracy on the results is 13 digits. Although such a high accuracy is not necessary in a classifier network, exhaustive tests have been performed, like for the MNIST neural network. Results reported here refer to two input images out of the 10,000 provided by CIFAR10. Figure 8 and Appendix C.2 present the type configurations given by PROMISE with respectively test_data[386] and test_data[731], considering one type per neuron. Appendix C.1 and Appendix C.3 show the results obtained with the same input images considering one type per layer.

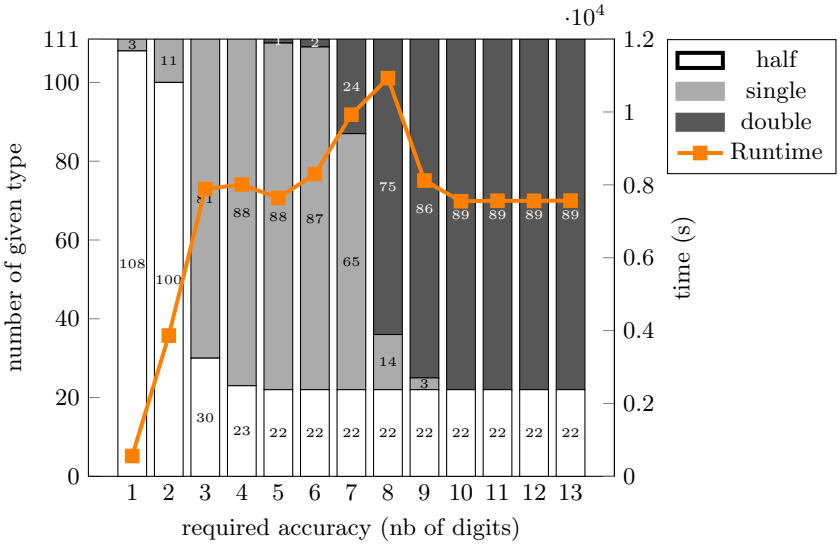

Fig. 8: Number of variables of each type and runtime for CIFAR neural network with test_data[386] input

Again, PROMISE runtime tends to increase with the required accuracy. As already mentioned in 4.2, this can be explained by the Delta Debug algorithm. As previously observed, considering one type per layer results in lower PROMISE runtimes, but often in uniform precision programs. Again, the input image slightly impacts the type configuration provided by PROMISE, the same trends can be observed.

Experiments have been carried out with neural networks also processing CIFAR10, but with more layers (up to 8 layers). Again, PROMISE could provide suitable type configurations taking into account accuracy requirements. However PROMISE execution (that includes the compilation and execution of various programs) makes exhaustive tests more difficult with such neural networks. Possible

PROMISE improvements described in Section 5 would enable precision tuning in larger neural networks that are themselves time consuming.

4.4 Inverted pendulum

We present here results obtained with a neural network introduced in [4] in the context of reinforcement learning for autonomous control. In [4] methods are proposed for certified approximation of neural controllers and this neural network, related to an inverted pendulum, is used in a program that provides an approximation of a Lyapunov function. This neural network consists of 2 dense layers and uses the tanh activation function. The input is a state vector $x \in \mathbb{R}^2$ and the output, a scalar value in $\mathbb{R}$ is an approximated value of the Lyapunov function for the state vector input. The first layer has 6 neurons and the second layer only one. Given the proximity between the two neural network models, results are expected to be close to the ones obtained for the sine approximation.

Figure 9 presents both the distribution of the different precisions and the execution time of PROMISE with respect to the accuracy requirement for input (0.5, 0.5). We consider here one type per neuron. As expected, the trend observed for the type configurations is the same as with the sine approximation. As the required accuracy increases, the precision of the network parameters also increases. If one digit is required, all the parameters can be declared in half precision, and if at least 11 digits are required all the parameters must be in double precision. The runtime remains reasonable whatever the required accuracy. As previously observed, the runtime tends to increase with the required accuracy because of the number of type configurations tested by PROMISE.

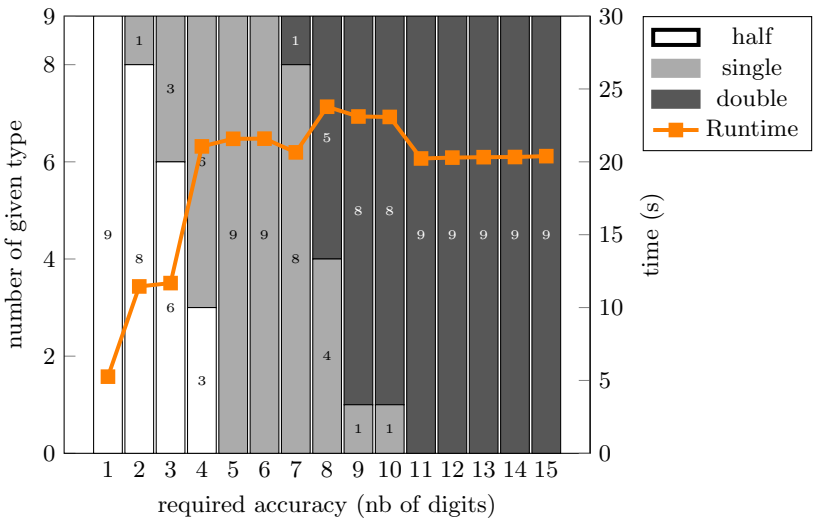

Fig. 9: Number of variables of each type and runtime for the pendulum neural network with input (0.5,0.5)

Appendix D.1 presents the results obtained considering one type per layer. Again, with this approach, the execution time of PROMISE is lower, but most configurations are actually in uniform precision. Results in Appendix D.2 and Appendix D.3 refer to an input consisting of two negative values (-3,-6). Again, changing the input induces no significant difference.

5 Conclusion and perspectives

We have shown with different kinds of neural networks having different types of layers how to lower the precision of their parameters while still satisfying a desired accuracy. Considering one type per neuron, mixed precision programs can be provided by PROMISE. Considering one type per layer enables one to reduce PROMISE execution time, however this approach often leads to uniform precision programs. It has been observed that with both approaches input values have actually a low impact on the type configurations obtained.

We plan to analyse the execution time of the mixed precision programs obtained with PROMISE on a processor with native half precision. Other perspectives consist in improving PROMISE. Another accuracy test more adapted to image classification networks could be proposed. We could improve the Delta-Debug algorithm used in PROMISE. Optimizations of the Delta-Debug algorithm are described in [19], including the parallelization potential. We could also consider the parallelization of PROMISE itself, i.e. applying PROMISE to different parts of a code in parallel. The extension of PROMISE and CADNA to other floating-point formats such as bfloat16 is another perspective. Taking benefit from the GPU version of CADNA, PROMISE could also be extended to GPUs. Floating-point auto-tuning in arbitrary precision is also a possible perspective that would enable the automatic generation of programs with a suitable type configuration for architectures such as FPGAs.

Acknowledgements

This work was supported by the InterFLOP (ANR-20-CE46-0009) project of the French National Agency for Research (ANR).

References

1. *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019 (Revision of IEEE 754-2008), (2019), pp. 1–84.
2. A. Adjé, D. B. Khalifa, and M. Martel, *Fast and Efficient Bit-Level Precision Tuning*, arXiv:2103.05241 [cs], (2021). arXiv: 2103.05241.
3. H. Benmaghnia, M. Martel, and Y. Seladji, *Fixed-Point Code Synthesis For Neural Networks*, Artificial Intelligence, Soft Computing and Applications, (2022), pp. 11–30. arXiv: 2202.02095.
4. Y.-C. Chang, N. Roohi, and S. Gao, *Neural Lyapunov Control*, 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), (2020). arXiv: 2005.00611.

5. S. Cherubin, D. Cattaneo, M. Chiari, A. D. Bello, and G. Agosta, *TAFFO: Tuning Assistant for Floating to Fixed Point Optimization*, IEEE Embedded Systems Letters, 12 (2020), pp. 5–8.
6. J.-M. Chesneaux, *L'arithmétique stochastique et le logiciel CADNA, Habilitation à diriger des recherches*, Université Pierre et Marie Curie, Paris, France, 1995.
7. J.-M. Chesneaux, S. Graillat, and F. Jezequel, *Numerical Validation and Assessment of Numerical Accuracy*, Oxford e-Research Center, (2009).
8. W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, *Rigorous Floating-Point Mixed-Precision Tuning*, in Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, New York, NY, USA, 2017, ACM, pp. 300–315.
9. T. Csendes, *Adversarial Example Free Zones for Specific Inputs and Neural Networks*, Proc. ICAI, (2020), pp. 76–84.
10. N. Damouche and M. Martel, *Mixed Precision Tuning with Salsa*, in Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, Porto, Portugal, 2018, SCITEPRESS - Science and Technology Publications, pp. 47–56.
11. E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, *Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper)*, in 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 10805 LNCS, Thessaloniki, Greece, 2018, pp. 270–287.
12. E. Darulova and V. Kuncak, *Towards a Compiler for Reals*, ACM Transactions on Programming Languages and Systems (TOPLAS), 39 (2017), pp. 8:1–8:28.
13. S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari, *Output Range Analysis for Deep Feedforward Neural Networks*, in NASA Formal Methods, A. Dutle, C. Muñoz, and A. Narkawicz, eds., vol. 10811, Springer International Publishing, Cham, 2018, pp. 121–138. Series Title: Lecture Notes in Computer Science.
14. M. Essam, T. B. Tang, E. T. W. Ho, and H. Chen, *Dynamic Point Stochastic Rounding Algorithm for Limited Precision Arithmetic in Deep Belief Network Training*, in 2017 8th International IEEE/EMBS Conference on Neural Engineering (NER), Shanghai, China, May 2017, IEEE, pp. 629–632.
15. S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, *Auto-Tuning for Floating-Point Precision with Discrete Stochastic Arithmetic*, Journal of Computational Science, 36 (2019), p. 101017.
16. H. Guo and C. Rubio-González, *Exploiting Community Structure for Floating-Point Precision Tuning*, in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam Netherlands, July 2018, ACM, pp. 333–343.
17. S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, *Deep Learning with Limited Numerical Precision*, Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, (2015). arXiv: 1502.02551.
18. N.-M. Ho, H. D. silva, and W.-F. Wong, *GRAM: A Framework for Dynamically Mixing Precisions in GPU Applications*, ACM Transactions on Architecture and Code Optimization, 18 (2021), pp. 1–24.
19. R. Hodován and Á. Kiss, *Practical Improvements to the Minimizing Delta Debugging Algorithm:*, in Proceedings of the 11th International Joint Conference on Software Technologies, Lisbon, Portugal, 2016, SCITEPRESS - Science and Technology Publications, pp. 241–248.

20. A. Ioualalen and M. Martel, *Neural Network Precision Tuning*, in Quantitative Evaluation of Systems, D. Parker and V. Wolf, eds., vol. 11785, Springer International Publishing, Cham, 2019, pp. 129–143. Series Title: Lecture Notes in Computer Science.
21. F. Jézéquel, S. s. Hoseininasab, and T. Hilaire, *Numerical Validation of Half Precision Simulations*, in Trends and Applications in Information Systems and Technologies, Á. Rocha, H. Adeli, G. Dzemyda, F. Moreira, and A. M. Ramalho Correia, eds., vol. 1368, Springer International Publishing, Cham, 2021, pp. 298–307. Series Title: Advances in Intelligent Systems and Computing.
22. P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, *AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications*, in Proceedings of the ACM International Conference on Supercomputing, Phoenix Arizona, June 2019, ACM, pp. 160–170.
23. I. Laguna, P. C. Wood, S. Ranvijay, and S. Bagchi, *GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications*, vol. 11501, Springer, 2019, pp. 227–246. M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan.
24. M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, *Automatically Adapting Programs for Mixed-Precision Floating-Point Computation*, in Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, New York, NY, USA, 2013, ACM, pp. 369–378.
25. M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, *Tool Integration for Source-Level Mixed Precision*, in 2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), 2019, pp. 27–35.
26. C. Lauter and A. Volkova, *A Framework for Semi-Automatic Precision and Accuracy Analysis for Fast and Rigorous Deep Learning*, arXiv:2002.03869 [cs], (2020). arXiv: 2002.03869.
27. W. Lin, Z. Yang, X. Chen, Q. Zhao, X. Li, Z. Liu, and J. He, *Robustness Verification of Classification Deep Neural Networks via Linear Programming*, Conference on Computer Vision and Pattern Recognition, (2019).
28. A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, *Towards Deep Learning Models Resistant to Adversarial Attacks*, 6th International Conference on Learning Representations, ICLR, (2019). arXiv: 1706.06083.
29. N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, *Mixed Precision Training with 8-bit Floating Point*, arXiv:1905.12334 [cs, stat], (2019). arXiv: 1905.12334.
30. H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, *ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning*, in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, Nov. 2018, IEEE, pp. 614–626.
31. T. Na, J. H. Ko, J. Kung, and S. Mukhopadhyay, *On-Chip Training of Recurrent Neural Networks with Limited Numerical Precision*, in 2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, USA, May 2017, IEEE, pp. 3716–3723.
32. A. S. Rakin, L. Yang, J. Li, F. Yao, C. Chakrabarti, Y. Cao, J.-s. Seo, and D. Fan, *RA-BNN: Constructing Robust & Accurate Binary Neural Network to Simultaneously Defend Adversarial Bit-Flip Attack and Improve Accuracy*, arXiv:2103.13813 [cs, eess], (2021). arXiv: 2103.13813.

33. C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, *Precimonious: Tuning Assistant for Floating-Point Precision*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'13, New York, NY, USA, 2013, ACM, pp. 27:1–27:12.
34. G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, *Fast and Effective Robustness Certification*, Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, NeurIPS, (2018), pp. 10825–10836.
35. C. Su, S. Zhou, L. Feng, and W. Zhang, *Towards High Performance Low Bitwidth Training for Deep Neural Networks*, Journal of Semiconductors, 41 (2020), p. 022404. https://iopscience.iop.org/article/10.1088/1674-4926/41/2/022404.
36. V. Tjeng, K. Xiao, and R. Tedrake, *Evaluating Robustness of Neural Networks with Mixed Integer Programming*, arXiv:1711.07356 [cs], (2019). arXiv: 1711.07356.
37. J. Vignes, *A Stochastic Arithmetic for Reliable Scientific Computation*, Mathematics and Computers in Simulation, 35 (1993), pp. 233–261.
38. ———, *Discrete Stochastic Arithmetic for Validating Results of Numerical Software*, Numerical Algorithms, 37 (2004), pp. 377–390.
39. N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, *Training Deep Neural Networks with 8-bit Floating Point Numbers*, Advances in Neural Information Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., Curran Associates, Inc., (2018), pp. 7686–7695. arXiv: 1812.08011, http://papers.nips.cc/ paper/7994-training-deep-neural-networks-with-8-bit-floating-point-numbers.pdf.
40. A. Zeller, *Why Programs Fail*, Morgan Kaufmann, Boston, second ed., 2009.
41. A. Zeller and R. Hildebrandt, *Simplifying and Isolating Failure-Inducing Input*, IEEE Trans. Softw. Eng., 28 (2002), pp. 183–200.
42. D. Zombori, *Verification of Artificial Neural Networks via MIPVerify and SCIP*, SCAN, (2020).
43. ———, *Fooling a Complete Neural Network Verifier*, The 9th International Conference on Learning Representations (ICLR), (2021).

Appendices

Appendix A Sine Neural Network

Appendix A.1 Type distribution for sine approximation with input value 2.37 considering one type per neuron

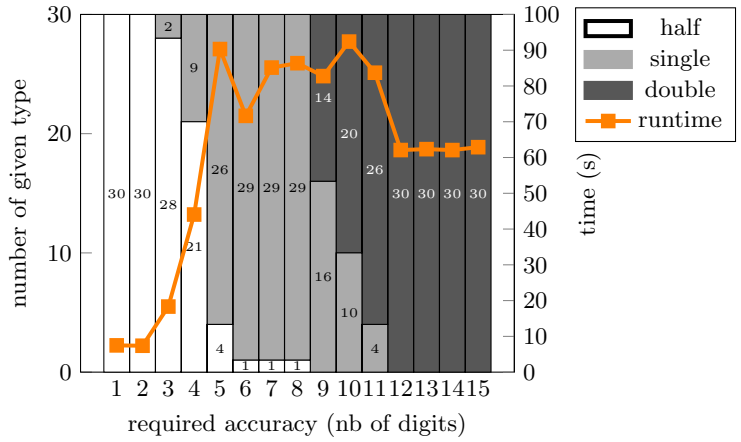

Fig. 10: Number of variables of each type and runtime for the sine neural network with input value 2.37

Appendix A.2 Type distribution for sine approximation with input value 2.37 considering one type per layer

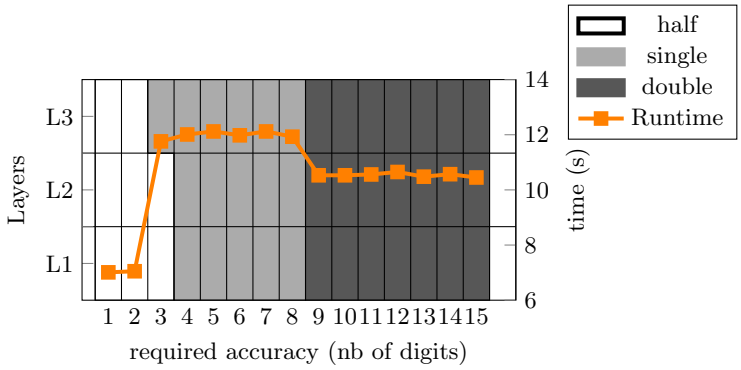

Fig. 11: Precision of each layer for the sine neural network with input value 2.37

Appendix B MNIST Neural Network

Appendix B.1 Type distribution for MNIST with test_data[61] input considering one type per layer

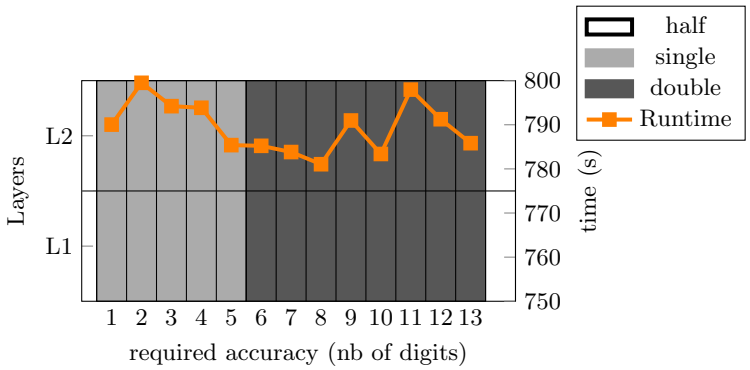

Fig. 12: Precision of each layer for MNIST neural network with test_data[61] input

Appendix B.2 Type distribution for MNIST with test_data[91] input considering one type per neuron

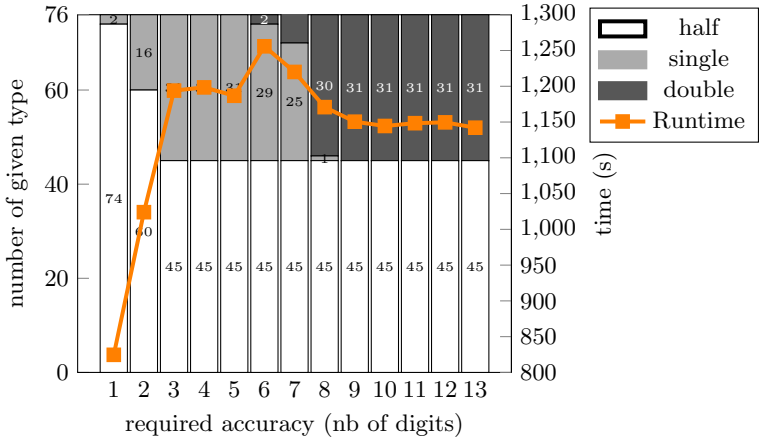

Fig. 13: Number of variables of each type and runtime for MNIST neural network with test_data[91] input

Appendix B.3 Type distribution for MNIST with test_data[91] input considering one type per layer

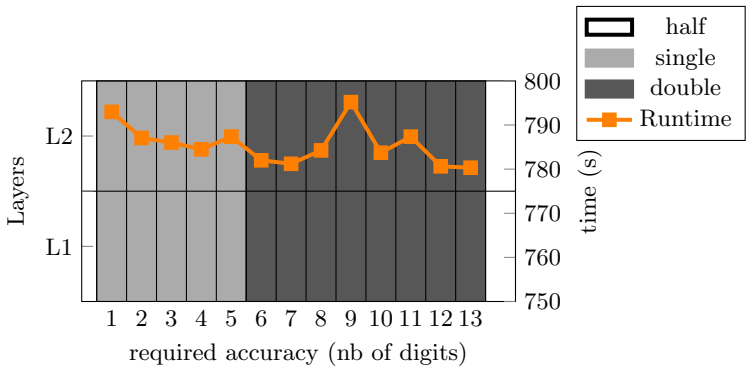

Fig. 14: Precision of each layer for MNIST neural network with test_data[91] input

Appendix C CIFAR Neural Network

Appendix C.1 Type distribution for CIFAR with test_data[386] input considering one type per layer

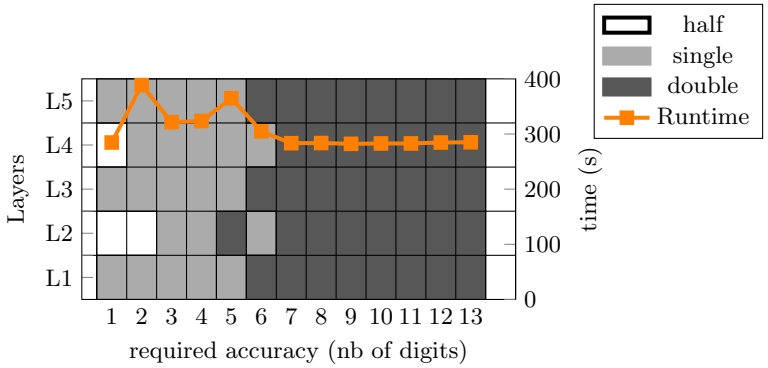

Fig. 15: Precision of each layer for CIFAR neural network with test_data[386] input

Appendix C.2 Type distribution for CIFAR with test_data[731] input considering one type per neuron

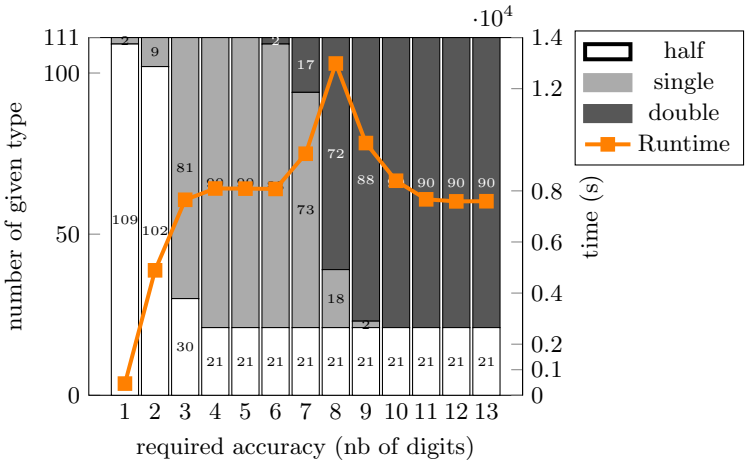

Fig. 16: Number of variables of each type and runtime for CIFAR neural network with test_data[731] input

Appendix C.3 Type distribution for CIFAR with test_data[731] input considering one type per layer

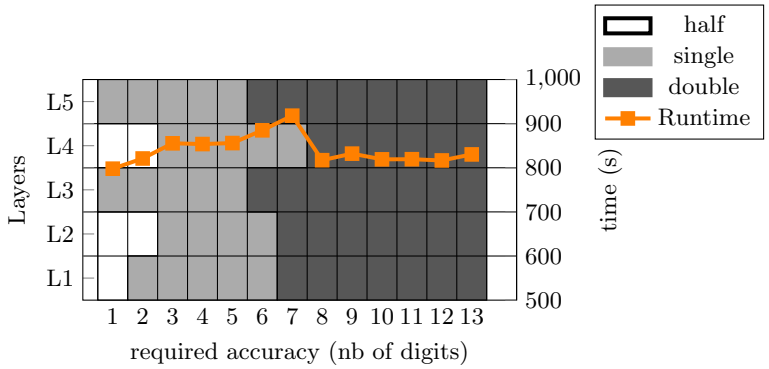

Fig. 17: Precision of each layer for CIFAR neural network with test_data[731] input

Appendix D Inverted pendulum

Appendix D.1 Type distribution for the inverted pendulum with input (0.5,0.5) considering one type per layer

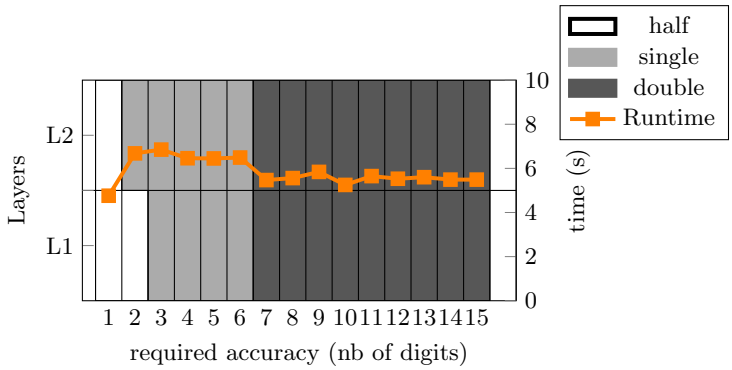

Fig. 18: Precision of each layer for the pendulum neural network with input (0.5,0.5)

Appendix D.2 Type distribution for the inverted pendulum with input (-3,-6) considering one type per neuron

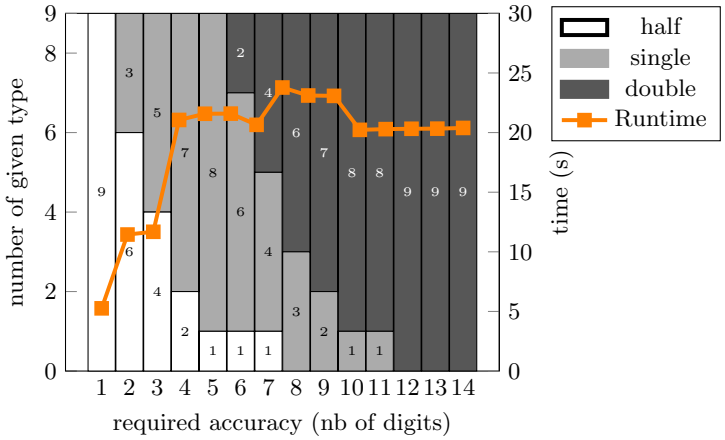

Fig. 19: Number of variables of each type and runtime for the pendulum neural network with input (-3,-6)

Appendix D.3 Type distribution for the inverted pendulum with input (-3,-6) considering one type per layer

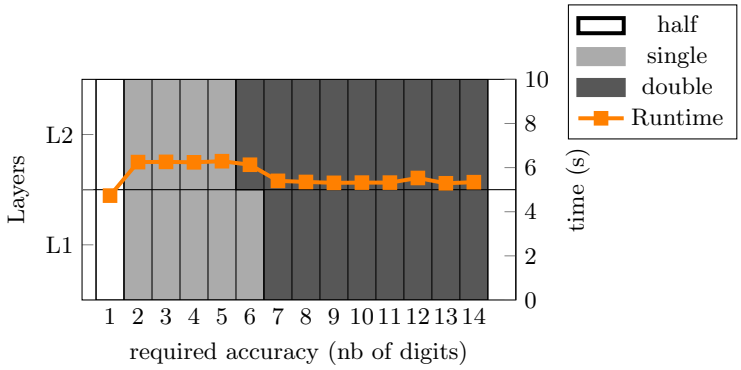

Fig. 20: Precision of each layer for the pendulum neural network with input (-3,-6)