# How Bounded Precision in Computing of Weights Affects the AI Computation Results, and What Is the Optimal Allocation of Weight Precisions

Christoph Q. Lauter[1][0000−0001−7335−8220], Martine Ceberio[1], Marcelo Frias[1], Esteban Rangel[2], Christopher J. Knight[2], Eric Petit[3], and Vladik Kreinovich[1][0000−0002−0244−1650]

[1] University of Texas at El Paso, El Paso TX 79968, USA
{cqlauter,mceberio,mfrias4,vladik}@utep.edu
[2] Argonne National Laboratory, Lemont, IL 60439, USA
{erangel,knightc}@anl.gov
[3] Intel Corporation, Portland, OR 97124, USA eric.petit@intel.com

**Abstract.** Many physical phenomena are described by complex systems of partial different equations. Even on high-performance parallel computers, numerical methods for solving these equations often require a large amount of computation time. Lately, AI techniques – namely, neural networks (NNs) – have been successfully used to speed up these computations: a NN is trained on several examples and, once trained, generates solutions for new initial and/or boundary conditions. These computations are faster, but still require a lot of computation resources: time, memory, and energy. It is known that in AI computations, we can often use fewer resources by using limited precision when computing and processing the weights of a NN. To utilize this idea, we need to be able to find out how bounded precision in computing weights affects the AI computation results, and – ideally – what is the *optimal* allocation of weight precisions. In this paper, we describe general algorithms for solving these two problems. As usual with algorithms, a lot of additional work is needed to make these algorithms practically efficient and easy to use – but what we show is that this is all doable.

**Keywords:** AI computations· neural networks for solving PDEs· precision of weight computations· optimal precision allocation.

## 1 Formulation of the Problem

### 1.1 Need to solve systems of partial differential equations

Real-world physical systems are usually described by systems of partial differential equations; see, e.g., [1, 4]. So, to analyze and to control such systems, we need to solve the corresponding systems of equations, i.e., to transform the appropriate input $x$ (e.g., the initial conditions and/or boundary conditions) into the desired result $y$ – e.g., the state of the system at some future moment of time. This is how we predict weather, this is how we predict how a building will react to an earthquake, etc.

### 1.2 Traditional way of solving such systems – and its limitation

There are many numerical techniques for solving these equations. These techniques are usually very time-consuming. As a result, even on modern high-performance computers, these computations may take hours or even days – and the resulting accuracy is often lower than desired.

### 1.3 How AI can help – and does help

In the last decades, very effective machine learning techniques have been developed, the most widely used is deep learning; see, e.g., [2]. In this technique, to compute the desired dependence $y = f(x)$ between the input $x$ and the output $y$, we:

- find many cases $(x^{(1)}, y^{(1)})$, ..., $(x^{(N)}, y^{(N)})$ in which we know both the input $x^{(k)}$ and the corresponding output $y^{(k)} = f\left(x^{(k)}\right)$, and
- train a neural network so that for the results $f_{NN}(x)$ generated by this network on input $x$ will fit the given data, i.e., that we should have $f_{NN}\left(x^{(k)}\right) \approx f\left(x^{(k)}\right)$ for all $k$.

Once the training is over and we get the weights $w_i$ for which the desired approximate equalities hold, we then "freeze" (= fix) these weights and use the resulting neural network to compute $y = f(x)$ based on $x$. In other words, for each input $x$, we return $f_{NN}(x)$ as an estimate for the desired value $f(x)$.

The main advantage of this technique is that while the training of a neural network may take a significant amount of time – e.g., the training for the Large Language Models like ChatGPT took years – once the network is trained, it produced its results very fast.

Machine learning has been successfully applied to many practical problems, in particular, to problems requiring solution of systems of partial differential equations. As a result, we have the following new method of solving the corresponding problems:

- first, we use traditional numerical methods to solve several cases of a given system, i.e., for find the outputs $y^{(1)}, \ldots, y^{(N)}$ corresponding to several inputs $x^{(1)}, \ldots, x^{(N)}$;
- then, we train a neural network on the resulting pairs

$$(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)});$$

- after that, given any input $x$, we apply the trained neural network to this input and return the result $f_{NN}(x)$ of applying this neural network as the solution $y = f(x)$ to the given system of equations.

### 1.4 Remaining challenge

Experience has shows that to get reliable results when solving systems of partial differential equations, it is often necessary to use double precision (or sometime

even higher precision) in all computations. Because of this, when the practitioners train the neural network to solve these equations, they also use double-precision to calculate all the weights. In many cases, the resulting network produces the desired result $y = f(x)$ and produces it fast.

However, there a remaining challenge: storing 64 or more bits of information for each weight means that overall, we we use a large portion of memory and use a lot of energy. A general experience of machine learning – experimentally confirmed on several applications of deep learning to solving systems of partial differential equations – has shown that we do not need such high precision for all the weights: some small weights can be just reduced to 0, and for some weights, we can use their 8-bit approximations and still get the desired accuracy of the final result $y$.

### 1.5 How can we predict the result of bounding weight precision? How can we determine the optimal precisions?

It is desirable to use lower bounded precision for computing weights – while still maintaining the desired accuracy of the final result. As of now, this is mostly done on a trial-and-error basis: we try some combinations of bounded precisions and see whether the accuracy remains good. This is a very time-consuming process, and the results are often far from impressive.

It is therefore desirable to be able:

- to predict how bounded precisions will affect the accuracy of the computation result without having to test the neural network every time, and
- to find the optimal precisions.

### 1.6 What we do in this paper

In the paper, we provide answers to both these questions:

- we provide an algorithm that, given proposed reduced precisions, estimates the effect of these reduced precisions on the accuracy of the computation result, and
- we provide an algorithm that, given the desired accuracy of the result, produces the optimal weight precisions – i.e., weight precisions that minimize the overall number of bits and thus, that minimize the use of resources such as memory and energy consumption.

*Comment.* Of course, what we will describe are *general* algorithms that provide the proof of the concept.

As usual with algorithms, a lot of additional work is needed to make these algorithms practically efficient and easy to use – but what we show is that this is all doable.

## 2 Analysis of the problem

### 2.1 How accuracy of $y$ depends on the weight's precision: general analysis

Each value $y$ in the numerical computation result depends on all the weights $w_1, \ldots, w_n$: $y = f(w_1, \ldots, w_n)$.

When we use $k_i$-bit precision to represent the $i$-th weight, we thus replace the original weight $w_i$ with a rounded value $\widetilde{w}_i = w_i + \Delta w_i$, where $\Delta w_i$ takes values from the interval $\left[-2^{-(k_i+1)}, 2^{-(k_i+1)}\right]$. Usually, the value $\Delta w_i$ is uniformly distributed on this interval, and the values $\Delta w_i$ corresponding to different $i$ are independent. In this case, for each $i$, the mean value of $\Delta w_i$ is 0, and the standard deviation is equal to

$$\sigma_i = \frac{1}{\sqrt{3}} \cdot 2^{-(k_i+1)}.$$

When we use the modified weights instead of the original weights, we get the value

$$\widetilde{y} = f\left(\widetilde{w}_1, \ldots, \widetilde{w}_n\right) = f(w_1 + \Delta w_1, \ldots, w_n + \Delta w_n)$$

that differs from the original value $y = f(w_1, \ldots, w_n)$ by the difference

$$\Delta y \stackrel{\text{def}}{=} \widetilde{y} - y = f(w_1 + \Delta w_1, \ldots, w_n + \Delta w_n) - f(w_1, \ldots, w_n). \tag{1}$$

Since the values $\Delta w_i$ are much smaller than the weights themselves, we can safely ignore terms that are quadratic or or higher order with respect to $\Delta w_i$. Thus, we can expand the expression (1) in Taylor series in terms of $\Delta w_i$ and ignore quadratic and higher order terms in this expansion. As a result, we get a linear expression

$$\Delta y = \frac{\partial f}{\partial w_1} \cdot \Delta w_1 + \ldots + \frac{\partial f}{\partial w_n} \cdot \Delta w_n. \tag{2}$$

Since all the values $\Delta w_i$ are independent and have mean 0, the mean value of this expression is 0, and the variance $V$ of $\Delta y$ is equal to:

$$V = E\left[(\Delta y)^2\right] = \left(\frac{\partial f}{\partial w_1}\right)^2 \cdot \sigma_1^2 + \ldots + \left(\frac{\partial f}{\partial w_n}\right)^2 \cdot \sigma_n^2. \tag{3}$$

The value $\Delta y$ is the sum of a large number of small independent random variables. So by the Central Limit Theorem (see, e.g., [3]), we can safely assume that the value $\Delta y$ is normally distributed.

Since the distribution is normal and its mean is 0, to get bounds on $\Delta y$ with certain confidence, we can have the bound $k_0 \cdot \sqrt{V}$, where $k_0$ depends on the desired confidence level [3]:

- for confidence level 95% we can take $k_0 = 2$;
- for confidence level 99.9%, we can take $k_0 = 3$;
- for confidence level $1 - 10^{-8}$, we can take $k_0 = 6$, etc.

The formula (3) describes the variance corresponding to a single input $x$. To get the variance $\sigma^2$ corresponding to all possible inputs, we should take the mean value $E_x[V]$ of the expression (3) over all the inputs $x$:

$$\sigma^2 = E_x[V] = E_x\left[\left(\frac{\partial f}{\partial w_1}\right)^2\right] \cdot \sigma_1^2 + \ldots + E_x\left[\left(\frac{\partial f}{\partial w_n}\right)^2\right] \cdot \sigma_n^2. \tag{4}$$

### 2.2 Where can we get the partial derivatives and their expected values?

To use the formulas (3) and (4) to predict the effect of limited weigh precision on the overall accuracy, we need to know the values of the corresponding partial derivatives with respect to $w_i$ – and the expected values of their squares. Where can we get this information?

To see where we can get these weights, let us recall that the weights in a neural network are determined by backpropagation, in which on each training step, each weight changes according to the following gradient descent formula:

$$w_i \to w_i - \alpha \cdot \frac{\partial J}{\partial w_i}, \tag{5}$$

where $J$ is the objective function, e.g.,

$$J = \sum_k \left(y_{NN}\left(x^{(k)}\right) - y^{(k)}\right)^2. \tag{6}$$

Thus, in the process of training, we already have the computed values of the partial derivative of $J$ with respect to $w_i$. The value $J$ depends on the result $y$ of computations in a known way. So, by the chain rule, we have

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial w_i}, \tag{7}$$

where the derivative of $J$ with respect to $y$ is easy to compute. For example, for the least squares objective function (6), we have

$$\frac{\partial J}{\partial y} = 2 \cdot \left(y - y^{(k)}\right). \tag{8}$$

Thus, based on the known derivatives of $J$, we can use the formula (7) to find the desired derivatives of $y$:

$$\frac{\partial y}{\partial w_i} = \left(\frac{\partial J}{\partial y}\right)^{-1} \cdot \frac{\partial J}{\partial w_i}. \tag{9}$$

Of course, the actual values of the derivatives depend on the values of the weights and thus, change from cycle to cycle. We need to only take into account the partial derivatives at the last cycle(s) of training, when the weights are close to the final ones.

### 2.3 How can we find the optimal weight precisions?

We want to minimize the overall number of bits. As we have mentioned, when we use $k_i$ bits, then

$$\sigma_i = \frac{1}{\sqrt{3}} \cdot 2^{-(k_i+1)}.$$

By taking binary (base-2) logarithm of both sides, we get

$$\log_2(\sigma_i) = -\frac{1}{2} \cdot \log_2(3) - (k_i + 1),$$

so

$$k_i = -\log_2(\sigma_i) - \frac{1}{2} \cdot \log_2(3) - 1, \tag{10}$$

i.e., $k_i = -\log_2(\sigma_i) + b$ for constant

$$b = - - \frac{1}{2} \cdot \log_2(3) - 1,$$

so

$$\sum_i k_i = -\sum_i \log_2(\sigma_i) + n \cdot b.$$

Adding a constant to a minimized function and multiplying the minimized function by a positive constant does not change where this function attains the minimum. Thus, minimizing the sum of the numbers of bits is equivalent to minimizing the sum of the $-\log_2(\sigma_i)$ terms.

Our constraint is that:

- either the average approximation error $\sigma^2$ should not exceed some given value $\sigma_0^2$,
- or the confident upper bound $k_0 \cdot \sigma$ should not exceed some given value $\Delta_0$ – which is equivalent to

$$\sigma^2 \le \sigma_0^2 \stackrel{\text{def}}{=} \frac{\Delta_0^2}{k_0^2}.$$

In both cases, the Lagrange multiplier approach to the corresponding constraint optimization problem reduces this problem to the problem of minimizing the expression

$$-\sum_i \log_2(\sigma_i) + \lambda \cdot \sum_i V_i \cdot \sigma_i^2, \text{ where } V_i \stackrel{\text{def}}{=} E_x\left[\left(\frac{\partial y}{\partial w_i}\right)^2\right]. \tag{11}$$

Here, $\lambda$ is the Lagrange multiplier that needs to be determined by the condition $\sigma^2 = \sigma_0^2$.

Differentiating the expression (11) with respect to an unknown $\sigma_i$, we conclude that

$$-\frac{1}{\ln(2) \cdot \sigma_i} + 2 \cdot \lambda \cdot V_i \cdot \sigma_i = 0,$$

i.e., that

$$\sigma_i^2 = \frac{c}{V_i}, \text{ where } c \stackrel{\text{def}}{=} \frac{1}{2 \cdot \ln(2) \cdot \lambda}.$$

Here, $V_i \cdot \sigma_i^2 = c$. Thus, the condition that $\sigma^2 = \sigma_0^2$ becomes

$$\sigma^2 = \sum_i V_i \cdot \sigma_i^2 = n \cdot c = \sigma_0^2,$$

so

$$c = \frac{\sigma_0^2}{n}.$$

Thus,

$$\sigma_i = \sqrt{\frac{c}{V_i}} = \frac{\sigma_0}{\sqrt{n \cdot V_i}}, \tag{12}$$

and

$$\log_2(\sigma_i) = \log_2(\sigma_0) - \frac{1}{2} \cdot \log_2(n) - \frac{1}{2} \cdot \log_2(V_i),$$

and, due to the formula (10), the optimal precision $k_i$ for each weight $w_i$ is

$$k_i = -\log_2(\sigma_0) + \frac{1}{2} \cdot \log_2(n) + \frac{1}{2} \cdot \log_2(V_i) - \frac{1}{2} \cdot \log_2(3) - 1. \tag{13}$$

Now, we are ready to describe the resulting algorithms.

## 3 Resulting algorithms

### 3.1 First algorithm: estimating the effect of different weight precision arrangements on the accuracy of the computation result

We have a trained neural network, and we have the precisions $k_i$ that we plan to assign to each neuron. The value $\widetilde{y}$ computed with thus bounded precisions will be somewhat different from the values $y$ returned by the neural networks in which each neuron has the original higb precision. We would like to estimate:

- either the mean squared value $\sigma^2$ of the inaccuracy $\Delta y \stackrel{\text{def}}{=} \widetilde{y} - y$ caused by these bounded precisions
- or the upper bound $\Delta$ on the resulting inaccuracy – that will be maintained with a given confidence level.

To perform these estimations:

- we need, during the last cycle(s) of the neural network training, to record the partial derivatives

  $$\frac{\partial J}{\partial w_i}$$

  of the objective function $J$ – the values that are computed and used during the backpropagation computations;

– then, we need to use these values – and the easy-to-compute values

$$\frac{\partial J}{\partial y}$$

– to compute the values

$$\frac{\partial y}{\partial w_i} = \left(\frac{\partial J}{\partial y}\right)^{-1} \cdot \frac{\partial J}{\partial w_i};$$

– finally, for each $i$, we need to take the arithmetic average $V_i$ of the squares of all the values of this derivative.

(If this was not done during the original training, we need to perform at least one cycle of additional training with the available pairs $\left(x^{(k)}, y^{(k)}\right)$ during which we collect the corresponding values of the partial derivatives.)

Then:

– the expected mean squared value $\sigma^2$ is equal to

$$\sigma^2 = \sum_i V_i \cdot \sigma_i^2, \text{ where } \sigma_i = \frac{1}{\sqrt{3}} \cdot 2^{-(k_i+1)};$$

– the upper bound $\Delta$ can be computed as $k_0 \cdot \sigma$, where the coefficient $k_0$ depends on the desired level of confidence: e.g., $k_0 = 2$ for confidence 0.85, $k_0 = 3$ for confidence 0.999, and $k_0 = 6$ for confidence $1 - 10^{-8}$.

### 3.2 Second algorithm: finding optimal weight precision arrangements for a given accuracy of the computation result

In this case, we are given:

– either the upper bound $\sigma_0^2$ on the mean squared value $\sigma^2$ of the inaccuracy $\Delta y \stackrel{\text{def}}{=} \widetilde{y} - y$ caused by these bounded precisions
– or the upper bound $\Delta_0$ on the resulting inaccuracy that will be achieved with a given degree of confidence.

Out of all the weight precisions $k_i$ that guarantee the desired accuracy, we need to find the values that minimize the overall number of bits (and thus, minimize the use of resources such as memory and energy).

In this case also, we first need to make sure that during the training, we compute the corresponding values $V_i$. If we are given the bound $\Delta_0$, then we need to compute the equivalent bound $\sigma_0 = \Delta_0/k_0$, where $k_0$ depends on the desired level of confidence. Then, we compute the optimal precisions $k_i$ by using the formula (13):

$$k_i = -\log_2(\sigma_0) + \frac{1}{2} \cdot \log_2(n) + \frac{1}{2} \cdot \log_2(V_i) - \frac{1}{2} \cdot \log_2(3) - 1. \qquad (13)$$

### 3.3 Discussion

**First comment** If for some small weights, their values with accuracy $k_i$ becomes 0, this means that the corresponding connection is not needed. If for some neuron, all the weights coming out of this neuron are small, this means that we do not need this neuron at all.

**Second comment: simplified version of the problem** The simplest version of the above problem is when we cannot change the precisions, but we can ignore some weights, i.e., we can make some weights equal to 0. In this setting, we can ask the two similar questions:

- How can we estimate the effect of this weight elimination on the result of the computations?
- For a given accuracy of the result, which weights should we eliminate to minimize the use of resources?

In this case, if denote by $D$, the set of indices of deletes weights, then the difference $\Delta y = \widetilde{y} - y$ takes the form

$$\Delta y = -\sum_{i \in D} \frac{\partial y}{\partial w_i} \cdot w_i. \tag{14}$$

It makes sense to assume that the derivatives are kind of randomly distributed, and that they are independent. In this case, the mean squared value $\sigma^2$ of $\Delta y$ is equal to

$$\sigma^2 = \sum_{i \in D} V_i \cdot w_i^2. \tag{15}$$

This formula provides the answer to the first question.

In the second question, the number of saved bits is proportional to the number of the deleted weights, i.e., to the number of weights in the set $D$. Thus, we need to select the largest possible set $D$ that still satisfies the desired equality $\sigma^2 \le \sigma_0^2$.

One can easily see that if in some arrangement, we keep a weight $i$ with a larger value of the product $V_i \cdot w_i^2$ and delete a weight $j$ with the smaller value $V_j \cdot w_j^2$, then by swapping these two weights, we can improve the situation.

Thus, to come up with the optimal arrangement, we need to sort all the weights in the increasing order of the product $V_i \cdot w_i^2$. Then, we delete the ones with the smallest product $V_i \cdot w_i^2$ and continue deleting the weights one by one until the sum of the corresponding small products reaches the required value $\sigma_0^2$.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Feynman, R., Leighton, R., Sands, M.: The Feynman Lectures on Physics. Addison Wesley, Boston, Massachusetts (2005)

2. Goodfellow, I., Y. Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge, Massachusetts (2016)
3. Sheskin, D. J.: Handbook of Parametric and Nonparametric Statistical Procedures. Chapman and Hall/CRC, Boca Raton, Florida (2011)
4. Thorne, K. S., Blandford, R. D.: Modern Classical Physics: Optics, Fluids, Plasmas, Elasticity, Relativity, and Statistical Physics. Princeton University Press, Princeton, New Jersey (2021)