Logarithmic Number System Is Optimal for AI Computations: Theoretical Explanation of Empirical Success

Olga Kosheleva[0000−0003−2587−4209], Vladik Kreinovich[0000−0002−1244−1650], Christoph Lauter[0000−0001−7335−8220], and Kristalys Ruiz-Rohena[0000−0002−7917−821X]

University of Texas at El Paso, El Paso TX 79968, USA
{olgak,vladik.cqlauter}@utep.edu, kruizrohena@miners.utep.edu

Abstract. Everyone knows the success story of machine-learning AI. However, the current AI tools are not perfect. We know how to make them better: every time we increase the amount of computations by the order of magnitude, we get a drastic improvement in the performance of the resulting machine learning tools. Training modern AI system requires a tremendous amount of computations – that already take a lot of time. So, to increase the number of computations, we need to make each computation step faster. One way to do that is to use low-precision arithmetic operations, e.g., with 1 byte per real number instead of the usual 8. It was shown that we can speed up computations even further if we apply an appropriate nonlinear transformation to all the values. Empirically, out of all transformations that were tried, logarithmic (log) transformation works the best. In this paper, we prove that under some reasonable condition, log transformation is indeed optimal. This way, not only we provide a theoretical explanation for the above empirical fact, but we are also proving that log transformation is better than all possible transformations, including the ones that have not been experimentally tried.

Keywords: Machine learning · AI · Nonlinear transformations · Logarithmic transformation · Optimal transformation.

1 Introduction

AI systems are very successful. Everyone is familiar with recent spectacular successes of deep-learning-based AI systems. Large Language Models (LLMs) – starting with ChatGPT – are so good that many of us have to struggle with LLM-produced solutions to homeworks that some students submit instead of their own work.

One of the main limitation to further successes is computation speed. Behind these successes is hard work of training. To achieve their successes, AI systems take thousands and millions of examples – and update billions of weights. Even

with state-of-the-art high-performance parallel computers, this training takes a lot of time – up to several years. And – at least so far – the more computational steps we perform, the better the results. Each order of magnitude speedup has led to new successes. So, to make further progress, we need to perform much more computations in the same period of time – i.e., we need to further speed up computations.

A natural way to speedup: low-precision computations. Computer engineers are constantly working on making electronic components as fast as possible. But for each state of these components, a natural way to further speed up AI-related computations is to use low-precision computations. Let us briefly explain why this is possible.

AI-related computations deal with real numbers – input data, output data, and the weights of the network. In a typical computer, each real number is represented by 8 bytes – 64 bits. This accuracy is important for many computations – e.g., for numerical solution of physics-motivated partial differential equations precision is critical, to the extend that some problems require double precision with 128 bits per number.

In contrast, even the best machine learning models do not need that accuracy. They process real data provided by measurements – and measurement accuracy is usually around a few percents, and they try to predict the outputs that also come from measurements. $1\% = 1/100$ accuracy corresponds to 7 bits, so it make sense to limit computations to 8-bit numbers. The fewer bits, the faster computations – and reducing the word length from 8 bytes to 1 byte provides an almost order-of-magnitude speedup. Experiments show that such a reduction indeed speeds up training without degrading the training quality – and there is a theoretical explanation for this success [3].

Need for a further speedup. The larger the range of possible values, the more bits we need to describe the exponent. A natural way to decrease this range is to take into account that many physical quantities have several different scales. For example, we can measure the intensity of an acoustic signal by its amplitude or by its power – which is the square of the amplitude. We can also measure the intensity by decibels – which is roughly equivalent to the number of levels up to this one that a human ear can distinguish. Decibels are proportional to the logarithm of intensity.

From the physical viewpoint, different scales represent the exact same physical quantity. However, from the computational viewpoint, the use of such nonlinear re-scaling may help. For example, if we go from power to amplitude, i.e., take a square root, then the range from 1 to 100 reduces to 1 to 10. If we use logarithms, the range decreases even more: to 0-to-2. Empirical data shows that such nonlinear re-scaling does speed up training. Among re-scalings that were tried, logarithmic re-scaling works the best; see, e.g., [1, 5, 7].

Natural question – and what we do in this paper. Log transformation is the best of all that were tried, but a natural question is: maybe some transformation

that was not tried will be even better? To answer this question, we formulate the question of selecting the best transformation in precise terms, and we prove that indeed, under some reasonable assumptions, every optimal transformation should be either $x \mapsto \log(x)$, or $x \mapsto \exp(x)$, or $x \mapsto x^\alpha$. Since empirically, log transformation is the best of these three, it is therefore optimal.

Thus, not only we provide a theoretical explanation for the empirical fact – that log was better than all transformations that were tried – but we also provide an assurance that log is better than all others transformations, including those that no one tried.

A comment on how we do it. The technique that we use in this paper is the technique of invariances. This is one the main techniques of modern physics; see, e.g., [2, 6], where invariances are known as *symmetries*. This technique has been also successfully used in AI to explain empirically successful selection of activation functions in neural networks and of "and"- and "or"-operations in fuzzy logic; see, e.g., [4].

2 Let Us Formulate This Problem in Precise Terms

What we are looking for. We want to find the optimal nonlinear transformation $x \mapsto y = f(x)$ between two scales for describing the same physical quantity – e.g., between amplitude and power. Since all values that we deal with are approximate, it makes sense to require that small changes in x should lead to equivalently small changes in y, i.e., in precise terms, that the function $f(x)$ be smooth (differentiable).

In principle, there may be several equally good transformations, maybe the whole family of them. So, to make the formation as general as possible, we need to consider looking for a *family* of transformations $x \mapsto F(x, c_1, \ldots, c_k)$ characterized by some parameters $c_1, \ldots, c_k$.

Definition 1. *Let k be fixed. By a* k-family*, we mean a smooth function $F(x, c_1, \ldots, c_k)$ whose dependence on x is nonlinear. We say that a function $f(x)$* belongs *to the family if for some c_i, we have $f(x) = F(x, c_1, \ldots, c_k)$ for all x.*

In general, the simpler the expression, the fewer parameters it has, the easier – and thus, the faster – it is to compute this expression. Since our goal is to speed up computations, we should be looking for families with the smallest number of parameters.

For each k, out of all possible families, we want to find the family that is, in some sense, optimal. How can we define "optimal"?

What we mean by optimal. In many applications, we have a well-defined objective function $J(a)$, and selecting an optimal alternative a simply means maximizing (or minimizing) this objective function. However, this formulation does not cover all possible optimization settings.

For example, when we are looking for the best sorting algorithm, a natural idea is to select the algorithm with the best average computations time. However, there are several such algorithms. We can use this non-uniqueness to optimize something else: e.g., we may want to select, among all the algorithms with the same average computation time, the one withe the best worst-case computation time. If we still have several equally good algorithm, this means that our selection method is not final: we can use the remaining non-uniqueness to optimize something else, etc. When we reach the final optimality criterion, there will be exactly one optimal alternative.

The more conditions we pile on, the more complex will the optimality criterion be. But for all of them, the most important feature is that each criterion allows us to compare pairs of alternatives (a, b) – at least some such pairs – and decide whether a is better than b (we will denote it by $a \succ b$) or b better than a ($b \succ a$), or that they are equivalent with respect to this criterion (we will denote it $a \sim b$). Of course, if a is better than b and b is better than c, then a should be better than c. Thus, we arrive at the following definition.

Definition 2. *Let $\mathcal{A}$ be a set. Its elements will be called* alternatives. *By an* optimality criterion *on $\mathcal{A}$, we mean a pair $(\succ, \sim)$ of binary relations for which, for all a, b and c, the following conditions are satisfied:*

- *if $a \succ b$ and $b \succ c$, then $a \succ c$;*
- *if $a \succ b$ and $b \sim c$, then $a \succ c$;*
- *if $a \sim b$ and $b \succ c$, then $a \succ c$;*
- *if $a \sim b$ and $b \sim c$, then $a \sim c$;*
- *if $a \sim b$ then $b \sim a$;*
- *if $a \succ b$, then $a \not\sim b$.*

We say that an optimality criterion is final *if there is exactly one* optimal *alternative a_{opt}, i.e., alternative for which for each a, we have $a_{\text{opt}} \succ a$ or $a_{\text{opt}} \sim a$.*

Need to take into account linear re-scalings. We want to find the optimal family of transformations from scale x to scale y. It is important to take into account that in each of these scales, the numerical value is not absolute: if we change a measuring unit to a one that is a times smaller, all numerical values are multiplied by a, so that $x \mapsto a \cdot x$. For example, 2 meters become $100 \cdot 2 = 200$ cm. Similarly, if we change the starting point to the one which is b units smaller, b is added to all numerical values: e.g., 20° C becomes $20 + 273.16 = 293.16^\circ$ K. In general, we can have linear transformations $x \mapsto a \cdot x + b$. Similarly, we can have a linear transformation $y \mapsto A \cdot y + B$.

In neural training, we can use data corresponding to different measuring units. The relative quality of a machine learning algorithm should not change if we simply use centimeters instead of meters. Use of different units means that instead of the original function $f(x)$, we consider a new function

$$T_{a,b,A,B}(f) \stackrel{\text{def}}{=} A \cdot f(a \cdot x + b). \tag{1}$$

Definition 3. *We say that functions $f(x)$ and $g(x)$ are* linearly equivalent *if there exist $a > 0$, b, $A < 0$, and B for which, for all x, we have $g(x) = A \cdot f(a \cdot x + b)$.*

If we apply the transformation $T_{a,b,A,B}$ to all functions from the original family $\mathcal{F}$, we thus get a new family that we will denote $T_{a,b,A,B}(\mathcal{F})$.

Definition 4. *We say that an optimality criterion is* linearly invariant *if for every two families $\mathcal{F}$ and $\mathcal{G}$ and for all $a > 0$, b, $A > 0$, and B, $\mathcal{F} \succ \mathcal{G}$ implies $T_{a,b,A,B}(\mathcal{F}) \succ T_{a,b,A,B}(\mathcal{G})$, and $\mathcal{F} \sim \mathcal{G}$ implies $T_{a,b,A,B}(\mathcal{F}) \sim T_{a,b,A,B}(\mathcal{G})$.*

Now, we are ready to formulate and prove our main result.

3 Main Result

Proposition 1. *The smallest k for which there exists a final linearly invariant optimality criterion on the set of all k-families is $k = 3$. For this k, every function from the optimal family is linearly equivalent either to $\log(x)$, or to $\exp(x)$, or to x^α for some α.*

Proof. Since the criterion is final, there exists the unique optimal family $\mathcal{F}_{\rm opt}$. Let us first prove that this optimal family is itself linearly invariant, i.e., that $T_{a,b,A,B}(\mathcal{F}_{\rm opt}) = \mathcal{F}_{\rm opt}$ for all a, b, A, and B. Indeed, by definition of the optimal family, for each other family $\mathcal{F}$, we have either $\mathcal{F}_{\rm opt} \succ \mathcal{F}$ or $\mathcal{F}_{\rm opt} \sim \mathcal{F}$. In particular, for each family $\mathcal{F}$, we have either $\mathcal{F}_{\rm opt} \succ T^{-1}_{a,b,A,B}(\mathcal{F})$ or $\mathcal{F}_{\rm opt} \sim T^{-1}_{a,b,A,B}(\mathcal{F})$, where $T^{-1}_{a,b,A,B}$ denotes the inverse linear transformation. Since the optimality criterion is linearly-invariant, we can apply the transformation $T_{a,b,A,B}$ to both sides of the corresponding relation, we can conclude that either $T_{a,b,A,B}(\mathcal{F}_{\rm opt}) \succ \mathcal{F}$ or $T_{a,b,A,B}(\mathcal{F}_{\rm opt}) \sim \mathcal{F}$. By definition of the optimal alternative, this means that the family $T_{a,b,A,B}(\mathcal{F}_{\rm opt})$ is also optimal. But since the optimality criterion is final, there can be only one optimal family. Thus, we indeed have $T_{a,b,A,B}(\mathcal{F}_{\rm opt}) = \mathcal{F}_{\rm opt}$.

This equality means, in particular, that for every function $f(x)$ from the optimal family $\mathcal{F}_{\rm opt}$ and for all possible values $a > 0$, b, $A > 0$, and B, the function $A \cdot f(a \cdot x + b) + B$ also belongs to this family. This expression has 4 parameters a, b, A, and B. For $k \le 3$, we cannot have all these functions to be different – that would lead to a 4-parametric family. Thus, at least some 1-parametric family of transformations – that starts with the identity for $t = 0$ – should keep the function un-changed, i.e., we should have

$$A(t) \cdot f(a(t) \cdot x + b(t)) + B(t) = f(x) \tag{2}$$

for all t. Differentiating both sides of the formula (2) with respect to t, we get

$$A'(t) \cdot f(a(t) \cdot x + b(t)) + A(t) \cdot f'(a(t) \cdot x + b(t)) \cdot (a'(t) \cdot x + b'(t)) = 0, \tag{3}$$

where, as usual, the dash – such as A' – means the derivative. In particular, for $t = 0$, taking into account that for $t = 0$ we have the identity transformation for

which $a(0) = A(0) = 1$ and $b(0) = B(0) + 1$, the formula (3) takes the form of the following (implicit) differential equation:

$$A_0 \cdot f(x) + f'(x) \cdot (a_0 \cdot x + b_0) + B_0 = 0, \tag{4}$$

where we denoted $A_0 \stackrel{\text{def}}{=} A'(0)$, $a_0 \stackrel{\text{def}}{=} a'(0)$, $b_0 \stackrel{\text{def}}{=} b'(0)$, and $B_0 \stackrel{\text{def}}{=} B'(0)$.

Let us move the terms proportional to $f'(x)$ to the right-hand side and take into account that $f(x) = y$. Then, the formula (7) takes the form

$$A_0 \cdot y + B_0 = -\frac{dy}{dx} \cdot (a_0 \cdot x + b_0). \tag{5}$$

We can separate the variables if we divide both sides of this equality by $A_0 \cdot y + B_0$ and by $a_0 \cdot x + b_0$ and multiply both sides by $-dx$

$$-\frac{dx}{a_0 \cdot x + b_0} = \frac{dy}{A_0 \cdot y + B_0}. \tag{6}$$

To solve this differential equation, let us consider four possible cases, depending on whether a_0 is equal to 0 and whether A_0 is equal to 0.

If $a_0 = 0$ and $A_0 = 0$, then integrating both sides of (6), we simply get $\text{const} \cdot x = \text{const} \cdot y + C$, where C is the integration constant. In this case, y is a linear function of x, which contradicts to our definition of a family as a class of nonlinear functions. Thus, the case $a_0 = A_0 = 0$ is not possible.

If $a_0 = 0$ and $A_0 \neq 0$, then (6) takes the form

$$-\frac{dx}{b_0} = \frac{dy}{A_0 \cdot y + B_0}. \tag{7}$$

If we introduce a new variable $Y \stackrel{\text{def}}{=} A_0 \cdot y + B_0$, then we have $dY = A_0 \cdot dy$. So, if we multiply both sides of (7) by A_0, we get

$$-\frac{A_0}{b_0} \cdot dx = \frac{A_0 \cdot dy}{A_0 \cdot y + B_0} = \frac{dY}{Y}. \tag{8}$$

If we now introduce a new variable $X \stackrel{\text{def}}{=} (-A_0/b_0) \cdot x$, then the equation (8) takes the form

$$dX = \frac{dY}{Y}. \tag{9}$$

Integrating both sides of the equation (9), we get $X + C = \ln(Y)$, hence $Y = \exp(X + C) = \text{const} \cdot \exp(X)$. Since Y is a linear function of y and C is a linear function of x, it follows that the function $y(x)$ is linearly equivalent to $\exp(x)$.

If $a_0 \neq 0$ and $A_0 = 0$, then (6) takes the form

$$-\frac{dx}{a_0 \cdot x + b_0} = \frac{dy}{B_0}. \tag{11}$$

If we introduce a new variable $X \stackrel{\text{def}}{=} a_0 \cdot x + b_0$, then we have $dX = a_0 \cdot dx$. So, if we multiply both sides of (11) by $-a_0$, we get

$$\frac{a_0 \cdot dx}{a_0 \cdot x + b_0} = \frac{dX}{X} = -\frac{a_0}{B_0} \cdot dy. \tag{12}$$

If we introduce a new variable $Y \stackrel{\text{def}}{=} -(a_0/B_0) \cdot y$, then (12) takes the form

$$dY = \frac{dX}{X}. \tag{13}$$

Integrating both sides of the equation (13), we get $Y = \ln(X) + C$. Since Y is a linear function of y and C is a linear function of x, it follows that the function $y(x)$ is linearly equivalent to $\ln(x)$.

If $a_0 \neq 0$ and $A_0 \neq 0$, then, if we multiply both sides of (6) by A_0, we get

$$-\frac{A_0}{a_0} \cdot \frac{a_0 \cdot dx}{a_0 \cdot x + b_0} = \frac{A_0 \cdot dy}{A_0 \cdot y + b_0}. \tag{14}$$

If we introduce new variables $X \stackrel{\text{def}}{=} a_0 \cdot x + b_0$ and $Y \stackrel{\text{def}}{=} A_0 \cdot y + B_0$, then (14) takes the form

$$\alpha \cdot \frac{dX}{X} = \frac{dY}{Y}, \tag{15}$$

where $\alpha \stackrel{\text{def}}{=} -A_0/a_0$. Integrating both sides of the equation (15), we get $\ln(Y) = \alpha \cdot \ln(X) + C$, hence

$$Y = \exp(\alpha \cdot \ln(X) + C) = \exp(\alpha \cdot \ln(X)) \cdot \exp(C) = \exp(C) \cdot (\exp(\ln(X))^\alpha =$$

$$\text{const} \cdot X^\alpha. \tag{16}$$

Since Y is a linear function of y and C is a linear function of x, it follows that the function $y(x)$ is linearly equivalent to x^α.

4 Conclusions and Future Plans

Conclusions. While modern machine-learning based AI techniques has led to spectacular successes, their results are often not perfect. To a large extend, the very success of these techniques is caused by the possibility to perform trillions of computational steps in reasonable time: so far, every order-of-magnitude increase in number of computations has led to a significant improvement of the machine learning results. So, a natural way to improve the quality of AI systems is to perform even more computations – and to be able to do it in reasonable time, we need to speed up each computational step. A significant speed up has been achieved by taking into account that with relatively low accuracy of most data and of most AI results, it is sufficient to use low-precision storage and processing of the real numbers: 8 bits are usually sufficient. We can speed up computations even more if we decrease the range of the values by applying an appropriate nonlinear transformation to all the values. Experiments showed that out of all tested transformation, logarithmic transformation leads to the best speedup.

In this paper, we provide a theoretical explanation for this result: namely, we prove that under reasonable conditions, log transformation is indeed optimal. This not only explains the empirical result, this also shows that log transformation is better than other possible transformation that no one tried.

Future work. In our study, we considered the families of transformations of the smallest possible dimension – i.e., with the smallest possible number of parameters. It is desirable to analyze what happens when we consider families of higher dimension: maybe, for an appropriate selection of parameters, we will be able to achieve even better speedup.

Acknowledgments. This work was supported in part by the National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science), HRD-1834620 and HRD-2034030 (CAHSI Includes), EAR-2225395 (Center for Collective Impact in Earthquake Science C-CIES), and by the AT&T Fellowship in Information Technology.

It was also supported by a grant from the Hungarian National Research, Development and Innovation Office (NRDI), and by the Institute for Risk and Reliability, Leibniz Universitaet Hannover, Germany.

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

References

1. Christ, M., de Dinechin, F., Pétrot, F.: Low-precision logarithmic arithmetic for neural network accelerators. In: Pericas, M., Pnevmatikatos, D., Sourdis, I. (eds.) Proceedings of the 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures, and Processes ASAP 2022, Gothenburg, Sweden, July 12–14, 2022, pp. 72–78.
2. Feynman, R., Leighton, R., Sands, M.: The Feynman lectures on physics. Addison Wesley, Boston, Massachusetts (2005)
3. Lauter, C. Q., Volkova, A.: A framework for semi-automatic precision and accuracy analysis for fast and rigorous deep learning. In: Cornea, M., Liu, W., Tisserand, A. (eds.), Proceedings of the 27th IEEE Symposium on Computer Arithmetic ARITH 2020, Portland, Oregon, June 7–10, 2020, pp. 103–110. IEEE Press (2020)
4. Nguyen, H. T., Kreinovich, V.: Applications of continuous mathematics to computer science. Kluwer, Dordrecht (1997)
5. Niu, Z. et al.: Hardware-efficient logarithmic floating-point multipliers for error-tolerant applications. IEEE Transactions on Circuits and Systems–I **71**(1), 209–222 (2024)
6. Thorne, K. S., Blandford, R. D.: Modern classical physics: optics, fluids, plasmas, elasticity, relativity, and statistical physics. Princeton University Press, Princeton, New Jersey (2021)
7. Zhao, J., et al.: LNS-Madam: low-precision training in logarithmic numer system using multiplicative weight upodate. IEEE Transactions on Computers **72**(12), 3179–3190 (2022)