Indian Journal of Artificial Intelligence and Neural Networking (IJAINN)
ISSN: 2582-7626 (Online), Volume-4 Issue-6, October 2024

Algorithms and Data Structures for Numerical Computations with Automatic Precision Estimation

Igor V. Netay

***Abstract*: We introduce data structures and algorithms to count numerical inaccuracies arising from usage of floating numbers described in IEEE 754. Here we describe how to estimate precision for some collection of functions most commonly used for array manipulations and training of neural networks. For highly optimized functions like matrix multiplication, we provide a fast estimation of precision and some hint how the estimation can be strengthened.**

***Keywords*: Digital Noise; Floating Point Numbers, Neural Networks, Numerical Analysis.**

I. INTRODUCTION

Numerical calculations with floating point numbers almost always are inexact. The simplest known example is

$$0.1 + 0.2 \ \ != 0.3,$$

which holds for almost all programming languages. This example illustrates the consequences of non-exact conversion from decimal number literals to binary float number representation. At the same time, computations with floating point binary numbers are themselves inexact.

Another well-known example shows that the values of roots of quadratic equation depend on way the solution was computed. Consider equation:

$$x^2 + 1000x - 2 \cdot 10^{-11} = 0. (1)$$

If we find the roots as $\frac{-b \pm \sqrt{D}}{2a}$ for $D = b^2 - 4ac$ for $ax^2 + bx + c = 0$, then for Eq.(1) we get one root $x = -1000$ with 53 exact bits (maximal number for 64-bit float) and other root $\approx 5.7 \cdot 10^{-14}$ without any exact bits. If we will compute the roots in other way:

$$x_1 = \frac{-b - sgn(b)\sqrt{D}}{2a}. \quad (2)$$

$$x_2 = \frac{c}{a \cdot x_1}, \ (3)$$

Then we get $x_1 = -1000$ and $x_2 = 2 \cdot 10^{-14}$ with all 53 bits exact for both roots.

The example above shows that accuracy of the result depends on way of computation frequently. Actually, usually one cannot easily guess a proper way to get maximally exact results. Moreover, one cannot get exact results for any arithmetic operation. Subtraction of numbers that are close to each other is the worst possible case.

Manuscript received on 07 October 2024 | Revised Manuscript received on 11 October 2024 | Manuscript Accepted on 15 October 2024 | Manuscript published on 30 October 2024.
*Correspondence Author(s)
Igor V. Netay*, JSRPC Kryptonite and Institute for Information Transmission Problems of Russian Academy of Sciences, Moscow, Russia. Email: i.netay@kryptonite.ru, ORCID ID: 0000-0002-0644-6148.

During this operation accuracy of the result drops drastically and there is no way to avoid it. Two mathematically equivalent ways to compute the same expression can lead to different numerical results, and it may be unclear which one is closer to correct mathematical value. This fact was the motivation for us to implement means to estimate loss of precision during computations and to track exact mantissa bits in the results. Also, this helps one to avoid wrong conclusions based upon possible interpretations of numerical noise as meaningful data.

Calculations with huge arrays of numbers and arithmetic operations involved like training of neural networks usually lead to high accumulated digital noise in the result (see [5][8][9][10]). Actually, even simpler numerical computations can lead to inexact (and unreliable) results frequently. For instance, among such computations are inversion of ill-conditioned matrices and solutions of ill-conditioned systems of linear equations.

We do not provide new ways to invert matrix or to solve linear systems or to perform any other computations. Instead, we enhanced standard library algorithms with additional digital error computation and its tracking that will automagically indicate whether the result is reliable or it should be discarded and more precise algorithms (or different computational approaches) are needed.

Usual goal of numerical analysis problem is to design algorithms to construct maximally or enough precise approximations of mathematical objects. Now for many applied problems numerical precision is ignored, and computations are constructed from some standard functions provided by well-known libraries. Here we propose the answer for the problem of inaccuracy estimation for a given model and algorithm. For practical potentially unreliable algorithm calculation of precision should be performed and may lead to searching for some more reliable algorithms.

II. NUMERICAL PRELIMINARIES

A. Errors

Suppose we calculate an approximation $\hat{x}$ of real value x. *Absolute error* is defined as $|\Delta x| = |\hat{x} - x|$.

Relative error is defined as δ such that $\frac{\hat{x}}{x} = 1 + \delta$.

We will denote by float numeric type the standard 64-bit floating point numbers (like it is called in Python language and not like C where it denotes 32-bit float). We implement our library in Rust and therefore use its notation where floating types are called f32 and f64. We extend standard float (equal to Rust f64) type and call it xf64 (extended float with 64 bits).

Example 1.3. Consider equation Eq. (1). One can see in Table 1 and Table 2, that computation with xf64 maintains the same number of mantissa bits (53) and additionally shows precision of computation. Method __str__ for xf64 was modified to substitute unreliable digits of the number with question marks.

Table 1. Solution in the First Way

Type	x_1, 15 Digits	x_1, Exp. Form
float (64-bit)	-1000.000000000000	-1.000000000000000e+03
xf64	-1000.00000000000?	-1.00000000000000?e+03
type	**x_2, 15 digits**	**x_2, exp. form**
float (64-bit)	0.000000000000057	5.684341886080801e-14
xf64	0.0000000000000??	?.???????????????e-14

Table 2. Solution in the Second Way (Same for x_1)

Type	x_2, 15 Digits	x_2, Exp. Form
float (64-bit)	0.000000000000020	2.000000000000000e-14
xf64	0.000000000000020	2.00000000000000?e-14

B. Error Propagation in Function Computation

Given an approximation $\hat{x}$ of real value x, we approximate $y = f(x)$ with some $\hat{y}$. For estimation of the absolute error in $\hat{y}$, linear error estimation is usually applied:

$$|\Delta y| = |\hat{y} - y| = |f(x + \Delta x) - f(x)| \approx \left|\frac{df(x)}{dx}\right|_{x=\hat{x}} \cdot |\Delta x|.$$

For the relative error we obtain

$$\left|\frac{\hat{y} - y}{\hat{y}}\right| \approx \left|\frac{\hat{x}}{f(\hat{x})} f'(\hat{x})\right|$$

The number $\left|\frac{\hat{x}}{f(\hat{x})} f'(\hat{x})\right|$ is called the *condition number* $C_f(\hat{x})$ of function $f(x)$ at $\hat{x}$. It corresponds to the error propagation coefficient of function $f(x)$ at $\hat{x}$. This means that relative error δ in $\hat{x}$ leads to error of magnitude $C_f(\hat{x}) \cdot \delta$ in $f(\hat{x})$. It is easy to check that if $\hat{x}$ has s significant bits, then $f(\hat{x})$ has $s - log_2 C_f(\hat{x})$ exact bits. Examples of some condition numbers for some standard floating point number operations are listed in Table 3.

Table 3. Condition Numbers for Some Standard Operations

$f(x)$	$C_f(x)$
$x + a$	$\frac{x}{x+a}$
ax	1
$1/x$	1
x^n	$\|n\|$
$ln(x)$	$\left\|\frac{1}{lnx}\right\|$
$sinx$	$\|x \cdot cotx\|$
$cosx$	$\|x \cdot tanx\|$

Function arcsin has poles at ± 1. So, near to poles we can see high precision loss:

arcsin(0.999999) = 1.56?

arcsin(0.999???) = 1.???

Here we calculate the function at 0.999999 with 6 and 3 exact digits. High condition number leads to high precision loss.

C. Other Reasons of Results Divergence

Another important reason of variations in results of numerical computations is lack of associativity for basic operations like addition and multiplication for floating point numbers used in programmatic computations (unlike computations with real numbers ℝ).

Therefore, any action influencing the order of arithmetic operations can change the precision of the result of the entire computations. For instance,

- vectorization instructions in CPU,
- flags used during compilations of imported libraries (e. g. -O2,-O3),
- any race condition of any form:
  - any use of async/await,
  - multithreading implementation details in your operation system,
  - Networking data transfer implementation details in you cluster.

Also, different versions of core systems libraries like **glibc** can cause some small differences and finally lead to some more visible difference.

III. RELATED WORK

Mainly, other ways to check precision are the following:

1. try to redo computation with more bits in floats and find the common part of the result;
2. start with some big enough number of digits/bits and see the loss of significance in systems like pari/gp or other libraries using gmp,
3. use interval arithmetics (like C++ boost/intervals).

Unfortunately, these ways have the following major issues:

- There is no any actual precision guarantee (1),
- Very low (2) or low (1, 3) performance,
- Very high (2) or high (1, 3) memory overhead,
- The computation should be redone from the beginning and lead to some new result of some number data type. So, there is no way to check precision of an existing computation.
- Need to reproduce computation in other programming languages (3 for most of cases).

We prefer to estimate precision of some existing computation algorithm and track the estimation of its errors alongside the main dataflow. Estimated inaccuracies are stored in memory as numbers of exact mantissa bits.

Given initial data with some values of inaccuracies, we cannot compute all the results and their inaccuracies precisely, so we should choose between:

Retrieval Number: 100.1/ijainn.F109204061024
DOI: 10.54105/ijainn.F1092.04061024
Journal Website: www.ijainn.latticescipub.com

Published By:
Lattice Science Publication (LSP)

1. estimate inaccuracies from below (so parts of results marked as inexact have no sense and should never be interpreted),
2. estimate inaccuracies from above (so data marked as inexact may have some computational errors, while all the other data is exact),
3. something in the middle without any guaranties.

The most reliable and expensive way is to combine (1) and (2). In this way mantissa bits are divided into three groups:

- "black" bits which are noisy and meaningless,
- "white" bits which are exact and reliable,
- "gray" bits in the middle part may depend on particular software and hardware details, computational process environment and other factors.

"Black" bits arise mainly from data inexactness and mathematical error propagation. At the same time, "gray" bits arise from numerically unstable algorithms. Actually, some computations evaluate results depending on initial data discontinuously and cannot be implemented with some numerically stable algorithm.

For instance, computation of eigenvectors for ill conditioned matrix would have big "gray" mantissa parts for any algorithm.

IV. ISOLATION LEVEL FOR PRECISION ESTIMATION

We can estimate precision of some computation in different ways. Given a complex function, we can estimate its common error propagation coefficient and can also do the same for its parts step by step.

Although bigger fragmentation may provide a better estimation, some standard functions may have specific software and hardware implementation in some external libraries. This imposes a restriction on subdivision level for precision estimation.

For instance, computation of trigonometric functions (usually dependent on **glibc**) or sum of array (that may be vectorized and dependent on availability of AVX/SSE instructions in CPU) should be estimated as atomic numerical operations. Another difficult atomic operation which is usually dependent on external libraries is matrix multiplication (see §6.6).

Deatomization of complex operations like matrix multiplication require reimplementation of corresponding dependencies like **blas/cublas/mkl**. Such very powerful tools cannot be easily modified and transferred to other platforms and hardware settings, so they stay considered as atomic for current work and near future releases (see §9).

V. DATA STRUCTURES AND BASIC PRECISION ESTIMATIONS

Let us introduce numerical data types xf64 and others (like xf32, xf16, xbf16) consisting of a

- floating point number of type f64 (or f32, f16, bf16) representing real value,
- A number of type u8 (aka byte or unsigned char) representing the number of exact bits of the real value.

We will call them *extended floating point numbers*.

We explain below how to make arithmetic operations and calculate some mathematical functions with these numbers.

In all cases we assume that the result of computation with real values is known and defined in IEEE 754, and we provide a way to estimate precision of this result.

Some special values are zero and NaN/Inf's. From the construction, zero should always be maximally exact and NaN/Inf's cannot have any exact bits.

VI. PRECISION ESTIMATION FOR PARTICULAR FUNCTIONS

A. Addition, Subtraction and Sum

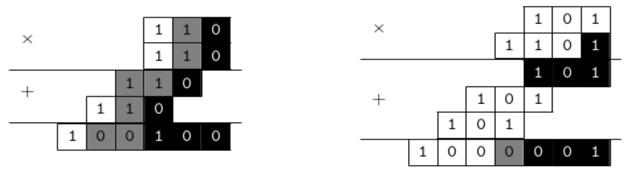
Suppose we add two binary numbers with some given white, gray and black bits. Then black bits of the result arise from addition of black bits of any of summands and from alignment to the left of the result. Estimation for gray bits can be obtained as sum of arguments inaccuracies.

Note that estimation of precision for sum of many numbers may differ from estimation if they were added one by one due to possible upper rounding for gray bits.

B. Multiplication, Division and Product

Estimation of precision for multiplication can be easily deduced to estimation of precision for addition. Let us illustrate it by some examples.

For example, take 6 with one white, gray and one black bits. Let us count exact bits of its square 36.



We see that gray bits can propagate its color to higher bits, but black bits cannot. Black bits can induce higher bits to become gray instead. If we multiply two bits, the color of result is the darkest of arguments colors.

C. Rounding

Any rounding either keeps the argument the same (in this case precision also is preserved), or change mantissa bits from some place to the right with some combination from some discrete predefined set. If the argument is changed, we should compare the first inexact bit position and the first changed bit position. If the first inexact bit position is lower than (to the right from) the first changed bit position, then the result of rounding have maximal possible number of exact bits. Otherwise, the number of exact bits is the same as in the argument. If the rounding gives zero, it is supposed to be exact.

D. Maxima and Minima

It is well known that floating point numbers do not satisfy axioms of linearly ordered set. Namely, checks of equality fails on the reflexivity rule on special values NaN/inf.

Extended floating point numbers unexpectedly break another common expectation that maximum and minimum of two numbers are always equal to one of the numbers. Namely, the numeric value is always equal to one of numeric values of arguments, but the precision may come from the other argument.

Retrieval Number: 100.1/ijainn.F109204061024
DOI: 10.54105/ijainn.F1092.04061024
Journal Website: www.ijainn.latticescipub.com

Published By:
Lattice Science Publication (LSP)

For example, for binary numbers 1.0_2 and 1.1_2 with 2 and 1 exact bits the minimum is 1.0 with 1 exact bit. Indeed, it is not reasonable to set the numeric value of minimum other than minimal of numerical values of arguments. At the same time, minimum of $1.??_2$ and $1.0?_2$ can not have more than 1 exact bit.

E. Differentiable Functions

Estimation of precision for differentiable functions is based on computation of the corresponding condition number. The standard floating point functions have explicit derivatives expressed in elementary functions. Given the number of exact bits in argument, we can compute relative error for value returned from the function. Thus, we can directly obtain estimation of result exact bits.

In this case we consider any such function as an atomic numeric operation which does not depend on particular implementation.

F. Matrix Multiplication and Tropical Semi-Ring

Estimation of matrix multiplication precision is a hard problem, because of performance issues. Actually, direct matrix multiplication by the definition is very rarely applied due to its computation complexity (naive implementation needs $O(n^3)$ operations for matrices $n \times n$).

Let us recall some notation.

Tropical semi-ring is a set of numbers and ∞ $(-\infty)$ with two binary operations *min* (*max*) and $+$ with axioms like for ones for usual numeric ring except group law for *min* (*max*): there is no inverse operation, so it is called semi-ring instead of ring.

Usually, it is considered with ∞ and *min*, but we need to complement this structure with $-\infty$ and *max* due to error estimation context.

Tropical matrix multiplication $\odot$ (also know as *min-plus matrix multiplication*) is a matrix product over tropical semi-ring. So, matrix elements can be obtained by usual matrix multiplication with replacement of $+$ and $\cdot$ with *max* and $+$.

Effective tropical matrix multiplication is a difficult problem with many approaches to particular optimizable cases (see [1,2,3][6][7]).

Unfortunately, our case does not fall into any of these cases. At the same time, we can estimate tropical matrix product from below instead of performing its precise numeric computation.

Theorem 5.1. *Let A, B, and C are matrices such that $A \cdot B = C$ of shapes ($m \times n$, $n \times k$ and $m \times k$). Suppose that A (resp. B, C) has inaccuracies $\mathcal{A}$ (resp. $\mathcal{B}$ and $\mathcal{C}$). Then*

$$\mathcal{C} \geq \mathcal{A} \odot \mathcal{B}.$$

In particular, the following inequality holds:

$$\mathcal{A} \odot \mathcal{B} \geq \frac{1}{n} \cdot log_2(2^{\mathcal{A}} \cdot 2^{\mathcal{B}}),$$

where power of 2 and log_2 are applied element-wise.

Proof. We can proof this element-wise over C and $\mathcal{C}$. So we can put $m = k = 1$. The needed inequality follows from the inequality between mean and max:

$$2^{\mathfrak{c}} \geq max(2^{\mathfrak{a}_1+\mathfrak{b}_1}, \dots, 2^{\mathfrak{a}_n+\mathfrak{b}_n}) \geq \frac{1}{n} \sum_{i=1}^{n} 2^{\mathfrak{a}_1+\mathfrak{b}_1}, \quad (4)$$

where $\mathcal{A}$ is the row $(\mathfrak{a}_1, \dots, \mathfrak{a}_n)$, and B is the column $(\mathfrak{b}_1, \dots, \mathfrak{b}_n)$, and $\mathcal{C}$ is the 1×1 matrix $(\mathfrak{c})$. This inequality is obvious.

Actually, this inequality (applied in xnumpy-1.0.0, see §9) gives an estimation that can be easily strengthened by the following. □

Theorem 5.2. *Let A, B, and C are matrices such that $A \cdot B = C$ of shapes ($m \times n$, $n \times k$ and $m \times k$). Suppose that A (resp. B, C) has inaccuracies $\mathcal{A}$ (resp. $\mathcal{B}$ and $\mathcal{C}$). Then*

$$2^{\mathcal{C}} \geq max(A \odot 2^{\mathcal{B}}, 2^{\mathcal{A}} \odot B),$$

where power of 2 and max are applied element-wise.

Proof. This estimate is stricter (applied in xnumpy-1.0.1, see §9) and gives more exact estimation of matrix multiplication precision.

The proof can be proceeded in almost the same manner like the proof of Theorem 5.1 with replacement of the initial precision estimation.

There we estimate absolute inaccuracy value for product of two numbers from below by the product of their absolute inaccuracies. We can replace the right-hand side by the maximum of one of multipliers multiplied with absolute value of inaccuracy of another one. So, Eq. (4) becomes

$$2^{\mathfrak{c}} \geq max(b_1 \cdot 2^{\mathfrak{a}_1}, a_1 \cdot 2^{\mathfrak{b}_1}, \dots, b_n \cdot 2^{\mathfrak{a}_n}, a_n \cdot 2^{\mathfrak{b}_n})$$

Remaining part of the proof is similar. □

Estimation for tropical product of matrices is identical for Theorems 6.1 and 6.2. We will propose a stronger estimation in §7.

This estimation can be performed with three usual matrix multiplications and additional memory usage for $\mathcal{A}, \mathcal{B}$ and twice for $\mathcal{C}$ (actually, they can be allocated by a single malloc/free-pair to avoid excessive heap allocations).

G. Gradients in Neural Networks

The most widely used frameworks for neural networks (Tensorflow and Torch) are based on automatic gradient computation and errors backpropagation. Gradients are computed as partial derivatives for provided collection of functions (basic collection can be extended by user-defined functions with gradients). So, the computation does not actually involve numeric differentiation. At the same time, training of neural network until its convergence can lead to subtraction of close numbers and very small inexact values of the loss function. As a result, gradients can become inexact, and the neural network can lose numerical stability.

Calculation of precision for gradients requires calculation of derivatives for arithmetic functions and standard collection of trigonometric and other functions. So, it can be deduced to previously discussed cases. Gradients of matrix multiplication can be deduced to matrix multiplication with transposed matrix. Namely, if we have matrices A and B, and $A \cdot B$ receives gradient G, then gradients for A and B are equal, respectively, $G \cdot B^T$ *and* $A^T G$. So, if some of matrices A and B is inexact, then its inexactness contribute to inexactness in both forward and backward propagation steps. Precision of convolution layers and their gradients are calculated analogously to the same for matrix multiplications.

Retrieval Number: 100.1/ijainn.F109204061024
DOI: 10.54105/ijainn.F1092.04061024
Journal Website: www.ijainn.latticescipub.com

Published By:
Lattice Science Publication (LSP)

VII. HÖLDER'S INEQUALITY AND MATRIX MULTIPLICATION PRECISION

Theorem 6.1. *Let A, B, and C are matrices such that $A \cdot B = C$. Suppose that A (resp. B, C) has inaccuracies $\mathcal{A}$ (resp. $\mathcal{B}$ and $\mathcal{C}$). Then*

$$\mathcal{C} \geq \mathcal{A} \odot \mathcal{B}.$$

In particular, the following inequality holds for any $p > 1$:

$$\mathcal{A} \odot \mathcal{B} \geq \frac{1}{np} \cdot log_2(2^{p\mathcal{A}} \cdot 2^{p\mathcal{B}}),$$

where power of 2 and log_2 are applied element-wise.

The proof is actually the same with replacement of inequality between the mean value and maximum with inequality between of generalized mean (also known as "*power mean*") of degree p.

This seems to be trivial due to addition of p into power of 2 and into the division outside of logarithm. Actually, it is not trivial, because the dot inside the brackets means matrix multiplication, so we can not reduce p in the formula above. Moreover, there is generalized mean inequality (also known as "Hölder's inequality")

$$M_p(x_1, \ldots, x_n) \leq M_q(x_1, \ldots, m_n),$$

for $p < q$, where

$$M_p(x_1, \ldots, x_n) := \sqrt[p]{\frac{1}{n}\sum_{i=1}^{n} x_i^p}.$$

and the limit property

$$\lim_{p\to\infty} M_p(x_1, \ldots, x_n) = max(x_1, \ldots, x_n).$$

It seems that taking this inequality for large p can give a better estimation of tropical product. But its application is limited by floating point overflows.

Theorem 6.2. *Let A, B, and C are matrices such that $A \cdot B = C$ of shapes $(m \times n, n \times k$ and $m \times k)$. Suppose that A (resp. B, C) has inaccuracies $\mathcal{A}$ (resp. $\mathcal{B}$ and $\mathcal{C}$). Then*

$$2^{p \cdot \mathcal{C}} \geq max(A \odot 2^{p \cdot \mathcal{B}}, 2^{p \cdot \mathcal{A}} \odot B),$$

for any $p > 1$, where power of 2 and max are applied element-wise.

Proof. This Theorem is simply a combination of Theorems 5.2 and 6.1. □

VIII. IMPLEMENTATION

We provide an implementation of our approach (see https://github.com/netay/xnumpy and https://gitflic.ru/project/kryptodpi/xnumpy) by extending widely applied NumPy library (see [4]). There "black bits" are estimated, i. e. if bits or digits are shown as inexact by this library, then they are meaningless.

One exception is precision computation for matrix product. If we compute matrix product, we rely on its numerical values provided by libraries like **blas/cublas/mkl**. At the same time, precision computation itself requires matrix multiplication. Due to performance reasons, these matrix multiplications are also computed by the same libraries, although in some cases it can give inexact matrix elements, and this can imply sometimes inexact precision estimations. But if we rely on some bit inexact numerical values, then we can rely on some bit inexact precision estimation in the same numerical computation.

Actually, some minor inaccuracies for the condition number computation can also happen, but they are usually can be omitted.

IX. CONCLUSIONS

Here we consider well known issues related to computations with floating point numbers. These computations are standard, but they introduce numerical inaccuracies that can lead to loss of significance for all the computation. It may be crucial for computations with huge number of floating point operations and high rate of quantization like neural networks training. We provide algorithms and their implementations for automatic estimation of inaccuracies during computations. It can help to avoid making unreliable conclusions based on inaccurate computations.

X. FUTURE WORK

All these estimations hold independent upon platform, software and hardware details. The implementation is already available for CPU and will be extended for GPU. Also, more advanced algorithms such as linear algebra and statistic methods will be overloaded to support precision loss tracking automagically.

XI. ACKNOWLEDGMENTS

The author is grateful to his Kryptonite colleagues Vasily Dolmatov, Dr. Nikita Gabdullin and Dr. Anton Raskovalov for fruitful discussions of topic and results and for assistance in testing the xnumpy library.

DECLARATION STATEMENT

After aggregating input from all authors, I must verify the accuracy of the following information as the article's author.

- **Conflicts of Interest/ Competing Interests:** Based on my understanding, this article has no conflicts of interest.
- **Funding Support:** This article has not been funded by any organizations or agencies. This independence ensures that the research is conducted with objectivity and without any external influence.
- **Ethical Approval and Consent to Participate:** The content of this article does not necessitate ethical approval or consent to participate with supporting documentation.
- **Data Access Statement and Material Availability:** The adequate resources of this article are publicly accessible.
- **Authors Contributions:** The authorship of this article is contributed equally to all participating individuals.

REFERENCES

1. Shucheng Chi, Ran Duan, Tianle Xie, and Tianyi Zhang. Faster min-plus product for monotone instances. In Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing, pages 1529–1542, 2022. https://doi.org/10.1145/3519935.3520057

2. Jana Cslovjecsek, Friedrich Eisenbrand, Michal Pilipczuk, Moritz Venzin, and Robert Weismantel. Efficient sequential and parallel algorithms for multistage stochastic integer programming using proximity. arXiv preprint arXiv:2012.11742, 2020. https://doi.org/10.48550/arXiv.2012.11742
3. Yuzhou Gu, Adam Polak, Virginia Vassilevska Williams, and Yinzhan Xu. Faster monotone min-plus product, range mode, and single source replacement paths. arXiv preprint arXiv:2105.02806, 2021. https://doi.org/10.48550/arXiv.2105.02806
4. Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. Nature, 585(7825):357–362, September 2020. https://doi.org/10.1038/s41586-020-2649-2
5. Igor V. Netay. Influence of digital fluctuations on behavior of neural networks. Indian Journal of Artificial Intelligence and Neural Networking (IJAINN), 3:1–7, December 2022. DOI: 10.54105/ijainn.A1061.123122
6. Mohanty, S., & Sahoo, B. (2020). Parallel Algorithms for Discovering Planted (l, d) Motif. In International Journal of Innovative Technology and Exploring Engineering (Vol. 9, Issue 4, pp. 1452–1461). https://doi.org/10.35940/ijitee.d1521.029420
7. Mustafa Basthikodi, Ahmed Rimaz Faizabadi, Waseem Ahmed, HPC Based Algorithmic Species Extraction Tool for Automatic Parallelization of Program Code. (2019). In International Journal of Recent Technology and Engineering (Vol. 8, Issue 2S3, pp. 1004–1009). https://doi.org/10.35940/ijrte.b1188.0782s319
8. Kumar, P., & Rawat, S. (2019). Implementing Convolutional Neural Networks for Simple Image Classification. In International Journal of Engineering and Advanced Technology (Vol. 9, Issue 2, pp. 3616–3619). https://doi.org/10.35940/ijeat.b3279.129219
9. Young, L., York, J. R., & Kil Lee, B. (2023). Implications of Deep Compression with Complex Neural Networks. In International Journal of Soft Computing and Engineering (Vol. 13, Issue 3, pp. 1–6). https://doi.org/10.35940/ijsce.c3613.0713323
10. Magapu, H., Krishna Sai, M. R., & Goteti, B. (2024). Human Deep Neural Networks with Artificial Intelligence and Mathematical Formulas. In International Journal of Emerging Science and Engineering (Vol. 12, Issue 4, pp. 1–2). https://doi.org/10.35940/ijese.c9803.12040324

AUTHORS PROFILE

Igor V. Netay, MS in Mathematics in MSU and IUM, Moscow, Russia; PhD in IITP, Moscow, Russia. He is a researcher at JSRPC Kryptonite and at IITP. The principal areas of research in which the author has worked are: Neural Networks, Machine Learning, Algebraic Geometry, Algebraic Topology, Representation Theory, Dynamical systems. (1) I. V. Netai, “Parabolically connected subgroups”, Mat. Sb., 202:8 (2011), 81–94; Sb. Math., 202:8 (2011), 1169–1182 (2) I. V. Netay, “Syzygy Algebras for Segre Embeddings”, Funkts. Anal. Prilozh., 47:3 (2013), 54–74; Funct. Anal. Appl., 47:3 (2013), 210–226 (3) V. V. Buchstaber, I. V. Netay, “Hirzebruch functional equation and elliptic functions of level d”, Funct. Anal. Appl., 49:4 (2015) (4) I. V. Netay, “Syzygies of quadratic Veronese embedding”, Sb. Math., 208:2 (2017) (5) Igor V. Netay, Alexei V. Savvateev, “Sharygin Triangles and Elliptic Curves", Bull. Korean Math. Soc., 54:5 (2017), 1597-1617. (6) I.