Propagating Uncertainty via Kolmogorov-Arnold Networks

Martine Ceberio, Christoph Lauter, Vladik Kreinovich, Olga Kosheleva,
Leobardo Valera, Christian Servin, Jose David Diaz Roman,
Boris Jesus Mederos Madrazo, Jose Manuel Mejia Muñoz, and
Nguyen Hoang Phuong

Martine Ceberio
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: mceberio@utep.edu

Christoph Lauter
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: cqlauter@utep.edu

Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: vladik@utep.edu

Olga Kosheleva
Department of Teacher Education, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: olgakk@utep.edu

Leobardo Valera
Computer Science and Information Technology Systems Department
El Paso Community College, 919 Hunter, El Paso, TX 79915, USA,
e-mail: leobardovalera@gmail.com

Christian Servin
Computer Science and Information Technology Systems Department
El Paso Community College, 919 Hunter, El Paso, TX 79915, USA, e-mail: cservin1@epcc.edu

Jose David Diaz Roman
Institute of Engineering and Technology, Universidad Autónoma de Ciudad Juárez (UACJ),
Ciudad Juárez 32310, México, e-mail: david.roman@uacj.mx

Boris Jesus Mederos Madrazo
Institute of Engineering and Technology, Universidad Autónoma de Ciudad Juárez (UACJ),
Ciudad Juárez 32310, México, e-mail: boris.mederos@uacj.mx

Jose Manuel Mejia Muñoz
Institute of Engineering and Technology, Universidad Autónoma de Ciudad Juárez (UACJ),
Ciudad Juárez 32310, México, e-mail: jose.mejia@uacj.mx

Nguyen Hoang Phuong
Artificial Intelligence Division, Information Technology Faculty, Thang Long University
Nghiem Xuan Yem Road, Hoang Mai District, Hanoi, Vietnam
e-mail: nhphuong2008@gmail.com

Abstract A significant amount of computations process data. Most of this data comes either from measurements, or from the results of some previous processing of measurement results. Measurement are not absolutely accurate: the measurement results are, in general, different from the actual (unknown) values of the corresponding quantities. It is therefore important to analyze how this difference affects the results of data processing. In particular, since a large number of data processing is currently performed by neural networks, it is important for provide this analysis for neural-based data processing – and techniques are known for this analysis. These techniques have been designed for deep neural networks, in which the activation function is fixed – and training means adjusting the weights which with inputs enter into a neuron. Experiments has shown that in many situations, it is advantageous to use different activation functions for different neuron – and to use training to select an appropriate activation function for each neuron. Such neural networks are known as Kolmogorov-Arnold networks. In this paper, we provide uncertainty propagation techniques for such networks.

1 Formulation of the problem

Traditional neural networks: a brief reminder. Neural networks are currently used to process data, i.e., to transform the input values $x_1 \ldots, x_n$ into the desired output $y = f(x_1, \ldots, x_n)$. A general neural network if formed by *neurons*, i.e., devices that transform values $z_1, \ldots, z_m$ into the new value $z = s(w_0 + w_1 \cdot z_1 + \ldots + w_m \cdot z_m)$ for some numbers $w_0, w_1, \ldots, w_m$. Here, $s(z)$ is a nonlinear function known as the *activation function*. Some neurons take, as inputs, the original inputs $x_1, \ldots, x_n$, other neurons take, as inputs, the values generated by some other neurons. One of the neuron produces the value y.

In a traditional neural network, usually all the neurons use the same activation function. Sometimes, several neurons use different activation functions – but still, for each neuron, the activation function is fixed. In this case, the only thing that we can change to compute different functions $y = f(x_1, \ldots, x_n)$ are the weights w_i corresponding to different neurons. To find the proper weights, neural networks use a special iterative *backpropagation* algorithm that implements gradient descent. Namely, on each iteration, each weight w_k is updated by using the following gradient descent formula:

$$w_k^{next} = w_k^{previous} - \alpha \cdot \frac{\partial J}{\partial w_k},$$

where $\alpha > 0$ is some appropriate constant – that changes from iteration to iteration – and J is the objective function that describes how close the result y_{NN} of applying a neural network is to the desired value y. The simplest such objective function is $J = (y_{NN} - y)^2$.

From traditional neural networks to Kolmogorov-Arnold networks (KAN). The efficiency of a neural network strongly depends on the selection of the ac-

tivation function. Empirically, in general, the so-called ReLU activation function $s(z) = \max(0, z)$ works the best, but for some specific classes of problems, other activation functions work better.

As of now, researchers and practitioners use trial-and-error to find the best activation function for their class of problems. A natural idea is to automate this process, i.e., to consider a whole family of activation functions, e.g., the family consisting of the functions

$$s(z) = c_1 \cdot e_1(z) + \ldots + c_p \cdot e_p(z), \tag{1}$$

where the basis functions $e_j(z)$ are fixed, and the coefficient c_j are selected based on a similar gradient descent approach; see, e.g., [1, 2, 3]. Such neural networks are known as *Kolmogorov-Arnold networks* (KAN, for short), in honor of two mathematicians who proved that every continuous function can be represented as a superposition of addition and functions of one variable – exactly what neural networks are doing.

The same back-propagation version of gradient descent can be used to find the values of the coefficients $c_{j,k}$:

$$c_{j,k}^{next} = c_{j,k}^{previous} - \alpha \cdot \frac{\partial J}{\partial c_{j,k}}.$$

For KAN networks, it is possible to simplify computations. At first glance, training a KAN network is more complicated than training a traditional neural networks. Indeed, now, in addition to the weights w_k, we also need to find the coefficients $c_{j.k}$. However, it is possible to simplify the corresponding computations by using the following alternative representation of the computations performed by KAN.

Indeed, the transformation performed by each neuron can be viewed as a composition of two transformations:

- first, we compute a linear combination $z_0 = w_0 + w_1 \cdot z_1 + \ldots + w_m \cdot z_m$, and
- then, we apply the activation function $s(.)$ to this linear combination, resulting in $z = s(z_0)$.

From this viewpoint, we can view the neural network as consisting of inter-changing layers: a *linear* (L) layer (that computes a linear combination is followed by a *nonlinear* (NL) layer that applies a nonlinear function of one variable. Each L layer is followed by a NL layer, and each NL later is followed by a L layer – until we come to the layer that produces the computation results.

In the traditional neural networks, usually, we combine each L later with the following NL layer. This combined pair of layers computes first the expression $z_0 = w_0 + w_1 \cdot z_1 + \ldots + w_m \cdot z_m$, and then the value $z = s(w_0 + w_1 \cdot z_1 + \ldots + w_m \cdot z_m)$ – which is exactly what is usually understood by a neuron. This combination is just a matter of convenience, both L and NL computations are computed anyway.

Alternatively, we can combine the L and NL layers into pairs differently: namely, we can combine each NL layer with the following L layer. In this case, first, the NL layer computes the values $t_j = s_j(t_j)$, and then the L layer computes a linear

combination of these results:

$$z = w_0 + w_1 \cdot t_1 + \ldots + w_m \cdot t_m = w_0 + w_1 \cdot s_1(z_1) + \ldots + w_m \cdot s_m(z_m).$$

If we take into account that each activation functions $s_j(t_j)$ is a linear combination (1) of basic functions, then we get

$$z = w_0 + \sum_{j=1}^{m} w_j \cdot \sum_{k=1}^{p} c_{j,k} \cdot e_k(z_j),$$

i.e.,

$$z = w_0 + \sum_{j=1}^{m} \sum_{k=1}^{p} w_j \cdot c_{j,k} \cdot e_k(z_j).$$

We can simplify this expression by denoting $C_{j,k} \stackrel{\text{def}}{=} w_j \cdot c_{j,k}$, $C_{j,0} \stackrel{\text{def}}{=} w_0$, and $e_0(z) \stackrel{\text{def}}{=} 1$. With these notations, the transformation performed by the pair of NL and L layers takes the following form:

$$z = \sum_{j=1}^{m} \sum_{k=0}^{p} C_{j,k} \cdot e_k(z_j). \tag{2}$$

Thus, we only need to train the coefficients $C_{j,k}$.

Need to take uncertainty propagation into account. Once a neural network is trained, it is used to process data, i.e., to compute the desired values $y = f(x_1, \ldots, x_n)$ based on the inputs $x_1, \ldots, x_n$. For example, in geophysical applications, data processing may take the results $x_1, \ldots, x_n$ of different measurements – seismic, magnetic, gravitational, etc. – and return the estimate for the amount of oil in a given area. Measurement are never absolute accurate: the measurement result $\widetilde{x}_i$ is, in general, somewhat different from the actual (unknown) value x_i of the corresponding quantity. The difference $\Delta x_i \stackrel{\text{def}}{=} \widetilde{x}_i - x_i$ is known as the *measurement uncertainty* or, alternatively, *measurement error*. Because of the measurement errors, the result $\widetilde{y} \stackrel{\text{def}}{=} f(\widetilde{x}_1, \ldots, \widetilde{x}_n)$ of applying the algorithm to the measurement results is, in general, somewhat different from the value $y = F(x_1, \ldots, x_n)$ that we would get if we knew the exact values $x_1, \ldots, x_n$.

In practical application, it is important to know how big is the difference $\Delta y \stackrel{\text{def}}{=} \widetilde{y} - y$ between these two numbers. For example, if our estimate for the amount of oil is $\widetilde{y} = 100$ million tons, and Δy is smaller than 20, this means that we have at least $100 - 20 = 80$ million tons – so the company can start exploration. However, if the inaccuracy is 100 ± 150 this means that maybe there is no oil at all – so it is desirable to perform additional measurements before making serious investments.

How propagating uncertainty is usually done. From the definition of the measurement errors Δx_i, we can conclude that $x_i = \widetilde{x}_i - \Delta x_i$. Thus, $y = f(x_1, \ldots, x_n) = f(\widetilde{x}_1 - \Delta x_1, \ldots, \widetilde{x}_n - \Delta x_n)$, and

$$\Delta y = \widetilde{y} - y = f(\widetilde{x}_1, \ldots, \widetilde{x}_n) - f(\widetilde{x}_1 - \Delta x_1, \ldots, \widetilde{x}_n - \Delta x_n). \tag{3}$$

Measurements are usually reasonably accurate, thus the measurement errors are usually much smaller that the actual values: $|\Delta x_i| \ll |x_i|$. Since the values Δx_i are small, we can expand the expression (3) in Taylor series in terms of Δx_i and keep only a few terms in this expansion.

Actually, we can already safely ignore quadratic terms: even if the accuracy is not so high, e.g., 10%, the square of 10% is 1% which is much much smaller. Thus, we get the following simplifies expression:

$$\Delta y = \sum_{i=1}^{n} d_i \cdot \Delta x_i,$$

where we denoted

$$d_i \stackrel{\text{def}}{=} \frac{\partial f}{\partial x_i}_{|x_1 = \widetilde{x}_1, \ldots, x_n = \widetilde{x}_n}.$$

This expression enables us to use information about the measurement errors Δx_i to get information about Δy.

Usually, the following two typical situations are considered. In the first situations, we assume that the mean value of each measurement error is 0 – this makes sense, since if it is now zero, we can simply subtract this mean value from all the measurement results and get it 0. We also assumed that all the measurement errors are independent – which is also usually the case – and that we know the standard deviation σ_i of each measurement error. In this case, the mean value of Δy is also 0, and its standard deviation σ takes the following form:

$$\sigma^2 = \sum_{i=1}^{n} d_i^2 \cdot \sigma_i^2. \tag{4}$$

Another case is when all we know about each measurement error Δx_i is the upper bound Δ_i on its absolute value: $|\Delta x_i| \le \Delta_i$. In this case, one can show that the only conclusion we can make about Δy is that it is located somewhere on the interval $[-\Delta, \Delta]$, where

$$\Delta = \sum_{i=1}^{n} |d_i| \cdot \Delta_i. \tag{5}$$

In both cases, we need to know the partial derivatives c_i. In some cases, the function $f(x_1, \ldots, x_n)$ is described by a simple expression. For example, if we use Ohm's law $V = I \cdot R$ with $x_1 = I$, $x_2 = R$, and $y = V$, then we have $f(x_1, x_2) = x_1 \cdot x_2$. In such cases, we can simply differentiate this expression, and get the desired derivatives.

Unfortunately, in many other situations, we do not have such an explicit expression – and this is exactly the case with data processing performed by neural networks. In such cases, one of the usual ways to estimate the partial derivatives is to use the fact that each derivative is the limit of the corresponding ratios

$$c_i = \lim_{h \to 0} \frac{f(\widetilde{x}_1, \ldots, \widetilde{x}_{i-1}, \widetilde{x}_i + h, \widetilde{x}_{i+1}, \ldots, \widetilde{x}_n) - \widetilde{y}}{h},$$

and thus, for sufficiently small h, can be approximated by such a ratio:

$$c_i \approx \frac{f(\widetilde{x}_1, \ldots, \widetilde{x}_{i-1}, \widetilde{x}_i + h, \widetilde{x}_{i+1}, \ldots, \widetilde{x}_n) - \widetilde{y}}{h}.$$

Another usual way to compute σ is to use Monte-Carlo simulations, i.e., to generate, for $k = 1, \ldots, K$, the values $x_i^{(k)} = \widetilde{x}_i + \eta_i$, where each η_i is normally distributed with mean 0 and standard deviation σ_i, compute $y^{(k)} = f(x_1^{(k)}, \ldots, x_1^{(k)})$, and then estimate σ as

$$\sigma^2 = \frac{1}{K} \cdot \sum_{k=1}^{K} (y^{(k)} - y)^2.$$

Why this does not work well for neural computations. The main limitation of the usual approach is that it is very time-consuming. If we use numerical differentiation, then we need to call the method for computing the value $f(x_1, \ldots, x_n)$ $n+1$ times: first, to compute the value $\widetilde{y} = f(\widetilde{x}_1, \ldots, \widetilde{x}_n)$, and then n times to compute n values $f(\widetilde{x}_1, \ldots, \widetilde{x}_{i-1}, \widetilde{x}_i + h, \widetilde{x}_{i+1}, \ldots, \widetilde{x}_n)$ corresponding to $i = 1, \ldots, n$. For large n, this means increasing the overall computation time by a factor of $n+1$. For $n = 20$, this would mean that estimating the uncertainty of the result $\widetilde{y}$ would take 20 times longer than computing this result $\widetilde{y}$; this is too much.

If we Monte-Carlo approach with K iteration, then we need to call the method f as many as $K+1$ times, where the value K is determined based on the fact that the accuracy of this approach is $1/\sqrt{K}$; see, e.g., [6]. To get a reasonable accuracy 20% in estimating the uncertainty, we need $K = 25$ – so we need to spend 25 times more time to estimate the uncertainty of the result than to estimate the result itself.

So what can we do? For the usual deep neural networks, it turned out that we can use the fact that when we train a neural network, we compute some derivatives. These are not exactly the same derivatives that we need, but it turns out that we can actually compute the desired derivatives based on the back-propagation ones; see, e.g., [4, 5]. This means that we can compute all the values c_i – and thus, the desired values σ and Δ – by simply adding a single backpropagation step to the original computations, i.e., crudely speaking, by only doubling the computation time.

In this paper, we show that a similar idea can be applied for KAN networks as well.

2 Analysis of the problem and the resulting solution

What we want and what we have. As we have mentioned in the previous section, what we have after training are the partial derivatives of the training's objective function J over the coefficients:

$$\frac{\partial J}{\partial C_{j,k}}.$$

What we want are the derivatives of y with respect to the unknown x_i:

$$\frac{\partial y}{\partial x_i}.$$

So, the question is how can we compute the desired derivatives based on the known ones.

Similar to the case of usual neural networks, let us consider the first layer. The very first layer is when the neuron directly process the inputs $x_1,\ldots,x_n$. For each neuron ℓ from this layer, the resulting signal takes the form

$$y_\ell = \sum_{i=1}^{n}\sum_{k=0}^{p} C_{\ell,i,k}\cdot e_k(x_i).$$

Due to the chain rule, the partial derivative of y with respect to x_i has the form

$$\frac{\partial y}{\partial x_i} = \sum_{\ell}\frac{\partial y}{\partial y_\ell}\cdot\frac{\partial y_\ell}{\partial x_i} = \sum_{\ell}\frac{\partial y}{\partial y_\ell}\cdot\left(\sum_{k=0}^{p} C_{\ell,i,k}\cdot e_k'(x_i)\right), \tag{6}$$

where $e'(x_i)$, as usual, means the derivative of the function. We know the coefficients $C_{\ell,i.k}$, we can easily compute the values $e_k'(x_i)$. Thus, to find the desired derivatives of y with respect to x_i, we need to compute the partial derivatives of y with respect to y_ℓ.

To find these derivatives, let us recall that from the training, we know, for some i_0 and k_0, the following derivative:

$$\frac{\partial J}{\partial C_{\ell,i_0,k_0}} = \frac{\partial J}{\partial y_\ell}\cdot\frac{\partial y_\ell}{\partial C_{\ell,i_0,k_0}} = \frac{\partial J}{\partial y_\ell}\cdot e_{k_0}(x_{i_0}).$$

Thus,

$$\frac{\partial J}{\partial y_\ell} = \frac{1}{e_{k_0}(x_{i_0})}\cdot\frac{\partial J}{\partial C_{\ell,i_0,k_0}}. \tag{7}$$

So, we can compute the derivative of J with respect to y_ℓ. What we want is the derivatives of y with respect to y_ℓ. To find this derivative, let us recall that the only way J depends on y_ℓ is through the dependence of y on y_ℓ, and that the objective function J depends on y the known way. Thus,

$$\frac{\partial J}{\partial y_\ell} = \frac{\partial J}{\partial y}\cdot\frac{\partial y}{\partial y_\ell}.$$

Therefore,

$$\frac{\partial y}{\partial y_\ell} = \left(\frac{\partial J}{\partial y}\right)^{-1}\cdot\frac{\partial J}{\partial y_\ell}.$$

Substituting (4) into this formula, we conclude that

$$\frac{\partial y}{\partial y_\ell} = \left(\frac{\partial J}{\partial y}\right)^{-1} \cdot \frac{1}{e_{k_0}(x_{i_0})} \cdot \frac{\partial J}{\partial C_{\ell,i_0,k_0}}. \tag{8}$$

Substituting the expression (8) into the formula (6), we get the expression for the desired partial derivative of y with respect to x_i:

$$d_i = \frac{\partial y}{\partial x_i} = \left(\frac{\partial J}{\partial y}\right)^{-1} \cdot \frac{1}{e_{k_0}(x_{i_0})} \cdot \sum_\ell \cdot \frac{\partial J}{\partial C_{\ell,i_0,k_0}} \cdot \left(\sum_{k=0}^{p} C_{\ell,i,k} \cdot e_k'(x_i)\right). \tag{9}$$

Resulting algorithm. For each inputs $x_1,\ldots,x_n$, to find the uncertainty of the results of KAN computations, after the computation, we perform a part of a back-propagation step – namely, we compute the partial derivatives without actually changing the coefficients. Once we compute these derivatives, we then take the derivative corresponding to the first layer, and compute the values (9), where the sum is over all neurons ℓ in this layer.

Then, depending on whether know the standard deviations σ_i of the inputs or the upper bounds Δ_i on the inputs' uncertainty, we:

- use the formula (4) to compute the standard deviation σ of the result of data processing, or
- use the formula (5) to compute the strict upper bound Δ on the uncertainty of the result of data processing.

3 What if we cannot add the backpropagation step

Possible problem with the proposed solution. The proposed solution requires that for KAN-based computations, to estimate the uncertainty of the computation result, we need to supplement the usual "forward" computational process, when we go layer-by-layer from the inputs $x_1,\ldots,x_n$ to the final result y, with one iteration of the "backward" process – back-propagation, when we go backwards layer by layer to compute the corresponding derivatives.

If we use this method, then the computation time increases by (at least) a factor of 2. In some computations, on the edge of computer abilities, it may not be possible to double computations time. For these situations, we need to come up with alternative methods that would not increase the computation time. In this section, we describe such methods.

Main idea. The reason why in the proposed method, we double the computation time, is that we need to compute the values d_i based on which we compute σ or Δ (by using the formula (4) and (5)). These values are, in general, different for different inputs.

Since we do not have time to compute the exact values d_i, a natural idea is to have some bounds on these values – and thus, on the corresponding value Δ and σ. For this purpose, we can use a usual statistics approach; see, e.g., [6]: namely, we can view d_i as random variables varying depending on the inputs. It makes sense to assume that, in general, these random variables are independent. From this viewpoint, according to the formulas (4) and (5), the expressions for σ^2 and Δ are the sum of a large number of relatively small independent variables. In this case, according to the Central Limit Theorem (see, e.g., [6]), the resulting distributions for Δ and σ^2 are close to Gaussian. A Gaussian distribution is uniquely determined by its mean m and standard deviation s, and, with some confidence, we can conclude that the actual value of the resulting variable is bounded by $m + k_0 \cdot s$, where:

- for k_0, we get confidence 95%;
- for $k_0 = 3$, we get confidence 99.9%;
- for $k_0 = 6$, we get confidence $1 - 10^{-8}$.

The values m and s can be estimates based on the values d_i computed during the training of the neural network. Computing these values practically does not require any additional computation time, since, by using the formula (9), we can easily compute these values based on the partial derivatives that are actually computed during the back-propagation step. Of course, for these computations, we need to only consider the values corresponding to the last iteration of the training process, when the KAN is already trained to compute the desired function $y = f(x_1, \ldots, x_n)$.

How to implement this idea: a general comment. In general, for the sum $\eta = \eta_1 + \ldots + \eta_n$ of n independent random variables, its mean value $E[\eta]$ is the sum of the mean values $E[\eta] = E[\eta_1] + \ldots + E[\eta_n]$, and its variance $V[\eta]$ is equal to the sum of variances. So, from the equations (4) and (5), we conclude that:

$$E[\sigma^2] = \sum_{i=1}^{n} E[d_i^2] \cdot \sigma_i^2, \;\; V[\sigma^2] = \sum_{i=1}^{n} V[d_i^2] \cdot \sigma_i^4, \tag{10}$$

$$E[\Delta] = \sum_{i=1}^{n} E[|d_i|] \cdot \Delta_i, \;\; V[\Delta] = \sum_{i=1}^{n} V[|d_i|] \cdot \Delta_i^2. \tag{11}$$

Thus, we arrive at the following algorithm.

How to implement this idea: case of random errors. During the last stages of the training of KAN, we use the technique for the previous section to transform the derivatives computed during back-propagation into the partial derivatives $d_1, \ldots, d_n$. As a result, for each i, we get a sample of values d_i corresponding to different inputs. Based on this sample, we compute the sample mean $E[d_i^2]$ and sample variance $V[d_i^2]$.

Then, when training is over, when we use KAN to process the inputs x_i with known uncertainties Δ_i, we conclude that with the level of confidence depending on our choice of k_0, we have $\sigma \le \sqrt{E[\sigma^2] + k_0 \cdot \sqrt{V[\sigma^2]}}$, where $E[\sigma^2]$ and $V[\sigma^2]$ are computed by using the formula (10).

How to implement this idea: case of interval uncertainty. During the last stages of the training of KAN, we use the technique for the previous section to transform the derivatives computed during back-propagation into the partial derivatives $d_1, \ldots, d_n$. As a result, for each i, we get a sample of values d_i corresponding to different inputs. Based on this sample, we compute the sample mean $E[|d_i|]$ and sample variance $V[|d_i|]$.

Then, when training is over, when we use KAN to process the inputs x_i with known uncertainties Δ_i, we conclude that with the level of confidence depending on our choice of k_0, we have $\Delta \le E[\Delta] + k_0 \cdot \sqrt{V[\Delta]}$, where $E[\Delta]$ and $V[\Delta]$ are computed by using the formula (11).

Comment. What if the KAN is already trained, and the derivatives were not recorded during training? In this case, we can perform a few additional training steps and thus, get the desired sample of values d_i.

Acknowledgments

This work was supported by the AT&T Fellowship in Information Technology, by the Institute for Risk and Reliability, Leibniz Universitaet Hannover, Germany, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) Focus Program SPP 100+ 2388, Grant Nr. 501624329, by the European Union under the project ROBOPROX (No. CZ.02.01.01/00/22 008/0004590), by the Center of Excellence in Econometrics, Faculty of Economics, Chiang Mai University, Thailand, by the Ho Chi Minh City University of Banking, Vietnam, and by Thang Long University, Hanoi, Vietnam.

References

1. R. Bouarah, V. Barbe, A. Volkova, and F. de Dinechin, *KANHard: Training Hardware-Friendly Kolmogorov-Arnold Networks*, Technical Report, https://hal.science/hal-05388006v1, 2025.
2. B. Hadj Kilani, *Convolutional Kolmogorov-Arnold Networks: A Survey*, Technical Report, https://hal.science/hal-05177765, 2025.
3. Y. Hou, T. Ji, D. Zhang, and A. Stefanidis, *Kolmogorov-Arnold Networks: A Critical Assessment of Claims, Performance, and Practical Viability*, arXiv: 2407.11075, https://arxiv.org/abs/2407.11075, 2025.
4. O. Kosheleva and V. Kreinovich, "How to propagate uncertainty via AI algorithms", *Proceedings of the 10th International Workshop on Reliable Engineering Computing REC'2024*, Beijing, China, October 26–27, 2024, pp. 5–12.
5. C. Q. Lauter, M. Ceberio, V. Kreinovich, and O. M. Kosheleva, "Uncertainty quantification for results of ai-based data processing: towards more feasible algorithms", In: F. Pavese, A. Forbes, A. Bosnjakovic, S. Eichstaedt, and J. Sousa (eds.), *Advanced Mathematical and Computational Tools for Metrology and Testing XIII*, World Scientific, Singapore, 2025, pp. 95–108.
6. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman and Hall/CRC, Boca Raton, Florida, 2011.