THE SIGMA-PI NEURON AND RESOLVING THE GENERAL "FACTOR MULTIPLICATION" PROBLEM IN ARTIFICIAL NEURAL NETWORKS

R. Yu. Portsev* and A. V. Makarenko**

*,**Trapeznikov Institute of Control Sciences, Russian Academy of Sciences, Moscow, Russia

*✉ poruss@mail.ru, **✉ avm.science@mail.ru

Abstract. This paper proposes a novel design of the Sigma-Pi neuron with the controlled automatic activation of the factor multiplication operation during the training phase of a neural network. The solution can be transparently integrated into both current and prospective neural architectures; it performs direct neural computation of a wide range of mathematical relations that are commonly implemented by neural networks in a memoization mode. The direct execution of the multiplication operation yields smaller network errors; this advantage is very clear when the network processes input data outside the training domain. As proved below, a single Sigma-Pi neuron can implement the XOR function of arbitrary dimension. The effectiveness and computational capabilities of the novel design, as well as its superiority over classical fully connected feedforward neural networks, are demonstrated for the neural modeling of polynomials and operations of vector-matrix algebra. The proposed Sigma-Pi neuron design is expected to find application both in surrogate neural network models for computation and simulation and in general AI+Science tasks.

Keywords: Sigma-Pi neuron, factor multiplication operation, XOR function, surrogate models.

INTRODUCTION

In 1943, W. McCulloch and W. Pitts published their seminal paper [1], proposing a formal model of an artificial neuron. In 1958, F. Rosenblatt generalized this model [2]. In modern notation, it takes the form

$$s = \mathbf{w}\cdot\mathbf{x} + b\,,\ z = \mathrm{g}(s)\,, \qquad (1)$$

and is well-known as the McCulloch–Pitts neuron. In formula (1), the symbols have the following meanings: $\mathbf{x}$ is the input data vector of dimension N, $\mathbf{x}\in\mathbb{R}^N$; $\mathbf{w}$ is the vector of trainable weights; b is a trainable bias; $\cdot$ is the scalar multiplication operation; $\mathrm{g}(\circ)$ is a nonlinear activation function; finally, z is the output. According to Rosenblatt's paradigm and modern concepts of deep learning [3], neurons of the form (1) and/or their derivatives are organized into regular layers, and the combination of layers, in turn, forms a neural network.

Direct analysis of the expression (1) shows that the main computations and adaptation in a network with Rosenblatt's architecture are concentrated in the trainable weighted summation unit $\mathbf{w}\cdot\mathbf{x}$. The activation function $\mathrm{g}(\circ)$ plays a secondary role: in the case of a multilayer structure, it prevents the composition of layers from degenerating into trivial vector-matrix multiplication (in fact, into a linear regression). Note that the vast majority of modern deep artificial neural networks (including fully connected, convolutional, recurrent, and transformer architectures, as well as combinations thereof) [3–5] operate based on this paradigm. In the KAN architecture proposed recently [6], the summation unit (1) is non-trainable (with fixed unit weights, $\mathbf{w}\equiv\mathbf{1}$), and the activation functions $\mathrm{g}(\circ)$ are trainable polynomials of limited degree.

Initially, the activation function $\mathrm{g}(\circ)$ was defined as a Heaviside threshold function:

$$z = \mathrm{g}_{01}(s) = \begin{cases} 1 & \text{for } s > a \\ 0 & \text{for } s \le a, \end{cases}$$

where a is the activation threshold, with a default

value of $a = 0$. (Neurons operated with binary signals.) Subsequently, Rosenblatt [2] introduced the sign function

$$z = \mathrm{g}_{11}(s) = \operatorname{sign} s;$$

in 1987, D.E. Rumelhart [7] proposed using either the sigmoid function or the hyperbolic tangent (making the network input continuous):

$$z = \mathrm{g}_{\mathrm{sg}}(s) = \frac{1}{1+e^{-s}}, \qquad (2)$$

$$z = \mathrm{g}_{\mathrm{th}}(s) = \operatorname{th} s.$$

In 1989, G. Cybenko proved the universal approximation theorem [8].

Theorem 1. *A feedforward artificial neural network with a single hidden layer can approximate any continuous function of several variables with arbitrary accuracy, provided that the network has a sufficient number of neurons N in the hidden layer with the sigmoid activation function* g_{sg}.

Note that Theorem 1 essentially complemented the convergence theorem for the perceptron [9]. In this context, we also mention the closely related Hecht–Nielsen theorem [10]. In 1991, K. Hornik generalized Theorem 1 to the case of arbitrary nonlinear activation functions [11]; see also [12]. It became clear that the universal approximation properties of artificial neural networks (ANNs) are, to a large extent, a property of the network structure itself. In 1992, M. Leshno et al. [13] further relaxed the requirements on the form of admissible activation functions, actually laying the theoretical foundation for using such functions as ReLU, Swish, GELU, etc. Finally, in 1999, V. Maiorov and A. Pinkus [14] established an analog of Theorem 1 for the case of a neural network with limited capacity (two hidden layers and a finite number of neurons in both layers) and the sigmoid activation function g_{sg}.

In a certain sense, Theorem 1 is a special version of the theorem on the representability of continuous functions of several variables as compositions of continuous functions of a single variable, proved in 1957 by A.N. Kolmogorov and V.I. Arnold [15–17].

Theorem 2 [Kolmogorov–Arnold Theorem]. *Each continuous function of several variables can be represented as a superposition of continuous functions of a single variable and binary summation.*

According to Theorem 2, a function f of N variables can be represented as

$$f(\mathbf{x}) = \sum_{i=1}^{2N+1} \Phi_i \left[\sum_{j=1}^{N} \phi_{i,j}\left(x_j\right) \right]. \qquad (3)$$

(This is a generalization of Doss's formulation.) In turn, Theorem 2 is related to K. Weierstrass's approximation theorem [18], dating back to 1885.

Theorem 3. *Let* f *be a continuous function taking real values and defined on a closed real interval* $[a, b]$. *For each* $\varepsilon > 0$, *there exists a polynomial* p *such that* $|f(x) - p(x)| < \varepsilon$ *for all* $x \in [a, b]$.

As a result, one faces a paradoxical situation.

Due to the conditions of Theorem 2 and formula (3), summation is the only true multidimensional operation, while other multidimensional operations (including multiplication) can be expressed as summation operations combined with one-dimensional functions.[1] And the mainstream of neural network approaches deals only with factor summation, while all other operations (including factor multiplication) are implemented through a memoization mode.[2]

However, in the practice of both mathematical modeling and computations in general, the results of the Kolmogorov–Arnold theorem (the representation (3)) are quite limited. One could even argue that they are infeasible when solving real-world problems: the analytical notation of the formulas and their computational implementations are replete with multiplication operations. Why? The answer lies in the compactness and precision of the formulas and computations in which the multiplication operation is implemented explicitly.[3]

Thus, "honest" and direct multiplication of factors is a very important and frequently used operation in neural networks. Here are some elementary examples:

- Input data specify the length and width of a rectangular emitting surface, with a uniform radiation flux density on the surface. It is required to determine the total radiation flux. Naturally, "inside" the model, one needs to compute the area of the surface.
- Input data specify the quantity of a product and its price. It is required to compute the total value for a

[1] Let us emphasize: the Kolmogorov–Arnold theorem guarantees such a representation only for continuous functions, claiming nothing regarding differentiable functions. In addition, see [14] and the results of A.G. Vitushkin [19].

[2] During the training phase, a static memory is formed within a neural network. In fact, this generates a table mapping the argument to the value of the memorized function.

[3] The following analogy seems appropriate here. According to Theorem 1, a perceptron with a single hidden layer is capable of solving any problem, but only if the number of neurons in the hidden layer is exponentially large (which is infeasible in practice). In reality, however, deep neural networks with fairly compact architectures allow solving complex problems; see the main theorem in [14]. In other words, the multiplication operation in this context is analogous to the depth of the network structure: both make the problems practically feasible.

batch of products. Naturally, the model shall be able to multiply the quantity of the product by its price internally.

Note that neural network approaches are being actively applied to build computational and simulation surrogate models[4] for estimating/predicting the properties and/or behavior of complex nonlinear systems of a physical or technical nature, including time-varying ones. In this class of problems, models often operate outside the training domain (and deviate significantly from its boundaries), which significantly degrades their quality and stability due to the limited generalization capability of the memoization mechanism.

The active factor multiplication operation in artificial neural networks can potentially be implemented as follows:

• **With the activation function defined as the square of the argument**, using the expression

$$(u+v)^2 = u^2 + 2uv + v^2, \qquad (4)$$

a two-layer network extracts the product of input factors (Fig. 1). The drawbacks of this approach are obvious: extensive use of neurons and layers; the impossibility to adaptively "grow" multiplication blocks in the necessary locations and quantities; and poor generalizability of the approach when multiplying three and more input variables.

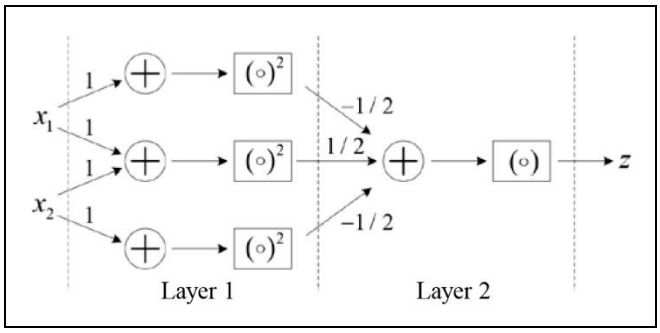

Fig. 1. The implementation of factor multiplication by a neural network with the quadratic activation function.

• **With neurons operating in a logarithmic basis,** multiplication is reduced to summation. Based on the formula

$$\log u + \log v = \log(uv),$$

by supplying logarithmic input values to the neuron and applying the exponential activation function, the network extracts the product of input factors.

This approach has an additional advantage (the automatic implementation of the ratio of factors) as well as a mixed mode (multiplication and division). Its drawbacks are also evident: the impossibility to adaptively "grow" summation blocks in the necessary locations and quantities; poor stability during network training due to the exponential activation function, which causes explosive gradient growth.

• **With the valve approach to building activation functions**, we have the element-wise product of two linear transformations of input data, one being activated by a sigmoid function [20] or more modern variants [21] (ReLU, GeLU, Swish, etc.); a variant with a linear function is also possible. (The bilinear form was originally proposed in 2007 by A. Mnih and J. Hinton [22].) Formally, this approach can be written as

$$\mathbf{z} = (\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \otimes \mathrm{a}(\mathbf{W}_2\mathbf{x} + \mathbf{b}_2), \qquad (5)$$

where $\otimes$ stands for the Hadamard product; $\mathrm{a}(\circ)$ is the activation function. With the expression (5), the neural network models quadratic relationships; in view of formula (4), this allows forming factor products.

Based on the above considerations, we emphasize two key aspects significantly influencing the performance characteristics of contemporary neural network models.

First, classical KAN architectures [6] implement the factor multiplication mechanism indirectly, through an extensive mechanism of the form (4). The MultKAN extension [23] introduces multiplication explicitly, albeit "forcibly," i.e., in the form of an additional intermediate layer after the summation layer; this worsens the network's compactness and the stability of its training. Furthermore, the number of factors to be multiplied in a network node is specified by the user (two factors are used by default) and is fixed before network training.

Second, modern transformers (consequently, large language models and foundational models in particular) implement a rather strange "multiplication scheme" in the form of (5). Moreover, this method is not only strange but also highly resource-intensive and unstable during training. (During training, a neural network may fail to find the combination of parameters required for multiplication.)

In this paper, we propose an alternative design of the so-called Sigma-Pi ($\Sigma\Pi$-) neuron for "directly" resolving the general "factor multiplication" problem in neural networks. This novel design is free from the shortcomings of the approaches mentioned above.

The remainder of this paper is organized as follows. In Section 1, we describe the design of the $\Sigma\Pi$-neuron and its main properties. Section 2 is devoted to

[4] These inexpensive and fast data-driven models (based on the black-box principle) approximate the behavior of more complex, accurate, and expensive computational (simulation) models.

numerical examples demonstrating the performance of neural networks based on the proposed solution and its advantages over the classical summation neuron.

1. THE DESIGN AND MAIN PROPERTIES OF THE SIGMA-PI NEURON

1.1. General Principles

Consider the so-called $\Sigma\Pi$-neuron consisting of the sum

$$\Sigma\Pi(\mathbf{x}) = \left[1 - \mathrm{g}_{\mathrm{sg}}(\gamma)\right] s(\mathbf{x}) + \mathrm{g}_{\mathrm{sg}}(\gamma)\, p(\mathbf{x}), \qquad (6)$$

where γ is the neuron's valve to control its operating mode (summation/multiplication), and $\mathrm{g}_{\mathrm{sg}}(\circ)$ is the sigmoid function (2). The term $s(\mathbf{x})$ is responsible for summing the components of the input vector $\mathbf{x}$:

$$s(\mathbf{x}) = b + \sum_{i=1}^{N} w_i\, x_i\,; \qquad (7)$$

it is equivalent to the summation unit in (1). The term $p(\mathbf{x})$ is responsible for multiplying the input data:

$$p(\mathbf{x}) = m \prod_{i=1}^{N} \left[1 - \mathrm{g}_{\mathrm{sg}}(\alpha_i) + \mathrm{g}_{\mathrm{sg}}(\alpha_i) x_i\right], \qquad (8)$$

where m denotes the multiplier's scaling coefficient, intended for the final renormalization of the product; α_i is the multiplier's gate for the ith factor. Note that the parameters γ, m, and α_i are trainable, as are w_i and b. Furthermore, all trainable parameters belong to the domain $\mathbb{R}$ and are continuous.

Due to formulas (6)–(8), the aggregation operator in a neuron (sum or product) is selected automatically during the training process. In pure summation mode, as $\mathrm{g}_{\mathrm{sg}}(\gamma) \to 0$, the $\Sigma\Pi$-neuron represents a classical McCulloch–Pitts neuron. As $\mathrm{g}_{\mathrm{sg}}(\gamma) \to 1$, the $\Sigma\Pi$-neuron selectively multiplies the inputs, and the factors are selected via the gates $\mathrm{g}_{\mathrm{sg}}(\alpha_i) \to 1$. In intermediate cases, the $\Sigma\Pi$-neuron implements a linear combination consisting of the sum and product of the components of the input vector $\mathbf{x}$. Note that the design of the gates (the use of the sigmoid function g_{sg}) is similar to that in the gate-based approach [20] and LSTM networks [24].

The design of the $\Sigma\Pi$-neuron can be generalized to the case of structural neuroplasticity[5] by introducing an adaptive dependence of γ on the current input data $\mathbf{x}$, the hidden state $\mathbf{h}$ of the neural network, and/or its output $\mathbf{z}$. Schematically, neuroplasticity can be written as

$$\gamma = \mathrm{G}(\circ),$$

where G is a certain mapping $\mathbb{R}^K \to \mathbb{R}$ tuned during the training/operation of the neural network.

1.2. Main Properties of the Sigma-Pi Neuron

1.2.1. The Topology of Separating Surfaces

Unlike the classical McCulloch–Pitts neuron, the $\Sigma\Pi$-neuron generates significantly more diverse and qualitatively different separating surfaces. For instance, in binary classification mode, the $\Sigma\Pi$-neuron generates three qualitatively different phase portraits for separating the input data into classes. In general form, these cases are written as

$$\mathrm{F_S} : z = \mathrm{g}_{11}\left[\sum_{i=1}^{N} x_i\right], \qquad (9)$$

$$\mathrm{F_{SP}} : z = \mathrm{g}_{11}\left[\sum_{i=1}^{N} x_i + \prod_{i=1}^{N} x_i\right], \qquad (10)$$

$$\mathrm{F_P} : z = \mathrm{g}_{11}\left[\prod_{i=1}^{N} x_i\right], \qquad (11)$$

where the case $\mathrm{F_S}$ corresponds to the classical summation neuron. For the 2D input vector, the phase portraits (9)–(11) are shown in Fig. 2; for the 3D one, in Fig. 3.

According to Figs. 2 and 3, when the *multiplication* operation is included in the neuron's aggregation operator (the topology of separating the input data vector into classes becomes essentially more complex), a single neuron can solve nonlinear problems.

Indeed, the case $\mathrm{F_S}$ (corresponding to the classical summation neuron) forms a single separating hyperplane and two regions (one per each class) for any dimension of the input data ($\dim \mathbf{x} \geq 2$). The case $\mathrm{F_{SP}}$ forms a nonlinear separating hypersurface. For $\dim \mathbf{x} = 2$, there are three isolated regions (two regions per class $z = +1$ and one per class $z = -1$). If $\dim \mathbf{x} \geq 3$, we have two regions (one per each class). And finally, the case $\mathrm{F_P}$ forms $2^{\dim \mathbf{x}}$ isolated regions (an equal number of regions per each class).

[5] Neuroplasticity in deep learning is an adaptive mechanism manifested as the capability of a neural network to dynamically change its structure and/or functionality during operation in response to changing input data or conditions/tasks.

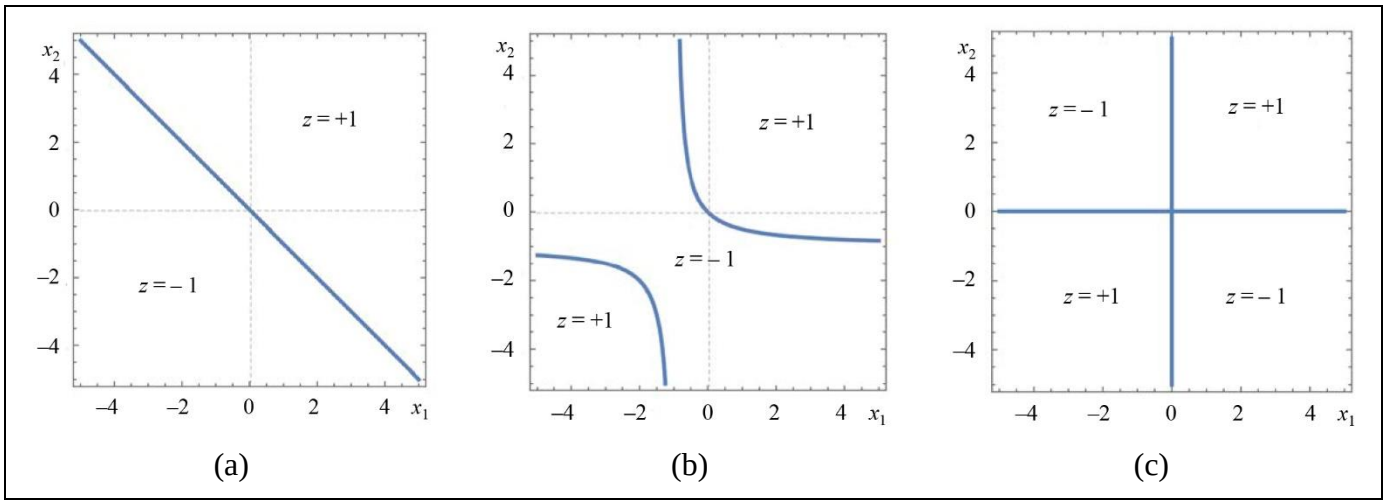

Fig. 2. The structure of separating surfaces in the case of binary classification of 2D input data by different neurons: (a) F_S, (b) F_{SP}, and (c) F_P.

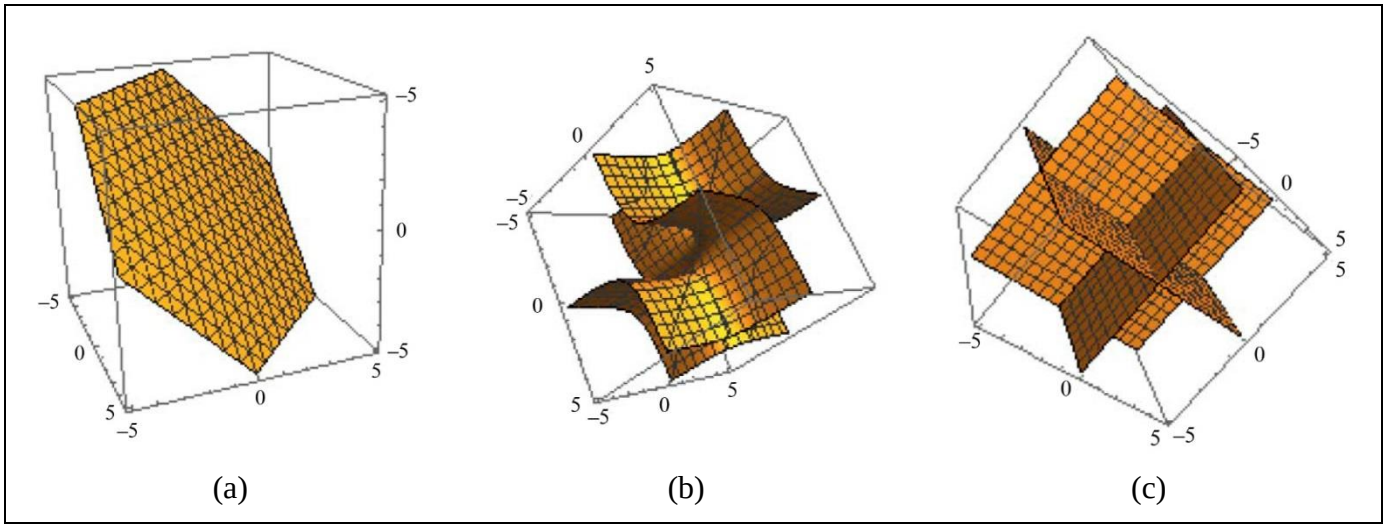

Fig. 3. The structure of separating surfaces in the case of binary classification of 3D input data by different neurons: (a) F_S, (b) F_{SP}, and (c) F_P.

1.2.2. Computing the XOR Function

As proved in [25], single-layer networks with neurons of the form (1) cannot solve even elementary non-separable problems, such as computing the XOR function (Exclusive OR). In turn, based on the results of subsection 1.2.1, the following is valid.

Theorem 4. *A single* $\Sigma\Pi$*-neuron in binary input* $\mathbf{x}\in\mathbb{Z}_2\times\ldots\times\mathbb{Z}_2$ *and output* $z\in\mathbb{Z}_2$ *bases can compute the XOR function* $\oplus_2$ *of arbitrary dimension* $N=\dim\mathbf{x}\geq 2$.

P r o o f. By the properties of the function $\oplus_2$, the result of the operation is TRUE only if the number of TRUE arguments in the current input set is odd. Therefore, we define the basis $\mathbb{Z}_2=\{-1,1\}$ and, in the case of an odd dimension of $\mathbf{x}$, represent the configuration of the $\Sigma\Pi$-neuron implementing $\oplus_2$ in the form

$$N \bmod 2 \neq 0:\ \Sigma\Pi=\prod_{i=1}^{N} x_i,\ z=\mathrm{g}_{11}(\Sigma\Pi). \qquad (12)$$

For an even dimension of $\mathbf{x}$, the configuration of the $\Sigma\Pi$-neuron is written as

$$N \bmod 2 = 0:\ \Sigma\Pi=\frac{1}{N}\sum_{i=1}^{N} x_i - 2\prod_{i=1}^{N} x_i - 1,\ z=\mathrm{g}_{11}(\Sigma\Pi). \qquad (13)$$

By verifying the expressions (12) and (13) for small N and applying induction, we finally arrive at the desired result. ♦

For clarity, Fig. 4 shows the phase portrait (13) in the case of two variables.

We emphasize the importance of Theorem 4, both from theoretical and practical standpoints: it confirms the capability of the $\Sigma\Pi$-neuron to solve certain classes of non-separable problems, while the XOR function finds application in a wide variety of fields, rang-

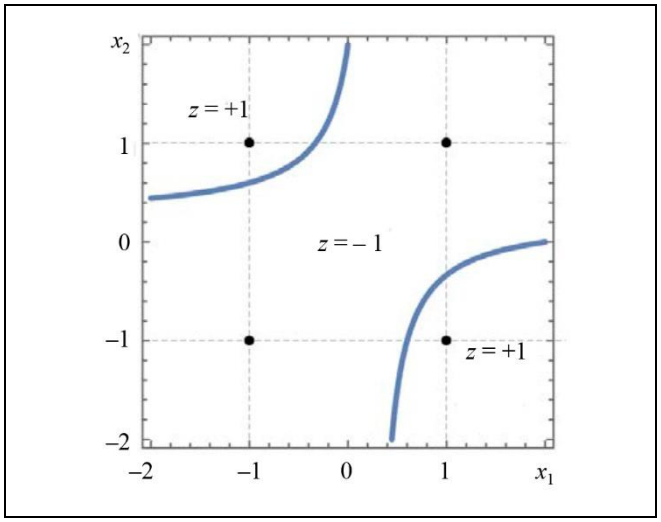

Fig. 4. The structure of separating surfaces in the case of computing the XOR function. (The circles indicate the input values in the basis $\mathbb{Z}_{\{-1,1\}}$.)

ing from low-level programming and cryptography to search and error detection algorithms. This potentially leads to the direct solution of such problems by neural networks based on $\Sigma\Pi$-neurons.

2. NUMERICAL EXPERIMENTS

2.1. General Conditions

To comprehensively analyze the properties of the $\Sigma\Pi$-neuron and to test schemes for constructing and training networks based on this neuron, we developed two frameworks. The first, purely research-oriented, was implemented in *Wolfram Mathematica*; the second, applications-relevant, was developed as a *PyTorch* extension.[6] The frameworks were used to conduct numerical experiments (modeling of various functional relationships by $\Sigma\Pi$-neural networks) and compare the results with those of classical fully connected networks. Some of the experimental results are presented below.

The initial problem statement was to investigate the modeling properties of $\Sigma\Pi$-neural networks in terms of approximating simple polynomials in two variables (subsection 2.2), scalar multiplication of vectors, and vector-matrix multiplication (subsection 2.3), both inside and outside the training domain.

For each variable (variable component), the training domain D^{TR} was defined as the interval

$$x \in \mathrm{D}^{\mathrm{TR}} \subset \mathbb{R},\ \mathrm{D}^{\mathrm{TR}} = [-2.2,\ 2.2], \qquad (14)$$

where x is the input variable (or variable component) of the relationships under analysis. The testing domain D^{TS} was slightly larger than the training one:

$$x \in \mathrm{D}^{\mathrm{TS}} \subset \mathbb{R},\ \mathrm{D}^{\mathrm{TS}} = [-3.5,\ 3.5].$$

In addition, an extended validation domain $\overline{\mathrm{D}}^{\mathrm{VL}}$ was defined:

$$x \in \overline{\mathrm{D}}^{\mathrm{VL}} \subset \mathbb{R},\ \overline{\mathrm{D}}^{\mathrm{VL}} = [-110,\ 110]. \qquad (15)$$

Direct comparison of formulas (15) and (14) shows that, in terms of amplitude, the extended validation domain was 50 times larger than the training counterpart.

Within the training D^{TR} and testing D^{TS} domains, two non-overlapping datasets were formed, namely, $\mathrm{DS}^{\mathrm{TR}}$ for training (40 000 samples) and $\mathrm{DS}^{\mathrm{TS}}$ for testing (10 000 samples), $\mathrm{DS}^{\mathrm{TR}} \bigcap \mathrm{DS}^{\mathrm{TS}} \equiv \varnothing$. Their arguments were initially represented by pseudorandom numbers with the uniform distribution:

$$\mathrm{DS}^{\mathrm{TR}}:\ x \in \mathcal{U}\left[\mathrm{D}^{\mathrm{TR}}\right],\ \mathrm{DS}^{\mathrm{TS}}:\ x \in \mathcal{U}\left[\mathrm{D}^{\mathrm{TS}}\right].$$

Then, Gaussian noise was added to the function values of these datasets:

$$z \mapsto z + \varepsilon,\ \varepsilon \in \mathcal{N}[\mu = 0,\ \sigma = 0.05].$$

On the one hand, the noise modeled certain measurement errors; on the other, it served for regularization.

Also, validation datasets were generated (100 000 samples each):

$$\mathrm{DS}^{\mathrm{VL}}:\ x \in \mathcal{U}\left[\mathrm{D}^{\mathrm{TR}}\right],\ \overline{\mathrm{DS}}^{\mathrm{VL}}:\ x \in \mathcal{U}\left[\overline{\mathrm{D}}^{\mathrm{VL}}\right],$$

to finally analyze the quality and stability of the neural networks under study. Naturally, $\mathrm{DS}^{\mathrm{TR}} \bigcap \mathrm{DS}^{\mathrm{VL}} \equiv \varnothing$ and $\mathrm{DS}^{\mathrm{TS}} \bigcap \mathrm{DS}^{\mathrm{VL}} \equiv \varnothing$.

$\Sigma\Pi$-neural networks were constructed based on the condition of minimal functional sufficiency, i.e., by specifying a minimum number of layers and neurons required to functionally implement the modeled relationship. We also selectively considered weaker and/or, conversely, overparameterized networks.

The solutions for $\Sigma\Pi$-neural networks were compared with similar classical MLP[7] fully connected architectures (hereinafter referred to as FC-neural networks) of commensurate complexity (in terms of the number of layers and trainable parameters). All neural

[6] The *PyTorch* extension was written by R.Yu. Portsev in collaboration with V.V. Kazakov, a researcher at Laboratory No. 77 of the Trapeznikov Institute of Control Sciences, the Russian Academy of Sciences.

[7] *Multilayer Perceptron* [3].

networks under study were trained using *PyTorch 2.1.0*. For training, the loss function was defined as the mean squared error (MSE):

$$L_{\text{MSE}} = \frac{1}{M}\sum_{i=1}^{M}\left(z_i - \hat{z}_i\right)^2,$$

where M is the number of samples in the dataset; z_i is the observed value of the i th sample; $\hat{z}_i$ is the predicted value of the ith sample. Each architecture was initialized with at least five different sets of weights, and each set was trained using the *RMSprop* optimizer for 200 epochs with a mini-batch size of 32 samples. (The mini-batch was formed in a special way depending on the epoch number.)

Next, the best (top-1) networks were selected for each function and architecture according to the rule

$$L_{\text{MSE}} \,|\, \text{DS}^{\text{TS}} \to \min$$

in the "initialization" × "training epoch" space.

To correctly compare the properties of neural networks across the domains DS^{VL} and $\overline{\text{DS}}^{\text{VL}}$ (including cross-domain analysis), the *Absolute Percentage Error* (APE) was used:

$$R_{\text{APE}} = \left|\frac{z_i - \hat{z}_i}{z_i}\right|. \qquad (16)$$

For this metric, statistical quantiles of the following orders were determined within each network and each validation dataset:

$$\alpha = \overline{0.25;\ 0.5;\ 0.75;\ 0.9;\ 0.99}. \qquad (17)$$

Other characteristics were also analyzed for each experiment when necessary.

2.2. Simple Polynomials in Two Variables

The polynomial is arguably the central object of study in classical algebra. Such objects play a crucial role in mathematical modeling as well. This is because any continuous function on an interval can be replaced by a polynomial with an arbitrarily small error (Theorem 3); hence, it is possible to approximate broad classes of functions by polynomials.

In this subsection, we investigate the capabilities of modeling simple polynomials in two variables by $\Sigma\Pi$-neural networks and classical fully connected networks (FC-neural networks). Five polynomials were selected for the experiments:

$$f_1 = -x^2 y^2, \qquad (18)$$

$$f_2 = (3x - y)^2, \qquad (19)$$

$$f_3 = x^2 - y^2, \qquad (20)$$

$$f_4 = x^2 y + x y^2, \qquad (21)$$

$$f_5 = x^2 + y^2. \qquad (22)$$

These expressions were chosen as the object of modeling due to the following considerations. On the one hand, they are relatively simple to interpret the results; on the other hand, they contain sufficiently nontrivial relationships (the presence of multiplication and exponentiation, and factorability/non-factorability).

Table 1 shows the configurations of the neural networks used to model the expressions (18)–(22). In all FC-neural networks, the ReLU activation function was applied between layers.

Table 1

Neural network configurations for polynomial modeling

Polynomial	$\Sigma\Pi$- neural network		FC-neural network		Experiment
	Architecture: layers / neurons	The number of trainable parameters	Architecture Layers / Neurons	The number of trainable parameters	
$f_1 = -x^2 y^2$	2 / 2+1	21	2 / 5+1	21	E1
$f_2 = (3x - y)^2$	2 / 2+1	21	2 / 5+1	21	E2
$f_3 = x^2 - y^2$	2 / 2+1	21	2 / 5+1	21	E3
$f_4 = x^2 y + x y^2$	2 / 2+1	21	2 / 5+1	21	E4
$f_5 = x^2 + y^2$	2 / 2+1	21 ↓	2 / 5+1	21	E5.1
$f_5 = x^2 + y^2$	3 / 3+2+1	46	3 / 6+4+1	51	E5.2
$f_5 = x^2 + y^2$	3 / 4+2+1	57 ↑	3 / 7+4+1	58	E5.3

Note that the polynomial f_5 was modeled by three different neural network configurations; see experiments E5.1 and E5.2 in Table 1. In the first case, we used a two-layer network (by definition, it does not implement the function f_5); in the second, a three-layer network with the minimum number of layers sufficient to implement the modeled relationship; and in the third, a three-layer network with an extra neuron in the first layer.

For the selected top-1 neural networks, the statistical quantiles (17) based on the metric (16) for the datasets $\mathrm{DS}^{\mathrm{VL}}$ and $\overline{\mathrm{DS}}^{\mathrm{VL}}$ are presented in Fig. 5.

According to the data in Fig. 5, the $\Sigma\Pi$-neural networks very accurately approximated the functions $f_1,\ldots, f_4$ in experiments E1,…, E4, respectively. This is indicated both by low errors (a median less than 0.04) and by almost identical values of the median error on the datasets $\mathrm{DS}^{\mathrm{VL}}$ and $\overline{\mathrm{DS}}^{\mathrm{VL}}$ within a single experiment. Recall that the extended validation dataset $\overline{\mathrm{DS}}^{\mathrm{VL}}$ exceeds the training domain by 50 times in terms of amplitude and by 2500 times in terms of area (two independent variables). In turn, on the dataset $\overline{\mathrm{DS}}^{\mathrm{VL}}$, the median errors of the FC-neural networks significantly exceed those of the $\Sigma\Pi$-neural ones: by over 50 times for the function f_1 and by over 100 times for the functions $f_2,\ldots, f_4$. Even within the training domain, the FC-neural networks exhibit error values that are an order of magnitude worse than those of the $\Sigma\Pi$-neural networks, and outside of it, they degrade completely. The experimental results for the function f_5 are more complex. Within the training domain, even the functionally insufficient $\Sigma\Pi$-neural network (experiment E5.1) demonstrates quite acceptable performance and an error of 2.5 times lower than that of its equivalent classical FC-neural network. The more complex networks (experiments E5.2 and E5.3) naturally yield better results, and the gap with FC-neural networks increases. However, only the overparameterized $\Sigma\Pi$-neural network (experiment E5.3) performs well outside the training domain, while its equivalent FC-neural network completely fails. The minimally sufficient $\Sigma\Pi$-neural network (experiment E5.2) operates improperly outside the training domain most likely because of its undertraining.

Clearly, due to architectural limitations, FC-neural networks are incapable of directly modeling the polynomials (18)–(22) and, in the series of experiments, were used in the full memoization mode. Therefore, it is of interest to investigate the actual mechanism by which $\Sigma\Pi$-neural networks implement the expressions (18)–(22). To do this, we consider the values of the trainable parameters of the trained $\Sigma\Pi$-neural networks. Let the coefficients of the polynomials (18)–(22) be compactly written as a square matrix of dimensions 3×3:

$$\mathcal{M}(f)=\begin{pmatrix} k^{(0,0)} & \cdots & k^{(0,2)} \\ \vdots & \ddots & \vdots \\ k^{(2,0)} & \cdots & k^{(2,2)} \end{pmatrix},$$

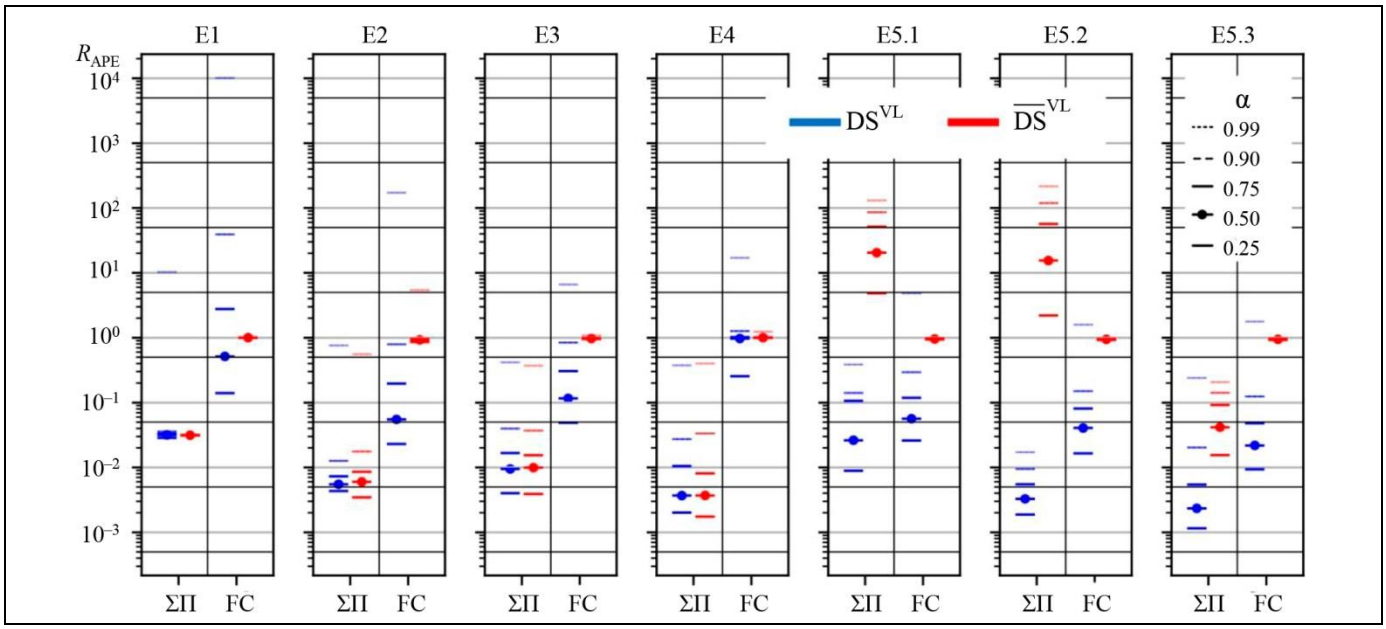

Fig. 5. The APE values of trained neural networks for the functions $f_1,\ldots, f_5$ in experiments E1,…, E5.3.

where the element $k^{(n,m)}$ corresponds to the coefficient at $x^n y^m$. After elementary transformations of the weights, for the functions under consideration and the trained $\Sigma\Pi$-neural networks, the coefficient matrices will be as follows.

According to Table 2, in all experiments except E5.1, the $\Sigma\Pi$-neural networks reproduce the theoretically derived polynomial coefficients quite well (in boldface) and have a few spurious coefficients of noticeable magnitude (values greater than 0.01 by magnitude, in boxes). This is evidence that the $\Sigma\Pi$-neural networks operate in a mode of direct modeling of the corresponding functional relationships. In the case of experiment E5.1, however, the functionally insufficient network architecture (Table 1) is trained in a memoization mode, actually implementing a function "close" to the theoretical one (22) within the training domain. Note that it is theoretically impossible to eliminate spurious coefficients for the architecture in experiment E5.1: the two-layer $\Sigma\Pi$-neural network cannot express the function (22) in its pure form.

In turn, direct comparison of the results of experiments E5.1 and E5.2 demonstrates the greater expressive capability of the $\Sigma\Pi$-neural networks of higher depth. (This property also holds for classical fully connected architectures.)

Table 2

Coefficient matrices for modeled polynomials and Sigma-Pi neural networks

Experiment	Polynomial	$\mathcal{M}\mid f$	$\mathcal{M}\mid$ $\Sigma\Pi$-neural network
E1	$f_1 = -x^2 y^2$	$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} 4.159\times10^{-4} & 1.394\times10^{-5} & -1.06\times10^{-7} \\ -6.988\times10^{-5} & -2.311\times10^{-3} & -6.646\times10^{-4} \\ 9.222\times10^{-10} & 3.149\times10^{-4} & \mathbf{-1.031} \end{pmatrix}$
E2	$f_2 = (3x-y)^2$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & -6 & 0 \\ 9 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 6.268\times10^{-3} & 3.737\times10^{-3} & \mathbf{0.9962} \\ \boxed{1.252\times10^{-2}} & \mathbf{-5.977} & -1.758\times10^{-4} \\ \mathbf{8.946} & 5.262\times10^{-4} & 7.736\times10^{-9} \end{pmatrix}$
E3	$f_3 = x^2 - y^2$	$\begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} -8.082\times10^{-6} & 2.078\times10^{-3} & \mathbf{-1.011} \\ 6.149\times10^{-4} & 5.871\times10^{-3} & -1.702\times10^{-5} \\ \mathbf{1.003} & 3.629\times10^{-5} & 2.557\times10^{-10} \end{pmatrix}$
E4	$f_4 = x^2 y + x y^2$	$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 2.241\times10^{-4} & 5.109\times10^{-4} & -1.856\times10^{-4} \\ 5.085\times10^{-4} & 4.455\times10^{-3} & \mathbf{0.9985} \\ -6.544\times10^{-4} & \mathbf{0.9988} & 1.177\times10^{-4} \end{pmatrix}$
E5.1	$f_5 = x^2 + y^2$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} \boxed{-2.213\times10^{-2}} & \boxed{-7.81\times10^{-2}} & \mathbf{1.058} \\ \boxed{-7.965\times10^{-2}} & \boxed{7.107\times10^{-2}} & -3.908\times10^{-3} \\ \mathbf{1.061} & -3.95\times10^{-3} & \boxed{-2.526\times10^{-2}} \end{pmatrix}$
E5.2	$f_5 = x^2 + y^2$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} -7.277\times10^{-4} & 3.595\times10^{-4} & \mathbf{0.9991} \\ 2.591\times10^{-3} & -6.276\times10^{-3} & 3.717\times10^{-3} \\ \mathbf{0.9956} & 3.969\times10^{-3} & -3.412\times10^{-6} \end{pmatrix}$
E5.3	$f_5 = x^2 + y^2$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} \boxed{1.562\times10^{-2}} & -7.9\times10^{-4} & \mathbf{0.9965} \\ -1.295\times10^{-4} & -1.577\times10^{-3} & -4.803\times10^{-4} \\ \mathbf{0.9938} & -2.412\times10^{-4} & -2.721\times10^{-5} \end{pmatrix}$

2.3. Scalar Multiplication

In addition to polynomials, operations involving vectors and matrices are significant in the practice and theory of mathematical modeling. In this subsection, we explore the capabilities of $\Sigma\Pi$-neural networks and classical fully connected networks for modeling two operations:

$$\mathbf{v}_1 = \mathbf{x} \cdot \mathbf{y}, \tag{23}$$

$$\mathbf{v}_2 = \mathbf{X}\mathbf{y}, \tag{24}$$

i.e., the scalar multiplication of vectors $\mathbf{x}$ and $\mathbf{y}$, and the left multiplication of a column vector $\mathbf{y}$ by a matrix $\mathbf{X}$, respectively. These expressions are chosen as the object of modeling due to the following considerations. On the one hand, they are relatively simple to interpret the results; on the other hand, these operations are key in vector-matrix algebra and have a wide range of practical applications.

The configurations of the neural networks used to model the expressions (23) and (24) are shown in Table 3. In all fully connected neural networks, the ReLU activation function was used between the layers.

For the selected top-1 neural networks, the statistical quantiles (17) based on the metric (16) for the datasets $\mathrm{DS}^{\mathrm{VL}}$ and $\overline{\mathrm{DS}}^{\mathrm{VL}}$ are presented in Fig. 6.

Figure 6 shows that in all experiments, the $\Sigma\Pi$-neural networks considerably outperform their classical FC-counterparts. The exception is the $\Sigma\Pi$-neural network in experiment E6 on the data outside the training domain. The anomalous increase in errors is most likely due to the network undertraining. Further research is needed for this phenomenon, a similar situation in experiment E5.2, and the training peculiarities of $\Sigma\Pi$-neural networks in general.

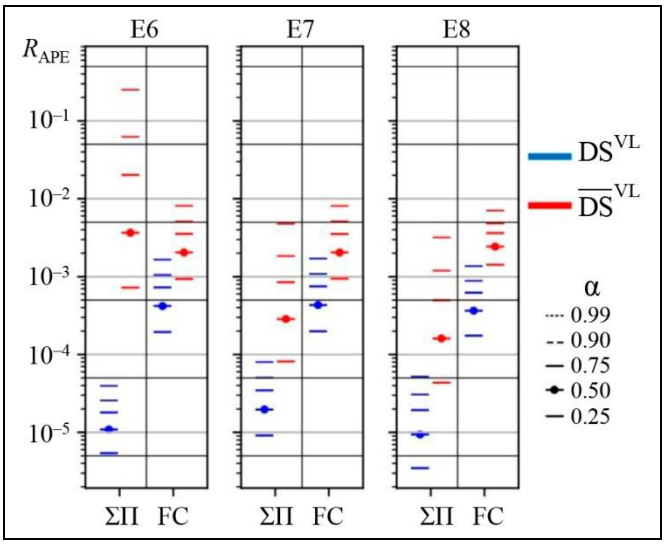

Fig. 6. The APE values of trained neural networks for the functions $\mathbf{v}_1$ and $\mathbf{v}_2$ in experiments E7 and E8.

CONCLUSIONS

In this paper, we have proposed and justified, in a first approximation, a novel design of the $\Sigma\Pi$-neuron for resolving the general "factor multiplication" problem in artificial neural networks. The solution has several undoubted advantages as follows:

- It can be transparently integrated with current and future neural network architectures in both inference and training modes. Moreover, due to the properties of the $\Sigma\Pi$-neuron, it is possible to rearrange the network layers rather flexibly;
- The factor multiplication operation has automatic activation controlled during the training phase. In addition, it is possible to implement a neuroplasticity mode: multiplication is activated for specific input data and/or when the network has a specific hidden state;

Table 3

Neural network configurations for modeling vector-matrix algebra operations

Operation	$\Sigma\Pi$-neural network		FC-neural network		Experiment
	Architecture: layers / neurons	The number of trainable parameters	Architecture layers / neurons	The number of trainable parameters	
$\mathbf{v}_1 = \mathbf{x} \cdot \mathbf{y}$, $\dim \mathbf{x} = 3$	2 / 3+1	54	2 / 7+1	57	E6
$\mathbf{v}_1 = \mathbf{x} \cdot \mathbf{y}$, $\dim \mathbf{x} = 4$	2 / 4+1	87	2 / 9+1	91	E7
$\mathbf{v}_2 = \mathbf{X}\mathbf{y}$, $\dim \mathbf{X} = 2 \times 2$	2 / 4+2	82	2 / 9+2	83	E8

• A single neuron can be used to solve certain non-separable problems. As proved in this work, a single $\Sigma\Pi$-neuron implements the XOR function of an arbitrary dimension.

• Compositional structures of polynomial, vector, and matrix nature can be identified in symbolic data. This aspect has been demonstrated above and is important for AI+Science tasks.

• The quality and stability of the operation of $\Sigma\Pi$-neural networks are consistently better than those of classical fully connected networks. In other words, $\Sigma\Pi$-neural networks produce fewer errors during training and inference than classical multilayer perceptrons, given the same number of parameters. This advantage is fully evident in tasks involving input data outside the training domain.

At the same time, the current implementation of the $\Sigma\Pi$-neuron has a clear drawback: no factor division. Furthermore, due to the mathematical properties (6)–(8), $\Sigma\Pi$-neural networks require some modification of the classical backpropagation algorithm and the development of an appropriate loss function to better realize their potential. All these issues are the subject of the authors' ongoing research.

Thus, the solutions proposed in this paper, on the one hand, increase the "expressiveness" of the neural network, making it more compact; and on the other hand, they improve its interpretability. The novel $\Sigma\Pi$-neuron design is expected to find application both in surrogate neural network models for computation and simulation and in general AI+Science tasks.

Acknowledgments. *The authors are grateful to Academician D.A. Novikov for his permanent attention to this work, to D.S. Gadzhiev for his active discussion of the related issues and valuable remarks, and to V.V. Kazakov for his implementation of the* $\Sigma\Pi$- *neuron in the PyTorch framework.*

REFERENCES

1. McCulloch, W.S. and Pitts, W., A Logical Calculus of the Ideas Immanent in Nervous Activity, *The Bulletin of Mathematical Biophysics*, 1943, vol. 5, pp. 115–133.
2. Rosenblatt, F., The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, *Psychological Review*, 1958, vol. 65, no. 6, pp. 386–408.
3. Makarenko, A.V., Deep Neural Networks: Origins, Development, Current Status, *Control Sciences*, 2020, no. 2, pp. 3–19. (In Russian.)
4. Vaswani, A., Shazeer, N., Parmar, N., et al., Attention Is All You Need, *arXiv: 1706.03762*, 2017. DOI: https://doi.org/10.48550/arXiv.1706.03762
5. Dehghani, M., Gouws, S., Vinyals, O., et al., Universal Transformers, *arXiv: 1807.03819*, 2018. DOI: https://doi.org/10.48550/arXiv.1807.03819
6. Liu, Z., Wang, Y., Vaidya, S., et al., KAN: Kolmogorov-Arnold Networks, *arXiv: 2404.19756*, 2024. DOI: https://doi.org/10.48550/arXiv.2404.19756
7. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Rumelhart, D.E. and McClelland, J.L., Eds., Cambridge, MA: MIT Press, 1986.
8. Cybenko, G., Approximation by Superpositions of a Sigmoidal Function, *Mathematics of Control Signals and Systems*, 1989, vol. 2, no. 4, pp. 303–314.
9. Rosenblatt, F., *Principles of Neurodynamics: Perceptions and the Theory of Brain Mechanisms*, New York: Cornell University, 1961.
10. Nielsen, R., Kolmogorov's Mapping Neural Network Existence Theorem, *Proceedings of the IEEE 1st International Conference on Neural Networks*, San Diego, CA, 1987, vol. 3, pp. 11–13.
11. Hornik, K., Approximation Capabilities of Multilayer Feedforward Networks, *Neural Networks*, 1991, vol. 4, no. 2, pp. 251–257.
12. Hornik, K., Stinchcombe, M., and White, H., Multilayer Feedforward Networks Are Universal Approximators, *Neural Networks*, 1989, vol. 2, no. 5, pp. 359–366.
13. Leshno, M., Lin, V.Ya., Pinkus, A., and Schocken, S., Multilayer Feedforward Networks with a Non-Polynomial Activation Function Can Approximate Any Function, *NYU Working Paper No. IS-92-13*, 1992.
14. Maiorov, V. and Pinkus, A., Lower Bounds for Approximation by MLP Neural Networks, *Neurocomputing*, 1999, vol. 25, no. 1–3, pp. 81–91.
15. Kolmogorov, A.N., On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition, in *American Mathematical Society Translations: Series 2*, 1963, vol. 28, pp. 55–59.
16. Arnold, V.I., On the Representation of Functions of Several Variables as a Superposition of Functions of a Smaller Number of Variables, in *Vladimir I. Arnold – Collected Works: Representations of Functions, Celestial Mechanics, and KAM Theory 1957–1965*, vol. 1, Berlin–Heidelberg: Springer, 2009, pp. 57–77.
17. Arnold, V.I., On Functions of Three Variables, in *American Mathematical Society Translations: Series 2*, 1963, vol. 28, pp. 51–54.
18. Weierstrass, K., Über die analytische Darstellbarkeit sogenannter willkürlicher Functionen einer reellen Veränderlichen, *Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften zu Berlin*, 1885, Bd. II, S. 633–639 (Mitteilung 1), 789–805 (Mitteilung 2). (In German.)
19. Vitushkin, A.G., *O mnogomernykh variatsiyakh* (On Multidimensional Variations), Moscow: Fizmatgiz, 1955. (In Russian.)
20. Dauphin, Y.M., Fan, A., Auli, M., and Grangier, D., Language Modeling with Gated Convolutional Networks, *arXiv: 1612.08083*, 2016. DOI: https://doi.org/10.48550/arXiv.1612.08083
21. Shazeer, N., GLU Variants Improve Transformer, *ArXiv: 2002.05202*, 2020. DOI: https://doi.org/10.48550/arXiv.2002.05202
22. Mnih, A. and Hinton, G., Three New Graphical Models for Statistical Language Modelling, *Proceedings of the 24th International Conference on Machine Learning*, Oregon, 2007, pp. 641–648.

23. Liu, Z., Ma, P., Wang, W., et al., KAN 2.0: Kolmogorov-Arnold Networks Meet Science, *arXiv: 2408.10205*, 2024. DOI: https://doi.org/10.48550/arXiv.2408.10205
24. Hochreiter, S. and Schmidhuber, J., Long Short-Term Memory, *Neural Computation*, 1997, vol. 9, no. 8, pp. 1735–1780.
25. Minsky, M. and Papert, S., *Perceptrons: An Introduction to Computational Geometry*, Cambridge, MA: MIT Press, 1969.

This paper was recommended for publication by RAS Academician D.A. Novikov, a member of the Editorial Board.

Received July 9, 2026, and revised July 10, 2026. Accepted July 10, 2026

Author information

Portsev, Ruslan Yur'evich. Junior Researcher, Trapeznikov Institute of Control Sciences, Russian Academy of Sciences, Moscow, Russia
✉ poruss@mail.ru
ORCID iD: https://orcid.org/0009-0009-4008-8725

Makarenko, Andrey Viktorovich. Cand. Sci. (Eng.), Trapeznikov Institute of Control Sciences, Russian Academy of Sciences, Moscow, Russia
✉ avm.science@mail.ru
ORCID iD: https://orcid.org/0000-0002-3875-7549

Cite this paper

Portsev, R.Yu. and Makarenko, A.V., The Sigma-Pi Neuron and Resolving the General "Factor Multiplication" Problem in Artificial Neural Networks. *Control Sciences* **4,** 12–23 (2026).

Original Russian Text © Portsev, R.Yu., Makarenko, A.V., 2026, published in *Problemy Upravleniya*, 2026, no. 4, pp. 15–27.

Translated into English by *Alexander Yu. Mazurov*,
Cand. Sci. (Phys.–Math.),
Trapeznikov Institute of Control Sciences,
Russian Academy of Sciences, Moscow, Russia
✉ alexander.mazurov08@gmail.com