J Multidisciplinary Scientific Journal

MDPI

Article

Polynomial-Computable Representation of Neural Networks in Semantic Programming

Sergey Goncharov *,† **and Andrey Nechesov** *,†

Sobolev Institute of Mathematics, Academician Koptyug Ave., 4, 630090 Novosibirsk, Russia
* Correspondence: s.s.goncharov@math.nsc.ru (S.G.); nechesov@math.nsc.ru (A.N.)
† These authors contributed equally to this work.

Abstract: A lot of libraries for neural networks are written for Turing-complete programming languages such as Python, C++, PHP, and Java. However, at the moment, there are no suitable libraries implemented for a p-complete logical programming language L. This paper investigates the issues of polynomial-computable representation neural networks for this language, where the basic elements are hereditarily finite list elements, and programs are defined using special terms and formulas of mathematical logic. Such a representation has been shown to exist for multilayer feedforward fully connected neural networks with sigmoidal activation functions. To prove this fact, special p-iterative terms are constructed that simulate the operation of a neural network. This result plays an important role in the application of the p-complete logical programming language L to artificial intelligence algorithms.

Keywords: polynomiality; polynomial algorithm; logical programming language; semantic programming; AI; neural networks; machine learning

check for updates

Citation: Goncharov, S.; Nechesov, A. Polynomial-Computable Representation of Neural Networks in Semantic Programming. *J* **2023**, *6*, 48–57. https://doi.org/10.3390/j6010004

Academic Editor: Christos Bouras

Received: 17 November 2022
Revised: 27 December 2022
Accepted: 4 January 2023
Published: 6 January 2023

1. Introduction

The conception of semantic programming [1] was developed in the 1970s and 1980s. The main objective of this direction was to create a logical programming language in which programs were given by the basic constructions of mathematical logic such as formulas and terms. A hereditarily finite list superstructure $HW(\mathfrak{M})$ of the signature σ was chosen as the virtual execution device. At first, the programs were Σ and Δ_0-formulas [2], but this language was Turing-complete [3].

In 2017, Goncharov proposed considering the programs as the terms and presented the conception of conditional terms [4]. In the new L_1 language, L_1-programs and L_1-formulas were inductively specified through the standard terms and Δ_0-formulas of the signature σ. The polynomial-computable (p-computable) model $HW(\mathfrak{M})$ of the signature σ was chosen as a virtual execution device. The authors of [5] showed that any L_1-program and L_1-formula has a polynomial computational complexity. The question arises as to whether all algorithms of polynomial computational complexity can be represented in this language or in its polynomial-computable extensions. This has been an open problem for several years.

Only in 2021 did Goncharov and Nechesov propose the conception of the p-iterative terms [6]. Extension of the L_1 language with p-iterative terms leads to the language L. In [6], it is shown that the class of all L-programs coincides with the class of all algorithms of a polynomial computational complexity P.

The main area of application of semantic programming is artificial intelligence. In artificial intelligence, the black box problem [7] has been around for a long time. Most often, AI gives a result without explaining it. It does not explicitly share how and why it reaches its conclusions. One of the main directions in AI is machine learning [8]. Most machine learning algorithms implemented using neural networks give a result but do not explain

it. Our work is the first step toward the separation of artificial intelligence algorithms in the stage where a result can be logically explained and the stage where a result is given without explanation.

The main objective of this paper is to build a polynomial-computable representation [9] for multilayer feedforward fully connected neural networks [10,11] using the basic elements of the model $HW(\mathfrak{M})$. This type of neural network plays a great role in many artificial intelligence algorithms [12], and such a library for them in the p-complete logical language L is necessary.

2. Preliminaries

The paper uses the results of the theory of semantic programming [13], which is based on a polynomial-computable hereditarily finite list superstructure $HW(\mathfrak{M})$ of a finite signature σ. The main set of $HW(\mathfrak{M})$ consists of the hereditarily finite list elements. The signature σ consists of the next constant, operations and predicates:

(1) nil: a constant that selects an empty list;
(2) $head^{(1)}$: returns the last element of the list or is nil otherwise;
(3) $tail^{(1)}$: returns a list without the last element or is nil otherwise;
(4) $getElement^{(2)}$: returns the ith element of the list or is nil otherwise;
(5) $NumElements^{(1)}$: returns the number of elements in the list;
(6) $first^{(1)}$: returns the first element of the list or is nil otherwise;
(7) $second^{(1)}$: returns the second element of the list or is nil otherwise;
(8) $\in^{(2)}$: the predicate "to be an element of a list";
(9) $\subseteq^{(2)}$: the predicate"to be an initial segment of a list".

Let us define the conception of L_0-formulas and L_0-programs in the basic language L_0 as Δ_0-formulas and standard terms of the signature σ, respectively.

The language L_1 is defined as an inductive extention of the basic language L_0 with conditional terms of the following form:

$$Cond(t_1, \varphi_1, \ldots, t_{n+1}) = \begin{cases} t_0, \text{ if } HW(\mathfrak{M}) \models \varphi_0 \\ t_1, \text{ if } HW(\mathfrak{M}) \models \varphi_1 \& \neg\varphi_0 \\ \ldots \\ t_n, \text{ if } HW(\mathfrak{M}) \models \varphi_n \& \neg\varphi_0 \& \ldots \& \neg\varphi_{n-1} \\ t_{n+1}, \text{ otherwise} \end{cases} \quad (1)$$

The conceptions of the L-formula, L-program and p-iterative term are defined as follows.

Basis of induction: Any L_1-program is an L-program, and any L_1-formula is an L-formula.

Induction step: Let $g(x)$ be an L-program and $\varphi(x)$ be an L-formula, where there is a constant C_g such that for any w, the following inequality is true:

$$|g(w)| \leq |w| + C_g \quad (2)$$

The notation $g^i(x)$ means the L-program g is applied i times:

$$g^i(x) = g(g^{i-1}(x)) \text{ where } g^0(x) = x$$

Suppose that there is a constant C_g such that for any $w \in HW(M)$, the following is true:

$$|g(w)| \leq |w| + C_g$$

The notation of the p-iterative term [6] is defined as follows:

$$Iteration_{g,\varphi}(w,n) = \begin{cases} g^i(w), \text{if } i \leq n\ HW(\mathfrak{M}) \models \varphi(g^i(w)) \\ \quad \text{and } \forall j < i\ HW(\mathfrak{M}) \not\models \varphi(g^j(w)) \\ false, \text{otherwise} \end{cases} \quad (3)$$

The L-program definitions in the inductive step are as follows:

- $Iteration_{g,\varphi}(t_1, t_2)$ is an L-program, where g, t_1, t_2 are L-programs and φ is an L-formula;
- $Cond(t_1, \varphi_1, \ldots, t_n, \varphi_n, t_{n+1})$ is an L-program, where $t_1, \ldots, t_{n+1}$ are L-programs and φ_1, φ_n are L-formulas;
- $F(t_1, \ldots, t_n)$ is an L-program, where $F \in \sigma$ and $t_1, \ldots, t_n$ are L-programs

The L-formula definitions in the inductive step are as follows:

- $t_1 = t_2$ is an L-formula, where t_1, t_2 are L-programs;
- $P(t_1, \ldots, t_n)$ is an L-formula, where $P \in \sigma$ and $t_1, \ldots, t_n$ are L-programs;
- $\Phi\&\Psi$, $\Phi \vee \Psi$, $\Phi \to \Psi$, $\neg\Phi$ are L-formulas, where Φ, Ψ are L-formulas
- $\exists x \delta t \Phi$, $\forall x \delta t \Phi$ are L-formulas, where t is an L-program, Φ is an L-formula and $\delta \in \{\in, \subseteq, \leq\}$.

Theorem 1 ((Solution to the problem P = L) [6]). *Let $HW(\mathfrak{M})$ be a p-computable hereditarily finite list superstructure $HW(\mathfrak{M})$ of the finite signature σ. Then, the following are true:*

(1) Any L-program has polynomial computational complexity.
(2) For any p-computable function, there is a suitable L-program that implements it.

In the current work, we modify the conception of the p-iterative term, and instead of the inequality $|g(x)| \leq |x| + C_g$, we require fulfillment of some conditions for an L-program g and L-formula φ.

Suppose the L-program $g(x)$ for a fixed $n \in N$ and some polynomial $h(x)$ are defined as follows:

$$g(w) = \begin{cases} w^*, \text{if } w =< w_1, \ldots, w_n > \\ false, \text{otherwise} \end{cases} \quad (4)$$

where $w^* =< w_1^*, \ldots, w_n^* >$ and the following conditions are fulfilled (up to a permutation):

(1) $|w_1^*| \leq |w_1| + C \cdot \sum_{i=2}^{n} |w_i|^p$;
(2) $|w_i^*| \leq |w_i|$, for all $i \in [2, \ldots, n]$.

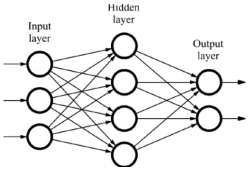
The next lemma almost completely repeats the proof of Theorem 1 from [6]. The same length and complexity estimates are used:

Lemma 1. *The term $Iteration_{g,\varphi}$ from Equation (3) with conditions (1–2) from Equation (4) on g is a p-computable function.*

3. Neural Networks

This work will consider multilayer feedforward fully connected neural networks of the type in [10] (see also [14,15]), which has one incoming layer, one output layer and several hidden layers. For simplicity of presentation, we will assume that there are no bias neurons in such neural networks. However, all results of this work for neural networks with bias neurons are also valid.

For example, such a neural network with one hidden layer has the following form:



By default, for all neurons of the neural network, the activation function will be a sigmoid of the form:

$$Sig(x) = \frac{1}{1+e^{-x}}$$

The derivative of this function has the form:

$$Sig'(x) = Sig(x) \cdot (1 - Sig(x))$$

Consider the approximation [16] for the function e^{-x} as the Taylor series expansion up to nine terms of the form:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots + \frac{(-1)^n x^n}{n!} \tag{5}$$

Denote the function f as the sigmoid approximation, which uses the Taylor series from Equation (5) for e^{-x}. In addition, denote $f'(x) = f(x)(1 - f(x))$ as an approximation for a derivative function $Sig'(x)$:

Remark 1. *f and f' are p-computable functions.*

Let $c(x, y) : N \times N \to N$ be a standard pair numbering function:

$$c(x, y) = \frac{(x + y + 1) \cdot (x + y)}{2} + y, \text{ where } x, y \in N$$

Let us define the notation for the mth neuron in kth layer of the neural network N as $n_{c(k,m)}$. Each neuron has the following list representation:

$$\underline{n_i} : < \underline{f}, < w_{ij_1}, \ldots, w_{ij_k} >>$$

where $\underline{f}$ is a constant symbol in a signature σ for an activation function and w_{ij} are the weights of the synapses which occur from neuron n_i to neuron n_j.

Neuron n_i of the output layer has the form:

$$\underline{n_i} : < \underline{f}, <>>$$

Let us define the p-computable predicate *Neuron* which selects the list encodings of the neurons. A characteristic function for this predicate is defined as follows:

$$Cond(1, \Phi(x), 0)$$

where *Cond* is a condition term from Equation (1) and $\Phi(x)$ has the following form:

$$\begin{aligned} \Phi(x) : \ & (first(x) = \underline{f}) \& (NumElements(x) = 2) \& \\ & \&((second(x) = nil) \vee (\forall l \in second(x) Number(l)) \end{aligned} \tag{6}$$

For some layer with a number i for our neural network, we can define the following code:

$$\underline{s_i} : < \underline{n_{c(i,1)}}, \ldots, \underline{n_{c(i,n_i)}} > \tag{7}$$

Let us define the p-computable predicate *Layer* which selects a layer. A characteristic function for this predicate is defined as follows:

$$Cond(1, \Phi(x), 0)$$

where $\Phi(x)$ has the form:

$$\Phi(x) : \ \forall l \in x \ Neuron(l) \tag{8}$$

Then, the following code is the list encoding for neural network N_i:

$$\underline{N_i} : < \underline{s_1}, \ldots, \underline{s_k} > \tag{9}$$

Let us define the p-computable predicate *NeuralNetwork* which selects a list encoding of the neural network. A characteristic function for this predicate is defined as follows:

$$Cond(1, \Phi(x), 0);$$

where $\Phi(x)$ has the form:

$$\begin{array}{l} \Phi(x) : \ \forall l \in x \ Layer(l) \& (\forall i \leq NumElements(x) - 1 \ \forall w \in getElement(x, i) \\ NumElements(second(w)) = NumElements(getElement(x, i+1))) \& \\ \& (\forall w \in head(x) \ second(w) = nil) \end{array} \tag{10}$$

Remark 2. *Any neural network N is uniquely restored from the list encoding* $\underline{N}$.

Consider the p-computable hereditarily finite list superstructure $HW(\mathfrak{M})$ of the signature

$$\sigma^* = \sigma \cup \{NeuralNetwork^{(1)}, Layer^{(1)}, Neuron^{(1)}\}$$

where neural networks are encoded as elements in $HW(M)$.

Let a be the number. Then, we define a function $\times$ as follows:

$$a \times l = \begin{cases} < a \cdot b_1, \ldots, a \cdot b_k >, \text{ where } l =< b_1, \ldots, b_k > \text{ and all } b_i \text{ are numbers} \\ false, \text{otherwise} \end{cases} \tag{11}$$

Remark 3. $\times$ *is a p-computable function.*

Let us define the function $\uplus$ as follows:

$$l \uplus w = \begin{cases} < a_1 + b_1, \ldots, a_k + b_k >, \text{ if } l =< a_1, \ldots, a_k >, w =< b_1, \ldots, b_k > \text{ and all } a_i, b_i \text{ are numbers} \\ false, \text{otherwise} \end{cases} \tag{12}$$

Remark 4. $\uplus$ *is a p-computable function.*

Let us define the operation $\otimes$ as follows:

$$< a_1, \ldots, a_{m_i} > \otimes \underline{s_i} = \begin{cases} < b_1, \ldots, b_{m_{i+1}} >, \text{ if } s_i \text{ is non-output layer} \\ < c_1, \ldots, c_{m_i} >, \text{ if } s_i \text{ is output layer} \\ false, \text{otherwise} \end{cases} \tag{13}$$

where

$$b_m = \sum_{j=1}^{m_i} f(a_j) \cdot GetElement(second(\underline{n_{c(i,j)}}), m), \ m \in [1, \ldots, m_{i+1}]$$

and where

$$\underline{n_{c(i,j)}} : GetElement(\underline{s_i}, j)$$

and

$$c_m = f(a_m), \ m \in [1, \ldots, m_i]$$

Lemma 2. $\otimes$ *is a p-computable function.*

Proof. We prove this lemma using the construction of the p-iterative term $Iteration_{g,\varphi}$. Let us define the operation of the L-program g for the non-output layer s_i as follows:

$$g(<< a_1,\ldots,a_{m_i} >, \underline{s_i}, <>>) =$$
$$=<< a_1,\ldots,a_{m_i-1} >, tail(\underline{s_i}), < b_1^{(1)},\ldots,b_{m_{i+1}}^{(1)} >>$$

where

$$< b_1^{(1)},\ldots,b_{m_{i+1}}^{(1)} >= f(a_{m_i}) \times second(head(\underline{s_i}))$$

and on the jth step, we have

$$g(<< a_1,\ldots,a_{m_i-j} >, tail^j(\underline{s_i}), < b_1^{(j)},\ldots,b_{m_{i+1}}^{(j)} >>) =$$
$$=<< a_1,\ldots,a_{m_i-j-1} >, tail^{j+1}(\underline{s_i}), < b_1^{(j+1)},\ldots,b_{m_{i+1}}^{(j+1)} >>$$

where the notation $tail^j$ means that the list function $tail$ is applied j times:

$$< b_1^{(j+1)},\ldots,b_{m_{i+1}}^{(j+1)} >=< b_1^{(j)},\ldots,b_{m_{i+1}}^{(j)} > \uplus f(a_j) \times second(head(tail^j(\underline{s_i})))$$

Using Remarks 3 and 4, we find that g is a p-computable function.
The L-formula φ is defined as follows:

$$\varphi : \ second(x) = nil$$

Conditions (1–2) from Equation (4) for the p-iterative term $Iteration_{g,\varphi}$ are met, and therefore, by Lemma 1 the operation $\otimes$ is a p-computable function where s_i is non-output layer.

If s_k is an output layer, then

$$g(<< a_1,\ldots,a_{m_k} >, \underline{s_k} >) =< f(a_1),\ldots,f(a_{m_k}) >$$

□

Let us define the operation W as follows:

$$W(< a_1,\ldots,a_{m_1} >, \underline{N}) =< o_1,\ldots,o_{m_k} >$$

where $\underline{N} =< \underline{s_1},\ldots,\underline{s_k} >$, a_i is the incoming signals and o_i is the outputs of the output layer neurons:

Lemma 3. *W is a p-computable function.*

Proof. We prove this fact by construction of a p-iterative term $Iteration_{g,\varphi}$. The function g is defined as follows:

$$g(<< a_1,\ldots,a_{m_i} >, < \underline{s_i},\ldots,\underline{s_k} >>) =<< a_1,\ldots,a_{m_i} > \otimes \underline{s_i}, < \underline{s_{i+1}},\ldots,\underline{s_k} >>$$

The construction g implies that g is a p-computable function.
The L-formula φ is defined as $head(x) = nil$.
Conditions (1–2) from Equation (4) for the p-iterative terms $Iteration_{g,\varphi}$ are met, and therefore, by Lemma 1, the function W is a p-computable function. □

Let us define the function O as follows:

$$O(\underline{a}, < \underline{s_1},\ldots,\underline{s_k} >) =< \underline{so_1},\ldots,\underline{so_k} >$$

where $\underline{so_i} =< o_{c(i,1)},\ldots,o_{c(i,m_i)} >$ represents the neuron outputs for the ith layer:

Lemma 4. *O is a p-computable function.*

Proof. The proof almost repeats the proof of Lemma 3 □

The backpropagation [17,18] algorithm will be used to configure the neural network. First, it is necessary to find the coefficients as follows:

$$\delta_j = \begin{cases} (o_j - t_j) \cdot o_j \cdot (1 - o_j), \text{ if j is an output neuron} \\ (\sum_{l \in L} w_{jl} \delta_l) \cdot o_j \cdot (1 - o_j), \text{ if j is an inner neuron} \end{cases}$$

where t_j is a target output for the jth neuron of the output layer.

The corrective weights are defined by the formula:

$$\Delta w_{ij} = -\eta \cdot o_i \cdot \delta_j$$

where η is a learning rate (some fixed constant ordinary from $[0, 1]$).

Let $\underline{a} =< a_1, \ldots, a_{m_1} >$ be an input and $\underline{so_i}$ be the neuron outputs for the ith layer.

Let us define the function Δ as follows:

$$\Delta(\underline{a}, \underline{t}, < \underline{s_1}, \ldots, \underline{s_k} >) =< \underline{s\delta_1}, \ldots, \underline{s\delta_k} >$$

where $\underline{s\delta_i} =< \delta_{c(i,1)}, \ldots, \delta_{c(i,m_i)} >$:

Lemma 5. *Δ is a p-computable function.*

Proof. We build the p-iterative term $Iteration_{g,\varphi}$, which simulates the operation of a function Δ as follows:

$$\begin{gathered} g(<<>, < \underline{so_1}, \ldots, \underline{so_k} >, < \underline{s_1}, \ldots, \underline{s_k} >, \underline{t} >) = \\ =<< \underline{s\delta_k} >, < \underline{so_1}, \ldots, \underline{so_k} >, < \underline{s_1}, \ldots, \underline{s_{k-1}} >, <>> \end{gathered}$$

where

$$\delta_{c(k,j)} = (o_{c(k,j)} - t_{c(k,j)}) o_{c(k,j)} (1 - o_{c(k,j)}),\ j \in [1, \ldots, m_k]$$

$$\begin{gathered} g(<< \underline{s\delta_{j+1}}, \ldots, \underline{s\delta_k} >, < \underline{so_1}, \ldots, \underline{so_{j+1}} >, < \underline{s_1}, \ldots, \underline{s_j} >, <>>) = \\ =<< \underline{s\delta_j}, \ldots, \underline{s\delta_k} >, < \underline{so_1}, \ldots, \underline{so_j} >, < \underline{s_1}, \ldots, \underline{s_{j-1}} >, <>> \end{gathered}$$

where

$$\delta_{c(j,i)} = (\sum_{l \in L} w_{c(j,i)c(j+1,l)} \delta_{c(j+1,l)}) o_{c(j,i)} (1 - o_{c(j,i)})$$

The L-formula φ is defined as follows:

$$\varphi :\ head(tail(x)) = nil$$

Conditions (1–2) from Equation (4) for the p-iterative term $Iteration_{g,\varphi}$ are met, and therefore, by Lemma 1 the operation Δ is a p-computable function. □

When all the parameters for weight correction are found, the weights can be changed using the formula:

$$w_{ij}^* = w_{ij} + \Delta w_{ij} = w_{ij} - \eta \cdot o_i \cdot \delta_j$$

Let us define the function T as follows:

$$T(<< \underline{s_1}, \ldots, \underline{s_k} >, < \underline{so_1}, \ldots, \underline{so_k} >, < \underline{s\delta_1}, \ldots, \underline{s\delta_k} >>) =< \underline{s_1}^*, \ldots, \underline{s_k}^* >$$

where the weights of the neurons $\underline{n_i^*} \in \underline{s_m}^*$ are derived from the weights of the neurons $\underline{n_i} \in \underline{s_m}$ by adding Δw_{ij}, where $j \in [\overline{1, \ldots, m_{i+1}}]$:

Lemma 6. *T is a p-computable function.*

Proof. The proof of this lemma is achieved by constructing a suitable p-iterative term $Iteration_{g,\varphi}$ as well as Lemma 5. □

The following theorem statement follows automatically from Remarks 3 and 4 and Lemmas 2–6:

Theorem 2. *Let $HW(\mathfrak{M})$ of the signature σ be a p-computable model. Then, $HW(\mathfrak{M})$ of the signature $\sigma \cup \{NeuralNetwork, Layer, Neuron\} \cup \{\times, \uplus, \otimes, W, O, \Delta, T\}$ is a p-computable model.*

4. Materials and Methods

This paper used the main tools and methods of semantic programming such as p-iteration terms, conditional terms and a p-computable hereditarily finite list superstructure $HW(\mathfrak{M})$ of the signature σ. These techniques allowed us to encode the elements of the multilayer feedforward fully connected neural networks with a sigmoidal activation function using hereditarily finite lists so that the p-iterative term satisfied conditions (1–2) from Equation (4) simulated the operation of the neural network itself. Moreover, the construction of the p-iterative term guarantees polynomial computational complexity of the neural network operation, as well as the backpropagation algorithm.

5. Results

The main result of this work is the construction of a polynomial-computable representation for a multilayer feedforward fully connected neural network with the sigmoidal activation function for a p-complete logical programming language L. This result is equivalent to the statement of the Theorem 2 and allows the introduction and use of neural networks as a basic library of L.

6. Discussion

A polynomial-computable representation of neural networks in the p-complete logical programming language L is extremely important in practice. This allows artificial intelligence algorithms to be implemented without the problems that arise in Turing-complete languages. One such problem is the halting problem.

An open question is the existence of polynomial-computable representations for other types of neural networks, such as recurrent neural networks [19], and modular neural networks [20]. Most often, the answer to the existence of such representations for neural networks in a general case will be negative. Is it possible to find such limitations for such a representation should exist?

Moreover, we are interested not only in the existence of such representations but also constructive implementations within the framework of the theory of semantic programming and p-complete language L.

7. Conclusions

This paper shows a constructive method for polynomial-computable representation of neural networks in the p-complete logical programming language L. Now, we can add a library for neural networks to the p-complete language L, which will allow us to implement algorithms both using a logical language and neural networks. Neural networks play a great role in any direction of AI, so this p-computable representation will help expand the expressive power of our core language L.

Since accuracy, speed and a logical explanation of the result are important criteria for the implementation of artificial intelligence algorithms, the given p-complete logical programming language L solves artificial intelligence problems much better than a similar implementation in Turing-complete languages. Moreover, in this language, there is no halting problem, which is very important for the stability and reliability of software solutions.

The main applications of this result are blockchains, smart contracts, robotics and artificial intelligence. Moreover, these results will be applied in the conception of building smart

cities [21], where almost all possible options for artificial intelligence are involved: image recognition, data mining, machine learning, neural networks, smart contracts, robotics, finance based on cryptocurrencies and blockchains.

Author Contributions: Conceptualization, S.G. and A.N.; methodology, S.G. and A.N.; formal analysis, S.G.; validation, S.G.; investigation, S.G. and A.N.; writing—original draft preparation, A.N.; writing—review and editing, A.N.; supervision, S.G.; project administration, S.G.; software, A.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work was performed within the state task of the Sobolev Institute of Mathematics (Project No. FWNF-2022-0011).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ershov Y.L.; Goncharov S.S.; Sviridenko D.I. Semantic programming. *Inf. Process.* **1986**, *86*, 1113–1120
2. Ershov Y.L. *Definability and Computability*; Springer: New York, NY, USA, 1996.
3. Michaelson, G. Programming Paradigms, Turing Completeness and Computational Thinking.*Art Sci. Eng. Program.* **2020**, *4(4)*. https://doi.org/10.22152/programming-journal.org/2020/4/4.
4. Goncharov, S. Conditional terms in semantic programming. *Sib. Math. J.* **2017** *58*, 794–800. https://doi.org/10.1134/S0037446617050068.
5. Ospichev, S.; Ponomarev, D. On the complexity of formulas in semantic programming. *Semr* **2018**. *15*, 987–995. https://doi.org/10.17377/semi.2018.15.083.
6. Goncharov, S.S.; Nechesov, A.V. Solution of the Problem P = L. *Mathematics* **2022**, *10, 113*. https://doi.org/10.3390/math10010113.
7. Bathaee Y. The artificial intelligence black box and the failure of intent and causation. *Harv. J. Law Technol.*. **2018**. *31*, pp.889-938.
8. Sarker, I.H. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Comput. Sci.* **2021**, *2 (160)*. https://doi.org/10.1007/s42979-021-00592-x.
9. Nechesov, A.V. Some Questions on Polynomially Computable Representations for Generating Grammars and Backus–Naur Forms. *Sib. Adv. Math.* **2022**, *32*, 299–309. https://doi.org/10.1134/S1055134422040058.
10. Leshno, M.; Lin, V.; Pinkuss, A.; Schocken, S. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Netw.* **1993**, *6*, 861–867. https://doi.org/10.1016/S0893-6080(05)80131-5.
11. Hagan, M.T; Demuth, H.B.; Beale, M.H; Jesus, O.D. *Neural Network Design*; Martin Hagan: USA, 2014.
12. Russel, S.J; Norvig, P. *Artificial Intelligence—A Modern Approach*, 3rd ed.; Prentice Hall, USA, 2010.
13. Goncharov, S.; Nechesov, A. Polynomial Analogue of Gandy's Fixed Point Theorem. *Mathematics* **2021**, *9*, 2102. https://doi.org/10.3390/math9172102.
14. Ahamed, I.; Akthar, S. A Study on Neural Network Architectures *Comput. Eng. Intell. Syst.* **2016**, *7*, pp.1–7
15. Wilamowski, B. Neural network architectures and learning algorithms. *IEEE Ind. Electron. Mag.* **2009**, *3*, 56–63. https://doi.org/10.1109/MIE.2009.934790.
16. Temurtas, F.; Gulbag, A.; Yumusak, N. A Study on Neural Networks Using Taylor Series Expansion of Sigmoid Activation Function; *Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany*, 2004; Volume 3046. https://doi.org/10.1007/978-3-540-24768-5_41.
17. Rumelhurt, D.; Hinton, G.; Williams, R. Learning representation by back-propagating errors. *Nature* **1986**, *323*, pp.533–536.
18. Backpropagation. Available online (accessed 10 November 2022): https://en.wikipedia.org/wiki/Backpropagation
19. Schmidt, R. Recurrent Neural Networks (RNNs): A gentle Introduction and Overview. *arXiv* **2019**, arXiv.1912.05911. https://doi.org/10.48550/arXiv.1912.05911.
20. Auda, G.; Kamel, M.; Raafat, H. Modular neural network architectures for classification. In Proceedings of International Conference on Neural Networks, Washington, DC, USA, 1996; Volume 2, pp.1279–1284. https://doi.org/10.1109/ICNN.1996.549082.
21. Nechesov, A.V.; Safarov, R.A. Web 3.0 and smart cities. In Proceedings of the International Conference "Current State and Development Perspectives of Digital Technologies and Artificial Intelligence", Samarkand, Uzbekistan, 2022.

Short Biography of Authors

Sergey Goncharov is a Russian mathematician and organizer of science, academician of the Russian Academy of Sciences (2016), professor, and a specialist of mathematical logic and the theory of computability. Fields of research interest: AI, machine learning, theory of algorithms, polynomials and their applications in approximation theory, model theory, algebra, and applications in theoretical computer science.

Andrey Nechesov is a Russian mathematician and blockchain expert. Principal fields of research: AI, XAI, IoT, DeFi, machine learning, smart contracts, theory of blockchain, theory of cryptocurrencies, theory of algorithms, polynomials and their applications in approximation theory, applications in theoretical computer science.