mathematics

MDPI

Article

On the Computability of Primitive Recursive Functions by Feedforward Artificial Neural Networks

Vladimir A. Kulyukin

Department of Computer Science, Utah State University, Logan, UT 84322, USA; vladimir.kulyukin@usu.edu

Abstract: We show that, for a primitive recursive function $h(\mathbf{x}, t)$, where $\mathbf{x}$ is a n-tuple of natural numbers and t is a natural number, there exists a feedforward artificial neural network $\mathfrak{N}(\mathbf{x}, t)$, such that for any n-tuple of natural numbers $\mathbf{z}$ and a positive natural number m, the first $m + 1$ terms of the sequence $\{h(\mathbf{z}, t)\}$ are the same as the terms of the tuple $(\mathfrak{N}(\mathbf{z}, 0), \ldots, \mathfrak{N}(\mathbf{z}, m))$.

Keywords: computability theory; theory of recursive functions; artificial neural networks

MSC: 03D75; 03D80

1. Introduction

Primitive recursive functions describe, albeit incompletely, the intuitive notion of a number-theoretic algorithm, a deterministic procedure to transform numerical inputs to numerical outputs in finitely many steps. This perception of primitive recursive functions as intuitive counterparts of number-theoretic algorithms may be rooted in the fact that any primitive recursive function can be mechanically constructed from a set of initial functions with finitely many applications of simple, well-defined operations of composition and primitive recursion. These functions and some of their properties have been investigated by Gödel [1], Péter [2,3], Kleene [4], Davis [5], and Rogers [6] in their studies of formal systems, foundations of mathematics, and computability theory. Although the confinement of the construction procedure to two operations may at first seem restrictive, many functions on natural numbers ordinarily encountered in mathematics and computer science are, in fact, primitive recursive (cf., e.g., Ch. 3 in [7]). Primitive recursive functions have been used to investigate the foundations of functional programming. Colson [8] presents a computational model in which a primitive recursive function is viewed as a rewriting system and gives a non-trivial necessary condition for an algorithm to be representable in the system. Paolini et al. [9] define a class of recursive permutations, which they call Reversible Primitive Permutations (RPP), and formalize it as a language that is sufficiently expressive to represent all primitive recursive functions. Petersen [10] uses induction and primitive recursion to develop resource conscious logics where the repeated recycling of assumptions, e.g., repeated applications of the successor function $f(n) = n + 1$ to enumerate natural numbers, has costs.

Feedforward artificial neural networks have their origin in the research by McCulloch and Pitts [11], which describes neural events with propositional logic. McCulloch and Pitts assume that the human nervous system is a finite set of neurons, each of which has an excitation threshold. When a neuron's threshold is exceeded, the neuron generates an impulse that propagates to other neurons across synapses connecting them to the origin of the impulse. A fundamental insight by McCulloch and Pitts is that if the response of a neuron can be formalized as a logical proposition specifying its stimulus, then behaviors of complex networks of neurons can, in principle, be described with symbolic logic. Artificial neural networks entered mainstream computer science almost half a century after the research by McCulloch and Pitts when Rumelhart, Hinton, and Williams [12] discovered *backpropagation*, a method for training networks to modify synapse weights by minimizing

check for updates

Citation: Kulyukin, V.A. On the Computability of Primitive Recursive Functions by Feedforward Artificial Neural Networks. *Mathematics* **2023**, *11*, 4309. https://doi.org/10.3390/math11204309

Academic Editor: Ke-Lin Du

Received: 30 August 2023
Revised: 29 September 2023
Accepted: 9 October 2023
Published: 16 October 2023

error between the output and the ground truth. Different types of such networks have been shown to be universal approximators of some classes of functions (e.g., [13–15]). Artificial neural networks are increasingly used in embedded artificial intelligence (AI) systems, i.e., systems that run on computational devices with finite amounts of computer memory (e.g., [16]). We will refer to embedded AI as *finite AI* to emphasize the fact that finite AI systems are realized on computational devices with finite amounts of computational memory.

In this investigation, we seek to relate, in a formal way, primitive recursive functions and feedforward artificial neural networks by investigating whether it is possible, for a given primitive recursive function, to construct a feedforward artificial neural network that arbitrarily computes many values of the function's co-domain from the corresponding values of the function's domain. We hope that our investigation contributes to the knowledge of the classes of functions that can be not only approximated, but provably computed by feedforward artificial neural networks. In particular, we formalize feedforward artificial neural networks with recurrence equations, propose a formal definition of the concept of $\mathfrak{N}$-computability, i.e., the property of a function to be computed by a feedforward artificial neural network $\mathfrak{N}$, and prove several lemmas and theorems to show how feedforward artificial neural networks can be constructed to arbitrarily compute many consecutive values of any primitive recursive function. Since these networks consist of finite sets of neurons and are used in some finite AI systems [17,18], our investigation will be of interest to mathematicians and computer scientists interested in the computability theory of finite AI.

The remainder of our article is organized as follows. In Section 2, we review several definitions of primitive recursive functions starting with the original definition by Gödel [1] and proceeding to the later definitions by Kleene [4], Davis [5], Rogers [6], and Davis et al. [7], and Meyer and Ritchie [19]. This section gives the reader a historical bird's eye view of how the concept of primitive recursive function and its formalization have co-evolved in time. In Section 3, we state the notational conventions and give the definition of a primitive recursive function we use in this article. This section is intended for reference. In Section 4, we offer a formalization of feedforward artificial neural networks in terms of recursive equations. In Section 5, we prove several lemmas and theorems that form the bulk of our theoretical investigation. In Section 6, we present some perspectives on the obtained results and summarize our conclusions in Section 7.

2. Recursive Functions

Gödel [1] (Sec. 2, p. 157) describes the class of number-theoretic functions as the class of functions whose domains are non-negative integers or n-tuples thereof and whose values are non-negative integers. Gödel [1] (Sec. 2, pp. 157–159) states that a number-theoretic function $\phi(x_1, x_2, \ldots, x_n)$ is *recursively defined in terms of* the number-theoretic functions $\psi(x_1, x_2, \ldots, x_{n-1})$ and $\mu(x_1, x_2, \ldots x_{n+1})$ if ϕ is obtained from ψ and μ by the following schema:

$$\begin{array}{lcl} \phi(0, x_2, \ldots, x_n) & = & \psi(x_2, \ldots, x_n), \\ \phi(k+1, x_2, \ldots, x_n) & = & \mu(k, \phi(k, x_2, \ldots, x_n), x_2, \ldots, x_n), \end{array} \quad (1)$$

where the equalities hold for all $k, x_2, \ldots, x_n$. Gödel [1] (Sec. 2, p. 159) defines a number-theoretic function ϕ to be *recursive* if there exists a finite sequence of number-theoretic functions ϕ_1, ϕ_2, …, $\phi_n = \phi$, where each function ϕ_i, $1 \leq i \leq n$, is a natural number constant, the successor function $x + 1$, or is defined from two preceding functions with Schema (1) or from one preceding function by substitution, i.e., the replacement of the arguments of a preceding function with some other preceding functions.

Kleene [4] (Chap. IX, § 43, p. 219) defines a number-theoretic function to be *primitive recursive* if it is definable by a finite number of applications of the six schemata in (2), where m and n are positive integers, i is an integer such that $1 \leq i \leq n$, q is a natural number, and $\psi, \chi_1, \ldots, \chi_m$, and χ are number-theoretic functions with the indicated numbers of arguments.

$$\begin{array}{ll}
(I) & \phi(x) = x + 1; \\
(II) & \phi(x_1, \ldots, x_n) = q; \\
(III) & \phi(x_1, \ldots, x_n) = x_i; \\
(IV) & \phi(x_1, \ldots, x_n) = \psi(\chi_1(x_1, \ldots, x_n), \ldots, \chi_m(x_1, \ldots, x_n)); \\
(Va) & \begin{cases} \phi(0) & = q, \\ \phi(y+1) & = \chi(y, \phi(y)); \end{cases} \\
(Vb) & \begin{cases} \phi(0, x_2, \ldots, x_n) & = \psi(x_2, \ldots, x_n), \\ \phi(y+1, x_2, \ldots, x_n) & = \chi(y, \phi(y, x_2, \ldots, x_n), x_2, \ldots, x_n). \end{cases}
\end{array} \tag{2}$$

Schema (I) defines the successor function, Schema (II) defines constant functions, and Schema (III) defines identity functions, which Kleene denotes with the symbol U_i^n. Kleene defines the functions satisfying Schemata (I), (II), and (III) in (2) as *initial functions*. Schema (IV) in (2) obtains ϕ from $\psi, \chi_1, \ldots, \chi_m$ by *substitution*. Schemata (Va) and (Vb) obtain ϕ from χ or from χ and ψ, respectively, by *primitive recursion*. Kleene [4] (Chap. XI, § 55, p. 275) defines a function to be *general recursive in* functions $\psi_1, \ldots, \psi_l$ if there is a system E of equations which defines ϕ recursively from $\psi_1, \ldots, \psi_l$.

Davis [5] (Chap. 2, Sec. 2, p. 36) defines the operation of *composition* as the operation to obtain the function $h(\mathfrak{x}^{(n)})$ from the functions $f(\mathfrak{y}^{(m)})$, $g_1(\mathfrak{x}^{(n)}), \ldots, g_m(\mathfrak{x}^{(n)})$ with Schema (3).

$$h(\mathfrak{x}^{(n)}) \quad = \quad f(g_1(\mathfrak{x}^{(n)}), \ldots, g_m(\mathfrak{x}^{(n)})), \tag{3}$$

where $\mathfrak{y}^{(m)}$ and $\mathfrak{x}^{(n)}$ are tuples of natural numbers with m and n elements, respectively. Davis [5] (Chap. 3, Sec. 4, p. 48) defines the operation of primitive recursion as the operation that uses Schema (4) to construct the function $h(\mathfrak{x}^{(n+1)})$ from the total functions $f(\mathfrak{x}^{(n)})$ and $g(\mathfrak{x}^{(n+2)})$, where $\mathfrak{x}^{(n)}$, $\mathfrak{x}^{(n+1)}$, and $\mathfrak{x}^{(n+2)}$ are tuples of natural numbers with n, $n+1$, and $n+2$ elements, respectively.

$$\begin{array}{lcl}
h(0, \mathfrak{x}^{(n)}) & = & f(\mathfrak{x}^{(n)}) \\
h(z+1, \mathfrak{x}^{(n)}) & = & g(z, h(z, \mathfrak{x}^{(n)}), \mathfrak{x}^{(n)}).
\end{array} \tag{4}$$

For a set of natural numbers A, Davis [5] (Chap. 3, Sec. 4, p. 49) defines an *A-primitive recursive* function or a function *primitive recursive in A* as a function that can be obtained by a finite number of applications of composition (cf. Schema (3)) and primitive recursion (cf. Schema (4)) from the following j functions:

$$\begin{array}{llcl}
(1) & C_A(x); & & \\
(2) & S(x) & = & x + 1; \\
(3) & N(x) & = & 0; \\
(4) & U_i^n(x_1, \ldots, x_n) & = & x_i, 1 \leq i \leq n,
\end{array} \tag{5}$$

where $C_A(x)$ is the characteristic function of A (i.e., $C_A(x)$ is a total function such that $C_A(x) = 1$ if $x \in A$ and $C_A(x) = 0$ if $x \notin A$), and $S(x)$ and U_i^n are identical to Kleene's Schemata (I) and (III) in (2). Davis [5] (Chap. 3, Sec. 4, p. 49) defines a function f to be primitive recursive if it is *$\emptyset$-primitive recursive*, where $\emptyset$ denotes the empty set.

Rogers [6] (Chap. 1, § 1.2, p. 6) defines the class C of primitive recursive functions as the smallest class of functions such that

(1) All constant functions $\lambda x_1 x_2 \cdots x_k[m]$, are in C, $1 \leq k$, $0 \leq m$;
(2) The successor function $\lambda x[x+1]$ is in C;
(3) All identity functions $\lambda x_1 \cdots x_k[x_i]$ are in C, $1 \leq i \leq k$;

(4) If f is a function of k variables in C and $g_1, \ldots, g_k$ are functions in C of m variables each, then the function $\lambda x_1 \cdots x_m[f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))]$ is in C, $1 \leq k, m$;
(5) If h is a function of $k+1$ variables in C, and g is a function of $k-1$ variables in C, then the unique function f of k variables satisfying Schema (6) is also in C, $1 \leq k$.

$$\begin{array}{lcl} f(0, x_2, \ldots, x_k) & = & g(x_2, \ldots, x_k), \\ f(y+1, x_2, \ldots, x_k) & = & h(y, f(y, x_2, \ldots, x_k), x_2, \ldots, x_k). \end{array} \tag{6}$$

Davis et al. [7] (Chap. 3, Sec. 3, p. 42) defines as initial the functions $s(x) = x+1$, $n(x) = 0$, and $u_i^n(x_1, \ldots, x_n) = x_i$, $1 \leq i \leq n$, and defines a function to be primitive recursive if it can be obtained from the initial functions by a finite number of applications of composition or primitive recursion where primitive recursion is defined by Schema (7) (Chap. 3, Sec. 2, p. 40 in [7]) and Schema (8) (Chap. 3, Sec. 2, p. 41 in [7]). In Schema (7), k is a natural number and g is a total function of two variables. In Schema (8), f and g are total functions of n and $n+2$ variables, respectively.

$$\begin{array}{lcl} h(0) & = & k, \\ h(t+1) & = & g(t, h(t)) \end{array} \tag{7}$$

$$\begin{array}{lcl} h(x_1, \ldots, x_n, 0) & = & f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n, t+1) & = & g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n). \end{array} \tag{8}$$

2.1. Computability and Turing Machines

Davis [5] (Chap. 1, Sec. 2, p. 10) gives the following definition of partially computable and computable functions.

Definition 1. *An n-ary function $f(x_1, \ldots, x_n)$ is partially computable if there exists a Turing machine Z such that*

$$f(x_1, \ldots, x_n) = \Psi_Z^{(n)}(x_1, \ldots, x_n).$$

In this case, we say that Z computes f. If, in addition, $f(x_1, \ldots, x_n)$ is a total function, then it is called computable.

In subsequent chapters of his monograph (cf. Chap. 2 and Chap. 3 in [5]), Davis separates the notion of computability from Turing machines to make it possible "to demonstrate the computability of quite complex functions without referring back to the original definition of computability in terms of Turing machines" (cf. Ch. 3, Sec. 1, p. 41 in [5]).

Davis et al. [7] (Chap. 2) continue this treatment of computability by designing the programming language $\mathcal{L}$ and then defining partially computable and computable functions in terms of $\mathcal{L}$ programs, viz., finite sequences of $\mathcal{L}$ instructions. In $\mathcal{L}$, the unique variable Y is designated as the output variable to store the output of an $\mathcal{L}$ program $\mathcal{P}$ on a given input. $X_1, X_2, \ldots$ are input variables and $Z_1, Z_2, \ldots$ are internal variables. All variables refer to natural numbers. $\mathcal{L}$ has conditional dispatch instructions, line labels, elementary arithmetic operations, comparisons of natural numbers, and macros.

Davis et al. [7] (Chap. 2, Sec. 3, p. 27) define a *computation* of an $\mathcal{L}$ program $\mathcal{P}$ on some inputs $x_1, \ldots, x_m$, and $m > 0$, as a finite sequence of *snapshots* $(s_1, \ldots, s_k)$, where each snapshot s_i, $1 \leq i \leq k$, $k > 0$ specifies the number of the instruction in $\mathcal{P}$ to be executed and the value of each variable in $\mathcal{P}$, and where each subsequent snapshot is uniquely determined by the previous snapshot (Theorem 3.2, Chap. 4, Sec. 3, pp. 74–75 in [7]). The snapshot s_1 is the *initial* snapshot, where the values of all input variables are set to their initial values, the program instruction counter is set to 1, i.e., the number of the first instruction in $\mathcal{P}$, and the values of all the other variables in $\mathcal{P}$ are set to 0. The snapshot s_k in $(s_1, \ldots, s_k)$ is a *terminal* snapshot, where the instruction counter is set to the number of the instructions in $\mathcal{P}$ plus 1. If some program $\mathcal{P}$ in $\mathcal{L}$ takes m inputs $X_1 = x_1$, $X_2 = x_2, \ldots$, $X_m = x_m$, then

$$\Psi_{\mathcal{P}}^{(m)}(x_1, x_2, \ldots, x_m) = \begin{cases} Y \text{ in } s_k & \text{if } (s_1, \ldots, s_k) \text{ is a computation}, k \geq 1, \\ \uparrow & \textit{otherwise}. \end{cases} \tag{9}$$

The definitions of partially computable and computable functions are made by Davis et al. [7] (Chap. 2, Sec. 4, p. 30) in terms of $\mathcal{L}$ programs as follows.

Definition 2. *An n-ary function f is partially computable if f is partial and there is a $\mathcal{L}$ program $\mathcal{P}$ such that Equation (10) holds for all $x_1, \ldots, x_n$.*

$$f(x_1, \ldots, x_n) = \Psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n). \tag{10}$$

Definition 3. *A n-ary function f is computable if it is total and partially computable.*

Equation (10) in Definition 2 is interpreted so that $f(x_1, \ldots, x_n)$ is defined if and only if $\Psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n)$ is defined. This treatment of computable functions in terms of programs in a formal language is by no means the only one in the literature. For example, as early as 1967, Meyer and Ritchie [19] formalize primitive recursive functions as loop programs consisting of assignment and iteration statements similar to DO statements of the programming language FORTRAN.

2.2. Computability of Primitive Recursive Functions

Davis et al. [7] (Chap. 3, Sec. 3, p. 42) introduce the concept of a *primitive recursively closed* (PRC) class of functions, which is a class of total functions that contains the initial functions and any functions obtained from the initial functions by a finite number of applications of composition or primitive recursion. Davis et al. [7] (Chap. 3, Sec. 3, pp. 42–43) show that (1) the class of computable functions is PRC; (2) the class of primitive recursive functions is PRC; and (3) a function is primitive recursive if and only if it belongs to every PRC class. A corollary of (3) is that every primitive recursive function is computable.

Péter [2,3] shows it is possible to define functions in terms of recursive equations that are not primitive recursive. In particular, Péter demonstrates that all unary primitive recursive functions are enumerable, i.e., $\phi_0(x), \phi_1(x), \phi_2(x), \ldots$ is an enumeration, with repetitions, of all unary primitive recursive functions. By Cantor's diagonalization (cf., e.g., pp. 6–8 in [4]), the unary function $f(x) = \phi_x(x) + 1$ is not in the enumeration and, hence, not primitive recursive. While f is not primitive recursive, it is computable (cf. Definition 3). Thus, the class of primitive recursive functions is a proper subset of computable functions and, in and of itself, cannot completely capture the intuitive notion of a number-theoretic algorithm. Péter's argument suffers no loss of generality, insomuch as any n-ary primitive recursive function, $n > 1$, can be reduced to an equivalent unary primitive recursive function (cf., Theorems 9.1 and 9.2, Chap. 4, Sec. 9, p. 108 in [7]). Kleene's separation of recursive functions into *general recursive* and *primitive recursive* may have been influenced by Péter's discovery (cited by Kleene [4] in Chap. XI, § 55, p. 272).

Rogers [6] (Chap. 1, § 1.2, p. 8) defines the *Ackermann generalized exponential*, a function for which there is no primitive recursive derivation, and formalizes it with the following recursive equations:

$$\begin{array}{lcl} f(0,0,y) & = & y, \\ f(0,x+1,y) & = & f(0,x,y)+1, \\ f(1,0,y) & = & 0, \\ f(z+2,0,y) & = & 1, \\ f(z+1,x+1,y) & = & f(z,f(z+1,x,y),y). \end{array}$$

3. Notational Conventions and Definitions

If f is a function, $dom(f)$ and $codom(f)$ are the domain and the co-domain of f. The expression $f : A \mapsto B$ abbreviates the logical conjunction $dom(f) = A \wedge codom(f) = B$, for some sets A and B. A function f is *partial* on A if $dom(f)$ is a proper subset of A, i.e., $dom(f) \subset A$. If f is partial on A and $a \in A$, the following statements are equivalent: (1) $a \in dom(f)$; (2) f is defined on a; (3) $f(a)$ is defined; (4) $f(a) \downarrow$. The following statements are also equivalent: (1) $a \notin dom(f)$; (2) f is undefined on a; (3) $f(a)$ is undefined; (4) $f(a) \uparrow$. If $dom(f) = A$, then f is *total* on A.

The notation $(a_1, \ldots, a_n)$ is used to denote *ordered n-tuples* or, simply, n-tuples over some set of numbers A. We will use bold lower-case variables, e.g., $\mathbf{a}, \mathbf{x}, \mathbf{y}$, to refer to n-tuples. Thus, $\mathbf{a} = (13, 17, 19)$ is a 3-tuple over the set of natural numbers $\mathbb{N} = \{0, 1, 2 \ldots\}$. We will use the symbol $\mathbb{N}^+$ to denote the set of positive natural numbers. If $\mathbf{x} = (x_1, \ldots, x_n)$ is an n-tuple over A, then $\mathbf{x}[j]$, $1 \leq j \leq n$, refers to individual elements of $\mathbf{x}$. Thus, if $\mathbf{x} = (2, 3, 5, 7, 11)$, then $\mathbf{x}[1] = 2$, $\mathbf{x}[2] = 3$, $\mathbf{x}[3] = 5$, $\mathbf{x}[4] = 7$, $\mathbf{x}[5] = 11$. The individual elements of an n-tuple are not required to be distinct. If $\mathbf{x}$ is an n-tuple, then $dim(\mathbf{x}) = n$, i.e., the number of elements in $\mathbf{x}$. The 0-tuple is denoted as $()$. In calculus, a *sequence* is an ordered set of numbers in a one-to-one correspondence with $\mathbb{N}$ or $\mathbb{N}^+$ (cf., e.g., Taylor [20], § 1.62, p. 67). Thus, if $f : \mathbb{N} \to \mathbb{N}$, then $\{f(n)\}$ denotes the sequence $f(0), f(1), \ldots, f(m), \ldots$ with countably many elements or terms. In computability theory, the term *sequence* sometimes refers to an n-tuple (cf., e.g., Ch. 3, p. 60 in [7]). Thus, in order to avoid confusion, when we want to emphasize the fact that we are dealing with a finite number of ordered elements, we refer to the collection of these elements as a *finite sequence*, a *tuple*, or an m-tuple, where m is the number of the elements.

For $n > 0$, A^n is the n-th *Cartesian power* of A, i.e., $A^n = \{(a_1, \ldots, a_n) | a_i \in A, 1 \leq i \leq n\}$. Thus, if $f : \mathbb{R}^2 \mapsto \mathbb{N}$, $dom(f) = \{(x_1, x_2) | x_1, x_2 \in \mathbb{R}\}$, where $\mathbb{R}$ is the set of real numbers. We use statements like $\mathbf{a} \in A^n$ to mean that $\mathbf{a}$ is an n-tuple over A. We do not distinguish between 1-tuples and individual elements, e.g., $a = (a)$, $a \in A$, and $h(a) = h((a))$ for some function h.

In formalizing feedforward artificial neural networks, it is sometimes convenient to treat n-tuples as vectors. Therefore, we occasionally use symbols like $\vec{x}$, $\vec{y}$, $\vec{z}$ to denote n-tuples. If $\vec{x} \in A^n$, then $dim(\vec{x}) = n$ and $\vec{x}[j]$, $1 \leq j \leq n$, is the j-th element of $\vec{x}$. E.g., if $\vec{x} = (1, 1, 11) \in \mathbb{N}^3$, then $\vec{x}[1] = \vec{x}[2] = 1$ and $\vec{x}[3] = 11$. If $\mathbf{a} \in A^n$ and $\vec{a} \in A^n$, and $\mathbf{a}[j] = \vec{a}[j]$, $1 \leq j \leq n$, then $\mathbf{a} = \vec{a}$. If $f : A^n \mapsto B^m$, $0 < n, m$, then $f(x_1, \ldots, x_n) = f(\vec{x}) = f(\mathbf{x}) = f(\vec{x}[1], \ldots, \vec{x}[n]) = f(\mathbf{x}[1], \ldots, \mathbf{x}[n]) = \vec{y} = \mathbf{y} = (\vec{y}[1], \ldots, \vec{y}[m]) = (\mathbf{y}[1], \ldots, \mathbf{y}[m])$. The empty tuple is discarded in function arguments. E.g., if $h : \mathbb{N} \mapsto \mathbb{N}$, then $h((), t) = h(t, ()) = h(t)$, $t \in \mathbb{N}$. We occasionally separate individual arguments of functions from the remaining arguments combined into tuples. E.g., if $f : \mathbb{N}^{n+2} \mapsto \mathbb{N}$, $0 < n$, then, for $\mathbf{z} \in \mathbb{N}^n$, $x \in \mathbb{N}$, $y \in \mathbb{N}$, $f(\mathbf{z}, x, y) = f(\mathbf{z}[1], \ldots, \mathbf{z}[n], x, y) = f(\mathbf{w})$, where $\mathbf{z}[i] = \mathbf{w}[i]$, $1 \leq i \leq n$, and $\mathbf{w}[n+1] = x$, $\mathbf{w}[n+2] = y$. If f is a function that maps $\mathbf{a}_1 \in A_1^{n_1}, \ldots, \mathbf{a}_k \in A_k^{n_k}$ to $\mathbf{c} \in C^m$, for some sets $A_1, \ldots, A_k$, $0 < n_j, m$, $1 \leq j \leq k$, then $f : A_1^{n_1}, \ldots, A_k^{n_k} \mapsto C^m$.

A total function $P : A^n \mapsto \{0, 1\}$ is a *predicate*, where 1 arbitrarily designates logical truth and 0 logical falsehood. The symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$ respectively refer to logical *not*, logical *and*, logical *or*, and logical *implication*. $P(\mathbf{x})$ is a shorthand for $P(\mathbf{x}) = 1$, and $\neg P(\mathbf{x})$ is a shorthand for $P(\mathbf{x}) = 0$. If P and Q are predicates, then $\neg P \vee Q$ is logically equivalent to $P \rightarrow Q$, i.e., $\neg P \vee Q \equiv P \rightarrow Q$. The symbols $\exists$ and $\forall$ refer to the logical *existential* (there exists) and *universal* (for all) quantifiers, respectively. Thus, the statement $(\exists \mathbf{x}) P(\mathbf{x})$ is logically equivalent to the statement that $P(\mathbf{x})$ holds for at least one $\mathbf{x}$ in $dom(P)$, while the statement $(\forall \mathbf{x}) P(\mathbf{x})$ is logically equivalent to the statement that $P(\mathbf{x})$ holds for every $\mathbf{x}$ in $dom(P)$.

Let, for $0 < k, n$, $f : \mathbb{N}^k \mapsto \mathbb{N}$, $g_j : \mathbb{N}^n \mapsto \mathbb{N}$, $1 \leq j \leq k$, and $\mathbf{x} \in \mathbb{N}^n$. We use the following definitions of composition and primitive recursion in our article. A function of $h : \mathbb{N}^n \mapsto \mathbb{N}$ is obtained from f, g_j by *composition* if h is obtained from f, g_j by Schema (11).

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), ..., g_k(\mathbf{x})). \tag{11}$$

Let $k \in \mathbb{N}$ and $\phi : \mathbb{N}^2 \mapsto \mathbb{N}$ be total. A function $h : \mathbb{N} \mapsto \mathbb{N}$ is obtained from ϕ by *primitive recursion* if it is obtained from ϕ by Schema (12).

$$\begin{array}{lcl} h(0) & = & k, \\ h(t+1) & = & \phi(t, h(t)). \end{array} \tag{12}$$

Let $f : \mathbb{N}^n \mapsto \mathbb{N}$ and $g : \mathbb{N}^{n+2} \mapsto \mathbb{N}$ be total, then $h : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ is obtained from f and g by *primitive recursion* if h is obtained from f and g by Schema (13), where $\mathbf{x} \in \mathbb{N}^n$.

$$\begin{array}{lcl} h(\mathbf{x}, 0) & = & f(\mathbf{x}), \\ h(\mathbf{x}, t+1) & = & g(t, h(\mathbf{x}, t), \mathbf{x}). \end{array} \tag{13}$$

If $\vec{x} \in \mathbb{N}^n$, Schema (13) can be expressed with the vector notation as

$$\begin{array}{lcl} h(\vec{x}, 0) & = & f(\vec{x}), \\ h(\vec{x}, t+1) & = & g(t, h(\vec{x}, t), \vec{x}). \end{array} \tag{14}$$

Let the set of *initial* functions consist of

$$\begin{array}{l} s(x) = x + 1, x \in \mathbb{N}; \\ n(x) = 0, x \in \mathbb{N}; \\ u_i^n(x_1, \ldots, x_n) = u_i^n(\vec{x}) = \vec{x}[i] = u_i^n(\mathbf{x}) = \mathbf{x}[i] = x_i, \quad 1 \le i \le n, \vec{x} = \mathbf{x} \in \mathbb{N}. \end{array} \tag{15}$$

Definition 4. *A function is primitive recursive if it can be obtained from the initial functions by a finite number of applications of Schemata (11)–(13).*

A corollary of Definition 4 is that if f is primitive recursive, then there is a sequence of functions $\phi_1, \ldots, \phi_n = f$ such that ϕ_i, $1 \le i \le n$, is either an initial function or is obtained from the previous functions in the sequence by composition or primitive recursion.

4. Feedforward Artificial Neural Networks

A *feedforward artificial neural network* $\mathfrak{N}_z$ is a finite set of *neurons*, each of which is connected to a finite number of the neurons in the same set through the *synapses*, i.e., directed weighted edges (cf. Figure 1). The neurons are organized into l layers $\mathfrak{C}_1, \ldots, \mathfrak{C}_l$, where $\mathfrak{C}_1$ is the input layer, $\mathfrak{C}_l$ is the output layer, and $\mathfrak{C}_e$, $1 < e < l$ are the hidden layers. We use the term *network* synonymously with the term *feedforward artificial neural network*.

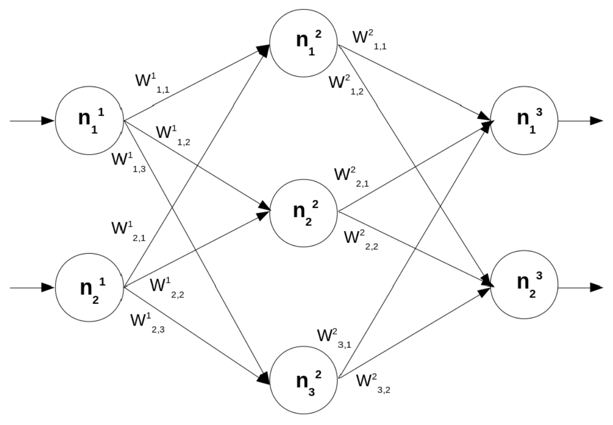

Figure 1. A 3-layer feedforward artificial neural network. Layer 1 includes the input neurons n_1^1 and n_2^1. Layer 2 includes the neurons n_1^2, n_2^2, n_3^2. Layer 3 includes the neurons n_1^3, n_2^3. The two arrows incoming into n_1^1 and n_2^1 signify that layer 1 is the input layer. The two arrows going out of n_1^3 and n_2^3 signify that layer 3 is the output layer. The weight of the synapse from n_i^e to n_j^{e+1} is $w_{i,j}^e$, $1 \le e \le 3$. E.g., $w_{1,1}^1$ is the weight of the synapse from n_1^1 to n_1^2 and $w_{3,1}^2$ is the weight of the synapse from n_3^2 to n_1^3.

Let $\mathfrak{z}_z$ denote the number of layers in $\mathfrak{N}_z$ and n_i^e refer to the i-th neuron in layer $\mathfrak{E}_e$, $1 \leq e \leq \mathfrak{z}_z$. The function $nn_z(e) : \mathbb{N}^+ \mapsto \mathbb{N}^+$ specifies the number of neurons in layer $\mathfrak{E}_e$ of $\mathfrak{N}_z$. We assume that $\mathfrak{N}_z$ is *fully connected*, i.e., there is a synapse from every neuron in layer $\mathfrak{E}_e$ to every neuron in layer $\mathfrak{E}_{e+1}$, $1 \leq e < \mathfrak{z}_z$. Each synaptic weight $w_{i,j}^e$ (cf. Figure 1) is a real number. The vector $\vec{w}^e$ is the vector of all synaptic weights in $\mathfrak{N}_z$ from $\mathfrak{E}_e$ to $\mathfrak{E}_{e+1}$. Thus,

$$\vec{w}^e = \left(w_{1,1}^e, \ldots, w_{1,nn_z(e+1)}^e, \ldots, w_{nn_z(e),1}^e, \ldots, w_{nn_z(e),nn_z(e+1)}^e\right).$$

We let $\vec{w}^0 = ()$ and assume, without loss of generality, that, for any synaptic weight $w_{i,j}^e$, $0 \leq w_{i,j}^e \leq 1$, because, if that is not the case, $w_{i,j}^e$ can be so scaled. No loss of generality is introduced with the assumption of full connectivity, because if full connectivity is not required, appropriate synaptic weights are set to zero. If, on the other hand, a given network is not fully connected, synapses with zero weights can be added as needed to make the network fully connected.

Each n_i^e, $e > 1$ computes an activation function

$$\alpha_i^e\left(\vec{a}^{e-1}, \vec{w}^{e-1}\right) : \mathbb{R}^{dim(\vec{a}^{e-1})}, \mathbb{R}^{dim(\vec{w}^{e-1})} \rightarrow \mathbb{R}, \tag{16}$$

where $\vec{a}^{e-1}$ is the vector of the activations of the neurons in layers $\mathfrak{E}_{e-1}$, $dim(\vec{a}^{e-1}) = nn_z(e-1)$, and $dim(\vec{w}^{e-1}) = nn_z(e-1)nn_z(e)$. If $\vec{x}$ is the input to $\mathfrak{N}_z$, then $\vec{a}^1 = \vec{x}$. For the input layer, we have

$$\alpha_i^1(\vec{x}, \vec{w}^0) = \alpha_1^1(\vec{x}, ()) = \vec{x}[i], 1 \leq i \leq nn_z(1). \tag{17}$$

The term *feedforward* means that the activations of the neurons are computed layer by layer from the input layer to the output layer, because the activation functions of the neurons in the next layer require only the weights of the synapses connecting the current layer with the next one and the activation values, i.e., the outputs of the activation functions of the neurons in the current layer. If $\vec{x}$ is the input vector, then

$$\begin{aligned} \vec{a}^1 &= \left(\alpha_1^1\left(\vec{x}, ()\right), \ldots \alpha_{nn_z(1)}^1\left(\vec{x}, ()\right)\right) = \vec{x}, \\ \vec{a}^e &= \left(\alpha_1^e\left(\vec{a}^{e-1}, \vec{w}^{e-1}\right), \ldots, \alpha_{nn(e)}^e\left(\vec{a}^{e-1}, \vec{w}^{e-1}\right)\right), 1 < e < \mathfrak{z}_z. \end{aligned} \tag{18}$$

The feedforward activation function f_z that computes the activations of $\mathfrak{N}_z$ layer by layer can be defined as

$$\begin{aligned} f_z(\vec{x}, 0) &= \vec{x}, \\ f_z(\vec{x}, e+1) &= \left(\alpha_1^{e+1}\left(f_z\left(\vec{x}, e\right), \vec{w}^e\right), \ldots, \alpha_{nn(e+1)}^{e+1}\left(f_z\left(\vec{x}, e\right), \vec{w}^e\right)\right). \end{aligned} \tag{19}$$

Thus, $f_z(\vec{x}, 0) = f_z(\vec{x}, 1) = \vec{a}^1 = \vec{x} = \mathbf{x}$ and $f_z(\vec{x}, e) = \vec{a}^e, 1 \leq e \leq \mathfrak{z}_z$. If $\mathfrak{N}_z$ maps $\vec{a}^1 \in A^n$ to $\vec{b}^{\mathfrak{z}_z} \in B^m$, for some sets A and B, we define the function $\zeta_z : A^n \rightarrow B^m$ computed by $\mathfrak{N}_z$ as

$$\zeta_z(\vec{x}) = f_z(\vec{x}, \mathfrak{z}_z). \tag{20}$$

Definition 5. *A function $f : A^n \rightarrow B^m$, for some sets A and B, is $\mathfrak{N}$-computable if there is a network $\mathfrak{N}_z$ such that, for all $\vec{x} = \mathbf{x} \in A^n$,*

$$\zeta_z(\vec{x}) = f(\vec{x}, \mathfrak{z}_z) = \vec{y} = \zeta_z(\mathbf{x}) = f(\mathbf{x}, \mathfrak{z}_z) = \mathbf{y} \in B^m.$$

If $\mathfrak{N}_z$ computes f, we refer to $\mathfrak{N}_z$ as $\mathfrak{N}_{f(\cdot)}$ and use the expression $\mathfrak{N}_{f(\cdot)} : A^n \mapsto B^m$ as a shorthand for $\zeta_z : A^n \mapsto B^m$. Furthermore, if $\mathfrak{N}_z$ computes f, then, for $\vec{x} = \mathbf{x} \in A^n$, the expressions $\zeta_z(\vec{x})$, $\zeta_z(\mathbf{x})$, $\mathfrak{N}_z(\vec{x})$, $\mathfrak{N}_z(\mathbf{x})$ are equivalent in that

$$\zeta_z(\vec{x}) = \mathfrak{N}_z(\vec{x}) = \vec{y} = \zeta_z(\mathbf{x}) = \mathfrak{N}_z(\mathbf{x}) = \mathbf{y} \in B^m. \tag{21}$$

A network $\mathfrak{N}_z$ can include other networks. Let $\mathfrak{N}_j$ and $\mathfrak{N}_k$ be two networks such as $\zeta_j : A^m \mapsto B^n$ and $\zeta_k : B^n \mapsto C^k$, for some sets A, B, C, and $0 < m, n, k$. Then we can construct a new network $\mathfrak{N}_l$ by feeding the output of $\mathfrak{N}_j$ to $\mathfrak{N}_k$ so that $\zeta_l : A^m \mapsto C^k$ (cf. Figure 2). We can generalize this case to a network that includes arbitrarily many networks whose outputs are the inputs to another network whose output is the output of the entire complex network (cf. Figure 3). Formally, let $\mathfrak{N}_{z_1}, \ldots, \mathfrak{N}_{z_l}$ be networks such that $\zeta_{z_1} : I^{n_{z_1}} \to O^{k_{z_1}}, \ldots, \zeta_{z_l} : I^{n_{z_l}} \to O^{k_{z_l}}$ for some sets I and O, $0 < n_{z_i}, k_{z_i}$, and $1 \le i \le l$. Let, for some set S, a network $\mathfrak{N}_j$ compute the function

$$\zeta_j : O^{k_{z_1}}, O^{k_{z_2}}, \ldots, O^{k_{z_l}} \to S^m$$

so that

$$\zeta_z(\vec{x}_{z_1}, \ldots, \vec{x}_{z_l}) = \zeta_j(\zeta_{z_1}(\vec{x}_{z_1}), \ldots, \zeta_{z_l}(\vec{x}_{z_l})) = \vec{s} \in S^m, \vec{x}_{z_i} \in I^{n_{z_i}}, 1 \le i \le l.$$

Then, for $\vec{x}_{z_i} \in I^{n_{z_i}}$ such that $\vec{x}_{z_i} = \mathbf{x}_{z_i}$, $1 \le i \le l$,

$$\mathfrak{N}_z(\mathbf{x}_{z_1}, \ldots, \mathbf{x}_{z_l}) = \mathfrak{N}_z(\mathbf{y}) = \mathfrak{N}_j(\mathfrak{N}_{z_1}(\mathbf{x}_{z_1}), \ldots, \mathfrak{N}_{z_l}(\mathbf{x}_{z_l})) = \mathbf{s} \in S^m, \tag{22}$$

where $\mathbf{y} = (\mathbf{x}_{z_1}[1], \ldots, \mathbf{x}_{z_1}[n_{z_1}], \ldots, \mathbf{x}_{z_l}[1], \ldots, \mathbf{x}_{z_l}[n_{z_l}])$, and $\mathbf{s} = \vec{s}$.

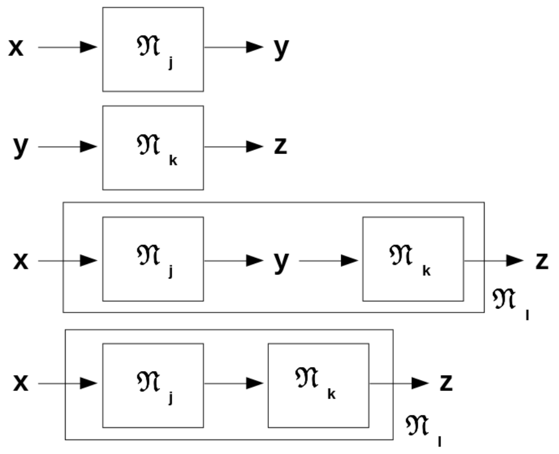

Figure 2. A chain network $\mathfrak{N}_l$ that consists of two networks $\mathfrak{N}_j$ (top) and $\mathfrak{N}_k$ (second from the top). The two bottom networks are functionally identical pictogrammatic renderings of the same network $\mathfrak{N}_l$. In the third network from the top the output $\mathbf{y}$ of $\mathfrak{N}_j$ is made explicit. In the bottom rendering of $\mathfrak{N}_l$, $\mathbf{y}$ is implicit in the arrow from $\mathfrak{N}_j$ to $\mathfrak{N}_k$. In sum, the output of $\mathfrak{N}_j$ is given to $\mathfrak{N}_k$, and the output of $\mathfrak{N}_k$ is the output of $\mathfrak{N}_l$. Thus, $\mathfrak{N}_l$ maps $\mathbf{x}$ to $\mathbf{z}$.

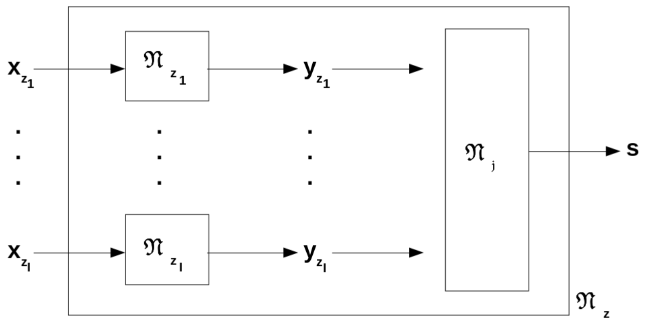

Figure 3. A network $\mathfrak{N}_z$ that includes networks $\mathfrak{N}_{z_1}, \ldots, \mathfrak{N}_{z_l}$ that take $\mathbf{x}_{z_1}, \ldots, \mathbf{x}_{z_k}$ as inputs and give their outputs to network $\mathfrak{N}_j$ (cf. Equation (22)). Thus, $\mathfrak{N}_z$ maps $\mathbf{x}_{z_1}, \ldots, \mathbf{x}_{z_k}$ to $\mathbf{s}$.

We use the symbol $\mathfrak{N}_{id}$ to denote an identity network such that, for $\vec{a} = \mathbf{a} \in A^n$, $0 < n$, $\zeta_{id}(\vec{a}) = \vec{a} = \zeta_{id}(\mathbf{a}) = \mathbf{a}$. One can think of $\mathfrak{N}_{id}$ as a single layer network of n neurons, where $\alpha_i^1(\vec{a}, ()) = \vec{a}[i] = \alpha_i^1(\mathbf{a}, ()) = \mathbf{a}[i]$, $1 \le i \le n$.

Our formalization of feedforward artificial neural networks as finite sets of neurons and synapses organized in finitely many layers is in compliance with the original definition by McCulloch and Pitts (Sec. 2, p. 103 in [11]) who state that the neurons of a given network may be assigned designations $c_1, c_2, \ldots, c_n$. It is also in compliance with the subsequent definition by Rumelhart, Hinton, and Williams [12] as well as with modern treatments of neural networks by Nielsen [17] and Goodfellow, Bengio, and Courville [18] that continue to describe neural networks as finite sets of neurons and synapses.

5. $\mathfrak{N}$-Computability of Primitive Recursive Functions

Lemma 1. *The initial functions are $\mathfrak{N}$-computable.*

Proof. Let $\mathfrak{N}_{n(\cdot)} : \mathbb{N} \mapsto \mathbb{N}$ be a network with a single input node n_1^1 and a single output node n_1^2 such that $w_{1,1}^1 = 0$ and $\alpha_1^2(\vec{a}^1, \vec{w}^1) = \vec{a}^1[1]\vec{w}^1[1]$. Then, $\zeta_{n(\cdot)}(x) = \alpha_1^2((x)(0)) = x \cdot 0 = 0 = n(x)$, $x \in \mathbb{N}$. Let $\mathfrak{N}_{s(\cdot)} : \mathbb{N} \mapsto \mathbb{N}$ be a network with a single input node n_1^1 and a single output node n_1^2 such that $w_{1,1}^1 = 1$ and $\alpha_1^2(\vec{a}, \vec{w}) = \vec{a}[1]\vec{w}^1[1] + 1$. Then, $\zeta_{s(\cdot)}(x) = \alpha_1^2((x), (1)) = xw_{1,1}^1 + 1 = s(x)$, $x \in \mathbb{N}$. Let $\mathfrak{N}_{u_i^n(\cdot)} : \mathbb{N}^n \mapsto \mathbb{N}$, $1 \le i \le n$, $n > 0$ be a network with n input nodes $n_1^1, \ldots, n_n^1$ and one output node n_1^2. Let $w_i^1 = 1$, $w_j^1 = 0$, $i \ne j$, $1 \le j \le n$, and

$$\alpha_1^2(\vec{a}^1, \vec{w}^1) = \sum_{j=1}^{n} \vec{a}^1[j]\vec{w}^1[j].$$

Then, if $\vec{a} = \mathbf{a} \in \mathbb{N}^n$,

$$\zeta_{u_i^n(\cdot)}(\vec{a}) = \alpha_1^2(\vec{a}^1, \vec{w}^1) = \vec{a}[i] = \alpha_1^2(\mathbf{a}, \vec{w}^1) = \mathbf{a}[i] = u_i^n(\mathbf{a}[1], \ldots, \mathbf{a}[n]).$$

□

We abbreviate $\mathfrak{N}_{u_i^n(\cdot)}$ as $\mathfrak{N}_{u(\cdot)}$, because n and i are always evident from the context.

Lemma 2. *Let $\mathbf{x} \in \mathbb{N}^n$, $n > 0$. Let $c_i^n(\mathbf{x})$, $1 \le i \le n$, be defined as*

$$c_i^n(\mathbf{x}) = \begin{cases} u_1^n(\mathbf{x}) & \text{if } i = 1, \\ u_2^n(\mathbf{x}) & \text{if } i = 2, \\ \ldots & \\ u_n^n(\mathbf{x}) & \text{if } i = n. \end{cases}$$

The c_i^n is $\mathfrak{N}$-computable.

Proof. Since u_i^n is primitive recursive, c_i^n is primitive recursive, by the definition by cases theorem and its corollary (cf. Theorem 5.4, Chap. 3, Sec. 5, pp. 50–51 in [7]). Let $\mathfrak{N}_{c_i^n(\cdot)}$ be a network with $n+1$ input nodes $n_1^1, \ldots, n_{n+1}^1$, where the first n nodes receive the n corresponding values of $\mathbf{x} \in \mathbb{N}^n$, and the last node n_{n+1}^1 receives $1 \le i \le n$. Let $\mathfrak{N}_{c_i^n(\cdot)}$ have one output node n_1^2 and let $w_{j,k}^1 = 1$, $1 \le j \le n$. Let the activation function of n_1^2 be defined as

$$\alpha_1^2(\vec{a}^1, \vec{w}^1) = \begin{cases} \vec{a}^1[1]\vec{w}^1[1] & \text{if } \vec{a}^1[n+1] = 1, \\ \vec{a}^1[2]\vec{w}^1[2] & \text{if } \vec{a}^1[n+1] = 2, \\ \ldots & \\ \vec{a}^1[n]\vec{w}^1[n] & \text{if } \vec{a}^1[n+1] = n. \end{cases}$$

Then,

$$\zeta_{c_i^n}(\mathbf{x}, j) = \mathbf{x}[j] = u_j^n(\mathbf{x}) = c_j^n(\mathbf{x}).$$

□

We abbreviate $\mathfrak{N}_{c_i^n(\cdot)}$ as $\mathfrak{N}_{c(\cdot)}$.

Lemma 3. *Let f be a $\mathfrak{N}$-computable function of k arguments, $k > 0$, and g_1, ..., g_k be $\mathfrak{N}$-computable functions of n arguments each, $n > 0$. Let a function h of n arguments be obtained from $f, g_1, \ldots, g_k$ by Schema (11). Then, h is $\mathfrak{N}$-computable.*

Proof. Let $f, g_1, \ldots, g_k$ be computable by $\mathfrak{N}_{f(\cdot)} : \mathbb{N}^k \mapsto \mathbb{N}$, $\mathfrak{N}_{g_1(\cdot)} : \mathbb{N}^n \mapsto \mathbb{N}, \ldots, \mathfrak{N}_{g_k(\cdot)} : \mathbb{N}^n \mapsto \mathbb{N}$. Then let $\mathfrak{N}_j : \mathbb{N}^n \mapsto \mathbb{N}$ be a network such that, for $\mathbf{x} \in \mathbb{N}^n$,

$$\mathfrak{N}_j(\mathbf{x}) = \mathfrak{N}_{f(\cdot)}(\mathfrak{N}_{g_1(\cdot)}(\mathbf{x}), \ldots, \mathfrak{N}_{g_k(\cdot)}(\mathbf{x})).$$

Then, for $\mathbf{z} \in \mathbb{N}^n$, we have

$$\mathfrak{N}_j(\mathbf{z}) = \mathfrak{N}_{f(\cdot)}(\mathfrak{N}_{g_1(\cdot)}(\mathbf{z}), \ldots, \mathfrak{N}_{g_k(\cdot)}(\mathbf{z})) = f(g_1(\mathbf{z}), \ldots, g_k(\mathbf{z})) = h(\mathbf{x}),$$

whence

$$\zeta_j(\mathbf{x}) \quad = \quad h(\mathbf{x}).$$

□

Lemma 4. *Let $k \in \mathbb{N}$. Then k is $\mathfrak{N}$-computable.*

Proof. Let $\mathfrak{N}_{n(\cdot)}$ and $N_{s(\cdot)}$ be as constructed in Lemma 1. Let $\{\mathfrak{N}_{s(\cdot)}\}^k$, $k \geq 0$ denote a network that consists of a finite sequence of k networks $\mathfrak{N}_{s(\cdot)}$, where the first $\mathfrak{N}_{s(\cdot)}$ receives its input from $\mathfrak{N}_{n(\cdot)}$ and each subsequent $\mathfrak{N}_{s(\cdot)}$ receives its input from the previous $\mathfrak{N}_{s(\cdot)}$ (cf. Figure 2). Let $\{\mathfrak{N}_{s(\cdot)}\}^0 = \mathfrak{N}_{n(\cdot)}$. Let $\mathfrak{N}_{\mathfrak{J}_k}(0) = \{\mathfrak{N}_{s(\cdot)}\}^k(\mathfrak{N}_{n(\cdot)}(0))$. Let $s^k(x)$ denote k compositions of $s(x)$ with itself, i.e., $s^1(x) = s(x)$, $s^2(x) = s(s(x))$, etc. Then,

$$\begin{array}{rcl} \mathfrak{N}_{\mathfrak{J}_0}(0) & = & \{\mathfrak{N}_{s(\cdot)}\}^0(\mathfrak{N}_{n(\cdot)}(0)) = 0; \\ \mathfrak{N}_{\mathfrak{J}_1}(0) & = & \{\mathfrak{N}_{s(\cdot)}\}^1(\mathfrak{N}_{n(\cdot)}(0)) = s(0) = 1; \\ \mathfrak{N}_{\mathfrak{J}_2}(0) & = & \{\mathfrak{N}_{s(\cdot)}\}^2(\mathfrak{N}_{n(\cdot)}(0)) = s^2(0) = 2; \\ & \cdots & \\ \mathfrak{N}_{\mathfrak{J}_k}(0) & = & \{\mathfrak{N}_{s(\cdot)}\}^k(\mathfrak{N}_{n(\cdot)}(0)) = s^k(0) = k. \end{array}$$

By induction on k, $\zeta_{\mathfrak{J}_k}(0) = k$. By construction, $\zeta_{\mathfrak{J}_k}(n) = k$, $n \in \mathbb{N}$. □

The next lemma, Lemma 5, is a technical result for Lemma 6. The function $x \dot{-} y$ is primitive recursive (cf. Chap. 3, Sec. 4, p. 46 in [7]).

Lemma 5. *Let the function $x \dot{-} y : \mathbb{N}^2 \to \mathbb{N}$ be defined as*

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq 0, \\ 0 & \text{if } x < y. \end{cases} \tag{23}$$

Then, $x \dot{-} y$ is $\mathfrak{N}$-computable.

Proof. Let $\mathfrak{N}_{\dot{-}(\cdot)}$ have two input nodes n_1^1, n_2^1 and one output node n_1^2. Let $w_{1,1}^1 = w_{2,1}^1 = 1$ and let

$$\alpha_1^2(\vec{a}^1, \vec{w}^1) = \begin{cases} \vec{a}^1[1]\vec{w}^1 - \vec{a}^1[2]\vec{w}^1[2] & \text{if } \vec{a}^1[1] \geq \vec{a}^1[2], \\ 0 & \text{if } \vec{a}^1[1] < \vec{a}^1[2]. \end{cases}$$

Then, for $\vec{a} = \mathbf{a} \in \mathbb{N}^2$, we have

$$\zeta_{\dot{-}(\cdot)}\left(\vec{a}\right) = \alpha_1^2(\vec{a}^1, \vec{w}^1) = \vec{a}^1[1] \dot{-} \vec{a}[2] = \alpha_1^2(\mathbf{a}, \vec{w}^1) = \mathbf{a}[1] \dot{-} \mathbf{a}[2].$$

□

Definition 6 confines the notion of $\mathfrak{N}$-computability of some function $f(\mathbf{x}, t)$ to the $\mathfrak{N}$-computability of the first k elements of the sequence $\{f(\mathbf{x}, t)\}$, $t \in \mathbb{N}$.

Definition 6. *A function $f : A^n \times \mathbb{N} \to B^m$, for some sets A and B, is $\mathfrak{N}$-computable elementwise for any $k > 0$ if there is a network $\mathfrak{N}_z$ such that, for any $\mathbf{z} \in A^n$, the first $k+1$ terms of the sequence*

$$\{f(\mathbf{z}, j)\} = f(\mathbf{z}, 0), f(\mathbf{z}, 1), \ldots, f(\mathbf{z}, k), \ldots$$

are the same as the terms of the tuple

$$(\mathfrak{N}(\mathbf{z}, 0), \quad \mathfrak{N}(\mathbf{z}, 1), \quad \ldots, \quad \mathfrak{N}(\mathbf{z}, k)),$$

i.e., $f(\mathbf{z}, i) = \mathfrak{N}(\mathbf{z}, i), 0 \leq i \leq k$.

Thus, if a function $f(\mathbf{x}, t)$ is $\mathfrak{N}$-computable, it is $\mathfrak{N}$-computable elementwise for any positive k.

Lemma 6. *Let $\phi : \mathbb{N}^2 \mapsto \mathbb{N}$ be $\mathfrak{N}$-computable elementwise and $h(t)$ be a function obtained from ϕ by Schema (12). Then, h is $\mathfrak{N}$-computable elementwise.*

Proof. Let ϕ be computable elementwise by $\mathfrak{N}_{\phi(\cdot)}$. Let $\mathfrak{N}_{\tilde{h}_0}(0) = \mathfrak{N}_{\mathfrak{I}_k}(0) = k$ as constructed in Lemma 4. In the equations below, we abbreviate $\mathfrak{N}_{n(\cdot)}(0)$ as 0, $\mathfrak{N}_{\mathfrak{I}_k}(0)$ as k, $\mathfrak{N}_{\mathfrak{I}_t}(0)$ as $\mathfrak{N}_{\mathfrak{I}_t}$, $\mathfrak{N}_{\dot{-}(\cdot)}(x, y)$ as $x \dot{-} y$, and $\mathfrak{N}_{\tilde{h}_i}(i)$ as $\mathfrak{N}_{\tilde{h}_i}$. Let

$$\begin{aligned} \mathfrak{N}_{\tilde{h}_0} &= k, \\ \mathfrak{N}_{\tilde{h}_{t+1}} &= \mathfrak{N}_{\phi(\cdot)}\left(\mathfrak{N}_{\mathfrak{I}_t}, \mathfrak{N}_{\tilde{h}_t}\right). \end{aligned} \tag{24}$$

By induction on t, $h(t) = \mathfrak{N}_{\tilde{h}(\cdot)}(t)$ (cf. Figure 4). Let

$$\mathfrak{N}_{h(\cdot)}(t) = \mathfrak{N}_{c(\cdot)}\left(\mathfrak{N}_{\tilde{h}_0}, \ldots, \mathfrak{N}_{\tilde{h}_m}, t+1\right), 0 \leq t \leq m, m > 0. \tag{25}$$

Then, the first $m+1$ terms of the sequence $\{h(t)\}$ are the same as the terms of the tuple $(\mathfrak{N}_{h(\cdot)}(0), \ldots, \mathfrak{N}_{h(\cdot)}(m))$ (cf. Figure 5 for $m = 3$). □

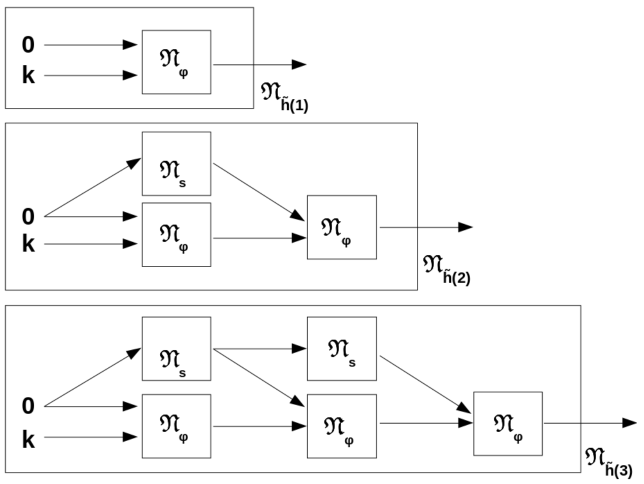

Figure 4. Networks $\mathfrak{N}_{\tilde{h}(1)}$, $\mathfrak{N}_{\tilde{h}(2)}$, $\mathfrak{N}_{\tilde{h}(3)}$ constructed with Schema (24) in Lemma 6. Note that **0** and **k** denote $\mathfrak{N}_{n(\cdot)}(0)$ and $\mathfrak{N}_{\mathfrak{I}_k}(0)$, respectively.

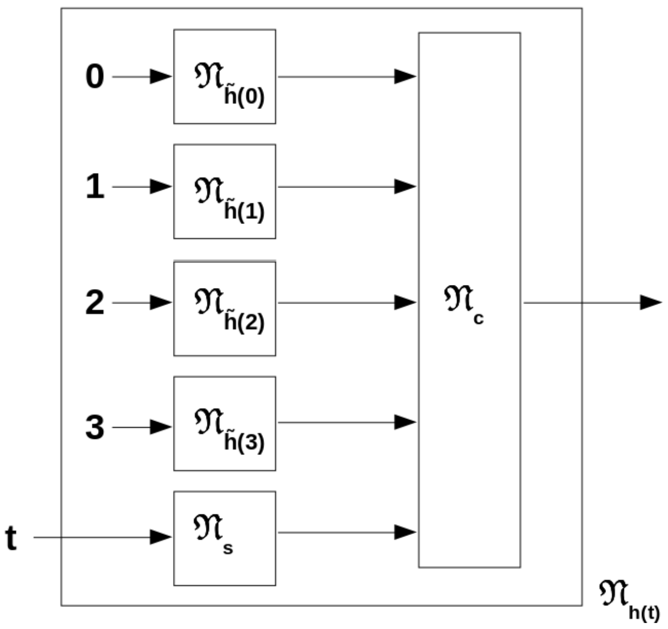
0
1
2
3
t
$\mathfrak{N}_{\tilde{h}(0)}$
$\mathfrak{N}_{\tilde{h}(1)}$
$\mathfrak{N}_{\tilde{h}(2)}$
$\mathfrak{N}_{\tilde{h}(3)}$
$\mathfrak{N}_s$
$\mathfrak{N}_c$
$\mathfrak{N}_{h(t)}$

Figure 5. Network $\mathfrak{N}_{h(\cdot)}(t)$, $0 \leq t \leq 3$, constructed with Equation (25) in Lemma 6. Since $h(0) = \mathfrak{N}_{h(\cdot)}(0), h(1) = \mathfrak{N}_{h(\cdot)}(1), h(2) = \mathfrak{N}_{h(\cdot)}(2), h(3) = \mathfrak{N}_{h(\cdot)}(3)$, the first four terms of the sequence $\{h(t)\}$ are the same as the terms of the 4-tuple $(\mathfrak{N}_{h(\cdot)}(0), \mathfrak{N}_{h(\cdot)}(1), \mathfrak{N}_{h(\cdot)}(2), \mathfrak{N}_{h(\cdot)}(3))$.

Lemma 7. *Let $f : \mathbb{N}^n \mapsto \mathbb{N}$ and $g : \mathbb{N}^2 \mapsto \mathbb{N}$ be $\mathfrak{N}$-computable elementwise and $h : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ be a function obtained from f and g by Schema (13). Then h is $\mathfrak{N}$-computable elementwise.*

Proof. Let $\mathbf{x} \in \mathbb{N}^n$ and $\mathbf{y} \in \mathbb{N}^{n+2}$, $n > 0$, such that $\mathbf{y} = (y_1, y_2, \mathbf{x}[\mathbf{1}], \ldots, \mathbf{x}[n])$. Let f and g be $\mathfrak{N}$-computable elementwise by $\mathfrak{N}_{f(\cdot)}$ and $\mathfrak{N}_{g(\cdot)}$, respectively. Let us abbreviate $\mathfrak{N}_{\tilde{h}_{\mathbf{x},t}}(\mathbf{x}, t)$ as $\mathfrak{N}_{\tilde{h}_{\mathbf{x},t}}$ and let

$$\begin{aligned} \mathfrak{N}_{\tilde{h}_{\mathbf{x},0}} &= \mathfrak{N}_{f(\cdot)}(\mathbf{x}), \\ \mathfrak{N}_{\tilde{h}_{\mathbf{x},t+1}} &= \mathfrak{N}_{g(\cdot)}\left(t, \mathfrak{N}_{\tilde{h}_{\mathbf{x},t}}, \mathbf{x}\right). \end{aligned} \tag{26}$$

By induction on t, $h(\mathbf{x}, t) = \mathfrak{N}_{\tilde{h}_{\mathbf{x},t}}$. Let

$$\mathfrak{N}_{h(\cdot)}(\mathbf{x}, t) = \mathfrak{N}_{c(\cdot)}\left(\mathfrak{N}_{\tilde{h}_{\mathbf{x},0}}, \ldots, \mathfrak{N}_{\tilde{h}_{\mathbf{x},m}}, t+1\right), 0 \leq t \leq m, m > 0. \tag{27}$$

Then the first $m+1$ terms of the sequence $\{h(\mathbf{x}, t)\}$, i.e., $h(\mathbf{x}, 0), \ldots, h(\mathbf{x}, m)$, agree elementwise with the tuple $(\mathfrak{N}_{h(\cdot)}(\mathbf{x}, 0), \ldots, \mathfrak{N}_{h(\cdot)}(\mathbf{x}, m))$. □

Figures 6 and 7 illustrate sample constructions of Lemma 7. If we treat $h(t)$ as a shorthand for $h((), t)$, then Lemmas 6 and 7 give us the following theorem.

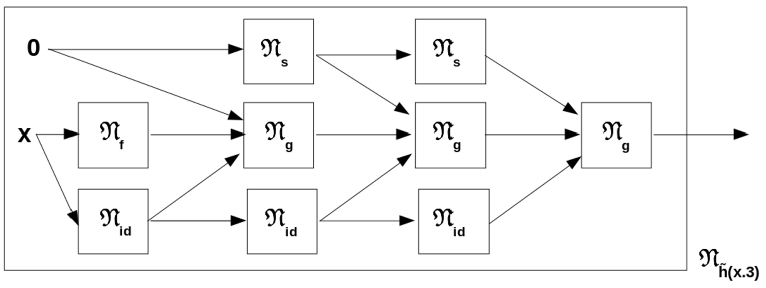

Figure 6. Network $\mathfrak{N}_{\tilde{h}(\mathbf{x},3)}$, constructed with Schema (26) in Lemma 7. Note that **0** denotes $\mathfrak{N}_{n(\cdot)}(0)$, and $\mathfrak{N}_{id}$ is the identity network.

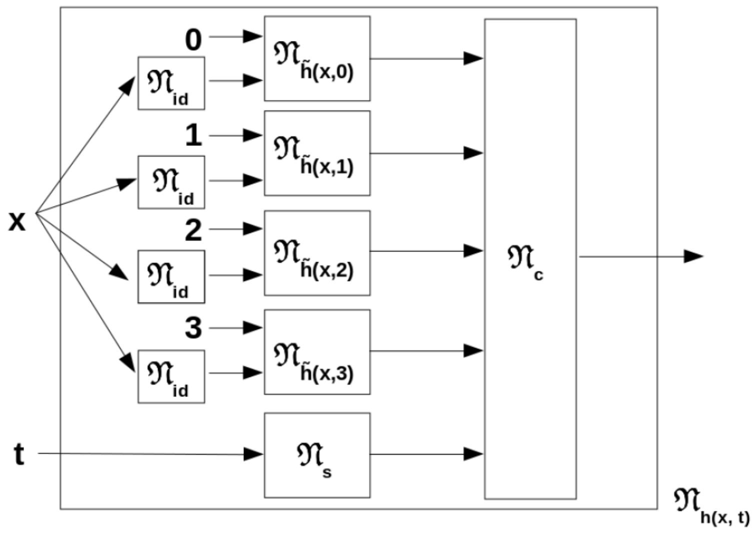
x
t
0
1
2
3
$\mathfrak{N}_{id}$
$\mathfrak{N}_{\tilde{h}(x,0)}$
$\mathfrak{N}_{\tilde{h}(x,1)}$
$\mathfrak{N}_{\tilde{h}(x,2)}$
$\mathfrak{N}_{\tilde{h}(x,3)}$
$\mathfrak{N}_{s}$
$\mathfrak{N}_{c}$
$\mathfrak{N}_{h(x,t)}$

Figure 7. Network $\mathfrak{N}_{h(\cdot)}(\mathbf{x},t)$, $0 \le t \le 3$, constructed with Equation (25) in Lemma 7. Since $h(\mathbf{x},0) = \mathfrak{N}_{h(\cdot)}(\mathbf{x},0)$, $h(\mathbf{x},1) = \mathfrak{N}_{h(\cdot)}(\mathbf{x},1)$, $h(\mathbf{x},2) = \mathfrak{N}_{h(\cdot)}(\mathbf{x},2)$, $h(\mathbf{x},3) = \mathfrak{N}_{h(\cdot)}(\mathbf{x},3)$, the first four terms of the sequence $\{h(\mathbf{x},t)\}$, i.e., $h(\mathbf{x},0), h(\mathbf{x},1), h(\mathbf{x},2), h(\mathbf{x},3)$, are the same as the terms of the tuple $(\mathfrak{N}_{h(\cdot)}(\mathbf{x},0), \mathfrak{N}_{h(\cdot)}(\mathbf{x},1), \mathfrak{N}_{h(\cdot)}(\mathbf{x},2), \mathfrak{N}_{h(\cdot)}(\mathbf{x},3))$.

Theorem 1. *Let $h(\mathbf{x},t)$ be a primitive recursive function, $\mathbf{x} \in \mathbb{N}^n$, $n \ge 0$. Then $h(\mathbf{x},t)$ is $\mathfrak{N}$-computable elementwise.*

We can ask if the elementwise $\mathfrak{N}$-computability of $h(\mathbf{x},t)$ (cf. Definition 6) can be generalized to $\mathfrak{N}$-computability. In other words, is it possible to have the sequences $\{h(\mathbf{x},t)\}$ and $\{\mathfrak{N}(\mathbf{x},n)\}$ agree term by term, i.e., $h(\mathbf{x},t) = \mathfrak{N}(\mathbf{x},t)$? Since $\mathfrak{N}$ has a finite set of neurons organized into a finite number of layers, $\mathfrak{N}$ can compute, per Lemmas 6 and 7, only the first $m+1$ values of $h(\mathbf{x},t)$, i.e., $h(\mathbf{x},t)$, $0 \le t \le m$, although m can be an arbitrarily large natural number. Thus, the answer to this question is negative.

Let us assume that $\mathfrak{N}_{h(\mathbf{x},t)}$ in Theorem 1 is allowed to have countably many neurons so that the number of neurons in the hidden layers of $\mathfrak{N}_{h(\mathbf{x},t)}$ is countable. Let $\zeta_{\mathfrak{N}}(\mathbf{x},t)$ be the function computed by $\mathfrak{N}_{h(\mathbf{x},t)}$. Since countably many neurons can be added to $\mathfrak{N}_{h(\mathbf{x},t)}$ to compute $h(\mathbf{x},t)$, for any t, we have the sequence $\{\zeta_{\mathfrak{N}}(\mathbf{x},t)\} = \{\mathfrak{N}(\mathbf{x},t)\}$, on the one hand, and the sequence $\{h(\mathbf{x},t)\}$, on the other hand. Let $f(\mathbf{x},t) = h(\mathbf{x},t) - \zeta_{\mathfrak{N}}(\mathbf{x},t)$. Since $h(\mathbf{x},t) = \zeta_{\mathfrak{N}}(\mathbf{x},t)$, for any $t \in \mathbb{N}$, $\{f(\mathbf{x},t)\}$ is vacuously convergent, i.e., $\lim_{t\to\infty} f(\mathbf{x},t) = 0$. Hence, we have the following theorem.

Theorem 2. *Let $h(\mathbf{x},t)$ be a primitive recursive function, $\mathbf{x} \in \mathbb{N}^n$, $n \ge 0$. Then there is a network $\mathfrak{N}(\mathbf{x},t)$ with countably many neurons such that for any $\mathbf{z} \in \mathbb{N}^n$, the sequences $\{h(\mathbf{z},t)\}$ and $\{\zeta_{\mathfrak{N}}(\mathbf{z},t\})\}$ agree term by term, i.e., $h(\mathbf{z},t) = \zeta_{\mathfrak{N}}(\mathbf{z},t)$, $t \in \mathbb{N}$.*

6. Discussion

As mathematical objects, feedforward artificial neural networks are more computationally powerful than primitive recursive functions inasmuch as the former can compute functions over real numbers whereas the latter, by definition, cannot. E.g., one can define a network that computes the sum of n real numbers, which no primitive recursive function can compute. However, the situation changes when networks cease to be mathematical objects and become computational objects by being realized on finite memory devices. A finite memory device is a computational device with a finite amount of memory available for numerical computation [21]. Such a device is analogous to a human scribe with a pencil and an eraser who is to carry out a numerical computation by writing and erasing symbols from a finite alphabet on a finite number of paper sheets. Finite memory devices are different from finite state automata of classical computability theory (e.g., a deterministic finite state machine (Chap. 2, Sec. 2.2 in [22]), non-deterministic finite state machine

(Chap. 2, Sec. 2.3 in [22]), a Mealy or Moore machine (Chap. 2, Sec. 2.7 in [22], a push down automaton (Chap. 5 in [22]), or a Turing machine (Chap. 6 in [7]), because the latter do not put any bounds on the number of cells in their tapes available for computation. A finite state automaton of classical computability becomes a finite memory device only when the number of its tape cells available for computation is bounded by a natural number.

A real number x is *signifiable* on a finite memory device D_j if and only if the finite amount of memory on D_j can hold its *sign*, where a sign is a sequence of arbitrary symbols from a finite alphabet [21]. Thus, if the alphabet is { ".", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" } and D_j has 8 memory cells to represent a real number, then the real numbers 1.41, 1.414, 1.4142, 1.41421, 1.414213 are signifiable on D_j as "1.41", "1.414", "1.4142", "1.41421", "1.414213", respectively, whereas the real numbers 1.4142135, 1.41421356, 1.414213562, 1.4142135623, and 1.41421356237 are not. A consequence of the finite amount of memory is that the set of real numbers signifiable on D_j is finite and, hence, vacuously countable. To put it differently, Cantor's theorem (§ 2 in [23]) does not apply insomuch as the number of signifiable reals on D_j in any interval $(\alpha \ldots \beta)$, $\alpha, \beta \in \mathbb{R}$, $\alpha < \beta$, is finite. Consequently, all computation of a feedforward artificial neural network $\mathfrak{N}_z : \mathbb{R}^n \mapsto \mathbb{R}^m$, $0 < n, m$, realized on D_j, can be packed into a unique natural number Ω_z such that there exists a primitive recursive function $\tilde{f} : \mathbb{N} \mapsto \mathbb{N}$ such that $\zeta_z(\vec{x}) = \vec{a}$ if and only if $\tilde{f}(\tilde{x}) = \tilde{a}$, where $\vec{x}$ uniquely corresponds to $\tilde{x}$ and $\vec{a}$ to $\tilde{a}$ (cf. Theorem 1, pp. 15–17 in [21]). Theorem 1 is, after a fashion, the converse of Theorem 1 in [21] in the sense that it shows how one can construct a network from a primitive recursive function.

Theorem 2 shows that all values of a primitive recursive function can be computed exactly by a feedforward artificial neural network if the network is allowed to have countably many neurons. This purely theoretical result contributes to the growing collection of universality theorems on feedforward neural networks and various classes of functions (cf. Ch. 4 in [17]). Thus, Hornik et al. [13] show that multilayer feedforward networks with a single hidden layer of neurons with arbitrary squashing activation functions can approximate any Borel measurable function from one dimensional space to another to any desired degree of accuracy so long as the number of the neurons in the hidden layer is unbounded. Gripenberg [14] shows that the general approximation property of feedforward perceptron networks is achievable when the number of perceptrons in each layer is bounded but the number of layers is allowed to grow to infinity and the perceptron activation functions are continuously differentiable and not linear. Guliyev and Ismailov [15] show that single hidden layer feedforward neural networks with the fixed weights of one and two neurons in the hidden layer approximate any continuous function on a compact subset of the real line and proceed to demonstrate that single layer feedforward networks with fixed weights cannot approximate all continuous multivariate functions.

We conclude our discussion with a caveat about universality results of feedforward neural networks with unbounded numbers of neurons. While these results provide valuable theoretical insights, they may not hold much sway with computer scientists interested in computability properties of finite AI, because networks with unbounded numbers of neurons cannot be realized on computational devices with finite amounts of computational memory.

7. Conclusions

We have formalized feedforward artificial neural networks with recurrence equations and proposed a formal definition of the concept of $\mathfrak{N}$-computability, i.e., the property of a function to be computed by a feedforward artificial neural network $\mathfrak{N}$. We have shown that, for a primitive recursive function $h(\mathbf{x}, t)$, where $\mathbf{x}$ is an n-tuple of natural numbers and t is a natural number, there exists a feedforward artificial neural network $\mathfrak{N}(\mathbf{x}, t)$ such that for any n-tuple of natural numbers $\mathbf{z}$, the first $m + 1$ terms of the sequence $\{h(\mathbf{z}, t)\}$ agree elementwise with the tuple $(\mathfrak{N}(\mathbf{z}, 0), \ldots, \mathfrak{N}(\mathbf{z}, m))$, for any positive natural number m. Our investigation contributes to the knowledge of the classes of functions that can be computed by feedforward artificial neural networks. Since such networks are used in some finite AI systems, our investigation may be of interest to mathematicians and computer scientists interested in the computability theory of finite AI.

Funding: This research received no external funding.

Data Availability Statement: No new data were created.

Acknowledgments: The author is grateful to the four anonymous reviewers for their feedback.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Gödel, K. On formally undecidable propositions of *Principia Mathematica* and related systems I. In *Kurt Gödel Collected Works Volume I Publications 1929–1936*; Feferman, S., Dawson, J.W., Kleene, S.C., Moore, G.H., Solovay, R.M., van Heijenoort, J., Eds.; Oxford University Press: Oxford, UK, 1986.
2. Péter, R. Konstruktion nichtrekursiver funktionen. *Math. Ann.* **1935**, *111*, 42–60. [CrossRef]
3. Péter, R. *Recursive Functionen*; Academinai Kiado: Budapest, Hungary, 1951.
4. Kleene, S.C. *Introduction to Metamathematics*; D. Van Nostrand: New York, NY, USA, 1952.
5. Davis, M. *Computability and Unsolvability*; Dover Publications, Inc.: New York, NY, USA, 1982.
6. Rogers, H., Jr. *Theory of Recursive Functions and Effective Computability*; The MIT Press: Cambridge, NY, USA, 1988.
7. Davis, M.; Sigal, R.; Weyuker, E. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd ed.; Harcourt, Brace & Company: Boston, MA, USA, 1994.
8. Colson, L. About primitive recursive algorithms. *Theor. Comput. Sci.* **1991**, *83*, 57–69. [CrossRef]
9. Paolini, L.; Piccolo, M.; Roversi, L. A class of recursive permutations which is primitive recursive complete. *Theor. Comput. Sci.* **2020**, *813*, 218–233. [CrossRef]
10. Petersen, U. Induction and primitive recursion in a resource conscious logic—With a new suggestion of how to assign a measure of complexity to primitive recursive functions. *Dilemmata Jahrb. ASFPG* **2008**, *3*, 49–106.
11. McCulloch, W.S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.* **1943**, *5*, 115–133. [CrossRef]
12. Rumelhart, D.E.; Hinton, G.E.; Williams, R.J. Learning representations by back-propagating errors. *Nature* **1986**, *323*, 533–536. [CrossRef]
13. Hornik, K.; Stinchcombe, M.; White, H. Multilayer feedforward networks are universal approximators. *Neural Netw.* **1989**, *5*, 359–366. . [CrossRef]
14. Gripenberg, G. Approximation by neural networks with a bounded number of nodes at each level. *J. Approx. Theory* **2003**, *122*, 260–266. https://.. [CrossRef]
15. Guliyev, N.; Ismailov, V. On the approximation by single hidden layer feedforward neural networks with fixed weights. *Neural Netw.* **2019**, *98*, 296–304. [CrossRef]
16. Zhang, Z.; Li, J. A review of artificial intelligence in embedded systems. *Micromachines* **2023**, *14*, 897. [CrossRef]
17. Nielsen, M. *Neural Networks and Deep Learning*; Determination Press: San Francisco, CA, USA, 2015.
18. Goodfellow, I.; Bengio, Y.; Courville, A. *Neural Networks*; MIT Press: Cambridge, MA, USA, 2016.
19. Meyer, M.; Ritchie, D. The complexity of loop programs. In Proceedings of the ACM National Meeting, Washington, DC, USA, 30 August 1967; pp. 465–469.
20. Taylor, A.E. *Advanced Calculus*; Ginn & Company: Boston, MA, USA, 1955.
21. Kulyukin, V.A. On correspondences between feedforward artificial neural networks on finite memory automata and classes of primitive recursive functions. *Mathematics* **2023**, *11*, 2620. [CrossRef]
22. Hopcroft, J.E.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*; Narosa Publishing Hourse: New Delhi, India, 2002.
23. Cantor, G. On a property of the class of all real algebraic numbers. *Crelle's J. Math.* **1874**, *77*, 258–262.