Descriptive Complexity for Neural Networks via Boolean Networks

Veeti Ahvonen
Tampere University, Finland

Damian Heiman
Tampere University, Finland

Antti Kuusisto
Tampere University, Finland

Abstract

We investigate the descriptive complexity of a class of neural networks with unrestricted topologies and piecewise polynomial activation functions. We consider the general scenario where the running time is unlimited and floating-point numbers are used for simulating reals. We characterize these neural networks with a rule-based logic for Boolean networks. In particular, we show that the sizes of the neural networks and the corresponding Boolean rule formulae are polynomially related. In fact, in the direction from Boolean rules to neural networks, the blow-up is only linear. We also analyze the delays in running times due to the translations. In the translation from neural networks to Boolean rules, the time delay is polylogarithmic in the neural network size and linear in time. In the converse translation, the time delay is linear in both factors. We also obtain translations between the rule-based logic for Boolean networks, the diamond-free fragment of modal substitution calculus and a class of recursive Boolean circuits where the number of input and output gates match.

2012 ACM Subject Classification Computing methodologies → Neural networks; Theory of computation → Finite Model Theory; Mathematics of computing → Numerical analysis; Computer systems organization → Parallel architectures

Keywords and phrases Descriptive complexity, neural networks, Boolean networks, floating-point arithmetic, logic

Digital Object Identifier 10.4230/LIPIcs.CSL.2024.9

Related Version *Full Version*: `https://arxiv.org/abs/2308.06277`

Funding Antti Kuusisto was supported by the Academy of Finland project Theory of computational logics, grant numbers 352419, 352420, 353027, 324435, 328987. Damian Heiman was supported by the same project, grant number 353027. Antti Kuusisto was also supported by the Academy of Finland consortium project Explaining AI via Logic (XAILOG), grant number 345612. Veeti Ahvonen was supported by the Vilho, Yrjö and Kalle Väisälä Foundation of the Finnish Academy of Science and Letters.

1 Introduction

This article investigates the descriptive complexity of neural networks, giving a logical characterization for a class of general neural networks which have the topology of directed graphs and unlimited running time. The characterization is based on *Boolean networks* [5, 14]. Boolean networks have a long history, originating from the work of Kauffman in the 1960s [10]. Current applications include a wide variety of research relating to topics varying from biology and medicine to telecommunications and beyond, see, e.g., [16, 15, 14].

Boolean networks are usually not defined via a logical syntax, but it is easy to give them one as follows. Consider the set $\mathcal{T} = \{X_1, \dots, X_k\}$ of Boolean variables. A *Boolean rule* over $\mathcal{T}$ is an expression of the form $X_i :- \varphi$ where $X_i \in \mathcal{T}$ is a *head predicate* and φ is a Boolean

formula over the syntax $\varphi ::= \top \mid X_j \mid \neg\varphi \mid \varphi \wedge \varphi$, where $X_j \in \mathcal{T}$. A *Boolean program* over $\mathcal{T}$ is then a set of Boolean rules (over $\mathcal{T}$), one rule for each X_i. Given an input $f : \mathcal{T} \to \{0,1\}$ and executing the rules in parallel, the program then produces a time-series of k-bit strings in a natural way (see the preliminaries section for the full details). An extended Boolean program over $\mathcal{T}$ is a Boolean program over some $\mathcal{S} \supseteq \mathcal{T}$ together with a *terminal clause* $X_j(0) :- b$ for each $X_j \in \mathcal{S} \setminus \mathcal{T}$, where $b \in \{\top, \bot\}$. Extended programs produce time-series just like regular programs, but they also contain *auxiliary variables* $X_j \in \mathcal{S} \setminus \mathcal{T}$ whose initial value is not part of the input but is instead given via a terminal clause (cf. the preliminaries section). The logic used in this paper, **Boolean network logic** BNL, consists of extended Boolean programs.

It turns out that BNL is also closely related to the diamond-free fragment of *modal substitution calculus* MSC used in [11] to characterize distributed message passing automata. Calling that fragment SC (for *substitution calculus*), we prove that programs of SC and BNL can be translated to each other with only a linear increase in program size. Thereby our characterization via BNL can alternatively be obtained via SC. Moreover, we also show that BNL is closely related to *self-feeding circuits*. Informally, self-feeding circuits are a class of Boolean circuits where the number of input and output gates match. Each self-feeding circuit is associated with an initializing function. An initializing function fixes the input for some subset of the set of input gates. The fixed gates are intuitively "auxiliary gates"; this is analogous to the terminal clauses for auxiliary variables in a BNL-program. To execute a self-feeding circuit, it is given an input that consists of values for the non-auxiliary input gates. With a given input, a self-feeding circuit induces a time-series of bit strings (whose length matches the number of output gates) as follows. In round zero, the bit string is obtained as a combination of the values given by the initializing function and input. In each subsequent round n, the string in round n is obtained by feeding the string from the previous round $n-1$ to the circuit. We prove that programs of BNL and self-feeding circuits can likewise be translated to each other with only a linear increase in size.

The neural network (NN) model we consider is very general. We allow unrestricted topologies for the underlying directed graphs, including loops, thereby considering the recurrent setting. The reals are modeled via floating-point numbers and the running times are unlimited. We show that for each NN, there exists a corresponding program of BNL that simulates the time series of the NN for each input, and vice versa, BNL-programs can – likewise – be simulated by NNs. Furthermore, the sizes of the NNs and BNL-programs are shown to be polynomially related.

In a bit more detail, let $S = (p, q, \beta)$ denote a floating-point system with *fraction precision* p, *exponent precision* q and *base* β (see Section 3.2 for the definitions). Let N denote the number of nodes in an NN and Δ the maximum degree of the underlying graph. Modeling activation functions via piecewise polynomial functions, let P denote the number of pieces required and Ω the maximum order of the involved polynomials. Then the following holds.

▶ **Theorem 14.** *Given a general neural network $\mathcal{N}$ for $S = (p, q, \beta)$ with N nodes, degree Δ, piece-size P and order Ω, we can construct a* BNL*-program Λ such that $\mathcal{N}$ and Λ are asynchronously equivalent in S where for $r = \max\{p, q\}$,*

1. *the size of Λ is $\mathcal{O}(N(\Delta + P\Omega^2)(r^4 + r^3\beta^2 + r\beta^4))$, and*
2. *the computation delay of Λ is $\mathcal{O}((\log(\Omega) + 1)(\log(r) + \log(\beta)) + \log(\Delta))$.*

Here (and also in the below theorem) *asynchronous equivalence* means that the modeled time series can be repeated but with a delay between *significant computation rounds*. The time delays in our results are not arbitrary but rather modest. For modeling Boolean network

logic via a general neural network, let the depth of a program refer to the maximum nesting depth of Boolean formulas appearing in rules. Our result is for NNs that use the activation function $\mathrm{ReLU}(x) = \max\{0, x\}$, but it can be generalized for other activation functions.

▶ **Theorem 15.** *Given a* BNL*-program* Λ *of size* s *and depth* d*, we can construct a general neural network* $\mathcal{N}$ *for any floating-point system* S *with at most* s *nodes, degree at most* 2*,* ReLU *activation functions and computation delay* $\mathcal{O}(d)$ *(or* $\mathcal{O}(s)$ *since* $s > d$*) such that* Λ *and* $\mathcal{N}$ *are asynchronously equivalent in binary.*

It is worth noting that in our setting, while we allow for general topologies and unlimited running times, our systems have inherently finite input spaces. In the framework of NN models, this is a well-justified assumption for a wide variety of modeling purposes. Our results stress the close relations between the size and time resources of general NNs and BNL-programs. Furthermore, as outputs we consider time series rather than a single-output framework. Indeed, it is worth noting that trivially a single Boolean function suffices to model any NN with a finite input space when limiting to single outputs only and not caring about size and time blow-ups in translations.

Related work. The closely related topic of descriptive complexity of graph neural networks (or GNNs) has been studied by Barceló et al. in [3], and by Grohe in [7], [6]. In [7], the GNNs operate via feedforward neural networks, and a natural connection between these models and the circuit complexity class TC^0 is established via logic. The feedforward model in [7] uses *rational piecewise linear approximable* activation functions, meaning that the parameters of the linear representations of activation functions are finitely representable in a binary floating-point system. In the current paper, we allow floating-point systems with an arbitrary base, which can be useful, as a change of base often allows inadmissible reals to become admissible. Moreover, our activation functions are piecewise polynomially definable, meaning that most of the widely used activation functions are directly representable in our framework, e.g., ReLU. Furthermore, practically all activation functions are reasonably approximable.

Neural networks are special kinds of distributed systems, and descriptive complexity of distributed computing was initiated in Hella et al. [8], Kuusisto [11] and Hella et al. [9] by relating distributed complexity classes to modal logics. While [8] and [9] gave characterizations to constant-time distributed complexity classes via modal logics, [11] lifted the work to general non-constant-time algorithms by characterizing finite distributed message passing automata via the modal substitution calculus MSC. This was recently lifted to circuit-based networks with identifiers in Ahvonen et al. [1], also utilizing modal substitution calculus. The logic MSC has been linked to a range of other logics. For example, in [11], it is shown to contain the μ-fragment of the modal μ-calculus, and it is easy to translate MSC into a fragment of partial fixed-point logic. Building on [11], Reiter shows in [13] that this fragment of the modal μ-calculus captures finite message passing automata in the asynchronous setting. It is worth noting here that also [3] utilizes modal logic, establishing a match between aggregate-combine graph neural networks and graded modal logic. The logics BNL and SC used in this article are rule-based systems. Rule-based logics are used widely in various applications, involving systems such as Datalog, answer-set programming (ASP) formalisms, and many others.

2 Preliminaries

First we introduce some basic concepts. For any set S, we let $\wp(S)$ denote the power set of S and we let $|S|$ denote the size (or cardinality) of S. We let $\mathbb{N}$ and $\mathbb{Z}_+$ denote the sets of non-negative and positive integers respectively. For every $n \in \mathbb{Z}_+$, we let $[n] = \{1, \dots, n\}$ and $[0; n] = \{0, \dots, n\}$. We let bold lower-case letters $\mathbf{a}, \mathbf{b}, \mathbf{c}, \dots$ denote strings. The letters of a

string are written directly next to each other, i.e. abc, or with dots in-between, i.e. $a \cdot b \cdot c$, or a mix of both, i.e. $abc \cdot def$. Omitted segments of strings are represented with three dots, i.e. $abcd \cdots wxyz$. If $\mathbf{s} = s_0 \cdots s_{k-1}$ is a string of length k, then for any $j \in [0; k-1]$, we let $\mathbf{s}(j)$ denote the letter s_j. The alphabet for the strings will depend on the context. We let $\mathrm{VAR} = \{ V_i \mid i \in \mathbb{N} \}$ denote the (countably infinite) set of all **schema variables**. Mostly, we will use meta variables X, Y, Z and so on, to denote symbols in VAR. We assume a linear order $<^{\mathrm{VAR}}$ over the set VAR. Moreover, for any set $\mathcal{T} \subseteq \mathrm{VAR}$, a linear order $<^{\mathcal{T}}$ is induced by $<^{\mathrm{VAR}}$. We let $\mathrm{PROP} = \{ p_i \mid i \in \mathbb{N} \}$ denote the (countably infinite) set of **proposition symbols** that is associated with the linear order $<^{\mathrm{PROP}}$, inducing a linear order $<^{P}$ over any subset $P \subseteq \mathrm{PROP}$. We let $\Pi \subseteq \mathrm{PROP}$ denote a finite subset of proposition symbols. When we talk about **rounds** in any context, we refer to non-negative integers that are interpreted as discrete steps of a computation.

2.1 Discrete time series

Next we consider infinite sequences of bit strings, i.e., we consider *discrete time series* of strings over the alphabet $\{0, 1\}$. To separate important strings from less important ones, we need to define when a time series produces an output; importantly, we allow an arbitrary number of outputs. We will define two separate general output conditions for time series. In the first approach, special bits indicate when to output. In the second approach the output rounds are fixed, and we do not include bits that indicate when to output.

The formal definition for the first approach is as follows. Let $k \in \mathbb{Z}_+$ and let B denote an infinite sequence $(\mathbf{b}_j)_{j \in \mathbb{N}}$ of k-bit strings $\mathbf{b}_j \in \{0, 1\}^k$. Let $A \subseteq [k]$ and $P \subseteq [k]$ be subsets of bits called **attention** bits and **print** bits respectively (or bit positions, strictly speaking). The sets A and P induce corresponding sequences $(\mathbf{a}_j)_{j \in \mathbb{N}}$ and $(\mathbf{p}_j)_{j \in \mathbb{N}}$ of substrings of the strings in B. More formally, $(\mathbf{a}_j)_{j \in \mathbb{N}}$ records the substrings with positions in A, and $(\mathbf{p}_j)_{j \in \mathbb{N}}$ records the substrings with positions in P. Next we define output conditions for B with respect to attention and print bits. If at least one bit in $\mathbf{a}_n$ is 1 (for some $n \in \mathbb{N}$), then we say that B **outputs** $\mathbf{p}_n$ in round n and that n is an **output round**. More precisely, B **outputs in round n with respect to (k, A, P)**, and $\mathbf{p}_n$ is the **output of B in round n with respect to (k, A, P)**. Let $O \subseteq \mathbb{N}$ be the set of output rounds induced by B w.r.t. (k, A, P); they induce a subsequence $(\mathbf{b}_j)_{j \in O}$ of B. We call the sequence $(\mathbf{p}_j)_{j \in O}$ the **output sequence** of B (w.r.t. (k, A, P)).

Next we define an output condition where output rounds are fixed by a set $O \subseteq \mathbb{N}$ and attention bits are excluded. We say that S **outputs** in rounds O (and also, in any particular round $n \in O$). Outputs and output sequences w.r.t. (k, O, P) are defined analogously.

We study the two approaches for the sake of generality. The difference between the two output frameworks is that the output rounds are induced internally from within the sequence in the first approach, while they are given externally from the outside in the second one. For instance, it is natural to indicate output conditions within a program if it is part of the program's design. Retroactively, it might be more natural to augment a program to draw attention to rounds the original design doesn't account for, and a different mechanism could be used to compute the output rounds, e.g., a Turing machine.

2.2 Modal substitution calculus MSC and Boolean network logic BNL

We next define modal substitution calculus MSC introduced in [11]. Let $\Pi \subseteq \mathrm{PROP}$ be a finite set of proposition symbols and $\mathcal{T} \subseteq \mathrm{VAR}$. A **terminal clause** (over $(\Pi, \mathcal{T})$) is a string of the form $V_i(0) :- \varphi$, where $V_i \in \mathcal{T}$ and φ is defined over the language $\varphi ::= \top \mid p_i \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Diamond\varphi$ where $p_i \in \Pi$ (i.e., φ is a formula of modal logic over Π). An **iteration clause** (over $(\Pi, \mathcal{T})$) is

a string of the form $V_i :- \psi$ where ψ is a $(\Pi, \mathcal{T})$**-formula of modal substitution calculus** (or MSC) defined over the language $\psi ::= \top \mid p_i \mid V_i \mid \neg\psi \mid \psi \wedge \psi \mid \Diamond\psi$ where $p_i \in \Pi$ and $V_i \in \mathcal{T}$. We also use symbols $\bot, \vee, \rightarrow$ and $\leftrightarrow$ as shorthand in the usual way. In a terminal clause $V_i(0) :- \varphi$, the symbol V_i is the **head predicate** and φ is the **body** of the clause. Analogously, for an iteration clause $V_i :- \psi$, we say that V_i is the head predicate and ψ is the body of the iteration clause.

Let $\mathcal{T} = \{X_1, \ldots, X_n\} \subseteq \mathrm{VAR}$ be a finite, nonempty set of $n \in \mathbb{Z}_+$ distinct schema variables. A $(\Pi, \mathcal{T})$-**program** of MSC is defined as a list

$$\begin{array}{ll} X_1(0) :- \varphi_1 & X_1 :- \psi_1, \\ \vdots & \vdots \\ X_n(0) :- \varphi_n & X_n :- \psi_n, \end{array}$$

where each schema variable in $\mathcal{T}$ has precisely one terminal and one iteration clause. Informally, the terminal and iteration clauses of a program can be seen as the "rules" of the program. The diamond-free fragment of MSC called **substitution calculus** SC simply restricts the terminal and iteration clauses, not allowing diamonds $\Diamond$. Note that in SC the bodies of terminal clauses are thereby formulae of propositional logic. Moreover, the programs of MSC (and SC) are also associated with a set $\mathcal{P} \subseteq \mathcal{T}$ of **print predicates** and either a set $\mathcal{A} \subseteq \mathcal{T}$ of **attention predicates** or an **attention function** $A\colon \{0,1\}^k \to \wp(\mathbb{N})$, where k is the number of distinct proposition symbols that appear in the program. Informally, the attention predicates and the attention function are analogous to the two output conditions discussed for time series. We will later discuss how either can be used to determine a set of output rounds for the program. We use attention predicates by default, and only discuss the attention function when specified.

Usually, a run of a program of MSC is defined over a Kripke-model, but a run of a $(\Pi, \mathcal{T})$-program of SC is defined over a **model** M of propositional logic, i.e., M is a valuation $\Pi \to \{0,1\}$ assigning a truth value to each proposition symbol in Π. The semantics of formulae of propositional logic in model M are defined as follows: $M \models p_i$ (read: p_i is true in M) iff the valuation of p_i is 1, and the semantics of $\wedge$, $\neg$ and $\top$ is the usual one. If $\Pi' \subseteq \Pi$ is the set of proposition symbols that appear in the program, the linear order $<^{\Pi'}$ and the model M induce a binary string $\mathbf{i} \in \{0,1\}^{|\Pi'|}$ that serves as input, i.e., the ith bit of $\mathbf{i}$ is 1 iff the valuation of the ith proposition in Π' is 1. Let Λ be a $(\Pi, \mathcal{T})$-program of MSC. The truth of a $(\Pi, \mathcal{T})$-formula ψ in round $n \in \mathbb{N}$ (written $M \models \psi^n$) is defined as follows: **1)** $M \models \top^n$ always holds, **2)** $M \models p_i^n$ iff $M \models p_i$ (where $p_i \in \Pi$), **3)** if $\psi := \neg\theta$, then $M \models (\neg\theta)^n$ iff $M \not\models \theta^n$, **4)** if $\psi := (\chi \wedge \theta)$, then $M \models (\chi \wedge \theta)^n$ iff $M \models \chi^n$ and $M \models \theta^n$, and **5)** the truth of a head predicate X_i is defined separately as follows. We define that $M \models X_i^0$ if $M \models \varphi_i$ where φ_i is the body of the terminal clause of X_i in Λ. Assume we have defined the truth of all $(\Pi, \mathcal{T})$-formulae in round n. We define that $M \models X_i^{n+1}$ iff $M \models \psi_i^n$ where ψ_i is the body of the iteration clause of X_i in Λ.

We then define **Boolean network logic** (or BNL) which we will later show to be equivalent to the fragment SC. Boolean network logic gets its name from Boolean networks, which are discrete dynamical systems commonly used in various fields, e.g., biology, telecommunications and various others. For example, they are used to describe genetic regulatory networks (e.g., [10]). A Boolean network consists of a set of Boolean variables, i.e. variables that only get Boolean values 0 or 1. Each variable is given an initial Boolean value called the "seed". The Boolean values of all variables are updated in discrete steps starting with the seed. In each step, each variable updates its Boolean value using its own Boolean function. The updated value is determined from the Boolean values of all variables in the previous step. There is no general syntax for Boolean networks, but BNL will give us a suitable one.

Let $\mathcal{T} \subseteq \mathrm{VAR}$. A **$\mathcal{T}$-formula of Boolean network logic** (or BNL) is defined over the language $\psi ::= \top \mid V_i \mid \neg\psi \mid \psi \wedge \psi$, where $V_i \in \mathcal{T}$ (i.e. we do not include propositions). Assume now that $\mathcal{T}$ is finite and nonempty. There are three main differences between $\mathcal{T}$-programs of BNL and SC: **1)** The terminal clauses of BNL are either of the form $X(0) :- \top$ or $X(0) :- \bot$. **2)** The bodies of iteration clauses of BNL are $\mathcal{T}$-formulae of BNL. **3)** Each schema variable in a BNL-program has exactly one iteration clause and either one or zero terminal clauses. We let $\mathcal{I}$ denote the predicates that do not have terminal clauses, which we call **input predicates**. For example, consider a BNL-program with the terminal clause $X(0) :- \top$ and the iteration clauses $Y :- Y \wedge X$ and $X :- \neg X$. Here Y is the sole input predicate and X acts as an auxiliary predicate. Bodies and head predicates of clauses are defined analogously to SC (and MSC). A BNL-program also includes print predicates and either attention predicates or an attention function $A\colon \{0,1\}^k \to \wp(\mathbb{N})$, where $k = |\mathcal{I}|$.

The run of a program of BNL is defined over a **model** $\mathcal{M}$, i.e. $\mathcal{M}$ is a valuation $\mathcal{I} \to \{0,1\}$. Analogously to a model of SC, any model for BNL and the set $\mathcal{I}$ induce a binary string $\mathbf{i} \in \{0,1\}^{|\mathcal{I}|}$ that serves as input. (Note that vice versa each string $\mathbf{i} \in \{0,1\}^{|\mathcal{I}|}$ induces a model with valuation $\mathcal{I} \to \{0,1\}$ such that $I_j \mapsto \mathbf{i}(j)$ if $I_0, \ldots, I_{|I|-1}$ enumerates the set $\mathcal{I}$ in the order $<^{\mathrm{VAR}}$.) The truth of a $\mathcal{T}$-formula is defined analogously to SC except for the truth value of head predicates in round 0. If $X \in \mathcal{I}$, we define that $\mathcal{M} \models X^0$ if the valuation of X in $\mathcal{M}$ is 1. If $X \notin \mathcal{I}$, then $\mathcal{M} \models X^0$ if the body of the terminal clause of X is $\top$.

Let $X_1, \ldots, X_n$ enumerate the set $\mathcal{T}$ of schema variables (in the order $<^{\mathrm{VAR}}$). Let Λ be a $\mathcal{T}$-program of SC (or BNL), and M a model for SC (or respectively $\mathcal{M}$ for BNL) that induces an input $\mathbf{i} \in \{0,1\}^k$, where k is the number of proposition symbols (or resp. the number of input predicates). Each time step (or round) $t \in \mathbb{N}$ defines a **global configuration** $g_t\colon \mathcal{T} \to \{0,1\}$. The global configuration at time step t is induced by the values of head predicates, i.e., $g_t(X_i) = 1$ iff $M \models X_i^t$ (or resp. $\mathcal{M} \models X_i^t$), for each $X_i \in \mathcal{T}$. Thus, an SC-program (or BNL-program) also induces an infinite sequence $(\mathbf{s}_t)_{t\in\mathbb{N}}$ called the **global configuration sequence** (with input $\mathbf{i}$), where $\mathbf{s}_t = g_t(X_1) \cdots g_t(X_n)$. The set of print predicates $\mathcal{P}$ corresponds to the set of print bits $\{\, i \mid X_i \in \mathcal{P} \,\}$. If the program has attention predicates $\mathcal{A}$, then $\mathcal{A}$ corresponds to the set of attention bits $\{\, i \mid X_i \in \mathcal{A} \,\}$. If the program has an attention function A, then the output rounds are given by $A(\mathbf{i})$. Therefore, analogously to the general output conditions defined for infinite sequences of bit strings, a program of SC or BNL with an input $\mathbf{i}$ also induces **output rounds** and an **output sequence** w.r.t. $(n, \mathcal{A}, \mathcal{P})$ (or resp. w.r.t. $(n, A(\mathbf{i}), \mathcal{P})$).

We say that a $(\Pi, \mathcal{T})$-program of SC and a $\mathcal{T}'$-program of BNL (or likewise, two BNL-programs) are **asynchronously equivalent** if they have the same output sequences with every input. We say that they are **globally equivalent** if they *also* have the same global configuration sequences and output rounds with each input (note that identical inputs require that $|\Pi'| = |\mathcal{I}|$, where $\Pi' \subseteq \Pi$ is the set of proposition symbols that appear in the SC-program and $\mathcal{I}$ is the set of input predicates of the BNL-program). We define the delay between two asynchronously equivalent objects x and y. Let $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ enumerate their (possibly infinite) sets of output rounds in ascending order. Assume that the cardinality of the sets of output rounds is the same and $x_n \geq y_n$ for every $n \in \mathbb{N}$. If T is the smallest amount of time steps (that might depend on x or y) such that $T \cdot y_n \geq x_n$ for every $n \in \mathbb{N}$, then we say that the **computation delay** of x is T. The case for $y_n \geq x_n$ is analogous.

The **size** of a program of SC or BNL is defined as the number of appearances of $\top$, proposition symbols p_i, head predicates V_i and logical connectives $\neg$ and $\wedge$ in its terminal and iteration clauses. The **depth** $d(\psi)$ of a BNL-formula or SC-formula is defined recursively:

1) $d(p_i) = d(\top) = d(X) = 0$, where p_i is a proposition symbol and X is a schema variable, **2)** $d(\neg\psi) = d(\psi) + 1$ and **3)** $d(\psi \wedge \theta) = \max\{d(\psi), d(\theta)\} + 1$. The **depth** of a BNL-program is the maximum depth of the bodies of its iteration clauses.

BNL-programs inherit a number of properties from Boolean networks. Each reachable combination of truth values for the head predicates (i.e., each reachable global configuration) is called a **state** and together they form a **state space**. Note that certain global configurations may not be reachable, because neither they nor their preceding states are possible states at round 0 due to the terminal clauses of the BNL-program. Given that the number of states is finite, a BNL-program will eventually either reach a single stable state or begin looping through a sequence of states. A stable state is called a **point attractor**, a **fixed-point attractor** or simply a **fixed point**, whereas a looping sequence of multiple states is a **cycle attractor**. The smallest amount of time it takes to reach an attractor from a given state is called the **transient time** of that state. The **transient time** of a BNL-program is the maximum transient time of a state in its state space [5]. The concept of transient time is also applicable to SC, since it is also deterministic and eventually stabilizes with each input.

Consider the fragment BNL_0 where no head predicate of a program is allowed to have a terminal clause. The programs of this logic BNL_0 are an exact match with Boolean networks; each program encodes a Boolean network, and vice versa. The logic BNL extends this framework by allowing terminal clauses.

A BNL-program that only has fixed points (i.e., no input leads to a cycle attractor) and outputs precisely at fixed points, is called a **halting** BNL**-program**. For a halting BNL-program Λ with input predicates $\mathcal{I}$ and print predicates $\mathcal{P}$, each input $\mathbf{i} \in \{0,1\}^{|\mathcal{I}|}$ results in a single (repeating) **output** denoted by $\Lambda(\mathbf{i})$, which is the output string determined by the fixed-point values of the print predicates. In this sense, a halting BNL-program is like a function $\Lambda\colon \{0,1\}^{|\mathcal{I}|} \to \{0,1\}^{|\mathcal{P}|}$. We say that Λ **specifies** a function $f\colon \{0,1\}^{\ell} \to \{0,1\}^{k}$ if $|\mathcal{I}| = \ell$, $|\mathcal{P}| = k$ and $\Lambda(\mathbf{i}) = f(\mathbf{i})$ for all $\mathbf{i} \in \{0,1\}^{\ell}$. The **computation time** of a halting BNL-program is its transient time.

We introduce two useful tools that are used when constructing BNL-programs (these tools are also definable via MSC or SC). Flagging is one of the most useful tools similar to adding "if-else" conditions in programming. Given two formulae φ and χ, and a rule $X :\!- \psi$, **flagging** X (w.r.t. φ and χ) means rewriting the rule $X :\!- \psi$ as $X :\!- (\varphi \wedge \psi) \vee (\neg\varphi \wedge \chi)$. Now, if φ is true then the truth value of X depends on the truth value of ψ, and if φ is false then the truth value of X depends on the truth value of χ. We call φ the **flag** and χ the **backup**. Often χ is X itself meaning that the truth value of X does not change if φ is false. Using flags, it is possible to create branches in a program, and thereby combine subprograms into a single, bigger program.

A **one-hot counter** is defined as a sequence of schema variables $T_0, T_1, \ldots, T_n$ with the terminal clauses $T_0(0) :\!- \top$ and $T_i(0) :\!- \bot$ for all $i \geq 1$, and iteration clauses $T_0 :\!- T_n$ and $T_i :\!- T_{i-1}$ for all $i \geq 1$. Exactly one of these schema variables is true in any one time step, and they turn on in a looping sequence from left to right. T_t is true in round t for all $t \leq n$. In round $n+1$, T_0 is true again and the cycle continues. This is ideal for flagging: T_n can be used as a flag for attention predicates to trigger an output round once every n time steps.

We are ready to prove that BNL is equivalent to SC.

▶ **Theorem 1.** *Each* SC*-program of transient time T has an asynchronously equivalent* BNL*-program of linear size and transient time $T+1$, and each* BNL*-program has a globally equivalent* SC*-program of linear size.*

Proof sketch. For the full proof, see [2]. From SC to BNL, we create a BNL-program that uses one time step to compute the terminal clauses of the SC-program; the terminal clauses of the SC-program are embedded into the iteration clauses of the BNL-program using a flag. From BNL to SC, we amend the BNL-program with the missing terminal clauses using proposition symbols. ◀

2.3 Link to self-feeding circuits

In this section we introduce self-feeding circuits. We will show that for every BNL-program, we can construct an equivalent self-feeding circuit and vice versa. We also pay special attention to the size and time complexities in the translations.

We first recall some basics related to circuits and then define a related self-feeding circuit model. A **Boolean circuit** is a directed, acyclic graph where at least each node of non-zero in-degree is labeled by one of the symbols $\wedge, \vee, \neg$. The nodes of a circuit are called **gates**. The in-degree of a gate u is called the **fan-in** of u, and the out-degree of u is the **fan-out**. The **input gates** of a circuit are precisely the gates that have zero fan-in and no label $\wedge, \vee$ or $\neg$. The **output gates** are the ones with fan-out zero; we allow multiple output gates in a circuit. The fan-in of every gate labeled with $\neg$ is 1.

The **size** $|C|$ of a circuit C is the number of gates in C. The **depth** $\mathsf{depth}(C)$ (or the **computation/evaluation time**) of C is the length of the longest path (number of edges) from an input gate to an output gate. The **height** $\mathsf{height}(G)$ of a gate G in C is the length of the longest path from an input gate to the gate G. The sets of input and output gates of a circuit are both linearly ordered. A circuit with n input gates and k output gates then **computes** (or specifies) a function of type $\{0,1\}^n \to \{0,1\}^k$. This is done in the natural way, analogously to the Boolean operators corresponding to $\wedge, \vee, \neg$; see, for example, [12] for the formal definition. The output of the circuit is the *binary string* determined by the bits of the output gates. Note that gates with the labels $\wedge, \vee$ can have any fan-in (also 0), meaning that by default, circuits have **unbounded** fan-in. In the elaborations below, we say a circuit is **fan-in bounded** (or the circuit has a bounded fan-in) if the fan-in of every $\wedge$-gate and $\vee$-gate of the circuit is at most 2. The $\wedge$-gates that have zero fan-in always output 1, and therefore correspond to the symbol $\top$. The $\vee$-gates that have zero fan-in always output 0 and therefore, similarly, correspond to the symbol $\bot$.

Analogously to circuits, a Boolean formula φ with n variables specifies a function of type $\{0,1\}^n \to \{0,1\}$. Let $\mathcal{B}$ denote the set of all Boolean formulas, and let $\mathcal{C}$ denote the set of all circuits. Given x and y in the set $\mathcal{B} \cup \mathcal{C}$, we say that x and y are **equivalent** if they specify the same function.

Let $k \in \mathbb{Z}_+$. A **self-feeding circuit for** k is a circuit C that specifies a function

$$f : \{0,1\}^k \to \{0,1\}^k.$$

The circuit C is associated with a set of **input positions** $I \subseteq [k]$ and an **initializing function** $\pi\colon [k] \setminus I \to \{0,1\}$. The elements of $[k] \setminus I$ are called **auxiliary positions**. Moreover, C is also associated with a set $P \subseteq [k]$ of **print positions** and either with a set $A \subseteq [k]$ of **attention positions** or an **attention function** $a\colon \{0,1\}^{|I|} \to \wp(\mathbb{N})$.

Informally, a self-feeding circuit computes as follows. The non-auxiliary input gates are fed with the input and the auxiliary input gates are fed with the values given by the initializing function; then the circuit produces an output in the ordinary way. After that in each round n the output from the previous round $n-1$ is fed back to the circuit itself to produce a new binary string. We then define the computation of self-feeding circuits formally. Let C be a self-feeding circuit for k with input positions I and input $i\colon I \to \{0,1\}$

(or the corresponding bit string $\mathbf{i} \in \{0,1\}^{|I|}$). Respectively, the function $\pi \cup i$ corresponds to the binary string $\mathbf{s}_{\pi \cup i} \in \{0,1\}^k$, where for each $j \in [k]$, if $j \in I$, then $\mathbf{s}_{\pi \cup i}(j-1) = i(j)$ and if $j \notin I$, then $\mathbf{s}_{\pi \cup i}(j-1) = \pi(j)$. Each round $n \in \mathbb{N}$ defines a **global configuration** $\mathbf{g}_n \in \{0,1\}^k$. The configuration of round 0 is the k-bit binary string $\mathbf{g}_0 = \mathbf{s}_{\pi \cup i}$. Recursively, assume we have defined $\mathbf{g}_n$. Then $\mathbf{g}_{n+1}$ is the output string of C when it is fed with the string $\mathbf{g}_n$. Now, consider the sequence $(\mathbf{g}_n)_{n \in \mathbb{N}}$ of k-bit strings that C produces. The circuit C with input i (or $\mathbf{i}$) also induces a set of **output rounds** and an **output sequence** w.r.t. (k, A, P) (or $(k, a(\mathbf{i}), P)$). Analogously to SC and BNL, we define asynchronous equivalence, global equivalence and computation delay between two self-feeding circuits or between a self-feeding circuit and a program.

We recall a well-known fact. The lemma below is related to the fact that the Boolean functions in the circuit complexity class NC^1 (with one output gate) are equivalent to the Boolean functions in the class of polynomial-size Boolean formulas.

▶ **Lemma 2** ([4]). *Given a Boolean formula of size n, there exists an equivalent circuit with bounded fan-in, one output gate, size $\mathcal{O}(n^2)$ and formula depth $\mathcal{O}(\log n)$.*

It is easy to obtain the following theorems.

▶ **Theorem 3.** *Given a* BNL*-program of size n and depth d, we can construct a globally equivalent self-feeding circuit with bounded fan-in, size $\mathcal{O}(n)$ and depth d. Moreover, we can also construct a globally equivalent self-feeding circuit with bounded fan-in, size $\mathcal{O}(n^3)$ and depth $\mathcal{O}(\log n)$.*

Proof. We prove both claims at once. Let Λ be a BNL-program of size n. We construct a globally equivalent self-feeding circuit C_Λ as follows. Let $X_1, \ldots, X_m$ enumerate the head predicates and $\psi_1, \ldots, \psi_m$ the corresponding rules of the head predicates of Λ. Let $\mathcal{I}$ denote the set of input predicates of Λ. Each rule ψ_i is transformed into a corresponding circuit C_i with bounded fan-in as follows. To prove the first claim, each C_i is obtained in a straightforward way from the tree representation of ψ_i, and therefore the size of each circuit C_i is linear in the size of ψ_i. Respectively to prove the second claim, each C_i is obtained by applying Lemma 2. We combine each circuit C_i to one circuit C_Λ such that they share the common input gates. The initializing function π is defined as follows. If $X_i \notin \mathcal{I}$, then $\pi(i) = 1$ if the rule of the terminal clause of X_i is $\top$ and respectively $\pi(i) = 0$ if the rule of the terminal clause of X_i is $\bot$. The depth of the obtained circuit C_Λ is $\mathcal{O}(\log n)$ if each circuit C_i is obtained by applying Lemma 2 and otherwise the depth is d, since combining circuits does not affect the depth. The size of C_Λ is $\mathcal{O}(n^3)$ if Lemma 2 was applied to circuits C_i since there are at most n head predicates and the size of each C_i is $\mathcal{O}(n^2)$. Otherwise the size of C is $\mathcal{O}(n)$ since each C_i was linear in the size of the corresponding rule ψ_i. The corresponding input positions, print positions and attention positions (or attention function) are straightforward to define. Clearly C_Λ is globally equivalent to Λ. ◀

▶ **Theorem 4.** *Given a self-feeding circuit C with size n, depth d and m edges, we can construct an asynchronously equivalent* BNL*-program of size $\mathcal{O}(n+m)$ and computation delay $\mathcal{O}(d)$. Moreover, if C has bounded fan-in then the size of the program is $\mathcal{O}(n)$.*

Proof. The proof is heavily based on the proof of Lemma 3 and Theorem 4 in [1], but we give a sketch of the proof. We assume that $d > 0$. The case for $d = 0$ is trivial. First we modify C so that we obtain a globally equivalent circuit C' of size $\mathcal{O}(n)$ such that the height of each output gate is $\mathcal{O}(d)$, see for example [1]. We define a one-hot counter $T_0, \ldots, T_{\mathsf{depth}(C')}$ as defined before. We define a head predicate X_G for each gate G in C' as follows. If G is an

$\wedge$-gate at height h and $Y_1, \ldots, Y_k$ are corresponding head predicates of gates that connect to G, then $X_G :- (T_h \wedge Y_1 \wedge \cdots \wedge Y_k) \vee (\neg T_h \wedge \psi_G)$, where ψ_G is X_G if G is not an output gate and otherwise ψ_G is $\bot$. Moreover, if the fan-in of G is zero, then $X_G :- (T_h \wedge \top) \vee (\neg T_h \wedge \psi_G)$. The cases for $\vee$-gates and $\neg$-gates are analogous. Intuitively, the one-hot counter is used as a flag to make sure that each X_G evaluates in the correct time. Let π be the initializing function of C'. If G is the ith input gate and G' is the ith output gate, then we define $X_G :- (T_0 \wedge X_{G'}) \vee (\neg T_0 \wedge X_G)$.

The input, print and attention predicates are the predicates corresponding to the output gates in input, print and attention positions respectively. If C' has an attention function a, then we define the attention function a' such that $a'(\mathbf{i}) = \{\, (\mathsf{depth}(C') + 1)n \mid n \in a(\mathbf{i}) \,\}$. The constructed program is clearly asynchronously equivalent to C. Moreover the computation delay is $\mathcal{O}(d)$ since it takes $\mathsf{depht}(C')$ rounds to "simulate" each round of C. The size is also clearly $\mathcal{O}(n+m)$ and $\mathcal{O}(n)$ if C is fan-in bounded. ◀

3 Arithmetic with BNL

In this section we first show how to carry out integer addition and multiplication in Boolean network logic in parallel. We then extend this demonstration to floating-point arithmetic, including floating-point polynomials and piecewise polynomial functions.

The algorithms we use for integers are mostly well known and thus some of the formal details are spared; they can be found in [2]. Informally, the idea is to split both addition and multiplication into simple steps that are executed in parallel. We will show that we can simulate integer arithmetic (respectively, floating-point arithmetic) by programs whose size is polynomial in the size of the integers (respectively, in the size of the floating-point system). We also analyze the time delays of the constructed programs. The time delay is polylogarithmic in the size of the integers (and resp. in the size of the floating-point system) and sometimes even a constant. Ultimately, the same applies to floating-point polynomials and piecewise polynomial functions.

3.1 Integer arithmetic

We next define how a *halting* BNL-program simulates integer functions in an arbitrary base $\beta \in \mathbb{Z}$, $\beta \geq 2$. Informally, we represent integers with bit strings that are split into substrings of length β, where exactly one bit in each substring is 1 and the others are 0. Formally, let $\mathbf{s}_1, \ldots, \mathbf{s}_k \in \{0,1\}^\beta$ be **one-hots**, i.e. bit strings with exactly one 1. We say that $\mathbf{s} = \mathbf{s}_1 \cdots \mathbf{s}_k$ **corresponds** to $b_1 \cdots b_k \in [0; \beta-1]^k$ if for every b_i, we have $\mathbf{s}_i(b_i) = 1$ (and other values in $\mathbf{s}_i$ are zero). For example, if $\beta = 5$, then $00100 \cdot 01000 \cdot 00001 \in \{0,1\}^{\beta \cdot 3}$ corresponds to $2 \cdot 1 \cdot 4 \in [0;4]^3$. We say that $\mathbf{s}$ is a **one-hot representation** of $b_1 \cdots b_k$.

Using the binary one-hot representations, we can present integers in BNL by assigning each bit with a head predicate that is true if and only if the bit is 1. The sign ($+$ or $-$) of a number can likewise be handled with a single bit that is true iff the sign is positive.

▶ **Definition 5.** *Let $\beta \in \mathbb{Z}$, $\beta \geq 2$, be a base. We say that a halting* BNL*-program Λ **simulates** (or computes) a function $f\colon [0;\beta-1]^\ell \to [0;\beta-1]^k$ if for each input string $\mathbf{i} \in \{0,1\}^{\ell\beta}$ that corresponds to $\mathbf{b} \in [0;\beta-1]^\ell$, the output $\Lambda(\mathbf{i})$ also corresponds to $f(\mathbf{b})$.*

We note that *comparison of two p-length integers in base β can be simulated* with a BNL-program of size $\mathcal{O}(p\beta^2 + p^2)$ and computation time 2. The one-hot representations of the numbers are first coded into input predicates. Then in round 1, auxiliary predicates determine which number has the higher digit in each position, which requires β^2 space for

each of the p positions. Finally in round 2, the attention/output predicates check that if the first number had a lower digit in some position i, then it has a greater digit in some position j to the left of i; this requires $\mathcal{O}(p^2)$ space.

Parallel addition

In this section we construct a parallel integer addition algorithm via BNL-programs. The algorithm is mostly well known and is based on how integer addition is computed in Nick's class NC^1 (sometimes called the carry-lookahead method), i.e., we parallelize the textbook method (sometimes called the long addition algorithm). Here the main difference to integer addition in Nick's class is that we generalize the algorithm for arbitrary bases.

To illustrate our method of carrying out integer addition, consider the following example of adding $\mathbf{x} = 614$ and $\mathbf{y} = 187$ in base 10. Let c_1, c_2 and c_3 denote the carry over digits and let s_1, s_2, s_3 and s_4 denote the digits of the sum $\mathbf{x} + \mathbf{y}$ from right to left. We have

$$c_1 = 1 = \lfloor (4+7)/10 \rfloor, \quad c_2 = 1 = \lfloor (1+8+c_1)/10 \rfloor, \quad c_3 = 0 = \lfloor (6+1+c_2)/10 \rfloor$$

and therefore $s_1 = 1$, $s_2 = 0$, $s_3 = 8$ and $s_4 = 0 = c_3$, since $(4+7) \equiv 1 (\bmod 10)$, $(1+8+1) \equiv 0 (\bmod 10)$ and $(6+1+1) \equiv 8 (\bmod 10)$. Therefore $\mathbf{x} + \mathbf{y} = 0801$, as wanted. As we can see, in order to know that $c_2 = 1$ we have to first check if $c_1 = 1$. In other words, we have to check if a carry from a previous position has been propagated forward.

Now we are ready to prove the following lemma.

▶ **Lemma 6.** *Given a base $\beta \in \mathbb{Z}$, $\beta \geq 2$ and $p \in \mathbb{Z}_+$, adding two numbers in $[0; \beta-1]^p$ can be simulated with a (halting)* BNL*-program of size $\mathcal{O}(p^3 + p\beta^2)$ and computation time $\mathcal{O}(1)$.*

Proof. We start with an informal description. Consider the addition of two p-digit integers $\mathbf{x} = x_p \cdots x_1$ and $\mathbf{y} = y_p \cdots y_1$ in a base $\beta \geq 2$ (we also allow leading zeros). We assume that the signs of both $\mathbf{x}$ and $\mathbf{y}$ are positive since this can be easily generalized for arbitrary signs. For $i \in [p]$, we let $c_1 = \left\lfloor \frac{x_1 + y_1}{\beta} \right\rfloor$ and $c_{i+1} = \left\lfloor \frac{x_{i+1} + y_{i+1} + c_i}{\beta} \right\rfloor$ denote the *carry digits*. We let $\mathbf{s}$ denote the result of the addition, that is, $\mathbf{x} + \mathbf{y} = (x_p \cdots x_1) + (y_p \cdots y_1) = s_{p+1} \cdots s_1 =: \mathbf{s}$, where for $j \in [p]$, $x_j + y_j + c_{j-1} \equiv s_j \bmod \beta$ (if $j = 1$, then c_{j-1} is omitted), and $s_{p+1} = c_p$. The hard part is to compute the carry digits c_i. We note that c_i is 1 if and only if the sum of x_i, y_i and c_{i-1} is at least β. The problem is that the sum of x_i and y_i might be less than β. Therefore, we have to also check if c_{i-1} is 1. To compute c_{i-1} we have to check if the sum of x_{i-1}, y_{i-1} and c_{i-2} is at least β and so on. So in order to compute c_i, we might have to check all previous carry digits.

So, in the worst case for c_i there are $\mathcal{O}(p)$ possibilities where adding x_j and y_j $(j < i)$ might lead the carry digit c_j to become 1 and in the worst case there are $\mathcal{O}(p)$ digits between j and i for whom we need to check if they carry c_j further. Since we are going to use one-hot representations this requires $\mathcal{O}(p^3 + p\beta^2)$ space in total but it can be done in $\mathcal{O}(1)$ time steps, as we will show next.

We will write a BNL-program of size $\mathcal{O}(p^3 + p\beta^2)$ that computes the sum of two integers (where β is the base and p is the length of the integers) in $\mathcal{O}(1)$ steps. We assume that integers $\mathbf{x} = x_p \cdots x_1$ and $\mathbf{y} = y_p \cdots y_1$ in $[0; \beta-1]^p$ (where we allow leading zeros) are encoded to variables $X_{j,m}$ and $Y_{j,m}$, where $j \in [p]$ and $m \in [0; \beta-1]$, using one-hot representation. For example, consider the integer 13. It can be represented in base 10 with the following variables: $Z_{1,0}, \ldots, Z_{1,9}$ and $Z_{2,0}, \ldots, Z_{2,9}$, where precisely $Z_{2,1}$ and $Z_{1,3}$ are true and the others are false. For $i \in [p]$ and $b \in \{0, 1\}$, we have

$$O_{i,b} :- \underbrace{\bigvee_{\lfloor (n+m+b)/\beta \rfloor = 1} (X_{i,n} \wedge Y_{i,m})}_{\mathcal{O}(\beta)}, \qquad C_i :- \underbrace{\bigvee_{1 \le j \le i} \left(O_{j,0} \wedge \bigwedge_{j<k<i} \left(O_{k,1} \right) \right)}_{\mathcal{O}(p^2)}.$$

The predicates $O_{i,0}$ determine whether the sum of the digits in position i will result in a carry-over digit. The predicates $O_{i,1}$ determine whether the sum of the digits in position i will result in a carry-over digit *presuming that the sum of digits in position* $i-1$ has resulted in a carry-over digit. Finally, the predicates C_i determine whether a carry-over digit is created in position i taking into account the whole sum.

Therefore we can write rules for variables $S_{i,k}$ ($i \in [p+1]$ and $k \in [0; \beta - 1]$) that will represent the sum of $\mathbf{x}$ and $\mathbf{y}$ in one-hot representation. For $i = 1$ and $k \in [0; \beta - 1]$, we have

$$S_{1,k} :- \underbrace{\bigvee_{n+m \equiv k \pmod{\beta}} (X_{1,n} \wedge Y_{1,m})}_{\mathcal{O}(\beta)}$$

and for $i \in \{2, \ldots, p\}$, we have

$$S_{i,k} :- \underbrace{\bigvee_{n+m \equiv k \pmod{\beta}} (X_{i,n} \wedge Y_{i,m} \wedge \neg C_{i-1})}_{\mathcal{O}(\beta)} \vee \underbrace{\bigvee_{n+m+1 \equiv k \pmod{\beta}} (X_{i,n} \wedge Y_{i,m} \wedge C_{i-1})}_{\mathcal{O}(\beta)}.$$

For $i = p+1$, we have $S_{p+1,0} :- \neg C_p$, $S_{p+1,1} :- C_p$, and $S_{p+1,k} :- \bot$ for every $k \in [\beta - 1]$. After three iteration rounds, the values of predicates $S_{i,k}$ have been computed. The program could be timed by using one-hot counters and flags correctly to avoid unwanted values for the predicates in steps one and two, but this is trivial to add and does not affect the size and time complexity. We have $\mathcal{O}(p)$ predicates $O_{i,b}$ of size $\mathcal{O}(\beta)$, $\mathcal{O}(p)$ predicates C_i of size $\mathcal{O}(p^2)$ and $\mathcal{O}(p\beta)$ predicates $S_{i,k}$ of size $\mathcal{O}(\beta)$. The total size of the program is thus $\mathcal{O}(p\beta + p^3 + p\beta^2) = \mathcal{O}(p^3 + p\beta^2)$.

If $\mathbf{x}$ and $\mathbf{y}$ both have negative signs, we can use the same addition algorithm; the output simply includes a negative sign. If $\mathbf{x}$ and $\mathbf{y}$ have opposite signs, we need to use a subtraction algorithm instead. First, we need to compare $\mathbf{x}$ and $\mathbf{y}$ with signs omitted; in other words, we compare which number has a greater absolute value (this also determines the sign of the output). As stated before, this requires $\mathcal{O}(p\beta^2 + p^2)$ space and 2 iteration rounds. Then, we modify the addition algorithm above in the following way. Instead of adding digits together, we subtract them; the digits of the number with the smaller absolute value are subtracted from the digits of the number with the greater absolute value. If the subtraction of two digits goes below 0, it results in a negative carry -1. Otherwise the algorithm works in the same exact way, and thus adds nothing to the size and time complexity of the program. ◀

Parallel multiplication

In this section we introduce a parallel multiplication algorithm. The parallelization method is mostly well known and is based on splitting the multiplication into simple addition tasks.

▶ **Lemma 7.** *Given a base $\beta \in \mathbb{Z}$, $\beta \ge 2$, multiplication of any two numbers in $[0; \beta - 1]^p$ can be simulated with a (halting)* BNL*-program of size $\mathcal{O}(p^4 + p^3\beta^2 + p\beta^4)$ and computation time $\mathcal{O}(\log(p) + \log(\beta))$.*

Proof sketch. The formal explanations and examples are in [2]. Assume that we have two p-digit integers (we allow leading zeros, i.e. the leftmost digits can be zeros): a multiplicand $\mathbf{x}$ and a multiplier $\mathbf{y} = y_p \cdots y_1$ in an arbitrary base $\beta \in \mathbb{Z}$, $\beta \geq 2$. The parallel multiplication algorithm computes in the following two steps. **(1)** We run p different multiplications in parallel where the multiplicand $\mathbf{x}$ is multiplied by $y_i 0 \cdots 0$ with $i-1$ zeros on the right (for each $i \in [p]$ in base β). Each multiplication is actually also computed in parallel by using the parallel addition algorithm to obtain relatively small space and time complexities. As a result we obtain p different numbers of length $2p$. **(2)** We add the numbers obtained in the first step together in parallel using the parallel addition algorithm. ◀

3.2 Floating-point arithmetic

In this section we consider floating-point arithmetic, including polynomials and piecewise polynomial functions. We show that BNL-programs can simulate these in polynomial space and in polylogarithmic time, and some simple arithmetic operations even in constant time.

Floating-point system

A **floating-point number** in a system $S = (p, q, \beta)$ (where $p, q, \beta \in \mathbb{Z}_+$, $\beta \geq 2$) is a number that can be represented in the form $\pm 0.d_1 d_2 \cdots d_p \times \beta^{\pm e_1 \cdots e_q}$, where $d_i, e_i \in [0; \beta - 1]$. For such a number in system S, we call $f = 0.d_1 d_2 \ldots d_p$ the **fraction**, the dot between 0 and d_1 the **radix point**, p the **fraction precision**, $e = \pm e_1 \cdots e_q$ the **exponent**, q the **exponent precision** and β the **base** (or **radix**).

A floating-point number in a system S may have many different representations such as 0.10×10^1 and 0.01×10^2 which are both representations of the number 1. To ensure that our calculations are well defined, we desire a single form for all non-zero numbers. We say that a floating-point number (or more specifically, a floating-point representation) is **normalized**, if **1)** $d_1 \neq 0$, or **2)** $f = 0$, e is the smallest possible value and the sign of the fraction is $+$.

For a floating-point system $S = (p, q, \beta)$, we define an extended system of **raw floating-point numbers** $S^+(p', q')$ (where $p' \geq p$ and $q' \geq q$) that possess a representation of the form $\pm d_0.d_1 d_2 \cdots d_{p'} \times \beta^{\pm e_1 \cdots e_{q'}}$. When performing floating-point arithmetic, the precise outcomes of the calculations may be raw numbers, i.e., no longer in the same system as the operands strictly speaking. Therefore, in practical scenarios, we have $p' = \mathcal{O}(p)$ and $q' = \mathcal{O}(q)$. Consider, e.g., the numbers 99 and 2 which are both in the system $S = (2, 1, 10)$, but their sum 101 is not, because 3 digits are required to represent the fraction precisely. For this purpose, we must round numbers.

The easiest way to round a number is **truncation**, where the least significant digits of the number are simply omitted, rounding the number toward zero. On the other hand, the most common method is to round to the nearest floating-point number, with ties rounding to the number with an even least significant digit. This is called **round-to-nearest ties-to-even**.

Representing floating-point numbers in binary

Our way of representing floating-points of arbitrary base in binary is based on international standards (e.g. IEEE 754). Informally, if $\mathbf{b}$ represents a floating-point number in a system $S = (p, q, \beta)$, then the first two bits encode the signs of the exponent and fraction. The next $q\beta$ bits encode the exponent in base β, and the last $p\beta$ bits encode the fraction in base β.

Before we go into the details, we have to define simulation of functions that compute with floating-point numbers in a system $S = (p, q, \beta)$. Let $F = \pm f \times \beta^{\pm e}$ be a floating-point number in system S. Let $\mathbf{p}_1, \mathbf{p}_2 \in \{0, 1\}$ and $\mathbf{s}_1, \ldots, \mathbf{s}_q, \mathbf{s}'_1, \ldots, \mathbf{s}'_p \in \{0, 1\}^\beta$. We say that

$\mathbf{s} = \mathbf{p}_1\mathbf{p}_2\mathbf{s}_1 \cdots \mathbf{s}_q\mathbf{s}'_1 \cdots \mathbf{s}'_p$ **corresponds** to F (or $\mathbf{s}$ is a **one-hot representation** of F) if **(1)** $\mathbf{p}_1 = 1$ iff the sign of the exponent is $+$, **(2)** $\mathbf{p}_2 = 1$ iff the sign of the fraction is $+$, **(3)** $\mathbf{s}_1 \cdots \mathbf{s}_q$ corresponds to $e = e_1 \cdots e_q$, and **(4)** $\mathbf{s}'_1 \cdots \mathbf{s}'_p$ corresponds to $f = 0.d_1d_2 \cdots d_p$ (or, more precisely, to $d_1 \cdots d_p$). Correspondence is defined analogously for raw floating-point numbers; we simply replace p and q with p' and q', and add one more bit string $\mathbf{s}_0 \in \{0,1\}^\beta$ that must correspond to the digit d_0 to the left of the radix point. Likewise, we say that a bit string $\mathbf{s}$ **corresponds** to a sequence $(F_1, \ldots, F_k)$ of floating-point numbers if $\mathbf{s}$ is the concatenation of the bit strings that correspond to $F_1, \ldots, F_k$ from left to right. For example, in the system $S = (4, 3, 3)$ the number $-0.2001 \times 3^{+120}$ has the corresponding string

$$\underbrace{1}_{\mathbf{p}_1} \cdot \underbrace{0}_{\mathbf{p}_2} \cdot \underbrace{010 \cdot 001 \cdot 100}_{\mathbf{s}_1\mathbf{s}_2\mathbf{s}_3} \underbrace{001 \cdot 100 \cdot 100 \cdot 010}_{\mathbf{s}'_1\mathbf{s}'_2\mathbf{s}'_3\mathbf{s}'_4}.$$

▶ **Definition 8.** *Let $S = (p, q, \beta)$ be a floating-point system, and let $S^+ = (p', q')$ be a raw floating-point system. We say that a halting* BNL*-program Λ **simulates** a function $f\colon S^\ell \to S^k$ (or respectively $f\colon (S^+)^\ell \to S^k$) if the output $\Lambda(\mathbf{i}_1 \cdots \mathbf{i}_\ell)$ corresponds to $f(F_1, \ldots, F_\ell)$ for any $F_1, \ldots, F_\ell \in S$ (or resp. $F_1, \ldots, F_\ell \in S^+$) and the corresponding inputs $\mathbf{i}_1, \ldots, \mathbf{i}_\ell \in \{0,1\}^{2+\beta(p+q)}$ (or resp. $\mathbf{i}_1, \ldots, \mathbf{i}_\ell \in \{0,1\}^{2+\beta(p'+1+q')}$).*

Later when we construct programs for the floating-point operations, e.g. normalization, we will use a tool called **shifting**, which means moving each digit of a fraction to the left or right by one (e.g. shifting a fraction 0.012 once to the left leads to 0.120)

Consider a raw floating-point number $\pm 0.d_1d_2 \cdots d_{p'} \times \beta^{\pm e_1 \cdots e_{q'}}$ where $d_1 \neq 0$, which we seek to round to the system $S = (p, q, \beta)$ (where $p \leq p'$ and $q \leq q'$) using round-to-nearest ties-to-even. First, we check whether $e_1 \cdots e_{q'-q} = 0 \cdots 0$; if not, then we have exceeded the maximum exponent and output the highest or lowest possible number depending on the sign of the fraction. If yes, we set $e' = e_{q'-q+1} \cdots e_{q'}$. Next, we check the value of d_{p+1}. If $d_{p+1} < \frac{\beta}{2}$, then we let $f' = d_1 \cdots d_p$. If $d_{p+1} > \frac{\beta}{2}$, then we let $f' = d_1 \cdots d_p + 0^{p-1}1$ using integer addition. (We round to the nearest number in both cases.) If $d_{p+1} = \frac{\beta}{2}$, then we let $f' = d_1 \cdots d_p$ if d_p is even and $f' = d_1 \cdots d_p + 0 \cdots 01$ if d_p is odd. (In other words, in the case of a tie we round to the nearest number whose rightmost digit is even.) Finally, if f' has precision $p + 1$, then we must shift the fraction to the right and round again; otherwise we output $\pm 0.f' \times \beta^{\pm e'}$ where the signs are the same as before rounding. A BNL-program that carries out the rounding clearly takes as much space as integer addition for the fractions, i.e., $\mathcal{O}(p^3 + p\beta^2)$. Instead of integer addition, we could use a different method using carries that would result in size $\mathcal{O}(p^2\beta)$, but this does not affect our other results.

Normalizing a floating-point number

We informally describe how the normalization of raw floating-point numbers can be done. By normalization we mean that a raw floating-point number is normalized as described above.

▶ **Lemma 9.** *Let $S = (p, q, \beta)$ be a floating-point system. Normalization of a raw floating-point number in $S^+(p', q')$ to the floating-point system S, where $p' = \mathcal{O}(p)$ and $q' = \mathcal{O}(q)$, can be simulated with a (halting)* BNL*-program of size $\mathcal{O}(r^3 + r^2\beta^2)$ and computation time $\mathcal{O}(1)$, where $r = \max\{p, q\}$.*

Proof sketch. The full proof can be found in [2]. Let $S = (p, q, \beta)$ be a floating-point system. The normalization of a raw floating-point number $f \times \beta^e$ (we do not write down the signs here) in system $S^+(p', q')$ to the system S, where $p' = \mathcal{O}(p)$ and $q' = \mathcal{O}(q)$ can be split into the following cases.

1. If $f = 0$, we only set e to the smallest possible value and the sign of the fraction to $+$.
2. If $0 < |f| < 1$, then we can calculate in a few steps how much we have to shift the fraction to the left (and decrease the exponent).
3. If $|f| \geq 1$, we shift the fraction to the right by one (and decrease the exponent by one) and, after that, round the number to match fraction precision p. The rounding might lead to a non-normalized floating-point number, but we only have to shift the number to the right again at most once (because after rounding, $|f| \leq 1$).

The hard part is to keep the time complexity as low as possible. We do not go into the details here (full proofs are in [2]), but the main idea is to apply parallel integer addition specified in Section 3.1. ◀

Addition of floating-point numbers

In this section we show that we can simulate floating-point addition via BNL-programs, which is possible even in constant time.

▶ **Lemma 10.** *Addition of two (normalized) floating-point numbers in $S = (p, q, \beta)$ can be simulated with a (halting)* BNL*-program of size $\mathcal{O}(r^3 + r^2\beta^2)$ and computation time $\mathcal{O}(1)$, where $r = \max\{p, q\}$.*

Proof sketch. The full proof can be found in [2]. In the parts where we add or normalize numbers, we apply the results obtained in earlier sections. The addition is done in the following steps. **(1)** We compare which of the exponents is greater and store it. **(2)** We determine the difference d between the exponents. If d is greater than the length of the fractions, we are done and output the number with the greater exponent. If d is smaller than the length of the fractions, then we shift the fraction of the number with the smaller exponent to the right d times. We then perform integer addition on the fractions and store the result. **(3)** We obtain a number whose exponent was obtained in the first step and whose fraction was obtained in the second step. We normalize this number. ◀

Multiplication of floating-point numbers

In this section we show that we can simulate floating-point multiplication via BNL-programs. The multiplication requires logarithmic time, since the proof applies the result obtained for integer multiplication in Lemma 7.

▶ **Lemma 11.** *Multiplication of two (normalized) floating-point numbers in $S = (p, q, \beta)$ can be simulated with a (halting)* BNL*-program of size $\mathcal{O}(r^4 + r^3\beta^2 + r\beta^4)$ and computation time $\mathcal{O}(\log(r) + \log(\beta))$, where $r = \max\{p, q\}$.*

Proof sketch. The full proof can be found in [2]. Informally, we do the following.

1. We add the exponents together by using the parallel (integer) addition algorithm and store the result. If the result is less than the maximum exponent, we move to the next step. Otherwise, we are done and output the largest possible number, i.e. the number with the highest possible fraction and exponent in the system.
2. We multiply the fractions using the integer multiplication algorithm and store the product.
3. We obtain a number whose exponent was obtained in the first step and whose fraction was obtained in the second step. We normalize this number.

Applying the results of parallel (integer) addition, parallel (integer) multiplication, and normalization described in the previous sections, we obtain the wanted results. ◀

Floating-point polynomials and piecewise polynomial functions

Next we consider floating-point polynomials and activation functions that are piecewise polynomial. A **piecewise polynomial** function (with a single variable) is defined as separate polynomials over certain intervals of real numbers. For instance, the function "$f(x) = x^2$ when $x \geq 0$ and $f(x) = -x$ when $x < 0$" is piecewise polynomial; the intervals are the sets of non-negative and negative numbers and the attached polynomials are x^2 and $-x$. In a floating-point system, a piecewise polynomial function is an approximation, much like addition and multiplication. We perform approximations after each addition and multiplication; as a result, the calculations must be performed in some canonical order because the order of approximations will influence the result. By the number of pieces, we refer to the number of intervals that the piecewise polynomial function is defined over; our example above has 2 pieces. We obtain the following theorem.

▶ **Theorem 12.** *Assume we have a piecewise polynomial function $\alpha\colon S \to S$, where each polynomial is of the form $a_n x^n + \cdots + a_1 x + a_0$ where $n \in \mathbb{N}$, $a_i \in S = (p, q, \beta)$ for each $0 \leq i \leq n$ and $r = \max\{p, q\}$ (addition and multiplication approximated in S). Let Ω be the highest order of the polynomials (or 1 if the highest order is 0) and let $P \in \mathbb{Z}_+$ be the number of pieces. We can construct a* BNL*-program Λ that simulates $\alpha(x)$ such that*
1. *the size of Λ is $\mathcal{O}(P\Omega^2(r^4 + r^3\beta^2 + r\beta^4))$, and*
2. *the computation time of Λ is $\mathcal{O}((\log(\Omega) + 1)(\log(r) + \log(\beta)))$.*

Proof sketch. The full proof can be found in [2]. We obtain BNL-programs that simulate the polynomials in polynomial space and polylogarithmic time. When calculating a floating-point polynomial $a_n x^n + \cdots + a_1 x + a_0$, the order of calculations is as follows: Multiplications are handled first. When carrying out the multiplication $x_1 \cdot x_2 \cdot \ldots \cdot x_k$, we simultaneously calculate the products $y_1 = x_1 \cdot x_2$, $y_2 = x_3 \cdot x_4$, etc. (If k is an odd number, the multiplicand x_k has no pair. In this case we define $y_{(k+1)/2} = x_k$.) Then, in similar fashion we calculate the products $z_1 = y_1 \cdot y_2$, $z_2 = y_3 \cdot y_4$, etc. We continue this until we have calculated the whole product. After multiplications, we handle the sums in identical fashion. We obtain the wanted results by simulating the additions and multiplications of each polynomial as described in Lemmas 10 and 11. ◀

4 Descriptive complexity for general neural networks

In this section, we establish connections between Boolean network logic and neural networks. Informally, we define a general neural network as a weighted directed graph (with any topology) operating on floating-point numbers in some system S. Each node receives either a fixed initial value or an input as its first activation value. In each communication round a node sends its activation value to its neighbours and calculates a new activation value as follows. Each node multiplies the activation values of its neighbours with associated weights, adds them together with a node-specific bias and feeds the result into a node-specific activation function. Note that floating-point systems are bounded, and the input space of a neural network is thus finite.

Before specifying neural networks formally, we introduce some concepts for infinite sequences of floating-point numbers analogous to infinite sequences of bit strings. Let $k \in \mathbb{Z}_+$ and let $S = (p, q, \beta)$ be a floating-point system. Let F denote an infinite sequence $(\mathbf{f}_j)_{j\in\mathbb{N}}$ of k-floating-point strings $\mathbf{f}_j \in S^k$. Let $A \subseteq [k]$ and $P \subseteq [k]$ be subsets of positions called **attention** positions and **print** positions respectively. The sets A and P induce corresponding sequences $(\mathbf{a}_j)_{j\in\mathbb{N}}$ and $(\mathbf{p}_j)_{j\in\mathbb{N}}$ of substrings of the strings in F. More formally, $(\mathbf{a}_j)_{j\in\mathbb{N}}$

records the substrings with positions in A, and $(\mathbf{p}_j)_{j\in\mathbb{N}}$ records the substrings with positions in P. Next we define output conditions for F with respect to attention and print positions. Let $\mathbf{t} \in S^{|A|}$ denote a set of thresholds for each attention position. If at least one floating-point number in $\mathbf{a}_n$ exceeds the threshold in the same position in $\mathbf{t}$ (for some $n \in \mathbb{N}$), then we say that $\mathcal{F}$ **outputs** $\mathbf{p}_n$ in round n and that n is an **output round**. More precisely, F **outputs in round** n **with respect to** $(k, A, P, \mathbf{t})$, and $\mathbf{p}_n$ is the **output of** F **in round** n **with respect to** $(k, A, P, \mathbf{t})$. Let $O \subseteq \mathbb{N}$ be the set of output rounds; they induce a subsequence $(\mathbf{f}_j)_{j\in O}$ of F. We call the sequence $(\mathbf{p}_j)_{j\in O}$ the **output sequence** of F.

Next we define an output condition where output rounds are fixed by a set $O \subseteq \mathbb{N}$ and attention bits are excluded along with thresholds. We say that S **outputs** in rounds O (and also, in any particular round $n \in O$). Outputs and output sequences w.r.t. (k, O, P) are defined analogously.

4.1 General neural networks

Next we define neural networks formally. A **(directed) graph** is a tuple (V, E), where V is a finite set of **nodes** and $E \subseteq V \times V$ is a set of **edges**. Note that we allow self-loops on graphs, i.e. edges $(v, v) \in E$. A **general neural network** $\mathcal{N}$ (for floating-point system S) is defined as a tuple $(G, \mathfrak{a}, \mathfrak{b}, \mathfrak{w}, \pi)$, where $G = (V, E, <^V)$ is a directed graph associated with a linear order $<^V$ for nodes in V. The network $\mathcal{N}$ contains sets $I, O \subseteq V$ of **input** and **output** nodes respectively, and a set $H = V \setminus (I \cup O)$ of **hidden nodes**. The tuples $\mathfrak{a} = (\alpha_v)_{v\in V}$ and $\mathfrak{b} = (b_v)_{v\in V}$ are assignments of a piecewise polynomial **activation function** $\alpha_v \colon S \to S$ and a **bias** $b_v \in S$ for each node. Likewise, $\mathfrak{w} = (w_e)_{e\in E}$ is an assignment of a **weight** $w_e \in S$ for each edge. The function $\pi \colon (V \setminus I) \to S$ assigns an initial value to each non-input node.

The computation of a general neural network is defined with a given input function $i \colon I \to S$. Similar to BNL-programs, an input function i also induces a floating-point string $\mathbf{i} \in S^{|I|}$, and respectively a floating-point string induces an input function. **The state of the network at time** t is a function $g_t \colon V \to S$, which is defined recursively as follows. For $t = 0$, we have $g_0(v) = i(v)$ for input nodes and $g_0(v) = \pi(v)$ for non-input nodes. Now assume we have defined the state at time t. The state at time $t + 1$ is defined as follows:

$$g_{t+1}(v) = \alpha_v\Big(b_v + \sum_{(u,v)\in E} \big(g_t(u) \cdot w_{(u,v)}\big)\Big).$$

More specifically, the sum is unfolded from left to right according to the order $<^V$ of the nodes $u \in V$. For each piece of an activation function, we assume a normal form $a_n x^n + \cdots + a_1 x + a_0$, which designates the order of operations (as in the proof sketch of Theorem 12). If we designate that $u_1, \ldots, u_k$ enumerate the set O of output nodes in the order $<^V$, then the state of the system induces an output tuple $o_t = (g_t(u_1), \ldots, g_t(u_k))$ at time t for all $t \geq 0$.

We once again define two frameworks for designating output rounds, one machine-internal and one machine-external framework. In the first framework, the set V contains a set A of **attention nodes** u, each of which is associated with a *threshold* $s_u \in S$; the order of the nodes induces a threshold string $\mathbf{t} \in S^{|A|}$. In the second framework, attention nodes are excluded and the neural network is associated instead with an **attention function** $a \colon S^{|I|} \to \wp(\mathbb{N})$.

Next we define how the output rounds and output sequence are obtained. Let $v_1, \ldots, v_n$ enumerate the nodes of the neural network (in the order $<^V$). Let $i \colon I \to S$ be an input function that induces an input $\mathbf{i} \in S^{|I|}$. A neural network induces an infinite sequence $(\mathbf{s}_t)_{t\in\mathbb{N}}$ called the **network state sequence** (with input $\mathbf{i}$), where $\mathbf{s}_t = g_t(v_1) \cdots g_t(v_n)$. The set of

output nodes O corresponds to the set of print positions $\{\, i \mid v_i \in O \,\}$. If the network has attention nodes A, then A corresponds to the set of attention positions $\{\, i \mid v_i \in A \,\}$. If the network has an attention function a, then the output rounds are given by $a(\mathbf{i})$. Therefore, analogously to the general output conditions defined for infinite sequences of floating-point strings, a neural network with an input $\mathbf{i}$ also induces **output rounds** and an **output sequence** w.r.t. $(n, A, O, \mathbf{t})$ (or resp. w.r.t. $(n, a(\mathbf{i}), O)$).

We then define some parameters that will be important when describing how neural networks and BNL-programs are related in terms of space and time complexity. The in-degree of a node v is the number of nodes u such that there is an edge $(u, v) \in E$; we say that u is a **neighbour** of v. Note that we allow reflexive loops so a node might be its own "neighbour". The **degree** of a general neural network $\mathcal{N}$ is the maximum in-degree of the underlying graph. The **piece-size** of $\mathcal{N}$ is the maximum number of "pieces" across all its piecewise polynomial activation functions. The **order** of $\mathcal{N}$ is the highest order of a "piece" of its piecewise polynomial activation functions.

A general neural network can easily emulate typical *feedforward neural networks*. This requires that the graph of the general neural network is connected and acyclic, the sets I, O and H are chosen correctly and the graph topology is as required, with all paths from an input node to an output node being of the same length. Unlike in a classical feedforward neural network, the hidden and output nodes of a general neural network have an initial value, but they are erased as the calculations flow through the network, so this is an inconsequential, essentially syntactic phenomenon. The inputs are also erased in the same way, likewise an inconsequential syntactic phenomenon. Finally, there is a round t where the general neural network outputs the same values as a corresponding feedforward network would.

In general, our neural network models are *recurrent* in the sense that they allow loops. They are *one-to-many* networks, in other words, they can map each input to a sequence of outputs unlike feedforward neural networks which always map each input to a single output.

4.2 Equivalence and time series problems

In order to translate neural networks to BNL-programs and vice versa, we define time series problems for both floating-point numbers and binary numbers, and two types of corresponding equivalence relations. The reason for this is obvious, as BNL-programs operate with binary numbers and neural networks with floating-point numbers. Informally, in the below asynchronous equivalence means that the modeled time series can be repeated but with a delay between output rounds. The time delays in our results are not arbitrary but rather modest. Moreover, we do not fix the attention mechanism for the programs or neural networks, and our definitions work in both cases.

First we define notions for floating-points. Let $k, \ell \in \mathbb{N}$, $P \subseteq [k]$ and let $S = (p, q, \beta)$ be a floating-point system. We let $\mathcal{F}(k, P, S)$ denote the family of sequences $F = (\mathbf{f}_n)_{n\in\mathbb{N}}$ of k-strings $\mathbf{f}_n \in S^k$ of numbers in S with print position set P. A **(floating-point) time series problem $\mathfrak{P}$ for** (ℓ, k, P) **in** S is a function $\mathfrak{P}\colon S^\ell \to \mathcal{F}(k, P, S) \times \wp(\mathbb{N})$. With a given input $(F_1, \ldots, F_\ell) \in S^\ell$, $\mathfrak{P}$ gives a sequence $(\mathbf{f}_n)_{n\in\mathbb{N}} \in \mathcal{F}(k, P, S)$ and a subset $O \in \wp(\mathbb{N})$ and therefore $\mathfrak{P}$ **induces** the output sequence of $(\mathbf{f}_n)_{n\in\mathbb{N}}$ w.r.t. (k, O, P) (P induces a subsequence $(\mathbf{p}_n)_{n\in\mathbb{N}}$, and O further induces the output sequence $(\mathbf{p}_n)_{n\in O}$). Let Λ be a BNL-program with $(\beta(p+q)+2)|P|$ print predicates and $(\beta(p+q)+2)\ell$ input predicates. We say that Λ **simulates a solution** for time series problem $\mathfrak{P}$ if for every input $\mathbf{i} \in \{0,1\}^{(\beta(p+q)+2)\ell}$ corresponding to $(F_1, \ldots, F_\ell) \in S^\ell$, the output sequence of Λ with input $\mathbf{i}$ corresponds to the output sequence induced by $\mathfrak{P}(F_1, \ldots, F_\ell)$, i.e., the output strings of Λ correspond to the output strings of $\mathfrak{P}$. A neural network $\mathcal{N}$ with ℓ input nodes and $|P|$ output nodes

solves $\mathfrak{P}$ if the output sequence of $\mathcal{N}$ with input $(F_1, \dots, F_\ell)$ is the output sequence induced by $\mathfrak{P}(F_1, \dots, F_\ell)$. We say that a BNL-program Λ and a neural network $\mathcal{N}$ (for S) are **asynchronously equivalent in** S if the time series problems in S simulated by Λ are exactly the ones solved by $\mathcal{N}$.

We define notions for binaries in similar fashion. Recall that $k, \ell \in \mathbb{N}$, $P \subseteq [k]$. Similarly, let $\mathcal{S}(k, P)$ denote the family of k-bit string sequences $B = (\mathbf{b}_n)_{n\in\mathbb{N}}$ with print bit set P. A **(binary) time series problem** $\mathfrak{P}$ **for** (ℓ, k, P) is a function $\mathfrak{P}\colon \{0,1\}^\ell \to \mathcal{S}(k, P) \times \wp(\mathbb{N})$ that assigns a k-bit string sequence and a set $O \in \wp(\mathbb{N})$ of output rounds to every input $\mathbf{i} \in \{0,1\}^\ell$; together they **induce** an output sequence w.r.t. (k, O, P). We say that a BNL-program Λ with ℓ input predicates and $|P|$ print predicates **solves** $\mathfrak{P}$ if the output sequence of Λ with any input $\mathbf{i} \in \{0,1\}^\ell$ is the output sequence induced by $\mathfrak{P}(\mathbf{i})$. We say that a neural network $\mathcal{N}$ for floating point system S with ℓ input nodes and $|P|$ output nodes **simulates a solution** for binary time series problem $\mathfrak{P}$ if for every input $(F_1, \dots, F_\ell) \in S^\ell$ that represents $\mathbf{i} \in \{0,1\}^\ell$ (i.e. every F_i represents 0 or 1), the output sequence of $\mathcal{N}$ with input $(F_1, \dots, F_\ell)$ corresponds to the output sequence induced by $\mathfrak{P}(\mathbf{i})$ (in the same fashion, where every floating-point number represents 0 or 1). We say that a BNL-program Λ and a general neural network $\mathcal{N}$ are **asynchronously equivalent in binary** if the time series problems in binary simulated by $\mathcal{N}$ are exactly the ones solved by Λ.

We define the **computation delay** between two objects that are asynchronously equivalent (in binary or in floating-point system S) analogously to the computation delay defined in the preliminaries.

▶ Remark 13. Asynchronous equivalence in binary could be extended to two BNL-programs; this is consistent with the asynchronous equivalence defined in preliminaries. Therefore asynchronous equivalence in binary could also extend for SC and self-feeding circuits. We could also define equivalence between two neural networks. Informally, two neural networks are asynchronously equivalent if they solve exactly the same floating-point time series problems. It is also possible to define a weakened equivalence relation for neural networks, where a neural network simulates another neural network in "binary" as follows. Let $\mathfrak{P}$ be a floating-point time series problem for (ℓ, k, P) in $S = (p, q, \beta)$ and let $\mathfrak{P}'$ be a binary time series problem for $((\beta(p+q)+2)\ell, (\beta(p+q)+2)k, P')$, where P' is the set of bit positions which corresponds to positions in P. We say that $\mathfrak{P}'$ corresponds to $\mathfrak{P}$ if for each $\mathbf{f} \in S^\ell$ and the unique bit string $\mathbf{i} \in \{0,1\}^{(\beta(p+q)+2)\ell}$ that corresponds to $\mathbf{f}$, we have that the output sequence induced by $\mathfrak{P}'(\mathbf{i})$ corresponds to the one induced by $\mathfrak{P}(\mathbf{f})$. We say that neural networks $\mathcal{N}$ and $\mathcal{N}'$ are weakly asynchronously equivalent in S if the time series problems in S solved by $\mathcal{N}$ are exactly the ones with a corresponding binary time series problem simulated by $\mathcal{N}'$, or respectively the time series problems in S solved by $\mathcal{N}'$ are exactly the ones with a corresponding binary time series problem simulated by $\mathcal{N}$.

4.3 From NN to BNL

We provide a translation from general neural networks to Boolean network logic. The proof is based on the results obtained for floating-point arithmetic in Section 3.2.

▶ **Theorem 14.** *Given a general neural network $\mathcal{N}$ for $S = (p, q, \beta)$ with N nodes, degree Δ, piece-size P and order Ω (or $\Omega = 1$ if the order is 0), we can construct a* BNL*-program Λ such that $\mathcal{N}$ and Λ are asynchronously equivalent in S where for $r = \max\{p, q\}$,*

1. *the size of Λ is $\mathcal{O}(N(\Delta + P\Omega^2)(r^4 + r^3\beta^2 + r\beta^4))$, and*
2. *the computation delay of Λ is $\mathcal{O}((\log(\Omega) + 1)(\log(r) + \log(\beta)) + \log(\Delta))$.*

Proof. First we consider the framework where output rounds are defined by attention nodes and attention predicates. We consider the setting where output rounds are fixed as a corollary.

We use separate head predicates $S_{u,e}$, $S_{u,f}$, $E_{u,i,b}$, and $F_{u,j,b}$ ($i \in [q]$, $j \in [p]$, $b \in [0; \beta-1]$) for each node u of $\mathcal{N}$. Together, they encode the **1)** exponent sign, **2)** fraction sign, **3)** exponent and **4)** fraction of the activation values of u in one-hot representation as described in Section 3.2. These calculations are done using the arithmetic algorithms from the same section. The program can not calculate a new activation value in one step like a neural network does, as each arithmetic operation takes some time to compute. The input of a single node is a floating-point number with q digits for the exponent, p digits for the fraction, and a sign for both. Its one-hot representation therefore has $(p+q)\beta+2$ bits; exactly the number of head predicates assigned for each node. Each of these predicates receives a corresponding bit as input. For instance, if the input floating-point number of u is $-0.314 \times 10^{+01}$, then the head predicates $S_{u,e}$, $E_{u,1,0}$, $E_{u,2,1}$, $F_{u,1,3}$, $F_{u,2,1}$ and $F_{u,3,4}$ get the input 1 while all the other head predicates for u get the input 0.

After receiving these inputs, the rest of the program is built by applying the programs for floating-point addition and multiplication constructed in Section 3.2 to the aggregations and activation functions of each node in the established canonical order of operations. The calculations are timed with a one-hot counter, i.e., predicates $T_0, \ldots, T_n$ as described in Section 2.2. Here n is the worst-case number of rounds required for the algorithms to calculate an activation value for a node in the network (based on the number of neighbours, as well as the order and number of pieces of the activation function). The predicates in this counter are used to stall the head predicates for each node such that they receive the bits corresponding to the new activation values at the same time (this includes the print predicates, which are all the predicates corresponding to output nodes). The attention nodes have additional predicates that correspond to the threshold values; during rounds where the activation values have been calculated, an attention predicate turns true if this value is exceeded.

We compute additions and multiplications for each node in the network; this can be done simultaneously for each node. Each node requires at most Δ multiplications and additions in the aggregation before the use of the activation function. Multiplications can be done simultaneously and sums in parallel as described in section 3. These steps require size $\mathcal{O}(N\Delta(r^4+r^3\beta^2+r\beta^4))$ (each of the N nodes performs $\mathcal{O}(\Delta)$ multiplications/additions; the size of the multiplication is $\mathcal{O}(r^4+r^3\beta^2+r\beta^4)$ which dwarfs the addition size $\mathcal{O}(r^3+r^2\beta^2)$) and the overall time required is $\mathcal{O}(\log(r)+\log(\beta))+\mathcal{O}(\log(\Delta))$ (multiplication + addition).

After the aggregation come the activation functions. Since they are piecewise polynomial, we may apply Theorem 12, using the piece-size and order of the network. If $\Omega = 0$ we are done, so assume that $\Omega \in \mathbb{Z}_+$. Each of the N nodes calculates at most P polynomial pieces of order at most Ω, which gives us a size of $\mathcal{O}(NP\Omega^2(r^4+r^3\beta^2+r\beta^4))$. This requires only $\mathcal{O}((\log(\Omega)+1)(\log(r)+\log(\beta)))$ time. The same predicates are used for the calculation of each subsequent global configuration of the network. Timing the calculations does not increase the size and time complexity. Adding the sizes and times together, the size of the program is $\mathcal{O}(N(\Delta+P\Omega^2)(r^4+r^3\beta^2+r\beta^4))$ and computing each global configuration of $\mathcal{N}$ requires time $\mathcal{O}(\log(r)+\log(\beta))+\mathcal{O}(\log(\Delta))+\mathcal{O}((\log(\Omega)+1)(\log(r)+\log(\beta))) = \mathcal{O}((\log(\Omega)+1)(\log(r)+\log(\beta))+\log(\Delta))$; the first $\mathcal{O}(\log(r)+\log(\beta))$ is not dwarfed if $\Omega = 1$.

The case for the second framework, where output rounds are given by an attention function, is obtained as a corollary. We simply take the worst case time for calculating a new activation value with the aggregations and piecewise polynomial functions in BNL; let's say the worst case is T rounds. If $a\colon S^k \to \wp(\mathbb{N})$ is the attention function of $\mathcal{N}$, then the attention function of Λ is the function $a'\colon \{0,1\}^{k(\beta(p+q)+2)} \to \wp(\mathbb{N}), a'(\mathbf{i}) = Ta(\mathbf{i}') = \{Tn \mid n \in a(\mathbf{i}')\}$ where $\mathbf{i} \in \{0,1\}^{k(\beta(p+q)+2)}$ corresponds to $\mathbf{i}' \in S^k$. ◀

4.4 From BNL to NN

Before the formal translation from BNL-programs to general neural networks, we introduce two typical piecewise polynomial activation functions with just two pieces and order at most 1. These are the well-known rectified linear unit and the Heaviside step function. Recall that an activation function is a function $S \to S$, where S is a floating-point system. The **rectified linear unit** ReLU is defined by $\mathrm{ReLU}(x) = \max\{0, x\}$ and the **Heaviside step function** H is defined by $H(x) = 1$ if $x > 0$, and $H(x) = 0$, otherwise. It is easy to generalize our results for other activation functions.

▶ **Theorem 15.** *Given a* BNL*-program* Λ *of size* s *and depth* d*, we can construct a general neural network* $\mathcal{N}$ *for any floating-point system* S *with at most* s *nodes, degree at most* 2*,* ReLU *(or Heaviside) activation functions and computation delay* $\mathcal{O}(d)$ *(or* $\mathcal{O}(s)$ *since* $s > d$*) such that* Λ *and* $\mathcal{N}$ *are asynchronously equivalent in binary.*

Proof sketch. The full proof can be found in [2]. The aggregation each node performs on the activation values of its neighbours weakens neural networks in the sense that much of the information related to specific neighbours is lost. Due to this, a single node of a neural network can't imitate an arbitrary iteration clause where each predicate has a precise role. Instead, the program Λ is first turned into an asynchronously equivalent "fully-open" program Λ' that is described in [2]. Informally, that means each body of the iteration clauses of Λ' includes at most one logical connective. This is turned into a neural network by creating a node for each predicate of Λ'. The network only uses the floating-point numbers $-1, 0, 1, 2$, and the iteration clauses can all be calculated with ReLU or Heaviside by choosing the weights and biases appropriately. ◀

We have shown a translation from neural networks to BNL-programs and vice versa. Using the translations in succession, it is possible to transform a neural network into a weakly asynchronously equivalent neural network that only uses 1 and 0 as activation values, and either ReLU or Heaviside activation functions in every node. Generalizing our result for other activation functions is possible.

The match between BNL and neural networks provides a concrete demonstration of the obvious fact that – in some relevant sense – there is no difference between symbolic and non-symbolic approaches. Under reasonable background assumptions, non-symbolic approaches can be technically reduced to symbolic ones. More than to the differences between the symbolic and non-symbolic realms, the clear advantages of modern AI methods relate to the difference between systems based on explicit programming and systems that involve an aspect of learning not based on explicit and fully controlled programming steps.

5 Conclusion

We have shown a strong equivalence between a general class of one-to-many neural networks and Boolean network logic in terms of discrete time series. The translations are simple in both directions, with reasonable time and size blow-ups. We receive similar results for the logic SC due to Theorem 1 and self-feeding circuits due to Theorem 3. The link to self-feeding circuits is novel, since it allows us to apply circuit based methods to reason about neural networks in the recurrent setting. Interesting future directions involve investigating extensions with randomization as well as studying the effects of using alternatives to floating-point numbers, such as, for example, fixed-point arithmetic.

References

1 Veeti Ahvonen, Damian Heiman, Lauri Hella, and Antti Kuusisto. Descriptive complexity for distributed computing with circuits. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*, volume 272 of *LIPIcs*, pages 9:1–9:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.MFCS.2023.9.

2 Veeti Ahvonen, Damian Heiman, and Antti Kuusisto. Descriptive complexity for neural networks via boolean networks. *CoRR*, abs/2308.06277, 2023. doi:10.48550/arXiv.2308.06277.

3 Pablo Barceló, Egor V. Kostylev, Mikaël Monet, Jorge Pérez, Juan L. Reutter, and Juan Pablo Silva. The logical expressiveness of graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

4 Maria Luisa Bonet and Samuel R. Buss. Size-depth tradeoffs for boolean formulae. *Information Processing Letters*, 49(3):151–155, 1994. doi:10.1016/0020-0190(94)90093-0.

5 Daizhan Cheng and Hongsheng Qi. A linear representation of dynamics of boolean networks. *IEEE Transactions on Automatic Control*, 55(10):2251–2258, 2010.

6 Martin Grohe. The logic of graph neural networks. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 – July 2, 2021*, pages 1–17. IEEE, 2021.

7 Martin Grohe. The descriptive complexity of graph neural networks, 2023. arXiv:2303.04613.

8 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. In *Proceedings of the 2012 ACM Symposium on Principles of distributed computing*, pages 185–194, 2012.

9 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Comput.*, 28(1):31–53, 2015.

10 Stuart Kauffman. Homeostasis and differentiation in random genetic control networks. *Nature*, 224(5215):177–178, 1969.

11 Antti Kuusisto. Modal Logic and Distributed Message Passing Automata. In *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 452–468, 2013.

12 Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

13 Fabian Reiter. Asynchronous distributed automata: A characterization of the modal mu-fragment. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 100:1–100:14, 2017.

14 Julian D Schwab, Silke D Kühlwein, Nensi Ikonomi, Michael Kühl, and Hans A Kestler. Concepts in boolean network modeling: What do they all mean? *Computational and structural biotechnology journal*, 18:571–582, 2020.

15 Massimiliano Zanin and Alexander N Pisarchik. Boolean networks for cryptography and secure communication. *Nonlinear Science Letters B: Chaos, Fractal and Synchronization. Vol*, 1(1):27–34, 2011.

16 Ranran Zhang, Mithun Vinod Shah, Jun Yang, Susan B Nyland, Xin Liu, Jong K Yun, Réka Albert, and Thomas P Loughran Jr. Network model of survival signaling in large granular lymphocyte leukemia. *Proceedings of the National Academy of Sciences*, 105(42):16308–16313, 2008.