Descriptive complexity for neural networks via Boolean networks

Veeti Ahvonen, Damian Heiman

Tampereen yliopisto

2023

Content

Content

Descriptive complexity for neural networks via Boolean networks

This seminar is based on the research article "Descriptive complexity for neural networks via Boolean networks" by authors Veeti Ahvonen, Damian Heiman and Antti Kuusisto, which has been accepted into the 32nd EACSL Annual Conference on Computer Science Logic 2024.

Motivation

- Neural networks relate to artificial intelligence and machine learning. A neural network does not perform some known algorithm with a computer; rather, the computer learns to produce classifications by itself. This is achieved by observing the classifications made in a previous iteration and fine-tuning the parameters of the network until the classification is sufficiently precise. A neural network can be taught, for example, to recognize hand-written numbers or people's faces from photographs.
- Because a neural network learns the algorithm itself, sometimes even its designer is unaware of how the network performs classifications. Ergo, the explainability of neural networks is a timely research problem. Descriptive complexity theory offers tools for solving it.

Information about the research

- The purpose of the article is to open the door to the explainability of neural networks by identifying a logical language that matches a class of general neural networks. A key aspect is turning neural networks into Boolean networks that operate with Boolean values 0 and 1. All the internal computation of the neural network is reduced into binary form and interpreted in the language of a logic we call Boolean network logic (BNL).
- As results, we find that a neural network can be interpreted as a program of BNL and vice versa. In particular, it follows that the non-linear (sometimes complicated) "activation functions" of neural networks can be translated into a very simple form, though this does increase the size of the network.

Content of the seminar

- The seminar will be in two parts; the first hour is presented by Veeti Ahvonen and the second by Damian Heiman.
- We start by introducing the logic BNL_0 and its expansion BNL.
- Next, we introduce how calculations with numbers can be carried out by BNL-programs using floating-point representations of real numbers.
- Lastly, we introduce neural networks and the translations we obtain in both directions.

Content

BNL_0-programs (informally)

First an informal description. Let $\mathcal{T} = \{X, Y, Z\}$ be an ordered set ($X < Y < Z$) with three Boolean variables (i.e., variables that can only contain the values 0 or 1). We attach a formula to each variable, that use the variables from $\mathcal{T}$ as atomic formulas. Each variable is given an initial value (upper index 0), which is either 0 or 1. Each variable calculates a new truth value for itself by slotting the previous truth values of all variables into its associated formula. This can carry on indefinitely.

$$\begin{array}{l|lllll}
X :- \; Y \wedge Z, & X^0 = 0, & X^1 = 0, & X^2 = 1, & X^3 = 1, & \dots \\
Y :- \; \neg X, & Y^0 = 0, & Y^1 = 1, & Y^2 = 1, & Y^3 = 0, & \dots \\
Z :- \; X \vee Z. & Z^0 = 1, & Z^1 = 1, & Z^2 = 1, & Z^3 = 1, & \dots
\end{array}$$

The list on the left, which contains variables and formulae associated with them, is called a **program of** BNL_0 (Boolean network logic).

Input and output (informally)

Because the variables of the program are ordered, the initial values of the variables can be seen as a bit string $X^0 Y^0 Z^0 = 001$; this is called the **input** of the program. We choose a set of print predicates from the variables of the program; let us choose the variables X and Z. The **output** of the program is likewise a bit string $X^t Z^t$, which consists of the truth values of the print predicates at some time $t \in \mathbb{N}$. For instance, if we print at time $t = 2$, then the output of the program is $X^2 Z^2 = 11$.

$$\begin{array}{l|lllll}
X :- \ Y \wedge Z, & {\color{blue}X^0 = 0}, & X^1 = 0, & {\color{red}X^2 = 1}, & X^3 = 1, & \dots \\
Y :- \ \neg X, & {\color{blue}Y^0 = 0}, & Y^1 = 1, & Y^2 = 1, & Y^3 = 0, & \dots \\
Z :- \ X \vee Z. & {\color{blue}Z^0 = 1}, & Z^1 = 1, & {\color{red}Z^2 = 1}, & Z^3 = 1, & \dots
\end{array}$$

The input of the program is marked in blue and the output in red.

Predicates

Let $\mathrm{VAR} = \{ V_i \mid i \in \mathbb{N} \}$ be a countably infinite set of **schema variables** (or simply variables), and let $<^{\mathrm{VAR}}$ denote a linear order of the variables. Typically, we use metavariables $X, Y, Z, \ldots$ or $X_1, X_2, \ldots$ to denote symbols in VAR.

Each subset $\mathcal{T} \subseteq \mathrm{VAR}$ induces an ordering $<^{\mathcal{T}}$ for the elements of $\mathcal{T}$. In other words, if $X, Y \in \mathcal{T}$ and $X <^{\mathrm{VAR}} Y$, then $X <^{\mathcal{T}} Y$. For simplicity, we may denote the ordering of the set $\mathcal{T}$ with $<$, if the set $\mathcal{T}$ is clear from the context. When using subindexing, we assume that variables $X_1, \ldots, X_n$ are in the order $X_1 < X_2 < \cdots < X_n$.

BNL_0-program

Definition 1

Let $\mathcal{T} = \{X_1, \ldots, X_n\} \subseteq \mathrm{VAR}$ be a set of $n \in \mathbb{Z}_+$ distinct schema variables. A **$\mathcal{T}$-program of the logic** BNL_0 is a triple $(L, \mathcal{P}, A)$, where L is a list

$$\begin{gathered} X_1 :- \psi_1, \\ \vdots \\ X_n :- \psi_n, \end{gathered}$$

where $\psi_1, \ldots, \psi_n$ are formulae of the language

$$\psi ::= \top \mid X_i \mid \neg\psi \mid \psi \wedge \psi$$

$(X_i \in \mathcal{T})$, $\mathcal{P} \subseteq \mathcal{T}$ is a set of **print predicates** and $A\colon \{0,1\}^{|\mathcal{T}|} \to \wp(\mathbb{N})$ is an **attention function**. A single item on the list $X_i :- \psi_i$ is a **rule** of the program, where X_i is the **head predicate** and ψ_i is the **body** of the rule.

Running BNL_0-programs

Let $\Lambda = (L, \mathcal{P}, A)$ be some $\mathcal{T}$-program of the logic BNL_0, and let $\pi\colon \mathcal{T} \to \{0,1\}$ be a function. We let X^t denote the truth value of a predicate $X \in \mathcal{T}$ at time $t \in \mathbb{N}$. It is defined recursively as follows:

- At time 0, we define that $X^0 = \pi(X)$.
- Assume that the truth values of all predicates is defined at time $t \in \mathbb{N}$. At time $t+1$, we define that $X^{t+1} = 1$ if the formula ψ associated with X is true, when the truth values of the predicates appearing in ψ are interpreted at time t. (Here ψ is the formula for which the rule $X :- \psi$ appears in the program.)

The **input** of Λ is the bit string $\overline{X}^0 = X_1^0 \cdots X_n^0$, where $\mathcal{T} = \{X_1, \dots, X_n\}$. Let $\mathcal{P} = \{Y_1, \dots, Y_\ell\}$ and $\overline{Y}^t = Y_1^t \cdots Y_\ell^t$ for all $t \in \mathbb{N}$. If $m \in A(\overline{X}^0)$, then we say that Λ **outputs** $\overline{Y}^m$ **in round** m and m is an **output round**. Also Λ induces an **output sequence** $(\overline{Y}^t)_{t \in A(\overline{X}^0)}$ with input $\overline{X}^0$.
Note that the predicates are arranged according to the indexes.

Example: Lexicographical ordering

Example 2

Let $\mathcal{T} = \{X_1, \ldots, X_n\}$. Observe the program

$$
\begin{aligned}
&X_1 :- \left(\neg X_1 \wedge \bigwedge_{k=2}^{n} X_k\right) \vee \left(X_1 \wedge \bigvee_{k=2}^{n} \neg X_k\right),\\
&X_2 :- \left(\neg X_2 \wedge \bigwedge_{k=3}^{n} X_k\right) \vee \left(X_2 \wedge \bigvee_{k=3}^{n} \neg X_k\right),\\
&\vdots\\
&X_n :- \neg X_n.
\end{aligned}
$$

Moreover assume that assume that $A(\overline{X}^0) = \{0, 2, 4, \ldots\}$ for all inputs $\overline{X}^0$. The program simply goes through all bit strings of length n in lexicographical order and outputs precisely on even rounds. For instance, if $n = 3$ and the input is 000, then the program will cycle through bit strings in the order $000 \to 001 \to 010 \to 011 \to 100 \to 101 \to 110 \to 111 \to 000 \to \ldots$. For example the program outputs 010 in round 2 and 011 in round 4.

Boolean networks

As its name would suggest, a program of Boolean network logic can be written as a directed network. The predicates X are interpreted as vertices v_X. There is an edge from v_X to v_Y if and only if X appears in the formula associated with Y. For example, in the example of the previous slides the predicate X appears in the formula $\neg X$ associated with Y. Thus, the graph would have an edge from v_X to v_Y. One can imagine that the vertices send their truth values to each other via the edges, and change their state by slotting the truth values into their formula.

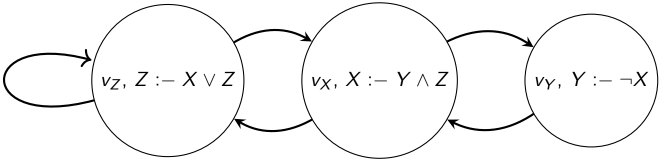


Figure: The graph representation of the BNL_0-program $X :- Y \wedge Z$; $Y :- \neg X$; $Z :- X \vee Z$.

BNL-program (informally)

The main weakness of BNL_0-programs is that each head predicate takes an input. This does not seem a big deal but it actually restricts which kinds of output sequences we can produce.
Let us modify the previous program by changing X and Z into "auxiliary predicates".

$$\begin{array}{rl|llll} X^0 = 1, & X :- Y \wedge Z, & X^0 = 1, & X^1 = 0, & X^2 = 0, & \dots \\ & Y :- \neg X, & Y^0 = 1, & Y^1 = 0, & Y^2 = 1, & \dots \\ Z^0 = 0, & Z :- X \vee Z. & Z^0 = 0, & Z^1 = 1, & Z^2 = 1, & \dots \end{array}$$

Now the length of the input is only 1, because X and Z don't receive any input. The program is an implementation of a function $\{0,1\} \to \{0,1\}^2$, and with the input 1 it outputs 01. A program that has two lists - one with initial values for auxiliary predicates and one with formulae corresponding to all predicates - is called a **program of the logic** BNL.

BNL-program

Definition 3

Let $\mathcal{T} = \{X_1, \ldots, X_n\} \subseteq \mathrm{VAR}$ be a set of $n \in \mathbb{Z}_+$ distinct schema variables. A $\mathcal{T}$**-program of the logic** BNL is a triple $(L, \mathcal{P}, A)$, where L is a list

$$\begin{array}{ll} Y_1^0 = b_1, & X_1 :- \psi_1, \\ \vdots & \vdots \\ Y_m^0 = b_m, & X_m :- \psi_m \\ & \vdots \\ & X_n :- \psi_n, \end{array}$$

where $\mathcal{A} = \{Y_1, \ldots, Y_m\} \subseteq \mathcal{T}$ is a set of **auxiliary predicates**, $b_1, \ldots, b_m \in \{0, 1\}$, $\mathcal{I} = \mathcal{T} \setminus \mathcal{A}$ is a set of **input predicates**, $\psi_1, \ldots, \psi_n$ are formulae defined over the language $\psi ::= \top \mid X_i \mid \neg\psi \mid \psi \wedge \psi$ $(X_i \in \mathcal{T})$, $\mathcal{P} \subseteq \mathcal{T}$ is a set of **print predicates** and $A\colon \{0, 1\}^{|\mathcal{I}|} \to \wp(\mathbb{N})$ is an **attention function**.

BNL: semantics

Let $\Lambda = (L, \mathcal{P}, A)$ be some $\mathcal{T}$-program of the logic BNL and let $\pi \colon \mathcal{T} \to \{0,1\}$ be a function. We let X^t denote the truth value of $X \in \mathcal{T}$ at time $t \in \mathbb{N}$. It is defined recursively as follows:

- At time 0 we define that $X^0 = b$ if the rule $X^0 = b$ appears in the program for some $b \in \{0,1\}$. Otherwise, we define that $X^0 = \pi(X)$.
- Assume we have defined the truth value of each predicate at time $t \in \mathbb{N}$. The truth value of X at time $t+1$ is defined exactly as in BNL_0.

The **input** of Λ is the bit string $\overline{X}^0 = X_1^0 \cdots X_n^0$, where $X_1, \ldots, X_n$ are exactly the input predicates of Λ. Let $\mathcal{P} = \{Y_1, \ldots, Y_\ell\}$ and $\overline{Y}^t = Y_1^t \cdots Y_\ell^t$ for all $t \in \mathbb{N}$. If $m \in A(\overline{X}^0)$, then we say that Λ **outputs** $\overline{Y}^m$ **in round** m and m is an **output round**.

Λ induces an **output sequence** $(\overline{Y}^t)_{t \in A(\overline{X}^0)}$ with input $\overline{X}^0$.

Note that the predicates are arranged according to the indexes.

Example: Flagging

Example 4

Let $\mathcal{T} = \{F, X, Y, Z\}$. Consider the following BNL-program.

$$\begin{aligned} F^0 = 0, \quad & F :- \top, \\ & X :- (\neg F \wedge (Y \wedge Z)) \vee (F \wedge X), \\ & Y :- (\neg F \wedge \neg X) \vee (F \wedge Y), \\ & Z :- (\neg F \wedge (X \vee Z)) \vee (F \wedge Z). \end{aligned}$$

The program is a modification of a previous BNL_0-program; the erstwhile rules of predicates X, Y and Z are written in blue, but now a predicate F (called a flag) acts as a precondition for them. No matter the input of the program, we have $F^t = F^1$, $X^t = X^1$, $Y^t = Y^1$ and $Z^t = Z^1$ for all $t \geq 2$. In other words, the program applies the rules of the BNL_0-program exactly once. A program of BNL_0 could produce the same output at time $t = 1$, but it is not capable of counting rounds, unlike a BNL-program.

Attractors and transient time

Given that the number of states is finite, a BNL-program (BNL_0-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. The concept of transient time is also applicable to SC, since it is also deterministic and eventually stabilizes with each input.

Halting BNL-programs

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 $\mathrm{i} \in \{0,1\}^{|\mathcal{I}|}$ results in a single (repeating) **output** denoted by $\Lambda(\mathrm{i})$, 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 $\Lambda(\mathrm{i}) = f(\mathrm{i})$ for all $\mathrm{i} \in \{0,1\}^{\ell}$. The **computation time** of a halting BNL-program is its transient time.

Size and depth

The **size** of a BNL-program is the number of appearances of symbols $\top$, X, $\neg$, and $\vee$ in the formulae ψ that appear in rules $X :- \psi$ in the program.
The **depth** of a BNL-program Λ is the maximum number of nested Boolean connectives in the formulae ψ that appear in rules $X :- \psi$ in the program.

Content

Motivation

- In various literature, neural networks usually use real numbers numbers for computing.
- In a practical scenarios real numbers are presented with floating-point numbers.
- Most of the neural networks that are used in applications are run by computers, i.e., most of the applied neural networks use floating-point arithmetic to compute.
- Therefore, in order to characterize neural networks with BNL-programs, we have to simulate floating-point arithmetic.

Floating-point numbers

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

$$\pm \underbrace{0.d_1 d_2 \cdots d_p}_{=f} \times \beta^{\pm e_1 \cdots e_q},$$

where $d_i, e_i \in [0; \beta - 1]$. For such a number in system S, we call f 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**).

Normalized fp-numbers

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 $+$.

Raw fp-numbers

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 $\pm d_0.d_1 d_2 \cdots d_{p'} \times \beta^{\pm e_1 \ldots e_{q'}}$.
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 e.g. **truncation** or **round-to-nearest ties-to-even**.

Representing integers in binary

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 $\mathsf{s}_1, \dots, \mathsf{s}_k \in \{0,1\}^\beta$ be **one-hots**, i.e. bit strings with exactly one 1. We say that $\mathsf{s} = \mathsf{s}_1 \cdots \mathsf{s}_k$ **corresponds** to $b_1 \cdots b_k \in [0; \beta - 1]^k$ if for every b_i, we have $\mathsf{s}_i(b_i) = 1$ (and other values in 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 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.

Representation of fp-numbers in binary

Let $F = \pm f \times \beta^{\pm e}$ be a floating-point number in system S. Let $\mathrm{p}_1, \mathrm{p}_2 \in \{0,1\}$ and $\mathrm{s}_1, \ldots, \mathrm{s}_q, \mathrm{s}'_1, \ldots, \mathrm{s}'_p \in \{0,1\}^\beta$. We say that

$$\mathrm{s} = \mathrm{p}_1\mathrm{p}_2\mathrm{s}_1 \cdots \mathrm{s}_q\mathrm{s}'_1 \cdots \mathrm{s}'_p$$

corresponds to F (or s is a **one-hot representation** of F) if

1. $\mathrm{p}_1 = 1$ iff the sign of the exponent is $+$,
2. $\mathrm{p}_2 = 1$ iff the sign of the fraction is $+$,
3. $\mathrm{s}_1 \cdots \mathrm{s}_q$ corresponds to $e = e_1 \cdots e_q$,
4. $\mathrm{s}'_1 \cdots \mathrm{s}'_p$ corresponds to $f = 0.d_1d_2 \cdots d_p$ (or, more precisely, to $d_1 \cdots d_p$).

Likewise, we say that a bit string s **corresponds** to a sequence $(F_1, \ldots, F_k)$ of floating-point numbers if 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

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

Simulating floating-point functions

Definition 5

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

Results

Lemma 6

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 $r = \max\{p, q\}$.

Lemma 7

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 $r = \max\{p, q\}$.

Lemma 8

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 $r = \max\{p, q\}$.

Theorem 9

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 $\mathcal{O}((\log(\Omega) + 1)(\log(r) + \log(\beta)))$.*

Content

What is machine learning?

- Informally, machine learning means solving problems by "teaching" a machine to find a sufficiently good algorithm that solves the problem instead of defining the algorithm oneself.
- Machine learning considers various "machines", and the most central of these in current research are neural networks.

Example 10

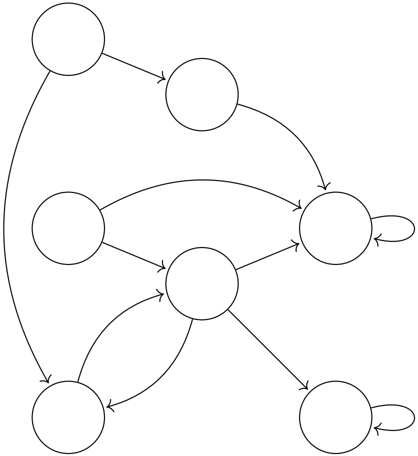
Consider a machine whose purpose is to recognize whether a picture contains a cat or a dog. If the machine gives a wrong answer, it is told about it, and the machine adjusts the algorithm to improve it by e.g. more closely examining the shape of the creature's ears.

Introduction to neural networks

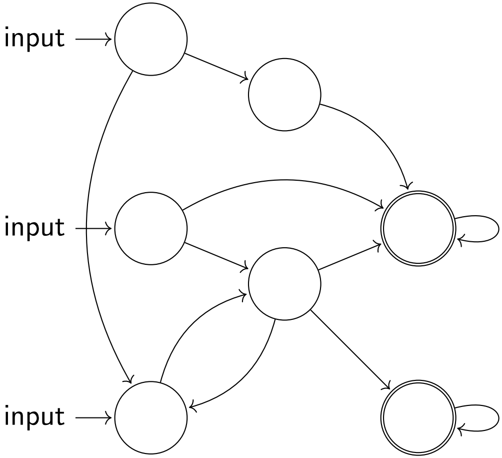
A neural network is a type of distributed system used in machine learning. Its design is modeled after the human brain. Neural networks typically perform calculations with real numbers, and they have an input and an output.

Consideration is often limited to feedforward neural networks (FNNs) whose topology is restricted. We consider a more general class of neural networks where the network topology is simply that of a directed graph. Furthermore, our neural networks perform calculations with floating-point numbers instead of real numbers, as this is more realistic for computers.

We start with an informal description of the type of neural network we're examining by constructing an example neural network step by step. After that we go over how it operates.



A neural network is shaped like a network (akin to the brain). Nodes of the network may be referred to as **neurons** and the edges as **synapses**. We assume there is an ordering for the nodes.



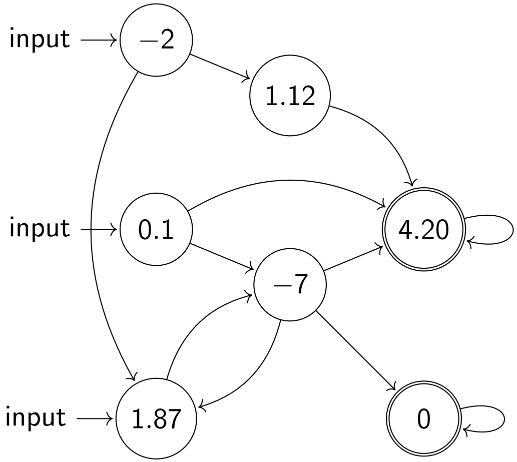
Neural networks have named **input nodes** and **output nodes**. We mark the input nodes with arrows and output nodes with doubled outlines.

Introduction to how neural networks operate

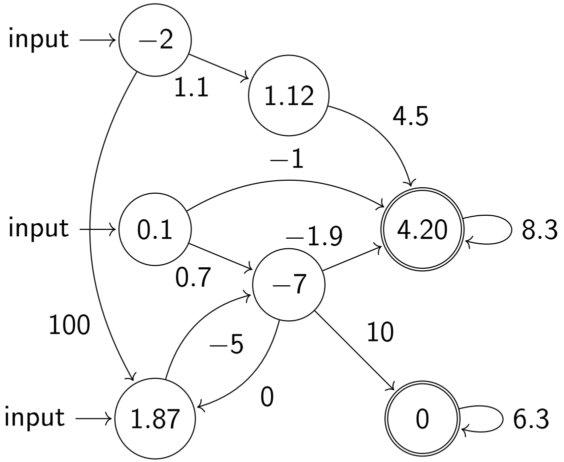
Before going any further, we consider the purpose of a neural network. Since a neural network is modeled after the brain, its operation should describe how a brain works in some way.

- An individual neuron has an **activation value** (a floating-point number) that changes while the network is running. Each neural network is thus associated with some floating-point system $S = (p, q, \beta)$.
- A neural network operates synchronically in discrete rounds. The initial activation value of input nodes depends on the input, while each non-input node has some fixed initial value. In each round, the neurons send their activation values via edges to other neurons according to the directions of the edges. Then each node calculates a new activation value based on the activation values received in that round.

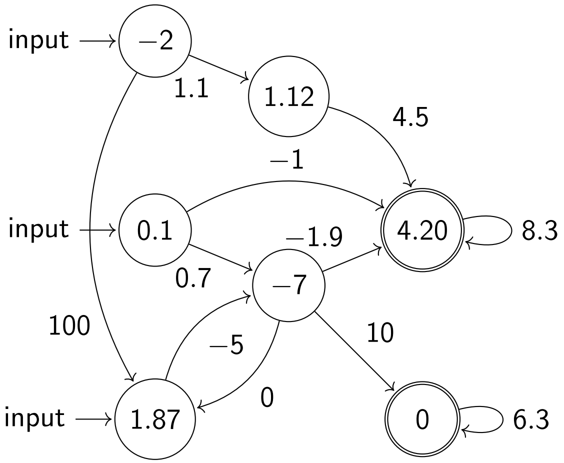
We must describe neural networks more precisely to show how they calculate activation values. We will do this one step at a time.



Each node v is equipped with a **bias term** $b_v \in S$.



Each edge e is equipped with a **weight** $w_e \in S$.



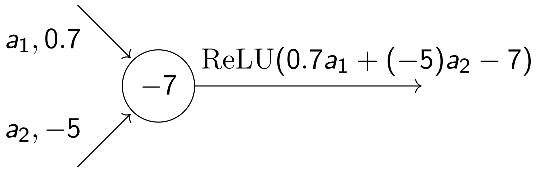
Finally, each node v is equipped with an **activation function** $\alpha_v \colon S \to S$. The function may be different for each node, but they are typically the same. We assume that the activation functions are floating-point approximations of piecewise polynomial functions. Well-known activation functions include $\mathrm{ReLU}(x) = \max\{0, x\}$ and the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$.

An informal description of an individual neuron

Let us examine the node from the previous slide with the bias term -7 and call it s. Let us assume that each node in the network has an activation value and that the activation function of s is $\mathrm{ReLU}(x) = \max\{0, x\}$. The neuron has 2 incoming edges, both of which have an individual weight. A new activation value for the neuron s is calculated in the following steps:

1. The predecessors of s send their activation values to s along edges.
2. Each of these activation values is multiplied by the weight of the edge it's arriving across.
3. These products are added together along with the bias term of s.
4. This sum becomes the argument for the activation function, and its image is the new activation value of s.

This is demonstrated in the next slide.



In the picture, a_1 and a_2 are the activation values arriving from other nodes. A new activation value is obtained with the following formula: $\mathrm{ReLU}(0.7a_1 + (-5)a_2 + (-7))$. If for instance $a_1 = -2$ and $a_2 = -0.9$, then $\mathrm{ReLU}(0.7(-2) + (-5)(-0.9) + (-7)) = \mathrm{ReLU}(-1.4 + 4.5 - 7) = \mathrm{ReLU}(-3.9) = 0$.

How a neural network functions informally

As stated before, the nodes of a neural network are separated into two categories: input nodes I and non-input nodes $V \setminus I$. A neural network is equipped with a function $\pi\colon V \setminus I \to S$ that assigns an initial value to each non-input node. A neural network performs calculations in discrete rounds $t \in \mathbb{N}$. Given an input $(F_1, \ldots, F_{|I|}) \in S^{|I|}$, we define the activation value of a node $v \in V$ at time $t \in \mathbb{N}$ (denoted by v^t) as follows.

1. If $t = 0$ and $v \in V \setminus I$, then $v^0 = \pi(v)$. If $v \in I$ and v is the ith input node, then $v^0 = F_i$.
2. Assume we have defined u^t for all $u \in V$. Let $u_1, \ldots, u_n$ denote the predecessors of v at time t, let $w_1, \ldots, w_n$ denote the weights of the associated edges, and let b_v denote the bias of v. Then

$$v^{t+1} = \alpha_v\Big(\sum_{i=1}^{n} u_i^t w_i + b_v\Big).$$

A neural network contains a set $O \subseteq V$ of **output nodes**. A neural network is also equipped with an **attention function** function $a\colon S^{|I|} \to \wp(\mathbb{N})$ that assigns a set of output rounds (natural numbers) to each input. We can, for instance, define that with a specific input a neural network outputs precisely in rounds $3n$ where $n \in \mathbb{N}$. This is consistent with the output method of BNL-programs, only generalized for floating-point numbers.

Let $v_1, \ldots, v_{|O|}$ denote the output nodes, and let $\overline{v}^t = (v_1^t, \ldots, v_{|O|}^t) \in S^{|O|}$ be the tuple of activation values of the output nodes in round t with an input $\mathcal{F} \in S^{|I|}$. If $t \in a(\mathcal{F})$, then we say that the **output of the neural network at time** t is $\overline{v}^t$. Given the input $\mathcal{F}$, a neural network produces an **output sequence** $(\overline{v}^t)_{t \in a(\mathcal{F})}$ (each component $\overline{v}^t$ of the output sequence is itself a sequence of floating-point numbers).

Formal definition for neural networks

Definition 11

A **neural network** in the system $S = (p, q, \beta)$ is an ordered directed graph $(V, E, <^V)$ (a directed graph where the nodes are ordered according to $<^V$), which also includes

- a set of **input nodes** $I \subseteq V$ and **output nodes** $O \subseteq V$,
- a **weight** $w_e \in S$ for each edge $e \in E$,
- a **bias** $b_v \in S$ for each node $v \in V$,
- an **activation function** $\alpha_v \colon S \to S$ for each node $v \in V$,
- **an initializing function** $\pi \colon V \setminus I \to S$,
- an **attention function** $a \colon S^{|I|} \to \wp(\mathbb{N})$.

If we want to be more precise, a neural network is the following tuple: $\mathcal{N} = ((V, E, <^V), S, I, O, (w_e)_{e \in E}, (b_v)_{v \in V}, (\alpha_v)_{v \in V}, \pi, a)$ (Phew!).

Important qualities of a neural network

The **degree** of a 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.

Content

Asynchronous equivalence

When translating a neural network into a program of BNL, notice that we are transitioning from a floating-point framework to a Boolean framework. We use one-hot representations of floating-point numbers to represent them in binary. By equivalence we basically mean that the neural network and the BNL-program have matching output sequences.

Definition 12

Let $\mathcal{N}$ be a neural network (for $S = (p, q, \beta)$) and let Λ be a BNL-program. Let I be the set of input nodes of $\mathcal{N}$, and let a and A be the attention functions of $\mathcal{N}$ and Λ respectively. We say that $\mathcal{N}$ and Λ are **asynchronously equivalent in** S if for all $\mathcal{F} \in S^{|I|}$ it holds that if $\mathrm{i} \in \{0,1\}^{(\beta(p+q)+2)|I|}$ is the bit string that corresponds to $\mathcal{F}$, then the elements of $(\overline{X}^t)_{t \in A(\mathrm{i})}$ correspond to the elements of $(\overline{v}^t)_{t \in a(\mathcal{F})}$.

Note that though the output sequences have to be corresponding, the output rounds may differ, hence the name "asynchronous".

Computation delay

Given two asynchronously equivalent objects x and y, we define a notion of computation delay. Let $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ enumerate the output rounds of x and y respectively. Furthermore, assume that $x_n \geq y_n$ for every $n \in \mathbb{N}$. The **computation delay** of x (w.r.t. y) is the smallest $T \in \mathbb{N}$ such that $T \cdot y_n \geq x_n$ for every $n \in \mathbb{N}$. (If such a number T does not exist, then we could define that the computation delay of x is ∞, but we do not consider such scenarios.)

From neural networks to BNL-programs

The first of our two results is given below.

Theorem 13

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 $\mathcal{O}((\log(\Omega) + 1)(\log(r) + \log(\beta)) + \log(\Delta))$.*

Asynchronous equivalence in binary

When translating a BNL-program into a neural network, we don't need complex floating-point representations; we can simply use the floating-point representations of 0 and 1 in the neural network. Instead of producing a corresponding output sequence, the neural network produces essentially the very same output sequence.

Definition 14

Let Λ be a BNL-program and let $\mathcal{N}$ be a neural network (for $S = (p, q, \beta)$). Let $\mathcal{I}$ be the set of input predicates of Λ, and let A and a be the output functions of Λ and $\mathcal{N}$ respectively. We say that Λ and $\mathcal{N}$ are **asynchronously equivalent in binary** if for all $\mathrm{i} \in \{0,1\}^{|\mathcal{I}|}$ we have that $(\overline{X}^t)_{t \in A(\mathrm{i})} = (\overline{v}^t)_{t \in a(\mathrm{i})}$.

From BNL-programs to neural networks

The second of our two main results is given below.

Theorem 15

Given a BNL*-program* Λ *of size* s *and depth* d*, we can construct a 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* $\mathcal{O}(d)$ *such that* Λ *and* $\mathcal{N}$ *are asynchronously equivalent in binary.*

(Reminder: The rectified linear unit is defined by $\mathrm{ReLU}(x) = \max\{0, x\}$. The Heaviside step function is defined by $H(x) = 1$ if $x > 0$, and $H(x) = 0$ otherwise.)