New Directions in Descriptive Complexity

Veeti Ahvonen

Tampere University

January 5, 2024

What is descriptive complexity?

- In classical descriptive complexity aim is to characterize complexity classes (or class of computational models) via logic, i.e., obtain "logical characterization" for the computational problems in complexity class.
- **Questions:** Does a computational model class (or a complexity class) $\mathcal{C}$ of have the same expressive power as a formula class Φ?
- Informally, we want to obtain an equivalence, where a single computational model in $\mathcal{C}$ corresponds to a logical formula in Φ and vice versa.

Example: Fagin's Theorem

- The complexity class NP contains all the (decision) problems that are solvable by a non-deterministic Turing machine in polynomial time.
- **Fagin's theorem:** Problems in the complexity class NP are precisely the problems expressed in the existential second order logic (also known as ESO).[1]
- Therefore, properties in NP apply to ESO and vice versa.

[1] Ronald Fagin. "Generalized first-order spectra and polynomial-time recognizable sets". In: *Complexity of computation (Proc. SIAM-AMS Sympos., New York, 1973)*. Vol. 7. SIAM-AMS Proc. 1974, pp. 43–73.

Why?

- So descriptive complexity theory gives as a highly useful tool; we can apply logic-based methods to reason about the complexity class.
- Many results in descriptive complexity theory only concerns classical models e.g. Turing machines and finite deterministic automata.

Recent developments

In this talk, I will introduce a recent developments in the descriptive complexity theory for modern computational models.[2]

Goal: We will show how to characterize a class of neural networks with a rule-based Boolean logic.

[2] Veeti Ahvonen, Damian Heiman, and Antti Kuusisto. "Descriptive complexity for neural networks via Boolean networks". In: *CoRR* abs/2308.06277 (2023). DOI: 10.48550/ARXIV.2308.06277. arXiv: 2308.06277. URL: https://doi.org/10.48550/arXiv.2308.06277

What are neural networks?

- The most central machine learning framework currently is neural networks (or simply NNs).

What are neural networks?

- The most central machine learning framework currently is neural networks (or simply NNs).
- Neural networks are inspired by human brain. They are often represented as weighted graphs, which compute in discrete rounds. Each node has an "activation value" and nodes update these values by communicating synchronously to each other.

Framework informally

- Neural networks are based on the weighted directed graphs (self-loops are allowed).
- Computation is carried by floating-point numbers instead of real numbers.
- The "activation values" are computed by using "activation functions" that are floating-point approximations of polynomial piecewise functions.

Preliminaries: 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

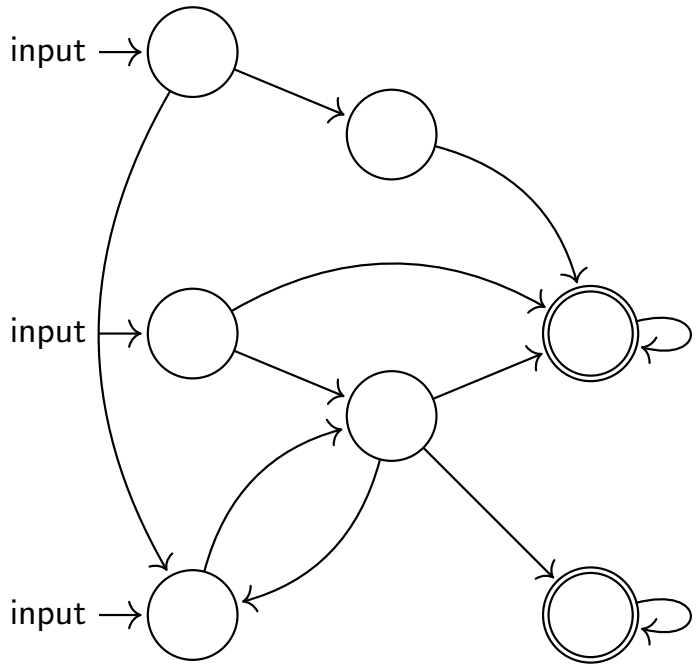
$$\pm \underbrace{0.d_1 d_2 \cdots d_p}_{=f} \times \beta^{\pm e_1 \cdots e_q},$$

where $d_i, e_i \in \{0, \ldots, \beta - 1\}$.

For example, 0.10×10^1 and 0.01×10^2 are both representations of the number 1.

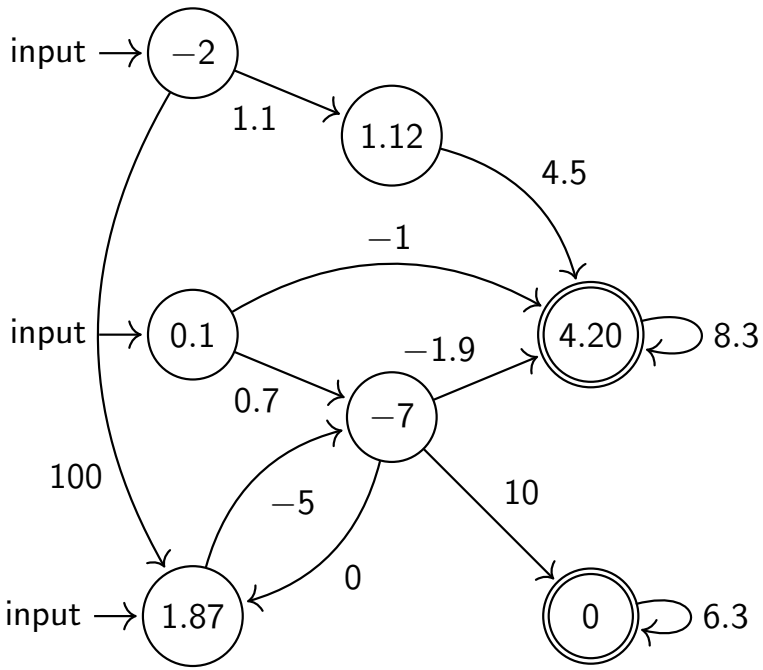
From now, assume that a neural network is fixed with a floating-point system S.

Closer look on our framework



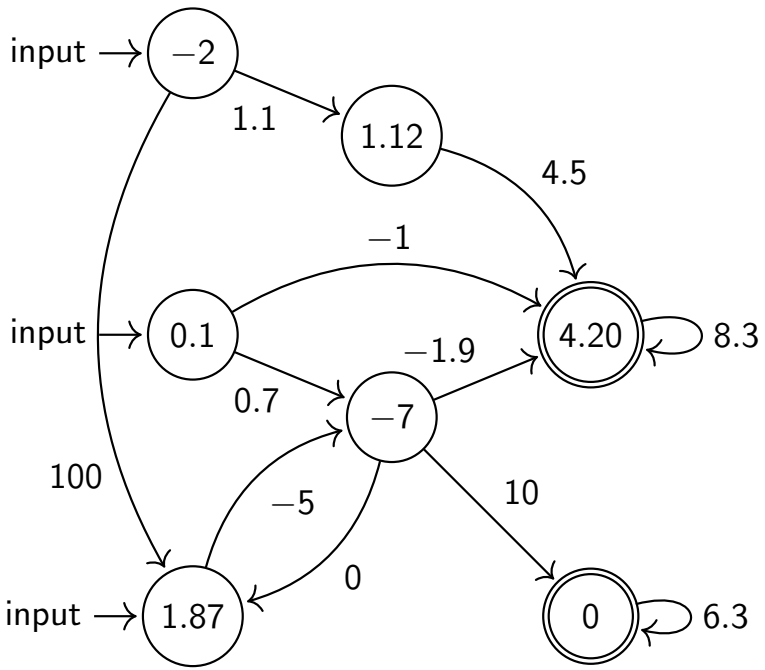
A neural network is a directed ordered graph $(V, E, <_V)$, which includes named **input nodes** $I \subseteq V$ and **output nodes** $O \subseteq V$. A neural network is also equipped with an **initializing function** $\pi\colon V \setminus I \to S$, which gives an initial value for each non-input node.

Bias terms and weights



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

Activation function



Also, each node v is equipped with an **activation function** $\alpha_v : S \to S$. We assume that the activation functions are floating-point approximations of piecewise polynomial functions.

How does a neural network compute?

- A neural network computes in discrete rounds $t \in \mathbb{N}$ and computes an **activation value** for each node as follows.

How does a neural network compute?

- A neural network computes in discrete rounds $t \in \mathbb{N}$ and computes an **activation value** for each node as follows.
- In round $t = 0$, each input node $v \in I$ is fed with a floating-point number $v^0 \in S$ and non-input nodes $V \setminus I$ are initialized by a function π.

How does a neural network compute?

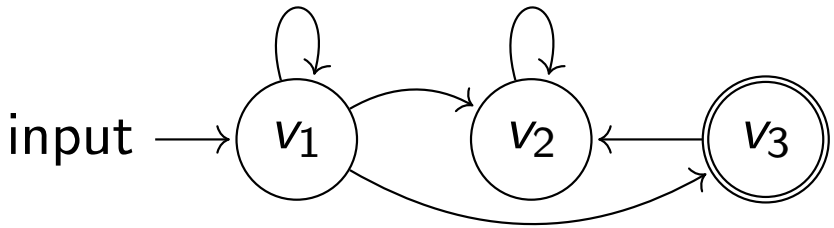
- A neural network computes in discrete rounds $t \in \mathbb{N}$ and computes an **activation value** for each node as follows.
- In round $t = 0$, each input node $v \in I$ is fed with a floating-point number $v^0 \in S$ and non-input nodes $V \setminus I$ are initialized by a function π.
- In round $t > 0$, each node v computes a new value v^t as follows. Let $u_1^{t-1}, \ldots, u_n^{t-1}$ denote the activation values of predecessors of v at round $t-1$, let $w_1, \ldots, w_n$ denote the weights of the associated edges, and let b_v denote the bias of v. Then the value of v at round t is computed by the formula

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

Neural network output conditions

Each neural network is also associated with an **attention function** $a\colon S^{|I|} \to \wp(\mathbb{N})$ that assigns a set of **output rounds** (a set of natural numbers) to each input for the neural network.

Neural network output conditions



v_1	$v_1^0 = 1.1$,	$v_1^1 = -0.1$,	$v_1^2 = -1.1$,	$v_1^3 = -2.1$, $\ldots$
v_2	$v_2^0 = 2.6$,	$v_2^1 = 2.7$,	$v_2^2 = 2.9$,	$v_2^3 = 3.4$, $\ldots$
v_3	$v_3^0 = 0.7$,	$v_3^1 = -0.71$,	$v_3^2 = 1.5$,	$v_3^3 = 5.0$, $\ldots$

The node v_1 is an input node and it is fed with value 1.1 in blue and other nodes has a fixed initial value. Assume that output rounds are $\{1, 3, 6, \ldots\}$ with input 1.1. The colored red values of v_3 are the output values in the output rounds.

Neural network output conditions

Therefore, a neural network $\mathcal{N}$ with a given input $\mathcal{F} \in S^{|I|}$ produces an **output sequence** $\mathcal{N}(\mathcal{F}) = (\overline{v}^t)_{t \in a(\mathcal{F})}$, where $\overline{v}^t \in S^{|O|}$ is floating-point sequence induces by the outputs of output nodes at round t.

BNL-programs

Let $\mathcal{T} = \{X, Y, Z\}$ be a finite ordered set ($X < Y < Z$) with three Boolean variables (or predicates) i.e., variables that can only contain the values 0 or 1.

BNL-programs

Let $\mathcal{T} = \{X, Y, Z\}$ be a finite ordered set ($X < Y < Z$) with three Boolean variables (or predicates) i.e., variables that can only contain the values 0 or 1.

- We attach a Boolean formula to each variable, that use the variables from $\mathcal{T}$ (we also allow to use $\top$ symbol).

$$X :- Y \wedge Z,$$
$$Y :- \neg X,$$
$$Z :- X \vee Z.$$

BNL-programs

Let $\mathcal{T} = \{X, Y, Z\}$ be a finite ordered set ($X < Y < Z$) with three Boolean variables (or predicates) i.e., variables that can only contain the values 0 or 1.

- We attach a Boolean formula to each variable, that use the variables from $\mathcal{T}$ (we also allow to use $\top$ symbol).
- Each variable is given an initial value in round 0 (upper index 0), which is either 0 or 1.

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

BNL-programs

Let $\mathcal{T} = \{X, Y, Z\}$ be a finite ordered set ($X < Y < Z$) with three Boolean variables (or predicates) i.e., variables that can only contain the values 0 or 1.

- We attach a Boolean formula to each variable, that use the variables from $\mathcal{T}$ (we also allow to use $\top$ symbol).
- Each variable is given an initial value in round 0 (upper index 0), which is either 0 or 1.
- Each variable calculates a new truth value for itself in each round $t > 0$ by slotting the truth values of all variables from previous round $t - 1$ into its associated formula. This can carry on indefinitely.

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

BNL-programs

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

The list on the left, which contains variables and formulae associated with them, is called a **program of** BNL (Boolean network logic). On the right is the run of the program, where upper index t for the variable means the value of the variable at the time step t.

BNL: input and output conditions

- Each program is associated with a set of **input predicates** $\mathcal{I} \subseteq \mathcal{T}$. The **input** for the program is a binary string $\mathsf{i} \in \{0,1\}^{|\mathcal{I}|}$ that gives an initial value for each input predicate (w.r.t. order of variables).

BNL: input and output conditions

- Each program is associated with a set of **input predicates** $\mathcal{I} \subseteq \mathcal{T}$. The **input** for the program is a binary string $\mathrm{i} \in \{0,1\}^{|\mathcal{I}|}$ that gives an initial value for each input predicate (w.r.t. order of variables).
- The predicates in $\mathcal{T} \setminus \mathcal{I}$ are called **auxiliary predicates**. Each auxiliary predicate $X \in \mathcal{T} \setminus \mathcal{I}$ is associated with a rule $X^0 = b$, where $b \in \{0,1\}$, which gives an initial value for auxiliary predicate X.

BNL: input and output conditions

- Each program is associated with a set of **input predicates** $\mathcal{I} \subseteq \mathcal{T}$. The **input** for the program is a binary string $\mathsf{i} \in \{0,1\}^{|\mathcal{I}|}$ that gives an initial value for each input predicate (w.r.t. order of variables).
- The predicates in $\mathcal{T} \setminus \mathcal{I}$ are called **auxiliary predicates**. Each auxiliary predicate $X \in \mathcal{T} \setminus \mathcal{I}$ is associated with a rule $X^0 = b$, where $b \in \{0,1\}$, which gives an initial value for auxiliary predicate X.
- Each program is associated with a set of **print predicates** $\mathcal{P} \subseteq \mathcal{T}$.

BNL: input and output conditions

- Each program is associated with a set of **input predicates** $\mathcal{I} \subseteq \mathcal{T}$. The **input** for the program is a binary string $\mathsf{i} \in \{0,1\}^{|\mathcal{I}|}$ that gives an initial value for each input predicate (w.r.t. order of variables).
- The predicates in $\mathcal{T} \setminus \mathcal{I}$ are called **auxiliary predicates**. Each auxiliary predicate $X \in \mathcal{T} \setminus \mathcal{I}$ is associated with a rule $X^0 = b$, where $b \in \{0,1\}$, which gives an initial value for auxiliary predicate X.
- Each program is associated with a set of **print predicates** $\mathcal{P} \subseteq \mathcal{T}$.
- Each program is also associated with an **attention function** $A\colon \{0,1\}^{|\mathcal{I}|} \to \wp(\mathbb{N})$ which assigns a set of **output rounds** for each input.

BNL: input and output conditions

For example, assume that the set of input and print predicates for the program below is $\{X, Z\}$ and $Y^0 = 0$. If the input for the program is $X^0Z^0 = 01$ and $A(01) = \{2\}$, then the output of the program with input 01 is $X^2Z^2 = 11$. The input of the program is marked in blue and the output in red.

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

BNL: input and output conditions

For example, assume that the set of input and print predicates for the program below is $\{X, Z\}$ and $Y^0 = 0$. If the input for the program is $X^0Z^0 = 01$ and $A(01) = \{2\}$, then the output of the program with input 01 is $X^2Z^2 = 11$. The input of the program is marked in blue and the output in red.

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

Therefore, a BNL-program Λ with a given input $\mathsf{i} \in \{0,1\}^{|\mathcal{I}|}$ induces a **output sequence** $\Lambda(\mathsf{i}) = (\overline{X}^t)_{t \in A(\mathsf{i})}$, where $\overline{X}^t \in \{0,1\}^{|\mathcal{P}|}$ induced by the print predicates at round t.

Representation of fp-numbers in binary

- Given a floating-point system $S(p, q, \beta)$ there are various ways to represent floating-points in binary. For example, IEEE 754 standard. So each floating-point number F in S has a **corresponding binary representation** $\mathrm{bit}(F)$.
- *The main idea is simple.* A single fp-number can be represented as a binary string in $\{0, 1\}^{2+(p+b)\beta}$, where signs are marked with 0 or 1 depending on the signs and each digit of the exponent and the fraction are represented by the string in $\{0, 1\}^{\beta}$ where precisely one bit that is 1 and others are 0.

Asynchronous equivalence in fp-system

By equivalence we basically mean that the neural network and the BNL-program have matching output sequences.

Definition 1

Let $\mathcal{N}$ be a neural network (in fp-system S) with input nodes I and let Λ be a BNL-program. We say that $\mathcal{N}$ and Λ are **asynchronously equivalent in** S if for all $\mathcal{F} \in S^{|I|}$ and the corresponding sequence of bit string $\mathsf{i} = \mathrm{bit}(\mathcal{F})$, the output sequence $\Lambda(\mathsf{i})$ correspond to the output sequence $\mathcal{N}(\mathcal{F})$.[3]

[3] The definition could be extend between two neural networks.

Computation delay

Let x and y be two asynchronously equivalent objects and 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 y (w.r.t. x) is the smallest $T \in \mathbb{N}$ such that $T \cdot y_n \geq x_n$ for every $n \in \mathbb{N}$.

From neural networks to BNL-programs

The first of our two results is given below.

Theorem 2

Given a neural network $\mathcal{N}$ in fp-system S, we can construct a BNL*-program Λ such that $\mathcal{N}$ and Λ are asynchronously equivalent in S and*

1. *the size of Λ is "small" (polynomial in the sizes of $\mathcal{N}$), and*
2. *the computation delay of "small" (polylogarithmic in the sizes of $\mathcal{N}$).*

From neural networks to BNL-programs

The first of our two results is given below.

Theorem 2

Given a neural network $\mathcal{N}$ in fp-system S, we can construct a BNL*-program Λ such that $\mathcal{N}$ and Λ are asynchronously equivalent in S and*

1. *the size of Λ is "small" (polynomial in the sizes of $\mathcal{N}$), and*
2. *the computation delay of "small" (polylogarithmic in the sizes of $\mathcal{N}$).*

Proof.

(Sketch) Each aggregation in the nodes can be simulated by the subprogram for floating-point addition and multiplication. The activation functions are simulated by the subprogram for piecewise polynomial activation functions. □

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.

[4] The definition could be extend between two BNL-programs.

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.

Definition 3

Let Λ be a BNL-program with input predicates $\mathcal{I}$ and let $\mathcal{N}$ be a neural network (in fp-system S) with input nodes I. We say that Λ and $\mathcal{N}$ are **asynchronously equivalent in binary** if for all $\mathsf{i} \in \{0,1\}^{|\mathcal{I}|}$ and corresponding fp-numbers $\mathcal{F} \in S^{|I|}$, the output sequence $\mathcal{N}(\mathcal{F})$ corresponds to the output sequence $\Lambda(\mathsf{i})$.[4]

[4] The definition could be extend between two BNL-programs.

From BNL-programs to neural networks

The second of our two main results is given below. Well-known activation functions include $\mathrm{ReLU}(x) = \max\{0, x\}$.

Theorem 4

Given a BNL*-program* Λ*, we can construct a neural network* $\mathcal{N}$ *for any floating-point system* S *with* ReLU *activation functions such that* Λ *and* $\mathcal{N}$ *are asynchronously equivalent in binary and*

1. *the size of* $\mathcal{N}$ *is linear in the size of* Λ*, and*
2. *the computation delay is "small".*

From BNL-programs to neural networks

The second of our two main results is given below. Well-known activation functions include $\mathrm{ReLU}(x) = \max\{0, x\}$.

Theorem 4

Given a BNL*-program* Λ*, we can construct a neural network* $\mathcal{N}$ *for any floating-point system* S *with* ReLU *activation functions such that* Λ *and* $\mathcal{N}$ *are asynchronously equivalent in binary and*

1. *the size of* $\mathcal{N}$ *is linear in the size of* Λ*, and*
2. *the computation delay is "small".*

Proof.

(Sketch) The BNL-program Λ is first transformed to the asynchronously equivalent BNL-program Γ (with "small" delay), where in each rule simplified. From Γ, it is easy to define asynchronously equivalent neural network by using ReLU (other activation functions are possible). □

Conclusion

- We obtained translations with small size and time explosions.
- Since the asynchronous equivalence could be extended between two neural networks, we can translate a single neural network to other asynchronously equivalent neural network which uses ReLU with our results.
- BNL is also highly linked to distributed computing and Boolean circuits.[5]

THANKS!

[5] Veeti Ahvonen et al. "Descriptive Complexity for Distributed Computing with Circuits". In: *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*. Vol. 272. 2023, 9:1–9:15.