Neural Networks and Kahn Networks: Relation Between Neurons, Logic and parallel programming

Ekaterina Komendantskaya

Department of Computer Science, University of St Andrews

Report in INRIA Sophia-Antipolis

Outline

Outline

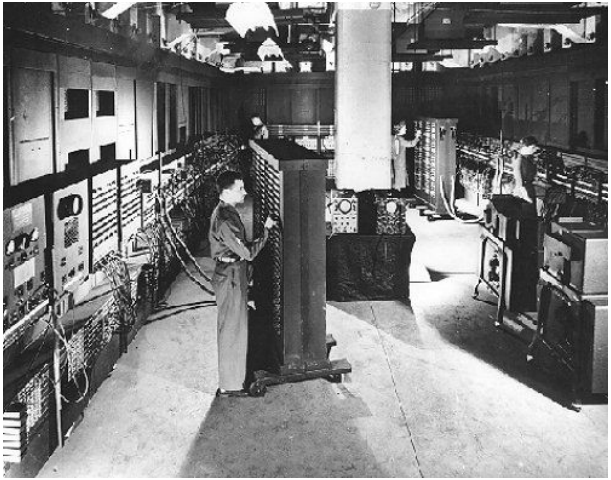
Outline

Motivation: a happy start for programmed computing

In 1946, the first useful electronic digital computer (ENIAC) is created: it was a happy start for the programmed computing.

Motivation: a happy start for programmed computing

In 1946, the first useful electronic digital computer (ENIAC) is created: it was a happy start for the programmed computing.



Motivation: a happy start?

One had to wait till 1976 to see the first personal computer -

Apple:

Motivation: a happy start?

One had to wait till 1976 to see the first personal computer -

Apple:

Programmed computing involves:

- devising an algorithm to solve a given problem;
- using software to encode it;
- using the hardware to implement the software

Note!

This has always had close theoretical relation to Turing machines, Mathematical Logic (Church, λ-calculus, Kleene, etc.), the theory of computable functions, and what we now call "Classical (conventional) models of computation".

Are there any other options?

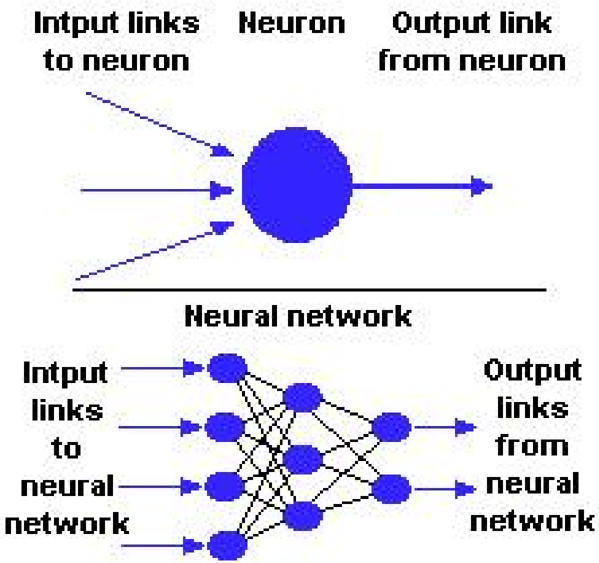
Neurocomputing

- Takes inspiration from (biological) neural networks, rather than from logic.
- Does not require a ready algorithm, but is capable to "learn" the algorithm from examples.
- Artificial Neural networks are parallel, distributed, adaptive processing systems that develop information processing capabilities in response to exposure to an information environment.

Note

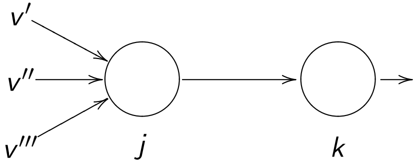
The major advantages: parallelism, learning, ability to adapt.

Neural Network: definitions



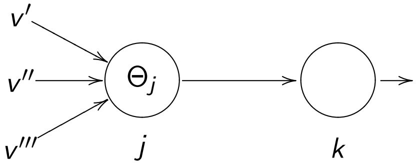
Neurons

$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t + \Delta t) = \psi(p_k(t))$$



Neurons

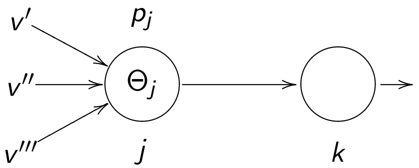
$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t + \Delta t) = \psi(p_k(t))$$



The following parameters can be trained: weights w_{kj}, biases b_k, b_j.

Neurons

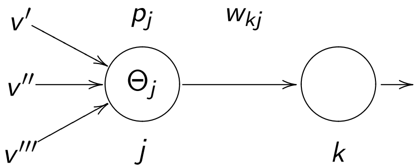
$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t + \Delta t) = \psi(p_k(t))$$



The following parameters can be trained: weights w_{kj}, biases b_k, b_j.

Neurons

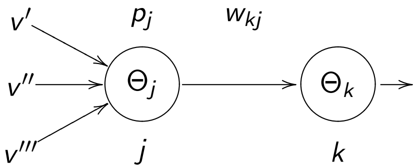
$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t + \Delta t) = \psi(p_k(t))$$



The following parameters can be trained: weights w_{kj}, biases b_k, b_j.

Neurons

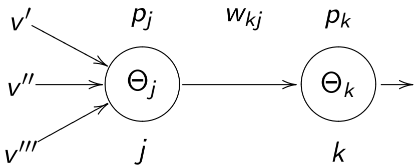
$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t + \Delta t) = \psi(p_k(t))$$



The following parameters can be trained: weights w_{kj}, biases b_k, b_j.

Neurons

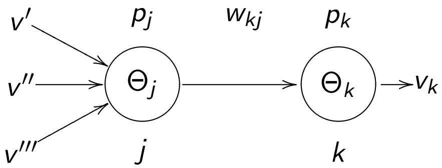
$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t+\Delta t) = \psi(p_k(t))$$



The following parameters can be trained: weights w_{kj}, biases b_k, b_j.

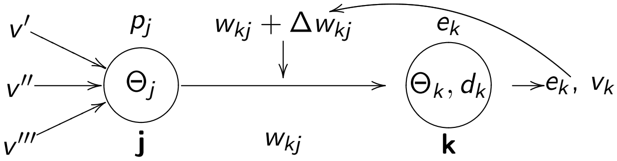
Neurons

$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) + b_k\right) - \Theta_k$$
$$v_k(t + \Delta t) = \psi(p_k(t))$$



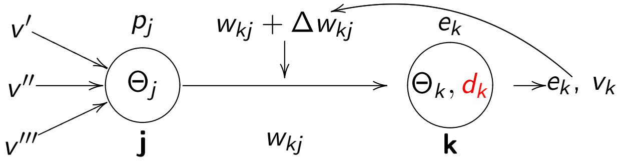
The following parameters can be trained: weights w_{kj}, biases b_k, b_j.

Error-Correction (Supervised) Learning



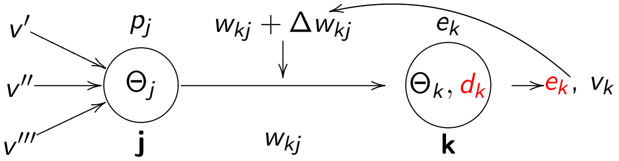
Error-Correction (Supervised) Learning

We embed a new parameter, **desired response** d_k into neurons;



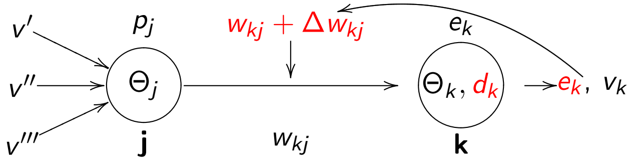
Error-Correction (Supervised) Learning

We embed a new parameter, **desired response** d_k into neurons;
Error-signal: $e_k(t) = d_k(t) - v_k(t)$;



Error-Correction (Supervised) Learning

We embed a new parameter, **desired response** d_k into neurons;
Error-signal: $e_k(t) = d_k(t) - v_k(t)$;
Error-correction learning rule: $\Delta w_{kj}(t) = \eta e_k(t) v_j(t)$.



First Engineering insights:

Mark 1 and Mark 2 Perceptrons (1948 - 1958)

First Engineering insights:

Frank Rosenblatt with 400 pixel Mark 1 Perceptron image sensor.

First Engineering insights:

The Mark 1 Perceptron patchboard. The connection patterns were typically random, so as to illustrate the ability of the perceptron to learn the desired pattern without need for precise writing (in contrast to a programmed computer).

Hardware facts:

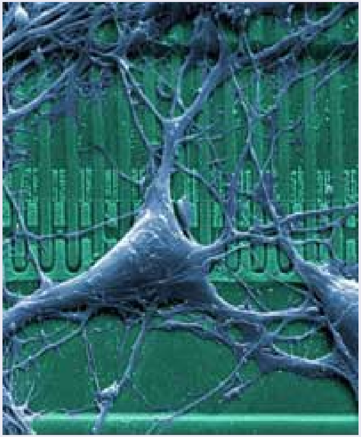
- Often, "Neurocomputers" are attached to "programmed" computers as coprocessors.
- The most successful neurocomputers were built from electronic or optical hardware, or their mixture.
- The other types of hardware that were taken for experiments were: chemical, acoustic, mechanical, and even ...

.

Hardware facts:

- Often, "Neurocomputers" are attached to "programmed" computers as coprocessors.
- The most successful neurocomputers were built from electronic or optical hardware, or their mixture.
- The other types of hardware that were taken for experiments were: chemical, acoustic, mechanical, and even ... biological.

Nerve cell connected to a silicon chip, 2001



Software facts:

For those who develop neural network software, there exists a variety of neural network simulators, written on programming languages: C (Stuttgart Neural Network Simulator), Matematica, R, Matlab, JAVA, Predictive Model Markup Language (PMML) - the latter is XML based.

Note:

I know nothing about NN software written on functional languages... I know nothing about Coq libraries... but whatever libraries for matrices exist, could be applied straightforwardly.

Common applications:

- Function approximation, or regression analysis, including time series prediction and modeling.
- Classification, including pattern and sequence recognition, novelty detection and sequential decision making.
- Data processing, including filtering, clustering, blind source separation and compression.

Application areas: system identification and control (vehicle control, process control), game-playing and decision making (backgammon, chess, racing), pattern recognition (radar systems, face identification, object recognition and more), sequence recognition (gesture, speech, handwritten text recognition), medical diagnosis, financial applications (automated trading systems), data mining (or knowledge discovery in databases, "KDD"), visualization and e-mail spam filtering.

Logic and networks; Logic networks

Among early results relating logic and neural networks were:

- Boolean networks - networks receiving and emitting boolean values and performing boolean functions. The Boolean functions could be pre-defined and given to a network; there exist networks that can learn how to perform Boolean functions from examples.
- XOR problem and perceptron.
- Finite Neural networks can simulate both Turing machines and Universal Turing machines.
- HALTING PROBLEM is proved to be solvable by infinite NNs.
- Neural networks and Automata; curcuits.

Example: NN - Automata (I.Alexander: Neurons and Symbols)

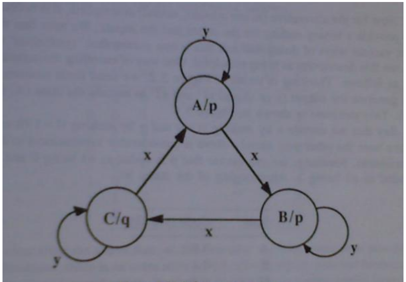


Figure 5.3 *A three-state automaton.*

Example: NN - Automata (I.Alexander: Neurons and Symbols)

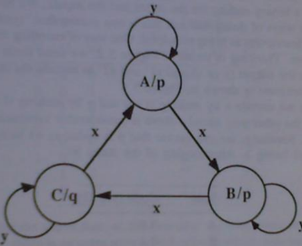


Figure 5.3 *A three-state automaton.*

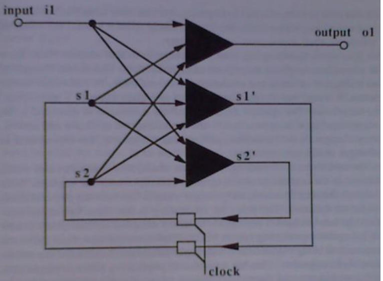


Figure 5.4 *The neural version of the automaton in Figure 5.3.*

Logic and NNs: summary

Finite Automata $\longrightarrow$ Binary threshold networks
Turing Machines $\longrightarrow$ Neural networks with rational weights
Probabilistic Turing Machines $\longrightarrow$ NNs with rational weights
Super-turing computations $\longrightarrow$ NNs with real weights.

Neuro-Symbolic Networks and Logic Programs

Logic Programs

- $A \leftarrow B_1, \ldots, B_n$

Neuro-Symbolic Networks and Logic Programs

Logic Programs

- $A \leftarrow B_1, \ldots, B_n$
- $T_P(I) = \{A \in B_P : \; A \leftarrow B_1, \ldots, B_n$
  is a ground instance of a clause in P and $\{B_1, \ldots, B_n\} \subseteq I\}$

Neuro-Symbolic Networks and Logic Programs

Logic Programs

- $A \leftarrow B_1, \ldots, B_n$
- $T_P(I) = \{A \in B_P : \; A \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in P and $\{B_1, \ldots, B_n\} \subseteq I\}$
- $\mathtt{lfp}(T_P \uparrow \omega) =$ the least Herbrand model of P.

Theorem

For each propositional program P, there exists a 3-layer feedforward neural network which computes T_P.

- No learning or adaptation;
- Require infinitely long layers in the first-order case.

A Simple Example

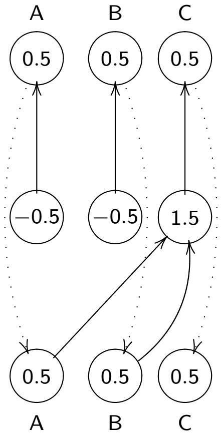
$B \leftarrow$
$A \leftarrow$
$C \leftarrow A, B$

$T_P \uparrow 0 = \{B, A\}$
$lfp(T_P) = T_P \uparrow 1 = \{B, A, C\}$

A Simple Example

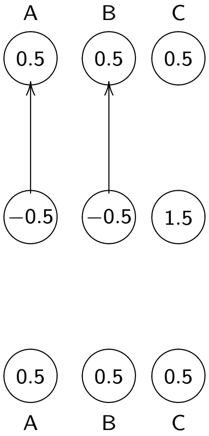
$B \leftarrow$
$A \leftarrow$
$C \leftarrow A, B$

$T_P \uparrow 0 = \{B, A\}$
$lfp(T_P) = T_P \uparrow 1 = \{B, A, C\}$



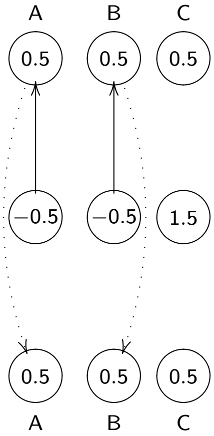
A Simple Example

$B \leftarrow$
$A \leftarrow$
$C \leftarrow A, B$
$T_P \uparrow 0 = \{B, A\}$
$lfp(T_P) = T_P \uparrow 1 = \{B, A, C\}$



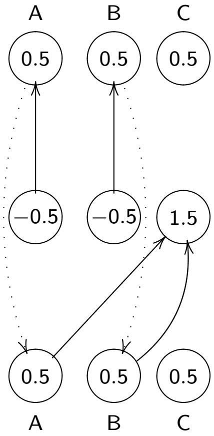
A Simple Example

$B \leftarrow$
$A \leftarrow$
$C \leftarrow A, B$
$T_P \uparrow 0 = \{B, A\}$
$lfp(T_P) = T_P \uparrow 1 = \{B, A, C\}$



A Simple Example

$B \leftarrow$
$A \leftarrow$
$C \leftarrow A, B$
$T_P \uparrow 0 = \{B, A\}$
$lfp(T_P) = T_P \uparrow 1 = \{B, A, C\}$



A Simple Example

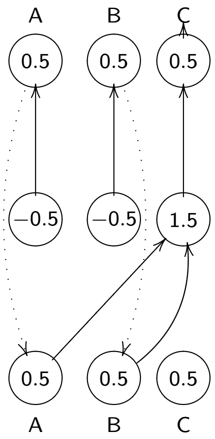
$B \leftarrow$

$A \leftarrow$

$C \leftarrow A, B$

$T_P \uparrow 0 = \{B, A\}$

$lfp(T_P) = T_P \uparrow 1 = \{B, A, C\}$



Another Example: First-Order Case

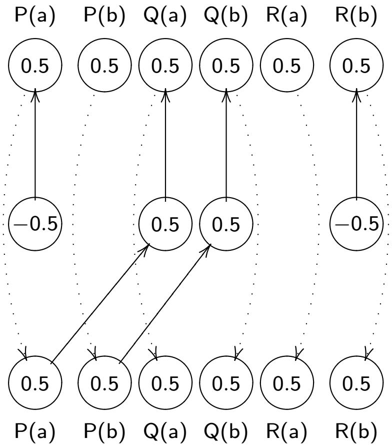
$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$

$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$

Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$

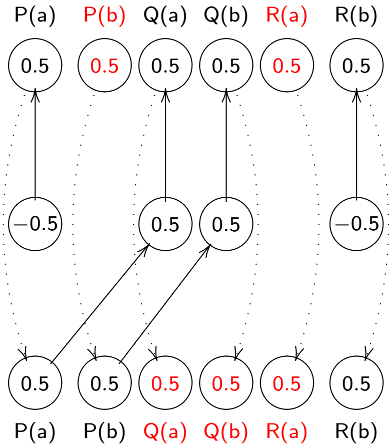
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$

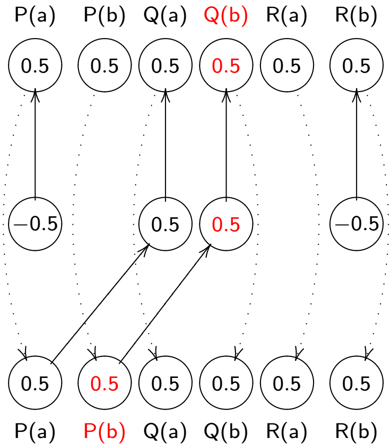
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$

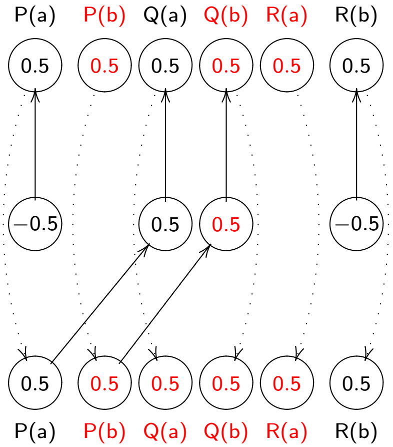
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



Another Example: First-Order Case

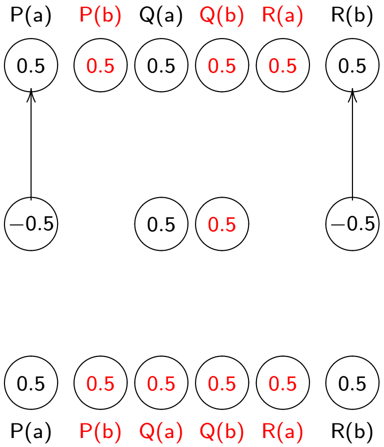
$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$

$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



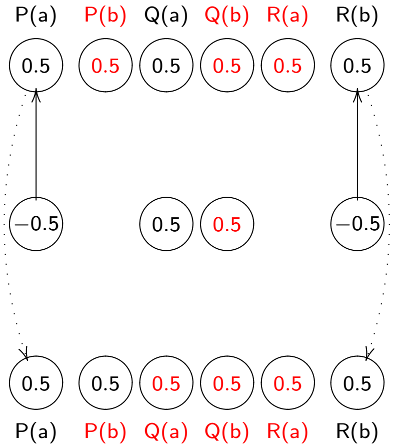
Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



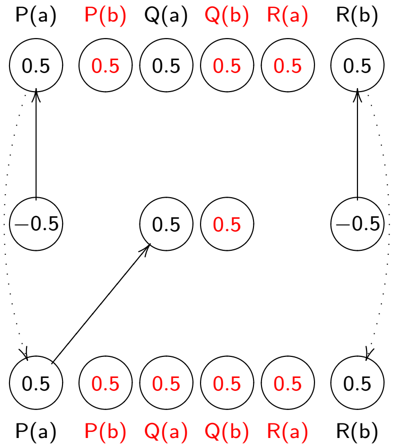
Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



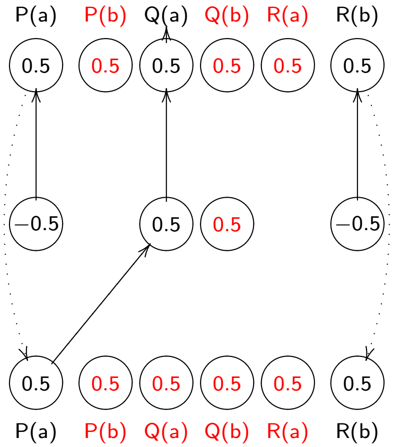
Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



Another Example: First-Order Case

$P(a) \leftarrow$
$Q(x) \leftarrow P(x)$
$R(b) \leftarrow$
$T_P \uparrow 0 = \{P(a), R(b)\}$
$lfp(T_P) = T_P \uparrow 1 =$
$\{P(a), R(b), Q(a)\}$



Example 3

$P(0) \leftarrow$
$P(s(x)) \leftarrow P(x)$

$T_P \uparrow 0 = \{P(0)\}$
$lfp(T_P) = T_P \uparrow \omega =$
$\{0, s(0), s(s(0)),$
$s(s(s(0))), \ldots\}$

Example 3

$P(0) \leftarrow$
$P(s(x)) \leftarrow P(x)$

$T_P \uparrow 0 = \{P(0)\}$
$lfp(T_P) = T_P \uparrow \omega =$
$\{0, s(0), s(s(0)),$
$s(s(s(0))), \ldots\}$

Example 3

$P(0) \leftarrow$
$P(s(x)) \leftarrow P(x)$

$T_P \uparrow 0 = \{P(0)\}$
$lfp(T_P) = T_P \uparrow \omega =$
$\{0, s(0), s(s(0)),$
$s(s(s(0))), \ldots\}$

Paradox:
(computability,
complexity,
proof theory)

Unification Algorithm

In my thesis, I suggested Neural networks of another architecture, so that a two-neuron network could perform a unification of two arbitrary first-order atoms.
Currently, I am implementing and testing these neural networks using MATLAB neural network simulator.
The most "inconvenient" part of the implementation is that conventional Neural networks do not accept anything but scalar numbers. They would not accept strings or lists, for example. Scalars are needed to mimic the 'unstructured' signals that excite a biological neuron.
A higher level of abstraction may be needed in the future?

Why is this interesting?

- Making use of parallelism when unifying atoms...
- Second-order unification: could the parallelism help?
- General interest of obtaining a neural network interpreter for logic programs.

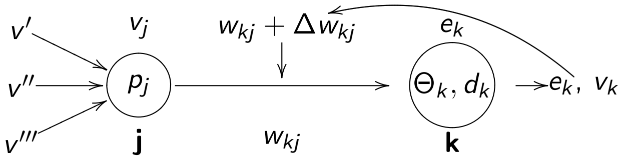
Definition of Kahn networks

Definition

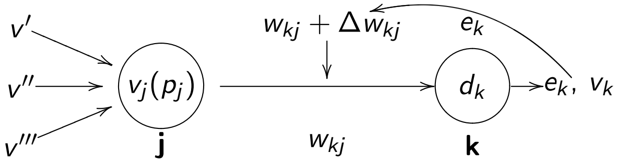
A Kahn network is a directed graph, where nodes represent processes or computing stations, and edges link the processes. The processes can be linked sequentially or in parallel. To each communication line, we assign a type of data that can be communicated. Each node can use the history of communications to produce an output, so it can be seen as a function from the streams of inputs to the streams of outputs. The function is continuous, which means that an output can be delivered without waiting for an infinite amount of information on the input lines.

Motivation: designed as a simple language for parallel programming, it was used later to give a semantics for parallel computations.

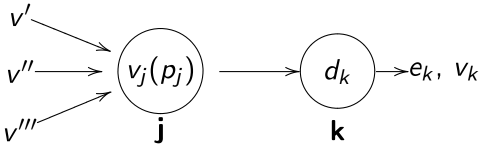
Visual transformation of a neural network into a Kahn network



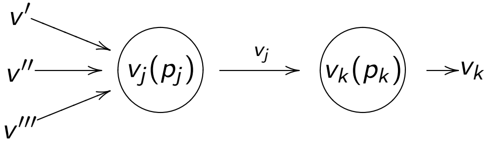
Visual transformation of a neural network into a Kahn network



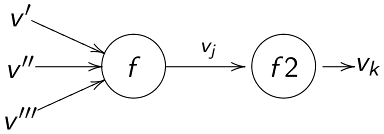
Visual transformation of a neural network into a Kahn network



Visual transformation of a neural network into a Kahn network


$$\begin{tikzcd} v' \arrow[dr] & & & \\ v'' \arrow[r] & \boxed{v_j(p_j)} \arrow[r, "v_j"] & \boxed{v_k(p_k)} \arrow[r] & v_k \\ v''' \arrow[ur] & & & \end{tikzcd}$$

Visual transformation of a neural network into a Kahn network



Visual transformation of a neural network into a Kahn network

$$\begin{tikzcd}
v' \arrow[dr] & & & \\
v'' \arrow[r] & f \arrow[r, "v_j"] & f2 \arrow[r] & v_k \\
v''' \arrow[ur] & & &
\end{tikzcd}$$

We specify the data types for channels v', v'', v''', v_j, v_k. E.g., N.
We associate continuous functions
$f : (v')^\omega \ * \ (v'')^\omega \ * \ (v''')^\omega -> {v_j}^\omega$ and $f2 : {v_j}^\omega -> {v_k}^\omega$ to the nodes. E.g., f outputs a stream whose nth element is the sum of nth elements of $(v')^\omega$, $(v'')^\omega$ and $(v''')^\omega$; and $f2$ outputs a stream whose nth elemnt is the sum of the first (n-1) elements.

Kahn network that computes Eratosthenes' sieve, [Kahn, Paulin-Mohring]

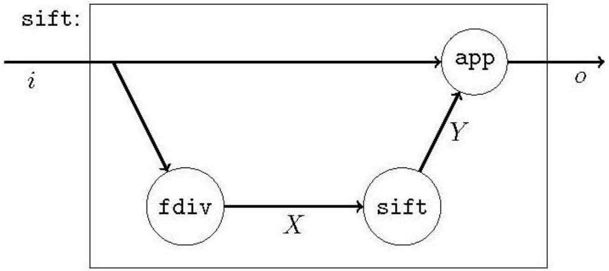


Figure 3: A Kahn network for the Sieve of Eratosthenes

Neural networks and Kahn networks: similarities

- The Structure: computing units with a function embedded in them, that are connected in parallel, and communicate signals in a particular direction.
- Neural networks have different functions, such as activation functions, transfer functions, learning functions, that are combined in a particular way and particular sequence, but their composition may be seen as one processing function in a Kahn network. And vice versa, a Kahn process function can be represented as a neuron whose transfer function is equal to the function used by Kahn networks, and all other functions are identities.

Neural networks and Kahn networks: similarities

- The Structure: computing units with a function embedded in them, that are connected in parallel, and communicate signals in a particular direction.
- Neural networks have different functions, such as activation functions, transfer functions, learning functions, that are combined in a particular way and particular sequence, but their composition may be seen as one processing function in a Kahn network. And vice versa, a Kahn process function can be represented as a neuron whose transfer function is equal to the function used by Kahn networks, and all other functions are identities.

If learning and reals are removed from the picture, and signals to the network are sent continuously, then one might prove the exact correspondence between the two models.

Neural networks and Kahn networks: differences

- Neural networks can use real or complex numbers.
- Neural networks do not accept structured data, such as lists and strings, for the reason that biological neurons can be only excited or inhibited, but there is no structure in the signals.
- There is learning in Neural networks.
- The Kahn networks were constructively formalised in Coq (Paulin-Mohring), using libraries for streams, corecursive functions and cpos. As for neural networks, following the tradition of NN implementations, such a formalisation could rely on libraries for matrices.

The notion of a "structure" of an input is the tricky one here: in what sense processing matrices is better than processing streams? Why networks processing streams are "logic" networks, and the nets processing matrices are "neural"? Can real numbers be seen as "streams"?

Thank you!