(Newtonian) Space-Time Algebra

J. E. Smith

NCAL Research Note 1

6/15/2019

1. Introduction

The *space-time* (*s-t*) *algebra* provides a mathematical structure for communicating and computing with values encoded as points in discretized linear time. Consequently, the input-output behavior of *s-t* functions must be consistent with the flow of Newtonian time. The *s-t* algebra, in effect, formally defines what it means to be "consistent with the flow of Newtonian time". Once formalized, the functional capabilities of the s-t algebra, and therefore the capabilities of temporal computing, can be fully explored.

The *s-t* algebra contrasts with conventional temporal algebras targeted at drawing conclusions regarding temporal relationships, i.e., *interval algebras* [1]. These algebras convert an interval of time into a spatial form which facilitates the analysis of temporal relationships using a spatial representation. This approach aligns with the spatial manner with which we (humans) tend to reason about time [2].

The *s-t* algebra operates at a lower level and is not based on a spatial representation of time. Rather, it employs a temporal computing paradigm, as might be used by biological neurons. With respect to the way it is intended to be used (the construction of computing networks), the *s-t* algebra is closer to Boolean algebra than it is to an interval algebra (primarily an analysis tool).

Note that this is a working document; some formal proofs remain to be done and constructions are informal.

1.1 Definition

Refer to Figure 1.

Definition: The *s-t algebra* is a bounded distributive lattice $\mathbb{S} = (N_0^{\infty}, <, \wedge, \vee, 0, \infty)$. $\mathbb{S}$ consists of a bottom element 0, a top element ∞, and the natural numbers. $\mathbb{S}$ is well-ordered and is closed under addition. □

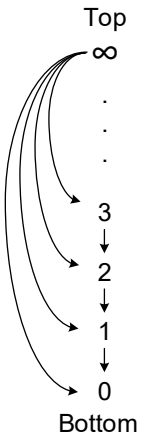


Figure 1. The *s-t* algebra is a bounded distributive lattice.

$\mathbb{S}$ is not complemented. Furthermore, subtraction cannot be performed because the algebra is closed only for addition, and all elements of the algebra are non-negative. When the algebra is given a temporal interpretation, both complementation and negation are tantamount to going backwards in time -- counter to the flow of Newtonian time.

By definition, $\wedge$ and $\vee$ are distributive, associative, commutative, and satisfy the absorption laws.

The top element is represented with the symbol "∞", not to be confused with the mathematical ∞. This symbol is chosen because the top element of the lattice has characteristics one would readily associate with an intuitive, conceptual "∞". E.g., $\infty = \infty + 1$.

1.2 Temporal Interpretation

The algebra is intended to support physical implementations of computing devices that communicate by encoding values with temporal events; for example, values communicated as the times of electrical transients. In some implementations transient events may be voltage pulses (*spikes*) transmitted over wires, or they may be changes in voltage levels (*edges*). In general, any method that relies on communication via transient temporal events may be used; photonic pulses communicated through free space is another example. In the remainder of this document, the term "spike" will be used as a generic term to denote a point in time.

2. Space-Time Functions

Of interest here is a certain class of functions defined over the *s-t* algebra. These functions take spikes as inputs, produce spikes as outputs, and the functional relationships between input spike times and output spike times are consistent with rules governing the flow of Newtonian time.

Definition: A function $z = F(x_1...x_q)$, $x_{1...}x_q$, $z \in N_0^\infty$, is a *Space-Time Function* if it satisfies the following:

1) *implementability:* F is implementable with a finite number of states.

2) *causality*: For all $x_j > z$, $F(x_1...,x_j,...x_q) = F(x_1...,\infty,...x_q)$, and if $z \neq \infty$, then $z \geq x_{\min}$.

3) *invariance*: $F(x_1 + 1, ..., x_q + 1) = F(x_{1,..}x_q) + 1$. □

The three properties are very general. *Implementability* assures that the function is implementable using a finite amount of physical state. The other two properties are consistent with the uniform passage of time. *Causality*: using the spike-based interpretation, an output spike can not be affected by input spikes that occur later in time. Furthermore, there are no spontaneous output spikes. *Invariance*: if all the input spike times uniformly "shift" by unit time, then the output spike time shifts by the same amount. Invariance naturally extends to any constant number of unit time shifts.

2.1 Primitive Space-Time Functions

In this section a set of *s-t* primitive functions or *operators* are defined in terms of the lattice's ordering relation $<$.

2.1.1 Unary Operators

The identity function, $a = a$, is one of two unary operators. The *increment* function, $a = b + 1$, is the other.

Defn: For $b \neq \infty$, $a = b + 1$ iff $b < a$ and there exists no $c < a$ such that $b < c$.
For $b = \infty$, $\infty = \infty + 1$.

2.1.2 2-ary Operators

Ordering relationships are a natural way of describing the 2-ary[1] operators. Figure 2 captures all such functions of two inputs. There are three disjoint relationships between inputs a and b: $a < b$, $a = b$, and $b < a$. For a given operation and for each of these three input relationships, there are only three possible outputs: a, b, or ∞. This would suggest a total of $3^3 = 27$ total functions. However, after accounting for symmetries and removing duplicates, there are 10 unique 2-ary operations; all are shown in Figure 2.

Commutative 2-ary operations: *min*, *max*, *x-min*, *x-max, equals*

Non-commutative 2-ary relationals: *not equals*, *less than*, *less or equals*, *greater*, *greater or equals*

For all of the relationals, if the stated relation is true, then the output is equal to one of the inputs (the first input, a, by convention here); if the relation is false, then the output is ∞. Hence, for example, $a \prec b$ and $b \succ a$ are not the same function.

$a < b$	$a = b$	$b < a$	function	name	symbol
a	$a=b$	b	if $a < b$ then a ; else b	*min*	$\wedge$
a	$a=b$	-	if $a \leq b$ then a ; else ∞	*less or equal*	$\preccurlyeq$
a	-	a	if $a \neq b$ then a ; else ∞	*not equal*	$\neq$
a	-	b	if $a < b$ then a else if $b < a$ then b ; else ∞	*exclusive min*	x$\wedge$
a	-	-	if $a < b$ then a ; else ∞	*less than*	$\prec$
b	$a=b$	a	if $a \geq b$ then a ; else b	*max*	$\vee$
b	-	a	if $a > b$ then a else if $b > a$ then b ; else ∞	*exclusive max*	x$\vee$
-	$a=b$	a	if $a \geq b$ then a ; else ∞	*greater or equal*	$\succcurlyeq$
-	$a=b$	-	if $a = b$ then a ; else ∞	*equal*	$\equiv$
-	-	a	if $a > b$ then a ; else ∞	*greater than*	$\succ$

Figure 2. All 2-ary *s-t* functions.

2.2 Symbols and Notation

Symbols for the primitive functions are shown in Figure 3. The symbol $\bowtie$ represents any of the non-commutative relational operations.

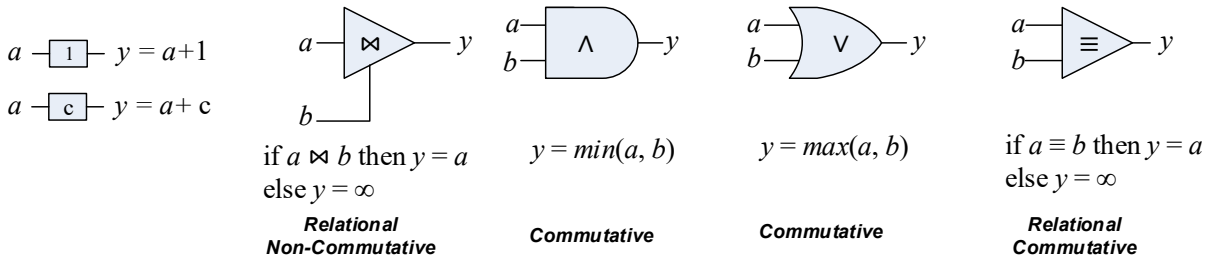


Figure 3. Symbols representing the various primitive operators that may be used in network schematics.

[1] Avoiding the term "binary" because of ambiguity with respect to binary encoded data.

In written notation, the asymmetry of inputs is represented by the order of the inputs; that is, $a \bowtie b$ is always interpreted as: "if $a \bowtie b$ then a; else ∞". The input asymmetry in the drawn relational symbols mnemonically express their non-commutative relationship.

3. Basic Identities and Theorems

As noted above, the $\wedge$ and $\vee$ are commutative, associative, distributive, and satisfy the absorption laws. There are also identities which involve the relations and the +1 function.

Basic Properties

1: $a \vee \infty = \infty$ ***lattice top/bottom***
2: $a \wedge 0 = 0$
3: $a \vee 0 = a$ ***identity***
4: $a \wedge \infty = a$
5: $a = a \wedge a$ ***idempotence***
6: $a = a \vee a$
7: $a \vee b = b \vee a$ ***commutative***
8: $a \wedge b = b \wedge a$
9: $a \vee (b \vee c) = (a \vee b) \vee c$ ***associative***
10: $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

11: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ ***distributive***
12: $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
13: $a \prec (b \wedge c) = (a \prec b) \vee (a \prec c)$
14: $a \prec (b \vee c) = (a \prec b) \wedge (a \prec c)$
15: $(a \vee b) \prec c = (a \prec c) \vee (b \prec c)$
16: $(a \wedge b) \prec c = (a \prec c) \wedge (b \prec c)$
17: $(a \wedge b) \prec a = b \prec a$ ***special case of 16***
18: $(a \prec b) \prec c = (a \prec b) \vee (a \prec c)$
19: $(a \bowtie_1 b) \bowtie_2 c = a \bowtie_1 b \wedge a \bowtie_2 c$ ***meta-theorem; holds for any relational operators***

20: $a \wedge (a \vee b) = a$ ***absorption***
21: $a \vee (a \wedge b) = a$
22: $a \wedge a{+}1 = a$
23: $a \vee a{+}1 = a + 1$

24: $(a \vee b) + 1 = (a + 1) \vee (b + 1)$ ***invariance***
25: $(a \wedge b) + 1 = (a + 1) \wedge (b + 1)$
26: $(a \prec b) + 1 = (a + 1) \prec (b + 1)$

27: $a \prec b \wedge a \prec b{+}1 = a \prec b{+}1$

pf:
$a \prec b \wedge a \prec b{+}1$
$= a \prec (b \vee b{+}1)$
$= a \prec b + 1$

Four theorems useful for reducing networks. They are general forms of a distributive property.

30: $(a \vee b) \prec (c \wedge d) = (a \prec c) \vee (a \prec d) \vee (b \prec c) \vee (b \prec d)$

Proof:

$(a \vee b) \prec (c \wedge d)$
$= (a \prec (c \wedge d)) \vee (b \prec (c \wedge d))$ *from* **15:** $(a \vee b) \prec c = (a \prec c) \vee (b \prec c)$
$= (a \prec c) \vee (a \prec d) \vee (b \prec c) \vee (b \prec d)$ *from* **13:** $a \prec (c \wedge d) = (a \prec c) \vee (a \prec d)$

31: $(a \vee b) \prec (c \vee d) = (a \prec c) \vee (a \prec d) \vee (b \prec c) \vee (b \prec d)$
$= (a \prec (c \vee d)) \vee (b \prec (c \vee d))$ *from* **15:** $(a \vee b) \prec c = (a \prec c) \vee (b \prec c)$
$= [(a \prec c) \wedge (a \prec d)] \vee [(b \prec c) \wedge (b \prec d)]$ *from* **14:** $a \prec (b \vee c) = (a \prec b) \wedge (a \prec c)$

32: $(a \wedge b) \prec (c \wedge d) = [(a \prec c) \vee (a \prec d)] \wedge [(b \prec c) \vee (b \prec d)]$
$= (a \prec (c \wedge d)) \wedge (b \prec (c \wedge d))$ *from* **16:** $(a \wedge b) \prec c = (a \prec c) \wedge (b \prec c)$
$= [(a \prec c) \vee (a \prec d)] \wedge [(b \prec c) \vee (b \prec d)]$ *from* **13:** $a \prec (c \wedge d) = (a \prec c) \vee (a \prec d)$

33: $(a \wedge b) \prec (c \wedge d) = [(a \prec c) \wedge (a \prec d)] \wedge [(b \prec c) \wedge (b \prec d)]$
$= (a \prec (c \wedge d)) \wedge (b \prec (c \wedge d))$ *from* **16:** $(a \wedge b) \prec c = (a \prec c) \wedge (b \prec c)$
$= (a \prec c) \wedge (a \prec d) \wedge (b \prec c) \wedge (b \prec d)$ *from* **14:** $a \prec (c \wedge d) = (a \prec c) \vee (a \prec d)$

These two theorems allow the collapsing of $\prec$ chains of the nasty type from two deep to one deep. The expression on the left contains two levels of relationals; the expression on the right contains only one.

34: $a \prec (b \prec c) = (a \prec b) \wedge ((a \succcurlyeq b) \vee (b \succcurlyeq c))$

35: $a \prec (b \succcurlyeq c) = (a \prec b) \wedge ((a \succcurlyeq b) \vee (b \prec c))$

4. Space-Time Computing Networks

Lemma 1: Any function implemented as a non-recurrent composition of space-time functions is a space-time function.

proof outline: A proof begins with a topological sort of the directed graph implied by the non-recurrent composition. Then, the proof proceeds by induction on the sequence of elements in the topological sort. □

Definition: A *Space-Time Computing Network* is a non-recurrent (feedforward) interconnection of space-time functional blocks. Each block implements an implementable, causal, and invariant function. □

From Lemma 1, all space-time computing networks implement space-time functions, so if the designer begins with a set of basic operators that implement space-time functions and the operators are interconnected in any feedforward manner, the overall system must implement a space-time function.

4.1 Primitive Completeness

All the 2-ary operations can be constructed from only three primitives +1, $\wedge$, and $\prec$. I.e., these three primitives are *functionally complete* with respect to all the 2-ary functions. For example, Figure 4 shows constructions of $\succcurlyeq$, $\preccurlyeq$, $\vee$, and $\equiv$ from these three. Similar constructions for all the other 2-ary operators are straightforward.

From the perspective of implementing useful designs, network designers should consider *all* 2-ary functions to be at their disposal -- to be used freely. Their implementation costs will likely be similar.

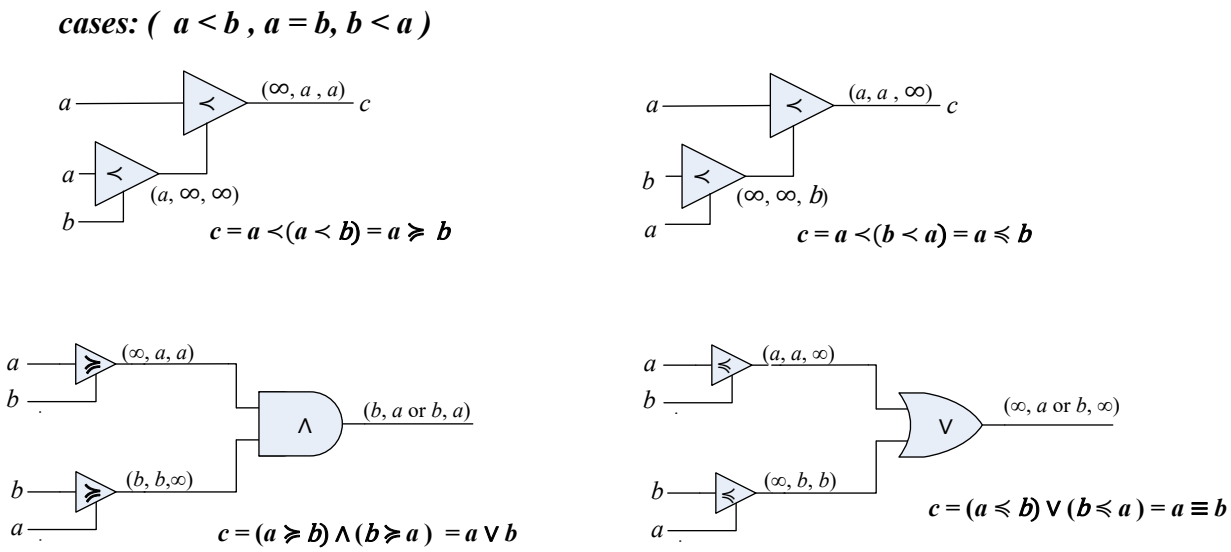


Figure 4. Examples of functional completeness for primitives +1, ∧, and ≺ follow: ≽ is implemented using only ≺. ≼ is implemented using only ≺. ∨ is implemented using only ≽ and ∧. ≡ is implemented using only ≼ and ∨.

Theorem: all the primitive operators in Figure 2 can be implemented using only +1, ∧, and ≺.

proof outline: by construction; some of the constructions are shown in Figure 4.

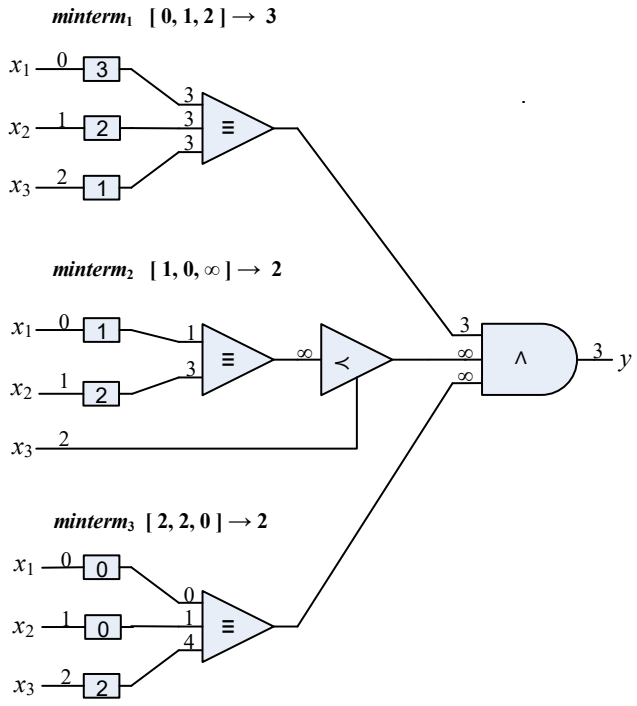
4.2 Bounded Functional Completeness

A way of specifying one class of *s-t* functions, *bounded s-t functions*, is to use a function table (FT), analogous to a Boolean truth table. Figure 5a is a small example. The table shown is in *normal form*. A bounded FT has a finite number of entries and is *normalized* if 1) at least one input in each row is a 0 and 2) the output is not ∞. All normalized input combinations that yield a non-∞ output have an entry in the table.

An important observation is that although a normalized function table contains only a finite number of rows, it specifies a total function over the infinite set N_0^∞ due to the invariance property. Given a normalized table, to compute the output for an arbitrary unnormalized input vector, one should first normalize the input vector by subtracting x_{min} from each element. If the normalized vector is in the table, then x_{min} should be added to the corresponding table entry to yield the output value. If the normalized entry is not in the table, then by definition the output is ∞.

x_1	x_2	x_3	y
0	1	2	3
1	0	∞	2
2	2	0	2

a) Function Table



b) Implementation

Figure 5. Example of minterm implementation.

To show completeness for functions specified by a bounded FT, one can construct an implementation for any bounded normalized FT.

Theorem: The primitive operations +1, $\prec$, $\equiv$, and $\wedge$ are complete for *s-t* functions specified as bounded FTs.

proof outline: by construction; very straightforward, but TBD. Figure 5b is an example for the given normalized FT.

Corollary: The primitives +1, $\prec$, and $\wedge$ are complete for *s-t* functions with bounded FTs.

proof outline: $\equiv$ can be implemented with $\wedge$ and $\prec$, by construction given in Figure 2.

Consequently, three of the primitives are sufficient for functional completeness; the primitive operators are a matter of convenience and intuitive expression.

Observe that this completeness result holds only for functions with bounded FTs. Consequently, this does not include all of the *s-t* functions and excludes some important ones. For example, the *max* function $a \vee b$ cannot be specified by a bounded FT.

5. Discussion and Open Problems

Completeness

The FT specifies *s-t* functions in a more-or-less conventional way -- rows consisting of input values and an associated output value. For bounded FT tables, +1, $\prec$, and $\wedge$ are functionally complete. The set of functions that can be represented via a bounded FT table is limited, however; neither $\wedge$ nor $\vee$ can be specified with a bounded FT. The same goes for many other simple functions like $(a \wedge b) \vee (c \vee d)$.

It is conjectured that the primitive *s-t* operators are complete for *s-t* functions with finite implementations, but completeness has only been shown for specifications involving bounded tables having a finite number of entries. This is not the more general completeness result we would like to have. For example, there may exist implementable *s-t* functions that cannot be specified via a bounded table of any form.

Open problem: are the primitive operators +1, $\prec$, and $\wedge$ complete for the entire set of implementable s-t functions?

Sub-problem: are the primitive operators +1, $\prec$, and $\wedge$ complete for the entire set of three-input s-t functions? This can be answered with an exhaustive approach, if necessary.

When considering these problems, we would like the broadest practical definition of "implementable". Begin with the minimal restriction that tangible physical resources in any implementation must be strictly finite and no physical resource can constructed to infinitesimal tolerances. Consequently, any particular implementation (and therefore its associated function) is restricted to a fixed number of input and output variables, each of which can take on a value from a discrete set (not the continuum). A large scale digital computer system satisfies these constraints, for example.

For Boolean functions, a truth table with a finite number of inputs, each of which can take a finite number of values (0 and 1) can specify all implementable Boolean functions. By using a minterm canonical form, for example, we can show that AND, OR, and NOT are functionally complete. Further, by showing that AND, OR, and NOT can all be implemented via a NAND, NAND alone is functionally complete.

Because *s-t* functions encode values as points in time (not a tangible physical resource), the function domains and ranges, although discrete, are not bounded. Hence, we may not necessarily assume that a bounded table of some kind can specify all implementable *s-t* functions.

Taking all the above into account, it is strongly conjectured that +1, $\prec$, and $\wedge$ are complete for the full set of implementable *s-t* functions.

Developing a Standard Form

A related problem is the reduction of an arbitrary feedforward network composed of the *s-t* primitives to a standard form akin to a two-level Boolean network. This is not a canonical form; a canonical form is yet to be defined. Rather it is a simple, well-structured way of representing functions.

Although details remain, one approach for arriving at a standard form is:

1) Reduce all 2-ary operators to +1, $\prec$, $\wedge$, and $\vee$.

2) Expand to network with fanout only at primary inputs.

3) Push all delays (δ) back to primary inputs using invariance ***24***, ***25***, ***26.***

4) Push $\prec$ operators toward inputs using ***30-33***. If internal fanouts are created, remove them by expanding subnetworks to primary inputs.

5) After step 4) the network will consist of layers of 1) fanned-out primary inputs feeding delays, 2) chains of $\prec$ operators, 3) a multi-level network consisting of $\wedge$, and $\vee$.

6) the $\prec$ chains can be reduced via theorems 18, 34, 35. After reduction, there is a single level of relational operators. However, in the reduction process an additional relational ($\succcurlyeq$) is introduced. That is, the standard form incorporates five operators: δ, $\prec$, $\succcurlyeq$, $\wedge$, $\vee$.

7) The $\wedge$, $\vee$ network can then be reduced to two levels using the associative, commutative, and distributive properties.

The final network will look like the example in Figure 6.

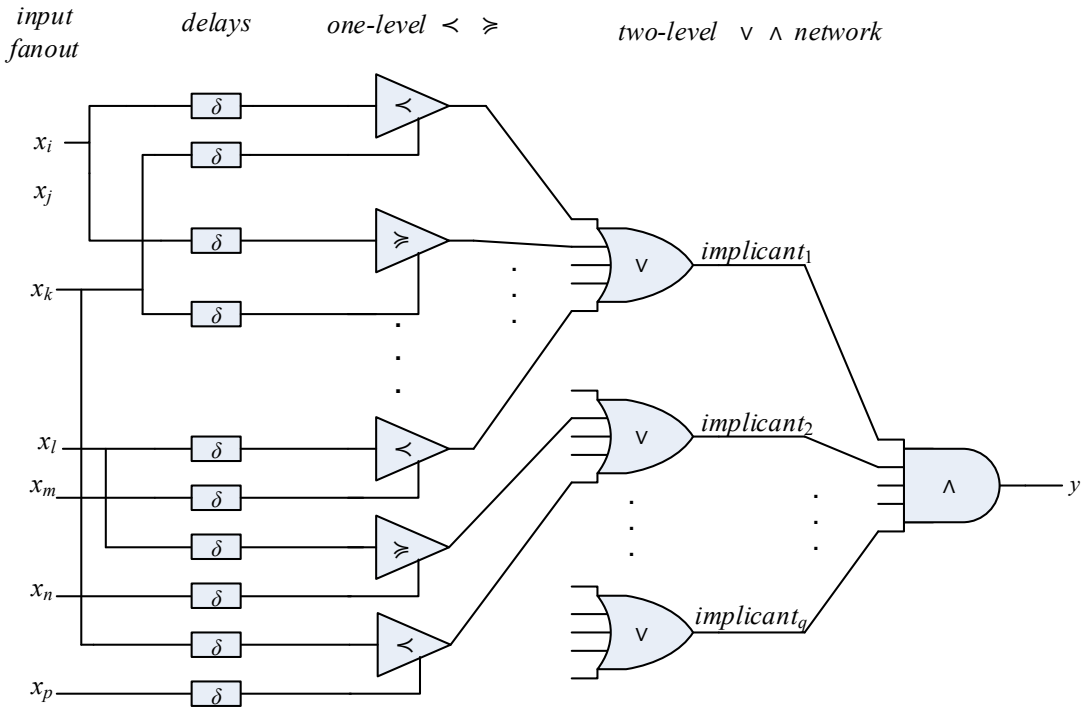


Figure 6. Proposed standard form: all feedforward *s-t* networks can be reduced to this form.

A non-causal function

To illustrate the limits of the *s-t* algebra, consider the following function: if $a < b < c$ then $f = a$; else $f = \infty$. (Note: "<" is the lattice-defining relation, not be confused with the relational operator "$\prec$" .) Although f seems like a reasonable temporal function, it is not causal so it is not an *s-t* function. If f evaluates to a, then a causal f can only use information available up until time a, but not after. At the time a arrives, it can be determined whether $a < b$ and $a < c$. However, this not good enough to determine that $f = a$. In order to determine $f = a$, one would have to look into the future to determine the eventual relationship between b and c.

6. The XOR Issue

An important part of artificial neural network lore is that after the first weighted artificial neurons were proposed -- *perceptrons* -- Minsky and Papert [3] showed that a perceptron can not implement all Boolean functions. In particular, XOR and its complement, XNOR (or equivalence), can not be implemented. Hence, perceptrons were generally considered to be deficient, and, according to the lore, this led to a long connectionist "winter". The apparent shortcoming was eventually resolved via multiple layer perceptrons, and artificial neural networks re-emerged as an active research topic in the late 1980s.

Given its significance in the development of ANNs, consider the XOR problem in the context of *s-t* functions. The issue to be addressed is whether XOR, or XOR-like, functions can be implemented using *s-t* operations.

The conventional XOR *logic* function is defined over a lattice using logical operators. In contrast, the *s-t* algebra is defined over a lattice restricted to operations consistent with the flow of time: causality and invariance.

Consider the *s-t* algebra with elements N_0^∞ and a set of causal and invariant operators. Although the following discussion holds for all points in the algebra, for simplicity, consider only 0 and ∞. Then, a temporal XOR function is given in Table 1.

Table 1. Temporal XOR

x_1	x_2	z
0	0	0
0	∞	∞
∞	0	∞
∞	∞	0

The temporal XOR function is not causal. In the bottom row, there is an output spike at $t = 0$ if neither of the inputs has a spike at any time in the future. Because it is not causal, there is no *s-t* network that will implement this function.

Next, consider the temporal XNOR. In Boolean terms, it is the complement of the XOR. The temporal XNOR is given in Table 2. (The full function is: *if* $x_1 \equiv x_2$ *then* $z = \infty$; *else* $z = x_1$.)

Table 2. Temporal XNOR

x_1	x_2	z
0	0	∞
0	∞	0
∞	0	0
∞	∞	∞

The temporal XNOR function is *both* causal and invariant, and therefore can be implemented as an *s-t* network.

Consequently, although there is no *s-t* XOR analog, there is a temporal XNOR analog. This is certainly better than not being able to implement either, and a temporal XNOR alone may be adequate for supporting useful *s-t* functions, especially the neocortical ones.

The *s-t* algebra is not functionally complete: one cannot implement all functions with ranges and domains of N_0^∞. But in the big picture, why should we care about completeness -- i.e. being able to implement all members of an extremely large set of functions? If the only functions we really care about are the neocortical functions, then we may not need temporal XOR capabilities (i.e., the ability to look into the future!).

Small notes

In general, for a function to be causal, the all-∞ input must always yield an ∞ output. This property holds for the XNOR but not the XOR.

In Boolean algebra, the XNOR is an XOR with an inverter. In the *s-t* algebra there is no inverter, because inversion is tantamount to going backward in time.

7. References

[1] Allen, James F. *"Maintaining knowledge about temporal intervals"* Communications of the ACM. ACM Press: 832–843, November 1983.

[2] Casasanto, Daniel, and Lera Boroditsky. "Time in the mind: Using space to think about time." *Cognition* 106, no. 2 (2008): 579-593.

[3] M. L. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass., 1969.