Vector space of finite prefix trees for dataflow matrix machines

Michael Bukatin

HERE North America LLC, Burlington, MA

Joint work with Jon Anthony

- - -

51st Spring Topology and Dynamical Systems Conference,
Special Session on Topology and Computer Science, March 10, 2017

Electronic coordinates

These slides are linked from my page on partial inconsistency and vector semantics of programming languages:

http://www.cs.brandeis.edu/∼bukatin/partial_inconsistency.html

E-mail:

`bukatin@cs.brandeis.edu`

(This is a continuation of my Leicester SumTopo 2016 talk.
I'll repeat some slides, while trying to minimize the overlap.)

Outline

Algebraic extensions and partial inconsistency

Algebraic extension of reals with respect to multiplication leads to imaginary numbers, and then to complex numbers.

Algebraic extension of the monoid of interval numbers with respect to addition leads to overdefined pseudosegments of negative length, such as $[3, 2]$, and then to the vector space of Warmus numbers.

Algebraic extension of the probabilistic measures over X with respect to multiplication by real scalars leads to positive and negative measures over X, and then to the vector space of signed measures over X.

In this fashion, extensions resulting in vectors spaces give rise to partially inconsistent elements, such as negative probabilities and pseudosegments of negative length.

Partial inconsistency landscape

- Negative distance/probability/degree of set membership
- Bilattices
- Partial inconsistency
- Non-monotonic inference
- Bitopology
- $x = (x \wedge 0) + (x \vee 0)$ or $x = (x \wedge \perp) \sqcup (x \vee \perp)$
- Scott domains tend to become embedded into vector spaces
- Modal and paraconsistent logic and possible world models
- Bicontinuous domains
- The domain of arrows, $D^{Op} \times D$ or $C^{Op} \times D$

Warmus numbers

Start with interval numbers, represented by ordinary segments.

Add pseudosegments $[a, b]$, such that $b < a$.

This corresponds to contradictory constraints, $x \leq b \& a \leq x$.

The new set consists of segments and pseudosegments.

Addition: $[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$.

True minus: $-[a, b] = [-a, -b]$.

$-[a, b] + [a, b] = [0, 0]$.

This gets us a group and a 2D vector space.

True minus is antimonotonic

$x \sqsubseteq y \Rightarrow -y \sqsubseteq -x$.

True minus maps precisely defined numbers, $[a, a]$, to precisely defined numbers, $[-a, -a]$.

Other than that, true minus maps segments to pseudosegments and maps pseudosegments to segments.

In the bicontinuous setup, true minus is a bicontinuous function from $[R]$ to $[R]^{Op}$ (or from $[R]^{Op}$ to $[R]$).

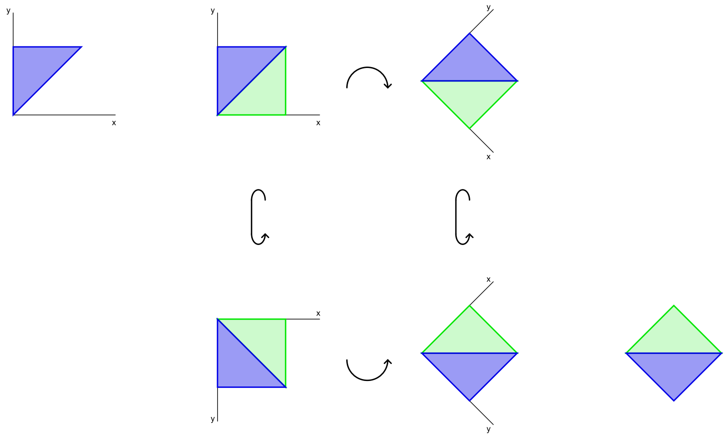
Multiple rediscoveries

Known under various names: Kaucher interval arithmetic, directed interval arithmetic, generalized interval arithmetic, modal interval arithmetic, interval algebraic extensions, etc.

First mention we know: M. Warmus, Calculus of Approximations. Bull. Acad. Pol. Sci., Cl. III, 4(5): 253-259, 1956, `http://www.cs.utep.edu/interval-comp/warmus.pdf`

A comprehensive repository of literature on the subject is maintained by Evgenija Popova: The Arithmetic on Proper & Improper Intervals (a Repository of Literature on Interval Algebraic Extensions),
`http://www.math.bas.bg/~epopova/directed.html`

From Cartesian to Hasse representation



Partially inconsistent interval numbers as a domain of arrows

$$[R] = \mathbb{R} \times \mathbb{R}^{Op}$$

(There is a tension between the group structure on $\mathbb{R}$ and $[R]$ and the axioms of domains requiring $\perp$ and $\top$ elements which can be satisfied by restricting to a segment of reals, or by adding $-\infty$ and $+\infty$. I am mostly being ambiguous about this in this slide deck, but this is something to keep in mind.)

Bitopology and *d*-frames

Achim Jung, M. Andrew Moshier. On the bitopological nature of Stone duality. Technical Report CSR-06-13. School of Computer Science, University of Birmingham, December 2006, 110 pages.

d-frame for the (lower, upper) bitopology on $\mathbb{R}$

d-frame elements are pairs $\langle L, U\rangle$ of open rays, $\langle(-\infty, a), (b, +\infty)\rangle$ (a and b are allowed to take $-\infty$ and $+\infty$ as values).

Non-overlapping pairs of open rays are consistent ($a \leq b$), overlapping pairs of open rays ($b < a$) are total.

Correspondence with partially inconsistent interval numbers

The bilattice isomorphism between d-frame elements and partially inconsistent interval numbers with "infinity crust":
$\langle(-\infty, a), (b, +\infty)\rangle$ corresponds to a partially inconsistent interval number $[a, b]$.

Consistent, i.e. non-overlapping, pairs of open rays ($a \leq b$) correspond to segments. Total, i.e. covering the whole space, pairs of open rays ($b < a$) correspond to pseudosegments.

Logic for fuzzy paraconsistent mathematics

Traditional fuzzy math: logic based on $[0, 1]$.

Paraconsistent math: logic based on the 4-valued bilattice.

Fuzzy paraconsistent math: logic based on the bilattice of Warmus numbers (probably within [0,1] or within [-1,1], or all reals with added “infinity crust”).

References

Section 4 of Bukatin and Matthews, *Linear models of computation and program learning*, GCAI 2015, EasyChair Proceedings in Computing, **36**, pages 66–78, 2015, `http://easychair.org/publications/download/Linear_Models_of_Computation_and_Program_Learning` (**"Tbilisi paper"**)

Slides of my November 2014 talk at Kent State University: `http://www.cs.brandeis.edu/~bukatin/PartialInconsistencyProgressNov2014.pdf`

Monotonic evolution of Warmus numbers by additions

Consider $x \sqsubseteq (x + x_1) \sqsubseteq (x + x_1 + x_2) \sqsubseteq \ldots$

Then every $x_i = [a_i, b_i]$ must be a pseudo-segment anti-approximating zero:

$[0,0] \sqsubseteq [a_i, b_i]$, that is $b_i \leq 0 \leq a_i$.

(`https://arxiv.org/abs/1610.00831`, Appendix A)

Rectifiers and quasi-metrics

It was typical to use sigmoid non-linearities as activation functions in neural nets, but a few years ago people discovered that **ReLU** (rectified linear units) often work much better:
$\mathbf{ReLU}(x) = max(0, x)$.

This is an integral of the Heaviside step function. Lack of smoothness at 0 does not seem to interfere with gradient methods, and otherwise it's nice when the derivatives are so simple.

Our standard quasi-metrics on reals are closely related to ReLU:

$q_1(x, y) = \mathbf{ReLU}(x - y) = q_2(y, x)$.

(`https://arxiv.org/abs/1610.00831`, Appendix B)

Linking two previous slides together

$$x \sqsubseteq (x + [\mathbf{ReLU}(a_1), -\mathbf{ReLU}(-b_1)]) \sqsubseteq$$
$$(x + \mathbf{ReLU}(a_1), -\mathbf{ReLU}(-b_1)] + [\mathbf{ReLU}(a_2), -\mathbf{ReLU}(-b_2)]) \sqsubseteq \dots$$

(The first new observation in this talk compared to Leicester)

[cf. the use of anti-monotonic involutions to perform anti-monotonic inference in Section 4.14, "Computational Models with Involutions", of the Tbilisi paper]

Concatenated ReLU

$x \mapsto (\mathbf{ReLU}(x), \mathbf{ReLU}(-x))$ from
Wenling Shang et al., `https://arxiv.org/abs/1603.05201`

We can think about this as “incorporating both (dual to each other) quasi-metrics on the reals”.

Addresses the ”problem of dying ReLUs” discussed in
[`https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b`]

In terms of scalar neurons, this is a neuron with two outputs.

TensorFlow authors thought **crelu** to be important enough to include it into their very short list of built-in activation functions, starting from version 0.11 (**discuss difficulties related to shape**).

Warmus numbers and split-complex numbers

Depending on how one defines multiplication, Warmus numbers can be viewed as the ring of split-complex numbers.

[D. H. Lehmer (1956) Review of "Calculus of Approximations" from Mathematical Reviews,
`http://www.ams.org/mathscinet/pdf/81372.pdf`]

see also `https://en.wikipedia.org/wiki/Split-complex_number#History`
and
`https://en.wikipedia.org/wiki/Interval_(mathematics)#Topological_algebra`

Neural networks as a model of computations

Computations with **streams of numbers**.

Straightforward extensions of recurrent neural networks are Turing-complete.

There is a variety of mechanisms to provide unbounded memory required for Turing completeness:

- Neural net as a controller for external memory [Pollack 1987, and a lot of modern research]
- Real numbers of unlimited precision [Sontag and Siegelmann, early 1990-s]
- Countable-sized neural net, with finite and possibly expanding part of it being active at any given moment of time [approach of **dataflow matrix machines**]

Recurrent neural networks - the core part

A finite set of neurons indexed by $i \in I$.

"Two-stroke engine":

"Up movement": for all i, $y_i := f(x_i)$.

"Down movement": for all i, $x_i := \sum_{j \in I} w_{ij} * y_j$.

(w_{ij}) is the matrix of weights.

(Core part only, input and output omitted.)

Dataflow matrix machines as a generalization of RNNs

Arbitrary linear streams.

A finite or countable collection of available **kinds of linear streams**.

A finite or countable collection of **neuron types**.

Each neuron type:

- a nonnegative input arity,
- a nonnegative output arity,
- a particular kind of linear streams is associated with each input and each output,
- a particular *built-in stream transformation*.

Dataflow matrix machines as a generalization of RNNs

Countable collection of neurons of each type.

Hence countable number of inputs x_i and outputs y_j.

Take countable matrix of weights with finite number of non-zero elements, and in particular make sure that w_{ij} can be non-zero only if the same kind of linear streams is associated with x_i and y_j.

Only neurons with at least one nonzero input or output weight are **active**, otherwise we keep them silent and treat their outputs as zeros. Hence only a finite number of neurons are active.

"Two-stroke engine"

"Up movement": for all active neurons C, $y_{1,C}, ..., y_{n,C} := f_C(x_{1,C}, ..., x_{m,C})$.

n, m, and f_C correspond to the type of the neuron C.

"Down movement": for all inputs i having non-zero weights associated with them, $x_i := \sum_{\{j \mid w_{ij} \neq 0\}} w_{ij} * y_j$.

RNNs as a general-purpose programming platform

RNNs are Turing-universal (unbounded memory is required).

However they are not a convenient general-purpose programming language, but belong to the class of

`https://en.wikipedia.org/wiki/Esoteric_programming_language`

and

`https://en.wikipedia.org/wiki/Turing_tarpit`

together with many other elegant and useful Turing-universal systems such as Conway's Game of Life and LaTeX.

DMMs as a general-purpose programming platform

DMMs are much more powerful than RNNs:

- arbitrary linear streams
- neurons with multiple input arity
- selection of convenient built-in transformations
- friendliness of DMMs towards sparse vectors and matrices
- self-referential facilities
- approximate representations of infinite-dimensional vectors (samples from probability distributions and signed measures)

Trend towards similar generalizations in machine learning

Trend towards speaking in terms of layers (i.e. non-sparse vectors) instead of individual neurons.

Trend towards implicit (and, occasionally, explicit) use of neurons with multiple inputs and outputs.

E.g. the usefulness of activation function $(x, y) \mapsto x * y$ was understood long ago [e.g. Pollack 1987], but it turns out that it is used implicitly in modern recurrent neural net architectures, such as LSTM and Gated Recurrent Unit networks: [`https://arxiv.org/abs/1610.00831`, Appendix C]

Reducing complexity of dataflow matrix machines

Many **kinds** of linear streams. Many **types** of neurons, with different input and output arities, and with different kinds of linear streams associated with each input and output.

Can we make this less complicated?

Consider a countable set L (in practice, a set of legal hash keys in your favorite programming language).

Consider finite ordered sequences of non-negative length of the elements of L (**paths**). Denote the set of those paths as P.

The vector space V is the space of formal finite linear combinations of elements of P.

Finite prefix trees = recurrent maps

V is the space of finite prefix trees with intermediate nodes from L and non-zero numbers as leaves.

V can also be understood as a space of recurrent maps, where elements of L are mapped into pairs of a number and an element of V (if an element is mapped into the pair of zeros, we omit it from the map description, and we consider the maps of finite overall description).

Finite prefix trees = "mixed rank tensors"

V can also be understood as a space of sparse "mixed rank tensors" with finite number of non-zero coordinates, where "tensor of rank N" is understood simply as an N-dimensional matrix, as it is customary in machine learning.

If the only path in the tree is of the length 0, we think about the vector in question as a scalar (tensor of rank 0).

If all paths are of the length 1, we think about the vector in question as a one-dimensional vector (tensor of rank 1).

If all paths are of the length 2, we think about the vector in question as a matrix (tensor of rank 2).

Et cetera...

Finite prefix trees = "mixed rank tensors"

Now consider an arbitrary finite prefix tree from V. It contains paths of various lengths. Paths of the length 1 if any correspond to coordinates of a one-dimensional vector, paths of the length 2 if any correspond to the elements of a matrix, etc.

Therefore, we call an arbitrary v from V a **"mixed tensor"**. It can contain coordinates for tensors of rank 1, 2, 3, etc, together with a scalar (the coordinate for the tensor of rank 0) at the same time.

(The alternative terminology might be a **"mixed rank tensor"**.)

Variadic neurons

The remaining element of complexity is different input and output arities in different neuron types.

The space V is rich enough to enable us to represent variadic neurons (neurons with variable input and output arities) via non-linear transformations of V with one argument and one result, by dedicating the first layer of keys of a recurrent map to serve as names of inputs and outputs.

Therefore a type of neuron is simply a (generally non-linear) transformation of V.

Variadic neurons

The network matrix columns naturally have a three-level hierarchy of indices (type of neuron, name of neuron of a given type, name of an output of a given neuron).

So the natural structure of a matrix row in this case is a sparse “tensor of rank 3”.

The network matrix rows themselves also naturally have a three-level hierarchy of indices (type of neuron, name of neuron of a given type, name of an input of a given neuron).

Therefore the natural structure of the matrix itself in this case is a sparse “tensor of rank 6”.

Dataflow matrix machines: “two-stroke engine”

V - vector space of prefix trees.

Countable number of functions, $f \in F, f : V \to V$.

For each f, countable number of neuron names, $n_f \in N_F$.

"Up movement": for all $f \in F$, all $n_f \in N_f$, $y_{f,n_f} := f(x_{f,n_f})$.

"Down movement": for all $f \in F$, all $n_f \in N_f$, all input names $i \in L$, $x_{f,n_f,i} := \sum_{g \in F} \sum_{n_g \in N_g} \sum_{o \in L} w_{f,n_f,i,g,n_g,o} * y_{g,n_g,o}$.

$(w_{f,n_f,i,g,n_g,o})$ is the “matrix” (6D-tensor) of weights usually obtained via the self-referential mechanism discussed below, computations are only done for the active part of the network.

Discussion

Last year Ralph Kopperman remarked that one way to look at dataflow matrix machines is to think about them as a formalism allowing to handle functions with variable number of arguments.

With this new formalism, single neurons also have this ability.

Discussion

Last year Andrey Radul formulated a principle stating that there is no reason to distinguish between a neuron and a subnetwork, and that it is a desirable property of a model to not have a difference between a generalized neuron and a subnetwork.

The formalism of vector space of finite prefix trees and variadic neurons brings us closer to fulfilling this principle.

Generalizations

Sometimes, we would like vectors for which even this generality is not enough, e.g. signed measures over some X (which we computationally might present via samples from X).

For those cases, one solution is to take $\mathbb{R} \oplus M$ instead of $\mathbb{R}$ as the space of leaves of finite prefix trees (e.g. M can be the space of signed measures over X).

Self-referential facilities

Self-modifying dynamical systems/neural nets/continuous programs.

Allow the kind of linear streams of countably-sized matrices (w_{ij}) with finite number of non-zero elements.

Introduce neuron `Self` having a stream of matrices (w_{ij}) on its output and use the current last value of that stream as the network matrix (w_{ij}) during the computations on each "down movement":

$x_i := \sum_{\{j \mid w_{ij} \neq 0\}} w_{ij} * y_j$.

Self-referential facilities

In the first version of our formalism, `Self` has a single input taking the same kind of stream of matrices and the identity transformation of streams, so it just passes its input through.

Its output is connected with weight 1 to its input, hence it is functioning as an accumulator of additive contributions of other neurons connected to its input with non-zero weights.

Self-referential facilities

Now we think, it is more convenient to have two separate inputs for `Self`, x_W and $x_{\Delta W}$, connect the output of `Self` y_W to x_W with weight 1, take additive contribution from other neurons at the $x_{\Delta W}$ input, and compute $x_W + x_{\Delta W}$ on the "up movement".

This is the mechanism we propose as a replacement of untyped lambda-calculus for dataflow matrix machines.

[Michael Bukatin, Steve Matthews, Andrey Radul, *Notes on Pure Dataflow Matrix Machines: Programming with Self-referential Matrix Transformations*, `https://arxiv.org/abs/1610.00831`]

Example: lightweight pure dataflow machines

Fixed-size matrices instead of theoretically prescribed countable-sized matrcies (for simplicity).

Let us describe the construction of **lightweight pure dataflow matrix machine**. We consider rectangular matrices $M \times N$. We consider discrete time, $t = 0, 1, \ldots$, and we consider $M + N$ streams of those rectangular matrices, $X^1, \ldots, X^M, Y^1, \ldots, Y^N$. At any moment t, each of these streams takes a rectangular matrix $M \times N$ as its value. (For example, X^1_t or Y^N_t are such rectangular matrices. Elements of matrices are real numbers.)

Example: lightweight pure dataflow machines

Let's describe the rules of the dynamical system which would allow to compute $X^1_{t+1}, \dots, X^M_{t+1}$, $Y^1_{t+1}, \dots, Y^N_{t+1}$ from $X^1_t, \dots, X^M_t$, $Y^1_t, \dots, Y^N_t$. We need to make a choice, whether to start with $X^1_0, \dots, X^M_0$ as initial data, or whether to start with $Y^1_0, \dots, Y^N_0$. Our equations will slightly depend on this choice. In our series of preprints we tend to start with matrices $Y^1_0, \dots, Y^N_0$, and so we keep this choice here, even though this might be slightly unusual to the reader. But it is easy to modify the equations to start with matrices $X^1_0, \dots, X^M_0$.

Example: lightweight pure dataflow machines

Matrix Y^1_t will play a special role, so at any given moment t, we also denote this matrix as A. Define $X^i_{t+1} = \sum_{j=1,\dots,N} A_{i,j} Y^j_t$ for all $i = 1, \dots, M$. Define $Y^j_{t+1} = f^j(X^1_{t+1}, \dots, X^M_{t+1})$ for all $j = 1, \dots, N$.

So, $Y^1_{t+1} = f^1(X^1_{t+1}, \dots, X^M_{t+1})$ defines Y^1_{t+1} which will be used as A at the next time step $t + 1$. This is how the dynamical system modifies itself in lightweight pure dataflow matrix machines.

Example: wave of weights in a self-modifying system

From [`https://arxiv.org/abs/1610.00831`, Appendix D.2.2]

Define $f^1(X_t^1, \ldots, X_t^M) = X_t^1 + X_t^2$. Start with $Y_0^1 = A$, such that $A_{1,1} = 1$, $A_{1,j} = 0$ for all other j, and maintain the condition that first rows of all other matrices $Y^j, j \neq 1$ are zero. These first rows of all $Y^j, j = 1, \ldots, N$ will be invariant as t increases. This condition means that $X_{t+1}^1 = Y_t^1$ for all $t \geq 0$.

Example: wave of weights in a self-modifying system

Let's make an example with 3 constant update matrices:
Y^2_t, Y^3_t, Y^4_t. Namely, say that
$f^2(X^1_t, \dots, X^M_t) = U^2$,
$f^3(X^1_t, \dots, X^M_t) = U^3$,
$f^4(X^1_t, \dots, X^M_t) = U^4$.

Then say that $U^2_{2,2} = U^3_{2,3} = U^4_{2,4} = -1$, and
$U^2_{2,3} = U^3_{2,4} = U^4_{2,2} = 1$, and that all other elements of U^2, U^3, U^4 are zero[1]. And imposing an additional starting condition on $Y^1_0 = A$, let's say that $A_{2,2} = 1$ and that $A_{2,j} = 0$ for $j \neq 2$.

[1]Essentially we are saying that those matrices "point to themselves with weight -1", and that "U^2 poiints to U^3, U^3 points to U^4, and U^4 points to U^2 with weight 1".

Example: wave of weights in a self-modifying system

Now, if we run this dynamic system, the initial condition on second row of A would imply that at the $t=0$, $X^2_{t+1} = U^2$. Also $Y^1_{t+1} = X^1_{t+1} + X^2_{t+1}$, hence now taking $A = Y^1_1$ (instead of $A = Y^1_0$), we obtain $A_{2,2} = 1 + U^2_{2,2} = 0$, and in fact $A_{2,j} = 0$ for all $j \neq 3$, but $A_{2,3} = U^2_{2,3} = 1$.

Continuing in this fashion, one obtains $X^2_1 = U^2, X^2_2 = U^3, X^2_3 = U^4, X^2_4 = U^2, X^2_5 = U^3, X^2_6 = U^4, X^2_7 = U^2, X^2_8 = U^3, X^2_9 = U^4, \ldots$, while the invariant that the second row of matrix Y^1_t has exactly one element valued at 1 and all other zeros is maintained, and the position of that 1 in the second row of matrix Y^1_t is 2 at $t=0$, 3 at $t=1$, 4 at $t=2$, 2 at $t=3$, 3 at $t=4$, 4 at $t=5$, 2 at $t=6$, 3 at $t=7$, 4 at $t=8$, ...

Example: wave of weights in a self-modifying system

This element 1 moving along the second row of the network matrix is a simple example of a circular wave pattern in the matrix $A = Y_t^1$ controlling the dynamical system in question.

It is easy to use other rows of matrices U^2, U^3, U^4 as "payload" to be placed into the network matrix Y_t^1 for exactly one step at a time, and one can do other interesting things with this class of dynamical systems.

Initial open-source prototypes

`https://github.com/anhinga/fluid`

`https://github.com/jsa-aerial/DMM`

Minimalistic self-referential example described above:

`https://github.com/anhinga/fluid/tree/master/Lightweight_Pure_DMMs/aug_27_16_experiment`

(in Processing 2.2.1)

`https://github.com/jsa-aerial/DMM/blob/master/examples/dmm/oct_19_2016_experiment.clj`

(in Clojure)

Electronic coordinates

These slides are linked from my page on partial inconsistency and vector semantics of programming languages:

http://www.cs.brandeis.edu/~bukatin/partial_inconsistency.html

E-mail:

`bukatin@cs.brandeis.edu`