Dataflow Matrix Machines and V-values: a Bridge between Programs and Neural Nets

Michael Bukatin

HERE Technologies

Joint work with Jon Anthony (Boston College)

- - -

Video presentation for Kálmán & Kornai workshop,
Budapest, December 18, 2017

Dataflow matrix machines (DMMs)

- A strong generalization of neural networks
- Powerful enough to write programs in this formalism
- A program is determined by a matrix of numbers
- Any program P can be transformed to any program Q in this formalism by continuously transforming the matrices defining those programs.
- Synthesize a matrix of numbers to synthesize a program.

Dataflow matrix machines (DMMs)

- A strong generalization of neural networks
- Powerful enough to write programs in this formalism
- A program is determined by a matrix of numbers
- Any program P can be transformed to any program Q in this formalism by continuously transforming the matrices defining those programs.
- Synthesize a matrix of numbers to synthesize a program.

Dataflow matrix machines (DMMs)

- A strong generalization of neural networks
- Powerful enough to write programs in this formalism
- A program is determined by a matrix of numbers
- Any program P can be transformed to any program Q in this formalism by continuously transforming the matrices defining those programs.
- Synthesize a matrix of numbers to synthesize a program.

Dataflow matrix machines (DMMs)

- A strong generalization of neural networks
- Powerful enough to write programs in this formalism
- A program is determined by a matrix of numbers
- Any program P can be transformed to any program Q in this formalism by continuously transforming the matrices defining those programs.
- Synthesize a matrix of numbers to synthesize a program.

Dataflow matrix machines (DMMs)

- A strong generalization of neural networks
- Powerful enough to write programs in this formalism
- A program is determined by a matrix of numbers
- Any program P can be transformed to any program Q in this formalism by continuously transforming the matrices defining those programs.
- Synthesize a matrix of numbers to synthesize a program.

Linear streams

- Neural networks process streams of numbers
- DMMs process linear streams
- Streams are sequences
- Linear streams: the notion of
  linear combination of several streams is defined
- Example: for a vector space V,
  streams of its elements form a space of linear streams

Linear streams

- Neural networks process streams of numbers
- DMMs process linear streams
- Streams are sequences
- Linear streams: the notion of
  linear combination of several streams is defined
- Example: for a vector space V,
  streams of its elements form a space of linear streams

Linear streams

- Neural networks process streams of numbers
- DMMs process linear streams
- Streams are sequences
- Linear streams: the notion of
  linear combination of several streams is defined
- Example: for a vector space V,
  streams of its elements form a space of linear streams

Linear streams

- Neural networks process streams of numbers
- DMMs process linear streams
- Streams are sequences
- Linear streams: the notion of
  linear combination of several streams is defined
- Example: for a vector space V,
  streams of its elements form a space of linear streams

Linear streams

- Neural networks process streams of numbers
- DMMs process linear streams
- Streams are sequences
- Linear streams: the notion of
  linear combination of several streams is defined
- Example: for a vector space V,
  streams of its elements form a space of linear streams

Kinds of linear streams

- To distinguish between different spaces of linear streams we talk about kinds of linear streams.
- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of streams of probabilistic samples drawn from X and decorated with +/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Most importantly: streams of V-values

Kinds of linear streams

- To distinguish between different spaces of linear streams we talk about kinds of linear streams.
- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of streams of probabilistic samples drawn from X and decorated with +/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Most importantly: streams of V-values

Kinds of linear streams

- To distinguish between different spaces of linear streams we talk about kinds of linear streams.
- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of streams of probabilistic samples drawn from X and decorated with $+$/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Most importantly: streams of V-values

Kinds of linear streams

- To distinguish between different spaces of linear streams we talk about kinds of linear streams.
- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of streams of probabilistic samples drawn from X and decorated with $+$/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Most importantly: streams of V-values

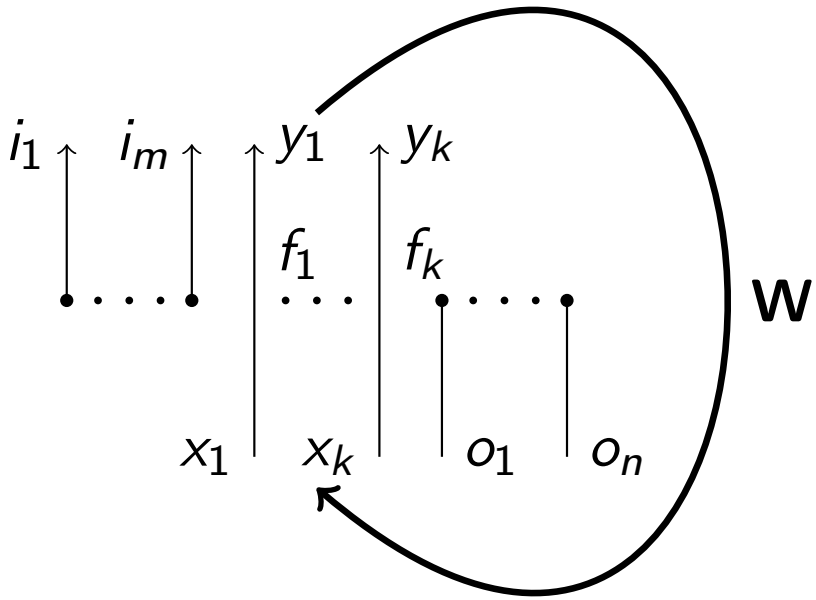
Kinds of linear streams

- To distinguish between different spaces of linear streams we talk about kinds of linear streams.
- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of streams of probabilistic samples drawn from X and decorated with $+$/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Most importantly: streams of V-values

Kinds of linear streams

- To distinguish between different spaces of linear streams we talk about kinds of linear streams.
- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of streams of probabilistic samples drawn from X and decorated with $+$/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Most importantly: streams of V-values

Recurrent neural networks

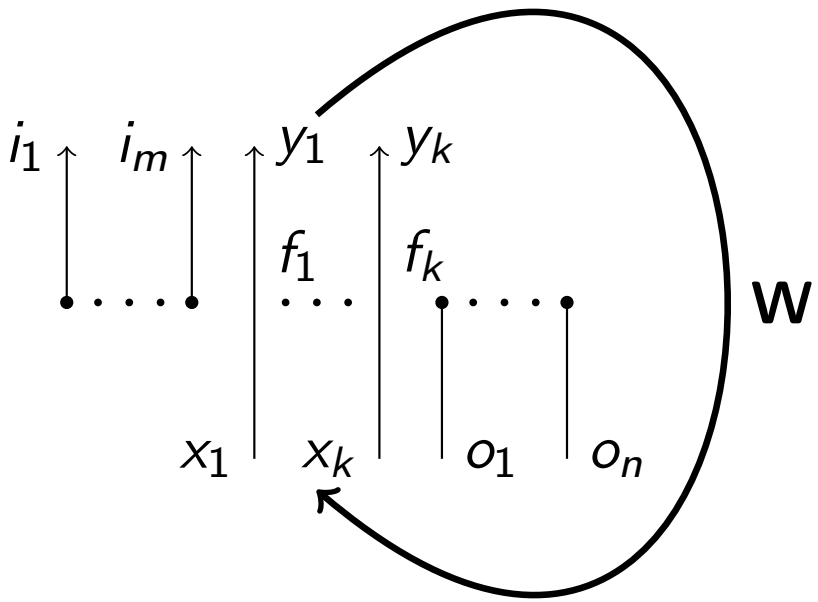


"Two-stroke engine" for an RNN:

"Down movement":
$(x_1^{t+1}, \ldots, x_k^{t+1}, o_1^{t+1}, \ldots, o_n^{t+1})^\top = \mathbf{W} \cdot (y_1^t, \ldots, y_k^t, i_1^t, \ldots, i_m^t)^\top$.

"Up movement":
$y_1^{t+1} = f_1(x_1^{t+1}), \ldots,$
$y_k^{t+1} = f_k(x_k^{t+1})$.

Recurrent neural networks

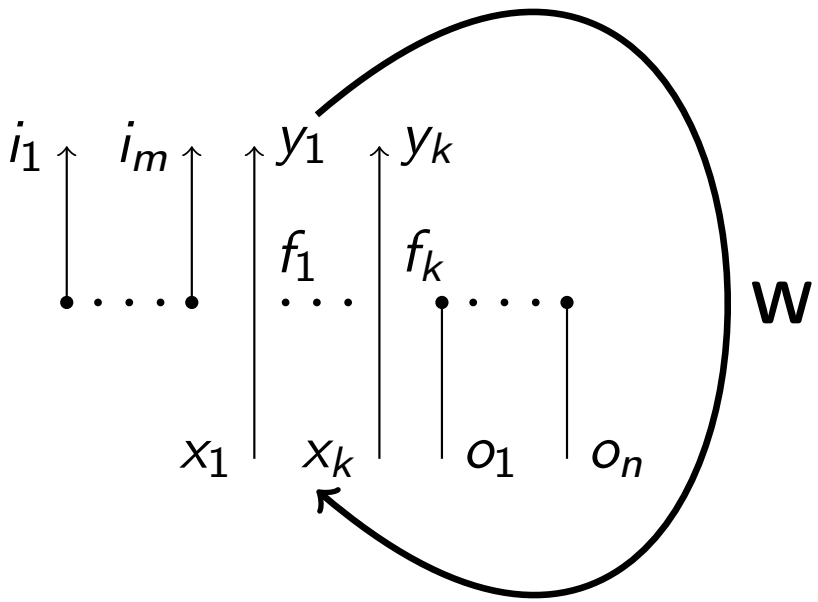


"Two-stroke engine" for an RNN:

"Down movement":
$(x_1^{t+1}, \ldots, x_k^{t+1}, o_1^{t+1}, \ldots, o_n^{t+1})^\top = \mathbf{W} \cdot (y_1^t, \ldots, y_k^t, i_1^t, \ldots, i_m^t)^\top$.

"Up movement":
$y_1^{t+1} = f_1(x_1^{t+1}), \ldots,$
$y_k^{t+1} = f_k(x_k^{t+1})$.

Recurrent neural networks



"Two-stroke engine" for an RNN:

"Down movement":
$(x_1^{t+1}, \ldots, x_k^{t+1}, o_1^{t+1}, \ldots, o_n^{t+1})^\top = \mathbf{W} \cdot (y_1^t, \ldots, y_k^t, i_1^t, \ldots, i_m^t)^\top$.

"Up movement":
$y_1^{t+1} = f_1(x_1^{t+1}), \ldots,$
$y_k^{t+1} = f_k(x_k^{t+1})$.

Dataflow matrix machines

Countable network with finite active part at any moment of time.

Countable matrix **W** with finite number of non-zero elements at any moment of time.

"Down movement":
For all inputs x_{i,C_k} where there is a non-zero weight $w^t_{(i,C_k),(j,C_l)}$:

$$x^{t+1}_{i,C_k} = \sum_{\{(j,C_l)|w^t_{(i,C_k),(j,C_l)}\neq 0\}} w^t_{(i,C_k),(j,C_l)} * y^t_{j,C_l}.$$

"Up movement":
For all active neurons C:

$$y^{t+1}_{1,C}, ..., y^{t+1}_{n,C} = f_C(x^{t+1}_{1,C}, ..., x^{t+1}_{m,C}).$$

Dataflow matrix machines

Countable network with finite active part at any moment of time.

Countable matrix **W** with finite number of non-zero elements at any moment of time.

"Down movement":
For all inputs x_{i,C_k} where there is a non-zero weight $w^t_{(i,C_k),(j,C_l)}$:

$$x^{t+1}_{i,C_k} = \sum_{\{(j,C_l)|w^t_{(i,C_k),(j,C_l)}\neq 0\}} w^t_{(i,C_k),(j,C_l)} * y^t_{j,C_l}.$$

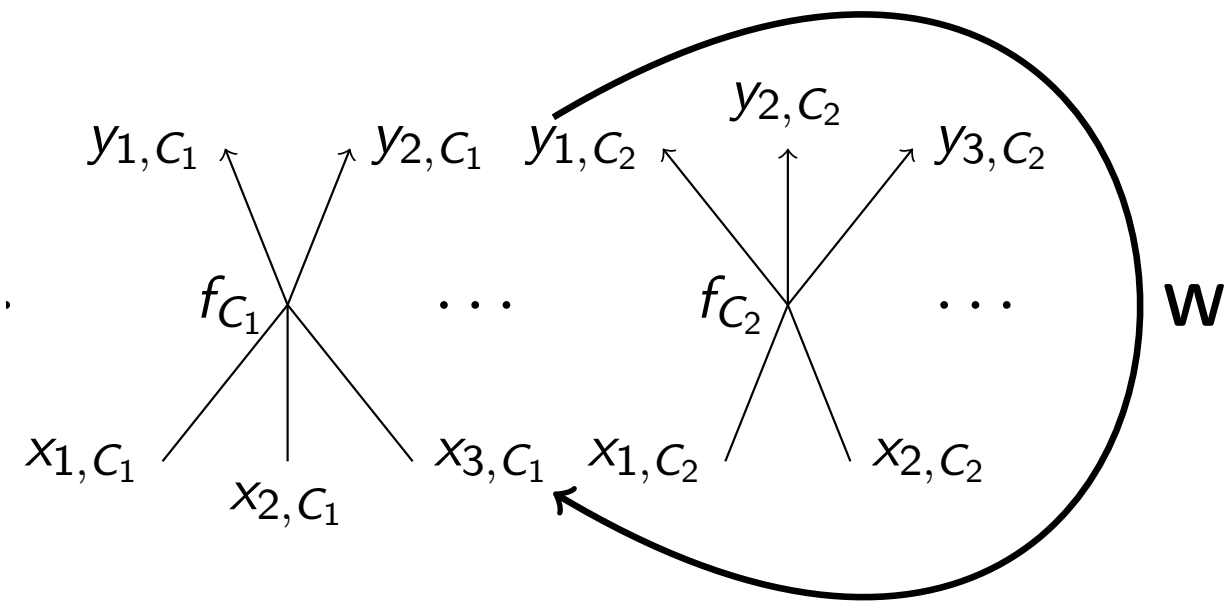
"Up movement":
For all active neurons C:

$$y^{t+1}_{1,C}, ..., y^{t+1}_{n,C} = f_C(x^{t+1}_{1,C}, ..., x^{t+1}_{m,C}).$$

Dataflow matrix machines

Countable network with finite active part at any moment of time.

Countable matrix **W** with finite number of non-zero elements at any moment of time.



"Down movement":
For all inputs x_{i,C_k} where there is a non-zero weight $w^t_{(i,C_k),(j,C_l)}$:

$$x^{t+1}_{i,C_k} = \sum_{\{(j,C_l)|w^t_{(i,C_k),(j,C_l)}\neq 0\}} w^t_{(i,C_k),(j,C_l)} * y^t_{j,C_l}.$$

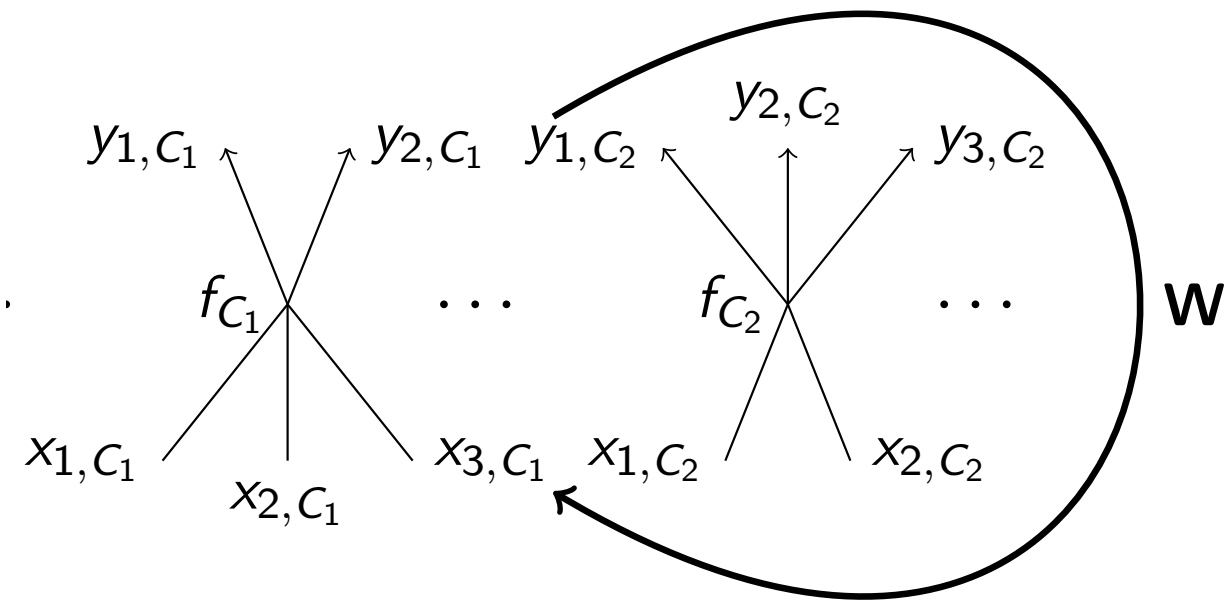
"Up movement":
For all active neurons C:

$$y^{t+1}_{1,C}, ..., y^{t+1}_{n,C} = f_C(x^{t+1}_{1,C}, ..., x^{t+1}_{m,C}).$$

Dataflow matrix machines

Countable network with finite active part at any moment of time.

Countable matrix **W** with finite number of non-zero elements at any moment of time.



"Down movement":
For all inputs x_{i,C_k} where there is a non-zero weight $w^t_{(i,C_k),(j,C_l)}$:

$$x^{t+1}_{i,C_k} = \sum_{\{(j,C_l)|w^t_{(i,C_k),(j,C_l)}\neq 0\}} w^t_{(i,C_k),(j,C_l)} * y^t_{j,C_l}.$$

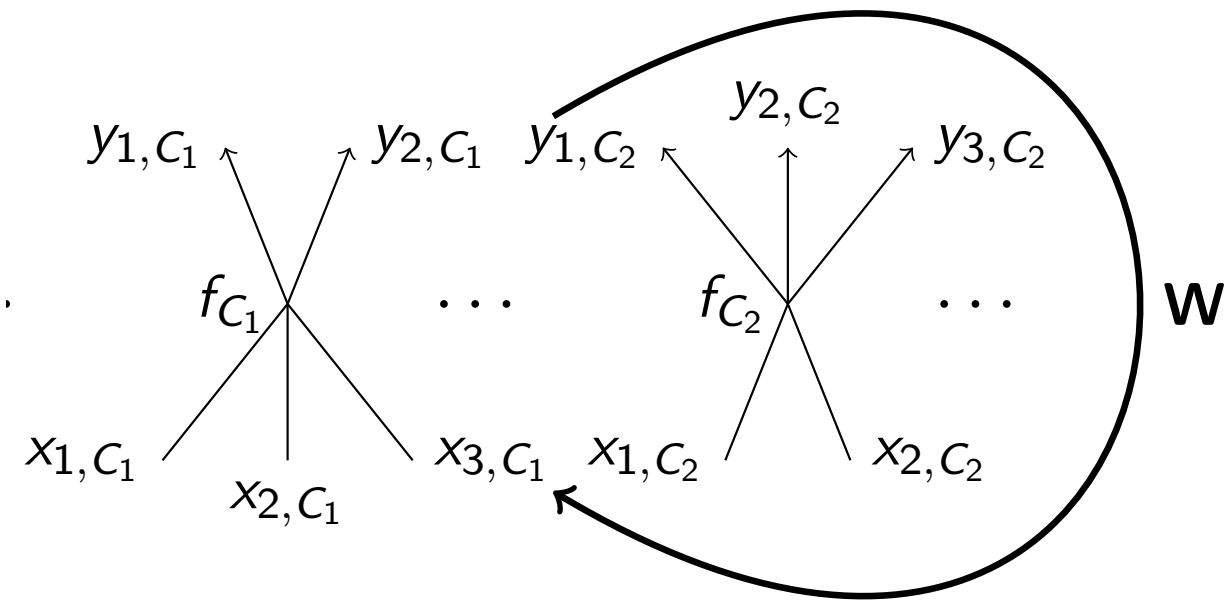
"Up movement":
For all active neurons C:

$$y^{t+1}_{1,C}, \ldots, y^{t+1}_{n,C} = f_C(x^{t+1}_{1,C}, \ldots, x^{t+1}_{m,C}).$$

Dataflow matrix machines

Countable network with finite active part at any moment of time.

Countable matrix **W** with finite number of non-zero elements at any moment of time.



"Down movement":
For all inputs x_{i,C_k} where there is a non-zero weight $w^t_{(i,C_k),(j,C_l)}$:

$$x^{t+1}_{i,C_k} = \sum_{\{(j,C_l)|w^t_{(i,C_k),(j,C_l)}\neq 0\}} w^t_{(i,C_k),(j,C_l)} * y^t_{j,C_l}.$$

"Up movement":
For all active neurons C:

$$y^{t+1}_{1,C}, ..., y^{t+1}_{n,C} = f_C(x^{t+1}_{1,C}, ..., x^{t+1}_{m,C}).$$

Type correctness condition

We need to keep input and output arity of each neuron in mind, and we also need to impose a type correctness condition in order for the formulas in the previous slide to be well-defined:

$w^t_{(i,C_k),(j,C_l)}$ is allowed to be non-zero only if x_{i,C_k} and y_{j,C_l} belong to the same *kind* of linear streams.

We are now going to introduce linear streams of *V-values* based on *nested dictionaries*, which are sufficiently universal and expressive to save us from the need to impose type correctness conditions.

Moreover, they will allow us to define *variadic neurons*, so that we don't need to keep track on input and output arity either.

Type correctness condition

We need to keep input and output arity of each neuron in mind, and we also need to impose a type correctness condition in order for the formulas in the previous slide to be well-defined:

$w^t_{(i,C_k),(j,C_l)}$ is allowed to be non-zero only if x_{i,C_k} and y_{j,C_l} belong to the same *kind* of linear streams.

We are now going to introduce linear streams of *V-values* based on *nested dictionaries*, which are sufficiently universal and expressive to save us from the need to impose type correctness conditions.

Moreover, they will allow us to define *variadic neurons*, so that we don't need to keep track on input and output arity either.

Type correctness condition

We need to keep input and output arity of each neuron in mind, and we also need to impose a type correctness condition in order for the formulas in the previous slide to be well-defined:

$w^t_{(i,C_k),(j,C_l)}$ is allowed to be non-zero only if x_{i,C_k} and y_{j,C_l} belong to the same *kind* of linear streams.

We are now going to introduce linear streams of *V-values* based on *nested dictionaries*, which are sufficiently universal and expressive to save us from the need to impose type correctness conditions.

Moreover, they will allow us to define *variadic neurons*, so that we don't need to keep track on input and output arity either.

Vector space based on nested dictionaries

Lisp introduced nested lists (S-expressions) in 1958.

It might be the case when computational resources are sufficient, that if one has to base a formalism on a single kind of nested data structure, this structure should be nested dictionaries.

The simplest way to build a vector space of nested dictionaries is to require all atoms (leaves) to be numbers.

We call nested dictionaries with numerical atoms *V-values*.

(A more general construction of V-values involving nested dictionaries with more complicated atoms is considered in the companion paper for this talk.)

Vector space based on nested dictionaries

Lisp introduced nested lists (S-expressions) in 1958.

It might be the case when computational resources are sufficient, that if one has to base a formalism on a single kind of nested data structure, this structure should be nested dictionaries.

The simplest way to build a vector space of nested dictionaries is to require all atoms (leaves) to be numbers.

We call nested dictionaries with numerical atoms *V-values*.

(A more general construction of V-values involving nested dictionaries with more complicated atoms is considered in the companion paper for this talk.)

Vector space based on nested dictionaries

Lisp introduced nested lists (S-expressions) in 1958.

It might be the case when computational resources are sufficient, that if one has to base a formalism on a single kind of nested data structure, this structure should be nested dictionaries.

The simplest way to build a vector space of nested dictionaries is to require all atoms (leaves) to be numbers.

We call nested dictionaries with numerical atoms *V-values*.

(A more general construction of V-values involving nested dictionaries with more complicated atoms is considered in the companion paper for this talk.)

Vector space based on nested dictionaries

Lisp introduced nested lists (S-expressions) in 1958.

It might be the case when computational resources are sufficient, that if one has to base a formalism on a single kind of nested data structure, this structure should be nested dictionaries.

The simplest way to build a vector space of nested dictionaries is to require all atoms (leaves) to be numbers.

We call nested dictionaries with numerical atoms *V-values*.

(A more general construction of V-values involving nested dictionaries with more complicated atoms is considered in the companion paper for this talk.)

Vector space based on nested dictionaries

Lisp introduced nested lists (S-expressions) in 1958.

It might be the case when computational resources are sufficient, that if one has to base a formalism on a single kind of nested data structure, this structure should be nested dictionaries.

The simplest way to build a vector space of nested dictionaries is to require all atoms (leaves) to be numbers.

We call nested dictionaries with numerical atoms *V-values*.

(A more general construction of V-values involving nested dictionaries with more complicated atoms is considered in the companion paper for this talk.)

Ways to understand V-values

Consider ordinary words ("labels") to be letters of a countable "super-alphabet" L.

V-values can be understood as

- Finite linear combinations of finite strings of letters from L;
- Finite prefix trees with numerical leaves;
- Sparse "tensors of mixed rank" with finite number of non-zero elements;
- Recurrent maps (that is, nested dictionaries) from $V \cong \mathbb{R} \oplus (L \to V)$ admitting finite descriptions.

Ways to understand V-values

Consider ordinary words ("labels") to be letters of a countable "super-alphabet" L.

V-values can be understood as

- Finite linear combinations of finite strings of letters from L;
- Finite prefix trees with numerical leaves;
- Sparse "tensors of mixed rank" with finite number of non-zero elements;
- Recurrent maps (that is, nested dictionaries) from $V \cong \mathbb{R} \oplus (L \to V)$ admitting finite descriptions.

Ways to understand V-values

Consider ordinary words ("labels") to be letters of a countable "super-alphabet" L.

V-values can be understood as

- Finite linear combinations of finite strings of letters from L;
- Finite prefix trees with numerical leaves;
- Sparse "tensors of mixed rank" with finite number of non-zero elements;
- Recurrent maps (that is, nested dictionaries) from $V \cong \mathbb{R} \oplus (L \to V)$ admitting finite descriptions.

Ways to understand V-values

Consider ordinary words ("labels") to be letters of a countable "super-alphabet" L.

V-values can be understood as

- Finite linear combinations of finite strings of letters from L;
- Finite prefix trees with numerical leaves;
- Sparse "tensors of mixed rank" with finite number of non-zero elements;
- Recurrent maps (that is, nested dictionaries) from $V \cong \mathbb{R} \oplus (L \to V)$ admitting finite descriptions.

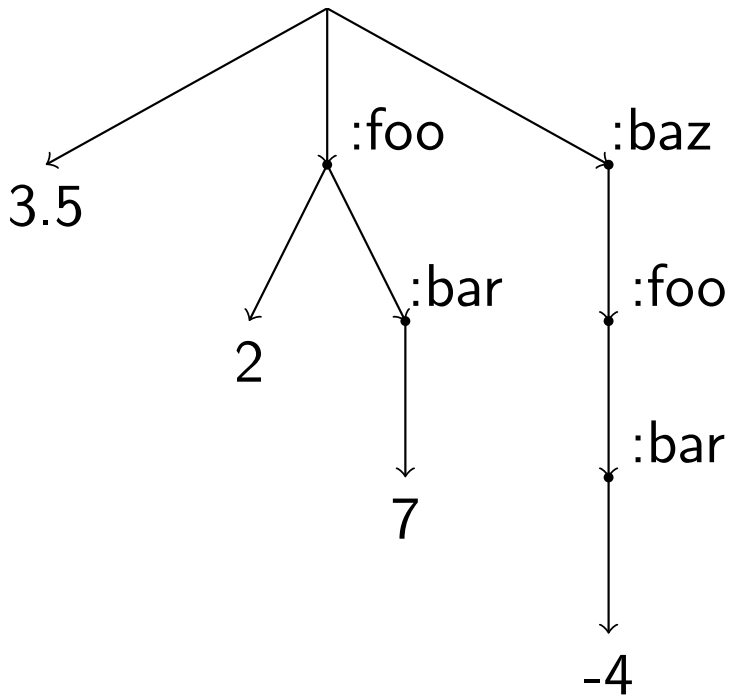
Ways to understand V-values

Consider ordinary words ("labels") to be letters of a countable "super-alphabet" L.

V-values can be understood as

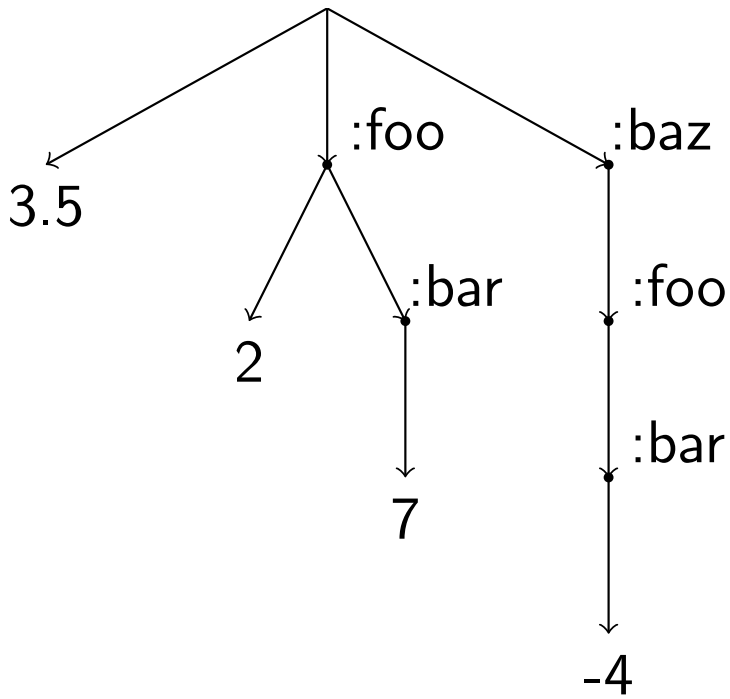
- Finite linear combinations of finite strings of letters from L;
- Finite prefix trees with numerical leaves;
- Sparse "tensors of mixed rank" with finite number of non-zero elements;
- Recurrent maps (that is, nested dictionaries) from $V \cong \mathbb{R} \oplus (L \to V)$ admitting finite descriptions.

Example of a V-value



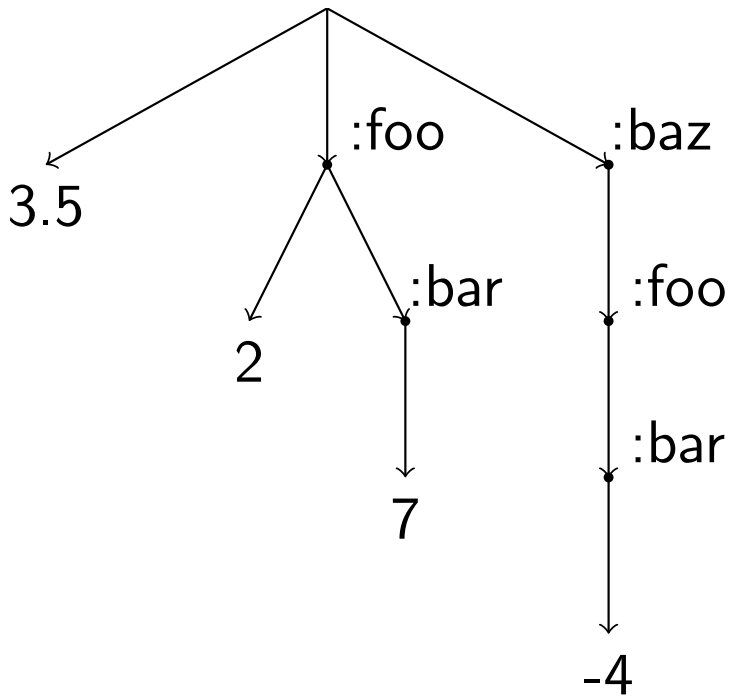
- $3.5 \cdot (\epsilon) + 2 \cdot (:\text{foo}) +$
  $7 \cdot (:\text{foo}\ :\text{bar})$ - $4 \cdot (:\text{baz}\ :\text{foo}\ :\text{bar})$
- $(\rightsquigarrow 3.5) + (:\text{foo} \rightsquigarrow 2) +$
  $(:\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow 7) +$
  $(:\text{baz} \rightsquigarrow :\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow \text{-}4)$
- scalar 3.5 + sparse 1D array `d1[:foo] = 2` +
  sparse 2D matrix `d2[:foo, :bar] = 7` +
  sparse 3D array `d3[:baz, :foo, :bar] = -4`
- `{:number 3.5,`
  `:foo {:number 2, :bar 7},`
  `:baz {:foo {:bar -4}}}`
  $(:\texttt{number} \notin L)$

Example of a V-value



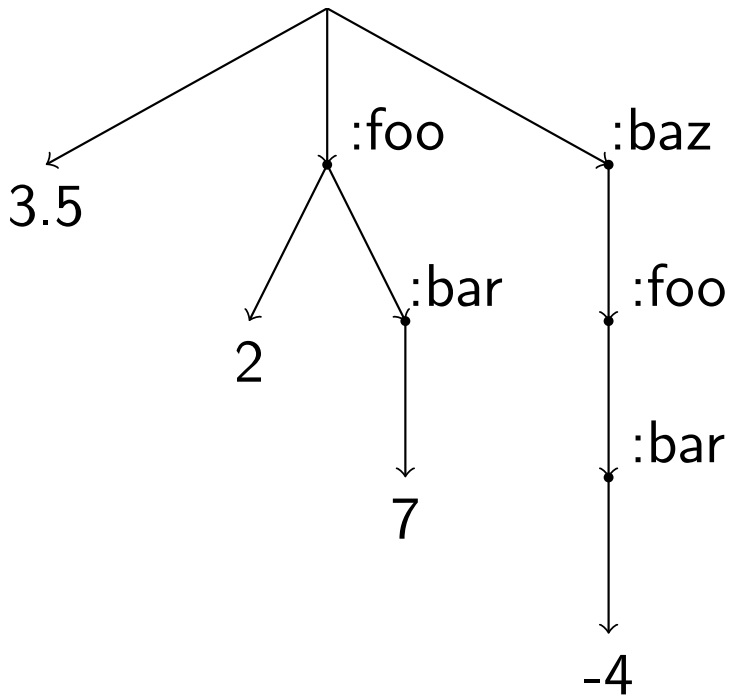
- $3.5 \cdot (\epsilon) + 2 \cdot (:\text{foo}) +$
  $7 \cdot (:\text{foo}\ :\text{bar})$ - $4 \cdot (:\text{baz}\ :\text{foo}\ :\text{bar})$
- $(\rightsquigarrow 3.5) + (:\text{foo} \rightsquigarrow 2) +$
  $(:\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow 7) +$
  $(:\text{baz} \rightsquigarrow :\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow \text{-}4)$
- scalar 3.5 + sparse 1D array `d1[:foo] = 2` +
  sparse 2D matrix `d2[:foo, :bar] = 7` +
  sparse 3D array `d3[:baz, :foo, :bar] = -4`
- `{:number 3.5,`
  `:foo {:number 2, :bar 7},`
  `:baz {:foo {:bar -4}}}`
  $(:\text{number} \notin L)$

Example of a V-value



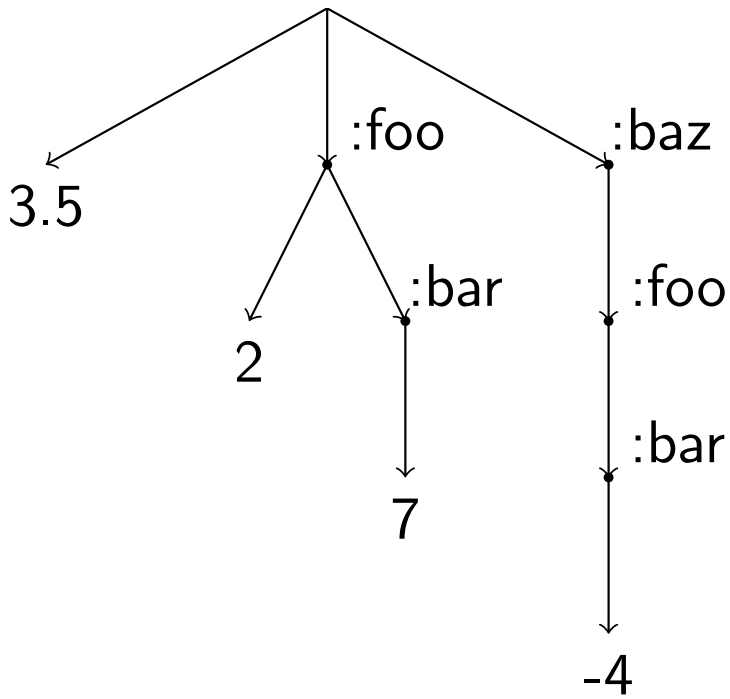
- $3.5 \cdot (\epsilon) + 2 \cdot (:\text{foo}) +$
  $7 \cdot (:\text{foo}\ :\text{bar})$ - $4 \cdot (:\text{baz}\ :\text{foo}\ :\text{bar})$
- $(\rightsquigarrow 3.5) + (:\text{foo} \rightsquigarrow 2) +$
  $(:\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow 7) +$
  $(:\text{baz} \rightsquigarrow :\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow \text{-}4)$
- scalar 3.5 + sparse 1D array `d1[:foo] = 2` +
  sparse 2D matrix `d2[:foo, :bar] = 7` +
  sparse 3D array `d3[:baz, :foo, :bar] = -4`
- `{:number 3.5,`
  `:foo {:number 2, :bar 7},`
  `:baz {:foo {:bar -4}}}`
  $(:\text{number} \notin L)$

Example of a V-value



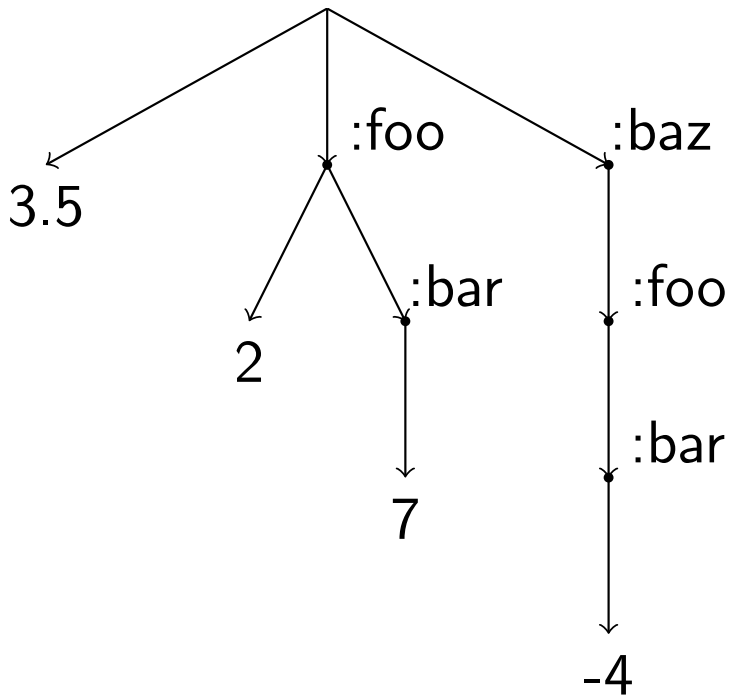
- $3.5 \cdot (\epsilon) + 2 \cdot (:\text{foo}) +$
  $7 \cdot (:\text{foo}\ :\text{bar})$ - $4 \cdot (:\text{baz}\ :\text{foo}\ :\text{bar})$
- $(\rightsquigarrow 3.5) + (:\text{foo} \rightsquigarrow 2) +$
  $(:\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow 7) +$
  $(:\text{baz} \rightsquigarrow :\text{foo} \rightsquigarrow :\text{bar} \rightsquigarrow \text{-}4)$
- scalar 3.5 + sparse 1D array `d1[:foo] = 2` +
  sparse 2D matrix `d2[:foo, :bar] = 7` +
  sparse 3D array `d3[:baz, :foo, :bar] = -4`
- `{:number 3.5,`
  `:foo {:number 2, :bar 7},`
  `:baz {:foo {:bar -4}}}`
  $(:\text{number} \notin L)$

Example of a V-value



- $3.5 \cdot (\epsilon) + 2 \cdot (\text{:foo}) +$
  $7 \cdot (\text{:foo :bar}) - 4 \cdot (\text{:baz :foo :bar})$
- $(\rightsquigarrow 3.5) + (\text{:foo} \rightsquigarrow 2) +$
  $(\text{:foo} \rightsquigarrow \text{:bar} \rightsquigarrow 7) +$
  $(\text{:baz} \rightsquigarrow \text{:foo} \rightsquigarrow \text{:bar} \rightsquigarrow \text{-4})$
- scalar 3.5 + sparse 1D array `d1[:foo] = 2` +
  sparse 2D matrix `d2[:foo, :bar] = 7` +
  sparse 3D array `d3[:baz, :foo, :bar] = -4`
- `{:number 3.5,`
  `:foo {:number 2, :bar 7},`
  `:baz {:foo {:bar -4}}}`
  $(\texttt{:number} \notin L)$

Example of a V-value

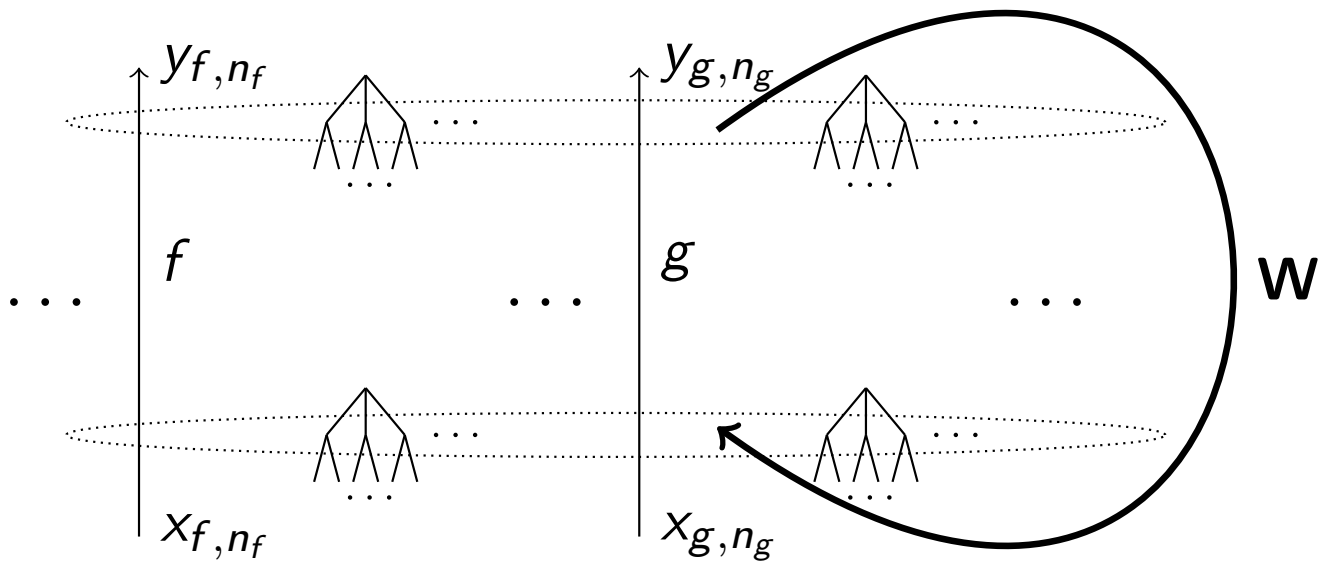


- $3.5 \cdot (\epsilon) + 2 \cdot (\text{:foo}) +$
  $7 \cdot (\text{:foo :bar}) - 4 \cdot (\text{:baz :foo :bar})$
- $(\rightsquigarrow 3.5) + (\text{:foo} \rightsquigarrow 2) +$
  $(\text{:foo} \rightsquigarrow \text{:bar} \rightsquigarrow 7) +$
  $(\text{:baz} \rightsquigarrow \text{:foo} \rightsquigarrow \text{:bar} \rightsquigarrow -4)$
- scalar 3.5 + sparse 1D array `d1[:foo] = 2` +
  sparse 2D matrix `d2[:foo, :bar] = 7` +
  sparse 3D array `d3[:baz, :foo, :bar] = -4`
- $\{$`:number 3.5,`
  `:foo {:number 2, :bar 7},`
  `:baz {:foo {:bar -4}}}`
  $(\texttt{:number} \notin L)$

Dataflow matrix machines based on streams of V-values and variadic neurons

$$x^{t+1}_{f,n_f,i} = \sum_{g \in F} \sum_{n_g \in L} \sum_{o \in L} w^t_{f,n_f,i;\,g,n_g,o} * y^t_{g,n_g,o} \text{ (down movement)}$$

$$y^{t+1}_{f,n_f} = f(x^{t+1}_{f,n_f}) \text{ (up movement)}$$



Powerful neurons

With powerful variadic neurons and streams of V-values we have a formalism which is much more expressive than neural nets based on streams of numbers.

Many tasks can be accomplished by compact networks, with single neurons functioning as layers or modules.

We only touch this topic lightly here; see the companion paper for more...

Powerful neurons

With powerful variadic neurons and streams of V-values we have a formalism which is much more expressive than neural nets based on streams of numbers.

Many tasks can be accomplished by compact networks, with single neurons functioning as layers or modules.

We only touch this topic lightly here; see the companion paper for more...

Powerful neurons

With powerful variadic neurons and streams of V-values we have a formalism which is much more expressive than neural nets based on streams of numbers.

Many tasks can be accomplished by compact networks, with single neurons functioning as layers or modules.

We only touch this topic lightly here; see the companion paper for more...

Sparse vectors of high or infinite dimension

Consider a neuron accumulating count of words in a given text.

The dictionary mapping words to their respective counts is an infinite-dimensional vector with a finite number of non-zero elements.

Don't need a neuron for each coordinate of our vector space

Don't have an obligation to reduce dimension by embedding.

Sparse vectors of high or infinite dimension

Consider a neuron accumulating count of words in a given text.

The dictionary mapping words to their respective counts is an infinite-dimensional vector with a finite number of non-zero elements.

Don't need a neuron for each coordinate of our vector space

Don't have an obligation to reduce dimension by embedding.

Sparse vectors of high or infinite dimension

Consider a neuron accumulating count of words in a given text.

The dictionary mapping words to their respective counts is an infinite-dimensional vector with a finite number of non-zero elements.

Don't need a neuron for each coordinate of our vector space

Don't have an obligation to reduce dimension by embedding.

Sparse vectors of high or infinite dimension

Consider a neuron accumulating count of words in a given text.

The dictionary mapping words to their respective counts is an infinite-dimensional vector with a finite number of non-zero elements.

Don't need a neuron for each coordinate of our vector space

Don't have an obligation to reduce dimension by embedding.

Streams of immutable data structures

It is convenient to represent a large variety of data structures via nested dictionaries. One can represent lists, matrices, graphs, and so on in this fashion.

The computational architecture promoted by the functional programming community is to work with immutable data sharing common substructures and garbage collecting the obsolete parts.

The experience of the field shows that this approach is pretty efficient in general and allows easy parallelization.

It is natural to use streams of immutable V-values in the implementations of DMMs.

Hence, the DMM architecture is friendly towards implementing algorithms working with immutable data structures in the spirit of functional programming.

Streams of immutable data structures

It is convenient to represent a large variety of data structures via nested dictionaries. One can represent lists, matrices, graphs, and so on in this fashion.

The computational architecture promoted by the functional programming community is to work with immutable data sharing common substructures and garbage collecting the obsolete parts.

The experience of the field shows that this approach is pretty efficient in general and allows easy parallelization.

It is natural to use streams of immutable V-values in the implementations of DMMs.

Hence, the DMM architecture is friendly towards implementing algorithms working with immutable data structures in the spirit of functional programming.

Streams of immutable data structures

It is convenient to represent a large variety of data structures via nested dictionaries. One can represent lists, matrices, graphs, and so on in this fashion.

The computational architecture promoted by the functional programming community is to work with immutable data sharing common substructures and garbage collecting the obsolete parts.

The experience of the field shows that this approach is pretty efficient in general and allows easy parallelization.

It is natural to use streams of immutable V-values in the implementations of DMMs.

Hence, the DMM architecture is friendly towards implementing algorithms working with immutable data structures in the spirit of functional programming.

Streams of immutable data structures

It is convenient to represent a large variety of data structures via nested dictionaries. One can represent lists, matrices, graphs, and so on in this fashion.

The computational architecture promoted by the functional programming community is to work with immutable data sharing common substructures and garbage collecting the obsolete parts.

The experience of the field shows that this approach is pretty efficient in general and allows easy parallelization.

It is natural to use streams of immutable V-values in the implementations of DMMs.

Hence, the DMM architecture is friendly towards implementing algorithms working with immutable data structures in the spirit of functional programming.

Streams of immutable data structures

It is convenient to represent a large variety of data structures via nested dictionaries. One can represent lists, matrices, graphs, and so on in this fashion.

The computational architecture promoted by the functional programming community is to work with immutable data sharing common substructures and garbage collecting the obsolete parts.

The experience of the field shows that this approach is pretty efficient in general and allows easy parallelization.

It is natural to use streams of immutable V-values in the implementations of DMMs.

Hence, the DMM architecture is friendly towards implementing algorithms working with immutable data structures in the spirit of functional programming.

Self-referential facilities

It is easy to represent the network matrix **W** as a V-value.

Therefore, it is easy to designate a particular neuron `Self` and a particular output of that neuron (say, `:active`) as emitting the streams of network matrices.

The V-value at that output of `Self` obtained during the most recent "up movement" is used as the network matrix **W** during the next "down movement".

This mechanism allows any DMM network to **modify its own topology and weights** while it is running.

Self-referential facilities

It is easy to represent the network matrix **W** as a V-value.

Therefore, it is easy to designate a particular neuron `Self` and a particular output of that neuron (say, `:active`) as emitting the streams of network matrices.

The V-value at that output of `Self` obtained during the most recent "up movement" is used as the network matrix **W** during the next "down movement".

This mechanism allows any DMM network to **modify its own topology and weights** while it is running.

Self-referential facilities

It is easy to represent the network matrix **W** as a V-value.

Therefore, it is easy to designate a particular neuron `Self` and a particular output of that neuron (say, `:active`) as emitting the streams of network matrices.

The V-value at that output of `Self` obtained during the most recent "up movement" is used as the network matrix **W** during the next "down movement".

This mechanism allows any DMM network to **modify its own topology and weights** while it is running.

Self-referential facilities

It is easy to represent the network matrix **W** as a V-value.

Therefore, it is easy to designate a particular neuron `Self` and a particular output of that neuron (say, `:active`) as emitting the streams of network matrices.

The V-value at that output of `Self` obtained during the most recent "up movement" is used as the network matrix **W** during the next "down movement".

This mechanism allows any DMM network to **modify its own topology and weights** while it is running.

Self-referential facilities

This is something which is theoretically possible, but practically very difficult to do in neural nets based on streams of numbers.

Our current implementation uses `Self` which works as an accumulator. It accumulates the value of the network matrix while accepting additive updates from other neurons in the network.

In turn, the other neurons can use `Self` outputs to become aware of the current network structure and weights.

Our experiments show that this mechanism together with simple constant update neurons is capable of producing waves of connectivity patterns propagating within the network matrix $\mathbf{W}$.

Self-referential facilities

This is something which is theoretically possible, but practically very difficult to do in neural nets based on streams of numbers.

Our current implementation uses `Self` which works as an accumulator. It accumulates the value of the network matrix while accepting additive updates from other neurons in the network.

In turn, the other neurons can use `Self` outputs to become aware of the current network structure and weights.

Our experiments show that this mechanism together with simple constant update neurons is capable of producing waves of connectivity patterns propagating within the network matrix $\mathbf{W}$.

Self-referential facilities

This is something which is theoretically possible, but practically very difficult to do in neural nets based on streams of numbers.

Our current implementation uses `Self` which works as an accumulator. It accumulates the value of the network matrix while accepting additive updates from other neurons in the network.

In turn, the other neurons can use `Self` outputs to become aware of the current network structure and weights.

Our experiments show that this mechanism together with simple constant update neurons is capable of producing waves of connectivity patterns propagating within the network matrix $\mathbf{W}$.

Self-referential facilities

This is something which is theoretically possible, but practically very difficult to do in neural nets based on streams of numbers.

Our current implementation uses `Self` which works as an accumulator. It accumulates the value of the network matrix while accepting additive updates from other neurons in the network.

In turn, the other neurons can use `Self` outputs to become aware of the current network structure and weights.

Our experiments show that this mechanism together with simple constant update neurons is capable of producing waves of connectivity patterns propagating within the network matrix $\mathbf{W}$.

Applications...

We hope that this formalism will be useful in various **learning to learn** setups
(the systems which learn how to learn better),
in **program synthesis**, and more...

Electronic resources

The companion paper will be linked from my page on partial inconsistency and vector semantics of programming languages:

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

It will also be linked from our open source project which implements core DMM primitives in Clojure:

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