Dataflow matrix machines and V-values: A bridge between programs and neural nets

MICHAEL BUKATIN
HERE North America LLC
bukatin@cs.brandeis.edu

JON ANTHONY
Boston College
jsa.aerial@gmail.com

KEYWORDS

dataflow matrix machines
linear streams
self-referential neural
  networks
continuous deformations
  of programs
V-values
variadic neurons

ABSTRACT

Dataflow matrix machines (DMMs) generalize neural nets by replacing streams of numbers with linear streams (streams supporting linear combinations), allowing arbitrary input and output arities for activation functions, countable-sized networks with finite dynamically changeable active part capable of unbounded growth, and a very expressive self-referential mechanism.

  DMMs are suitable for general-purpose programming, while retaining the key property of recurrent neural networks: programs are expressed via matrices of real numbers, and continuous changes to those matrices produce arbitrarily small variations in the associated programs.

  Spaces of V-values (vector-like elements based on nested maps) are particularly useful, enabling DMMs with variadic activation functions and conveniently representing conventional data structures.

1. Introduction

András Kornai wrote his *Mathematical Linguistics* book (Kornai 2008) while he and the first author of the present paper were working in the same office at MetaCarta during the previous decade and sharing many fruitful moments. The book published ten years ago was written as an introduction to the mathematical foundations of linguistics for computer scientists, engineers, and mathematicians interested in natural language processing.

Dataflow matrix machines emerged two years ago (see Bukatin & Matthews 2015b) and a series of technical papers have been written on the subject since then (Bukatin et al. 2016a;b;c;d; Bukatin & Anthony

2017). The present paper is meant to be an introduction to the subject for researchers and engineers working in other fields. We tried to keep the style of chapters and sections of *Mathematical Linguistics* in mind while writing this article.

*   *   *

Artificial neural networks are a powerful machine learning platform based on processing the streams of numbers.

It is long known that recurrent neural networks are expressive enough to encode any algorithm, if they are equipped with a reasonable form of unbounded memory (McCulloch & Pitts 1943; Pollack 1987; Siegelmann & Sontag 1995). There is a long history of synthesis of algorithms expressed as neural networks both by compilation and by machine learning methods.

However, conventional neural networks belong to the class of *esoteric programming languages* and do not constitute a convenient platform for manual software engineering.

In particular, there is a considerable history of using neural networks to synthesize and modify other neural networks, including self-modification. However, the limitations of conventional neural networks as a software engineering platform make efforts of this kind quite challenging.

The main key point of the approach of *dataflow matrix machines* is that the natural degree of generality for the neural model of computations is not the streams of numbers, but arbitrary *linear streams*.

The other enhancements dataflow matrix machines make to the neural model of computations are neurons of *variable input and output arity*, novel models of unbounded memory based on countable-sized weight-connectivity matrices with finite number of non-zero weights at any given time and, more generally, on streams of countable-dimensional vectors, and *explicit self-referential facilities*.

This results in a more powerful and expressive machine learning platform.

When one considers dataflow matrix machines as a software engineering framework, it turns out that the restriction to linear streams and to programs which admit continuous deformations is less severe than one could have thought a priori given the discrete nature of conventional programming languages.

Dataflow matrix machines are considerably closer to being a general-purpose programming platform than recurrent neural nets, while retaining the key property of recurrent neural nets that large classes of programs

can be parametrized by matrices of numbers, and therefore synthesizing appropriate matrices is sufficient to synthesize programs.

Linear streams. Dataflow matrix machines are built around the notion of linear streams. Generally speaking, we say that a space of streams is a *space of linear streams*, if a meaningful notion of linear combination of several streams with numerical coefficients is well-defined.

The simplest example of a space of linear streams is the space of sequences of numbers. A slightly more complicated example comes from considering a vector space V and the space of sequences of its elements, $(v_1, v_2, \ldots)$.

In the first few sections of this paper, this simple version of the notion of linear streams would be sufficient. The discrete time is represented by non-negative integers, a particular vector space V is fixed, and the space of functions from time to V forms the space of linear streams in question.

To distinguish between streams based on different linear spaces, e.g., V_1 and V_2, we talk about different *kinds of linear streams*.

In section 5, we describe a sufficiently general notion of linear streams which includes, for example, streams of samples from sequences of probability distributions over an arbitrary measurable space X. All constructions in the present paper work in this degree of generality.

Neuron types. The originally developed formalism of dataflow matrix machines was heavily typed (Bukatin et al. 2016b). One considered a diverse collection of *kinds of linear streams*, and a diverse collection of *types of neurons* with explicit fixed input and output arities.

The more recent version is close to being type free. It uses a single kind of linear streams based on a "sufficiently universal" space of elements we call V-values. V-values enable the use of *variadic neurons* which have arbitrary input and output arities, eliminating the need to keep track of input and output arities. The neuron types still exist in that they have different activation functions.

Structure of this paper. We start with an informal introduction to recurrent neural networks and to the typed version of dataflow matrix machines (DMMs) in section 2. In section 3, we present the theory of V-values. In section 4, we describe DMMs based on V-values and variadic neurons. We discuss linear streams in section 5. In that section we also discuss embeddings of discrete objects into vector spaces, and into spaces of linear streams.

Programming patterns in DMMs are presented in section 6. A very expressive self-referential mechanism which is a key element of our approach is presented in section 7.

The issues related to expressing the network topology are the subject of section 8 and the issues related to subnetworks and modularization are the subject of section 9.

We discuss some of the potential approaches to using DMMs in machine learning in section 10. The concluding section 11 contains historical remarks and discussion of related work.

2. From recurrent neural networks to dataflow matrix machines

The essence of artificial neural architectures is that linear and nonlinear transformations are interleaved. Then one can control neural computations by only modifying the linear part and keeping the non-linear part fixed.

Therefore, neural architectures such as *recurrent neural networks* (RNNs) can be viewed as "two-stroke engines" (Figure 1), where the "two-stroke cycle" of a linear "down movement" followed by a typically non-linear "up movement" is repeated indefinitely.

The network consists of the weight matrix $\mathbf{W}$ and the neurons. The neuron k has input and output streams of numbers, x_k^t and y_k^t, associated with it, where t is discrete time. The network also has streams of numbers i_m^t representing external inputs, and streams of numbers o_n^t representing external outputs.
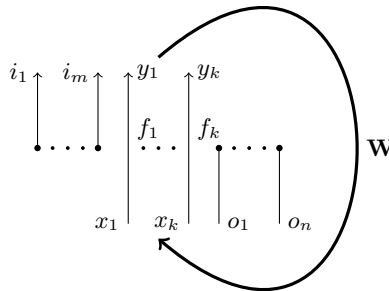

Figure 1: "Two-stroke engine" for an RNN. Figure from Bukatin & Anthony (2017).

On the "down movement", neuron inputs and network external outputs are computed by applying linear transformation $\mathbf{W}$ to the neuron outputs and network external inputs: $(x_1^{t+1}, \ldots, x_k^{t+1}, o_1^{t+1}, \ldots, o_n^{t+1})^\top = \mathbf{W} \cdot (y_1^t, \ldots, y_k^t, i_1^t, \ldots, i_m^t)^\top$. On the "up movement", the neurons calculate their outputs from their inputs using activation functions f_k which are built into each neuron k and are usually non-linear: $y_1^{t+1} = f_1(x_1^{t+1}), \ldots, y_k^{t+1} = f_k(x_k^{t+1})$.

Note that the computations during the "up movement" are local to the neuron in question, while the computations during the linear "down movement" are potentially quite global, as any neuron output might potentially be linked to any neuron input by a non-zero element of $\mathbf{W}$.

Now, moving from RNNs to *dataflow matrix machines* (DMMs), consider a finite or countable collection of *kinds of linear streams*, a finite or countable collection of *neuron types*, with every neuron type specifying non-negative integer number of inputs, non-negative integer number of outputs, the kind of linear streams associated with each input and each output, and an activation function transforming the inputs to the outputs.

Take a countable collection of neurons of each type, so that we have a countable-sized overall network. However, we'll make sure that only a finite part of this network is active at any given time (similarly to only a finite part of the Turing machine tape having non-blank symbols at any given time), and that processing time and memory are only spent on working with the currently active part, while the rest exists simply as potentially infinite address space.

The network consists of a countable-sized connectivity matrix $\mathbf{W}$ and a countable-sized collection of neurons described in the previous paragraph. The connectivity matrix $\mathbf{W}$ depends[1] on discrete time t.

The matrix element $w_{(i,C_k),(j,C_l)}^t$ is the weight linking the output j of the neuron C_l to the input i of the neuron C_k at the moment t. We impose the condition that at any given moment of time t only finite number of matrix elements $w_{(i,C_k),(j,C_l)}^t$ are non-zero. Hence the connectivity matrix is inherently sparse, and the structure of its non-zero weights determines the actual connectivity pattern of the network at any given moment of time.

The DMM "two-stroke engine" in a fashion similar to that of RNNs has a "two-stroke cycle" consisting of a linear "down movement" followed by an "up movement" performed by the activation functions of neurons (Figure 2). This "two-stroke cycle" is repeated indefinitely.

---

[1] When $\mathbf{W}$ changes with time, this change can be controlled from the outside or by the network itself via the self-referential mechanism described in section 7.
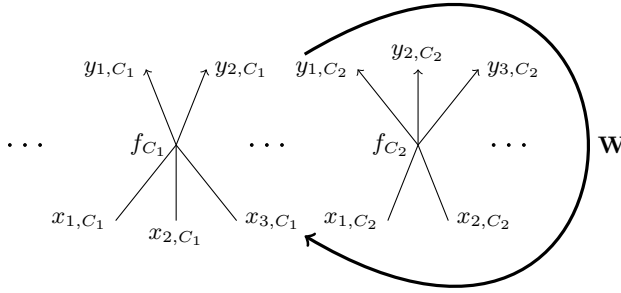
Figure 2: "Two-stroke engine" for a standard DMM (Bukatin & Anthony 2017). Two of its neurons, C_1 and C_2, are explicitly shown.

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

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

Note that x^{t+1}_{i,C_k} and y^t_{j,C_l} may no longer be numbers, but elements of linear streams x_{i,C_k} and y_{j,C_l}, so in order for Equation 1 to be well-defined we impose the type correctness condition which states that $w^t_{(i,C_k),(j,C_l)}$ 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.[2]

We call a neuron C active at the time t, if there is at least one non-zero connectivity weight from **W** associated with one of its inputs or outputs. Since **W** has only a finite number of non-zero weights at any given time, there are only a finite number of active neurons in the network at any given time.

"Up movement" is defined as follows. For all active neurons C:

$$y^{t+1}_{1,C}, ..., y^{t+1}_{n,C} = f_C(x^{t+1}_{1,C}, ..., x^{t+1}_{m,C}). \tag{2}$$

Here m is the input arity of neuron C and n is its output arity, so f_C has m inputs and n outputs. If $m = 0$, then f_C has no arguments. If $n = 0$, then the neuron just consumes data, and does not produce streams on

---

[2] Recall that the number of inputs and outputs of a neuron C, the kind of linear streams associated with each particular input or output of this neuron, and the built-in activation function f_C of this neuron are determined by the type of the neuron in question.

the "up movement". Given that the input and output arities of neurons are allowed to be zero, special handling of network inputs and outputs which was necessary for RNNs (Figure 1) is not needed here. The neurons responsible for network input and output are included on par with all other neurons.

The resulting formalism is very powerful, and we discuss what can be done with it later in the paper. However, its complexity is a bit unpleasant. The need to keep track of various kinds of linear streams and of the details of various neuron types is rather tiresome. It would be great to have only one sufficiently expressive kind of linear streams, and, moreover, to avoid the need to specify the arity of activation functions, while still enjoying the power of having multiple inputs and outputs within a single neuron. The spaces of *V-values* discussed in the next section allow us to achieve just that, while further increasing the power of a single neuron.

3. V-values

In this section, we define vector space V which is sufficiently rich to represent vectors from many other spaces encountered in practice.

In section 5.3, we show how to enrich this construction in those situations where it is not sufficiently universal, resulting in a family of vector spaces.

We call both the elements of these vector spaces and the hash-map-based representations of those elements *V-values*. In our context, V-values play the role somewhat similar to the role of S-expressions in the context of Lisp.

Implementation-wise, we create a version of S-expressions which is dictionary-based, rather than list-based. In this section, we require all atoms of those dictionary-based S-expressions to be numbers. A more general form of leaves in V-values is considered in section 5.3.

Speaking more formally, we start with a finite or countable alphabet L of labels (which we sometimes call tokens or keys). One can think about elements of L as words from some language defined over some other alphabet, which allows us to think about meaningful languages of labels.

We are going to consider several equivalent ways to define tree-like structures with intermediate nodes labeled by elements of L and with leaves labeled by numbers. Some of these ways are "depth-first", and they are easier to present mathematically, and some are "breadth-first", and they are more fundamental to us, as we use them in our implementation and as they enable the use of variadic neurons.

These tree-like structures can be viewed as

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

3.1. Finite linear combinations of finite strings

To start with "depth-first" methods, consider space L^* of finite sequences of elements of L (including the empty sequence), and construct V as the vector space of formal finite linear combinations[3] of the elements of L^* over $\mathbb{R}$.

We denote the empty sequence of elements of L as ε, and we denote non-empty sequences of elements of L, $(l_1, \ldots, l_n)$, as $l_1 \rightsquigarrow \ldots \rightsquigarrow l_n$. Since we are talking about formal finite linear combinations of elements of L^*, we need a notation for the multiplication of real number α and generator $l_1 \rightsquigarrow \ldots \rightsquigarrow l_n$, $\alpha \cdot (l_1 \rightsquigarrow \ldots \rightsquigarrow l_n)$.

For reasons, which become apparent in the next subsection, it is convenient to denote $\alpha \cdot (l_1 \rightsquigarrow \ldots \rightsquigarrow l_n)$ as $l_1 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \alpha$.

3.2. Finite prefix trees with numerical leaves

One can think of $l_1 \rightsquigarrow \ldots \rightsquigarrow l_n$ as a path in a prefix tree (*trie*), with intermediate nodes being labeled by letters from L. So when one considers $\alpha \in \mathbb{R}$, one can express the presence of term $\alpha \cdot (l_1 \rightsquigarrow \ldots \rightsquigarrow l_n)$ in our linear combination as presence of path with the intermediate nodes labeled by $l_1, \ldots, l_n$ and the leaf labeled by α. We denote this path as $l_1 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \alpha$. Because we have finite linear combinations (terms and paths corresponding to $\alpha = 0$ tend to be omitted), we are talking about *finite prefix trees with numerical leaves*.

Example. The linear combination $3.5 \cdot (\varepsilon) + 2 \cdot (\texttt{:foo}) + 7 \cdot (\texttt{:foo :bar}) - 4 \cdot (\texttt{:baz :foo :bar})$, i.e., $(\rightsquigarrow 3.5) + (\texttt{:foo} \rightsquigarrow 2) + (\texttt{:foo} \rightsquigarrow \texttt{:bar} \rightsquigarrow 7) + (\texttt{:baz} \rightsquigarrow \texttt{:foo} \rightsquigarrow \texttt{:bar} \rightsquigarrow -4)$.

---

[3] the space of functions $f : L^* \to \mathbb{R}$ such that $f(w) \neq 0$ for no more than finite number of $w \in L^*$; the operations are pointwise: $(f + g)(w) = f(w) + g(w)$ and $(\alpha f)(w) = \alpha f(w)$
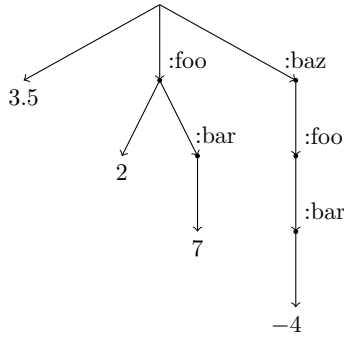
Figure 3: The prefix tree for this example

The empty string ε with non-zero coefficient β, written as $\beta \cdot (\varepsilon)$ or simply $\rightsquigarrow \beta$, corresponds to the leaf with non-zero β attached directly to the root of the tree.

The prefix tree (trie) for this example is shown in Figure 3. In this particular example, we label intermediate nodes with Clojure keywords, which start with ":" character. This example reminds us about the earlier remark that one can often think about letters from alphabet L as words from some language defined over some other, more conventional alphabet, and that this allows us to think about meaningful languages of labels.

3.3. Sparse "tensors of mixed rank"

Yet another "depth-first" way of looking at this situation is to consider $l_1 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \alpha$ to be an element of sparse multidimensional array with n dimensions. E.g., $l_1 \rightsquigarrow l_2 \rightsquigarrow \alpha$ is an element of a sparse matrix with its row labeled by l_1 and its column labeled by l_2 and α being the value of the element.

The non-zero leaf attached directly to the root of the tree, $\beta \cdot (\varepsilon) = \rightsquigarrow \beta$, is considered to be a scalar with value β.

Each string of length one with non-zero coefficient γ, that is $l \rightsquigarrow \gamma$ (the leaf with non-zero γ attached to the end of the path of length one with the intermediate node in the path labeled by l), is considered to be a coordinate of a sparse array, where the coordinate is labeled by l and has value γ.

Each string (path) of length three with non-zero coefficient γ', written as $l_1 \rightsquigarrow l_2 \rightsquigarrow l_3 \rightsquigarrow \gamma'$, is considered to be an element of sparse three-dimensional array, etc.

The standard convention in machine learning is to call multidimensional arrays with n dimensions "tensors of rank n". Because our linear combinations generally include sequences from L^* of different lengths, we have to talk about *sparse "tensors of mixed rank"*. For example, the vector $(\rightsquigarrow 3.5) + (\texttt{:foo} \rightsquigarrow 2) + (\texttt{:foo} \rightsquigarrow \texttt{:bar} \rightsquigarrow 7) + (\texttt{:baz} \rightsquigarrow \texttt{:foo} \rightsquigarrow \texttt{:bar} \rightsquigarrow -4)$ from the previous subsection is the sum of scalar 3.5, sparse array with one non-zero element $\texttt{d1[:foo] = 2}$, sparse matrix with one non-zero element $\texttt{d2[:foo, :bar] = 7}$, and sparse three-dimensional array with one non-zero element $\texttt{d3[:baz, :foo, :bar] = -4}$, so it is a typical "tensor of mixed rank".

Therefore all usual vectors, matrices, and tensors of any dimension can be represented in V. This is convenient from the viewpoint of machine learning as multidimensional tensors often occur in the machine learning practice.

The space V is a direct sum of the one-dimensional space of scalars and spaces of n-dimensional arrays: $V = V_0 \oplus V_1 \oplus V_2 \ldots$ Since L is countably infinite, $V_1, V_2, \ldots$ are infinite-dimensional vector spaces. If L is finite and consists of $\text{Card}(L)$ elements, then the dimension of V_i as a vector space is $\text{Card}^i(L)$.

This is also a good point to transition to "breadth-first" representations. A sparse matrix can also be viewed as a map from elements of L labeling its non-zero rows to the sparse vectors representing those rows (these sparse vectors are maps from elements of L labeling columns of the matrix to the values of the actual non-zero matrix elements; zero elements are omitted, as usual).

3.4. Recurrent maps

To obtain a "breadth-first" representation for a general element $v \in V$, we first note that there is a possibility that the $\alpha \cdot (\varepsilon) = \rightsquigarrow \alpha$ belongs to v with non-zero coefficient α, in which case the corresponding coordinate of v is a non-zero "tensor of rank 0", i.e., the scalar with value α.

Then for each letter $l_1 \in L$, such that some non-zero term $l_1 \rightsquigarrow l_2 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \beta$ belongs to v, we consider all terms from v which share the same first letter l_1, namely $l_1 \rightsquigarrow l_2 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \beta$, $l_1 \rightsquigarrow l_2' \rightsquigarrow \ldots \rightsquigarrow l_m' \rightsquigarrow \gamma, \ldots$, and consider $v^{l_1} \in V$ consisting of those terms with the first letter removed, namely $l_2 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \beta$, $l_2' \rightsquigarrow \ldots \rightsquigarrow l_m' \rightsquigarrow \gamma, \ldots$

We map each such letter l_1 to v^{l_1}, and we map each letter $l \in L$ for which v does not have a non-zero term starting from l to zero vector (zero element of V). The finite description of our map only needs to include the finite set of $\langle l_1, v^{l_1} \rangle$ pairs, and pairs $\langle l, 0 \rangle$ and $\langle l_1, 0 \rangle$ can all be omitted.

An element of v is then a pair consisting of a scalar and a map from L to V admitting a finite description. Either or both elements of this pair can be zero.

As a vector space, V satisfies the following equation:

$$V \cong \mathbb{R} \oplus (L \to V). \tag{3}$$

Here $L \to V$ is a space of such maps from L to V that only a finite number of elements of L map to non-zero elements of V. Let us call such maps *finitary*. This equation is an *isomorphism of vector spaces*. It reflects the fact that every element $v \in V$ can be represented as a pair of $\alpha \in \mathbb{R}$ and finitary map $l \mapsto v^l$, as was shown earlier in the present subsection, and vice versa every pair consisting of $\alpha \in \mathbb{R}$ and a finitary map $L \to V$ is obtained in this fashion.

We would like to be able to represent elements of V not by pairs of a number and a map, but simply by maps. To include numbers α into the map itself, we would need a separate label for them that does not appear in L. Thus we take a new key $n \notin L$ and $L' = L \cup \{n\}$, then represent $\langle \alpha, \{\langle l_1, v^{l_1} \rangle, \ldots, \langle l_n, v^{l_n} \rangle\} \rangle$ as $\{\langle n, \alpha \rangle, \langle l_1, v^{l_1} \rangle, \ldots, \langle l_n, v^{l_n} \rangle\}$.

Since $\mathbb{R}$ is embedded into V via representation of $\alpha \in \mathbb{R}$ as $\langle \alpha, 0 \rangle$, space V is isomorphic to a subspace of $L' \to V$. This isomorphism is why we call this space a space of *recurrent maps*: every element of V is represented as a finitary map from extended alphabet L' to the space V itself.

This representation of V via finitary maps to V is fundamental to our constructions in the present paper, because it translates directly to our Clojure implementation of core DMMs primitives (DMM 2016–2017) and because it allows us to introduce *variadic neurons*.

Implementation. We implement elements $v \in V$ as recurrent maps. Usually, programming languages provide *dictionaries* or *hash-maps* suitable for this purpose. In our Clojure implementation, elements $v \in V$ are represented by hash-maps, which map elements of L' to V.

Typically, L will be the set of all legal hash-map keys available in our language with the exception of a few keys reserved for other purposes. In particular, we reserve Clojure keyword `:number` to be mapped into the scalar component of a pair $\langle \alpha, \{\langle l_1, v^{l_1} \rangle, \ldots, \langle l_n, v^{l_n} \rangle\} \rangle$. So, in our case $L' = L \cup \{$`:number`$\}$.

Therefore, $\langle \alpha, \{\langle l_1, v^{l_1} \rangle, \ldots, \langle l_n, v^{l_n} \rangle\}\rangle$ is represented in Clojure by the hash-map `{:number` $\alpha, l_1\ v^{l_1}, \ldots, l_n\ v^{l_n}$`}`.

Here $v^{l_1}, \ldots, v^{l_n}$ are represented by similar hash-maps themselves resulting in *nested hash-maps*, and keys from L can have rather complex structure, if desired, taking advantage of great variety of hash-map keys allowed in Clojure.

When element $v \in V$ is simply a scalar (the pair $\langle \alpha, 0 \rangle$), the implementation is allowed to simply use number α instead of the hash-map `{:number` α`}`.

Example from section 3.2. The sum $(\rightsquigarrow 3.5) + ($`:foo` $\rightsquigarrow 2) + ($`:foo` $\rightsquigarrow$ `:bar` $\rightsquigarrow 7) + ($`:baz` $\rightsquigarrow$ `:foo` $\rightsquigarrow$ `:bar` $\rightsquigarrow -4)$ is represented as Clojure hash-map `{:number 3.5, :foo {:number 2, :bar 7}, :baz {:foo {:bar -4}}}`.

Variadic Neurons. We use the formalism of V-values to eliminate the need to keep track of the number of input and output arguments of the activation functions. We describe variadic neurons and DMMs based on variadic neurons in section 4.

To conclude the present section, V-values are essentially a dictionary-based version of S-expressions. Section 5.3 removes the restriction that all atoms must be numbers and allows to incorporate complex objects under reserved keywords.

4. Variadic neurons

The activation functions of variadic neurons transform a single stream of V-values into a single stream of V-values.

However, the labels at the first level of those V-values are dedicated to serve as the names of input and output arguments. Therefore, a neuron is a priori *variadic* and can potentially handle a countable collection of inputs and produce a countable collection of outputs (although our usual restrictions of keeping the active part of the network finite would in practice limit those collections to finite at any given moment of time).

Here is an example of an activation function for neuron with two arguments, x and y, outputting two results, difference $(x - y)$ and negative difference $(y - x)$. This and all subsequent code examples in this paper are written in Clojure.[4] Assume for the purpose of this example that `my-minus`

---

[4] https://clojure.org

function is available to compute these subtractions of one V-value from another V-value:[5]

```
(defn symmetric-minus [input]
  (let [x (get input :x {})
        y (get input :y {})]
    {:difference (my-minus x y)
     :negative-difference (my-minus y x)}))
```

The `input` is a hash-map, representing a V-value. The arguments are sub-trees of `input` corresponding to $:x, :y \in L$. These subtrees are computed by the expressions `(get input :x {})` and `(get input :y {})`. If the subtree in question is not present, the empty hash-map `{}` is returned. The empty hash-map represents zero vector (zero element of V) in our implementation. The function outputs a V-value with two subtrees corresponding to `:difference, :negative-difference` $\in L$.

So, the arguments are combined into a single V-value, `input`, and the outputs are combined into a single return V-value.

The non-trivial aspect of this approach is that instead of mapping neuron outputs to neuron inputs, the network matrix **W** now maps sub-trees at the first level of the neuron outputs to subtrees at the first level of the neuron inputs. We shall see in the present section that the network matrix **W** naturally acquires a structure of multidimensional tensor under this approach.

4.1. Space *U*

Because we are going to use the keys at the first level of V-values as names of inputs and outputs, we do not want any top-level leaves, that is we do not want non-zero scalars ("tensors of rank 0") in our V-values. Hence we will be using space $U = L \to V$, namely we'll use values $u \in U$ as *inputs and outputs of the neuron activation function* (which will always have arity one), and those values u will contain *actual inputs and outputs of the neuron* at their first level.

Observe that $V \cong \mathbb{R} \oplus (L \to V) = \mathbb{R} \oplus U$, and that U therefore satisfies the equation $U \cong L \to (\mathbb{R} \oplus U)$.

---

[5] Generally, one would want to define `my-minus` in terms of DMM core primitives for V-values implemented in DMM (2016–2017): `(defn my-minus [x y] (rec-map-sum x (rec-map-mult -1 y)))`.

The activation functions of the neurons map U to U, transforming single streams of elements of U. The labels at the first level of the elements of U serve as names of inputs and outputs.

4.2. Multidimensional structure on **W**

The network matrix **W** must provide a linear map from the concatenation of the first levels of elements of U which are the outputs of all neurons, to the concatenation of the first level of elements of U which are the inputs of all neurons. In our example above, a neuron with `symmetric-minus` activation function would have two V-values at its input, one labeled by `:x` and one labeled by `:y`, assembled into one input map. Such a neuron would also have two V-values at its output, one labeled by `:difference` and one labeled by `:negative-difference`, assembled into one output map.

However, we consider an infinite collection of V-values on input of each neuron, with V-values labeled by all elements of L, and an infinite collection of V-values on output of each neuron, with V-values labeled by all elements of L, since nothing restricts activation functions from using any of those labels.

Now we want to take infinite collections of V-values on *output* of each neuron for all neurons and join those infinite collections together into a single infinite collection of V-values, and we shall apply matrix **W** to this unified infinite collection (imposing the usual condition about only a finite number of relevant elements or vectors being non-zero at any given moment of time).

We also want to take infinite collections of V-values on *input* of each neuron for all neurons and join those infinite collections together into a single infinite collection of V-values, and matrix **W** will produce this unified infinite collection each time it is applied to the collection described in the previous paragraph.

Below we follow closely the material from section 3.2 of Bukatin & Anthony (2017).

Consider *one input* situated on the first level of the element of U serving as the argument of the activation function for one neuron, and the row of the network matrix **W** responsible for computing *that input* from the concatenation of the first levels of elements of U which are the outputs of all neurons.

The natural structure of indices of this row is not flat, but hierarchical. At the very least, there are two levels of indices: neurons and their outputs.

We currently use three levels of hierarchy in our implementation: neuron types (which are Clojure *vars* referring to implementations of activation functions $U \rightarrow U$), neuron names, and names of the outputs. Hence, matrix rows are three-dimensional sparse arrays (sparse "tensors of rank 3") in our current implementation.

The natural structure of indices of the array of rows is also not flat, but hierarchical. At the very least, there are two levels of indices: neurons and their inputs. We currently use three levels of hierarchy in our implementation: neuron types, neuron names, and names of the inputs.

Hence, the network matrix $\mathbf{W}$ is a six-dimensional sparse array (sparse "tensor of rank 6") in our current implementation.

4.3. DMMs based on V-values and variadic neurons

The DMM is a "two-stroke engine" similar to that of section 2, and consists of a linear "down movement" followed by an "up movement" performed by the activation functions of neurons. This "two-stroke cycle" is repeated indefinitely (Figure 4).
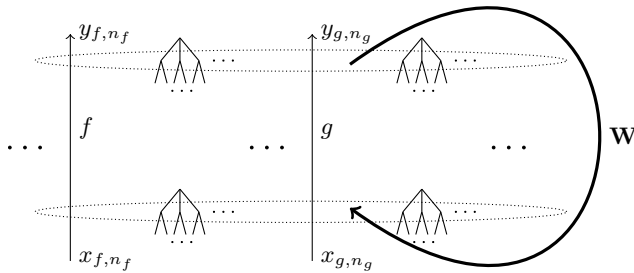

Figure 4: "Two-stroke engine" for a DMM based on variadic neurons. Two neurons, n_f and n_g, are explicitly pictured. Their inputs and outputs, $x_{f,n_f}, x_{g,n_g}, y_{f,n_f}, y_{g,n_g}$, are depicted as trees belonging to U. $\mathbf{W}$ is a linear map from the concatenation of the first levels of all y_{h,n_h} trees to the concatenation of the first levels of all x_{h,n_h} trees (Equation 4).

We allow names for neurons and their inputs and outputs to be any elements of L. The address space is such that the network is countably-sized, but since the network matrix $\mathbf{W}$ has only a finite number of non-zero elements at any given time, and elements of U have only a finite number of

non-zero coordinates at any given time, we are always working with finite representations.

The network matrix $\mathbf{W}$ (sparse "tensor of rank 6") depends on t, and its element $w^t_{f,n_f,i;\,g,n_g,o}$ is non-zero, if the output o of neuron n_g with the built-in activation function g is connected to the input i of neuron n_f with the built-in activation function f at the moment of time t, with number $w^t_{f,n_f,i;\,g,n_g,o}$ being the non-zero weight of this connection.

On the "down movement", the network matrix $(w^t_{f,n_f,i;\,g,n_g,o})$ is applied to an element of U which contains all outputs of all neurons. The result is an element of U which contains all inputs of all neurons to be used during the next "up movement". Each of those inputs is computed using the following formula:

$$x^{t+1}_{f,n_f,i} = \sum_{g \in F} \sum_{n_g \in L} \sum_{o \in L} w^t_{f,n_f,i;\,g,n_g,o} * y^t_{g,n_g,o}. \tag{4}$$

Indices f and g belong to the set of neurons types F, which is simply the set of transformations of U. Potentially, one can implement countable number of such transformations in a given programming language, but at any given time only finite number of them are defined and used. Indices n_f and n_g are the names of input and output neurons, and indices i and o are the names of the respective input and output arguments of those neurons.

In the formula above, $w^t_{f,n_f,i;\,g,n_g,o}$ is a number (the connection weight), and $x^{t+1}_{f,n_f,i}$ and $y^t_{g,n_g,o}$ are elements of V (not necessarily of U, since the presence of scalars is allowed at this level). The product $w^t_{f,n_f,i;\,g,n_g,o} * y^t_{g,n_g,o}$ multiplies vector $y^t_{g,n_g,o}$ by real number $w^t_{f,n_f,i;\,g,n_g,o}$.

This operation is performed for all $f \in F$, all $n_f \in L$, all input names $i \in L$, such that the corresponding three-dimensional row of $\mathbf{W}$ has some non-zero elements at time t.

The result is finitely sized map $\{f \mapsto \{n_f \mapsto x^{t+1}_{f,n_f}\}\}$, where each x^{t+1}_{f,n_f} is a finitely sized map from the names of neuron inputs to the values of those inputs, $\{i \mapsto x^{t+1}_{f,n_f,i}\}$.

On the "up movement", each f is simply applied to the elements of U representing the single inputs of the activation function f. This application is performed for all neurons $\langle f, n_f \rangle$ which are present in the finitely sized map described in the previous paragraph:[6]

---

[6] The neuron is fully determined by a pair $\langle f, n_f \rangle$. For each $f \in F$, there can be many active neurons $\langle f, n_f \rangle$ with different n_f. For each $n_f \in L$, there can be many active neurons $\langle f, n_f \rangle$ with different f.

$$y_{f,n_f}^{t+1} = f(x_{f,n_f}^{t+1}). \tag{5}$$

An example of an activation function of a variadic neuron was given in the beginning of the present section. For more examples of this kind see section 6.1 and section 6.3.

We gave the construction for the space V, but similar constructions also work for more general variants described in section 5.3.

5. Linear streams

Informally speaking, we say that a space of streams is a *space of linear streams*, if a meaningful notion of linear combination of several streams with numerical coefficients is well-defined.

We would like to formalize this notion, while keeping the following examples in mind:

– The space of sequences of numbers;
– For a vector space V, the space of sequences of its elements, $(v_1, v_2, \ldots)$;
– For a measurable space X, spaces of samples and signed samples drawn from X.

We consider linear streams in somewhat limited generality here. First of all, we consider discrete sequential time, while other models of time, e.g., continuous, are also potentially of interest.[7] We consider linear streams over real numbers, while other systems of coefficients (especially, complex numbers) can also be quite fruitful. Finally, we ground each space of linear streams in a vector space, while one could consider a more abstract approach to the notion of linear combination, where one works solely with streams of abstract representations without grounding them in a vector space.

To define a particular *kind* of linear streams k in a more formal manner, we specify background vector space V_k and streams of *approximate representations* of the underlying vectors from V_k. The approximate representations provide some information about the underlying vectors. Moreover, for every kind of linear streams k, we specify a procedure computing an approximate representation of a linear combination $\alpha_1 v_{1,k} + \ldots + \alpha_n v_{n,k}$ from

---

[7] We understand streams as functions from time to a set of objects, so that an object corresponds to any given moment of time. In our paper, time tends to have a starting point, to be discrete, and to continue indefinitely, so we usually model time by nonnegative integers starting from 0 or 1.

approximate representations of vectors $v_{1,k}, \ldots, v_{n,k}$. We say informally that the approximate representations in question belong to a vector-like space and we call them *vector-like elements*.

5.1. Streams of samples and signed samples

First, let us consider streams of samples from sequences of probability distributions over an arbitrary measurable space X and their linear combinations with positive coefficients.

A sequence of probability distributions, $(\mu_1, \mu_2, \ldots)$, can be represented by a sequence of elements of X sampled from those distributions, $(x_1, x_2, \ldots)$. Note that X is not required to be a vector space. Consider $0 < \alpha < 1$ and sequences of probability distributions $(\mu_1, \mu_2, \ldots)$ and $(\nu_1, \nu_2, \ldots)$ represented by streams of samples $(x_1, x_2, \ldots)$ and $(y_1, y_2, \ldots)$. Produce a stream of samples representing the sequence of probability distributions $(\alpha * \mu_1 + (1-\alpha) * \nu_1, \alpha * \mu_2 + (1-\alpha) * \nu_2, \ldots)$ as follows. Sample a random number uniformly from [0,1], and if this number is smaller than α, pick x_1 as the first element of our stream, otherwise pick y_1. Repeat this procedure for x_2 and y_2, and so on. Let us call this procedure a *stochastic linear combination* of the streams of samples.

To include this example into our framework, we need a background vector space, and we need stochastic linear combinations with positive and negative coefficients to be well-defined.

Let us observe that probability distributions over some measurable space X belong to the vector space of finite signed measures[8] over X. Let us consider *signed samples*, that is, samples marked as being positive or negative.

Streams of signed samples are mentioned in section 1.2 of Bukatin & Matthews (2015c). Here we follow a more detailed treatment of them given in Appendix A.1 of Bukatin & Anthony (2017). The underlying vector space is the space of all finite signed measures over arbitrary measurable space X, and samples are pairs $\langle x, s \rangle$, with $x \in X$ and flag s taking values -1 and 1.

One considers streams of finite signed measures over X, $\mu_1, \ldots, \mu_n$, and streams of corresponding samples, $\langle x_1, s_1 \rangle, \ldots, \langle x_n, s_n \rangle$.

The procedure of computing a sample representing a signed measure $\alpha_1 * \mu_1 + \ldots + \alpha_n * \mu_n$ is as follows. We pick index i with probability $\mid \alpha_i \mid / \sum_j \mid \alpha_j \mid$ using the absolute values of coefficients α, and we pick

---

[8] measures which can take any finite real values, including negative values

the sample $\langle x_i, \mathrm{sign}(\alpha_i) * s_i \rangle$ (reversing the flag if the selected value α_i is negative) to represent the linear combination $\alpha_1 * \mu_1 + \ldots + \alpha_n * \mu_n$.

For further discussion of expressive power of signed measures and signed samples see section 1.2 of Bukatin & Matthews (2015c). Issues related to missing samples and zero measures are mentioned in Appendix A.2 of Bukatin & Anthony (2017). A generalization to complex-valued measures and linear combinations with complex coefficients is considered in the design notes for DMM project (see DMM 2016–2017).[9]

5.2. Embedding

The ability to represent characters, words, and other objects of discrete nature as vectors is one of the cornerstones of success of modern neural networks.

The embedding of characters into a vector space generated by the alphabet ("one-hot encoding") is a basis for rather spectacular results obtained by modern forms of recurrent neural networks, such as LSTMs (Karpathy 2015). The ability to learn an optimal embedding of words into vectors is an integral part of a number of applications of recurrent neural nets to linguistics (Mikolov et al. 2013).

When it comes to representing compound structures in vector spaces, there is obviously a lot of freedom and variety. Throughout this paper, we work with embeddings of classes of dataflow matrix machines (considering each time a class of DMMs over some signature of neuron types) into corresponding vector spaces. One can argue that a class of dataflow matrix machines forms a sufficiently rich space of objects, and that since one finds a meaningful natural embedding of such a space into a vector space, one should expect to be able to find meaningful natural embeddings for a large variety of spaces of objects.

As the previous subsection indicates, the notion of embedding into linear streams is more general than the notion of embedding into vector spaces. As long as one is willing to consider a stream of objects of arbitrary nature as drawn from some sequence of probability distributions over those objects, this stream belongs to a space of linear streams of samples equipped with the stochastic version of linear combination.

Hence one is able to obtain an embedding of a stream of objects into a space of linear streams without embedding individual objects into a vector space.

---

[9] https://tinyurl.com/y8bfwmre

5.3. A family of spaces of V-values

The space V is very expressive, but sometimes it is not enough. If one wants to accommodate vectors from some other vector space V', a convenient way to do so is to allow elements of V' in the leaves of the prefix trees. Usually one still wants to be able to have just numbers inside the leaves as well, so one uses $\mathbb{V} = \mathbb{R} \oplus V'$ for the space of leaves.

This means that elements of $\mathbb{V}$ are now used as coefficients α instead of real numbers in $\alpha \cdot (l_1 \ldots l_n) = l_1 \rightsquigarrow \ldots \rightsquigarrow l_n \rightsquigarrow \alpha$. Considering a basis in $\mathbb{V}$, one can see that this whole construction corresponds to tensor product $V \otimes \mathbb{V}$.

Elements of $\mathbb{V}$ are also used as elements of sparse "tensors of mixed rank". The equation $V \cong \mathbb{R} \oplus (L \to V)$ becomes $V \cong \mathbb{V} \oplus (L \to V)$.

Implementation-wise, if, for example, $\mathbb{V} = \mathbb{R} \oplus V' \oplus V'' \oplus V'''$, then in addition to the reserved key `:number`, one would reserve additional keys for each of the additional components V', V'', V''' to incorporate the non-zero instances of those components into leaves (and into hash-maps).

For example, Appendix A.3 of Bukatin & Anthony (2017) shows how to accommodate signed samples discussed in section 5.1 above within this framework. One considers $\mathbb{V} = \mathbb{R} \oplus M$, where M is the space of finite signed measures over X, and one uses the reserved keyword `:sample` to incorporate signed samples into the leaves as necessary.

So, in this fashion a space $V \cong \mathbb{V} \oplus (L \to V)$ will still be represented by nested hash-maps, and we are still going to call those nested hash-maps *V-values*.

6. Programming

Here we discuss some of the programming patterns in dataflow matrix machines. Further programming tools come from the presence of self-referential mechanism (section 7) and from modularization facilities (section 9).

6.1. Linear and multiplicative constructions

Linear and multiplicative constructions in dataflow matrix machines are well-covered in Bukatin et al. (2016d). The most fundamental of them is a neuron with *identity activation function*. Consider some argument name, e.g., `:accum`. If we connect output `:accum` of such a neuron to its

input :accum with weight 1, this neuron becomes an *accumulator*. It adds together and accumulates in its :accum arguments the contributions to its input :accum made during each "two-stroke cycle" by all other outputs in the network connected to the input :accum of our neuron by non-zero weights.

Among multiplicative constructions, the most fundamental one is multiplication of an otherwise computed neuron output by the value of one of its scalar inputs. This is essentially a fuzzy conditional, which can selectively turn parts of the network on and off in real time via multiplication by zero, attenuate or amplify the signal, reverse the signal via multiplication by -1, redirect flow of signals in the network, etc. For further details see Bukatin et al. (2016d).
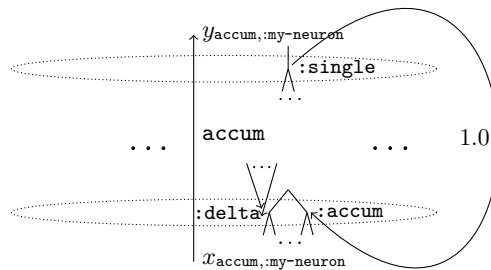

Figure 5: Connectivity of a neuron [accum :my-neuron] with activation function accum, neuron name :my-neuron, and input arguments :accum and :delta, when this neuron is used as an accumulator

These facilities are quite powerful even for scalar flows of reals, and even more so for vector flows. The lack of these facilities hinders the approaches which insist on only having non-linear activation functions, or on only having neurons with a single input.

Sometimes, it is convenient to take advantage of having multiple inputs and use a neuron with $+$ activation function, $y = x + \Delta x$, as an accumulator, connecting y to x with weight 1 and accepting contributions from other outputs in the network on input Δx (Bukatin et al. 2016c). For details of this connectivity pattern in the current DMM architecture with variadic neurons see Figure 5 .

The code for the activation function accum of this accumulator looks as follows (DMM 2016–2017):

```
(defn accum [input]
   {:single (rec-map-sum (input :accum) (input :delta))})
```

The function `accum` takes a V-value `input` as argument, then it adds together the values of the two neuron inputs,[10] `(input :accum)` and `(input :delta)`, obtaining the desired result, and then returns `{:single result}` as the output. It can add together arbitrary elements of V.

In this example, the `:accum` name for a neuron input stands for x, the `:delta` name for a neuron input stands for Δx, and the `:single` is the name of the only output this particular neuron has.

If we want a particular neuron `:my-neuron` with built-in activation function `accum` to work as an accumulator, then the element of the network matrix connecting the output `[accum :my-neuron :single]` together with the input `[accum :my-neuron :accum]` should be 1 (see Figure 5). The corresponding matrix element $w_{\text{accum},:\text{my-neuron},:\text{accum};\ \text{accum},:\text{my-neuron},:\text{single}}$ is expressed as

```
{v-accum {:my-neuron {:accum
  {v-accum {:my-neuron {:single 1}}}}}}
```

in our current Clojure implementation, where `v-accum` is defined as

```
(def v-accum (var accum))
```

because the value of `accum` itself used as a hash-key is not stable from recompilation to recompilation, whereas `(var accum)` is stable.

The contributions from other neurons are accepted on the input `:delta`.

6.2. Sparse vectors

The ability to handle sparse vectors in a straightforward manner is not available in scalar-based neural networks. One has to allocate a neuron for every coordinate, and there is no straightforward way to avoid processing those coordinates which happen to be zero at the moment.

In DMMs one can handle compact sparse vectors of very high, or even infinite dimensions, and the network size does not depend on those dimensions. An example of a compact DMM built around an accumulator

---

[10] `rec-map-sum` is one of DMM core primitives for V-values implemented in DMM (2016–2017) which performs addition of V-values

of sparse vectors in the vector space generated by a given alphabet in order to keep track of the number of occurrences of each character in a given string is studied in detail in Bukatin et al. (2016d). The savings can be drastic, as, for example, Unicode alphabet exceeds 100,000 characters, and with sparse representation only the actually occurring characters are stored and processed.

The same technique can be applied to keeping track of the number of occurrences of each word in the text, where the underlying vector space generated by all possible words is infinitely-dimensional.

6.3. Data structures

An earlier paper (Bukatin et al. 2016d) focuses on allocating data structures in the body of the network itself, an approach encouraged by the network's potentially infinite size and powerful self-modification facilities.

However, with the ability to process complicated vectors such as V-values, it is natural to encode and process data structures on that level. The use of structure sharing immutable data structures, the default in Clojure, should make passing complicated structures through "up movements" and "down movements" reasonably efficient, as seen in our own preliminary explorations.

For example, a list can be encoded by using `:this` and `:rest` keys on the same level of a V-value, and having neurons with `first`, `rest`, and `cons` activation functions. The only decision one needs to make is whether to consider all lists as having infinite sequences of zeros at the tail, or whether to incorporate an explicit list terminator (e.g., a keyword `:end` mapped to `1`) in the formalism.

For example, the following activation function for the neuron used to accumulate a list of "interesting" events is similar to the function used in our example of a DMM accumulating a list of mouse clicks:[11]

```
(defn dmm-cons [accum-style-input]
  (let [old-self (get accum-style-input :self {})
        new-signal (get accum-style-input :signal {})]
    (if (interesting? new-signal)
      {:self {:this new-signal :rest old-self}}
      {:self old-self})))
```

In order to maintain the accumulator metaphor, the `:self` output and the `:self` input of the corresponding neuron are connected with weight 1.

---

[11] https://tinyurl.com/yxgagfjl

Any linked structures which can be encoded inside the network matrix, can be encoded inside a similar matrix not used as the network matrix (the only difference is that data structures encoded within the network matrix tend to be "active", as they are built over actively working neurons; this difference can potentially be profound).

6.4. From programming via composition of transformers of streams of V-values to Dataflow Matrix Machines

There is a rather long history of programming via composition of transformers of linear streams. In each of those cases, linear streams can be represented as sufficiently general streams of V-values.

Perhaps the most well known example is the discipline of audio synthesis via composition of *unit generators*, which are transformers of streams of audio samples (streams of numbers, if one considers a single monophonic channel). That discipline was created by Max Mathews in 1957 at Bell Labs (Mathews 1963). It is typical for a modern audio synthesis system to be crafted along those lines, even though the syntax can differ greatly. One of the classical textbooks in that discipline is Farnell (2010). We found the tutorial (Merz 2011) on Beads, a realtime audio and music library for Java and Processing, to be a convenient introductory text.

In this cycle of studies we explored a number of different examples of programming via composition of linear streams, starting from dataflow programming of animations via composition of transformers of image streams (Bukatin & Matthews 2015a). During that series of experiments we discovered that if one does not want to impose the condition of the dependency graph being acyclic, then one needs to maintain two elements of each stream at any given time, the "current" element, used by the transformers depending on that stream in their computations, and the "next" element, the element which is being computed. Then, after transformers computed their respective "next" elements, there is a *shift* operation which makes all the "next" elements current.

This cycle "transform-shift" is a version of the two-stroke cycle used in this paper, and the shift operation is what the "down movement" of a DMM would look like, if all non-zero matrix rows would only have a single non-zero element in each of them, and that element would be 1. In fact, any program built as a composition of transformers of linear streams can be converted into an equivalent DMM by inserting a linear transformation described by a matrix row with one non-zero element with weight 1 at each connection.

A number of examples of DMMs we explored in this cycle of studies, such as the character-processing DMM described in section 3 of Bukatin et al. (2016d) or the DMM accumulating a list of mouse clicks mentioned in section 6.3 of the present paper, come from programs built as compositions of transformers of linear streams converted to DMMs by inserting weight 1 connectors.

One aspect which used to be somewhat limited within the discipline of programming via composition of transformers of linear streams was higher-order programming, i.e., transforming the transformers. In particular, the higher-order constructions themselves were not expressed as transformations of linear streams.

Dataflow matrix machines allow to continuously transform any composition of stream transformers into any other composition of stream transformers, and we present one way to do so *within the discipline of transforming linear streams* and in a self-referential manner in the next section.

7. Self-referential mechanism

The ability to handle arbitrary linear streams implies the ability to handle streams of vectors shaped like network connectivity matrices (be those flat two-dimensional matrices, or sparse multidimensional tensors described in section 4). This enables a rather straightforward mechanism to access and modify the network matrix $\mathbf{W}$. We designate a neuron `Self` emitting a stream of such matrices, and use the most recent value from that stream as $\mathbf{W}$ for the purpose of the next "down movement" step.

We currently prefer to use an accumulator with $+$ activation function $y = x + \Delta x$ as `Self` following Bukatin et al. (2016c) (see section 6.1). `Self` takes additive updates from other neurons in the network on its Δx input, and other neurons can take the stream of the current values of $\mathbf{W}$ from the output of `Self` making them aware of the current state of the network connectivity.

Network self-modification based on the streams of network matrices was first introduced in Bukatin et al. (2016b), and the principle of "self-referential completeness of the DMM signature relative to the language available to describe and edit the DMMs" was formulated there. That principle states that it is desirable to have a sufficient variety of higher-order neurons to perform updates of the network matrix, so that any modifications of a DMM (understood as a network or as a program) can be made by triggering an appropriate higher-order neuron.

Paper Bukatin et al. (2016d) explored ways for the network to modify itself by making deep copies of its own subgraphs. The possibility of using self-referential matrix transformations as a new foundation for programming with linear streams, somewhat similarly to lambda-calculus being a foundation for symbolic programming, was studied in Bukatin et al. (2016c).

Also, it was demonstrated in Bukatin et al. (2016c) that this self-referential mechanism together with a few constant update matrices gives rise to a wave pattern dynamically propagating in the network matrix; this result was also verified in computer experiments (see Appendix B of Bukatin & Anthony (2017) for a more polished presentation).

However, all these studies were so far merely scratching the surface of what is possible with this mechanism. In principle, it should allow the network to maintain an evolving population of its own subnetworks, to maintain an evolving population of network update methods, to train network update methods as a linear combination of available network update primitives, etc.

We hope that some of this potential will be explored in the future.

8. Network topology

The network topology, such as layers, is defined by the pattern of sparsity of **W**: some of the connectivity weights are kept at zero, and some are allowed to deviate from zero, and the network topology comes from that.

However, in the model of synchronous time we follow in this paper, all layers still work simultaneously. If it is desirable for layers to work strictly in turn, one can use multiplicative mechanisms described in section 6.1 to orchestrate the computations by turning off the layers at the appropriate moments of time using zero *multiplicative masks*, and then by further optimizing implementation to save processing time in such situations.

Not all weights which are non-zero need to be variable weights. It is often the case that some of the non-zero weights are set to 1, and then it is the user's choice which of those should be allowed to vary. The particularly frequent are the cases when the weight 1 in question is the only non-zero element of its matrix row. The examples of cases where weights set to 1 occur naturally include

– accumulators (sections 6.1, 6.3);
– cases when a program expressed as composition of V-value transform-ers is translated into a DMM (section 6.4);
– special neural network topology.[12]

9. Subnetworks and modularization

The modern trend in artificial neural nets is to build the networks not from a large number of single neurons, but from a relatively small number of modules such as layers, etc.

In this sense, it is convenient that neurons in DMMs are powerful enough to express the "up movement" action of whole subnetworks. This allows to build DMMs from a relatively small number of powerful neurons, if so desired.

In the old style DMMs (Bukatin et al. 2016a;b;c;d), the neurons had fixed arity and were powerful enough to express the "up movement" action of the subnetworks of fixed size. However, the networks themselves became variadic networks of unbounded size, so the gap between single neurons and general subnetworks remained.

With variadic neurons this particular gap is eliminated.

In 2016, 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 V-values and variadic neurons allows DMMs to *fulfill this principle in the following limited sense*: single neurons are powerful enough to express one "up movement" action of any subnetwork as one "up movement" action of an appropriately crafted single neuron.

---

[12] For example, let us express LSTMs and Gated Recurrent Unit networks as networks built from sigmoid neurons, linear neurons, and neurons performing multiplication for gating following Appendix C of Bukatin et al. (2016c), and let us assume that we are writing this in terms of neurons processing scalar streams (streams of numbers). Then each of the two inputs for each neuron performing multiplication of scalar streams is connected with weight 1 from a single output of an appropriate neuron.

10. Learning

There are various indications that dataflow matrix machines have strong potential for future machine learning applications.

DMMs contain well-known classes of neural networks with good machine learning properties, such as LSTM and Gated Recurrent Unit networks (Appendix C of Bukatin et al. 2016c).

At the same time, they allow to naturally express a number of various algorithms within a formalism which allows arbitrarily small modifications of programs, where one can transform programs continuously by continuously transforming the matrices defining those programs.

The presence of well-developed self-referential facilities means that network training methods can be made part of the network itself, making this a natural setting for a variety of "learning to learn" scenarios.

DMM architecture is conductive to experiments with "fast weights" (e.g., Ba et al. 2016).

Recently, we have been seeing very interesting suggestions that synthesis of small functional programs and synthesis of neural network topology from a small number of modules might be closely related to each other (Olah 2015; Nejati 2016). The ability of single DMM neurons to represent neural network modules suggests that DMMs might provide the right degree of generality to look at these classes of problems of network and program synthesis.

We are seeing evidence that syntactic shape of programs and their functionality provide sufficient information about each other for that to be useful during program synthesis by machine learning methods (e.g., Murali et al. 2017). If a corpus of human-readable programs manually written in the DMM architecture emerges eventually, this should be quite helpful for solving the problem of synthesis of human-readable programs performing non-trivial tasks.

Given that DMMs form a very rich class of computational models, it makes sense to search for its various subclasses for which more specialized methods of machine learning might be applicable.

11. Historical remarks and related work

There are two ways one can arrive at the dataflow matrix machines. One can focus on programming with linear streams and then notice that by interleaving linear and non-linear transformations of those streams, it is

possible to obtain parametrization of large classes of programs by matrices of numbers.

Another way is to focus on recurrent neural networks as a programming platform and to try to generalize them as much as possible while retaining their key characteristic, which is parametrization by connectivity matrices of network weights.

In this section of the present paper we discuss some of the related work under both of these approaches.

11.1. Recurrent neural networks as programs

It was recognized as early as 1940s, that if one provides a neural network with a suitable model of unbounded memory one obtains a Turing-universal formalism of computations (McCulloch & Pitts 1943). Research studies formally establishing Turing-universality for neural networks processing streams of reals include Pollack (1987); Siegelmann & Sontag (1995).

More recent studies include such well-known approaches as Graves et al. (2014); Weston et al. (2014), which are currently under active development.

Yet, as these approaches are gradually becoming more successful at learning neural approximations to known algorithms, they do not seem to progress towards human-readable programs. In fact, it seems that while the expressive power of scalar-based neural networks is sufficient to create Turing-complete esoteric programming languages, they are not expressive enough to become a pragmatic programming platform. The restriction to scalar flows seems to either necessitate awkward encoding of complex data within reals (as in Siegelmann & Sontag 1995, where binary expansions of real numbers are used as tapes of Turing machines), or to force people to create networks depending in their size on data dimensionality and with any modularization and memory capabilities being external to the network formalism, rather than being native to the networks in question.

The awkward encodings of complex data within reals hinder the ability to use self-modification schemas for scalar-based neural networks. E.g., a remarkable early paper (Schmidhuber 1993) has to use such encodings for addresses in the network matrix, and such encodings lead to very high sensitivities to small changes of numbers involved, while the essence of correct neural-based computational schemas is their robustness in the presence of noise.

Even such natural constructions within the scalar flow formalism as neurons with linear activation functions, such as identity, and neurons performing multiplication of two arguments, each expressing a different linear combination of neuron outputs, encounter resistance in the field.

The power of linear and multiplicative neurons was well understood at least as early as 1987 (Pollack 1987). Yet, when the LSTMs were invented in 1997 (Hochreiter & Schmidhuber 1997), the memory and gating mechanisms were understood as mechanisms external to neural networks, rather then the mechanisms based straightforwardly on neurons with linear activation functions for memory and neurons performing multiplication for gating, which provide a natural way to express memory and gating mechanisms in neural nets (see Appendix C of Bukatin et al. 2016c).

Recently, the power of having linear activation functions, in particular identity, in the mix together with other activation functions is finally getting some of the recognition it deserves (the paper, He et al. 2016, is now a well-cited paper). However, the explicit activation functions of arity two, such as multiplication, are still quite exotic and often difficult to explicitly incorporate into existing software frameworks for neural networks.

We think that dataflow matrix machines as presented here, with their vector flows and multiple arities for activation functions, provide the natural degree of generality for neural networks.

11.2. Programming with linear streams

Continuous computations (which tend to be computations with linear streams) have a long history, starting with electronic analog computers. The programs, however, were quite discrete: a pair of single-contact sockets was either connected with a patch cord, or it was not connected with a patch cord.

More modern dataflow architectures focusing on work with linear streams representing continuous data include, for example, LabVIEW (Johnston et al. 2004) and Pure Data (e.g., Farnell 2010). The programs themselves are still quite discrete.

To incorporate higher-order programming methods within the paradigm of programming with linear streams, the space of programs themselves needs to become continuous. Neural networks represent a step in this direction. While the network topology itself is discrete (the connection between nodes is either present, or not), when one expresses the network topology via its weight-connectivity matrix, the degree to which any particular edge is present (the absolute value of the weight associated with

it) can be made as small as desired, and this provides the continuity we are after.

The particular line of work we are presenting in this paper emerged in 2012–2013, when it was recognized by our group that the approximation domains providing continuous denotational semantics in the theory of programming languages can acquire the structure of vectors spaces, when equipped with cancellation properties missing in the standard theory of interval numbers. Namely, there must be enough overdefined (partially inconsistent) elements in those spaces to produce zero on addition by the mechanism of cancellation with underdefined (partially defined) elements. For interval numbers, this corresponds to introduction of pseudosegments $[a, b]$ with the contradictory property that $b < a$, following Warmus (1956). For probabilistic spaces, this corresponds to allowing negative values of probabilities in addition to usual non-negative values. The mathematics of the resulting *partial inconsistency landscape* is presented in section 4 of Bukatin & Matthews (2015c).

By 2015 it became apparent to our group that programming with linear streams was a rich formalism which included programming with streams of probabilistic samples and programming with *generalized animations.* This framework seemed to provide methods for *continuous higher-order programming*, and, moreover, it had good potential for obtaining more efficient schemas for genetic programming by allowing to introduce the motives similar to *regulation of gene expression* into genetic programming frameworks (Bukatin & Matthews 2015c). Crucially, it also became apparent at that time that if one introduced the discipline of interleaving linear and non-linear transformations of linear streams, then one could parametrize large classes of programs by matrices of numbers (Bukatin & Matthews 2015b;c).

The first open-source software prototypes associated with this approach also appeared in 2015 (Fluid 2015–2017).

In 2016 we understood that the resulting formalism generalized recurrent neural networks, and the term *dataflow matrix machines* was coined (Bukatin et al. 2016a). The modern version of the self-referential mechanism in DMMs and the first precise description of how dataflow matrix machines function, given that their matrices can be dynamically expanded, appeared in Bukatin et al. (2016b). The programming patterns for the resulting software framework were studied in Bukatin et al. (2016d). A more theoretical paper (Bukatin et al. 2016c) explored the possibility of using self-referential matrix transformations instead of lambda-calculus as the foundation in this context and established further connections

between neural networks and the mathematics of the partial inconsistency landscape.

The formalism of finite prefix trees with numerical leaves (the vector space of recurrent maps) was introduced in the Fall of 2016. This formalism was inspired by our work with Clojure programming language (Hickey 2018). The first open-source implementation of a version of DMMs based on that formalism and written in Clojure was produced in that time frame (DMM 2016–2017), and a research paper based on this architecture was presented recently at LearnAut 2017 (Bukatin & Anthony 2017).

Acknowledgements

References

Ba, J., G. Hinton, V. Mnih, J. Leibo and C. Ionescu. 2016. Using fast weights to attend to the recent past. October 2016. https://arxiv.org/abs/1610.06258

Bukatin, M. and J. Anthony. 2017. Dataflow matrix machines as a model of computations with linear streams. LearnAut 2017 ("Learning and Automata" Workshop at LICS 2017). https://arxiv.org/abs/1706.00648

Bukatin, M. and S. Matthews. 2015a. Almost continuous transformations of software and higher-order dataflow programming. July 2015. https://arxiv.org/abs/1601.01050

Bukatin, M. and S. Matthews. 2015b. Dataflow graphs as matrices and programming with higher-order matrix elements. August 2015. https://arxiv.org/abs/1601.01050

Bukatin, M. and S. Matthews. 2015c. Linear models of computation and program learning. In G. Gottlob, G. Sutcliffe and A. Voronkov (eds.) GCAI 2015. Global Conference on Artificial Intelligence, vol. 36. 66–78.
https://easychair.org/publications/paper/Q4lW

Bukatin, M., S. Matthews and A. Radul. 2016a. Dataflow matrix machines as a generalization of recurrent neural networks. March 2016. http://arxiv.org/abs/1603.09002

Bukatin, M., S. Matthews and A. Radul. 2016b. Dataflow matrix machines as programmable, dynamically expandable, self-referential generalized recurrent neural networks. May 2016. https://arxiv.org/abs/1605.05296

Bukatin, M., S. Matthews and A. Radul. 2016c. Notes on pure dataflow matrix machines: Programming with self-referential matrix transformations. October 2016.
https://arxiv.org/abs/1610.00831

Bukatin, M., S. Matthews and A. Radul. 2016d. Programming patterns in dataflow matrix machines and generalized recurrent neural nets. June 2016.
https://arxiv.org/abs/1606.09470

DMM. 2016–2017. DMM project GitHub repository. https://github.com/jsa-aerial/dmm

Farnell, A. 2010. Designing sound. Cambridge, MA: MIT Press.

Fluid, X. 2015–2017. Project "Fluid" GitHub repository. https://github.com/anhinga/fluid

Graves, A., G. Wayne and I. Danihelka. 2014. Neural Turing machines. October 2014. https://arxiv.org/abs/1410.5401

He, K., X. Zhang, S. Ren and J. Sun. 2016. Identity mappings in Deep Residual Networks. March 2016. https://arxiv.org/abs/1603.05027

Hickey, R. 2008. The Clojure programming language. In Proceedings of the 2008 Symposium on Dynamic Languages. Association for Computing Machinery.

Hochreiter, S. and J. Schmidhuber. 1997. Long short-term memory. Neural Computation 9. 1735–1780.

Johnston, W., J. Hanna and R. Millar. 2004. Advances in dataflow programming languages. ACM Computing Surveys 36. 1–34.

Karpathy, A. 2015. The unreasonable effectiveness of Recurrent Neural Networks. May 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Kornai, A. 2008. Mathematical linguistics. London: Springer.

Mathews, M. 1963. The digital computer as a musical instrument. Science 142. 553–557.

McCulloch, W. and W. Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics 5. 115–133.

Merz, E. 2011. Sonifying processing: The Beads tutorial. http://evanxmerz.com/?page_id=18

Mikolov, T., I. Sutskever, K. Chen, G. S. Corrado and J. Dean. 2013. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani and K. Q. Weinberger (eds.) Advances in neural information processing systems 26. Red Hook, NY: Curran Associates. 3111–3119

Murali, V., S. Chaudhuri and C. Jermaine. 2017. Bayesian sketch learning for program synthesis. March 2017. https://arxiv.org/abs/1703.05698

Nejati, A. 2016. Differentiable programming. August 2016. https://pseudoprofound.wordpress.com/2016/08/03/differentiable-programming.

Olah, C. 2015. Neural networks, types, and functional programming. September 2015. http://colah.github.io/posts/2015-09-NN-Types-FP.

Pollack, J. 1987. On connectionist models of natural language processing. Doctoral dissertation. University of Illinois at Urbana-Champaign.

Schmidhuber, J. 1993. A "self-referential" weight matrix. In S. Gielen and B. Kappen (eds.) ICANN 93: Proceedings of the International Conference on Artificial Neural Networks. Berlin & New York: Springer. 446–450.

Siegelmann, H. and E. Sontag. 1995. On the computational power of neural nets. Journal of Computer and System Sciences 50. 132–150.

Warmus, M. 1956. Calculus of approximations. Bulletin of the Polish Academy of Sciences Cl III 4. 253–259.

Weston, J., S. Chopra and A. Bordes. 2014. Memory networks. October 2014. https://arxiv.org/abs/1410.3916