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

Michael Bukatin† Jon Anthony‡

July 7, 2017

Abstract

Dataflow matrix machines generalize neural nets by replacing streams of numbers by streams of vectors (or other kinds of *linear streams* admitting a notion of linear combination of several streams) and adding a few more changes on top of that, namely arbitrary input and output arities for activation functions, countable-sized networks with finite dynamically expandable active part, and a very expressive *self-referential mechanism*.

While recurrent neural networks are Turing-complete, they form an esoteric programming platform, not conductive for practical general-purpose programming. Dataflow matrix machines are more suitable as a general-purpose programming platform, although it remains to be seen whether this platform can be made fully competitive with more traditional programming platforms currently in use. At the same time, dataflow matrix machines retain the key property of recurrent neural networks: *programs are expressed via matrices of real numbers, and arbitrary small changes of those matrices and of the programs associated with those matrices are possible.*

Spaces of vector-like elements are of particular importance in this context. In particular, we focus on the vector space V of finite linear combinations of strings, which can be also understood as the vector space of finite prefix trees with numerical leaves, the vector space of "mixed rank tensors", or the vector space of recurrent maps $V \cong \mathbb{R} \oplus (L \to V)$. This space, and a family of spaces of vector-like elements derived from it, are sufficiently expressive to cover all cases of interest we are currently aware of, and allow a much more compact and streamlined version of dataflow matrix machines based on a single space of vector-like elements and variadic neurons. We call elements of this family of spaces *V-values*. Their role in our context is somewhat similar to the role of S-expressions in Lisp.

*Manuscript in preparation

†HERE North America LLC, Burlington, MA, USA
‡Boston College, Chestnut Hill, MA, USA

1 Introduction

András Kornai wrote his *Mathematical Linguistics* book [19] while he and the first author were working in the same office at MetaCarta during the previous decade and sharing many fruitful moments. The book published 10 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 2 years ago [4] and a series of technical papers had been written on the subject since then [5, 6, 7, 8, 2]. This article 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 neural networks are expressive enough to encode any algorithm, if they are equipped with a reasonable form of unbounded memory [20, 25, 27]. 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 all 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, a novel model of unbounded memory based on countable-sized weight-connectivity matrices with finite number of non-zero elements, 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 programs which can be continuously deformed 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.

1.1 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 two or several streams with numerical coefficients is well-defined.

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

A rather non-trivial example comes from an observation that probability distributions over some measurable space X belong to the vector space of finite signed measures over X. A sequence of probability distributions, $(\mu_1, \mu_2, \ldots)$, can be represented by the 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.

It turns out that by considering samples marked as being positive or negative, one can generalize this construction to arbitrary numerical coefficients and arbitrary vectors from the space of finite signed measures over X. The sequences of samples marked as being positive or negative form the space of linear streams in this case. For details of this and further discussion of linear streams see Section 2.

1.2 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 [18]. 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 [21].

When it comes to representing compound structures in vector spaces, there is obviously a lot of freedom and variety. Throughout this paper, we will consider embedding of dataflow matrix machines into vector spaces. One can argue that dataflow matrix machines is a sufficiently rich space of objects, and that since one finds a meaningful natural embedding of this space into vector spaces, one should expect to be able to find a meaningful natural embedding for a large variety of objects.

As the last example of 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 statistical version of the operation of taking linear combinations.

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.

1.3 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.

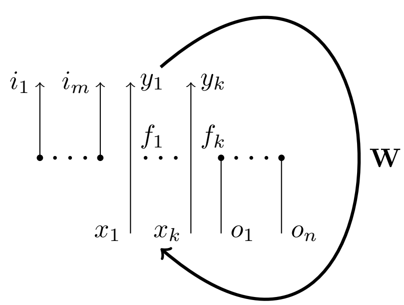

Figure 1: “Two-stroke engine” for an RNN [2].

Therefore, neural architectures such as *recurrent neural networks* (RNNs) can be viewed as “two-stoke engines” (Figure 1). 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: $(x_1^{t+1}, \dots, x_k^{t+1}, o_1^{t+1}, \dots, o_n^{t+1})^\top = \mathbf{W} \cdot (y_1^t, \dots, y_k^t, i_1^t, \dots, i_n^t)^\top$. 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: $y_1^{t+1} = f_1(x_1^{t+1}), \dots, y_k^{t+1} = f_k(x_k^{t+1})$.

This “two-stroke cycle” of a “down movement” followed by an “up movement” is repeated indefinitely.

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 $\mathbf{W}$.

In the above, all variables x_i^t, y_j^t, etc. are real numbers, so variables x_i, y_j, etc. are actually streams of reals parametrized by discrete time t.

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, what kind of linear streams is 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.

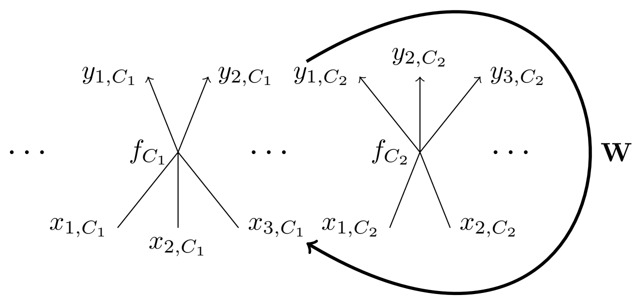

Figure 2: "Two-stroke engine" for a standard DMM [2]. Two of its neurons, C_1 and C_2, are explicitly shown.

The "two-stroke engine" (Figure 2) is a "two-stroke cycle" consisting of a "down movement" followed by an "up movement" which is repeated indefinitely. The connectivity matrix $\mathbf{W}$ depends on discrete time t. $\mathbf{W}$ is a countable-sized matrix, but we impose the condition that at any given moment of time t only finite number of its elements $w^t_{(i,C_k),(j,C_l)}$ are non-zero.

"Down movement" is defined as follows. For all inputs x_{i,C_k} such that there is a non-zero weight $w^t_{(i,C_k),(j,C_l)}$ (non-zero real number linking the output y_j of the neuron C_l and the input x_i of the neuron C_k at the moment t):

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

Note that x^{t+1}_{i,C_k} and y^t_{j,C_l} are no longer numbers, but elements of various linear streams, so in order for our 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.

We call a neuron active, if there is at least one non-zero connectivity weight from $\mathbf{W}$ associated with one of its inputs or outputs. Since $\mathbf{W}$ 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:

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

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 subsection allow us to achieve just that, while further increasing the power of a single neuron.

1.4 V-values

We start with defining a vector space which is sufficiently rich to represent vectors from many other spaces encountered in practice. Then we show how to enrich this construction in those situations, when it is not sufficiently universal.

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.

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, and some are "breadth-first", and they are more fundamental to us.

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 $V \cong \mathbb{R} \oplus (Ł \to V)$ admitting finite descriptions.

1.4.1 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 of the elements of L^* over $\mathbb{R}$.

1.4.2 Finite Prefix Trees with Numerical Leaves

One can think about $l_1 \dots l_n$ as a path in a prefix tree (*trie*), with nodes being labeled by letters from L. So when one considers $\alpha \in \mathbb{R}$, one can express the presence of $\alpha(l_1 \dots l_n)$ in our linear combination as presence of path with the intermediate nodes labeled by $l_1, \dots, l_n$ and the leaf labeled by α. 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.*

1.4.3 Sparse "Tensors of Mixed Rank"

Yet another "depth-first" way of looking at this situation is to consider $\alpha(l_1 \dots l_n)$ to be an element of sparse multidimensional array with n dimensions. E.g. $\alpha(l_1 l_2)$ 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.

Similarly, the empty string ϵ with non-zero coefficient β, written as $\beta(\epsilon)$ (the leaf with non-zero β attached directly to the root of the tree), is considered to be a scalar with value β.

Each string of length one with non-zero coefficient γ, that is $\gamma(l)$ (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 $\gamma'(l_1 l_2 l_3)$ 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 of L of different lengths, we have to talk about *sparse "tensors of mixed rank"*. This means that all usual vectors, matrices, and tensors of any dimension can be represented in our space.

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 rows to the sparse vectors expressing 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).

1.4.4 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 empty sequence from L^* belongs to v with non-zero coefficient α, in which case the prefix tree contains a path of no intermediate nodes and the leaf labeled by α, i.e. the corresponding coordinate of v contains non-zero "tensor of rank 0", which is the scalar.

Then for each letter $l_1 \in L$, such that a non-zero term $\beta(l_1 l_2 \dots l_n)$ belongs to v, we consider all terms from v which share the same first letter l_1, namely $\beta(l_1 l_2 \dots l_n), \gamma(l_1 l'_2 \dots l'_m), \dots$, and consider $v^{l_1} \in V$ consisting of those terms with the first letter removed, namely $\beta(l_2 \dots l_n), \gamma(l'_2 \dots l'_m), \dots$

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 $\langle l, 0 \rangle$ 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:

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

This representation of V is fundamental to our construction, because it translates directly to our Clojure implementation of core DMMs primitives and because it allows us to introduce *variadic neurons*.

Implementation. 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 the key `:number` to be mapped into the scalar component of a pair $\langle \alpha, \{\langle l_1, v^{l_1}\rangle, \ldots, \langle l_n, v^{l_n}\rangle\}\rangle$.

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, and keys from L can have rather complex structure, if desired, taking advantage of great variety of hash-map keys allowed in Clojure.

When the element of 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` $\alpha\}$.

Example. 3.5 + 2 (:foo) + 7 (:foo :bar) - 4 (:baz :foo :bar) is represented as `{: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. 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 (prefix trees, recurrent maps) 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).

The non-trivial aspect of this approach is that instead of mapping neuron outputs to neuron inputs, the network matrix $\mathbf{W}$ now maps subtrees at the first level of the neuron outputs to subtrees at the first level of the neuron inputs. We discuss variadic neurons in detail in Section 3, and in particular we shall see that the network matrix $\mathbf{W}$ naturally acquires a structure of multidimensional tensor under this approach.

1.4.5 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 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 $\alpha(l_1 \ldots l_n)$. 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 "tensor 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 them into leaves (and into hash-maps).

For example, Appendix A.3 of [2] shows how to accommodate samples marked as being positive or negative (Section 1.1 and Section 2 of the present paper) in this framework. One considers $\mathbb{V} = \mathbb{R} \oplus M$, where M is the space of 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 hash-maps, and we are still going to call those hash-maps *V-values*.

The role of V-values in the context of DMMs is somewhat similar to the role of S-expressions in the context of Lisp.

1.5 Structure of this Paper

In Section 2 we discuss linear streams in more details. In Section 3 we define DMMs based on variadic neurons.

Programming patterns in DMMs are presented in Section 4. A very expressive self-referential mechanism which is the key element of this approach is presented in Section 5.

The issues related to expressing the network topology are the subject of Section 6 and the issues related to subnetworks and modularization are the subject of Section 7.

We discuss some of the potential approaches to using DMMs in machine learning in Section 8. The concluding Section 9 contains historical remarks and discussion of related work.

2 Linear Streams: Further Discussion

We consider linear streams in somewhat limited generality here. First of all, we consider discrete sequential time, while other models of time, such as continuous real line, are also potentially of interest. 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.

Specifically, for every kind of linear stream k we introduce background vector space V_k, and we consider streams of approximate representations providing some information about the underlying vectors from V_k. For every kind of linear streams k, we define a procedure computing an approximate representation of vector $\alpha_1 v_{1,k} + \ldots + \alpha_n v_{n,k}$ from approximate representations of vectors

$v_{1,k}, \ldots, v_{n,k}$. We say informally that the representations in question belong to a vector-like space.

Streams of signed samples are an important non-trivial example of this approach. They are mentioned in Section 1.2 of [3], and a more detailed treatment of them is given in Appendix A.1 of [2]. The underlying vector space is the space of all finite signed measures over 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 the sample $\langle x_i, \text{sign}(\alpha_i) * s_i \rangle$ (reversing the flag if the selected value α_i is negative) to represent the linear combination $\alpha_1 * \mu_1 + \ldots + \alpha_n * \mu_n$.

Issues related to missing samples and zero measures are mentioned in Appendix A.2 of [2]. A generalization to complex-valued measures and linear combination with complex coefficients is considered in the design notes for [10][1].

3 Variadic Neurons

We use the space of V-values we have defined earlier to construct DMMs based on variadic neurons, following Section 1.4.4. We give the construction for the space V, but similar constructions also work for variants $V \otimes \mathbb{V}$ described in Section 1.4.5.

Because we are going to use the keys at the first level of V-values as names of inputs and outputs (as explained in Section 1.4.4), we don't want any top-level leaves, that it is we don't want non-zero scalars ("tensors of rank 0") in our V-values. Hence we'll 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

$$U \cong L \to (\mathbb{R} \oplus U). \tag{4}$$

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.

The network matrix $\mathbf{W}$ 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. Below we follow closely the material from Section 3.2 of [2].

[1] `https://github.com/jsa-aerial/DMM/blob/master/design-notes/Early-2017/sampling-formalism.md`

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 $\mathbf{W}$ 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 indexes: 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 $U \to U$), 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 indexes: 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 (Figure 3).

We allow names for neurons and their inputs and outputs to be any elements of L.

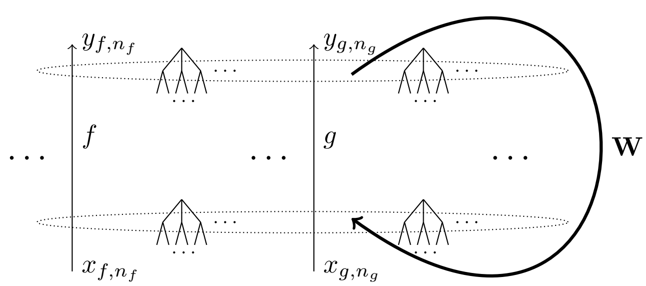

Figure 3: "Two-stroke engine" for a DMM based on variadic neurons. Two neurons, n_f and n_g, are explicitly pictured. Their inputs and outputs, $x_{f,n_f}, x_{g,n_g}, y_{f,n_f}, y_{g,n_g}$, are depicted as trees belonging to U. $\mathbf{W}$ is a linear map from the concatenation of the first levels of all y_{h,n_h} trees to the concatenation of the first levels of all x_{h,n_h} trees.

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 hence 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:

$$x^{t+1}_{f,n_f,i} = \sum_{g \in F} \sum_{n_g \in L} \sum_{o \in L} w^t_{f,n_f,i,g,n_g,o} * y^t_{g,n_g,o}. \tag{5}$$

Indices f and g belongs to the set of neurons types F, which is simply the set of transformations of U. Potentially, one can have countable number of such transformations implemented 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, 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.

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 for all the neurons n_f which are present in this map:

$$y^{t+1}_{f,n_f} = f(x^{t+1}_{f,n_f}). \tag{6}$$

For an example of an activation function of a variadic neuron see Section 4.1.

4 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 5) and from modularization facilities (Section 7).

4.1 Linear and Multiplicative Constructions

Linear and multiplicative constructions in dataflow matrix machines are well-covered in [7]. The most fundamental of them is a neuron with *identity activation function.* If we connect the output of such a neuron to its input with weight 1, this neuron becomes an accumulator, which adds together and stores the contributions to its input from all other outputs in the network connected with the input of this neuron by non-zero weights.

Among multiplicative constructions, the most fundamental one is multiplication of an otherwise computed neuron output by the value of its scalar input. This is essentially a fuzzy conditional, which can selectively turn parts of the network on and off in real time, attenuate, amplify, and reverse the signal, redirect flow of signals in the network, etc. For further details see [7].

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 have non-linear activation functions, or on only have 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 [8].

The code for the activation function `accum` of this accumulator looks as follows [10]:

```
(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, `(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.

4.2 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 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 [7]. The savings can be drastic, as, for example, Unicode alphabet exceeds 100,000 characters, and with sparse representation only 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.

4.3 Data Structures

An earlier paper [7] focuses on allocating data structures in the body of the network itself, an approach encouraged by the network 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 standard Clojure style of working with immutable data structures and automatically sharing their common parts should make passing complicated structures through "up movements" and "down movements" reasonably efficient.

For example, a list can be encoded by using `:this` and `:next` 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.

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).

5 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 3). 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 [8] (see Section 4.1). `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 [6], 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 network matrix updates, 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 [7] 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 [8].

Also, it was demonstrated in [8] 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 [2] 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.

6 Network Topology

The network topology, such as layers, is defined by the pattern of sparsity of $\mathbf{W}$: some of the connectivity weights are kepts 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 4.1 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.

7 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 [5, 6, 7, 8], 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 the single "up movement" action of any subnetwork as the single "up movement" action of an appropriately crafted neuron.

8 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 [8]).

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 various "learning to learn" scenarios.

DMMs architecture is conductive to experiments with "fast weights" (e.g. [1]).

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 [24, 23]. The ability of single DMM neurons to express 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. [22]). 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 solving 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.

9 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 transformation 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.

9.1 Recurrent Neural Networks as Programs

It was recognized as early as 1940-s, that if one provides a neural network with a suitable model of unbounded memory one obtains a Turing-universal formalism of computations [20]. Research studies formally establishing Turing-universality for neural networks processing streams of reals include [25, 27].

More recent studies include such well-known approaches as [13, 28], 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 progress towards human-readable programs. In fact, it seems that the expressive power of scalar-based neural networks is sufficient to create Turing-complete esoteric programming languages, but is not sufficient to create a pragmatic programming platform. The restriction to scalar flows seems to either necessitate awkward encoding of complex data within reals, or to force people to create networks depending in their size on data dimensionality and with any modularization capabilities being external to the network formalism itself, rather than being native to the networks.

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 [26] has to use such encodings, 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 [25]. Yet, when the LSTMs were invented in 1997 [17], 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 yield the natural way to express memory and gating mechanisms in neural nets (see Appendix C of [8]).

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 [16] is now a well-cited paper). However, the explicit activation functions of arity two, such as multiplication, are still quite exotic and difficult to 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.

9.2 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 [15] and Pure Data (e.g. [11]). The programs themselves are still quite discrete.

To incorporate higher-order programming methods, 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 there are enough overdefined (i.e. partially inconsistent) elements in those spaces to produce zero on addition by the mechanism of cancellation with underdefined (i.e. partially defined) elements. The mathematics of the resulting *partial inconsistency landscape* is presented in Section 4 of [3].

By 2015 it become apparent to our group that programming with linear streams is a rich formalism which includes programming with streams of probabilistic samples and programming with generalized animations, that this framework seems to provide methods for continuous higher-order programming, and that, moreover, it has good potential for obtaining more efficient schemas for generic programming by allowing to introduce the motives similar to regulation of gene expressions into genetic programming frameworks [3]. It also became apparent at that time that if one introduces the discipline of interleaving linear and non-linear transformation of linear streams, then one can parametrize large classes of programs by matrices of numbers [4, 3].

The first open-source software prototypes associated with this approach also appeared in 2015 [12].

In 2016 we understood that the resulting formalism generalized recurrent neural networks, and the term *dataflow matrix machines* was coined [5]. 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 [6]. The programming patterns for resulting software framework were studied in [7]. A more theoretical paper [8] explored the possibility of using self-referential matrix transformations instead of lambda-calculus in this context and established further connections between neural networks and 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 [14, 9]. The first open-source implementation of a version of DMMs based on that formalism and written in Clojure was produced in that time frame [10], and a research paper based on this architecture was presented recently at LearnAut 2017 [2].

Acknowledgments

We would like to thank Dima-David Datjko, Elena Machkasova, and Elena Nekludova for helpful discussions of the material presented in [2] and here.

References

[1] J. Ba, G. Hinton, V. Mnih, J. Leibo, C. Ionescu. *Using Fast Weights to Attend to the Recent Past.* `https://arxiv.org/abs/1610.06258`

[2] M. Bukatin, J. Anthony. *Dataflow Matrix Machines as a Model of Computations with Linear Streams.* In LearnAut 2017 ("Learning and Automata" Workshop at LICS 2017). `https://arxiv.org/abs/1706.00648`

[3] M. Bukatin, S. Matthews. *Linear Models of Computation and Program Learning.* In G. Gottlob et al., editors, GCAI 2015, EasyChair Proceedings in Computing, vol. 36, pages 66–78, 2015. `https://easychair.org/publications/paper/245311`

[4] M. Bukatin, S. Matthews. *Dataflow Graphs as Matrices and Programming with Higher-order Matrix Elements.* Preprint, August 2015. `https://arxiv.org/abs/1601.01050`

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

[6] M. Bukatin, S. Matthews, A. Radul. *Dataflow Matrix Machines as Programmable, Dynamically Expandable, Self-referential Generalized Recurrent Neural Networks*, May 2016. `https://arxiv.org/abs/1605.05296`

[7] M. Bukatin, S. Matthews, A. Radul. *Programming Patterns in Dataflow Matrix Machines and Generalized Recurrent Neural Nets*, June 2016. `https://arxiv.org/abs/1606.09470`

[8] M. Bukatin, S. Matthews, A. Radul. *Notes on Pure Dataflow Matrix Machines: Programming with Self-referential Matrix Transformations*, October 2016. `https://arxiv.org/abs/1610.00831`

[9] Clojure programming language. `https://clojure.org`

[10] DMM project. GitHub repository: `https://github.com/jsa-aerial/DMM`

[11] A. Farnell. *Designing Sound*. MIT Press, 2010.

[12] Fluid: Project "Fluid" GitHub repository, 2015-2017.
`https://github.com/anhinga/fluid`

[13] A. Graves, G. Wayne, I. Danihelka. *Neural Turing Machines*, October 2014.
`https://arxiv.org/abs/1410.5401`

[14] R. Hickey. *The Clojure Programming Language*. In Proceedings of the 2008 Symposium on Dynamic Languages, Association for Computing Machinery.

[15] W. Johnston, J. Hanna, R. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, 2004.

[16] K. He, X. Zhang, S. Ren, J. Sun.*Identity Mappings in Deep Residual Networks*, March 2016. `https://arxiv.org/abs/1603.05027`

[17] S. Hochreiter, J. Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8): 1735–1780, 1997.

[18] A. Karpathy. *The unreasonable effectiveness of recurrent neural networks*, May 2015. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

[19] A. Kornai. *Mathematical Linguistics*. Springer-Verlag, 2008.

[20] W. McCulloch, W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[21] T. Mikolov, I.'Sutskever, K. Chen, G. Corrado, J. Dean. *Distributed Representations of Words and Phrases and their Compositionality*, October 2013.
`https://arxiv.org/abs/1310.4546`

[22] V. Murali, S. Chaudhuri, C. Jermaine. *Bayesian Sketch Learning for Program Synthesis*, March 2017. `https://arxiv.org/abs/1703.05698`

[23] A. Nejati. *Differentiable Programming*, August 2016.
`https://pseudoprofound.wordpress.com/2016/08/03/differentiable-programming`

[24] C. Olah. *Neural Networks, Types, and Functional Programming*, September 2015. `http://colah.github.io/posts/2015-09-NN-Types-FP`

[25] J. Pollack. *On Connectionist Models of Natural Language Processing.* PhD thesis, University of Illinois at Urbana-Champaign, 1987. Chapter 4 is available at `http://www.demo.cs.brandeis.edu/papers/neuring.pdf`

[26] J. Schmidhuber. *A "Self-Referential" Weight Matrix*, In: S. Gielen, B. Kappen, eds., ICANN 93: Proceedings of the International Conference on Artificial Neural Networks, Springer, 1993, pp. 446–450.

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

[28] J. Weston, S. Chopra, A. Bordes. *Memory Networks*, October 2014. https://arxiv.org/abs/1410.3916