Self-referential mechanism for dataflow matrix machines and generalized recurrent neural networks

Michael Bukatin

HERE North America LLC, Burlington, MA

Joint work with Steve Matthews and Andrey Radul

- - -

New England Programming Languages and Systems Symposium,
Northeastern University, October 7, 2016

Electronic coordinates

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

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

E-mail:

`bukatin@cs.brandeis.edu`

Motivation

Trying to take steps towards two goals.

- Recurrent neural networks are Turing-universal. Can we increase their expressive power in such a way as to produce a convenient general-purpose programming system?
- New methods of machine learning, including new methods for training neural networks and for program synthesis.

Towards these goals, we introduce dataflow matrix machines (DMMs), which is a new formalism generalizing recurrent neural networks.

Outline of the talk

DMMs generalize RNNs

- **Recurrent neural networks** — RNNs.
- **Dataflow matrix machines** — DMMs.
- DMMs — a natural generalization of RNNs.

- RNNs process streams of numbers.
- DMMs process streams of representations of arbitrary vectors.

  (Examples: stream of vectors from linear space V,
  stream of samples of probability distributions over X,
  stream of samples of signed measures over X.)

The next 5 slides describe DMMs.

DMMs: kinds of streams and types of neurons

Arbitrary **linear streams:** streams admitting the notion of linear combination of several streams.

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

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

Each neuron type is defined by:

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

Classical neural nets (RNNs)

Only one kind of linear streams - stream of numbers.

Main neurons - input and output arity is 1.

One or a very few types of built-in transformations.

Input neurons - input arity zero, output arity 1, emit input streams.

Output neurons - input arity 1, output arity 0, with side-effect (recording the results).

Occasionally, people allow input arity 2 (resulting in much more expressive nets; we'll talk about it later).

DMMs: fixed size nets vs variable size nets

Weights and topology can change while the network is running.

Simple case: Fixed size nets - finite number of neurons with specified fixed types.

Determined by finite rectangular matrices of fixed size.

Main case: Variable size nets - countable number of neurons of each possible type (hence countable number of overall neurons).

Only a finite number of neurons is active at any given moment.

Determined by **countable matrices with finite number of non-zero elements** at any given moment.

DMMs, main case: network matrix and active neurons

Countable collection of neurons of each type.

Hence countable number of inputs x_i and outputs y_j.

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

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

DMMs and RNNs: "Two-stroke engine"

One should think about a DMM/RNN as a **"two-stroke engine"**.

One element is added to each output stream y_j on the "up movement"; one element is added to each input stream x_i on the "down movement".

Writing for the case of DMMs:

"Up movement": for every active neuron C,
$y_{1,C}, \ldots, y_{n,C} := f_C(x_{1,C}, \ldots, x_{m,C})$.

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

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

The next 4 slides describe
the self-referential mechanism in the DMMs.

DMMs: linear neurons as accumulators/memory

Linear neurons are neurons with linear transformation of input streams as their built-in transformation.

Simplest case: **accumulators with arity 1**.

Consider a neuron with a single input taking a particular kind of linear streams and with the built-in **identity transformation of streams**, so it just passes its input through.

Connect its output with weight 1 to its input and keep this connection fixed. This neuron is functioning as an **accumulator of additive contributions** of other neurons connected to its input with non-zero weights.

DMMs: linear neurons as accumulators/memory

Accumulators with arity 2.

$$y = x_{\text{main}} + x_{\text{delta}}$$

Connect y to x_{main} with weight 1, and keep other connections to x_{main} at zero.

Take additive contribution from other neurons at the x_{delta} input.

DMMs: self-referential mechanism

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

(For the fixed size case, use streams of finite rectangular matrices.)

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

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

In our version of self-refential mechanism, `Self` is an accumulator.

Self-referential mechanism as the foundation for DMMs

Self-referential matrix transformations seem to fit the DMM architecture better, than string rewriting.

This self-referential mechanism might be a reasonable replacement of untyped lambda-calculus for dataflow matrix machines.

(If we limit ourselves to one kind of linear streams, namely streams of matrices (w_{ji}), this might be a "moral equivalent" of programming within pure untyped lambda-calculus.)

Our joint preprints on DMMs

`https://arxiv.org/abs/1603.09002`

`https://arxiv.org/abs/1605.05296`

`https://arxiv.org/abs/1606.09470`

`https://arxiv.org/abs/1610.00831`

Part 2

Motivation: reminder

Trying to take steps towards two goals.

- **Recurrent neural networks are Turing-universal. Can we increase their expressive power in such a way as to produce a convenient general-purpose programming system?**
- New methods of machine learning, including new methods for training neural networks and for program synthesis.

Turing universality vs general-purpose programming

There are plenty of elegant and/or useful systems for which there are claims of Turing-universality, but which are not suitable as a general-purpose programming platform. See:

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

and

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

Conway's Game of Life, Rule 110, LaTeX, C++ templates, . . .

This list includes Recurrent Neural Networks.

Turing universality requires unbounded memory

We are aware of three ways to equip RNNs with unbounded memory.

- Make RNN a controller for external memory, e.g. tape of a Turing machine [e.g. Pollack 1987];
- Allow to use reals of unlimited precision, effectively using a binary expansion of a real number as a tape of a Turing machine [Siegelmann, Sontag, early 1990-s];
- Consider a countable neural network, all of which except for a finite part is silent at any given moment of time [our recent preprints, 2015-2016].

"The unreasonable effectiveness of recurrent neural networks"

Modern neural networks, such as LSTM and Gates Recurrent Unit networks are really powerful as a machine learning platform.

Unlike old-style RNNs, they can learn long-range dependencies.

A nice set of examples is in this blog post by **Andrej Karpathy**:

`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

(See in particular his examples of the networks learning patterns of programs in C and patterns of mathematical LaTeX texts.)

Problem of vanishing gradients

The training only started to work well after people figured out how to fight the problem of vanishing gradients.

LSTM (original flavor): 1997.

A lot of options now, including a variety of LSTM flavors and other schemas. For a nice compact overview see this paper from Nanjing University: Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, Zhi-Hua Zhou, **Minimal Gated Unit for Recurrent Neural Networks**. `http://arxiv.org/abs/1603.09420`

I want to discuss the nature of these architectures, and for this I need to first talk about bilinear neurons.

Bilinear neurons

For example $y = x_1 * x_2$ (bilinear transformation of streams).

Used in [Pollack 1987, `http://www.demo.cs.brandeis.edu/papers/neuring.pdf`].

Greatly increase the expressive power (in terms of convenience, not strictly required for Turing universality). Require input arity 2.

For example, if one selects one of the inputs of $x_1 * x_2$ as "carrying the main signal", and another input as "doing the modulation of the main signal", then one can use the modulation to conditionally turn parts of the network on and off, redirect the flow of the signal in the network, attenuate the signal, etc.

Sometimes people use the terminology of **multiplicative masks** or **gates** in this context.

Linear and bilinear neurons for LSTM and GRU networks

People tend to de-emphasize the role of linear and bilinear neurons.

For example, modern LSTM and gated recurrent unit networks are usually described as networks of sigmoid neurons augmented with external memory and gating mechanisms.

A more straightforward description of them: conventional RNNs built from a mixture of sigmoid, linear, and bilinear neurons.

See Section 2 and Table 1 of the Nanjing paper.

In all these flavors of RNNs (LSTM, GRU, etc) some weights are variable and subject to training and some are fixed as zeros or ones to establish a particular network topology.

How many weights are needed in RNN of bilinear neurons

I'd like to briefly mention the assertion from [Pollack 1987] that in order to use an RNN consisting of N bilinear neurons, N^3 weights are required, and that therefore the training of these networks should be very difficult.

If one simply multiplies linear combinations accumulated on both inputs, then the actual number seems to be much more manageable $2N^2$ variable weights, just like for Minimal Gated Unit networks (and similar to $3N^2$ for Gated Recurrent Unit networks, and $4N^2$ for LSTM).

So the networks based on bilinear neurons might actually train well, just like LSTM/GRU/MGU, but experimental work needs to be performed to see, if it is actually the case.

Part 3

DMMs as a general-purpose programming platform

DMMs are much more powerful and expressive than RNNs, but share the key property: the programs are simply matrices of numbers, so one can deform them in a continuous fashion, etc.

The question is: **if we replace RNNs (an esoteric programming language) with DMMs**, how far would this move us towards creating a general-purpose programming platform?

This is very much work in progress. Let's say a few words about its current state.

DMMs as a general-purpose programming platform

DMMs are much more powerful than RNNs:

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

The next few slides

The next few slides illustrate the roles of various DMM capabilities in using DMMs as a general-purpose programming platform:

- Sparse arrays and string processing
- Multiple input arity and multiplicative masks
- Explicit linear and bilinear neurons for memory and conditionals
- Stochastic mechanisms and memory within neurons
- Various uses of self-referential mechanism

DMMs as a general-purpose programming platform

Representing characters as vectors: “1-in-N” representation is standard in RNNs.

Take the alphabet as the basis, represent characters as vectors with 1 at the corresponding coordinate, and zeros at all others.

Sparse arrays are extremely important here, especially for large alphabets like Unicode (100,000+ characters).

E.g. in DMMs one neuron can accumulate a vector representing the dictionary of counts of character occurrences in a string.

Independence of the size of the network from the size of the alphabet (impossible in RNNs).

DMMs as a general-purpose programming platform

Multiple inputs allow us to program via **multiplicative masks**, which can dynamically turn on and off, attenuate and amplify parts of the network.

By turning parts of the network on and off one can implement conditionals, redirect flows of data within the network, and precisely orchestrate coordination.

This aspect revives the insights from [Pollack 1987].

DMMs as a general-purpose programming platform

The lack of linear neurons (identity transformation, sum) in usual RNNs is incredibly inconvenient, because one needs them for accumulators, leaky accumulators, and more.

The linear and bilinear neurons are implicitly present in LSTM and Gates Recurrent Unit networks, but one needs to consider them explicitly to be able to fluently program with them.

DMMs as a general-purpose programming platform

Let's look again at the "two-stroke engine".

One element is added to each output stream y_j on the "up movement"; one element is added to each input stream x_i on the "down movement". But it's OK for the neurons to have memory if desired, and it's OK to use the "statistical sum" linear combination on the "down movement" to implement streams of samples.

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

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

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

More uses of self-referential mechanism

In the next few slides we use the self-referential mechanism in several different ways

- Programming language experiments to interactively update the network while it is running by triggering update neurons and using self-referential mechanism
- Deep copy of subnetworks while the DMM is running
- Create and traverse linked structures in the body of network
- Create oscillations and waves of network connectivity patterns by using simple constant matrices as update matrices for self-referential mechanism

More uses of self-referential mechanism

We did some experimental work on sketching a programming language to interactively update a DMM while its running.

We try to follow the following informal principle:

Principle of **self-referential completeness** of the DMM signature relative to the language available to describe and edit the DMMs.

(DMM signature = available kinds of linear streams and types of neurons.)

The principle states that for all updates one can do in the language, one should be able to accomplish those updates by triggering appropriate neurons producing additive changes to `Self`.

More uses of self-referential mechanism

It is possible to allocate deep copies of subnetworks/subgraphs in the empty “silent space” while the network is running.

One uses column-shaped and row-shaped vectors as “multiplicative row masks” and “multiplicative column masks” respectively to indicate a subgraph.

One can iterate and nest this construction to create intricate “pseudo-fractal patterns” within the network.

See `https://arxiv.org/abs/1606.09470` for more details.

More uses of self-referential mechanism

Using the ability to change the network matrix while it is running, it should be possible to create and traverse linked structures within the network.

This capability is something yet to be explored in more details in the future.

More uses of self-referential mechanism

The update mechanism is really powerful. Just using the update mechanism and a few constant matrices, it is possible to create oscillations and waves of network connectivity patterns.

See Appendix D.2 of `https://arxiv.org/abs/1610.00831` for more details (**and I'll show a demo of that material at the end of the talk, if there is time**).

It is easy to embed an arbitrary deterministic finite state machine into this control mechanism simply by making the update matrices dependent on e.g. input symbols, and it is easy to include "payload" into the update matrices.

Part 4

DMMs: history remarks

The defining feature of DMMs is using arbitrary linear streams instead of streams of numbers used in RNNs.

One might expect that the idea as basic as this one would be known for decades at this point. Strangely enough, the feedback we are getting so far indicates that this is new.

Our own path to this generalization of recurrent neural networks was rather indirect.

DMMs: history remarks

First, we noticed that in the presence of partial inconsistency domains for denotational semantics tend to become embedded into vector spaces, and this encouraged us to focus on programming with linear streams:

Michael Bukatin and Steve Matthews. *Linear Models of Computation and Program Learning.* GCAI 2015, EasyChair Proceedings in Computing, 36, 66-78.

`http://easychair.org/publications/download/Linear_Models_of_Computation_and_Program_Learning`

DMMs: history remarks

Then while researching dataflow programming with linear streams, we noticed that if one does not allow to connect non-linear transformations of linear streams directly to each other, but only allows to connect them via linear transformations, then one gets a parametrization of large classes of programs by matrices:

Michael Bukatin and Steve Matthews. *Dataflow Graphs as Matrices and Programming with Higher-order Matrix Elements.* `https://arxiv.org/abs/1601.01050`

DMMs: history remarks

Only then we realized that this formalism is a generalization of recurrent neural networks, and that this is a productive way to look at this subject matter, resulting in the current series of preprints:

Michael Bukatin, Steve Matthews, Andrey Radul. *Dataflow Matrix Machines as a Generalization of Recurrent Neural Networks.* `https://arxiv.org/abs/1603.09002`

Michael Bukatin, Steve Matthews, Andrey Radul. *Dataflow Matrix Machines as Programmable, Dynamically Expandable, Self-referential Generalized Recurrent Neural Networks.* `https://arxiv.org/abs/1605.05296`

DMMs: history remarks

Michael Bukatin, Steve Matthews, Andrey Radul.
Programming Patterns in Dataflow Matrix Machines and Generalized Recurrent Neural Nets.
`https://arxiv.org/abs/1606.09470`

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

Initial open-source prototypes

Project Fluid: our initial experiments in **programming with linear streams**:

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

Done in Processing 2.2.1. (This is a simplified, more readable version of Java for the artistic community. It is architected around repeatedly calling top-level function `draw` which renders the next frame of an animation. This resonates well with the "two-stroke engine" cyclic architecture of RNNs and DMMs.)

It might be productive to switch to a more flexible modern language such as Clojure or Racket.

Demo

If there is time at the end of the talk:

Waves of network connectivity patterns obtained simply by using the self-referential update mechanism with a few constant update matrices. Appendix D.2.2 of `https://arxiv.org/abs/1610.00831`

Each update matrix in this example essentially says: "subtract me from the network and add the next update matrix instead".

A demo from `https://github.com/anhinga/fluid`

`Lightweight_Pure_DMMs/aug_27_16_experiment` subdirectory.

Conclusion

Revisiting our first goal:

- Recurrent neural networks are Turing-universal. Can we increase their expressive power in such a way as to produce a convenient general-purpose programming system?

It is possible to program solely in terms of linear streams. One of the questions we are trying to answer is: how much of our overall programming practice is convenient to express in this fashion?

It is clear at this point that DMMs allow to progress quite a bit from matrix-based programming technology as an esoteric programming language (RNNs), to a matrix-based programming technology which is much closer to the usual software engineering needs. Can DMMs go all the way in this direction?

Electronic coordinates

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

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

E-mail:

`bukatin@cs.brandeis.edu`

Looking for collaboration on research and engineering aspects of dataflow matrix machines.