Exploring synthesis of flexible neural machines with Zygote.jl

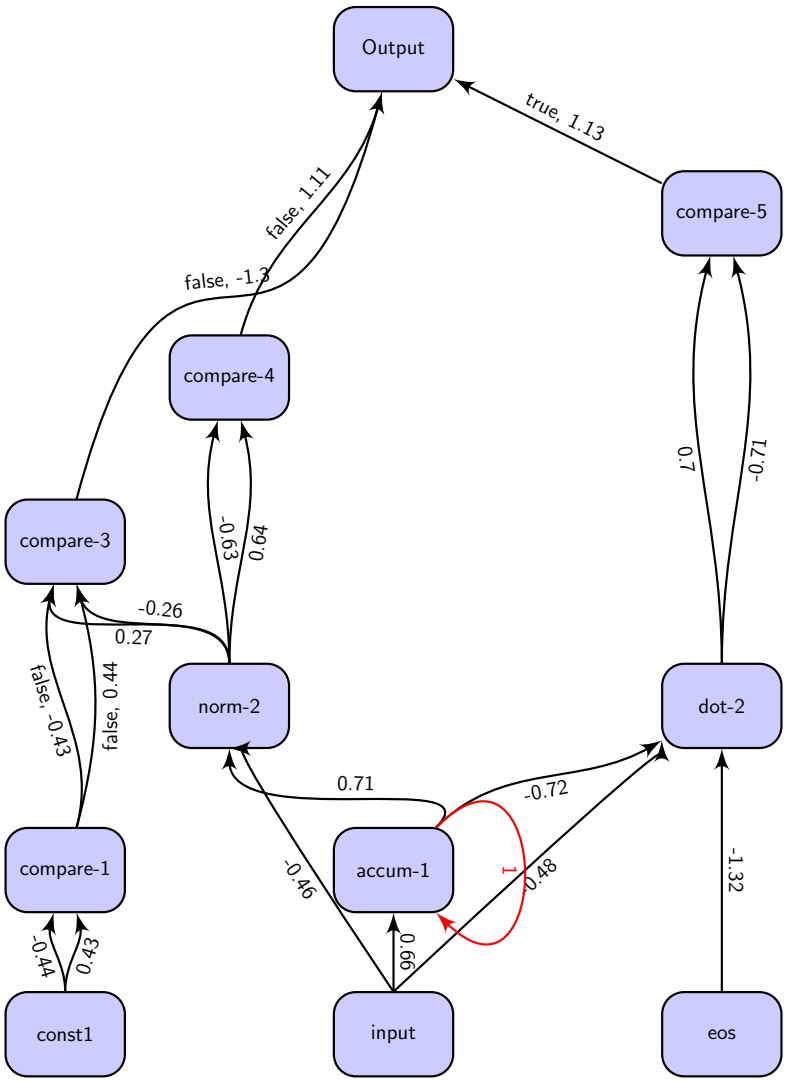
Mishka (Michael Bukatin)

`github:anhinga`

Dataflow Matrix Machines project

JuliaCon 2023
Cambridge, MA, July 28, 2023

Vectors represented by dictionaries flow through the wires



This talk:

- Flexible neural machines (flexible attention machines) discovered and studied by our group a few years ago. We call them dataflow matrix machines (DMMs).

- My recent experiments with Zygote.jl

I am looking for collaborators

Transitions:

- Traditional neural nets ⇒ Transformers (modern AI magic)
  - Much more expressive than traditional neural nets
  - Easier to train (better optimized for training)

- Transformers ⇒ Flexible neural machines
  - Much more expressive than even Transformers
    - General-purpose continuously deformable dataflow programs
    - Convenient and flexible self-modification facilities
    - ...
  - But not optimized for training (too flexible)
    - Zygote.jl is also very flexible and can help today
    - Optimize those neural machines better in the future?

The essence of neural model of computations is that
linear and non-linear computations are interleaved.

What is the most general (and most flexible) way
to make this linear/non-linear alternating pattern possible?

. . . . . .

The natural degree of generality for neural-like computations is:

~~use only streams of numbers~~
use any streams supporting the notion of
linear combination of several streams ("linear streams")

Dataflow matrix machines (DMMs)

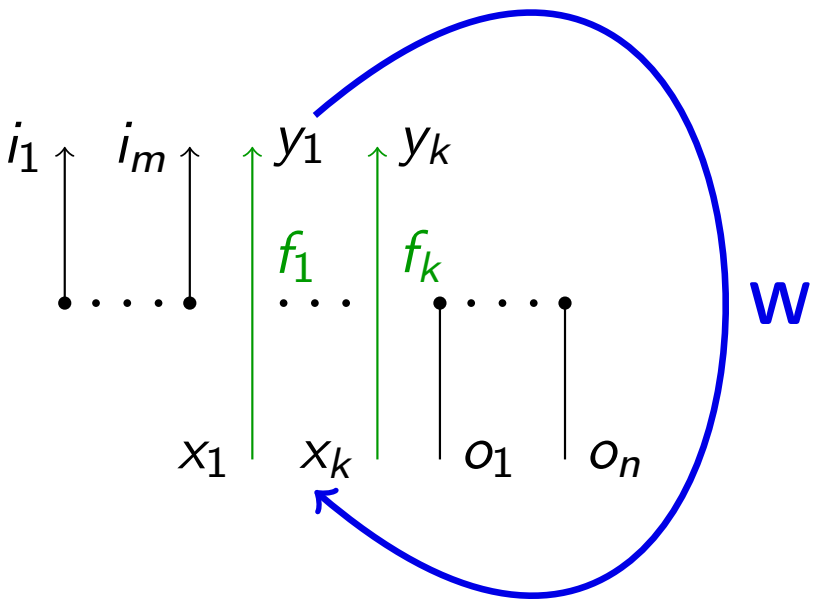
DMMs: a very natural class of **neural machines**.

They use arbitrary **linear streams** instead of streams of numbers.

Use of linear streams by single neurons is the main source of their power compared to RNNs. The extra power also comes from:

- Arbitrary fixed or variable arity of neurons;
- Highly expressive linear streams of V-values (tree-shaped flexible tensors);
- Unbounded network size ($\Rightarrow$ unbounded memory);
- Self-referential facilities: ability to change weights, topology, and the size of the active part dynamically, on the fly.

RNNs: linear and non-linear computations are interleaved



"Two-stroke engine" for an RNN:

"Down movement" (linear[1]):

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

"Up movement" (non-linear[2]):

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

[1] linear and global in terms of connectivity

[2] usually non-linear, local

Linear streams

The key feature of **DMMs** compared to **RNNs**: they use **linear streams** instead of streams of numbers.

The following streams all support the pattern of alternating linear and non-linear computations:

- Streams of numbers
- Streams of vectors from fixed vector space V
- Linear streams: such streams that the notion of **linear combination of several streams** is defined.

"Artificial attention":
taking linear combinations of high-dimensional objects.

Kinds of linear streams

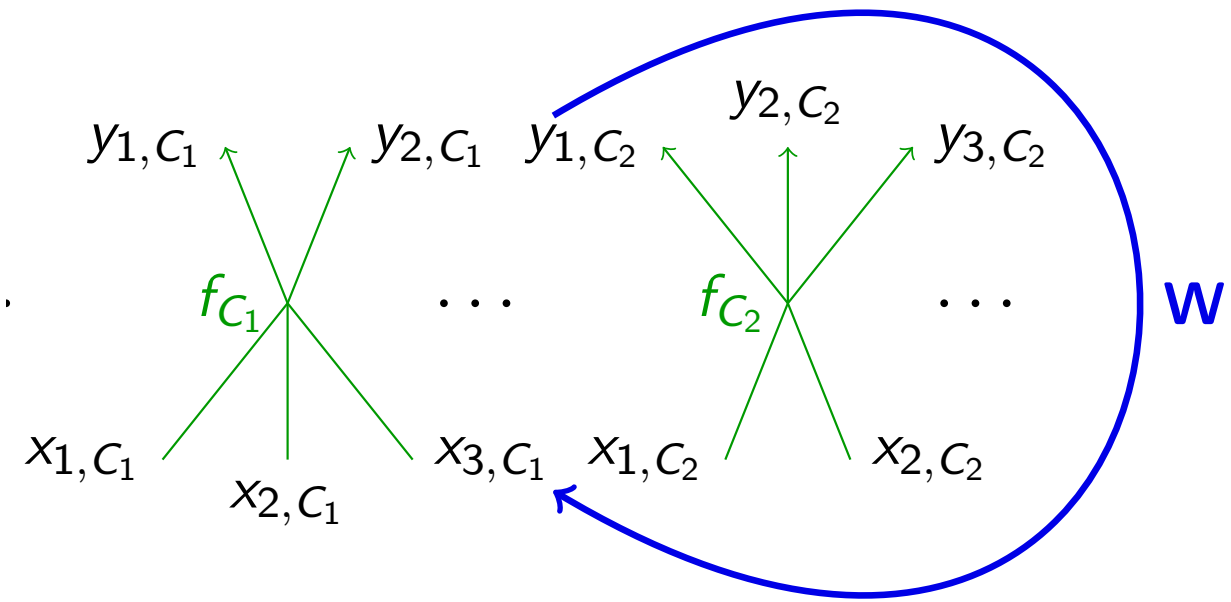
The notion of **linear streams** is more general than the notion of **streams of vectors**. Some examples:

- Every vector space V gives rise to the corresponding kind of linear streams (streams of vectors from that space)
- Every measurable space X gives rise to the space of **streams of probabilistic samples** drawn from X and decorated with +/- signs (linear combination is defined by a stochastic procedure)
- Streams of images of a particular size (that is, animations)
- Streams of matrices; streams of multidimensional arrays
- Streams of V-values based on nested maps (trees)

Dataflow matrix machines (DMMs, continued)

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

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



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

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

"Up movement":
For all active neurons C:

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

Here x_{i,C_k} and y_{j,C_m} are linear streams.
Neurons in DMMs have arbitrary arity!

Type correctness condition for mixing different kinds of linear streams in one DMM

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

Next: linear streams of **V-values** based on **nested dictionaries**:

- sufficiently universal and expressive to save us from the need to impose type correctness conditions.
- allow us to define **variadic neurons**, so that we don't need to keep track of input and output arity either.

V-values: vector space based on nested dictionaries

V-values play the role of Lisp S-expressions in this formalism.

We want a vector space.

Take prefix trees with numerical leaves
implemented as nested dictionaries.

We call them **V-values** ("vector-like values").

(A more general construction of V-values with "linear stream" leaves: Section 5.3 of `https://arxiv.org/abs/1712.07447`)

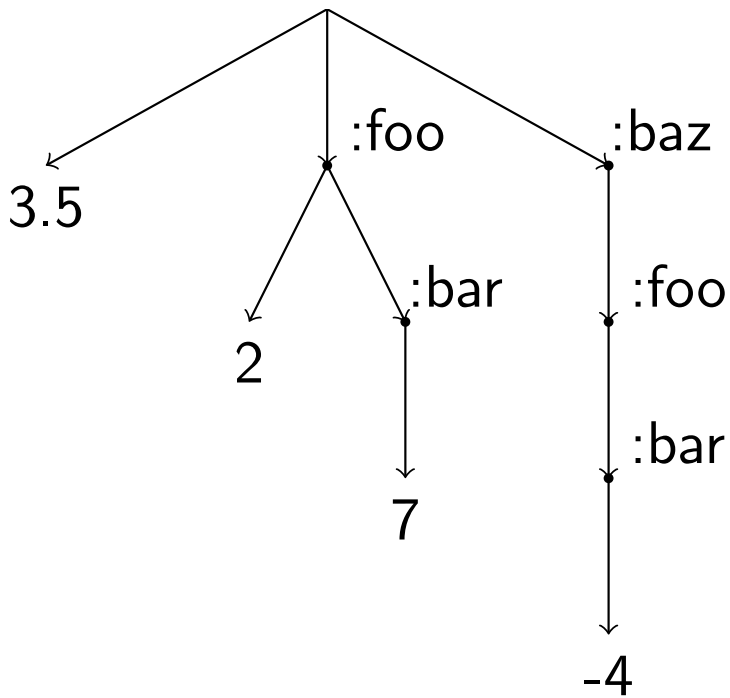
Ways to understand V-values

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

V-values can be understood as

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

Example of a V-value: different ways to view it

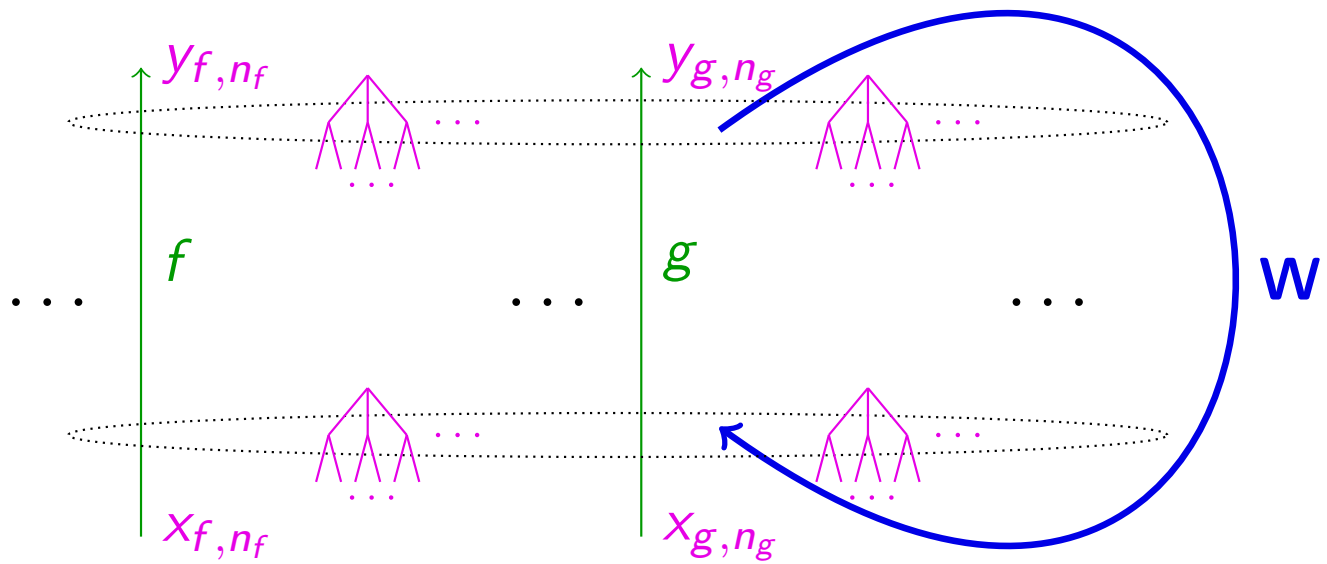


- $3.5 \cdot (\epsilon) + 2 \cdot (\text{:foo}) +$
  $7 \cdot (\text{:foo :bar}) - 4 \cdot (\text{:baz :foo :bar})$
- $(\rightsquigarrow 3.5) + (\text{:foo} \rightsquigarrow 2) +$
  $(\text{:foo} \rightsquigarrow \text{:bar} \rightsquigarrow 7) +$
  $(\text{:baz} \rightsquigarrow \text{:foo} \rightsquigarrow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

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

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

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



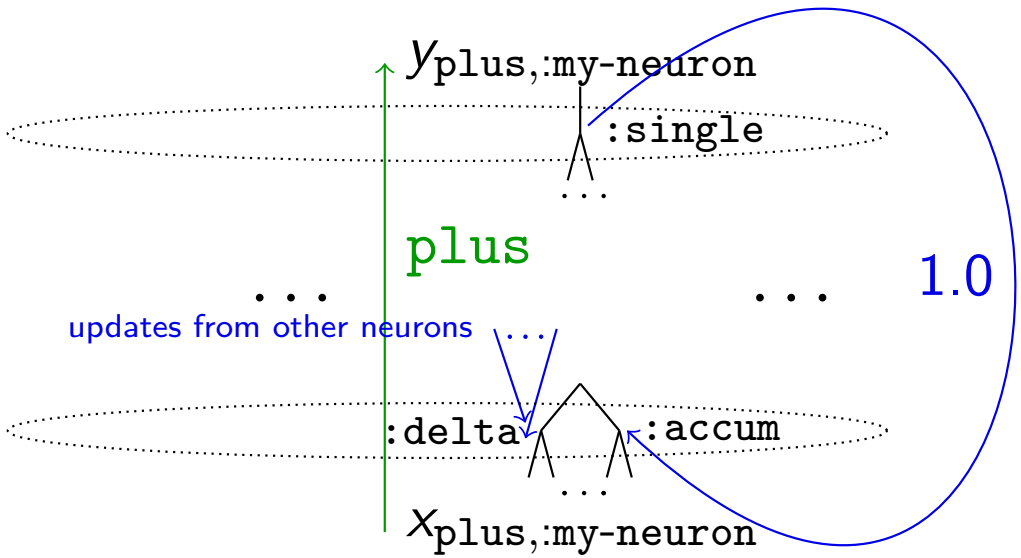
DMMs: programming with powerful neurons

Powerful variadic neurons and streams of V-values ⇒ a much more expressive formalism than networks based on streams of numbers.

⇒ Many tasks can be accomplished by compact DMM networks, where **single neurons function as layers or modules**.

- Accumulators and memory
- Multiplicative constructions and "fuzzy if"
- Sparse vectors
- Data structures
- Self-referential facilities

Accumulators and memory



In this implementation, activation function `plus` adds V-values from `:accum` and `:delta` together and places the result into `:single`.

Multiplicative masks and fuzzy conditionals (gating)

Many multiplicative constructions enabled by **input arity** > 1.

The most notable 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
- ...

Sparse vectors of high or infinite dimension

Example: a neuron accumulating count of words in a given text.

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

- Don't need a neuron for each coordinate of our vector space.
- Don't have an obligation to reduce dimension by embedding.

Streams of immutable data structures

One can represent **lists, matrices, graphs,** and so on via nested dictionaries.

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

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

But more imperative styles can be accommodated as well.

Self-referential facilities

Difficult to do well with streams of scalars because of dimension mismatch: typically one has $\sim N$ neurons and $\sim N^2$ weights.

Easy in DMMs:

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

Emit the stream of network matrices from neuron `Self`.

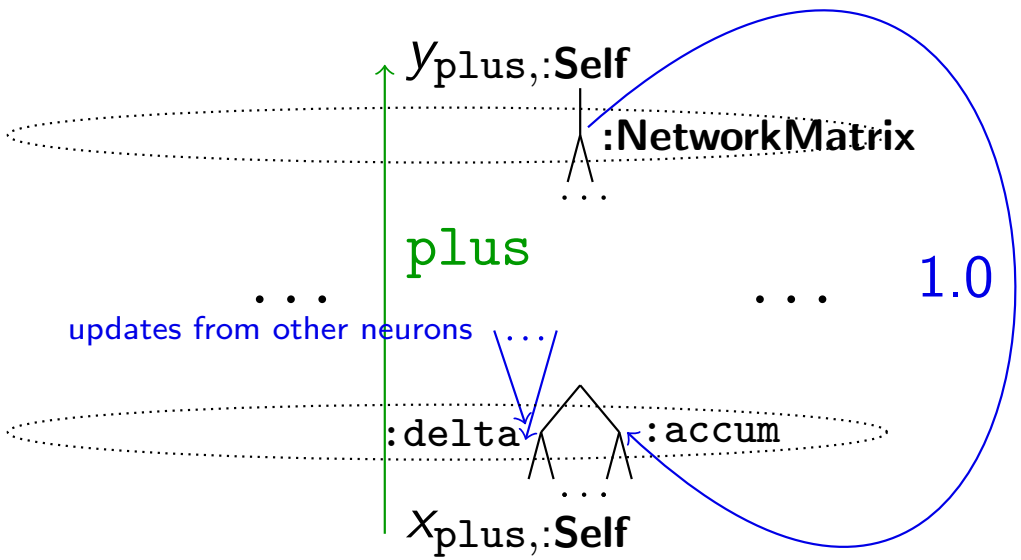
Use the most recent V-value from that stream as the network matrix **W** during the next "down movement".

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

Self-referential facilities

Our current implementation: `Self` connected as an accumulator.

It accumulates the value of the network matrix and accepts additive updates from other neurons in the network.



The most recent V-value at the `:NetworkMatrix` output of $y_{\texttt{plus},:\textbf{Self}}$ neuron is used as **W**.

Self-referential facilities

Other neurons can use `Self` outputs to take into account the structure and weights of the current network **(reflection)**.

We have used self-referential mechanism to obtain waves of connectivity patterns propagating within the network matrix.

We have observed interesting self-organizing patterns in self-referential networks.

We also use this mechanism for "pedestrian purposes": to allow a user to edit a running network on the fly.

AI safety issues are quite real here

Promise of DMMs

- **Learning to learn**

  We expect that networks which modify themselves should be able to **learn to modify themselves better**.

- **Program synthesis**

  DMMs combine
  - aspects of **program synthesis** setup (compact, human-readable programs);
  - aspects of **program inference** setup (continuous models defined by matrices).

- **But how to turn this promise into reality?**

Julia Flux and Zygote.jl

Julia Flux/Zygote.jl is the first machine learning system with enough flexibility to work with flexible DMMs and V-values "as is".

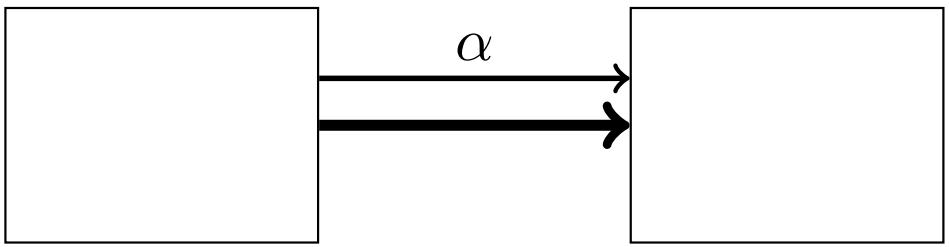
In particular, this has been the first machine learning system which can take **gradients with respect to trees**.

There is some competition now (e.g. JAX, Enzyme.jl), but Zygote.jl remains my favorite so far.

Neural architecture search

We use neural architecture search to synthesize a neural machine with desired properties:

Consider a neural machine consisting of modules (DMM neurons) connected with data pipelines gated by scalars (DMM weights).



Data in the pipeline are multiplied by the scalar α

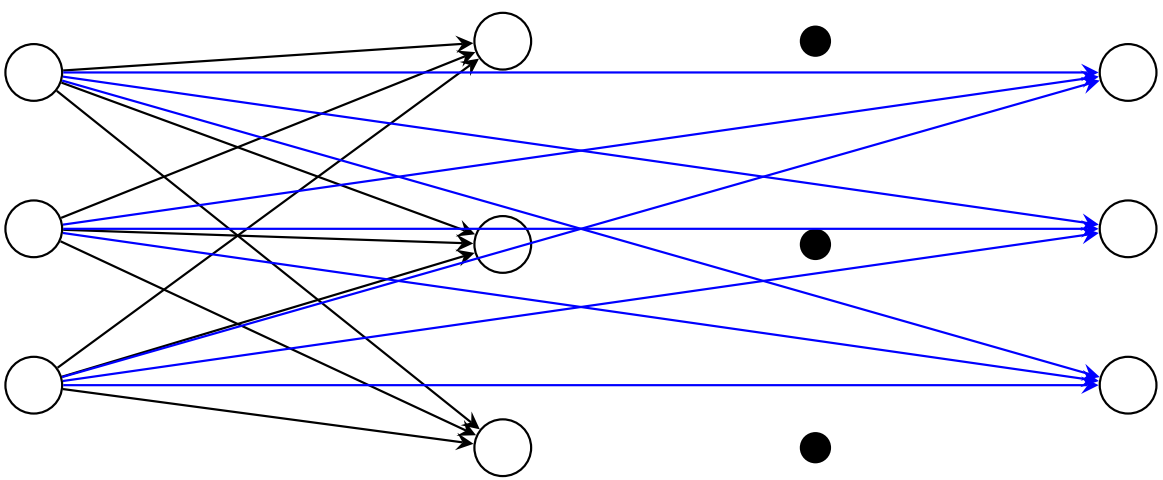
Take a large network with modules connected with each other in this fashion and start training it with *sparsifying regularization*, and at some point start **gradually pruning** those connections which have small α while continuing to train.

DMM synthesis via neural architecture search

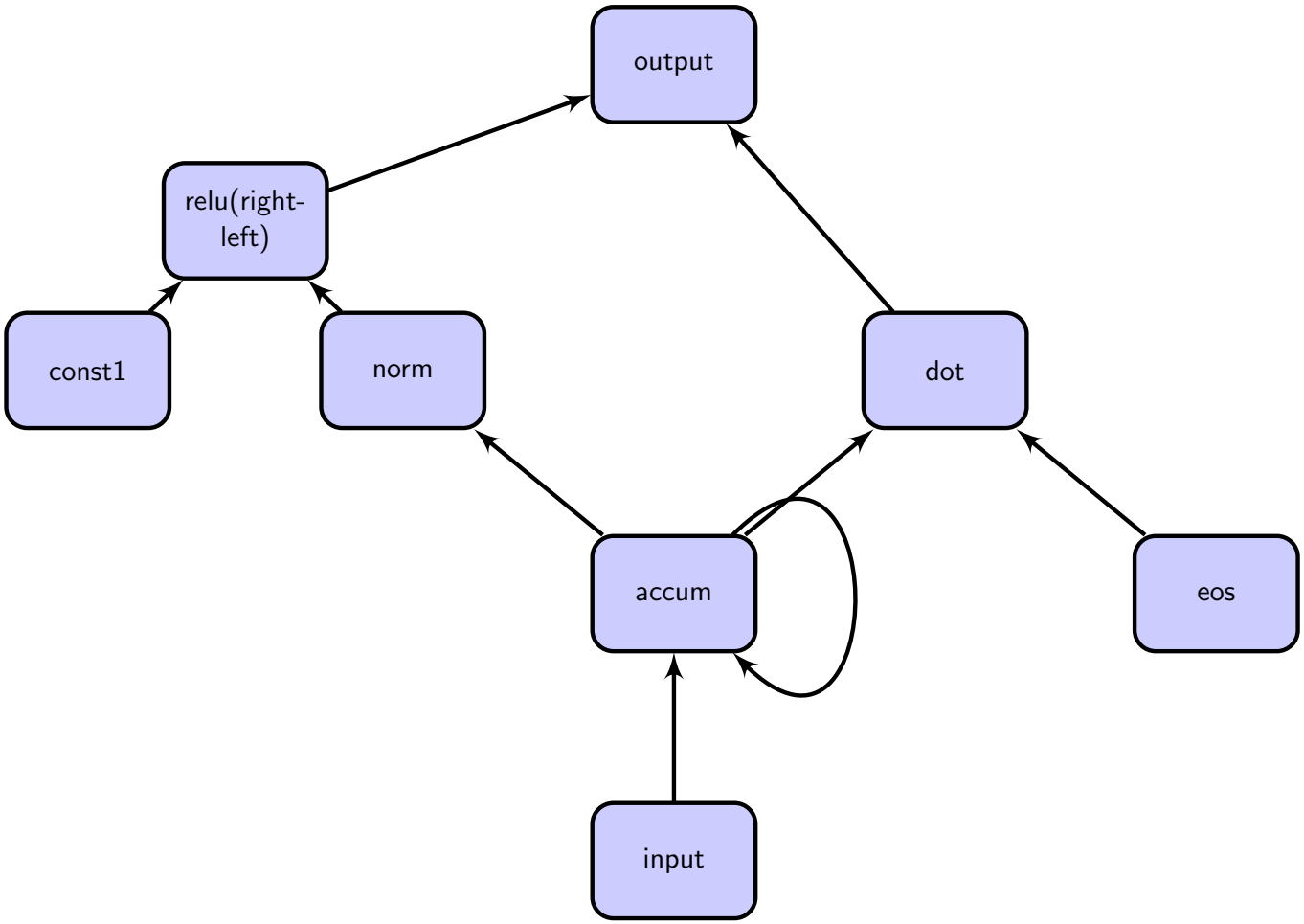
In our case, modules are DMM neurons, data pipelines carry linear streams of dictionaries, and α are DMM weights.

I am going to synthesize a feedforward DMM.

Let's take a feedforward network with fully connected layers, and with skip connections, so that each layer is fully connected to each subsequent layer (before we start pruning).



A handcrafted DMM implementing a duplicate detector



Starting point

We want to synthesize a DMM which obtains results similar to the handcrafted DMM.

We form the following feedforward network (DMM) with fully connected layers and skip connections.

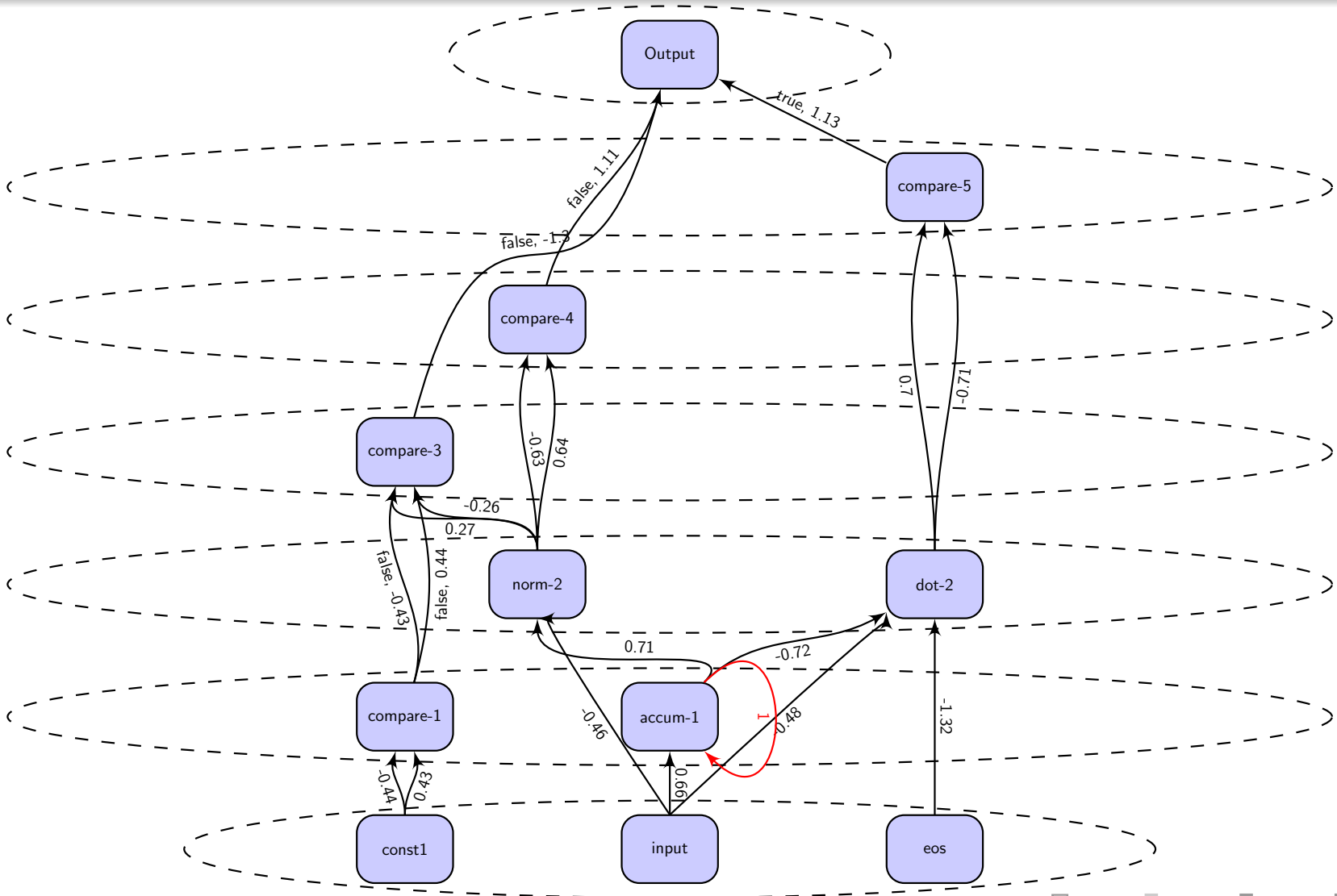
Initial layer: one of each of `input`, `const1`, `eos` neurons.

5 intermediate layers with 2 neurons of each of the following types in each layer: `accum`, `norm`, `dot`, `(relu(x-y), relu(y-x))`.

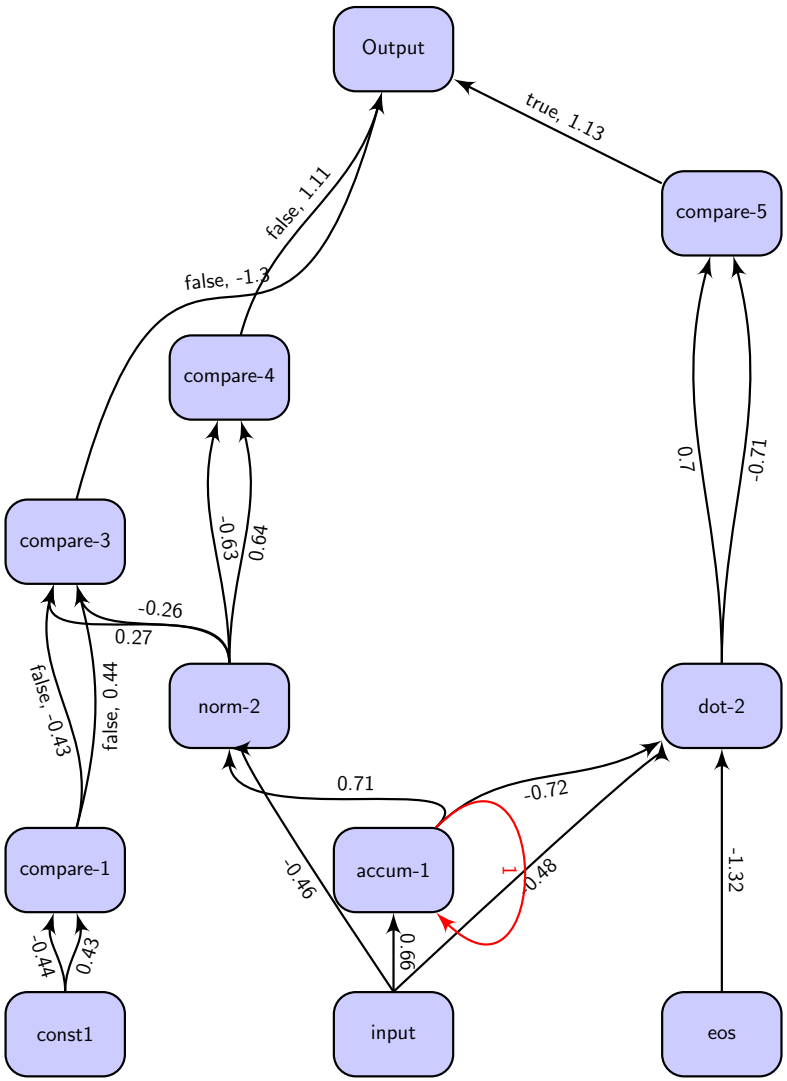
Final layer: the `output` neuron.

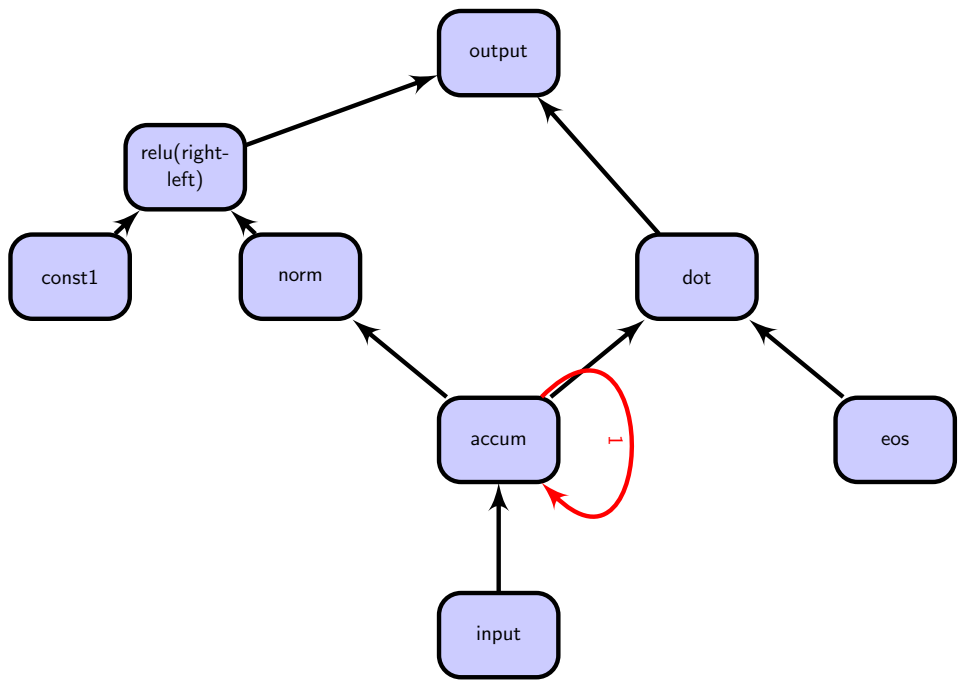
This network has slightly under 1500 parameters.

Most connections are pruned during sparsifying training



Synthesized machine vs handcrafted machine





Synthesized network matrix as V-value (JSON)

19 trainable parameters:


```
{
      "norm-2-1": {
                    "dict": {
                              "accum-1-2": {
                                            "dict": 0.7146392
                                          },
                                 "input": {
                                            "char": -0.46103984
                                          }
                                }
                  },
       "dot-2-2": {
                    "dict-2": {
                                 "eos": {
                                           "char": -1.3165319
                                         }
                               },
                    "dict-1": {
                                 "accum-1-2": {
                                                "dict": -0.7224683
                                              },
                                    "input": {
                                                "char": -0.47540766
                                              }
                                }
                  },
          ...
```

Training data and generalization properties

Minimalistic training data: `"test string..."`

Feeding 1 character per 10 time steps, so `Dict("t"=>1.0f0)`, then `DictString, Float32()` 9 times, then `Dict("e"=>1.0f0)`, etc.

Despite training data being so tiny, it generalizes almost perfectly on all kinds of short strings.

But the learned algorithm does have defects, so when one goes for longer strings it stops performing well (it is supposed to accurately count maximal degree of duplication it has observed and the number of "end of sentence" symbols it has seen).

Remarks about Zygote.jl

- Zygote takes gradients with respect to dictionaries and nested dictionaries just fine, as long as you don't use it in "old style" $\Rightarrow$ don't use `params`
- Sometimes the generated code computes gradients incorrectly $\Rightarrow$ do selective tests comparing vs. numerical derivatives
- "Backpropagation theorem" promises that gradient computations take no longer than 4*(time taken by forward computation), but sometimes generated code is slower than that for various reasons, and one has to be ready for this
- It's really good to have help from people who are experts in your favorite autodiff system! (Thanks to Mike Innes for help.)

This is an exploration, we want to make it a technology

Next steps:

- Scale this up, make it more performant
- Further experimental studies on various tasks
- Julia code professional enough to make a package
  (the current code base is "an open source prototype")
- This really needs a small team rather than a single person

Thanks to `GPT-4` for help with `TikZ` pictures in the last section.

Contact: `github:anhinga` (e.g. open an issue)

bukatin@cs.brandeis.edu (or michael.bukatin at gmail)

I am looking for collaborators