Dataflow matrix machines, tree-shaped flexible tensors, neural architecture search, and PyTorch

Michael A. Bukatin

January 4, 2021[1]

Dataflow matrix machines (DMMs) form a **flexible class of neural machines** with remarkable properties. They combine **general-purpose programming powers of stream-oriented architectures** such as traditional dataflow programming and more novel functional reactive programming with **good machine learning properties of conventional neural networks.**

As a programming framework, DMMs have affinity with synchronous versions of dataflow and functional reactive programming. They generalize digital audio synthesis based on composition of unit generators (transformers of streams of numbers), thus providing potential to generalize the style of programming via composition of unit generators to visual animations, virtual reality, and eventually to general-purpose programming.

A class of spaces of V-values (flexible tensors based on tree-shaped indices) is extremely convenient for a variety of purposes. V-values allow to bring conventional data structures into the neural context, are convenient for hierarchies and for neuro-symbolic methods, are used to allow variadic activation functions, and have good potential for use in creating and training flexible neural interfaces between pre-existing software systems.

When considered as neural machines, DMMs are remarkable for their strong self-referential facilities, allowing a neural network of this class to analyze and modify its current configuration on the fly. They form a natural framework for modular neural networks. This suggest a strong potential for their use in learning to learn, and also in neuroevolutionary methods.

Dataflow matrix machines have strong affinity with methods of **Neural Architecture Search (NAS)**. In particular, when one reformulates program synthesis as DMM synthesis, the task of *learning program sketches* is reformulated as neural architecture search. At the same time, the ability of DMMs to transform themselves and other DMMs make them a strong platform for metalearning, and, in particular, make them a strong platform for generating and learning novel Neural Architecture Search methods.

One can choose to implement DMMs within one of the next generation ultra-flexible machine learning platforms such as **JAX** or **Julia Flux**, or one can choose to implement DMMs within one of the mainstream machine learning platforms such as **PyTorch**. We consider some of the trade-offs and technical aspects associated with this choice.

1 Introduction

When one considers undertaking a project of implementing and using a new class of neural machines, one needs to address two classes of questions. One needs to identify promising applications for the new class of neural machines and to decide which of those applications to focus upon initially. One also needs to choose the most appropriate implementation environment (programming language, machine learning framework, and so on).

I want to address these two questions in the present short draft with regard to dataflow matrix machines. If an organization already has an ongoing program working with methods of Neural Architecture Search, there is a lot to be said in favor of choosing Neural Architecture Search as the initial focus of DMM applications.

On one hand, one of the key promises of DMMs is to produce breakthroughs in program synthesis, and the most natural route to making this promise a reality is to bring DMMs into a robust NAS program. When one reformulates program synthesis as DMM synthesis, the task of *learning program sketches* is reformulated as Neural Architecture Search.

On the other hand, if one implements transformations of weight-connectivity matrices of neural machines as DMMs, one can then flexibly compose such transformations and combine them with coefficients, and one can learn such compositions and linear combinations of transformations of weight-connectivity matrices as new DMMs. So one can expect DMMs to contribute to creating better NAS methods.

So, overall, there is natural synergy between dataflow matrix machines and neural architecture search.

The second question is the choice of implementation platform. We want to consider trade-offs associated with choosing between one of the mainstream machine learning platforms such as **PyTorch** and one of the next generation ultra-flexible machine learning platforms such as **JAX** or **Julia Flux**.

[1]This is an open source research draft published in a repository available under MIT license. It includes modified fragments from our earlier open source notes; relevant references are included below.

Of course, labor-wise it is much easier to implement DMMs in something like JAX or Julia Flux. JAX and Julia Flux are already equipped with capabilities to handle tree-like data structures and to compute gradients with respect to such structures. Their ability to compute gradients of functions expressed by almost arbitrary Python or Julia code is quite helpful, because we often find it useful to introduce novel activation functions which sometimes can be quite complicated, and it's nice to have gradients of those functions available automatically.

But in terms of impact and in terms of the potentially unfolding competition between mainstream platforms such as PyTorch, and the next generation of ultra-flexible machine learning platforms, a fairly strong argument for implementing DMMs within something like PyTorch can be made.

First of all, having DMMs within PyTorch would greatly increase its flexibility without changing its architecture, which is important when one thinks about competition between PyTorch and frameworks like JAX and Julia Flux.[2]

Given that DMMs provide general-purpose programming capabilities, having DMMs within PyTorch would provide Software 2.0 capabilities within PyTorch.

What are the trade-off associated with having Software 2.0 capabilities implemented as DMMs rather than as Python or Julia? On one hand, DMMs is an unusual platform: programming in DMMs means essentially programming in dataflow or functional reactive style. It is a version of stream-oriented programming, and the implementing team needs to provides capabilities of combining several streams with numerical coefficients to make sure that these programs have properties of neural machines.

So the constraint here is that one is forced to program in a rather unfamiliar stream-oriented style. However, the benefit from making this choice is that better metalearning is available. In ordinary Software 2.0, the metalearning in its full generality is equivalent to program synthesis, and our current progress in program synthesis is relatively slow compared to many other areas of machine learning. However, if one implements Software 2.0 via DMMs, then metalearning is DMM synthesis, which we expect to be a much more tractable problem.

Also, there is a good chance that the work needed to fit flexible class of neural machines such as DMMs into PyTorch would lead to a more efficient implementation of DMMs, compared to a straightforward JAX or Julia Flux implementation. Another benefit of doing this in PyTorch is that a PyTorch implementation can immediately reach a much wider community of professional users.

In the rest of this draft, I include some further details relevant to what I am saying in this Introduction.

2 Background: how DMMs work

The essence of neural model of computations is that linear and non-linear computations are interleaved. Hence, the natural degree of generality for neuromorphic computations is to work not with streams of numbers, but with arbitrary streams supporting the notion of linear combination of several streams (**linear streams**).

Dataflow matrix machines (DMMs) form a novel class of neural machines, which work with wide variety of **linear streams** instead of streams of numbers. The neurons have arbitrary arity (arity of a neuron can be fixed or variable). Of particular note are self-referential facilities: ability to change weights, topology, and the size of the active part of the network dynamically, on the fly, and the reflection capability (the ability of the network to analyze its current configuration).

There are various kinds of linear streams. They include streams of numbers, sparse vectors and sparse tensors (both of finite and infinite dimension), streams of functions and distributions. We found streams of V-values (**flexible tensors** based on tree-shaped indices) to be of particular use.

A single dataflow matrix machine can process a large variety of different kinds of linear streams, or it can be based on a single kind of linear streams, sufficiently expressive for a given class of situations.

This allows us to obtain neural machines which combine **general-purpose programming powers of stream-oriented architectures** such as traditional dataflow programming and more novel functional reactive programming with **good machine learning properties of conventional neural networks.**

There are deep connections between DMMs and attention-based models including Transformers. Each input of a neuron computes a linear combination of linear streams (which tend to be high-dimensional or infinite dimensional entities), so each input of each neuron performs a (generalized) attention operation. Transformer-like rewrites of DMM attention operations in terms of matrix multiplication are also available in many situations.

[2]Some members of the PyTorch community seem to advocate a closer collaboration between PyTorch and JAX, in particular, suggesting to use JAX **pytree** protocol via a PyTorch wrapper and to have an ecosystem of projects depending on both PyTorch and JAX. If this route is taken, it might be better to implement DMMs on the JAX side using JAX **pytree** capability directly.

Dataflow Matrix Machines resources:

Reference paper: `https://arxiv.org/abs/1712.07447`

Reference slide deck: `https://researcher.watson.ibm.com/researcher/files/us-lmandel/aisys18-bukatin.pdf`

GitHub Pages: `https://anhinga.github.io`

Open source implementation (Clojure): `https://github.com/jsa-aerial/DMM`

The present draft includes modified fragments from:

A white paper on DMMs: `https://www.cs.brandeis.edu/~bukatin/dmm-white-paper-2019.pdf`

An interdisciplinary collaborative research agenda related to DMMs:

`https://www.cs.brandeis.edu/~bukatin/dmm-collaborative-research-agenda.pdf`

3 Conventional programming and program synthesis

The dimension of the network and the dimension of data are decoupled, so compact neural machines for solving conventional programming problems are available. For example, by considering streams of maps from words to numbers, one can build a dataflow matrix machine counting words in a given text which uses only a few neurons (Section 3 of `https://arxiv.org/abs/1606.09470`). Similarly, by considering streams of V-values (flexible tensors based on tree-shaped indices) and embedding of lists into trees, one can build a similarly compact dataflow matrix machine accumulating a list of asynchronous incoming events (e.g. mouse clicks, see Section 6.3 of the DMM reference paper, `https://arxiv.org/abs/1712.07447`).

For more examples of DMMs as programs see *Map of DMM-related programming examples and techniques*:

`https://github.com/anhinga/2020-notes/tree/master/programming-overview`

The task of synthesis of dataflow matrix machines should be more tractable than conventional program synthesis. When one works with DMMs, the task of **learning program sketches** is reformulated as **Neural Architecture Search**, and converting a program sketch to a full program should be done by conventional methods of neural net training.

Dataflow matrix machines allow us to combine

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

4 Self-modification, learning to learn, and neuroevolution

Using neural networks for metalearning is always non-trivial. In particular, dimension mismatch, namely the number of neuron outputs being much smaller than the number of network weights, means that a neural network can only modify itself in a highly constrained manner. Dataflow matrix machines address this problem and have **powerful and flexible self-modification facilities**.

Therefore, a dataflow matrix machine can be equipped with a variety of primitives which perform self-modifications, and it can fruitfully learn various linear combinations and compositions involving those primitives.

Self-modification facilities of dataflow matrix machines are not limited to the weight changes for the existing connections in the network. *The available primitives allow to modify the network topology as well.* For example, primitives allowing the network to control its own fractal-like growth by the means of cloning its own subnetworks are available.

Therefore, this is a very promising architecture not only for methods of learning to learn better in a traditional sense, but also for *methods of learning to perform Neural Architecture Search better.*

A dataflow matrix machine can comfortably host an evolving population of other DMMs inside itself, so it is an excellent environment for neuroevolution experiments and, in particular, for the experiments aiming to learn to evolve better (or to evolve to evolve better).

In our software experiments, we used self-modification facilities to

- produce controlled wave patterns in the network matrix (see Appendix B.2 of our LearnAut 2017 paper, `https://arxiv.org/abs/1706.00648`);

- create randomly initialized self-referential DMMs which generated interesting emerging behaviors (see Section 1.2 of our 11-2018 technical report, `dmm-notes-2018.pdf`);
- edit a running network on the fly by sending it requests to edit itself (in particular, this enables **live-coding**, but this is also quite open-ended, since it enables a population of networks to tell each other to modify themselves; of course, the receiving network doesn't have to follow an incoming instruction to self-modify blindly, although in the most simple-minded case it would do so; see Section 1.1 of our 11-2018 technical report, `dmm-notes-2018.pdf`).

5 From DMM research to DMMs as a machine learning platform

5.1 The current state and the default path of further development

We typically think of implementing DMMs as an embedded DSL into an existing programming language and, optionally, into an existing machine learning framework.

The current open-source reference implementation of DMMs is a DSL embedded into Clojure and it works with *immutable streams of flexible tensors with tree-shaped indices also known as V-values.* We have also produced open-source implementations for some rigid subclasses of DMMs in Processing (a simplified Java for digital artists) using ordinary mutable multidimensional arrays.

While we have performed a number of software experiments with self-modifying neural machines (DMMs), and while the class of DMMs includes known neural networks as subclasses, our group has only done preliminary design work for future machine learning experiments with DMMs.

Some of the dichotomies here are between gradient-based methods and gradient-free methods, and also between GPU/TPU acceleration and just using CPU cores.

For moderate scale experiments, one can simply use derivative-free methods and CPU cores. For example, as a derivative-free method, one can use the modern incarnation of evolution strategies, introduced by researchers from OpenAI[3] and further elucidated by researchers from Uber AI Labs[4].

We have also started to do exploratory work towards using an adaptive "population coordinate descent" derivative-free schema. The essence of that schema is to maintain an evolving population of directions which forms an overdefined coordinate system and to use an adaptive probability distribution/adaptive sampling schema to repeatedly sample a direction for the next step of coordinate descent[5].

In any case, there are plenty of derivative-free methods to choose from, and experiments of modest size using derivative-free methods on CPU can be performed using our existing implementations.

However, looking forward, one really wants to have an option of using gradient-based methods and GPU/TPU acceleration. As we noted above, it is most economical in terms of labor to implement DMMs in something like JAX or Julia Flux, using our Clojure code as initial guidance. JAX and Julia Flux are already equipped with capabilities to handle tree-like data structures and to compute gradients with respect to such structures. Their ability to compute gradients of functions expressed by almost arbitrary Python or Julia code is quite helpful, because we often find it useful to introduce novel activation functions which sometimes can be quite complicated, and it's nice to have gradients of those functions available automatically.

It is convenient that our Clojure implementation works with immutable streams since both JAX and Julia Flux impose some immutability requirements. JAX arrays must all be immutable. We can use JAX **pytree** protocol to implement streams of V-values (we'll have to express scalars as 1-element arrays whenever necessary), and **pytree** protocol might provide a practically convenient generalization of our default flavor of V-values.

Julia Flux, "the ML library that doesn't make you tensor", is also a very good fit for our use case[6]. Its incredibly flexible *Zygote* differentiable programming system is particularly impressive[7]. It is a feature of Zygote that it is comfortably handling any *nested dictionaries* and other nested data structures, and those structures can generally be *mutable*, with one exception: the arrays must still be immutable.

[3]Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, Ilya Sutskever, *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*, March 2017. `https://arxiv.org/abs/1703.03864`

[4]Xingwen Zhang, Jeff Clune, Kenneth Stanley, *On the Relationship Between the OpenAI Evolution Strategy and Stochastic Gradient Descent*, December 2017. `https://arxiv.org/abs/1712.06564`

[5]This work is still in its exploratory design stage: `https://github.com/anhinga/population-of-directions`

[6]`https://github.com/FluxML/Flux.jl`; the reference paper for Julia Flux is Michael Innes et al., *Fashionable Modelling with Flux*, November 2018, `https://arxiv.org/abs/1811.01457`

[7]Michael Innes, *Don't Unroll Adjoint: Differentiating SSA-Form Programs*, October 2018, `https://arxiv.org/abs/1810.07951` and Michael Innes et al., *A Differentiable Programming System to Bridge Machine Learning and Scientific Computing*, July 2019, `https://arxiv.org/abs/1907.07587`

5.2 Flexibility vs. parallelization and optimization

Extreme flexibility of DMMs is provided by the use of V-values (flexible tensors with tree-shaped indices, see Section 3 the DMM reference paper, `https://arxiv.org/abs/1712.07447`, for the theory of V-values), sparse connections, and the ability of the network to self-reconfigure on the fly.

Obviously, there is a lot of tension between that and the ability to provide an efficient implementation, and especially with the ability to provide parallelized, GPU-friendly, and batching-friendly implementation.

The task of doing so is not impossible, but can be quite involved (see, for example, TensorFlow `Fold`, a library for working with dynamic computation graphs). See Appendix F of `https://arxiv.org/abs/1610.00831` for further discussion.

We started the initial design work towards reconciling tree-shaped tensor indices and GPUs/TPUs in `https://github.com/anhinga/2019-design-notes` repository. It is also promising to consider more flexible parallel architectures than GPUs, such as, for example, architectures based on FPGAs, and also commercial flexible alternatives to GPUs, which are under development at a number of organizations.

5.3 PyTorch-only route

The most popular machine learning frameworks, such as PyTorch, are a nice fit for rigid subclasses of DMMs, like those we have implemented in Processing. Sparse matrices are still highly desirable even for rigid subclasses of DMMs, but recent progress in the state of sparse matrices in PyTorch itself and in the PyTorch ecosystem in general is very encouraging.

However, traditional mainstream machine learning frameworks are oriented towards standard tensors (multidimensional arrays of a fixed number of dimensions and fixed sizes along each of those dimensions), and using them for the more flexible variety of DMMs based on flexible tensors with tree-shaped indices is non-trivial.

One possibility is to consider methods of reshaping and flattening of the tree-shaped indices into flat shapes of indices of multidimensional arrays. After all, reshaping of the same tensor, say, between one-dimensional and two-dimensional representations is frequent in machine learning applications. So reshaping from tree-shaped structures to flat structures and back would be in the same spirit.

A design sketch for one possible way of flattening and reshaping tree-shaped indices to fit the "fixed number of dimensions/fixed size" framework can be found here:

`https://github.com/anhinga/2019-design-notes/blob/master/automated-synthesis/flattening-of-v-values.md`.

We should try to see if this can be done via the usage of *minimal perfect hashes*, e.g. following the scheme by Czech et al., "Perfect Hashing", Theor.Comp.Sci. **182** (1997):

`https://www.sciencedirect.com/science/article/pii/S0304397596001466`.

In particular, the following implementation might be quite useful:

`https://github.com/eddieantonio/perfection/blob/master/perfection/czech.py`.

5.4 PyTorch + JAX (or Julia Flux)

Some members of the PyTorch community would like to see tighter collaboration between PyTorch and JAX communities. For example, Sabrina Mielke and Sasha Rush implemented an immutable, stochastic module system for PyTorch, and they would like to see it ported to JAX: `https://github.com/srush/parallax` (see `https://twitter.com/srush_nlp/status/1262741628810190850` for discussion, May 2020).

Sabrina Mielke wrote a very useful essay, "From PyTorch to JAX: towards neural net frameworks that purify stateful code": `https://sjmielke.com/jax-purify.htm`
(see `https://twitter.com/sjmielke/status/1237042622029385728` for discussion, March 2020).

We also see a discussion between Sasha Rush and Adam Paszke on methods to enable autograd with respect to tree-like structures in PyTorch, and the consensus seems to be that it's OK to use a wrapper around JAX **pytree**, at least tentatively: `https://twitter.com/srush_nlp/status/1262868401921368065`.

So we can make two conclusions here. First, it's OK to tentatively use a PyTorch wrapper around JAX **pytree**, at least for the proof-of-concept projects. Whether it is a good permanent solution should be considered separately.

The second conclusion is that if people discuss implementing and using DMMs in the context of PyTorch, the somewhat uncertain possibility of closer relationships between PyTorch and JAX (or, perhaps, even between PyTorch and Julia Flux) is a factor to consider.