Dataflow Matrix Machines: a Collaborative Research Agenda

Michael A. Bukatin

December 24, 2024[1]

Dataflow matrix machines form a quintessential interdisciplinary field. They emerged as a class of neural machines expressive enough to also serve as a viable programming framework. Their historical roots are in the synthesis of domains for denotational semantics and vector spaces. There are deep connections between vector semantics of programming languages and fuzzy and multivalued logic of partial inconsistency. The dynamical systems based on dataflow matrix machines exhibit a variety of interesting emerging properties.

As a programming framework, dataflow matrix machines 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, 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, dataflow matrix machines 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 were discovered and studied by a series of small-scale academic research collaborations. To unlock their full interdisciplinary potential, it would be necessary to generate a wider interest in this class of programmable neural machines.

In this note, I provide a bit more details and key references for some of the promising interdisciplinary research directions I see at the moment. I hope readers from various fields would find this of interest.

[1]Version 2.5. Version 2.0 (08-07-20) is at `https://github.com/anhinga/2020-notes/tree/master/research-agenda`

1 Background: how they 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.

Dataflow Matrix Machines resources:

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

Reference slide deck: `https://github.com/jsa-aerial/DMM/blob/master/doc/IBM-AI-Systems-Day-2018/aisys18-bukatin.pdf`

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

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

2 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).

First successful experiments in program synthesis/circuit synthesis/DMM synthesis via neural architecture search were performed in June 2022[2].

[2]See `history.md` at `https://github.com/anhinga/DMM-synthesis-lab-journal`

3 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`).

4 Dynamical systems based on DMMs and emerging properties

The dynamical systems based on dataflow matrix machines exhibit a variety of interesting emerging properties. We conducted two series of experimental studies which yielded interesting emerging properties. These experiments were implemented in Processing programming language.

The first series of experiments was conducted in August 2015 and involved continuous cellular automata (see Section 4 of the preprint introducing the class of neural machines which we later started to call DMMs, `https://arxiv.org/abs/1601.01050`). The following videos show some of the emerging patterns we observed:

- `https://youtu.be/KZHQxdZUlSU`
- `https://youtu.be/-pFil1_GEA4`

The second series of experiments was conducted in 2016-2018 and involved "pure lightweight dataflow machines" (see Section 1.2 of our 11-2018 technical report, `dmm-notes-2018.pdf`). We observed, for example, the following emerging patterns:

- `https://youtu.be/_mZVVU8x3bs` - emerging sleep-wake patterns
- `https://youtu.be/CKVwsQEMNjY` - emerging oscillations

This is a fertile setup to discover new emerging patterns of behavior, to study those patterns mathematically, and, perhaps, to eventually design novel emerging patterns of behavior.

5 Synchronous vs. async, and artificial vs. biological neural nets

Like all neural machines, DMMs tend to be synchronous, because it is much easier to combine several streams with coefficients in the synchronous model of computations. As such, they have affinity with synchronous dataflow and functional reactive languages and libraries, such as, for example, `Lucid Synchrone` programming language and `Yampa` (a Haskell library).

By default, DMMs tend to provide extreme sparseness in connectivity patterns ("sparseness in space"), but not sparseness in time.

On the other hand, biological networks are asynchronous, providing sparseness in time, just like general dataflow programming tends to be asynchronous, and general functional reactive programming, in particular, tends to provide sparseness in time.

At the same time, biological neural networks tend to be equipped with various synchronization mechanisms. E.g. on the lower level, mechanisms such as, for example, **leaky integrate-and-fire** enable combining asynchronous inputs with coefficients. On higher levels, there are various mechanisms synchronizing firing times of different neurons (see, for example, research by the group led by Nancy Kopell at Boston University starting from around 2005 for various mathematical models of those mechanisms).

Reconciling synchronous and asynchronous models of computation, and, in particular, borrowing from what we know about biological neural nets and bringing it into realm of artificial neural machines is an important direction. Just like all "sparseness in time", it is also quite relevant to the field of low energy neural computations, which keeps growing in importance.

6 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.

This is an important and potentially very fruitful area of collaboration between specialists in parallel and efficient implementation of algorithms and people focusing on flexible neural architectures.

6.1 Design for DMMs as a machine learning platform: PyTorch/TensorFlow vs. JAX/Julia Flux

While we have performed a number of 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[3].

Some of the dichotomies here are between gradient-based methods and gradient-free methods, and also between GPU 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[4] and further elucidated by researchers from Uber AI Labs[5].

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

[3]This subsection is a copy of Appendix A.3 of Michael Bukatin, *Synergy between AI-generating algorithms and dataflow matrix machines*, March 2020. `https://github.com/anhinga/2020-notes/tree/master/research-notes`

[4]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`

[5]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`

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[6].

However, looking forward, one really wants to have an option of using gradient-based methods and GPU/TPU acceleration. The most popular machine learning frameworks, such as PyTorch, are a nice fit for rigid subclasses of DMMs. However, they 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[7].

A better fit might be a machine learning framework, designed from start with the degree of flexibility we would like to have here. In particular, Julia Flux, "the ML library that doesn't make you tensor", might be a good fit for our use case[8]. Its incredibly flexible *Zygote* differentiable programming system is particularly impressive[9].

JAX has a comparable degree of flexibility as well with its **pytree** protocol and differentiable programming capabilities, and a benefit of having the DeepMind JAX Ecosystem built around it[10].

For a more detailed analysis of choosing a PyTorch-like platform for DMM implementation vs. choosing one of the next generation ultra-flexible machine learning platforms such as Julia Flux or JAX see our preliminary research draft[11] .

JAX and Zygote.jl (Julia Flux) are capable of taking gradients with respect to variables stored inside nested dictionaries[12].

First successful experiments in DMM training and in program synthesis/circuit synthesis/DMM synthesis via neural architecture search were performed in June 2022 using Zygote.jl[13].

7 DMMs vs. differentiable programming: a meta-learning aspect

Given that DMMs are neural machines with general-purpose programming capabilities, using DMMs as a programming formalism would provide differentiable programming ("Software 2.0") capabilities[14]

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 taking linear combinations of streams in question* (that is, the ability to combine several streams with numerical coefficients) *so that the resulting programs are 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 includes 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.

At the same time, the whole point of modern differentiable programming (mostly, in its Julia Flux and Python/JAX incarnations) is that one can incorporate fragments of more traditional models straight into a reasonably general subset of Julia or Python, using the resulting programs as more expressive models with strong priors. In particular, one can incorporate fragments of DMMs and DMM-related constructions straight into Julia or Python; therefore one can start using various DMM-related methods within a differentiable

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

[7]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`

[8]`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`

[9]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`

[10]`https://deepmind.com/blog/article/using-jax-to-accelerate-our-research`

[11]Michael Bukatin, *Dataflow matrix machines, tree-shaped flexible tensors, neural architecture search, and PyTorch*, January 2021. `https://github.com/anhinga/2021-notes/tree/main/research-drafts`

[12]See `https://github.com/anhinga/jax-pytree-example` and
`https://github.com/anhinga/julia-flux-drafts/tree/main/arxiv-1606-09470-section3`

[13]See `history.md` at `https://github.com/anhinga/DMM-synthesis-lab-journal`

[14]In this section we use "differentiable programming" and "Software 2.0" interchangeably while recognizing that modern differentiable programming permits some discontinuities, e.g. the use of ReLU brings discontinuities in derivatives, etc.

programming framework without being purist and without waiting until one is in position to move one's whole Software 2.0 process entirely to DMMs.

8 Links to fuzzy and multivalued logic of partial inconsistency

The historical roots of dataflow matrix machines are in the synthesis of domains for denotational semantics and vector spaces. It was that synthesis, which informed us that it was likely that programming with linear streams could be made powerful enough to be used for general-purpose programming.

The key element of that synthesis is the ability to handle partial contradictions (that is, to handle overdefined elements in addition to partially defined elements). For interval arithmetic, this corresponds to introduction of pseudosegments $[a, b]$ with the contradictory property that $b < a$ (the first known discovery of those overdefined interval numbers is by Mieczyslaw Warmus in his "Calculus of Approximations", 1956). For probability theory, this corresponds to allowing negative values of probabilities, which originally comes from physics (e.g. as quasiprobabilities by Eugene Wigner in 1932, and then as quasiprobabilities for phase-space formulation of quantum mechanics independently developed by José Moyal and Hilbrand Groenewold in 1940-es).

For the detailed overview of the material linking vector semantics with partial inconsistency, see Section 4 (and, in particular, Sections 4.2-4.4, and Sections 4.7-4.12) of our GCAI 2015 paper, "Linear Models of Computation and Program Learning", `https://easychair.org/publications/paper/Q4lW`. Because in addition to Lawvere duality between fuzzy orders and quasi-metrics, a similar duality exists between multivalued equalities and partial metrics (`https://www.cs.brandeis.edu/~bukatin/distances_and_equalities.html`), here we have both partial metrics and fuzzy multivalued predicates taking values in Warmus' partially inconsistent interval numbers.

There are interesting possible connections to be explored between this material and probabilistic logic networks by Goertzel et al. (2008).

8.1 Domain equations for bicontinuous domains

Beginnings of high-order domain theory for bicontinuous domains are present in both of the respective papers by Klaus Keimel and by Dexter Kozen[15].

However, no theory of domain equations for bicontinuous domains has emerged so far to the best of my knowledge. Even the question of the choice of morphisms in this context is non-trivial. Generally speaking, one can use each of the two Scott topologies on the bicontinuous domain in question for computations, and one can also use order-reversing involutions (Sections 4.5 and 4.14 of our GCAI 2015 paper).

At the same time, on the level of operational semantics we have a rich theory of self-referential dataflow matrix machines (see Section 3 of the present text). What would constitute an adequate theory of reflexive bicontinuous domains in this context is an important open question.

9 DMMs and probabilistic programming

Our preprint[16] describes mechanisms allowing to integrate externally generated streams of probabilistic samples into neural machines by combining those streams with coefficients and using stream transformers built into the neurons. These mechanisms cover streams of quasi-probabilities where positive and negative probability values are allowed, and also complex-valued streams. Here we include the material from Section 8 of that preprint.

Observe that the neural machines in question are recurrent, so rather interesting streams might result from this setup. A dataflow matrix machine is capable of changing the coefficients used to combine streams (and can even reconfigure its own topology governing the connectivity between neurons), so there is good potential to create new methods to synthesize various desired streams in this setup.

There are various intriguing possibilities of further development in connection with this formalism.

In particular, two motives are prominent in probabilistic programming: probability distributions are defined by constraints expressed by probabilistic programs, and a variety of sampling methods is used to compute those

[15] K. Keimel. Bicontinuous domains and some old problems in domain theory. *Electronic Notes in Theoretical Computer Science*, **257**:35-54, 2009; D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, **22**(3):328-350,1981.

[16] M. Bukatin. *Using streams of probabilistic samples in neural machines.* January 2020.
`https://github.com/anhinga/2020-notes/tree/master/research-notes`

distributions[17]. It would be interesting to explore the following applications of DMMs:

- the situations where probability distributions are defined by constraints expressed by programs in the form of dataflow matrix machines (**DMMs as probabilistic programs**);
- the situations where the sampling computing those distributions is performed by running dataflow matrix machines (**sampling DMMs**).

Sampling DMMs might take DMMs used as probabilistic programs as inputs[18].

10 Real-time functional programming: visual animation and virtual reality via composition of unit generators

Dataflow matrix machines generalize digital audio synthesis based on composition of unit generators (transformers of streams of numbers). Real-time digital audio synthesis and real-time generation of computer animation and virtual reality form very natural classes of soft real-time programming.

Moving from streams of numbers to more general linear streams, such as streams of V-values, should provide sufficient expressive power to program 2D and 3D visual animation based on composition of unit generators understood as transformers of linear streams. We explored this approach in various pre-DMM prototypes, and later in DMM systems, both for fixed computational graphs and for computational graphs dynamically changing on the fly. For our initial pre-DMM work in this direction see, for example, Sections 3-5 of `https://arxiv.org/abs/1601.00713` and associated videos:

- `https://youtu.be/fEWcg_A5UZc` - fixed dataflow graph
- `https://youtu.be/gL2L7otx-qc` - dataflow graph changing on the fly

Note that the old-fashioned analog video synthesizers also work in the style of composition of unit generators. Modern computer graphics tends to be imperative, oriented towards specific data flows implemented inside GPUs, and to require the software practitioners to focus on manual optimizations. The promise of doing animations via functional reactive programming is to focus again on semantically meaningful data flows, and to leave the hardware-oriented optimization to the underlying system.

I hope that collaboration between people specializing in visual effects and computer animation, people who understand how to compile flexible data flows and dynamic computation graphs to GPUs, and people who focus on DMMs will make this promise a reality.

11 DMMs and Transformers

There are extremely interesting connections between DMMs and Transformers and other attention-based models. We started to make rough sketches exploring those connection available[19] beginning in July 2020, and we are going to add a brief summary here.

Transformers are models with multi-layer attention mechanism which have been introduced 3 years ago[20]. Their attention mechanism is based on linear combinations of high-dimensional vectors and the representations of those linear combinations via matrix multiplication.

Transformers are the only widely known subclass of DMMs built around linear combinations of high-dimensional vectors. Note, that the transition from linear combinations of numbers to linear combinations of high-dimensional vectors on the level of single neurons is the key factor responsible for increased expressive power of DMMs compared to RNNs.

However, that is where the similarities between Transformers and DMMs end so far. Transformers is the leading state-of-the-art architecture with demonstrated remarkable capabilities to train and learn, whereas DMMs is a formalism with very interesting and remarkable expressive powers, but without any track record in terms of training and learning at this point in time.

[17] See, for example, van de Meent et al., *An Introduction to Probabilistic Programming*, `https://arxiv.org/abs/1809.10756` and references at `http://probabilistic-programming.org`

[18] In the self-referential world of DMMs there is no precise boundary between containing a subnetwork and taking a subnetwork as an input, as a DMM has facilities to take another network as an input and to incorporate it as a subnetwork on the fly.

[19] `https://github.com/anhinga/2020-notes/tree/master/attention-based-models`

[20] A. Vaswani et al., *Attention Is All You Need*, June 2017, `https://arxiv.org/abs/1706.03762`

The Transformers were introduced with relatively modest goals of faithfully capturing long-range connections in data and enhancing parallelization capabilities during training, but it soon became apparent that their actual capabilities go well beyond that. In particular, it turned out that when one trains sufficiently large Transformers on a diverse natural language corpus in an unsupervised fashion for a task of predicting an excised word, these models capture deep hidden structure present in data and useful for this kind of prediction. In particular, it turns out that Transformers learn natural language grammar quite well in this scenario, as demonstrated both by surface performance of many models of this class and by investigation of their internal structure[21].

The initial training of Transformers tends to be quite costly, but their adaptation ("fine-tuning") to new tasks tends to be relatively inexpensive. So one can say that there is "hidden meta-learning" which is built into this architecture: after initial pretraining the model's ability to learn new things increases greatly.

These impressive capabilities were taken to an entirely new level with the introduction of GPT-3 and OpenAI API in May-June of this year[22]. I think the community is only at the beginning of the quest to understand why this architecture and especially its GPT-3 incarnation work so well[23].

It's quite promising at this point to consider interplay between attention-based models including Transformers and other approaches. Hybrid approaches of this kind are bringing progress along multiple axes[24].

We started an exploration of connections between DMMs and Transformers in July 2020[25]. So far, we have been focusing this exploration along two dimensions:

- Could what we know about DMMs shed some light on the remarkable properties of Transformers?
- What are the ways to incorporate key elements from Transformer architecture into a more flexible DMM setup, and, in particular, could we obtain interesting compact and low training cost models by incorporating attention-inspired and Transformer-inspired motives into DMMs?

In particular, we have been taking a closer look at the properties of matrix multiplication and at combining matrix multiplication and softmax together with other Transformer and DMM primitives to build small flexible neural machines. We have been exploring computational properties of those neural machines and solving selected machine learning problems involving those machines[26].

Some of our efforts in this direction were presented as a virtual poster at JuliaCon 2021[27].

The flexibility afforded by the latest generation of machine learning frameworks such as Julia Flux and JAX is extremely helpful and saves a lot of coding labor for people exploring these classes of problems.

I believe this is an extremely promising direction and I advocate its further research and exploration.

[21] E.g. A. Coenen et al., *Visualizing and Measuring the Geometry of BERT*, June 2019, `https://arxiv.org/abs/1906.02715`

[22] See, for example, my note at `https://www.cs.brandeis.edu/~bukatin/transformer_revolution.html`

[23] As of early August 2020 when I was writing this; the flood of Transformer-related literature and Transformer-related advances has been rather overwhelming in the last year, and adequately monitoring and summarizing Transformer-related research has to be a group effort these days.

[24] For example, the AlphaFold 2 model is a hybrid model with attention-based elements playing some of the key roles.

[25] `https://github.com/anhinga/2020-notes/tree/master/attention-based-models`

[26] `https://github.com/anhinga/2021-notes/tree/main/matrix-mult-machines`

[27] Michael Bukatin. *Multiplying monochrome images as matrices: A*B and softmax.* Virtual poster at JuliaCon 2021, July 2021. `https://github.com/anhinga/JuliaCon2021-poster`