Using streams of probabilistic samples in neural machines

Michael A. Bukatin

January 5, 2020

We describe a class of neural machines capable of processing streams of probabilistic samples. The samples themselves can be, e.g., compound objects of discrete nature if so desired, and the neural machines in question don't need to embed those samples into vector spaces, but can process them "as is".

While it is customary to process one stream element per one cycle of a neural machine, the mechanism we describe allows for temporal sparsity, that is, it allows to provide a stream element only occasionally, if so desired. We describe mechanisms allowing to combine streams of probabilistic samples with positive coefficients, with arbitrary real coefficients, and with complex coefficients.

In recent years my colleagues and I introduced and studied a new class of neural machines called dataflow matrix machines[1]. In particular, this class of neural machines is capable of more versatile processing of streams of probabilistic samples compared to conventional neural networks. We are looking for ways in which this would be applicable to probabilistic programming.

1 Dataflow matrix machines and linear streams

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

More specifically, the "linear" part involves combining streams of data with coefficients ("linear combinations"), while "non-linear" part involves arbitrary processing of streams inside neurons.

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 combining several streams of data with coefficients (**linear streams**).

We call neural machines working with arbitrary linear streams **dataflow matrix machines (DMMs)**.

It turns out that the transition from streams of numbers to arbitrary linear streams greatly increases the expressive power of the resulting neural machines. The dimension of the network and the dimension of data become decoupled, so compact neural networks (often consisting of just several neurons) are capable of processing streams of high-dimensional and even infinite-dimensional data. It turns out that unlike conventional neural networks, the resulting class of neural machines is a viable general-purpose programming platform. Moreover, while conventional neural networks have highly constrained and rather inconvenient self-modification options, dataflow matrix machines have powerful and flexible reflection and self-modification mechanisms, which is of considerable interest from various angles of view including potential for metalearning.

2 Streams of discrete objects in the neural framework

We believe that a general purpose programming platform should be able to process discrete data without distortions (in particular, it should be possible to process discrete data without learning their embedding into vector spaces).

We know of at least two different ways to do so in context of dataflow matrix machines and linear streams.

One approach is to consider the space of objects of interest, X, and a space of formal linear combinations of elements of X. We consider streams of such formal linear combinations. To control the size of stream elements, one can introduce a sparsity constraint, for example, requiring that a linear combination in question has no more than N non-zero terms. This means that on performing an operation of adding several such combinations with coefficients, one needs to drop the less meaningful terms from the resulting combination before proceeding further. If N is at least 2, one can have smooth transitions between discrete objects in question, although in practice one would probably want to use somewhat higher values of N.

Another approach is to think about a stream of elements of X as a stream of objects sampled from an unknown probability measure over X. This is the approach we are focusing on in the present extended abstract.

[1]For an up-to-date compact overview with detailed references see:
Michael Bukatin. *Dataflow matrix machines: a collaborative research agenda.* December 2019.
`https://www.cs.brandeis.edu/∼bukatin/dmm-collaborative-research-agenda.pdf`

3 Statistical interpretation of convex linear combinations

Consider streams of coefficients $\alpha_1^t, \ldots, \alpha_n^t$, such that $\alpha_1^t + \ldots + \alpha_n^t = 1$ and for all i, $0 \leq \alpha_i^t \leq 1$. Assume that there are streams of samples $x_1^t, \ldots, x_n^t$, and we'll think informally that on background they represent streams of probability measures $\mu_1^t, \ldots, \mu_n^t$. We would like a heuristics to sample from the probability measure $\alpha_1^t * \mu_1^t + \ldots + \alpha_n^t * \mu_n^t$.

We are going to sample an index, i, according to probability distribution $\alpha_1^t, \ldots, \alpha_n^t$ on indices $1, \ldots, n$, and then we take sample x_i^t as representing the linear combination $\alpha_1^t * \mu_1^t + \ldots + \alpha_n^t * \mu_n^t$.

4 Arbitrary real coefficients and signed samples

However, in the neural model of computations, it is really important to be able to use arbitrary real coefficients. In particular, it is really important to allow negative coefficients in order to encode inhibitory connections. In general, the ability to have things cancel each in order to obtain zero is quite important in neural models of computation[2].

Therefore, instead of probability measures, we are going to consider **the space of all finite signed measures** over X, and we are going to consider **signed samples**, that is, pairs $\langle x, s \rangle$, where $x \in X$ and s is a flag taking value 1 to denote a positive sample, and value -1 to denote a negative sample.

The Hahn-Jordan decomposition allows to represent a finite signed measure μ as a sum of a positive and a negative measure: $\mu = \mu^+ + \mu^-, \mu^+ = \mu \vee 0, \mu^- = \mu \wedge 0$. This decomposition holds because $y = (y \vee 0) + (y \wedge 0)$ is a theorem for all lattice-ordered groups.

The "perfect sample" only produces samples $\langle x, 1 \rangle$ for the part of X covered by nonzero μ^+ and samples $\langle x, -1 \rangle$ for the part of X covered by nonzero μ^-. However, we don't insist on the samples being perfect. Instead we allow positive and negative samples to cancel each other.

If we select a bin of X (a convenient measurable subset $Y \subseteq X$), we would simply want the overall sign and frequency of samples in that bin correspond to $\mu(Y)$. In particular, for a "perfect sample", if for all measurable subsets $Z \subseteq Y$, $\mu(Z) = 0$, then we never get elements of Y in our sample. However, generally speaking, we will not try too hard to achieve that, but would just want positive and negative samples from Y to balance each other.

With this in mind, we define the following sampling procedure for $\alpha_1^t * \mu_1^t + \ldots + \alpha_n^t * \mu_n^t$, assuming that the streams of finite signed measures μ_i^t are represented by streams of samples $\langle x_i^t, s_i^t \rangle$. We pick index i with probability $|\alpha_i^t| / \sum_j |\alpha_j^t|$, and then we pick the sample $\langle x_i^t, \text{sign}(\alpha_i^t) * s_i^t \rangle$ to represent the linear combination of signed measures in question.

5 Ability to omit samples: temporal sparsity

Note that the formula in the previous section does not work, if all α_i are zero. Generally speaking, it is nice to have ability to provide less than one sample per unit of time (per cycle of a neural machine). On one hand, a machine can handle a variety of kinds of linear streams at once, and it might be the case that producing samples is more expensive than other kinds of processing. It might also be the case that an external source of samples is slow and cannot keep up[3].

We are going to allow missing samples, and we are going to require that samples from zero measure are always missing.

More specifically, if we pick index i while computing a sample from $\alpha_1^t * \mu_1^t + \ldots + \alpha_n^t * \mu_n^t$, and measure μ_i^t is represented by a missing sample, then the linear combination is represented by the missing sample. This approach requires somewhat ad hoc handling of the situation when $\sum_j |\alpha_j^t|$ is small, namely adding an extra term, for example, $(1 - \sum_j |\alpha_j^t|) * \mu_0$, where μ_0 is a zero measure, resulting in $(1 - \sum_j |\alpha_j^t|) * \mu_0 + \alpha_1^t * \mu_1^t + \ldots + \alpha_n^t * \mu_n^t$.

[2]This section and the next two sections generally follow Appendix A of Michael Bukatin, Jon Anthony. *Dataflow matrix machines as a model of computations with linear streams.* In LearnAut 2017, `https://arxiv.org/abs/1706.00648`; however here we provide more detailed presentation and the degree of generality is improved.

[3]At the same time, we don't think that not being able to squeeze multiple samples into one cycle is a serious problem, just like not being able to produce colors above maximal brightness does not seem to be a serious problem.

6 Probabilistic samples within tensors

In the above we have focused on replacing separate streams of numbers with streams of probabilistic samples. It is customary to organize numbers into tensors and to consider streams of those tensors. It is convenient to be able to organize probabilistic samples in the same fashion.

What we say is applicable to traditional tensors (multidimensional arrays of a fixed shape) and to spaces of V-values (flexible tensors with tree-shaped indices we found useful in our studies). In the context of DMMs, these tensors are often quite sparse.

Elements of those tensors are numbers, and in order to integrate other kinds of data one needs to augment those numbers with something else. For example, one can consider pairs $\langle$number, sample$\rangle$ as tensor elements. So instead of having tensor elements in $\mathbb{R}$, we have tensor elements in $\mathbb{R} \oplus M$, where M is the space of finite signed measures over X.

So, the original space of tensors or V-values, V, is replaced by $V' = V \otimes (\mathbb{R} \oplus M)$. Streams of elements of V' will be represented by streams of appropriately shaped tensors with $\langle$number, sample$\rangle$ pairs as elements.

In this situation, the correct handling of zero measures and the ability to omit samples is of particular importance, since sparsity tends to be quite vital in this context.

7 Complex coefficients and samples with phase

Here we extend the considerations of Section 4 for the case of complex coefficients, which can be useful when one would like to allow complex-valued synaptic weights[4].

We replace the signed samples $\langle x_i^t, s_i^t \rangle$ with $\langle x_i^t, \phi_i^t \rangle$, where ϕ is the phase. The value -1 of the flag s_i^t corresponds to $\phi_i^t = \pi$.

A "perfect sample" would record the true phase ϕ from the underlying distribution, but this is not our goal here, just like in Section 4. Instead we adopt the following procedure for sampling from $\alpha_1^t * \mu_1^t + \ldots + \alpha_n^t * \mu_n^t$, assuming that the streams of complex-valued measures μ_i^t are represented by streams of samples $\langle x_i^t, \phi_i^t \rangle$.

We pick index i with probability $|\alpha_i^t| / \sum_j |\alpha_j^t|$, just like in Section 4. Then assuming that ψ_i^t is the phase of α_i^t, we use the sample $\langle x_i^t, \psi_i^t + \phi_i^t \rangle$ to represent the linear combination of complex-valued measures in question.

8 Next steps: probabilistic programming

We have exposed 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. Observe that the neural machines in question are recurrent, so rather interesting streams might result from this.

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 distributions[5]. 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[6].

[4]We follow the January 2017 sketch in `https://github.com/jsa-aerial/DMM/blob/master/design-notes/Early-2017/sampling-formalism.md`

[5]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`

[6]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.