Exploring functional reactive architectures for neural machines

Michael A. Bukatin

January 22, 2020 (draft)

Is it fruitful to consider functional reactive implementations of neural machines? This draft starts to explore this question.

Neural machines come in different flavors. The most familiar are various neural networks currently used in machine learning. I tend to mostly focus on a more expressive flavor, where single neurons can process streams of complex objects, such as images, trees, etc.[1]. Neural machines of this more expressive flavor can be used as a programming platform, with the property that programs can be continuously deformed. As neural networks, this flavor of neural machines has unusually powerful facilities for reflection and self-modification.

Because this is a novel programming platform, we are only aware of a small fraction of possible programming idioms and styles which can be fruitfully used here. The style of underlying implementation of these neural machines might matter quite a bit. On one hand, there are various efficiency considerations, but more importantly the way we implement these neural machines might inspire us to come up with *new programming idioms and styles* for these machines.

1 Interactive visual animations via composition of unit generators

A neural machine of this class generalizes digital audio synthesis via composition of unit generators. In the simplest case of monophonic sound, unit generators are transformers of streams of numbers, and the sound is synthesized by a contour composing those transformers.

So for neural machines processing streams of images, trees, and other complex objects, the natural next step is to consider transformers of those streams of complex objects as unit generators, and start exploring, for example, programming of interactive visual animations in this paradigm[2].

[1] We call this new class of neural machines **dataflow matrix machines**. 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`

[2] See Section 8 of the reference above for more details, and also Sections 1 and 2 of Michael Bukatin. *DMMs for VR and worldmaking; a modest proposal on effects and qualia.* December 2019. https://github.com/anhinga/2019-design-notes/blob/master/research-notes/research-notes-dec-2019.pdf

2 Pluralism of architectures and styles

Both traditional neural networks and interactive animations tend to assume that each of their underlying machines has a core running synchronously under its own central clock, but is often capable of handling asynchronous communications to some extent. So, generally speaking, we would be talking about a population of different asynchronously communicating neural machines running on the same computer or on a computer network, and different neural machines in that population can be implemented using different principles and programming styles.

3 Imperative programming, mutable style

The most traditional, down-to-earth style of implementing neural machines is an ordinary mutable style. The neural network typically works as something like $X_{t+1} = W(Y_t), Y_{t+1} = F(X_{t+1})$, and X and Y are overwritten on each time step t. This is quite efficient, especially if X and Y change completely on each step[3]. Of course, if changes in X and Y on each step are sparse, the overhead of rewriting them completely each time is nasty. And this architecture does not give us much help in inventing new programming paradigms for dataflow matrix machines.

4 Immutable streams

One can program with immutable data and treat X_t, Y_t as immutable streams sharing common parts[4]. As streams unfold in time, the unaccessible parts left in the past can be garbage collected (or we can maintain the references to the entire past, then the whole trace will be kept in memory, which is convenient, but costly). This architecture has good potential to give us some hints about new programming idioms for dataflow matrix machines.

5 Functional reactive programming with signals

When one programs in this style, with streams controlled by a global clock, it is tempting to explore programming with **signals**, that is functions $t \mapsto X_t$. One of my favorite examples of this approach is Yampa, a domain-specific language for the programming of hybrid (discrete and continuous time) systems embedded in Haskell[5].

[3] This is how traditional neural nets tend to be implemented, and this is how our interactive and non-interactive animations in Processing are implemented in `https://github.com/anhinga/fluid`.

[4] This is how our canonical Clojure implementation of dataflow matrix machines works: `https://github.com/jsa-aerial/DMM`.

[5] See `https://wiki.haskell.org/Yampa` and links in that page;
see also `https://wiki.haskell.org/Embedded_domain_specific_language`.

Given the emphasis on machines controlled by their global clock with the bulk of their computations being synchronized, it might make sense to port Yampa onto an eager language, making it a domain-specific language embedded in another functional language instead of Haskell. One could embed into an eager typed functional language, or one could even make a drastic step of embedding a Yampa-like domain-specific language into something like Clojure.

It might be extremely interesting to experiment in this general direction. My primary goal in such experiments would be to figure out new programming idioms and styles stemming from such experiments.

Efficiency questions are also of great interest here, as trade-off can be quite non-trivial (this style of programming might be costly, but also might allow for unexpected drastic optimizations).

6 What about Elm?

The current practice of many members of Haskell and Elixir communities is to use Elm for their interface needs. So whether Elm might be an appropriate platform for such experiments is an interesting question. And this question is really split into two parts, because there are two Elms: the original functional reactive Elm[6], and the new, modern Elm, which is not functional reactive[7]. I would like to ask whether old Elm might be a good platform for this, and whether modern Elm might still be a good platform for this.

It might be that the answer to both questions is negative, since Elm was always oriented first of all towards handling extremely well the interfaces with relatively rare asynchronous events and making sure that those are extremely non-latent and reactive, and computationally efficient. So this might turn to be a mismatch to our use case, where the bulk of computational power is spent generating a video from scratch in real time by neuromorphic computations (so this is not the case of playing a video created in advance).

It might be, for example, that neural computations should happen elsewhere, and in those cases where Elm use in the interfaces is desirable, one should simply communicate with Elm via something like Web sockets. I am going to ask the members of Elm community about this.

[6] The original Elm was introduced in Evan Czaplicki. *Elm: Concurrent FRP for Functional GUIs.* March 2012. `https://elm-lang.org/assets/papers/concurrent-frp.pdf`; see also Evan Czaplicki, Stephen Chong, *Asynchronous Functional Reactive Programming for GUIs*, June 2013. `http://people.seas.harvard.edu/~chong/abstracts/CzaplickiC13.html`

[7] The non-FRP version was introduced in Elm 0.17 in May 2016, see `https://elm-lang.org/news/farewell-to-frp`