A Technical Overview of the Neural Engineering Framework

Terrence C. Stewart

Centre for Theoretical Neuroscience technical report. Oct 29, 2012

This is an extended version of an article written for AISB Quarterly: The Newsletter of the Society for the Study of Artificial Intelligence and Simulation of Behaviour, Issue 135 (Autumn, 2012).

The Neural Engineering Framework (NEF) is a general methodology that allows you to build large-scale, biologically plausible, neural models of cognition [1]. In particular, it acts as a neural compiler: you specify the properties of the neurons, the values to be represented, and the functions to be computed, and it solves for the connection weights between components that will perform the desired functions. Importantly, this works not only for feed-forward computations, but recurrent connections as well, allowing for complex dynamical systems including integrators, oscillators, Kalman filters, and so on [2]. It also incorporates realistic local error-driven learning rules, allowing for online adaptation and optimization of responses [3]. The NEF has been used to model visual attention [4], inductive reasoning [5], reinforcement learning [6], and many other tasks. Recently, we used it to build Spaun, the world's largest functional brain model, using 2.5 million neurons to perform eight different cognitive tasks by interpreting visual input and producing hand-written output via a simulated 6-muscle arm [7,8]. Our open-source software Nengo was used for all of these, and is available at http://nengo.ca, along with tutorials, demos, and downloadable models.

Motivation

At first, it may seem odd to some that we would want to create biologically plausible models of cognition. Why bother? Why add the extra computational overhead of having realistically modelled neurons? Why would we want to put this extra constraint on our models, when it's hard enough to create models that produce realistic cognitive behaviour as it is?

For us, there are two major reasons. First, using realistic neurons allows us to better evaluate our theories. If we want to understand how brains work, then our models should not only produce the correct behaviour, but should also do so *in the same way* as real brains. That is, we should see comparable firing patterns and neural connectivity. We should see the same effects of neural degeneration, lesioning, deep brain stimulation, and even various drug treatments. We should see the same timing effects caused by the biophysical properties of neurons. Indeed, when we implemented a production system using the NEF, and constrained the model to have the properties of the various neuron types found in the brain regions involved, it produced the classic 50 millisecond cognitive cycle time *without parameter fitting* [9]. Furthermore, it produced a novel prediction that some types of productions take ~40 milliseconds, and others take ~70 milliseconds, which matches well to some unexplained behavioural data [10]. In other words, realistic neurons allow us to create new types of predictions, and allow us to set model parameters based on biology.

While this first reason allows you to do a better job of evaluating a model, the second reason is that this *suggests new types of algorithms*. When using the NEF, you do not get an exact implementation of whatever algorithm you specify. Instead, the neurons approximate that algorithm, and the accuracy of that approximation depends not only on the neural properties but also on the functions being computed. That is, instead of using any computations at all in your model, the NEF forces you to use the basic

operations that are available to neurons. This has allowed us to make strong claims about the classes of algorithms that could not be implemented in the human brain (given the constraints on timing, robustness, and numbers of neurons involved) [11]. In particular, finding a plausible method for implementing symbol-like cognitive reasoning, including symbol tree manipulation, has led us towards a relatively unexplored family of algorithms. As we discuss below, these alternate approaches turn out to be particularly useful for induction and pattern completion tasks which are difficult to explain with classical symbol structures.

Of course, once these new types of algorithms are identified, they can be explored without the use of realistic neurons. Indeed, our software package Nengo allows for the use of neurons to be turned on and off arbitrarily, either for the whole model or just for particular parts, even while a simulation is running. However, we have found some cases where the approximations incurred in the neural implementation actually improve the model (in terms of its match to human data) [12]. In other words, even when a standard mathematical implementation of a theory does not match empirical results, the NEF model of that theory may do a much better job.

Representation

The NEF uses distributed representations. In particular, it makes a sharp distinction between the *activity* of a group of neurons and the *value* being represented. The value being represented is usually thought of as a vector **x**. For example, you may use 100 neurons to represent a two-dimensional vector. Different vector values correspond to different patterns of activity across those neurons.

To map between **x** and neuron activity a, every neuron i has an *encoding* vector $\mathbf{e}_i$. This can be thought of as the *preferred direction vector* for that neuron: the vector for which that neuron will fire most strongly. This fits with the general neuroscience methodology of establishing *tuning curves* for neurons, where the activity of a neuron peaks for some stimulus or condition. In particular, the NEF makes the strong claim that the input current to a neuron is a linear function of the value being represented. If G is the neural non-linearity, α_i is a gain parameter, and b_i is the constant background bias current for the neuron, then we can compute neural activity given x as follows:

$$a_i = G(\alpha_i \mathbf{e}_i \cdot \mathbf{x} + b_i) \qquad (1)$$

Importantly, the function G can be any neural model, including simple rate-based sigmoidal neurons, spiking Leaky-Integrate-and-Fire neurons, or more complex biologically detailed models. The only requirement is that there be some mapping between input current and neuron activity, which can include complex spiking behaviour.

In standard connectionist models, each neuron responds to a distinct component of the input. We can think of this as a special case of Equation 1 where all of the $\mathbf{e}_i$ values are aligned along the standard basis vectors ([1,0] and [0,1] for the two-dimensional case). If a single non-spiking neuron is supposed to actually be a pool of spiking neurons, then this would be like having identical $\mathbf{e}_i$, α_i, and b_i values. However, real neurons have widely varying values (see Figure 1). As discussed in the next section, allowing these values to vary vastly increases the computational power of these neurons.

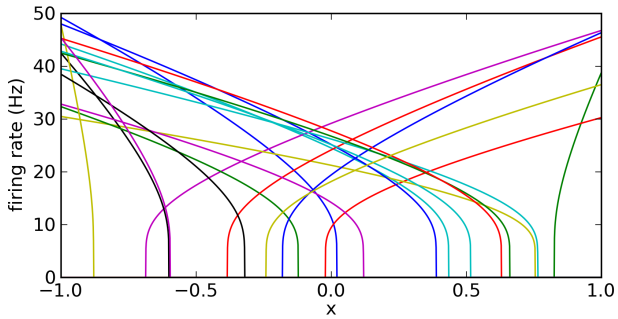

Figure 1: Average firing rates for 20 different leaky-integrate-and-fire neurons, computed using equation 2. α_i and b_i are randomly chosen to give a realistic range of responses. Neurons whose firing increases with **x** have $\mathbf{e}_i$=1, while the other neurons have $\mathbf{e}_i$=-1.

While Equation 1 allows us to convert a vector **x** into neural activity a_i, it is also important to go the other way around. That is, given some neural activity, what value is represented? The simplest method is to find a linear *decoder* $\mathbf{d}_i$. This is a set of weights that maps the activity back into an estimate of **x**, as follows:

$$\hat{\mathbf{x}} = \sum a_i \mathbf{d}_i \qquad (2)$$

Finding this set of decoding weights $\mathbf{d}_i$ is a least-squares minimization problem, as we want to find the set of weights that minimizes the difference between **x** and its estimate. This is a standard algebra problem, and can be solved as follows, where the sum is over a random sampling of the possible **x** values:

$$\mathbf{d} = \Gamma^{-1}\Upsilon \qquad \Gamma_{ij} = \sum_{\mathbf{x}} a_i a_j \qquad \Upsilon_j = \sum_{\mathbf{x}} a_j \mathbf{x} \qquad (3)$$

Having this decoder allows you to determine how accurately a group of neurons is representing some value, and provides a high-level interpretation of the spiking activity of the group. Importantly, it also turns out that these decoders also allow you to directly *solve for* the neural connection weights that will compute some desired transformation, as shown below.

Computation

So far, we have specified how to encode vectors into the distributed activity of a population of neurons, and how to interpret that activity back into a vector. However, to do anything useful, neurons need to be connected together. Consider the simple case where you have two neural populations (A and B) and you want to pass the information from one population to the next. That is, if you set A to represent the value 0.2, then the synaptic connections between A and B should cause the activity in B to also represent the value 0.2. In other words, you want the connections between A and B to compute the function **f**(**x**)=**x**.

Importantly, you can't do the simple approach of just connecting the first neuron in A to the first neuron in B, and so on for the other neurons. Not only might there be different numbers of neurons in the two groups, with different α_i and b_i values, but the neural non-linearity *G* makes this naïve approach highly inaccurate.

To create this connection in an accurate and robust manner, let us first assume that we have an intermediate group of perfectly ideal linear neurons, and we have one of these for each dimension

being represented. We then note that, thanks to Equation 2, **d** is exactly the set of connection weights needed to compute **x** given the activity in A. Furthermore, using the encoder values for group B ($\mathbf{e}_j$) as connection weights from this **x** representation is exactly what is needed to compute the dot product in Equation 1, which would in turn cause group B to represent **x**. This is shown in Figure 2a.

Of course, the real brain does not have these idealized intermediate neurons. However, they are completely unneeded. You can remove them and directly connect A to B using the connection weights found by *multiplying the two sets of weights* (see Figure 2b). That is, the optimal weights to pass the information from A to B are simply $\omega_{ij}=\mathbf{d}_i \cdot \mathbf{e}_j$.

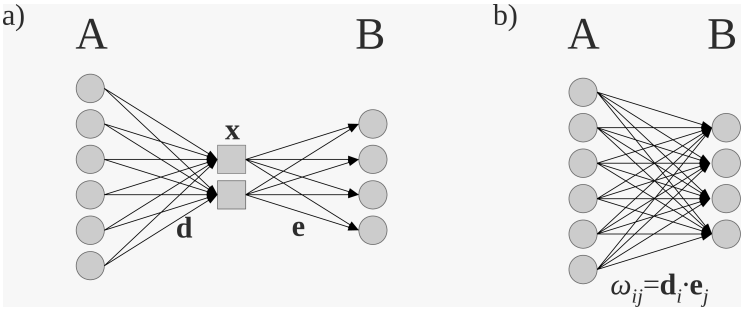

Figure 2: Computing with the NEF. Circles are any complex neuron model *G* with gain α_i and bias b_i. Squares are idealized perfectly linear components. Part a) shows how Equation 2 computes **x** from a_i using weights **d** and how Equation 1 combines **x** with **e** to compute the input current to the next layer of neurons. Part b) eliminates these idealized components, giving a realistic neuron model functionally identical to a).

This approach is not limited to simple functions like **f**(**x**)=**x**. By adjusting Equation 3 you can find decoding weights $\mathbf{d}^{\mathbf{f}(\mathbf{x})}$ to approximate any function **f**(**x**), as shown in Equation 4. Indeed, you can even use the input/output pairs used for training data in standard neural network approaches (the input patterns give the **x** values to sum over, and the output patterns give the corresponding **f**(**x**) values). Importantly, this means that nonlinear functions can be computed with a single layer of connections – no back-propagation of error is required. This includes not only the classic XOR problem, but also more complex functions such as multiplication, trigonometric functions, or even circular convolution (see *Symbol Processing*, below). That said, it cannot compute *any* function, and in general, the more non-linear and discontinuous the function is, the lower the accuracy. This accuracy is also affected by the neuron properties and the encoding method used: for example, if the $\mathbf{e}_i$ values are all aligned with the standard basis vectors, then nonlinear functions of multiple variables cannot be computed (this is the case for standard connectionist models, which is why they require multiple layers). This allows you to determine what neuron properties would be ideal for particular computations, which can then be used as neurological predictions [1].

$$\mathbf{d}^{\mathbf{f}(\mathbf{x})} = \Gamma^{-1}\Upsilon^{\mathbf{f}(\mathbf{x})} \qquad \Gamma_{ij} = \sum_{\mathbf{x}} a_i a_j \qquad \Upsilon_j^{\mathbf{f}(\mathbf{x})} = \sum_{\mathbf{x}} a_j \mathbf{f}(\mathbf{x}) \tag{4}$$

One way to think of this surprising feature (that distributed representations allow for complex functions to be computed in a single set of connections) is that the NEF is using the same trick seen in support vector machines: project your data into a high dimensional space. If you randomly choose $\mathbf{e}_i$ (as we generally do in our models, as it maps well to neural observations), then this is a *random projection*, which has been widely applied to machine learning problems. Furthermore, random variation in α_i and b_i is also required: the function **f**(**x**) being approximated ends up being built up out of linear sums of the tuning curves (Figure 1), so a wider variety of tuning curves leads to better function approximation.

The method of representation used in the NEF also allows you to *add* values by simply feeding two inputs into the same group of neurons. If you connect group A to group C with connection weights that compute **f**(**a**) and if you connect B to C with connection weights that compute **g**(**b**), then the neural group C will end up with the activity pattern that represents **f**(**a**)+**g**(**b**). This is a consequence of the linear representation given in Equation 1, and is vital for constructing larger networks.

A further advantage of this approach is that it is fast to simulate. Instead of multiplying the activity of group A by the full weight matrix, you multiply by the decoders and then multiply by the encoders to produce the input current to group B. This produces considerable savings of memory usage and simulation time, and was vital to our work connecting a neural model of a Kalman filter to real-time recordings from monkey motor cortex [13]. It should be noted, however, that this method produces connection weight matrices that freely mix positive and negative values, which are not seen in the real brain. We have shown elsewhere how to fix this problem by introducing inhibitory interneurons [14].

Dynamics

The previous two sections are sufficient to produce biologically realistic models capable of computing functions of the form $\mathbf{y}=\mathbf{f}(\mathbf{x})$. However, the NEF also provides a direct method for computing *dynamic* functions of the form $d\mathbf{x}/dt=\mathbf{A}(\mathbf{x})+\mathbf{B}(\mathbf{u})$, where **x** is the value being represented, **u** is some input, and **A** and **B** are arbitrary functions. A particularly useful special case of this equation is $d\mathbf{x}/dt=\mathbf{u}$, an *integrator*. This is a neural system that, if given no input ($\mathbf{u}=0$), will *maintain its current state* (since $d\mathbf{x}/dt=0$). Given a positive input, the stored value will increase, and given a negative input it will decrease. This sort of component appears in many models of working memory and in accumulator models of decision making.

While a full proof is outside the scope of this article (see [1]), building this system requires you to know the neurotransmitter time constant τ of a *recurrent* connection from the neurons in the group back to themselves. This time constant is a standard property of biological neuron models, and reflects how quickly the neurotransmitter released by a spike is reabsorbed. It varies widely across different types of neurons, from 2ms up to 200ms. Given this value, we have shown that you can compute the desired $d\mathbf{x}/dt=\mathbf{A}(\mathbf{x})+\mathbf{B}(\mathbf{u})$ function by using the above method to create a set of feedback connection weights that compute $\tau\mathbf{A}(\mathbf{x})+\mathbf{x}$, and setting the input connection to compute $\tau\mathbf{B}(\mathbf{u})$.

For the special case of the integrator ($d\mathbf{x}/dt=\mathbf{u}$), this results in a feedback connection computing exactly the same identity function $\mathbf{f}(\mathbf{x})=\mathbf{x}$ as in the simple model that passed information from group A to group B, only here the neurons are passing that information *back to themselves*. Much more complex models are possible, however, including oscillators ($d\mathbf{x}/dt=[\mathbf{x}_2,-\mathbf{x}_1]$), frequency-controlled oscillators ($d\mathbf{x}/dt=[\mathbf{x}_3\mathbf{x}_2,-\mathbf{x}_3\mathbf{x}_1]$), Kalman filters, and chaotic attractors. This can also be thought of as a kind of reservoir computing, where instead of randomly choosing the recurrent connections, you derive the connectivity that gives the desired dynamics.

Symbol Processing

While manipulating vectors is extremely powerful, many cognitive algorithms rely on manipulating *symbols* with some sort of syntactic structure. How can neurally realistic models possibly represent something like “Dogs chase cats” in such a way as to distinguish it from “Cats chase dogs”? How can we manipulate these representations in useful ways?

It turns out that there are a family of models that already exist for converting symbolic logic into vector

manipulations. These are known as Vector Symbolic Architectures [15], and all follow the approach of using high-dimensional vectors for each basic symbol, and then combining these vectors with various mathematical operations to produce new vectors that encode full symbol structures. Unlike ideal classic symbol systems, VSAs are lossy, in that as the symbol tree structure gets more complex, the accuracy of extracting the original vectors from that vector gradually decreases.

Furthermore, the vectors maintain similarity, so that if "pink" and "red" have similar vectors, then "pink square" and "red square" will also have similar vectors. This feature allows inductive reasoning over complex patterns. For example, our neural model of the Raven's Progressive Matrix task (a standard intelligence test where participants are given 8 visual patterns in a 3x3 grid and are asked to determine what pattern should be placed in the missing square) works by forming the vector representation of each pattern and computing the average transformation that that will take one pattern to the next [5].

As a simple example of this approach, you can create high-dimensional (~500 dimensions for adult-level vocabularies) unit vectors for each basic symbol (DOG, CAT, CHASE, SUBJECT, OBJECT, VERB, etc.). These can be randomly chosen, or chosen so as to reflect standard similarity measures. To create a symbol structure, you need two operations: addition (+) and circular convolution (⊗). The sentence "Dogs chase cats" would then be S=DOG⊗SUBJECT+CHASE⊗VERB+CAT⊗OBJECT). Given this sentence, you can extract a particular component by computing $S \otimes SUBJECT^{-1} \approx DOG$, where the inverse operation is a simple reordering of the elements in the vector. Addition is easy to implement in neurons, as noted above. Interestingly, while circular convolution seems like a complicated operation, you can break it down into a linear transformation, a large number of pairwise multiplications, and another linear transformation. All of these operations are accurately approximated by the NEF methods.

Spaun

The ability to perform symbol-like manipulations using vectors allows you to build very large-scale cognitive models. Our largest model to date is Spaun, a 2.5 million spiking neuron model with a vision system (formed by implementing a Restricted Boltzmann Machine Deep Belief Network with the NEF), a single 6-muscle 3-joint arm for output, and a selective routing system (analogous to a production system) implemented in spiking neurons comprising the cortex (for working memory storage), the basal ganglia (for action selection), and the thalamus (for selectively routing information between cortical areas) [7]. Various other cortical areas are also modeled, allowing for transformations between visual, conceptual, and motor spaces, inductive pattern finding, and list memory. The model is capable of performing eight different psychological tasks, including recognizing hand written digits, memorizing digit lists and recalling particular items, pattern completion, reinforcement learning, and mental addition. No changes to the model are made between tasks: instead, a visual input is provided telling the model which task to perform next. We are aware of no other realistic neural model with this combination of flexibility and biological realism.

Nengo

While the math provided in this article is sufficient for implementing the Neural Engineering Framework on your own, we have also developed *Nengo*, an open-source cross-platform Java application which implements the NEF. It is meant as both a teaching tool (with hands-on classroom demos) and a research tool (all of our large-scale models are built with it, including Spaun). Neural groups can be created through a drag-and-drop interface (Figure 3, left) or Python scripting. The functions to approximate are similarly specified, with Nengo automatically computing the optimized

connection weights. Also included is a visualization interface for viewing and interacting with running models, including support for simulated environments and physical robots (Figure 3, right). The software, extensive documentation, and various tutorials are available at <http://nengo.ca>.

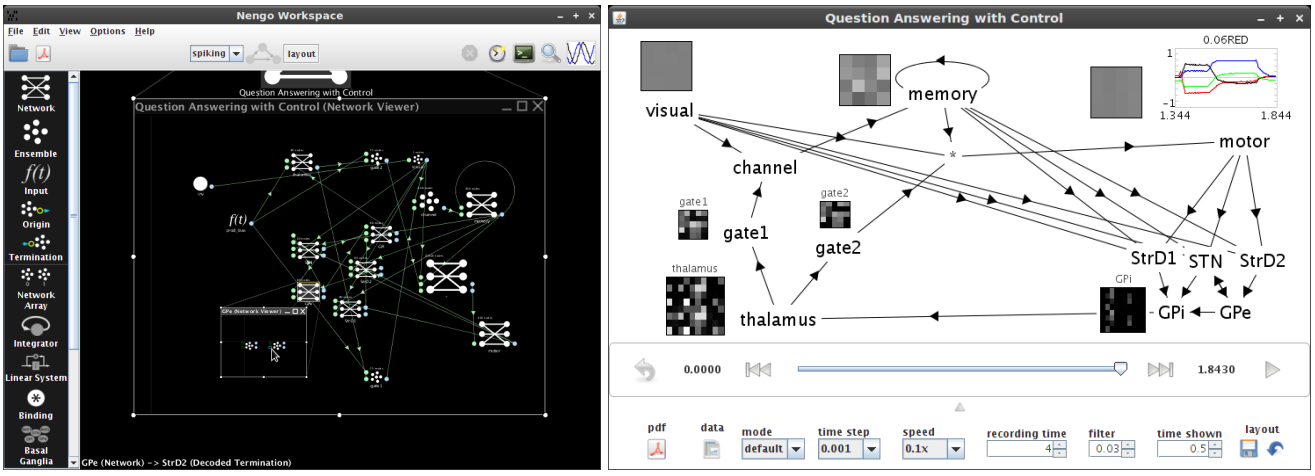

Figure 3: The Nengo software package. Left: the graphical interface for constructing models. Right: the interactive display showing a running model, allowing for real-time interaction.

The purpose of Nengo is to provide you easy access to the biologically realistic modelling techniques described in this article. The idea is that you should be able to take a hypothesized algorithm and quickly create a biologically realistic neural model that implements that algorithm. Given that model, you can make novel predictions as to the neural organization required, and the patterns of spiking activity that should be observed if your hypothesis is correct. Furthermore, you get direct behavioural predictions in terms of both timing and the effects of the neural approximation. This opens up completely new avenues of research, bridging the gap between brains and behaviour.

[1] Eliasmith, C., & Anderson, C. H. (2003). Neural engineering: Computation, representation and dynamics in neurobiological systems. Cambridge, MA: MIT Press.
[2] Eliasmith, C. (2005). A unified approach to building and controlling spiking attractor networks. Neural computation. 7, 1276-1314.
[3] MacNeil, D., & Eliasmith C. (2011). Fine-tuning and the stability of recurrent neural networks. PLoS ONE. 6(9).
[4] Bobier, B., Stewart T. C., & Eliasmith C. (2011). The attentional routing circuit: receptive field modulation through nonlinear dendritic interactions. Cognitive and Systems Neuroscience Poster.
[5] Rasmussen, D., & Eliasmith, C. (2011). A neural model of rule generation in inductive reasoning. TopiCS 3, 140-153.
[6] Stewart, T.C., Bekolay, T., & Eliasmith, C. (2012). Learning to select actions with spiking neurons in the basal ganglia. Frontiers in Decision Neuroscience, 6.
[7] Stewart, T., Choo, F-X, & Eliasmith, C. (2012). Spaun: A perception-cognition-action model using spiking neurons. Proceedings of the 34th Annual Conference of the Cognitive Science Society.
[8] Eliasmith, C. (2013). How to build a brain: A neural architecture for biological cognition. New York: Oxford University Press.
[9] Stewart, T.C., Choo, F-X., & Eliasmith, C. (2010). Dynamic behaviour of a spiking model of action selection in the basal ganglia. In Proceedings of the 10th International Conference on Cognitive Modeling, 235–240.
[10] Gunzelmann, G., Moore, R., Salvucci, D., & Gluck, K. (2011). Sleep loss and driver performance:

Quantitative predictions with zero free parameters. Cognitive Systems Research, 12(2), 154-163.
[11] Stewart, T., & Eliasmith C. (2012). Compositionality and biologically plausible models. Oxford Handbook of Compositionality.
[12] Choo, F-X & Eliasmith, C. (2010). A spiking neuron model of serial-order recall. In Proceedings of the 32nd Annual Conference of the Cognitive Science Society.
[13] Dethier, J., Nuyujukian P., Eliasmith C., Stewart T.C., Elassaad S. A., Shenoy K., & Boahen, K. (2011). A brain-machine interface operating with a real-time spiking neural network control algorithm. Neural Information Processing Systems (NIPS) 24.
[14] Parisien, C., Anderson C. H., & Eliasmith C. (2008). Solving the problem of negative synaptic weights in cortical models. Neural Computation. 20, 1473-1494.
[15] Gayler, R. (2003). Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience. ICCS/ASCS International Conference on Cognitive Science, Sydney, Australia: University of New South Wales. 133-138.