Learning nonlinear functions on vectors: examples and predictions

Trevor Bekolay

Centre for Theoretical Neuroscience technical report. Dec 17, 2010

Abstract

One of the underlying assumptions of the Neural Engineering Framework, and of most of theoretical neuroscience, is that neurons in the brain perform functions on signals. Models of brain systems make explicit the functions that a modeller hypothesizes are being performed in the brain; the Neural Engineering Framework defines an analytical method of determining connection weight matrices between populations to perform those functions in a biologically plausible manner.

With the recent implementation of general error-modulated plasticity rules in Nengo, it is now possible to start with a random connection weight matrix and learn a weight matrix that will perform an arbitrary function. This technical report confirms that this is true by showing results of learning several non-linear functions performed on vectors of various dimensionality. It also discusses trends seen in the data, and makes predictions about what we might expect when trying to learn functions on very high-dimensional signals.

1 Introduction

In neuroscience, the term "function" is used in many different contexts. In experimental neuroscience, and in large part to the layperson, the function of the brain or a subsystem of the brain refers to the verb that best describes what it does. The basal ganglia perform action selection, the prefrontal cortex plans and performs abstract thinking, and so on. In theoretical neuroscience, functions are the mathematical operations that can be combined to bring about that higher-level verbal description. While experimenters and philosophers can describe "brain function" without making assertions about the nature of what the brain does, theoretical neuroscience necessarily treats the brain as a computational device; it takes in input from the environment, transforms that input using mathematical functions, and produces some output in the form of internal changes or the activation of motor systems.

The goal of the theoretician is, then, to identify which functions the brain is performing, and how neurons can be employed to perform those functions. Any well written paper describing a model makes explicit both of these elements. Papers on learning in neural systems, however, do not. Typically, a learning paper will attempt to match neural data, showing that a learning rule is able to elicit from a population of simulated neurons a firing pattern similar to that shown in the neural data. While it seems logical to then attempt to identify the mathematical function that the learned connection weight matrix is performing, few papers do so. In part, this is due to an ignorance of tools available that will allow the investigation of the function a connection weight matrix is performing.

One such tool is Eliasmith & Anderson's Neural Engineering Framework (NEF). The NEF defines a method of encoding and decoding signals in spiking neurons and an analytical method of determining connection weight matrices to perform transformations on encoded signals. The NEF can be thought of as a framework that models the *result* of learning; while it may be true that neuronal connectivity is in some cases genetic, one should make the assumption that the connection weight matrices found analytically by

the NEF would be learned in the biological brain. While this obviates the need for learning in the NEF, it does not preclude it. Quite the opposite; using learning in models built with the NEF allows a modeller to analyze the functions that a learning rule can and does learn, and can fine-tune models created with the NEF's analytical connection weight matrices.

This paper describes the analysis of a local error-modulated learning rule. It follows the structure of describing networks described in Eliasmith & Anderson's book Neural Engineering (2003). In section 2, the network in which the learning rule will be analyzed is presented, and the particular learning rule is described and related to other similar learning rules. In section 3, additional constraints are discussed, and the control networks that will be used for comparison are presented. The mathematical functions that will be learned are enumerated. In section 4, the implementation of the described networks is discussed, and results from simulations are presented. In section 5, the results are discussed and predictions are made that will direct future work.

2 System description

Unlike models that aim to replicate the function of a particular biological system, the networks created for the report are idealized systems created specifically for the purpose of testing a learning rule. The neurons used are the Nengo defaults: leaky integrate-and-fire neurons with typical parameters and tuning curves.

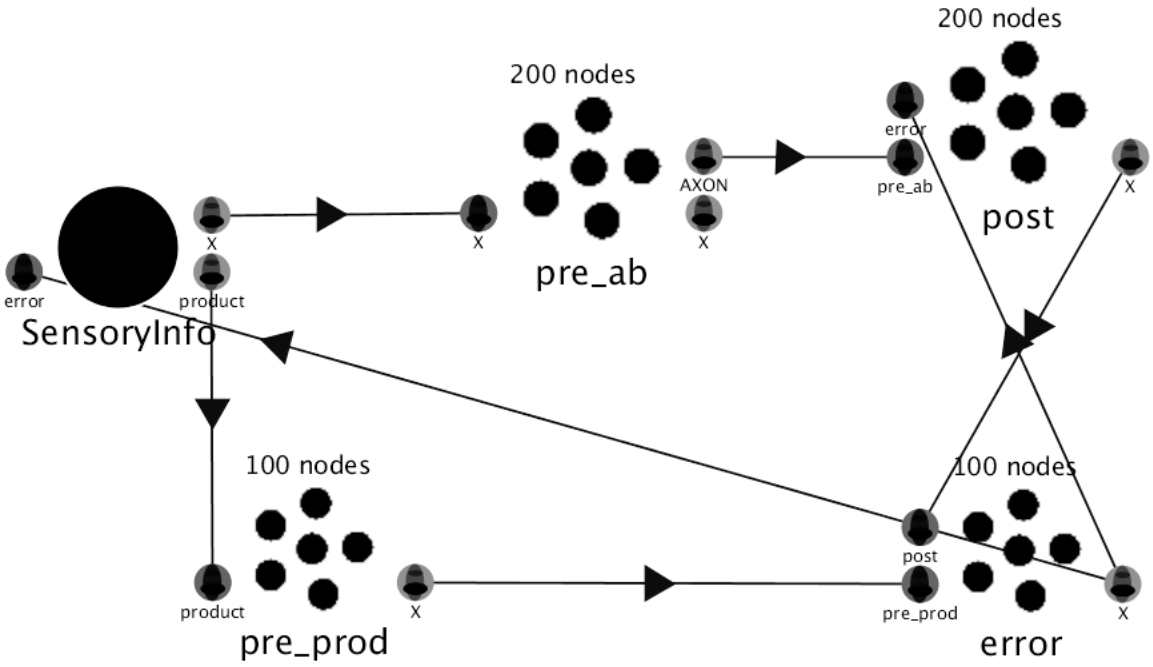

Figure 1: Network for learning multiplication.

Figure 1 is a network designed for learning the multiplication of two numbers. There are five interconnected nodes.

- `SensoryInfo` is an `nef.simplenode` that provides the input to the network. The 'X' origin emits a two-dimensional signal: the two values to be multiplied. These values change by a small amount on each timestep (determined randomly from a Gaussian distribution with mean 0 and variance 0.05) – i.e., the values are Gaussian white noise. The 'product' origin emits the actual value produced by

multiplying those two signals. The 'error' termination accepts input from the `error` population for the purpose of determining system performance (discussed in section 4).

- `pre_ab` is an `NEFEnsemble` made up of 200 LIF neurons. It receives as input the 'X' signal generated by `SensoryInfo`. It represents this value, and emits spikes accordingly.
- `post` is an `NEFEnsemble` made up of 200 LIF neurons. It receives as input the spikes emitted by the `pre_ab` population, which encodes the two-dimensional signal emitted by `SensoryInfo`. Those spikes are available to the 'pre_ab' termination; this termination is plastic, and has an error-modulated learning rule applied to it. The connection weight matrix is initially random, and is learned over the course of the simulation. The modulatory 'error' termination accepts input from the `error` population, which is used by the learning rule modifying the 'pre_ab' termination. The signal encoded by `post` is projected to the `error` population.
- `pre_prod` is an `NEFEnsemble` made up of 100 LIF neurons. It receives as input the 'product' signal emitted by `SensoryInfo`. It encodes this signal in spikes and emits those spikes to the `error` population.
- `error` is an `NEFEnsemble` made up of 100 LIF neurons. It receives as input the precomputed product from the `pre_prod` population, and the value represented by the `post` population. It computes the difference between these two values, and emits that difference to both the `post` population, to direct learning, and to the `SensoryInfo` node, to track system performance.

All of the other networks that learn transformations are similar, differing primarily in the number of dimensions represented by various populations and then number of neurons used to represent those signals. As a second example, figure 2 shows the network used to learn circular convolution on two 3-dimensional signals.

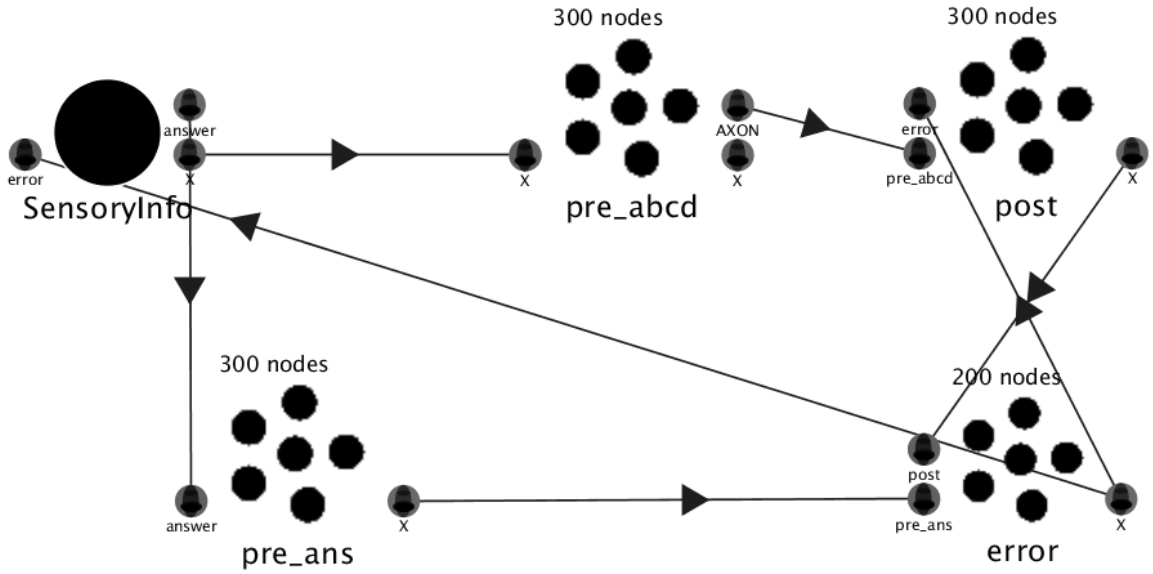

Figure 2: Network for learning 3-dimensional circular convolution.

Ostensibly, these networks are nearly identical. The visually discernible differences are the number of neurons in each population. Because this network deals with three- and six-dimensional signals, the number of neurons must increase to represent these signals well. The only other differences are the dimensionality of the represented signals, and the answer that `SensoryInfo` computes.

2.1 Learning rule

The learning rule used in the experiments described in this report is MacNeil & Eliasmith's general error-based learning rule (2010, preprint). This rule is local (Hebbian), error-modulated, and has already been incorporated into the NEF. The rule, in delta-omega notation, is

$$\Delta\omega_{ij} = \kappa\alpha_j\tilde{\phi}_j\mathbf{e}a_i$$

where

- κ is the learning rate (always 1×10^{-7} in the simulations described in this report),
- α_j is the gain of the neuron in the post population,
- $\tilde{\phi}_j$ is the encoding vector of the neuron in the post population,
- $\mathbf{e}$ is an n-dimensional error signal, and
- a_i is the activity of the neuron in the pre population. In the NEF, this is the pre neuron's spike train filtered with a post-synaptic current (PSC) curve.

This rule belongs to the class of error-modulated learning rules. These rules are typically motivated by the finding that dopamine from the substantia nigra pars compacta and the ventral tegmental area appear to encode something similar to reward prediction error, and related studies that show that dopamine modulates learning in the striatum and parts of cortex.

The most important difference between this rule and other rules in this class is that the error signal is considered to be n-dimensional. For the functions learned in this paper, using a multidimensional error signal makes network construction much easier. While it is possible to solve any of these problems with multiple scalar error signals operating on separate connection weight matrices (or separate post populations), we predict that using a multidimensional error signal increases the amount of information that a network can transmit post-learning, and will scale better in the number of neurons for learning functions on high-dimensional vectors. This should be especially true if the dimensions in the output are related, or rely on the same inputs.

3 Design specification

In creating the networks that will learn the example functions, there are no specific neural constraints; however, care was taken to not tweak neuron parameters and network structure such that learning was made easier. Nengo defaults are used whenever possible.

To get an idea of baseline performance and to ensure that controllable parameters (number of neurons in each population, etc.) are well chosen, a control network without learning was created for each learning example. That is, for each function that is learned, a corresponding network was created using connection weight matrices analytically determined by the NEF.

In most cases this only required a small change (two or three lines) in the Python script file that generates the learning networks. The exceptions to this are the networks that learn circular convolution, as will be discussed in the subsequent sections.

3.1 Example functions

Networks to learn five non-linear functions were created. The functions differed in the dimensionality of the input and output, among other factors. In this section, we make explicit the five functions and the properties that make them interesting targets for testing the performance of the error-modulated learning rule.

3.1.1 Multiplying two numbers

$$f(x_1, x_2) = x_1 \times x_2$$

While not the simplest non-linear function, multiplying two numbers is the simplest function that was learned, in terms of the dimensionality of the input and output. The input is two-dimensional, and the output is one-dimensional. Creation of the learning and non-learning versions of this network are straightforward, and was discussed in section 2. Being the simplest non-linear function, this network requires relatively few neurons, and quick prototyping can be done to evaluate learning rules and other tunable parameters. It also serves as evidence that a particular learning rule can indeed learn a connection weight matrix that computes a non-linear function.

3.1.2 Combining two products

$$f(x_1, x_2, x_3, x_4) = x_1 \times x_2 + x_3 \times x_4$$

This function combines two separate, unrelated products. The input is four-dimensional, and the output is one-dimensional. An important feature of this function is that it is a linear combination of non-linear functions. Despite that, the error signal is scalar, matching the dimensionality of the output of this function; so, while the network is learning two non-linearities and the linear combination of them, the only reinforcement it receives is how incorrect it is at the current time.

3.1.3 Three separate products

$$f(x_1, x_2, x_3) = [x_1 \times x_2, x_1 \times x_3, x_2 \times x_3]$$

This function computes three separate, but related, products. The input is three-dimensional, and the output is three-dimensional. This is the first attempt at learning a function that produces multidimensional output. While it is not a minimal example of this, it is sufficiently simple and well understood that a learning rule that can learn the above examples should be able to learn this example.

This particular function may be useful in the future, to motivate the use of multidimensional error signals, compared to an approach that splits this problem into three separate subnetworks that each learn one of the three products.

3.1.4 Two-dimensional circular convolution

$$f(x_1, x_2, x_3, x_4) = [x_1, x_2] \otimes [x_3, x_4]$$

Circular convolution is a non-linear function used in many neural simulations – especially those created with the NEF – to bind together two vectors of the same length. Circular convolution is typically computed by multiplying the two vectors in the frequency domain; that is, the discrete Fourier transform of the two input vectors are computed, those vectors are multiplied, and then are transformed back to the signal

domain with the inverse discrete Fourier transform. Because of this, performing circular convolution in neural circuits is more complicated than simple products, and the neural implementation using the NEF requires an additional subnetwork to compute the Fourier transform.

However, the learning version of this network does not use this subnetwork, and attempts to learn this function in a two-layer network. The input is four-dimensional, and the output is two-dimensional. No additional modifications are made to the learning network (as mentioned in section 2), so this function can give support to the hypothesis that the learning rule can learn arbitrary transformations on n-dimensional vectors.

3.1.5 Three-dimensional circular convolution

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = [x_1, x_2, x_3] \otimes [x_4, x_5, x_6]$$

Adding a third dimension to the circular convolution network described above gives insight into how additional dimensions affects the nature of the learning in the network. It helps answer questions such as: how much longer does it take to learn a transformation with additional dimensions? How many additional neurons are needed in each population, if any? How is the accuracy of the learned transformation affected?

4 Implementation

Implementation of the example functions described in section 3.1 was straightforward, following the example shown in section 2. There were additional complication for the control versions of the circular convolution networks; however, the `nef.convolution` module, implemented by Terry Stewart, made this process much simpler.

Implementing the learning rule itself presented a number of separate challenges; those efforts have been discussed in a previous technical report.[1]

4.1 Evaluation

The most difficult challenge of implementation is in how to evaluate the performance of the networks. The ideal evaluation method would be to choose large number of evaluation points distributed over the range of possible input vectors, and sum up the amount of error in the value represented by the post population across those evaluation points. However, this approach is computationally costly and difficult to implement.

Instead, the existing infrastructure (i.e. the `SensoryInfo` node and the `error` population) are leveraged to get a general sense of the performance of the network. Because these elements are also present in the control networks, we can easily compare the learned networks to the analytically determined ones.

The specific evaluation method used in these simulation is as follows:

- A trail run is broken up into equal parts training and testing.
- For a defined length of time (between 2.5 and 5 seconds), the system is tested. For the same amount of time, the network learns. These two phases are repeated until the trail run is complete.
- When the testing phase begins, learning is stopped in the `post` population (using `post.setLearning(false)`). Until the end of the testing phase, the absolute value of the filtered output from the `error` population

[1] "Using and extending plasticity rules in Nengo", Bekolay, 2010

is accumulated. When the testing phase finishes, the amount of accumulated error is recorded and reset to 0.

- When the training phase beings, learning starts in the `post` population (using `post.setLearning(true)`).
- The input signal is Gaussian white noise at all times. The error is always calculated as the difference between the value represented by the `post` population and the actual answer computed by `SensoryInfo`.

There are many sources of variability in this evaluation scheme.

- The input signal varies randomly. For both the learning and control examples, the amount of variation in the signal during the testing phase can be quite large, causing unusually large accumulated error values at random points in the simulation. This is mitigated by doing multiple trials for each network; however, due to time constraints, only 10 trails of each network were done for the purpose of this report.
- The neurons used in each trial of each network are created with random parameters. Even in the control networks, while the amount of accumulated error is relatively similar across one entire trail, there is variation across trials. This could be mitigated by creating networks with more neurons than are necessary to perform the computations, but because of the additional computational demand of plasticity, populations were made to be as small as possible while still being able to reasonably compute the example functions.

Despite this, as will be shown below, the results are statistically significant.

4.2 Results

In the five functions that were attempted, the error-modulated learning rule was able to learn a connection weight matrix that performed the function as well as or almost as well as the control network. In the included plots, the blue line represents the accumulated error averaged over 10 runs of the learned network, and the red line represents the accumulated error averaged over 10 runs of the control network. The dashed grey lines represent bootstrapped 95% confidence intervals. The accumulated error on the y-axis is in arbitrary units, and the value of the accumulated error in one example should not be compared to the value of the accumulated error in another example, due to different length of testing phases and error in all dimensions being summed together. The time indicated on the x-axis is the amount of time that the network has been allowed to learn. At 0 seconds, the pre and post populations are connected with a completely random connection weight matrix.

As shown in figure 3, the learning rule was able to learn a connection weight matrix that multiplied two numbers as well as the control network after approximately 30 simulated seconds.

As shown in figure 4, the learning rule was able to learn a connection weight matrix that combined two products as well as the control network after approximately 70 simulated seconds, though it should be noted that the network comes very close at approximately 40 seconds. That this example takes longer to learn than simple multiplication is unsurprising, though the increase in dimensionality and complexity does not have as large an effect as one might hypothesize.

As shown in figure 5, the learning rule was not able to learn a connection weight matrix that computed three separate products as well as the control network after running for 100 seconds. However, the rule did stabilize at around 45 seconds and had a consistently low value for accumulated error that was only slightly higher than the control. It is feasible that with changes to the number of neurons in various populations, the learning rule could learn to perform the function as well as the control network; however, it is also possible that there is some particular feature about this function (e.g. that the three dimensions of the output are related) that makes it hard or impossible to learn with this learning rule.

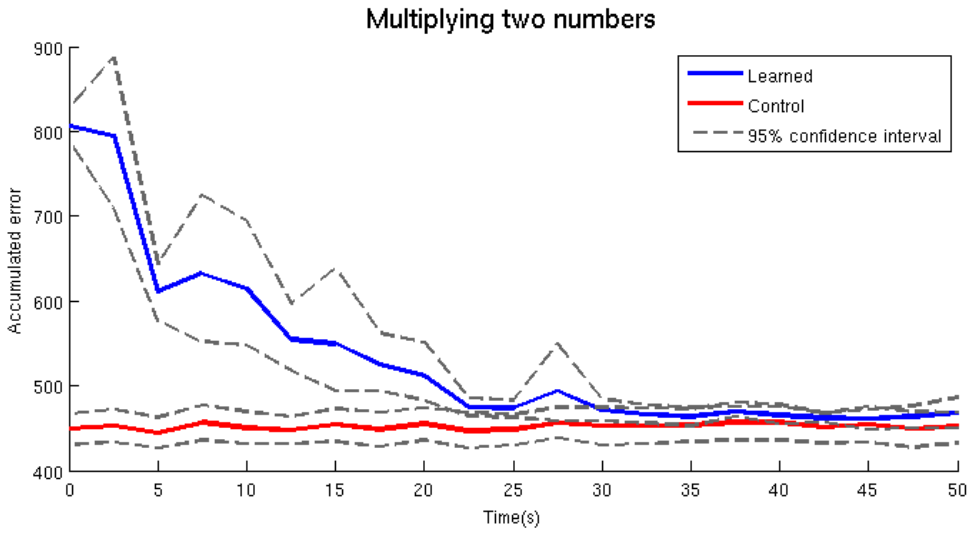

Figure 3: Accumulated error for the function $f(x_1, x_2) = x_1 \times x_2$.

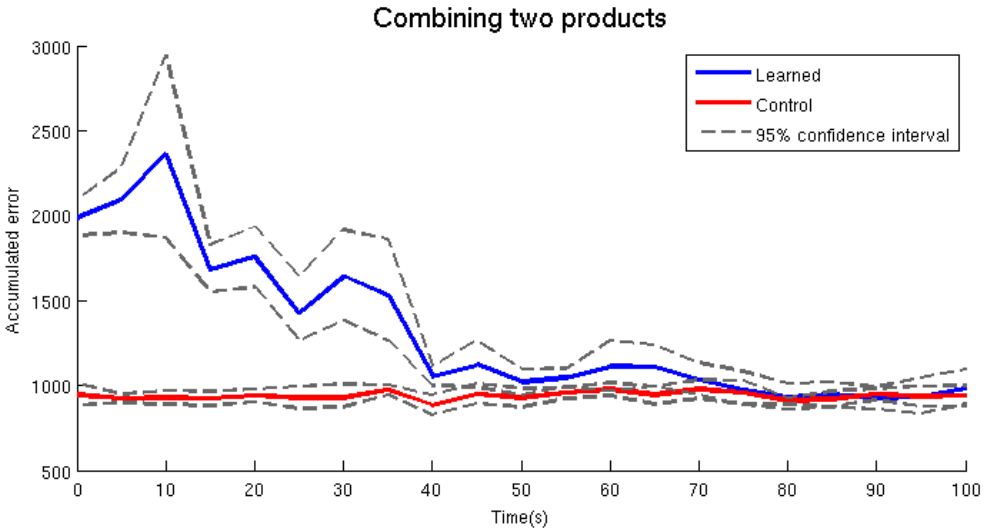

Figure 4: Accumulated error for the function $f(x_1, x_2, x_3, x_4) = x_1 \times x_2 + x_3 \times x_4$.

As shown in figure 6, the learning rule was able to learn a connection weight matrix that convolved two two-dimensional vectors as well as the control network after approximately 65 simulated seconds. There are some interesting features of the learned network; specifically, the spikes in variability at 90 and 120 seconds. Further investigation into these spikes may reveal something about either the learning rule, circular convolution in neurons, or the difference between the control network and the learned network.

As shown in figure 7, the learning rule was not able to learn a connection weight matrix that convolved two three-dimensional vectors as well as the control network after 400 seconds of simulation time. However, the rule largely stabilized at around 200 seconds and had a consistently low value for accumulated error that was only slightly higher than the control. This example, in particular, warrants further investigation into the reason behind the discrepancy in accumulated error, though one should note the scale of the accumulated error, which begins at 3000.

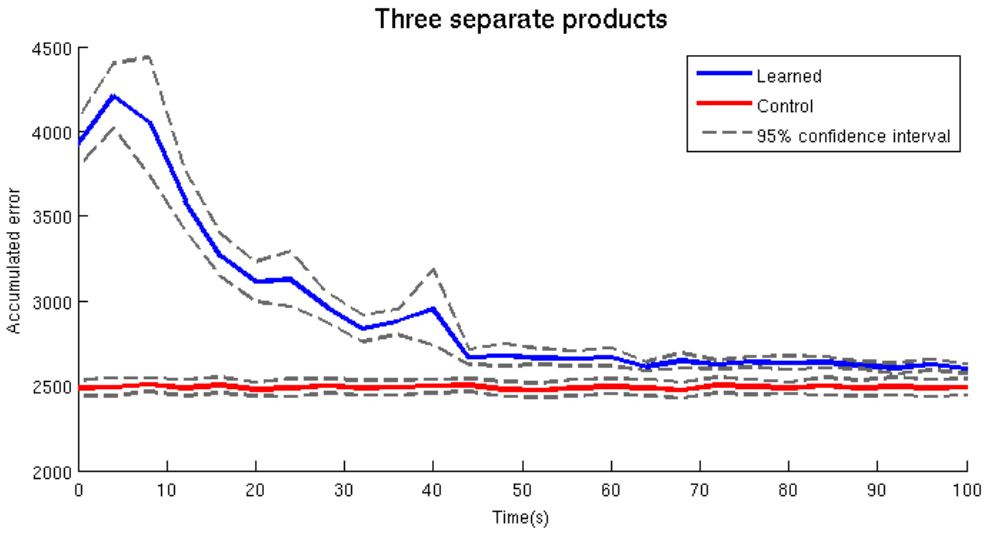

Figure 5: Accumulated error for the function $f(x_1, x_2, x_3) = [x_1 \times x_2, x_1 \times x_3, x_2 \times x_3]$.

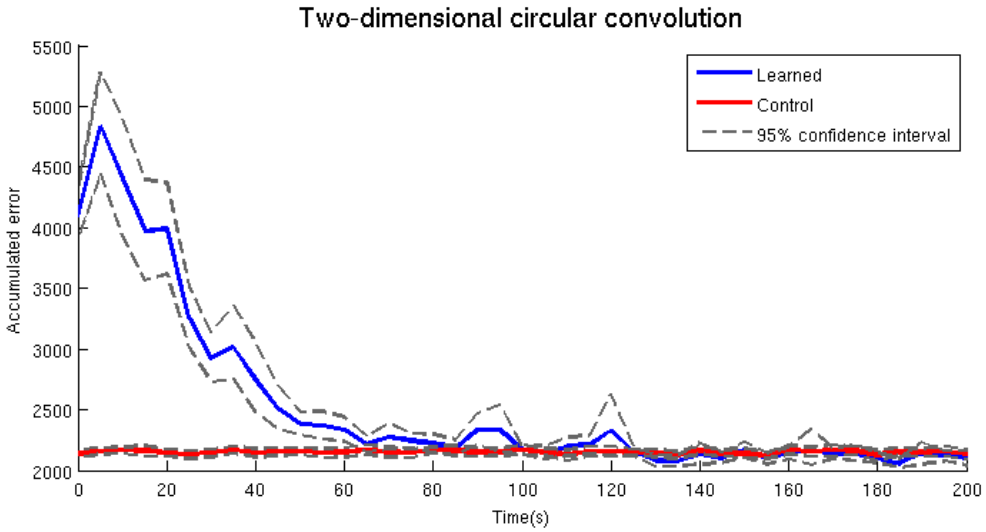

Figure 6: Accumulated error for the function $f(x_1, x_2, x_3, x_4) = [x_1, x_2] \otimes [x_3, x_4]$.

5 Discussion

The most important finding of this report, which may be overlooked if one focuses on certain examples, is that the error-based learning rule decreases the amount of error significantly in all cases, and in the majority of cases, is able to perform non-linear functions on multidimensional signals as well as the control network whose connection weight matrices are analytically determined. A particular example that demonstrates the power of the rule is the example of two-dimensional convolution, in which the learned network is able to learn the function with a two-layer network of 500 neurons in total, while the control network does no better with a three-layer network of 1300 neurons.

An interesting feature of all but one of the examples is that the accumulated error in the learned network

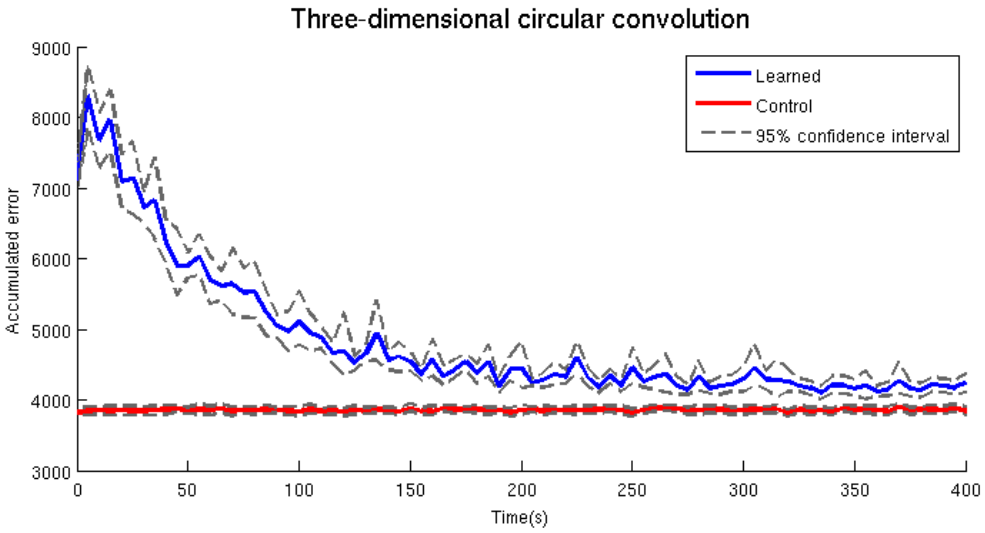

Figure 7: Accumulated error for the function $f(x_1, x_2, x_3, x_4, x_5, x_6) = [x_1, x_2, x_3] \otimes [x_4, x_5, x_6]$.

grows slightly before dropping somewhat consistently. One might make the assumption that the random connection weight matrix that the network begins with is a "worst case," and that the learning rule would decrease error monotonically. However, these results show that the first several seconds of learning can be detrimental to the performance of the system. This may occur for a number of reasons.

- The randomly connected weight matrix, in most cases, makes the post population insensitive to any input from the pre population, meaning that the post population is representing values near 0 most of the time. If the function being computed is also biased to being close to 0 most of the time, then a random connection weight matrix would have lower accumulated error than a weight matrix that has only explored a small portion of the domain of a function.
- The learning rate may be too high, biasing the network to the values it experiences early on.
- The learning rate may be too low, making the learning rule slow to generalize particular examples to the overall solution.
- This is desired or expected behaviour.

While the learning rule appears to have little difficulty with multidimensional functions in general (e.g. the two-dimensional convolution example), the two examples in which the output was three-dimensional were the two examples that did not match the performance of the control network. This points to a need for a more disciplined approach to investigating the effects of increasing the dimensionality of the input and output of a function. A good direction for future work is to focus on one particular example and investigate the effect on the efficacy and speed of learning when the following parameters are modified:

- dimensionality of the input,
- dimensionality of the output,
- number of neurons in the pre population,
- number of neurons in the post population,
- radius of the pre population,
- radius of the post population, and
- learning rate (κ) of the learning rule.