Spike-based learning of transfer functions with the SpiNNaker neuromimetic simulator

Sergio Davies*†, Terry Stewart‡, Chris Eliasmith‡ and Steve Furber†
*Corresponding author: daviess@cs.man.ac.uk
†Advanced Processor Technologies Group, School of Computer Science, The University of Manchester, United Kingdom
‡Centre for Theoretical Neuroscience, University of Waterloo, Ontario, Canada

***Abstract*—Recent papers have shown the possibility to implement large scale neural network models that perform complex algorithms in a biologically realistic way. However such models have been simulated on architectures not able to perform real-time simulations. In previous work we presented the possibility to simulate simple models in real-time on the SpiNNaker neuromimetic architecture. However such models were "static": the algorithm performed was defined at design-time. In this paper we present a novel learning rule, implemented on the SpiNNaker system, that enables models designed with the Neural Engineering Framework (NEF) to learn the function to perform using a supervised framework. We show that the proposed learning rule, belonging to the Prescribed Error Sensitivity (PES) class, is able to learn effectively both linear and non-linear functions.**

I. Introduction

In recent papers (e.g. [1]) some large scale neural network models have been proposed, which focus on the functional connection between computational neural blocks. Each of these blocks comprises multiple populations of neurons, each performing a simpler function, or algorithm.

The functions and algorithms performed by these populations and their interconnections are designed on the basis of a mathematical framework, called the Neural Engineering Framework (NEF) [2]. These algorithms are grouped in large scale architectures to perform complex tasks (e.g. [1]) that resemble biological behaviour.

Simulations of such systems may be performed on general-purpose computational substrates, such as a standard personal computer [3], or on dedicated neuromorphic or neuromimetic hardware [4]. The computation may, or may not, be performed within a real-time constraint, depending on the type of hardware in use for the simulation [5].

A spiking neural network simulator for the NEF has been implemented using a software (named *"Nengo"*) running on a standard computer desktop. This simulator has the limitation that is able to run slower than real-time, and the bigger the network, the slower the simulation runs, therefore with long waiting before results are presented. The idea of performing real time simulations has been introduced with the use of the SpiNNaker architeture, which is presented in the next section.

Earlier work [6] presented the possibility to perform real-time simulation of "static" functional connections on the SpiNNaker system [7]: such connections perform a single, immutable function defined at system-design time.

However, synaptic plasticity is known to happen in biology: the human brain is subject to continuous evolution in the properties of the interconnection pattern. Various learning rules have been proposed [8] [9], implemented [10] and tested [11] [12] on simulators; some of these were extracted from biological observations [13] [14].

Currently, the majority of the learning algorithms [15] for the third generation of neural networks [16] are based on the STDP algorithm [17], or modified version of it (e.g.[18], [19]). In fact, although this algorithm (extracted from biological observations) intrinsically requires an unsupervised learning framework, it has been adapted also for use with supervised frameworks (e.g. [20] and [21]) and reinforcement learning frameworks (e.g. [22] and [23]).

This work focuses on the implementation of a novel learning rule on the SpiNNaker neuromimetic architecture. This rule is based on synaptic weight modification applied in a supervised learning paradigm using a modulatory signal. The set of trainee synapses benefits from the presence of the learning (modulatory) signal to represent the desired transfer appropriately function between populations of neurons. We show that there is no limit on the type of function that may be taught using this learning rule: in fact the network is shown to be able to learn linear as well as non-linear functions.

II. Introduction to SpiNNaker

SpiNNaker (acronym of Spiking Neural Network Architecture, see fig. 1) is a hardware-based real-time universal neural network simulator following an event-driven computational approach [24]. This project involved the design of a chip and the development of dedicated software to simulate neural networks [25]. This system tries to mimic the features of biological neural networks in various ways:

- **Native parallelism**: each neuron is a primitive computational element within a massively parallel system. Likewise, SpiNNaker uses parallel computation;
- **Spiking communications**: in biology, neurons communicate through spikes. The SpiNNaker architecture uses source-based AER (Address-Event Representation) packets to transmit the equivalent of neural signals [26];
- **Event-driven behaviour**: neurons are very power efficient, and consume much less power than modern hardware. To reduce power consumption we put the hardware in "sleep" state waiting for an event [25];

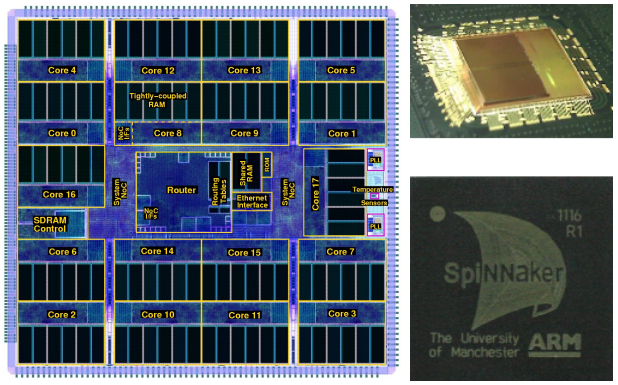


Fig. 1. SpiNNaker chip layout with labels for functional blocks and pictures of the chip before and after packaging. Image taken from the GDS2 plot sent to the manufacturer. The chip was fabricated at UMC using their 130e-llsp low-power process. Die size is $\approx 10 \times 10$ mm.

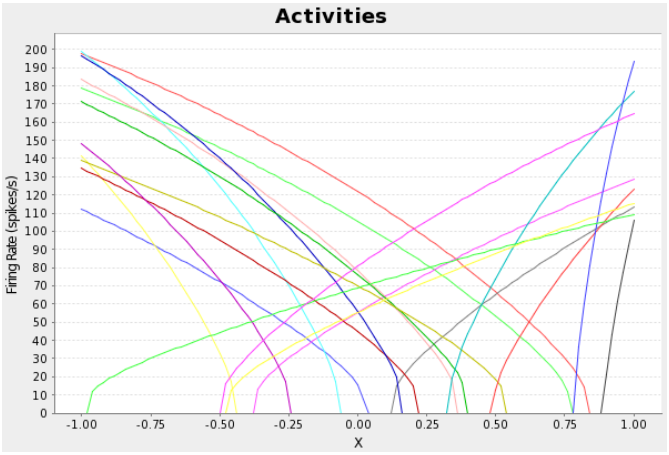


Fig. 2. Example of tuning curves: the horizontal axis represents the input value, the vertical axis represents the neural firing rate in spikes per second. Each curve represents a neuron of the population.

- **Distributed memory**: in biology, neurons use only local information to process incoming stimuli. In the SpiNNaker architecture we use memory local to each of the cores and an SDRAM local to each chip;
- **Reconfigurability**: in biology, synapses are plastic. This means that the neural connectivity changes in both shape and strength. The SpiNNaker architecture allows on-the-fly reconfiguration.

Here we introduced the main features of the SpiNNaker system. Complete details of the system and of its performance can be found in [7], [24], [27] and [28]. This architecture has been used for the experiments described in this paper with the purpose of speeding up the simulation.

III. The Neural Engineering Framework

The Neural Engineering Framework (NEF) is a theoretical framework to convert high-level mathematical functions into realistic spiking neural networks [2]. This framework represents vectors in a distributed fashion, using neural activity to encode vector values. The algorithm to be implemented is encoded in the connections between populations of neuron. The mathematical foundations of this framework is in the definition of the set of encoders and decoders: an input vector x is encoded by neural activity using an encoding vector particular to each neuron. On the decoder side, to decode the spike train in an estimated value $\hat{x}$, a set of decoders needs to be defined.

This section introduces the mathematical background of the NEF using models and formulas from [2]. The next section presents the actual authors' contribution: a learning rule that may be applied to the NEF and that adapts to the peculiarities of the SpiNNaker architecture.

If we define $a_i(x)$ as the neural activity of the i-th neuron in correspondence of the input vector x, the activity can be computed as:

$$a_i(x) = G(\alpha_i e_i x + b_i) \tag{1}$$

where G is a non-linear function that represents the link between the current injected in a neuron and its firing rate (this function is typical for each mathematical model); α_i is a gain factor; e_i is the randomly chosen encoder value, which represents the value for the maximum spiking rate; and b is the background current flowing into the neuron. Such values are specific for each neuron, and in a population of neurons encoding unidimensional vectors (x and e are unidimensional vectors), the plot of neural activity shows the "tuning curves" of the population. An example of tuning curves is presented in fig. 2. On the horizontal axis is the input value, x in eq. (1). On the vertical axis is the firing rate of the neuron corresponding to each input value. The function between the two dimensions is expressed by eq. (1), and each curve in fig. 2 represents the activity of each neuron in a population.

While eq. (1) represents the neural activity following an input value, the inverse pass is computed using eq. (2): the output vector $\hat{x}$ is estimated so that the value decoded has the minimum square error from the input vector x.

$$\hat{x} = \sum_i (d_i a_i(x)) \tag{2}$$

$$E = \frac{1}{2}\int (x - \hat{x})^2 dx = \frac{1}{2}\int (x - \sum_i (d_i a_i(x)))^2 dx \tag{3}$$

$$\frac{\delta E}{\delta d_i} = -(x - \sum_j (d_j a_j(x)))a_i(x) = 0 \tag{4}$$

$$d_i = \Gamma_{ij}^{-1}\Upsilon_j \tag{5}$$

$$\Gamma_{ij} = \int a_i(x)a_j(x)dx \tag{6}$$

$$\Upsilon_j = \int a_j(x)x dx \tag{7}$$

The set of decoder values is therefore computed using the mean square error function (eq. 3), and setting its derivative to 0 (eq. 4).

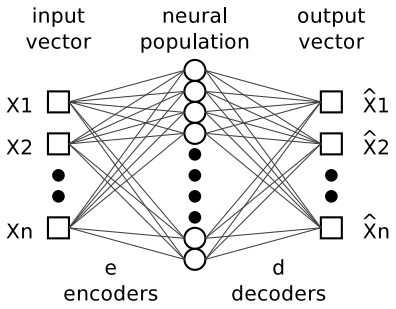


Fig. 3. Encoding and decoding values from one population.

Encoders and decoders are used to transform a vector into spikes, and from spikes back to the vector (see fig. 3). If a connection between two population of neurons is desired, to transfer the value from the first to the second population is possible to specify adequately the interconnections weights to perform the operation. In first instance, we can assume that between the two population of neurons there are as many units as there are dimensions of the input vector (see fig. 4(a)). If we suppose that such units are perfectly linear, the output is a reproduction of the input value. The input value is decoded from the spikes received from the first population, and the output of such units is then encoded for the second population.

Since the linear units are just idealized neurons, not present in biology, we can eliminate them (see fig. 4(b)) computing directly the interconnecting weights as a matrix product between decoders of the first population and the encoders of the second population: $w_{ij} = d_i \cdot e_j$

If an algorithm (or equivalently a function) is desired, then the $f(x)$ may be computed using the appropriate decoders:

$$d_i^{f(x)=\Gamma_{ij}^{-1}\Upsilon_j^{f(x)}} \tag{8}$$

$$\Upsilon_j^{f(x)} = \int a_j(x) f(x) dx \tag{9}$$

The transfer function may be defined at network description time, or learned, modifying appropriately the synaptic weights interconnecting the input and output populations. A method to implement the latter case is presented in the next section

IV. The PES learning rule

Previous work described the implementation of networks with known and fixed transfer functions. In this section we present a learning rule which modifies the synaptic weight using a teaching signal. This learning rule has been extracted by the authors from the NEF and forms the main contribution of this paper.

Starting with random initial synaptic weights, if we want the network to learn a new transfer function $y = f(x)$ using a supervised paradigm [21], the synaptic weighs w_{ij} need to be modified so that, keeping e_j constants, the decoders d_i are modified to implement a function $y = f(x)$.

To determine the amount of change in the synaptic weights, we need to minimize the mean square error between the desired output y and the current output $\hat{y}$. In this case, mean square error is computed as:

$$E = \frac{1}{2}\int (y-\hat{y})^2 dx = \frac{1}{2}\int (f(x) - \hat{f}(x))^2 dx) =$$
$$= \frac{1}{2}\int (f(x) - \sum_i (d_i^{f(x)} a_i(x)))^2 dx \tag{10}$$

Minimizing this error we obtain:

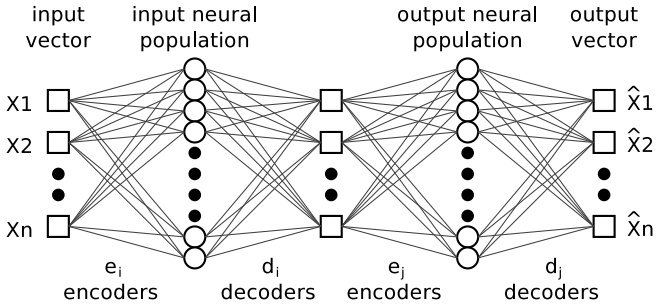
$$\frac{\delta E}{\delta d_i} = -(f(x) - \sum_j (d_j^{f(x)} a_j(x))) a_i(x) = 0 \tag{11}$$

Converting this function into a standard delta rule, and including the learning rate parameter k, we obtain:

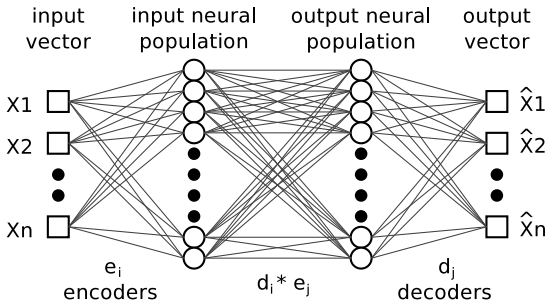
$$\Delta d_i = k \cdot m \cdot a_i(x) \tag{12}$$

where:

- Δd_i is the variation of the decoder;
- k is the learning rate parameter;



(a) Encoding and deconding values from two interconnected population with perfectly linear elements in between. The input units provide a value that, multipled by the encoders is transformed by the input neural population in spikes. The tuning curves in fig. 2 provides an example of the relation between neural input and output. Spikes are then decoded and transferred to the second population.



(b) Encoding and deconding values from two interconnected population. The input units provide a value that, multipled by the encoders is transformed by the neurons in spikes. These are then transferred to the second population through synapses whose weight is function of decoders and encoders, and finally decoded.

Fig. 4. Network structures considering encoders and decoders.

- m is a modulatory signal which is related to the error between the desired function and the implemented function;
- $a_i(x)$ represents the activity of the i-th neuron for the value x.

Multiplying both sides of (12) by e_j we obtain:

$$e_j \cdot \Delta d_i = k(e_j \cdot m)a_i(x) = \frac{k}{\alpha_j}(\alpha_j \cdot e_j \cdot m)a_i(x) \qquad (13)$$

If we include the term α_j into the learning rate parameter k and set $I_{e_j} = \alpha_j \cdot e_j \cdot m$, the equation (13) represents the weight variation:

$$\Delta w_{ij} = e_j \cdot \Delta d_i = k \cdot I_{e_j} \cdot a_i(x) \qquad (14)$$

where:

- $\Delta w_i j$: is the additive weight variation of the synapse between neuron i in the input population and neuron j in the output population;
- k: is the learning rate;
- I_{e_j}: is the modulatory error current for neuron j received from the error population;
- $a_i(x)$: this function represents the activity for the i-th neuron in the input population. It takes value 1 when this neuron emits an action potential.

Following this learning rule, only synapses, through which there is activity, are modified following the modulatory error signal; in this environment the error signal provides a teaching signal to the learning synapses, so that the output value generated by the output population agrees with the training value.

This learning rule belongs to the class PES (Prescribed Error Sensitivity): the learning synapses have a more rapid efficacy variation where the modulatory error current is greater. It may be worth noting that this learning rule allows the learning of both linear and non-linear transfer functions.

The work presented in this paper builds on the work presented in [6] which presented the first implementation of the NEF on the SpiNNaker platform. The NEF implementation on SpiNNaker noes not implement learning rules, and this work is intended to extend the learning capabilities of the NEF implementation on SpiNNaker in a way that exploits the unique features of the architecture, such as multicast packet routing [26] and inter-neuron delay modelled as pure dendritic delay.

Alternative, biologically plausible, learning rules such as Spike Timing Dependent Plasticity (STDP), use a different background, without such direct association and that follow the Hebb's postulate: *"Cells that fire together wire together"*. The STDP learning rule has been used in unsupervised and supervised frameworks [20], together with a modulatory signal to reinforce correct behaviour and depress incorrect ones. Even though the PES and STDP learning rules present similarities, their application field is different and a direct comparison between the two approaches is not immediate.

Another comparison may be done with a Multi-Layer Perceptron (MLP), a second generation neural network [16]

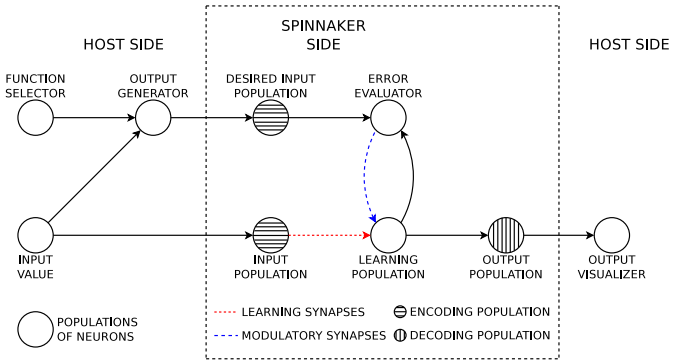


Fig. 5. Structure of the network used to test the learning feature.

which resembles the structure in fig. 4(b). However, a direct comparison between networks of different generation is not an easy task that we leave, together with a comparison with the STDP learning rule, to future publication(s).

In this paper we intend to show the validity of the presented PES learning rule, and therefore limit the experiments to simple trials. More complex, real world interactions are left to future applications of the NEF framework.

V. Neural network structure

To test the PES learning rule we used a spiking neural network to generate the input and the correct output required from the synapses under test, and then to visualize the result of the algorithm.

The network used (see fig. 5) comprises three stages: the first, simulated on a host computer (the computer attached to the SpiNNaker board), computes the input and the desired output of the network. The second is simulated on the SpiNNaker system and involves the two sets of trainer and trainee synapses. The output of the learning stage is then sent back to the host computer for visualization (third part of the test network). The dashed box in the image identifies the part of the network which is simulated on the SpiNNaker system. The set of trainee synapses is initially set to random weights, so that it does not represent any particular transfer function.

A schematic model of the network in use is presented in fig. 5. In this image the circles represent populations of neurons, black solid arrows represent static connections between populations; the red dotted line represents the set of trainee synapses; the blue dashed line represents the set of modulatory synapses.

The filled populations simulated on the SpiNNaker system represent a set of "interface" neural populations. The horizontally-shaded populations represent the encoders of the value received from the host: on the basis of the value to be represented, a neural activity is generated (see eq. 1). The vertically-shaded population performs the decoding operation: from the neural activity received by the "learning population", the set of decoders (see eq. 2) extract the value to be sent to the host computer for visualization.

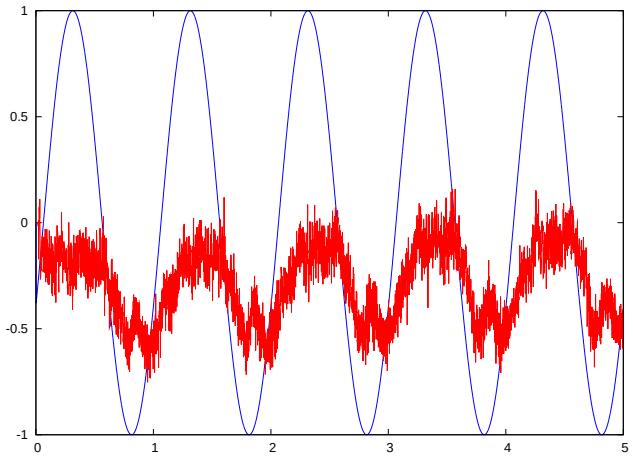


Fig. 6. Output of the network without training: no particular pattern is visible. The blue line represents the input and the desired output. The red line shows the network response.

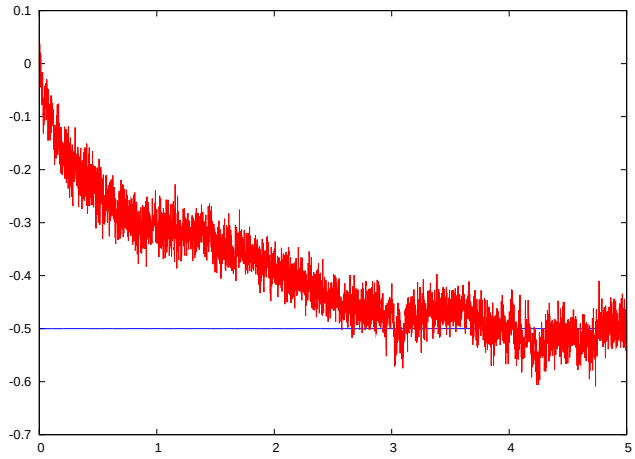


Fig. 7. Example of adaptation in the learning process. The horizontal axis represents the simulation time in sec, the blue line represents the desired output, and the red output shows the actual network output. The input and the desired output value are changed in correspondance with the beginning of the graph from 0 to -0.5. The network adapts to the new stimulus after a transient period.

The input value, generated by the population "Input Value", (x in eq. 1) is propagated both to the input population on the SpiNNaker system, to provide the input for the set of trainee synapses, and to the population named "Output Generator", which computes the desired output for each input. The "Function Selector" population is used to select which function to apply to the input value. The possible functions to apply are:

- $y = x$: a communication channel;
- $y = -x$: an inverse communication channel.
- $y = \sin(x)$: a sine evaluator;

As this set includes both linear and non-linear functions, we show that the set of trainee synapses is able to learn both kinds of transfer functions.

VI. Results of the learning process

The learning process described here is carried out using a learning parameter $k = 5 \cdot 10^{-5}$ in eq. (14). The PES learning rule has been tested using the network described earlier through a two-stage process: the first is a purely learning stage: an input value is constantly provided to the network, together with the required output value, for 20 seconds. The input values provided in this training phase are in order: 0, -0.25, -0.5, -0.75, -1, 0.25, 0.5, 0, 75, 1. The required output is generated using one of the three functions described above, and the simulation on the SpiNNaker board has been run 10 times slower than biological time to allow the extraction and storage of all data during the simulation.

Before training the network, synaptic weights are set to random values. Fig. 6 shows the initial transfer function between the input and the output values. Whilst there is a correlation between the two, the range of the output values does not cover the complete interval $[-1; 1]$, and the noise is more evident when compared with the output generated after training (see figures 9, 11 and 12).

Using the PES algorithm, the set of trainee synapses adapt their weight to reproduce the desired output value. An example of this process is presented in fig. 7: here the horizontal axis represents the simulation time (in seconds), the blue line represents the desired output for which the network is being trained, and the red line is the output of the network. In this example the input and the desired value are being changed, at time 0 of the graph, from 0 to -0.5. It is possible to see that, after a transitory period, the network adapts to the new desired value.

The second stage of the testing process is the testing phase: the input provided to the network for 5 seconds is a sinusoidal signal with period of 1 second. The output of the network is compared with the desired output signal. In this phase the learning algorithm is still active.

During this testing phase it is spossible to noice the interpolation made by the neural network: although the training input provided does not include all the possible values, the network provides a continuous output inpterpolating between known values on which it has been initially trained.

This process, comprising the two stages described earlier, is repeated three times, once for each of the transfer functions described, which include both linear (the communication channel, and the inverse communication channel) and non-linear (the sinusoid) functions.

An example of the input signal is presented by the blue line in fig. 8. Figures 9, 11 and 12 present this input signal (in green) together with the desired output signal, in blue, as generated by the "output generator" population in fig. 5, and the signal generated by the "output population" of the spiking neural network simulated on the SpiNNaker system, in red, after the training. As the output value is estimated (see eq. (2)), some noise is present on the output signal. The output value presented here is filtered using a sliding window filter

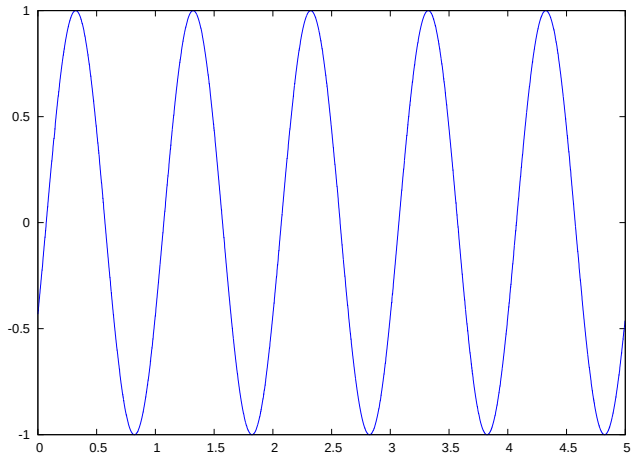


Fig. 8. Input signal to test the transfer function. On the horizontal axis the simulation time in sec. On the vertical axis the input value.

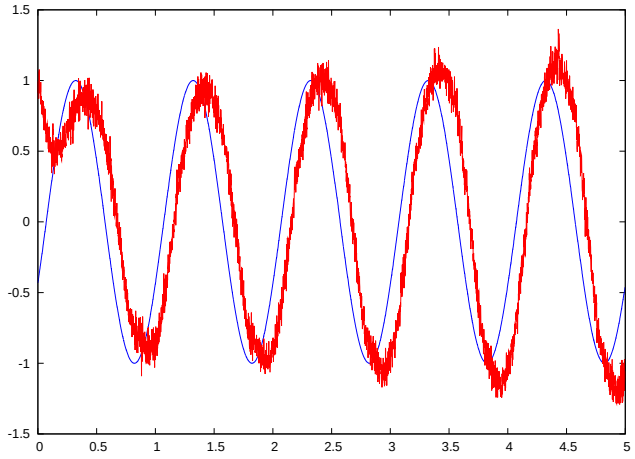


Fig. 9. Output of the communication channel ($y = x$) test. In blue the desired output, in red the output of the neural network. On the horizontal axis the simulation time in sec. On the vertical axis the output value.

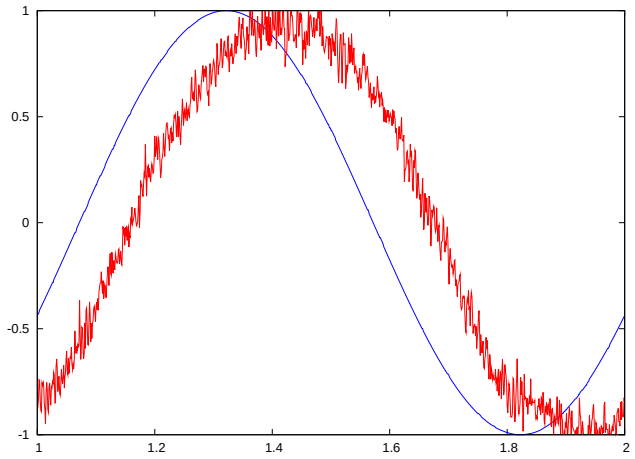


Fig. 10. Output of the communication channel ($y = x$) test. In blue the desired output, in red the output of the neural network. On the horizontal axis the simulation time in sec. On the vertical axis the output value. Ths image is a portion of fig. 9 representing the values between the seconds 1 and 2 of the simulation time

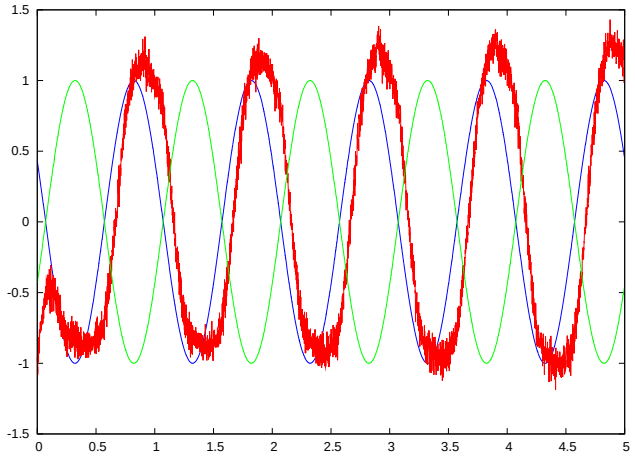


Fig. 11. Output of the inverse communication channel ($y = -x$) test. In green the input signal, in blue the desired output and in red the output of the neural network. On the horizontal axis the simulation time in sec. On the vertical axis the output value.

that computes the average value over 10 samples: the current value and the past 9 values received from the network.

Fig. 9 presents the output for a linear transfer function ($y = x$). It is possible to see that the output consistently follows the desired signal with a constant delay. However, even though the learning phase is concluded, the learning algorithm is always active, and the result is that the output value has the tendency to increase, slightly, the value range during the testing phase.

Fig. 10 details the section of fig. 9 related to the simulation between seconds 1 and 2. It is possible to note how the output follows the input, even if there is a delay in the response.

The same set of comments can apply also to the inverse communication channel transfer function ($y = -x$): as it is possible to see in fig. 11, the output signal follows the desired output signal with a constant delay, its values cover the whole interval $[-1; 1]$, and has the tendency to increase slightly, as described before.

The next transfer function presented is non-linear; a sinusoidal function. Fig. 12 presents the outcome of this training experiment; it is possible to see the desired output, in this case, covers the interval between $\sin(1) \approx 0.841$ and $\sin(-1) \approx -0.841$ (where the argument of the $\sin()$ function is expressed in radiant).

Here it is possible to see how the output signal adapts to the desired transfer function: even though the learning process was not completed in the given time frame, the neural network still learns during this testing phase. In particular, the negative peak value progressively approaches the desired output value, reaching it in the last sinusoid.

VII. Conclusions

We presented a novel learning rule of the Prescribed Error Sensitivity (PES) class, that we applied in a supervised learn-

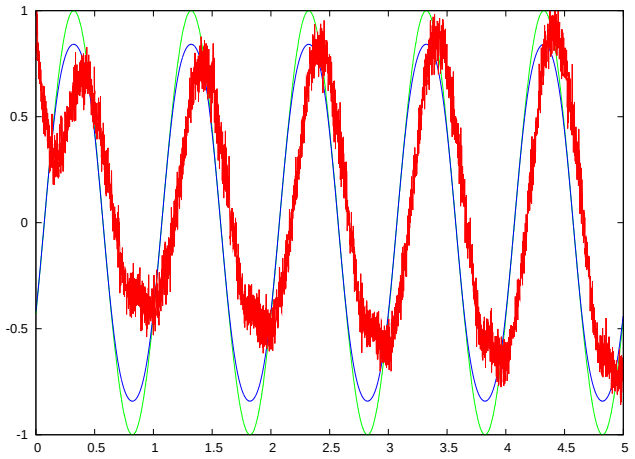


Fig. 12. Output of the sinusoidal transfer function ($y = \sin(x)$). In green the input signal, in blue the desired output and in red the output of the neural network. On the horizontal axis the simulation time in sec. On the vertical axis the output value.

ing framework. This rule, in fact, requires a set of modulatory synapses that identify the size of the synaptic weight change for each synapse. The learning rule has been initially characterized on theoretical basis, providing mathematical support.

In a second step, the rule has been applied to the SpiNNaker neuromimetic simulator for practical evaluation. The rule has been used to train a set of learning synapses to represent three transfer functions (linear and non-linear), showing that through this learning rule they can adapt to the function required.

From the graph of the outgoing signal it is possible to note a delay in the neural response. There are two main reasons: firstly synapses are not instantaneous, but have an intrinsic delay of $1msec$. Secondly, each of the neural populations requires some time to decode the new incoming spike rate and produce the required outgoing spikes according to the newly decoded value. This happens because some time is needed for the neurons to change their state, from "active" to "quiescent" and even longer for the inverse change.

The learning rate is a parameter that influences how quickly the network adapts to the new transfer function. The higher, the better, from a theoretical point of view, because it implies shorter learning times. However the higher the learning rate, the greater are the synaptic weight variations, which may lead to an unstable system that starts oscillating.

This problem is strictly connected with the response delay described earlier: since there is a delay between the application of the input and the generation of the response by the neural population, the feedback loop in the neural network (composed by the two populations "Error evaluator" and "Learning population" of fig. 5) is influenced by this. As a consequence, if sudden variation of the output signal occurs, the "error evaluator" population requires some time before sensing and correcting it. In the meanwhile the variation increases even more and the training loop results in a positive feedback with the consequent oscillations.

Therefore, the learning rate needs to be tuned according to the activity of the pre-synaptic population and the error evaluator population (according to eq. 14), so that such behaviour does not occur.

The PES learning rule has been discussed with regard to some particular, elementary, transfer functions. In fact, communication channels and sinusoidal functions can be implemented easily using "static" network structures. The novelty presented in this paper is the learning rule, that allows a single set of synapses to learn different transfer functions in subsequent training phases.

The Neural Engineering Framework has been designed to provide an engineering approach to the design of neural networks using attractor networks. Such a framework allows the realization of large scale neural network models that are able to simulate, to some extent, the behaviour of a human brain. This learning rule adds another facility to the simulation of biological neural networks, in particular, with regard to the simulation of the ability of biological neural networks to adapt to the environemnt in which the neural network lives.

Big neural network models, like SPAUN [1], may benefit from this learning rule and it introduces a new functionality that is simple to implement in the SpiNNaker simulation substrate, allowing behaviour similar to what happens in biology and taking advantage of the computational power provided by the SpiNNaker neuromimetic architecture.

ACKNOWLEDGMENTS

The research presented here was initiated during the Telluride Neuromorphic Engineering Workshop 2012, and extended beyond this workshop. The authors would like to thank all the organizers.

The SpiNNaker project is supported by the Engineering and Physical Science Research Council (EPSRC), grant EP/4015740/1, and also by ARM and Silistix. We appreciate the support of these sponsors and industrial partners.

The authors would like to thank Dr. Simon Davidson and Dr. John Viv Woods for the time dedicated to discussions and reviews of this paper.

REFERENCES

[1] C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen, "A Large-Scale Model of the Functioning Brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, Nov. 2012.

[2] C. Eliasmith and C. H. Anderson, *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems (Computational Neuroscience)*, new ed ed. A Bradford Book, Aug. 2004.

[3] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. Bower, M. Diesmann, A. Morrison, P. Goodman, F. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. Davison, S. E. Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: a review of tools and strategies." *Journal of computational neuroscience*, vol. 23, no. 3, pp. 349–398, Dec. 2007.

[4] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1-3, pp. 239–255, Dec. 2010.

[5] S. Davies, "Learning in Spiking Neural Networks," Ph.D. dissertation, School of Computer Science, The University of Manchester, Kilburn Building, Oxford Road, M13 9PL, Manchester, UK, Feb 2013.

[6] F. Galluppi, S. Davies, T. Stewart, C. Eliasmith, and S. Furber, "Real Time On-Chip Implementation of Dynamical Systems with Spiking Neurons," in *WCCI 2012, The 2012 IEEE World Congress on Computational Intelligence*. IEEE, Jun. 2012, pp. 2455–2462.

[7] S. B. Furber, S. Temple, and A. D. Brown, "High-Performance Computing for Systems of Spiking Neurons," in *Proc. AISB'06 workshop on GC5: Architecture of Brain and Mind*, Apr. 2006.

[8] E. L. Bienenstock, L. N. Cooper, and P. W. Munro, "Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex," *J. Neurosci.*, vol. 2, no. 1, pp. 32–48, Jan. 1982.

[9] H. Markram and M. Tsodyks, "Redistribution of synaptic efficacy between neocortical pyramidal neurons." *Nature*, vol. 382, no. 6594, pp. 807–810, Aug. 1996.

[10] C. Clopath, L. Busing, E. Vasilaki, and W. Gerstner, "Connectivity reflects coding: a model of voltage-based STDP with homeostasis," *Nature Neuroscience*, vol. 13, no. 3, pp. 344–352, Mar. 2010.

[11] S. Davies, A. Rast, F. Galluppi, and S. Furber, "A forecast-based biologically-plausible STDP learning rule," in *IJCNN 2011*, Aug. 2011, pp. 1810–1817.

[12] S. Davies, F. Galluppi, A. D. Rast, and S. B. Furber, "A forecast-based STDP rule suitable for neuromorphic implementation," *Neural Networks*, vol. 32, no. 0, pp. 3–14, 2012.

[13] G.-Q. Bi and M.-M. Poo, "Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type," *J. Neurosci.*, vol. 18, no. 24, pp. 10 464–10 472, Dec. 1998.

[14] D. J. Bakkum, Z. C. Chao, and S. M. Potter, "Long-term activity-dependent plasticity of action potential propagation delay and amplitude in cortical networks," *PLoS ONE*, vol. 3, no. 5, p. e2088, May 2008.

[15] L. F. Abbott and S. B. Nelson, "Synaptic plasticity: taming the beast." *Nature neuroscience*, vol. 3 Suppl, pp. 1178–1183, Nov. 2000.

[16] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[17] J. Sjöström and W. Gerstner, "Spike-timing dependent plasticity," *Scholarpedia*, vol. 5, no. 2, pp. 1362+, 2010.

[18] T. Masquelier and S. J. Thorpe, "Unsupervised learning of visual features through spike timing dependent plasticity," *PLoS Comput Biol*, vol. 3, no. 2, p. e31, 02 2007.

[19] T. Masquelier, R. Guyonneau, and S. J. Thorpe, "Competitive STDP-Based Spike Pattern Learning." *Neural Computation*, vol. 21, no. 5, pp. 1259–1276, Dec. 2008.

[20] T. J. Strain, L. J. McDaid, L. P. Maguire, and T. M. McGinnity, "A Supervised STDP Based Training Algorithm with Dynamic Threshold Neurons," in *Neural Networks, 2006. IJCNN 2006. International Joint Conference on*. IEEE, 2006, pp. 3409–3414.

[21] D. MacNeil and C. Eliasmith, "Fine-Tuning and the Stability of Recurrent Neural Networks," *PLoS ONE*, vol. 6, no. 9, pp. e22 885+, Sep. 2011.

[22] M. A. Farries and A. L. Fairhall, "Reinforcement Learning With Modulated Spike Timing–Dependent Synaptic Plasticity," *Journal of Neurophysiology*, vol. 98, no. 6, pp. 3648–3665, Dec. 2007.

[23] T. C. Stewart, T. Bekolay, and C. Eliasmith, "Learning to select actions with spiking neurons in the basal ganglia," *Frontiers in Neuroscience*, vol. 6, no. 2, 2012.

[24] A. D. Rast, X. Jin, F. Galluppi, L. A. Plana, C. Patterson, and S. Furber, "Scalable event-driven native parallel processing: the SpiNNaker neuromimetic system," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 21–30.

[25] X. Jin, S. B. Furber, and J. V. Woods, "Efficient modelling of spiking neural networks on a scalable chip multiprocessor," *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pp. 2812–2819, Sep. 2008.

[26] S. Davies, J. Navaridas, F. Galluppi, and S. Furber, "Population-Based Routing in the SpiNNaker Neuromorphic Architecture," in *WCCI 2012, The 2012 IEEE World Congress on Computational Intelligence*. IEEE, Jun. 2012, pp. 1932–1939.

[27] L. A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A GALS infrastructure for a massively parallel multiprocessor," *Design & Test of Computers, IEEE*, vol. 24, no. 5, pp. 454–463, Oct. 2007.

[28] A. Rast, F. Galluppi, S. Davies, L. Plana, C. Patterson, T. Sharp, D. Lester, and S. Furber, "Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware," *Neural Networks*, vol. 24, pp. 961–978, 2011.