A Unified Learning Framework for Memristive Neuromorphic Hardware

Anatoli Gorchetchnikov, Massimiliano Versace, Heather Ames, Ben Chandler, Jasmin Léveillé, Gennady Livitz, and Ennio Mingolla, Boston University
Greg Snider, Rick Amerson, Dick Carter, Hisham Abdalla, and Muhammad Shakeel Qureshi
Hewlett-Packard Laboratories

Abstract—Realizing adaptive brain functions subserving perception, cognition, and motor behavior on biological temporal and spatial scales remains out of reach for even the fastest computers. Newly introduced memristive hardware approaches open the opportunity to implement dense, low-power synaptic memories of up to $10^{15}$ bits per square centimeter. Memristors have the unique property of “remembering” the past history of their stimulation in their resistive state and do not require power to maintain their memory, making them ideal candidates to implement large arrays of plastic synapses supporting learning in neural models. Over the past decades, many learning laws have been proposed in the literature to explain how neural activity shapes synaptic connections to support adaptive behavior. To ensure an optimal implementation of a large variety of learning laws in hardware, some general and easily parameterized form of learning law must be designed. This general form learning equation would allow instantiation of multiple learning laws through different parameterizations, without rewiring the hardware. This paper characterizes a subset of local learning laws amenable to implementation in memristive hardware. The analyzed laws belong to four broad classes: Hebb rule derivatives with various methods for gating learning and decay; Threshold rule variations including the covariance and BCM families; Input reconstruction-based learning rules; and Explicit temporal trace-based rules.

I. INTRODUCTION

IMPLEMENTING brain functions enabling perception, cognition, and learning on biological temporal and spatial scales, while easily achieved by mammalian nervous systems, remains infeasible for current computational architectures. In the past few years work on memristive devices has brought modelers closer to realizing digital brain architectures that can interact with the world in real time while learning [18][19][21]. The mathematical definition of learning equations that are compatible with the newly designed hardware is a major outstanding issue.

One such example of co-development of adaptive hardware and software for memristive devices is the collaboration between Hewlett-Packard and Boston University. The jointly developed Cog Ex Machina (Cog) software architecture is targeted for a variety of hardware platforms including Dendra chips, which are massively parallel processors characterized by dense, memristive memories integrated with conventional circuitry for inference and learning, such that memory and computational circuits are closer together. Cog represents a computation as a directed graph where nodes hold computational state and exchange information across edges. Edges relay information from one node to another using a multidimensional array of tensors (tensor field). Cog implements plasticity in adaptive transformations that use incoming tensor fields to produce output tensor fields on outgoing edges [19].

Plasticity rules implemented in adaptive transformations should be as flexible as possible to accommodate a wide variety of existing learning laws and at the same time should be as simple as possible to reduce the computational cost of learning. One step toward accomplishing this goal is to analyze a large sample of candidate learning laws and recast them as instantiations of one general model.

Existing learning laws can be roughly categorized in the following classes: Basic Hebb rule derivatives; Threshold-based rules; Input reconstruction-based rules; and Explicit temporal trace-based rules. Most of the cases discussed in this paper only require a two layer network, while for networks containing more than two layers, learning proceeds independently between each pair of consecutive layers. A notable exception to this is the back-propagation rule, where the learning in each weight matrix depends on an error term propagated all the way from the top-most layer to the bottom-most layer. A network design capable of implementing this kind of propagation will be introduced in the corresponding section. For all other sections we use the network depicted in Figure 1a.

Here $x_i$ represents the activity of presynaptic neuron $i$ (or vector $x$ represents the activity of all presynaptic neurons); $y_j$ represents the activity of postsynaptic neuron $j$ ($y$ in vector notation); and $w_{ij}$ represents the synaptic weight from $i$ to $j$ ($w$ is the matrix of all weights). The postsynaptic activity $y$ can be calculated via a differential equation:
long-term memory. Cog combines the state of a linear transformation, transformations cooperate to implement the learning that corresponds to a slightly different form similar to equation (3): 

\[ dw_{ij} = \lambda (x_i y_j - w_{ij}) f(x_i, y_j) \]  

(4)

Here the gating function \( f() \) is applied not only to the decay, but also to the learning process. As a result, the equilibrium value of the weight always follows the Hebbian correlation, but the dynamics of this process are governed by the function \( f() \). Both forms can be combined as:

\[ dw_{ij} = \lambda (x_i y_j - g(x_i, y_j) y_{ij}) h(x_i, y_j) \]  

(5)

Hardware constraints put additional limitations on the shapes of the functions \( g() \) and \( h() \). Since these functions are dependent on the value of \( y_j \) only, they have to be computed in the nodes. At the same time, \( x_i \) is not available at this level unless it is transmitted from the pre-synaptic node directly to the post-synaptic node, effectively bypassing the linear transformation. If the functions \( g() \) and \( h() \) are only functions of \( y_j \), the data transfer fits into the one depicted in Figure 1. Our analysis shows that all the rules in the Hebbian family have separable components of these functions that depend on \( x_i \) and \( y_j \) independently. Furthermore, the component that depends on \( x_i \) is linear in all cases and an integral power of \( x_i \) in the most general case. Thus, equation (4) can be recast as:

\[ dw_{ij} = \lambda (x_i y_j - (\beta_1 x_i^n + g(y_j)) w_{ij}) (\beta_2 x_i^m + h(y_j)) \]  

(6)

where \( \beta_1 \) and \( \beta_2 \) are coefficients. Note that the \( y_j \) in the Hebbian term was replaced with a function of \( y_j \), which will be convenient for some of the rules. All functions \( f() \), \( g() \), and \( h() \) now only depend on \( y_j \) and not \( x_i \). The recasting of single-equation Hebb rule derivatives are summarized in Tables 1 and 2 where Table 2 indicates the rules that are used for STDP.

For all cases of the single-equation Hebb rule derivatives, parameters \( n \) and \( m \) in equation (6) are 1, and the most complex function of \( y_j \) is cubic. The general form equation (6) is able to accommodate all rules listed in Tables 1 and 2 without requiring an expensive data transfer of \( x_i \).

II. BASIC HEBB RULE DERIVATIVES

The premise of the simple mathematical interpretation of the Hebb rule [9] is that there is a direct dependency between synaptic plasticity and repeated and persistent stimulation of a postsynaptic cell by a presynaptic cell, which is usually expressed as a correlation between weight change and the product of presynaptic and postsynaptic activities. All of the rules discussed in this section are based on this fundamental rule and are widely applied in a class of continuous firing dynamical models expressed in differential form; equation (1). A general form for Hebbian rules is:

\[ \frac{dy_{ij}}{dt} = \phi(y_j) \sum_{i=1}^{N} x_i w_{ij} + \psi(y_j) \]  

(1)

where \( \phi() \) represents the resulting activation from the current input, and \( \psi() \) represents the internal dynamics of activation. Another way to calculate the activation is through current input, and \( \phi(x) \).

Some rules were designed to work with the differential form of the postsynaptic activation while others were set to work with the algebraic form. We will refer to these forms or equations in the sections for corresponding rules. In the following sections, we analyze each learning rule class and discuss its implementation in Cog (Figure 1b).

III. THRESHOLD-BASED RULES

The primary difference between threshold-based rules and Hebbian rules is the introduction of a threshold in considering the influence of cell activities on learning. While the Hebbian correlation term is positive and leads to an increase in weight whenever both presynaptic and postsynaptic activities are positive, this is not always the case for threshold rules. These rules are set up such that if the activity of the cell is higher than a certain threshold the weight increases, but if it is lower than the threshold the weight decreases. Effectively, the Hebbian weight change is proportional to \( x_i y_j \), while threshold-based weight change is proportional to \( x_i (y_j - \theta) \).

The equations summarized in this section include covariance laws [3] and multiple variations of the BCM law (e.g.
These learning laws are commonly used in networks with unit activations governed by equation (2), together with a second equation governing the dynamics of the threshold \( \theta_{x,y} \):

\[
\frac{d\theta_{x,y}}{dt} = \psi(\theta_{x,y}, x_i, y_j)
\]  

(7)

which is used for learning only. As a result, equation (7) is usually considered a part of the learning rule rather than being a component of cell activation. Learning laws based on such thresholds are then formulated as a system of two equations, one with faster dynamics (threshold in equation (7)), and another with slower dynamics (synaptic weight change). Technically, one can consider, and it is discussed in some variations of the BCM rule, that equation (2) is a limit case of activation with very fast dynamics. Note that in some variations of the BCM rule, that equation (2) is a reduced to:

\[
\Delta w_{ij}^m = \eta \delta_j^m V_j^{m-1}
\]  

(12)

where \( m \) is the layer and:

\[
V_j^m = g(h_i^m) = g(\sum_j w_{ij}^m v_j^{m-1})
\]  

(13)

\[
\delta_i^m = g'(h_i^m)[\zeta_i^m - V_i^M]
\]  

(14)

for \( m = \text{max}(m) \) and:

\[
\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ij}^m \delta_j^m
\]  

(15)

for \( m < \text{max}(m) \). Typically, node activation spreads up the layers, whereas learning occurs offline when the error is backpropagated from the output to the hidden and input layers. The process is supervised, and requires multiple iterations through the same input/output pairs in an interleaved fashion. This makes it difficult to use backpropagation in contexts where learning must be fast, such as when learning should occur on-line as the system “behaves”, or when the correct output is not known. It is possible to recast the backpropagation algorithm in terms of our general form equation (11) by using the substitutions listed in Table 4. The general form equation of learning in backpropagation is found in Table 6.

### IV. Input Reconstruction-based Rules

The rules described in this section require a feedforward pass to calculate the activation of nodes in the network, and a feedback pass through the transposed weight matrix to calculate the values necessary to update the synaptic weights. These rules can generally be implemented in Cog by using a transform to broadcast the transpose of the synaptic weight matrix from the output nodes to the input nodes. The equations summarized in this section include backpropagation [2][17], Harpur’s law [8] and the contrastive divergence law [10].

#### A. Backpropagation

A typical backpropagation network can have multiple layers, with a typical architecture consisting of three layers, where recurrence may be introduced [4][12]. The change in synaptic weights is governed by the following equation:

\[
\Delta w_{ij}^m = \eta \delta_j^m V_j^{m-1}
\]  

(12)

where \( m \) is the layer and:

\[
V_j^m = g(h_i^m) = g(\sum_j w_{ij}^m v_j^{m-1})
\]  

(13)

\[
\delta_i^m = g'(h_i^m)[\zeta_i^m - V_i^M]
\]  

(14)

for \( m = \text{max}(m) \) and:

\[
\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ij}^m \delta_j^m
\]  

(15)

for \( m < \text{max}(m) \). Typically, node activation spreads up the layers, whereas learning occurs offline when the error is backpropagated from the output to the hidden and input layers. The process is supervised, and requires multiple iterations through the same input/output pairs in an interleaved fashion. This makes it difficult to use backpropagation in contexts where learning must be fast, such as when learning should occur on-line as the system “behaves”, or when the correct output is not known. It is possible to recast the backpropagation algorithm in terms of our general form equation (11) by using the substitutions listed in Table 4. The general form equation of learning in backpropagation is found in Table 6.

### TABLE IV

<table>
<thead>
<tr>
<th>Substitutions to convert the canonical backpropagation algorithm into our general form equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canonical backpropagation notation</td>
</tr>
<tr>
<td>( V_i^{m-1} )</td>
</tr>
<tr>
<td>( \delta_i^m )</td>
</tr>
<tr>
<td>( h_i^m )</td>
</tr>
<tr>
<td>( \zeta_i^m )</td>
</tr>
</tbody>
</table>

In this case equation (14) becomes
\[ \xi(y_j) = g'(y_j)[r_j - g(y_j)] \]  

(16)

for the top-most layer, where \( r_j \) is the external supervision; equation (15) becomes

\[ \xi(y_j) = g'(y_j) \sum_k w_{jk} \xi(y_k) \]  

(17)

for all other layers, where the index \( k \) denotes the cells in the upper layer and \( y \) denotes partial inference in this upper layer. Rule (12) simplifies to:

\[ \Delta w_{ij} = \lambda x_i \xi(y_j) \]  

(18)

where \( \xi(y_j) \) is calculated on the node according to equation (16) or (17). Equation (18) is just the first term of equation (11). Table 6 depicts an implementation of the backpropagation rule. Compared with the rules below, backpropagation is simpler since it only requires one forward pass through the transform, and the weights are calculated directly from the results of reverse pass. On the other hand, it is more complex than the rules below since for each layer all reverse passes through the network have to be completed before learning can happen at the bottom-most layer.

B. Harpur’s Rule

Harpur’s algorithm, unlike backpropagation, requires two forward passes through the transform, first with the original input and second with the reconstructed error term as input. It is possible to recast Harpur’s algorithm in terms of our general form equation (11) by using the substitutions listed in Table 5.

<table>
<thead>
<tr>
<th>Canonical Harpur notation</th>
<th>Adjusted notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_p - Y_d )</td>
<td>( x_i )</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( \xi(y_j) )</td>
</tr>
</tbody>
</table>

Then the learning law for Harpur’s rule is defined as:

\[ \Delta w_{ij} = \lambda \xi(y_j)x_i \]  

(19)

Here the input signal \( x_i \) is actually an error term computed through input reconstruction:

\[ x_i = x_p - Y_d = x_p - \sum_{j=1}^{M} \xi(y_j)w_{ij} \]  

(20)

and \( \xi(y_j) \) is a dynamic function of partial inference updated on each step by the following increment

\[ \Delta \xi(y_j) = \mu y_j = \mu \sum_{i=1}^{N} x_i w_{ij} \]  

(21)

Implementing Harpur’s rule in Cog by requires that a complete cycle of learning will two computational cycles, one per forward pass, with the learning itself only occurring during the second pass.

C. Hinton’s Contrastive Divergence

In matrix notation, Hinton’s contrastive divergence algorithm can be summarized in five steps: \( Y = WX, \) \( W = k(XY) \), \( X_2 = W^T Y \) (first Gibbs bounce), \( Y_2 = WX_2 \) (second Gibbs bounce), and \( W = -k(X_2Y_2) \). This rule requires two passes and transmission of \( X_2 \) values instead of \( I \) on the second pass. This raises similar concerns as storing reconstructed input values with other error correcting rules, where it is necessary to store temporary values on nodes at the previous stage. The contrastive divergence rule could be implemented in Cog by requiring two forward passes, but unlike Harpur’s rule learning has two stages, with each stage happening after their respective forward pass. To integrate these learning rules into the general form equation, we need to specify the functions that are passed from nodes to the transform and modify the first term of equation (11). They are presented in Table 6.

V. Explicit Temporal Trace-based Rules

This class of learning laws includes rules that require the maintenance of an explicit temporal trace to calculate the synaptic weight change. These rules appear both in the reinforcement learning literature, such as Rescorla-Wagner [16] and Temporal Difference [20], and in perceptual learning [5][6].

A. Rescorla-Wagner

In the Rescorla-Wagner learning equation, the synaptic change is defined as:

\[ y_j = \sum_{i=1}^{N} w_{ij}x_i \]  

(22)

\[ \dot{w}_{ij} = \lambda (r_j - y_j)x_i \]  

(23)

This second equation is then fit into the general form equation; see Table 7. This law is useful, but fails in many complex classical conditioning paradigms. Rescorla-Wagner has a Hebbian component, where \( x \) is the input which corresponds to the stimulus (Conditioned Stimulus, or CS), \( y \) is the prediction of reward that the stimulus will give, \( r \) is the actual reward, and \( f() \) is a difference of \( y_j \) and reward signal \( r \) computed on the nodes and then passed to the transform.

B. Temporal Difference

In the Temporal Difference learning equation, the synaptic change is defined as:

\[ \dot{w}_{ij}(t+1) = \lambda(r_j(t+1) + \gamma y_j(t+1) - y_j(t))x_i(t) \]  

(24)

where
\[
y_j = \sum_{i=1}^{N} w_{ij} x_i
\]  
(25)

Equation (24) can be rewritten as:

\[
\dot{w}_{ij}(t) = \lambda (r_j(t) + \gamma y_j(t) - y_j(t-1))x_i(t-1)
\]  
(26)

The value of \(\gamma\) illustrates the importance of the rewards that are expected in the future. If \(\gamma = 0\), only the reward at the current time matters. If \(\gamma > 0\), then in addition to current rewards all future rewards also matter with an exponential decay of the importance. The temporal difference rule works in discrete time steps with the weight change happening on the next time step relative to predicting inputs. This might lead to an issue of having to know the next time step relative to predicting inputs. This might result in a decision about whether to make it a two-pass rule, when on the first pass we use \(x_{t-1}\) with \(x_t\), but this depends on how fast the input changes. The problem with this approximation is that we reduce (if not eliminate) the predictive power of the input, which reduces the utility of learning. By setting \(\gamma = 0\) and \(\Delta t \to 0\), this rule reduces to the Rescorla-Wagner rule. Another solution is to make it a two-pass rule, when on the first pass we use \(x_t\) to compute \(y_j\) and on the second pass we use \(x_t(t-1)\) to compute weights. This requires double storage in the pre-synaptic nodes. One more solution is to set up the following algorithm by, at each step \(t\): compute \(y_j(t)\); compute \(\dot{w}_{ij}(t)\) using a precomputed \(-y_j(t-1)x_i(t-1)\) from the previous step \(t-1\); pre-compute \(-y_j(t)x_i(t)\) and store it on the chip for weight update on the next step \(t+1\). This procedure requires double storage in the transform.

C. Foldiak

Foldiak proposed a modified Hebbian rule where learning is proportional to the product of the presynaptic activity and the trace of the postsynaptic activity, with the postsynaptically-gated decay also being based on the trace:

\[
\Delta u_{ij}^{(t)} = \lambda \tilde{y}_j^{(t)} (x_i^{(t)} - w_{ij}^{(t)})
\]  
(27)

where:

\[
\tilde{y}_j^{(t)} = (1 - \delta)\tilde{y}_j^{(t-1)} + \delta y_j^{(t)}
\]  
(28)

With \(\Delta t \to 0\) the latter equation reduces to the differential form \(\frac{dy_j}{dt} = \delta (y_j - \tilde{y}_j)\) and the rule becomes close to the ones discussed in the threshold-based rules section. With a finite \(\Delta t\) the rule is similar to the ones used in other reinforcement learning settings (e.g., [20]), hence its inclusion in this section. The trace of the postsynaptic activity has the role of keeping a running average of the “categorization” cell to sample changes in the lower-level cells, mirroring the fact that that the features to be learned are relatively stable in the environment. By renaming \(\tilde{y}_j\) as \(\theta_j\) and \(\delta\) as \(\epsilon\), the Foldiak rule can be fit into the general form equation; see Table 7.

VI. Conclusion

Synaptic-level plasticity is one of the critical mechanisms subserving learning and memory, and is therefore an essential component in the development of a hardware and software framework aimed at implementing adaptive behavior at biological scale. This article contains a preliminary general form equation encompassing several learning law candidates for implementation on massively parallel memristive hardware realized in the Hewlett-Packard and Boston University collaboration. The Cog software architecture implements the above mentioned laws on Dendra chips, which are processors characterized by dense memristive memories coupled with conventional circuitry for inference and learning. This paper characterizes and implements four main families of rules: Basic Hebb rule derivatives, threshold-based rules, input reconstruction-based rules, and temporal trace-based rules. All equations of the four families fit in a general form equation that can be implemented on the Dendra chip.

REFERENCES

**TABLE I**

Hebb rule derivatives used in firing rate-based models, and conventionally captured by equation (3). These rules are recast in terms of equation (6), where \( \alpha \) is parameter.

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
<th>( f(y_j) )</th>
<th>( \beta_1 )</th>
<th>( g(y_j) )</th>
<th>( \beta_2 )</th>
<th>( h(y_j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic Hebb</td>
<td>( w_{ij} = \lambda x_i y_j )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hebb with passive decay</td>
<td>( \dot{w}_{ij} = \alpha x_i y_j )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \alpha )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Outstar</td>
<td>( \dot{w}<em>{ij} = -\alpha w</em>{ij} )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \alpha )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Instar</td>
<td>( \dot{w}<em>{ij} = \alpha y_j w</em>{ij} )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \alpha )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Oja [15]</td>
<td>( \dot{w}<em>{ij} = \alpha y_j w</em>{ij} )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \alpha )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dual OR</td>
<td>( \dot{w}_{ij} = \lambda x_i y_j - \alpha (x_i + y_j) )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \lambda )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dual AND</td>
<td>( \dot{w}<em>{ij} = \lambda x_i y_j - \alpha x_i y_j w</em>{ij} )</td>
<td>1</td>
<td>0</td>
<td>( \alpha )</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE II**

Spike-timing dependent plasticity (STDP) Hebb rule derivatives used in spiking models and represented by equation (4) recast in terms of equation (6), where \( \alpha \), \( \alpha_1 \), and \( \alpha_2 \) are parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
<th>( f(y_j) )</th>
<th>( \beta_1 )</th>
<th>( g(y_j) )</th>
<th>( \beta_2 )</th>
<th>( h(y_j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - w</em>{ij}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Pre-synaptic gating</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - w</em>{ij}) \alpha x_i )</td>
<td>( y_j )</td>
<td>0</td>
<td>1</td>
<td>( \alpha )</td>
<td>0</td>
</tr>
<tr>
<td>Post-synaptic gating</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - w</em>{ij}) \alpha y_j^2 )</td>
<td>( y_j )</td>
<td>0</td>
<td>1</td>
<td>( \alpha )</td>
<td>( \alpha y_j^2 )</td>
</tr>
<tr>
<td>Dual OR</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - w</em>{ij}) (\alpha_1 x_i + \alpha_2 y_j^2) )</td>
<td>( y_j )</td>
<td>0</td>
<td>1</td>
<td>( \alpha_1 )</td>
<td>( \alpha )</td>
</tr>
<tr>
<td>Dual AND</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - w</em>{ij}) \alpha x_i y_j )</td>
<td>( y_j^3 )</td>
<td>0</td>
<td>( \alpha )</td>
<td>( \alpha y_j^2 )</td>
<td>( y_j^2 )</td>
</tr>
</tbody>
</table>

**TABLE III**

Threshold-based rule variations that are usually represented by equation (8) and are recast in terms of equation (9), where \( \alpha \) and \( \epsilon \) are parameters and \( \delta' \) is a derivative of the sigmoidal activation function.

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
<th>( f(y_j) )</th>
<th>( \beta_1 )</th>
<th>( g(y_j) )</th>
<th>( \beta_3 )</th>
<th>( h(y_j, \theta) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covariance 1</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - x_i \theta</em>{y_j}) )</td>
<td>( y_j )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Covariance 2</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - y_j \theta</em>{x_i}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \theta_{x_i} )</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \theta_{x_i} = \epsilon (x_i - \theta_{x_i}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \theta_{x_i} )</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Original BCM</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - x_i \theta</em>{y_j}) y_j - \alpha w_{ij} )</td>
<td>( y_j )</td>
<td>0</td>
<td>( \alpha )</td>
<td>( y_j )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \theta_{y_j} = \frac{\lambda}{\lambda} - \theta_{y_j} )</td>
<td>( y_j )</td>
<td>1</td>
<td>0</td>
<td>( \alpha )</td>
<td>( y_j )</td>
</tr>
<tr>
<td>IBCM [11]</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - x_i \theta</em>{y_j}) \frac{\lambda}{\lambda} \theta_{y_j} )</td>
<td>( y_j )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \theta_{y_j} = \epsilon (y_j^2 - \theta_{y_j}) )</td>
<td>( y_j )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td>LBCM [13]</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - x_i \theta</em>{y_j}) y_j )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \theta_{y_j} = \epsilon (y_j^2 - \theta_{y_j}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td>Textbook BCM [3]</td>
<td>( \dot{w}<em>{ij} = \lambda (x_i y_j - x_i \theta</em>{y_j}) y_j )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
</tbody>
</table>

**TABLE VI**

Input reconstruction-based rules recast in terms of equation (11)

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
<th>( f(y_j) )</th>
<th>( \beta_1 )</th>
<th>( g(y_j) )</th>
<th>( \beta_3 )</th>
<th>( h(y_j, \theta) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpropagation</td>
<td>( \dot{w}_{ij} = \lambda x_i \xi(y_j) )</td>
<td>( \xi(y_j) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \xi(y_j) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
<td></td>
</tr>
<tr>
<td>Harpur</td>
<td>( \dot{w}_{ij} = \lambda x_i \xi(y_j) )</td>
<td>( \xi(y_j) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \xi(y_j) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
<td></td>
</tr>
<tr>
<td>Contrasting Divergence</td>
<td>( \dot{w}_{ij} = \lambda x_i y_j )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \theta_{y_j} = \epsilon (y_j^2 - \theta_{y_j}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \theta_{y_j} = \epsilon (y_j^2 - \theta_{y_j}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
<tr>
<td></td>
<td>( \theta_{y_j} = \epsilon (y_j^2 - \theta_{y_j}) )</td>
<td>( y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( y_j \delta'(y_j) )</td>
</tr>
</tbody>
</table>
### TABLE VII
**Temporal trace-based rules recast in terms of equation (11)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
<th>( f(y_j) )</th>
<th>( \beta_1 )</th>
<th>( g(y_j) )</th>
<th>( \beta_3 )</th>
<th>( h(y_j, \theta) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rescorla-Wagner</td>
<td>( \dot{w}_{ij} = \lambda x_i (r_j - y_j) )</td>
<td>( r_j - y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Temporal Difference</td>
<td>( \dot{w}_{ij}(t) = \lambda x_i(t-1) (r_j(t) + \gamma y_j(t) - y_j(t-1)) )</td>
<td>( r_j(t) + \gamma y_j(t) - y_j(t-1) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Foldiak</td>
<td>( \dot{w}<em>{ij} = \lambda \theta y_j (x_i - w</em>{ij}) )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( \theta y_j )</td>
</tr>
<tr>
<td></td>
<td>( \dot{\theta}_y = \epsilon (y_j - \theta y_j) )</td>
<td>( r_j - y_j )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>