# Error Backpropagation with Generalized Functional Composition

Alejandro Bassi and Pedro Ortega {abassi|peortega}@dcc.uchile.cl
Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile

***Abstract*— We present a flexible implementation scheme to build learning machines that are trained with backpropagation. The proposed approach departs from standard feedforward artificial neural networks by using general vector functions as the basic calculating units instead of simple neurons with single outputs. Elaborate structures are created from these basic building blocks by combining them with composition operators. The operators can express intricate dataflow interactions in a much straightforward way than what is achieved with networks of connected neurons. Nested structures are built transparently. Error and regularization terms may be inserted at arbitrary points within a network. In all cases the gradient computation is based on the same standard procedures, independently of the learning algorithm. No special training methods are needed for complex architectures. Our approach is easy to implement and to extend with custom basic units and composition operators. As one of its major practical advantages, it provides a framework to rapidly create and test many network designs for a given problem, easing the search of a suitable model.**

***Index Terms*— Backpropagation, functional composition, neural network, modularity, compilation.**

## I. Introduction

Artificial neural networks (ANN) are computational models whose processing units can be viewed as simplified simulations of biological neurons. The calculation of a unit generally involves a weighed sum of its inputs, reshaped by a non-linear activation function with bounded output. Following the early success of the perceptron [5], the decisive breakthrough of artificial neural networks came after the discovery of the backpropagation algorithm [8] which allows to adjust the weights of multilayered networks using gradient descent. The key improvement was to replace the discontinuous step activation of the original artificial neuron with a continuous logistic function [6], [7].

In spite of the historical importance of the neuronal metaphor, most multilayered networks can be described in a much more compact way relying on general vector functions rather than the usual single output artificial neuron. For example, let the vector functions $\mathit{affine}^{N,M}_{\mathbf{w}}$ and $\mathit{logistic}^{N}$ be defined as

$$\mathit{affine}^{N,M}_{\mathbf{w}}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$$

$$\mathit{logistic}^{N}(\mathbf{x}) = \left[ \frac{1}{1+e^{-x_i}} \right]_{i=1}^{N\ \ T}$$

where $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{A}$ is matrix of size $M \times N$ and $\mathbf{b}$ is a vector of $M$ components. Both $\mathbf{A}$ and $\mathbf{b}$ are embedded in the parameter vector $\mathbf{w} \in \mathbb{R}^{M\times N+M}$. A standard network with an input of three components, a hidden layer of four neurons and an output layer of two neurons with linear activations, can be represented by the composite function

$$\mathit{affine}^{3,4}_{\mathbf{w_1}} \circ \mathit{logistic}^{4} \circ \mathit{affine}^{4,2}_{\mathbf{w_2}}$$

which is also a vector function that can be evaluated as

$$\mathit{affine}^{4,2}_{\mathbf{w_2}}(\mathit{logistic}^{4}(\mathit{affine}^{3,4}_{\mathbf{w_1}}(\mathbf{x})))$$

The example above illustrates a simple case of function composition. It can be argued that this kind of representation is less flexible than a neuron based one. Particularly, it is not well suited to growing and pruning algorithms [25]–[27], where the very structure of the network is dynamically adapted. However, the great majority of practical ANN are structured in homogeneous processing layers easily described with this functional scheme.

We present a generalized functional approach to create learning machines that can be used as an alternative to standard feedforward ANN. Other complex neural architectures are also considered, including nonfeedforward ones. We define a set of vector functions and composition operators and explain how these building blocks are used to define network structures and how they are implemented in an efficient way. Composite structures are designed to behave the same way than basic vector functions and may become part of higher level structures. All building blocks are compatible with backpropagation. Unlike the usual practice in which the training algorithm is hardwired into the learning machine, our approach separates the gradient computation. It permits to apply any first order parameter optimization algorithm regardless of the network architecture.

Similar modular approches to implement neural nets have been proposed [9], [12]. Nevertheless, their framework restricts function composition to cascade evaluation and do not state how to generalize to other dataflow layouts.

### A. Notation

Lower-case bold letters, for example $\mathbf{x}$, denote vectors, while upper-case bold letters, such as $\mathbf{M}$, denote matrices. Elements are referred as $x_i$ and $M_{i,j}$. Calligraphic upper-case letters denote finite indexed sets, as in $\mathcal{X}$, which contain $|\mathcal{X}|$ elements. The notation $\{\mathbf{x}^k\}_{k=1}^{|\mathcal{X}|}$ is used to emphasize its members, each one identified by a superscript. The term *vector* refers to both column and row vectors, as it should be clear from the context. For objects such as scalars, matrices and indexed sets, a default encoding as a vector will be assumed, for example $[M_{i,j}]_{i,j}$ with $i = 1, \ldots, n$ and $j = 1, \ldots, m$ can be rewritten as $[M_k]_k$ with $k = 1, \ldots, m \times n$. If an object has been encoded as a vector $\mathbf{y}$ and embedded into a vector $\mathbf{x}$, then $\mathbf{x}[\mathbf{y}]$ denotes those elements. For a tuple $U$, the notation $U.I$ refers to its member $I$. The dot notation $\dot{\mathbf{x}}$ stands for the error gradient with respect to $\mathbf{x}$, i.e. $\partial E/\partial \mathbf{x}$.

Along this text, the following symbols appear repeatedly

| | |
|---|---|
| $U$ | Processing unit |
| $C$ | Composite unit |
| $E$ | Error or regularized unit |
| $M$ | Learning Machine |
| $\mathcal{Q}$ | Execution sequence |
| $\mathcal{X}, \mathcal{Y}, \mathcal{D}$ | Input, output and target set |
| $\mathbf{x}, \mathbf{w}, \mathbf{y}, \mathbf{d}$ | input, parameter, output and target vectors |
| $\dot{\mathbf{x}}, \dot{\mathbf{w}}, \dot{\mathbf{y}}$ | input, parameter and output gradient vectors |

## II. Gradient based learning

There are several approaches to automatic machine learning, but much of the successful approaches can be categorized as *gradient-based learning methods* [11]. The training task consists in feeding a learning system with a training set $\mathcal{T}$ and parameters $\mathbf{w}$ and to compute an associated error cost $E$, which measures the performance of the system (Figure 1a). The aim of the training task is to adjust the parameters $\mathbf{w}$ using a gradient descent technique, so that $E$ is minimized given $\mathcal{T}$ (Figure 1b).

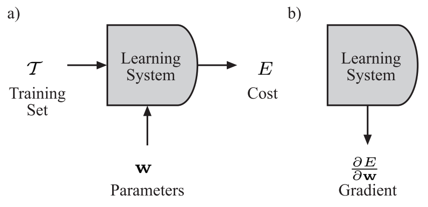

Fig. 1. Learning system

The learning system can be divided up into two components: the learning machine and the cost function. This division allows to restate the problem as follows. The training task feeds a learning machine $M$ with input vectors $\mathcal{X} = \{\mathbf{x}^i\}$, retrieves the output vectors $\mathcal{Y} = \{M(\mathbf{x}^i, \mathbf{w})\}$, and computes a cost $E$, making use of additional desired target vectors $\mathcal{D} = \{\mathbf{d}^i\}$ in the supervised case (Figure 2). The parameters are then adjusted using the cost gradient. Figure 3 depicts this idea. The end result of this setup is the trained learning machine $M$.

A learning machine may be regarded as a processing unit that implements a vector function and its gradient. A machine may be simple (atomic) or complex, consisting of the composition of several units possibly composite as well. The key here is to recognize that recursive function composition allows for arbitrarily complex (neural) structures and to exploit

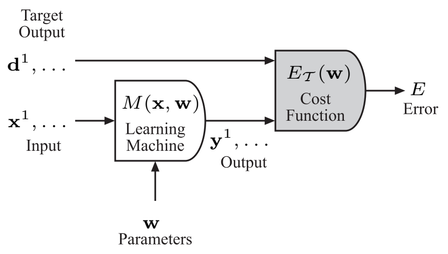

Fig. 2. Error evaluation scheme

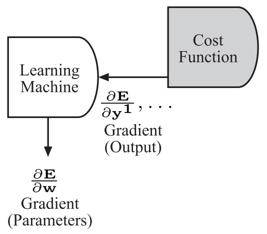
Learning Machine
Cost Function
$\frac{\partial \mathbf{E}}{\partial \mathbf{y^1}}, \ldots$
Gradient (Output)
$\frac{\partial \mathbf{E}}{\partial \mathbf{w}}$
Gradient (Parameters)

Fig. 3. Gradient evaluation scheme

this feature by designing appropriate processing units. To ensure the seamless integration of the units, they must respect additional design constraints as explained below.

### A. Compositionality and forward evaluation

Function composition can be achieved by defining appropriate functional operators. For example, consider the *Multilayer Perceptron* architecture (MLP) where processing layers are connected in cascade. This setup is viewed as a set of processing units assembled by a serial composer defined as

$$serial(U^1, U^2, \ldots, U^n) \equiv U^n(\ldots U^2(U^1(x)) \ldots)$$

where $U^1, U^2, \ldots, U^n$ are units which compute the desired vector functions. The MLP given in the previous example would then be defined by

$$serial(affine^{3,4}_{\mathbf{w_1}}, logistic^4, affine^{4,2}_{\mathbf{w_2}}). \tag{1}$$

This expression describes a composite unit consisting of three basic calculating units and a dataflow layout given by the serial composition operator (Figure 4). The resulting composite unit embodies itself a vector function, whose output depends on its subunits' outputs. This dependency can be explicited using a syntax tree (Figure 5).

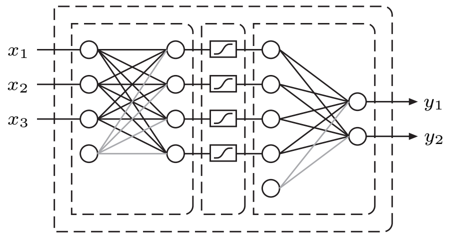

Fig. 4. MLP 3-4-2

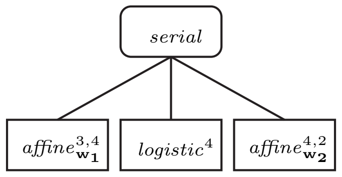

Fig. 5. MLP Tree

The output of a basic unit results from a direct evaluation of the associated vector function. In contrast, a composite unit could require the evaluation of several subunits according to its dataflow layout. A functional composition is defined through a dataflow layout that implies an evaluation order. The chained ordered evaluation from the inputs to the outputs is called the *forward propagation*. In our example, the forward propagation consists of the following steps:

1. execute $affine^{3,4}_{\mathbf{w_1}}(\mathbf{x}) \rightarrow \mathbf{v}_1$
2. execute $logistic^4(\mathbf{v}_1) \rightarrow \mathbf{v}_2$
3. execute $affine^{4,2}_{\mathbf{w_2}}(\mathbf{v}_2) \rightarrow \mathbf{y}$

where $\mathbf{v}_1$ and $\mathbf{v}_2$ are intermediate results. Here, the serial composer fixes the execution order, allocates the buffers for intermediate results and establishes the dataflow. This *execution sequence*, depicted in figure 6, is basically the compiled form of expression (1). It contains all the information needed to compute the output $\mathbf{y}$ of the serial unit given input $\mathbf{x}$.

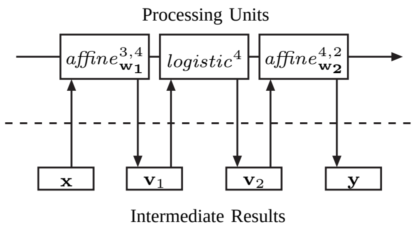

Fig. 6. Execution sequence.

The example above can be easily extended to the general case. Basic units contain all the information to carry out the computation of a vector function given an input and output buffer. Functional operators allow the construction of composite units described by expressions which are compiled to execution sequences.

## B. Error backpropagation

In addition to the previous forward evaluation procedure, the MLP needs an efficient procedure to compute the gradient of the error with respect to its weights. This is achieved by the error backpropagation algorithm [6]–[8], which is basically a practical application of the chain rule [9]. To support this algorithm, the execution sequence is extended with gradient buffers and backward procedures. For each buffer $\mathbf{v}$, a corresponding gradient buffer $\dot{\mathbf{v}}$ of the same size is created. In addition, each unit is equipped with two error backpropapation routines: the gradient evaluation procedure and the backward evaluation procedure.

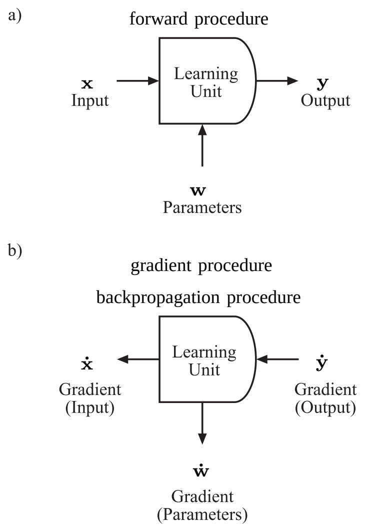

Fig. 7. (a) Forward and (b) backward evaluation

Consider a basic unit that produces output $\mathbf{y}$ from input $\mathbf{x}$ with parameters $\mathbf{w}$. Identifying

$$\dot{\mathbf{x}} = \frac{\partial E}{\partial \mathbf{x}}, \qquad \dot{\mathbf{y}} = \frac{\partial E}{\partial \mathbf{y}}, \qquad \dot{\mathbf{w}} = \frac{\partial E}{\partial \mathbf{w}}$$

and assuming that $\dot{\mathbf{y}}$ is already computed, then, applying the chain rule for partial derivatives, $\dot{\mathbf{w}}$ and $\dot{\mathbf{x}}$ are given by the following equations

$$\dot{w}_i = \sum_k \frac{\partial y_k}{\partial w_i} \dot{y}_k \tag{2}$$

$$\dot{x}_i = \sum_k \frac{\partial y_k}{\partial x_i} \dot{y}_k. \tag{3}$$

So, given $\dot{\mathbf{y}}$, a basic unit calculates $\dot{\mathbf{w}}$ by executing its gradient procedure (2), and it backpropagates the error gradient to $\dot{\mathbf{x}}$ with the backward propagation procedure (3), as depicted in figure 7.

In the case of composite units, the error gradient is obtained by following the forward execution sequence in reverse order. Continuing with our previous MLP example, the *backward propagation* sequence is

1.a execute gradient $affine^{4,2}(\dot{\mathbf{y}}) \rightarrow \dot{\mathbf{w}}_2$
1.b execute backprop $affine^{4,2}(\dot{\mathbf{y}}) \rightarrow \dot{\mathbf{v}}_2$
2.a execute gradient $logistic^4(\dot{\mathbf{v}}_2) \rightarrow null$
2.b execute backprop $logistic^4(\dot{\mathbf{v}}_2) \rightarrow \dot{\mathbf{v}}_2$
3.a execute gradient $affine^{3,4}(\dot{\mathbf{v}}_1) \rightarrow \dot{\mathbf{w}}_1$
3.b execute backprop $affine^{3,4}(\dot{\mathbf{v}}_1) \rightarrow \dot{\mathbf{x}}$

It is worth noticing that the buffers involved in this procedure are the gradient counterparts of the forward evaluation (Figure 8). Subunits are always connected through shared buffers. This invariant embedding assumption ensures the seamless integration of units. With this extension and the forward and backward propagation procedures, a composite unit represented by a syntax tree is correctly compiled to calculate its outputs and gradients.

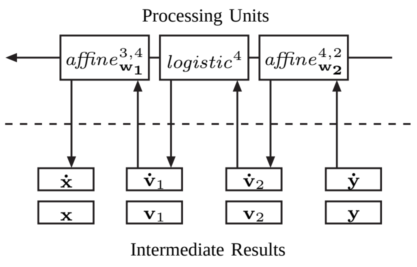

Fig. 8. Backward execution sequence.

## III. Building blocks

In order to easily create a neural architecture we propose three kinds of building blocks: basic units that implement atomic vector functions, composite units that are built from subunits, themselves possibly composite, and error or regularized units which are the source of the gradient computation. Composite units define different dataflow layouts depending on the composition operator they embody.

### A. *Basic units*

A basic unit $U$ can be minimally described as a tuple

$$U = (U_{fw}, U_{grad}, U_{bk}, I, O, \mathbf{w}, \dot{\mathbf{w}})$$

where $U_{fw}$ is the forward evaluation procedure, $U_{grad}$ the gradient calculating procedure, $U_{bk}$ the backpropagation procedure, $I$ the size of the input vector, $O$ the size of the output vector, $\mathbf{w}$ the trainable parameters (weights) of the unit and $\dot{\mathbf{w}}$ the storage for the gradient of the weights. This description is minimal in the sense that additional data may be necessary for particular basic units, for example to store non trainable parameters. The forward evaluation procedure $U_{fw}$ of a basic unit is associated to a vector function $F$ whose derivatives $\{\partial F/\partial w_i\}_{i=1}^{|\mathbf{w}|}$ and $\{\partial F/\partial x_i\}_{i=1}^{I}$ must be defined to implement $U_{grad}$ and $U_{bk}$ respectively. There is an implicit order between these procedures. Before evaluating $U_{bk}$ and $U_{grad}$ the results of $U_{fw}$ are needed.

The procedures $U_{fw}$, $U_{grad}$ and $U_{bk}$ are of the form

$$\begin{aligned} U_{fw} &= \mathit{fw}(U, \mathbf{x}, \mathbf{y}) \\ U_{grad} &= \mathit{grad}(U, \mathbf{x}, \mathbf{y}, \dot{\mathbf{y}}) \\ U_{bk} &= \mathit{bk}(U, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}}) \end{aligned}$$

where all arguments are references to external data provided by the embedding execution context: $U$ is a unit reference and $\mathbf{x}$, $\mathbf{y}$, $\dot{\mathbf{x}}$ and $\dot{\mathbf{y}}$ are the input, output, input gradient and output gradient buffers, respectively. The result of $U_{fw}$ is obtained in $\mathbf{y}$ and that of $U_{bk}$ in $\dot{\mathbf{x}}$. $U_{grad}$ modifies $\dot{\mathbf{w}}$.

In order to create adequately initialized representations of specific units, each type of unit is associated to a creation function, or *constructor* in object oriented terminology. A constructor produces the representation of a unit of the corresponding type as the result of its invocation with appropriate defining arguments.

### B. *Composite units*

A composite unit $C$ is described as a tuple

$$C = (C_{compile}, I, O, \mathcal{S})$$

where $\mathcal{S}$ is the list $U^1, U^2, \ldots U^L$ of subunits of $C$ and $C_{compile}$ the compilation procedure that creates the dataflow layout of the composite unit from $\mathcal{S}$. The $C_{fw}$, $C_{grad}$ and $C_{bk}$ procedures are implicitely defined by this layout. Composite units do not define trainable parameters $\mathbf{w}$, but they subsume the parameters of their subunits, concatenating them: $C.\mathbf{w} = [U^1.\mathbf{w}, U^2.\mathbf{w}, \ldots U^L.\mathbf{w}]$.

The list $\mathcal{S}$ represents the first level of a tree structure whose final leaves are basic units. When a learning machine is created, the tree is compiled to generate an execution sequence of appropriately ordered and contextualized invocations to basic units. All the calculating procedures of a composite unit are then ultimately implemented using the procedures associated to the leaves of the tree.

The procedure $C_{compile}$ is of the form

$$C_{compile} \quad = \mathit{compile}(C, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$$

Since composite units have subunits, the compiling process is recursive. A default $\mathit{compile}$ procedure is needed for terminal basic units:

```
$compile(U, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$ :
    $appendStep(U, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$
```

Following a standard compilation technique [28], each call to $\mathit{appendStep}$ produces a single execution step which is appended to the execution sequence of the learning machine that is being created.

### C. *Error or regularized units*

An error or regularized unit $E$ is described as a tuple

$$E = (E_{fw}, E_{grad}, E_{bk}, I, O, U)$$

where $U$ is the subunit to which the error criterium is applied (it usually encompasses the whole structure of a machine). The vector function associated to $E$ is the same than for $U$. A regularized unit can therefore be used as a building block exactly like its subunit. This equivalence is achieved transparently by using in $E$ the same buffers $\mathbf{x}$, $\mathbf{y}$, $\dot{\mathbf{x}}$ and $\dot{\mathbf{y}}$ of $U$. To ensure it, a default $\mathit{compile}$ procedure is defined for error units:

```
$compile(E, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$ :
    $compile(E.U, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$
    $appendStep(E, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$
```

where $E.U$ is a notation to indicate the $U$ component of $E$ and the call to $\mathit{compile}(E.U, \ldots)$ refers to the compiling method of $U$.

The procedures associated to a regularized unit have the same interface than those of basic units. However, the forward procedure $E_{fw}$ does not implement a vector function because the procedure $U_{fw}$ of the underlying unit is assumed to carry out this task. Instead, when the embedding machine is being trained, it calculates the error function associated to $E$. Two error types are considered in our design: errors over the outputs

$\mathbf{y}$ of $U$ and errors over its parameters $\mathbf{w}$. In the former case $E_{bk}$ adds the contribution of the error to $\dot{\mathbf{y}}$. In the latter case, $E_{grad}$ adds the contribution of the error to $\dot{\mathbf{w}}$.

The compiler generates the execution sequence of $U$ before the execution step of its parent error unit $E$. When the forward evaluation is carried out, the function associated to $U$ is calculated as expected before the error. During the backward evaluation, the error contribution to $\dot{\mathbf{y}}$ is available before the backpropagation. Errors need to be calculated only when the embedding machine is beeing trained.

### D. MLP building blocks

A MLP can be constructed relying on two types of basic vector processing units: *affine* transformations and *logistic* non-linear projections. Additionally, a *serial* connector is needed to glue these basic units to form composite multilayered structures. In order to evaluate its gradients, the MLP must also be associated to an error function.

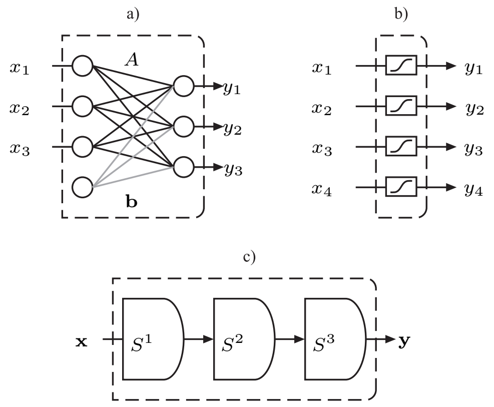

Fig. 9. MLP building blocks.

Panel a) shows an *affine* basic unit $\mathit{affine}(3,3)$, b) a *logistic* basic unit $\mathit{logistic}(4)$, and c) a serial connector $\mathit{serial}(S^1, S^2, S^3)$.

*1) Affine basic units :* The constructor for affine transformations is invoked as $\mathit{affine}(N, M)$, where $N$ is the dimension of the input of the unit to be created and $M$ that of the output. The attributes of the unit are initialized as follows:

$$\begin{aligned}&U = \mathit{affine}(N,M):\\ &\quad U \leftarrow \mathit{new}(\mathit{Affine})\\ &\quad U.I \leftarrow N\\ &\quad U.O \leftarrow M\\ &\quad U.\mathbf{w} \leftarrow \mathit{vector}(M\times N+M)\\ &\quad U.\dot{\mathbf{w}} \leftarrow \mathit{vector}(M\times N+M)\end{aligned}$$

where the parameter vector $\mathbf{w}$ and its gradient $\dot{\mathbf{w}}$ are assigned with vectors of appropriate size. The methods $U_{fw}$, $U_{grad}$ and $U_{bk}$ are the same for every unit of the $\mathit{Affine}$ type. The operation $\mathit{new}(\mathit{Affine})$ is responsable of linking these methods to the particular instance that is beeing created. Assuming a standard mapping of the parameters into a $O \times I$ matrix $\mathbf{A}$ and a $O$ vector $\mathbf{b}$, the methods are defined as follows:

$$\begin{aligned}&\mathit{fw}(U,\mathbf{x},\mathbf{y}):\\ &\quad \mathbf{y} \leftarrow \mathbf{A}\mathbf{x}+\mathbf{b}\\ \\ &\mathit{grad}(U,\mathbf{x},\mathbf{y},\dot{\mathbf{y}}):\\ &\quad \dot{\mathbf{w}}[\mathbf{A}] \leftarrow \dot{\mathbf{w}}[\mathbf{A}] + \dot{\mathbf{y}}\mathbf{x}^T\\ &\quad \dot{\mathbf{w}}[\mathbf{b}] \leftarrow \dot{\mathbf{w}}[\mathbf{b}] + \dot{\mathbf{y}}\\ \\ &\mathit{bk}(U,\mathbf{x},\mathbf{y},\dot{\mathbf{x}},\dot{\mathbf{y}}):\\ &\quad \dot{\mathbf{x}} \leftarrow \mathbf{A}^T\dot{\mathbf{y}}\end{aligned}$$

where the notation $\mathbf{v}[\mathbf{M}]$ refers to ‘the locations in vector $\mathbf{v}$ associated to the values of matrix $\mathbf{M}$’.

In spite that some arguments of the methods are not used in this particular case, they are kept for the sake of uniformity. The guiding design principle is that, externally, all types of units have the same interface.

The gradient of basic units is calculated by adding the contribution of the current training example to $\dot{\mathbf{w}}$. Provided $\dot{\mathbf{w}}$ is given an initial zero value, the gradient for a set of examples is obtained by merely executing the calculations successively with each one of them.

*2) Logistic basic units :* The constructor for logistic units needs only one argument since its inputs and outputs have the same dimension:

$$\begin{aligned}&U = \mathit{logistic}(N):\\ &\quad U \leftarrow \mathit{new}(\mathit{Logistic})\\ &\quad U.I \leftarrow N\\ &\quad U.O \leftarrow N\end{aligned}$$

The methods are defined as follows:

$$\begin{aligned}&\mathit{fw}(U,\mathbf{x},\mathbf{y}):\\ &\quad \mathbf{y} \leftarrow [1/(1+e^{-x_i})]_{i=1}^{O}{}^{T}\\ \\ &\mathit{grad}(U,\mathbf{x},\mathbf{y},\dot{\mathbf{y}}):\\ &\quad \text{null method}\\ \\ &\mathit{bk}(U,\mathbf{x},\mathbf{y},\dot{\mathbf{x}},\dot{\mathbf{y}}):\\ &\quad \dot{\mathbf{x}} \leftarrow [\dot{y}_i y_i (y_i-1)]_{i=1}^{I}{}^{T}\end{aligned}$$

where the null gradient method is needed for interface compatibility.

*3) Serial composite units :* The constructor for serial composite units is invoked with a list of subunits to be connected. It generates a tree structure that must be compiled before execution. The constructor is defined as:

$$\begin{aligned}&C = \mathit{serial}(U^1,U^2,\ldots,U^L):\\ &\quad C \leftarrow \mathit{new}(\mathit{Serial})\\ &\quad C.I \leftarrow U^1.I\\ &\quad C.O \leftarrow U^L.O\\ &\quad C.\mathcal{S} \leftarrow U^1,U^2,\ldots,U^L\end{aligned}$$

where each $U^k$ is a previously created unit. The compilation method for serial units links the output of each subunit to the input of its successor in the list. This is achieved by using buffers to store intermediate results as shown below:

```
$compile(C, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$ :
    $\mathbf{v}^1 \leftarrow vector(U^1.O)$ ; $\dot{\mathbf{v}}^1 \leftarrow vector(U^1.O)$
    $compile(U^1, \mathbf{x}, \mathbf{v}^1, \dot{\mathbf{x}}, \dot{\mathbf{v}}^1)$
    $\mathbf{v}^2 \leftarrow vector(U^1.O)$ ; $\dot{\mathbf{v}}^2 \leftarrow vector(U^1.O)$
    $compile(U^2, \mathbf{v}^1, \mathbf{v}^2, \dot{\mathbf{v}}^1, \dot{\mathbf{v}}^2)$
    $\ldots$
    $\mathbf{v}^{L-1} \leftarrow vector(U^1.O)$ ; $\dot{\mathbf{v}}^{L-1} \leftarrow vector(U^1.O)$
    $compile(U^{L-1}, \mathbf{v}^{L-2}, \mathbf{v}^{L-1}, \dot{\mathbf{v}}^{L-2}, \dot{\mathbf{v}}^{L-1})$
    $compile(U^L, \mathbf{v}^{L-1}, \mathbf{y}, \dot{\mathbf{v}}^{L-1}, \dot{\mathbf{y}})$
```

where each $U^k$ belongs to the $S$ list of $C$ and the call to $compile(U^k, \ldots)$ refers to the compiling method of $U^k$.

*4) Mean square error units :* The constructor for mean square error units allows to associate this error to an underlying unit which represents a whole ANN structure. The constructor is defined as:

```
$E = mse(U)$ :
    $E \leftarrow new(Mse)$
    $E.I \leftarrow U.I$
    $E.O \leftarrow U.O$
    $E.S \leftarrow U$
```

From the point of view of the vector function implemented, the newly created $mse$ unit $E$ behaves exacly like its underlying unit $U$. However the methods of $E$ define how to calculate the error and its contribution to the gradient:

```
$fw(E, \mathbf{x}, \mathbf{y})$ :
    $Error \leftarrow Error + \frac{1}{2N} \sum_{i=1}^{O} (d_i - y_i)^2$

$grad(E, \mathbf{x}, \mathbf{y}, \dot{\mathbf{y}})$ :
    null method

$bk(E, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$ :
    $\dot{\mathbf{y}} \leftarrow \dot{\mathbf{y}} + (\mathbf{y} - \mathbf{d})/N$
```

where $Error$ is a global accumulator for the error, $N$ is the total number of training examples $|\mathcal{T}|$ and $\mathbf{d}$ is the desired value for $\mathbf{y}$ in $\mathcal{T}$. The variables $Error$, $N$ and $\mathbf{d}$ are global and must be initialized by the training task. The error accumulator allows to include several error or regularization terms within a given network: they are all summed up. The gradient method of the $mse$ unit is null because this error does not directly depend on the weights $\mathbf{w}$ of $U$.

## E. Machine construction

A backpropagation learning machine $M$ is described as a tuple

$$M = (U, \mathcal{Q}, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}})$$

where $\mathcal{Q}$ is the sequence of compilation steps of the machine and $\mathbf{x}$, $\mathbf{y}$, $\dot{\mathbf{x}}$ and $\dot{\mathbf{y}}$ are global buffers for the input, output, input gradient and output gradient, respectively. A machine is created from a given unit $U$ which describes its structure. The constructor is defined as:

```
$M = machine(U)$ :
    $M \leftarrow new(Machine)$
    $M.U \leftarrow U$
    $M.\mathbf{x} \leftarrow vector(U.I)$
    $M.\mathbf{y} \leftarrow vector(U.O)$
    $M.\dot{\mathbf{x}} \leftarrow vector(U.I)$
    $M.\dot{\mathbf{y}} \leftarrow vector(U.O)$
    $M.\mathcal{Q} \leftarrow$ void
    $compile(U, M.\mathbf{x}, M.\mathbf{y}, M.\dot{\mathbf{x}}, M.\dot{\mathbf{y}})$
```

The global buffers are initialized with vectors of appropriate size and the compilation is recursively carried out to create in $\mathcal{Q}$ the sequence of steps defined for the unit $U$. In our previous MLP example, extended to include the $mse$ error,

$$mse(serial(affine(3,4), logistic(4), affine(4,2)))$$

the corresponding generated sequence is:

1. $affine(3,4), \mathbf{x}, \mathbf{v}_1, \dot{\mathbf{x}}, \dot{\mathbf{v}}_1$
2. $logistic(4), \mathbf{v}_1, \mathbf{v}_2, \dot{\mathbf{v}}_1, \dot{\mathbf{v}}_2$
3. $affine(4,2), \mathbf{v}_2, \mathbf{y}, \dot{\mathbf{v}}_2, \dot{\mathbf{y}}$
4. $mse, \mathbf{x}, \mathbf{y}, \dot{\mathbf{x}}, \dot{\mathbf{y}}$

The forward and backward propagation are carried out by two default methods $M_{fw}$ and $M_{bk}$. They depend on the stored compilation sequence $M.\mathcal{Q}$, where each $Q^k \in M.\mathcal{Q}$ is of the form $(U^k, \mathbf{u}^k, \mathbf{v}^k, \dot{\mathbf{u}}^k, \dot{\mathbf{v}}^k)$. The methods are defined as follows:

```
$fw(M, \mathbf{x}, \mathbf{y})$ :
    $M.\mathbf{x} \leftarrow \mathbf{x}$
    for $1 \leqslant k \leqslant |\mathcal{Q}|$
        $fw(U^k, \mathbf{u}^k, \mathbf{v}^k)$
    $\mathbf{y} \leftarrow M.\mathbf{y}$

$bk(M)$ :
    for $|\mathcal{Q}| \geqslant k \geqslant 1$
        $grad(U^k, \mathbf{u}^k, \mathbf{v}^k, \dot{\mathbf{v}}^k)$
        $bk(U^k, \mathbf{u}^k, \mathbf{v}^k, \dot{\mathbf{u}}^k, \dot{\mathbf{v}}^k)$
```

The forward propagation copies the input vector to the input buffer of the machine, then it executes the forward procedure for each step of the sequence, in order, and copies the output buffer of the machine to the output vector. The backward propagation is carried out after the forward pass. It updates the weight gradient and backpropagates the input gradient for each step of the sequence, in reverse order.

## IV. Extended architectures

The units previously presented illustrate our implementation scheme. Of course, other popular neural architectures can be implemented from custom made units defined within this framework. New units integrate seamlessly as long as they conform to the same functional composition scheme. Complex network architectures, tailored to specific pattern recognition tasks, can be created from simple building blocks, opening a wide range of applications within a single unified framework.

To further illustrate the flexibility of this approach, let us consider a few neural architectures which are difficult to implement in practice.

### A. Higher-order neurons

The basic idea behind classifying with a multi-layer perceptron is to use a number of perceptrons, where each one implements a linear decision plane, and to combine them to approximate the decision boundary of the different classes. Although it has been shown that a MLP is an universal approximator [29], [30], depending on the structure of the data it may be advantageous to use neurons with non-linear 'decision surfaces' to separate classes. These so-called *higher-order* neurons typically achieve complex decision boundaries at the cost of increasing their computational complexity [31]–[33].

Here we outline how to implement a simple variant of higher-order neurons whose decision surfaces are quadratic forms. Consider a layer of 3 input and 3 output neurons, where the output $\mathbf{y}$ is given by

$$\mathbf{y} = logistic(\mathbf{z}) \tag{4}$$

and where $\mathbf{z}$ is

$$\mathbf{z} \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1,10} \\ a_{21} & a_{22} & \dots & a_{2,10} \\ a_{31} & a_{32} & \dots & a_{3,10} \end{pmatrix} \underbrace{\begin{pmatrix} 1 \\ \hline x_1 \\ x_2 \\ x_3 \\ \hline x_1^2 \\ x_2^2 \\ x_3^2 \\ \hline x_1x_2 \\ x_1x_3 \\ x_2x_3 \end{pmatrix}}_{\mathbf{x}'} \tag{5}$$

As shown in (5), vector $\mathbf{x}'$ contains constant, linear and quadratic terms. This layer of higher-order neurons resembles a standard neuron layer in that it performs an affine transform followed by a logistic function. The difference lies in the input vector, which has extra quadratic terms. These are easily appended by copying the input vector $\mathbf{x}$, computing its quadratic terms, and joining both linear and quadratic terms into a single vector $\mathbf{x}'$.

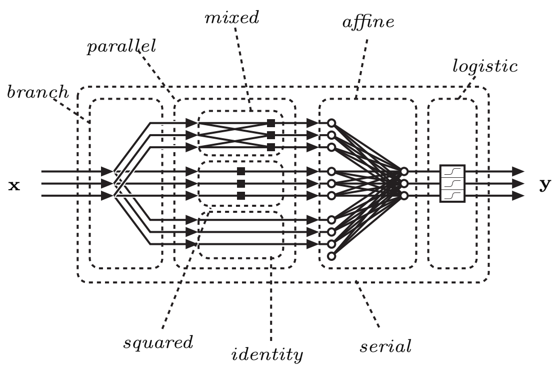

Fig. 10. 3-input 3-output quadratic neuron layer.

The resulting architecture is depicted in figure 10, where five new units have been introduced. The $branch$ basic unit produces an enlarged output by concatenating several copies of its input. The basic units $square$ and $mixed$ compute the squares and mixed products of its inputs, respectively. The $parallel$ composite unit combines several subunits in parallel, thus applying different functions on each vector window. The $identity$ basic unit is self-explanatory.

Constructors for these units are as follows: $branch(N, L)$ produces $L$ copies of its $N$-sized input vector; $squared(N)$ calculates $N$ squared terms; $mixed(N)$ produces $N(N-1)/2$ mixed products from its $N$ inputs; $identity(N)$ builds an identity function of $N$ inputs; $parallel(U^1, \dots, U^L)$ applies units $U^1$ to $U^L$ in parallel.

With this clarification, a layer of quadratic neurons with $N$ inputs and $M$ outputs can be represented by the expression

$$\begin{aligned} & serial( \\ & \quad branch(N, 3), \\ & \quad parallel(\ mixed(N), squared(N), identity(N)\ ), \\ & \quad affine(N(N-1)/2 + 2N, M), \\ & \quad logistic(M)\ ). \end{aligned}$$

This composite expression may be referred to as $quadratic(N, M)$ by noticing that it depends only on $N$ and $M$, thus allowing the rapid construction of higher-order neuron layers.

### B. Repetition

A simple kind of recursion is given by the iterative application of the same function. More specifically, consider the functional operator $repeat$ defined by

$$repeat(f, N) = \underbrace{f \circ f \circ \dots \circ f}_{N \text{ times}}$$

where $f$ is a vector function with inputs and outputs of same size and $N$ is the number of times that $f$ is applied in cascade. As defined, a $repeat$ composite unit is a particular case of a $serial$ composite unit, since

$$repeat(U, N) = serial(\underbrace{U, U, \dots, U}_{N \text{ times}})$$

where $U$ is a unit implementing a vector function (Figure 11). Interestingly, the compiled expression generates automatically a machine that implements *weight sharing* [19], because all partial contributions to the weight gradient (generated on each backward pass through unit $U$) are summed up.

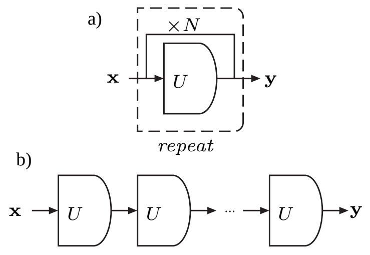

Fig. 11. A $repeat$ composite unit.

The $repeat$ composite unit shown in panel (a) behaves like a $serial$ composite unit, applying a given unit $U$ several times in cascade (b).

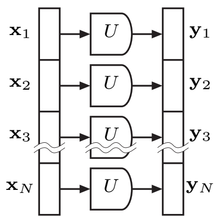

Fig. 12. A *scan* composite unit.

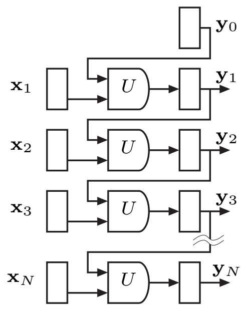

Fig. 13. Extension for temporal processing.

A *repeat* composite unit is not exactly a relaxation-type recurrent network because the relaxation depth is fixed. However, when those kind of networks are trained with gradient descent, they are partially unfolded to generate a non-recurrent aproximate architecture that can be trained using the usual algorithms. In our approach, a simple redefinition of the forward procedure, to be used only after training, allows to implement relaxation.

## C. Scanning

Another class of architectures that repeatedly apply a single function are neural networks which perform a convolution on its input, such as *FIR Multilayer Perceptrons* [16], [17] and *Convolutional Neural Networks* [18]. Basically, both can be thought as applying a function on a window that moves along a single input vector. Based on this observation, let us derive a simple implementation.

Without loss of generality, consider a functional operator $scan$, which given a function $f$ and an input $\mathbf{x}$ computes

$$scan(f, \mathbf{x}) = \begin{pmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_N) \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{pmatrix} = \mathbf{y} \qquad (6)$$

where $f(\mathbf{x}_i) = \mathbf{y}_i$ and $\mathbf{x}$ is the concatenation of subvectors $\mathbf{x}^1$ to $\mathbf{x}^N$ each of the same size. Similarly as $repeat$ composite units, a $scan$ composite unit can be expressed in terms of a $parallel$ unit as follows

$$scan(U, N) = parallel(\underbrace{U, U, \ldots, U}_{N \text{ times}}).$$

The $scan$ functional operator provides a powerful glue to build highly parallel architectures (Figure 12). A wide class of architectures can be built by feeding the $scan$ composite unit using a projection unit that extracts local features and concatenate them in a single vector. Local features typically consist of a 1-D or 2-D window over the input vector (assumed to be appropiately encoded). Let $buildviews(\ldots, N)$ denote the constructor of such a projection unit that builds a concatenation of $N$ vector views from its input, then the expression

$$serial(buildviews(\ldots, N), scan(U, N))$$

implements a convolutional architecture. It is worth emphasizing that the embedded filtering unit $U$ is arbitrarily complex.

Temporal recurrence is implemented as an extension to $scan$ (Figure 13). Time is expanded spatially: the input represents the whole time series and the output the result series. Thus, the training set consist of a single input-target pair. Recurrence is implemented by letting the underlying unit $U$ to receive as input the outputs from previous applications of $U$ during the scanning procedure (assumed to be sequential). This scheme is a form of *backpropagation through time* [16], [17]. Note that this architecture has been reduced to a common backpropagation case. There is no need for a specific training algorithm.

## D. Parameter injection

Some architectures require a direct access to view or manipulate their parameters. For instance, the approach proposed in [34] details a self-referential network architecture that can 'speak' about its own weight matrix in terms of activations. Examples like this are cases of *role switching*, where the inputs/outputs of a unit act as another unit's parameters, or *vice versa*.

This can be achieved by parameter injection/ejection operators (Figure 14). An $injector$ composite unit splits its input $\mathbf{x}$ into two parts $\mathbf{x}'$ and $\mathbf{w}$, which are repectively the input and parameter vectors of its subunit. Similarly, an $ejector$ composite unit concatenates the output of its subunit $\mathbf{y}'$ together with its weights $\mathbf{w}$ to form $\mathbf{y}$. Although the implementation of these composite units is simple, some care has to be taken in order to define appropriate forward and backward procedures.

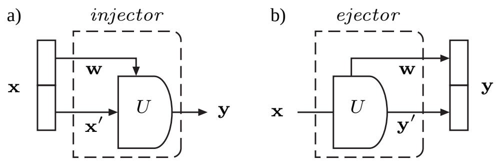

Fig. 14. Weight injector and ejector.

## E. Regularizers

Complex non-linear neural network models often have an excess of free parameters which tend to generate mappings with a lot of curvature and structure. This phenomenon, known as the *bias/variance dilemma*, arises as the result of overfitting to the noise of the data and leads to poor generalization [1], [20].

To overcome this problem, regularization techniques have been developed, which encourage smoother network mappings by adding a penalization term to the error function [21]–[24]. Althought it has been found empirically that they can lead to significant improvements in network generalization, their use is not widespread, mainly due to their difficult implementation.

*Weight Decay*, one of the simplest regularizers, consists of the sum of squared weights scaled by a decay constant $\alpha$, penalizing large weights:

$$\frac{\alpha}{2}\sum_i w_i^2. \tag{7}$$

This penality is easily stated as a regularizer operator, thus allowing it to be applied to any underlying unit, contributing to the global error without changing the unit's output.

Let $weightdecay(U, \alpha)$ be the associated constructor, taking the weights of a unit $U$ with decay constant $\alpha$. The example MLP in figure 4 could then be regularized layer-wise as

$$\begin{array}{l} mse( \\ \quad serial( \\ \quad\quad weightdecay(\mathit{affine}(3,4), \alpha_1), \\ \quad\quad logistic(4), \\ \quad\quad weightdecay(\mathit{affine}(4,2), \alpha_2) \\ \quad )). \end{array}$$

This expression produces a MLP whose global error is given by

$$\tilde{E} = E + \frac{\alpha_1}{2}\sum_i w_i^2 + \frac{\alpha_2}{2}\sum_j {w'}_j^2.$$

where $E$ is the mean square error, $w_i$ and $w'_j$ are the weights of the first and second layer respectively.

### F. Other neural architectures

The preceeding examples present implementations of important concepts found in the literature, but the list is incomplete. More examples that have been left out but are straightforward to implement, are:

- *Units*: hyperbolic tangent and softmax activation functions, cross entropy error function, parametric models.
- *Unsupervised learning*: training without target vectors, dimensional reduction techniques using appropriately defined regularizers.
- *Mixtures and comitees*: mixtures of experts, comitees of networks.
- *Kernel methods*: kernel regression, radial basis function networks with trainable centers.

## V. Implementation and testing

To test our design, we developed a Java library and a C stand-alone implementation. The Java library has been integrated to Matlab allowing a flexible manipulation and rapid development.

We present below a selection of experiments using the Java library which have been carried out on a PC with an AMD Athlon 2400 Mhz processor and 512 Mb RAM. The objective of the experiments is to validate the correct definition and compilation of complex networks. Our results confirm that the proposed framework leads to well defined machines trainable with backpropagation.

### A. Tight-encoder

The encoder/decoder problem is a well known test case for neural networks. It is usually implemented using a MLP N-M-N architecture. The weights are adjusted using an autoassociative training where the desired output is the same as the input. The training set contains the $N$ canonical basis vectors of the input space. The encoding is achieved as the result of the middle layer. When $M = log_2 N$ the encoder is called 'tight' because this is the minimal size for an optimal binary encoding of the training set (however the network may use a non binary encoding). Here we implemented the 256-8-256 tight-encoder. The resulting MLP architecture has $4360$ trainable parameters. The expression that defines the encoder is given by

$$\begin{array}{l} cross\text{-}entropy( \\ \quad serial( \\ \quad\quad \mathit{affine}(256,8), \\ \quad\quad logistic(8), \\ \quad\quad \mathit{affine}(8,256), \\ \quad\quad logistic(256) \\ \quad )). \end{array}$$

where $cross\text{-}entropy(U)$ computes the cross-entropy error over the outputs of unit $U$ [1], [3]. Minimizing this error function is equivalent to minimize the *Kullback-Leibler distance* if the outputs are interpreted as modelling a probability distribution, which is convenient in this case.

Table I shows the results for 10 independent test runs after 50 epochs using the *resilient backpropagation* training algorithm [14]. Figure 15 illustrates the classification of the network before and after training, togheter with the training error evolution.

The results show that the compiled machines rapidly learn an efficient encoding by a gradient based parameter optimization algorithm.

TABLE I
Tight-encoder results.

| | |
|---|---|
| Mean error [bits] | $1.1098 \pm 0.3058$ |
| Correct classification [%] | $95.70 \pm 0.0157$ |
| Execution time [s] | $8.656 \pm 0.1223$ |

### B. Iris data

The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant (*Iris Setosa*, *Iris Versicolour* and *Iris Virginica*). One class is linearly separable from the other 2, while the latter two are not linearly separable from each other [35], [36]. Features are sepal length, sepal width, petal length and petal width in centimeters.

In this case, a single layer of 3 quadratic neurons has been used:

$$cross\text{-}entropy(quadratic(4,3))$$

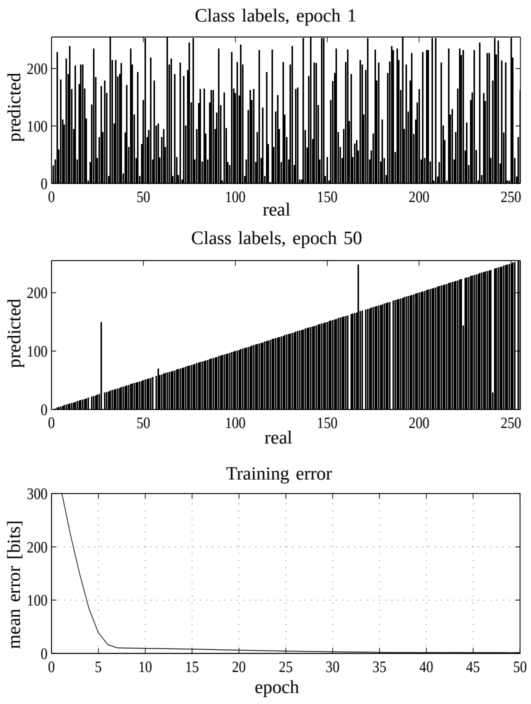

Fig. 15. Tight-encoder classification and training error during 50 epochs.

The quadratic layer expands to the nested architecture of simpler processing units described in subsection IV-A. As in the previous experiment, the chosen error function was the cross-entropy error. Table II shows the results for 10 randomly initialized test runs after 300 epochs using the *resilient back-propagation* training algorithm [14]. The resulting confusion matrix for a typical run is shown in table III.

From the traning error graph illustrated in figure 16, we can validate the compilation of the complex underlying architecture. The compositional operators define machines that seamlessly compute their forward and backward evaluations.

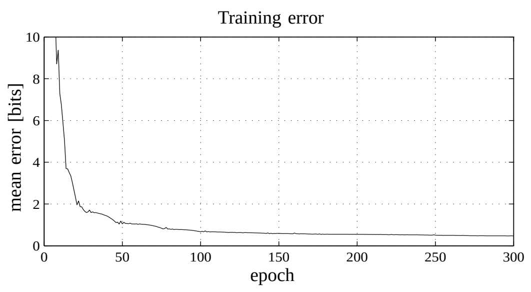

Fig. 16. Training error for the Iris data set.

## C. Additional remarks

In our implementation the network parameters can be frozen by using a mask over the gradient vector. The masking technique defines a constant mask vector $\mathbf{m}$ that is multiplied

TABLE II
IRIS SET RESULTS.

| | |
|---|---|
| Mean error [bits] | $0.5056 \pm 0.1390$ |
| Correct classification [%] | $98.13 \pm 0.0042$ |
| Execution time [s] | $1.2719 \pm 0.0475$ |

TABLE III
CONFUSION MATRIX FOR IRIS SET.

| real \ estimated | Setosa | Versicolour | Virginica |
|---|---|---|---|
| Iris Setosa | 49 | 1 | 0 |
| Iris Versicolour | 1 | 49 | 0 |
| Iris Virginica | 0 | 0 | 50 |

element-wise with $\dot{\mathbf{w}}$ before the update is performed. This technique allows the definition of a degree of sensitivity of the parameters to changes. The parameters are frozen by setting $\mathbf{m} = \mathbf{0}$.

The compilation of some complex architectures could require a prohibitive amount of storage space for the generated execution sequence and the corresponding intermediate buffers. There is trade-off between time and space. Either the intermediate results are stored or they can be regenerated on demand. The latter solution is to be preferred when there is not enough space in memory. This problem arises for example in convolutional neural networks, specially when the filtering unit itself is complex and memory demanding. A small footprint is also relevant for an optimal use of cache memory.

## VI. CONCLUSIONS

We developed a flexible computational design to build gradient-based learning machines. The scheme has been successfully implemented and tested on complex architectures. This framework is a step towards the generalization of gradient-based training. It extends the applicability of back-propagation to a wide class of learning machines, namely, continuous vector functions with computable gradients. This includes much of the existent neural architectures, avoiding the need for *ad-hoc* training algorithms. Furthermore, nonfeed-forward architectures are also considered, provided a partial unfolding of their structure is sufficient for training, which is the common case. The major achievements can be summarized as follows:

1) *Modular building design*: Complex machines, tailored to solve specific learning tasks, can be created using simple building blocks. Invariant design constraints associated to composition rules ensure the seamless integration of subunits, so that the resulting learning system is capable of computing its gradient in a recursive way.
2) *Uncoupling of gradient computation*: The parameter gradient calculation procedure has been modularized, allowing the application of the same first order training algorithms regardless of the machine architectural complexity.
3) *Standarization of architecture description*: Our framework proposes an operator based expression language to describe neural architectures in a rapid, compact and orthogonal way.

4) *Easy regularization*: Different types of regularization terms may be inserted at arbitrary points within a network to provide more stable training results and better generalization.
5) *Efficient implementation*: The compiling procedures defined for composition operators allow for a fast computation of the gradient.

## ACKNOWLEDGEMENTS

The authors would like to thank Gonzalo Ruz and Michel Tesmer for reviewing this text and for their fruitful comments. This work was supported by FONDECYT (grant 1030141).

## REFERENCES

[1] Bishop C. M., "Neural Networks for Pattern Recognition", *Oxford University Press*, 1995.

[2] Simon Haykin, "Neural Networks: A Comprehensive Foundation", *Macmillan College Publishing Company*, 1994.

[3] Theodoridis S., Koutroumbas K., "Pattern Recognition", *Academic Press*, 1999.

[4] Ripley, B.D., "Pattern Recognition and Neural Networks", *Cambridge University Press*, Cambridge, 1996.

[5] McCulloch, W. S., and Pitts, W., "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics*, 5:115-137, 1943.

[6] Bryson, A.E.Jr., Ho, Y.C., "Applied optimal control", Blaisdel Publishing Company, 1969.

[7] Werbos, P.J., "Beyond regression: New tools for prediction and analysis in the behavioral sciences", Ph.D. thesis, Harvard University, Cambridge, MA, 1974.

[8] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning internal representations by error propagation". In Rumelhart, D. E. and McClelland, J.L., editors, *Parallel Distributed Processing*, Volume 1, chapter 8, pp. 318-362. MIT Press, Cambridge, Massachusetts, 1986.

[9] Bottou, L., "Une approche théorique de l'apprentissage connexionniste: applications à la reconnaissance de la parole", Ph.D. thesis, Université de Paris XI, Orsay, France, 1991.

[10] LeCun, Y., "A theoretical framework for back-propagation". In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pp. 21-28, CMU, Pittsburgh, PA, 1988. Morgan Kaufmann.

[11] Y. LeCun, L. Bottou, G.B. Orr and K.-R. Müller, "Efficient backprop", in "Neural Networks – Tricks of the Trade", *Springer Lecture Notes in Computer Sciences 1524*, pp. 5-50, 1998. `http://citeseer.ist.psu.edu/lecun98efficient.html`

[12] Collobert, R., Bengio, S. and Mariéthoz J., "Torch: a modular machine learning software library", *Technical Report IDIAP-RR*, pp. 02-46, IDIAP, 2002.

[13] Saarinen, S., Bramley, R.B. and Cybenko, G., "Neural networks, backpropagation, and automatic differentiation". *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, In Griewank, A. and Corliss, G.F., editors, pp. 31-42, Philadelphia, PA, SIAM, 1992.

[14] Riedmiller M., Braun H., "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", Neural Networks for Pattern Recognition", *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco, CA, March 28-April 1, 1993.

[15] Hestenes, M.R. and Stiefel, E., "Methods of conjugate gradients for solving linear systems". *Journal of Research of the National Bureau of Standards*, Vol. 49 (6), pp. 409-436, 1952.

[16] Wan, E.A., "Temporal backpropagation for FIR neural networks", *IEEE International Joint Conference on Neural Networks*, Vol. 1, pp. 575-580, San Diego, CA, 1990.

[17] Wan, E.A., "Temporal backpropagation: An efficient algorithm for finite impulse response neural networks". In Touretzky, D.S., Elman, J.L., Sejnowski, T.J., and Hinton, G.E., editors, *Proceedings of the 1990 Connectionist Models Summer School*, pp. 131-140, San Mateo, CA, Morgan Kaufmann, 1990.

[18] LeCun, Y. and Bengio, Y., "Convolutional networks for images, speech, and time series", *The handbook of brain theory and neural networks*, pp. 255-258, MIT Press, Cambridge, Massachusetts, 1998.

[19] Shawe-Taylor, J. "Introducing invariance: a principled approach to weight sharing", *Proceedings of the IEEE International Conference on Neural Networks*, Congress on Computational Intelligence, pp. 345-349, Orlando, FL, 1994.

[20] Geman, S., Bienenstock, E. and Doursat, R., "Neural Networks and the Bias/Variance Dilemma", *Neural Computation*, Vol. 4, pp. 1-58, 1992.

[21] Hinton, G.E., "Connectionist learning procedures". *Technical Report CMU-CS-87-115*, Carnegie-Mellon University, Pittsburgh, PA, 1987.

[22] Krogh, A. and Hertz, J. A., "A simple weight decay can improve generalization". In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors. "Advances in Neural Information Processing Systems", Vol. 4, pp 450-957, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[23] Barlett, P.L., "For valid generalization, the size of the weights is more important than the size of the network". In Mozer, M.C., Jordan, M.I., and Petsche, T., editors. *Advances in Neural Information Processing Systems*, Vol. 9, pp. 134-140, The MIT Press, Cambridge, MA, 1997.

[24] Weigend, A. S., Rumelhart, D. E., & Huberman, B. A., "Generalization by weight-elimination with application to forecasting". In R. P. Lippmann, J. Moody, & D. S. Touretzky, editors. *Advances in Neural Information Processing Systems*, Vol. 3, San Mateo, CA. Morgan Kaufmann, 1991.

[25] Lee, T.-C., Peterson, A.M., and Tsai, J.-C., "A multi-layer feed-forward neural network with dynamically adjustable structures". *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 367-369, Los Angeles, CA, 1990.

[26] LeCun, Y., Denker, J.S., and Solla, S.A., "Optimal brain damage". *Advances in Neural Information Processing Systems 2*, In Touretzky, D.S., editor, pp. 598-605, San Mateo, CA. Morgan Kaufmann, 1990b.

[27] Hassibi, B., Stork, D.G., and Wolff, G.J., "Optimal brain surgeon and general network prunning". *IEEE International Conference on Neural Networks*, Vol. 1, pp. 293-299, San Francisco, CA, 1993.

[28] Aho, A. V., Sethi, R., Ullman, J. D., "Compilers", *Addison Wesley*, 1986.

[29] Cybenko, G., "Approximation by superposition of a sigmoidal function". *Mathematics of Control, Signals and Systems*, Vol. 2, pp. 303-314, 1989.

[30] Hornik, K., “Approximation capabilities of multilayer feedforward neural networks”. *Neural Networks*, Vol. 4, pp. 251-257, 1990.

[31] Buchholz, S. and Sommer, G., “A hyperbolic multilayer perceptron”. *International Joint Conference on Neural Networks*, *Como*, *Italy*, Vol. 2, pp. 129-133. IEEE Computer Society Press, 2000.

[32] Lipson, H. and Siegelmann, H.T., “Clustering irregular shapes using high-order neurons”. *Neural Computation*, 12(10):2331-2353, 2000.

[33] Banarer, V., Perwass, C., Sommer, G., “The hypersphere neuron”. *ESANN’2003 proceedings - European Symposium on Artificial Neural Networks*, Vol. 4, pp. 469-474, Bruges, Belgium, 2003.

[34] Schmidhuber, J., “A self-referential weight matrix”, *Proceedings of the International Conference on Artificial Neural Networks*, Springer, pp 446-451, Amsterdam, 1993.

[35] Fisher, R. A., “The use of multiple measurements in taxonomic problems”, *Annual Eugenics*, Springer, Vol. 7, Part II, pp. 179-188, 1936; also in “Contributions to Mathematical Statistics”, John Wiley, NY, 1950.

[36] Duda, R.O. and Hart, P.E., “Pattern Classification and Scene Analysis”, (Q327.D83) John Wiley & Sons, ISBN 0-471-22361-1, pp. 218, 1973.