*Journal of Universal Computer Science, vol. 7, no. 11 (2001), 1006-1023*
*submitted: 7/6/01, accepted: 1/10/01, appeared: 28/11/01* 

# A Neural Abstract Machine

Egon Börger
Dipartimento di Informatica, Università di Pisa
boerger@di.unipi.it

Diego Sona
Dipartimento di Informatica, Università di Pisa
sona@di.unipi.it

**Abstract:** In an attempt to capture the fundamental features that are common to neural networks, we define a parameterized *Neural Abstract Machine* (*NAM*) in such a way that the major neural networks in the literature can be described as natural extensions or refinements of the *NAM*. We illustrate the refinement for feedforward networks with back-propagation training. The *NAM* provides a platform and programming language independent basis for a comparative mathematical and experimental analysis and evaluation of different implementations of neural networks. We concentrate our attention here on the computational core (*Neural Kernel NK*) and provide abstract interfaces for the other *NAM* components.
**Key Words:** neural networks, neural abstract machine, abstract state machines, distributed computation.
**Category:** D.2.1 - D.2.2 - F.1.1 - I.2.5

## 1 Introduction

There is a variety of models for neural networks, tailored to process different tasks (classification, regression, clustering, etc.), using different structures (like arrays, sequences, directed acyclic graphs [Giles and Gori(1998)]) and different training techniques [Haykin(1999), Bishop(1995)]. Although most of these models share certain basic features that are characteristic for neural networks, the simulators that have been developed to support the fine tuning of a supposedly general neural network to the particular task to be performed, typically only make use of built-in libraries for different models, instead of being based upon model similarities.

In this paper we define a parameterized Neural Abstract Machine (*NAM*) in an attempt to capture the fundamental features that are common to neural networks, in such a way that the major models in the literature can be described as natural extensions or refinements of the *NAM*, e.g. by instantiating the appropriate parameters. We illustrate the refinement for the case of feedforward networks with backpropagation training. The *NAM* can also be used as a platform and programming language independent basis for a comparative mathematical and experimental analysis and evaluation of different implementations of neural networks. We concentrate our attention on the computational core, called *Neural Kernel* (*NK*), and provide abstract interfaces for the other components.

The *NK* we define comes with abstract and parameterized classes of basic objects (e.g. computational units) and of actions to be performed (e.g. initializing the internal state of neural units, scheduling of computational tasks, etc.). The mathematical framework we use is that of Abstract State Machines (ASMs [Gurevich(1995)]), which provides the appropriate rigour together with the freedom of abstraction needed for our task. We suppose the reader to be familiar with this notion. For a textbook definition we refer to [Stärk et al.(2001)]. We exploit the enhancement of the characteristic built-in parallelism of ASMs by the composition technique introduced in [Börger and Schmid(2000)] for structuring large machines. This approach allows to view an entire submachine computation as happening simultaneously with abstract atomic (non-durative) actions, providing the possibility to reuse machine components as shown in [Börger(2001)].

## 2 Neural Networks

A *Neural Network* (NN) can be defined as a directed graph of simple computing units that are connected by weighted links. Its overall behaviour can be seen as that of a black box: after having taken an input, it does some internal computation and then yields an output. Thus for the input providing environment, a neural network computation appears as a global atomic action on the input, although the network behaviour is appropriately described by a finite sequence of atomic actions of its computing units.

These units are divided into input units (the ones which for their subcomputation take some input from the environment), output units (the ones which send their output to the environment and possibly to other units), and internal units (the other ones). Through the graph structure and the computational model, the units come with a partial order that determines the scheduling of unit computations, which are usually of data-flow type. Every computing unit may have internal states associated to it and expects input to be provided, by the environment and by the units preceding it with respect to the partial order, and to be scheduled for its computation. In a given state, when scheduled, a computing unit consumes its input by performing a computation that is considered as internal and atomic, resulting in a possible change of the internal state and a propagation of the output to the units which follow in the given partial order. For example, such a computation step for unit $k$ may take the following form:

$$net_k = \sum_{i=0}^{n} w_{ki} \cdot u_{ki} \tag{1}$$

$$y_k = f(net_k) \tag{2}$$

where $n+1$ is the number of units sending their output to unit $k$, $w_{ki}$ and $u_{ki}$ are respectively the weight and the input of the link connecting unit $i$ to unit $k$, and $f$ is the activation function, which usually is nonlinear. The result $y_k$ is propagated as input to all the following units.

The well-definedness of the global neural network step is guaranteed by the condition that per input, each unit performs only one computation, together with the common property of network graphs to be acyclic. When cycles are admitted, they are usually appropriately delayed with respect to the environmental input, but for example this does not hold for the Boltzmann machine [Haykin(1999), Chap. 11].

Neural networks have two functioning modes, which can be called *operative* and *training*. During the operative mode the data flow propagation as described above goes in the direction of the arcs, starting at the input units, until the output units are reached. For this reason we refer to this phase also as *forward propagation*. Practically all neural networks exhibit this form of forward propagation. A differentiation between different models comes through the training mode, to which we refer also as *backward propagation* mode for reasons to be explained below. In this phase the internal parameters of the network may be modified, in the example above the weights of the links connecting the elements (the neurons). The appropriate adaptation of these parameters is usually achieved via a procedure, called training algorithm. In the case of so-called supervised models, this algorithm optimizes the mean error made by the network on a set of classified examples, which are supposed to come with the information on the responses expected from the model.

A typical measure of the error is given by the following cost function:

$$J(W) = \frac{1}{2} \sum_{i \in O_u} ||d_i - y_i||^2 \tag{3}$$

where $W$ is the matrix of all weights (adaptive parameters), $y_i$ and $d_i$ are respectively the actual and the desired output of the $i$-th output element, and $O_u$ is the set of output units.

The majority of training algorithms are based on the backward propagation of error information. In *back-propagation training*, which is also called *gradient descent training*, the adaptive parameters are modified by an amount which is proportional to the gradient of the cost function and can be computed on the basis of information which is available at a computational unit and its successor units. This inverse dependency implies an inverse propagation of data from the output units to the input units [see Fig. 1]. Our abstract model for *NK* defined below comes with this forward/backward feature, which can however be appropriately refined also for unsupervised models [Sona and Sperduti(2001)].

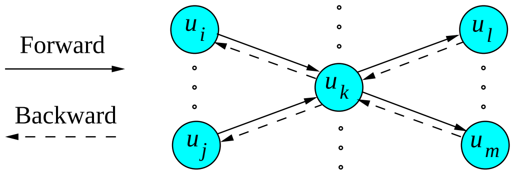



Figure 1: The bidirectional flow of data in a Neural Network trained with a Gradient Descent based algorithm

# 3 The Neural Kernel

## 3.1 States of the Neural Kernel

The Neural Kernel $NK$ interacts with the environment by taking input from it, via a 0-ary monitored function *input*, and by providing its response as output via a controlled 0-ary function *output*:

$$input : INPUT$$
$$output : OUTPUT$$

where we take *input* and *output* as sequences of abstract $DATA$, to be instantiated later:

$$INPUT : DATA*$$
$$OUTPUT : DATA*$$

As a consequence we access by appropriate abstract functions certain types of information that comes with the input and is needed to drive the network computation, for example information on the initialization of units, tokens and identifiers associated to the input, or whether the $NK$ should work in forward or backward propagation mode, etc.

$$inputType : INPUT \rightarrow INPUT\_TYPE$$
$$INPUT\_TYPE = \{forward,\ backward\}$$

This allows one to refine $INPUT$ for specialized neural networks.

To be prepared for both a synchronous and asynchronous models of interaction between the environment and $NK$, we use a shared function

$$newInputToBeConsumed : BOOL$$

through which the environment signals to the $NK$ the presence of new unused data, whereas when consuming the input, $NK$ resets that function to avoid the reuse of the same (occurrence of the) input. For the same reason we indicate the control state of the $NK$ by another Boolean valued function:

$readyForNewComputation : BOOL$

This function is controlled by $NK$ and allows the environment to know if the $NK$ is performing an internal computation or whether it has terminated its previous computation, so that the result can be found in the appropriate *output* location and that $NK$ is ready for a new input. In the next section we model *readyForNewComputation* by $modeNK = input$.

As explained in the introduction, we view neural networks as composed out of standard computational blocks, formally reflected as elements of a domain *UNIT*. As a consequence of this block diagram view [Santini et al.(1995)] [Wan and Beaufays(1998)] [Nerrand et al.(1993)] [Berthold and Fischer(1997)] [Campolucci et al.(2000)], units can be instantiated to neurons, synapses, layers of neurons, even to neural networks, but also to products of vectors and matrixes, if one wants to reflect the architectural view [Tsoi(1998b)] and [Tsoi(1998a)]. Each unit can be seen as an agent, recognizable by the $NK$ via a function that identifies all the units which currently constitute the neural network:

$network : \mathcal{P}(UNIT)$

where $\mathcal{P}$ stands for power-set. The above described input units, which constitute the input side of the interface of a Neural Network, and the analogous output units, which constitute the output side of the interface with the environment, are formalized by the following two functions:

$inputUnits : \mathcal{P}(UNIT)$
$outputUnits : \mathcal{P}(UNIT)$

The information flow among the units is reflected by two functions, both monitored by $NK$ and controlled by the environment, which intrinsically describe the *network topology*. By this separation of responsibilities one can reflect that for each new input elaboration performed by $NK$, the environment could update these functions, with the $NK$ automatically adapting itself to the new network topology. In particular, these two functions describe the units from which a given unit receives information, respectively the units which are the destination of the information produced by the given unit:

$source : UNIT \rightarrow \mathcal{P}(UNIT)$
$dest : UNIT \rightarrow \mathcal{P}(UNIT)$

We treat the input of a unit as part of its internal state, which is updated when the source units send out the result of their computation. In this way the output of a unit can be retrieved from the input locations of its destination units. This is formalized, for forward and backward input type, by two functions describing the input coming from the external environment:

$$inForward^{\mathrm{ext}} : UNIT \rightarrow DATA*$$
$$inBackward^{\mathrm{ext}} : UNIT \rightarrow DATA*$$

These two functions for external input can be defined as a function of the given unit and of (a copy of) the environmental input.

$$inForward^{\mathrm{ext}}(u) = getNetInput(u,\ inputCopy)$$
$$inBackward^{\mathrm{ext}}(u) = getNetError(u,\ inputCopy)$$

where *getNetInput* and *getNetError* are two auxiliary functions, providing the appropriate projection of the external input vector and to be instantiated when the necessity arises.

$$getNetInput : UNIT \times INPUT \rightarrow DATA*$$
$$getNetError : UNIT \times INPUT \rightarrow DATA*$$

The two functions for internal input are functions of the owner unit and of the sender unit.

$$inForward^{\mathrm{int}} : UNIT \times UNIT \rightarrow DATA$$
$$inBackward^{\mathrm{int}} : UNIT \times UNIT \rightarrow DATA$$

This separation of different input functions allows us to associate the (free) parameters—the weights of the connections—with the respective input signal to be transmitted.

### 3.2 Rules of the Neural Kernel

We define *NK* as a machine that operates in two distinct and mutually exclusive phases. In its *input* phase, *NK* waits for external data. When new input has arrived, *NK* starts the corresponding neural computation and enters its *compute* phase, in which it is isolated from the environment and performs all the unit computations until no more units can be executed (we will see below that as part of this isolation, in the rule *activateNeuralKernel NK* makes an internal copy of the input), then the *NK* returns to the waiting mode for new input, preparing itself for a new input elaboration, and possibly providing the result of the terminated data elaboration. This view of *NK* is formalized by the following ASM rule (see also the equivalent flowchart formulation in [Fig. 2], which uses the equivalence between these diagrams and ASM rules defined in [Börger(1999), Section 3.2] and [Stärk et al.(2001), pg.16 Fig.2.1]).

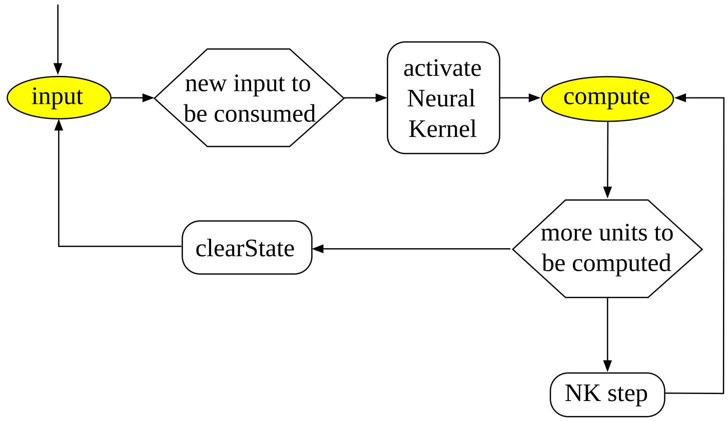



**Figure 2:** The structure of one macro step of NK

$NeuralKernel =$
  **if** $(modeNK = input \;\; and \;\; newInputToBeConsumed)$ **then**
    $modeNK := compute$
    $activateNeuralKernel$
  **if** $(modeNK = compute)$ **then**
    **if** $(moreUnitsToBeComputed)$ **then**
      *NK-Step*
    **else**
      $clearState$
      $modeNK := input$

As we are going to illustrate in the following sections, by refining this abstract high-level description, i.e. defining the submachines occurring in it, one can obtain a variety of instantiations of the $NK$, depending on the particular computational model to be described.

#### 3.2.1 Activation, Step, and Termination of *NK*

The $activateNeuralKernel$ machine has to initialize the $NK$ for the new input computation, which should not be influenced by newly arriving input. This can be guaranteed by switching to the *compute* mode and by working in this mode with an internal copy of the current input, simultaneously updating the function $newInputToBeConsumed$ (whereby it is signaled also to the environment that the new input has been read for the elaboration). Working with a copy of the input taken at the moment of starting the $NK$-subcomputation is compatible with both a synchronous and an asynchronous model of interaction between the environment and $NK$. The elaboration itself is prepared by scheduling the units that are to be executed next, in reaction to the new external input. This is formalized by the following definition:

$$
\begin{array}{l}
\mathit{activateNeuralKernel} = \\
\quad \mathit{newInputToBeConsumed} := \mathit{false} \\
\quad \mathit{copyNetInput}(\mathit{input}) \\
\quad \mathit{scheduledUnits} := \mathit{nextExecutableUnits}(\emptyset,\ \mathit{inputType}(\mathit{input}))
\end{array}
$$

where copying the environmental input can be defined as:

$$
\begin{array}{l}
\mathit{copyNetInput}(\mathit{input}) = \\
\quad \mathit{inputCopy} := \mathit{input} \\
\quad \mathit{inputType} := \mathit{inputType}(\mathit{input})
\end{array}
$$

$NK$ has terminated its current internal computation if there are no more units that need to be processed, i.e. when the set *scheduledUnits* becomes empty:

$$\mathit{moreUnitsToBeComputed} = \mathit{scheduledUnits} \neq \emptyset$$

For the updates of *scheduledUnits*, which happen during the initialization of $NK$ and at each $NK$-step (see below), we use an auxiliary unit selection function *nextExecutableUnits*, which schedules the units to be activated in the next step, taking into account the set of currently computing units and the input type. In this way we separate the scheduling and the unit computation (see below) into two independent machine components, which can then be refined in an orthogonal manner to reflect different scheduling and unit computation strategies.

Technically, the *NK-Step* rule realizes the orthogonality of unit computation and unit scheduling by exploiting the parallelism of ASMs. All the units that have to process their input information are simultaneously called for execution, and the set of units that will be processed in the next $NK$ step is determined.

$$
\begin{array}{l}
\mathit{NK\text{-}Step} = \\
\quad \textbf{forall}\ \ (u \in \mathit{scheduledUnits})\ \ \textbf{do} \\
\quad\quad \mathit{computeUnit}(u) \\
\quad \mathit{scheduledUnits} := \mathit{nextExecutableUnits}(\mathit{scheduledUnits},\ \mathit{inputType})
\end{array}
$$

The *clearState* rule has been introduced only to prepare instantiations of $NK$ where upon termination of a round, $NK$ does some specific actions. For the moment the reader may consider it as a skip statement.

$$
\begin{array}{l}
\mathit{clearState} = \\
\quad \textbf{skip}
\end{array}
$$

### 3.2.2 The Unit Computation

The *computeUnit* rule changes the state of the given unit and propagates the result of the unit computation as input to the following units or as network *output* to the environment. The state of the units can be rather complex, and our abstract formulation can be refined by a variety of neural network instantiations. Here we specify only that it can change in two different ways, depending on the input type.

$$
\begin{array}{l}
\mathit{computeUnit}(u) = \\
\quad \textbf{if}\ \ (\mathit{inputType} = \mathit{forward})\ \ \textbf{then} \\
\quad\quad \textbf{let}\ \ (\mathit{result} = \mathit{forwardValue}(u))\ \ \textbf{in} \\
\quad\quad\quad \mathit{propagateForward}(u,\ \mathit{result}) \\
\quad\quad\quad \mathit{updateLocalStateForward}(u,\ \mathit{result}) \\
\quad \textbf{if}\ \ (\mathit{inputType} = \mathit{backward})\ \ \textbf{then} \\
\quad\quad \textbf{let}\ \ (\mathit{result} = \mathit{backwardValue}(u))\ \ \textbf{in} \\
\quad\quad\quad \mathit{propagateBackward}(u,\ \mathit{result}) \\
\quad\quad\quad \mathit{updateLocalStateBackward}(u,\ \mathit{result})
\end{array}
$$

For reasons of modular design, in this rule we use abstract functions to hide the specifics of the internal unit computation that determines the result to be propagated and the resulting state change:

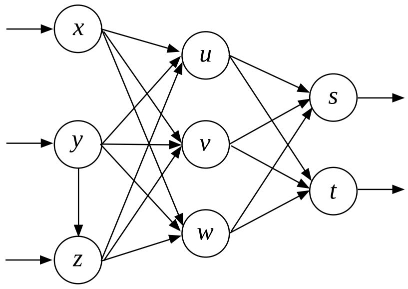
$$
\begin{array}{l}
\mathit{forwardValue} : \mathit{UNIT} \rightarrow \mathit{RESULT} \\
\mathit{backwardValue} : \mathit{UNIT} \rightarrow \mathit{RESULT} \\
\mathit{updateLocalStateForward}(\mathit{UNIT},\ \mathit{RESULT}) \\
\mathit{updateLocalStateBackward}(\mathit{UNIT},\ \mathit{RESULT})
\end{array}
$$

The forward propagation transmits the computed value to every destination unit of the given unit, and in case of an output unit also to the corresponding output location.

$$
\begin{array}{l}
\mathit{propagateForward}(u,\ \mathit{dataToPropagate}) = \\
\quad \textbf{forall}\ \ (d \in \mathit{dest}(u))\ \ \textbf{do} \\
\quad\quad \mathit{inForward}^{\mathrm{int}}(d,u) := \mathit{intValueForw}(d,u,\ \mathit{dataToPropagate}) \\
\quad \textbf{if}\ \ (u \in \mathit{outputUnits})\ \ \textbf{then} \\
\quad\quad \mathit{output}(u) := \mathit{extValueForw}(u,\ \mathit{dataToPropagate})
\end{array}
$$

We proceed similarly for the backward propagation where the result of the local computation is transmitted to the units that precede the current unit.

$$
\begin{array}{l}
\mathit{propagateBackward}(u,\ \mathit{dataToPropagate}) = \\
\quad \textbf{forall}\ \ (s \in \mathit{source}(u))\ \ \textbf{do} \\
\quad\quad \mathit{inBackward}^{\mathrm{int}}(s,u) := \mathit{intValueBack}(s,u,\ \mathit{dataToPropagate}) \\
\quad \textbf{if}\ \ (u \in \mathit{inputUnits})\ \ \textbf{then} \\
\quad\quad \mathit{outputBack}(u) := \mathit{extValueBack}(u,\ \mathit{dataToPropagate})
\end{array}
$$




$$orderedUnitsForward = [\{x, y\}, \{z\}, \{u, v, w\}, \{s, t\}]$$
$$orderedUnitsBackward = [\{s, t\}, \{u, v, w\}, \{x, z\}, \{y\}]$$

**Figure 3:** Coding the data-flow propagation into the scheduler

As part of the propagation there is also a local data transformation, which we describe in terms of abstract functions, to be refined for each particular unit instantiation.

$$\begin{aligned}
&intValueForw : UNIT \times UNIT \times RESULT \rightarrow DATA\\
&extValueForw : UNIT \times RESULT \rightarrow DATA\\
&intValueBack : UNIT \times UNIT \times RESULT \rightarrow DATA\\
&extValueBack : UNIT \times RESULT \rightarrow DATA^*
\end{aligned}$$

### 3.2.3 Scheduling

The scheduling process appears in our model in the form of two functions.

$$\begin{aligned}
&scheduledUnits : \mathcal{P}(UNIT)\\
&nextExecutableUnits : \mathcal{P}(UNIT) \times INPUT_TYPE \rightarrow \mathcal{P}(UNIT)
\end{aligned}$$

The controlled function *scheduledUnits* maintains the information on the units which are currently ready for the computation, and is updated using the scheduling function *nextExecutableUnits*. This description reflects the multitude of possible schedulers. One possible refinement is to select the units depending upon an ordering among them, which is reasonably assumed to be updatable by the environment and thereby appropriately reflected by two functions:

$$\begin{aligned}
&orderedUnitsForward : \mathcal{P}(UNIT)^*\\
&orderedUnitsBackward : \mathcal{P}(UNIT)^*
\end{aligned}$$

The graph depicted in [Figure 3] shows an example how the flow of data in a neural network can be scheduled within the two ordering functions.

The scheduler itself then becomes a function that is derived from the unit ordering as follows:

$$
\begin{array}{l}
nextExecutableUnits(U,\ inputType) = \\
\quad \textbf{case}\ \ (inputType)\ \ \textbf{of} \\
\qquad forward\ \rightarrow\ selectNextUnits(U,\ orderedUnitsForward) \\
\qquad backward\ \rightarrow\ selectNextUnits(U,\ orderedUnitsBackward) \\
\quad \textbf{endcase}
\end{array}
$$

where an auxiliary function *selectNextUnits* allows one to still vary the actual schedule, e.g. by different search algorithms through the given ordering.

$$selectNextUnits : \mathcal{P}(UNITS) \times \mathcal{P}(UNITS)^* \rightarrow \mathcal{P}(UNITS)$$

As another example we consider the specialization of *nextExecutableUnits* for a typical feedforward neural network model without internal feedbacks, trained with the backpropagation algorithm. What characterizes this model is that the activation of neural units follows the data-flow principle for both forward and backward information flow, in the sense that a unit can be activated only when all its inputs are ready.

It suffices to associate to each unit a flag *actionState* indicating whether the unit has been already activated during the elaboration of the current environmental input. It is assumed (and has to be guaranteed upon instantiating *NK*) that initially, all the units get their flag set to *waiting*. Upon activating a unit, its flag is updated to *executed* as an extension of the machines *updateLocalStateForward* respectively *updateLocalStateBackward*. In this way, a unit has all the inputs ready and becomes eligible for the scheduler, when all the preceding units have been activated. This allows to refine the scheduler as follows:

$$
\begin{array}{l}
nextExecutableUnits(U,\ inputType) = \\
\quad \textbf{case}\ \ (inputType)\ \ \textbf{of} \\
\qquad forward\ \rightarrow \\
\qquad\quad \{u \in network \mid actionState(u) \neq executed\ \ and \\
\qquad\qquad \forall u' \in source(u) : u' \in U\ \ or\ actionState(u') = executed\} \\
\qquad backward\ \rightarrow \\
\qquad\quad \{u \in network \mid actionState(u) \neq executed\ \ and \\
\qquad\qquad \forall u' \in dest(u) : u' \in U\ \ or\ actionState(u') = executed\}
\end{array}
$$

The *clearState* rule has to be refined to clear all the *actionState* flags.

$$
\begin{array}{l}
clearState = \\
\quad \textbf{forall}\ \ (u \in network)\ \ \textbf{do} \\
\qquad actionState(u) := waiting
\end{array}
$$

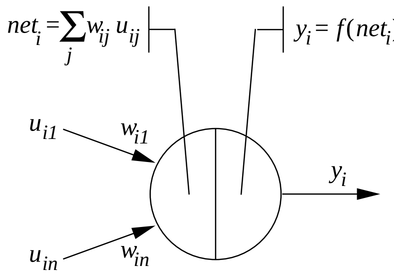



Figure 4: The figure schematize the typical computations performed by a neural unit during its forward propagation.

# 4 Refinement for Feedforward NNs with Backpropagation Training

## 4.1 Feedforward and Backpropagation Training

In a feedforward neural network, the computation of the neurons—the computational units— in the operative mode is described by the equations 1 and 2 [see also Fig. 4].

In the backpropagation training mode, the weights of the links connecting the neurons are modified by an amount($\Delta w_{ij}$), computed deriving the cost function (3) with respect to the weights. This yields an inverse propagation of data from the output neurons to the input neurons, as described by the following equations for the backpropagation in the $i$-th computational unit [see Fig. 1]:

$$\Delta w_{ij} = -\eta \delta_i u_{ij} \tag{4}$$

$$\delta_i = \begin{cases} -f_i'(net_i) \cdot (d_i - y_i) & \text{if } i \text{ is an output unit} \\ \\ -f_i'(net_i) \cdot \left(\sum_{k \in dest(i)} \delta_k w_{ki}\right) & \text{otherwise} \end{cases} \tag{5}$$

where $w_{ij}$ are the weight parameters associated to the links from unit $j$ to unit $i$, $\Delta w_{ij}$ is the amount of weight change inferred by the error made by the network, $f_i'$ is the first derivative of the activation function, and $\eta$ is the learning rate of the algorithm. If the neuron is an output unit, its internal weights update depends on the error information $(d_i - y_i)$ coming from the environment; otherwise, the internal weights update depends on the $\delta$s backpropagated from the following units.

### 4.2 Refining the State for Neurons

The state—we refer to it as status—of the neural units, which comprises the internal and external input functions, is enriched by additional functions for the internal and external *weights* (the adaptive parameters) and the *deltaWeights*, where the latter are used for storing the information on the weight changes for training the network. We also include the *net* function into the refined state, to be computed and used during both forward and backward propagation. Formally this comes up to have a *status* function that associates to each unit its *STATUS*.

$$status : UNIT \to STATUS$$

where the ASM domain *STATUS* can be defined as:

$$\begin{aligned} STATUS = (&inForward^{\mathrm{int}} &&: UNIT \to DATA,\\ &inForward^{\mathrm{ext}} &&: DATA^*,\\ &inBackward^{\mathrm{int}} &&: UNIT \to DATA,\\ &inBackward^{\mathrm{ext}} &&: DATA^*,\\ &weights^{\mathrm{int}} &&: UNIT \to WEIGHT,\\ &weights^{\mathrm{ext}} &&: WEIGHT^*,\\ &deltaWeights^{\mathrm{int}} &&: UNIT \to WEIGHT,\\ &deltaWeights^{\mathrm{ext}} &&: WEIGHT^*,\\ &net &&: NET) \end{aligned}$$

Without loss of generality we set

$$WEIGHT = float$$
$$NET = float$$

Below, instead of *status* we use directly its projections. In particular, for each function defined in *STATUS*, we define a function with the same signature as expressed in object-oriented notation in the following example:

$$inForward^{\mathrm{int}}(u, v) = status(u).inForward^{\mathrm{int}}(v)$$

where $status(u).inForward^{\mathrm{int}}(v)$ stands for $inForward^{\mathrm{int}}(status(u))(v)$.
When transmitting information from the environment to neural units or vice versa, also the receiving or sending unit name is considered as part of the sent data. One can reflect this by stipulating that data are pairs of units and message content (the proper value of the data):

$$DATA = (UNIT,\ RESULT\ )$$

$$RESULT = float$$

Below we will only use the corresponding second projection function, defined both for elements and sequences of *DATA*:

$result : DATA \rightarrow RESULT$
$result : DATA* \rightarrow RESULT*$

### 4.3 Refining the Forward Unit Computation

The submachine *computeUnit* consists of three constituent machines for computing and propagating the local result and for updating the local state, which we are going to instantiate in this section for input of type *forward*.

The local result computing function *forwardValue* can be refined as follows to compute the forward output of a neuron as described by equations 1 and 2:

$$
\begin{aligned}
& \mathit{forwardValue}(u) = \\
& \quad \mathbf{let} \ \ \mathit{netValue} = \mathit{scalarProduct}\ (\mathit{result}(\mathit{inForward}^{\mathrm{ext}}(u)),\ \mathit{weights}^{\mathrm{ext}}(u)) \\
& \qquad\qquad + \textstyle\sum_{s \in source(u)} \mathit{result}(\mathit{inForward}^{\mathrm{int}}(u,s)) \cdot \mathit{weights}^{\mathrm{int}}(u,s) \\
& \quad \mathbf{in} \\
& \qquad \mathit{activationFunction}(\mathit{netValue})
\end{aligned}
$$

where *activationFunction* stands for the nonlinear function $f$ in equation 2 and *scalarProduct* is a derived function that computes the product of two vectors. The two value forwarding functions *extValueForw* and *intValueForw*, used by *propagateForward*, in this case simply pass the data to be propagated unchanged, formally:

$$
\begin{aligned}
& \mathit{extValueForw} : UNIT \times RESULT \rightarrow DATA \\
& \mathit{extValueForw}(u,\ \mathit{dataToPropagate}) = (u,\ \mathit{dataToPropagate}) \\
& \mathit{intValueForw} : UNIT \times UNIT \times RESULT \rightarrow DATA \\
& \mathit{intValueForw}(d, u,\ \mathit{dataToPropagate}) = (\mathit{undef},\ \mathit{dataToPropagate})
\end{aligned}
$$

The update of the local state of a unit is refined for input of type *forward* as computing *net*, formally:

$$
\begin{aligned}
& \mathit{updateLocalStateForward}(u,\ \mathit{forwardResult}) = \\
& \quad \mathit{net}(u) := \mathit{scalarProduct}(\mathit{result}(\mathit{inForward}^{\mathrm{ext}}(u)),\ \mathit{weights}^{\mathrm{ext}}(u)) + \\
& \qquad\qquad \textstyle\sum_{s \in source(u)} \mathit{result}(\mathit{inForward}^{\mathrm{int}}(u,s)) \cdot \mathit{weights}^{\mathrm{int}}(u,s)
\end{aligned}
$$

The same equation could obviously be used to define *net* not as controlled but as derived function.

### 4.4 Refining the Backward Unit Computation

In this section we instantiate the three abstract constituents of the submachine *computeUnit* for inputs of type *backward*.

The local result computing function *backwardValue* can be refined as follows to compute the backward output of a neuron as described by equations 4 and 5:

$$backwardValue(u) = derivedActivationFunction(net(u)) \cdot \\ (\sum result(inBackward^{\text{ext}}(u)) + \\ \sum_{d \in dest(u)} result(inBackward^{\text{int}}(u,d)))$$

where *derivedActivationFunction* is the first derivative of the neural activation function, the first sum is done over all *result* elements of the sequence in *inBackward*$^{\text{ext}}(u)$ coming from the environment, and the second sum is done over all $\delta$s coming from the following units. In this way we take the sum of all error contributions that are backpropagated to the considered unit.

The machine *propagateBackward* uses two abstract value functions which can here be refined to pass their arguments to an auxiliary function *vectorResult* that computes a vector of *DATA* containing the backpropagated $\delta$s which are to be transmitted to the environment. Formally:

$$extValueBack : UNIT \times RESULT \rightarrow DATA^*$$
$$extValueBack(u,\ dataToPropagate) = \\ \quad vectorResult(u,\ dataToPropagate,\ weights^{\text{ext}}(u))$$

$$intValueBack : UNIT \times UNIT \times RESULT \rightarrow DATA$$
$$intValueBack(d, u,\ dataToPropagate) = \\ \quad (undef,\ dataToPropagate \cdot weights^{\text{int}}(u,d))$$

where *vectorResult* is a function that given the current unit name, the message to transmit, and the list of weights, prepares the list of *DATA* to be passed to the environment.

$vectorResult : UNIT \times RESULT \times WEIGHTS^* \rightarrow DATA^*$
$vectorResult(u,\ data,\ weights) =$
    **if** $(length(weights) \geq 1)$ **then**
        **let** $(value = head(weights) \cdot data)$ **in**
            $[(u,\ value)] \cdot vectorResult(u,\ data,\ tail(weights))$
    **else**
        $[\,]$

The update of the unit local state is refined for input of type *backward* as follows:

$$
\begin{array}{l}
\mathit{updateLocalStateBackward}(u,\ \mathit{backwardResult}) = \\
\quad \textbf{if}\ \ (\mathit{updateWeights}(\mathit{inputCopy}))\ \ \textbf{then} \\
\qquad \mathit{weights}^{\text{ext}}(u) := \mathit{weights}^{\text{ext}}(u) + \eta(u)\ \cdot\ (\mathit{deltaWeights}^{\text{ext}}(u) \\
\qquad\qquad\qquad + \mathit{backwardResult} \cdot \mathit{result}(\mathit{inForward}^{\text{ext}}(u))) \\
\qquad \mathit{deltaWeights}^{\text{ext}}(u) := [0, \ldots, 0] \\
\qquad \textbf{forall}\ \ (s \in \mathit{source}(u))\ \ \textbf{do} \\
\qquad\quad \mathit{weights}^{\text{int}}(u,s) := \mathit{weights}^{\text{int}}(u,s) + \eta(u) \cdot\ (\mathit{deltaWeights}^{\text{int}}(u,s) \\
\qquad\qquad\qquad + \mathit{backwardResult} \cdot \mathit{result}(\mathit{inForward}^{\text{int}}(u,s))) \\
\qquad\quad \mathit{deltaWeights}^{\text{int}}(u,s) := 0 \\
\quad \textbf{else} \\
\qquad \mathit{deltaWeights}^{\text{ext}}(u) := \mathit{deltaWeights}^{\text{ext}}(u) + \\
\qquad\qquad\qquad \mathit{backwardResult} \cdot \mathit{result}(\mathit{inForward}^{\text{ext}}(u))) \\
\qquad \textbf{forall}\ \ (s \in \mathit{source}(u))\ \ \textbf{do} \\
\qquad\quad \mathit{deltaWeights}^{\text{int}}(u,s) := \mathit{deltaWeights}^{\text{int}}(u,s) + \\
\qquad\qquad\qquad \mathit{backwardResult} \cdot \mathit{result}(\mathit{inForward}^{\text{int}}(u,s)))
\end{array}
$$

where *updateWeights* is a projection function that extracts the required flag from the copy of the *input* location. With this flag the environment signals to the *NK* whether to update the adaptive parameters of the neural network or to temporarily store the changes leaving the parameters unchanged.

## 5 Conclusion and Future Work

We have defined the computational kernel of a Neural Abstract Machine, making explicit the basic computational paradigm of a generic neural network. We have refined this abstract machine in a natural way to a machine for feedforward neural networks with backpropagation training. In [Sona and Sperduti(2001)] an alternative refinement for such nets is provided where the units can be synapses, neurons and functional units. Also the Boltzmann machine is derived there as an instance of our abstract neural machine. Work in progress is on *NAM* specification for training algorithms for recurrent neural networks (BPTT [Rumelhart et al.(1986)] and RTRL [Williams and Zipser(1989)]) for sequences and structures [Giles and Gori(1998)]. We plan also to work on further refinements reflecting other computational paradigms (fuzzy, probabilistic, symbolic, etc.) and on other abstract components, one for controlling the interaction between the environment and the neural kernel (e.g. providing inputs and taking outputs, dynamically creating and initializing or destroying networks or single computational units), one for a scheduler, one for a debugger, one to access databases for the persistence of data, models and programs, etc.

## Acknowledgements

We are grateful to Alessandro Sperduti for suggesting to build an abstract neural machine and for helpful discussions.

## References

[Berthold and Fischer(1997)] Berthold, M. and Fischer, I., 1997. Formalizing Neural Networks Using Graph Transformations. In *Proc. of the IEEE Int. Conf. on Neural Networks*, vol. 1, pp. 275–280. IEEE.

[Bishop(1995)] Bishop, C., 1995. *Neural Networks for Pattern Recognition*. Oxford University Press.

[Börger(1999)] Börger, E., 1999. High Level System Design and Analysis using Abstract State Machines. In *Current Trends in Applied Formal Methods (FM-Trends 98)*, eds. D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, vol. 1641 of *LNCS*, pp. 1–43. Berlin Heidelberg New York: Springer-Verlag.

[Börger(2001)] Börger, E., 2001. Design for Reuse via Structuring Techniques for ASMs. In *Proc. EUROCAST'2001 (Las Palmas)*. Springer LNCS.

[Börger and Schmid(2000)] Börger, E. and Schmid, J., 2000. Composition and Submachine Concepts for Sequential ASMs. In *Computer Science Logic (Gurevich Festschrift). Proc. 14th International Workshop CSL*, eds. P. Clote and H. Schwichtenberg, Springer LNCS 1862, pp. 41–60.

[Campolucci et al.(2000)] Campolucci, P., Uncini, A., and Piazza, F., 2000. A Signal-Flow-Graph Approach to On-line Gradient Calculation. *Neural Computation* 12, no. 8:1901–1927.

[Giles and Gori(1998)] Giles, C. and Gori, M., eds., 1998. *Adaptive Processing of Sequences and Data structures*. LNAI. Heidelberg, Germany: Springer.

[Gurevich(1995)] Gurevich, Y., 1995. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, ed. E. Börger, pp. 9–36. Oxford University Press.

[Haykin(1999)] Haykin, S., 1999. *Neural Networks, A Comprehensive Foundation*. Prentice Hall, second edn.

[Nerrand et al.(1993)] Nerrand, O., Roussel-Ragot, P., Personnaz, L., Dreyfus, G., and Marcos, S., 1993. Neural Networks and Nonlinear Adaptive Filtering: Unifying Concepts and New Algorithms. *Neural Computation* 5, no. 2:165–199.

[Rumelhart et al.(1986)] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., 1986. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, eds. D. Rumelhart, J. McClelland, and the PDP Research Group. MIT Press.

[Santini et al.(1995)] Santini, S., Bimbo, A. D., and Jain, R., 1995. Block structured recurrent neural networks. *Neural Networks* 8:135–147.

[Sona and Sperduti(2001)] Sona, D. and Sperduti, A., 2001. Modeling in the neural abstract machine framework. LFTNC 2001 - NATO Advanced Research Workshop on Limitations and Future Trends in Neural Computation- Siena, Italy 22 - 24 October 2001. Accepted.

[Stärk et al.(2001)] Stärk, R., Schmid, J., and Börger, E., 2001. *Java and the Java Virtual Machine: Definition, Verification, Validation.*. Berlin-Heidelberg-New York: Springer-Verlag.

[Tsoi(1998a)] Tsoi, A., 1998a. Gradient Based Learning Methods. *LNCS* 1387:27–62.

[Tsoi(1998b)] Tsoi, A., 1998b. Recurrent Neural Network Architectures: An Overview. *LNCS* 1387:1–26.

[Wan and Beaufays(1998)] Wan, E. and Beaufays, F., 1998. Diagrammatic Methods for Deriving and Relating Temporal Neural Network Algorithms. *Lecture Notes in Computer Science* 1387:63–98.

[Williams and Zipser(1989)] Williams, R. J. and Zipser, D., 1989. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation* 1:270–280.