Modelling in the Neural Abstract Machine Framework

Diego Sona, Alessandro Sperduti
Dipartimento di Informatica, Università di Pisa
email: {sona,sperduti}@di.unipi.it

Abstract. The variety and complexity of learning tasks and neural networks models is quite large. Moreover, there are other relevant factors which add complexity to the management of a neural network for a complete solution of the faced task. A unified computational framework for neural computation is thus needed. Here we propose to use *Abstract State Machines* to define a *Neural Abstract Machine*, a machine able to manipulate the basic objects and functions which constitute the essence of neural computation. A partial definition of the core of this machine, namely the *Neural Kernel*, is presented and a couple of examples of instantiation to classic neural models is given.

1 Introduction

Defi ning and training a neural network for solving a given problem is not a trivial task. Several issues must be considered, including the format of input (vectors, sequences, structures), the learning paradigm (supervised, unsupervised, reinforcement), the network model (feed-forward backpropagation, recurrent, recursive, Boltzmann, ART-map, SOM, etc.), the choice of the "right" architecture (number of neurons, hidden units, weights, and feed-back connections, input window size, etc.), the choice of the error function (squared error, entropy, regularized error functions, etc.), and so on.

This complexity is usually faced by a procedural approach, where each choice is taken without the help of a unifi ed computational approach. Some help is given by neural network simulators. Most of them, however, allow the implementation of specifi c neural network architectures and training algorithms, and they do not allow for major modifi cations of the neural model and training algorithm. Moreover, none of them is based on a unifi ed computational theory of neural computation, including structured data.

The fi nal goal of our work is to defi ne a formal computational machine, namely the Neural Abstract Machine (*NAM*), able to manipulate the basic objects and functions which constitute the essence of neural computation. In this way, it should be possible to specify the computation of each neural model as an appropriate combination of basic instructions and data structures that constitute the *NAM*. Moreover, it should be easy to defi ne new neural models and learning algorithms by simply giving the specifi cation of the model: the *NAM* will take care to execute the specifi cation on the input data.

In the present work we briefly present the (partial) specifi cation of the core of the *NAM*, that we call *Neural Kernel* (*NK*). Using an *ASM* composition technique, as explained in [4], we defi ne the *NK* as a composition of many parallel sub-machines.

2 Overview of the *ASMs* Formalism

The formal notion of *Abstract State Machine* (*ASM*) also referred as *Gurevich's Evolving Algebras*, defined in [7], furnishes a rigorous mathematical method for modeling computing processes on different abstraction levels. This formalism, belonging to the software engineering area, presents many useful features for the formal definition of the *Neural Abstract Machine*. First of all, its intrinsic parallelism which is well suited for expressing the internal parallelism of a neural network. Furthermore, it has been intensively used for the formal design and analysis of various hardware systems, algorithms, and programming languages semantics, showing a good scalability to complex real-world systems. Moreover, due to the imperative specification style the formalism shows easiness while learning and using it, nevertheless preserving mathematical soundness and completeness. This leads to an easy definition of formal and informal requirements, turning them into a satisfactory *ground model*[1] [3].

As stated in [2], the *ASMs* have many features that can help when devising a system or defining a language semantics. The most important of such features is the *freedom of abstraction*, which allows incremental development of systems by stepwise refinement through a vertical hierarchy of intermediate models. This property is well sustained by the *information hiding* and the *precise definition of interfaces*, which help the horizontal organization of modules.

Last but not least, due to the formality of the *ASMs*, it will be possible to verify by rigorous proofs some properties of the *NAM*, and due to the existence of *ASM* tools (see *ASM-Workbench, ASMGofer, XASM, asmL* by Microsoft©) we can validate experimentally the prototype, refining the system with different neural model implementations.

When modeling a system, the basic concepts on which it works must be defined. These basic concepts, which are the atomic entities of the system, in the *ASM* terminology are the *objects*, each one belonging to a specific class, forming in this way many sets of coherent objects, which we call *domains*. All the sets constitute the *universe* over which the *ASM* model operates.

The notion of *ASM state* is related to the mathematical concept of *algebra*, which can be defined as a set of domains equipped with a set of basic operations (partial *functions*) and predicates, defined over the domains.

A state transition, which corresponds to an *ASM* computation step, changes these functions point-wise with the so called *updates*, which are triples of the form:

$$(f, (t_1, \ldots, t_n), t)$$

where f is an arbitrary n-ary function, $t_1, \ldots, t_n$ are the function parameters defined in some domains, and t is the value at which the function is set. In the *ASM* framework the *dynamic function update* corresponds to the *destructive assignment*:

$$f(t_1, \ldots, t_n) := t$$

which changes (or defines for the first time) the value of the function f at the given parameters.

The function definition is so general that there are no restrictions on its abstraction level, its complexity, and its behavior. However, functions are distinguished in terms of their basic characteristics, i.e. they can be *dynamic* or *static*, depending on the possibility of the function

[1]The ground model is a specification that satisfies all the system requirements.

to change or not during any *ASM* run as consequence of updates. The dynamic functions are further divided into four subclasses. *Controlled* functions are updated by and only by the rules of the defined *ASM* machine, and are not updated by the environment. *Monitored* functions are dynamic functions which are updated only by the environment and not by the machine. *Interaction* functions, which can be updated by both the environment and the machine. *Derived* function, which are not updatable neither by the environment nor by the machine, but are nevertheless dynamic because defined in terms of static and dynamic functions

The dynamic function updates are the basic operations by which the behavior of a system can be described. However, they are inadequate for a complete system description, so the model needs to be enriched with control operators, frequently termed *rules*. Thus, the *ASMs* are systems of finitely many transition rules of the form:

```
if (Condition) then Updates
```

where *Condition* is a first-order formula (the constraint), and *Updates* consists of finitely many function updates. The meaning is very simple: if the *Condition* is *true* all the *Updates* are simultaneously executed.

An *ASM* run can be defined as a sequence of *ASM* steps. At each step, all updates of all transition rules with the verified guard are executed simultaneously. The *ASM* terminates its run when no more rules have the guard verified.

The *ASM* formalism is so general that a free use of programming notation is permitted. In order to simplify the designer work, some simple and generic rules have been introduced, the most important of which is:

```
forall $x$ with $\phi$ do
    Rule
```

which is a concise notation for the simultaneous execution of the *ASM* *Rule* for each x object satisfying the condition ϕ. In a similar way we use notations such as **if-then-else**, **case**, **let**, and so on. We also use parameterized rules that allow the composition of concepts expressed as submachine definitions [4], which are a sort of macro definition:

$r(x_1, \ldots, x_n)$

3 The Neural Kernel Specification

The *NK* gives the basic computational paradigm for a generic neural network, in the sense that it performs all internal computations of a neural network, without any concern about network creation and initialization. Specifically, we assume that the environment "magically" controls the network creation and maintenance. Within this scheme the environment furnishes the input to the *NK* and gathers its output using two interface (interaction) functions

input : *INPUT*, *output* : *OUTPUT*

where *INPUT* and *OUTPUT* are sequences of *DATA*. Moreover, the input may contain many information, accessed with specific functions, such as the type of signal propagation and the initialization of units.

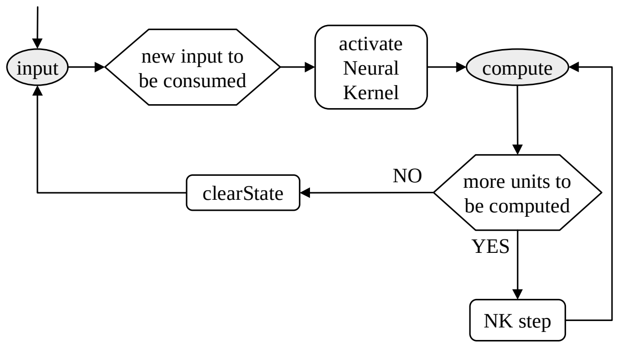

Figure 1: Description of the macro step of the *NK*, which operates in the two states *input* and *compute*. The transitions from one state to the other correspond to the ASM rules execution. If the condition in the hexagonal box is verified, then there is a transition and the rule in the rectangular box is executed.

3.1 Neural Networks Description

Some researchers, especially for recurrent architectures, have devised different unifi cation theories using two main approaches: *Architectural*, where the NNs are classifi ed accordingly to topological aspects, and the NN behavior is described as operations on vectors and matrixes [10]; *Block diagram*, where the NNs are defi ned as compositions of standard computational blocks, and the NN behavior is decomposed and analyzed as composition of simple blocks behavior [11, 6, 1, 8, 9]. We believe that the second approach is more general for a tool development, i.e. a NN is described as a set of connected computational units, which can be formalized as having two flows of information (see [11, 1]): the forward flow in the fi ring state, and the backward flow in the training state. We will see that this formalization is valid also for those NNs that are not trained by gradient descent algorithms.

Thus, within the *NK* each neural network is defi ned as a composition of blocks (connected according to the desired topology) specifi ed by the user, which we call *computational units*. Each unit has an internal state which is partially defi ned by the *NK*, and partially defi ned by the end user. The input changes when the connected units send the result of their computation, and this is managed by the *NK*. With this approach the *NK* furnishes the skeleton for the flow of information, and the basic neural computational paradigm, while the user defi nes the units behavior according to the model she wants to instantiate.

In order to guarantee the correct propagation of data within the network we use two different inputs depending on the flow direction. Moreover, we distinguish between the input coming from the environment (external input) and the one coming from other units (internal input). For this reason, we need to defi ne four functions:

$$\begin{array}{ll} \mathit{inForward}^{\mathrm{ext}} : \mathit{UNIT} \rightarrow \mathit{DATA*} & \mathit{inBackward}^{\mathrm{ext}} : \mathit{UNIT} \rightarrow \mathit{DATA*} \\ \mathit{inForward}^{\mathrm{int}} : \mathit{UNIT} \times \mathit{UNIT} \rightarrow \mathit{DATA} & \mathit{inBackward}^{\mathrm{int}} : \mathit{UNIT} \times \mathit{UNIT} \rightarrow \mathit{DATA} \end{array}$$

where the internal input is a function of the owner and sender unit names.

3.2 The Neural Kernel

As formerly pointed out, the *NK* is an agent operating over an input stream. In particular, it operates in two distinct and mutually exclusive phases (see Figure 1). In the fi rst phase (*input*), the *NK* waits for external data, and when the input arrives the *NK* activate itself, starting the neural computations, and insulating itself from the environment. In the second phase (*compute*), the *NK* performs all the neural network computations till no more units can be executed. Then, the *NK* returns in the *input* state waiting for new data.

Our purpose here is not to give a complete specifi cation of the *NK*, which is better explained in [5], but to give a feeling of the framework, in order to show some NNs descriptions. The core of the *NK* is constituted by the rule governing the internal computation, which in Figure 1 is represented by the "NK step" box. At each iteration, this rule simultaneously activates all the scheduled units (with the *computeUnit* rule) that have to process their input information, if there is any:

```
NK-Step =
    forall (u ∈ scheduledUnits) do
        computeUnit(u)
    scheduledUnits := nextExecutableUnits(scheduledUnits, inputType)
```

At the same time, the rule determines the set of units that need to be processed at the next machine step. When the set of selected units is empty the *NK* will commute itself in the *input* state. The solution is highly modular since it separates the unit computation from the scheduling strategy, rendering the two concepts perfectly orthogonal.

The *computeUnit* rule modifi es the internal state of the unit, and propagates the result to the input of the connected units, and eventually to the *output* of the network. The unit state can change in two different ways, depending on the type of propagation:

```
computeUnit(u) =
    if (inputType = forward) then
        let (result = forwardValue(u)) in
            propagateForward(u, result)
            updateLocalStateForward(u, result)
    if (inputType = backward) then
        let (result = backwardValue(u)) in
            propagateBackward(u, result)
            updateLocalStateBackward(u, result)
```

The functions returning the internal unit computation (*forwardValue* and *backwardValue*), as well as the rules that update the internal state (*updateLocalStateForward* and *updateLocalStateBackward*), are left to the user defi nition, which has the responsibility for the correct implementation. The result of the unit computation is then transmitted to the connected units, if there is any, and eventually to the output of the network. In the forward propagation the connected units are identifi ed by the *observed* function *dest*:

```
propagateForward(u, dataToPropagate) =
    if (u ∈ outputUnits) then
        output(u) := extValueForw(u, dataToPropagate)
    forall (d ∈ dest(u)) do
        $inForward^{\mathrm{int}}(d, u)$ := intValueForw(d, u, dataToPropagate)
```

while in the backward propagation they are identifi ed by the *observed* function *source*:

```
propagateBackward(u, dataToPropagate) =
    if (u ∈ inputUnits) then
        outputBack(u) := extValueBack(u, dataToPropagate)
    forall (s ∈ source(u)) do
        $inBackward^{\mathrm{int}}(s, u)$ := intValueBack(s, u, dataToPropagate)
```

In the above two rules, before the propagation, data is transformed by two functions which appropriately prepare the result for the connected computational units, or the output of the network.

4 Examples of *NK* instantiation to Different NNs

As formerly pointed out, the type of data used during information transmission need to be defined. Here we assume that this is done accordingly with the instance of neural model to be implemented. Moreover, there are some functions and rules that need to be defined. Among all, here we are interested in the ones involved in the forward (*forwardValue*, *extValueForw*, *intValueForw*, *updateLocalStateForward*) and backward (*backwardValue*, *extValueBack*, *intValueBack*, *updateLocalStateBackward*) propagation. The first three functions, respectively for the forward or the backward phase, return the result of the internal unit computation, the transformed value addressed to the environment (external) and addressed to other units (internal). The last one instead is a rule that modifies the local state of the units.

4.1 Feed-forward NNs with Back-Propagation Training

To specify a feedforward neural network trained by Back-Propagation we decided to adopt the *Signal Flow Graph* approach [11, 6] or equivalently the *Graph Grammars* approach [1], where a single neuron in a network is seen as a composition of many simple computational elements, which are the synapses, the summing junction and the activation function. Thus, the feedforward NN with backpropagation can be designed as a composition of three types of units: *SYNAPSE, JUNCTION* and *FUNCTIONAL*.

4.1.1 Forward Computation

The *forwardValue* function, which returns the result of the unit computation is defined accordingly to the type of unit it refers to. The synapses return their single internal input multiplied by the *weight* parameter, the junctions return the sum of all inputs, and the functionals return the result of the activation function applied to the single input:

```
$\mathit{forwardValue}(u : \mathit{SYNAPSE}) =$
    if $(u \in \mathit{inputUnits})$ then
        let $([\mathit{in}] = \mathit{inForward}^{\mathrm{ext}}(u))$ in $\mathit{dataValue}(\mathit{in}) \cdot \mathit{weight}(u))$
    else
        let $(\{s\} = \mathit{source}(u))$ in $\mathit{dataValue}(\mathit{inForward}^{\mathrm{int}}(u, s)) \cdot \mathit{weight}(u)$

$\mathit{forwardValue}(u : \mathit{JUNCTION}) = \sum_{s \in source(u)} \mathit{dataValue}(\mathit{inForward}^{\mathrm{int}}(u, s))$

$\mathit{forwardValue}(u : \mathit{FUNCTIONAL}) =$
    let $(\{s\} = \mathit{source}(u))$ in $\mathit{activationFunction}(\mathit{dataValue}\ (\mathit{inForward}^{\mathrm{int}}(u, s)))$
```

After the unit computation, the result is propagated using the rule *propagateForward*, which in turn uses the two functions:

$\mathit{extValueForw}(u, \mathit{dataToPropagate}) = (u, \mathit{dataToPropagate})$
$\mathit{intValueForw}(d, u, \mathit{dataToPropagate}) = (\mathit{undef}, \mathit{dataToPropagate})$

which return a structure containing the resulting value with attached appropriate information.

Finally, the local state of all units need to be updated, however, since the units have no short term memory, the update of the state for all the units correspond to the **skip** operation.

4.1.2 Backward Computation

The computation of the backward value inherits the three different modalities of the forward propagation:

```
$\mathit{backwardValue}(u : \mathit{SYNAPSE}) =$
    **let** $(\{d\} = \mathit{dest}(u))$ **in** $\mathit{dataValue}(\mathit{inBackward}^{\mathrm{int}}(u, d))$

$\mathit{backwardValue}(u : \mathit{JUNCTION}) =$
    **let** $(\{d\} = \mathit{dest}(u))$ **in** $\mathit{dataValue}(\mathit{inBackward}^{\mathrm{int}}(u, d))$

$\mathit{backwardValue}(u : \mathit{FUNCTIONAL}) =$
    **let** $(\{s\} = \mathit{source}(u))$ **in**
        $\mathit{derivedActivationFunction}(\mathit{dataValue}(\mathit{inForward}^{\mathrm{int}}(u, s))) \cdot$
                $(\sum \mathit{dataValue}(\mathit{inBackward}^{\mathrm{ext}}(u)) + \sum_{d \in dest(u)} \mathit{dataValue}(\mathit{inBackward}^{\mathrm{int}}(u, d)))$
```

Then the result is backward propagated trough the rule *propagateBackward*, which rely on the functions:

$\mathit{extValueBack}(u : \mathit{SYNAPSE}, \mathit{dataToPropagate}) = (u, \mathit{dataToPropagate} \cdot \mathit{weight}(u))$
$\mathit{intValueBack}(d, u : \mathit{SYNAPSE}, \mathit{dataToPropagate}) = (\mathit{undef}, \mathit{dataToPropagate} \cdot \mathit{weight}(u))$

$\mathit{extValueBack}(u : \mathit{JUNCTION}, \mathit{dataToPropagate}) =$ **skip**
$\mathit{intValueBack}(d, u : \mathit{JUNCTION}, \mathit{dataToPropagate}) = (\mathit{undef}, \mathit{dataToPropagate})$

$\mathit{extValueBack}(u : \mathit{FUNCTIONAL}, \mathit{dataToPropagate}) =$ **skip**
$\mathit{intValueBack}(d, u : \mathit{FUNCTIONAL}, \mathit{dataToPropagate}) = (\mathit{undef}, \mathit{dataToPropagate}(u))$

Finally, the local state needs to be updated by the rule *updateLocalStateBackward* instantiated over the three types of units:

```
$\mathit{updateLocalStateBackward}(u : \mathit{SYNAPSE}, \mathit{backwardResult}) =$
    **if** $(\mathit{updateWeights}(\mathit{inputCopy}))$ **then**
        **if** $(u \in \mathit{inputUnits})$ **then**
            **let** $([\mathit{in}] = \mathit{inForward}^{\mathrm{ext}}(u))$ **in**
                $\mathit{weight}(u) := \mathit{weight}(u) + \eta(u) \cdot (\mathit{deltaWeight}(u) + \mathit{backwardResult} \cdot \mathit{dataValue}(\mathit{in})$
        **else**
            **let** $(\{s\} = \mathit{source}(u))$ **in**
                **let** $(\mathit{in} = \mathit{inForward}^{\mathrm{int}}(u, s)))$ **in**
                    $\mathit{weight}(u) := \mathit{weight}(u) + \eta(u) \cdot (\mathit{deltaWeight}(u) + \mathit{backwardResult} \cdot \mathit{dataValue}(\mathit{in})$
        $\mathit{deltaWeight}(u) := 0$
    **else**
        **if** $(u \in \mathit{inputUnits})$ **then**
            **let** $([\mathit{in}] = \mathit{inForward}^{\mathrm{ext}}(u))$ **in**
                $\mathit{deltaWeight}(u) := \mathit{deltaWeight}(u) + \mathit{backwardResult} \cdot \mathit{dataValue}(\mathit{in})$
        **else**
            **let** $(\{s\} = \mathit{source}(u))$ **in**
                **let** $(\mathit{in} = \mathit{inForward}^{\mathrm{int}}(u, s)))$ **in**
                    $\mathit{deltaWeight}(u) := \mathit{deltaWeight}(u) + \mathit{backwardResult} \cdot \mathit{dataValue}(\mathit{in})$

$\mathit{updateLocalStateBackward}(u : \mathit{JUNCTION}, \mathit{backwardResult}) =$ **skip**

$\mathit{updateLocalStateBackward}(u : \mathit{FUNCTIONAL}, \mathit{backwardResult}) =$ **skip**
```

For this example we assume the network composed of the two basic computational units *neurons* and *synapses*, which in terms of *ASMs* are represented by the domains *SYNAPSE* and *NEURON*. Thus the parameters to be trained (the weights) are contained in the synapse units. This distinction is due to the need of weight sharing. Actually, in the Boltzmann machine, each synapse is bi-directional, however, since in the *NK* all units can have only one direction for each flow type (forward and backward), we have to simulate the bi-directionality using two connections with a shared weight. The connections can share a part of their internal state using the function:

$$\mathit{share} : \mathit{UNIT} \rightarrow \mathit{UNIT}$$

which returns the name of the unit sharing the state with the current unit. For example, when creating two synapses that share the weights, there should be an update as follows:

$$\mathit{share}(s_1) := s_1$$
$$\mathit{share}(s_2) := \mathit{share}(s_1)$$

Thus, when accessing the weights of the synapses we access always the same shared weight:

$$\mathit{weight}(\mathit{share}(s_1)) \equiv \mathit{weight}(s_1)$$
$$\mathit{weight}(\mathit{share}(s_2)) \equiv \mathit{weight}(s_1)$$

During the forward computation the neurons perform the stochastic computation, while the synapses transmit the weighted input. In the backward computations, instead, the neurons simply transmit to the synapses the result of their previous computation, while the synapses perform the weight update. Since the weights are shared by couples of synapses, in the training phase only one of the shared links should be updated.

Notice that while for classic feedforward networks the scheduling policy of the active units is driven by the topology of the network, in the Boltzmann machine it is driven by the annealing policy (e.g. the *Gibbs Sampling*). Moreover, during training we can perform a simultaneous computation of the units followed by a simultaneous computation of part of the synapses which contain the shared weight.

4.2.1 Forward Computation

In this example, the *forwardValue* function, which returns the result of a unit computation is defi ned according to the unit type. The synapses return their input multiplied by their weight, while the neurons return the result of the stochastic computation over the sum of the inputs:

```
$\mathit{forwardValue}(u : \mathit{SYNAPSE}) =$
    let $(\{s\} = \mathit{source}(u))$ in $\mathit{inForward}^{\mathrm{int}}(u, s)) \cdot \mathit{weight}(\mathit{shared}(u))$

$\mathit{forwardValue}(u : \mathit{NEURON}) =$
    if $(\mathit{random}(0,1) \leq \mathit{sigmoidalFunction}(-\frac{1}{T} \sum_{s \in source(u)} \mathit{dataValue}(\mathit{inForward}^{\mathrm{int}}(u, s)))$ then
        $+1$
    else
        $-1$
```

The result of the forward computation is then propagated to all the following units with the rule *propagateForward*, which in turn uses the functions:

$\textit{extValueForw}(u, \textit{dataToPropagate}) = \textbf{skip}$
$\textit{intValueForw}(d, u, \textit{dataToPropagate}) = (\textit{undef}, \textit{dataToPropagate})$

Finally the local state of the units is updated by the rules:

$\textit{updateLocalStateForward}(u : \textit{SYNAPSE}, \textit{forwardResult}) = \textbf{skip}$
$\textit{updateLocalStateForward}(u : \textit{NEURON}, \textit{forwardResult}) = \textit{state}(u) := \textit{forwardResult}$

4.2.2 Backward Computation

In the backward propagation, the neural units only need to transmit their actual state to all the adjacent input synapses, while the synapses do not need to transfer any information.

$\textit{backwardValue}(u : \textit{NEURON}) = \textit{state}(u)$
$\textit{backwardValue}(u : \textit{SYNAPSE}) = \textbf{skip}$

Then the result should be transmitted to all the preceding units of the current one using the rule *propagateBackward*, which in turn uses the functions:

$\textit{extValueBack}(u, \textit{dataToPropagate}) = \textbf{skip}$
$\textit{intValueBack}(d, u : \textit{SYNAPSE}, \textit{dataToPropagate}) = \textbf{skip}$
$\textit{intValueBack}(d, u : \textit{NEURON}, \textit{dataToPropagate}) = (\textit{undef}, \textit{dataToPropagate})$

Finally the local state is updated according to the following rules:

```
$\textit{updateLocalStateBackward}(u : \textit{NEURON}, \textit{backwardResult}) = \textbf{skip}$

$\textit{updateLocalStateBackward}(u : \textit{SYNAPSE}, \textit{backwardResult}) =$
  let $(\{s\} = \textit{source}(u), \{d\} = \textit{dest}(u))$ in
      case $(\textit{backwardType}(\textit{inputCopy}))$ of
          $\textit{counting} \rightarrow$
              if $(\textit{clamped}(\textit{inputCopy}))$ then
                  $\textit{clampedCount}(u) := \textit{inForward}^{\mathrm{int}}(u, s) \cdot \textit{inBackward}^{\mathrm{int}}(u, d)$
                  $\textit{clampedCyclesCount}(u) + +$
              else
                  $\textit{freeRunningCount}(u) := \textit{inForward}^{\mathrm{int}}(u, s) \cdot \textit{inBackward}^{\mathrm{int}}(u, d)$
                  $\textit{freeRunningCyclesCount}(u) + +$
          $\textit{endPattern} \rightarrow$
              $\textit{patternNumber}(u) + +$
              $\textit{clampedMeans}(u) + = \textit{clampedCount}(u) \ / \ \textit{clampedCyclesCount}(u)$
              $\textit{clampedCount}(u) := 0$
              $\textit{clampedCyclesCount}(u) := 0$
              $\textit{freeRunningMeans}(u) + = \textit{freeRunningCount}(u) \ / \ \textit{freeRunningCyclesCount}(u)$
              $\textit{freeRunningCount}(u) := 0$
              $\textit{freeRunningCyclesCount}(u) := 0$
          $\textit{updateWeights} \rightarrow$
              let $\textit{clampedProb} = \textit{clampedMeans}(u) \ / \ \textit{patternNumber}(u)$
                  $\textit{freeRunningProb} = \textit{freeRunningMeans}(u) \ / \ \textit{patternNumber}(u)$
              in
                  $\textit{weight}(\textit{shared}(u)) + = \textit{epsilon} \cdot (\textit{clampedProb} - \textit{freeRunningProb}) \ / \ T$
                  $\textit{clampedMeans}(u) := 0$
                  $\textit{freeRunningMeans}(u) := 0$
                  $\textit{patternNumber}(u) := 0$
          endcase
```

Note that the training algorithm is implemented by the synapses.

5 Conclusion and Future Works

In the introduction we briefly argued on the need for a *Neural Abstract Machine*, i.e., a formally precise defi nition of the basic (and relevant) computational objects as well as operations which are manipulated and performed, respectively, by neural computation. To defi ne the *Neural Abstract Machine*, we suggested to use *Abstract State Machines*, which have been extensively used for the formal design and analysis of various hardware systems, algorithms and programming languages semantics. They allow to specify formal systems in a very simple way, while preserving mathematical soundness and completeness.

We have reported the partial defi nition of the *Neural Kernel*, i.e. the core of the *Neural Abstract Machine*, and we have given two examples of how different neural models can be modeled in this framework.

Of course, we are only at an initial stage of development of the abstract machine. We need to consider recurrent and recursive models, topology growing algorithms, and so on. Moreover, we will need to defi ne a Neural Programming Language, based on the abstract machine, which should allow the user to describe novel models and learning algorithms by simply giving a formal specifi cation. Finally, the implementation of the abstract machine by using the Microsoft© tool *asmL* will be considered.

References

[1] M.R. Berthold and I. Fischer. Formalizing neural networks using graph transformations. In *Proc. of the IEEE Int. Conf. on Neural Networks*, volume 1, pages 275–280. IEEE, 1997.

[2] E. Börger. Why use evolving algebras for hardware and software engineering? In J. Wiedermann M. Bartosek, J. Staudek, editor, *SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer-Verlag, 1995.

[3] E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, pages 1–43. Springer-Verlag, 1999.

[4] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Proc. CSL'2000 (Gurevich Festschrift)*, LNCS. Springer-Verlag, 2000. To appear.

[5] E. Börger and D. Sona. A neural abstract machine. *To appear in LNCS*, 2001.

[6] P. Campolucci, A. Uncini, and F. Piazza. A signal-fbw-graph approach to on-line gradient calculation. *Neural Computation*, 12(8):1901–1927, 2000.

[7] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[8] O. Nerrand, P. Roussel-Ragot, L. Personnaz, G. Dreyfus, and S. Marcos. Neural networks and nonlinear adaptive filtering: Unifying concepts and new algorithms. *Neural Computation*, 5(2):165–199, 1993.

[9] S. Santini, A. Del Bimbo, and R. Jain. Block structured recurrent neural networks. *Neural Networks*, 8:135–147, 1995.

[10] A.C. Tsoi and A. Back. Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15:183–223, 1997.

[11] E.A. Wan and F. Beaufays. Diagrammatic methods for deriving and relating temporal neural network algorithms. *Lecture Notes in Computer Science*, 1387:63–98, 1998.