Neural Networks in Maude[⋆]

Gustavo Santos-García[1]

Universidad de Salamanca

Miguel Palomino[2]

Departamento de Sistemas Informáticos y Computación, UCM

Alberto Verdejo[3]

Departamento de Sistemas Informáticos y Computación, UCM

Abstract

In this work we study the representation of the computational model of artificial neural networks in rewriting logic, along the lines of several models of parallelism and concurrency that have already been mapped into it. We show how crucial is the right choice for the representation operations and the availability of strategies to guide the application of our rules. Finally, we also apply our specification to data used in the diagnosis of glaucoma.

Keywords: Neural networks, rewriting logic, Maude, strategies, executability.

1 Introduction

Rewriting logic [11] is a logic of concurrent change that can naturally deal with states and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency. Indeed, rewriting logic was proposed as a unifying framework in which many models of concurrency could be represented, such as labeled transition systems, phrase structure grammars, Petri nets, concurrent object-oriented programming, or CCS, to name a few. For many of these models, concrete maps have actually been defined into rewriting logic; see e.g. [15,14,12,7,16] and the references in [8].

[⋆] Research supported by Spanish project DESAFIOS TIN2006–15660–C02–01 and by Comunidad de Madrid program PROMESAS S–0505/TIC/0407.

[1] Email:santos@usal.es

[2] Email:miguelpt@sip.ucm.es

[3] Email:alberto@sip.ucm.es

Artificial neural networks [6] are another important model of parallel computation. This model attempts to imitate the operation of biological neural networks. In [10] it was argued that rewriting logic was also a convenient framework in which to embed artificial neural nets, and a possible representation was sketched. However, and to the best of our knowledge, no concrete map has ever been constructed either following those ideas or any others. Our goal with this paper is to fill this gap.

In our representation of artificial neural networks we consider two stages: a first one, in which we are only concerned with the evaluation of patterns by the network, and a second one, in which the output produced by those patterns is used to adapt (or *train*) the net so as to make it more precise (that is, so that the error between the actual network output and the target output provided by the training set is reduced). It turns out that while the representation suggested in [10] allows a straightforward implementation of the first stage, we were unable to make use of it for the training phase. Therefore, we propose an alternative representation in which the specification of both stages proceeds smoothly.

Since its conception, rewriting logic was proposed as the foundation of an efficient executable system called Maude [1]. Here we write our representation directly in Maude to be able to run our neural networks and apply them to a real case-study, that is the analysis of campimetric fields and nerve fibres of the retina for the diagnosis of glaucoma on patients from the University Hospital in Salamanca [3].

The paper is organized as follows. In Section 2 we review those aspects of Maude that will be used in the specification of artificial neural nets, mainly object-oriented modules and strategies, to make the paper (almost) self-contained. Section 3 introduces multilayer perceptron nets, their specification in Maude, and an appropriate strategy for their evaluation. The backpropagation algorithm for neural network training is presented in Section 4, together with the changes necessary in the specification of the network and a training strategy. Section 5 discusses the application of our implementation to the study of the diagnosis of glaucoma and Section 6 concludes. The Maude code, the data used for the examples, and the output can be downloaded from `http://maude.sip.ucm.es/~miguelpt/`.

2 Maude

Maude [1] is a high performance language and system supporting both equational and rewriting logic computation for a wide range of applications. The key novelty of Maude is that besides efficiently supporting equational computation and algebraic specification it also supports rewriting logic computation. Mathematically, a rewrite rule has the form $l : t \longrightarrow t'$ *if Cond* with t, t' terms of the same kind which may contain variables. Intuitively, a rule describes a local concurrent transition in a system: anywhere a substitution instance $\sigma(t)$ is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place.

Full Maude [1] is an extension of Maude with a *richer module algebra* of parameterized modules and module composition operations and with special syntax for object-oriented specifications. These object-oriented modules have been exploited for specifying artificial neural networks.

2.1 Object oriented modules

An *object* in a given state is represented as a term `< O : C | a1 : v1,..., an : vn >` where `O` is the object's name, belonging to a set `Oid` of object identifiers, `C` is its *class*, the `ai`'s are the names of the object's *attributes*, and the `vi`'s are their corresponding values. *Messages* are defined by the user for each application.

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms of associativity, commutativity, and identity) using rules that describe the effects of *communication events* between some objects and messages. The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The general form of such rules is

$$\begin{aligned}
& M_1 \ldots M_n \ \langle O_1 : F_1 \mid \mathit{atts}_1\rangle \ldots \langle O_m : F_m \mid \mathit{atts}_m\rangle \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid \mathit{atts}'_{i_1}\rangle \ldots \langle O_{i_k} : F'_{i_k} \mid \mathit{atts}'_{i_k}\rangle \\
& \qquad \langle Q_1 : D_1 \mid \mathit{atts}''_1\rangle \ldots \langle Q_p : D_p \mid \mathit{atts}''_p\rangle \\
& \qquad M'_1 \ldots M'_q \\
& \quad \mathit{if}\ \mathit{Cond}
\end{aligned}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and *Cond* is a rule condition. The result of applying a rewrite rule is that the messages $M_1, \ldots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \ldots, Q_p$ are created; and new messages $M'_1, \ldots, M'_q$ are sent.

Since the above rule involves several objects and messages in its lefthand side, we say that it is a *synchronous rule.* It is conceptually important to distinguish the special case of rules involving at most one object and one message in their lefthand side. These rules are called *asynchronous* and have the form

$$\begin{aligned}
& (M)\ \langle O : F \mid \mathit{atts}\rangle \\
& \longrightarrow (\langle O : F' \mid \mathit{atts}'\rangle) \\
& \qquad \langle Q_1 : D_1 \mid \mathit{atts}''_1\rangle \ldots \langle Q_p : D_p \mid \mathit{atts}''_p\rangle \\
& \qquad M'_1 \ldots M'_q \\
& \quad \mathit{if}\ \mathit{Cond}
\end{aligned}$$

By convention, the only object attributes made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only in the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only in the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged.

2.2 Maude's strategy language

Rewrite rules in rewriting logic need to be neither confluent nor terminating. This theoretical generality requires some control when the specifications become executable, because it must be ensured that the rewriting process does not go in undesired directions and eventually terminates. Maude's strategy language can be used to control how rules are applied to rewrite a term [9,2]. Strategies are defined in a separate module and are run from the prompt through special commands.

The simplest strategies are the constants `idle`, which always succeeds by doing nothing, and `fail`, which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and with the possibility of providing a substitution for the variables in the rule. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control how the rewrite conditions are solved by means of strategies. An operation `top` to restrict the application of a rule just to the *top* of the term is also provided. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation (`;`), union (`|`), and iteration (`*` for 0 or more iterations, `+` for 1 or more, and `!` for a 'repeat until the end' iteration). Another strategy combinator is a typical 'if-then-else', but generalized so that the first argument is also a strategy. By using this combinator, we can define many other useful strategy combinators as derived operations: for example a binary `orelse` combinator that applies the second argument strategy only if the first fails, and a unary `not` combinator that fails when its argument is successful and vice versa. The language also provides a `matchrew` combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten. Moreover, an extended `matchrew` combinator, that is `amatchrew`, is also provided to support rewriting modulo axioms of associativity, commutativity, identity, and idempotency is considered, when declared. Recursion is also possible by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies.

For our implementation, the full expressive power of the strategy language will not be needed and all our strategies will be expressed as combinations of the application of certain rules (possibly instantiated), concatenation (`;`), and 'repeat until the end' iteration (`!`). For efficiency reasons, we have extended the previous strategy language with a new combinator `one`(S) which, when applied to a term t, returns one of the possible solutions of applying S to t. However, our use of this combinator together with the `!` combinator guarantees that no final solution is lost.

3 Specification of neural networks

Artificial neural network models have been extensively studied and applied in recent years in the hope of reaching human performance in different fields, including, for instance, automatic speech recognition, image processing, and biomedical applications [3,5,6,17]. Models of artificial neural networks are made up of many nodes or computational elements, called "neurons", which work in parallel and are con-

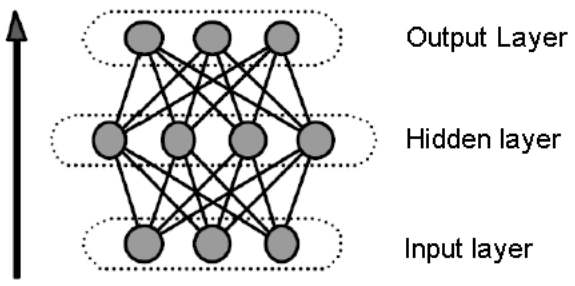

Fig. 1. A fully-connected, three-layer feed-forward perceptron neural network.

nected to each other by "weights"—transmission coefficients—that are usually updated throughout the computation in order to increase the precision of the network diagnoses.

Among the different models of neural networks we have chosen to work with multilayer perceptrons. Perceptrons are classifiers that learn to build complex separating hyperplanes that partition the input space in decision regions based on the information provided during training; a multilayer perceptron with one hidden layer with step transfer functions can solve any problem with arbitrary decision regions [18].

3.1 Multilayer perceptron networks

A *neural network* is defined in mathematical terms as a graph with the following properties: (1) each node i, called *neuron*, is associated with a state variable x_i storing its current output; (2) each junction between two neurons i and k, called *synapse* or *link*, is associated with a real weight ω_{ik}; (3) a real threshold θ_i, called *activation threshold*, is associated with each neuron i; (4) a *transfer function* $f_i[n_k, \omega_{ik}, \theta_i, (k \neq i)]$ is defined for each neuron, and determines the activation degree of the neuron as a function of its threshold, the weights of the input junctions and the outputs n_k of the neurons connected to its input synapses. In our case, the transfer function has the form $f(\sum_k \omega_{ik} n_k - \theta_i)$, where $f(x)$ is, in our case, a sigmoidal function, defined by $f(x) = 1/(1 + e^{(\nu - x)})$, which corresponds to the continuous and derivable generalization of the step function.

Multilayer perceptrons are networks with one or more layers of nodes between the layer of input units and the layer of output nodes; Figure 1 shows a three-layer perceptron. These layers contain hidden units or nodes which obtain their input from the previous layer and output their results to the next layer, to both of which they are fully-connected. Nodes within each layer are not connected and have the same transfer function. The strength of the multilayer perceptron originates from the use of non-linear sigmoidal functions in the nodes. If the nodes were linear elements, then monolayer networks with appropriately selected weights could repeat the calculations carried out by a multilayer network [18]. A multilayer perceptron with a non-linear step function and a hidden layer can solve problems in which the decision regions are open or closed convex regions. *In the case of perceptrons with one hidden layer, problems with arbitrary decision regions can be solved, but more complex regions will need a greater number of nodes in the network* [4,5].

The accuracy of the multilayer perceptron depends basically on the correct learning of the connection weights between nodes. The backpropagation training algorithm is an algorithm which uses a gradient descent method to minimize the mean quadratic error between the actual output of the perceptron and the desired output. In this section we focus on specifying three-layers perceptrons in Maude and designing a strategy for evaluation; we consider the backpropagation algorithm in Section 4.

3.2 Evaluation of perceptrons

The core of our representation of perceptrons in Maude revolves around the definition of two object-oriented classes to represent neurons and synapses as individual objects.

```
class Neuron | x : Float, t : Float, st : Nat .
class Link | w : Float, st : Nat .
```

Each neuron object carries its current activation value `x`, depending on its threshold `t`, and an attribute `st` that will be used to determine whether the neuron has already fired or not, that is, whether it is still waiting for input or has already output a value. Similarly, synapse objects store their numerical weight and contain an attribute `st` to flag whether some value has already passed through them or not. A net then is a "soup" (a multiset) of neurons and synapses.

Neurons and links are identified by a name. We define two operations that take natural numbers as arguments and return an object identifier: for neurons, the numbers correspond to the layer and the position within the layer; for links, the numbers correspond to the output layer and the respective positions within each layer of the neurons connected.

```
op neuron : Nat Nat -> Oid .
op link : Nat Nat Nat -> Oid .
```

The evaluation of the network is essentially performed by repeated application of the following two rules:

```
rl [feedForward] :
  < neuron(  L, I) : Neuron |  x : X1 , st : 1 >
  < neuron(s L, J) : Neuron |  x : X2 , st : 0 >
  < link(s L, I, J) : Link | w : W , st : 0 >
=>
  < neuron(  L, I) : Neuron |  x : X1 , st : 1 >
  < neuron(s L, J) : Neuron |  x : (X2 + (X1 * W)) , st : 0 >
  < link(s L, I, J) : Link | w : W , st : 1 > .

rl [sygmoid] :
  < neuron(L, I) : Neuron |  x : X , t : T , st : 0 >
=>
  < neuron(L, I) : Neuron |  x : syg(_-_(X, T), L) ,  st : 1 > .
```

Rule `feedForward` calculates the weighted sum of the inputs to the neuron, whereas `sygmoid` just applies the sigmoidal function `syg` (defined somewhere else in the code) to the net input.

As can be seen in `feedForward`, the attribute `st` of links is assumed to be 0 prior to their firing and becomes 1 once the information has been sent from one neuron to the other. Hence, pending some kind of reset, links can only be used once. Similarly, the rule `sygmoid` sets the attribute `st` of a neuron to 1 once the sigmoidal function has been applied. Note that this last rule can be triggered before `feedForward` has considered all possible links, thus producing an incorrect result: our evaluation strategy will take care of not executing `sygmoid` while `feedForward` is still enabled.

3.3 The introduction of data

In order to have a running net we need to specify the number of layers, the neurons in each of them, the weights of all links, and the input patterns which, in general, will be multiple. Whereas the object-oriented representation presented in the previous section is very convenient for specifying their behavior, it is clear that introducing all these data directly in this form would be very cumbersome. Hence, we have decided to use matrices and vectors of values to specify thresholds and weights, defining equations and rules to transform them into the object representation.

We define two operations to construct neurons and links:

```
op neuronGeneration : Nat FloatList FloatList Nat -> Configuration .
op linkGeneration : Nat Matrix Nat Nat -> Configuration .
```

The operation `neuronGeneration` constructs *all* neurons in the layer specified in its first argument and `linkGeneration` constructs *all* links between two consecutive layers. For that, the second and third arguments of `neuronGeneration` are vectors or lists of real numbers with the corresponding value and threshold for each neuron. Similarly, the second argument of `linkGeneration` is a matrix of reals, that is, a list of lists where each row (list) contains the weights of all links from a neuron in a certain layer to all neurons in the next layer. The third and fourth arguments are used during the process of building the object-oriented representation; essentially, they act as counters to identify neurons and links and are increased every time a value is removed from one of the lists or the matrices.

Objects for individual neurons with the correct values from the vector representation are obtained by repeated application of the following two equations:

```
eq  neuronGeneration(L, X LX, T LT, S)
=   neuronGeneration(L, LX, LT, s S)
    < neuron(L, s S) : Neuron |  x : X , t : T , st : 0 > .

eq  neuronGeneration(L, X, T, S)
=   < neuron(L, s S) : Neuron |  x : X , t : T , st : 0 > .
```

The argument `L` in the operation `neuronGeneration` identifies the layer the data pertain to.

Similarly, individual links are obtained from the matrix representation by means of the equations:

```
eq  linkGeneration(L, ((W LW) ; MW), S, S2)
=   linkGeneration(L, (LW ; MW), s S, S2)
    < link(L, s S2, s S) : Link | w : W , st : 0 > .

eq  linkGeneration(L, (W ; MW), S, S2)
=   linkGeneration(L, MW, 0, s S2)
    < link(L, s S2, s S) : Link | w : W , st : 0 > .
```

```
eq  linkGeneration(L, (W LW), S, S2)
=   linkGeneration(L, LW, s S, S2)
    < link(L, s S2, s S) : Link | w : W , st : 0 > .

eq  linkGeneration(L, W, S, S2)
=   < link(L, s S2, s S) : Link | w : W , st : 0 > .
```

Besides the information about neurons and links, we also need a means to specify the input, that is, the patterns the neural network is going to work with and, when available, the desired output they should produce. (This output is not needed for evaluation purposes, but it is required to train the net.) Again, we will use two operations to build our object-oriented representation in which the input/output to a single neuron constitutes an individual message, from vectors storing all patterns to a layer.

```
ops inPatternConversion : Nat FloatList Nat -> Configuration .
op outPatternConversion : Nat FloatList Nat -> Configuration .

--- Input-output for a single neuron
msg inputPattern : Nat Nat Float -> Msg .
msg outputPattern : Nat Nat Float Nat -> Msg .
```

The representation of input patterns is constructed as for neurons and links:

```
eq inPatternConversion(N, X LX, S)
=  inPatternConversion(N, LX, s S)
   inputPattern(N, s S, X) .

eq inPatternConversion(N, X, S)
=  inputPattern(N, s S, X) .
```

As for the output patterns, objects of a new class `Net` will also be created to hold the difference between the generated output and the desired value.

```
class Net | e : Float, r : Bool, st : Nat .

eq outPatternConversion(N, X LX, S)
=  outPatternConversion(N, LX, s S)
   outputPattern(N, s S, X, 0) .

eq outPatternConversion(N, X, S)
=  outputPattern(N, s S, X, 0)
   < net(N) : Net |  e : 0.0 , r : false , st : 0 > .
```

Evaluation of a perceptron starts by obtaining an input pattern through the rule `nextPattern`, which is guided by the message `netStatus`. A message of the form `netStatus(N0, 0, 0, N1)` means that the `s N0`-th pattern should be considered, and then the following patterns until the `N1`-th.

```
msg netStatus : Nat Nat Nat Nat -> Msg .

crl [nextPattern] :   --- use next pattern
netStatus(N , N1 , N2 , N0)  =>
netStatus(s N , N1 , N2 , N0)
inPatternConversion(s N, inputPattern(s N), 0)
outPatternConversion(s N, outputPattern(s N), 0)
if  N < N0 .
```

Before starting the feedforward process, the values of the neurons in the input layer and the corresponding weights are reset:

```
rl [resetNeuron] :
< neuron(L, I) : Neuron | x : X , st : s S >
=>
< neuron(L, I) : Neuron | x : 0.0 , st : 0 > .

rl [resetLink] :
< link(L, I, J) : Link  | w : W , st : 1 >
=>
< link(L, I, J) : Link  | w : W , st : 0 > .
```

After the perceptron has been initialized, the rule `introducePattern` inserts the input pattern in the neurons of the input layer and removes them from the configuration. Note that the input layer is identified with the number 0.

```
rl [introducePattern] :
inputPattern(N, I, X0)
< neuron(0, I)  : Neuron  |  x : X , st : 0 >
=>
< neuron(0, I)  : Neuron  |  x : X0 , st : 1 > .
```

Once we are done with the evaluation of all patterns (by means of the rules `feedForward` and `sygmoid` of the previous section), we compute the error and mark the current object `net(N)` as completed.

```
rl [computeError] :
< neuron(2, I) : Neuron | x : X0 , st : 1 >
outputPattern(N, I, X1, 0)
< net(N0) : Net | e : E , st : 0 >
=>
< neuron(2, I) : Neuron | x : X0 , st : 1 >
outputPattern(N, I, X1, 1)
< net(N0) : Net | e : (E + ((_-_(X1, X0)) * (_-_(X1, X0)))) , st : 0 > .
```

After the error has been computed for all neurons in the output layer (again, a strategy will take care of that), the attribute `r` stores whether the result is admissible by checking if the error is less than a previously defined constant `tol`.

```
rl [setNet] :
< net(N) : Net | e : E , r : B , st : 0 >
=>
< net(N) : Net | e : E , r : (E <= tol) , st : 2 > .
```

3.4 Shepherding the perceptron: evaluation

As remarked at the end of Section 3.2, our specification is nondeterministic and not all of its possible executions correspond to valid behaviors of a perceptron. Hence, in order to be able to use the specification to simulate the evaluation of patterns we need to control the order of application of the different rules by means of strategies.

The main strategy is

```
strat feedForwardStrat : Nat @ Configuration .
sd feedForwardStrat(L') :=
   ( one(feedForward[L <- L']) ! ; one(sygmoid[L <- s L']) ! )  .
```

This strategy, which takes a natural number as argument an applies to a `Configuration` (that is, a perceptron), chooses a layer `L'` and applies rule `feedForward`, at random positions and as long as it is enabled, to compute the weighted sum of values associated to each neuron at the layer. When all sums have been calculated, it applies the sigmoidal function to all of them by means of rule `sygmoid` which, again, is applied at random positions and as long as it is enabled.

There are two additional auxiliary strategies:

```
strat inputPatternStrat : @ Configuration .
sd inputPatternStrat :=
 ( one(resetNeuron) ! ; one(resetLink) ! ; one(nextPattern) ) .

strat computeOutput : @ Configuration .
sd computeOutput :=
   ( one(computeError) ! ; setNet )  .
```

The strategy `inputPatternStrat` takes care of making the successive patterns available and of resetting the appropriate attributes of the neurons and links, whereas `computeOutput` is invoked to compute the error once a pattern has been evaluated.

Last, all these previous strategies are combined into the evaluation strategy, which inputs the next pattern, computes the values of the neurons in the hidden and the output layers, and returns the error:

```
strat evaluateANN : @ Configuration .
sd evaluateANN := ( inputPatternStrat ; feedForwardStrat(0) ;
                    feedForwardStrat(1) ;  computeOutput ) .
```

Then, to force the evaluation of the first `M` patterns by the multilayer perceptron the following command would be executed:

```
(srew ann netStatus(0, 0, 0, M) using one(evaluateANN) ! .)
```

where `ann` would be a term of the form

```
neuronGeneration(0, input0, threshold0, 0)
neuronGeneration(1, input1, threshold1, 0)  linkGeneration(1, link1, 0, 0)
neuronGeneration(2, input2, threshold2, 0)  linkGeneration(2, link2, 0, 0)
```

and the input patterns would have been suitable defined.

4 The backpropagation algorithm

The backpropagation training algorithm is an iterative gradient algorithm designed to minimize the mean square error between the actual and the desired network output. We recall it here as presented in [6].

Let us consider a perceptron with $c+1$ layers where layer 0 is the input layer and layer c is the output layer. Let N_i be the number of nodes in the i-th layer, where $i = 0, 1, \ldots, c$. Let x_{ij}^k be the input for the i pattern of node j of layer k. Let y_{ij}^k be the output for the i pattern of node j of layer k. By definition, the input and output values coincide for the nodes of the input layer.

Let ω_{ij}^k be the weight of the connection of neuron j of layer k with neuron i of the previous layer. By definition of the perceptron by layers, the following relationships are fulfilled

$$x_{ij}^k = \sum_l y_{il}^{k-1} \cdot \omega_{lj}^k; \quad y_{ij}^k = f(x_{ij}^k). \tag{1}$$

The mean quadratic error function between the real output of the perceptron and the desired output, for a particular pattern i, is defined as

$$E_i = \frac{1}{2}\sum_{j,k}(y_{ij}^k - d_{ij}^k)^2\,,$$

where d_{ij}^k is the desired output for pattern i of node j of layer k.

In order to minimize the error function we use the descending gradient function, considering the error function E_p and the weight sequence $\omega_{ij}^k(t)$, started randomly at time $t = 0$, and adapted to successive discrete time intervals. We then have

$$\omega_{ij}^k(t+1) = \omega_{ij}^k(t) - \eta\frac{\partial E_l}{\partial \omega_{ij}^k}(t), \tag{2}$$

where η is the so-called *learning rate constant*.

Learning is usually faster if a term of the moment of inertia is added and the changes of the connection weights are adapted as

$$\omega_{ij}^k(t+1) = \omega_{ij}^k(t) - \eta\frac{\partial E_l}{\partial \omega_{ij}^k}(t) + \alpha\left[\omega_{ij}^k(t) - \omega_{ij}^k(t-1)\right], \tag{3}$$

where α is a value between 0 and 1 called the *inertia constant*.

It is now necessary to calculate the quantities $\partial E_l/\partial\omega_{ij}^k$ and substitute in the expressions (2) or (3) for each one of the learning patterns, thus optimizing the values of the connection weights of the network.

Applying the rule of the chain in $\partial E_l/\partial\omega_{ij}^k$ and using the relationships (1), we can conclude that

$$\partial E_l/\partial\omega_{ij}^k = -(d_{ij}^k - y_{lj}^k)\cdot y_{lj}^k \cdot f'(x_{lj}^k).$$

From this equation it is possible to calculate $\partial E_l/\partial\omega_{ij}^k$ only for the nodes of the last layer, since in this layer the desired values are known. For the remaining layers we use the rule of the chain

$$\frac{\partial E_l}{\partial y_{lj}^k} = \sum_m \frac{\partial E_l}{\partial x_{lm}^{k+1}}\cdot\frac{\partial x_{lm}^{k+1}}{y_{lj}^k} = \sum_m \frac{\partial E_l}{\partial x_{lm}^{k+1}}\cdot\omega_{jm}^{k+1},$$

and, substituting,

$$\frac{\partial E_l}{\partial \omega_{ij}^k} = \frac{\partial E_l}{\partial x_{lj}^k}\cdot\frac{\partial x_{lj}^k}{\partial \omega_{ij}^k} = \left(\frac{\partial E_l}{\partial y_{lj}^k}\frac{\partial y_{lj}^k}{\partial x_{lj}^k}\right)\frac{\partial x_{lj}^k}{\partial \omega_{ij}^k} = \left(\sum_m \frac{\partial E_l}{\partial x_{lm}^{k+1}}\cdot\omega_{jm}^{k+1}\right)\cdot f'(x_{lj}^k)\cdot y_{li}^{k-1}. \tag{4}$$

We next review how it works [13,6], when particularized to the case in which the sigmoidal function is $f(x) = 1/(1+e^{\nu - x})$.

(i) Set all weights and node states to small random values.

(ii) Introduce the input vector $x_1,\ldots,x_n$ and the desired output $d_1,\ldots,d_m$.

(iii) Use the net to evaluate the actual output $y_1,\ldots,y_m$ produced by $x_1,\ldots,x_n$.

(iv) Adjust weights according to the equation

$$w_{ij}(t+1) = w_{ij}(t) + \eta\delta_j x_i'\,.$$

In this equation, $w_{ij}(t)$ is the weight of the link connecting neuron i to neuron j at time t, x_i' is the output of neuron i (input and output values coincide for the neurons of the input layer), η is a gain term or learning constant, and δ_j is an error term for node j. For output neurons:

$$\delta_j = y_j(1-y_j)(d_j - y_j)\,.$$

If node j belongs to a hidden layer, then

$$\delta_j = x'_j(1 - x'_j) \sum_k \delta_k w_{jk} \,,$$

where k ranges over all neurons in the layers above neuron j. Internal node thresholds are adapted in a similar manner. (However, since training thresholds is not necessary to optimally classify the data, we do not consider it in our implementation.)

The speed of convergence to a solution and the propensity to fall into local minima both depend heavily on the learning constant. Standard backpropagation is very sensitive to the initial learning rate chosen for a given learning task. In general the optimum value of the learning constant depends on the problem being solved: there is no single value suitable for different training cases.

4.1 Backpropagation in Maude

For training the net we need neurons and links to hold additional information, namely the error terms δ_j and the adjusted weights. Since evaluation is part of backpropagation, we define `NeuronTR` and `LinkTR` as *subclasses* of `Neuron` and `Link` with an additional attribute to store the extra information.

```
class LinkTR   | w+1 : Float .   subclass LinkTR < Link .
class NeuronTR | dt : Float .    subclass NeuronTR < Neuron .
```

Note that the rules for evaluating a net also apply to these new objects; the new attributes are simply ignored. Hence, we already have the code for the first three steps of the algorithm.

The next step demands the evaluation of the error terms δ_j before adjusting the weights. The calculation of these δ_j depends on whether we are working with the output layer or not. For the output layer, the corresponding rule is straightforward:

```
rl [delta2] :
< neuron(2, I) : NeuronTR | x : X , dt : DT , st : 2 >
outputPattern(N, I, D, 1)
=>
< neuron(2, I) : NeuronTR | x : X ,
      dt : (X * ((_-_(1.0, X)) * (_-_(D, X))) ) , st : 3 > .
```

The case for the remaining layers is a bit more involved and is split in three phases: the operation `delta1A` initializes `dt` to zero, `delta1B` below takes care of calculating the sum of the weights multiplied by the corresponding error term, and `delta1C` computes the final product.

```
rl [delta1A] :
  < neuron(1, I)  : Neuron |  dt : DT , st : 1  >  =>
  < neuron(1, I)  : Neuron |  dt : 0.0 , st : 2 > .

rl [delta1B] :
  < neuron(1, J)  : Neuron | dt : DT1 , st : 2 >
  < neuron(2, K)  : Neuron | dt : DT2 , st : 2 >
  < link(2, J, K) : Link   |  w : W ,   st : 2 >
=>
  < neuron(1, J)  : Neuron | dt : (DT1 + (DT2 * W)) , st : 2 >
  < neuron(2, K)  : Neuron | dt : DT2 , st : 2 >
  < link(2, J, K) : Link   |  w : W ,   st : 3 > .
```

```
rl [delta1C] :
  < neuron(1, J)  : Neuron |  x : X , dt : DT , st : 2 >  =>
  < neuron(1, J)  : Neuron |  x : X , dt : (DT * (X * (_-_(1.0, X)))) , st : 3 > .
```

Again, in all these rules the status attribute `st` is correspondingly updated.

Once the error terms are available, the updated weights can be calculated: rule `link1` does it for the hidden layer and `link2` for the output layer.

```
rl [link1] :
  < neuron(0, I)  : Neuron |  x : X1 , st : 1  >
  < neuron(1, J)  : Neuron | dt : DT , st : 3 >
  < link(1, I, J) : Link   |  w : W ,  w+1 : W1 , st : 1 >
=>
  < neuron(0, I)  : Neuron |  x : X1 , st : 1  >
  < neuron(1, J)  : Neuron | dt : DT , st : 3 >
  < link(1, I, J) : Link   |  w : W ,  w+1 : (W + (eta * (DT * X1))) , st : 3 > .

rl [link2] :
  < link(2, I, J) : Link   |  w : W ,  w+1 : W1 , st : 1 >
  < neuron(1, I)  : Neuron |  x : X1 , st : 2 >
  < neuron(2, J)  : Neuron | dt : DT , st : 1 >
=>
  < link(2, I, J) : Link   |  w : W ,  w+1 : (W + (eta * (DT * X1))) , st : 1 >
  < neuron(1, I)  : Neuron |  x : X1 , st : 2 >
  < neuron(2, J)  : Neuron | dt : DT , st : 2 > .
```

Last, the old weights are replaced by the adjusted ones.

```
rl [switchLink] :
  < link(N, I, J) : Link |  w : W ,  w+1 : W1 , st : s S >  =>
  < link(N, I, J) : Link |  w : W1 , w+1 : W ,  st : 0 > .
```

The reason why this rule (and a fortiori, the attribute `w+1`) is needed, instead of simply updating the value of the attribute `w` in the `link` rules, is because the old weights are used in the computation of the error terms δ and could be lost otherwise.

4.2 *Shepherding the perceptron: training*

As we did for evaluation, we need to define an appropriate strategy for training the perceptron. Assuming we have already calculated the output associated to a pattern, we next must calculate the error terms δ, use them to obtain the adjusted weights, and transfer them to the right attribute. That can be easily done by applying the rules defined in the previous section in the right order.

```
strat backpropagateANN : @ Configuration .
sd backpropagateANN :=
   ( one(delta2) ! ; one(link2) ! ;
     one(delta1A) ! ; one(delta1B) ! ; one(delta1C) ! ;  one(link1) !  ;
     one(switchLink) !  ) .
```

Finally, training a net consists in evaluating a pattern, with the strategy defined in Section 3.4, and then adjusting the weights accordingly.

```
strat stratANN :  @ Configuration .
sd stratANN := ( evaluateANN ; backpropagateANN ) .
```

5 Example: Diagnosis of glaucoma

For the diagnosis of glaucoma, Gustavo Santos-García participated in a project that proposed the use of an intelligent system that employs artificial neural networks and integrates the analysis of the nerve fibers of the retina from the study with scanning laser polarimetry (NFAII;GDx), perimetry and clinical data [3]. The resulting multilayer perceptron was trained using MatLab.

We used the data from that project as a test bed for our specification of the backpropagation algorithm in Maude. Our results coincided and the succes rate was of 100% but the execution time of our implementation lagged far behind, which motivated us to optimize our code. Since equations are executed much faster than rules by Maude and, in addition, do not give rise to branching but linear computations, easily handled by strategies, we simplified rules as much as possible. The technique used was the same in all cases and is illustrated here with `feedForward`:

```
rl [feedForward] : C => feedForward(C) .
op feedForward : Configuration -> Configuration .
eq feedForward(C < link(s L, I, J) : Link  | w : W , st : 0 >
        < neuron(  L, I) : Neuron | x : X1 , st : 1 >
        < neuron(s L, J) : Neuron | x : X2 , st : 0 >)
=  feedForward(C < link(s L, I, J) : Link  | w : W , st : 1 >
        < neuron(  L, I) : Neuron | >
        < neuron(s L, J) : Neuron | x : (X2 + (X1 * W)) >) .
eq feedForward(C) = C [owise] .
```

The evaluation and training strategies had to be correspondingly modified since the combinator `!` was no longer needed. The resulting specification is obviously less natural, but more efficient; however, it is still not competitive with MatLab.

6 Conclusions

We have presented in this paper a specification of a class of artificial neural networks, namely multilayer perceptrons, in a two step fashion. First we have shown how to use rewrite rules guided by strategies to simulate the evaluation of patterns by a perceptron, and then we have enhanced the specification to make the training of the net possible.

The evaluation process is straightforward, essentially amounting to the repeated application of two rules, `feedForward` and `sygmoid`, which further does credit to the suitability of rewriting logic as a framework for (yet another model of) concurrency. The training algorithm requires more rules, but the strategy is also rather simple.

However, the simplicity of the resulting specification should be put in perspective. First of all, the election of our concrete representation in which neurons and links are individual entities and which, at first sight, might not strike as the most appropriate, is of paramount importance. Indeed, our first attempts at specifying perceptrons made use of a vector representation like the one we have used here for inputting the data and similar to that proposed in [10]. Such representation was actually suitable for the evaluation of patterns but proved unmanageable when considering the training algorithm.

In addition to the representation, the availability of strategies turned out to be crucial. While with the vector representation layers could be considered as a whole and there was no much room for nondeterminism, the change to the object-oriented representation gave rise, as we have observed, to the possible interleaving of rules in an incorrect order. It then became essential the use of the strategy language to guide the rewriting process in the right direction.

As a result, our specification is the happy crossbreed of an object-oriented representation and the use of strategies: without the first the resulting specification would have been much more obscure, whereas without the availability of the strategy language, its interest would have been purely theoretical.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude — A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. A. Muñoz, editors, *6th International Workshop on Strategies in Automated Deduction, STRATEGIES'06, Seattle, Washington, August 16, 2006, Part of FLOC 2006*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007.

[3] E. Hernández Galilea, G. Santos-García, and I. Franco Suárez-Bárcena. Identification of glaucoma stages with artificial neural networks using retinal nerve fibre layer analysis and visual field parameters. In E. Corchado, J. M. Corchado, and A. Abraham, editors, *Innovations in Hybrid Intelligent Systems*, Advances in Soft Computing, pages 418–424. Springer, 2007.

[4] K. Hornik, M. Stinchcombe, H. White, and P. Auer. Degree of approximation results for feedforward networks approximating unknown mappings and their derivatives. *Neural Computation*, 6(6):1262–1275, 1994.

[5] A. Ilachinski. *Cellular automata*. World Scientific Publishing Company, 2001.

[6] R. P. Lippman. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, 1987.

[7] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay, editor, *Handbook of Philosophical Logic. Second Edition*, volume 9, pages 1–81. Kluwer Academic Press, 2002.

[8] N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[9] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Workshop on Rewriting Logic and its Applications, WRLA 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.

[10] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic: Marktoberdorf, Germany, July 29 – August 6 1997*, volume 165, pages 347–398. NATO Advanced Study Institute.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[12] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations.* MIT Press, 1986.

[14] M.-O. Stehr. A rewriting semantics for algebraic nets. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering — A Guide to Modeling, Verification, and Applications.* Springer, 2001.

[15] C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Workshop on Rewriting Logic and its Applications, WRLA'96*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 360–383. Elsevier, 1996.

[16] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Workshop on Rewriting Logic and its Applications, WRLA'02*, volume 71 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2002.

[17] B. Widrow and M. A. Lehr. 30 years of adaptative neural networks: Perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[18] B. Widrow and S. D. Sterns. *Adaptive signal processing.* Prentice Hall, 1985.