PARALLEL REWRITING IN NEURAL NETWORKS

Ekaterina Komendantskaya
School of Computer Science, University of St Andrews, U.K.

Keywords: Computational logic in neural networks, Neuro-symbolic networks, Abstract rewriting, Parallel term-rewriting, Unsupervised learning, Computer simulation of neural networks.

Abstract: Rewriting systems are used in various areas of computer science, and especially in lambda-calculus, higher-order logics and functional programming. We show that the unsupervised learning networks can implement parallel rewriting. We show how this general correspondence can be refined in order to perform parallel *term* rewriting in neural networks, for any given first-order term. We simulate these neural networks in the MATLAB Neural Network Toolbox and present the complete library of functions written in the MATLAB Neural Network Toolbox.

1 INTRODUCTION

Term rewriting (Terese, 2003) is a major area of research in theoretical computer science, and has found numerous applications in lambda calculus, higher-order logics and functional programming. Different forms of term-rewriting techniques underly various areas of automated reasoning.

A simple example of an abstract rewriting system would be a string together with a rule for rewriting the elements of the string. In more complex cases, the string can be given by some first-order term, there can be a system of rewriting rules rather than one rule, and, of course, the rewriting rules can be such that the initial string would be shortened or extended through the rewriting process. Certain rewriting systems would always lead to normal forms, some - not, and the process of reducing to a normal form can be finite or infinite. We will give formal definitions and explanations in Section 2.

If we have to build neural networks capable of automated reasoning, we would need to implement term-rewriting techniques into them; (Komendantskaya, 2009a). These methods can be further used in hybrid systems research.

There are several obstacles on our way. First problem is that, according to a general convention, neural networks do not process strings, or ordered sequences. Every neuron can accept only a scalar as a signal, and output a scalar in its turn. This general convention has been developed through decades of discussion, and different views on it are best summarised in (Aleksander and Morton, 1993; Smolensky and Legendre, 2006). However, it happens to be that some order is innate to neural networks: and this order is imposed by position of neurons in a given layer, and by positions of layers in a network. So, although each neuron accepts only a scalar number as an input, a layer of neurons accepts a vector of such numbers, and the whole network can accept a matrix of numbers.

This gives us the first basic assumption of the paper: **a vector of neurons in a layer mirrors the structure of a string**. This is why, we will use one layer networks throughout.

Related literature that concerns the structure processing with neural networks falls within three areas of research: the core method to deal with symbolic formulae and prolog terms (Bader et al., 2008); recursive networks which can deal with string trees (Strickert et al., 2005); and kernel methods for structures (Gärtner, 2003). The approach we pursue here does not follow any of the mentioned mainstream directions, but, as a pay-off, it is very direct and simple.

Having made the *first assumption* above, we still need to determine which of the parameters of a neural network will hold information about the elements of a given string. One easy solution could be to send a vector consisting of the elements of a string as an input to a chosen network. However, in this case the task of rewriting this string would be delegated to a processing function of the layer, whereas we wish to realise the process by means of learning. This reduces our options: conventionally, there are two parameters that

Komendantskaya E. (2009).
PARALLEL REWRITING IN NEURAL NETWORKS.
In *Proceedings of the International Joint Conference on Computational Intelligence*, pages 452-458
DOI: 10.5220/0002319704520458

can be trained in neural networks: these are weights and biases. Weights are used more often in learning and training, and so we choose weights to represent the string we wish to rewrite.

Thus, the second major assumption is: **adjusting weights of a network is similar to rewriting terms**.

So, given a string s, we construct one layer of neurons, with the weight vector $\mathbf{w}$ equal to s, and the linear transfer function $F(x) = x$. We will work with input signals equal to 1, so as to preserve the exact value of $\mathbf{w}$ at the first step. Next, we wish the process of training of this weight to correspond to steps of parallel rewriting. How close is conventional unsupervised learning implemented in neural networks to the term rewriting known in computational logic?

Consider a simple form of a Hebbian learning: given an input $\mathbf{x} = 1$ to the layer, and having received an output $\mathbf{y}$, the rate of change $\Delta\mathbf{w}$ for $\mathbf{w}$ is computed as follows: $\Delta\mathbf{w} = L(\mathbf{y},\mathbf{x})$, with L some chosen function. In a special case, it may be $\Delta\mathbf{w} = \eta\mathbf{yx}$, where η is a positive constant called the *rate of learning*. We take, for example, $\eta = 2$. At the first iteration, the output will be equal to $\mathbf{w}$, and so the network will compute $\Delta\mathbf{w} = 2\mathbf{w}$. At the next iteration, the network will modify its weight as follows: $\mathbf{w}^{\mathsf{new}} = \mathbf{w} + \Delta\mathbf{w} = \mathbf{w} + 2\mathbf{w} = 3\mathbf{w}$. And this value will be sent as the output, see also Section 3.

Interestingly enough, the conventional Hebbian network we have just described above does rewriting as we know it in computer science. In terms of term rewriting, it takes any string, and rewrites it according to the rewriting rule $\rho : x \to 3x$, albeit, as we will see in Section 2, we can use only ground instances of ρ. Given a string $[1,2,3,1,2,3,3,1,2]$ the network will transform it into $[3,6,9,3,6,9,9,3,6]$.

This justifies the third main assumption we use throughout: **unsupervised (Hebbian) learning provides a natural and elegant framework for implementing parallel rewriting in neural networks**.

These three assumptions lay the basis for the main definitions of Section 3. Additionally, in Sections 3 and 4, we show the ways to formalise the more complex cases of term-rewriting by means of unsupervised learning. These cases arise when one has more than one rewriting steps, and these steps are not instances of one rewriting rule, when the length of a given string changes in the process of rewriting, and also, when one uses first-order terms instead of abstract strings. In Section 3, we define the architecture and a simple unsupervised learning rule for neural networks that can perform abstract rewriting, with some restrictions on the shape and the number of rewriting steps. In Section 4, we refine the architecture of these neural networks and adapt them for the purpose of first-order *term* rewriting. We prove that for an arbitrary Term Rewriting System, these neural networks perform exactly the parallel term rewriting.

When moving from simple examples of rewriting systems to more specific and complex ones, all we have to do is to re-define the function L used in the definition of the learning rule $\Delta\mathbf{w} = L(\mathbf{y},\mathbf{x})$. While for some examples, as the one we have just considered, L is completely conventional, for other examples we define and test new functions (`rewrite`, `rewrite_mult`), using MATLAB Neural Network Simulator. The most complex of these functions - `rewrite_mult` - can support *rewriting by unsupervised learning* for any given Abstract or Term rewriting System.

Finally, in Section 5, we conclude the paper.

2 REWRITING SYSTEMS

In this section, we outline some basic notions used in the theory of Term-Rewriting, see (Terese, 2003).

The most basic and fundamental notion we encounter is the notion of an abstract reduction (or rewriting) system.

Definition 1. *An* abstract rewriting system *(ARS) is a structure $\mathcal{A} = (A, \{\to_\alpha \mid \alpha \in I\})$ consisting of a set A and a set of binary relations $\to_\alpha$ on A, indexed by a set I. We write $(A, \to_1, \to_2)$ instead of $(A, \{\to_\alpha \mid \alpha \in \{1,2\}\})$.*

A *term rewriting system* (TRS) consists of terms and rules for rewriting these terms. So we first need the terms. Briefly, they will be just the terms over a given first-order signature, as in the first-order predicate logic. Substitution is the operation of filling in terms for variables. See (Terese, 2003) for more details. Given terms, we define rewriting rules:

Definition 2. *A* reduction rule *(or rewrite rule) for a signature Σ is a pair $\langle l, r\rangle$ of terms of $Ter(\Sigma)$. It will be written $l \to r$, often with a name: $\rho : l \to r$. Two restrictions on reduction rules are imposed:*

- *the left-hand side l is not a variable;*
- *every variable occurring in the right-hand side r occurs in the left-hand side l as well.*

A reduction rule $\rho : l \to r$ can be viewed as a scheme. An instance of ρ is obtained by applying a substitution σ. The result is an atomic reduction step *$l^\sigma \to_\rho r^\sigma$. The left-hand side l^σ is called a* redex *and the right-hand side r^σ is called its* contractum.

Given a term, it may contain one or more occurrences of redexes. A rewriting step consists of contracting one of these, i.e., replacing the redex by its contractum.

Definition 3. *A* rewriting step *according to the rewriting rule* $\rho : l \to r$ *consists of contracting a redex within an arbitrary context:*

$$C[l^\sigma] \to_\rho C[r^\sigma]$$

We call $\to_\rho$ *the one-step rewriting relation generated by* ρ.

Definition 4.
- *A* term rewriting system *is a pair* $\mathcal{R} = (\Sigma, R)$ *of a signature* Σ *and a set of rewriting rules R for* Σ.
- *The* one-step rewriting *relation of* $\mathcal{R}$*, denoted by* $\to_R$*, is defined as the union* $\bigcup\{\to_\rho | \rho \in R\}$. *So we have* $t \to_R s$ *when* $t \to_\rho s$ *for one of the rewriting rules* $\rho \in R$.

Example 1. *Consider a rewrite rule* $\rho : F(G(x), y) \to F(x,x)$. *Then a substitution* σ, *with* $\sigma(x) = 0$ *and* $\sigma(y) = G(x)$, *yields the atomic reduction step*

$$\rho : F(G(0), G(x)) \to_\rho F(0,0)$$

with redex $F(G(0), G(x))$ *and contractum* $F(0,0)$. *The rule gives rise to (e.g.) the rewriting step*

$$F(z, G(F(G(0), G(x)))) \to_\rho F(z, G(F(0,0)))$$

Here the context is $F(z, G(\Box))$.

Example 2. *Consider the TRS with rewriting rules*

$$\rho_1 : F(a,x) \to G(x,x) \quad (1)$$
$$\rho_2 : b \to F(b,b) \quad (2)$$

- *The substitution* $[x := b]$ *yields the atomic rewriting step* $F(a,b) \to_{\rho_1} G(b,b)$.
- *A corresponding one-step rewriting is* $G(F(a,b),b) \to_{\rho_1} G(G(b,b),b)$.
- *Another one-step rewriting is* $G(F(a,b),b) \to_{\rho_2} G(F(a,b),F(b,b))$.

The notion of a *parallel rewriting* is central for establishing confluence; (Terese, 2003).

Definition 5. *Let a term t contain some disjoint redexes* $s_1, s_2, \ldots, s_n$*; that is, suppose we have* $t \equiv C[s_1, s_2, \ldots, s_n]$, *for some context C. Obviously, these redexes can be contracted in any order. If their contracta are respectively* $s'_1, s'_2, \ldots, s'_n$, *in n steps the reduct* $t' \equiv C[s'_1, s'_2, \ldots, s'_n]$ *can be reached. These n steps together are called a* parallel step.

Performing disjoint reductions in parallel brings significant speed-up to computations. However, very often the parallel steps are conceived or implemented as a sequence of disjoint rewriting steps. As we show in the next sections, term-rewriting implemented in neural networks does the parallel step not as a sequence, but truly in parallel.

3 UNSUPERVISED LEARNING AND ABSTRACT REWRITING

In this section, we define neural networks, following (Hecht-Nielsen, 1990; Haykin, 1994).

An *artificial neural network* (also called a neural network) is a directed graph. A *unit* k in this graph is characterised, at time t, by its *input vector* $(v_{i_1}(t), \ldots v_{i_n}(t))$, its potential $p_k(t)$, its bias b_k and its *value* $v_k(t)$. In what follows, we will use integers.

Units are connected via a set of directed and weighted connections. If there is a connection from unit j to unit k, then w_{kj} denotes the *weight* associated with this connection, and $i_k(t) = w_{kj} v_j(t)$ is the *input* received by k from j at time t. At each update, the potential and value of a unit are computed with respect to an *input (activation)* and an *output (transfer) functions* respectively. The units considered here compute their potential as the weighted sum of their inputs:

$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) \right). \quad \star$$

The units are updated synchronously, time becomes $t + \Delta t$, and the output value for k, $v_k(t + \Delta t)$, is calculated from $p_k(t)$ by means of a given *transfer function* F, that is, $v_k(t + \Delta t) = F(p_k(t))$.

A unit is said to be a *linear unit* if its transfer function is the identity. In this case, $v_k(t + \Delta t) = p_k(t)$.

We will consider networks where the units can be organised in layers. A *layer* is a vector of units.

In the rest of the paper, we will normally work with layers of neurons rather than with single neurons, and hence we will manipulate with vectors of weights, output signals, and other parameters. In this case, we can drop the subscripts and write simply w for the vector of weights.

There are two major kinds of learning distinguished in Neurocomputing: supervised and unsupervised learning. In this paper, we focus only on unsupervised learning.

Unsupervised learning in its different forms has the following common features. A network is given a learning rule, according to which it trains its weights. Adaptation is achieved by means of processing external signals, and applying the learning rule L. To train the weight $w_{kj}(t)$, we apply a learning function L to the input and output signals $v_j(t)$ and $v_k(t)$, and get $\Delta w_{kj}(t) = L(v_k(t), v_j(t))$. We will call the vector Δw the *change vector* for the weight vector w. As a particular case of this formula, one can have $\Delta w_{kj}(t) = \eta(v_k(t), v_j(t))$, where η is a positive constant called the *rate of learning*. At the next time step $t + 1$, the weight is changed to $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$.

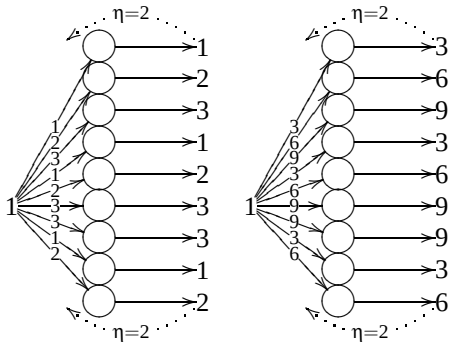

Figure 1: ARNN `net` at training steps 1 and 2.

We could perceive this learning function L as a rewriting rule for the weight w_{kj}, and the process of training would be the process of rewriting in this case. A suitable architecture for a network capable of performing abstract rewriting by unsupervised learning is given in the next definition, under the name *abstract rewriting neural network* (ARNN). For simplicity, we will first cover only ARS with one rewriting rule.

We adopt the following notation. For a given vector v, we denote its length by l_v. For a given string s, the vector that corresponds to it is denoted v_s, and the length of this vector is denoted by l_{v_s}.

Definition 6. *Given an ARS $\mathcal{A} = (A, \{\rightarrow_1\})$, and a sequence s of elements of A, an architecture for the* abstract rewriting neural network *(ARNN)* `net` *for s is defined as follows. Let v_s be the vector of elements of s. Let l_{v_s} be the length of v_s. Then* `net` *is constructed from one layer k of l_{v_s} neurons. Its weight vector w_{k1} is equal to v_s. The transfer function is taken to be identity. The network receives input signal* 1.

This definition realises the first two basic assumptions we outlined in Introduction. In the future, we will freely transform sequences of symbols into vectors, in the way we have done in the Definition 6. Because the input signal is equal to 1, the network built as in Definition 6 will always output v_s, as we further illustrate in the next example.

Example 3. *Given a set $A = \{1,2,3\}$, and a sequence $s = 1,2,3,1,2,3,3,1,2$, the corresponding ARNN is constructed as follows. We take one layer k of 9 neurons, and define the weight w_{k1} to be the vector $v_s = [1;2;3;1;2;3;3;1;2]$. Once initialised, the network will output the same vector: If we look at the equation $\star$, and put $j = 1$ (there is only one input), and $v_j = 1$, then the potential p_k will be equal to w_k. See Figure 1. This example and many more are also available in the file* `experiments_s` *in (Komendantskaya, 2009b).*

We have a learning rule to add, in order to enable the network to rewrite. Suppose we have a rewriting rule $\rho_1 : [a] \rightarrow [b]$, with vectors a and b of equal length, and we want to apply this rewriting rule. Following the usual conventions, and taking the input signal to be 1, the learning rule will take the output vector v_k and apply some learning function L to v_k, to form the change vector Δw_{1k}, and compute $w1k^{\mathsf{new}} = w_{1k}^{\mathsf{old}} + \Delta w_{1k}$. The only thing left is to define L.

As we mentioned in the introduction, in some cases we can use conventional Hebbian learning. For example, taking the rate of learning to be equal to 2, we can obtain the difference vector $\Delta w_{1k} = 2v_s$, for the network from Example 3. This will amend the weight $\mathbf{w_{1k}}^{\mathsf{new}} = \mathbf{w_{1k}} + \Delta\mathbf{w_{1k}} = \mathbf{v_s} + 2\mathbf{v_s} = 3\mathbf{v_s}$. Such a network would perform rewriting for ground instantiations of the rule $x \rightarrow 3x$. Applied to Example 3, it would give the result $[3;6;9;3;6;9;9;3;6]$, see Figure 1. But note that Definition 2 prohibits the use of the rewriting rules which contain a variable as a redex, and so we use three ground instances of this rule, substituting $1,2,3$ for x.

However, transformation of a rewriting rule into a linear function is not normally given, and not always possible. Therefore, we need to develop a more general approach. We define L, and call it `rewrite`; in the MATLAB library (Komendantskaya, 2009b) this function is called `rewrite2`.

Definition 7. (Function `Rewrite`). *Let $\mathcal{A} = (A, \{\rightarrow_1\})$ be an ARS, s_A be the given string, v_s its corresponding vector, and $\rho_1 : [a] \rightarrow [b]$ be the rewriting rule, where a and b are vectors of the same length m. Take zero vector Z of length l_s. Compute $\rho_1' = -([a] - [b])$. For every $n = \{1, \ldots, l_s\}$, do the following: If $n, n+1, \ldots (n + l_a - 1)$th elements of the vector v_k are equal to a, put ρ_1' on the $n \ldots (n + l_a - 1)$th place of Z.*

The function `rewrite` takes three arguments - the output vector v_k, and two vectors $[a]$ and $[b]$ that correspond to the left-hand side and the right-hand side of the rewriting rule ρ. It outputs the change vector that contains ρ_1' at precisely those positions where $[a]$ appeared in v_s, and zeros at all other positions. To simulate this in MATLAB, one has to choose a training mode - in the standard library the unsupervised training function is called `trainbuwb`. Then we define a new learning function `learnr` that is used by the training function. The learning function implements the function `rewrite`.

Note these subtle interconnections between the functions participating in training. The unsupervised training function (`trainbuwb`) activates the learning function (`learnr`) that computes Δw, and the latter is given by implementing the function `rewrite`. This

hierarchy is imposed by MATLAB Neural Network Toolbox, and we respect it throughout the paper.

Example 4. *We continue Example 3, and introduce a rewriting rule* $\rho_1 : [2\ 3] \rightarrow [2\ 1]$. *Now we can compute* `rewrite`$(v_s, [2\ 3], [2\ 1]) = [0;0;-2;0;0;-2;0;0;0]$. *After one iteration,* `net` *performs a parallel rewriting step computing:* $[1;2;1;1;2;1;3;1;2]$*; see the file* `experiments_s.mat` *in (Komendantskaya, 2009b).*

Lemma 1. *Given an ARS* $\mathcal{A} = (A, \{\rightarrow_1\})$*, such that the rewriting rule's redex and contractum are of the same size, given a sequence* s_A *of elements of A, there exists an Abstract Rewriting Neural Network (ARNN) that performs the parallel rewriting step for* s_A *in* $\mathcal{A}$.

Proof. The architecture of such a network is given in Definition 6, and the learning function (called `learnr` in the MATLAB library) implements $L =$ `rewrite` from Definition 7. □

We have illustrated, on a limited class of ARSs, that term-rewriting evolves naturally in unsupervised learning neural networks. In the next section, we want to exploit this idea to its full potential and apply it to more complex rewriting systems.

4 TERM REWRITING NETS

In this section we consider ARSs and TRSs in their full generality. Two major extensions will be needed. We will need to arrange special training functions that would allow to replace a redex by contractum when they have different sizes. This first problem arises because in neural networks, we use vectors instead of strings. Secondly, we must enrich the learning rule in such a way that several rewriting steps, possibly arising from several rewriting rules, can be applied in parallel.

Suppose we have a string s from Example 3 and a rewriting rule $\rho_2 : [1\ 2] \rightarrow [4\ 5\ 6]$. Following the method described in the previous section, we can build a network with weight $w = v_s$. The training function will automatically attempt to compute $w + \Delta w$, this is possible only if the error vector Δw is of the same length as w; otherwise the vector addition is not defined. But clearly, the rewriting rule ρ_2 will produce Δw that is longer than w. To bring w into appropriate form, we introduce `completion`.

Definition 8. (Completion Algorithm). *Let s be a given string, and* v_s *be the corresponding vector. Let* $\rho_1 : [a] \rightarrow [b]$ *be a given rewriting step, such that a and b are vectors of length* l_a *and* l_b*, and* $l_b > l_a$*. Then* complete v_s *as follows. Compute* $l = l_b - l_a$*, and form a zero vector* v_Z *of length l. Find occurrences of the subvector a in* v_s*. Concatenate* v_Z *with each such subvector in* v_s*.* Completion *outputs the vector* v'_s *that contains* v_Z *after each occurrence of a, but otherwise contains all the elements of* v_s *in their given order.*

In the library of functions we present (Komendantskaya, 2009b), this function is called `completion_r`. Completion can easily be embedded into definitions of the term-rewriting networks.

We now generalise `rewrite` from Definition 7 by adding `completion` to it.

Definition 9. (Generalised Rewrite). *Let s be a given string, and* v_s *be the corresponding vector. Let* $\rho_1 : [a] \rightarrow [b]$ *be a given rewriting rule, such that a and b are vectors of arbitrary length* l_a *and* l_b*. Let* v'_s *be* completed v_s.

Form a zero vector Z of length $l_{v'_s}$.

Compute $\rho' = -([a] - [b])$*, if* $l_a = l_b$*; otherwise concatenate the shortest of them with the vector of zeros of the length* $|l_a - l_b|$*, and compute* $\rho' = -([a] - [b])$ *of length m.*

For every $n = \{1, \ldots, l_{v'_s}\}$*, do the following: If* $n, n+1, \ldots (n+l_a-1)$*th elements of the vector* v'_s *are equal to a, put* ρ' *on the* $n, n+1, \ldots (n+l_a-1)$*th place of Z. The resulting vector is the* change *vector* Δw.

Generalised `rewrite` outputs the change vector for v'_s, its implementation in MATLAB Neural Network toolbox can be found in (Komendantskaya, 2009b). As in the previous section, the *reduced* (or *rewritten*) term can be found by computing: $v_s^{new} = v_s + \Delta w$. This agrees with the training mechanism used in neural networks and we use `rewrite` to generalise Lemma 1:

Lemma 2. *Given an ARS* $\mathcal{A} = (A, \{\rightarrow_1\})$*, and a sequence* s_A *of elements of A, there exists a term-rewriting neural network (TRSNN) that performs the parallel rewriting step for* s_A *in* $\mathcal{A}$.

Proof. The architecture of such a network is given in Definition 6, and the learning rule (`learn_trs` in the MATLAB library) implements $L =$ `rewrite` from Definition 9, see (Komendantskaya, 2009b). □

So far, we have considered only rewriting on numbers. If we wish to apply the TRNN to terms, we would need some numerical vector representation of the first-order syntax. We simply take the standard ASCII encoding provided by MATLAB and command `double`. In general, any one-to-one encoding will be as good.

Example 5. *We take the atomic rewriting step* ρ^σ *from Example 1. We train the TRNN constructed in Lemma 2 to rewrite the term* $F(z, G(F(G(0), G(x))))$*. For this, we take numerical vector encoding v of*

$F(z,G(F(G(0),G(x))))$. *The weight vector is set to v. We get the learning function* `learn_trs` *to implement the generalised* `rewrite`. *On the next iteration, the network outputs the answer* $F(z,G(F(0,0)))$*; see* `experiments_TRS.mat` *in (Komendantskaya, 2009b).*

The last extension we wish to introduce here concerns the number of rewriting rules. So far, we considered only cases with one rewriting rule. However, there can be several disjoint redexes to which different rewriting steps are applied. Clearly, composition of rewriting steps does not convey this idea, (Terese, 2003). To implement the parallel term rewriting for several rules, we need to customise the functions `completion_r` and `rewrite`. Thus, they need to have as many arguments as desired - depending on the number of different and disjoint rewriting steps. For example, `rewrite` was defined to have three arguments v - the vector we rewrite, and $r1, r2$, if the rewriting rule is $\rho_1 : r1 \to r2$. In case of two rewriting rules, we will additionally have arguments $r3$ and $r4$ - for the rule $\rho_2 : r3 \to r4$.

Similarly to the TRSNN that process TRSs with one rewriting rule, `completion` and `rewrite` will be applied hand-in-hand. We assume now that we already have the generalised completion defined for several rewriting rules, see (Komendantskaya, 2009b). We define the generalised `rewrite` for several rewriting rules (`rewrite_mult` in MATLAB).

Definition 10. ***(`Rewrite` for Several Rewriting Rules.)*** *Let s be the given string, and v_s be the corresponding vector. Let $\rho_1 : [a_1] \to [b_1]$, ..., $\rho_n : [a_n] \to [b_n]$ be disjoint atomic rewriting steps, such that each a_i and b_i are vectors of arbitrary length l_{a_i} and l_{b_i}. Let v'_s be the* completed *v_s.*

Form a zero vector Z of length $l_{v'_s}$.

For every $i \in \{1,\dots,n\}$, do the following: compute $\rho'_i = -([a_i]-[b_i])$, if $l_{a_i} = l_{b_i}$; otherwise concatenate the shortest of them with the vector of zeros of the length $|l_a - l_b|$, and then compute $\rho'_i = -([a_i]-[b_i])$ of length $l_{\rho'_i}$.

For every $i \in \{1,\dots,n\}$, find the occurrences of the first element of the vector a_i in v'_s, and form the vector v_i of indexes of the occurrences. Concatenate all such v_i in one vector v_n, and sort its elements in ascending order. For all $k \in v_n$, for all $i \in \{1,\dots,n\}$, do the following. If $k, k+1, \dots, (k+l_{\rho'_i}-1)$th elements of the vector v'_s are equal to a_i, put ρ'_i on the $k, k+1, \dots (k+l_{\rho'_i}-1)$th place of Z.

Rewrite_mult outputs the difference vector Δw for $w = v'_s$, if v'_s is taken to be the weight vector of a network. And we come to the main theorem of the paper.

Theorem 3. *Given an arbitrary ARS $\mathcal{A}$ (or an arbitrary TRS $\mathcal{R}$), and a string s of elements of A (or any term t of $\mathcal{R}$), there exists a neural network that performs a parallel rewriting step for s according to the rewriting rules of $\mathcal{A}$ (or $\mathcal{R}$).*

Proof. The architecture of such a network is given by Definition 6, the training function is conventional (`trainbuwb`), the learning rule (`learn_mult`) implements `rewrite_mult` from Definition 10. The initial weight of the network is equal to the vector v'_s (respectively, v'_t), where v'_s and v'_t are *completed* vectors obtained by applying the function `completion_mult` to v_s and v_t, respectively. See (Komendantskaya, 2009b) for a ready-to-use library. □

Note that the network described in this paper is built in a very generic way, and in practice, we only have to define such a network once (as we did in Figure 1), for one string or term. For other terms or strings of different length, one would simply need to re-define the length of the layer, given by MATLAB command `net.layers{1}.size`, the new value of the weight **w**, given by command `net.iw{1,1}`, and plug in the given rewriting rules into the learning function. This can be easily automatised.

Example 6. *We return to Example 2. Suppose we have chosen the substitution $\sigma = [x := c]$, and need to perform a parallel rewriting step for $G(F(a,c),b)$ using ρ_1 and ρ_2. We again take the template definition of a neural network* `net` *from Example 3. We customize it by computing the numerical vector v for $G(F(a,b),b)$, and taking l_v be the length of the network's only layer. The learning function* `learn_mult` *implements* `rewrite_mult`. *The network outputs $G(G(c,c),F(b,b))$ - the result of performing parallel rewriting step for $G(F(a,c),b)$, ρ_1^σ, and ρ_2. See also the file* `experiments_TRS.mat` *in (Komendantskaya, 2009b) for the MATLAB implementation of it.*

In order to perform a *sequence of parallel rewriting steps*, one needs to iterate the unsupervised training of the given network: n parallel rewriting steps will be performed in n time steps. Additionally, we will need to embed the function `completion_mult` into the training function, such that at each iteration of learning, the network could amend the number of neurons and the weights.

When embedded into the training function, the `complete_mult` will give an effect of a *growing neural gas* (Fritzke, 1994), that is, the network may grow at each training step. The growth will always be bound by the length of the contracta appearing in the rewriting rules, and the contracta are always finite, and often not too big.

5 CONCLUSIONS

We have shown that unsupervised learning used in Neurocomputing implements naturally the parallel rewriting, both for ARSs and TRSs. For a simple and limited class of rewriting systems, where only one rewriting rule is allowed, and its redex and contractum are of the same length, the abstract rewriting is described naturally by a simple form of unsupervised learning. For ARSs and TRSs in their full generality, we have constructed neural networks that perform parallel rewriting steps with the help of *completion* algorithm embedded into the learning rule.

The neural networks defined here are fully formalised in the MATLAB Neural Network Toolbox, and the library of functions is available in (Komendantskaya, 2009b). The implementation brings computational optimisation to the theory of TRS, in that it achieves true parallelism, as opposed to the classical view on parallel term rewriting as a "sequence of disjoint reductions". Since term-rewriting plays a central role in typed theories and functional programming, this implementation may prove to be an important step on integration of the computational logic with learning techniques of neurocomputing; see also (Komendantskaya, 2009a).

The arguable part of the presented work is whether the new (unconventional) learning functions we defined are admissible in neural networks. There can be two responses to this criticism. The first and general response (see also (Komendantskaya, 2008)) is that the devision between unconventional ("symbolic") and conventional ("arithmetic", "statistical") functions is arguable, as there is no formal criteria that separates the two. Depending on a programming language we use, arithmetic functions can be represented symbolically (Komendantskaya, 2008), or, as we did here, symbolic functions can be represented numerically. Another, more concrete and practical response, is that the clear advantage of the networks we presented here is the ease of implementation in hybrid systems: one and the same network can easily switch between the conventional and "symbolic" learning functions, without any structural or other transformations.

ACKNOWLEDGEMENTS

The work was sponsored by EPSRC PF research grant EP/F044046/1. I thank Roy Dyckhoff for useful discussions. Finally, I thank the authors and presenters of EIDMA/DIAMANT minicourse *Lambda Calculus and Term Rewriting Systems* Henk Barendregt and Jan Willem Klop for inspiration.

REFERENCES

Aleksander, I. and Morton, H. (1993). *Neurons and Symbols*. Chapman and Hall.

Bader, S., Hitzler, P., and Hölldobler, S. (2008). Connectionist model generation: A first-order approach. *Neurocomputing*, 71:2420–2432.

Fritzke, B. (1994). Fast learning with incremental rbf networks. *Neural Processing Letters*, 1:1–5.

Gärtner, T. (2003). A survey of kernels for structured data. *SIGKDD Explorations*, 5(1):49–58.

Haykin, S. (1994). *Neural Networks. A Comprehensive Foundation*. Macmillan College Publishing Company.

Hecht-Nielsen, R. (1990). *Neurocomputing*. Addison-Wesley.

Komendantskaya, E. (2008). Unification by error-correction. In *Proceedings of NeSy'08 workshop at ECAI'08, 21-25 July 2008, Patras, Greece*, volume 366. CEUR Workshop Proceedings.

Komendantskaya, E. (2009a). Neurons or symbols: why does or remain exclusive? In *Proceedings of ICNC'09*.

Komendantskaya, E. (2009b). Term rewriting in neural networks: Library of functions and examples written in MATLAB neural network toolbox. www.cs.st-andrews.ac.uk/˜ek/Term-Rewriting.zip.

Smolensky, P. and Legendre, G. (2006). *The Harmonic Mind*. MIT Press.

Strickert, M., Hammer, B., and Blohm, S. (2005). Unsupervised recursive sequence processing. *Neurocomputing*, 63:69–97.

Terese (2003). *Term Rewriting Systems*. Cambridge University Press.