First-order deduction in neural networks

Ekaterina Komendantskaya[1]

Department of Mathematics, University College Cork, Cork, Ireland
e.komendantskaya@mars.ucc.ie *

Abstract. We show how the algorithm of SLD-resolution for first-order logic programs can be performed in connectionist neural networks. The most significant properties of the resulting neural networks are their finiteness and ability to learn.
Key words: Logic programs, artificial neural networks, SLD-resolution, connectionism, neuro-symbolic integration

1 Introduction

The field of neuro-symbolic integration is stimulated by the fact that formal theories (as studied in mathematical logic and used in automated reasoning) are commonly recognised as deductive systems which lack such properties of human reasoning as adaptation, learning and self-organisation. On the other hand, neural networks, introduced as a mathematical model of neurons in human brains, claim to possess all of the mentioned abilities, and moreover, they perform parallel computations and hence can compute faster than classical algorithms. As a step towards integration of the two paradigms, there were built connectionist neural networks [7, 8] which can simulate the work of semantic operator T_P for propositional and (function-free) first-order logic programs. Those neural networks, however, were essentially deductive and could not learn or perform any form of self-organisation or adaptation; they could not even make deduction faster or more effective. There were several attempts to bring learning and self-adaptation in these neural networks, see, for example, [1–3, 10] for some further developments.

The other disconcerting property of the connectionist neural networks computing semantic operators is that they depend on ground instances of clauses, and in case of first-order logic programs containing function symbols will require infinitely long layers to compute the least fixed point of T_P. This property does not agree with the very idea of neurocomputing, which advocates another principle of computation: effectiveness of both natural and artificial neural networks depends primary on their architecture, which is finite, but allows very sophisticated and "well-trained" interconnections between neurons.

In this paper we draw our inspiration from the neural networks of [7, 8], but modify them as follows. In Section 3, we build SLD neural networks which

* I thank the Boole Centre for Research in Informatics (BCRI) at University College Cork for substantial support in the preparation of this paper. I am also grateful to the anonymous referees for their useful comments and suggestions.

simulate the work of SLD-resolution, as opposed to computation of semantic operator in [7, 8]. We show that these neural networks have several advantages comparing with neural networks of [7, 8]. First of all, they embed several learning functions, and thus perform different types of supervised and unsupervised learning recognised in neurocomputing. Furthermore, SLD neural networks do not require infinite number of neurons, and are able to perform resolution for any first-order logic program using finite number of units. The two properties of the SLD neural networks - finiteness and ability to learn - bring the neuro-symbolic computations closer to the practically efficient methods of neurocomputing [5], see also [10] for a more detailed analysis of the topic.

2 Background definitions

We fix a first-order language $\mathcal{L}$ consisting of constant symbols $a_1, a_2, \ldots$, variables $x_1, x_2, \ldots$, function symbols of different arities $f_1, f_2, \ldots$, predicate symbols of different arities $Q_1, Q_2, \ldots$, connectives $\neg, \wedge, \vee$ and quantifiers $\forall, \exists$. We follow the conventional definition of a term and a formula.

Formula of the form $\forall \overline{x}(A \vee \neg B_1 \vee \ldots \vee \neg B_n)$, where A is an atom and each B_i is either an atom or a negation of an atom is called a *Horn clause*. A *Logic Program* P is a set of Horn clauses, and it is common to use the notation $A \leftarrow B_1, \ldots, B_n$, assuming that $B_1, \ldots, B_n$ are quantified using $\exists$ and connected using $\wedge$, see [12] for further details. If each B_i is positive, then we call the clause *definite*. The logic program that contains only definite clauses is called a *definite logic program*. In this paper, we work only with definite logic programs.

Example 1. Consider the following logic program P_1, which determines, for each pair of integers x_1 and x_2, whether $\sqrt[x_1]{x_2}$ is defined. Let Q_1 denote the property to be "defined", $(f_1(x_1, x_2))$ denote $\sqrt[x_1]{x_2}$; Q_2, Q_3 and Q_4 denote, respectively, the property of being an even number, nonnegative number and odd number.

$$Q_1(f_1(x_1, x_2)) \leftarrow Q_2(x_1), Q_3(x_2)$$
$$Q_1(f_1(x_1, x_2)) \leftarrow Q_4(x_1).$$

Logic programs are run by the algorithms of unification and SLD-resolution, see [12] for detailed exposition of it. We briefly survey the notions following [11–13]. Some useful background definitions and the unification algorithm are summarised in the following table:

Let S be a finite set of atoms. A substitution θ is called a unifier for S if S is a singleton. A unifier θ for S is called a *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$. To find the *disagreement set* D_S of S locate the leftmost symbol position at which not all atoms in S have the same symbol and extract from each atom in S the term beginning at that symbol position. The set of all such terms is the disagreement set.
Unification algorithm:

1. Put $k = 0$ and $\sigma_0 = \varepsilon$.
2. If $S\sigma_k$ is a singleton, then stop; σ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\sigma_k$.
3. If there exist a variable v and a term t in D_k such that v does not occur in t, then put $\theta_{k+1} = \theta_k\{v/t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.

The *Unification Theorem* establishes that, for any finite S, if S is unifiable, then the unification algorithm terminates and gives an mgu for S. If S is not unifiable, then the unification algorithm terminates and reports this fact.

The background notions needed to define SLD-resolution are summarised in the following table:

Let a goal G be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and a clause C be $A \leftarrow B_1, \ldots, B_q$. Then G' is derived from G and C using mgu θ if the following conditions hold:

- A_m is an atom, called the *selected* atom, in G.
- θ is an mgu of A_m and A.
- G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

An *SLD-derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\Box$ as the last goal of derivation. If $G_n = \Box$, we say that refutation has length n. The *success set* of P is the set of all $A \in B_P$ such $P \cup \{\leftarrow A\}$ has an SLD-refutation.
If $\theta_1, \ldots, \theta_n$ is the sequence of mgus used in SLD-refutation of $P \cup \{G\}$, then a *computed answer* θ for $P \cup \{G\}$ is obtained by restricting $\theta_1, \ldots, \theta_n$ to variables of G. We say that θ is a *correct answer* for $P \cup \{G\}$ if $\forall((G)\theta)$ is a logical consequence of P.

The SLD-resolution is *sound and complete.* We illustrate the work of SLD-resolution by means of our running example as follows.

Example 2. Consider the logic program P_1 from Example 1. To keep computations simple, we chose a ground goal $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$, where $a_1 = 2$ and $a_2 = 3$, and add $Q_2(a_1) \leftarrow$ and $Q_3(a_2) \leftarrow$ to the database. Now the process of SLD-refutation will proceed as follows:

1. $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$ is unifiable with $Q_1(f_1(x_1, x_2))$, and the algorithm of unification can be applied as follows:
   Form a set $S = \{Q_1(f_1(a_1, a_2)), Q_1(f_1(x_1, x_2))\}$. Form the disagreement set $D_S = \{x_1, a_1\}$. Put $\theta_1 = x_1/a_1$. Now $S\theta_1 = \{Q_1(f_1(a_1, a_2)), Q_1(f_1(a_1, x_2))\}$. Find the new disagreement set $D_{S\theta} = \{x_2, a_2\}$ and put $\theta_2 = x_2/a_2$. Now $S\theta_1\theta_2$ is a singleton, and a new goal can be formed.
2. Form the next goal $G_1 = \leftarrow (Q_2(x_1), Q_3(x_2))\theta_1\theta_2 = \leftarrow Q_2(a_1), Q_3(a_2)$. $Q_2(a_1)$ can be unified with the clause $Q_2(a_1) \leftarrow$ and no substitutions are needed.
3. Form the goal $G_2 = \leftarrow Q_3(a_2)$, and it is unifiable with the clause $Q_3(a_2) \leftarrow$.
4. Form the goal $G_3 = \Box$.

There is a refutation of $P_1 \cup G_0$, the answer is the substitution $\theta_1\theta_2$.

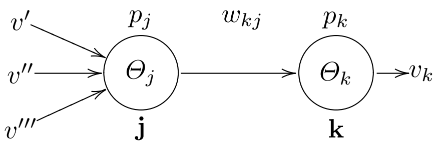
Connectionist Neural Networks.

We follow the definitions of a connectionist neural network given in [7, 8], see also [1] and [6] for further developments of the connectionist neural networks.

A *connectionist 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 *threshold* Θ_k, and its *value* $v_k(t)$. Note that in general, all v_i, p_i and Θ_i, as well as all other parameters of a neural network can be performed by different types of data, the most common of which are real numbers, rational numbers [7, 8], fuzzy (real) numbers, complex numbers, numbers with floating point, and some others, see [5] for more details. We will use Gödel (integer) numbers to build SLD neural networks in Section 3.

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 $v_k(t) = w_{kj}v_j(t)$ is the *input* received by k from j at time t. The units are updated synchronously. In each update, the potential and value of a unit are computed with respect to an *activation* and an *output function* respectively. Most units considered in this paper compute their potential as the weighted sum of their inputs minus their threshold: $p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj}v_j(t)\right) - \Theta_k$. 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 *output function* F, that is, $v_k(t+\Delta t) = F(p_k(t))$. For example, the output function we most often use in this paper is the binary threshold function H, that is, $v_k(t+\Delta t) = H(p_k(t))$, where $H(p_k(t)) = 1$ if $p_k(t) > 0$ and $H(p_k(t)) = 0$ otherwise. Units of this type are called *binary threshold units.*

Example 3. Consider two units, j and k, having thresholds Θ_j, Θ_k, potentials p_j, p_k and values v_j, v_k. The weight of the connection between units j and k is denoted by w_{kj}. Then the following graph shows a simple neural network consisting of j and k. The neural network receives input signals v', v'', v''' and sends an output signal v_k.



We will mainly consider connectionist networks where the units can be organised in layers. A *layer* is a vector of units. An n-layer *feedforward network* $\mathcal{F}$ consists of the *input* layer, $n-2$ *hidden* layers, and the *output* layer, where $n \geq 2$. Each unit occurring in the i-th layer is connected to each unit occurring in the $(i+1)$-st layer, $1 \leq i < n$.

3 SLD-Resolution in Neural Networks

In this section we adapt techniques used both in connectionism and neurocomputing to simulate SLD-resolution, the major first-order deductive mechanism of logic programming. The resulting neural networks have finite architecture, have learning abilities and can perform parallel computations for certain kinds of program goals. This brings connectionist neural networks of [7, 8] closer to artificial neural networks implemented in neurocomputing, see [5], for example. Furthermore, the fact that classical first-order derivations require the use of learning mechanisms if implemented in neural networks is very interesting on its own right and suggests that first-order deductive theories are in fact capable of acquiring some new knowledge, at least to the extent of how this process is understood in neurocomputing.

In order to perform SLD-resolution in neural networks, we will allow not only binary threshold units in the connectionist neural networks, but also units which may receive and send Gödel numbers as signals. We encode first-order atoms directly in neural networks, and this enables us to perform unification and resolution directly in terms of operations of neural networks.

We will use the fact that the first-order language yields a Gödel enumeration. There are several ways of performing the enumeration, we just fix one as follows.

Each symbol of the first-order language receives a **Gödel number** as follows:

– variables $x_1, x_2, x_3, \ldots$ receive numbers $(01), (011), (0111), \ldots$;
– constants $a_1, a_2, a_3, \ldots$ receive numbers $(21), (211), (2111), \ldots$;
– function symbols $f_1, f_2, f_3, \ldots$ receive numbers $(31), (311), (3111), \ldots$;
– predicate symbols $Q_1, Q_2, Q_3, \ldots$ receive numbers $(41), (411), (4111), \ldots$;
– symbols (,) and , receive numbers 5, 6 and 7 respectively.

It is possible to enumerate connectives and quantifiers, but we will not need them here and so omit further enumeration.

Example 4. The following is the enumeration of atoms from Example 1, the rightmost column contains short abbreviations we use for these numbers in further examples:

Atom	Gödel Number	Label
$Q_1(f_1(x_1, x_2))$	41531501701166	g_1
$Q_2(x_1)$	4115016	g_2
$Q_3(x_2)$	411150116	g_3
$Q_3(a_2)$	411152116	g_4
$Q_2(a_1)$	4115216	g_5
$Q_1(f_1(a_1, a_2))$	41531521721166	g_6

We will reformulate some major notions defined in Section 2 in terms of Gödel numbers. We will define some simple (but useful) operations on Gödel numbers in the meantime.

Disagreement set can be defined as follows. Let g_1, g_2 be Gödel numbers of two arbitrary atoms A_1 and A_2 respectively. Define the set $g_1 \ominus g_2$ as follows. Locate the leftmost symbols $j_{g_1} \in g_1$ and $j_{g_2} \in g_2$ which are not equal. If j_{g_i}, $i \in \{1, 2\}$ is 0, put 0 and all successor symbols $1, \ldots, 1$ into $g_1 \ominus g_2$. If j_{g_i} is 2, put 2 and all successor symbols $1, \ldots, 1$ into $g_1 \ominus g_2$. If j_{g_i} is 3, then extract first two symbols after j_{g_i} and then go on extracting successor symbols until number of occurrences of symbol 6 becomes equal to the number of occurrences of symbol 5, put the number starting with j_{g_i} and ending with the last such 6 in $g_1 \ominus g_2$. It is a straightforward observation that $g_1 \ominus g_2$ is equivalent to the notion of the disagreement set D_S, for $S = \{A_1, A_2\}$ as it is defined in Section 2.

We will also need the operation $\oplus$, **concatenation** of Gödel numbers, defined by $g_1 \oplus g_2 = g_1 8 g_2$.

Let g_1 and g_2 denote Gödel numbers of a variable x_i and a term t respectively. We use the number $g_1 9 g_2$ to describe the substitution $\sigma = \{x/t\}$, and we will call $g_1 9 g_2$ the **Gödel number of substitution** σ. If the substitution is obtained for $g_m \ominus g_n$, we will write $s(g_m \ominus g_n)$.

If g_1 is a Gödel number of some atom A_1, and $s = g_1' 9 g_2' 8 g_1'' 9 g_2'' 8 \ldots 8 g_1''' 9 g'''$ is a concatenation of Gödel numbers of some substitutions $\sigma', \sigma'', \ldots, \sigma'''$, then

$g_1 \odot s$ is defined as follows: whenever g_1 contains a substring $(g_1)^*$ such that $(g_1)^*$ is equivalent to some substring s_i of s such that either s_i contains the first several symbols of s up to the first symbol 9 or s_i is contained between 8 and 9 in s, but does not contain 8 or 9, substitute this substring $(g_1)^*$ by the substring s'_i of symbols which success $s_i 9$ up to the first 8. It easy to see that $g_1 \odot s$ reformulates $(A_1)\sigma_1\sigma_2 \ldots, \sigma_n$ in terms of Gödel numbers. In Neural networks, Gödel numbers can be used as positive or negative signals, and we put $g_1 \odot s$ to be 0 if $s = -g_1$.

Unification algorithm can be restated in terms of Gödel numbers as follows: Let g_1 and g_2 be Gödel numbers of two arbitrary atoms A_1 an A_2.

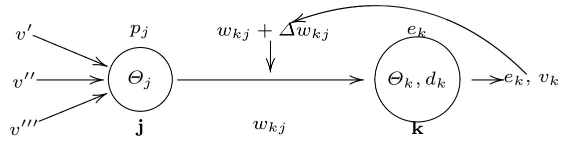
1. Put $k = 0$ and the Gödel number s_0 of substitution σ_0 equal to 0.
2. If $g_1 \odot s_k = g_2 \odot s_k$ then stop; s_k is an mgu of g_1 and g_2. Otherwise, find the disagreement set $(g_1 \odot s_k) \ominus (g_2 \odot s_k)$ of $g_1 \odot s_k$ and $g_2 \odot s_k$.
3. If there exists a number g' starting with 0 and a number g'' in $g_1 \ominus g_2$ such that g' does not occur as a sequence of symbols in g'', then put $s_{k+1} = s_k \oplus g' 9 g''$, increment k and go to 2. Otherwise, stop; g_1 and g_2 are not unifiable.

The algorithm of unification can be simulated in neural networks using the learning technique called *error-correction learning*, see the table below.

[5] Let $d_k(t)$ denote some *desired response* for unit k at time t. Let the corresponding value of the *actual response* be denoted by $v_k(t)$. The response $v_k(t)$ is produced by a *stimulus* (vector) $v_j(t)$ applied to the input of the network in which the unit k is embedded. The input vector $v_k(t)$ and desired response $d_k(t)$ for unit k constitute a particular *example* presented to the network at time t. It is assumed that this example and all other examples presented to the network are generated by an environment. We define an *error signal* as the difference between the desired response $d_k(t)$ and the actual response $v_k(t)$ by $e_k(t) = d_k(t) - v_k(t)$.
The *error-correction learning rule* is the adjustment $\Delta w_{kj}(t)$ made to the weight w_{kj} at time n and is given by

$$\Delta w_{kj}(t) = \eta e_k(t) v_j(t),$$

where η is a positive constant that determines the rate of learning.
Finally, the formula $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$ is used to compute the updated value $w_{kj}(t+1)$ of the weight w_{kj}. We use formulae defining v_k and p_k as in Section 2.
The neural network from Example 3 can be transformed into an error-correction learning neural network as follows. We introduce the *desired response* value d_k into the unit k, and the error signal e_k computed using d_k must be sent to the connection between j and k to adjust w_{kj}:



Lemma 1. *Let k be a neuron with the desired response value $d_k = g_B$, where g_B is the Gödel number of a first-order atom B, and let $v_j = 1$ be a signal sent to k with weight $w_{kj} = g_A$, where g_A is the Gödel number of a first-order atom A. Let h be a unit connected with k. Then there exists an error signal function e_k and an error-correction learning rule Δw_{kj} such that the **unification algorithm for A and B** is performed by **error-correction learning** at unit k, and the unit h outputs the Gödel number of an mgu of A and B if an mgu exists, and it outputs 0 if no mgu of A and B exists.*

Proof. We set $\Theta_k = \Theta_h = 0$, and the weight $w_{hk} = 0$ of the connection between k and h. We use the standard formula to compute $p_k(t) = v_j(t)w_{kj}(t) - \Theta_k$, and put $v_k(t) = p_k(t)$ if $p_k(t) \geq 0$, and $v_k(t) = 0$ otherwise.

The error signal is defined as $e_k(t) = s(d_k(t) \ominus v_j(t))$. That is, $e_k(t)$ computes the disagreement set of g_B and g_A, and $s(d_k(t) \ominus v_j(t))$ computes the Gödel number of substitution for this disagreement set, as described in item 3 of the Unification algorithm. If $d_k(t) \ominus v_j(t) = \emptyset$, set $e_k(t) = 0$. This corresponds to item 1 of the unification algorithm. If $d_k(t) \ominus v_j(t) \neq \emptyset$, but $s(d_k(t) \ominus v_j(t))$ is empty, set $e_k(t) = -w_{kj}(t)$. The latter condition covers the case when g_A and g_B are not unifiable.

The error-correction learning rule is defined to be $\Delta w_{kj}(t) = v_j(t)e_k(t)$. In our case $v_j(t) = 1$, for every t, and so $\Delta w_{kj}(t) = e_k(t)$. We use $\Delta w_{kj}(t)$ to compute

$$w_{kj}(t+1) = w_{kj}(t) \odot \Delta w_{kj}(t) \text{ and } d_k(t+1) = d_k(t) \odot \Delta w_{kj}(t).$$

That is, at each new iteration of this unit, substitutions are performed in accordance with item 2 of the Unification algorithm.

We update the weight of the connection from the unit k to the unit h:
$w_{hk}(t+1) = w_{hk}(t) \oplus \Delta w_{kj}(t)$, if $\Delta w_{kj}(t) > 0$, and $w_{hk}(t+1) = 0$ otherwise. That is, the Gödel numbers of substitutions will be concatenated at each iteration, simulating item 3 of the unification algorithm.

It remains to shows how to read the Gödel number of the resulting mgu. Whenever $e_k(t + \Delta t)$ is 0, compute $p_h(t + \Delta t) = v_k(t + \Delta t) \odot w_{hk}(t + \Delta t)$. If $p_h(t + \Delta t) > 0$, put $v_h(t + \Delta t) = w_{hk}$, and put $v_h(t + \Delta t) = 0$ otherwise. (Note that $p_h(t + \Delta t)$ can be equal to 0 only if $v_k(t + \Delta t) = 0$, and this is possible only if $w_{kj}(t + \Delta t) = 0$. But this, in its turn, is possible only if $\Delta w_{kj}(t + \Delta t - 1) = e_k(t + \Delta t - 1)$ is negative, that is, if some terms appearing in A and B are reported to be non-unifiable according to the Algorithm of Unification.) Thus, if an mgu of A and B exists, it will be computed by $v_h(t + \Delta t)$, and if it does not exists, the unit h will give $v_h(t + \Delta t) = 0$ as an output.

Now we are ready to state and prove the main Theorem of this paper.

Theorem 1. *Let P be a definite logic program and G be a definite goal. Then there exists a 3-layer recurrent neural network which computes the Gödel number s of substitution θ if and only if SLD-refutation derives θ as an answer for $P \cup \{G\}$. (We will call these neural networks* SLD neural networks*).*

Proof. Let P be a logic program and let $C_1, \ldots, C_m$ be definite clauses contained in P.

The SLD neural network consists of three layers, *Kohonen's layer k* (see the table below) of input units $k_1, \ldots, k_m$, layer h of output units $h_1, \ldots, h_m$ and layer o of units $o_1, \ldots, o_n$, where m is number of clauses in the logic program P, and n is the number of all atoms appearing in the bodies of clauses of P.

[9] The general definition of the Kohonen layer is as follows. The Kohonen layer consists of N units, each receiving n input signals $v_1, \ldots, v_n$ from another layer of units. The v_j input signal to Kohonen unit i has a weight w_{ij} assigned to it. We denote by $\mathbf{w_i}$ the vector of weights $(w_{i1}, \ldots, w_{in})$, and we use $\mathbf{v}$ to denote the vector of input signals $(v_1, \ldots, v_n)$.
Each Kohonen unit calculates its *input intensity* I_i in accordance with the following formula: $I_i = D(\mathbf{w_i}, \mathbf{v})$, where $D(\mathbf{w_i}, \mathbf{v})$ is the distance measurement function. The common choice for $D(\mathbf{w_i}, \mathbf{v})$ is the Euclidian distance $D(\mathbf{w_i}, \mathbf{v}) = |w_i - v|$.
Once each Kohonen unit has calculated its input intensity I_i, a competition takes place to see which unit has the smallest input intensity. Once the winning Kohonen unit is determined, its output v_i is set to 1. All the other Kohonen unit output signals are set to 0.

Similar to the connectionist neural networks of [7, 8], each input unit k_i represents the head of some clause C_i in P, and is connected to precisely one unit h_i, which is connected, in its turn, to units $o_k, \ldots, o_s$ representing atoms contained in the body of C_i. This is the main similar feature of SLD neural networks and connectionist neural networks of [7, 8]. Note that in neural networks of [7, 8] o was an output layer, and h was hidden layer, whereas in our setting h will be an output layer and we require the reverse flow of signals comparting with [7, 8].

Thresholds of all the units are set to 0.

The input units $k_1, \ldots, k_m$ will be involved in the process of error-correction learning and this is why each of $k_1, \ldots, k_m$ must be characterised by the value of the *desired response* d_{k_i}, $i \in \{1, \ldots, m\}$, and each d_{k_i} is the Gödel number of the atom A_i which is the head of the clause C_i. Initially all weights between layer k an layer h are set to 0, but an error-correction learning function is introduced in each connection between k_i and h_i, see Lemma 1. The weight from each h_i to some o_j is defined to be the Gödel number of the atom represented by o_j.

Consider a definite goal G that contains atoms $B_1, \ldots, B_n$, and let $g_1, \ldots g_n$ be the Gödel numbers of $B_1, \ldots, B_n$. Then, for each g_l, do the following: at time t send a signal $v_l = 1$ to each unit k_i.

Predicate threshold function will be assumed throughout the proof, and is stated as follows. Set the weight of the connection $w_{k_i,l}(t)$ equal to g_l ($l \in \{1, \ldots, n\}$) if g_l has the string of 1s after 4 of the same length as the string of 1s succeeding 4 in d_{k_i} (there may be several such signals from one g_l, and we denote them by $v_{l1}, \ldots, v_{lm}$). Otherwise, set the weight $w_{k_i l}(t)$ of each connection between l and k_i equal to 0.

Step 1 shows how the input layer k filters excessive signals in order to process, according to SLD-resolution algorithm, only one goal at a time. This step will involve the use of Kohonen competition and Grossberg's laws defined in the table above:

[5] Consider the situation when a unit receives multiple input signals, $v_1, v_2, \ldots, v_n$, with v_n distinguished signal. In Grossberg's original neurobiological model [4], the v_i, $i \neq n$, were thought of as "conditioned stimuli" and the signal v_n was an "unconditioned stimulus". Grossberg assumed that v_i, $i \neq n$ was 0 most of the time and took large positive value when it became active.
Choose some unit c with incoming signals $v_1, v_2, \ldots, v_n$. *Grossberg's law* is expressed by the equation

$$w_{ci}^{\text{new}} = w_{ci}^{\text{old}} + a[v_i v_n - w_{ci}^{\text{old}}]U(v_i), (i \in \{1, \ldots, n-1\}),$$

where $0 \leq a \leq 1$ and where $U(v_i) = 1$ if $v_i > 0$ and $U(v_i) = 0$ otherwise.
We will also use *the inverse form of Grossberg's law* and apply the equation

$$w_{ic}^{\text{new}} = w_{ic}(t)^{\text{old}} + a[v_i v_n - w_{ic}^{\text{old}}]U(v_i), (i \in \{1, \ldots, n-1\})$$

to enable (unsupervised) change of weights of connections going from some unit c which sends outcoming signals $v_1, v_2, \ldots v_n$ to units $1, \ldots, n$ respectively. This will enable outcoming signals of one unit to compete with each other.

Suppose several input signals $v_{l1}(t), \ldots, v_{lm}(t)$ were sent from one source to unit k_i. At time t, only one of $v_{l1}(t), \ldots, v_{lm}(t)$ can be activated, and we apply the *inverse Grossberg's law* to filter the signals $v_{l1}(t), \ldots, v_{lm}(t)$ as follows. Fix the *unconditioned signal* $v_{l1}(t)$ and compute, for each $j \in \{2, \ldots, m\}$, $w_{k_i l_j}^{\text{new}}(t) = w_{k_i l_j}^{\text{old}}(t) + [v_{l1}(t)v_{l_j}(t) - w_{k_i l_j}^{\text{old}}(t)]U(v_{l_j})$. We will also refer to this function as $\psi_1(w_{k_i l_j}(t))$. This filter will set all the weights $w_{k_i l_j}(t)$, where $j \in \{2, \ldots, m\}$ to 1, and the *Predicate threshold* will ensure that those weights will be inactive.

The use of the inverse Grossberg's law here reflects the logic programming convention that each goal atom unifies only with one clause at a time. Yet several goal atoms may be unifiable with one and the same clause, and we use Grossberg's law *to filter signals of this type as follows.*

If an input unit k_i receives several signals $v_j(t), \ldots, v_r(t)$ from different sources, then fix an *unconditioned signal* $v_j(t)$ and apply, for all $m \in \{(j+1), \ldots, r\}$ the equation $w^{\mathrm{new}}_{k_i m}(t) = w^{\mathrm{old}}_{k_i m}(t) + [v_m(t)v_j(t) - w^{\mathrm{old}}_{k_i m}(t)]U(v_m)$ at time t, we will refer to this function as $\psi_2(w_{k_i m}(t))$. The function ψ_2 will have the same effect as ψ_1: all the signals except $v_j(t)$ will have to pass through connections with weights 1, and the *Predicate threshold* will make them inactive at time t.

Functions ψ_1 and ψ_2 will guarantee that each input unit processes only one signal at a time. At this stage we could start further computations independently at each input unit, but the algorithm of SLD-refutation treats each non-ground atom in a goal as dependent on others via variable substitutions, that is, if one goal atom unifies with some clause, the other goal atoms will be subjects to the same substitutions. This is why we must avoid independent, parallel computations in the input layer and we apply the principles of competitive learning as they are realized in Kohonen's layer*:*

At time $t+1$, compute $I_{k_i}(t+1) = D(\mathbf{w_{k_i j}}, \mathbf{v_j})$, for each k_i. The unit with the least $I_{k_i}(t+1)$ will proceed with computations of $p_{k_i}(t+1)$ and $v_{k_i}(t+1)$, all the other units $k_j \neq k_i$ will automatically receive the value $v_{k_j}(t+1) = 0$. Note that if neither of $w_{k_i j}(t+1)$ contains symbol 0 (all goal atoms are ground), we don't have to apply Kohonen's competition and can proceed with parallel computations for each input unit.

Now, given an input signal $v_j(t+1)$, the potential $p_{k_i}(t+1)$ will be computed using the standard formula: $p_{k_i}(t+1) = v_j(t+1)w_{k_i j} - \Theta_k$, where, as we defined before, $v_j(t+1) = 1, w_{k_i j} = g_j$ and $\Theta_k = 0$. The output signal from k_i is computed as follows: $v_{k_i}(t+1) = p_{k_i}(t+1)$, if $p_{k_i}(t+1) > 0$, and $v_{k_i}(t+1) = 0$ otherwise.

At this stage the input unit k_i is ready to propagate the signal $v_{k_i}(t+1)$ further. However, the signal $v_{k_i}(t+1)$ may be different from the desired response $d_{k_i}(t+1)$, and the network initialises the *error-correction learning* in order to bring the signal $v_{k_i}(t+1)$ in correspondence with the desired response and compute the Gödel number of an mgu. We use here Lemma 1, and conclude that at some time $(t+\Delta t)$ the signal $v_{h_i}(t+\Delta t)$ (the Gödel number of substitutions) is sent both as the input signal to the layer o and as an output signal of the network which can be read by external recipient.

The next two paragraphs describe amendments to the neural networks to be done in cases when either mgu was obatined, or the unification algorithm reported that no mgu exists.

If $e_{k_i}(t+\Delta t) = 0$ ($\Delta t \geq 1$), set $w_{kj}(t+\Delta t+2) = 0$, where j is the impulse previously trained via error-correction algorithm; change input weights leading from all other sources r, $r \neq j$, using $w_{k_n r}(t+\Delta t+2) = w_{k_n r}(t) \odot w_{h_i k_i}(t+\Delta t)$.

Whenever at time $t+\Delta t$ ($\Delta t \geq 1$), $e_{k_i}(t+\Delta t) \leq 0$, set the weight $w_{h_i k_i}(t+\Delta t+2) = 0$. Furthermore, if $e_{k_i}(t+\Delta t) = 0$, initialise at time $t+\Delta t+2$ new activation of *Grossberg's function* ψ_2 (for some fixed $v_m \neq v_j$); if $e_{k_i}(t+\Delta t) < 0$, initialise at time $t+\Delta t+2$ new activation of *inverse Grossberg's function* ϕ_1 (for some $v_{l_i} \neq v_{l_1}$). In both cases initialise Kohonen's layer competition at time $t+\Delta t+3$.

Step 2. As we defined already, h_i is connected to some units $o_l, \ldots, o_r$ in the layer o with weights $w_{o_l h_i} = g_{o_l}, \ldots, w_{o_r h_i} = g_{o_r}$. And v_{h_i} is sent to each $o_l, \ldots, o_r$ at time $t+\Delta t+1$. The network will now compute, for each o_l, $p_{o_l}(t+\Delta t+1) = w_{o_l h_i} \odot v_{h_i} - \Theta_{o_l}$, with $\Theta_{o_l} = 0$. Put $v_{o_l}(t+\Delta t+1) = 1$ if $p_{o_l}(t+\Delta t+1) > 0$ and $v_{o_l}(t+\Delta t+1) = 0$ otherwise.

At step 2 the network applies obtained substitutions to the atoms in the body of the clause whose head has been unified already.

Step 3. At time $t+\Delta t+2$, $v_{o_l}(t+\Delta t+1)$ is sent to the layer k. Note that all weights $w_{k_j o_l}(t+\Delta t+2)$ were defined to be 0, and we introduce the learning function $\vartheta = \Delta w_{k_j o_l}(t+\Delta t+1) = p_{o_l}(t+\Delta t+1)v_{o_l}(t+\Delta t+1)$, which can be seen as a kind of Hebbian function, see [5]. At time $t + \Delta t + 2$ the network computes $w_{k_j o_l}(t + \Delta t + 2) = w_{k_j o_l}(t + \Delta t + 1) + \Delta w_{k_j o_l}(t + \Delta t + 1)$.

At step 3, the new goals, which are the Gödel numbers of the body atoms (with applied substitutions) are formed and sent to the input layer.

Once the signals $v_{o_l}(t + \Delta t + 2)$ are sent as input signals to the input layer k, the *Grossberg's functions* will be activated at time $(t + \Delta t + 2)$, *Kohonen competition* will take place at time $(t + \Delta t + 3)$ as described in Step 1 and thus the new iteration will start.

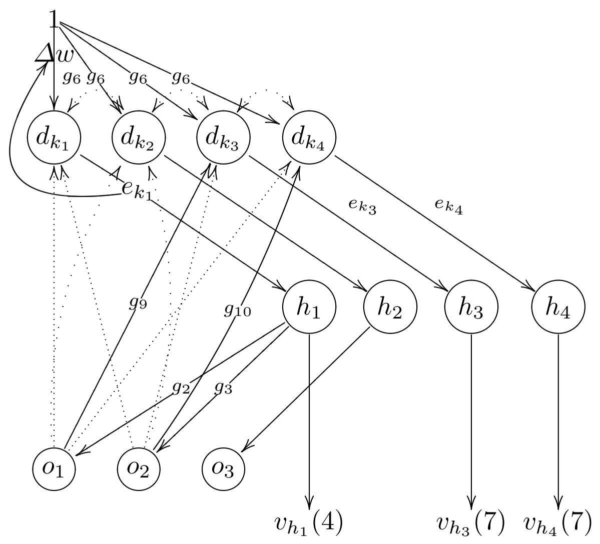
Computing and reading the answer. The signals v_{h_i} are read from the hidden layer h, and as can be seen, are Gödel numbers of relevant substitutions. We say that an SLD neural network *computed an answer for* $P \cup \{G\}$, if and only if, for each external source i and internal source o_s of input signals $v_{i_1}(t), v_{i_2}(t), \dots, v_{i_n}(t)$ (respectively $v_{o_{s1}}(t), v_{o_{s2}}(t), \dots, v_{o_{sn}}(t)$), the following holds: for at least one input signal $v_{i_l}(t)$ (or $v_{o_{sl}}(t)$) sent from the source i (respectively o_s), there exists $v_{h_j}(t + \Delta t)$, such that $v_{h_j}(t + \Delta t)$ is a string of length $l \geq 2$ whose first and last symbol is 0. If, for all $v_{i_l}(t)$ ($v_{o_{sl}}(t)$ respectively), $v_{h_j}(t + \Delta t) = 0$ we say that the computation failed.

Backtracking is one of the major techniques in SLD-resolution. We formulate it in the SLD neural networks as follows. Whenever $v_{h_j}(t + \Delta t) = 0$, do the following.

1. Find the corresponding unit k_j and $w_{k_j o_l}$, apply the inverse Grossberg's function ψ_1 to some v_{o_s}, such that v_{o_s} has not been an *unconditioned signal* before.
2. If there is no such v_{o_s}, find unit h_f connected to o_s and go to item 1.

The rest of the proof proceeds by routine induction.

Example 5. Consider the logic program P_1 from Example 2 and SLD neural networks for it. Gödel numbers $g_1, \dots, g_6$ are taken from Example 4. Input layer k consists of units k_1, k_2, k_3 and k_4, representing heads of four clauses in P_1, each with the *desired response value* $d_{k_i} = g_i$. The layers o consists of units o_1, o_2 and o_3, representing three body atoms contained in P_1. Then the steps of computation of the answer for the goal $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$ from Example 2 can be performed by the following Neural network:



The answer is: $v_{h_1}(4) = 0019218011921180$, $v_{h_3}(7) = 080192180$, $v_{h_4}(7) = 08011921180$, natural numbers in brackets denote time at which the signal was emitted. It is easy to see that the output signals correspond to Gödel numbers of substitutions obtained as an answer for $P_1 \cup G_0$ in Example 2.

Note that if we built a connectionist neural network of [7, 8] which corresponds to the logic program P_1 from Examples 1-2, we would need to built a neural networks with infinitely many units in all the three layers. And, since such networks cannot be built in the real world, we would finally need to use some approximation theorem which is, in general, non-constructive.

4 Conclusions and Further Work

Several conclusions can be made from Lemma 1 and Theorem 1.

SLD neural networks have finite architecture, but their effectiveness is due to several learning functions: two Grossberg's filter learning functions, error-correction learning functions, *Predicate threshold* function, Kohonen's competitive learning, Hebbian learning function ϑ. The most important of those functions are those providing supervised learning and simulating the work of algorithm of unification.

Learning laws implemented in SLD neural network exhibit a "creative" component in SLD-resolution algorithm. Indeed, the search for successful unification, choice of goal atoms and program clause at each step of derivation are not fully determined by the algorithm, but leave us (or program interpreter) to make personal choice, and in this sense, allow certain "creativity" in the decisions. The fact that process of unification is simulated by means of error-correction learning algorithm reflects the fact that the unification algorithm is, in essence, a correction of one peace of data relatively to the other piece of data. This also suggests that unification is not totally deductive algorithm, but an adaptive process.

Atoms and substitutions of the first-order language are represented in SLD neural networks internally via Gödel numbers of weights and other parameters. This distinguishes SLD neural networks from the connectionist neural networks of [7, 8], where symbols appearing in a logic program were not encoded in the corresponding neural network directly, but each unit was just "thought of" as representing some atom. This suggests that SLD neural networks allow easier machine implementations comparing with the neural networks of [7, 8].

The SLD neural networks can realize either depth-first or breadth-first search algorithms implemented in SLD-resolution, and this can be fixed by imposing some conditions on the choice of unconditioned stimulus during the use of Grossberg's law in layer k.

The future work may include both practical implementation of SLD neural networks, and their further theoretical development. For example, SLD neural networks we have presented here, unlike the neural networks of [7, 8], allow almost straightforward generalisations to higher-order logic programs. Further extension of these neural networks to higher-order Horn logics, hereditary Harrop logics, linear logic programs, etc. may lead to other novel and interesting results.

References

1. A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications.* Springer-Verlag, 2002.
2. A. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied intelligence, Special Issue on Neural networks and Structured Knowledge*, 11(1):59–77, 1999.
3. A. d'Avila Garcez, G. Zaverucha, and L. A. de Carvalho. Logical inference and inductive learning in artificial neural networks. In C.Hermann, F.Reine, and A.Strohmaier, editors, *Knowledge Representation in Neural Networks*, pages 33–46. Logos Verlag, Berlin, 1997.
4. S. Grossberg. Embedding fields: A theory of learning with physiological implications. *J. Math. Psych.*, 6:209–239, 1969.
5. R. Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, 1990.
6. P. Hitzler, S. Hölldobler, and A. K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.
7. S. Hölldobler and Y. Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.
8. S. Hölldobler, Y. Kalinke, and H. P. Storr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.
9. T. Kohonen. *Self-Organization and Associative memory.* Springer-Verlag, Berlin, second edition edition, 1988.
10. E. Komendantskaya. Learning and deduction in neural networks and logic, 2006. Submitted to the Special Issue of TCS, "From Gödel to Einstein: computability between logic and physics".
11. R. A. Kowalski. Predicate logic as a programming language. In *Information Processing 74*, pages 569–574, Stockholm, North Holland, 1974.
12. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
13. J. A. Robinson. A machine-oriented logic based on resolution principle. *Journal of ACM*, 12(1):23–41, 1965.