Unification by Error-Correction

Ekaterina Komendantskaya [1]

Abstract. The paper formalises the famous algorithm of first-order unification by Robinson by means of the error-correction learning in neural networks. The significant achievement of this formalisation is that, for the first time, the first-order unification of two arbitrary first-order atoms is performed by finite (two-neuron) network.

1 Introduction

The field of neuro-symbolic integration is stimulated by the fact that logic theories are commonly recognised as deductive systems that 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 brain, possess all of the mentioned abilities, and moreover, they provide parallel computations and hence can perform certain calculations faster than classical algorithms.

As a step towards integration of the two paradigms, there were built connectionist neural networks (also called neuro-symbolic networks), see [1, 8] for a very good survey. In particular, there were built neural networks [16, 17] that can simulate the work of the semantic operator T_P for logic programs. These neural networks processed classical truth values 0 and 1 assigned to clauses and clause atoms. These values were presented to the neural networks as input vectors and emitted by the neural networks as output vectors.

The connectionist neural networks of different architectures [8, 15, 7, 1] bore different advantages, but one similar feature: the unification of first-order terms, atoms, or clauses was achieved by building a separate neuron for each of the ground instances of an atom. Then all neurons were connected in a particular way that they reflected intended logical relations between the ground atoms. In many cases, e.g., in the presence of function symbols in logic programs, the number of the required ground instances can become infinite. This makes building corresponding neural networks impractical. The problem gave rise to a series of papers about possibility of approximation of potentially infinite computations by (a family of) finite neural networks; see [13, 3, 25, 2].

In this paper, I propose a different direction for the development of the connectionist neural networks. In particular, I propose to use two-neuron networks with error-correction learning to perform the first-order unification over two arbitrary first-order atoms. A simple form of error-correction learning is adapted to syntax of a first-order language in such a way that unification of two atoms is seen as a correction of one piece of data relative to the other piece of data.

The problem of establishing a way of how to perform first-order unification in finite neural networks has been tackled by many researchers over the last 30 years; [4, 21, 15, 14, 8, 26, 27, 1, 28].

[1] INRIA Sophia Antipolis, France, email: ekaterina.komendantskaya@inria.fr

The way of performing unification that I propose here is novel, fast and simple, and can be easily integrated into a number of various existing neuro-symbolic networks. The paper develops ideas which were first spelt out in [18, 19]. Here, I simplify the construction of the networks, generalise the theorem about unification by error correction and give a more subtle analysis of the new functions introduced into the neural networks. Notably, the statement and the proof of the main theorem do not depend anymore on Gödel numbers, as in [18, 19].

The structure of the paper is as follows. Section 2 outlines the classical algorithm of unification. Section 3 defines artificial Neurons and Neural Networks following [12, 13]. Section 4 describes the error-correction learning algorithm. In Section 5, I re-express several logic notions in terms of recursive functions over terms. These functions are then embedded into the network. In Section 6, I prove that the algorithm of Unification for two arbitrary first-order atoms can be simulated by a two-neuron network with error-correction function. Finally Section 7 concludes the discussion.

2 Unification algorithm

The algorithm of unification for first-order atoms was introduced in [24] and has been extensively used in Logic programming [22] and theorem proving.

I 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 $\forall, \exists$. This syntax is sufficient to define first-order language or first-order Horn clause programs, [22].

I follow the conventional definition of a term and an atomic formula. Namely, a constant symbol is a term, a variable is a term, and if f_i^n is a n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f_i^n(t_1, \ldots t_n)$ is a term. If Q_i^n is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $Q_i(t_1, \ldots t_n)$ is an atomic formula, also called an atom.

Let S be a finite set of atoms. A substitution θ is called a unifier for S if $S\theta$ 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.

The unification algorithm [24, 20, 22] is described as follows.

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 $\sigma_{k+1} = \sigma_k\{v/t\}$, 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.

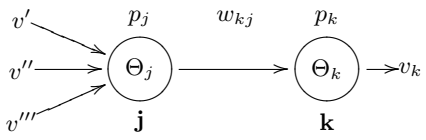
3 Connectionist Neural Networks

I follow the definitions of a connectionist neural network given in [16, 17], see also [7, 13, 9].

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 [16, 17], fuzzy (real) numbers [23], complex numbers, numbers with floating point, Gödel numbers [19], and some others, see also [12].

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. 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 and [16] 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 using $p_k(t)$ by means of a given *output function* F, that is, $v_k(t + \Delta t) = F(p_k(t))$. For example, F can be an identity function $\mathtt{id}$, or 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.

Example 3.1 *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 signals v', v'', v''' from external sources and sends an output signal v_k.*



4 Error-Correction Learning

Error-correction learning is one of the algorithms among the paradigms that advocate *supervised learning*; see [12, 11] for further details.

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 input signal $i_k(t)$ 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. It is common to define an *error signal* as the difference between the desired response $d_k(t)$ and the actual response $v_k(t)$ by $e_k(t) = d_k(t) - v_k(t)$.

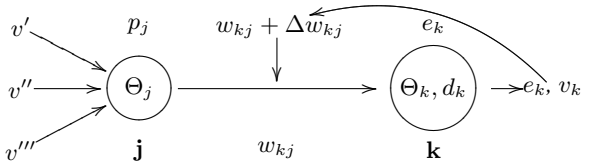
The *error-correction learning rule* is the adjustment $\Delta w_{kj}(t)$ made to the weight w_{kj} at time n and is given by

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

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}. I use formulae defining v_k and p_k as in Section 3.

Example 4.1 *The neural network from Example 3.1 can be transformed into an error-correction learning neural network as follows. I 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}.*



This learning rule has been extensively used for "recognition" tasks, such as image and speech recognition.

5 The data type of parameters of the neural network

In order to perform the algorithm of unification in neural networks and not to depend on truth values of formulae, I need to allow the syntax of first-order formulae directly into the neural network.

Initially, Gödel numbers were used as parameters of the novel neural networks, [19]. It was inspired by the idea that some sort of numerical representation is crucial because Neural networks are numerical machines, and can process only numbers. However, from computational point of view the numerical encoding of the first-order syntax plays no crucial role in the development of the neural networks, and so I omit enumeration here. Instead, I give a more subtle analysis of the new functions that I embed into neural networks.

The significant feature of the Gödel enumeration in [19] was that the notions of the *disagreement set*, *substitution*, and *application of the computed substitutions* were formally expressed as functions over first-order atoms viewed as lists. These functions were embedded into the neural network. The appearance of new functions in the neural network architecture was natural because the neural networks used the new data type - Gödel numbers of atoms.

The algorithm of Unification from Section 2 can be reformulated functionally. Thus, we can define a function $\ominus$ computing a disagreement set of two given atoms, or a function $\odot$ applying substitutions, and some other functions, and then compose them into a more complex function that is able to unify terms. There exist several functional formalisations of the algorithm of unification in different functional languages, [6, 10, 29]. It can be quite hard to formalise this algorithm, as [6, 10, 29] indicate. In this paper, we will use only two simple auxiliary functions $\odot$ and $\ominus$, and will not go into further details.

In the rest of the section, we give an example of how the two functions $\odot$ and $\ominus$ can be formalised using the language of Coq. The reader not familiar with functional languages can pass on to the next Section, simply bearing in mind that $\ominus$ will denote the function computing a disagreement set, and $\odot$ is a function that applies substitutions. The complete Coq file containing more details and examples can be found in [30].

When formalised functionally, the definitions of the *disagreement set*, *substitution*, and *application of the computed substitutions* defined over first-order terms and atoms bear no serious computational difference from formalisations of arithmetic functions $+$, $-$, $*$ defined over integers. In fact, all of the above-mentioned operations, both logical and arithmetic, are defined recursively by computing a fixpoint.

Example 5.1 *This is how conventional $+$ over natural numbers is defined in Coq:*

```
Fixpoint plus (n m: nat){struct n}: nat :=
match n with
O => m | S p => S (plus p m) end.
```

We will see in this section that logical operations can be defined analogously, by fixpoint. I will define functions that find a disagreement set, a substitution, and apply the computed substitutions using the Coq code, the same functions expressed in terms of Gödel numbers can be found in [19, 18].

We need to specify the inductive data type of first-order terms and atoms. We will define recursive functions over this data type:

```
Inductive term : Set :=
App (t1 t2: term) | Const (c: cindex)
   | Var (v: vindex).
```

Example 5.2 *For example, Q(x , y) will be denoted by* `(App (App (Const 0) x) y)`.

I start with the function applying substitutions:

```
Fixpoint subst (v: vindex) (t1 t2: term)
  {struct t2} : term :=   match t2 with
 | App t3 t4 =>
      App (subst v t1 t3) (subst v t1 t4)
 | Const _   => t2
 | Var v1    => if v_eq v v1 then t1 else t2
  end.
```

To define the function computing the disagreement set, one needs to inductively define the type of possible outputs of the function. Unification algorithm of Section 2 could output either "failure", or a computed mgu, or an "empty" substitution ε:

```
Inductive dresult: Set :=
Dempty | Dfail | Dsubst (v: vindex) (t: term).
```

The function computing the disagreement set reformulated in Gödel numbers was called $\ominus$ in [19].
Operation $\ominus$, written $(t_1 \ominus t_2)$, or `delta (t1 t2)`:

```
Fixpoint delta (t1 t2: term)
        {struct t1} : dresult :=
match t1, t2 with
| Var v1, Var v2 =>
  if v_eq v1 v2 then Dempty else Dsubst v1 t2
| Var v1, _ =>
  if v_is_in v1 (free_vars t2) then
      Dfail else Dsubst v1 t2
| _, Var v2 =>
   if v_is_in v2 (free_vars t1) then
      Dfail else Dsubst v2 t1
| Const c1, Const c2 =>
   if c_eq c1 c2 then Dempty else Dfail
| App t11 t12, App t21 t22 =>
   match delta t11 t21 with Dempty =>
     delta t12 t22 | r => r end
  | _, _ => Dfail
end.
```

The function above is built using another function, `free_vars`, that performs the occur check. The definition of this function can be found in [30]; it is a simple recursive function defined by fixpoint.

The function that applies *computed* substitutions was formulated in Gödel numbers and denoted $\odot$ in [19]. We give its Coq code here:
Operation $\odot$, written $(t \odot d)$ or `apply_delta t d`:

```
Definition apply_delta t d :=
 match d with Dsubst v t1 =>
subst v t1 t | _ => t end.
```

The functions $\ominus, \odot$ will be taken as new parameters of a neural network.

6 Unification in Neural Networks

Neural networks constructed in this section perform unification by error-correction.

The next theorem and its proof present this novel construction. The construction is effectively based upon the error-correction learning algorithm defined in Section 4 and makes use of the operations $\odot$ and $\ominus$ defined in Section 5. I will also use $\oplus$ - the conventional list concatenation that, given lists x and y, forms the list $x \oplus y$. The standard Coq formalisation of this function can be found in [5].

Remark 1. The next Theorem requires the last short remark. The item 3 in the Unification algorithm of Section 2 requires composition of computed substitutions at each iteration of the algorithm. That is, given a set S of atoms, the unification algorithm will compute $S(\sigma_0\sigma_1 \dots \sigma_n)$, which means that the composition of substitutions $\sigma_0\sigma_1 \dots \sigma_n$ is applied to S. However, one can show that $S(\sigma_0\sigma_1 \dots \sigma_n) = (\dots((S\sigma_0)\sigma_1) \dots \sigma_n)$. That is, one can alternatively concatenate substitutions and apply them one by one to atoms in S. We will use this fact when constructing the neural networks. The two functions - concatenation $\oplus$ and application $\odot$ of substitutions will be used to model $S(\sigma_0\sigma_1 \dots \sigma_n)$.

Theorem 1 *Let k be a neuron with the desired response value $d_k = g_B$, where g_B is (the encoding of) a first-order atom B, and let $v_j = 1$ be the signal sent to k with weight $w_{kj} = g_A$, where g_A is (the encoding 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 encoding of) an mgu of A and B if an mgu exists, and it outputs 0 if no mgu of A and B exists.*

Construction:
We construct the two neuron network as follows. The unit k is the input unit, the signal from the unit k goes to the unit h, and the unit h outputs the signal.

Parameters:
Set thresholds $\Theta_k = \Theta_h = 0$, and the initial weight $w_{hk}(0) = 0$ of the connection between k and h. The input signal $i_k(t) = v_j(t)w_{kj}(t) = w_{kj}(t)$. The initial input $i_k(0) = w_{kj}(0) = g_A$.

Because $v_j(t) = 1$ for all t, the standard formula that computes potential $p_k(t) = v_j(t)w_{kj}(t) - \Theta_k$ transforms into $p_k(t) = w_{kj}(t)$. We put $v_k(t) = p_k(t)$.

The error signal is defined as follows: $e_k(t) = d_k(t) \ominus v_k(t)$. The initial desired response $d_k(0) = g_B$.

The error-correction learning rule is as defined in Section 4: $\Delta w_{kj}(t) = v_j(t)e_k(t)$. In our case $v_j(t) = 1$, at every time t, and so $\Delta w_{kj}(t) = e_k(t)$. The $\Delta w_{kj}(t)$ is used to compute

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

Connection between k and h is "trained" as follows:

$$w_{hk}(t+1) = \begin{cases} w_{hk}(t) \oplus \Delta w_{kj}(t), & \text{if } \Delta w_{kj}(t) = \texttt{Dsubst v t} \\ w_{hk}(t) \oplus 0, & \text{if } \Delta w_{kj}(t) = \texttt{Dempty} \\ 0, & \text{if } \Delta w_{kj}(t) = \texttt{Dfail}. \end{cases}$$

Reading the resulting mgu.
Compute $p_h(t+\Delta t) = w_{hk}(t+\Delta t)$. Put $v_h(t+\Delta t) = p_h(t+\Delta t)$.
If the signal $v_h(t + \Delta t) \neq 0$ and the first and the last symbol constituting the list $v_h(t + \Delta t)$ is 0, stop. The signal $v_h(t + \Delta t)$ is the mgu of A and B.
If $v_h(t + \Delta t) = 0$, then stop. Unification failed.

Sketch of a proof: *Item 1 of Unification algorithm (Section 2).* The network works in discrete time, and the sequence of time steps, starting with 0, corresponds to the parameter k in the Unification algorithm, that changes from 0 to 1, from 1 to 2, etc. The initial empty substitution $\sigma_0 = \varepsilon$ corresponds to the initial weight $w_{hk}(0) = 0$.

Item 2 of Unification algorithm. The application $S\sigma_0 \dots \sigma_k$ of substitution $\sigma_0 \dots \sigma_k$ to S is performed by the function $\odot$ that, by **Remark 1**, applies σ_k to $(\dots(S\sigma_0)\dots\sigma_{k-1})$. The check whether the disagreement set D_k for $S\sigma_1 \dots \sigma_k$ is empty is done by $\ominus$. If it is empty, $d_k(t) \ominus v_k(t) = \texttt{Dempty}$. If this happens at time t, $e_k(t) = \Delta w_{kj}(t) = \texttt{Dempty}$, and $v_h(t+1) = w_{hk}(t+1) = w_{hk}(t) \oplus \Delta w_{kj}(t) = 0 \oplus e_k(0) \oplus \dots \oplus e_k(t-1) \oplus e_k(t) = 0 \oplus e_k(0) \oplus \dots \oplus e_k(t-1) \oplus 0$ is sent as an output from the unit h. This will be read as "Stop, the mgu is found". In case D_k for $S\sigma_1 \dots \sigma_k$ is not empty, the function $d_k(t) \ominus v_k(t)$ computes the disagreement set for $g_B(\sigma_1 \dots \sigma_k) = d_k(t)$ and $g_A(\sigma_1 \dots \sigma_k) = v_k(t)$ in $S\sigma_1 \dots \sigma_k$; $\Delta w_{kj}(t)$ is computed, and the new iteration starts.

Item 3 of Unification algorithm. The occur check is hidden inside the function $\ominus$ used to define the error signal e_k, thanks to the auxiliary function `free_vars` used when defining $\ominus$. The step "put $\sigma_{k+1} = \sigma_k\{v/t\}$" is achieved by using concatenation of substitutions by function $\oplus$: $w_{hk}(t+1) = w_{hk}(t) \oplus \Delta w_{kj}(t)$. By **Remark 1**, we concatenate the substitutions $\sigma_0 \dots \sigma_k\sigma_{k+1}$, and apply them to atoms in S stepwise, that is, at each iteration of the network we use $\odot$ to apply the new computed substitution σ_{k+1} given by $\Delta w_{kj}(t+1)$ to the two atoms in $S\sigma_1 \dots \sigma_k$ given by $w_{kj}(t)$ and $d_k(t)$.

The step "increment k and go to 2" is achieved by starting a new iteration. The condition "otherwise, stop; S is not unifiable" is achieved as follows. When the substitution is not possible, or the "occur check" is not passed, $\Delta w_{kj}(t) = d_k(t) \ominus v_k(t) = \texttt{Dfail}$, and this sets $w_{hk}(t+1) = 0$. But then, $p_h(t+1) = w_{hk}(t+1) = 0$. But then, the output value $v_h(t+1) = p_h(t+1)$ is set to 0. And this will be read as "substitution failed".

Note that in case when $A = B$ and hence the mgu of A and B is ε, the neural networks will give output $v_h(t+2) = 0 \oplus 0$.

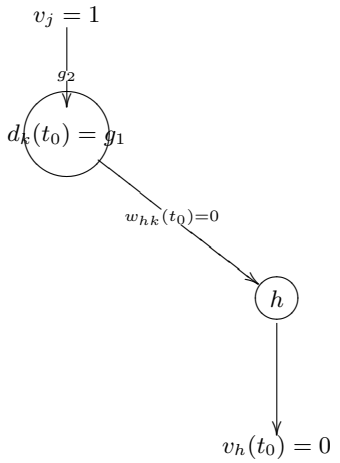
Unlike the Unification algorithm of Section 2, the neural network outputs the concatenation of the substitutions computed at each iteration, and not their composition. However, given an ordered list of computed substitutions, composing them is trivial, and can be done using the function $\odot$ instead of $\oplus$ in the networks above. The function $\oplus$ bears an advantage that one can easily check whether the output list of substitutions ends with 0, and thus it is easy to decide whether Unification algorithm has come to an answer. In general, there is no serious obstacles for using $\odot$ instead of $\oplus$ in the construction above, and thus to output composition of substitutions instead of their concatenation.

The next example illustrates how the construction of Theorem 1 works.

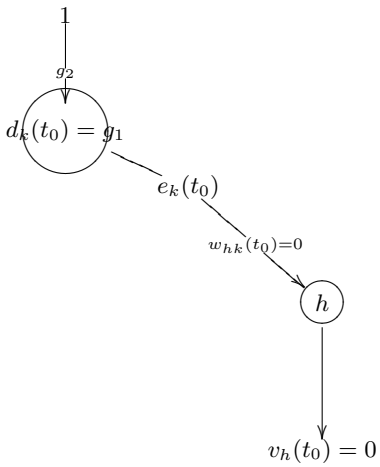
Example 6.1 *Let $S = (Q_1(f(x_1, x_2)), Q_1(f(a_1, a_2)))$ be a set of first-order atoms. Then $\theta = \{x_1/a_1;\ x_2/a_2\}$ is the mgu.*

Next I show how this can be computed by neural networks. Let g_1 and g_2 denote the chosen encoding for $Q_1(f(x_1, x_2))$ and $Q_1(f(a_1, a_2))$.

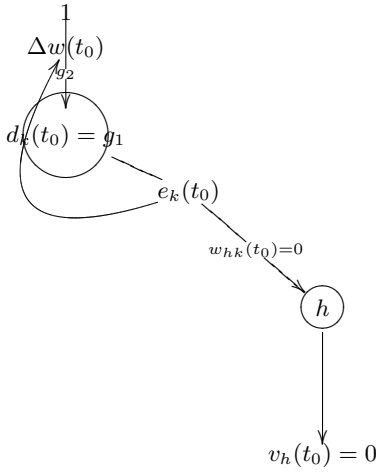
The neural network I build will consist of two units, k and h. External signal $v_j(t_0) = 1$ is sent to the unit k. I use the numbers g_1 and g_2 as parameters of the neural network, as follows: $w_{kj}(t_0) = g_2$; and hence $i_k(t_0) = v_j w_{kj} = g_2$. The desired response $d_k(t_0)$ is set to g_1. See the next diagram.



At time t_0, this neural network computes $e_k(t_0) = d_k(t_0) \ominus v_k(t_0)$ - the disagreement set $\{x, a\}$:



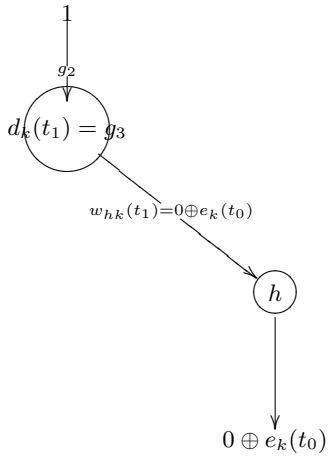
The error signal $e_k(t_0)$ will then be used to change the weight of the outer connection w_{kj} to k and some other parameters; and $\Delta w(t_0)$ is computed for this purpose as follows: $\Delta w(t_0) = e_k(t_0)$. And this is shown on the next diagram.



At time t_1, the input weight w_{kj} is amended using $e_k(t_0)$, and the desired response value $d_k(t_1)$ is "trained", too: $w_{kj}(t_1) = w_{kj}(t_0) \odot \Delta w_{kj}(t_0)$ and $d_k(t_1) = d_k(t_0) \odot \Delta w_{kj}(t_0)$. At this stage the computed substitution $e_k(t_0) = \Delta w_{kj}(t_0)$ is applied to the numbers g_1 and g_2 of the atoms $Q_1(f(x_1, x_2)), Q_1(f(a_1, a_2))$ that we are unifying.

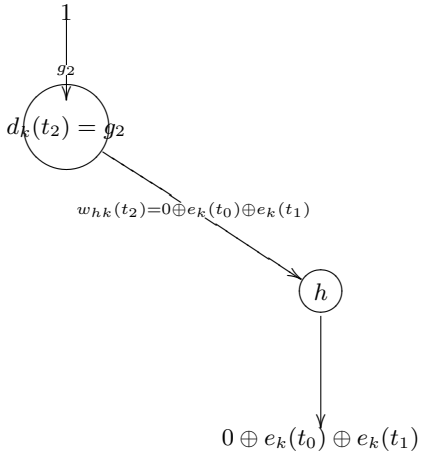
The weight between k and h is amended at this stage: $w_{hk}(t_1) = w_{hk}(t_0) \oplus \Delta w_{kj}(t_0) = 0 \oplus e_k(t_0)$.

As a result, at time t_1 all the parameters are updated as follows: input signal $v_j(t_1)w_{kj}(t_1) = g_2$ is the encoding of $Q_1(f(a_1, a_2))$; the desired response $d_k(t_1) = g_3$ is $Q_1(f(a_1, x_2))$. And this is shown on the next diagram:



Note that on the diagram above, the unit h emits its first non-zero output signal, that is, the number of substitution $e_k(t_0)$.

Because the parameters $w_{kj}(t_1) = g_2$ and $d_k(t_1) = g_3$ are not equal yet and the list v_h does not end with 0, the same iteration as above starts again. And at times $t_1 - t_2$, the number $e_k(t_1)$ of a new substitution $\{x_2/a_2\}$ is computed and applied, as follows:
$e_k(t_1) = g_2 \ominus g_3$; the input signal $v_j(t_2)w_{kj}(t_2) = w_{kj}(t_1) \odot e_k(t_1) = g_2$ is the encoding of $Q_1(f(a_1, a_2))$;
$d_k(t_2) = d_k(t_1) \odot e_k(t_1) = g_2$ is $Q_1(f(a_1, a_2))$:



At time t_2, new iteration will be initialised. But, because $d_k(t_2) = w_{kj}(t_2) = g_2$, the error signal $e_k(t_2) =$ `Dempty` and the error-correction learning rule will compute $\Delta w_{kj}(t_2) =$ `Dempty`. And then, $w_{hk}(t_3) = 0 \oplus e_k(t_0) \oplus e_k(t_1) \oplus 0$.

Note that the neuron h finally emits the signal that contains both substitutions computed by the network. The fact that the last symbol of the list $0 \oplus e_k(t_0) \oplus e_k(t_1) \oplus 0$ is 0, tells the outer reader that the unification algorithm finished its computations.

7 CONCLUSIONS

The main conclusions to be made from the construction of Theorem 1 are as follows:

- First-order atoms are embedded directly into a neural network. That is, I allow not only binary threshold units (or binary truth values 0 and 1) as in traditional neuro-symbolic networks [16, 17, 1], but also units that can receive and send a code of first-order formulae.
- Numerical encoding of first-order atoms by means of neural network signals and parameters forced us to introduce new functions, $\ominus$ and $\odot$ that work over these encodings.
- Resulting neural network is finite and gives deterministic results.
- The error-correction learning recognised in Neurocomputing is used to perform the algorithm of unification.
- Unification algorithm is performed as an adaptive process, which corrects one piece of data relatively to the other piece of data.

Discussion

The main result of this paper can raise the following questions.

Does Theorem 1 really define a connectionist neural network?

The graph defined in Theorem 1 complies with the general definition of a neural network of Section 3. Indeed, it is a directed graph, with usual parameters such as thresholds, weights, with potentials and values computed using the same formulae as in Section 3. The main new feature of these novel neural networks is the new data type of inputs and outputs, that is the type of first-order terms.

Does the network really learn?

The network follows the same scheme of error-correction learning as conventional error-correction learning neural networks, [12]. That is, we introduce the desired response value d_k in the neuron k, which is repeatedly compared with the output value v_k of the neuron k. Through the series of computations, the network amends the weight w_{kj} in such a way that the difference between d_k and v_k diminishes.

This agrees with the conventional error-correction learning rule that "trains" the weight of the given connection, and minimises the difference between the desired response and the output value. The main novelty is that the error signal is defined using $\ominus$, - the function computing the difference between terms, as opposed to using conventional "$-$" in Section 4. The error-correction learning rule is defined using $\odot$, - the function applying substitutions, as opposed to using conventional "$+$" in Section 4.

Can we use these neural networks for massively parallel computations?

There is no obstacles for composing the networks of Theorem 1. One can unify arbitrary many sets S_1, S_2, S_n, using composition of n networks from Theorem 1 working in parallel.

What is the significance of these networks?

As was shown in [18, 19], the networks of Theorem 1 can be conveniently used to formalise the algorithm of SLD-resolution for first-order logic programs. This construction can be further developed and refined in order to obtain the first neuro-symbolic theorem prover. The main advantages of the networks simulating SLD resolution [18, 19] as opposed to those simulating semantic operators [17, 8, 15, 7, 1] are their finiteness, possibility to extend to higher-order terms and the use of unification and variable substitutions, as opposed to working with ground instances of atoms and their truth values in case of semantic operators. This opens new horizons for implementing a goal-oriented proof search in neural networks.

Further work

The construction we have given here can be extended to higher-order terms and atoms. This should be relatively easy, because we do not depend on ground instances anymore.

Another direction for research would be to embed the neurons performing unification into the existing neuro-symbolic networks, such as those described in [7, 9].

The networks we present here can be useful for further development of neuro-symbolic networks formalising inductive logic and inductive reasoning.

Finally, we envisage to complete in the near future the full Coq formalisation of Theorem 1.

ACKNOWLEDGEMENTS

I would like to thank Laurent Théry for Coq formalisations of functions $\ominus$, $\odot$ ([30]), and for stimulating discussions of the unification algorithm in Coq.

I thank the anonymous referees for their useful comments and suggestions.

REFERENCES

[1] S. Bader and P. Hitzler, 'Dimensions of neural symbolic integration - a structural survey', in *We will show them: Essays in honour of Dov Gabbay*, ed., S. Artemov, volume 1, 167–194, King's College, London, (2005).

[2] S. Bader, P. Hitzler, S. Hölldobler, and A. Witzel, 'A fully connectionist model generator for covered first-order logic programs', in *Proceedings of the 20th International Conference On Artificial Intelligence IJCAI-07*, Hyderabad, India, (2007).

[3] S. Bader, P. Hitzler, and A. Witzel, 'Integrating first-order logic programs and connectionist systems — a constructive approach', in *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05*, eds., A. S. d'Avila Garcez, J. Elman, and P. Hitzler, Edinburgh, UK, (2005).

[4] J.A. Barnden, 'On short term information processing in connectionist theories', *Cognition and Brain Theory*, **7**, 25–59, (1984).

[5] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Constructions*, Springer-Verlag, 2004.

[6] A. Bove, *General Recursion in Type Theory*, Ph.D. dissertation, Department of Computing Science, Chalmers University of Technology, 2002.

[7] A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay, *Neural-Symbolic Learning Systems: Foundations and Applications*, Springer-Verlag, 2002.

[8] H. W. Güsgen and S. Hölldobler, 'Connectionist inference systems', in *Parallelization in Inference Systems*, eds., B. Fronhöfer and G. Wrightson, Springer, LNAI *590*, (1992).

[9] B. Hammer and P. Hitzler, *Perspectives of Neural-Symbolic Integration*, Studies in Computational Intelligence, Springer Verlag, 2007.

[10] J. Harrison, *Introduction to Logic and Automated Theorem Proving*, 2004.

[11] S. Haykin, *Neural Networks. A Comprehensive Foundation*, Macmillan College Publishing Company, 1994.

[12] R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, 1990.

[13] P. Hitzler, S. Hölldobler, and A. K. Seda, 'Logic programs and connectionist networks', *Journal of Applied Logic*, **2(3)**, 245–272, (2004).

[14] S. Hölldobler and F. Kurfess, 'CHCL – A connectionist inference system', in *Parallelization in Inference Systems*, eds., B. Fronhöfer and G. Wrightson, 318 – 342, Springer, LNAI *590*, (1992).

[15] S. Hölldobler, 'A structured connectionist unification algorithm', in *Proceedings of the AAAI National Conference on Artificial Intelligence*, pp. 587–593, (1990).

[16] 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*, pp. 68–77. ECCAI, (1994).

[17] S. Hölldobler, Y. Kalinke, and H. P. Storr, 'Approximating the semantics of logic programs by recurrent neural networks', *Applied Intelligence*, **11**, 45–58, (1999).

[18] E. Komendantskaya, 'First-order deduction in neural networks', *Submitted to the Journal of Information and Computation Postproceedings volume of LATA'07*, 26 pages, (2007).

[19] E. Komendantskaya, *Learning and Deduction in Neural Networks and Logic*, Ph.D. dissertation, Department of Mathematics, University College Cork, Ireland, 2007.

[20] R. A. Kowalski, 'Predicate logic as a programming language', in *Information Processing 74*, pp. 569–574, Stockholm, North Holland, (1974).

[21] T. E. Lange and M. G. Dyer, 'High-level inferencing in a connectionist network', *Connection Science*, **1**, 181 – 217, (1989).

[22] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 2nd edn., 1987.

[23] D. Nauck, F. Klawonn, R. Kruse, and F.Klawonn, *Foundations of Neuro-Fuzzy Systems*, John Wiley and Sons Inc., NY, 1997.

[24] J.A. Robinson, 'A machine-oriented logic based on resolution principle', *Journal of ACM*, **12**(1), 23–41, (1965).

[25] A. K. Seda, 'On the integration of connectionist and logic-based systems', in *Proceedings of MFCSIT2004, Trinity College Dublin, July, 2004*, eds., T. Hurley, M. Mac an Airchinnigh, M. Schellekens, A. K. Seda, and G. Strong, volume 161 of *Electronic Notes in Theoretical Computer Science*, pp. 109–130. Elsevier, (2005).

[26] L. Shastri and V. Ajjanagadde, 'An optimally efficient limited inference system', in *Proceedings of the AAAI National Conference on Artificial Intelligence*, pp. 563–570, (1990).

[27] L. Shastri and V. Ajjanagadde, 'From associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony', *Behavioural and Brain Sciences*, **16**(3), 417–494, (1993).

[28] P. Smolensky, 'On the proper treatment of connectionism', *Behavioral and Brain Sciences*, **11**, 1–74, (1988).

[29] L. Théry. Formalisation of unification algorithm, 2008.

[30] L. Théry. Functions for neural unification: Coq code, 2008. `www.cs.ucc.ie/~ek1/Neuron.v`.