

# Unification Neural Networks: Manual

Ekaterina Komendantskaya

*School of Computer Science, University of St Andrews*

---

## Abstract

This preliminary version of the paper published in the Journal of Algorithms in Cognition, Informatics and Logic, is based on the MATLAB library implementing Unification Neural Networks. It currently serves as a manual for the library. We show that the conventional first-order algorithm of unification can be simulated by finite artificial neural networks with one layer of neurons. In these *unification neural networks*, the unification algorithm is performed by error-correction learning. In particular, the unification neural network is given a target, - to reach the state when the difference between two atoms it unifies is zero. It then sees the non-empty disagreement set as an “error” between the output and the target, and learns the computed substitution as a change vector. It uses this vector to adapt its biases and weights. Each time step of adaptation of the network corresponds to a single iteration of the unification algorithm. We present this result together with the library of learning functions and examples fully formalised in MATLAB Neural Network Toolbox.

*Key words:* Unification, Neuro-Symbolic Networks, Neural Network Learning, Error-correction learning, Connectionism

---

## 1. Introduction

Unification is a fundamental process that occurs in several fields of computer science, including theorem proving, logic programming, natural language processing, computational complexity, and computability theory. This paper considers the problem of implementing unification in Neuro-Symbolic (Connectionist) networks.

Connectionism [1, 2] is a movement in the fields of artificial intelligence, cognitive science, neuroscience, psychology and philosophy of mind, which hopes to explain human intellectual abilities using artificial neural networks. Connectionism relies on the assumption that artificial neural networks are simplified models of the brain.

Neuro-symbolic integration is a particular area within Connectionism; it investigates ways of integration of logic and formal languages in neural networks, in order to better understand the essence of symbolic (deductive) and human (developing, spontaneous) reasoning, and to show interconnections between them. The books [3, 4] and papers [5, 6, 7, 8, 9, 10, 11, 12] are good examples of this approach.

The joint efforts of researchers in many areas have given many insights on how logic and neuroscience can relate: Boolean (binary) networks can compute logical connectives ([13, 14, 15]); binary threshold networks can simulate Finite Automata [16], and (universal) Turing machines can be simulated by neural networks with rational weights [17], and neural networks in their full potential are as powerful as analog computing [17].

The natural question that arises is how neural networks can cope with logical theories and calculi. There were built neural networks that can simulate the work of the semantic operator  $T_P$  for propositional and (function-free) first-order logic programs; [8, 9]. We will call these networks  $T_P$  *neural networks*.  $T_P$  networks process classical truth values 0 and 1 assigned to ground instances of formulae contained within a given logic program. These binary values are presented to the  $T_P$  networks as input vectors and emitted as output vectors. Essentially, the  $T_P$  networks implement Boolean networks of McCulloch and Pitts [13] to processing the  $T_P$  operator. This approach has been very popular, and inspired research into implementations of different kinds of semantic operators in neural networks; [18, 19, 3, 4, 12, 20, 21, 22].

Research into  $T_P$  neural networks opened a graceful way to avoid the problem of implementing unification and goal-oriented proof search in neural networks. The  $T_P$  operator does not employ any form of unification, and deals only with ground instances of first-order formulae. Once only ground instances are required for computations, one can easily process the truth values 0 and 1 assigned to ground formulae, instead of processing the formulae syntactically.

The  $T_P$  networks have two major technical disadvantages. The first technical disadvantage arises from the fact that it can take an infinite number of ground first-order atomic formulae for the  $T_P$  operator to reach its fixed

point, and in this case the  $T_P$  neural networks have to have an infinite size, which is fatal for practical implementations. This problem was tackled using topological arguments and approximations of infinite computations by finite, [18, 19, 12, 9, 23].

Also,  $T_P$ -neural networks for classical logic programs could not learn or perform any form of self-organisation or adaptation, which set them aside from the learning networks traditionally studied in neurocomputing. There were several attempts to bring learning and self-adaptation in  $T_P$  neural networks by means of generalising them to non-classical and inductive logic programs, see, for example, [3, 24, 25, 26, 27, 28].

The theoretical disadvantage of the  $T_P$  neural networks is that they did not provide any clue to how non-ground reasoning may be handled in the neural networks. How exactly would neural networks “reason” if sentences are non-ground and it is impossible to assign truth values 0 and 1 to them? For example, the sentences “If Tweety is a bird, then Tweety can fly”, “7 is a natural number” are examples of ground sentences, hence we can assign truth values to them. But in every-day life, in logic, or in mathematics we can reason perfectly well about sentences containing variables, such as “If X flies, then X is a bird”; “If a number X can be obtained from the number 0 by adding 1 several times, then X is a natural number.” That is, we can reason without assigning truth values. And indeed, one does not have to examine the whole population of birds, or compute the whole infinite set of natural numbers in order to reason about them. And this is why Logic programs are run by the SLD-resolution that can handle non-ground sentences and incorporates unification.

In this paper, we show how to implement the algorithm of unification in neural networks. To do this, we have to abandon Boolean networks that can process only 0 and 1, for we will not unify sentences by taking their ground instances and truth values. Instead, finite neural networks that can process integers will be sufficient for performing unification. These neural networks will have linear activation function, and will consist of one layer of neurons. The interesting feature of these neural networks is that they employ an error-correction learning function in order to unify terms.

The standard error-correction learning in neural networks has the following behaviour. A network is supplied with targets. Then, a network receives some input, and processes it according to the processing and transfer functions that are predefined and embedded into the network. As a result, the network sends an output. This output is compared with the target. The net-

work computes an error - the difference between the target and the output. Then the network changes its weights or biases using this computed error, in order to minimize the error on the next iteration. If, at the next iteration, the error is equal to 0, the learning stops, otherwise the network goes on learning and minimizing the error.

Surprisingly, the conventional Unification algorithm for first-order atoms [29, 30, 31] follows a pattern similar to the error-correction. Suppose we are given two first-order formulae  $A$  and  $B$ . And we apply the algorithm of Unification [30]. The algorithm has its “target” - to reach the state when no terms contained in  $A$  and  $B$  disagree: that is, the state when the difference between  $A$  and  $B$  is equal to 0. The algorithm finds the disagreement set containing the first two non-equal terms in  $A$  and  $B$ , and finds a substitution for them. This computed substitution, derived from the disagreement set, is similar to a computed “error signal” in a neural network. Then the substitution is applied to  $A$  and  $B$ , similarly to how the error signal is applied to a bias and a weight, and the algorithm starts its new iteration, that is, it finds a new disagreement set, and so on, until it reaches the state when the disagreement set is empty (the “error” is 0).

In this paper, we exploit this simple analogy to its full potential. The idea of doing unification by error-correction was first spelled out in my PhD Thesis [26], but it took some time to refine its realisation [32, 33] and fit it into the framework of a conventional neural network simulator, in order to implement and test the whole idea.

For example, in early versions [26, 32], the tools for numeric representations of logic formulae played an important role, whereas in the final version we present here, we avoid this problem altogether and use just arbitrary (ASCII) encoding provided by standard MATLAB library. In all the earlier descriptions [26, 32, 33], the neural networks had to process lists of numbers (“Gödel numbers”), whereas now we abandoned this idea in favour of vectors, which fits nicely into the framework of neurocomputing and is definable in the environment of a neural network simulator.

These two major modifications to the initial architecture made possible to complete the formalization of the *Unification Neural Networks* within the MATLAB Neural network toolbox [34], and inspired numerous minor changes on the way. The library [35] is one of important contributions of this paper, and we describe it in detail here. It is the first implementation of the Unification algorithm in neural networks; and it opens the way to further experiments, applications and developments.

The structure of the paper is as follows. In Section 2, we define the first-order language and the algorithm of first-order unification for it, following [29, 30]. In Section 3, we define artificial neural networks following [36, 37, 34], and in particular, we describe the mechanism of error-correction learning in neural networks.

In Section 4, we show how to construct a unification neural networks (UNNs) for any two function-free first-order atoms; and how it can be implemented in the MATLAB Neural Network Simulator (MNNS). We describe how to simulate the UNNs. Starting with this Section, we support all our descriptions with experiments conducted in the MNNS using the library [35]. In Section 5, we introduce the error-correction into the UNNs we build. We explain how error-correction performs unification for atoms not containing function symbols. In Section 6 we extend these results to the more complex cases, when function symbols are contained in the first-order atoms, and the terms (as well as UNNs) may grow in the process of unification. We prove that UNNs implement the algorithm of unification in a sound way.

In Conclusions, we discuss the significance and possible future development and implementations of the UNNs.

Appendices A, B, C contain definitions of the three major functions we defined in [35].

## 2. First-order Unification

In this section, we consider standard definitions of first-order alphabet and language. The first-order language  $\mathbf{L}$  consists of the well-formed formulae built from the symbols of the alphabet  $\mathbf{A}$ .

**Definition 1.** We fix the *alphabet*  $\mathbf{A}$  to consist of

- constant symbols  $a_{01}, a_{02}, \dots, a_{99}$ ,
- variables  $x_{01}, x_{02}, \dots, x_{99}$ ,
- function symbols  $f_{01}, f_{02}, \dots, f_{99}$ ,
- predicate symbols  $P_{01}, P_{02}, \dots, P_{99}$ ,
- connectives  $\neg, \wedge, \vee$ , also called *negation*, *conjunction* and *disjunction*.
- quantifiers  $\forall, \exists$  and

- punctuation symbols “(”, “,” “)”.

Note that we have chosen to use only finitely many symbols in the alphabet. This is done for convenience, and we will explain the use and impact of this decision in the later sections. Finite alphabet does not mean we restrict ourselves to a finite language or finite computations. An alphabet containing only one constant, one variable, one function symbol and one predicate would be sufficient to define a logic program that can compute the whole set of natural numbers; see Example 3. So, for the time being, we assume that this finite version of the first-order alphabet is sufficient for our purposes.

We follow the conventional (inductive) definition of a term and a formula. Namely, every constant symbol is a term, every variable is a term, and if  $f_i$  is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms, then  $f_i^n(t_1, \dots, t_n)$  is a *term*. We will sometimes write  $\bar{t}$  instead of  $(t_1, \dots, t_n)$ .

Let  $P^n$  be a predicate symbol of arity  $n$  and  $t_1, \dots, t_n$  be terms. Then  $P(t_1, \dots, t_n)$  is a *formula* (also called an atomic formula or an *atom*). If  $F_1, F_2$  are formulae and  $\bar{x}$  are variables, then  $\neg F_1, F_1 \wedge F_2, F_1 \vee F_2, \forall \bar{x} F_1$  and  $\exists \bar{x} F_1$  are formulae.

**Definition 2.** The *first-order language*  $\mathbf{L}$  given by the alphabet  $A$  consists of the set of all formulae constructed from the symbols of the alphabet.

**Example 1.**  $\forall x_{01} \forall x_{02} (P_{01}(x_{01}, x_{02}) \vee \neg P_{01}(x_{01}, x_{02}))$  is a formula of the language  $\mathbf{L}$ .

**Definition 3.** A *ground term* is a term not containing variables. Similarly, a *ground atom* is an atom not containing variables.

Next, we define the algorithm of unification as it was introduced by [30] and [31]; see also [38].

**Definition 4** (Unifier). Let  $S$  be a finite set of atoms. A substitution  $\theta$  is called a *unifier* for  $S$  if  $S\theta$  is a singleton. A unifier  $\theta$  for  $S$  is called a *most general unifier (mgu)* for  $S$  if, for each unifier  $\sigma$  of  $S$ , there exists a substitution  $\gamma$  such that  $\sigma = \theta\gamma$ .

Given substitutions  $\theta_1, \dots, \theta_n$ , we can compose them, and we will denote their composition by  $\theta_1 \dots \theta_n$ .

**Definition 5** (Disagreement set). Let  $S$  be a finite set of atoms. 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 subexpression beginning at that symbol position. The set of all such terms is the disagreement set.

**Unification algorithm**[29, 30]:

1. Put  $k = 0$  and  $\theta_0 = \varepsilon$ .
2. If  $S\theta_k$  is a singleton, then stop;  $\theta_k$  is an mgu of  $S$ . Otherwise, find the disagreement set  $D_k$  of  $S\theta_k$ .
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.

**Theorem 1** (Unification Theorem [30]). Let  $S$  be a finite set of atoms. 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.

For technical convenience we refine the notion of the disagreement set as follows.

**Definition 6.** Given atoms  $A$  and  $B$ , restricted disagreement set (*r-disagreement set*) for  $A$  and  $B$  is a disagreement set for  $A$  and  $B$  that contains at least one variable.

That is, the restricted disagreement set cannot consist of two constants, or two terms starting with two different function symbols. In practice, only r-disagreement sets contribute substitutions in the process of unification of two given atoms. In what follows, we will use r-disagreement sets instead of the disagreement sets whenever we unify two atoms. If we replace the occurrences of the “disagreement set” with “r-disagreement set” in the algorithm of unification, only one sentence needs to be added in item 2: “If the r-disagreement set  $D_k$  does not exist, then stop.  $S$  is not unifiable.” The two formulations of the unification algorithm are clearly equivalent. The latter reformulation is restricted to sets that contain only two atoms and it makes explicit the assumption that has been implicitly contained in the initial algorithm.

**Example 2.** We will illustrate how this algorithm works on a simple example. Consider the set  $S = \{P_{01}(f_{01}(a_{01}, a_{02})), P_{01}(f_{01}(x_{01}, x_{02}))\}$  and form  $D_S = \{x_{01}, a_{01}\}$ . Put  $\theta_1 = x_{01}/a_{01}$ . Now  $S\theta_1 = \{P_{01}(f_{01}(a_{01}, a_{02})), P_{01}(f_{01}(a_{01}, x_{02}))\}$ . Find  $D_{S\theta_1} = \{x_{02}, a_{02}\}$  and put  $\theta_2 = x_{02}/a_{02}$ . Now  $S\theta_1\theta_2$  is a singleton. Unification stops.

**Example 3.** Although here, we do not consider logic programs in any further details, it is useful to note that the algorithm of unification would play an important role in processing the following small logic program that can compute the set of natural numbers:

$P_{01}(a_{01}) \leftarrow$

$P_{01}(f_{01}(x_{01})) \leftarrow P_{01}(x_{01})$

We think of  $a_{01}$  as 0,  $f_{01}$  as a successor, and  $P_{01}$  as the property that is being defined - that is, the property of being a natural number. The goal to find a natural number would be stated as:  $\leftarrow P_{01}(x_{02})?$ . And then the interpreter will be able to unify the goal first with  $P_{01}(a_{01})$ , and then - with  $P_{01}(f_{01}(a_{01}))$ , and so on...

Already this simple example would require an infinite number of neurons if we simulate it in a  $T_P$ -neural network.

### 3. Neural Networks and Error-correction Learning

In this section, we give formal definitions of neural networks. We follow [36, 37, 34].

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*  $(v_{i_1}(t), \dots, v_{i_n}(t))$ , its potential  $p_k(t)$ , its bias  $b_k$  and its *value*  $v_k(t)$ . In general, all  $v_i$ ,  $p_i$ ,  $b_k$ , as well as all other parameters of a neural network can be represented by different types of data, the most common of which are real numbers, rational numbers [9], fuzzy (real) numbers [21], complex numbers, numbers with floating point, and some others, see [36] for more details. 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  $t$ . In 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 plus their bias:

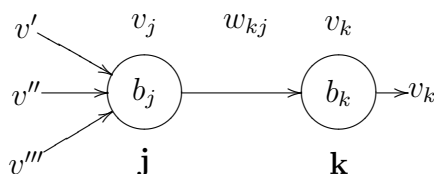
$$p_k(t) = \left( \sum_{j=1}^{n_k} w_{kj} v_j(t) \right) + b_k.$$

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 *transfer function*  $F$ , that is,  $v_k(t + \Delta t) = F(p_k(t))$ .

For example, the transfer function used in [8] 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 0 otherwise. Units of this type are called *binary threshold units*.

A unit is said to be a *linear unit* if its transfer function is the identity, that is,  $v_k(t + \Delta t) = p_k(t)$ . In MATLAB Neural Network Simulator, it is called `purelin`.

**Example 4.** Consider two units,  $j$  and  $k$ , having potentials  $p_j, p_k$  and values  $v_j, v_k$ . The weight of the connection between units  $j$  and  $k$  is denoted  $w_{kj}$ . 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  $v_k$ .



Among all the parameters of neural networks, there are two parameters that are conventionally considered as capable of learning, training and adapting: they are the weights and biases.

We will consider networks where the units can be organised in layers. A *layer* is a vector of units. An  $n$ -*layer 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,  $1 \leq i < n$ . Neural networks consisting of layers are sometimes called *associative neural networks* [36]. Here, we will work with one-layer networks.

*Error-correction learning* is a kind of *supervised learning*. Supervised learning is the most popular type of learning implemented in artificial neural networks, and we give a brief sketch of error-correction algorithm in this subsection; see, for example, [37] for further details.

Let  $d_k(t)$  denote some *desired response*, or target, 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  $e_k(t) = d_k(t) - v_k(t)$ .

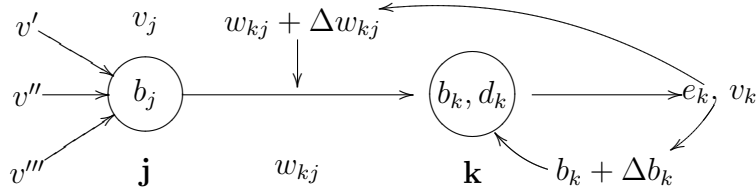
The *error-correction learning rule* is the adjustment (the change signal)  $\Delta w_{kj}(t)$  (or  $\Delta b_k(t)$ ) made to the weight  $w_{kj}$  or the bias  $b_k$  at time  $n$  and is given by

$$\begin{aligned}\Delta w_{kj}(t) &= \eta e_k(t) v_j(t) \\ \Delta b_k(t) &= \eta e_k(t) v_j(t)\end{aligned}$$

where  $\eta$  is a positive constant that determines the rate of learning. Note that one can train either weights, or biases, or both.

Finally, the formulae  $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$  and  $b_k(t+1) = b_k(t) + \Delta b_k(t)$  are used to compute the updated values for the weight  $w_{kj}(t)$  and the bias  $b_k(t)$ .

**Example 5.** The neural network from Example 4 can be transformed into an error-correction learning neural network as follows. We introduce the *desired response* value  $d_k$  into the unit  $k$ . The change signal  $\Delta w_{kj}$  computed using  $e_k$  must be sent to the connection between  $j$  and  $k$  to adjust  $w_{kj}$ ; similarly for  $b_k$ .



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, biases, targets, errors, and other parameters. In this case, we will have to drop the subscripts and write simply  $w$ ,  $b$ ,  $t$ ,  $e$  for vectors of weights, biases, targets (desired responses), and errors respectively. We will call the vector of errors  $e$  an *error vector*, and the vectors  $\Delta w$  and  $\Delta b$  - the *change*

vectors. To be consistent with MATLAB notation, we will write  $dw$  and  $db$  for  $\Delta w$  and  $\Delta b$ .

#### 4. Construction of the Unification Neural Networks

In this section, we describe the structure of the neural networks that can simulate the algorithm of unification, we will call such networks *Unification Neural Networks* (UNNs). To simplify the description, we first explain how to build networks that can unify only first-order atoms not containing function symbols. Having achieved that, we will devote Section 6 to the additional algorithm of completion needed for unifying atoms containing function symbols. The implementation was done in MATLAB neural network simulator (MNNS), and the library of functions and examples can be found here [35].

##### Construction of the unification neural network (UNN):

1. The network we build consists of 1 layer of neurons. This layer has its bias  $b$ , the vector of input weights  $w$  and will normally receive only one input signal - 1. After processing the input signal, it will emit an output vector. The Figure 1 shows a network `net1` that can unify two atoms  $P_{01}(x_{12}, a_{15})$  and  $P_{01}(a_{13}, x_{02})$ . The single layer of `net1` contains 12 neurons, one neuron for each symbol contained in  $P_{01}(a_{13}, x_{02})$ , but the size of the layer is not our concern at this stage. We follow the MATLAB notation and abbreviate layers by a single box, as shown on the Figure 1 generated by MATLAB. For various first-order atoms, only the length of this single layer will vary. This is why in this notation, UNNs for any two first-order atoms will look exactly the same. The reader can find this example and many more in the file `test_learning.mat` in [35].

Unification networks use several pre-defined in MNNS functions. For example, we use the transfer function `purelin` - the linear transfer function, also employed in the famous Perceptron. It is depicted as a diagonal line inside of the layer of `net1` in Figure 1. Another pre-defined function we employ here is the activation function `netsum` (denoted by  $+$  in Figure 1), one can set it in MNNS by simply typing `net1.layers1.netinputFcn = 'netsum'`. The function returns element-wise sum of the weighted input signal and the bias. That is, if the input is 1, the vector  $w$  representing input weights has the shape  $[n_1; n_2; \dots; n_r]$ , and the bias vector  $b = [m_1; m_2; \dots; m_r]$ , the result

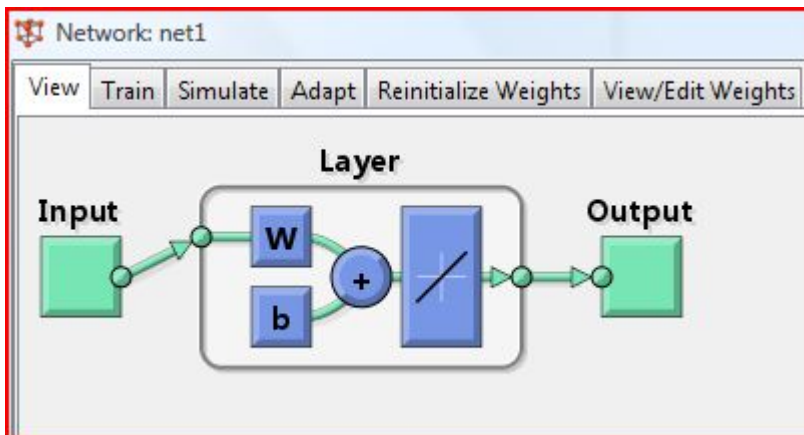


Figure 1: The Unification network `net1`

of application of this function will be  $[n_1 + m_1, n_2 + m_2, \dots, n_r + m_r]$ . For further details, see also Documentation of MNNS, [34].

No other special structural settings are needed to simulate the unification networks. Note also that all these general settings (also called “object settings” in MNNS) are defined once and for all UNNs, irrespective of the size and shape of the first-order atoms we unify.

**2.** Next step will be to decide the subobject properties: size of the layer, and values assigned to the weight ( $w$ ) and bias ( $b$ ) vectors. These will vary from one UNN to another, depending on the atoms we unify. Unlike any kind of Boolean or  $T_P$  neural network, we embed the first-order atoms directly into the UNN, via the values of  $w$  and  $b$ . It is a strict requirement that neural networks can have only numerical signals and parameters. Therefore, we need some numerical encoding for atoms we unify.

In general, given first-order atoms  $A$  and  $B$ , and their numerical vector encodings  $v_A$  and  $v_B$ , we set the size of the layer to be equal the size of  $v_B$ , and  $w = v_A$  and  $b = -v_B$ . The parameters  $w$  and  $b$  are chosen to represent the first-order atoms, because precisely these two parameters can be trained in conventional neural networks and in MNNS.

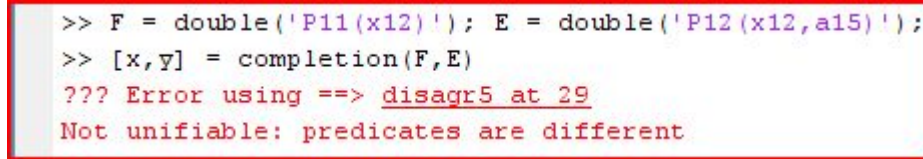
**3. Vector encoding.** The vector encoding is done in two steps. First we assign a number to each of the symbols of the alphabet  $\mathbf{A}$ . The only natural condition we impose on the assignment is that it should be a one-to-one map from symbols of the alphabet to integers. In the library we develop, we simply use ASCII encoding provided by MATLAB library. Other kinds of encoding

may add efficiency, but our example serves to illustrate that essentially, the way the encoding is made is unimportant for the development. The next step is to represent a given formula or term as a vector of numbers that are determined by the assignment.

**Example 6.** In MATLAB, we can type “`double(P01(a01))`”, and receive the vector: `[80,48,49,40,97,48,49,41]`. The inverse function, `char`, is also defined in MATLAB and will turn the vector back into the symbolic shape. MATLAB also determines the size of the vector - in this case it is 8.

Let  $l_A$  and  $l_B$  be the sizes of vectors  $v_A$  and  $v_B$ . In this section, we simply assume that  $l_A$  and  $l_B$  are equal. As soon as we assume that we consider meaningful cases - that is, when predicates are the same, then the lack of function symbols will effectively mean that  $l_A = l_B$ .

In general, there is a special checker (function `completion`) embedded into the library, that detects non-miningful cases, and reports if two different predicates are contained in  $A$  and  $B$ , see Figure 2. Details will follow in Section 6.



```
>> F = double('P11(x12)'); E = double('P12(x12,a15)');
>> [x,y] = completion(F,E)
??? Error using ==> disagr5 at 29
Not unifiable: predicates are different
```

Figure 2: Formulae containing two different predicates

**4. Simulation.** We are ready to simulate the network. Having set the input signal to 1, we simply type `sim(net,1)`, and the network outputs a vector,  $\text{out} = w + b = v_A - v_B$ , that is, the difference between  $v_A$  and  $v_B$ .

**Example 7.** We simulate the network `net1` from Figure 1, with the layer of size 12. We set `net1.iw1,1 = transpose(double('P01(x12,a15)'))` and `net1.b1 = - transpose(double('P01(a13,x02)'))`.

The function `transpose` simply changes the horizontal vector into a vertical vector, to fit into the MNNS settings. We type `sim(net1,1)` and get the answer: `[0; 0; 0; 0; 23; 0; -1; 0; -23; 1; 3; 0]`. One can use graphical interface, see Figure 3.

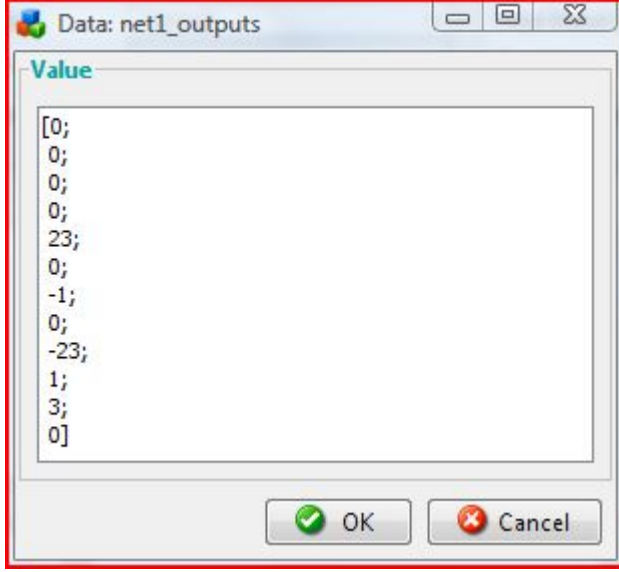


Figure 3: Simulation of `net1`

As yet, the output bears little symbolic meaning, but it signals that the atoms are not the same; and also the first non-zero symbol in the vector `net1.outputs` shows the exact position where the differences start. That is, the number 23 is precisely the difference between numbers encoding  $x$  and  $a$ . These data will be important for the learning functions we are about to introduce.

**Example 8.** We return to Example 3 of a logic program computing natural numbers. We can create the network `netN` that can unify  $P_{01}(a_{01})$  and  $P_{01}(x_{02})$  using the template given by `net1`. We simply have to change the size of the layer from 12 to 8; and set `netN.iw1,1 = transpose(double('P01(x02)'))` and `netN.b1 = - transpose(double('P01(a01)'))`. The result of simulation will be the difference between the two vectors, which is `netN_out = [0; 0; 0; 0; 23; 0; 1; 0]`.

These examples and many more are formalised and available in the file `test_learning.mat` in [35].

We summarise the construction algorithm we have described in the following Lemma:

**Lemma 2.** Given two first-order atoms  $A$  and  $B$  of the same length and built from the same predicates, there exists a UNN corresponding to them. It will have a single layer of length  $l_A = l_B$  with the linear activation function `purelin`. The UNN's input weight vector will be computed as  $w = v_A$ , and the bias vector  $b = -v_B$ . Moreover, if a signal 1 is sent to the UNN, it will output the vector  $\text{out} = v_A - v_B$ .

*Proof.* To construct the UNN, one needs to follow the steps 1 - 3 described in this section. The output function is computed using the standard formula  $\text{out} = \text{purelin}(v_{\text{in}}w + b)$ . Substituting  $v_{\text{in}}$  by 1; and values  $w$  and  $b$  as described in Steps 1 - 3, one gets precisely the required formula  $\text{out} = v_A - v_B$ . If  $A$  and  $B$  are equal,  $v_A = v_B$ , and so the output will be the vector of zeros.  $\square$

## 5. Unification by Error-correction Learning

We can now train the networks. The ultimate goal of the unification algorithm is to make two first-order atoms equal. So, in terms of UNNs, we wish the difference between  $w$  and  $b$  to be 0, and so we supply the UNN with the target vector  $t$  that contains only 0's.

MNNS contains the zero target vector as a default target for training, and so we do not have to type the value for  $t$  when calling the function `adapt` in MNNS. So, instead of typing `adapt(net,in,t)`, we simply type `adapt(net,in)`, `in` stands for the input value.

The error  $e$  is computed using the conventional formula  $e = t - \text{net}_{\text{out}}$ ; in our case  $e = -\text{net}_{\text{out}}$ .

We use the standard training and adaptation functions pre-defined in MNNS. Namely - the adaptation will be performed by the function `adapt` that, given a network, an input signal, and a target, changes biases and weights according to a learning function, and outputs the adapted network, together with its output and its error.

Another predefined function we use is the training and adaptation function `trains`. This function is traditionally used inside `adapt`; it trains a network with weight and bias learning rules with *sequential updates*. The latter means that the sequence of inputs is presented to the network with updates occurring after each time step. This incremental training algorithm is commonly used for adaptive applications. To set it in MNNS, we simply type `net.adaptfcn = 'trains'`.

Now we have come to the point when we introduce the first original function - the learning function `learn_disagr`. It seems to be significant and positive that at least till this point, we could develop the UNNs smoothly using only standard settings of the conventional neural network simulator.

The new learning function will be able to process the error signal in a more clever way than arithmetical operations do. It relies on two auxiliary functions - `disagreement`<sup>1</sup> and `substitution`, see also Appendices A and B, or [35].

`Disagreement` detects the first non-zero element in the error vector, and, using the values of  $b$  and  $w$  as parameters, infers two vectors  $D_1$  and  $D_2$  that encode the two terms this error originates from. For given atoms  $A$  and  $B$ , such that  $v_A = w$  and  $v_B = b$ , the pair  $\{D_1, D_2\}$  is effectively numerical representation of the r-disagreement set for  $A$  and  $B$ . That is, `disagreement` also detects cases when neither of the two terms in the disagreement set is a variable, and it reports an error in this case.

Having this data, `substitution` infers a substitution (change) vector from the disagreement set  $\{D_1, D_2\}$  of  $v_A = w$  and  $v_B = -b$ . It also performs the occur check; if the check fails, `substitution` reports an error. It is also responsible for encoding this substitution as a vector of the same size as the layer of neurons, otherwise the change vector will be rejected by the network.

Finally, we put `substitution` inside of the learning function `learn_disagr`: `dw = substitution(w, (-e))`. Additionally, `learn_disagr` has a number of pre-defined in the MNNS parameters and settings that make it possible for the adaptation function `adapt` to recognise and use it as a well-defined learning function.

**Example 9.** To illustrate the way of defining a learning function in MNNS, consider the Perceptron learning rule. The MATLAB learning function for it, `learnp`, contains many lines of standard definitions and settings, but the actual computation is performed using the function `dw = e*p'`. The function `learn_disagr` will differ from `learnp` (or any other standard learning function), only in this single last line, namely, we have `dw = substitution(w, (-e))`. Both functions use the error vector  $e$ , but then `learnp` uses the input vector  $p$ , whereas we use the input weight vector  $w$ , which is another possible argument for learning functions. The major difference is that we use the function `substitution` instead of multiplication.

---

<sup>1</sup>In the library [35], I call this function `disagr5`.



Weights and biases of the network receive the change signal computed by `substitution` and communicated through `learn_disagr`, and update their values. The weight and bias update is entirely delegated to the conventional functions `adapt` and `trains` predefined in MNNS.

For technical convenience, in [35] we split the learning function `learn_disagr` into `learn_disagr_w` that learns the change vector  $dw$  for weights, and `learn_disagr_b` that calculates a similar vector  $db$  for biases. Then it is convenient to have two variants of `substitution` from Appendix B - `substitution1` and `substitution2`. The former takes the weight and error vectors as arguments, the latter takes the bias and error vectors as arguments. Then, `substitution1` outputs the change vector for  $w$ , and `substitution2` - for  $b$ .

**Example 10.** We continue to develop Example 8. For the two atoms  $P_{01}(a_{01})$  and  $P_{01}(x_{02})$ , we have set  $w$  and  $-b$  of `netN` be their numerical encodings. Figure 4 shows the change vector computed as a result of applying `learn_disagr_w` to two vectors  $w = WN$  and  $e = -(WN - BN) = -(w + b)$ .

```
>> learn_disagr_w(WN, -(WN-BN))
x02
a01

ans =

      0
      0
      0
      0
     -23
      0
     -1
      0

fx >>
```

Figure 4: The substitution vector for  $w$  in `netN`

The terms  $x_{02}$  and  $a_{01}$  are computed by `disagreement` and displayed: they are the disagreeing terms. The answer `ans` is the change vector  $dw$

for  $w$ . Note that  $dw$  and  $w$  have the same size. Adding this vector to  $w$  would yield precisely the encoding of  $P_{01}(a_{01})$ , see Figure 5. And so, only one iteration would suffice to unify the two terms.

```
>> WN = char(transpose(WN+ans))

WN =

P01(a01)
```

Figure 5: The change vector  $dw$  ( $=$  ans) applied to the weight  $w$  of `netN`

Calling `[netN,y,e,pf] = adapt(netN,1)` will automatically complete the steps we have just done manually: it will call the learning functions `learn_disagr_w` and `learn_disagr_b` automatically, add the change vectors  $dw$  and  $db$  to weigh and bias vectors  $w$  and  $b$ , and update the network. And on the next iteration, the network will output zeros - which means that the two terms are unified. See also Figure 6.

```
>> [netN,y,e,pf] = adapt(netN,1);
x02
a01
>> out = transpose(sim(netN,1))

out =

    0    0    0    0    0    0    0    0
```

Figure 6: Adaptation of `netN`

**Example 11.** The atoms from Example 7 are more complex than what we have just seen in `netN`. Two iterations of the learning and training functions will be needed to compute the two substitutions that are needed to unify the two atoms. And so, we call the adaptation function 3 times, until it outputs no new substitutions. See Figure 7. Conveniently, there is a command in MNNS that can set the number of passes for adaptation. Setting this parameter to 3 (or any number greater than 3), would yield the same result, but will not require re-typing; see Figure 8.

```

>> [net1,y,e,pf] = adapt(net1,1);
x12
a13
>> [net1,y,e,pf] = adapt(net1,1);
x02
a15
>> [net1,y,e,pf] = adapt(net1,1);
fx >> |

```

Figure 7: Adaptation of `net1`

```

>> net1.adaptParam.passes = 3;
>> [net1,y,e,pf] = adapt(net1,1);
x12
a13
x02
a15
fx >> |

```

Figure 8: Adaptation of `net1`, in 3 passes.

To summarize, we return to the Algorithm of Unification from Section 2, and we outline the parts of the Unification algorithm that were delegated to the function `learn_disagr`.

**Unification algorithm**[29, 30]:

1. Put  $k = 0$  and  $\theta_0 = \varepsilon$ .
2. If  $S\theta_k$  is a singleton, then stop;  $\theta_k$  is an mgu of  $S$ . Otherwise,  
find the r-disagreement set  $D_k$  of  $S\theta_k$ .  
If  $D_k$  does not exist, stop;  $S$  is not unifiable.
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.

As one can see, the iterative part is left to be done by the error-correction learning, while routine occur-check and the detection of the disagreement set

were formalised by means of the embedded functions `substitution` and `disagreement`. Function `disagreement` covers the underlined part of the item 2; and the function `substitution` covers the underlined part of the item 3. Notably, the application of the newly calculated substitution ( $\theta_{k+1} = \theta_k\{v/t\}$ ) is again done by the UNN.

The reader can find the functions `disagreement` and `substitution` in Appendices A and B, or in [35]. They do nothing more than simply adapt the underlined part of the algorithm above to the settings where we process vectors instead of lists of symbols. The two lemmas about the properties of these two functions will follow in the next section, where we introduce function symbols as well.

## 6. Processing Atoms with Function Symbols.

In this section, we extend the UNNs to process atoms containing function symbols. We postponed introducing this case because function symbols bring several technical complications. These are the reasons why introduction of function symbols needs to be treated with care:

1. When atoms are allowed to contain function symbols, their length can be different. And to be able to encode them into UNNs, we need them to have the same length.

**Example 12.** Two atoms  $P_{01}(x_{02})$  and  $P_{01}(f_{01}(x_{01}))$  from Example 3 have different length. If represented by vectors, the vectors will have the length 8 and 13 respectively. So, these two vectors could not possibly be used as weight and bias vectors in the UNN. One could complete the vector representing  $P_{01}(x_{02})$  by adding a subvector consisting of five 0s to this vector; and this will make the length match.

Note that 0 is not a meaningful code in ASCII encoding, and it is convenient to use it for completion. If we demand to see the symbolic image of 0, we receive an empty symbol: `char(0) = .` For convenience, we will denote this empty symbol by `_` throughout this section. So, we will say that we completed  $P_{01}(x_{02})$  by  $P_{01}(x_{02})\_ \_ \_ \_ \_$ .

2. We need to make sure that the vectors  $w$  and  $b$  that bear information about the first-order atoms were brought in the form that allows the change vectors to be added to  $w$  and  $b$  in a meaningful way.

**Example 13.** Consider two atoms  $P_{01}(f_{11}(x_{15}), x_{13})$  and  $P_{01}(x_{13}, x_{14})$ . Suppose they were encoded as  $w$  and  $b$  in a UNN. Suppose we simply added the

tail of zeros to the second atom:  $P_{01}(x_{13}, x_{14}) - - - -$ . Then, the change vector  $db$  will be sent to  $b$ , but the length of the subvector  $v_{f_{11}(x_{15})}$  representing  $f_{11}(x_{15})$  is larger than the length provided by  $v_{x_{13}}$ , and hence the length of the subvector  $v_D$  that encodes the difference between  $x_{13}$  and  $f_{11}(x_{15})$  will be greater than the length of  $v_{x_{13}}$ . As a result, when the change vector  $db$  is added to  $b$ ,  $v_D$  will interfere with the meaningful remainder of the atom, in our case - with “ $, x_{14}$ ” . However, precisely this part should be left intact for the next iterations of unification.

This means that completion must be made in a sensitive way, that is, it should rely on the function that computes the disagreement set, and then use the length of the terms in the disagreement set as a guide for completion. It should detect the right place for adding zeros, too.

**Example 14.** In the previous Example, the disagreement set would contain  $x_{13}$  and  $f_{11}(x_{15})$ . The difference in their length is 5, so a subvector consisting of five 0s should be added immediately after  $v_{x_{13}}$  in the vectors  $v_{P_{01}(f_{11}(x_{15}), x_{13})}$  and  $v_{P_{01}(x_{13}, x_{14})}$ . So, the completion will work as follows:  $P_{01}(f_{11}(x_{15}), x_{13} - - - - -)$ , and  $P_{01}(x_{13} - - - - -, x_{14})$ . Now, we should also make the length match and complete the second term as in Step 1:  $P_{01}(x_{13} - - - - -, x_{14}) - - - - -$ .

**3.** The last difficulty is that one completion is not enough. As terms grow through the series of substitutions, more completions will be needed, and in general, we cannot predict how many.

**Example 15.** In the previous example, we have completed  $P_{01}(f_{11}(x_{15}), x_{13})$  and  $P_{01}(x_{13}, x_{14})$ . But then, after the first substitution, the atoms will be transformed into  $P_{01}(f_{11}(x_{15}), f_{11}(x_{15}))$  and  $P_{01}(f_{11}(x_{15}), x_{14}) - - - - -$ . They again need to be completed to allow the next substitution. Namely, the latter atom will be completed as follows:  $P_{01}(f_{11}(x_{15}), x_{14} - - - - -)$ .

This means that the mechanism of completion needs to be embedded into the training and adapting functions, so that at each iteration the neural network could “complete” itself and thus enable the next iteration of the unification and learning.

So, the library [35] contains the functions `completion` applicable to vectors, and `net_complete` applicable to networks. The latter function applies the former to the whole network, and updates the layer’s size as well as  $w$  and

b. After each iteration of the adaptation, `net_complete` performs the network update. Through this process, the layer of the UNN may grow, and this resembles to the concept of the *growing neural gas* [39]. At each time step, the growth is bounded by the size of the longest term in the disagreement set, which is always finite.

**Example 16.** In this example, we consider the network `net3` that unifies  $P_{01}(f_{11}(x_{15}), x_{13})$  and  $P_{01}(x_{13}, x_{14})$ . In order to encode these two atoms as a weight and a bias, we need to make an initial completion, see Figure 9. Now, we set the layer's size to 22 - the length of the completed terms `W3C`

```
>> W3C = transpose(double('P01(f11(x15),x13)'));
>> B3C = transpose(double('P01(x13,x14)'));
>> [W3C,B3C] = completion(W3C,B3C);
>> W3C = char(transpose(W3C))

W3C =

P01(f11(x15),x13    )

>> B3C = char(transpose(B3C))

B3C =

P01(x13    ,x14)
```

Figure 9: Initial completion for `net3`

and `B3C`; and put  $w = W3C$ ,  $b = B3C$ . The rest of the work on unification will be done by function `adapt`. As in previous sections, `adapt` calls the learning functions `learn_disagr`, and depends on `disagreement` and `substitution`. Additionally, it completes the network at each iteration. The adaptation for `net3` will have 3 meaningful steps, and then it will output empty substitutions. See Figure 10.

**Example 17.** Our running example 3 will have a less tricky completion for atoms  $P_{01}(x_{02})$  and  $P_{01}(f_{01}(x_{01}))$ . See Figure 11. Also, only initial completion will be needed for unifying these atoms.

```

>> net3.iw{1,1} = W3C;
>> net3.b{1} = -B3C;
>> [net3,y,e,pf] = adapt(net3,1);
x13
f11(x15)
>> [net3,y,e,pf] = adapt(net3,1);
x14
f11(x15)
>> [net3,y,e,pf] = adapt(net3,1);
fx >> |

```

Figure 10: Adaptation of `net3`

```

>> WN = transpose(double('PO1(x02)'));
>> BN = transpose(double('PO1(f01(x01))'));
>> [WNC,BNC] = completion(WN,BN);
>> WNC = char(transpose(WNC))

WNC =

PO1(x02      )

>> BNC = char(transpose(BNC))

BNC =

PO1(f01(x01))

```

Figure 11: Initial completion for `netN`

As we have seen in Section 4 and Figure 2, `completion` can also serve as an early detector for the non-unifiable cases, e.g., when atoms are constructed from different predicates. The function `completion` is included as a part of the library in [35] and can be found in Appendix C.

We are ready to formally state the properties of functions from the library [35] and the properties of the UNNs.

**Lemma 3.** Let  $A$  and  $B$  be two first-order atoms, and  $v_A, v_B$  be vectors encoding them. The function **disagreement** computes two vectors  $x$  and  $y$  that are numerical encodings of the two terms in the r-disagreement set  $D$  of  $\{A, B\}$ , if the r-disagreement set for  $A$  and  $B$  exists. It outputs vectors  $x = [000]$  and  $y = [000]$  if  $A$  and  $B$  are equal; and it outputs “error(Not unifiable:*explanation*)” otherwise.

*Proof.* If  $A$  and  $B$  are equal, **disagreement** detects that  $v_A = v_B$ , and outputs  $x = [000]$  and  $y = [000]$ . See Appendix A. If  $A$  and  $B$  have different length, **disagreement** completes the shortest vector by adding the vector of zeros to it. After this, the error vector  $E = v_A - v_B$  can be computed.

The proof proceeds by induction on symbols in  $A$  and  $B$  that can disagree. Any two atoms in the language  $L$  start with predicates symbols  $P_{ij}$  and  $P_{kl}$ . So, the indexes of the predicates is the first possible place for differences.

If  $A$  and  $B$  are built from different predicates, **disagreement** (“Case 1”) detects that the difference between  $v_A$  and  $v_B$  is on one of the first 3 positions in the error vector, and outputs **error(’Not unifiable: predicates are different’)**.

If the predicates are the same, then, the predicate symbols will be followed by brackets, and then - by terms. The terms can disagree, and the three following cases are possible. The terms are: 1) a variable and a constant; 2) a variable and a function symbol; 3) a constant and a function symbol. The third case will not contribute to the r-disagreement set. These three cases are covered as Cases 3, 4, 5 in **disagreement**, Case 5 outputs an error.

Another possibility is that the disagreeing terms are two variables, two constants, or two function symbols with different subscripts. Only the first of the three cases will contribute to the r-disagreement set. These three possibilities are covered in Case 2 of **disagreement**; the error is sent as an output if two constants or two function symbols are detected.

All the rest disagreements will arise in case the predicates  $P_{ij}$  and  $P_{kl}$  have arities greater than 1, and all the cases we have just considered will be repeated again.  $\square$

**Lemma 4.** For two arbitrary first-order atoms  $A$  and  $B$ , and their vector encodings  $v_A$  and  $v_B$ , the function **completion** transforms  $v_A$  and  $v_B$  into two vectors  $v'_A$  and  $v'_B$  of the same length. Moreover, if the r-disagreement set for  $A$  and  $B$  exists, and consists of a variable  $x$  and a complex term  $t$ , then **completion** finds occurrences of  $v_x$  in  $v_A$  and  $v_B$ , and adds a vector of zeros of the length  $l_t - l_x$  following each occurrence of  $v_x$  in  $v_A$  and  $v_B$ .



The resulting vectors  $v'_A$  and  $v'_B$  are the shortest of all possible completions sufficient to perform the next substitution.

*Proof.* We rely on Lemma 3, and assume that, if r-disagreement set  $D = \{x, t\}$  for  $A$  and  $B$  exists, then `disagreement` finds its vector encoding for us.

Another auxiliary function that we use, `sort_vect`, is capable of detecting places in  $v_A$  and  $v_B$  where  $v_x$  appears, and it outputs a vector  $v_A^*$  of indexes (places) of such occurrences for  $v_A$ ; similarly for  $v_B$ . The function `completion` then starts a `for` loop which performs, for every  $i$  in  $v_A^*$ , the following transformation. It cuts the vector  $v_A$  into two parts - part  $Y$  contains the first  $i + 3$  elements of  $v_A$ , part  $X$  - the rest. Then it constructs  $Y$  and  $X$  back together, putting the vector of  $l_t - l_x$  zeros between them. Similar for  $v_B$ .

At the final stage, it removes all zeros at the end of  $v_A$  and  $v_B$ , and then adds zeros only to the one that is the shortest of the two.

Since we ensured that there are no excessive zeros at the end of  $v_A$  and  $v_B$ , and have added zeros only at the places detected in  $v_A^*$  and  $v_B^*$ , we have a guarantee that it is the shortest completion needed for the next substitution performed by `adapt`.  $\square$

**Lemma 5.** Let  $A$  and  $B$  be two first-order atoms, and  $v_A$  and  $v_B$  their completed vector encodings. Let  $D$  be their r-disagreement set, and let  $x$  and  $t$  be the variable and the term it contains. Let  $v_x$  and  $v_t$  be the vector encodings of  $x$  and  $t$ , with the length  $l_x$  and  $l_t$ . Let  $Z$  be a vector of zeros of the length  $l_t - l_x$ .

The function `substitution` produces two vectors:

- the vector  $dw$  that contains the subvector  $-(v_x - v_t)$  at those positions where the subvector  $[v_x; Z]$  appears in  $A$ , and
- the vector  $db$  that contains the subvector  $-(v_x - v_t)$  at those positions where the subvector  $[v_x; Z]$  appears in  $B$ .

This happens if and only if the item 3 of the Unification algorithm produces a substitution  $\theta_j$  for  $A$  and  $B$ .

*Proof.* We rely on the Lemmas 3 and 4. The item 3 of the unification algorithm performs the occurrence check. If this check fails, the function `substitution` outputs: 'Occurrence check: the variable  $x$  occurs in

the term `t`. `Substitution is blocked.`'. The rest of the proof is trivial.  $\square$

**Theorem 6.** Let  $A$  and  $B$  be two first-order atoms, and  $v_A$  and  $v_B$  be their numerical encodings. Then there exists a UNN `net` that unifies  $v_A$  and  $v_B$  and outputs the computed substitutions  $\theta_1, \dots, \theta_k$  as answers, if and only if, the algorithm of unification succeeds and outputs  $\theta_1, \dots, \theta_k$  as an answer.

*Proof.* We rely on Lemma 2 when defining the object and subobject properties of `net`, additionally, we use `completion` and Lemma 4 to ensure that vectors  $v_A$  and  $v_B$  are of the same length. For adaptation, we use adaptation function `adapt` that uses the learning function `learn_disagr` based on `substitution`.

Suppose the unification algorithm succeeded, with the output  $\theta_1, \dots, \theta_k$ . We use induction on number  $k$  of iterations of the algorithm. If  $k = 0$ , then  $A$  and  $B$  were the same, but then  $v_A = v_B$ , and the error signal consists of zeros, and so the network outputs zero error vector, so the adaptation is void.

Suppose, at step  $k$ , the unification algorithm stops and outputs substitutions  $\theta_1, \dots, \theta_k$ . This means that, at step  $k - 1$ , the variants  $A'$  and  $B'$  of  $A$  and  $B$  were obtained, and their r-disagreement set  $D$  contained a variable  $x$  and a term  $t$ , such that  $x$  did not occur in  $t$ . And  $D$  was used to obtain the substitution  $\theta_k$ , such that  $A'\theta_k = B'\theta_k$ . By inductive assumption we know that for  $k - 1$ , there exists a UNN that can adapt its weights and parameters  $w = v_A$  and  $b = -v_B$  to become  $w = v_{A'}$  and  $b = -v_{B'}$  in  $k - 1$  steps. But then, by Lemma 3, `disagreement` can find the disagreement vectors  $D1$  and  $D2$  for  $w = v_{A'}$  and  $-b = v_{B'}$ .

Also, by definition of `adapt`, at the step  $k - 1$  the network has been completed by `complete_net` using the value of  $D1$  and the length of  $D2$  as parameters. This means, in its turn, that the length of the layer of `net` will be just the same as the length of the change vector computed by `substitution` that is called by the learning function `learn_disagr`.

Moreover, by Lemma 5, we know that `substitution` produces the change vectors  $dw$  and  $db$  that soundly encode the substitution  $\theta_k$ . And the function `adapt` will add the change vectors  $dw$  and  $db$  to  $w(k - 1)$  and  $b(k - 1)$ . By Lemma 5, the only non-zero entries in  $dw$  and  $db$  are those representing  $-([v_x; Z] - v_t)$ , where  $Z$  contains a sufficient number of zeros to make the length of the two vectors  $[v_x; Z]$  and  $v_t$  match. Moreover, these non-zero entries in  $dw$  and  $db$  match the entries of  $[v_x; Z]$  in  $w$  and  $b$ . But then, these

non-zero entries of  $dw$  will have the following effect on the matching entries of  $w$ :  $[v_x; Z] + (-([v_x; Z] - v_t)) = v_t$ , similarly with  $b$ .

This way, the encoding of the substitution  $\theta$  is applied to  $v_{A'}$  and  $v_{B'}$ . At step  $k$ ,  $w(k) = v_{A'\theta_k} = -b(k) = v_{B'\theta_k}$ . But then, the network will output the zero error vector at time step  $k$ , and will not output any further substitutions.

The opposite direction of “if and only if” is a similar argument by induction on the number of the time steps taken by the UNN to reach the zero error signal.  $\square$

## 7. Conclusions

We have constructed the unification neural networks that can simulate the algorithm of first-order unification. We implemented and tested them in the MATLAB Neural Network Simulator, and described the library of functions we needed to employ for it.

This result is the first implementation of the unification algorithm in artificial neural networks, and it has several theoretical and practical implications.

From the theoretical point of view, it shows one possible way of how one of the basic and most important logic algorithms may be realised in neural networks. And so, the UNNs contribute to the line of research that seeks to develop efficient neuro-symbolic networks. Unlike most of the neuro-symbolic networks we know of, the UNNs have the benefit of being finite in the first-order case, and their size is bound by the length of terms and atoms we choose to unify. Therefore, the UNNs are highly resource-conscious.

The UNNs is the first experiment on how non-boolean networks can be employed in a logic algorithm. That is, we no longer rely on the truth-values of the first-order atoms when unifying them. And this is a step towards the proof-theoretic style of building the neuro-symbolic networks, as opposed to the traditional, model-theoretic style.

Because the unification is the basic procedure in automated reasoning, the UNNs can be further used for simulation of the SLD resolution for different kinds of logic programming, or decision procedures for various sequent calculi.

The result has a significance for cognitive science, because the UNNs demonstrate how non-ground reasoning may be implemented in neural networks. The UNNs are fully defined in the environment of MATLAB Neural Network Simulator and are ready to be implemented in robotics.

From the point of view of logic, the UNNs may prove to be useful because the parallelism of neural networks can add the efficiency in cases when we need to perform unifications massively and in parallel. Moreover, the UNNs we have described here is the first step to the major goal of tackling the second-order unification in neural networks. The outstanding computational abilities of neural networks would be especially useful in the second order (undecidable) case.

Only a slight technical improvement to the function **disagreement** is needed to extend the UNNs to the arbitrary first-order alphabet with the infinite number of symbols. Here, we have limited the number of symbols contained in the alphabet, and used first-order symbols  $x_{..}$ ,  $a_{..}$ ,  $f_{..}$ ,  $P_{..}$  only with double digit subscripts. The reason for this restriction was expository rather than theoretical or technical. In non-restricted case, instead of checking a symbol and the two digits after it, we could use an auxiliary function similar to the **find\_term** we have already employed in **disagreement**. This function would decide when the numbers representing indexes  $\bar{j}$  of the given symbol  $P_{\bar{j}}$ ,  $x_{\bar{j}}$ ,  $a_{\bar{j}}$ , or  $f_{\bar{j}}$  end, and the new term begins.

## References

- [1] G. Markus, The Algebraic Mind: Integrating Connectionism and Cognitive Science, Cambridge, MA: MIT Press, 2001.
- [2] P. Smolensky, G. Legendre, The Harmonic Mind, MIT Press, 2006.
- [3] A. d’Avila Garcez, K. B. Broda, D. M. Gabbay, Neural-Symbolic Learning Systems: Foundations and Applications, Springer-Verlag, 2002.
- [4] A. d’Avila Garcez, L. C. Lamb, D. M. Gabbay, Neural-Symbolic Cognitive Reasoning, Cognitive Technologies, Springer-Verlag, 2008.
- [5] H. W. Gsgen, S. Hlldobler, Connectionist inference systems, in: B. Fronhfer, G. Wrightson (Eds.), Parallelization in Inference Systems, Springer, LNAI 590, 1992, pp. 82–100.
- [6] S. Hlldobler, Towards a connectionist inference system, Computational Intelligence III (1991) 25–38.
- [7] S. Hlldobler, F. Kurfess, CHCL – A connectionist inference system, in: B. Fronhfer, G. Wrightson (Eds.), Parallelization in Inference Systems, Springer, LNAI 590, 1992, pp. 318 – 342.

- [8] S. Hölldobler, Y. Kalinke, Towards a massively parallel computational model for logic programming, in: Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing, ECCAI, 1994, pp. 68–77.
- [9] S. Hölldobler, Y. Kalinke, H. P. Storr, Approximating the semantics of logic programs by recurrent neural networks, *Applied Intelligence* 11 (1999) 45–58.
- [10] L. Shastri, 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) (1993) 417–494.
- [11] T. E. Lange, M. G. Dyer, High-level inferencing in a connectionist network, *Connection Science* 1 (1989) 181 – 217.
- [12] P. Hitzler, S. Hölldobler, A. K. Seda, Logic programs and connectionist networks, *Journal of Applied Logic* 2(3) (2004) 245–272.
- [13] W. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, *Bulletin of Math. Bio.* 5 (1943) 115–133.
- [14] D. Rumelhart, J. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, I & II*, MIT Press, Cambridge MA, 1986.
- [15] I. Aleksander, H. Morton, *Neurons and Symbols*, Chapman and Hall, 1984.
- [16] I. Cloete, J. M. Zurada, *Knowledge-Based Neurocomputing*, MIT Press, 2000.
- [17] H. Siegelmann, *Neural Networks and Analog Computation. Beyond the Turing Limit*, Birkhauser, 1999.
- [18] S. Bader, P. Hitzler, A. Witzel, Integrating first-order logic programs and connectionist systems — a constructive approach, in: A. S. d’Avila Garcez, J. Elman, P. Hitzler (Eds.), *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy’05*, Edinburgh, UK, 2005.

- [19] S. Bader, P. Hitzler, S. Hölldobler, 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.
- [20] L. Ding, Neural prolog - the concepts, construction and mechanism, in: Proceedings of the 3rd Int. Conference Fuzzy Logic, Neural Nets, and Soft Computing, 1995, pp. 181–192.
- [21] D. Nauck, F. Klawonn, R. Kruse, F. Klawonn, Foundations of Neuro-Fuzzy Systems, John Wiley and Sons Inc., NY, 1997.
- [22] L. Zadeh, Interpolative reasoning in fuzzy logic and neural network theory, Fuzzy Systems (1992) 1–20.
- [23] A. K. Seda, On the integration of connectionist and logic-based systems, in: T. Hurley, M. Mac an Airchinnigh, M. Schellekens, A. K. Seda, G. Strong (Eds.), Proceedings of MFCSIT2004, Trinity College Dublin, July, 2004, Vol. 161 of Electronic Notes in Theoretical Computer Science, Elsevier, 2005, pp. 109–130.
- [24] A. d’Avila Garcez, G. Zaverucha, The connectionist inductive learning and logic programming system, Applied intelligence, Special Issue on Neural networks and Structured Knowledge 11 (1) (1999) 59–77.
- [25] A. d’Avila Garcez, G. Zaverucha, L. A. de Carvalho, Logical inference and inductive learning in artificial neural networks, in: C. Hermann, F. Reine, A. Strohmaier (Eds.), Knowledge Representation in Neural Networks, Logos Verlag, Berlin, 1997, pp. 33–46.
- [26] E. Komendantskaya, Learning and deduction in neural networks and logic, Ph.D. thesis, Department of Mathematics, University College Cork, Ireland (2007).
- [27] E. Komendantskaya, M. Lane, A. Seda, Connectionist representation of multi-valued logic programs, in: B. Hammer, P. Hitzler (Eds.), Perspectives of Neural-Symbolic Integration, Computational Intelligence, Springer Verlag, 2007, pp. 259–289, to appear.

- [28] E. Komendantskaya, A. K. Seda, Logic programs with uncertainty: neural computations and automated reasoning, in: Proc. CiE'06, Swansea, Wales, 2006, pp. 170–182.
- [29] J. Herbrand, Investigations in proof theory, in: J. van Heijenoort (Ed.), From Frege to Gödel: A source book in Mathematical Logic, 1879-1931, Harvard University Press, Cambridge, Mass., 1967, pp. 525–581.
- [30] J. Robinson, A machine-oriented logic based on resolution principle, Journal of ACM 12 (1) (1965) 23–41.
- [31] R. A. Kowalski, Predicate logic as a programming language, in: Information Processing 74, Stockholm, North Holland, 1974, pp. 569–574.
- [32] E. Komendantskaya, First-order deduction in neural networks, in: Proc. 1st Int. Conf. on Language and Automata Theory and Applications (LATA'07), Tarragona, Spain, 2007, pp. 307–319.
- [33] E. Komendantskaya, Unification by error-correction, in: Proceedings of NeSy'08 workshop at ECAI'08, 21-25 July 2008, Patras, Greece, Vol. 366, CEUR Workshop Proceedings, 2008.
- [34] MathWorks, Neural network toolbox 6, [www.mathworks.com](http://www.mathworks.com) (2008).
- [35] E. Komendantskaya, Unification neural networks: Library of functions and examples written in MATLAB neural network toolbox (2009). URL [www.cs.st-andrews.ac.uk/~ek/Error-Correction.zip](http://www.cs.st-andrews.ac.uk/~ek/Error-Correction.zip)
- [36] R. Hecht-Nielsen, Neurocomputing, Addison-Wesley, 1990.
- [37] S. Haykin, Neural Networks. A Comprehensive Foundation, Macmillan College Publishing Company, 1994.
- [38] J. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer-Verlag, 1987.
- [39] B. Fritzke, Fast learning with incremental rbf networks, Neural Processing Letters 1 (1994) 1–5.

## A. Function disagreement

```
function [x,y] = disagreement(W, B)
% Finds r-disagreement set for vectors W and B representing two first-order
% atoms.
% Works with either vertical or horizontal vectors.

[m,n] = size(W);
if (~((m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('Input must be a vector')
end
[m2,n2] = size(B);
if (~((m2 == 1) || (n2 == 1)) || (m2 == 1 && n2 == 1))
    error('Input must be a vector')
end

if m==1 && n>1
    W = transpose(W);
end

if m2==1 && n2>1
    B = transpose(B);
end

% In order to compute the error vector for $W$ and $B$, we
% make the length of these vectors equal:
if m ~= m2
    if m>m2
        dm = m-m2;
        B = [B; zeros(dm,1)];
    end
    if m2>m
        dm = m2-m;
        W = [W; zeros(dm,1)];
    end
end

[m,n] = size(W);
[m2,n2] = size(B);

E = (W - B);

% Case when W = B
[rows] = find(E,1);
[m0,n0] = size(rows);
if ((m0 == 0) && (n0 == 1)) || ((n0 == 0) && (m0 == 1))
    x = [0 0 0]; y = [0 0 0];
end
```



```

else

% Case 1. Predicates are different
[rowsp] = find(B == 40, 1); % double('(') = 40
[rowsp2] = find(W == 40, 1);

if ((rows < rowsp) || (rows < rowsp2))
    error('Not unifiable: predicates are different')
end

% Case 2. Two variables, or two constants, or two function symbols.
% Any two variables x_jj and x_ii will differ only in the subscript part.
if ((48 < W(rows)< 58) && (48 < B(rows)< 58))
    if (W(rows - 1) == 120) && (B(rows - 1) == 120)
        if ((W(rows + 2) == 41) && (B(rows + 2) == 41)) ||
            ((W(rows + 2) == 44) && (B(rows + 2) == 44))
            x = [W(rows - 1), W(rows), W(rows+1)];
            y = [B(rows -1), B(rows),B(rows+1)]; % Actual computation
        else error('Unification error: Cannot infer the syntax of the term')
        end
    end
    if (W(rows - 1) == 97) && (B(rows - 1) == 97) %Two constants
        error('Not unifiable: Two different constants in the disagreement set')
    end
    if (W(rows - 1) == 102) && (B(rows - 1) == 102) %Two function symbols
        error('Not unifiable: Two different function symbols in
            the disagreement set')
    end
end
if ((48 < W(rows - 1)< 58) && (48 < B(rows - 1)< 58))
    if (W(rows - 2) == 120) && (B(rows - 2) == 120)
        x = [W(rows - 2), W(rows - 1), W(rows)];
        y = [B(rows - 2), B(rows -1), B(rows)]; % Actual computation
    end
    if (W(rows - 2) == 97) && (B(rows - 2) == 97)
        error('Not unifiable: Two different constants in
            the disagreement set')
    end
    if (W(rows - 2) == 102) && (B(rows - 2) == 102)
        error('Not unifiable: Two different function symbols in
            the disagreement set')
    end
end
end
end
end

```

```

% Case 3. A variable and a constant (2 subcases)
% double(x) = 120; double(a) = 97;
if ((W(rows) == 120) && (B(rows) == 97))
    x = [W(rows), W(rows+1), W(rows+2)];
    y=[B(rows), B(rows+1), B(rows+2)];    % Actual computation
end

if ((B(rows) == 120) && (W(rows) == 97))
    x = [B(rows), B(rows+1), B(rows+2)];
    y = [W(rows), W(rows+1), W(rows+2)];    % Actual computation
end

% Case 4. A variable and a complex term (2 subcases)
% double(x) = 120; double(f) = 102;
if ((W(rows) == 120) && (B(rows) == 102))
    if ((48 < W(rows + 1) < 58) && (48 < B(rows + 1) < 58) &&
        (48 < W(rows + 2) < 58) && (48 < B(rows + 2) < 58))
        B2 = B(rows:end);
        z = find_term(B2); %the function find_term finds the index of
        %the last bracket contained in the term starting with f...
        x = [W(rows), W(rows+1), W(rows+2)];
        y = transpose(B(rows : (rows + z-1)));    % Actual computation
    else
        error('indexes are not double-digit.')
    end
end

if ((B(rows) == 120) && (W(rows) == 102))
    if ((48 < W(rows + 1) < 58) && (48 < B(rows + 1) < 58) &&
        (48 < W(rows + 2) < 58) && (48 < B(rows + 2) < 58))
        W2 = W(rows:end);
        z = find_term(W2);
        x = [B(rows), B(rows+1), B(rows+2)];
        y = transpose(W(rows: (rows + z -1)));    % Actual computation
    else
        error('indexes are not double-digit.')
    end
end

% Case 5. A constant and a complex term.
% double(a) = 97; double(f) = 102;
if ((W(rows) == 97) && (B(rows) == 102))
    error('Not unifiable: a constant and a function symbol in
        the disagreement set')
end

```

```

if ((B(rows) == 97) && (W(rows) == 102))
error('Not unifiable: a constant and a function symbol in
      the disagreement set')
end

end

end

```

## B. Function substitution

```

function [y,z] = substitution(W, E)
%SUBSTITUTION takes the vectors W and E: W is representing a first-order atom,
% E is a difference between W and the vector that represents another
% first-order atom.
% It relies on D - the r-disagreement set given by D = disagreement(W, (W-E)),
% and outputs two vectors: (y = dw) - a substitution vector for the first
% atom, and (z = db) - a substitution vector for the second atom.
% In the network, dw will be the change vector for weights, and db will be the
% change vector for biases.
%

[m,n] = size(W);
if (~(m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('Input must be a vector')
end
[m2,n2] = size(E);
if (~(m2 == 1) || (n2 == 1)) || (m2 == 1 && n2 == 1))
    error('Input must be a vector')
end

B = W - E;

%r-disagreement set:
[D1,D2] = disgr5(W,B);
disp(char(D1));
disp(char(D2));
%D1 and D2 are horizontal vectors

[p,q] = size(D1);
[p1,q1] = size(D2);

if (q1-q)>0
    qn = q1-q;

```

```

    D1 = [D1, zeros(1,qn)];
end

% These are needed to form dw.

Z1 = zeros(m,1);
Z2 = zeros(m2,1);

if ~ (D1(1) == 0 && D1(2) == 0 && D1(3) == 0)

%Find occurrences of the variable x_ij in the r-disagreement set D1 in W and B:
INDw = sortvect(W,D1(1),D1(2),D1(3));
INDb = sortvect(B,D1(1),D1(2),D1(3));

[m3,n3] = size(INDw);
[m4,n4] = size(INDb);

if ~ isempty(sortvect(D2,D1(1),D1(2),D1(3)))
    error('Occur
           it is going to be substituted with. Substitution is blocked.')
end

D3 = -(D1 - D2);

if ((m3 > 0) && (n3 > 0))
    for s = INDw
        for l=0:(q1-1)
            Z1(s+1,:) = D3(l+1);
        end
    end
Z1;
end

if ((m4 > 0) || (n4 > 0))
    for t = INDb
        for l=0:(q1-1)
            Z2(t+1,:) = D3(l+1);
        end
    end
Z2;
end
y = Z1;
z = Z2;

else

```

```

y = Z1;
z = Z2;
end
end

```

### C. Function completion

```

function [x, y] = completion(A, B)
%Given two vectors A and B, encoding two first-order formulae,
%the function compares the vectors and completes the shortest of them to
%have the same length as the longest.
%Additionally, if v and t are the variable and the term in the
% r-disagreement set, it adds
% the subvector of zeros after each occurrence of v in A and B,
% to enable substitution for the term t.

[m,n] = size(A);
if (~(m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('Input must be a vector')
end
[m2,n2] = size(B);
if (~(m2 == 1) || (n2 == 1)) || (m2 == 1 && n2 == 1))
    error('Input must be a vector')
end

if n==1 && m>1
    A = transpose(A);
end

if n2==1 && m2>1
    B = transpose(B);
end

% Step 1. Complete variables:

% Step 1.2. Disagreement set:
[D1,D2] = disagr5(transpose(A),transpose(B));

[p,q] = size(D1);
[p1,q1] = size(D2);

q2 = q1-q;
if q2 > 0
Z = zeros(1,q2);

```

```

else Z = []
end

[ZA] = sortvect(A,D1(1),D1(2),D1(3));
% sortvect finds places where the variable D1 occurs
[ZB] = sortvect(B,D1(1),D1(2),D1(3));

% Step 1.3 Actual completion
if isempty(ZA)
    A1 = A;
else
    for s=ZA
        X = A(s+3:end);
        A(:,s+3: end) = [];
        A1 = [A, Z, X];
    end
end

if isempty(ZB)
    B1 = B;
else
    for r=ZB
        Y = B(r+3:end);
        B(:,r+3: end) = [];
        B1 = [B, Z, Y];
    end
end

%Step 2. Adding zeroes at the end of the shortest list.

[c,d] = size(A1);
[c1,d1] = size(B1);

fd = find(A1,1,'last');
fd1 = find(B1,1,'last');

% Step 2.1. First we remove excessive 0's at the ends
% of both lists, and then compare meaningful parts.

if fd ~= d
    A1 = A1(1:fd);
end

if fd1 ~= d1
    B1 = B1(1:fd1);
end

```

```

end

%Step 2.2 Add zeros to the shortest vector.
[c,d] = size(A1);
[c1,d1] = size(B1);

if d == d1
    A2 = A1;
    B2 = B1;
else
    if d>d1
        d2 = d-d1;
        A2 = A1;
        B2 = [B1, zeros(1,d2)];
    end

    if d1>d
        d3 = d1-d;
        A2 = [A1, zeros(1,d3)];
        B2 = B1;
    end
end

x = transpose(A2);
y = transpose(B2);

end

```