Using Inductive Types for Ensuring Correctness of Neuro-Symbolic Computations

Ekaterina Komendantskaya[1], Krysia Broda[2], and Artur d'Avila Garcez[3]

[1] School of Computing, University of Dundee, UK
[2] Department of Computing, Imperial College, London, UK
[3] Department of Computing, City University London, UK *

Abstract. We propose a new method for ensuring correctness of neuro-symbolic computations. We consider important examples when checking the data type of the network's inputs/outputs is crucial for ensuring that it performs correctly. We construct neuro-symbolic networks that can recognise the type of the input/output data; they are capable of recognising inductive and even dependent types.
Key words: Nature Inspired Computing, Neural Networks and Connectionist Models, Hybrid Neuro-Symbolic Systems, Type Systems and Type Theory, Artificial Intelligence.

1 Introduction

Computational logic and Neurocomputing are two different paradigms that underly numerous attempts to expand and refine the qualities and capacities of automated reasoning and artificial intelligence.

Computational logic, including type theory, higher-order calculi, interactive higher-order theorem provers, aims at representing and automating the *logical, deductive, mathematical and constructive* reasoning; and hence the main advantages achieved here are soundness of computations, correctness of typed and functional programs, also known as *correctness-by-construction*; [3, 10].

Neurocomputing aims at covering the *"illogical"' — situational, inductive and adaptive* — reasoning. The main achievements here are the networks capable of recognising images and sounds and classifying objects into classes, [6]. Some methods of Neurocomputing are applied in machine learning; for example, to handle inductive logics and logic programs [4, 5] or inductive classification of logically structured data [12, 8].

Neuro-Symbolic Integration is the area of research that endeavours to synthesize the best of the two worlds. The area was given a start by the pioneering paper of McCulloch and Pitts [9] that showed how propositional Boolean logic can be represented in neural networks; we will call these networks *Boolean networks*. The Neuro-Symbolism has since developed different approaches to inductive, probabilistic, and fuzzy logic programming; [4, 5, 14, 13]. However, it

* The work was supported by the Engineering and Physical Sciences Research Council, UK; Postdoctoral Fellow research grant EP/F044046/1.

generally follows the methodology of McCulloch and Pitts, in that logical information is given only by means of logical connectives and truth values; as opposed to encoding the higher-order syntax and logical structure of sentences in neural networks, [7, 11].

In this paper, we stretch the fundamental quest for correctness of computations from the area of Computational Logic to the area of Neuro-Symbolic computation. In particular, we examine the question: How reliable and secure the existing Neuro-Symbolic networks are from the logical point of view? And how can they be made secure? Answers to these questions are decisive for the future implementation of neuro-symbolic networks in computing.

Consider the following example. Figure 1 shows the Boolean network of McCulloch and Pitts that computes boolean function $(x \vee y) \wedge \neg z$. For the networks built in this style, we have no tools to check whether the network performs correctly. In particular, such network would not be able to distinguish "logical" data (values 0 and 1) from any other type of data, and would output the same result both for sound inputs like $x := 1, y := 1, z := 0$, and for non-logical values such as $x := -100.555, y := 200.3333\ldots, z := 0$. Imagine the situation when a user monitors the outputs of a big network, and sees outputs 1, standing for "true", whereas in reality the network is receiving some uncontrolled, excessive or noisy, data. Therefore, such computations do not guarantee the "correctness" of computations. The network will perform correctly only if someone has already tested the input data and insured that it is of the type `bool`. Most of existing neuro-symbolic networks are vulnerable in this respect.

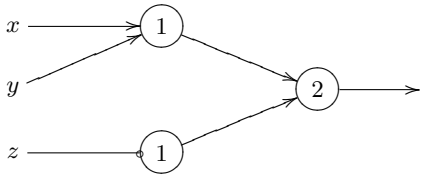

Fig. 1. The traditional McCulloch and Pitts network computing $(x \vee y) \wedge \neg z$. The numbers $1, 2$ are the thresholds, and negation is formalised by inhibitory link $-\circ$.

A different example that indicates the same problem comes from inductive logic. Reasoning by induction involves making general conclusions from particular examples. E.g., one can generalise from "This dog has four legs, and hence it can run" to the statement "Everything that has four legs can run". However, intuitively, this generalisation does not sound convincing: we know that there are some objects, such as chairs, that have four legs but do not move. And hence, both in common reasoning and science, it is usual to use some kind of implicit typing, like "All animals that have four legs can run". This typing does not guarantee the correctness of inductive generalisations, but it provides an essential preliminary check for correctness. In [5], we considered inductive relational learning in neuro-symbolic networks, by taking relations *father, mother,*

grandparent as particular examples. However, our attempts to generalise this approach have shown that the implicit typing has to be explicitly present in the neuro-symbolic networks; or otherwise we sacrifice correctness.

In this paper, we propose neuro-symbolic networks that can check types of the expressions given in a functional language. Such networks can be used to perform the initial type recognition, e.g, for the first example above. They can also be used as an integral part of the neuro-symbolic networks that perform inductive learning; e.g., as in the second example above.

The paper is organised as follows. Section 2 contains background definitions. In Section 3, we define symbol recognisers in neural networks. In Section 4, we show how the recursive recognisers can be used to recognise expressions of inductive and dependent types. In Section 5 we conclude the paper.

2 Preliminaries

2.1 Types

The formal language we use as the "symbolic" part of the neuro-symbolic system is a higher-order typed language, *dependently-typed λ-calculus*. Moreover, we use the functional programming syntax throughout; as e.g. in [2, 10]. The higher-order approach is relatively new to Neuro-Symbolic systems: the leading neuro-symbolic networks are based on propositional [4, 5, 14] or first-order logics [1, 13]. The only functional approach we are aware of uses kernel methods: [12, 8].

Although this paper may be seen as a first and initial step to define a neuro-symbolic specification language for a dependently-typed λ-calculus, this ambitious goal lies far beyond the scope of this paper. Instead, we will stay very close to the motivating examples we gave in the introduction; and will simply consider the method of recognising, using the machinery of neural networks, whether a given expression is of a given inductive data type.

We will therefore skip the definitions of terms and expressions that conventionally open the description of dependently typed languages; we take the base library of Coq, the reader can consult e.g., [2, 10]. We will instead proceed straight to the definitions of inductive data types given in functional programming style.

Every expression in the language we use is assumed to be typed. The functional language provides a variety of *base types* - sets of simple, unstructured values such as numbers, booleans and characters. Among them we will distinguish the *atomic types* and *data types*. *Atomic types* have no internal structure as far as the type system is concerned. Examples of atomic types are `Prop` - the type of propositions; `Set` - the type of sets; and `Type`.

The inductive definitions of *data types* always follow one syntactic pattern. The word `Inductive` is a declaration that an inductive type is being defined; it is followed by the name of the type, for example, `bool`; and by one or more terms, also called constructors, $c_1, \ldots c_n$, (cf. `t` and `f` in Example 1) with the typing relation given by $c_i : T$ that assigns type T to the term c_i.

Example 1. `Inductive bool : Type := | t : bool  | f : bool.`

One can imagine defining other types in the manner similar to `bool`; for example, one could define the days of the week or the months of the year by simply listing them. If inductive definition of a type does not involve recursion, we will call this inductive type *primitive*.

Consider an *inductive type* of natural numbers `nat`. Unlike the inductive type `bool`, it involves recursion. We will call such data types *recursive*. Using the syntax below, number 3 will be written as `SSSO`.

Example 2. `Inductive nat : Set := | O : nat  | S : nat -> nat.`

Constructors used in an inductive definition provide a method for generating elements of the inductive type. More importantly, they give a general way of specifying and recognising a well-formed term of this type. No matter how big or complex the term of a type `T` is, the computations will always follow the pattern given by its constructors.

The last distinction we need to make is between *simple* and *dependent* types. All the types we have considered so far were simple, in that their definitions did not depend on other types. Consider the example of the dependent type of lists of elements of a type `nat`. The definition of this type not only involves recursion, but it is also *dependent* on another type - `nat`.

Example 3.
```
Inductive list : Set :=
  | nil : list   | cons : nat -> list -> list.
```

Most programming languages provide a variety of ways of building *compound data structures*. Types can be composed using `->`: e.g, `Set -> Prop` is a type of predicate symbols. The product type can be defined as follows:

```
Inductive prod (A B:Set) : Set :=
  pair : A -> B -> prod A B.
```

Then the set `prod A B` is a cartesian product `A` $\times$ `B` of sets `A` and `B`. All the types we consider in this paper are zero-order – or quantifier free – types.

2.2 Neural networks

An **artificial neural network** [6] is a directed graph with nodes (called *units* or *neurons*), and edges (called *connections*). If there is a connection from unit j to unit k, then w_{kj} denotes the *weight* associated with this connection, and $i_{kj}(t) = w_{kj}v_j(t)$ is the *input* received by k from j at time t. A neuron k is characterised, at time t, by its activation *input vector* $(v_{i_1}(t), \ldots v_{i_n}(t))$, its input *potential* $p_k(t)$, its *bias* b_k and its *value* $v_k(t)$. In each update, the potential and value of a unit are computed with respect to an *input* function and a *transfer* function, 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, $v_k(t + \Delta t) = F(p_k(t))$.

3 Type Recognition by Symbol Recognition in Networks

In this section, we show how neural symbol-recognisers can handle recognition of primitive inductive types. We define type recognisers as follows.

Definition 1. *Given a definition of an inductive data type* `T`*, we say that a neural network N recognises* `T`*, or is a* T-recogniser*, if, for any given well-formed expression E, the following holds. If a numerical encoding of E is sent to N as an input, then there exist unique vectors v_s and v_f, called* success *and* failure vectors*, such that N outputs v_s if and only if E is of type* `T`*, and it outputs v_f if and only if E is not of type T.*

We start with explaining how the problem of data recognition can be solved using traditional methods of neurocomputing. Figure 2 shows two neural networks that can recognise zero and non-zero input data. That is, if one of such networks outputs 1, the external recipient can read this as a confirmation that the input data was of the desired format. These two networks show a "naive", but conventional, approach to "type" recognition in neural networks. However, this method is not systematic, and depends on the fact that the properties yield direct numerical checks in the neural networks. This method is not extendable to the types that describe "symbolic", rather than numerical, data.

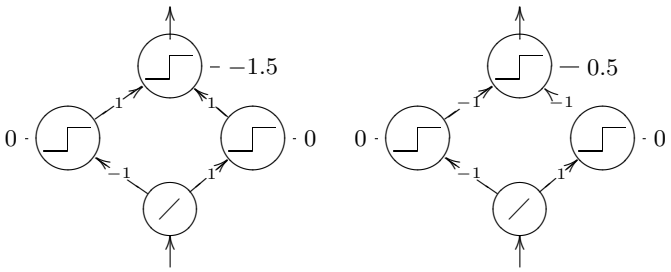

Fig. 2. "Zero" and "non-zero" recognisers. Inside of the units, we show the graphs of the transfer functions — linear and hard-limit (or threshold) functions; see [6].

This is why we propose a systematic way of representing types in neural networks, irrespective of the nature of the objects they describe. The useful feature of the types that we rely on here, is that a definition of a type gives a general skeleton for computations, and this skeleton can be used to deal with all elements/members of this type, irrespective of their complexity.

We assume that the logical syntax has a suitable numerical encoding; cf. [7].

Definition 2. *The* symbol recogniser *for a given symbol 's', also called an s -* recogniser*, is a neural network consisting of a single neuron, defined as follows. Take the numerical encoding n_s of the symbol s. The input weight is set to* 1*,*

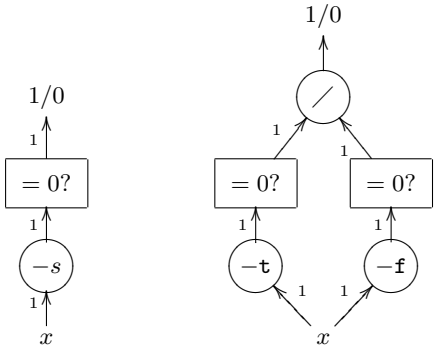

Fig. 3. Left: Recognising symbol s. The input is sent to the neuron with bias $-s$; it outputs 0 if the input matches s and some non-zero value otherwise. This neuron may be connected to a zero-recogniser (in the square box) that outputs 1 whenever the signal 0 is computed. **Right:** Recognising type `bool` with constructors `t` and `f`.

the neuron's transfer function is linear (identity), and the neuron's bias is set to $-n_s$.

Because the bias is $-n_s$, the input weight is 1, and the transfer function is the identity, the output will be equal to $i - n_s$, where i is the input value. If $i = n_s$ is sent as an input, the output will be equal to 0. This value can then be sent to the 0-recogniser network from Figure 2, as shown in Figure 3, in case we want output value "1", rather than "0", to signify "success".

Note that this simple architecture will suffice for recognising primitive inductive data types. The right-hand side of Figure 3 shows the network that can recognise type `bool` given by the two constructors `t` and `f`.

Lemma 1. *Given a definition* `Inductive X : Type := | c1 : X... | cn : X.` *of a primitive inductive data type* `X`*, there exists an* `X`*-recogniser for* `X`*.*

Construction 1 *The network* N_X *is built using* n *symbol recognisers, each recognising* `c1`$, \ldots,$ `cn`*, with built-in networks working as zero-recognisers, as shown in Figure 3. Moreover, the output signals from each of these* `ci`*-recognisers are collected in the upper neuron, as shown in Figure 3. The upper neuron computes its potential as defined in Subsection 2.2, and has a linear activation function. The output success vector* v_s *is* $[1]$*; and the failure vector* v_f *is* $[0]$.

The networks we have described would be applicable when one works with finite sets (e.g, finitely many truth values or days of the week). For some purposes of inductive and relational reasoning, such finite sets may well suffice; [5]. However, we will extend this to potentially infinite structures; and in the next section, we will use the symbol recognisers to identify "base cases" of inductive definitions.

4 Recognition of Recursive and Dependent Types

For inductive types that require recursion, we propose to use the recursive connections in neural networks. We will first consider the inductive definitions of simple inductive types; our running example is `nat`.

Definition 3. *Given a definition of a simple inductive type X, and its constructor `C` of type $X \to X$, the* recursive recogniser *for `C` is a one layer network, consisting of $n > 1$ neurons with the following properties. The length n of the layer is the length of the input vector that the network will process. Each neuron has one input connection, with the weight* 1*. The biases of all but the first neuron are set to* 0*; the bias of the first neuron is set to $-n_C$, where n_C is a numerical representation of C. The first neuron has an* output connection *that can be received by an external user. The outputs of the* $2nd - nth$ *neurons, called* recursive outputs*, are connected to the same layer, as follows: the output connection of the kth neuron ($k \in 2, \ldots, n$) is sent as an input to the $k-1$ neuron.*

Note that the first neuron of such network is the symbol recogniser for the constructor `C`. The other $n-1$ neurons in the layer are designed to recursively process the remaining $n-1$ elements of the input vector. That is, for processing the number 3, written as `SSS0`, one will need a layer of four neurons to build the recursive recogniser; see Figure 4. The recursive connections are set in such a way, that, once the numerical signal standing for `SSS0` is sent as an input to this layer, the network will process each of the symbols recursively.

We assume here that the type recognisers are built for the purposes of checking whether a given expression of specified length belongs to a given type. In case one wishes to apply the same method to some data of unspecified size - e.g. streams - one can remove the neurons $2 - n$ from the recursive layer of length n, and use the symbol recognisers instead of recursive recognisers, while sending the input stream not as a vector, but in element-by-element fashion.

Lemma 2. *Given a definition `Inductive X : Type := | c1 : X... |ck: X | ck+1 : X -> X... | cn : X -> X.` of a simple recursive inductive data type `X`, there exists an `X`-recogniser for `X`.*

Construction 2 *For every `ci`, $i \in \{1 \ldots k\}$, build a symbol recogniser, as in Definition 2; and join all such symbol recognisers into one network, as shown in Construction 1. For every `ci`, $i \in \{k+1 \ldots n\}$, build a recursive recogniser, as in Definition 3; and join all the recursive recognisers by sending their outputs to one neuron with linear activation function; similarly to Construction 1. Thus, the network has two outputs — O_B and O_I — read by the external user; they come from two neurons: one neuron collects signals from base cases recognisers, and another — from inductive case recognisers. The numerical vector encoding the expression E that we need to check for being of type `X` will be sent as an input to each of the subnetworks, as shown in Figure 4. The success vector v_s is* $[1; 0]$*, the failure vector v_f is* $[0; 0]$*; while computing, the network must output* $[0; 1]$*.*

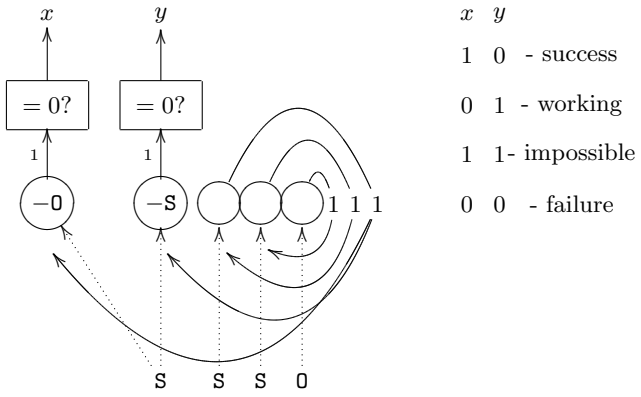

Fig. 4. This network decides whether an expression `S(S(S(O)))` is of type `nat`. Such network receives an input vector i, given by numerical encoding of the term - `S(S(S(O)))`. The dotted arrows show the initial input to the network; the solid arrows show the connections with weight 1. The network has two components: `O`-recogniser, and recursive `S`-recogniser: the recursive outputs from neurons in the `S`-recogniser layer are sent to the same layer. The success vector is $[1; 0]$, signifying that the first symbol in the input vector is `'O'`. In case the first symbol is neither `O` nor `S`, the failure output will be $[0; 0]$. The output $[0; 1]$ signifies that the expression is being processed recursively.

Note that in Lemma 2 and Construction 2, we used only inductive constructors of type `X -> X`, but types can be composed, and in case we need to give an account to compound types, we build up more layers, and connect them recursively, as we will illustrate in Figure 5.

Next, we extend these results to dependent types.

Theorem 1. *For any zero-order inductive data type `X`, there exists an `X`-recogniser.*

Construction 3 *We use Lemmas and Constructions 1 and 2. Additionally, for dependent types like `list nat` (cf. Example 3), that is, when a type `X1` being defined depends on a type `X2`, the types should be build in a cascade, as follows. The input vector is first sent to the `X1`-recogniser. The `X1`-recogniser is connected to the `X2`-recogniser at each time step, as shown in Figure 5. Each of `X1`- and `X2`- recognisers has two outputs - one for base cases (O_B) and the other for inductive cases (O_I) of inductive definitions, as described in Construction 2. The `X2`-recogniser has two sets of recursive output connections (cf. Figure 5): one set of recursive connections sends its output back to the `X2`-recogniser, and it is active as long as the recursive recognition of `X2` continues, that is, while `X2`$_{O_B}$ outputs 0, and `X2`$_{O_I}$ outputs 1. The second set of recursive output connections goes to the `X1`-recogniser; these connections become active as soon as the recursive*

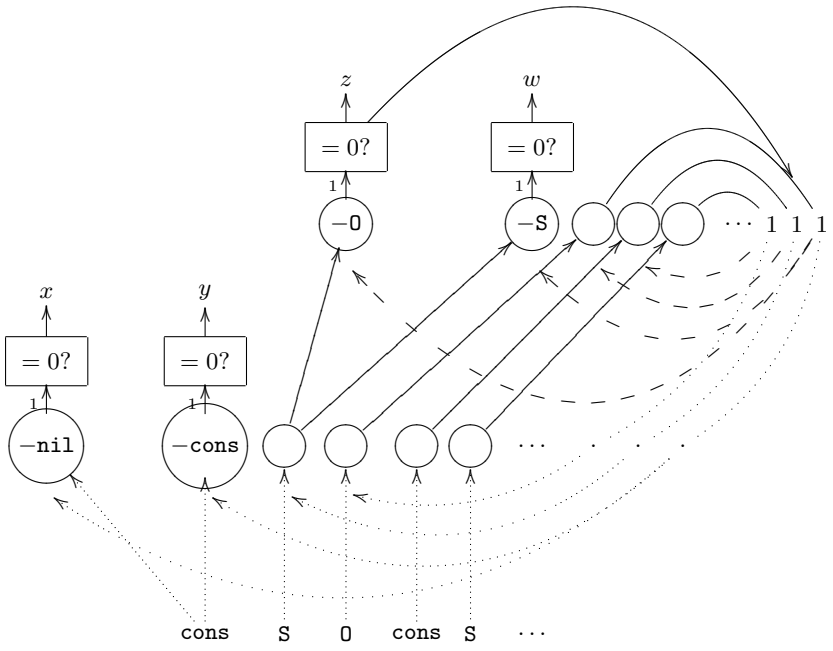

Fig. 5. This network structure can recognise a list of natural numbers. This is more complex since it has to be able to recognise a list structured input, as well as check the individual elements are of the type `nat`. It therefore consists of two recursive recognisers. At time step 1, either `nil` is recognised, and computation stops; or `cons` is recognised, in this case the rest of the input vector is transferred to the `nat` recogniser, which recognises the second element of the input vector at time step 2. Then the `nat` recogniser works using recursive output connections $--\!>$, until `0` is recognised, and then switches to the recursive output connections $\ldots\!>$ that activate the `list`-recogniser again; and the process repeats. If at some time t, there is a symbol in the input vector that does not have the form of the constructors, the network comes to the error state $[0;0;0;0]$.

recognition of `X2` *is finished and* $\texttt{X2}_{O_B}$ *outputs* 1. *For* $\texttt{X1}_{O_B}$, $\texttt{X1}_{O_I}$, $\texttt{X2}_{O_B}$, $\texttt{X2}_{O_I}$, *the success and failure vectors are* $v_s = [1;0;0;0]$ *and* $v_f = [0;0;0;0]$. *One of the three output vectors is acceptable as indicating "computing in progress":* $[0;1;0;0], [0;0;0;1], [0;0;1;0]$. *If the type being defined depends on more than one type, or constructors take more than one argument of type* `X2`, *the type recognisers are connected according to the dependencies.*

5 Conclusions and Future Work

We have proposed a general neural architecture that can recognise and check the type of a given expression written in a functional language. To the best of our knowledge, this is the first paper to show how inductive data types can be

recognised using the machinery of neural networks; and used to insure correctness of neuro-symbolic computations. We have used symbol recognisers to deal with the base cases of inductive definitions, recurrent connections in the neural networks to implement recursive computation, and cascading of neural layers to implement dependency. Composition of types can nicely be reflected by composition of layers in networks. Given such building blocks, one is free to define new type recognisers, and use them for training or inductive learning.

We believe that the typed networks may offer a good initial architecture for network learning, as done normally by the use of background knowledge (or inductive bias) in symbolic machine learning. We also believe that the proposed model can serve as a cognitive model of massively-parallel symbolic computation, especially useful when one considers such data types as trees.

Further developments of the networks we have proposed will include the study of first-order or higher-order types and practical experiments on both reasoning and learning capabilities.

References

1. S. Bader, P. Hitzler, and S. Hölldobler. Connectionist model generation: A first-order approach. *Neurocomputing*, 71:2420–2432, 2008.
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Constructions*. Springer-Verlag, 2004.
3. R. Constable and M. Bickford. Formal foundations of computer security. *NATO Science for Peace and Security Series, D: Information and Communication Security*, 14:29 – 52, 2008.
4. A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag, 2002.
5. A. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Cognitive Technologies. Springer-Verlag, 2008.
6. S. Haykin. *Neural Networks. A Comprehensive Foundation*. Macmillan College Publishing Company, 1994.
7. E. Komendantskaya. Unification neural networks: Unification by error-correction learning. *Journal of Algorithms in Cognition, Informatics, and Logic (in print)*, 2009.
8. J. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Springer, Cognitive Technologies Series, 2003.
9. W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Math. Bio.*, 5:115–133, 1943.
10. B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
11. P. Smolensky and G. Legendre. *The Harmonic Mind*. MIT Press, 2006.
12. J. L. Thomas Gartner and P. Flach. Kernels and distances for structured data. *Machine Learning*, 3(57):205–232, 2004.
13. J. Wang and P. Domingos. Hybrid markov logic networks. In *AAAI*, pages 1106–1111, 2008.
14. L. Zadeh. Interpolative reasoning in fuzzy logic and neural network theory. *Fuzzy Systems*, pages 1–20, 1992.