UNIVERSITY OF CASTILLA-LA MANCHA

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DEGREE IN COMPUTER SCIENCE

MAJORING IN COMPUTING

UNDERGRADUATE DISSERTATION PROJECT

DEVELOPING FUZZY NEURAL NETWORKS WITH THE FLOPER ENVIRONMENT

Jesús Pérez Martínez

July, 2019

UNIVERSITY OF CASTILLA-LA MANCHA

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DEGREE IN COMPUTER SCIENCE

MAJORING IN COMPUTING

UNDERGRADUATE DISSERTATION PROJECT

DEVELOPING FUZZY NEURAL NETWORKS WITH THE FLOPER ENVIRONMENT

Author:	Jesús Pérez Martínez
Supervisors:	Ginés Damián Moreno Valverde
	José Antonio Riaza Valverde

July, 2019

Authorship Statement

I, the undersigned, Jesús Pérez Martínez, with National Identification Number ******** declare that I am the sole author of the Undergraduate Dissertation titled DEVELOPING FUZZY NEURAL NETWORKS WITH THE FLOPER ENVIRONMENT and that this work does not break the current intellectual property law and that all the non-original material contained in this work is properly attributed to their legitimate authors.

Albacete, 29/06/2018

Signed by: Jesús Pérez Martínez

Abstract

The "*Fuzzy LOgic Programming Environment for Research*" built in the DEC-TAU research group of the UCLM is intended for developing flexible applications (coded with the fuzzy loic languages MALP and FASILL) and supporting approximated reasoning and uncertain knowledge in the fields of AI, machine learning, symbolic computation, soft-computing, semantic web, and so on. The tool, which is able to directly translate fuzzy programs into standard Prolog code, currently offers running/debugging/tracing capabilities with close connections to other sophisticated manipulation techniques for program optimization, thresholded tabulation, unfolding, tuning, etc.

On the other hand, when specifying a MALP program, it might sometimes be difficult to assign weights (truth degrees) to program rules, as well as to determine the right connectives. This is a common problem with fuzzy control system design, where some trial-and-error is often necessary. Unfortunately, this is a tedious and time consuming operation, and actually, it might be impractical when the program should correctly model a large number of test cases provided by the user. In order to overcome this drawback, in the DEC-TAU group it has been recently designed a symbolic extension of MALP programs called symbolic multi-adjoint logic programming (sMALP). Here, we can write rules containing symbolic truth degrees and symbolic connectives, i.e., connectives which are not defined on its associated lattice. sMALP programs can be used to tune a program w.r.t. a given set of test cases, thus easing what is considered the most difficult part of the process: the specification of the right weights and connectives for each rule. The online versión of this tool has been used in this work.

In this Final Year Project, we are concerned with a new tool that integrates artificial neural networks and fuzzy logic programming to create symbolic neural networks and tune it in order to get a more accurate model. This tool, called Neuro-FLOPER, allows the user to translate neural networks (modeled by Keras) into MALP fuzzy logic programs. Then, the user can define some parts of the fuzzy program modeling the network as symbolic constants and aggregators (thus obtaining a sMALP program) which are equivalents to weights and activation functions of the network, respectively. With these ingredients, together with a a set of test cases provided by the users, Neuro-FLOPER can perform a tuning process for finding the best substitution of these parameters with a big number of options that better satisfy the user expectations.

Acknowledgements

I would like to thank the support and collaboration of the people who made this work possible, especially:

- To Ginés Moreno and Jose Antonio Riaza, for allowing me to carry out this project and help me at all times throughout the process. Thanks for your closeness and kindness.
- To my parents and my sister Laura, for always supporting and encouraging me to dedicate myself to what I really like.
- To Ana Rosa, for her inestimable love and understanding. Thank you for always encouraging me to go one step further.

Contents

INDEX OF FIGURES

INDEX OF CHARTS

Chapter 1

INTRODUCTION

1.1 Motivation

The FLOPER fuzzy logic programming environment has been developed at the University of Castilla-La Mancha from six other end-of-degree projects that precede the one we propose now. In its current state, the tool is capable of executing and debugging fuzzy programs (by pre-processing them into standard Prolog code) on any Prolog platform, and is ready to incorporate various techniques for transformation, optimization, specialization, among others, designed by DEC-TAU research group in recent years. In addition, the system incorporates a graphical environment written in Java that allows interacting with the system in an agile and flexible way on standardized platforms (such as Windows, Linux and even Android). It is also possible to run its latest online through `http://dectau.uclm.es/floper/?q=sim/test`.

The *symbolic multi-adjoint logic programming* (sMALP) is a symbolic extension of a well-known fuzzy logic language MALP, capable of to assing truth degrees to program rules, as well as to determine the right connectives [28]. An artificial neural network defined as a fuzzy logic program can be easily modeled with sMALP. The translation performed define that each layer corresponds to a rule, operations and activation functions with connectives, and the result of a test case with truth degrees.

Tuning in fuzzy logic programs corresponds to a set of algorithms that generate a symbolic substitution with a sMALP program and a set of test cases. For this project, the algorithm chosen is thresholded tuning, which determines when a partial solution is acceptable from a set of symbolic substitutions that are computed along the execution of the program [29].

Keras is a high-level, open source neural network library written in Python. It is designed to enable fast experimentation with deep neural networks, and to be modular, extensible and user-friendly [7]. This last characteristic is due to the capability of running on top of Tensorflow, Theano, MXNet or Microsoft Cognitive Toolkit. It was developed as part of project ONEIROS (*Open-ended Neuro-Electronic Intelligent Robot Operating System*).

NeuroProlog is a reasoning system developed in 1990 and which integrates a Prolog core[1] and SunNet neural networks simulator [11]. The main objective for this system is to exploit symbolic reasoning of Prolog, and predictive skills of neural networks, capable of managing inaccurate and incomplete information. This tool was built with the available technology at that time, so obviously it has been widely surpassed in these almost forty years within the dynamic world of neural networks.

Starting from the basic idea of NeuroProlog, this Final Degree Project aims to create a new system that allows using neural networks created and trained with Keras, to translate them to FLOPER in order to use fuzzy tuning techniques to optimize determined parts of the network based on a series of test cases.

1.2 Objectives

Throughout this project, there are three main objectives that have been developed. These objectives are concerned about creating a translation, carry on the tuning process over it, and finally integrate this new part with the existing FLOPER tools.
First, as we have said, is essential to create a transformation system that allows the user to translate, according to a set of limitations, a Keras neural network model into a fuzzy logic program. This first objective can be subdivided into four more concrete sub-objectives.

- **Constraints for Keras models**: defining in Keras some rules and limitations to create a specific framework. For example, activation functions with learning capabilities are not allowed.

- **Syntactic structure using a fuzzy logic language**: defining a syntactical structure for neural networks in a fuzzy logic program.

[1]Prolog interpreter C-Prolog

- **Intermediate structures**: defining intermediate structures to adapt a neural network from Keras to a fuzzy logic program.
- **Activation functions for fuzzy logic programs**: defining some of the most common activation functions used in neural networks in order to represent them as a complete lattice, usable by the FLOPER environment.

The second main objective wants to cover all the tuning process, studying different techniques of tuning that can be performed over a neural network converted into a fuzzy logic program. Also, is interesting to study the results and performance that the system can get, and highlight some useful technical considerations.

Finally, the last objective covers the integration of this new tool with the FLOPER environment. To get this fact, in this project is going to be created a new interface for allowing the user to transform Keras models and databases as .csv files, into fuzzy logic programs and .test files correspondingly.

1.3 Structure of the document

The content in this Final Year Project is divided into five important chapters. As this is a multidisciplinary project, is fundamental to start explaining the theoretical basis of artificial neural networks, logic programming and fuzzy logic (Chapter 2). Knowing this and some tools used nowadays in this fields, as for example *Keras*, the document continues explaining the main core of this project corresponding to the part of fuzzy logic, that is the tuning of fuzzy logic programs. Along the Chapter 3 are explained in detail various concepts, that are essential in order to understand the practical approach that will be given by our new system Neuro-FLOPER in the next chapters.

Then, in Chapter 4, a new concept is introduced, the translation of neural networks as graphs into fuzzy logic programs. This translation is made by Neuro-FLOPER, and for this version is necessary to take into account some limitations. Chapter 5 have a set of experiments that allow us to analyze experimental results of this new system, as for example validation of the model and tuning different parameters. Finally, in Chapter 6 is presented the final conclusion of this work, and also future works that could be created in the future taking this Final Year Project as a reference.

At the end of the document, inside the different Appendixes, there are useful documentation that is attached to this project. These Appendixes contain programs in Python and MALP, and information regarding the content of the CD.

Chapter 2

BACKGROUND AND CURRENT STATUS

2.1 Artificial Neural Networks

An artificial neural network is defined as a computational model based on the structure and functions of the biological neural networks.

The three key concepts that artificial neural systems want to emulate from the biological ones are distributed memory, computational parallelism and adaptability [23].

- **Distributed memory**: in a computer, information is stored in well-defined positions inside memory, but in an artificial neural network the information is stored along the network. If a neuron is damaged, only a small part of the information is lost. Furthermore, biological neural systems contain redundant information, so this kind of systems are fault tolerant.

- **Computational parallelism**: this is an essential point, and the base of some libraries as, for example, *Keras*. To ensure the computational efficiency when one of these models is executed, is necessary to parallelize its calculations. For example, a sequential execution of a small image of 299x299 pixels might take some seconds to recognize an element inside of it. But, if the execution is parallelized with a GPU, the same action might take only 100 milliseconds.

- **Adaptability**: the artificial neural systems have the capability of adapting to the environment modifying their synapses and learning from the experience. In the field of artificial neural networks, this property is known as **generalization based on examples**.

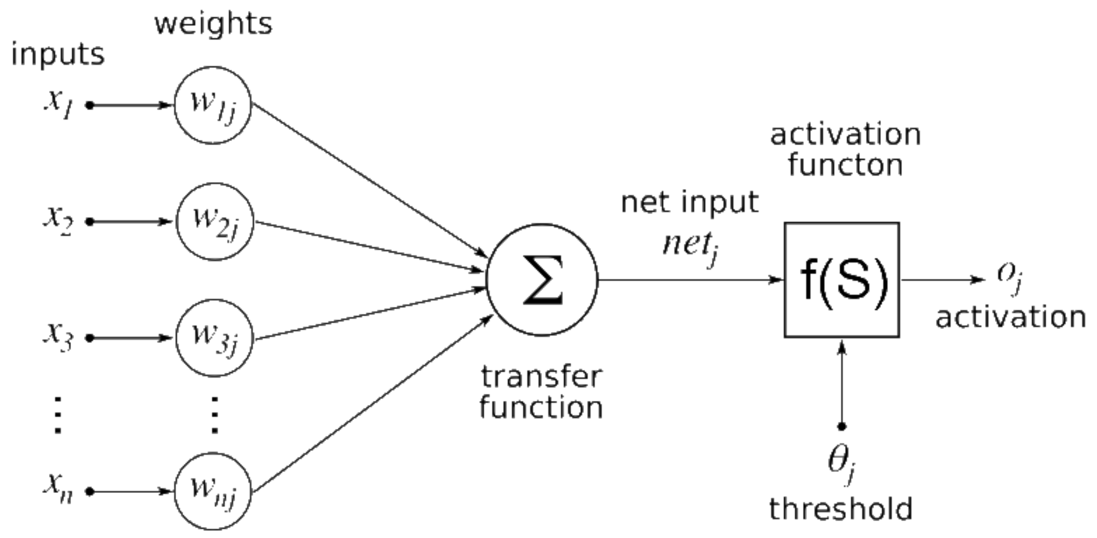


Figure 2.1: Representation of an artificial neuron.

In this section is going to be explained the general structure of an artificial neural network, but first it is necessary to introduce the concept of artificial neuron established by the PDP group in [34].

An **elemental processor or neuron** is a simple computational model that, from an input vector provided by other neurons or the outside, provides a unique output as seen in Figure 2.1. It is composed of the following elements:

1. Set of **inputs**.
2. **Synaptic weights** of the neuron, that represent the postsynaptic power value of a biological neuron.
3. **Activation function**, that provides an actual state of activation based on the previous state and the postsynaptic power value. One of the most-known is the sigmoid function, but nowadays is easy to find more complex functions that learn with other parts of the network.
4. **Output function**, that gives an output in base of the activation state.

Being defined the basic elements of an artificial neuron, we can define a set of these elements, ordered in a determined structure as a **graph**, in which a node corresponds to a neuron and an edge to a connection between neurons. An artificial neural network is a graph [30] with the following properties:

1. For each node i, a state variable x_i is defined.
2. For each connection i,j, of nodes i and j, a weight $w_{ij} \in \mathbb{R}$ is defined.

3. For each node i, a threshold θ_i is defined.

4. For each node i, a function $f_i(x_j, w_{ij}, \theta_i)$ that depends on the weights, the threshold and the states of nodes j connected to it, is defined. This function provides a new state for the node.

2.1.1 The Multi-layer Perceptron

This architecture is one of the most representative exponents of artificial neural networks due to its simplicity and power. Usually, this kind of computational model is trained with the *backpropagation algorithm*, or one of its variants. For this reason, the combination of *Multi-Layer Perceptron + Backpropagation algorithm* is known as **Backpropagation network** or **BP** [23].

The work of the PDP group popularized the Multi-Layer Perceptron with Backpropagation (see [35, 33]) from 1986. They presented a useful tool for solving complex problems, a thing that raised the interest of the scientific community in the field of neurocomputation.

The structure of a Multi-Layer Perceptron (or MLP) is represented in Figure 2.2. We will designate x_i for the inputs of the network, x_j for the outputs of the hidden layer, and z_k for the outputs of the network; t_k will be the output targets. On the other hand, w_{ij} are the weights of the hidden layer and θ_j their thresholds, w'_{kj} the weight of the output layer and θ'_k their correspondent thresholds.

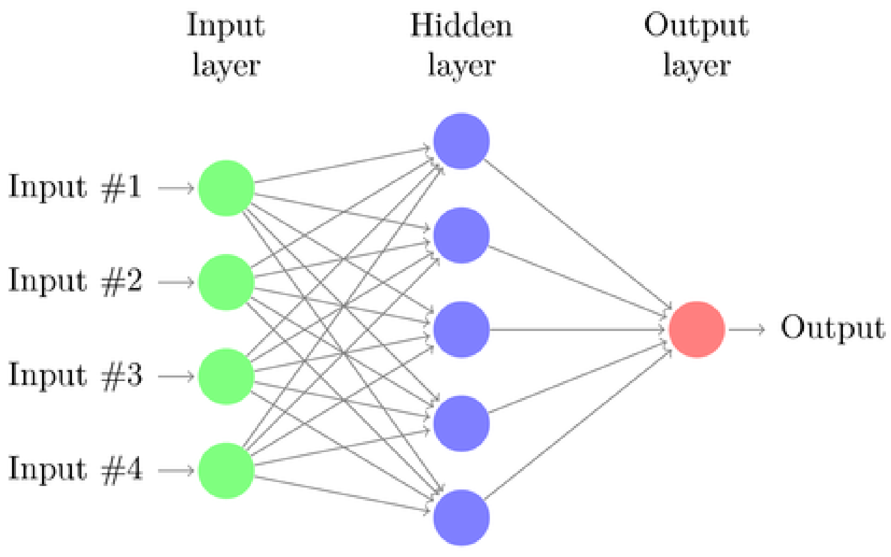


Figure 2.2: Graphical representation of a Multi-Layer Perceptron.

The function of a MLP with a hidden layer and lineal output neurons (a universal function estimator [33]) is defined as:

$$z_k = \sum_j w'_{kj} y_j - \theta'_i = \sum_j w'_{kj} f(\sum_i w_{ji} x_i - \theta_j) - \theta'_i$$

Being $f(x)$ of sigmoid type, as seen in the following examples:

$$f(x) = \frac{1}{1+e^{-x}} \quad (I) \qquad\qquad f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = tanh(x) \quad (II)$$

This is the most common architecture for a MLP, and gives the output in the interval $[0, +1]$ for (I), and in $[-1, +1]$ for (II). In this Final Year Project we will use the first one.

The development of the MLP during the last fourty years has shown that experimentally it is able to represent complex *mappings* and approach huge classification problems in an effective and simple way. The theoretical demonstration of this computational capabilities was defined almost at the same time with two very similar theorems in [10] and [9]. As an example, we will enunciate the theorem corresponding to K. I. Funahashi.

Theorem 1 ([9])**.** Being $f(x)$ a non constant function, bounded and monotonously increasing. Being K a compact subset (bounded and closed) of $\mathbb{R}^n$. Being a real number $\xi \in \mathbb{R}$, and an integer $k \in Z$ such that $k \geq 3$. With this conditions, every *mapping* **g:x** $\in K \rightarrow (g_1(\mathbf{x}), g_2(\mathbf{x}), ..., g_m(\mathbf{x})) \in \mathbb{R}^m$, with $g_i(\mathbf{x})$ additive in K, can be approximated in the sense of L_2 topology in K by the *mapping* input-output represented by an unidirectional neural network (MLP) of k layers ($k-2$ hidden layers), with $f(x)$ as a transfer function of the hidden neurons, and lineal functions for the input and output layers. In other words, $\forall \xi > 0$, there is an MLP with the previous characteristics that implement the mapping:

$$g : x \in K \rightarrow (g'_1(\mathbf{x}), g'_2(\mathbf{x}), ..., g'_m(\mathbf{x})) \in \mathbb{R}^m$$

so that

$$d_{L_2(K)}(\mathbf{g,g'}) = (\sum_{i=1}^{m} \int_K |g_i(x_1, ..., x_n) - g'_i(x_1, ..., x_n)| dx)^{\frac{1}{2}} < \epsilon$$

In summary, *a MLP of one hidden layer can approach every continuous function in a certain interval*, therefore, unidirectional multilayer neural networks are **universal function estimators**. In [10], the result was similar, considering sigmoid activation functions not necessarily continuous.

2.1.2 Backpropagation

Backpropagation algorithm, henceforth BP, was born as a solution for training nodes in hidden layers for multilayer architectures [35, 33]. It starts as a natural consequence for extending LMS algorithm to multilayer networks.

Definition 1. [23] Being a MLP of three layers as seen in Figure 2.2, given an input pattern x^μ, $(\mu = 1, ..., p)$, the global operation for this kind of architecture is expressed as:

$$Z_k^\mu = g(\sum w'_{kj} y_j^\mu - \theta'_k) = g(\sum_j w'_{kj} f(\sum_i w_{ji} x_i^\mu - \theta_j) - \theta'_k)$$

$g(x)$ corresponds to the activation function of the output neurons and $f(x)$ is for the hidden ones. Both can be of type sigmoid, but frequently the function for the output $g(x)$ is considered as the identity. Now we start from the mean squared error as a cost function

$$E(w_{ji}, \theta_j, w'_{kj}, \theta'_k) = \frac{1}{2} \sum_\mu \sum_k [t_k^\mu - g(\sum_j w'_{kj} y_j^\mu - \theta'_k)]^2$$

whose minimization is carried out as the descent for the gradient. There will be a gradient with respect to the weights in the output layer (w'_{kj}), and another one respect to the ones of the hidden layer (w_{ji}).

$$\delta w'_{kj} = -\epsilon \frac{\partial E}{\partial w'_{kj}} \qquad \delta w_{ji} = -\epsilon \frac{\partial E}{\partial w_{ji}}$$

The expressions for weight actualization are obtained for the derivation of the last two expressions, taking into account functional dependencies and applying the chain rule

$$\delta w'_{kj} = \epsilon \sum_\mu \Delta_k'^\mu y_j^\mu, \quad \textit{with} \quad \Delta_k'^\mu = [t_k^\mu - g(h_k'^\mu)] \frac{\partial g(h'\mu_k)}{\partial h_k'^\mu} \quad \textit{and} \quad h_k'^\mu = \sum w'_{kj} y_j^\mu - \theta'_k$$

$$\delta w_{ji} = \epsilon \sum_\mu \Delta_j'^\mu y_i^\mu, \quad \textit{with} \quad \Delta_j^\mu = (\sum_k \Delta_k'^\mu w'_{kj}) \frac{\partial g(h'\mu_j)}{\partial h_j^\mu} \quad \textit{and} \quad h_j^\mu = \sum_i w_{ji} x_i^\mu - \theta_j$$

$h_k'^\mu$ and h_j^μ are the postsynaptic potentials. For bias update, the process to be followed is the same, considering that the bias is a particular case of synaptic weight.

The algorithm can be applied easily in architectures with more than one hidden layer following the same schedule. It is important to start always with initial random weights, because if the algorithm starts with null weights, the process of learning is not going to evolve. This is produced by the fact that the output of the neurons and the increment of the weights are going to be null.

In summary, the procedure to train a MLP with BP is the following one:

1. Establish a random starting value for weights and thresholds ($t = 0$).
2. For each pattern μ of the training set:
   - i) Carry out an execution phase in order to obtain the outputs of the network until the μth pattern.
   - ii) Compute the associated error signals $\Delta_k'^{\mu}$ and Δ_j^{μ}.
   - iii) Compute the partial increment of weights and thresholds due to each pattern μ.
3. Compute the total increment for all the patterns. For the weights $\partial w'_{kj}$ and ∂w_{ji}. Do the same for the thresholds.
4. Update weights and thresholds.
5. Compute the actual error $t = t + 1$ and repeat 2) if is not the expected value.

In this previous schedule, the process of applying gradient descent is carried out over all the patterns of the training set. The variation of weights for each pattern is calculated, stored, and when there is a certain number of patterns that were executed, then the weights are updated. This is known as **batch learning**, but there is also another way of using BP updating weights with every evaluation of each pattern μ, called **online learning**.
The use of these techniques depends on the main characteristics of the problem and the dataset. For example, online learning is specially useful for problems in which we have a huge training set.

2.2 Libraries for automatic learning

Over the last few years, machine learning and more specifically neural networks have become a powerful computational tool capable of solving really interesting use-cases as, for example, recognize diseases in plants with images of leafs [32], compose music [14], or predict the electricity consuption demand of a region [23]. Neural networks can be applied to a lot of real world problems.

In this Final Year Project are going to be used only Python libraries due to the following reasons:

- **Amount of machine learning libraries:** nowadays Python has a huge amount of machine learning libraries, and some of them like Tensorflow and Keras are considered as *State of art*.
- **Simplicity:** thanks to its simple syntax and the huge community, Python is becoming one of the most important programming languages.
- **Power:** machine learning algorithms can be easily executed with Python and the corresponding libraries. This programming language has become one of the main options for data scientists.

Also, another fundamental point whereby Python and libraries used for deep learning are very popular nowadays is due to the compatibility with GPUs. This fact allows the user to parallelize executions and gain performance on the execution of complex tasks as image recognition. The success of Tensorflow and Keras, the libraries that are going to be described in the following subsections, is largely by GPU compatibility.

2.2.1 Tensorflow

Tensorflow is an open source software library for high performance numerical computation. Its architecture is very flexible and allows easy deployment of computation across a huge variety of platforms, as, for example, GPUs, CPUs and TPUs. This library can be executed from desktops to clusters of servers to mobile and edge devices.

Created by researches and engineers from *Google Brain* team within Google's AI organization, *Tensorflow* gives a strong support for deep learning and machine learning, but its flexible numerical computation core is used across many other scientific domains. It is considered the successor of *DistBelief*, because the codebase of this system was modified to create a faster and a more robust application which became *Tensorflow*.

Version 1.0.0 was released on February 11, 2017 under the Apache 2.0 license. Its name derives from the operations that this library performs on multidimensional data arrays, named *tensors*. This fact allows a high-parallelism level in the execution, more if it is combined with graphical processing units.

The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models [1]. Some of the fields of computer science in which this library has been applied are robotics, computer vision, speech recognition, natural language processing, information retrieval and geographic information extraction.

As said before, *Tensorflow* computations are described by a *directed graph*. The graph represents a dataflow computation, with extensions for allowing some kind of nodes to administrate states, branching and looping control structures. Each *node* represents an *operation*, which has zero or more inputs, and the same range for outputs. The values that flow along *nodes* are called *tensors*, and their data representations correspond to arbitrary dimensional arrays with a certain type, specified at graph-construction time [1]. A graphical illustration of a *tensor* appears in Figure 2.3.

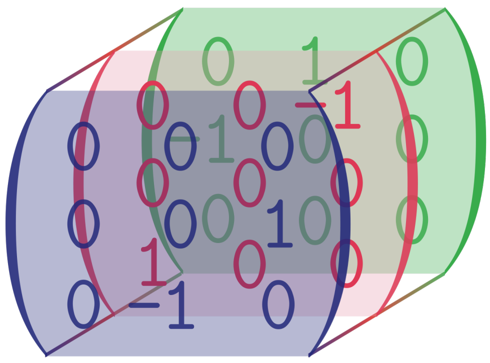


Figure 2.3: Graphical representation of a tensor.

2.2.2 Keras

Keras is a high-level, open source library for neural networks. It is written in Python, and is capable of running *Tensorflow*, *Theano*, *MXNet* and *Microsoft Cognitive Toolkit* as a backend. This library was developed for enabling fast experimentation.

The main characteristics of Keras are the following, according to its documentation [7].

- Easy and fast prototyping.
- Support a high amount of types of neural networks, including recurrent and convolutional networks, as well as combinations of them.
- Modules provide a huge range of examples, and new ones are very simple to add, just as new classes and functions.
- Neural networks are defined as a graph with fully-configurable modules (nodes) that can be plugged together with as little restrictions as possible. An example of the representation appears in Figure 2.4.
- The user APIs reduce the number of actions required for the user, focusing on the user experience.
- Runs on CPU and GPU.

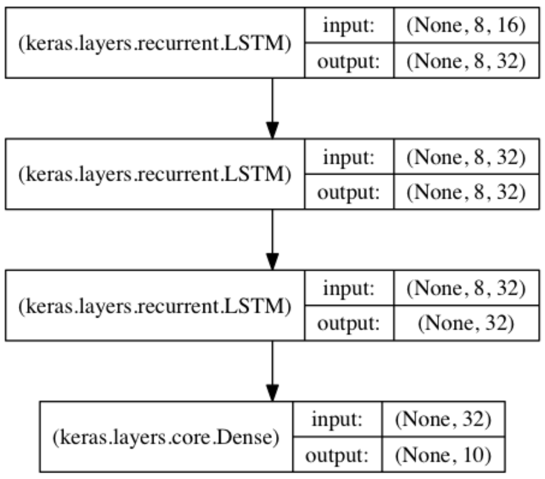


Figure 2.4: Graphical representation of a Keras model as a graph.

Keras was developed as a part of the the research project called *ONEIROS* (Open-ended Neuro-Electronic Intelligent Robot Operating System). The first version was released in 2015, and two years later *Google's Tensorflow Team* decided to support *Keras* in *Tensorflow's* core library.

Keras has stronger adoption in industry and in the research community that any other deep learning framework (except Tensorflow). In november 2017, the number of individual users that uses *Keras* were over two hundred thousand.

This deep learning library is especially popular among startups, and a huge number of features built with *Keras* are used in important companies as for example *Netflix*, *Square* and *Uber*.

Keras has also been adopted by researchers from around the world, highlighting *NASA* and *CERN* research teams.

2.3 Logic programming

Logic programming (LP) was originated in the research on Automatic Theorem Proving. Is based on a subset of predicate logic, concretely in Horn clauses, that are used as the core of a programming language in addition with an operational semantics, SLD-resolution, for which there are a very efficient implementation.

In essence, a logic program is a set of Horn clauses. A clause has the form $A \leftarrow B_1, \dots, B_n$, and can be considered as a part of the formulation of a routine. A clause of the form $\leftarrow C_1, \dots, C_k$ is a goal, and each C_k can be perceived as a call to a routine. To execute a program is to query a goal. If the goal is $\leftarrow C_1, \dots, C_k$, a computation step implies unifying some C_j with the head A of a clause $A \leftarrow B_1, \dots, B_n$, thus obtaining:

$$\leftarrow (C_1, \dots, C_{j-1}, B_1, \dots, B_n, C_{j+1}, \dots, C_k)\theta$$

where θ is a unifier substitution. A program in LP is conceived as a formal theory in a certain logic, and computation is understood as a logic deduction in this logic. The base logic has to include the following elements (see [17]):

- A language expressive enough to address an interesting field of application,
- An operational semantics, that is, a calculation mechanism to execute programs,
- A declarative semantics to provide a meaning to programs independently of their possible execution, and
- Results of soundness and completion to assure that the computed result coincides with what is considered true according to the notion of truth given by the declarative semantics.

Also, this declarative semantics specifies the meaning of the syntactic objects of the language by means of its translation to components and structures in a known (generally mathematical) domain.

The operational semantics in LP is based on a method of proof by refutation called SLD-resolution, that is a case of the resolution strategy. SLD-resolution is based on the unification algorithm and allows the retrieval of responses, i.e., the link of a value to a logical variable. SLD-resolution, whose name comes from "Selective Linear Definite clause resolution", is a refinement of Robinson's resolution, that was first described by R. A. Kowalski [19]. Besides, it is a complete method for the referred logic.

Now, we introduce some basic notions of logic programming. These concepts are addressed with more detail in [22, 15].

Unification

First of all, is necessary to present the concept of unification, a fundamental in logic programing and automatic proving [15]. In an intuitive way, in order to unify two expressions is necessary to make them syntactically equal by using over them a substitution called **unifier** (i.e., both expressions become equal to the cases resulting from them trough some substitution).

Definition 2. A substitution θ is a unifier of the expressions E_1, E_2 if, and only if, $\theta(E_1) = \theta(E_2)$.

We can extend this definition in a natural way to an infinite number of expressions $E_1, \ldots, E_n$, and then, we use the unifier of the set $S = \{E_1, \ldots, E_n\}$.

Definition 3 ([15])**.** A unifier σ of a set of expressions S is the most general unifier for S if, and only if, any other unifier θ is such that $\sigma \leq \theta$.

We write *mgu* for the *most general unifier* of a set of expressions. The *mgu* always exists and is unique (not taking renaming into account, see [20]).

Substitution

Let $\mathcal{V}$ be an infinite set of variables and Σ a set of function symbols f/n, each one of them with an arity n associated. $\mathcal{T}(\Sigma, \mathcal{V})$[1] and $\mathcal{T}(\Sigma)$ stand, respectively, for the set of terms and ground terms (terms with no variables) built upon $\Sigma \cup \mathcal{V}$ and Σ. The set of variables in an expression E is denoted by $\mathcal{V}ar(E)$. A term, then, is said to be ground if $\mathcal{V}ar(t) = \emptyset$.

[1] Occasionally we only write $\mathcal{T}$.

Definition 4 ([15]). A substitution σ is an application $\sigma : \mathcal{V} \longrightarrow \mathcal{T}$ that assigns to each variable x the set of variables $\mathcal{V}$ of a first order language $\mathcal{L}$, a term $\sigma(x)$ of the set of terms $\mathcal{T}$.

It is usual to require $\sigma(x) \neq x$ only for a finite number of variables and, also, to express the substitution in terms of sets, identifying (in some sense) the application σ to the set of images. That is, we write $\sigma = \{x_1/t_1, \dots, x_n/t_n\}$, where $t_i = \sigma(x_i)$ is different from x_i and each pair x_i/t_i is called "binding" or substitution element.

The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} : \sigma(x) \neq x\} = \{x_1, \dots, x_n\}$ is said to be the domain of σ, and its range is $\mathcal{R}an(\sigma) = \{\sigma(x) : x \in \mathcal{D}om(\sigma)\} = \{t_1, \dots, t_n\}$. Additionally, we represent by *id* the identity substitution, that can be understood as the set of empty bindings, so $\mathcal{D}om(id) = \emptyset$, that is, $id(x) = x$ for all $x \in \mathcal{V}$. Also, σ is said to be ground if the terms t_i are ground (the include no variables).

Definition 5. Given an expression E and a substitution σ, $\sigma(E)$ is called *instance* and is the result of applying σ over E, replacing simultaneously all instance of x_i in E by the corresponding term t_i, being x_i/t_i an element of substitution σ.

Usually the previous instance is written $E\sigma$ instead of $\sigma(E)$. Whenever a substitution applies to the more general formulae of language $\mathcal{L}$, and not only to expressions in a clausal language, it is convenient to rename the bound variables before applying the substitution (see, [12]).

Definition 6. Given the substitutions $\sigma = \{x_1/t_1, \dots, x_n/t_n\}$, $\theta = \{y_1/s_1, \dots, y_m/s_m\}$, the composition $\sigma \circ \theta$[2] is the substitution determined from the set $\sigma \circ \theta = \{x_1/\theta(t_1), \dots, x_n/\theta(t_n), y_1/s_1, \dots, y_m/s_m\}$, removing the bindings $x_i/\theta(t_i)$ such that $x_i = \theta(t_i)$ and removing from θ the bindings y_j/s_j such that $y_j \in \{x_1, \dots, x_n\}$.

This composition verifies, over an expression E, that $(\sigma \circ \theta)(E) = \sigma(\theta(E))$ is associative and the identity substitution is the (two-sided) identity element. In the other hand, given σ, θ with $\mathcal{V}ar(\sigma) \cap \mathcal{V}ar(\theta) = \emptyset$, the union $\sigma \cup \theta$ is defined by the union set of both, that is, $(\sigma \cup \theta)(x) = \sigma(x), x \in \mathcal{D}om(\sigma)$ and $(\sigma \cup \theta)(x) = \theta(x), x \in \mathcal{D}om(\theta)$.

A substitution ρ is called *renaming substitution*, or simply *renaming*, if there is ρ^{-1} (inverse substitution) such that $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$. Two expressions E_1, E_2 are variant if there are renaming substitutions ρ, ρ' such that $E_1 = \rho(E_2)$ and $E_2 = \rho'(E_1)$.

Combination of substitutions induces the usual preorder among substitutions: $\theta \leq \sigma$ if, and only if, there is γ that $\sigma = \theta \circ \gamma$, and we say that θ is a more general substitution than σ. This preorder produces a partial preorder over terms given by $t \leq t'$ if

[2]Occasionally we write only $\sigma\theta$ instead of $\sigma \circ \theta$ to abbreviate.

there is γ that $t' = t\gamma$.

Two terms t and t' are variants (one another) if there is a renaming ρ that $t\rho = t'$. Given a substitution θ and a set of variables $W \subseteq \mathcal{V}$, we denote by $\theta_{|W}$ the substitution obtained from θ by restricting $Dom(\theta)$ only to the variables W. We write $\theta = \sigma\ [W]$ if $\theta_{|W} = \sigma_{|W}$, and $\theta \leq \sigma\ [W]$ denotes the existence of a substitution γ such that $\theta \circ \gamma = \sigma\ [W]$.

2.3.1 Prolog

Prolog (PROgrammation en LOGique) is a general purpose logic programming language associated with computational linguistics and artificial intelligence. It was designed in 1972 by Alain Colmerauer and Philippe Roussel, in the University of Aix-Marseille. This programming language has its roots in first-order logic.

The program logic is expressed in terms of relations, represented as rules and facts. Then, a computation is initiated by running a *query* over these relations. In essence, a Prolog program is a set of Horn clauses that are treated as a sequence. Given a certain *objective*, the program starts executing using the operational mechanism of SLD resolution.

There are two types of clauses, facts and rules. Then, there is an example of fact.

```
food(pizza).
```

This built-in predicate is always true, and it is equivalent to the following rule.

```
food(pizza) :- true.
```

Given the above fact, we can ask *pizza is food?*.

```
?- food(pizza).
Yes
```

And also *what things are food?*

```
?- food(X).
X = pizza
```

On the other hand, we can find clauses with bodies, that are called rules, as we can see in the next example.

```
Car :- Wheels.
```

This example is read as “Car is true if Wheels is true”. A rule’s body consists of calls to predicates, which are called the rule’s **goal**. The built-in predicate **/3* means a 3-arity operator with name ***. Conjunctions and disjunctions can only appear in the body, and never in the head of a rule.

Prolog is specially useful for working with databases, language parsing applications and symbolic mathematics.

Operational mechanism

The adopted operational mechanism by the systems that implement *pure Prolog*, is a modified version of the SLD resolution strategy with the following peculiarities:

1. A Prolog *computational rule* deals with objectives as sequences of *atoms* and always choose for resolution the *atom* at the left inside the objective.

2. Prolog tries to unify the selected *atom* by the computational rule with the heads that compounds the program clauses, with the same order in which appears. In other words, the *ordering rule* takes clauses from top to bottom.

3. Prolog uses depth-search with backtracking.

4. Prolog skips *occur check* when makes the unification of two expressions.

5. Prolog allows *ambivalent* syntax. This means that the same function or predicate can be used with different arities in the same program.

2.4 Fuzzy logic

In contrast with traditional logic, fuzzy logic applies to the field of vague or imprecise statements used for describing complex systems with unclear boundaries. Some of this imprecise statements are, for example, *average weight*, *high temperature* and *a lot of humidity*.

The first formulation of fuzzy logic was maded by Lofti A. Zadeh [39, 40] and widen by Goguen and Pavelka, in order to incorporate to formal logic the imprecise predicates of the common language, and create an approximate type of reasoning.

According to Zadeh, fuzzy logic has two different meanings. In a narrow sense, fuzzy logic is a logical system, which is an extension of multivalued[3] logic. On the other hand, in a wider sense fuzzy logic is almost synonymous with the theory of fuzzy sets, a theory which relates to classes of objects with unsharp boundaries in which membership is a matter of degree.

Classical logic, classical set theory and probability are not well suited to address the vagueness, complexity, uncertainty, inconsistency, lack of specificity, and imprecision of the real world. This restriction in the traditional tools motivates the investigation on fuzzy sets and, in parallel, fuzzy logic.

In [39] the author introduces for the first time a theory of fuzzy sets, that are sets with non precise borders and whose membership function gives a degree. One of the main goals of this theory is to provide a basis for approximate reasoning that uses vague hypotheses as a tool for formulating knowledge. Although its nature is different from other logics, this logic of infinite truth values can be seen as an extension of the bivalent logic, of the trivalent logic defined by Łukasiewicz in 1922 and, in general, of the multivalued logic [2].

We can say that fuzzy logic can be seen as a reasoning model that takes into account qualitative or approximated features. This is a great aptitude to work with poorly defined or very complex problems. Its natural basis is conformed by the above mentioned fuzzy sets, that are the mathematical supports that allow fuzzy predicates to perform the logic calculations needed to perform inferences.

This logic is one of the most interesting and recent theories to model a numerous number of systems for which classical logic, multivalent logics and the probability framework are not enough or inappropriate. For this problems, fuzzy logic offers symbols and operators that operate with the notion of vagueness, and inference rules that preserve, delimit and transmit the truth values from the hypothesis to the thesis.

[3]A multivaluated logic corresponds to a logic system which allows more truth degrees that true or false

2.4.1 Fuzzy sets

In *Fuzzy logic and approximate reasoning* [39], Zadeh introduces a notion of fuzzy set through which the concepts of fuzzy interpretation, fuzzy logic operations and linguistic modifiers, among others are formalized. Consider ordinary sets like

$$A = \{x \in \mathbb{Z} : x \text{ is prime}\},\ \ A = \{x \in \mathbb{N} : x \text{ is odd}\},\ \ A = \{x : x \text{ is immortal}\}$$

for which the membership relation is discrete, i.e., an element (of the corresponding universe) belongs or does not belong to the set:

$$\forall x \in \mathcal{U}, \quad x \in A \ \vee\ x \notin A; \quad A \subset \mathcal{U},$$

In the case of fuzzy sets, the membership is associated to a degree; is the case of sets like:

$$A = \{x \in \mathbb{Z} : x \text{ is small}\}$$
$$A = \{x : x \text{ is a hot day}\}$$
$$A = \{x : x \text{ is a modern store}\}$$

in which the nature of the property (predicate) that designates them is not clear (as it is in ordinary sets), but fuzzy. Therefore, we cannot say that the elements of the universe satisfy or not a certain predicate, but they satisfy it at some degree. These sets are formalised providing this new notion of membership, as we do next.

A fuzzy set A, in a universe $\mathcal{U}$, is expressed as:

$$A = \{x|\mu_A(x) : \mu_A(x) \neq 0, x \in \mathcal{U}\},$$

where the application

$$\mu_A : \mathcal{U} \to [0,1]$$

is the membership degree function.

In other words, a fuzzy set is determined by a function μ_A. For each $x \in \mathcal{U}$, $\mu_A(x) \in [0,1]$ is a real number that indicates the compatibility of x with the characteristic (predicate) that defines the set A.

It is also possible to consider that the ordinary membership is determined by the characteristic function

$$\chi_A : \mathcal{U} \to \{0,1\}, \quad \chi_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

that is,

$$x \in A \Leftrightarrow \chi_A(x) = 1, \quad x \notin A \Leftrightarrow \chi_A(x) = 0, \quad \forall x \in \mathcal{U}$$

The fuzzy membership, given by μ_A, is a generalisation of the classical one, since χ_A is a particular case of μ_A.

From this relevant observation follows that the notion of fuzzy set extends the one of the classical set. That is, an ordinary set is a fuzzy set. Particularly, the universe $\mathcal{U}$ (that we take as ordinary in the definition of A) and the empty set $\emptyset$ is fuzzy. Indeed,

$$\mu_\emptyset = \chi_\emptyset \text{ such that } \mu_\emptyset(x) = \chi_\emptyset(x) = 0,\ \forall x \in \mathcal{U}$$

$$\mu_\mathcal{U} = \chi_\mathcal{U} \text{ such that } \mu_\mathcal{U}(x) = \chi_\mathcal{U}(x) = 1,\ \forall x \in \mathcal{U}$$

Once characterized a fuzzy set by its membership degree function, μ_A, all its properties are referred to this, so the content, complementary, operations, etc., are expressed in terms of the corresponding membership functions.

2.4.2 T-norm, T-conorm and aggregators

The syntax of fuzzy logic has not too many novelties with respect to the interpretation of connectives. Once an elemental expression has been interpreted, the compounded expressions take their values applying ad hoc formulae [21]. Thus, for instance, the conjunction is usually defined by the formula

$$\mathcal{I}(A(x_0) \wedge B(x_0)) = min\{\mathcal{I}(A(x_0)), \mathcal{I}(B(y_0))\},$$

where $A(x), B(y)$ are some predicates in universes $\mathcal{U}$, $\mathcal{V}$, respectively, and $x_0 \in \mathcal{U}, y_0 \in \mathcal{V}$.

If we take predicates $A(x), B(x)$ over the same universe $\mathcal{U}$ and they define, respectively, fuzzy sets $A, B \subset \mathcal{U}$, it also follows

$$\mathcal{I}(A(x_0) \wedge B(x_0)) = \mu_{A\cap B}(x_0),$$

where $\mu_{A\cap B}(x_0)$ is the membership degree of x_0 to the intersection set $A \cap B$. That is, it is allowed to define the fuzzy conjunction by means of the corresponding intersection of sets.

In a more natural way, the truth function of the fuzzy conjunction can be defined also by the wide range of functions known as triangular norms, introduced in *Probabilistic Metric Spaces* [36].

Now, we define these functions in the interval $[0, 1]$.

Definition 7 ([31]). An operation $T : [0, 1] \times [0, 1] \longrightarrow [0, 1]$ is a triangular norm or t-norm if, and only if, it verifies

i) is commutative, i.e., $T(x, y) = T(y, x), \forall x, y \in [0, 1]$.

ii) is associative, i.e., $T(x, T(y, z)) = T(T(x, y), z), \forall x, y, z \in [0, 1]$.

iii) $T(x, 1) = x, \forall x \in [0, 1]$.

iv) is monotonic in each component, i.e.[4], if $x_1 \leq x_2$, then $T(x_1, y) \leq T(x_2, y)$, $\forall x_1, x_2, y \in [0, 1]$.

Analogously, the disjunction is usually characterized by the expression

$$\mathcal{I}(A(x_0) \vee B(x_0)) = max\{\mathcal{I}(A(x_0)), \mathcal{I}(B(y_0))\},$$

and if we consider predicates $A(x), B(x)$ on the same universe $\mathcal{U}$ defining, respectively, the fuzzy sets $A, B \subset \mathcal{U}$, we also have

$$\mathcal{I}(A(x_0) \vee B(x_0)) = \mu_{A \cup B}(x_0),$$

where $\mu_{A\cup B}(x_0)$ is the membership degree of x_0 to the union set $A \cup B$. Consequently, this logic operation is associated with the union of sets. More precisely, it is possible to formalize fuzzy disjunction (of propositions and also of predicates) by the union of fuzzy sets.

Furthermore, as in the conjunction, the (truth function of the) fuzzy disjunction can be defined by the wide range of functions called t-conorms, characterized in the following way in the interval $[0, 1]$.

Definition 8 ([31])**.** An operation $S : [0, 1] \times [0, 1] \longrightarrow [0, 1]$ is a triangular conorm, or t-conorm, if, and only if, it verifies

i) is commutative, i.e., $S(x, y) = S(y, x), \forall x, y \in [0, 1]$.

ii) is associative, i.e., $S(x, S(y, z)) = S(S(x, y), z), \forall x, y, z \in [0, 1]$.

iii) $S(x, 0) = x, \forall x \in [0, 1]$.

iv) is monotonic in each component, i.e.[5], if $x_1 \leq x_2$, then $S(x_1, y) \leq S(x_2, y)$, $\forall x_1, x_2, y \in [0, 1]$.

If T is a t-norm in $[0, 1]$, then $S(x, y) = 1 - T(1 - x, 1 - y)$ defines a t-conorm and S is said to derive from T. More generally, given a t-norm T and a strong negation[6] N, then function $S_N : [0, 1] \times [0, 1] \longrightarrow [0, 1]$, defined as $S_N(x, y) = N(T(N(x), N(y)))$, is a t-conorm called N-dual of T.

[4]From the given characterization (only for the first component) follows also the monotonicity in the second one using conditions *i*) and *iv*).

[5]The monotonicity in the second component follows also from *i*) and *iv*).

[6]A strong negation in $[0, 1]$ is a function $N : [0, 1] \longrightarrow [0, 1]$ that is continuous, strictly decreasing and $N(0) = 1, N(N(x)) = x$.

By the elemental properties of negation we have $T(x,y) = N(S_N(N(x),N(y)))$, that is, T is the N-dual t-norm of S_N. Given a t-conorm S and a strong negation N, the function $T_N : [0,1] \times [0,1] \longrightarrow [0,1]$, defined as $T_N(x,y) = N(S(N(x),N(y)))$, is a t-norm called N-dual t-norm of S.

Again, since N is a negation, $S(x,y) = N(T_N(N(x),N(y)))$, that is, S is the N-dual t-conorm of T_N. Concluding, we say that T and S are N-dual if $\forall x,y \in [0,1]$ it holds:

$$T(x,y) = N(S(N(x),N(y))) \qquad S(x,y) = N(T(N(x),N(y)))$$

Particularly, taking the usual negation $N(x) = 1-x$, T and S are dual if $\forall x \in [0,1]$ it holds:

$$T(x,y) = 1 - S(1-x,1-y) \qquad S(x,y) = 1 - T(1-x,1-y)$$

We present below basic pairs of basic t-norms and t-conorms (dual) ([6]):

- Zadeh's (or the Minimum/Maximum) defined by

$$T(x,y) = min\{x,y\} \qquad S(x,y) = max\{x,y\}$$

- Łukasiewicz's defined by

$$T(x,y) = max\{x+y-1,0\} \qquad S(x,y) = min\{x+y,1\}$$

- Of the Product, defined by

$$T(x,y) = xy \qquad S(x,y) = x+y-xy$$

- Weak/Strong, defined by

$$T(x,y) = \begin{cases} min\{x,y\}, & \text{if } max\{x,y\} = 1 \\ 0, & \text{otherwise} \end{cases} \qquad S(x,y) = x+y-xy$$

- Hamacher's, defined for each $\gamma \geq 0$ by

$$T_\gamma(x,y) = \frac{xy}{\gamma + (1-\gamma)(x+y-xy)} \qquad S(x,y) = \frac{x+y-(2-\gamma)xy}{1-(1-\gamma)xy}$$

- Yager's, defined for each $p > 0$ by

$$T_p(x,y) = 1 - min\{1, \sqrt[p]{(1-x)^p + (1-y)^p}\} \quad S_p(x,y) = min\{1, \sqrt[p]{x^p + y^p}\}$$

It is common to use t-norms and t-conorms to produce new connectives [26, 8]. T-norms and t-conorms are particular cases of aggregation operators[7] and, also, certain combinations of them originate new aggregation operators [5].

It is possible to produce aggregators (see [26]) by convex combinations of a t-norm T and a t-conorm S, that is, produce the aggregator $@(x, y) = \alpha T(x, y)+(1-\alpha)S(x, y)$, that preserves symmetry and idempotence.

Aggregators are common in the development of many intelligent systems, as is the case of **neuronal networks**, expert systems, fuzzy controllers and, specially, in decision theory. Aggregators allow the competent and flexible combination of information, which has become a main assignment in multiple-criteria decision problems, where it is necessary to process a great deal of information with different precision and quality.

The most general definition for the aggregation operator, in the interval $[0, 1]$, is the one given in [18], that we show here.

Definition 9. An aggregation operator $@$ is an application $@ : [0, 1]^n \longrightarrow [0, 1]$ that fulfils:

i) $@(0, \dots, 0) = 0, @(1, \dots, 1) = 1$ (boundary conditions)

ii) $\forall (x_1, \dots, x_n), (y_1, \dots, y_n) \in [0, 1]^n$,
$(x_1, \dots, x_n) \leq (y_1, \dots, y_n)$[8] $\Longrightarrow @(x_1, \dots, x_n) \leq @(y_1, \dots, y_n)$ (monotonicity)

Sometimes other conditions are required together with the ones mentioned above, such as continuity, symmetry and idempotence. Particularly, $@$ is symmetric if, and only if, for all permutation σ of $\{1, \dots, n\}$ and all n-uple $(x_1, \dots, x_n) \in [0, 1]^n$ the next holds: $@(x_1, \dots, x_n) = @(x_{\sigma(1)}, \dots, x_{\sigma(n)})$; also, $@$ is idempotent (i.e., $@(x, \dots, x) = x$) if and only if for all n-uple $(x_1, \dots, x_n) \in [0, 1]^n$, $min\{x_1, \dots, x_n\} \leq @(x_1, \dots, x_n) \leq max\{x_1, \dots, x_n\}$ holds.

The most general definition for the aggregation operator, in the interval $[0, 1]$, is the one given in [18], that we reproduce here.

Definition 10. An aggregation operator $@$ is an application $@ : [0, 1]^n \longrightarrow [0, 1]$ that fulfils:

i) $@(0, \dots, 0) = 0, @(1, \dots, 1) = 1$ (boundary conditions)

ii) $\forall (x_1, \dots, x_n), (y_1, \dots, y_n) \in [0, 1]^n$,
$(x_1, \dots, x_n) \leq (y_1, \dots, y_n)$[9] $\Longrightarrow @(x_1, \dots, x_n) \leq @(y_1, \dots, y_n)$ (monotonicity)

[7]See Definition 10 for a characterization of the former.
[8]Where $(x_1, \dots, x_n) \leq (y_1, \dots, y_n)$ if, and only if, $x_i \leq y_i, i = 1, \dots, n$.
[9]Where $(x_1, \dots, x_n) \leq (y_1, \dots, y_n)$ if, and only if, $x_i \leq y_i, i = 1, \dots, n$.

Sometimes other conditions are necessary together with the ones mentioned above, such as symmetry, idempotence and continuity. Particularly, @ is symmetric if, and only if, for all permutation σ of $\{1, \ldots, n\}$ and all n-uple $(x_1, \ldots, x_n) \in [0,1]^n$ the next holds: $@(x_1, \ldots, x_n) = @(x_{\sigma(1)}, \ldots, x_{\sigma(n)})$; also, @ is idempotent (i.e., $@(x, \ldots, x) = x$) if and only if for all n-uple $(x_1, \ldots, x_n) \in [0,1]^n$, $min\{x_1, \ldots, x_n\} \leq @(x_1, \ldots, x_n) \leq max\{x_1, \ldots, x_n\}$ holds.

2.4.3 Fuzzy logic programming

Fuzzy logic programming (FLP) emerge as an extension of LP in the same sense that fuzzy logic extends classical logic. FLP is defined formally as a portion of fuzzy logic focused on the study of fuzzy programs or fuzzy theories, which are a set of fuzzy logic expressions in a first order language directly executables in a computer.

This style of programming applies to areas where the high level of expressiveness and abstraction of traditional declarative languages is required, but those are not able to neither model vague or imprecise scenarios nor formulate approximate reasoning. For this, new expressive resources from fuzzy logic are incorporated.

The area of fuzzy logic programming is in a relatively beginning state, although it is being consolidated by a solid net of researchers that provide maturity in practice aspects as well as in theoretical ones. However there are still neither standards nor a unified framework, but different procedures that take divergent paths.

Due to this diversity of approaches on FLP, it is possible to define many classifications, analyzing the procedural mechanism to deal with vagueness, the extension of syntactic unification, the extension of SLD-resolution, or other considerations where we include interesting concepts as the implementation or not of the negation in this area, or different fuzzy logics [38].

Many of the different procedures of fuzzy logic programming replace the classical inference mechanism SLD-resolution, by a fuzzy variant that allows to deal with uncertainty and estimate truth degrees. It is possible to establish two main trends:

- The first one includes truth degrees together with facts, rules and goals and, therefore, it needs to modify the resolution mechanism to perform operations over those degrees, and unification remains intact.
- The other modifies the unification algorithm and preserves the resolution mechanism by handling truth values separately.

Thus, there is no common method to "fuzzify" the resolution principle of Prolog (see [37]): the majority of these languages implement the fuzzy resolution principle introduced by [21], like the system Fril Prolog [3], Prolog-Elf [13] and the language F-Prolog. Other fuzzy languages Bousi Prolog only take into account the fuzzy component of predicates by introducing the notion of similarity. In these languages there are many schemes to fuzzify the knowledge, to handle it and represent it. The completeness properties for different types of procedural semantics has been projected related to a suitable declarative semantics that, in a lot of cases, has been conceptualized as a fuzzy extension of the classical least Herbrand model [22].

2.4.4 Fuzzy logic languages

MALP

In multi-adjoint logic programming (MALP in brief) [24], each program is associated with a certain lattice that provides truth degrees and allows to encapsulate different types of fuzzy logics inside each rule.

Given a MALP program, goals are evaluated in two separated computational phases. During the resolution phase, the fuzzy counterpart of SLD-derivation steps, called admissible steps, are applied. This first phase produces a substitution and an expression where all atoms have been exploited. This last expression is interpreted afterwards (in the so-called interpretative phase) in the multi-adjoint lattice associated to the program, thus obtaining a pair (*truth degree; substitution*) that is the fuzzy analogous to the classical notion of computed answer traditionally used in LP.

Furthermore, it is noteworthy that the framework based on similarity can be emulated by means of a certain multi-adjoint lattice [25], particularly, the real interval $[0, 1]$ with the t-norm of Gödel, and extending the original program with a set of rules defining a similarity relation (in a similar way to the extension of first order logic with equality axioms). This illustrates the generality and expressiveness of the multi-adjoint logic language.

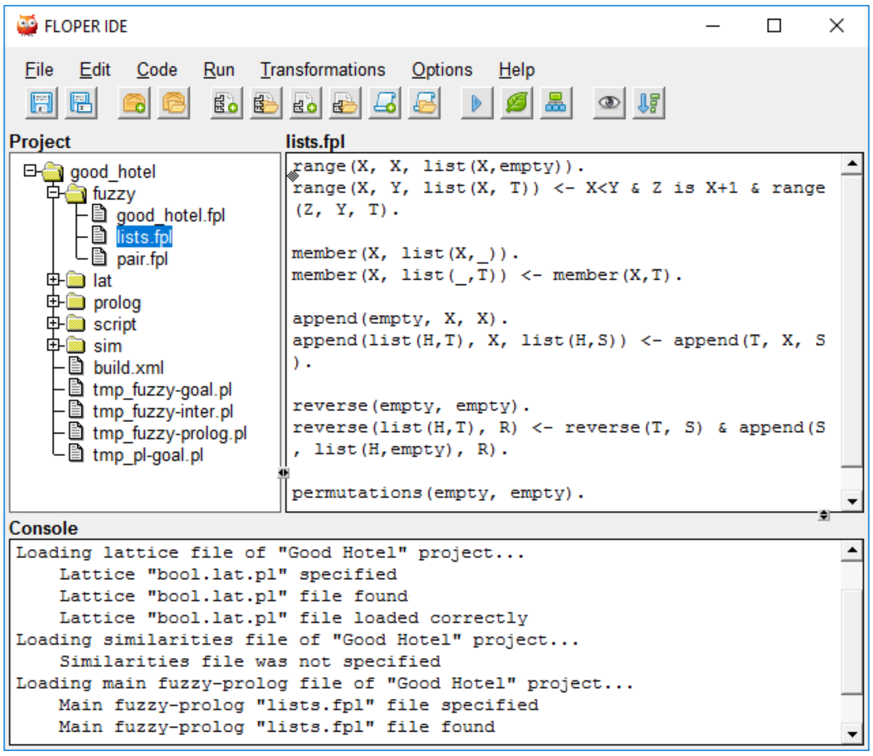
FASILL

FASILL (*Fuzzy Aggregators and Similarity Into a Logic Language*) is a first order language that integrates and extends the capabilities of MALP. It includes some new characteristics as, for example, similarity relations. As well as in other previous fuzzy languages, FASILL accepts the format of rules with explicit weights ($A \leftarrow_{label} B$ *with* r, translated as $A \leftarrow r \,\&_{label}\, B$).

As a fuzzy language, FASILL inherits concepts seen in Section 2.3. Also, substitution and unification have been extended for working with similarity relations. It adopts the line of Bousi Prolog, in which the most general unificator is replaced by the concept of *weak most general unificator*, that introduces the algorithm of *weak unification* for computing it.

The FLOPER environment

FLOPER (*Fuzzy LOgic Programming Environment for Research*) is a tool developed in the University of Castilla-La Mancha by the investigation group DEC-TAU. In their actual state, FLOPER is capable of analyzing and debug fuzzy logic programs with MALP and FASILL syntax in any Prolog platform. Also, FLOPER can execute goals and generate derivation trees in order to obtain fuzzy answers. In Illustration 2.5 there is an screenshot of the desktop version.



```
range(X, X, list(X,empty)).
range(X, Y, list(X, T)) <- X<Y & Z is X+1 & range
(Z, Y, T).

member(X, list(X,_)).
member(X, list(_,T)) <- member(X,T).

append(empty, X, X).
append(list(H,T), X, list(H,S)) <- append(T, X, S
).

reverse(empty, empty).
reverse(list(H,T), R) <- reverse(T, S) & append(S
, list(H,empty), R).

permutations(empty, empty).
```

```
Loading lattice file of "Good Hotel" project...
    Lattice "bool.lat.pl" specified
    Lattice "bool.lat.pl" file found
    Lattice "bool.lat.pl" file loaded correctly
Loading similarities file of "Good Hotel" project...
    Similarities file was not specified
Loading main fuzzy-prolog file of "Good Hotel" project...
    Main fuzzy-prolog "lists.fpl" file specified
    Main fuzzy-prolog "lists.fpl" file found
```

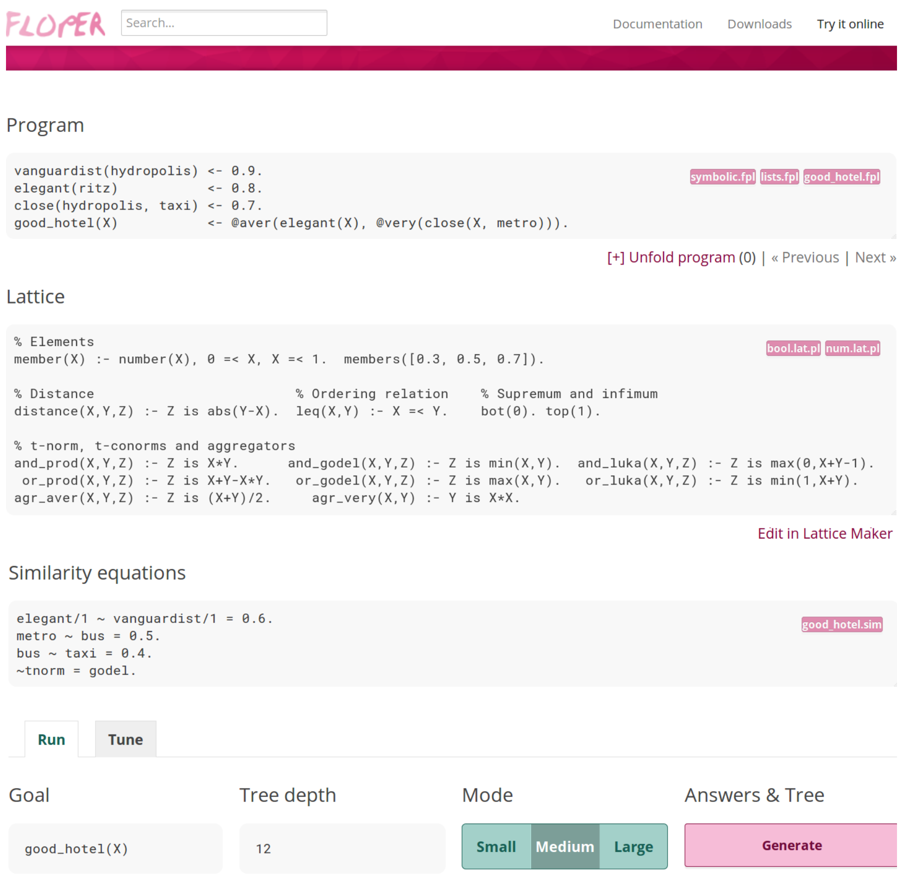
Figure 2.5: Screenshot of the FLOPER Desktop Version.

In [16] a powerful method is proposed to translate fuzzy programs to standard Prolog code that can be directly executable, showing the basic capabilities of this tool that aims to be a decisive experimental platform in specialization, optimization, transformation, and debugging of fuzzy programs.

FLOPER has some differences with Prolog. For example, FLOPER performs a *width search* instead of a *depth search*, and returns all the answers found at a specified depth. Also, FLOPER has three different modes (*long, medium and short*) for the computation of the interpretative steps.

- The **long mode** computes a unique derivation for the computed fuzzy answer.
- The **medium mode** computes in each step a connective of the expression.
- The **short mode** computes in each step one of the internal operations that compounds the different connectives.

In addition, FLOPER has an online tool (as we can see in Illustration 2.6) for analyzing, debug and execute fuzzy logic programs, that can be accessed from `http://dectau.uclm.es/unfold/try`.



Program

```
vanguardist(hydropolis) <- 0.9.
elegant(ritz)           <- 0.8.
close(hydropolis, taxi) <- 0.7.
good_hotel(X)           <- @aver(elegant(X), @very(close(X, metro))).
```

Lattice

```
% Elements
member(X) :- number(X), 0 =< X, X =< 1.  members([0.3, 0.5, 0.7]).

% Distance                           % Ordering relation    % Supremum and infimum
distance(X,Y,Z) :- Z is abs(Y-X).  leq(X,Y) :- X =< Y.    bot(0). top(1).

% t-norm, t-conorms and aggregators
and_prod(X,Y,Z) :- Z is X*Y.      and_godel(X,Y,Z) :- Z is min(X,Y).  and_luka(X,Y,Z) :- Z is max(0,X+Y-1).
 or_prod(X,Y,Z) :- Z is X+Y-X*Y.   or_godel(X,Y,Z) :- Z is max(X,Y).   or_luka(X,Y,Z) :- Z is min(1,X+Y).
agr_aver(X,Y,Z) :- Z is (X+Y)/2.     agr_very(X,Y) :- Y is X*X.
```

Similarity equations

```
elegant/1 ~ vanguardist/1 = 0.6.
metro ~ bus = 0.5.
bus ~ taxi = 0.4.
~tnorm = godel.
```

Goal: good_hotel(X)
Tree depth: 12

Figure 2.6: Screenshot of the FLOPER Online Version.

Chapter 3

TUNING FUZZY LOGIC PROGRAMS

In this part, some fundamental concepts will be explained that compound the basis of this project. The following sections show how tuning is made, based on the papers presented in the RuleMl+RR 2017 and LOPSTR 2017 conferences, for more details, see [29] and [27] respectively. In the next chapters, these concepts will be adapted to our specific domain of neural networks and their fuzzy logic translation.

3.1 The fuzzy logic languages MALP and sMALP

In this chapter we focus on the *multi-adjoint logic programming* approach MALP [25], a powerful language in the area of fuzzy logic programming. Intuitively speaking, logic programming is extended with a *multi-adjoint lattice* L of truth values (typically, a real number between 0 and 1), equipped with a collection of *adjoint pairs* $\langle \&_i, \leftarrow_i \rangle$ and connectives: implications, conjunctions, disjunctions, and other operators called aggregators, which are interpreted on this lattice. Consider, for instance, the following MALP rule:

$$good(X) \leftarrow_{\mathtt{P}} @_{\mathtt{aver}}(nice(X), cheap(X)) \ with\ 0.8$$

where the adjoint pair $\langle \&_{\mathtt{P}}, \leftarrow_{\mathtt{P}} \rangle$ is defined as

$$\&_{\mathtt{P}}(x,y) \triangleq x * y \qquad\qquad \leftarrow_{\mathtt{P}} (x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x/y & \text{if } 0 < x < y \end{cases}$$

and the aggregator $@_{\mathtt{aver}}$ is defined as $@_{\mathtt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Therefore, the rule specifies that X is good (with a truth degree of 0.8) if X is nice and cheap. Assuming that X is nice and cheap with, e.g., truth degrees n and c, respectively, then X is good with a truth degree of $0.8 * ((n + c)/2)$.

When specifying a MALP program, it might sometimes be difficult to assign weights (truth degrees) to program rules, as well as to determine the right connectives.[1] This is a common problem with fuzzy control system design, where some trial-and-error is often necessary. In our context, a programmer can develop a prototype and repeatedly execute it until the set of answers is the intended one. Unfortunately, this is a tedious and time consuming operation. Actually, it might be impractical when the program should correctly model a large number of test cases provided by the user [27].

In order to overcome this obstacle, in this chapter we propose a symbolic extension of MALP programs called *symbolic multi-adjoint logic programming* (sMALP). In this new language we can write rules with *symbolic* connectives and *symbolic* truth degrees, i.e., connectives which are not defined on its associated multi-adjoint lattice. In order to evaluate these programs, there is a symbolic operational semantics that delays the evaluation of symbolic expressions. Therefore, a *symbolic answer* could now include symbolic (unknown) truth values and connectives. Using the symbolic semantics and then changing the unknown values and connectives by concrete ones gives the same result as replacing these values and connectives in the original sMALP program and, then, applying the concrete semantics on the resulting MALP program [29].

We assume the existence of a multi-adjoint lattice $\langle L, \preceq, \&_1, \leftarrow_1, \ldots, \&_n, \leftarrow_n \rangle$, provided with a collection of *adjoint pairs* $\langle \&_i, \leftarrow_i \rangle$ (where each $\&_i$ is a conjunctor that is intended to be used for the evaluation of *modus ponens*) [25]. In addition, on each program rule, we can have a different adjoint implication ($\leftarrow_i$), conjunctions (denoted by $\wedge_1, \wedge_2, \ldots$), disjunctions ($|_1, |_2, \ldots$), adjoint conjunctions ($\&_1, \&_2, \ldots$), and other operators called aggregators (usually denoted by $@_1, @_2, \ldots$); see [31] for more details. More exactly, a multi-adjoint lattice fulfills the following properties:

- $\langle L, \preceq \rangle$ is a (bounded) complete lattice.[2]
- For each truth function of $\&_i$, an increase in any of the arguments results in an increase of the result (they are *increasing*).
- For each truth function of $\leftarrow_i$, the result increases as the first argument increases, but it decreases as the second argument increases (they are *increasing* in the consequent and *decreasing* in the antecedent).

[1]For instance, we have typically several adjoint pairs: *Łukasiewicz logic* $\langle \&_{\texttt{L}}, \leftarrow_{\texttt{L}} \rangle$, *Gödel logic* $\langle \&_{\texttt{G}}, \leftarrow_{\texttt{G}} \rangle$ and *product logic* $\langle \&_{\texttt{P}}, \leftarrow_{\texttt{P}} \rangle$, which might be used for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively.

[2]A complete lattice is a (partially) ordered set $\langle L, \preceq \rangle$ such that every subset S of L has infimum and supremum elements. It is bounded if it has bottom and top elements, denoted by $\bot$ and $\top$, respectively. L is said to be the *carrier set* of the lattice, and $\preceq$ its ordering relation.

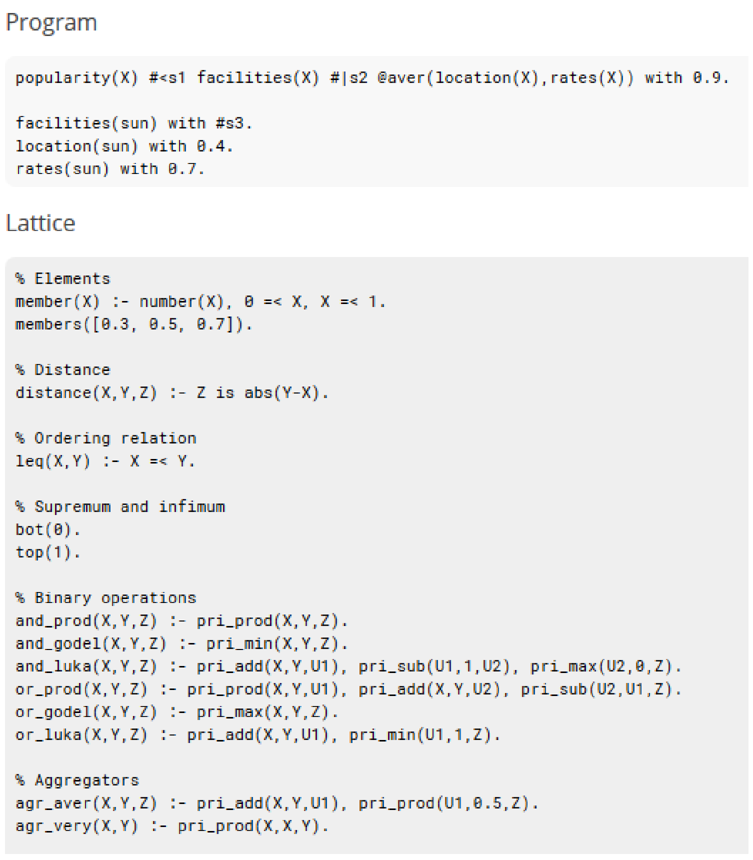
- $\langle \&_i, \leftarrow_i \rangle$ is an *adjoint pair* in $\langle L, \preceq \rangle$, namely, for any $x, y, z \in L$, we have that: $x \preceq (y \leftarrow_i z)$ if and only if $(x \&_i z) \preceq y$.

The last condition, called the *adjoint property*, could be considered the most important aspect of the framework (in contrast with other approaches) which justifies most of its properties regarding decisive results for completeness, applicability, soundness, etc. [25].

Aggregation operators are appropriate to describe or specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arithmetic mean, a weighted sum or in general any monotone function whose arguments are values of a multi-adjoint lattice L.

Although, formally, these connectives are binary operators, we often use them as n-ary functions so that $@(x_1, \ldots, @(x_{n-1}, x_n), \ldots)$ is denoted by $@(x_1, \ldots, x_n)$. In these cases, we consider $@$ an n-ary operator.

The truth function of an n-ary connective ς is denoted by $[\![\varsigma]\!] : L^n \mapsto L$ and is required to be monotonic and fulfill the following conditions: $[\![\varsigma]\!](\top, \ldots, \top) = \top$ and $[\![\varsigma]\!](\bot, \ldots, \bot) = \bot$.


Program

```
popularity(X) #<s1 facilities(X) #|s2 @aver(location(X),rates(X)) with 0.9.

facilities(sun) with #s3.
location(sun) with 0.4.
rates(sun) with 0.7.
```

Lattice

```
% Elements
member(X) :- number(X), 0 =< X, X =< 1.
members([0.3, 0.5, 0.7]).

% Distance
distance(X,Y,Z) :- Z is abs(Y-X).

% Ordering relation
leq(X,Y) :- X =< Y.

% Supremum and infimum
bot(0).
top(1).

% Binary operations
and_prod(X,Y,Z) :- pri_prod(X,Y,Z).
and_godel(X,Y,Z) :- pri_min(X,Y,Z).
and_luka(X,Y,Z) :- pri_add(X,Y,U1), pri_sub(U1,1,U2), pri_max(U2,0,Z).
or_prod(X,Y,Z) :- pri_prod(X,Y,U1), pri_add(X,Y,U2), pri_sub(U2,U1,Z).
or_godel(X,Y,Z) :- pri_max(X,Y,Z).
or_luka(X,Y,Z) :- pri_add(X,Y,U1), pri_min(U1,1,Z).

% Aggregators
agr_aver(X,Y,Z) :- pri_add(X,Y,U1), pri_prod(U1,0.5,Z).
agr_very(X,Y) :- pri_prod(X,X,Y).
```

Figure 3.1: Screenshot of the program and the lattice.

Example 1. In Figure 3.1, we show the shape of the lattice of truth degrees $([0,1], \leq)$ loaded by default in the FLOPER online tool. In general, lattices are described by means of a set of Prolog clauses where the definition of the following predicates is mandatory: `member/1`, that determine the elements of the lattice; `members/1`, that highlights into a list a subset of truth degrees to be used at tuning time; `bot/1` and `top/1` stand for the infimum and supremum elements of the lattice; and finally `leq/2`, that implements the ordering relation. Connectives are defined as predicates whose meaning is given by a number of clauses. The name of a predicate has the form `and_`*label*, `or_`*label* or `agr_`*label* depending on if it implements a conjunction, a disjunction or an aggregator correspondingly, where a *label* is an identifier of that particular connective.

The arity of the predicate is $n+1$, where n is the arity of the connective that it implements, so its last parameter is a variable to be unified with the truth value resulting of its evaluation [29]. This example can be tested online via the following URL:

http://dectau.uclm.es/tuning/

Given a multi-adjoint lattice L, we consider a first order language $\mathcal{L}_{Lukasiewicz}$ built upon a signature $\Sigma_{Lukasiewicz}$, that contains the elements of a calculable infinite set of variables $\mathcal{V}$, function and predicate symbols (denoted by $\mathcal{F}$ and Π, respectively) with an associated arity (usually considered as pairs f/n or p/n, respectively, where n represents its arity), and the truth degree literals $\Sigma^T_{Lukasiewicz}$ and connectives $\Sigma^C_{Lukasiewicz}$ from L. Therefore, a well-formed formula in $\mathcal{L}_{Lukasiewicz}$ can be either:

- A *value* $v \in \Sigma^T_{Lukasiewicz}$, which will be interpreted as itself, i.e., as the truth degree $v \in L$.
- $p(t_1, \ldots, t_n)$, if $t_1, \ldots, t_n$ are terms over $\mathcal{V} \cup \mathcal{F}$ and p/n is an n-ary predicate. This formula is called *atomic* (or just an atom).
- $\varsigma(e_1, \ldots, e_n)$, if $e_1, \ldots, e_n$ are well-formed formulas and ς is an n-ary connective with truth function $[\![\varsigma]\!] : L^n \mapsto L$.

As usual, a *substitution* σ is a mapping from variables from $\mathcal{V}$ to terms over $\mathcal{V} \cup \mathcal{F}$ such that $Dom(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. Substitutions are usually denoted by sets of pairs like, e.g., $\{x_1/t_1, \ldots, x_n/t_n\}$. Substitutions are extended to morphisms from terms to terms in a natural way. The identity substitution is denoted by *id*. The composition of substitutions is denoted by juxtaposition, i.e., $\sigma\theta$ denotes a substitution δ such that $\delta(x) = \theta(\sigma(x))$ for all $x \in \mathcal{V}$ [27].

$$
\begin{array}{llllll}
\&_{\texttt{Product}}(x,y) & \triangleq & x * y & \leftarrow_{\texttt{Product}}(x,y) & \triangleq & \begin{cases} 1 & \text{if } y \leq x \\ x/y & \text{if } 0 < x < y \end{cases} \\
\&_{\texttt{Gödel}}(x,y) & \triangleq & \min(x,y) & \leftarrow_{\texttt{Gödel}}(x,y) & \triangleq & \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases} \\
\&_{\texttt{Lukasiewicz}}(x,y) & \triangleq & \max(0, x+y-1) & \leftarrow_{\texttt{Lukasiewicz}}(x,y) & \triangleq & \min(x-y+1, 1)
\end{array}
$$

Figure 3.2: Adjoint pairs of three different fuzzy logics over $\langle [0,1], \leq \rangle$.

In the following, an *L-expression* is a well-formed formula of $\mathcal{L}_{Lukasiewicz}$ which is composed only by values and connectives from L, i.e., expressions over $\Sigma^T_{Lukasiewicz} \cup \Sigma^C_{Lukasiewicz}$.

In what follows, we assume that the truth function of any connective ς in L is given by a corresponding definition of the form $[\![\varsigma]\!](x_1, \ldots, x_n) \triangleq E$.[3] For instance, in this work, we will be mainly concerned with the classical set of adjoint pairs (conjunctors and implications) over $\langle [0,1], \leq \rangle$ shown in Figure 3.2, where labels L, G and P mean respectively *Łukasiewicz logic*, *Gödel logic* and *Product logic* (which might be used for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively).

A MALP *rule* over a multi-adjoint lattice L is a formula $H \leftarrow_i \mathcal{B}$, where H is an *atomic formula* (usually called the *head* of the rule), $\leftarrow_i$ is an implication symbol belonging to some adjoint pair of L, and $\mathcal{B}$ (which is called the *body* of the rule) is a well-formed formula over L without implications [27].
A *goal* is a body submitted as a query to the system. A MALP program is a set of expressions R *with* v, where R is a rule and v is a *truth degree* (a value of L) expressing the confidence of a programmer in the truth of rule R. By abuse of the language, we often refer to R *with* v as a rule. See, e.g., [25] for a complete formulation of the MALP framework.

3.2 Symbolic substitution

The *symbolic* extension of multi-adjoint logic programming allows some undefined values (truth degrees) and connectives in the program rules, and these elements can be systematically computed afterwards. In the following, we will use the abbreviation sMALP to refer to programs belonging to this setting.

[3]For convenience, in the following sections, we do not distinguish between the connective ς and its truth function $[\![\varsigma]\!]$.

Here, given a multi-adjoint lattice L, we consider an augmented language $\mathcal{L}_L^s \supseteq \mathcal{L}_L$ which may also include a number of symbolic values, symbolic adjoint pairs and symbolic connectives which do not belong to L. Symbolic objects are usually denoted as o^s with a superscript s.

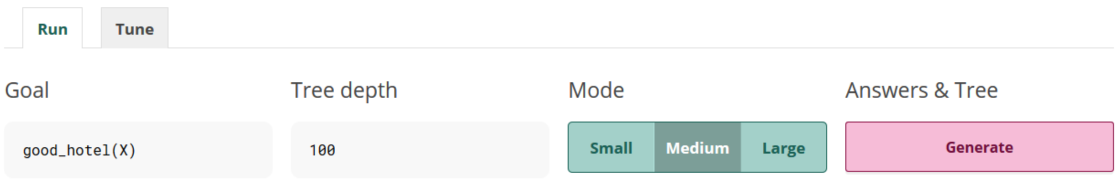
Definition 11 (sMALP program)**.** Let L be a multi-adjoint lattice. A sMALP program over L is a set of symbolic rules, where each symbolic rule is a formula $(H \leftarrow_i \mathcal{B}\ \mathit{with}\ v)$ that meets the following conditions:

- H is an atomic formula of $\mathcal{L}_L$ (the head of the rule);
- $\leftarrow_i$ is a (possibly symbolic) implication from either a symbolic adjoint pair $\langle \&^s, \leftarrow^s \rangle$ or from an adjoint pair of L;
- $\mathcal{B}$ (the body of the rule) is a symbolic goal, i.e., a well-formed formula of $\mathcal{L}_L^s$;
- v is either a truth degree (a value of L) or a symbolic value.

Example 2. We consider the multi-adjoint lattice $\langle [0,1], \leq, \&_{\mathtt{P}}, \leftarrow_{\mathtt{P}}, \&_{\mathtt{G}}, \leftarrow_{\mathtt{G}}, \&_{\mathtt{L}}, \leftarrow_{\mathtt{L}} \rangle$, where the adjoint pairs are defined in Section 3.1, also including $@_{\mathtt{aver}}$ which is defined as follows: $@_{\mathtt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Then, the following is a sMALP program $\mathcal{P}$:

$$\begin{array}{lll} p(X) \leftarrow^{s_1} & \&^{s_2}(q(X), @_{\mathtt{aver}}(r(X), s(X))) & \mathit{with}\ \ 0.9 \\ q(a) & & \mathit{with}\ \ v^s \\ r(X) & & \mathit{with}\ \ 0.7 \\ s(X) & & \mathit{with}\ \ 0.5 \end{array}$$

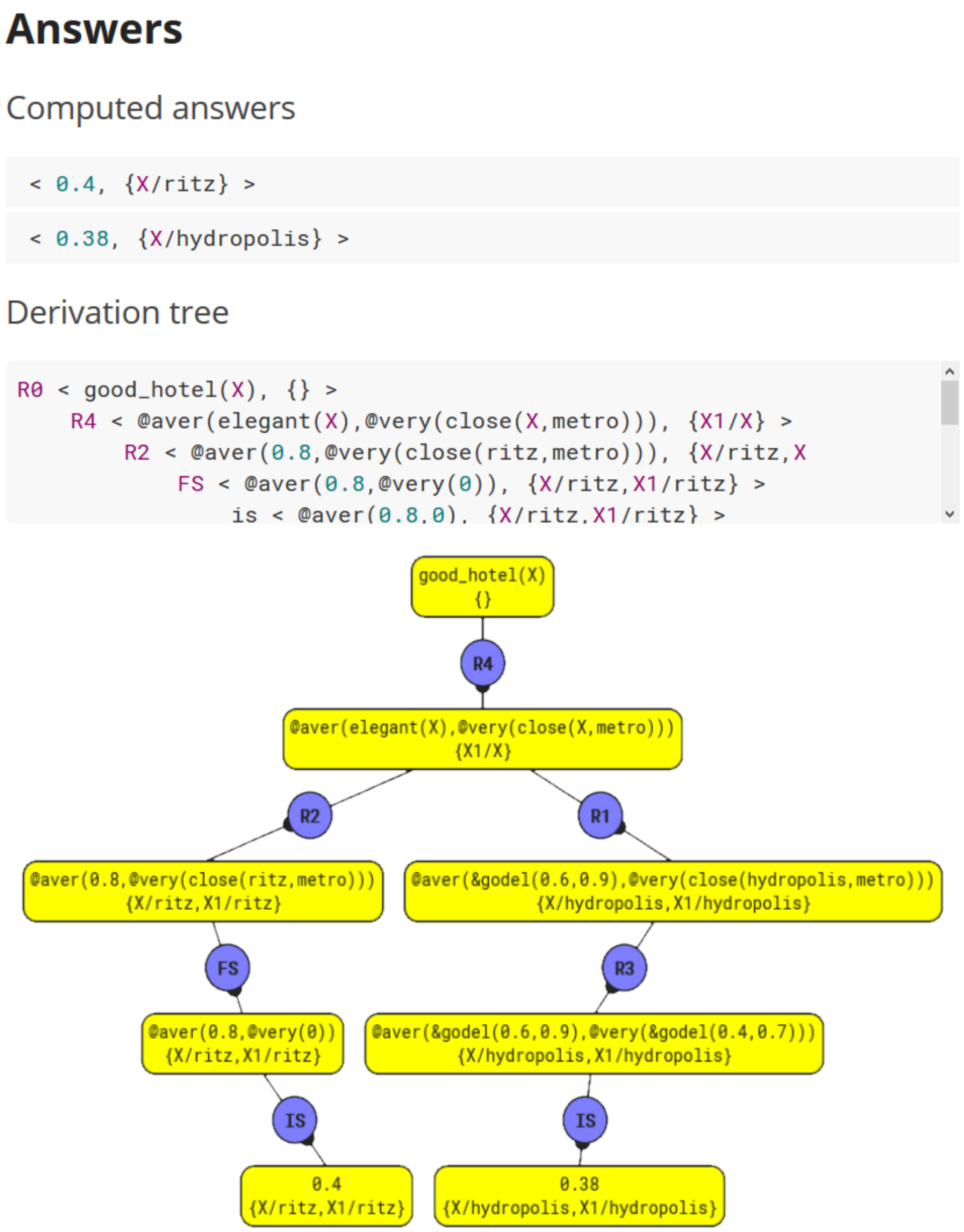
where $\langle \&^{s_1}, \leftarrow^{s_1} \rangle$ is a symbolic adjoint pair (i.e., a pair not defined in L), $\&^{s_2}$ is a symbolic conjunction, and v^s is a symbolic value.



Goal: good_hotel(X)
Tree depth: 100
Mode: Medium
Answers & Tree

Figure 3.3: Screenshot of the FLOPER online tool showing the run area.

In Figure 3.3 we can see the run area of our online tool. After introducing a goal and clicking on the Answers & Tree button, the system computes the goal and generates both the whole set of its sfca's as well as its associated derivation tree (in plain text and also graphically), as seen in Figure 3.4. Each sfca appears on a different leaf of the tree, where each state contains its corresponding goal and substitution components and they are drawn inside yellow ovals. Computational steps, colored in blue, are labeled with the program rule they exploit in the case of *admissible* steps or with the word “is”, corresponding to *interpretive* steps [29].



Answers

Computed answers

```
< 0.4, {X/ritz} >
< 0.38, {X/hydropolis} >
```

Derivation tree

```
R0 < good_hotel(X), {} >
    R4 < @aver(elegant(X),@very(close(X,metro))), {X1/X} >
        R2 < @aver(0.8,@very(close(ritz,metro))), {X/ritz,X
            FS < @aver(0.8,@very(0)), {X/ritz,X1/ritz} >
                is < @aver(0.8,0), {X/ritz,X1/ritz} >
```

good_hotel(X)
{}
R4
@aver(elegant(X),@very(close(X,metro)))
{X1/X}
R2
R1
@aver(0.8,@very(close(ritz,metro)))
{X/ritz,X1/ritz}
@aver(&godel(0.6,0.9),@very(close(hydropolis,metro)))
{X/hydropolis,X1/hydropolis}
FS
R3
@aver(0.8,@very(0))
{X/ritz,X1/ritz}
@aver(&godel(0.6,0.9),@very(&godel(0.4,0.7)))
{X/hydropolis,X1/hydropolis}
IS
IS
0.4
{X/ritz,X1/ritz}
0.38
{X/hydropolis,X1/hydropolis}

Figure 3.4: Answers found for a symbolic program.

The procedural semantics of sMALP is defined in a stepwise manner as follows. First, an *operational* stage is introduced which proceeds similarly to SLD resolution in pure logic programming. In contrast to standard logic programming, though, our operational stage returns an expression still containing a number of (possibly symbolic) values and connectives. Then, an *interpretive* stage evaluates these connectives and produces a final answer (possibly containing symbolic values and connectives).
The procedural semantics of both MALP and sMALP programs is based on a similar scheme. The main difference is that, for MALP programs, the interpretive stage always returns a value, while for sMALP programs we might get an expression containing symbolic values and connectives that should be first instantiated in order to compute a value.

In the following, $\mathcal{C}[A]$ denotes a formula where A is a sub-expression which occurs in the (possibly empty) context $\mathcal{C}[]$. Moreover, $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$, whereas $\mathcal{V}ar(s)$ refers to the set of distinct variables occurring in the syntactic object s, and $\theta[\mathcal{V}ar(s)]$ denotes the substitution obtained from θ by restricting its domain to $\mathcal{V}ar(s)$. A sMALP *state* has the form $\langle \mathcal{Q}; \sigma \rangle$ where $\mathcal{Q}$ is a symbolic goal and σ is a substitution. We let $\mathcal{E}^s$ denote the set of all possible sMALP states.

Definition 12 (admissible step)**.** Let L be a multi-adjoint lattice and $\mathcal{P}$ a sMALP program over L. An *admissible step* is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E}^s \times \mathcal{E}^s)$ is the smallest relation satisfying the following transition rules:[4]

1. $\langle \mathcal{Q}[A]; \sigma \rangle \;\rightarrow_{AS}\; \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$,
   if $\theta = mgu(\{H = A\}) \neq fail$, $(H \leftarrow_i \mathcal{B}\; with\; v) \ll \mathcal{P}$ and $\mathcal{B}$ is not empty.[5]

2. $\langle \mathcal{Q}[A]; \sigma \rangle \;\rightarrow_{AS}\; \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$,
   if there is no rule $(H \leftarrow_i \mathcal{B}\; with\; v) \ll \mathcal{P}$ such that $mgu(\{H = A\}) \neq fail$.

Here, $(H \leftarrow_i \mathcal{B}\; with\; v) \ll \mathcal{P}$ denotes that $(H \leftarrow_i \mathcal{B}\; with\; v)$ is a renamed apart variant of a rule in $\mathcal{P}$ (i.e., all its variables are fresh). Note that symbolic values and connectives are not renamed.

Observe that the second rule is needed to cope with expressions like $@_{aver}(p(a), 0.8)$, which can be evaluated successfully even when there is no rule match-

[4] Here, we assume that A in $\mathcal{Q}[A]$ is the selected atom. Furthermore, as it is common practice, $mgu(E)$ denotes the *most general unifier* of the set of equations E [20].

[5] For simplicity, we consider that facts $(H\; with\; v)$ are seen as rules of the form $(H \leftarrow_i \top\; with\; v)$ for some implication $\leftarrow_i$. Furthermore, in this case, we directly derive the state $\langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$ since $v \&_i \top = v$ for all $\&_i$.

ing $p(a)$ since $@_{aver}(0, 0.8) = 0.4$.

In the following, given a relation $\to$, we let $\to^*$ denote its reflexive and transitive closure. Also, an *L^s-expression* is now a well-formed formula of $\mathcal{L}_L^s$ which is composed of values and connectives from L as well as by symbolic values and connectives.

Definition 13 (admissible derivation)**.** Let L be a multi-adjoint lattice and $\mathcal{P}$ be a sMALP program over L. Given a goal $\mathcal{Q}$, an *admissible derivation* is a sequence $\langle \mathcal{Q}; id\rangle \to_{AS}^* \langle \mathcal{Q}'; \theta\rangle$. When $\mathcal{Q}'$ is an L^s-expression, the derivation is called *final* and the pair $\langle \mathcal{Q}'; \sigma\rangle$, where $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, is called a *symbolic admissible computed answer* (saca, for short) for goal $\mathcal{Q}$ in $\mathcal{P}$.

Example 3. Consider again the multi-adjoint lattice L and the sMALP program $\mathcal{P}$ of Example 2. Here, we have the following final admissible derivation for $p(X)$ in $\mathcal{P}$ (the selected atom is underlined):

$$\begin{array}{lll}
\langle \underline{p(X)};\ id\rangle & \to_{AS} & \langle \&^{s_1}(0.9, \&^{s_2}(\underline{q(X_1)}, @_{\texttt{aver}}(r(X1), s(X1)))); \{X/X_1\}\rangle \\
 & \to_{AS} & \langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\texttt{aver}}(\underline{r(a)}, s(a)))); \{X/a, X_1/a\}\rangle \\
 & \to_{AS} & \langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\texttt{aver}}(0.7, \underline{s(a)}))); \{X/a, X_1/a, X_2/a\}\rangle \\
 & \to_{AS} & \langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\texttt{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\}\rangle
\end{array}$$

Therefore, the associated saca is $\langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\texttt{aver}}(0.7, 0.5))); \{X/a\}\rangle$.

Given a goal $\mathcal{Q}$ and a final admissible derivation $\langle \mathcal{Q}; id\rangle \to_{AS}^* \langle \mathcal{Q}'; \sigma\rangle$, we have that $\mathcal{Q}'$ does not contain atomic formulas. Now, Q' can be *solved* by using the following interpretive stage:

Definition 14 (interpretive step)**.** Let L be a multi-adjoint lattice and $\mathcal{P}$ be a sMALP program over L. Given a saca $\langle Q; \sigma\rangle$, the *interpretive* stage is formalized by means of the following transition relation $\to_{IS} \subseteq (\mathcal{E}^s \times \mathcal{E}^s)$, which is defined as the least transition relation satisfying:

$$\langle \mathcal{Q}[\varsigma(r_1, \ldots, r_n)]; \sigma\rangle \ \to_{IS}\ \langle \mathcal{Q}[\varsigma(r_1, \ldots, r_n)/r_{n+1}]; \sigma\rangle$$

where ς denotes a connective defined on L and $[\![\varsigma]\!](r_1, \ldots, r_n) = r_{n+1}$. An interpretive derivation of the form $\langle \mathcal{Q}; \sigma\rangle \to_{IS}^* \langle \mathcal{Q}'; \theta\rangle$ such that $\langle \mathcal{Q}'; \theta\rangle$ cannot be further reduced, is called a *final* interpretive derivation. In this case, $\langle \mathcal{Q}'; \theta\rangle$ is called a *symbolic fuzzy computed answer* (sfca, for short). Also, if $\mathcal{Q}'$ is a value of L, we say that $\langle \mathcal{Q}'; \theta\rangle$ is a fuzzy computed answer (fca, for short).

Example 4. Given the saca of Ex. 3: $\langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\texttt{aver}}(0.7, 0.5))); \{X/a\}\rangle$, we have the following final interpretive derivation (the connective reduced is underlined):

$$\langle \&^{s_1}(0.9, \&^{s_2}(v^s, \underline{@_{\texttt{aver}}(0.7, 0.5)})); \{X/a\}\rangle \to_{IS} \langle \&^{s_1}(0.9, \&^{s_2}(v^s, 0.6)); \{X/a\}\rangle$$

with $[\![@_{\texttt{aver}}]\!](0.7, 0.5) = 0.6$. Therefore, $\langle \&^{s_1}(0.9, \&^{s_2}(v^s, 0.6)); \{X/a\}\rangle$ is a sfca of $p(X)$ in $\mathcal{P}$ since it cannot be further reduced.

Given a multi-adjoint lattice L and a symbolic language $\mathcal{L}^s_L$, in the following we consider *symbolic substitutions* that are mappings from symbolic values and connectives to expressions over $\Sigma^T_L \cup \Sigma^C_L$. Symbolic substitutions are denoted by $\Theta, \Gamma, \ldots$ Furthermore, for all symbolic substitution Θ, we require the following condition: $\leftarrow^s/\leftarrow_i \in \Theta$ iff $\&^s/\&_i \in \Theta$, where $\langle \&^s, \leftarrow^s\rangle$ is a symbolic adjoint pair and $\langle \&_i, \leftarrow_i\rangle$ is an adjoint pair in L. Intuitively, this is required for the substitution to have the same effect both on the program and on an L^s-expression.

Given a sMALP program $\mathcal{P}$ over L, we let $\mathcal{S}ym(\mathcal{P})$ denote the symbolic values and connectives in $\mathcal{P}$. Given a symbolic substitution Θ for $\mathcal{S}ym(\mathcal{P})$, we denote by $\mathcal{P}\Theta$ the program that results from $\mathcal{P}$ by replacing every symbolic symbol e^s by $e^s\Theta$. Trivially, $\mathcal{P}\Theta$ is now a MALP program.

The following theorem is our key result in order to use sMALP programs for tuning the components of a MALP program:

Theorem 2. Let L be a multi-adjoint lattice and $\mathcal{P}$ be a sMALP program over L. Let $\mathcal{Q}$ be a goal. Then, for any symbolic substitution Θ for $\mathcal{S}ym(\mathcal{P})$, we have that $\langle v; \theta\rangle$ is a fca for Q in $\mathcal{P}\Theta$ iff there exists a sfca $\langle Q'; \theta'\rangle$ for $\mathcal{Q}$ in $\mathcal{P}$ and $\langle \mathcal{Q}'\Theta; \theta'\rangle \to^*_{IS} \langle v; \theta'\rangle$, where θ' is a renaming of θ.

For simplicity, we consider that the same fresh variables are used for renamed apart rules in both derivations. Consider the following derivations for goal $\mathcal{Q}$ w.r.t. programs $\mathcal{P}$ and $\mathcal{P}\Theta$, respectively:

$$\begin{aligned}
\mathcal{D}_{\mathcal{P}} &: \langle \mathcal{Q}; id\rangle \to^*_{AS} \langle \mathcal{Q}''; \theta\rangle \quad \to^*_{IS} \langle \mathcal{Q}'; \theta\rangle\\
\mathcal{D}_{\mathcal{P}\Theta} &: \langle \mathcal{Q}; id\rangle \to^*_{AS} \langle \mathcal{Q}''\Theta; \theta\rangle \to^*_{IS} \langle \mathcal{Q}'\Theta; \theta\rangle
\end{aligned}$$

Our proof proceeds now in three stages:

1. Firstly, observe that the sequences of symbolic admissible steps in $\mathcal{D}_{\mathcal{P}}$ and $\mathcal{D}_{\mathcal{P}\Theta}$ exploit the whole set of atoms in both cases, such that a program rule R is used in $\mathcal{D}_{\mathcal{P}}$ iff the corresponding rule $R\Theta$ is applied in $\mathcal{D}_{\mathcal{P}\Theta}$ and hence, the saca's of the derivations are $\langle \mathcal{Q}''; \theta\rangle$ and $\langle \mathcal{Q}''\Theta; \theta\rangle$, respectively.

2. Then, we proceed by applying interpretive steps until reaching the sfca $\langle \mathcal{Q}'; \theta\rangle$ in the first derivation $\mathcal{D}_{\mathcal{P}}$ and it is easy to see that the same sequence of interpretive steps are applied in $\mathcal{D}_{\mathcal{P}\Theta}$ thus leading to state $\langle \mathcal{Q}'\Theta; \theta\rangle$, which is not necessarily a sfca.

3. Finally, it suffices to instantiate the sfca $\langle \mathcal{Q}'; \theta\rangle$ in the first derivation $\mathcal{D}_{\mathcal{P}}$ with the symbolic substitution Θ, for completing both derivations with the same sequence of interpretive steps until reaching the desired fca $\langle v; \theta\rangle$.

□

Example 5. Consider again the multi-adjoint lattice L and the sMALP program $\mathcal{P}$ of Example 2. Let $\Theta = \{\leftarrow^{s_1}/\leftarrow_{\mathtt{P}}, \&^{s_1}/\&_{\mathtt{P}}, \&^{s_2}/\&_{\mathtt{G}}, v^s/0.8\}$ be a symbolic substitution. Given the sfca from Example 4, we have:

$$\langle \&^{s_1}(0.9, \&^{s_2}(v^s, 0.6))\Theta; \{X/a\}\rangle = \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, 0.6)); \{X/a\}\rangle$$

So, we have the following interpretive final derivation for the instantiated sfca:

$$\langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(0.8, 0.6)}); \{X/a\}\rangle \rightarrow_{IS} \langle \underline{\&_{\mathtt{P}}(0.9, 0.6)}; \{X/a\}\rangle \rightarrow_{IS} \langle 0.54; \{X/a\}\rangle$$

By Theorem 2, we have that $\langle 0.54; \{X/a\}\rangle$ is also a fca for $p(X)$ in $\mathcal{P}\Theta$.

3.3 Distance

In order to tune programs and get the symbolic substitution θ that minimize the sum of the erros, we have to include in our lattice a new predicate, that allows to compute the distance between to any elements of the lattice.

In other words, to compute the difference between the expected truth degree and the truth degree of a computed fuzzy answer, is necessary to include in the lattice associated to out symbolic program a new operation that calculates the distance between both elements. This concept is going to be formalized in the following definition.

Definition 15. Let X be a set. An application $d : XxX \rightarrow R$ defines a distance in X if, for all $x, y \in X$, the following conditions are satisfied:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) \quad | \quad d(y, z) \quad (tringle \quad inequality)$

For example, the most common distance in $\mathbb{R}$ is given by the absolute value, called *usual distance* and corresponding to $d(x, y) = |x - y|$. This is the one that will be considered along this work, denoted as *d(x,y)*.

3.4 Tuning algorithm

We summarize the automated technique for tuning multi-adjoint logic programs using sMALP programs initially presented in [27] and implemented in the FLOPER online tool as we are going to explain.
Consider a typical Prolog clause "$H : -B_1, \ldots, B_n$". It can be fuzzified in order to become a MALP rule "$H \leftarrow_{label} \mathcal{B}$ with v" by performing the following actions:

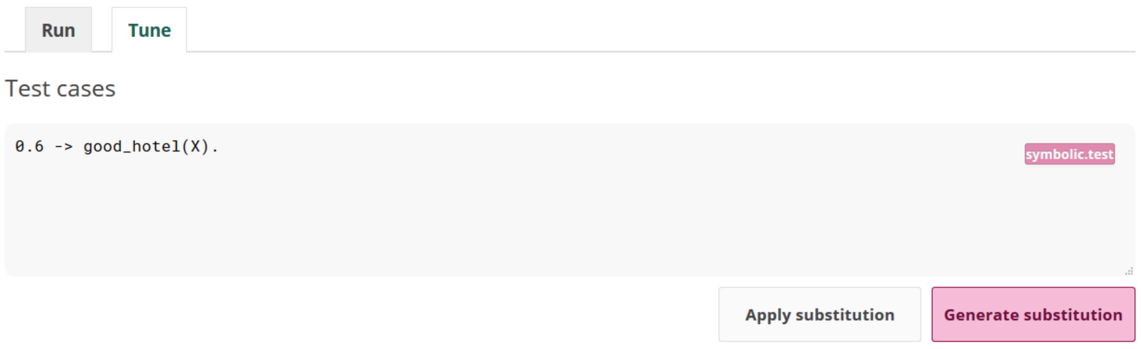
1. Weighting it with a truth degree v.

2. Connecting its head and body with a fuzzy implication symbol $\leftarrow_{label}$ (belonging to a concrete adjoint pair $\langle \leftarrow_{label}, \&_{label} \rangle$).

3. Linking the set of atoms $B_1, \ldots, B_n$ on its body $\mathcal{B}$ by means of a set of fuzzy connectives (i.e., conjunctions $\&_i$, disjunctions $|_j$ or aggregators $@_k$).

Introducing changes on each one of the three fuzzy components just described above may affect (sometimes in an unexpected way) the set of fuzzy computed answers for a given goal.

The first action for initializing the tuning process in the online tool obviously consists in introducing a set of test cases as shown in Figure 3.5. Each test case appears in a different line with syntax: r −> $\mathcal{Q}$, where r is the desired truth degree for the fca associated to query $\mathcal{Q}$ (which obviously does not contain symbolic constants). Before directly using the test cases introduced in the input box, the system firstly tries to refine them by automatically instantiating the variable symbols as much as possible.



```
0.6 -> good_hotel(X).
```

Figure 3.5: Screenshot of the online tool for starting the tuning process.

In our particular example, after introducing the test case `0.6−> popularity(X)`, the tool generates the three sfca's shown in Figure 3.6 and then uses their substitution components for instantiating the original query, thus changing the original test

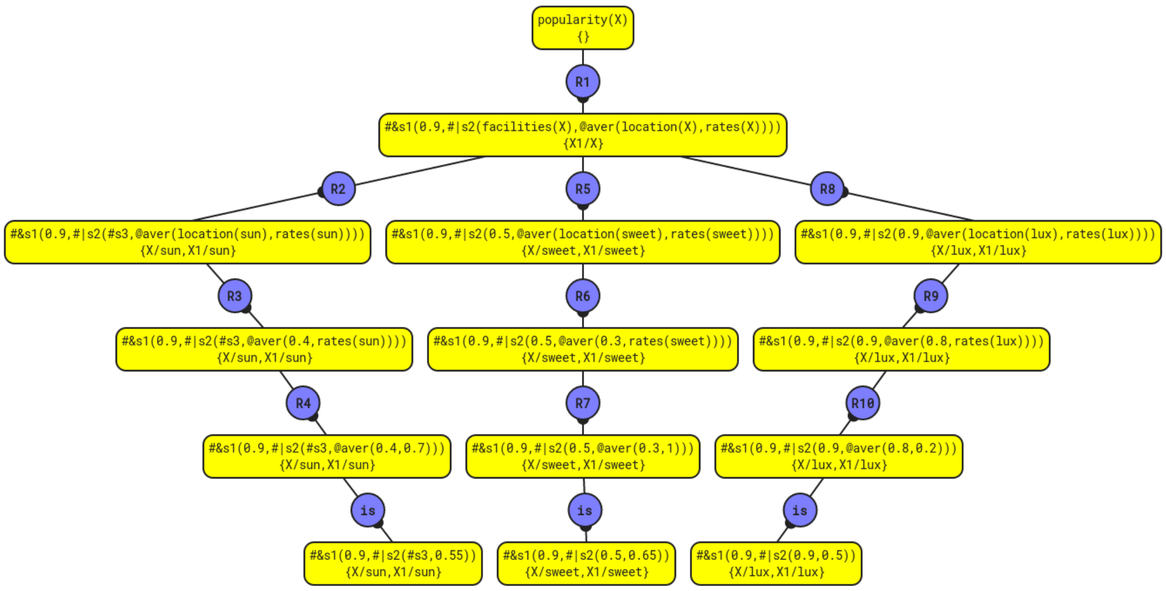

```
popularity(X)
{}
R1
#&s1(0.9,#|s2(facilities(X),@aver(location(X),rates(X))))
{X1/X}
R2
#&s1(0.9,#|s2(#s3,@aver(location(sun),rates(sun))))
{X/sun,X1/sun}
R3
#&s1(0.9,#|s2(#s3,@aver(0.4,rates(sun))))
{X/sun,X1/sun}
R4
#&s1(0.9,#|s2(#s3,@aver(0.4,0.7)))
{X/sun,X1/sun}
is
#&s1(0.9,#|s2(#s3,0.55))
{X/sun,X1/sun}
R5
#&s1(0.9,#|s2(0.5,@aver(location(sweet),rates(sweet))))
{X/sweet,X1/sweet}
R6
#&s1(0.9,#|s2(0.5,@aver(0.3,rates(sweet))))
{X/sweet,X1/sweet}
R7
#&s1(0.9,#|s2(0.5,@aver(0.3,1)))
{X/sweet,X1/sweet}
is
#&s1(0.9,#|s2(0.5,0.65))
{X/sweet,X1/sweet}
R8
#&s1(0.9,#|s2(0.9,@aver(location(lux),rates(lux))))
{X/lux,X1/lux}
R9
#&s1(0.9,#|s2(0.9,@aver(0.8,rates(lux))))
{X/lux,X1/lux}
R10
#&s1(0.9,#|s2(0.9,@aver(0.8,0.2)))
{X/lux,X1/lux}
is
#&s1(0.9,#|s2(0.9,0.5))
{X/lux,X1/lux}
```

Figure 3.6: Screenshot of the symbolic derivation tree generated by the online tool.

case by the following three: 0.6−> popularity(sun), 0.6−> popularity(sweet), and 0.6−> popularity(lux). Next, the user can manually update the truth degrees of some refined test cases.

Once the set of test cases has been appropriately customized, users simply need to click on the Generate substitution button for proceeding with the tuning process. The precision of the technique depends on the set of symbolic substitutions considered at tuning time.

For assigning values to the symbolic constant (starting with #), the tuning tool takes into account all the truth values defined on the members/1 predicate (which in our case is defined as members([0.3, 0.5, 0.8])) as well as the set of connectives defined in the lattice of Figure 3.1, which in our running example coincides with the three conjunction and disjunction connectives based on the so-called *Product*, *Gödel* and *Łukasiewicz* logics, as shown in Figure 3.7. Obviously, the larger the domain of values and connectives is, the more precise the results are (but the algorithm is more expensive, of course).

$$
\begin{array}{lclllcl}
\&_{\texttt{Product}}(x,y) & = & x*y & \quad & |_{\texttt{Product}}(x,y) & = & x+y-x*y \\
\&_{\texttt{Gödel}}(x,y) & = & \min(x,y) & & |_{\texttt{Gödel}}(x,y) & = & \max(x,y) \\
\&_{\texttt{Lukasiewicz}}(x,y) & = & \max(x+y-1,0) & & |_{\texttt{Lukasiewicz}}(x,y) & = & \min(x+y,1)
\end{array}
$$

Figure 3.7: Conjunctions and disjunctions of three different fuzzy logics over $\langle[0,1],\leq\rangle$.

The following definition resumes the tuning algorithm described in [27] and used for this project:

Definition 16 (algorithm for thresholded tuning of MALP programs)**.**

Input: an sMALP program $\mathcal{P}^s$ and a number of test cases $v_i \to Q_i$, $i = 1, \ldots, k$.

Output: a symbolic substitution Θ_τ.

1. For each test case $v_i \to Q_i$, compute the sfca $\langle Q'_i, \theta_i \rangle$ of $\langle Q_i, id \rangle$ in $\mathcal{P}^s$.
2. Then, consider a finite number of possible symbolic substitutions for $\mathcal{S}ym(\mathcal{P}^s)$, say $\Theta_1, \ldots, \Theta_n$, $n > 0$.
3. $\tau = \infty$; For each symbolic substitution $j \in \{1, \ldots, n\}$ and $\tau \neq 0$

```
        $z = 0$; For each test case $i = \{1, \ldots, k\}$ and $\tau > z$
                compute $\langle Q'_i \Theta_j, \theta_i \rangle \to^*_{IS} \langle v_{i,j}; \theta_i \rangle$
                let $z = z + |v_{i,j} - v_i|$.
        if $z < \tau$ then {  $\tau = z$;   $\Theta_\tau = \Theta_j$  }.
```

4. Finally, return the best symbolic substitution Θ_τ.

This algorithm makes use of a threshold τ for determining when a *partial* solution is acceptable. The value of τ is initialized to ∞ (in practice, a very huge number). Then, this threshold dynamically decreases whenever we find a symbolic substitution with an associated deviation which is lower that the actual value of τ. Moreover, a partial solution is discarded as soon as the cumulative deviation computed so far is greater than τ. In general, the number of discarded solutions grows as the value of τ decreases, thus improving the pruning power of thesholding.

Chapter 4

TRANSLATING NEURAL NETWORKS TO FUZZY LOGIC PROGRAMS

Now that we know the theoretical basis and context of this project, it is necessary to define a certain process, combining all the different elements in the proper way. As said in previous chapters, this project starts from the basic idea of *Neuro-Prolog* [11] that consists of neural networks working together with logic programs to create a *Neuro-Fuzzy system*, called *Neuro-FLOPER*. In this project, instead of working only with Prolog we are going to use The FLOPER Environment to deal with fuzzy logic rules, in order to create a symbiosis between neural networks created with Keras that allows to introduce and explore the capabilities of tuning with FLOPER. This functionality is going to be used for addressing some issues relating architecture of neural networks and how it is modificated in the training process.

First of all, is necessary to define rules and a certain context in order to translate a Keras model into a fuzzy logic program written in MALP that allows FLOPER to use tuning over the model. In order to get a more simple approach, is necessary to define an intermediate data structure between both representations, to make a little simple representation and data managing. There are a huge amount of different types of data that introduce one simple neural network, and in order to increase the simplicity of the concept that wants to be represented, only one kind of neural network architecture is going to be used, the *multilayer perceptron*. MLP probably is the most well-known architecture of neural networks, one of the most simple, and also one with great power in combination with *Backpropagation algorithm* [35]. For these three reasons, it is the one that has been chosen for this project.

In this chapter, there are some key points that are necessary to take into account for making the translation:

1. Activation functions: there are a huge amount of activation functions that have been created during the last fourty years. For this first version is necessary to make a selection of some of them.
2. Weights and bias: are two essential parts of every neuron. Is important to find an optimal representation in our intermediate data structure.
3. Structure of the network: for this version, is going to be as the universal representation of a multilayer perceptron, in which each node in one layer connects with a certain weight w_{ij} to every node in the following layer (except for nodes in the output layer).
4. Lattice: in order to work with fuzzy logic, is necessary to define a lattice where all our activation functions are going to be represented. The lattice defines the universe of our translated neural network.

The first point in our chapter starts with the last in the previous list. Now we introduce our lattice, the first element to represent neural networks as a fuzzy logic program.

4.1 Lattice definition

In this section, we are going to introduce the basic structure of a lattice used for tuning neural networks as fuzzy logic programs. In the next chapter we will use three different lattices for carrying on a set of experiments. Here we introduce a part of the lattice **L95** with not all the activation functions used in this work. The complete lattice appears in Appendix D.

The first point is the Elements paragraph. In this, we have to define a set of members included in our lattice that will be candidates for the substitution of symbolic constants. As FLOPER is based in fuzzy logic, we have to work with numbers using this way, that is quite different from other languages as for example Python.

```
% Elements
member(pos_inf).    member(neg_inf).    member(X) :- number(X).
members([0.95, 0.25, 0.50, -0.25, -0.95, -0.50]).
```

The next important point that has to be defined is the Distance. In our case we will work with the *usual distance* given by the absolute value of the difference between

Y and X.

```
% Distance
distance(X,Y,Z) :- (X=pos_inf;X=neg_inf;Y=pos_inf;Y=neg_inf),!,
    current_prolog_flag(max_tagged_integer,Z).
distance(X,Y,Z) :- Z is abs(Y-X).
```

The Ordering relation indicates the order of the members in our lattice.

```
% Ordering relation
leq(_,pos_inf).    leq(neg_inf,_).    leq(X,Y) :- X =< Y.
```

Also, we have to define the upper and lower bounds of the lattice, given by infinite and minus infinite.

```
% Supremum and infimum
bot(neg_inf).     top(pos_inf).
```

The Binary Operators correspond to the definition of T-norms and T-conorms. In this example are only defined two basic operators, the supremum and the infimum.

```
% Binary operations
or_sup(X,Y,Y) :- leq(X,Y).
or_sup(X,_,X).
and_sup(X,Y,X) :- leq(X,Y).
and_sup(_,Y,Y).
```

The basic aggregators of our lattice are the addition and the product. This two will make the basic structure of a neural network, and as we have seen are encapsulated activation functions.

```
% Aggregators
agr_add(pos_inf,_,pos_inf). agr_add(_,pos_inf,pos_inf).
agr_add(neg_inf,_,neg_inf). agr_add(_,neg_inf,neg_inf).
agr_add(X,Y,Z) :- Z is X+Y.
```

```

agr_prod(pos_inf,_,pos_inf). agr_prod(neg_inf,_,neg_inf).
agr_prod(_,pos_inf,pos_inf). agr_prod(_,neg_inf,neg_inf).
agr_prod(X,Y,Z) :- Z is X*Y.
```

The rest of the aggregators correspond to the different activation functions that we have. In the next piece of code we can find the *sigmoid* and the *softplus* function.

```
%Sigmoid
agr_sigmoid(pos_inf,pos_inf).
agr_sigmoid(neg_inf,neg_inf).
agr_sigmoid(X,Y) :- Y is 1 / (1+exp(-X)).

%Softplus
agr_softplus(pos_inf,pos_inf).
agr_softplus(neg_inf,neg_inf).
agr_softplus(X,Y) :- Y is log(1+exp(X)).
```

In the next subsection all of them that compounds the final lattice will be explained more in detail.

4.1.1 Activation functions

The way in which these functions are defined is very important in order to create a consistent lattice. For each one of them, is necessary to define boundaries in addition to the expression itself. The selection for this first version has been made taking into account popularity of each function in the scientific community, and if the expression includes some terms defined in training mode. This last fact is going to be a limitation of our Keras models, that will be explained later more in detail.

Now, all functions defined in our lattice are going to be presented:

- **Identity function**: this function (Figure 4.1) always return the same value that was used in its argument, and is given by $f(x) = x$. It does not help with the complexity or various parameters of usual data that is fed to the neural networks. For this aim is not used in MLP architectures, but we are going to introduce it in our lattice to test how tuning behaves. This function is going to be defined as a binary agregator, only taken 0 or 1 as possible values. The **range** of this function is $(-\infty\ to\ \infty)$.

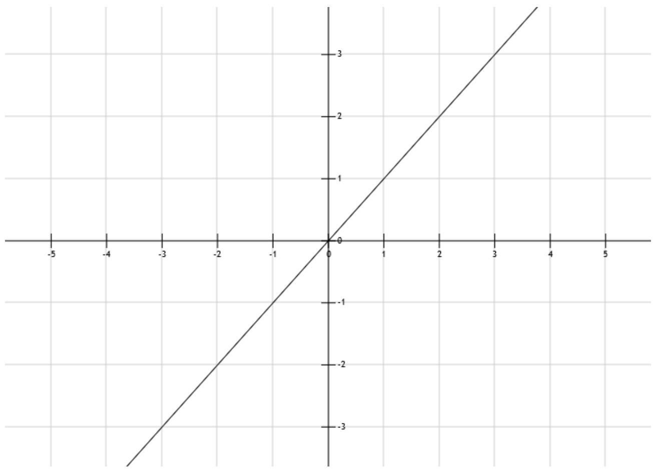


Figure 4.1: Graphical representation of Identity function.

- **Sigmoid function**: it is especially used for models that have to predict the **probability** as an output, because its range is $[0, 1]$. This function is differentiable (we can find the slope of the function curve at any two points), and is monotonic (but not its derivative).

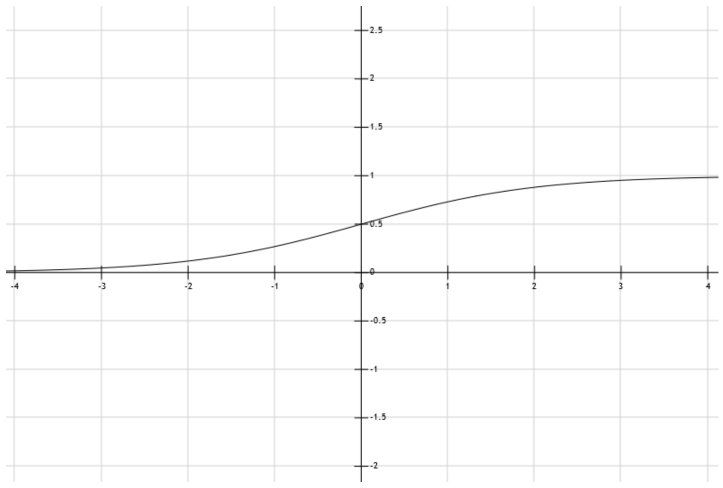


Figure 4.2: Graphical representation of Sigmoid function.

Their function is graphically represented in Figure 4.2, and its expression is given by:

$$f(x) = \frac{1}{1+e^{-x}}$$

- **Softplus function**: the main difference respect other functions is that the range of softplus is in the scale of $(0, \infty)$. The expression of this function is:

$$f(x) = ln(1+e^{x})$$

This function is less used than others, as for example ReLU, because compute $log(.)$ and $exp(.)$ is not so efficient. Its derivative corresponds to the sigmoid function. The graphical representation is given by Figure 4.3.

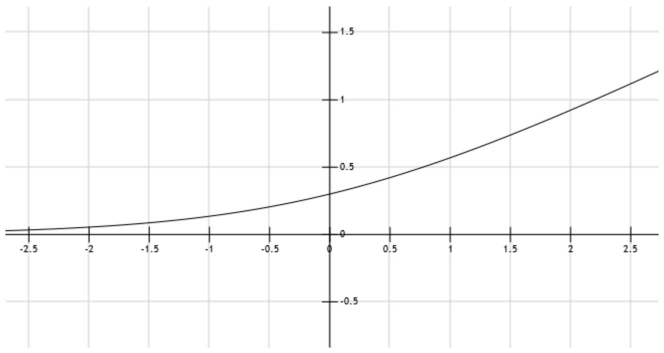


Figure 4.3: Graphical representation of Softplus function.

- **Softsign function**: this function (Figure 4.4) is not adopted in practice as much as tanh, its main alternative, and this makes it uncommon. But for some problems, specially in image classification, softsign is more effective [4].

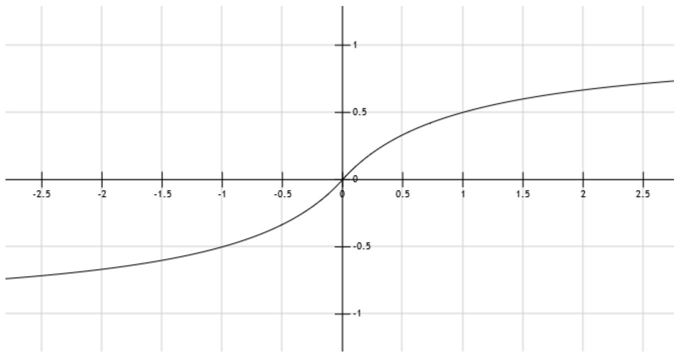


Figure 4.4: Graphical representation of Softsign function.

The range is in the scale of $[-1, 1]$ and its expression is:

$$f(x) = \frac{1}{1+|x|}$$

- **Rectified Linear Unit (ReLU)**: is the most used activation function in the field of neural networks, being in a huge amount of architectures as, for example, convolutional neural networks. Its range is $[0, \infty)$, as we can see in Figure 4.5. This function is given by the following expression:

$$\begin{cases} x < 0, \ y = 0 \\ x \geq 0, \ y = x \end{cases}$$

  Its derivative and the original expression **are monotonic**. One of the most important issues for this function is that negative values become zero inmediately, and they are not mapped propertly in the graph. This is the reason for which the ability of the model to fit or train from data decreases.

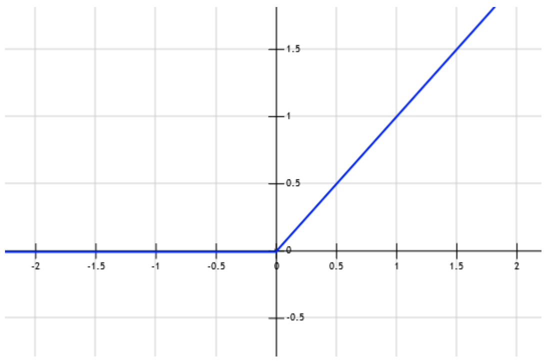


Figure 4.5: Graphical representation of ReLU function.

- **Leaky ReLU**: this function is an attempt to solve the dying ReLU problem and to map negative values more propertly. For values less than 0, coordinate x is multiplied by a *leak*, by default 0.01, as we can see in the expression:

$$\begin{cases} x < 0, \ y = 0.01x \\ x \geq 0, \ y = x \end{cases}$$

  When the *leak* is not 0.01, the function is called **Randomized Relu**. For *Neuro-FLOPER*, only the original version of this function (as seen in Figure 4.6) is going to be implemented.

- **Hyperbolic Tangent (tanh)**: this function is sigmoidal and has a range of $[-1, 1]$, as we can see in Figure 4.7. Is differentiable and monotonic (but not its derivative, the same as for the sigmoid function). The main advantage is that negative inputs will be mapped strongly negative, and zero inputs will be mapped near zero in its graph. Is mainly used in classification problems with two classes.

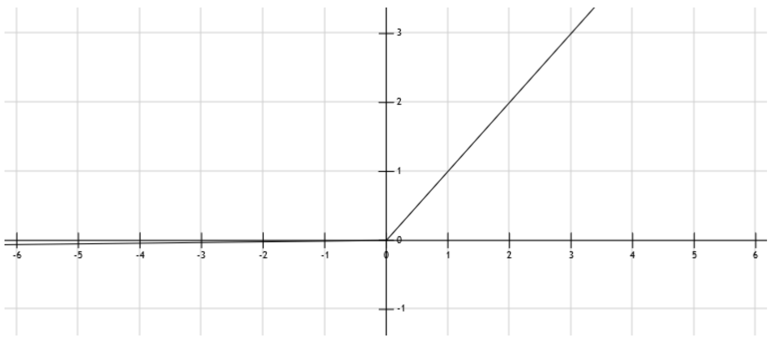


Figure 4.6: Graphical representation of Leaky ReLU function.

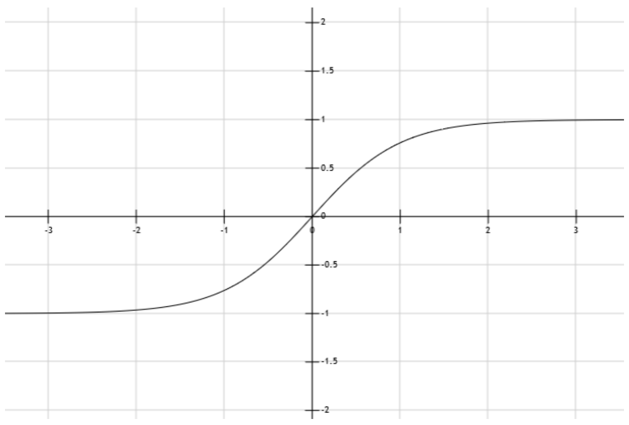


Figure 4.7: Graphical representation of tanh function.

- **Inverse tangent (arctan)**: is very similar to sigmoid and tanh, but arctan (Figure 4.8) is slightly flatter and due to this reason this funtion may have a little better ability to discriminate between similar input values. Its range is in $[\frac{\pi}{2}, \frac{\pi}{2}]$.

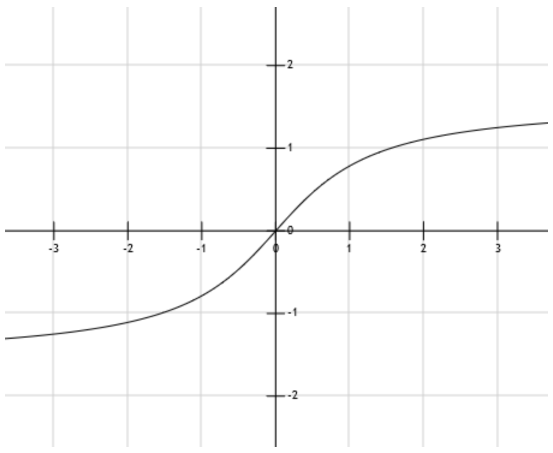


Figure 4.8: Graphical representation of Arctan function.

- **Sinusoid**: periodic functions as cosine are not very popular in neural networks, but it would be saturated when input increases positively or negatively just like other common activation functions such as tanh or sigmoid.

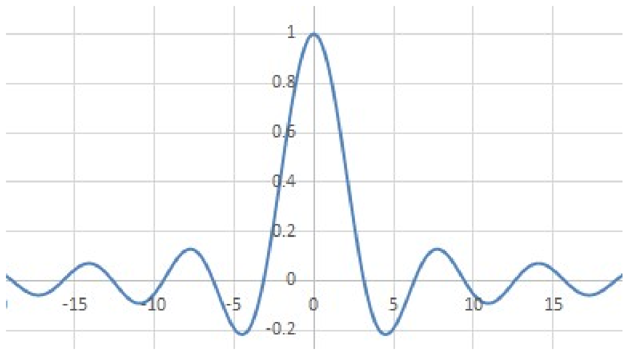


Figure 4.9: Graphical representation of Sinusoid function.

Sinusoid function (Figure 4.9) is defined as *sine of x* over x, and is important to know that this kind of function is undefined for x equals to 0, so is necessary to define an exception in this point, being by default 1.

$$\begin{cases} x = 0, \ y = 1 \\ else \ f(x) = \frac{sin(x)}{x} \end{cases}$$

4.2 Model representation

First of all, is essential to know some particularities of Keras models, in order to distinguish between what points are useful in our translation. A Keras model can be represented as a graph, as seen in 2.2.2. In this brief representation shown in Figure 4.10 (that can be displayed in horizontal or vertical), we can see at the top of each node the type of layer. For Keras, *Dense* means fully connected, in other words, a typical layer of a multilayer perceptron in which each node is connected with all the nodes of the next layer.

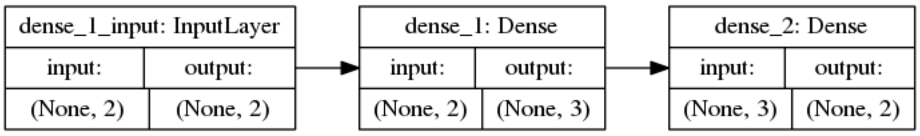


Figure 4.10: Graphical representation of a multilayer perceptron in Keras.

In the next level, two columns are displayed in a very similar way, denoting the labels *inputs* and *outputs*. Each tuple at the top of a node indicates in its first parameter dimensionality of the batch size (*None* means that the dimension is variable). The second corresponds to the number of arches that each neuron in a layer gets for the *inputs*, and the number of neurons that a layer has for the *outputs*.

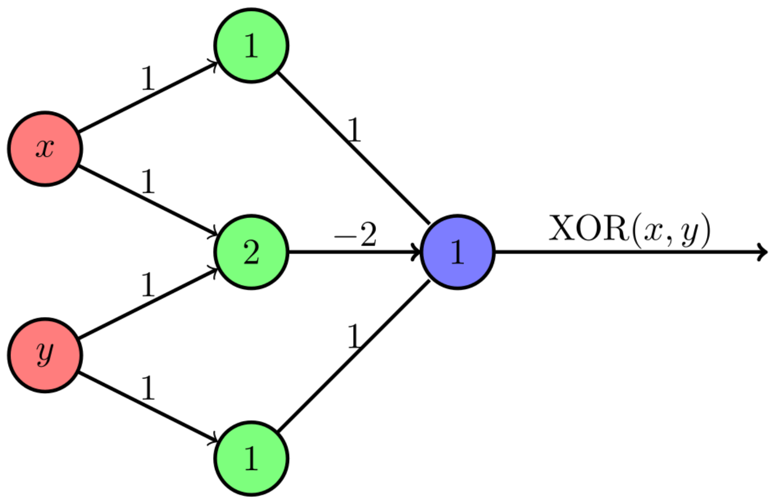


Figure 4.11: Representation of the XOR MLP as a graph.

This representation is the translation of the model proposed in Figure 4.11 to a Keras model. As we can see, Keras display of a representation of each model, but some of the essential points of a neural network can be accessed easily by the programmer, in contrast to other libraries as for example Tensorflow. For the translation process, two fundamental methods are going to be used:

- *model.layer.get_weights*(): it returns a numpy array in which each component has two different arrays for a layer, one for weights and other for bias.
- *model.layer.activation*: returns an object containing the name of the activation function used in a certain layer.

Now that we know how to get all the required parameters that define an artificial neural network, is required to define an intermediate structure to represent the whole structure in a proper way. This structure is composed of three levels that correspond to different abstractions of a neural networks architecture. It starts with the word **node**, which represents all the information of each node:

- **id_n**: the name of the node (neuron), translated to *FLOPER* sintax. An example would be *node_1_2(.)*, this id indicates that the node is the first node of the second layer. If the node corresponds to a node of an output layer, it would have the name of the class that represents, for example *rain_probability(.)* symbolize the class with the same name for the problem that wants to be approached.
- **p_nodes**: array with parent nodes, represents each input in a neuron and each component of the array is a tuple with two components, the id of the parent node and the weight of the input.

- **bias**: float number corresponding with the bias of the neuron.

With this first level, is possible to define a set of neurons (layer) and determinate a more general abstraction with a collection of key components. A layer is defined as:

- **n_layers**: number of the given layer inside the network.
- **n_nodes**: number of nodes in the layer.
- **nodes**: an array of nodes that build the layer.
- **activation**: name of the activation function. It has to be equal for all the nodes in the layer.

Finally, the last level corresponds to the network itself, and is defined by the following parameters:

- **input_f**: input layer of the network. In practice it will be used only for knowing the number of inputs for the neural network.
- **hidden_fs**: an array of layers with all the hidden layers of the network.
- **output_f**: output layer of the network.

This data structure is only valid for representing a multilayer perceptron as a fuzzy logic program. Is an intermediate step in the process of translation to order all the necessary information and prepare some components in a valid format for *FLOPER*.

The final step of the translation corresponds to transform all the content given by the intermediate data structure into rules. Due to this fact, is not necessary to declare explicitly a rule for each neuron in input nodes, so they will be codified **always** in the first hidden layer. An example will be:

```
node_1_2(X1, X2, X3, X4)<- @relu( @add(0.6244668, @add ( @prod
    (0.70569754,td(X2)), @add(@prod(-0.3849561,td(X4)), @add(@prod
    (0.731073,td(X1)),@prod(-0.7602397,td(X3))))))).
```

With this rule we can know that corresponds to the first neuron of the second layer (first hidden layer), which receives four inputs directly of the input layer, as we can see in each parent neuron defined as $td(Xi)$, being i the number of neurons in the input layer. The activation function is ReLU, given by the aggregator $@relu$, and also we can find the aggregator of the product (*@prod*) and for the binary sum (*@add*). The first number, in this case 0.6244668 defines the bias of the network, and the rest that come along with each parent node are weights for the different inputs of the neuron.

For other neurons, the corresponding rules are quite similar. The next example corresponds to a neuron in the output layer, and represents the class $iris_{setosa}$. The rule was shortened in order to make it more readable.

```
iris_setosa(X1, X2, X3, X4)<- @sigmoid( @add(-0.011225972, @add ( @prod
    (1.4837127,node_3_2(X1, X2, X3, X4)), ...)))))).
```

As we can see, in this example all the ids follow the principles defined before in our intermediate data structure. This is the unique structural change with respect to the previous example. In this case the activation function is the sigmoid (*@sigmoid*) and the bias for this neuron corresponds to the negative value −0.011225972.

4.3 Model limitations

For tuning a neural network with our first version of *Neuro-FLOPER*, is necessary to know all the limitations that this application has. As we have said in previous sections, the structure of a neural network that is going to be tuned with *Neuro-FLOPER* has to follow the classical architecture of a *multilayer perceptron*. In other words, the minimum number of layers has to be three, one for receive inputs, a hidden layer and an output layer. Between layers, each neuron has to be connected with all the neurons of the previous layer. Also, each neuron has to have a *bias*.

That kind of structure can be easily defined in *Keras* using some methods provided by this framework. First, is necessary to define our model (neural network) as a *sequential*, meaning that is a linear stack of layers.

```
model = Sequential()
```

Then, we have to add some layers to our model, indicating the number of neurons in the layer, the activation function, and, if the layer corresponds to the first hidden one, is necessary to specify the dimensionality of the input (input_dim) or the number of neurons in the input layer. The following lines of code show how to create a neural network of three layers with four, eight and three neurons in each one of the layers respectively.

```
model.add(Dense(8, input_dim=4, activation='relu'))
model.add(Dense(3, activation='sigmoid'))
```

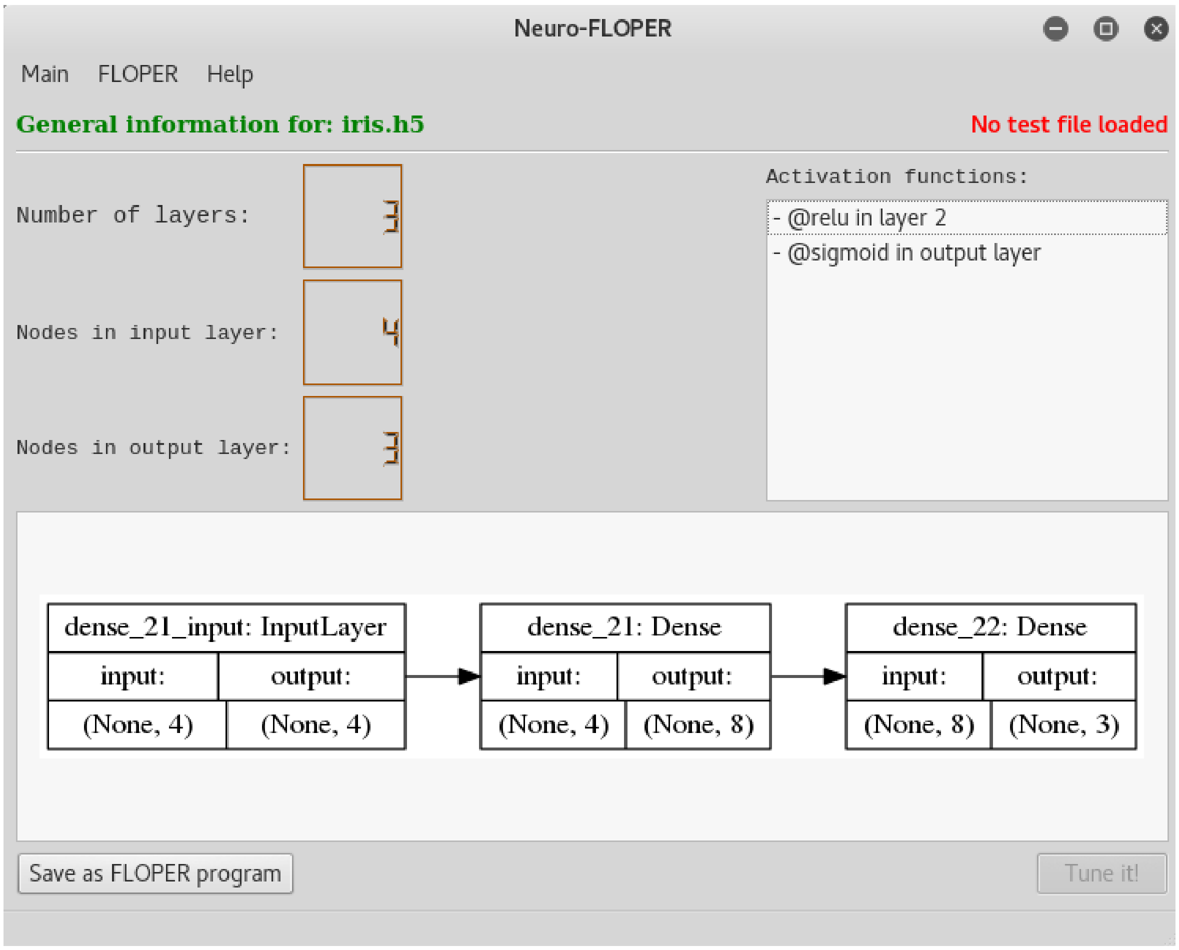
For creating the layers we have used the *Dense* method, indicating that each neuron in the layer is going to be connected with all the neurons of the previous one, or that the layer is going to be *fully connected* with its predecessor.

Probably, the most important point for this first version is the existence of a delimited set of activation functions. Some of them are included in *Keras*, but others are not included in the original library as for example the sinusoid function seen in Section 4.1.1, because is interesting to see if some of the non-common functions are the best option for tuning a neural network. Also, there are functions that are included in *Keras* but not in the first version of Neuro-FLOPER, as for example ELU. This kind of functions have learnable parameters, so this is the main reason for which are not included because we only want the classical functions.

Finally, we have to compile the model before training it. This point can be done using the method *compile* provided by *Keras*. All the code used in this section is a part of the one used for creating a neural network in order to address the well-known dataset *Iris*, and can be found in Appendix A.

4.4 Implementation of translation

For making the translation between a *Keras* model and a fuzzy logic program, is necessary to follow the steps provided in *Restructure_Floper.py* (see Appendix E). For starting the translation process is essential to save the neural network as a *.h5* file, and the labels in a *.txt* in the same order as the output neurons from top to bottom. Both files will be opened with Neuro-FLOPER and it will create an intermediate data structure with the architecture of the trained neural network. It is recommended to use a similar structure as shown in Appendix A for creating and training a neural network with *Keras*, in order to get the two necessary files for Neuro-FLOPER. Following the example of the neural network trained with the *Iris* dataset, the result of load it in Neuro-FLOPER can be seen in Illustration 4.12.



General information for: iris.h5

No test file loaded

Number of layers: 3

Nodes in input layer: 4

Nodes in output layer: 3

Activation functions:
- @relu in layer 2
- @sigmoid in output layer

dense_21_input: InputLayer	
input:	output:
(None, 4)	(None, 4)

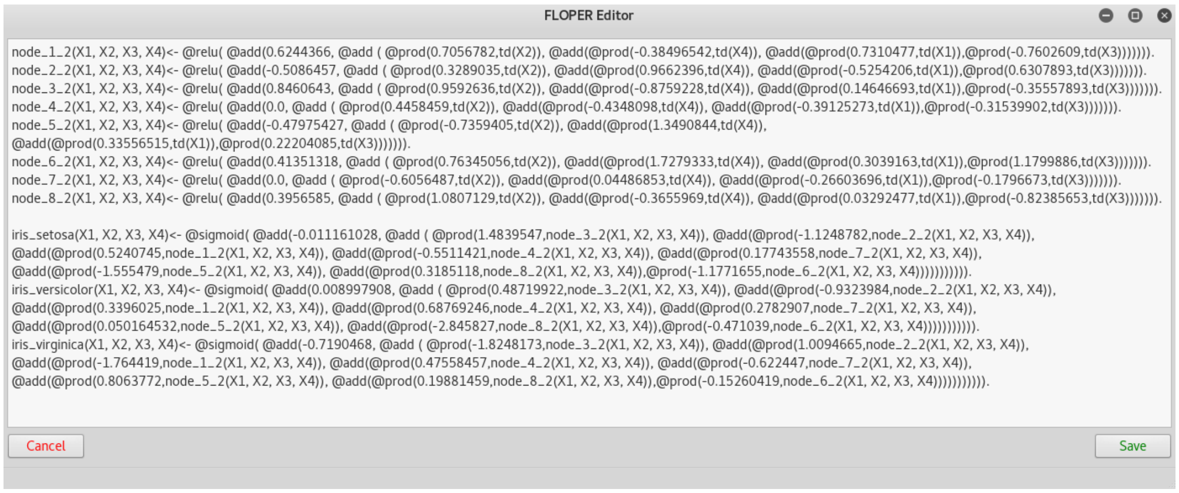
dense_21: Dense	
input:	output:
(None, 4)	(None, 8)

dense_22: Dense	
input:	output:
(None, 8)	(None, 3)

Figure 4.12: Neuro-FLOPER with iris.h5 and labels.txt loaded.

Then, the process starts with the creation of an input layer. This layer only has the id $td(Xi)$ of each input, where i is the number of the neuron in the layer. When all the inputs are saved in a unique layer, the system starts encoding each one of the hidden layers. The id of a neuron in a hidden layer corresponds to the head of a rule, and the body is a combination of an activation function as an aggregator followed by the bias and the rest of the weights codified with the binary addition and multiplication defined in our lattice. Finally, the creation of the last layer is very similar to the hidden ones, but in this case the labels contained in the *.txt* file are used for naming each one of the output neurons. This is for making the final program more readable and gives representative outputs.

Each node will be a rule in the final fuzzy program, starting only with hidden layers because inputs can be encapsulated inside them. Our transformed program appears in Illustration 4.13, which corresponds to a screenshot of the "FLOPER Editor" tool included in Neuro-FLOPER.



```
node_1_2(X1, X2, X3, X4)<- @relu( @add(0.6244366, @add ( @prod(0.7056782,td(X2)), @add(@prod(-0.38496542,td(X4)), @add(@prod(0.7310477,td(X1)),@prod(-0.7602609,td(X3))))))).
node_2_2(X1, X2, X3, X4)<- @relu( @add(-0.5086457, @add ( @prod(0.3289035,td(X2)), @add(@prod(0.9662396,td(X4)), @add(@prod(-0.5254206,td(X1)),@prod(0.6307893,td(X3))))))).
node_3_2(X1, X2, X3, X4)<- @relu( @add(0.8460643, @add ( @prod(0.9592636,td(X2)), @add(@prod(-0.8759228,td(X4)), @add(@prod(0.14646693,td(X1)),@prod(-0.35557893,td(X3))))))).
node_4_2(X1, X2, X3, X4)<- @relu( @add(0.0, @add ( @prod(0.4458459,td(X2)), @add(@prod(-0.4348098,td(X4)), @add(@prod(-0.39125273,td(X1)),@prod(-0.31539902,td(X3))))))).
node_5_2(X1, X2, X3, X4)<- @relu( @add(-0.47975427, @add ( @prod(-0.7359405,td(X2)), @add(@prod(1.3490844,td(X4)),
@add(@prod(0.33556515,td(X1)),@prod(0.22204085,td(X3))))))).
node_6_2(X1, X2, X3, X4)<- @relu( @add(0.41351318, @add ( @prod(0.76345056,td(X2)), @add(@prod(1.7279333,td(X4)), @add(@prod(0.3039163,td(X1)),@prod(1.1799886,td(X3))))))).
node_7_2(X1, X2, X3, X4)<- @relu( @add(0.0, @add ( @prod(-0.6056487,td(X2)), @add(@prod(0.04486853,td(X4)), @add(@prod(-0.26603696,td(X1)),@prod(-0.1796673,td(X3))))))).
node_8_2(X1, X2, X3, X4)<- @relu( @add(0.3956585, @add ( @prod(1.0807129,td(X2)), @add(@prod(-0.3655969,td(X4)), @add(@prod(0.03292477,td(X1)),@prod(-0.82385653,td(X3))))))).

iris_setosa(X1, X2, X3, X4)<- @sigmoid( @add(-0.011161028, @add ( @prod(1.4839547,node_3_2(X1, X2, X3, X4)), @add(@prod(-1.1248782,node_2_2(X1, X2, X3, X4)),
@add(@prod(0.5240745,node_1_2(X1, X2, X3, X4)), @add(@prod(-0.5511421,node_4_2(X1, X2, X3, X4)), @add(@prod(0.17743558,node_7_2(X1, X2, X3, X4)),
@add(@prod(-1.555479,node_5_2(X1, X2, X3, X4)), @add(@prod(0.3185118,node_8_2(X1, X2, X3, X4)),@prod(-1.1771655,node_6_2(X1, X2, X3, X4)))))))))))).
iris_versicolor(X1, X2, X3, X4)<- @sigmoid( @add(0.008997908, @add ( @prod(0.48719922,node_3_2(X1, X2, X3, X4)), @add(@prod(-0.9323984,node_2_2(X1, X2, X3, X4)),
@add(@prod(0.3396025,node_1_2(X1, X2, X3, X4)), @add(@prod(0.68769246,node_4_2(X1, X2, X3, X4)), @add(@prod(0.2782907,node_7_2(X1, X2, X3, X4)),
@add(@prod(0.050164532,node_5_2(X1, X2, X3, X4)), @add(@prod(-2.845827,node_8_2(X1, X2, X3, X4)),@prod(-0.471039,node_6_2(X1, X2, X3, X4)))))))))))).
iris_virginica(X1, X2, X3, X4)<- @sigmoid( @add(-0.7190468, @add ( @prod(-1.8248173,node_3_2(X1, X2, X3, X4)), @add(@prod(1.0094665,node_2_2(X1, X2, X3, X4)),
@add(@prod(-1.764419,node_1_2(X1, X2, X3, X4)), @add(@prod(0.47558457,node_4_2(X1, X2, X3, X4)), @add(@prod(-0.622447,node_7_2(X1, X2, X3, X4)),
@add(@prod(0.8063772,node_5_2(X1, X2, X3, X4)), @add(@prod(0.19881459,node_8_2(X1, X2, X3, X4)),@prod(-0.15260419,node_6_2(X1, X2, X3, X4)))))))))))).
```

Figure 4.13: Transformed program showed in the FLOPER Editor.

The final program is represented in Appendix B, in which we can see twelve rules representing the hidden layer and the output layer with eight and four neurons respectively.

// Chapter 5

TUNING NEURAL NETWORKS AS FUZZY LOGIC PROGRAMS

In this chapter, all contents created in the previous ones for this Final Year Project are going to be taken into account to make a new implementation of the threshold tuning algorithm seen in Section 3.4. We will introduce a part of Neuro-FLOPER capable of load test cases from the well-known *.csv* files for storing databases. Later, there are two types of components inside our fuzzy logic programs that we will study in terms of execution time and given results by the tuning process.

5.1 Generating test cases

For making the tuning process in Neuro-FLOPER is necessary to implement some capabilities on it. First, is essential to have a set of test cases in a certain format that can be understood by the FLOPER environment. As seen in Chapter 3, each test case has to follow this syntax:

```
1 -> iris_setosa(5.1,3.5,1.4,0.2).
```

This test case can be read as "is true at 100% that this element belongs to the class iris setosa with parameters (5.1,3.5,1.4,0.2)". This is a useful way of representing this kind of information, but unfortunately there is no a high amount of resources in this format.

As said before, one of the most common formats for representing databases and high amounts of information is *.csv*, and this type of format can be translated to our syntax. For making this, Neuro-FLOPER implements an option named *Load test cases from .csv* that we can see in Figure 5.1.

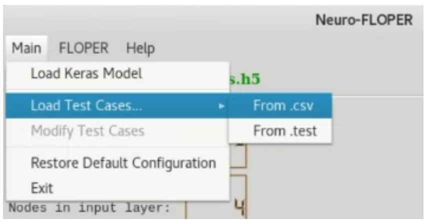
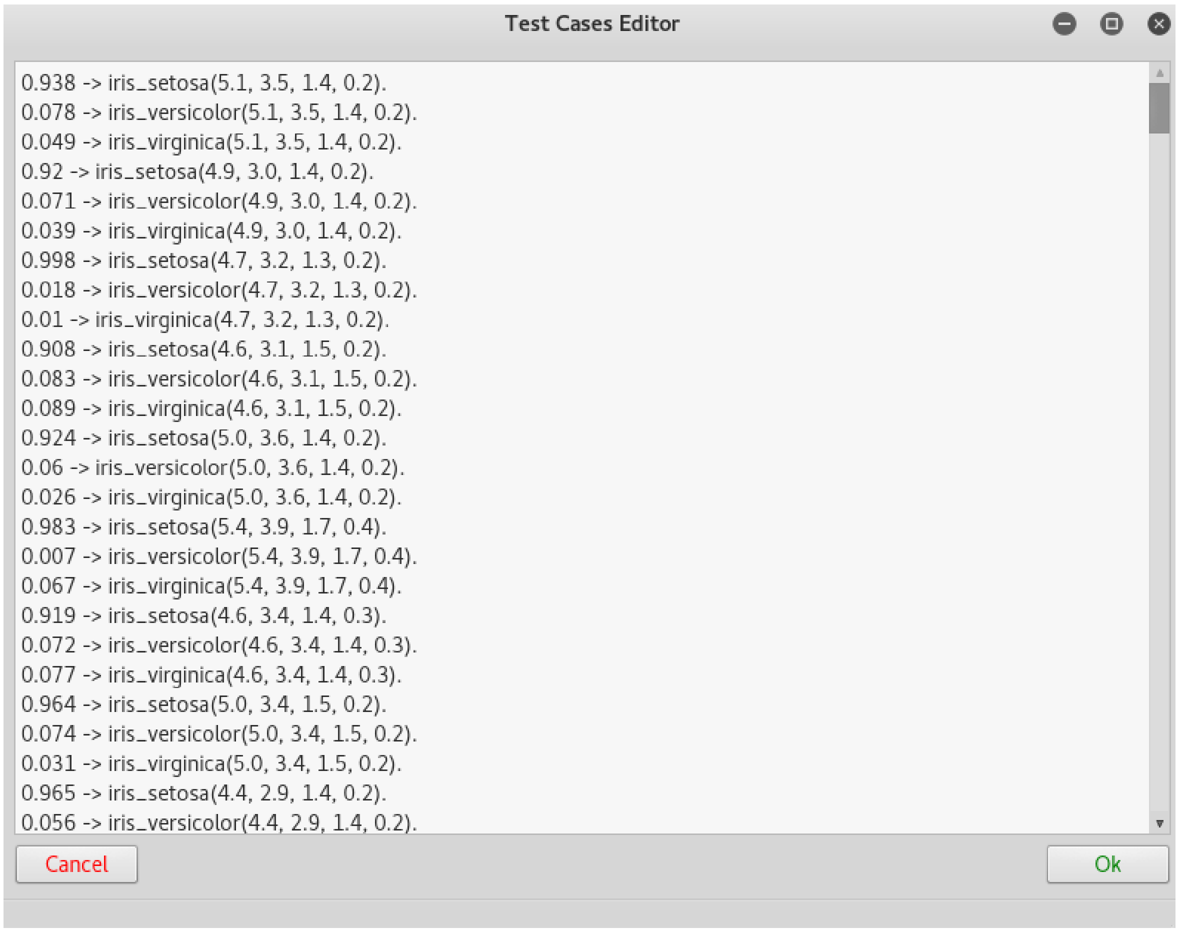


Figure 5.1: An option of Neuro-FLOPER for loading test cases from a .csv file.

For the following experiments, we are going to load different versions of the *Iris* dataset, in which each one of them corresponds to small versions of the original one, with all the elements chosen randomly. In Appendix C there is the complete dataset, with 449 different rules and with *.test* extension. This will be our *.test* file for all the performed experiments.

Neuro-FLOPER, for translating a .csv allows the user to define a trustability percentage for the data contained in the file. This factor will be used to make the translation, given random numbers to each component in the expected range. For example, if we introduce a **trustability factor** of 0.90 for a certain *.csv* file, the program will assign values between 0.9 and 1 for each register that belong to a class, and values between 0 and 0.1 for those who do not belong to the class with the same set of variables. In the following Illustration (5.2) appears a screenshot of the "Test Cases Editor" included in Neuro-FLOPER, and we can see the final result of a translated *.csv* file.



```
0.938 -> iris_setosa(5.1, 3.5, 1.4, 0.2).
0.078 -> iris_versicolor(5.1, 3.5, 1.4, 0.2).
0.049 -> iris_virginica(5.1, 3.5, 1.4, 0.2).
0.92 -> iris_setosa(4.9, 3.0, 1.4, 0.2).
0.071 -> iris_versicolor(4.9, 3.0, 1.4, 0.2).
0.039 -> iris_virginica(4.9, 3.0, 1.4, 0.2).
0.998 -> iris_setosa(4.7, 3.2, 1.3, 0.2).
0.018 -> iris_versicolor(4.7, 3.2, 1.3, 0.2).
0.01 -> iris_virginica(4.7, 3.2, 1.3, 0.2).
0.908 -> iris_setosa(4.6, 3.1, 1.5, 0.2).
0.083 -> iris_versicolor(4.6, 3.1, 1.5, 0.2).
0.089 -> iris_virginica(4.6, 3.1, 1.5, 0.2).
0.924 -> iris_setosa(5.0, 3.6, 1.4, 0.2).
0.06 -> iris_versicolor(5.0, 3.6, 1.4, 0.2).
0.026 -> iris_virginica(5.0, 3.6, 1.4, 0.2).
0.983 -> iris_setosa(5.4, 3.9, 1.7, 0.4).
0.007 -> iris_versicolor(5.4, 3.9, 1.7, 0.4).
0.067 -> iris_virginica(5.4, 3.9, 1.7, 0.4).
0.919 -> iris_setosa(4.6, 3.4, 1.4, 0.3).
0.072 -> iris_versicolor(4.6, 3.4, 1.4, 0.3).
0.077 -> iris_virginica(4.6, 3.4, 1.4, 0.3).
0.964 -> iris_setosa(5.0, 3.4, 1.5, 0.2).
0.074 -> iris_versicolor(5.0, 3.4, 1.5, 0.2).
0.031 -> iris_virginica(5.0, 3.4, 1.5, 0.2).
0.965 -> iris_setosa(4.4, 2.9, 1.4, 0.2).
0.056 -> iris_versicolor(4.4, 2.9, 1.4, 0.2).
```

Figure 5.2: A .csv file translated into a .test file with a trustability factor of 0.90.

Also, Neuro-FLOPER allows to load normal *.test* files as seen in Figure 5.3 that were created previously by the programmer, or translated from a *.csv*.

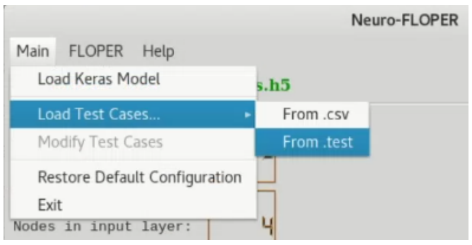


Figure 5.3: An option of Neuro-FLOPER for loading test cases from a .test file.

5.2 Tuning based on weights

Two important parts of an artificial neural network that can be tuned with this system are *weights* and *bias*. Each one of them can be easily replaced with symbolic constants like $\#n1$, in order to get a symbolic program and start tuning. This may be very useful for correcting some parts of the neuron that do not train as expected, producing unwanted effects in our neural network.

In our trained network with the *Iris* dataset, that we can find in Appendix B, we can see two neurons inside the hidden layer with suspicious *bias*. In the next portion of code where they appear, we can see that *bias* is exactly 0.0 for both of them, and this could be a mark of problems in the original training. Bias are represented in the first number of the body, after $@relu(@add$.

```
node_4_2(X1, X2, X3, X4)<- @relu( @add(0.0, @add ( @prod(0.4458459,td(X2)
    ), @add(@prod(-0.4348098,td(X4)), @add(@prod(-0.39125273,td(X1)),@prod
    (-0.31539902,td(X3))))))).

node_7_2(X1, X2, X3, X4)<- @relu( @add(0.0, @add ( @prod(-0.6056487,td(X2
    )), @add(@prod(0.04486853,td(X4)), @add(@prod(-0.26603696,td(X1)),
    @prod(-0.1796673,td(X3))))))).
```

The bias value allows the activation function to be shifted to the right or left, to better fit the data. Hence changes to the weights alter the steepness of the function curve, whilst the bias offsets it, shifting the entire curve so it fits better. Note also that the bias only influences the output values.

For studying if there is a better option for both *bias*, we are going to define a symbolic program named **sNN_2** with only two symbolic constants, one for each *bias*. In addition, there will be two more symbolic programs that modify some random weights. These programs are:

- **sNN_3**: symbolic program with three symbolic constants.
- **sNN_4**: symbolic program with four symbolic constants.

In both cases are included the symbolic constants defined in **sNN_2**. In total there are 56 weights in our neural network, 32 in the hidden layer and 24 in the output layer. Also, there are 12 bias, one for each neuron of the hidden layer (8) and the output one (4). The number of constants with possible errors in the first training given by *Keras* is approximately a 3% of the total, being constants the sum of weights and bias.

5.3 Tuning based on activation functions

This type of tuning has a very simple implementation in our neural network. It is true that in the fuzzy logic representation we could find different activation functions inside the same layer after applying tuning defining different symbolic aggregators for the same layer. But, respecting the original structure of a multilayer perceptron, for our network only must be two symbolic aggregators at most.

In our experiment, there will be three different symbolic programs in which there exist only symbolic aggregators, that corresponds to:

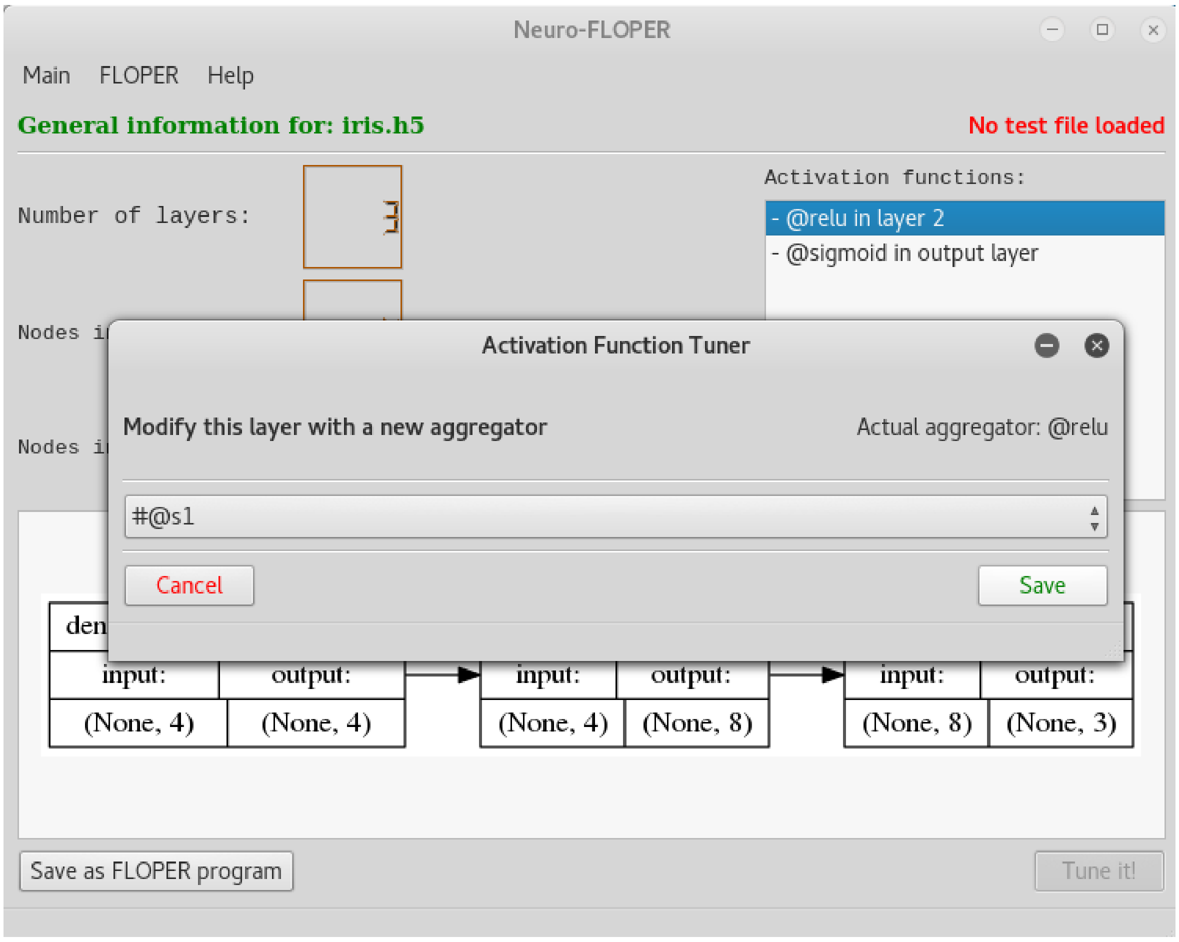
- **sNN_s1**: only a symbolic aggregator placed in the hidden layer, in substitution of the *relu* function.
- **sNN_s2**: only a symbolic aggregator placed in the output layer, in substitution of the *sigmoid* function.
- **sNN_s1s2**: this symbolic program is the combination of the previous ones, in which there are no activation functions.

In the next example, there is a rule (neuron) with a symbolic aggregator. This example belongs to the symbolic program **sNN_s1**, more concretely to the first neuron of the hidden layer.

```
node_1_2(X1, X2, X3, X4)<- #@s1( @add(0.6244668, @add ( @prod(0.70569754,
    td(X2)), @add(@prod(-0.3849561,td(X4)), @add(@prod(0.731073,td(X1)),
    @prod(-0.7602397,td(X3))))))).
```

All symbolic aggregators are expressed like #@$s1$, being $s1$ the name of the aggregator. Neuro-FLOPER allows the user to modify intuitively the activation functions of each layer. They can be changed by functions showed in Section 4.1 or by symbolic aggregators with only clicking each one of the layers showed in the section named as "Activation functions", inside the main window of Neuro-FLOPER.

Then, a small window is shown to the user with a Combo Box divided into three different sections. The first one contains the defined activation functions, the second one the symbolic aggregators that have been used in other layers (and due to this reason it can be empty), and finally a small section with only one symbolic aggregator that is free to be used. In Illustration 5.4 there is the Combo Box selecting one symbolic aggregator.


Neuro-FLOPER

General information for: iris.h5

No test file loaded

Number of layers: 3

Activation functions:
- @relu in layer 2
- @sigmoid in output layer

Activation Function Tuner

Modify this layer with a new aggregator

Actual aggregator: @relu

#@s1

Figure 5.4: Symbolic aggregator selected for tune the first hidden layer.

5.4 Experimental results

Before starting with our experiments defined in Sections 5.2 and 5.3, is essential to check if our model created with *Keras* and the translated program are **equal** and give the same results. This validation is made calling to *Keras* for getting the outputs for each one of the classes (as a numpy array), and using the online interpreter for testing the result of each output with the defined values for the four different inputs.

We have created three different experiments to validate our translation, each one with fixed values for the inputs $X1$, $X2$, $X3$ and $X4$. The tuples used for the inputs can be found in the *Iris* original dataset, and the results obtained are without applying normalization.

- **Inputs tuple (4.4,3.2,1.3,0.2):** this experiment uses a case that belongs to the class *iris setosa*. The results are shown in Table 5.1.

Class	**Keras**	**FLOPER**
iris_setosa	0.90089	0.90089
iris_versicolor	6.39109e-04	6.39109e-04
iris_virginica	3.69933e-08	3.69933e-08

Table 5.1: Output results for (4.4,3.2,1.3,0.2)

- **Inputs tuple (6.2,2.9,4.3,1.3):** this case belongs to the class *iris versicolor*. In Table 5.2 we can see the results.

Class	**Keras**	**FLOPER**
iris_setosa	7.90427e-07	7.90427e-07
iris_versicolor	1.14727e-02	1.14727e-02
iris_virginica	1.14912e-04	1.14912e-04

Table 5.2: Output results for (6.2,2.9,4.3,1.3)

- **Inputs tuple (5.7,2.5,5.0,2.0):** for this last case, we have chosen a case from the class *iris virginica*. The results are shown in Table 5.3.

Class	**Keras**	**FLOPER**
iris_setosa	2.96895e-10	2.96895e-10
iris_versicolor	5.88269e-04	5.88269e-04
iris_virginica	0.10928	0.10928

Table 5.3: Output results for (5.7,2.5,5.0,2.0)

For all the three experiments, we obtain the same results in both representations, and furthermore all the three cases get the correct classification. In all cases we have rounded to five significant numbers. As a curiosity, in Figure 5.5 appears a part of the derivation tree given by FLOPER for the experiment with inputs (6.2,2.9,4.3,1.3), more concretely for *iris setosa*.

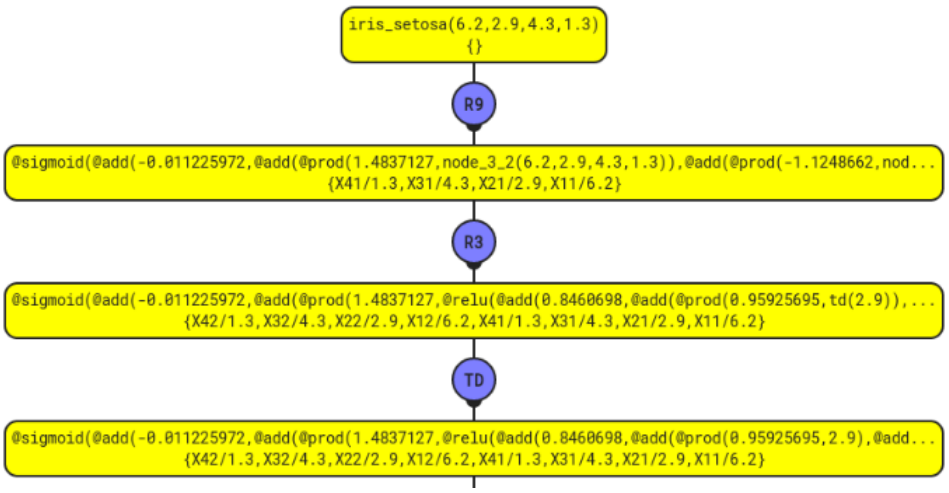

iris_setosa(6.2,2.9,4.3,1.3)
{}
R9
@sigmoid(@add(-0.011225972,@add(@prod(1.4837127,node_3_2(6.2,2.9,4.3,1.3)),@add(@prod(-1.1248662,nod...
{X41/1.3,X31/4.3,X21/2.9,X11/6.2}
R3
@sigmoid(@add(-0.011225972,@add(@prod(1.4837127,@relu(@add(0.8460698,@add(@prod(0.95925695,td(2.9)),...
{X42/1.3,X32/4.3,X22/2.9,X12/6.2,X41/1.3,X31/4.3,X21/2.9,X11/6.2}
TD
@sigmoid(@add(-0.011225972,@add(@prod(1.4837127,@relu(@add(0.8460698,@add(@prod(0.95925695,2.9),@add...
{X42/1.3,X32/4.3,X22/2.9,X12/6.2,X41/1.3,X31/4.3,X21/2.9,X11/6.2}

Figure 5.5: Part of the derivation tree for the class iris_setosa.

Now, we can start to carry on the tuning process. In order to set a complete comparative, we are going to use three lattices, in which the only point that will be different is the *member's* section. As FLOPER is based in fuzzy logic, is a little difficult to work with numbers in an easy way as for example other languages do, so this is the main reason for which we have to test three different lattices, especially for tuning weights.

Is essential to have a comparative that gives us a *baseline* for start working. In order to find it, we have used the original neural network translated (see Appendix B) allowing FLOPER to make the empty substitution with no symbolic aggregators nor constants. So, we get the **expected error of the original neural network**, that corresponds to **103.82526**.

```
< 103.82526, {} >
```

The error given at the end of the tuning process is defined as the error committed by the answers with the best substitution found based on the introduced test cases.
Our first candidate is the lattice **L25**, defined by the following lines of code:

```
% Elements
member(pos_inf).    member(neg_inf).    member(X) :- number(X).
members([0.75, 0.25, 0.50, -0.25, -0.75, -0.50]).
```

The second one is **L30**, with the following members:

```
% Elements
```

```
member(pos_inf).    member(neg_inf).    member(X) :- number(X).
members([0.90, 0.30, 0.60, -0.30, -0.90, -0.60]).
```

And the last one is **L95**, with the same number of elements that in the previous cases, but in this the maximum and the infimum are 0.95 and −0.95 respectively.

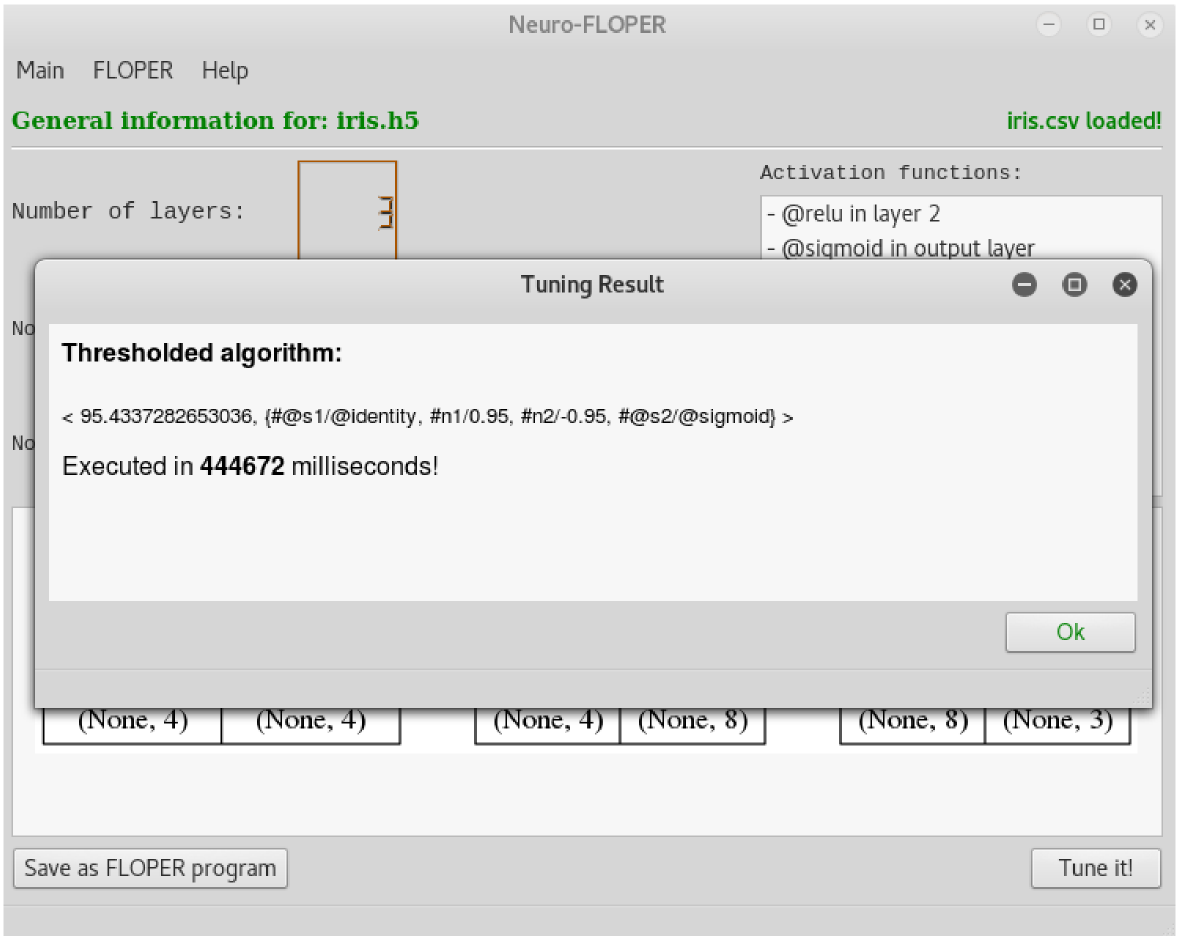
```
% Elements
member(pos_inf).    member(neg_inf).    member(X) :- number(X).
members([0.95, 0.25, 0.50, -0.25, -0.95, -0.50]).
```

As we are going to be focused in tuning the *bias* of our MLP, is not necessary to increment the range of the members, because do not have sense a *bias* with a value greater than one in this kind of architecture. A neuron with a *bias* equal or greater than one will always be fired.

For comparing the three lattices, they are going to be used with the symbolic neural networks described below:

- **sNN**$_2$: symbolic program with two symbolic constants, that are the suspicious *bias*.
- **sNN**$_3$: symbolic program with three symbolic constants, the two *bias* and one weight that has been chosen randomly.
- **sNN**$_4$: symbolic program with four symbolic constants, the two *bias* and two weight that have been chosen randomly.
- **sNN**$_{s1}$: only a symbolic aggregator placed in the hidden layer, in substitution of the *relu* function.
- **sNN**$_{s2}$: only a symbolic aggregator placed in the output layer, in substitution of the *sigmoid* function.
- **sNN**$_{s1s2}$: this symbolic program is the combination of **sNN**$_{s1}$ and **sNN**$_{s1}$.
- **sNN**$_{2s1s2}$: this symbolic program is the combination of **sNN**$_{s1}$ and **sNN**$_{s1}$ and **sNN**$_2$. It will be used for study how tuning works with a more complex symbolic neural network.

Illustration 5.6 shows how the result of tuning appears in Neuro-FLOPER, more concretely for the case of tuning sNN$_{2s1s2}$ with the lattice L95.



Thresholded algorithm:

< 95.4337282653036, {#@s1/@identity, #n1/0.95, #n2/-0.95, #@s2/@sigmoid} >

Executed in **444672** milliseconds!

Figure 5.6: Result for tuning the symbolic network sNN_{2s1s2} with the lattice L95.

We can compare the three lattices in three different categories: time, error and substitutions found. Here in Table 5.4 there is the comparison in terms of time (in miliseconds). Each execution corresponds to the average of repeat each case ten times.

Lattice	sNN_2	sNN_3	sNN_4	sNN_{s1}	sNN_{s2}	sNN_{s1s2}	sNN_{2s1s2}
L25	6436	25196	154790	4674	2868	14178	439802
L30	6432	25151	152090	4662	2863	14102	439617
L95	6580	25321	148868	4619	2768	13919	444672

Table 5.4: Time comparative for the different lattices used (in ms).

For the three cases of tuning weights, we can see that there is an exponential increasing of the time with the addition of new symbolic constants (Figure 5.7).

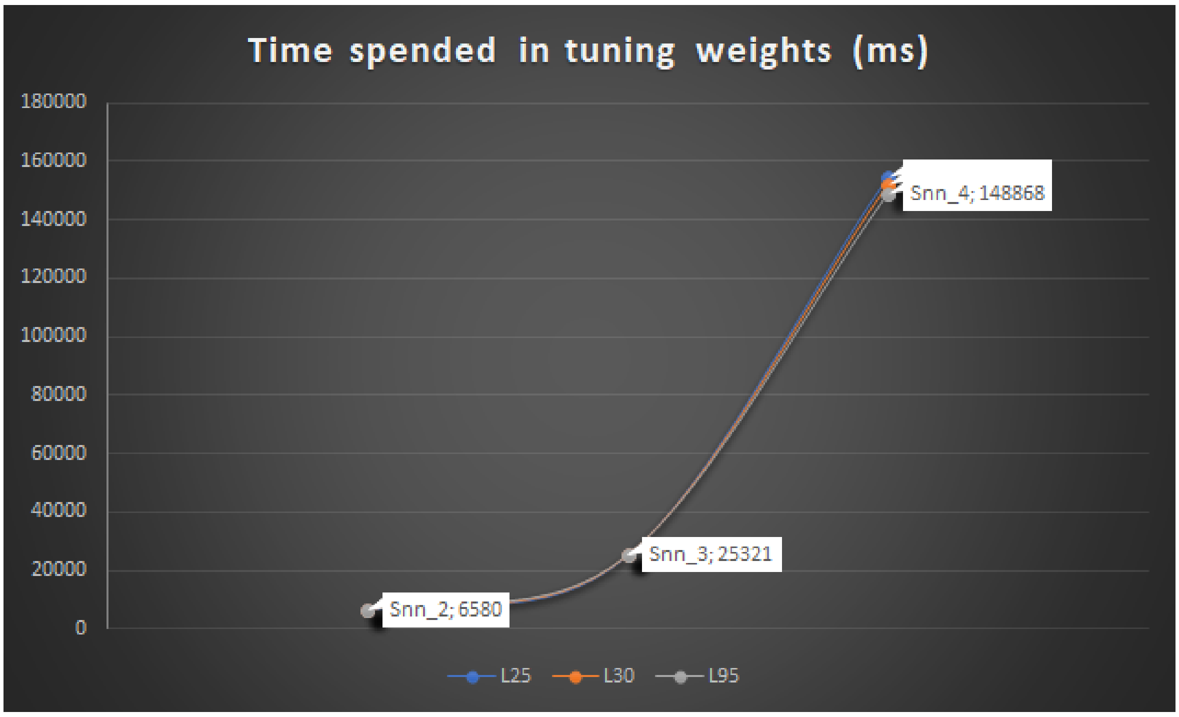


Figure 5.7: Representation of sNN_2, sNN_3 and sNN_4 in terms of time.

In Tables 5.5 and 5.6, appears the different substitutions found for each case. As we can see, the substitutions describe a pattern for weights due to the small number of them and the accuracy found.

Lattice	**sNN_2**	**sNN_3**	**sNN_4**
L25	#n1/0.25, #n2/0.75	#n1/0.25, #n2/0.75, #n3/0.75	#n1/0.75, #n2/0.75, #n3/0.75, #n4/-0.75
L30	#n1/0.3, #n2/0.9	#n1/0.3, #n2/0.9, #n3/0.9	#n1/0.9, #n2/0.9, #n3/0.9, #n4/-0.9
L95	#n1/0.25, #n2/0.95	#n1/0.25, #n2/0.95, #n3/0.95	#n1/0.95, #n2/0.95, #n3/0.95, #n4/-0.95

Table 5.5: Substitutions found for weights.

Lattice	**sNN**$_{s1}$	**sNN**$_{s2}$	**sNN**$_{s1s2}$	**sNN**$_{2s1s2}$
L25	#@s1/@identity	#@s2/@binary	#@s1/@identity, #@s2/@sigmoid	#@s1/@identity, #n1/0.75, #n2/-0.75, #@s2/@sigmoid
L30	#@s1/@identity	#@s2/@binary	#@s1/@identity, #@s2/@sigmoid	#@s1/@identity, #n1/0.9, #n2/-0.9, #@s2/@sigmoid
L95	#@s1/@identity	#@s2/@binary	#@s1/@identity, #@s2/@sigmoid	#@s1/@identity, #n1/0.95, #n2/-0.95, #@s2/@sigmoid

Table 5.6: Substitutions found for the rest of cases.

Remembering that our baseline error is **103.82526**, we can see in Table 5.7 that the results of tuning weights are practically equal or worse than the baseline. For only a small number of weights as symbolic constants, tuning is not very useful.

Lattice	**sNN**$_2$	**sNN**$_3$	**sNN**$_4$
L25	103.82526	104.87536	103.8210
L30	103.82526	104.33544	103.78791
L95	103.82526	104.11573	103.74545

Table 5.7: Error comparative in weights for the different lattices used.

But, for tuning aggregators or activation functions in our case, we can find more useful solutions with a smaller error. The combination of symbolic constants (that represents weights or bias with possible problems) and symbolic aggregators give us better results that applying both techniques separately. These results can be seen in Table 5.8.

Lattice	**sNN**$_{s1}$	**sNN**$_{s2}$	**sNN**$_{s1s2}$	**sNN**$_{2s1s2}$
L25	98.16823	99	98.16823	96.16193
L30	98.16823	99	98.16823	95.62387
L95	98.16823	99	98.16823	95.43372

Table 5.8: Error comparative for the rest of cases.

Looking back in Table 5.6, we found some strange substitutions for symbolic aggregators that decreases our errors. Lets analyze each case more in detail:

- **sNN**$_{s1}$: for the substitution of the activation function in the hidden layer, we get that is better to use the *identity* function with an increment of approximately a 5% over the original baseline. This has a simple explanation.

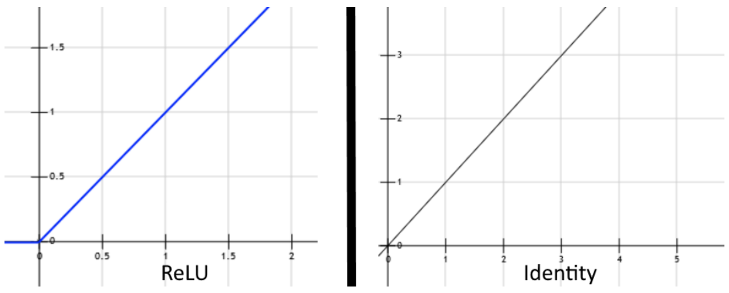


Figure 5.8: Comparison of ReLU and Identity function for positive values.

As we know, in the original layer is used the *ReLU function*. The different inputs are always positive, because each of them represent the length of a certain part of the flower, so the cannot be negative. In Figure 5.8 we can see that both functions are equivalent for working with positive values.

- **sNN**$_{s2}$: in this case, the substitusion of the activation function of the last layer gives us a *binary* function. This function, used in the first models of neural networks, is not used for real purposes nowadays, but in this case our tuning algoritmh chose it with an increment of approximately a 4% over the original baseline. This fact is due to our representation of test cases to making the tuning. Remember that each case was codified as:

```
1 -> iris_setosa(5.1, 3.5, 1.4, 0.2).
0 -> iris_versicolor(5.1, 3.5, 1.4, 0.2).
0 -> iris_virginica(5.1, 3.5, 1.4, 0.2).
```

So we are forcing each case to be one or zero. The *binary* function only can give this two values, so is logic that this function has less error for this case than the others. If we repeat the experiment using a trustability factor of, for example, 90% in the given data for creating the test file, the result will change to a sigmoid function as in sNN$_{2s1s2}$. Is important not to use trustability factors of 100% when importing data, adding a little uncertainty degree in order to avoid strange results and overfitting.

- **sNN**$_{s1s2}$ and **sNN**$_{2s1s2}$: for this last two cases we find that the *identity* appears again, but the *binary* is changed by the *sigmoid* function, the same that in the original one. When having more parameters that interfere in the process, out tuning algorithm must adapt the substitutions to a more realistic behaviour. We have an increasing of approximately a 5% for **sNN**$_{s1s2}$ as in sNN$_{s1}$, and an aproximate increasing of 8% for **sNN**$_{2s1s2}$ over the original baseline.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this Final Year Project a new point of view for neuro-fuzzy systems has been developed. Starting with the basic idea of combining Prolog with neural networks seen in [23], we have created a new tool that is more flexible and focused on research. This tool is the combination of two different parts, Keras and the FLOPER environment, that has been used to created Neuro-FLOPER, a tool for translating neural networks represented as graphs to fuzzy logic programs.

In this first version of Neuro-FLOPER, there is only support for an specific architecture of neural networks called multilayer perceptron. One of the main characteristics of Neuro-FLOPER is that we can create symbolic programs from the fuzzy logic translation of a neural network. This symbolic neural networks can be tuned by applying symbolic constants for weights and bias, and also symbolic aggregators for activation functions.

Applying both types separately has a small impact on the improvement of the whole network, but combining them gives significant results, specially when trying to solve some weights or bias that probably did not train well in Keras. It is important to interpret correctly the results given by the tuning algorithm over the new neural network, because combining neural networks and fuzzy logic can produce unusual results, due to some elements in the lattice or in the test cases. It is necessary to have a good framework of both of them in order to start working and create new models without overfitting and structural deficiencies.

6.2 Future work

The work developed in this Final Year Project would be a good starting point for future projects related to neuro-fuzzy systems and machine learning techniques. Some examples could be:

- Improve the efficiency for Neuro-FLOPER in the tuning process: the actual algorithm used is not efficient enough for tuning neural networks with big numbers of symbolic constants. It would be useful to improve the algorithm (using for instance, SMT solvers) to work with a greater number of symbolic weights and aggregators.
- Improve the number of functionalities for Neuro-FLOPER: starting at this point, there are a lot of capabilities that can be added to our tool. One of the most important one is the conversion of tuned models with Neuro-FLOPER to Keras again, in order to test how the tuned network behaves.
- Add new activation functions: the current version of Neuro-FLOPER only have some of the most basic activation functions traditionally used in the field, and nowadays there exists avilable functions with learnable parameters. Some examples of this kind of functions are ELU, SELU, RReLU, and ISRLU.
- Applying Neuro-FLOPER to a real-world problem: the combination of Neuro-FLOPER with other tools and techniques might be useful for giving solutions to real-world problems. An actual example is the Xylella disease, a new bacterium in Spain that affects a huge number of species of trees. There is no cure for this disease, and it could be interesting to improve XyApp (a prototype created by the author as an internship with CSIC) to help in the detection of symptoms only with images because the only existing method nowadays corresponds to an ADN analysis.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016. 12

[2] Robert Ackermann. An introduction to many-valued logics. 1968. 19

[3] James Frederick Baldwin, Trevor P Martin, and Bruce W Pilsworth. *Fril-fuzzy and evidential reasoning in artificial intelligence*. John Wiley & Sons, Inc., 1995. 26

[4] James Bergstra, Guillaume Desjardins, Pascal Lamblin, and Yoshua Bengio. Quadratic polynomials learn better image features. *Technical report, 1337*, 2009. 48

[5] Tomasa Calvo, Anna Kolesárová, Magda Komorníková, and Radko Mesiar. Aggregation operators: properties, classes and construction methods. In *Aggregation operators*, pages 3–104. Springer, 2002. 24

[6] Christer Carlsson, Robert Fullér, and Szvetlana Fullér. Possibility and necessity in weighted aggregation. In *The ordered weighted averaging operators*, pages 18–28. Springer, 1997. 23

[7] François Chollet et al. Keras, 2015. 2, 13

[8] János Fodor and Tomasa Calvo. Aggregation functions defined by t-norms and t-conorms. In *Aggregation and Fusion of Imperfect Information*, pages 36–48. Springer, 1998. 24

[9] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989. 8

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 8

[11] Takeshi Imanaka, Masato Soga, Kuniaki Uehara, and Junichi Toyoda. An integration of prolog and neural networks to deal with sensibility in logic programs. In *Systems Integration, 1990. Systems Integration'90., Proceedings of the First International Conference on*, pages 738–746. IEEE, 1990. 2, 43

[12] Pascual Julián Iranzo. *Lógica simbólica para informáticos*. Ra-Ma, 2004. 16

[13] Mitsuru Ishizuka and Naoki Kanai. Prolog-elf incorporating fuzzy logic. *New Generation Computing*, 3(4):479, 1985. 26

[14] Daniel D Johnson. Generating polyphonic music using tied parallel networks. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, pages 128–143. Springer, 2017. 11

[15] P Julián and M Alpuente. *Programación Lógica. Teoría y Práctica*. Pearson Prentice Hall, 2007. 15, 16

[16] Pascual Julián, Ginés Moreno, and Jaime Penabad. Operational/interpretive unfolding of multi-adjoint logic programs. *J. UCS*, 12(11):1679–1699, 2006. 28

[17] P Julián-Iranzo. *Especialización de Programas Lógico-Funcionales Perezosos*. PhD thesis, Tesis Doctoral, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2000. 14

[18] Anna Kolesárová and Magda Komorníková. Triangular norm-based iterative compensatory operators. *Fuzzy Sets and Systems*, 104(1):109–120, 1999. 24

[19] Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974. 15

[20] Jean-Louis Lassez, Michael J Maher, and Kim Marriott. Unification revisited. In *Foundations of deductive databases and logic programming*, pages 587–625. Elsevier, 1988. 15, 36

[21] Richard CT Lee. Fuzzy logic and the resolution principle. In *Readings in Fuzzy Sets for Intelligent systems*, pages 442–452. Elsevier, 1993. 21, 26

[22] John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012. 15, 26

[23] Bonifacio Martín and Alfredo Sanz. Redes neuronales y sistemas borrosos. *Zaragoza: Editorial Ra-Ma*, 1997. 5, 7, 9, 11, 73

[24] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. A multi-adjoint logic approach to abductive reasoning. In *International Conference on Logic Programming*, pages 269–283. Springer, 2001. 26

[25] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. Similarity-based unification: a multi-adjoint approach. *Fuzzy sets and systems*, 146(1):43–62, 2004. 26, 29, 30, 31, 33

[26] Masaharu Mizumoto. Pictorial representations of fuzzy connectives, part i: cases of t-norms, t-conorms and averaging operators. *Fuzzy sets and systems*, 31(2):217–242, 1989. 24

[27] Ginés Moreno, Jaime Penabad, José A Riaza, and Germán Vidal. Symbolic execution and thresholding for efficiently tuning fuzzy logic programs. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 131–147. Springer, 2016. 29, 30, 32, 33, 40, 42

[28] Ginés Moreno, Jaime Penabad, and José Antonio Riaza. Symbolic unfolding of multi-adjoint logic programs. In *9th European Symposium on Computational Intelligence and Mathematics.*, 2017. 1

[29] Ginés Moreno and José A Riaza. An online tool for tuning fuzzy logic programs. In *International Joint Conference on Rules and Reasoning*, pages 184–198. Springer, 2017. 1, 29, 30, 32, 35

[30] Berndt Müller, Joachim Reinhardt, and Michael T Strickland. *Neural networks: an introduction*. Springer Science & Business Media, 2012. 6

[31] Hung T Nguyen and Elbert A Walker. *A first course in fuzzy logic*. CRC press, 2005. 21, 22, 30

[32] Amanda Ramcharan, Kelsee Baranowski, Peter McClowsky, Babuali Ahmed, James Legg, and David Hughes. Using transfer learning for image-based cassava disease detection. *arXiv preprint arXiv:1707.03717*, 2017. 11

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986. 7, 8, 9

[34] David E Rumelhart and James L McClelland. Parallel distribution processing: exploration in the microstructure of cognition. *MA: MIT Press, Cambridge*, 1986. 6

[35] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing*, volume 1. MIT press Cambridge, MA, 1987. 7, 9, 43

[36] Berthold Schweizer and Abe Sklar. *Probabilistic metric spaces*. Courier Corporation, 2011. 21

[37] Claudio Vaucheret, Sergio Guadarrama, and Susana Mumnoz. Fuzzy prolog: A simple general implementation using clp (f) claudio vaucheret1, sergio guadarrama1, and susana mumnoz2 1 departamento de inteligencia artificial claudio@ clip. dia. fi. upm. es. 26

[38] Carlos Vázquez Pérez-Íñigo. Foundations and applications of fuzzy logic programming with weights and similarity relations. 2015. 25

[39] L. A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese*, 30:407–428, 1965. 18, 19, 20

[40] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965. 18

Appendix A

Training a multilayer perceptron with the Iris dataset

```python
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from keras.models import load_model

# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(8, input_dim=4, activation='relu'))
  model.add(Dense(3, activation='sigmoid'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
  return model

if __name__ == "__main__":

    # fix random seed for reproducibility
    seed = 7
    numpy.random.seed(seed)

    # load dataset
    dataframe = pandas.read_csv("iris.csv", header=None)
    dataset = dataframe.values
    X = dataset[:,0:4].astype(float)
    Y = dataset[:,4]

    # encode class values as integers
    encoder = LabelEncoder()
    encoder.fit(Y)

    #Store the labels for using it in Neuro-FLOPER
    f = open('labels.txt','w')
    for label in list(encoder.classes_):
            f.write(label)
            f.write("\n")
    f.close()

    encoded_Y = encoder.transform(Y)

    # convert integers to dummy variables (i.e. one hot encoded)
    dummy_y = np_utils.to_categorical(encoded_Y)

    estimator = KerasClassifier(build_fn=baseline_model, epochs=200,
   batch_size=5, verbose=0)

 #Create a KFold for cross validation
    kfold = KFold(n_splits=10, shuffle=True, random_state=seed)

    results = cross_val_score(estimator, X, dummy_y, cv=kfold)
    print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std
   ()*100))

    estimator.fit(X, dummy_y)

 #Store the complete neural network
    estimator.model.save('iris.h5')
    print("Model saved successfully!")
```

Appendix B

Translated Keras model into a fuzzy logic program

```
node_1_2(X1, X2, X3, X4)<- @relu( @add(0.6244668, @add ( @prod
    (0.70569754,td(X2)), @add(@prod(-0.3849561,td(X4)), @add(@prod
    (0.731073,td(X1)),@prod(-0.7602397,td(X3))))))).
node_2_2(X1, X2, X3, X4)<- @relu( @add(-0.5086394, @add ( @prod
    (0.32892075,td(X2)), @add(@prod(0.9662539,td(X4)), @add(@prod
    (-0.52540934,td(X1)),@prod(0.63079953,td(X3))))))).
node_3_2(X1, X2, X3, X4)<- @relu( @add(0.8460698, @add ( @prod
    (0.95925695,td(X2)), @add(@prod(-0.8759079,td(X4)), @add(@prod
    (0.1464666,td(X1)),@prod(-0.35556462,td(X3))))))).
node_4_2(X1, X2, X3, X4)<- @relu( @add(0.0, @add ( @prod(0.4458459,td(X2)
    ), @add(@prod(-0.4348098,td(X4)), @add(@prod(-0.39125273,td(X1)),@prod
    (-0.31539902,td(X3))))))).
node_5_2(X1, X2, X3, X4)<- @relu( @add(-0.47974947, @add ( @prod
    (-0.73593247,td(X2)), @add(@prod(1.3490778,td(X4)), @add(@prod
    (0.33557042,td(X1)),@prod(0.2220378,td(X3))))))).
node_6_2(X1, X2, X3, X4)<- @relu( @add(0.41330096, @add ( @prod
    (0.76326245,td(X2)), @add(@prod(1.7278047,td(X4)), @add(@prod
    (0.30372941,td(X1)),@prod(1.1798365,td(X3))))))).
node_7_2(X1, X2, X3, X4)<- @relu( @add(0.0, @add ( @prod(-0.6056487,td(X2
    )), @add(@prod(0.04486853,td(X4)), @add(@prod(-0.26603696,td(X1)),
    @prod(-0.1796673,td(X3))))))).
node_8_2(X1, X2, X3, X4)<- @relu( @add(0.39581802, @add ( @prod
    (1.0809461,td(X2)), @add(@prod(-0.36609808,td(X4)), @add(@prod
    (0.033086,td(X1)),@prod(-0.8241007,td(X3))))))).

iris_setosa(X1, X2, X3, X4)<- @sigmoid( @add(-0.011225972, @add ( @prod
    (1.4837127,node_3_2(X1, X2, X3, X4)), @add(@prod(-1.1248662,node_2_2(
    X1, X2, X3, X4)), @add(@prod(0.52394736,node_1_2(X1, X2, X3, X4)),
    @add(@prod(-0.5511421,node_4_2(X1, X2, X3, X4)), @add(@prod
    (0.17743558,node_7_2(X1, X2, X3, X4)), @add(@prod(-1.5553585,node_5_2(
    X1, X2, X3, X4)), @add(@prod(0.31823215,node_8_2(X1, X2, X3, X4)),
    @prod(-1.1773438,node_6_2(X1, X2, X3, X4))))))))))).
iris_versicolor(X1, X2, X3, X4)<- @sigmoid( @add(0.009011258, @add (
    @prod(0.48723626,node_3_2(X1, X2, X3, X4)), @add(@prod(-0.9324108,
    node_2_2(X1, X2, X3, X4)), @add(@prod(0.33959225,node_1_2(X1, X2, X3,
    X4)), @add(@prod(0.68769246,node_4_2(X1, X2, X3, X4)), @add(@prod
    (0.2782907,node_7_2(X1, X2, X3, X4)), @add(@prod(0.05014925,node_5_2(
    X1, X2, X3, X4)), @add(@prod(-2.844793,node_8_2(X1, X2, X3, X4)),@prod
    (-0.47109953,node_6_2(X1, X2, X3, X4))))))))))).
iris_virginica(X1, X2, X3, X4)<- @sigmoid( @add(-0.7190765, @add ( @prod
    (-1.8247896,node_3_2(X1, X2, X3, X4)), @add(@prod(1.0094312,node_2_2(
    X1, X2, X3, X4)), @add(@prod(-1.7643892,node_1_2(X1, X2, X3, X4)),
    @add(@prod(0.47558457,node_4_2(X1, X2, X3, X4)), @add(@prod(-0.622447,
    node_7_2(X1, X2, X3, X4)), @add(@prod(0.80634564,node_5_2(X1, X2, X3,
    X4)), @add(@prod(0.20066284,node_8_2(X1, X2, X3, X4)),@prod
    (-0.15255216,node_6_2(X1, X2, X3, X4))))))))))).
```

Appendix C

Iris dataset as a set of test cases

```
1 -> iris_setosa(5.1, 3.5, 1.4, 0.2).
0 -> iris_versicolor(5.1, 3.5, 1.4, 0.2).
0 -> iris_virginica(5.1, 3.5, 1.4, 0.2).
1 -> iris_setosa(4.9, 3.0, 1.4, 0.2).
0 -> iris_versicolor(4.9, 3.0, 1.4, 0.2).
0 -> iris_virginica(4.9, 3.0, 1.4, 0.2).
1 -> iris_setosa(4.7, 3.2, 1.3, 0.2).
0 -> iris_versicolor(4.7, 3.2, 1.3, 0.2).
0 -> iris_virginica(4.7, 3.2, 1.3, 0.2).
1 -> iris_setosa(4.6, 3.1, 1.5, 0.2).
0 -> iris_versicolor(4.6, 3.1, 1.5, 0.2).
0 -> iris_virginica(4.6, 3.1, 1.5, 0.2).
1 -> iris_setosa(5.0, 3.6, 1.4, 0.2).
0 -> iris_versicolor(5.0, 3.6, 1.4, 0.2).
0 -> iris_virginica(5.0, 3.6, 1.4, 0.2).
1 -> iris_setosa(5.4, 3.9, 1.7, 0.4).
0 -> iris_versicolor(5.4, 3.9, 1.7, 0.4).
0 -> iris_virginica(5.4, 3.9, 1.7, 0.4).
1 -> iris_setosa(4.6, 3.4, 1.4, 0.3).
0 -> iris_versicolor(4.6, 3.4, 1.4, 0.3).
0 -> iris_virginica(4.6, 3.4, 1.4, 0.3).
1 -> iris_setosa(5.0, 3.4, 1.5, 0.2).
0 -> iris_versicolor(5.0, 3.4, 1.5, 0.2).
0 -> iris_virginica(5.0, 3.4, 1.5, 0.2).
1 -> iris_setosa(4.4, 2.9, 1.4, 0.2).
0 -> iris_versicolor(4.4, 2.9, 1.4, 0.2).
0 -> iris_virginica(4.4, 2.9, 1.4, 0.2).
1 -> iris_setosa(4.9, 3.1, 1.5, 0.1).
0 -> iris_versicolor(4.9, 3.1, 1.5, 0.1).
0 -> iris_virginica(4.9, 3.1, 1.5, 0.1).
1 -> iris_setosa(5.4, 3.7, 1.5, 0.2).
0 -> iris_versicolor(5.4, 3.7, 1.5, 0.2).
0 -> iris_virginica(5.4, 3.7, 1.5, 0.2).
1 -> iris_setosa(4.8, 3.4, 1.6, 0.2).
0 -> iris_versicolor(4.8, 3.4, 1.6, 0.2).
0 -> iris_virginica(4.8, 3.4, 1.6, 0.2).
1 -> iris_setosa(4.8, 3.0, 1.4, 0.1).
0 -> iris_versicolor(4.8, 3.0, 1.4, 0.1).
0 -> iris_virginica(4.8, 3.0, 1.4, 0.1).
1 -> iris_setosa(4.3, 3.0, 1.1, 0.1).
0 -> iris_versicolor(4.3, 3.0, 1.1, 0.1).
0 -> iris_virginica(4.3, 3.0, 1.1, 0.1).
1 -> iris_setosa(5.8, 4.0, 1.2, 0.2).
0 -> iris_versicolor(5.8, 4.0, 1.2, 0.2).
0 -> iris_virginica(5.8, 4.0, 1.2, 0.2).
1 -> iris_setosa(5.7, 4.4, 1.5, 0.4).
0 -> iris_versicolor(5.7, 4.4, 1.5, 0.4).
0 -> iris_virginica(5.7, 4.4, 1.5, 0.4).
1 -> iris_setosa(5.4, 3.9, 1.3, 0.4).
0 -> iris_versicolor(5.4, 3.9, 1.3, 0.4).
0 -> iris_virginica(5.4, 3.9, 1.3, 0.4).
1 -> iris_setosa(5.1, 3.5, 1.4, 0.3).
0 -> iris_versicolor(5.1, 3.5, 1.4, 0.3).
0 -> iris_virginica(5.1, 3.5, 1.4, 0.3).
```

```
1 -> iris_setosa(5.7, 3.8, 1.7, 0.3).
0 -> iris_versicolor(5.7, 3.8, 1.7, 0.3).
0 -> iris_virginica(5.7, 3.8, 1.7, 0.3).
1 -> iris_setosa(5.1, 3.8, 1.5, 0.3).
0 -> iris_versicolor(5.1, 3.8, 1.5, 0.3).
0 -> iris_virginica(5.1, 3.8, 1.5, 0.3).
1 -> iris_setosa(5.4, 3.4, 1.7, 0.2).
0 -> iris_versicolor(5.4, 3.4, 1.7, 0.2).
0 -> iris_virginica(5.4, 3.4, 1.7, 0.2).
1 -> iris_setosa(5.1, 3.7, 1.5, 0.4).
0 -> iris_versicolor(5.1, 3.7, 1.5, 0.4).
0 -> iris_virginica(5.1, 3.7, 1.5, 0.4).
1 -> iris_setosa(4.6, 3.6, 1.0, 0.2).
0 -> iris_versicolor(4.6, 3.6, 1.0, 0.2).
0 -> iris_virginica(4.6, 3.6, 1.0, 0.2).
1 -> iris_setosa(5.1, 3.3, 1.7, 0.5).
0 -> iris_versicolor(5.1, 3.3, 1.7, 0.5).
0 -> iris_virginica(5.1, 3.3, 1.7, 0.5).
1 -> iris_setosa(4.8, 3.4, 1.9, 0.2).
0 -> iris_versicolor(4.8, 3.4, 1.9, 0.2).
0 -> iris_virginica(4.8, 3.4, 1.9, 0.2).
1 -> iris_setosa(5.0, 3.0, 1.6, 0.2).
0 -> iris_versicolor(5.0, 3.0, 1.6, 0.2).
0 -> iris_virginica(5.0, 3.0, 1.6, 0.2).
1 -> iris_setosa(5.0, 3.4, 1.6, 0.4).
0 -> iris_versicolor(5.0, 3.4, 1.6, 0.4).
0 -> iris_virginica(5.0, 3.4, 1.6, 0.4).
1 -> iris_setosa(5.2, 3.5, 1.5, 0.2).
0 -> iris_versicolor(5.2, 3.5, 1.5, 0.2).
0 -> iris_virginica(5.2, 3.5, 1.5, 0.2).
1 -> iris_setosa(5.2, 3.4, 1.4, 0.2).
0 -> iris_versicolor(5.2, 3.4, 1.4, 0.2).
0 -> iris_virginica(5.2, 3.4, 1.4, 0.2).
1 -> iris_setosa(4.7, 3.2, 1.6, 0.2).
0 -> iris_versicolor(4.7, 3.2, 1.6, 0.2).
0 -> iris_virginica(4.7, 3.2, 1.6, 0.2).
1 -> iris_setosa(4.8, 3.1, 1.6, 0.2).
0 -> iris_versicolor(4.8, 3.1, 1.6, 0.2).
0 -> iris_virginica(4.8, 3.1, 1.6, 0.2).
1 -> iris_setosa(5.4, 3.4, 1.5, 0.4).
0 -> iris_versicolor(5.4, 3.4, 1.5, 0.4).
0 -> iris_virginica(5.4, 3.4, 1.5, 0.4).
1 -> iris_setosa(5.2, 4.1, 1.5, 0.1).
0 -> iris_versicolor(5.2, 4.1, 1.5, 0.1).
0 -> iris_virginica(5.2, 4.1, 1.5, 0.1).
1 -> iris_setosa(5.5, 4.2, 1.4, 0.2).
0 -> iris_versicolor(5.5, 4.2, 1.4, 0.2).
0 -> iris_virginica(5.5, 4.2, 1.4, 0.2).
1 -> iris_setosa(4.9, 3.1, 1.5, 0.1).
0 -> iris_versicolor(4.9, 3.1, 1.5, 0.1).
0 -> iris_virginica(4.9, 3.1, 1.5, 0.1).
1 -> iris_setosa(5.0, 3.2, 1.2, 0.2).
0 -> iris_versicolor(5.0, 3.2, 1.2, 0.2).
0 -> iris_virginica(5.0, 3.2, 1.2, 0.2).
1 -> iris_setosa(5.5, 3.5, 1.3, 0.2).
0 -> iris_versicolor(5.5, 3.5, 1.3, 0.2).
0 -> iris_virginica(5.5, 3.5, 1.3, 0.2).
1 -> iris_setosa(4.9, 3.1, 1.5, 0.1).
0 -> iris_versicolor(4.9, 3.1, 1.5, 0.1).
0 -> iris_virginica(4.9, 3.1, 1.5, 0.1).
1 -> iris_setosa(4.4, 3.0, 1.3, 0.2).
0 -> iris_versicolor(4.4, 3.0, 1.3, 0.2).
0 -> iris_virginica(4.4, 3.0, 1.3, 0.2).
1 -> iris_setosa(5.1, 3.4, 1.5, 0.2).
0 -> iris_versicolor(5.1, 3.4, 1.5, 0.2).
0 -> iris_virginica(5.1, 3.4, 1.5, 0.2).
1 -> iris_setosa(5.0, 3.5, 1.3, 0.3).
0 -> iris_versicolor(5.0, 3.5, 1.3, 0.3).
0 -> iris_virginica(5.0, 3.5, 1.3, 0.3).
1 -> iris_setosa(4.5, 2.3, 1.3, 0.3).
0 -> iris_versicolor(4.5, 2.3, 1.3, 0.3).
0 -> iris_virginica(4.5, 2.3, 1.3, 0.3).
1 -> iris_setosa(4.4, 3.2, 1.3, 0.2).
0 -> iris_versicolor(4.4, 3.2, 1.3, 0.2).
0 -> iris_virginica(4.4, 3.2, 1.3, 0.2).
1 -> iris_setosa(5.0, 3.5, 1.6, 0.6).
0 -> iris_versicolor(5.0, 3.5, 1.6, 0.6).
0 -> iris_virginica(5.0, 3.5, 1.6, 0.6).
```

```
1 -> iris_setosa(5.1, 3.8, 1.9, 0.4).
0 -> iris_versicolor(5.1, 3.8, 1.9, 0.4).
0 -> iris_virginica(5.1, 3.8, 1.9, 0.4).
1 -> iris_setosa(4.8, 3.0, 1.4, 0.3).
0 -> iris_versicolor(4.8, 3.0, 1.4, 0.3).
0 -> iris_virginica(4.8, 3.0, 1.4, 0.3).
1 -> iris_setosa(5.1, 3.8, 1.6, 0.2).
0 -> iris_versicolor(5.1, 3.8, 1.6, 0.2).
0 -> iris_virginica(5.1, 3.8, 1.6, 0.2).
1 -> iris_setosa(4.6, 3.2, 1.4, 0.2).
0 -> iris_versicolor(4.6, 3.2, 1.4, 0.2).
0 -> iris_virginica(4.6, 3.2, 1.4, 0.2).
1 -> iris_setosa(5.3, 3.7, 1.5, 0.2).
0 -> iris_versicolor(5.3, 3.7, 1.5, 0.2).
0 -> iris_virginica(5.3, 3.7, 1.5, 0.2).
1 -> iris_setosa(5.0, 3.3, 1.4, 0.2).
0 -> iris_versicolor(5.0, 3.3, 1.4, 0.2).
0 -> iris_virginica(5.0, 3.3, 1.4, 0.2).
0 -> iris_setosa(7.0, 3.2, 4.7, 1.4).
1 -> iris_versicolor(7.0, 3.2, 4.7, 1.4).
0 -> iris_virginica(7.0, 3.2, 4.7, 1.4).
0 -> iris_setosa(6.4, 3.2, 4.5, 1.5).
1 -> iris_versicolor(6.4, 3.2, 4.5, 1.5).
0 -> iris_virginica(6.4, 3.2, 4.5, 1.5).
0 -> iris_setosa(6.9, 3.1, 4.9, 1.5).
1 -> iris_versicolor(6.9, 3.1, 4.9, 1.5).
0 -> iris_virginica(6.9, 3.1, 4.9, 1.5).
0 -> iris_setosa(5.5, 2.3, 4.0, 1.3).
1 -> iris_versicolor(5.5, 2.3, 4.0, 1.3).
0 -> iris_virginica(5.5, 2.3, 4.0, 1.3).
0 -> iris_setosa(6.5, 2.8, 4.6, 1.5).
1 -> iris_versicolor(6.5, 2.8, 4.6, 1.5).
0 -> iris_virginica(6.5, 2.8, 4.6, 1.5).
0 -> iris_setosa(5.7, 2.8, 4.5, 1.3).
1 -> iris_versicolor(5.7, 2.8, 4.5, 1.3).
0 -> iris_virginica(5.7, 2.8, 4.5, 1.3).
0 -> iris_setosa(6.3, 3.3, 4.7, 1.6).
1 -> iris_versicolor(6.3, 3.3, 4.7, 1.6).
0 -> iris_virginica(6.3, 3.3, 4.7, 1.6).
0 -> iris_setosa(4.9, 2.4, 3.3, 1.0).
1 -> iris_versicolor(4.9, 2.4, 3.3, 1.0).
0 -> iris_virginica(4.9, 2.4, 3.3, 1.0).
0 -> iris_setosa(6.6, 2.9, 4.6, 1.3).
1 -> iris_versicolor(6.6, 2.9, 4.6, 1.3).
0 -> iris_virginica(6.6, 2.9, 4.6, 1.3).
0 -> iris_setosa(5.2, 2.7, 3.9, 1.4).
1 -> iris_versicolor(5.2, 2.7, 3.9, 1.4).
0 -> iris_virginica(5.2, 2.7, 3.9, 1.4).
0 -> iris_setosa(5.0, 2.0, 3.5, 1.0).
1 -> iris_versicolor(5.0, 2.0, 3.5, 1.0).
0 -> iris_virginica(5.0, 2.0, 3.5, 1.0).
0 -> iris_setosa(5.9, 3.0, 4.2, 1.5).
1 -> iris_versicolor(5.9, 3.0, 4.2, 1.5).
0 -> iris_virginica(5.9, 3.0, 4.2, 1.5).
0 -> iris_setosa(6.0, 2.2, 4.0, 1.0).
1 -> iris_versicolor(6.0, 2.2, 4.0, 1.0).
0 -> iris_virginica(6.0, 2.2, 4.0, 1.0).
0 -> iris_setosa(6.1, 2.9, 4.7, 1.4).
1 -> iris_versicolor(6.1, 2.9, 4.7, 1.4).
0 -> iris_virginica(6.1, 2.9, 4.7, 1.4).
0 -> iris_setosa(5.6, 2.9, 3.6, 1.3).
1 -> iris_versicolor(5.6, 2.9, 3.6, 1.3).
0 -> iris_virginica(5.6, 2.9, 3.6, 1.3).
0 -> iris_setosa(6.7, 3.1, 4.4, 1.4).
1 -> iris_versicolor(6.7, 3.1, 4.4, 1.4).
0 -> iris_virginica(6.7, 3.1, 4.4, 1.4).
0 -> iris_setosa(5.6, 3.0, 4.5, 1.5).
1 -> iris_versicolor(5.6, 3.0, 4.5, 1.5).
0 -> iris_virginica(5.6, 3.0, 4.5, 1.5).
0 -> iris_setosa(5.8, 2.7, 4.1, 1.0).
1 -> iris_versicolor(5.8, 2.7, 4.1, 1.0).
0 -> iris_virginica(5.8, 2.7, 4.1, 1.0).
0 -> iris_setosa(6.2, 2.2, 4.5, 1.5).
1 -> iris_versicolor(6.2, 2.2, 4.5, 1.5).
0 -> iris_virginica(6.2, 2.2, 4.5, 1.5).
0 -> iris_setosa(5.6, 2.5, 3.9, 1.1).
1 -> iris_versicolor(5.6, 2.5, 3.9, 1.1).
0 -> iris_virginica(5.6, 2.5, 3.9, 1.1).
```

```
0 -> iris_setosa(5.9, 3.2, 4.8, 1.8).
1 -> iris_versicolor(5.9, 3.2, 4.8, 1.8).
0 -> iris_virginica(5.9, 3.2, 4.8, 1.8).
0 -> iris_setosa(6.1, 2.8, 4.0, 1.3).
1 -> iris_versicolor(6.1, 2.8, 4.0, 1.3).
0 -> iris_virginica(6.1, 2.8, 4.0, 1.3).
0 -> iris_setosa(6.3, 2.5, 4.9, 1.5).
1 -> iris_versicolor(6.3, 2.5, 4.9, 1.5).
0 -> iris_virginica(6.3, 2.5, 4.9, 1.5).
0 -> iris_setosa(6.1, 2.8, 4.7, 1.2).
1 -> iris_versicolor(6.1, 2.8, 4.7, 1.2).
0 -> iris_virginica(6.1, 2.8, 4.7, 1.2).
0 -> iris_setosa(6.4, 2.9, 4.3, 1.3).
1 -> iris_versicolor(6.4, 2.9, 4.3, 1.3).
0 -> iris_virginica(6.4, 2.9, 4.3, 1.3).
0 -> iris_setosa(6.6, 3.0, 4.4, 1.4).
1 -> iris_versicolor(6.6, 3.0, 4.4, 1.4).
0 -> iris_virginica(6.6, 3.0, 4.4, 1.4).
0 -> iris_setosa(6.8, 2.8, 4.8, 1.4).
1 -> iris_versicolor(6.8, 2.8, 4.8, 1.4).
0 -> iris_virginica(6.8, 2.8, 4.8, 1.4).
0 -> iris_setosa(6.7, 3.0, 5.0, 1.7).
1 -> iris_versicolor(6.7, 3.0, 5.0, 1.7).
0 -> iris_virginica(6.7, 3.0, 5.0, 1.7).
0 -> iris_setosa(6.0, 2.9, 4.5, 1.5).
1 -> iris_versicolor(6.0, 2.9, 4.5, 1.5).
0 -> iris_virginica(6.0, 2.9, 4.5, 1.5).
0 -> iris_setosa(5.7, 2.6, 3.5, 1.0).
1 -> iris_versicolor(5.7, 2.6, 3.5, 1.0).
0 -> iris_virginica(5.7, 2.6, 3.5, 1.0).
0 -> iris_setosa(5.5, 2.4, 3.8, 1.1).
1 -> iris_versicolor(5.5, 2.4, 3.8, 1.1).
0 -> iris_virginica(5.5, 2.4, 3.8, 1.1).
0 -> iris_setosa(5.5, 2.4, 3.7, 1.0).
1 -> iris_versicolor(5.5, 2.4, 3.7, 1.0).
0 -> iris_virginica(5.5, 2.4, 3.7, 1.0).
0 -> iris_setosa(5.8, 2.7, 3.9, 1.2).
1 -> iris_versicolor(5.8, 2.7, 3.9, 1.2).
0 -> iris_virginica(5.8, 2.7, 3.9, 1.2).
0 -> iris_setosa(6.0, 2.7, 5.1, 1.6).
1 -> iris_versicolor(6.0, 2.7, 5.1, 1.6).
0 -> iris_virginica(6.0, 2.7, 5.1, 1.6).
0 -> iris_setosa(5.4, 3.0, 4.5, 1.5).
1 -> iris_versicolor(5.4, 3.0, 4.5, 1.5).
0 -> iris_virginica(5.4, 3.0, 4.5, 1.5).
0 -> iris_setosa(6.0, 3.4, 4.5, 1.6).
1 -> iris_versicolor(6.0, 3.4, 4.5, 1.6).
0 -> iris_virginica(6.0, 3.4, 4.5, 1.6).
0 -> iris_setosa(6.7, 3.1, 4.7, 1.5).
1 -> iris_versicolor(6.7, 3.1, 4.7, 1.5).
0 -> iris_virginica(6.7, 3.1, 4.7, 1.5).
0 -> iris_setosa(6.3, 2.3, 4.4, 1.3).
1 -> iris_versicolor(6.3, 2.3, 4.4, 1.3).
0 -> iris_virginica(6.3, 2.3, 4.4, 1.3).
0 -> iris_setosa(5.6, 3.0, 4.1, 1.3).
1 -> iris_versicolor(5.6, 3.0, 4.1, 1.3).
0 -> iris_virginica(5.6, 3.0, 4.1, 1.3).
0 -> iris_setosa(5.5, 2.5, 4.0, 1.3).
1 -> iris_versicolor(5.5, 2.5, 4.0, 1.3).
0 -> iris_virginica(5.5, 2.5, 4.0, 1.3).
0 -> iris_setosa(5.5, 2.6, 4.4, 1.2).
1 -> iris_versicolor(5.5, 2.6, 4.4, 1.2).
0 -> iris_virginica(5.5, 2.6, 4.4, 1.2).
0 -> iris_setosa(6.1, 3.0, 4.6, 1.4).
1 -> iris_versicolor(6.1, 3.0, 4.6, 1.4).
0 -> iris_virginica(6.1, 3.0, 4.6, 1.4).
0 -> iris_setosa(5.8, 2.6, 4.0, 1.2).
1 -> iris_versicolor(5.8, 2.6, 4.0, 1.2).
0 -> iris_virginica(5.8, 2.6, 4.0, 1.2).
0 -> iris_setosa(5.0, 2.3, 3.3, 1.0).
1 -> iris_versicolor(5.0, 2.3, 3.3, 1.0).
0 -> iris_virginica(5.0, 2.3, 3.3, 1.0).
0 -> iris_setosa(5.6, 2.7, 4.2, 1.3).
1 -> iris_versicolor(5.6, 2.7, 4.2, 1.3).
0 -> iris_virginica(5.6, 2.7, 4.2, 1.3).
0 -> iris_setosa(5.7, 3.0, 4.2, 1.2).
1 -> iris_versicolor(5.7, 3.0, 4.2, 1.2).
0 -> iris_virginica(5.7, 3.0, 4.2, 1.2).
```

```
0 -> iris_setosa(5.7, 2.9, 4.2, 1.3).
1 -> iris_versicolor(5.7, 2.9, 4.2, 1.3).
0 -> iris_virginica(5.7, 2.9, 4.2, 1.3).
0 -> iris_setosa(6.2, 2.9, 4.3, 1.3).
1 -> iris_versicolor(6.2, 2.9, 4.3, 1.3).
0 -> iris_virginica(6.2, 2.9, 4.3, 1.3).
0 -> iris_setosa(5.1, 2.5, 3.0, 1.1).
1 -> iris_versicolor(5.1, 2.5, 3.0, 1.1).
0 -> iris_virginica(5.1, 2.5, 3.0, 1.1).
0 -> iris_setosa(5.7, 2.8, 4.1, 1.3).
1 -> iris_versicolor(5.7, 2.8, 4.1, 1.3).
0 -> iris_virginica(5.7, 2.8, 4.1, 1.3).
0 -> iris_setosa(6.3, 3.3, 6.0, 2.5).
0 -> iris_versicolor(6.3, 3.3, 6.0, 2.5).
1 -> iris_virginica(6.3, 3.3, 6.0, 2.5).
0 -> iris_setosa(5.8, 2.7, 5.1, 1.9).
0 -> iris_versicolor(5.8, 2.7, 5.1, 1.9).
1 -> iris_virginica(5.8, 2.7, 5.1, 1.9).
0 -> iris_setosa(7.1, 3.0, 5.9, 2.1).
0 -> iris_versicolor(7.1, 3.0, 5.9, 2.1).
1 -> iris_virginica(7.1, 3.0, 5.9, 2.1).
0 -> iris_setosa(6.3, 2.9, 5.6, 1.8).
0 -> iris_versicolor(6.3, 2.9, 5.6, 1.8).
1 -> iris_virginica(6.3, 2.9, 5.6, 1.8).
0 -> iris_setosa(6.5, 3.0, 5.8, 2.2).
0 -> iris_versicolor(6.5, 3.0, 5.8, 2.2).
1 -> iris_virginica(6.5, 3.0, 5.8, 2.2).
0 -> iris_setosa(7.6, 3.0, 6.6, 2.1).
0 -> iris_versicolor(7.6, 3.0, 6.6, 2.1).
1 -> iris_virginica(7.6, 3.0, 6.6, 2.1).
0 -> iris_setosa(4.9, 2.5, 4.5, 1.7).
0 -> iris_versicolor(4.9, 2.5, 4.5, 1.7).
1 -> iris_virginica(4.9, 2.5, 4.5, 1.7).
0 -> iris_setosa(7.3, 2.9, 6.3, 1.8).
0 -> iris_versicolor(7.3, 2.9, 6.3, 1.8).
1 -> iris_virginica(7.3, 2.9, 6.3, 1.8).
0 -> iris_setosa(6.7, 2.5, 5.8, 1.8).
0 -> iris_versicolor(6.7, 2.5, 5.8, 1.8).
1 -> iris_virginica(6.7, 2.5, 5.8, 1.8).
0 -> iris_setosa(7.2, 3.6, 6.1, 2.5).
0 -> iris_versicolor(7.2, 3.6, 6.1, 2.5).
1 -> iris_virginica(7.2, 3.6, 6.1, 2.5).
0 -> iris_setosa(6.5, 3.2, 5.1, 2.0).
0 -> iris_versicolor(6.5, 3.2, 5.1, 2.0).
1 -> iris_virginica(6.5, 3.2, 5.1, 2.0).
0 -> iris_setosa(6.4, 2.7, 5.3, 1.9).
0 -> iris_versicolor(6.4, 2.7, 5.3, 1.9).
1 -> iris_virginica(6.4, 2.7, 5.3, 1.9).
0 -> iris_setosa(6.8, 3.0, 5.5, 2.1).
0 -> iris_versicolor(6.8, 3.0, 5.5, 2.1).
1 -> iris_virginica(6.8, 3.0, 5.5, 2.1).
0 -> iris_setosa(5.7, 2.5, 5.0, 2.0).
0 -> iris_versicolor(5.7, 2.5, 5.0, 2.0).
1 -> iris_virginica(5.7, 2.5, 5.0, 2.0).
0 -> iris_setosa(5.8, 2.8, 5.1, 2.4).
0 -> iris_versicolor(5.8, 2.8, 5.1, 2.4).
1 -> iris_virginica(5.8, 2.8, 5.1, 2.4).
0 -> iris_setosa(6.4, 3.2, 5.3, 2.3).
0 -> iris_versicolor(6.4, 3.2, 5.3, 2.3).
1 -> iris_virginica(6.4, 3.2, 5.3, 2.3).
0 -> iris_setosa(6.5, 3.0, 5.5, 1.8).
0 -> iris_versicolor(6.5, 3.0, 5.5, 1.8).
1 -> iris_virginica(6.5, 3.0, 5.5, 1.8).
0 -> iris_setosa(7.7, 3.8, 6.7, 2.2).
0 -> iris_versicolor(7.7, 3.8, 6.7, 2.2).
1 -> iris_virginica(7.7, 3.8, 6.7, 2.2).
0 -> iris_setosa(7.7, 2.6, 6.9, 2.3).
0 -> iris_versicolor(7.7, 2.6, 6.9, 2.3).
1 -> iris_virginica(7.7, 2.6, 6.9, 2.3).
0 -> iris_setosa(6.0, 2.2, 5.0, 1.5).
0 -> iris_versicolor(6.0, 2.2, 5.0, 1.5).
1 -> iris_virginica(6.0, 2.2, 5.0, 1.5).
0 -> iris_setosa(6.9, 3.2, 5.7, 2.3).
0 -> iris_versicolor(6.9, 3.2, 5.7, 2.3).
1 -> iris_virginica(6.9, 3.2, 5.7, 2.3).
0 -> iris_setosa(5.6, 2.8, 4.9, 2.0).
0 -> iris_versicolor(5.6, 2.8, 4.9, 2.0).
1 -> iris_virginica(5.6, 2.8, 4.9, 2.0).
```

```
0 -> iris_setosa(7.7, 2.8, 6.7, 2.0).
0 -> iris_versicolor(7.7, 2.8, 6.7, 2.0).
1 -> iris_virginica(7.7, 2.8, 6.7, 2.0).
0 -> iris_setosa(6.3, 2.7, 4.9, 1.8).
0 -> iris_versicolor(6.3, 2.7, 4.9, 1.8).
1 -> iris_virginica(6.3, 2.7, 4.9, 1.8).
0 -> iris_setosa(6.7, 3.3, 5.7, 2.1).
0 -> iris_versicolor(6.7, 3.3, 5.7, 2.1).
1 -> iris_virginica(6.7, 3.3, 5.7, 2.1).
0 -> iris_setosa(7.2, 3.2, 6.0, 1.8).
0 -> iris_versicolor(7.2, 3.2, 6.0, 1.8).
1 -> iris_virginica(7.2, 3.2, 6.0, 1.8).
0 -> iris_setosa(6.2, 2.8, 4.8, 1.8).
0 -> iris_versicolor(6.2, 2.8, 4.8, 1.8).
1 -> iris_virginica(6.2, 2.8, 4.8, 1.8).
0 -> iris_setosa(6.1, 3.0, 4.9, 1.8).
0 -> iris_versicolor(6.1, 3.0, 4.9, 1.8).
1 -> iris_virginica(6.1, 3.0, 4.9, 1.8).
0 -> iris_setosa(6.4, 2.8, 5.6, 2.1).
0 -> iris_versicolor(6.4, 2.8, 5.6, 2.1).
1 -> iris_virginica(6.4, 2.8, 5.6, 2.1).
0 -> iris_setosa(7.2, 3.0, 5.8, 1.6).
0 -> iris_versicolor(7.2, 3.0, 5.8, 1.6).
1 -> iris_virginica(7.2, 3.0, 5.8, 1.6).
0 -> iris_setosa(7.4, 2.8, 6.1, 1.9).
0 -> iris_versicolor(7.4, 2.8, 6.1, 1.9).
1 -> iris_virginica(7.4, 2.8, 6.1, 1.9).
0 -> iris_setosa(7.9, 3.8, 6.4, 2.0).
0 -> iris_versicolor(7.9, 3.8, 6.4, 2.0).
1 -> iris_virginica(7.9, 3.8, 6.4, 2.0).
0 -> iris_setosa(6.4, 2.8, 5.6, 2.2).
0 -> iris_versicolor(6.4, 2.8, 5.6, 2.2).
1 -> iris_virginica(6.4, 2.8, 5.6, 2.2).
0 -> iris_setosa(6.3, 2.8, 5.1, 1.5).
0 -> iris_versicolor(6.3, 2.8, 5.1, 1.5).
1 -> iris_virginica(6.3, 2.8, 5.1, 1.5).
0 -> iris_setosa(6.1, 2.6, 5.6, 1.4).
0 -> iris_versicolor(6.1, 2.6, 5.6, 1.4).
1 -> iris_virginica(6.1, 2.6, 5.6, 1.4).
0 -> iris_setosa(7.7, 3.0, 6.1, 2.3).
0 -> iris_versicolor(7.7, 3.0, 6.1, 2.3).
1 -> iris_virginica(7.7, 3.0, 6.1, 2.3).
0 -> iris_setosa(6.3, 3.4, 5.6, 2.4).
0 -> iris_versicolor(6.3, 3.4, 5.6, 2.4).
1 -> iris_virginica(6.3, 3.4, 5.6, 2.4).
0 -> iris_setosa(6.4, 3.1, 5.5, 1.8).
0 -> iris_versicolor(6.4, 3.1, 5.5, 1.8).
1 -> iris_virginica(6.4, 3.1, 5.5, 1.8).
0 -> iris_setosa(6.0, 3.0, 4.8, 1.8).
0 -> iris_versicolor(6.0, 3.0, 4.8, 1.8).
1 -> iris_virginica(6.0, 3.0, 4.8, 1.8).
0 -> iris_setosa(6.9, 3.1, 5.4, 2.1).
0 -> iris_versicolor(6.9, 3.1, 5.4, 2.1).
1 -> iris_virginica(6.9, 3.1, 5.4, 2.1).
0 -> iris_setosa(6.7, 3.1, 5.6, 2.4).
0 -> iris_versicolor(6.7, 3.1, 5.6, 2.4).
1 -> iris_virginica(6.7, 3.1, 5.6, 2.4).
0 -> iris_setosa(6.9, 3.1, 5.1, 2.3).
0 -> iris_versicolor(6.9, 3.1, 5.1, 2.3).
1 -> iris_virginica(6.9, 3.1, 5.1, 2.3).
0 -> iris_setosa(5.8, 2.7, 5.1, 1.9).
0 -> iris_versicolor(5.8, 2.7, 5.1, 1.9).
1 -> iris_virginica(5.8, 2.7, 5.1, 1.9).
0 -> iris_setosa(6.8, 3.2, 5.9, 2.3).
0 -> iris_versicolor(6.8, 3.2, 5.9, 2.3).
1 -> iris_virginica(6.8, 3.2, 5.9, 2.3).
0 -> iris_setosa(6.7, 3.3, 5.7, 2.5).
0 -> iris_versicolor(6.7, 3.3, 5.7, 2.5).
1 -> iris_virginica(6.7, 3.3, 5.7, 2.5).
0 -> iris_setosa(6.7, 3.0, 5.2, 2.3).
0 -> iris_versicolor(6.7, 3.0, 5.2, 2.3).
1 -> iris_virginica(6.7, 3.0, 5.2, 2.3).
0 -> iris_setosa(6.3, 2.5, 5.0, 1.9).
0 -> iris_versicolor(6.3, 2.5, 5.0, 1.9).
1 -> iris_virginica(6.3, 2.5, 5.0, 1.9).
0 -> iris_setosa(6.5, 3.0, 5.2, 2.0).
0 -> iris_versicolor(6.5, 3.0, 5.2, 2.0).
1 -> iris_virginica(6.5, 3.0, 5.2, 2.0).
```

```
0 -> iris_setosa(6.2, 3.4, 5.4, 2.3).
0 -> iris_versicolor(6.2, 3.4, 5.4, 2.3).
1 -> iris_virginica(6.2, 3.4, 5.4, 2.3).
0 -> iris_setosa(5.9, 3.0, 5.1, 1.8).
0 -> iris_versicolor(5.9, 3.0, 5.1, 1.8).
1 -> iris_virginica(5.9, 3.0, 5.1, 1.8).
```

Appendix D

Lattice L95

```
% Elements
member(pos_inf).    member(neg_inf).    member(X) :- number(X).
members([0.95, 0.25, 0.50, -0.25, -0.95, -0.50]).

% Distance
distance(X,Y,Z) :- (X=pos_inf;X=neg_inf;Y=pos_inf;Y=neg_inf),!,
    current_prolog_flag(max_tagged_integer,Z).
distance(X,Y,Z) :- Z is abs(Y-X).

% Ordering relation
leq(_,pos_inf).    leq(neg_inf,_).    leq(X,Y) :- X =< Y.

% Supremum and infimum
bot(neg_inf).     top(pos_inf).

% Binary operations
or_sup(X,Y,Y) :- leq(X,Y).
or_sup(X,_,X).
and_sup(X,Y,X) :- leq(X,Y).
and_sup(_,Y,Y).

% Aggregators
agr_add(pos_inf,_,pos_inf). agr_add(_,pos_inf,pos_inf).
agr_add(neg_inf,_,neg_inf). agr_add(_,neg_inf,neg_inf).
agr_add(X,Y,Z) :- Z is X+Y.

agr_prod(pos_inf,_,pos_inf). agr_prod(neg_inf,_,neg_inf).
agr_prod(_,pos_inf,pos_inf). agr_prod(_,neg_inf,neg_inf).
agr_prod(X,Y,Z) :- Z is X*Y.

agr_sigmoid(pos_inf,pos_inf).
agr_sigmoid(neg_inf,neg_inf).
agr_sigmoid(X,Y) :- Y is 1 / (1+exp(-X)).
```

```

agr_binary(pos_inf,pos_inf).
agr_binary(neg_inf,neg_inf).
agr_binary(X,Y) :- (X<0 -> Y=0; Y=1).

agr_softplus(pos_inf,pos_inf).
agr_softplus(neg_inf,neg_inf).
agr_softplus(X,Y) :- Y is log(1+exp(X)).

agr_softsign(pos_inf,pos_inf).
agr_softsign(neg_inf,neg_inf).
agr_softsign(X,Y) :- Y is (X)/(1+abs(X)).

agr_relu(pos_inf,pos_inf).
agr_relu(neg_inf,neg_inf).
agr_relu(X,Y) :- (X<0 -> Y=0; Y=X).

agr_leaky_relu(pos_inf,pos_inf).
agr_leaky_relu(neg_inf,neg_inf).
agr_leaky_relu(X,Y) :- (X<0 -> Y is 0.01*X; Y=X).

agr_tanh(pos_inf,pos_inf).
agr_tanh(neg_inf,neg_inf).
agr_tanh(X,Y) :- Y is ((exp(X)-exp(-X))/(exp(X)+exp(-X))).

agr_arctan(pos_inf,pos_inf).
agr_arctan(neg_inf,neg_inf).
agr_arctan(X,Y) :- Y is atan(X).

agr_sinusoid(pos_inf,pos_inf).
agr_sinusoid(neg_inf,neg_inf).
agr_sinusoid(X,Y) :- (X=0 -> Y is 1; Y is (sin(X)/X)).

agr_identity(pos_inf,pos_inf).
agr_identity(neg_inf,neg_inf).
agr_identity(X,Y) :- Y is X.
```

Appendix E

CD Content

The artifact attached with this document (called CD), contains all the necessary files for running Neuro-FLOPER in addition to this document itself. In the root directory we can find:

- The folder "Neuro-FLOPER", with all the files (in Python) that compounds the program. For running it, is necessary to have the corresponding framework in the current machine (Tensorflow, Keras and other packages in Python 2.7).
- The folder "Tests", with all the files used in Chapter 5.
- The folder "Iris", with the dataset, script and labels file for generating a neural network in Keras.
- This document.