Marcus Vinícius Mazega Figueredo

A Learning Algorithm for Constructive Neural Networks Inspired on Decision Trees and Evolutionary Algorithms

A thesis submitted for the degree of Doctor of Philosophy at the Pontifícia Universidade Católica do Paraná of Curitiba, Paraná, Brazil.

Curitiba
2013

Marcus Vinícius Mazega Figueredo

A Learning Algorithm for Constructive Neural Networks Inspired on Decision Trees and Evolutionary Algorithms

A thesis submitted for the degree of Doctor of Philosophy at the Pontifícia Universidade Católica do Paraná of Curitiba, Paraná, Brazil.

Major Field: Computer Science

Adviser: Júlio Cesar Nievola

Curitiba
2013

Figueredo, Marcus Vinícius Mazega.
A Learning Algorithm for Constructive Neural Networks Inspired on Decision Trees and Evolutionary Algorithms. Curitiba, 2013.

Thesis - Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática.

1. Constructive neural networks 2. Neural trees 3. Easy learning problems I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e Tecnologia. Programa de Pós-Graduação em Informática.

To God, because of everything.
To my family, because of who I am.
To my masters, because of their lessons.

Acknowledgements

Initially, I would like to say that all my work would be impossible without the scholarship that was granted to me by the University. I would like to thank Prof. Júlio Cesar Nievola, who has been my advisor during all my postgraduate life. His important lessons were essential to my understanding of Artificial Intelligence and neural networks.

It is important to remind several teachers from my past, like professor João Dias da Silva, Robert Carlisle Burnett, James Baraniuk, Henri Eberspächer, Lourival Lippmann Jr and Maria Te Vaarwerk. Each one was very important during my academic journey.

I cannot forget to thank my friends Sérgio Renato Rogal Jr, Alfredo Beckert Neto and Carlos Eduardo Liparotti Chaves, who were always present in my projects.

I also want to remember my family: Liliane, Álvaro, Maria, Fernanda and Thiago. They supported me during my entire work.

Finally, I would like to thank everyone who directly or indirectly helped me in the execution of this work and, by accident, is not mentioned here.

Contents

List of Figures

List of Tables

List of Abbreviations

ML	*machine learning*
AI	*artificial intelligence*
MLP	*multilayer perceptron*
FNT	*flexible neural tree*
GP	*genetic programming*
PSO	*particle swarm optimization*
EP	*evolutionary programming*
PIPE	*probabilistic incremental program evolution*
GGGP	*Grammar Guided Genetic Programming*
MIMO-FNT	*multiple-input and multiple-output flexible neural tree*
IP	*immune programming*
PIPE	*probabilistic incremental program evolution*
SA	*simulation annealing*
PCA	*principal component analysis*
LDA	*linear discriminant analysis*
GUWL	*growing up with learning*
SGU	*simple growing up*
GNT	*generalized neural tree*
LTU	*linear threshold units*
SGNT	*Self-Generating Neural Tree*

QUANT	*quadratic-neuron-based neural tree*
UTM	*Universal Turing Machine*

List of Algorithms

Abstract

Inspired on decision trees and evolutionary algorithms, this thesis proposes a learning algorithm of constructive neural networks that relies on three statements: to layout the neurons in a tree-like structure; to train each neuron individually; and, to optimizate all the weights using an evolutionary approach. This way, it is expected to advance in two main questions concerning multilayer perceptrons (MLPs): how to determine the network architecure and how to build more comprehensible models. Based on the normalized information gain of each attribute, the constructive algorithm builds the network architecture. In the process, the algorithm automatically creates a set of training examples for each individual neuron and executes single-cell learning. Once the network is complete and trained, particle swarm optimization is utilized to evolve the connections of the network. Six metrics are utilized to validate the method when compared to decision trees and MLPs: accuracy, sensitivity, specificity, precision, computational cost and comprehensibility. The method was tested in 13 different data sets and proved to create neural networks that are more comprehensible than traditional MLPs, without degradation in the classification performance.

Keywords: Constructive algorithms, neural trees, neural networks, easy learning problems, comprehensibility.

Chapter 1

Introduction

Among the several machine learning methods, artificial neural networks are one of the most popular approaches. They are succesfully utilized in a large sort of applications, such as pattern classification, optimization, function aproximation, and many others. However, what are neural networks? And, most important, why are they so interesting?

Artificial neural networks have their inspiration on biological neurons, composed of dendrites, cell body and axons. The main idea is summarized by the reductionist approach: if you can reproduce with sufficient detail a biological "machine" that conducts an intelligent behavior (like the brain), then your system will exhibit an intelligent behavior (ALMEIDA, 2006).

The natural neurons are generalized in mathematical models based on the following statements (FAUSSET, 1994):

- Information is processed by simple units called neurons.
- Signals are transmitted between neurons through connections.
- Each connection has an associated weight, that is multiplied by the transmitted signal.
- Each neuron utilizes an activation function to calculate the output signal.

This way, a neural network can be defined as a connectionist model, with interconnected neurons in many possible topologies. They can be applied in problems where: resolution rules are unknow or hard to formalize; there is a large amount of examples and solutions; great speed is needed in the problem solving; or, there are not current tecno-

logical solutions. The application domains involve shape recognition, signal processing, view, speech, prediction, modelling, decision support and robotics (GURNEY, 1997).

One of the most important steps during neural networks utilization is the training, when the network "learns" how to solve a problem. This learning can be supervised (with previously solved examples) or unsupervised. Supervised training are mostly utilized in classification problems, while the unsupervised methods are preffered to grouping problems.

Despite the fact that there are many possible topologies and learning algorithms, multilayer feedforward networks trained by backpropagation algorithm are probably one of the most utilized strategies. Indeed, this technique, that utilizes mean squared error and gradient descent, became the most important and most widely used algorithm for connectionist learning (GALLANT, 1994). Much of this success and popularity probably came from the good results that this approach often presents.

It is relevant to remind that this algorithm searchs for a solution in *a priori* fixed network topology. During the search, only the weights are changed. This situation forces the user to select an adequate topology as a requirement for a good network behavior. Nevertheless, there are no efficient method for determining the optimal network topology for a given problem (PAREKH; YANG; HONAVAR, 2000). This is very significative, because a too small network may not learn the domain and a too large network can mask an overfitting problem.

1.1 Motivations and Challenges

The topology paradox stimulated many authors to develop constructive neural-network learning algorithms. The motivation for these algorithms is to transform the hard task of building a network into the easier problem of single-cell learning (GALLANT, 1994). Usually, they start with a single neuron and, as necessary, grow the network through the addition and training of new neurons. According to Parekh, Yang and Honavar (PAREKH; YANG; HONAVAR, 2000), the key motivations for studying constructive neural-network learning algorithms are:

1. They overcome the limitation of searching in *a priori* fixed topology.

2. They usually produce smaller networks, which are more efficient [1].

[1] Honavar believes that a network is more efficient if it can achieve a similar classification rate with

3. You can determine the complexity of a problem by measuring the size of the constructed network. More complex problems need extra neurons.

4. You can choose some variables to optimize, such as learning time or network size.

5. You can use prior knowledge to start the network architecture.

6. They present lifelong learning.

A particular case of constructive learning algorithms involves the *easy learning problems*. These are problems where it is given a network, and where the training examples specify input values and correct activations for all output and intermediate cells (GALLANT, 1994). They are called "easy" because it is possible to decompose the problem and train each single cell. Theoretically, an easy learning algorithm may provide a faster solution, since training each cell separately is an easier task than synchronously training interconnected cells. However, practical work shows that is very hard to determine correct activations of the intermediate cells. Indeed, it was proven that this is a NP-complete problem (BLUM; RIVEST, 1992).

Thus, it is possible to say that the main problem concerning easy learning algorithms involves the discovery of correct activations of the intermediate cells. Three solutions may be considered:

1. To utilize a database with training examples that specifies the input and output values for all cells.

2. A specialist may define the topology of network and establish the input and output values for all cells.

3. To find an automatic way (an algorithm) to define the topology of network and establish the input and output values for all cells.

The two first hypotheses are easily discarded, because they are not feasible. It is very hard to find a database with the inputs and outputs of the intermediate cells. This way, this option could not be considered to a generic method. On the other hand, it is not convenient to ask a specialist to define these values. Indeed, if you have a specialist, there are better artificial intelligence methods instead of easy learning algorithms.

So, finding an heuristic should be the better hypothesis to follow. This heuristic must comply with the following statements:

---

less neurons and connections

- It should learn automatically from a database with examples.

- It should define automatically the network's topology.

- It should establish automatically the input and output values of all cells, including the intermediate ones. Each neuron should be trained alone, in a single-cell learning approach.

Decision tree algorithms learn from examples and create topologies of *if-then* clauses. Theses clauses can be utilized to create inputs and outputs for each cell. In other words, it is possible to train each neuron separately, according to single-cell training method. Therefore, it is legitimate to suppose that one can derive a decision tree construction algorithm in a easy learning algorithm for constructive neural networks.

Hybrid systems involving neural networks and decision trees are not something new. In fact, these two methods have been combined in several ways over the years. Indeed, neural networks with tree-based structures were introduced since 1989 (UTGOFF, 1989). Moreover, these hybrid models are usually called *neural trees*. This approach involves the creation of a decision tree and the conversion into a matching neural network. Nonetheless, there is no method in the literature that proposes an unique algorithm (inspired in decision trees) that directly creates the network from the examples. Such method would be classified as a constructive learning algorithm of neural networks, which is conceptually the opposite of usual neural trees.

Another question that arises from the creation of neural trees is the post-training approach. The post-training is very important, because a newborn network has the same accuracy of the original decision tree. This way, it is necessary to train this tree so it can overcome the primal perfomance. It is clear that a conventional algorithm, such as backpropagation, could do this training. However, backpropagation is the complete opposite of the concept of constructive networks. Then, it is very important to search for another way in this training, in order to keep its consistency.

Evolutionary algorithms may fit very well in this training. In truth, there are many papers that describe the good results of different techniques, like genetic algorithms or particle swarm optimization, in neural networks training. Considering that this work is about constructive techniques and reminding that evolutionary algorithms are conceptually parallel methods, it is reasonable to propose their utilization as post-training method.

1.2 Research Contributions

Inspired on decision trees and evolutionary algorithms, this thesis aims to propose a learning algorithm of constructive neural networks for classification applications. The algorithm relys on the following statements:

1. To layout the neurons in a tree structure.
2. To train each neuron individually.
3. To optimizate all the weights using an evolutionary approach.

Following this scheme, it is expected to find a feasible solution to the implementation of automatic easy learning algorithms. Also, it should reduce the computational cost of initialising and training multilayer perceptrons, since the construction of the network is guided with a very clear heuristic. Additionaly, evolutionary algorithms may considerably refine the accuracy of the model, providing good results. And, finally, there is the interest in analyzing the consequences of mixing white-box (decision trees) and black-box (neural networks) models, mainly in the context of knowledge representation.

1.3 Thesis Structure

This document is organized as follows.

In chapter 2, important definitions about artificial neural networks are introduced, mainly the representation issues and basic learning algorithms for single-neurons systems (perceptrons). Multilayer perceptrons are also presented, including the backpropagation learning algorithm.

Chapter 3 is based on the constructive neural networks and their advantages compared to MLPs. Several constructive learning algorithms are described in this chapter.

Then, the chapter 4 reviews the various hybrid models that involve neural networks, decision trees and evolutionary algorithms. The methods are classified according to their characteristic and the most significant algorithms are described.

Chapter 5 describes the problem and the assumed hypothesis. The proposal is also explained in this chapter, in details. The three main aspects of the theory (topology construction, single-cell learning and weights optimization) are described separately.

The methodology of the project is defined in chapter 6 as so the chosen databases and the reasons that motivated these choices. Also, the chapter defines the ML (*machine learning*) techniques that will be compared to the proposed method. Additionaly, the utilized metrics are explained as well as the structure of each experiment.

Chapter 7 reveals the results of this work, which are deeply analyzed and compared to traditional methods.

Finally, chapter 8 brings the conclusion of this thesis.

Chapter 2

Foundations of Neural Networks

The brain is composed by billions of interconnected neurons, where each neuron is a cell that receives, processes and transmits signals using electrochemical reactions. As shown in figure 2.1, a typical neuron is composed by three parts: dendrites, cell body (or soma) and axon.

Dendrites are the several filaments that branch from the cell body. They conduct the electrochemical signals that came from the neighbor neural cells to the soma. On the other hand, the axon is the single projection that outputs electrochemical signals from the cell body to neighbor neurons.

This way, the brain can be defined as a network of neurons, where the axon of each neural cell is connected to the dendrites of many others. The signals are transmitted electrochemically via synapses, which are the functional connections between neural cells. Actually, a synapse is a small gap that exists between the axon of a neuron and the dendrite of other cell. The axon launchs neurotransmitters in the synapse that excite the dendrites, transmitting the information. The strength of the transmitted signal is determinated by physical and chemical characteristics of each synapse.

The brain can be viewed as a massive parallel processor. The input signals from many sensors (sight, hearing, taste, smell, touch, and others) are transmitted and processed through billions of synapses, resulting in output actions. Any change in the synapses circuit affects the output. In a trial and error process, the synapses are updated, improving the output. In other words, the brain "learns".

> *"Indeed, learning occurs as a result of changing the effectiveness of synapses so that their influence on other neurons also changes. Learning is a function of the effectiveness of synapses to propagate signals and initiate new signals*

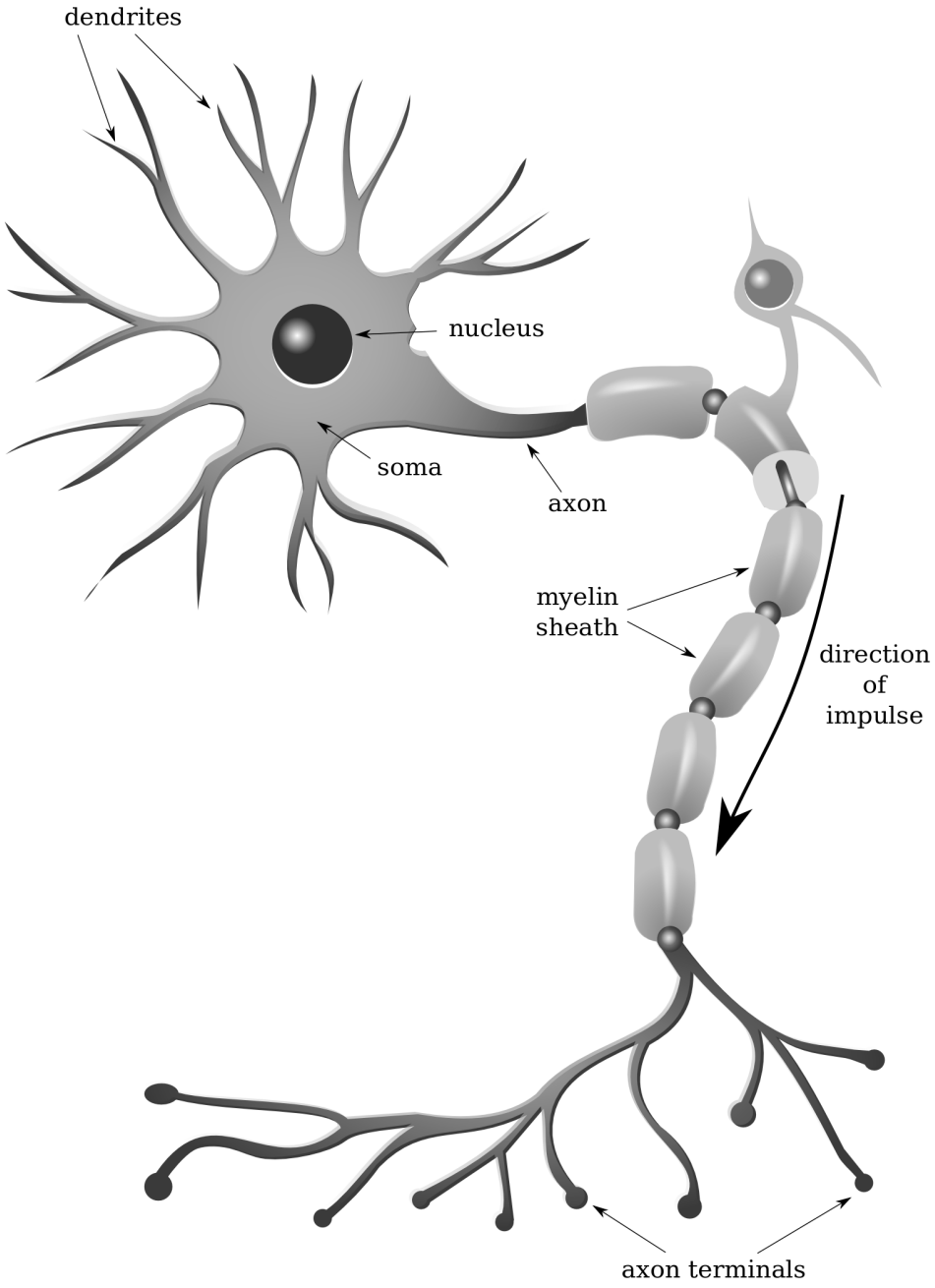

Figure 2.1: Biological neuron.

along connecting neurons. Learning and experience change the structure of the neural networks."(HINTON, 1992)

2.1 Artificial Neurons

Artificial neural networks are biologically inspired on the brain, which is composed by billions of interconnected neurons. So, to reproduce the intelligent behavior of the brain, one can reproduce the behavior of the biological neurons and their connections. It is a reductionist approach, where an intelligent behavior is achieved by mimetizing the behavior of the components of an intelligent system.

The first mathematical model of a neuron was introduced by Mcculloch and Pitts in 1943 (MCCULLOCH; PITTS, 1943). In that paper, they described a neural unit that sums the inputs of several sources and, if the total value overcomes a defined threshold, transmits an output of 1.0 (active). If not, it produces an output of 0.0 (inactive). Each input (or synapse) can be weighted, where the weight multiplies the respective input during the sum. Although being a very simple model, it reproduces the behavior of a biological neuron with sufficient detail. Indeed, their artificial neuron is a tunable linear threshold unit that can compute any logical expression, when it is arranged in a network with many interconnected units and tuned weights. However, they did not describe how to tune the weights, which means that there was no training algorithm for those initial networks.

In 1949, Donald Hebb proposed that the activation (or not) of neural synapses make them stronger (or weaker). In this hypothesis, the utilized synapses during the execution of a task are reinforced. Actually, it is stated that synapses could be created or destroyed depending of the frequency that the respective neurons are activated or not:

"Let us assume that the persistence or repetition of a reverberatory activity (or 'trace') tends to induce lasting cellular changes that add to its stability. The assumption can be precisely stated as follows: When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." (HEBB, 1949)

This neurophysiological postulated originated the "Hebb's rule", that was used as

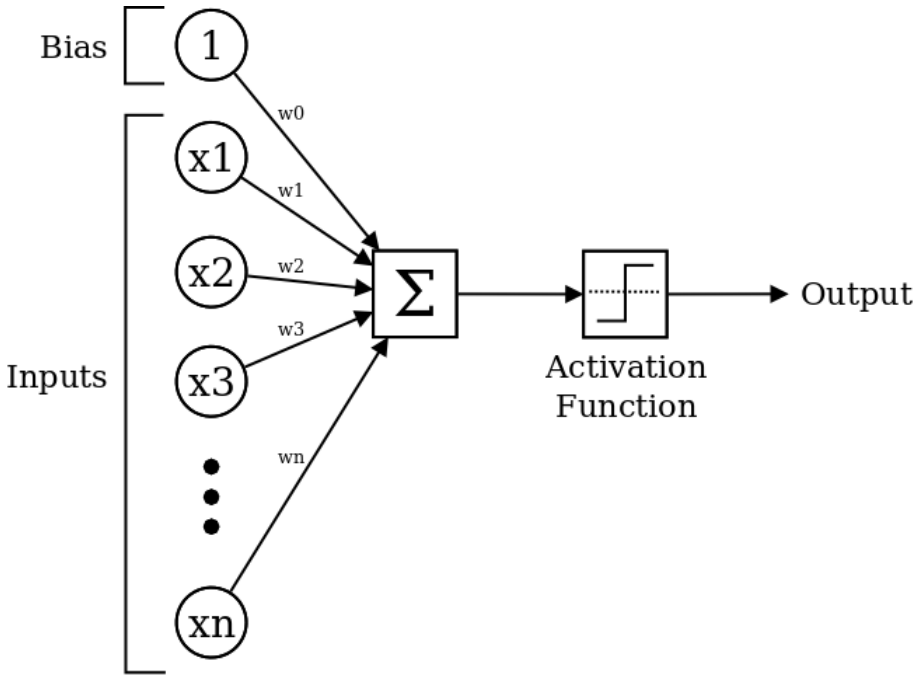

Figure 2.2: The perceptron.

the heuristic of several learning algorithms. This principle is utilized to alter the weights between artificial neurons. Basically, when two connected neurons are activated at the same time, the weight of their connection is increased. If just one of them is active, the same weight is reduced.

2.2 Perceptron

The "Hebb's rule" was succesfuly implemented in 1958 by Frank Rosenblatt, when he invented the perceptron (ROSENBLATT, 1958). As seen in the figure 2.2, it is a binary classifier that maps its inputs to a single binary value.

The classification function of perceptron is defined by:

$$f(x) = \begin{cases} 1 & if \quad w \cdot x + b > 0 \\ -1 & else \end{cases}$$

where w is a vector of real-valued weights, x is a vector of real input values and b is the "bias", a constant real value that does not vary according the inputs.

Given an instance represented by its input values, the perceptron will classify it as 1 or -1. So, it acts as a linear classifier that separates several instances with a hyperplane, as seen in the figure 2.3.

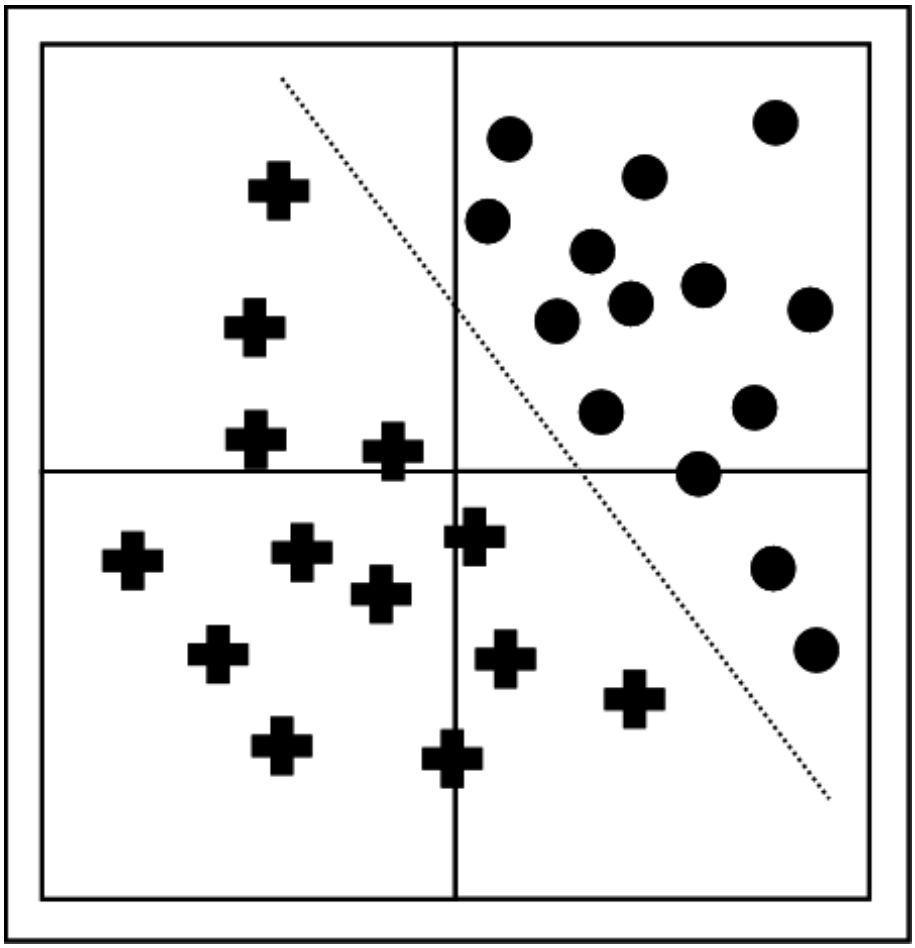

Figure 2.3: Perceptron uses a hyperplane to separate instances.

The intelligent behavior of a perceptron is determined by its weights and bias. Indeed, one can tell that the "knowledge" of a perceptron is stored in these values. Actually, the perceptron "learns" by updating its weights and bias. One of the great advances of Rosenblatt's work was to present the algorithm for perceptron learning, which made it possible to automatically train perceptrons with examples and solve several problems.

The perceptron learning algorithm is given as pseudocode in algorithm 1:

Algorithm 1 Perceptron learning algorithm

Require: A group of training examples and corresponding classifications.
Ensure: A vector of weights w that correctly classifies all training examples if they are linearly separable.

```
w ← 0
while w does not correctly classifies all examples do
    x ← training example
    c ← correct classification
    if (w · x >= 0 and c = +1) or (w · x < 0 and c = −1) then
        do nothing
    else
        w ← w + c · x
    end if
end while
```

The perceptron learning algorithm can be geometrically interpreted (figure 2.4), as did Stephen Gallant:

> *"Each training example corresponds to a vector with 'tail' at the origin and 'head' at x. The set of weights is also a vector, with head at w. The algorithm seeks a vector w that has positive projection (inner product) with all training examples labeled '1' and negative projection with all training examples labeled '-1'."(GALLANT, 1994)*

For example, consider a single-cell problem proposed by Gallant (GALLANT, 1994). His training examples are given in table 2.1. Note that the bias are added to the examples as a fixed input ("1"). This way, the algorithm can update the weights and the bias at the same time. Applying the perceptron learning algorithm to these examples, it produces the interactions sequence listed in table 2.2.

After 12 iterations, the algorithms finds an array of weights that solves the problem. It is relevant to observe that the algorithm randomly chose the order of the training examples, during learning process. A different order will result in more or less iterations.

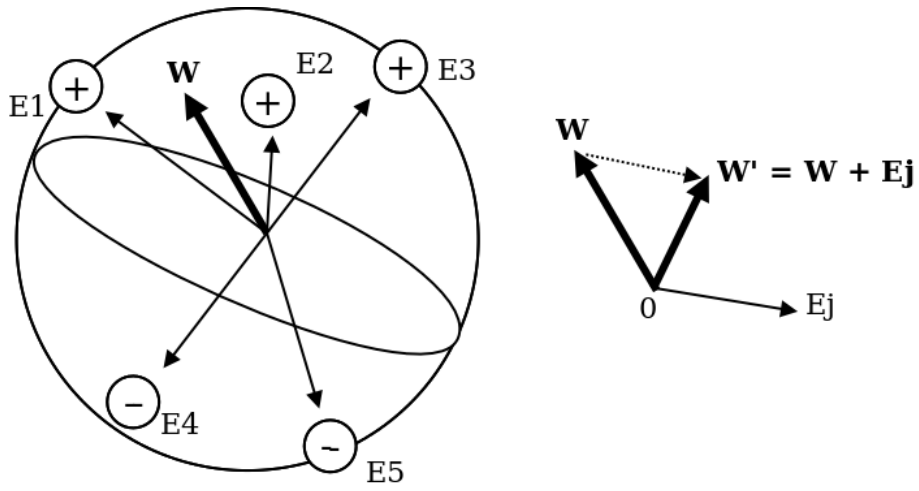

Figure 2.4: Geometrical interpretation of perceptron learning algorithm.

Table 2.1: Training examples of "AND" logical operation.

Training Example	(Bias, x1, x2)	c
X1	(+1, -1, -1)	-1
X2	(+1, +1, -1)	-1
X3	(+1, +1, +1)	+1

Table 2.2: Perceptron learning interactions over training examples of "AND" logical operation.

Interation	Current Example	Current Weights	OK?	Action
1	(1 -1 -1 -1)	(0 0 0 0)	No	Update weights
2	(1 1 -1 -1)	(1 -1 -1 -1)	No	Update weights
3	(1 -1 -1 -1)	(0 -2 0 0)	Yes	Do nothing
4	(1 1 1 1)	(0 -2 0 0)	No	Update weights
5	(1 -1 -1 -1)	(1 -1 1 1)	No	Update weights
6	(1 1 1 1)	(2 -2 0 0)	No	Update weights
7	(1 -1 -1 -1)	(3 -1 1 1)	Yes	Do nothing
8	(1 1 -1 -1)	(3 -1 1 1)	No	Update weights
9	(1 -1 -1 -1)	(2 -2 2 2)	No	Update weights
10	(1 -1 -1 -1)	(3 -3 1 1)	Yes	Do nothing
11	(1 1 -1 -1)	(3 -3 1 1)	Yes	Do nothing
12	(1 -1 -1 -1)	(3 -3 1 1)	Yes	End algorithm

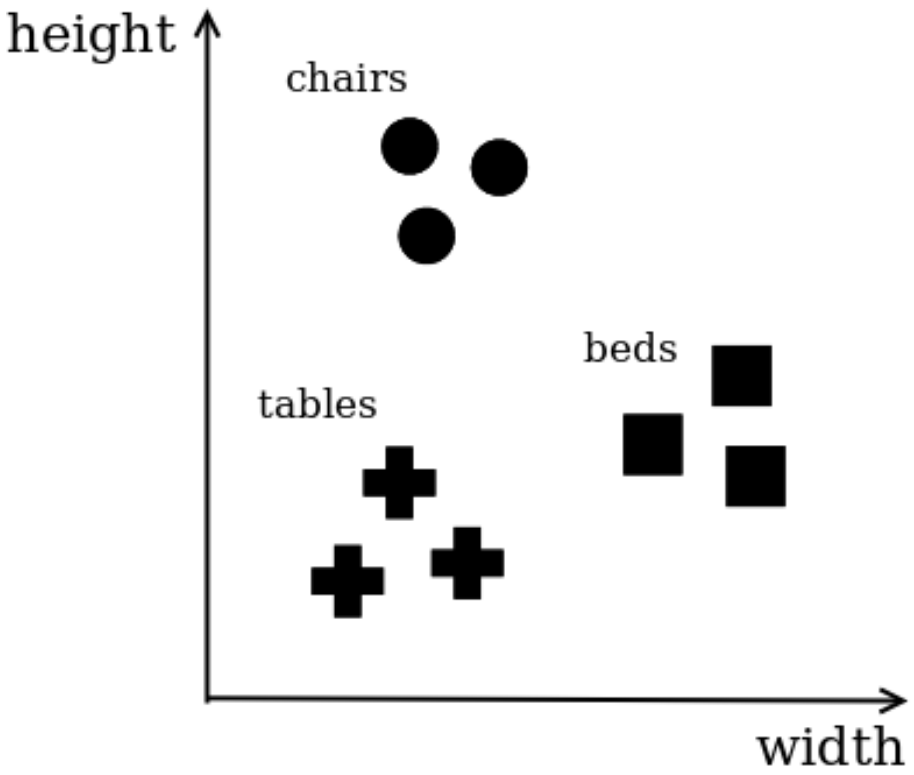

Figure 2.5: A set with three linearly separable classes.

Anyway, the algorithm will always find a solution for a finite set of linearly separable training examples. The statement that perceptron learning algorithm always converges when a solution exists is called "Perceptron Convergence Theorem" and was proposed by Frank Rosenblatt (ROSENBLATT, 1962).

The previous example is a 2-classes problem (-1 and +1), which could be solved with a single neuron. However, any number of classes is possible, provided that they are linearly separable (like the training examples of figure 2.5). In these cases, more than one neuron is necessary, as the single layer perceptron shown in figure 2.6. Basically, each perceptron is trained separately and divides the instances with a hyperplane (figure 2.7).

2.3 Pocket Algorithm

It is relevant to notice that the perceptron learning algorithm only uses the negative reinforcement to adjust the weights. In other words, it fully ignores the correct classifications. Aware of this situation, Stephen Gallant proposed the Pocket algorithm (GALLANT, 1990), which also utilizes positive reinforcement during the training. To do this, it keeps a separate set of weights named w^{pocket} "in the pocket", where w^{pocket} has

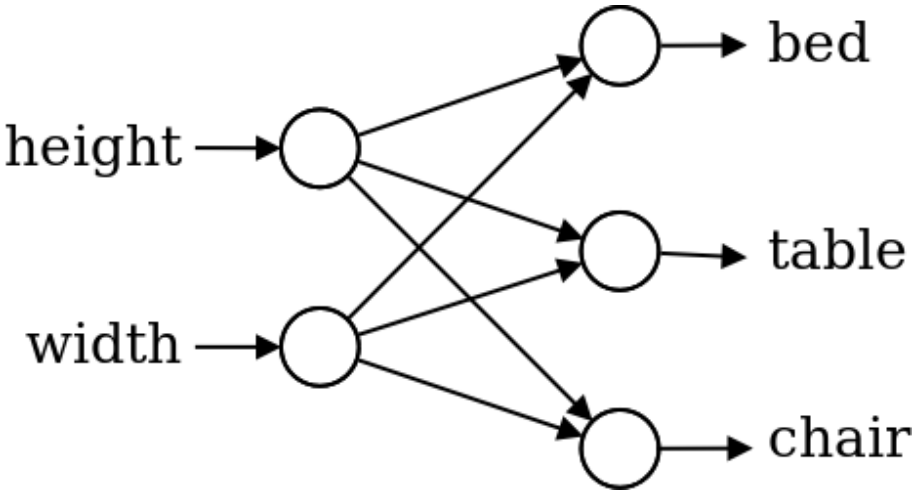

Figure 2.6: A single layer perceptron.

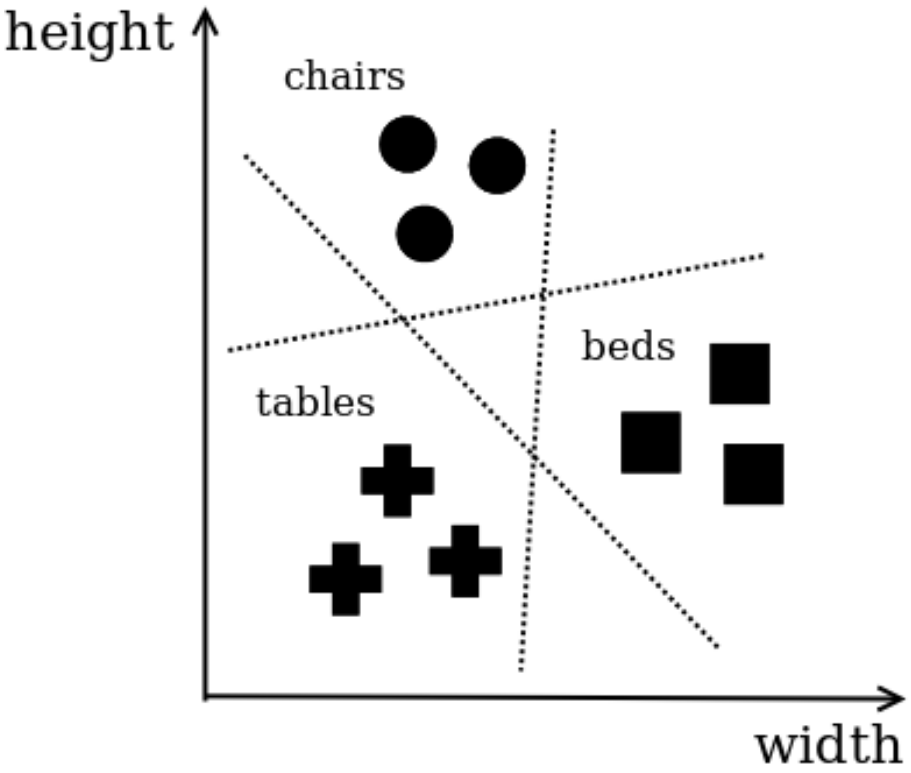

Figure 2.7: Each perceptron divides the instances with a hyperplane.

the largest number of consecutive correct classifications.

The Pocket algorithm is presented as pseudocode in algorithm 2:

Algorithm 2 Pocket algorithm

Require: A group of training examples and corresponding classifications.
Ensure: A vector of weights w that correctly classifies all training examples if they are linearly separable.

```
w ← 0
w^pocket ← 0
while w does not correctly classifies all examples do
  x ← training example
  c ← correct classification
  if (w · x >= 0 and c = +1) or (w · x < 0 and c = −1) then
    if w has more consecutive correct classifications than w^pocket then
      w^pocket ← w
    end if
  else
    w ← w + c · x
  end if
end while
```

Despite being a very interesting optimization, this algorithm introduces another weakness. It is possible to exist a set of weights with the best count of consecutive correct classifications, but without the largest total of correct classifications. Stephen Gallant called this situation as a "lucky run". To avoid that, it proposed another optimization called "ratchet". Basically, it consists in checking if the new set of weights has the largest total of correct classifications before adding it to the pocket. This way, the algorithm will always choose the best set. However, this kind of approach can only be applied in situations with a limited number of training examples, once it demands that all examples must be tested before a change in the pocket.

This algorithm, which is called the Pocket algorithm with ratchet, is presented as pseudocode in algorithm 3.

2.4 Multilayer Perceptron

Although the initial success of perceptrons, they increasily lost attractiveness after Marvin Minsky and Seymour Papert (MINSKY; PAPERT, 1969) showed that a single layer perceptron cannot learn the "XOR" logical function. Indeed, they proved that perceptrons could only deal with linearly separable problems.

Algorithm 3 Pocket algorithm with ratchet

Require: A group of training examples and corresponding classifications.
Ensure: A vector of weights w that correctly classifies all training examples if they are linearly separable.

```
w ← 0
w^pocket ← 0
while w does not correctly classifies all examples do
  x ← training example
  c ← correct classification
  if (w · x >= 0 and c = +1) or (w · x < 0 and c = −1) then
    if w correctly classifies more training examples than w^pocket then
      w^pocket ← w
    end if
  else
    w ← w + c · x
  end if
end while
```

This dramatically slowed down the work with perceptrons and other connectionist models. In fact, the research of AI (*artificial intelligence*) in the 1970s and early 1980s was dominated by symbolic models. Stephen Gallant (GALLANT, 1994) reminds a comment (from an anonymous referee who was analyzing a paper about neural networks on March 1985) that captures the general feeling about then: "perceptron learning has quite correctly been discarded as a technique worthy of study".

Despite of that, some authors achieved significant advances during these "dark ages", like Grossberg, Anderson, Kohonen, Fukushima and Hopfield. The resumption of neural networks was incrementally done by several works that could deal with non linearity. For example, in 1973, Stephen Grossberg proposed new arranges of perceptrons that could solve the "XOR" problem and other non linear problems (GROSSBERG, 1937).

However it was in the mid-1980s that neural networks "reborn" when Hinton, Sejnowski and Ackley (HINTON; SEJNOWSKI; ACKLEY, 1984) came up with the backpropagation algorithm. Actually, they "rediscovered" this algorithm, since it was firstly described in 1969 by Arthur E. Bryson and Yu-Chi Ho (BRYSON-JR; HO, 1969). Curiously, the backpropagation algorithm was also "reinvented" by P.J. Werbos on 1974 (WERBOS, 1974) and D.B. Parker on 1982 (PARKER, 1982).

Initially, the main problem of the backpropagation learning was the speed. This was solved when Rumelhart, Hinton and Williams (RUMELHART; HINTON; WILLIAMS, 1986) popularized a much faster algorithm, that was utilized by several researchers in many problems. Since then, multilayers perceptrons trained with backpropagation algo-

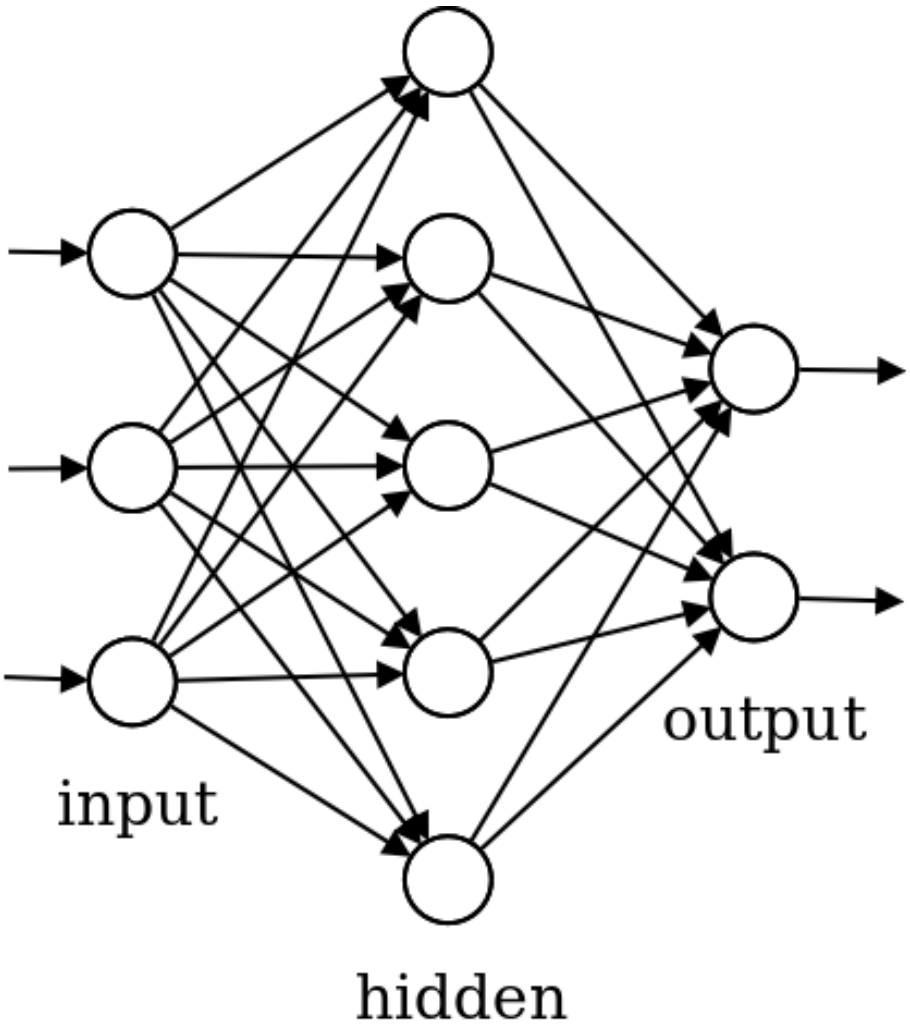

Figure 2.8: A typical multilayer perceptron.

rithm became the symbol of the "ressurection" of neural networks.

A MLP (*multilayer perceptron*) is a feedforward neural network that, differently from a single layer perceptron, has two or more layers with nonlinear activation functions. Because of that, it is able to map nonlinear models, which means that is able to solve problems that linear perceptrons cannot.

It consists of one input and one output layer (as single layer perceptrons) plus one or more "hidden" layers. Each layer is fully connected to the next, as exemplified in figure 2.8. Every connection between neurons is balanced with a weight w_{ij}. The number of input neurons is defined by the number of parameters whereas the output neurons quantity is the same of the mapped classes.

The activation function defines the output of a neuron from the sum of inputs multiplied by their weights, as previously shown in the figure 2.2. Unlike the linear perceptrons

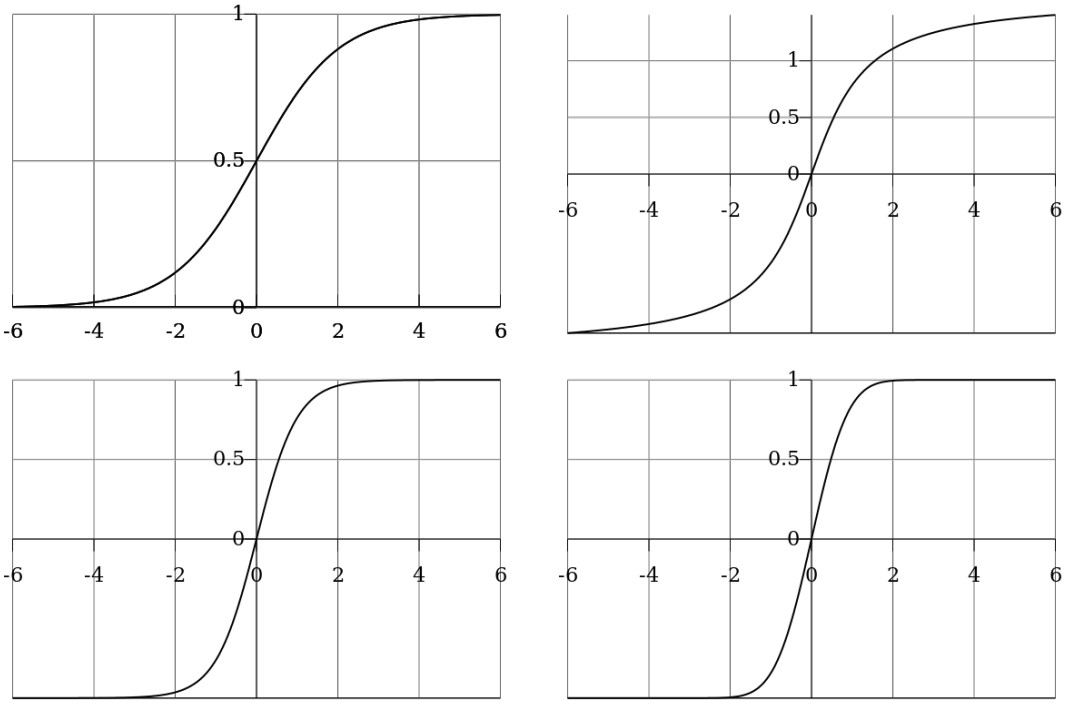

Figure 2.9: Some sigmoid functions.

that were described earlier, a MLP uses nonlinear activation functions, mainly sigmoids. A sigmoid function is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior (HAYKIN, 1994). There are several sigmoid functions: logistic, arctangent, hyperbolic tangent, and others. Some of them are displayed in the figure 2.9. They all share three important characteristics:

- They are real-valued, which means that they assume a continuous range of values from -1 to 1 or 0 to 1 (depending of the adopted mathematical model).
- They are differentiable and the first derivative must be non-negative or non-positive.
- They have one inflection point, that can be seen in the "s-shaped" graph.

2.5 Backpropagation Learning Algorithm

The backpropagation learning algorithm is the most used method of training MLPs. It is composed by two phases. In the first phase, the training examples are presented to the network that tries to predict results. In the second phase, these predicted results are compared to the correct values and an error value is calculated. This error is propagated

backwards through the network, from the output neurons to the inner ones. The weights are adjusted according to the propagated error. The two phases are iteratively repeated until the network correctly matches all training examples or another stopping criterion is satisfied.

Jeff Heaton (HEATON, 2005) describes the algorithm as follows:

1. Initialization: Randomly set the weights (including bias) between -1 and 1;

2. Training examples are presented to the network:

   a) On-line training: Execute steps 3 and 4 for each training example of the data set.

   b) Batch training: Execute steps 3 and 4 for each training epoch.

3. Propagation: After presenting the training example $t = x(n), d(n)$, where $x(n)$ is the network input and $d(n)$ is the desired network output, calculate the activation value v_j and the outputs of the neurons, as follows:

   a) $v_j =_{i=1}^{m} \sum w_{ji} x_i + b$, to calculate the activation value;

   b) $f(v) = \frac{1}{1+e^{-av}}$, to calculate the output y of the neuron k, using the sigmoid function or another, if necessary.

   c) Use the output units of a layer as inputs to the next, until the last layer. The output of the units of the last layer is the network response.

4. Calculate the error signal: Considering that $y_j = O_j(n)$ is the network response, calculate the error signal as follows:

   a) $e_j(n) = d_j(n) - O_j(n)$, where $d_j(n)$ is the desired output on the n_{th} iteration.

   b) This error signal is used to compute the values of the errors of the previous layers and make the necessary corrections of synaptic weights.

5. Retropropagation: Calculate the local errors δ for each unit on the network. The local gradient is defined as follows:

   a) $\delta_j(n) = e_j(n) O_j(n)(1 - O_j(n))$, for the output units.

   b) $\delta_j(n) = O_j(n)(1 - O_j(n)) \sum \delta_k w_{jk}$, for the other units. Where:

   i) $O_j(1 - O_j)$ is the activation function derived according the argument, i.e., the activation value;

   ii) δ_k is the error of the previous layer units that are connected to the unit j;

iii) w_{jk} are the weights of the connections with the previous layer.

c) After the errors calculation, adjust the weights of the connections following the generalized delta rule:

i) $\Delta w_{kj}(n+1) = \alpha w_{kj}(n) + \eta \delta_j y_j$ to adjust the weights.

d) Do:

i) $w(n+1) = w(n) + \Delta w_{kj}(n+1)$, where: α is the momentun constant (when $\alpha = 0$, this function acts as a normal delta rule); η is the learning rate; δ_j is the error of this unit; $y_j j$ is the output of the j_{th} unit.

6. Iteration: Repeat the steps 3, 4 and 5 (propagation, error calculation and retropropagation), presenting new training examples until the stop conditions are satisfied:

   a) The network error is low and is not changing during the training;

   b) The maximum number of iterations was reached.

The backpropagation learning algorithm is given as pseudocode in algorithm 4:

Algorithm 4 Backpropagation learning algorithm

Require: A group of training examples and corresponding classifications.
Ensure: A vector of weights w that correctly classifies all training examples.

```
w ← randomvalues
while w does not correctly classifies all examples or a stopping criterion is satisfied do
  for all training example e in the set do
    o ← output(w, e)
    c ← correct ouput from the training set
    error ← c − o
    w ← w + calculate_delta(error)
  end for
end while
```

2.6 Chapter Review

This chapter presented the foundations of neural networks. Initially, it presented the biological inspiration and the way the first neurons were implemented. Then, the chapter focused on the perceptron and its most famous evolution, the MLP trained with backpropagation algorithm.

The Pocket learning algorithm with ratchet (developed for training perceptrons) will be further utilized in this work to train the neurons in a single-cell learning approach.

Also, the backpropagation learning algorithm will train some MLPs that will be utilized as control group during the experiments. Other kinds of neural networks were intentionally left away in this chapter since they are not utilized in the project.

The next chapter will present the constructive algorithms as an attractive option to the MLP/backpropagation approach. This is why these algorithms were excluded from the foundations chapter.

Chapter 3

Constructive Neural Networks

One of the main limitations of MLPs trained with backpropagation is the search area. This learning algorithm only searches for an appropriate set of weights in an *a priori* fixed topology. In other words, a specialist must define the network architecture, including the number of hidden layers and neurons. Because there is no definitive method, nobody can assure that a chosen topology is optimal. Extra neurons can overfit data while scarcity may compromise the model accuracy. To overcome this shortage, most researchers adopt the trial-and-error strategy.

However, testing many options does not guarantee an optimum solution. Because of that, constructive algorithms are so appealing. They iteratively create the topology, so the generated networks tend to use less neurons as possible. Parekh, Yang and Honavar (PAREKH; YANG; HONAVAR, 2000) listed some of the main attractives of this approach:

- Flexibility of exploring the space of neural network topologies;
- Potential for matching the intrinsic complexity of the learning task;
- Estimation of the case complexity of the learning task;
- Tradeoffs among performance measures;
- Incorporation of prior knowledge;
- Lifelong learning.

There are several constructive algorithms in the literature. In this chapter, some of them are described:

- Tower;
- MTower;
- Pyramid;
- MPyramid;
- Cascade-correlation;
- Tiling;
- MTiling-real.
- Upstart;
- Easy learning problems.

3.1 Tower Algorithm

The Tower algorithm was independently discovered by Stephen Gallant (GALLANT, 1985) (GALLANT, 1990), Jean-Pierre Nadal (NADAL, 1989) and Marcus Roland Frean (FREAN, 1990). It employs single-cell learning to build a tower of cells, where each cell sees the original inputs and the single cell immediately below it (GALLANT, 1990). An example of network generated with this algorithm can be seen in figure 3.1. Each cell is trained separately using the Pocket algorithm with ratchet. After the training, the weights of a neuron do not change. Then, its output becomes one of the inputs of the next neuron. It is interesting to observe that the same algorithm can be applied with linear machines instead of single neurons.

The algorithm was proposed by Gallant as follows (GALLANT, 1994):

1. Use the Pocket algorithm to generate a single-cell model and freeze these weights.
2. Create a new cell that sees the p inputs to the network and the activation from the cell that was most recently trained. Run the Pocket algorithm with ratchet to train the $p + 2$ weights (including bias) for this cell.
3. If the network with this added cell gives improved performance, then freeze its coefficients and go to step 2; otherwise remove this last added cell and output that network.

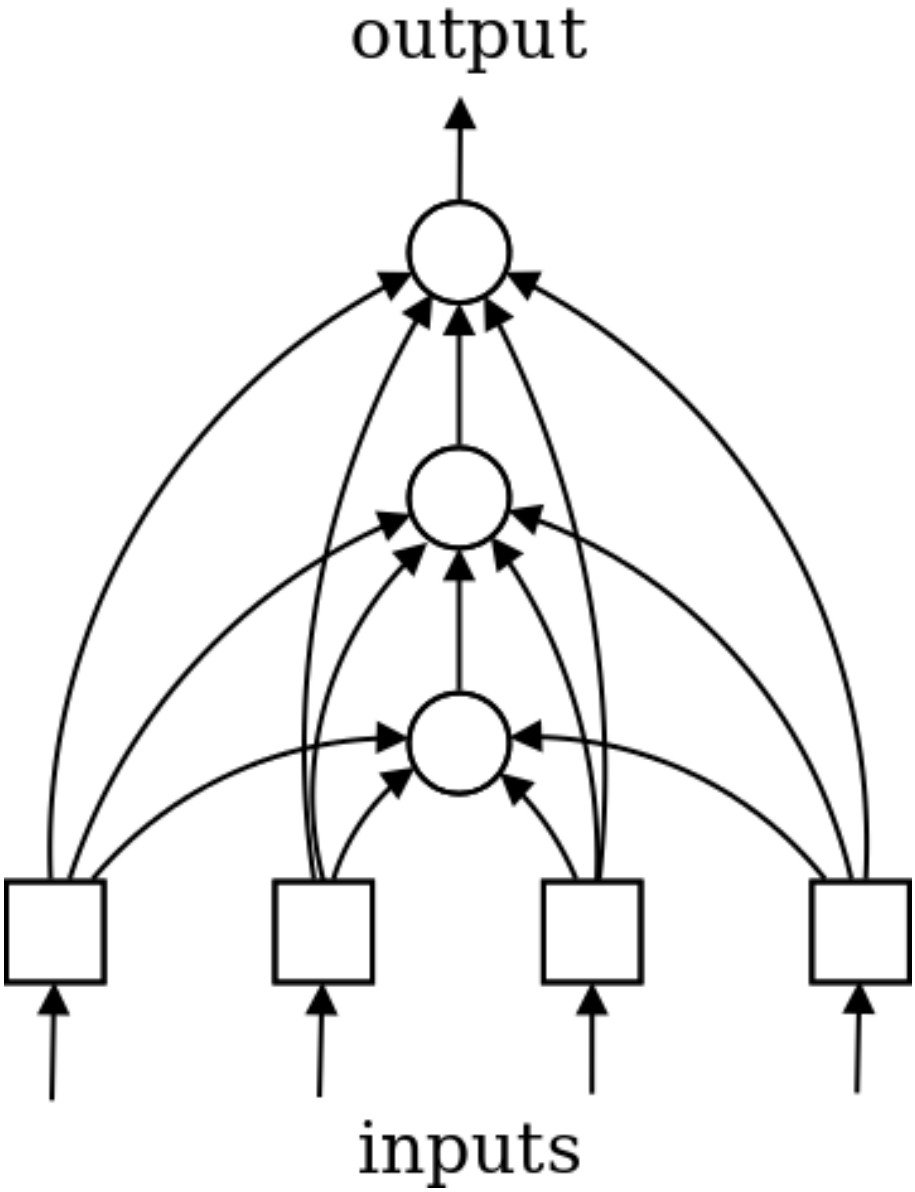

Figure 3.1: A network generated with the Tower algorithm.

The Tower algorithm tends to converge, provided there is a training set with noncontradictory examples and enough iterations. Its proof of convergence is based on the fact that Pocket algorithm also converges. Basically, if a n-cell model cannot classify a new training example, one can separately train a new cell with Pocket algorithm and insert it into the model. This way, it will have a $n + 1$ cell model that correctly classifies this example and leaves others unchanged.

Nevertheless, the practice shows that the algorithm may not deal well with large amounts of data. It mainly happens because not enough iterations can be made to ensure an optimal solution at each step (GALLANT, 1990).

3.2 MTower Algorithm

The Tower algorithm is only suitable for two-class problems. However, Parekh, Yang and Honavar (PAREKH; YANG; HONAVAR, 1997) proposed an extended version called MTower that can deal with multiclass problems. While the Tower algorithm adds a single neuron per layer, its multiclass version adds M neurons per layer, where M is the number of classes (NICOLETTI; BERTINI-JR., 2007). Figure 3.2 shows an example of a network that was generated by MTower algorithm.

Parekh, Yang and Honavar proposed the MTower algorithm as follows (PAREKH; YANG; HONAVAR, 1997):

1. Set the current output layer index $L = 0$.

2. Repeat the following steps until the desired training accuracy is achieved or the maximum number of hidden layers is exceeded:

   (a) $L = L + 1$. Add M output neurons to the network layer at layer L. This forms the new output layer of the network. Connect each neuron in layer L to the $N + 1$ input neurons and to each neuron in the preceding layer, $L - 1$, if one exists.

   (b) Train the weights associated with neurons in layer L (the rest of the weights in the network are left unchanged).

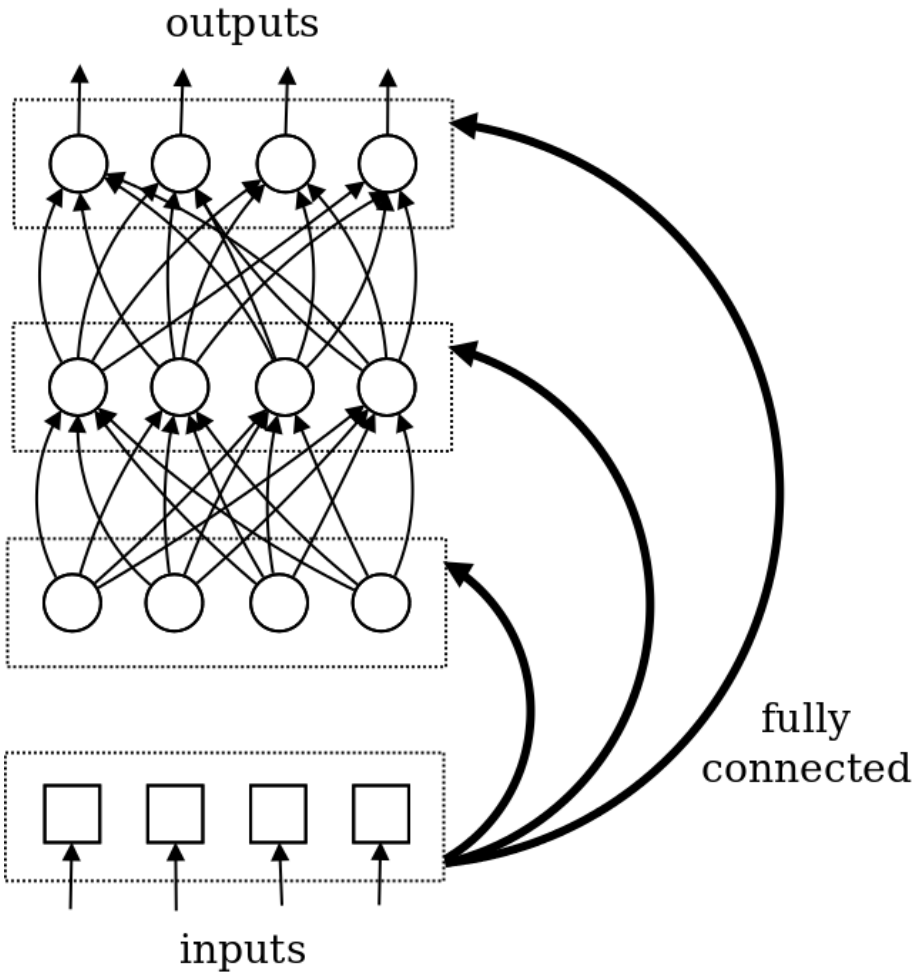

Figure 3.2: A network generated with the MTower algorithm.

3.3 Pyramid Algorithm

As the Tower algorithm, the Pyramid algorithm was proposed by Stephen Gallant (GALLANT, 1985). They are very similar and their only difference lies in the connections between hidden neurons. In Pyramid algorithm, each new neuron is connected to all other previous neurons, not just the neuron below. Thus the algorithm generates networks like the one that is displayed in the figure 3.3.

The Pyramid algorithm is:

1. Use the Pocket algorithm to generate a single-cell model and freeze these weights.
2. Create a new cell that sees the p inputs to the network and the activations from all previously trained cells. Run the Pocket algorithm with ratchet to train the weights for this cell.
3. If the network with this added cell gives improved performance, then freeze its coefficients and go to step 2; otherwise remove this last added cell and output that network.

3.4 MPyramid Algorithm

As the MTower, the MPyramid algorithm is the extended version of the Pyramid algorithm for multiclass problems. Each newly added layer of M neurons receives inputs from the $N + 1$ input neurons and the outputs of each neuron in each of the previously added layers (PAREKH; YANG; HONAVAR, 1997). The algorithm produces a net as the one displayed in figure 3.4.

The MPyramid algorithm is proposed by Parekh, Yang and Honavar (PAREKH; YANG; HONAVAR, 1997) as follows:

1. Set the current output layer index $L = 0$.
2. Repeat the following steps until the desired training accuracy is achieved or the maximum number of hidden layers allowed exceeded:
   (a) $L = L + 1$. Add M neurons to the network at layer L. This forms the new output layer of the network. Connect each neuron in the layer L to the $N + 1$ input neurons and each neuron in each of the previous layers if they exist.

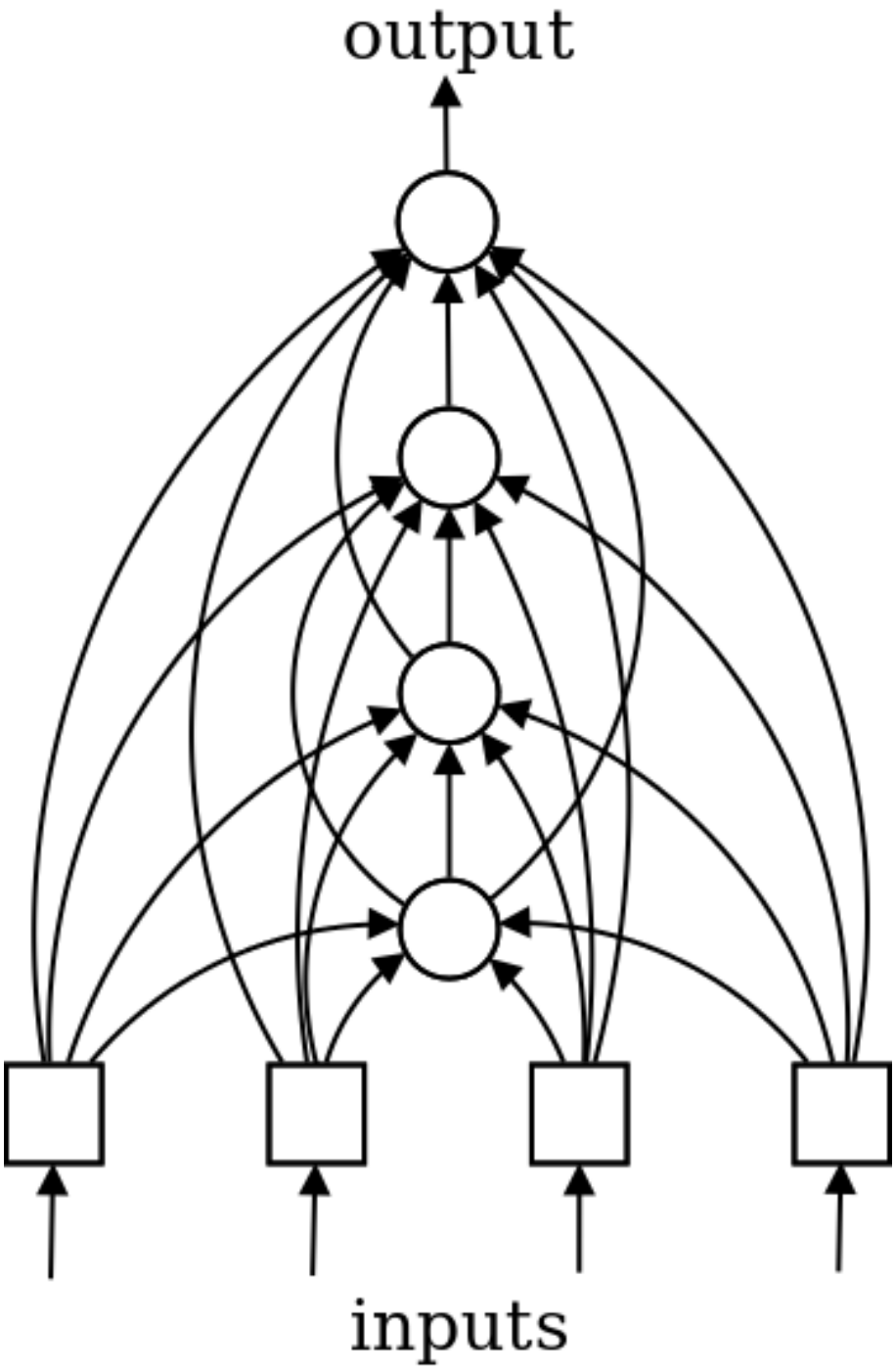

Figure 3.3: A network generated with the Pyramid algorithm.

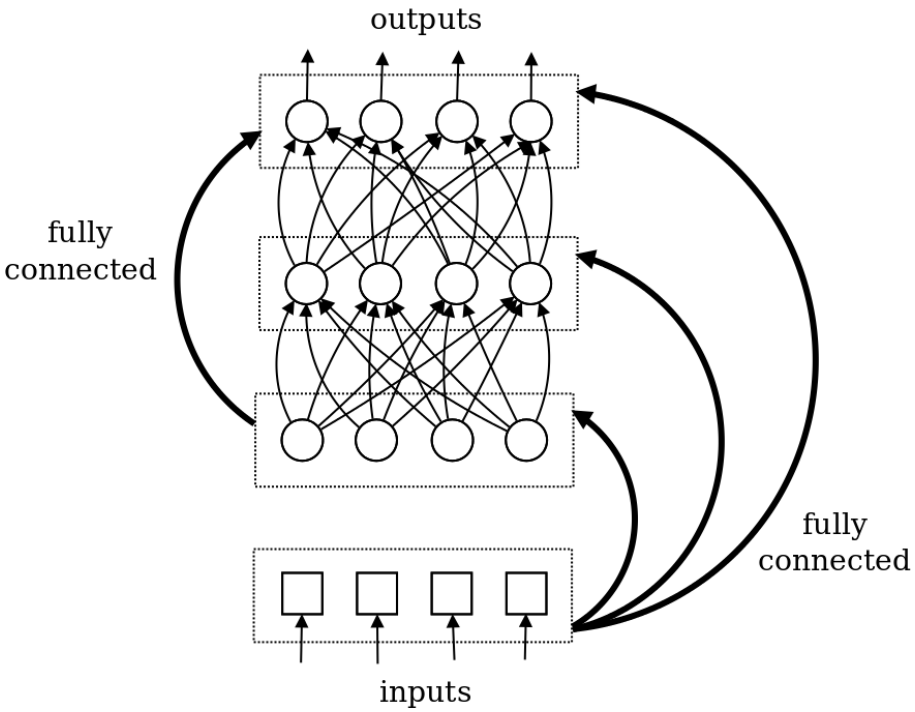

Figure 3.4: A network generated with the MPyramid algorithm.

(b) Train the weights associated with the neurons in layer L (the rest of the weights in the network are left unchanged).

3.5 Cascade-Correlation Algorithm

Scott Fahlman and Christina Lebiere proposed the Cascade-Correlation algorithm in 1990 during a research sponsored by the National Science Foundation (USA) and the Defense Advanced Research Projects Agency (USA) (FAHLMAN; LEBIERE, 1990). The main objective of the study was to find out why the backpropagation learning algorithm was so slow and then propose a faster alternative. The authors suggested that the speed problems of the backpropagation algorithm where mainly caused by two factors:

- **The step-size problem:** it is hard to determine the size of the update steps, during the learning. Small steps implicate on slower learning while larger steps can overpass the best solution.
- **The moving target problem:** each hidden neuron is trying to evolve as part of the classifier, but its connected neurons are evolving too. This constant and parallel mutation makes harder the training of hidden neurons.

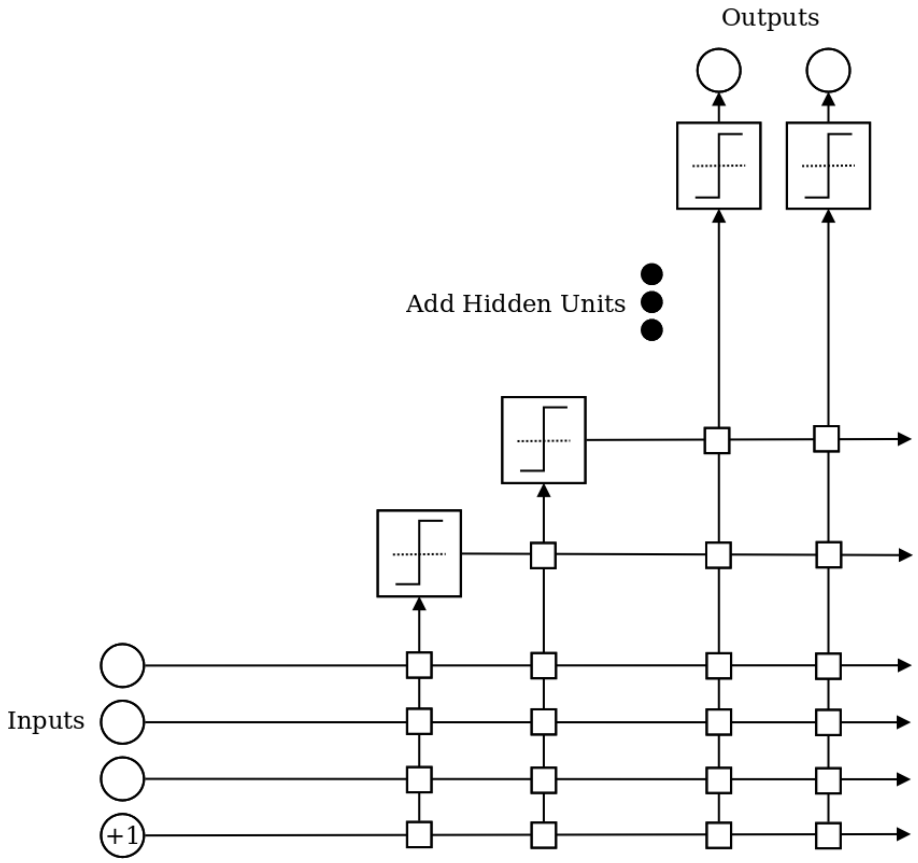

Figure 3.5: A network generated with the Cascade-Correlation algorithm.

The utilized network model is similar to the pyramid's one, as seen in figure 3.5. The key difference is that, in the Cascade-Correlation method, new hidden layers are added instead of output layers. In other words, the new neurons are always between the input neurons and the last added neurons.

3.6 Tiling Algorithm

In 1989, Marc Mézard and Jean-Pierre Nadal proposed the Tiling algorithm, a constructive method that builds layered networks (MÉZARD; NADAL, 1989). The neurons in a layer only see the activations of the neurons at immediately below layer.

This algorithm generates a network like exemplified in figure 3.6 and was described by Stephen Galland (GALLANT, 1994) as follows:

1. Set layer $L = 2$ (layer 1 is the input layer).

2. Use the pocket algorithm with ratchet to create the master cell for layer L using all

activations from layer $L-1$.

3. If the master cell for layer L correctly classifies all training examples, then quit.
4. Otherwise,continue to add ancillary cells ("tile") until layer L becomes faithful:
   (a) Find a maximum-sized subset of training examples with *more than one classification* that produces the *same activations* for all cells in layer L.
   (b) Use the pocket algorithm with ratchet to train a new ancillary cell for layer L using only the subset of training examples from step 4a.
5. Layer L is now faithful. Set $L = L + 1$ and go to step 2.

3.7 MTiling-Real Algorithm

An extended version of Tiling algorithm was proposed to learn *real* to *M-ary mappings*, which are problems that involve real inputs and multiple classes. It is called MTiling-Real and was proposed in 2000 by Rajesh Parekh, Jihoon Yang and Vasant Honavar (PAREKH; YANG; HONAVAR, 2000).

Unlike the Tiling algorithm that adds one master neuron per layer, it adds M master neurons where M is the number of classes. This way, it generates nets like the one displayed in figure 3.7.

3.8 Upstart Algorithm

Marcus Roland Frean proposed the Upstart algorithm on 1990, which uses a simple recursive rule to build the net's structure by adding units as they are needed (FREAN, 1990). The algorithm only works with binary inputs and activations:

1. Train a neuron Z with the perceptron algorithm to correctly classify the examples. The outputs should be +1 or -1.
2. Fix the weights of Z.
3. If the neuron Z wrongly classifies a +1:
   (a) Build a neuron X that will be an input from Z.

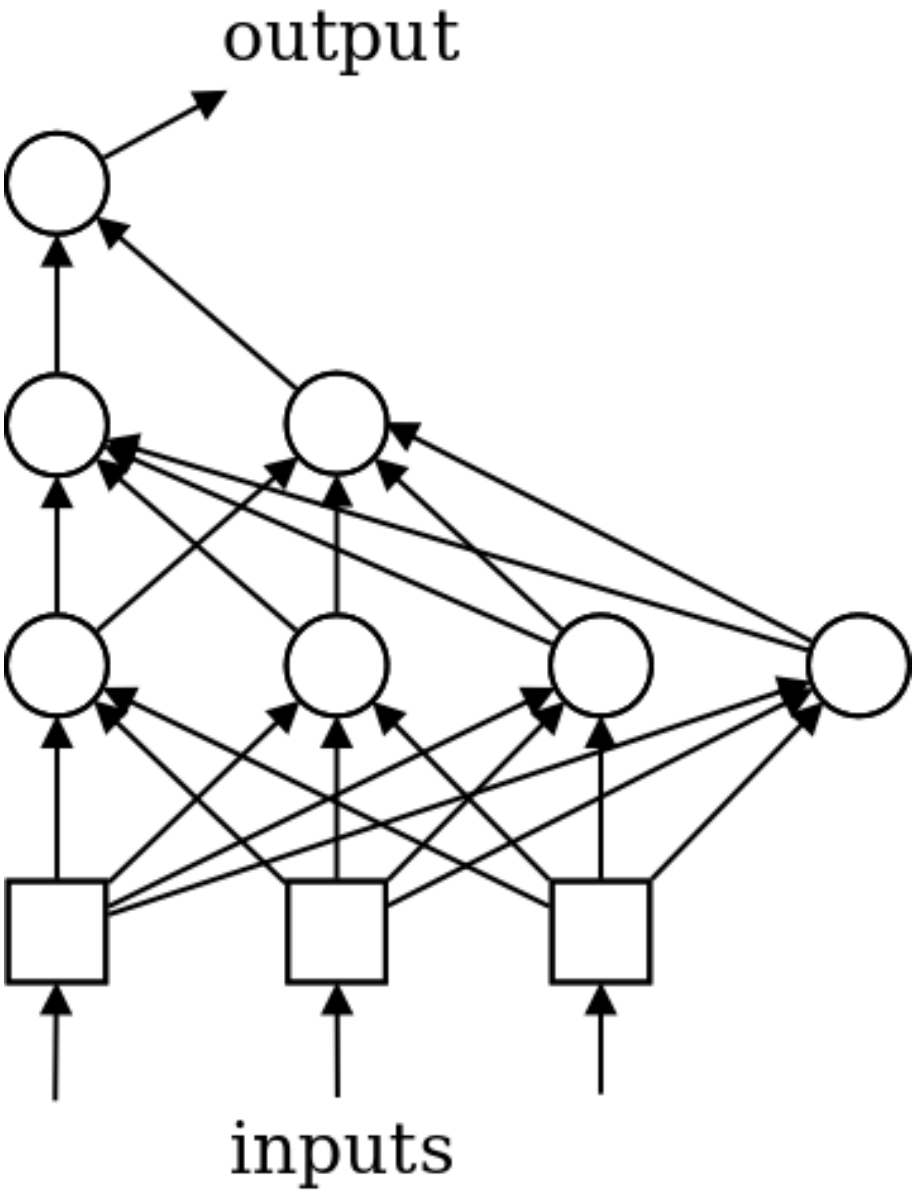

Figure 3.6: A network generated with the Tiling algorithm.

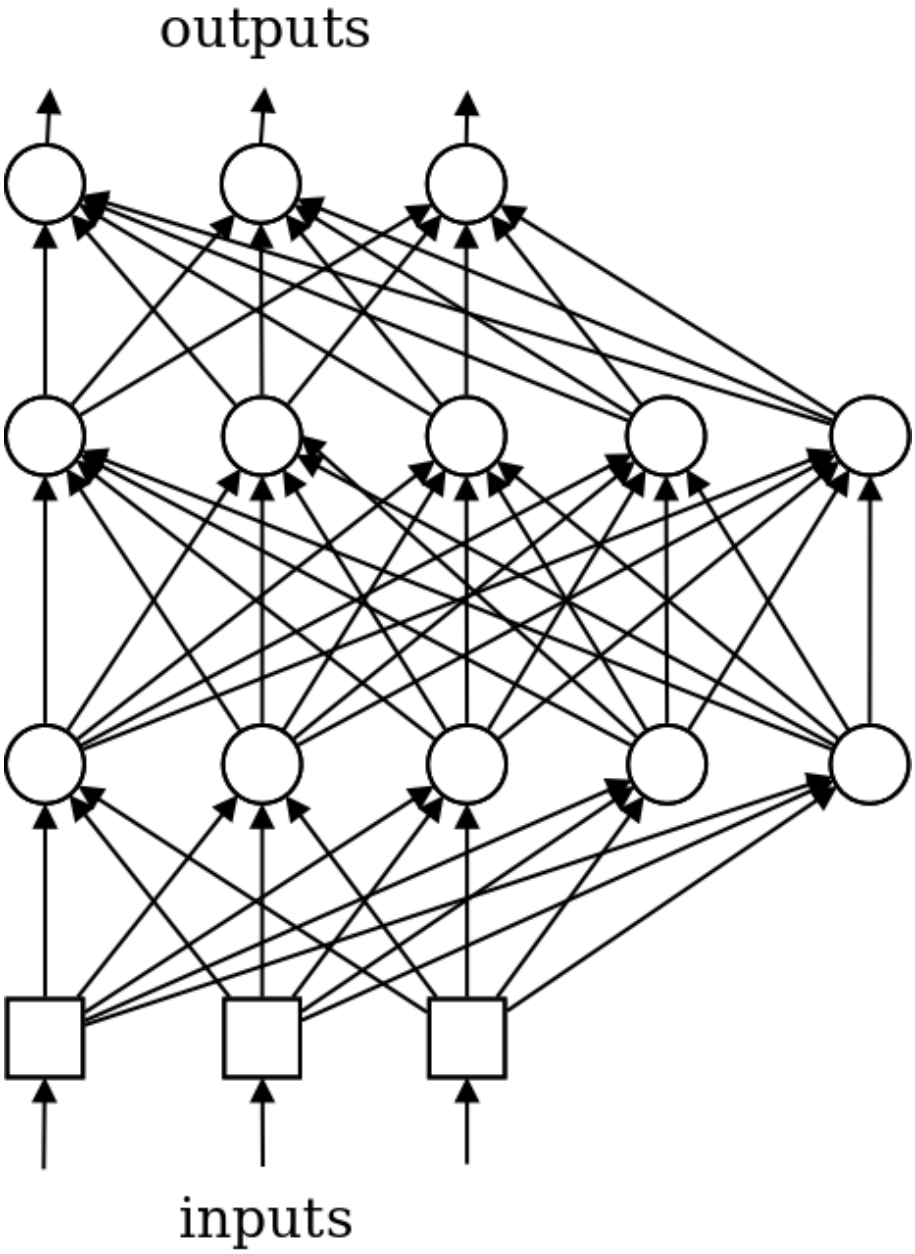

Figure 3.7: A network generated with the MTiling-real algorithm.

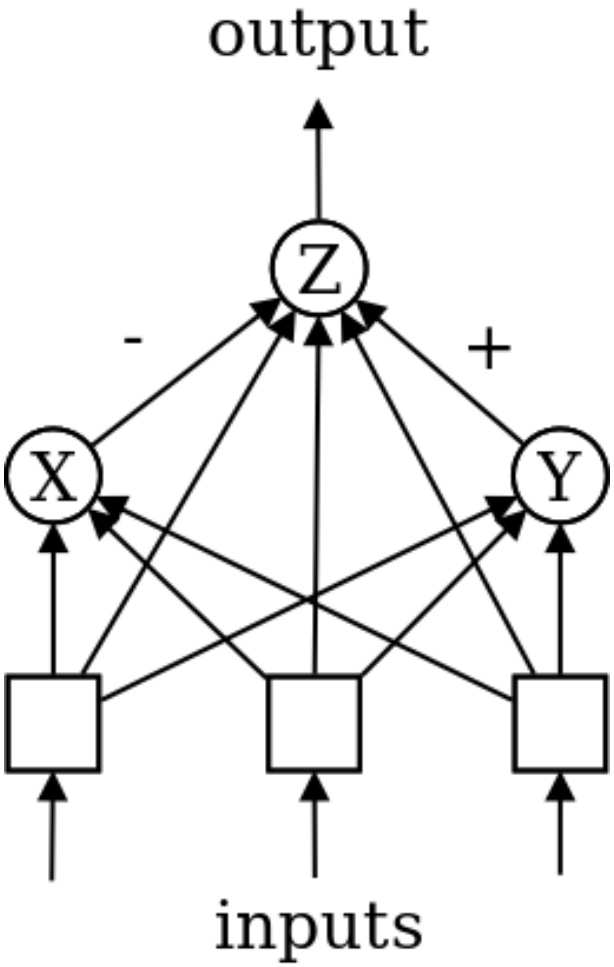

Figure 3.8: A network generated with the Upstart algorithm.

(b) Train the neuron X to correctly classifies the wrong examples.

4. If the neuron Z wrongly classifies a -1:

   (a) Build a neuron Y that will be an input of Z.

   (b) Train the neuron Y to correctly classifies the wrong examples.

5. Repeat the steps for the neurons X and Y recursively until the network correctly classifies all examples.

The Upstart algorithm generates a network like the one in the figure 3.8.

3.9 Easy Learning Problems

Stephen Gallant describes a very particular group of constructive algorithms where there is no freedom to choose the network topology (GALLANT, 1994). He called them "easy learning problems" and listed their two main characteristics:

- The network topology is previously defined.
- The training examples specify, besides the input and output values, the correct activations for all intermediate neurons.
- Each neuron can be separately trained (single-cell learning).

As an example, Gallant describes an easy learning version of XOR problem. As cited before, the XOR logical operation is probably the most classic example of linearly inseparable pattern. It can not be mapped by a single neuron. However, the XOR function can be decomposed as follows:

$p \oplus q$
$(p \wedge \neg q) \vee (\neg p \wedge q)$
$(p \vee q) \wedge (\neg p \vee \neg q)$
$(p \vee q) \wedge \neg(p \wedge q)$

This decomposition is interesting because $(p \vee q)$, $(p \wedge q)$ and $(p \wedge \neg q)$ are linearly separable patterns. This way, three neurons can be separately trained to solve the XOR logical problem, as described in figure 3.9.

Nevertheless, some problems cannot be decomposed on linearly separable subproblems. In this case, Gallant considers the use of other constructive methods to solve the remaining nonseparable subproblems. He calls them "expandable network problems".

Besides the fact that easy-learning algorithms are theoretically an attractive way to solve problems, Gallant reinforces the existence of a very significant problem. In many situations it is very hard to find the correct activations of the intermediate neurons. In fact, Blum and Rivest proved that it is a NP-complete problem (BLUM; RIVEST, 1992). This way, easy-learning algorithms were relegated to the rare cases when the intermediate activations are previously known.

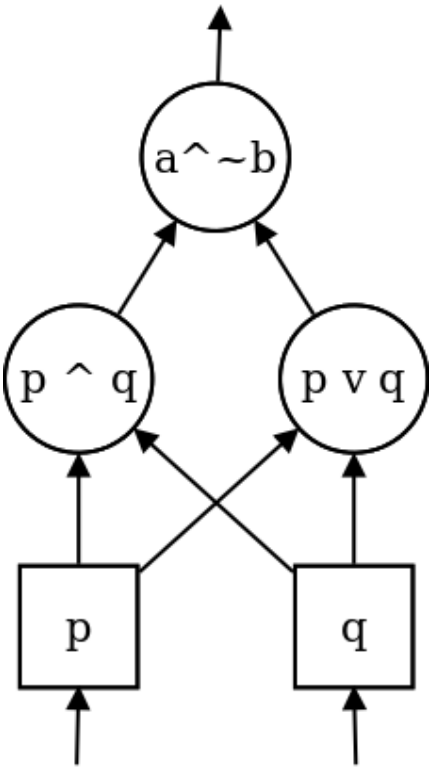

Figure 3.9: An easy learning version of XOR problem

3.10 Chapter Review

The most traditional constructive algorithms were presented in this chapter. They offer an attractive option instead of the MLP/backpropagation solution. In common, all these algorithms are able to build the network topology together with the weights learning. The older constructive algorithms were only able to deal with two-classes problems. However, authors had improved them to solve n-classes applications. For example, the MTower is the extended version of the classic Tower algorithm.

Additionaly, this chapter presented the easy learning problems, a very interesting type of context where the training examples include the correct activations for all intermediate neurons. This way, it is possible to separately train each neuron (single-cell learning). Unfortunately, there are no many data sets that include so much information. Another weak point of these special algorithms is that they demand a previously defined topology, differently from the majority of constructive algorithms.

This work proposes a new kind of constructive algorithm that is able to produce the network topology and also execute the single-cell learning approach. For this, it utilizes concepts that are utilized in decision trees and evolutionary algorithms. So, before

presenting this new algorithm, the next chapter will describe the hybrid methods that join neural networks, decision trees and evolutionary algorithms. Many of these hybrid algorithms could be classified as constructive algorithms since they build the network architecture.

Chapter 4

Hybrid Models

The present work proposes a new constructive learning algorithm for neural networks inspired on decision trees and evolutionary algorithms. Mixing these three concepts, however, is not an original idea. Indeed, many authors investigated several methods that combined some of these concepts. To understand the originallity of this research, it is necessary to analyze these previous works. So, this chapter lists the most relevant authors and their theories about hybrid models with neural networks, decision trees and/or evolutionary algorithms.

Qiangfu Zhao detected five main types of integration between neural networks and decision trees (ZHAO, 2001):

- Derivation of a neural network from a previously induced decision tree;
- Derivation of a decision tree from a previously trained neural network;
- Design of a tree structured neural network using genetic programming;
- Derivation of a modular neural network by softening the nodes of a decision tree;
- Design of decision trees with embedded neural networks.

4.1 Derivation of a neural network from a previously induced decision tree

In 1994, Arunava Banerjee proposed a method that initializes neural networks from decision trees (BANERJEE, 1994). The described algorithm proved to be faster than

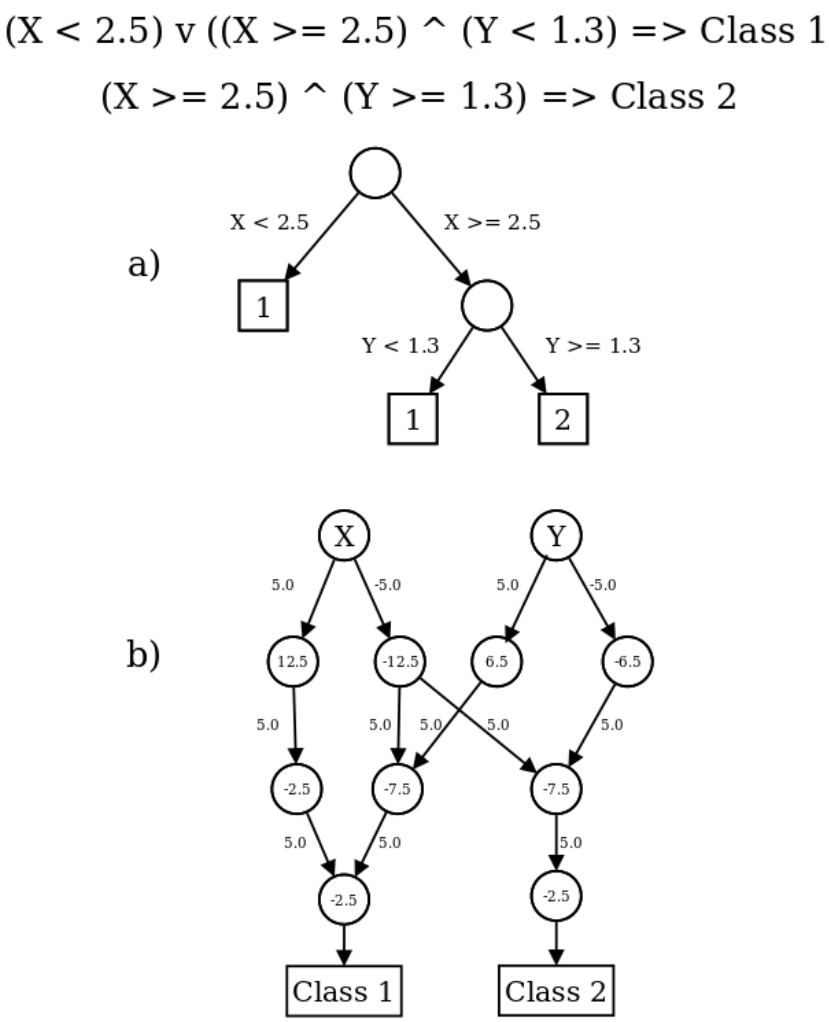

Figure 4.1: (a) Original decision tree. (b) Converted neural network using Banerjee's method.

backpropagation and matched its accuracy. It was done in three steps: construction of a decision tree from the data set, conversion into an equivalent neural network and network tuning using the same data set. The converted networks are 4-layers MLPs, with 1 input layer, 2 hidden layers and 1 output layer. The hidden layers are called "literal" and "conjunction" layers. On the other hand, the output layer is called "disjunction" layer. Figure 4.1 shows an original decision tree and the converted MLP.

Banerjee presents his algorithm as follows (BANERJEE, 1994):

1. Initialize parameters σ and β to 5.0 and 0.025 respectively.

2. Run C4.5 on the training dataset to generate a decision tree.

3. Traverse the decision tree to create a disjunctive normal form formula for each class.

4. Eliminate all redundant literals from each disjunct.

5. For each distinct literal of the form $(attrib) > (value)$, create a hidden unit in the literal layer with a bias of $-\sigma * (value)$. Connect it to the input unit corresponding to $(attrib)$ with a weight of σ. Connect it to all other input units with weights $+\beta$ or $-\beta$ with equal probabilities.

6. For each literal of the form $(attrib) < (value)$, repeat step 5 with the signs for the bias and weights inverted.

7. For each disjunct in a class, create a new hidden unit in the conjuction layer. Connect it to all relevant hidden units in the literal layer with weight σ. Connect it to the rest of the hidden units in the literal layer with weights $+\beta$ or $-\beta$ with equal probabilities. Set the bias to $-\sigma * (2n - 1)/2$, where n stands for the number of relevant hidden units in the literal layer. (In effect, each node represents an AND.)

8. For each class, create an output unit and connect it to the relevant hidden units in the conjuction layer with weights σ. Connect it to the rest of the hidden units in the conjuction layer with weights $+\beta$ or $-\beta$ with equal probabilities. Set the bias to $-\sigma * 1/2$. (Each node is effectively an OR.)

The main motivations about deriving a neural network from a decision tree are:

- Obtain a more accurate classifier;
- Train faster the neural network.

Nevertheless, for a long time, no author had proved that these statements are true for every case. To answer this, Nathan Rountree investigated a new initialisation method of neural networks from previously induced decision trees (ROUNTREE, 2006). Similarly to Bannerje's method, his algorithm always creates 4-layers networks independently of the size of the original tree, as shown in figure 4.2.

After testing the technique against six different databases, the author concluded that the statements are true, at least for several situations. Indeed, his work brings three main contributions:

- Often, there is a MLP that is more accurate than the original decision tree.
- His algorithm creates MLPs with an efficient quantity of neurons.

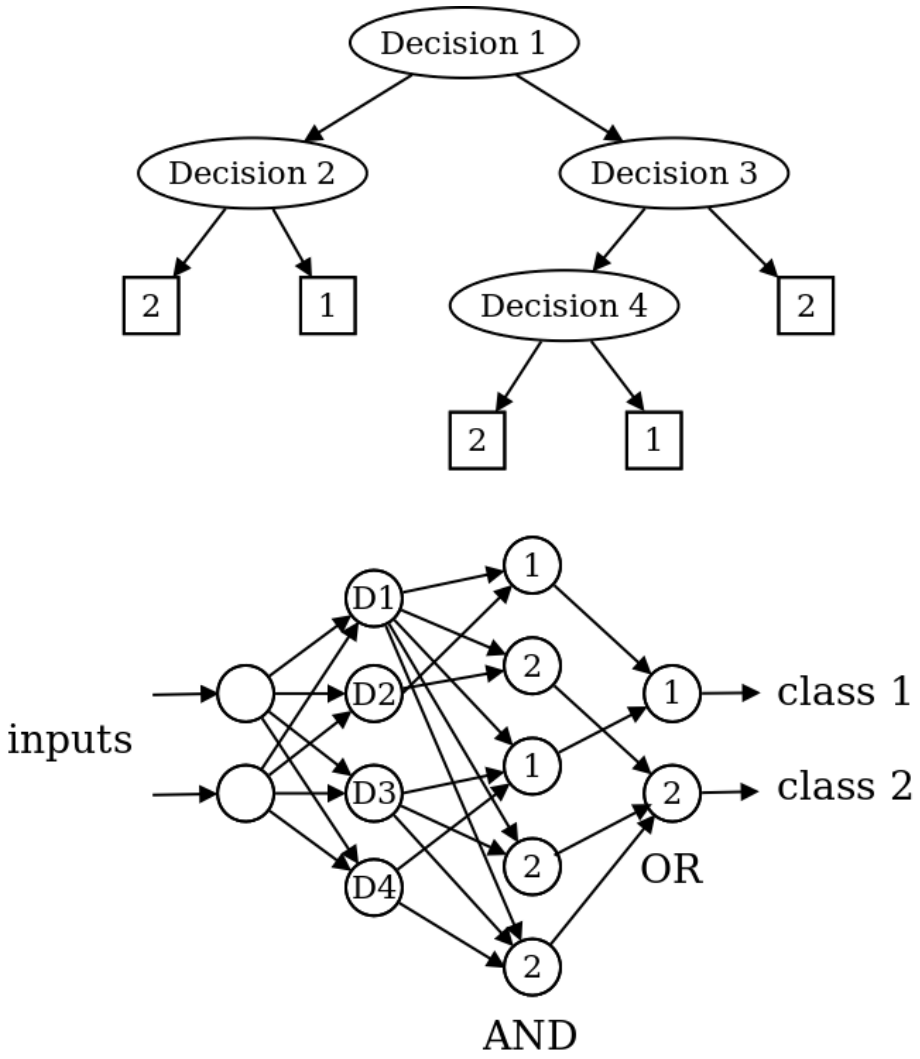

Figure 4.2: Initialisation of a neural network from a decision tree using Nathan Rountree's approach.

- He tried to be "fair" during the experiments, which means that the MLPs were compared to optimized decision trees.

An adapted version of the Banerjee's method was utilized on 2004 by Nerijus Remeikis, Ignas Skucas and Vida Melninkaite to categorize texts (REMEIKIS; SKUcAS; MELNINKAITė, 2004). One of the harder problems involving text categorization applications is that they generate too large trees, because of the large amount of data. The authors found out that changes in the certainty factor of the pruning algorithm significantly affect the tree size. The technique was tested on the Reuters-21578 corpus, one of the standard benchmarks for text categorization tasks. The adapted method achieved better results when compared to original Banerjee's method, MLPs and decision trees.

4.2 Derivation of a decision tree from a previously trained neural network

Neural networks are famous to be superior in performance to many other AI techniques. Nevertheless, they are also famous as "black-box" methods, because of their poor interpretability. This weakness stimulated several authors to find ways to extract more understandable models from them, such as the decision trees. Indeed, the extraction of decision trees from previously trained neural networks is probably the most common integration of these two techniques.

Darren Dancey, Dave McLean and Zuhair Bandar relate two main approaches for extracting decision trees from ANNs (DANCEY; MCLEAN; BANDAR, 2004): decompositional and pedagogical. Decompositional methods, like the KT algorithm (FU, 1991), analyze individually each connection and its respective weight. On the other hand, the pedagogical methods utilize the trained ANN to generate artificial training examples for the decision tree learning algorithm.

The pedagogical approach can usually be illustrated as follows:

1. Learn an ANN using the training examples from the data set.
2. Use the ANN to generate artificial training examples and increase the size of the data set.
3. Learn the decision tree using the new data set.

In 1996, the PhD thesis of Mark W. Craven presented a pedagogical algorithm called TREPAN (CRAVEN, 1996). It utilizes the ANN as an "oracle" that can "answer" questions made by the decision tree learning algorithm. The TREPAN algorithm constructs the decision tree by recursively partitioning the instances space, like other conventional algorithms (e.g., CART and C4.5). However, it differs from other techniques in many ways:

- It constructs the tree in a "best-first" manner, while other algorithms utilize the "depth-first" principle.
- When there is a lack of training examples, the algorithm can produce new instances and ask the "oracle" to classify them. This resource is very interesting because the algorithm can make "better" choices having more training examples.

- It uses "m-of-n" expressions for the splitting tests whereas other algorithms prefer single-feature tests.
- It uses global stopping criteria in addition to local stopping criteria.
- Its prunning method detects the subtrees that predict the same class and tries to merge them into a single leaf.

Inspired on the TREPAN algorithm, Vijaya Hari improved the efficiency of CART algorithm by using an ANN to generate extra training examples (HARI, 2009). Basically, the method provides an "oracle" to the CART algorithm. This way, it is possible to generate extra training instances (and classify them with the ANN) when there is no enough examples during the creation of a tree node.

The TREPAN also inspired the creation of ExTree algorithm, which was proposed by Darren Dancey, Dave McLean and Zuhair Bandar (DANCEY; MCLEAN; BANDAR, 2004). The ExTree differs from the TREPAN because it uses single-feature tests, like C4.5 and CART algorithms. Additionally, it uses a different prunning method based on the replacement of subtrees. It aims to create a smaller tree with better generalization.

4.3 Design of a Tree Structured Neural Network using Genetic Programming

In 1997, Byoung Tak Zhang, Peter Ohm and Heinz Mühlenbein introduced the term "neural trees" to describe an evolutionary method that induces networks using both genetic programming and genetic algorithms (ZHANG; OHM; MÜHLENBEIN, 1997). The technique evolves the weights and the topology of the network, always following the minimum description length principle. To guarantee the generalization, it creates high-order nets with sparse structures, as displayed in figure 4.3. The method was succesfully utilized to predict two chaotic time-series data sets.

A FNT (*flexible neural tree*) can be defined as a multilayer feedforward neural network with the following characteristics:

- It has connections over layers;
- The different nodes have variable activation functions;

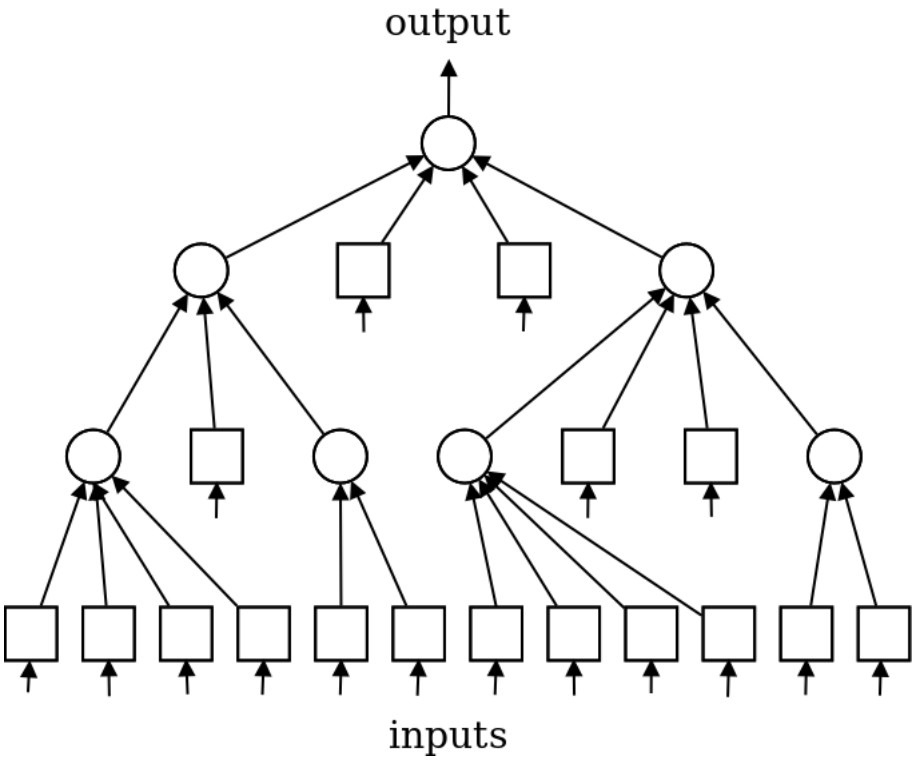

Figure 4.3: An example of a neural tree.

- There are sparse connections between the nodes.

A typical FNT is represented on figure 4.4. Each node of the tree can be described as a flexible neuron operator such as shown on figure 4.5. The activation function is variable because it calculates, besides the neuron inputs (x_i) and the weights (w_i), some additional parameters (a_i and b_i) that are unique to each neuron. For example, Yuehui Chen, Bo Yang, Jiwen Dong and Ajith Abraham utilized $f(a_i, b_i, x) = e^{-((x-a_i)/b_i)^2}$, where x is defined as $x = \sum_i^n x_i * w_i$. They utilized it to forecast time-seris using FNTs (CHEN et al., 2005).

Yuehui Chen, Bo Yang and Ajith Abraham utilized GP (*genetic programming*) and PSO (*particle swarm optimization*) to train FNTs for stock market predictions (CHEN; YANG; ABRAHAM, 2007). Their method interleaves both methods: the GP utilizes special genetic operators to create the topology while the PSO optimizates the weights. The steps are repeated until a reasonable model is created.

The authors utilized the standard "crossover" and "selection" operators in the GP. Additionaly, they created four specific "mutation" operators, as follows (CHEN; YANG; ABRAHAM, 2007):

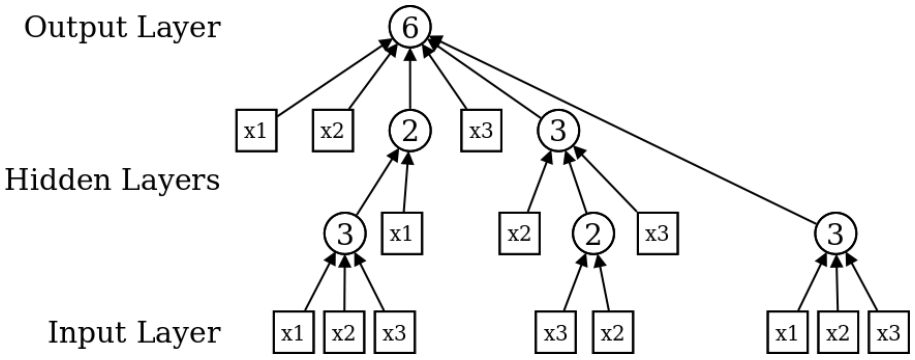

Figure 4.4: A typical flexible neural tree.

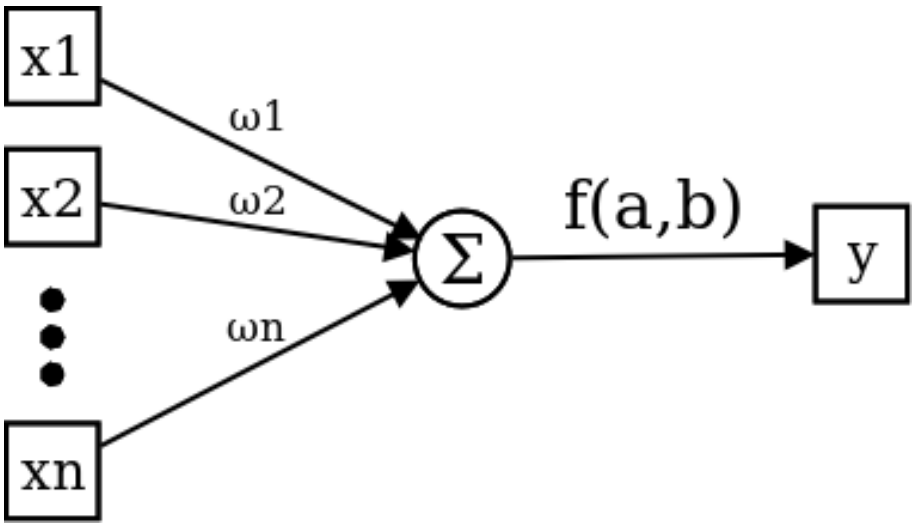

Figure 4.5: A flexible neuron operator.

- **Changing one terminal node:** randomly select one terminal node in the neural tree and replace it with another terminal node.
- **Changing all the terminal nodes:** select each and every terminal node in the neural tree and replace it with another terminal node.
- **Growing:** select a random leaf in hidden layer of the neural tree and replace it with a newly generated subtree.
- **Pruning:** randomly select a function node in the neural tree and replace it with a terminal node.

The learning method was described by the authors in this way:

1. Create an initial population of FNTs randomly.
2. Repeat until a satisfactory result is found:
   (a) Optimize the topology of the FNTs using the GP.
   (b) Select the FNT with the best topology.
   (c) Fix its structure.
   (d) Optimize its weights using PSO.

This hybrid method was also utilized to forecast exchange rates (CHEN; PENG; ABRAHAM, 2006a). Their paper proposed a FNT trained with GP and PSO that was able to predict the exchange rates of three major currencies: euros, British pounds and Japanese yen. The results suggested that FNTs can overcome conventional MLPs in this kind of problem.

In like manner, Chen, Peng and Abraham utilized FNTs to classify gene expression profiles (CHEN; PENG; ABRAHAM, 2006b). The FNTs were created and evolved using GP and PSO. The method was utilized to classify two cancer databases (leukemia and colon cancer) and presented very attractive results.

The same authors successfully utilized a similar strategy to detect network intrusions (CHEN; ABRAHAM; YANG, 2007)(CHEN; ABRAHAM, 2005). However, in this particular paper, they preferred using EP (*evolutionary programming*) instead of GP. The other steps of the process remained unchanged. Still in the same approach, Chen, Yang and Jiwen Dong tested “ant programming” as an alternative to GP on some time series

prediction problems (CHEN; YANG; DONG, 2004). Again, the method proved to work nicely.

An ensemble of FNTs were successfully utilized to detect breast cancer (CHEN; ABRAHAM; ZHANG, 2006). Six different FNTs were trained with PIPE (*probabilistic incremental program evolution*) and PSO algorithms. The PIPE algorithm created the topology of the trees, while the PSO optimized their weights. Each training utilized different sets of input variables to learn the different FNTs. This way, they could be combined in a global classifier with better accuracy. This approach proved to be correct and the combined FNTs achieved higher classification rates.

Several techniques were utilized to evolve FNTs, such as GP, PIPE and ant programming. Peng Wu and Yuehui Chen investigated a new hybrid learning method with GGGP (*Grammar Guided Genetic Programming*) and PSO (WU; CHEN, 2007). Using a predefined grammar, the GGGP is able to evolve the topology of a FNT, which weights can be optimized by PSO. The method was empirically tested to predict stock indexes and overcame other approaches based on genetic algorithms or neural networks.

Yuehui Chen, Feng Chen and Jack Y. Yang also proposed a different kind of FNT with multiple inputs and multiple outputs, which they called MIMO-FNT (*multiple-input and multiple-output flexible neural tree*) (CHEN; CHEN; YANG, 2007). To deal with the MIMO problem, the authors added some "dummy" neurons (Out_1, Out_2, ..., Out_m) as shown in figure 4.6. Their method utilizes a hybrid approach to learn the MIMO-FNTs, using IP (*immune programming*) and PSO as follows:

1. Define the IP and PSO parameters.
2. Create an initial population of MIMO-FNTs limited by the given parameters.
3. Optimize the weights of each MIMO-FNT using the PSO algorithm.
4. Optimize the structure of each MIMO-FNT using the IP algorithm.
5. Repeat the optimization steps until a stopping criteria is met.

In 2009, Qu Shou-ning, Liu Zhao-lian,Cui Guang-qiang and Fuai-fang trained FNTs using PIPE (*probabilistic incremental program evolution*) and SA (*simulation annealing*) (SHOU-NING et al., 2009). The general dynamics of their method resembles the Chen's approach, but with different techniques (PIPE and SA instead of evolutionary methods). The strategy was successfully utilized to control fluids in industry.

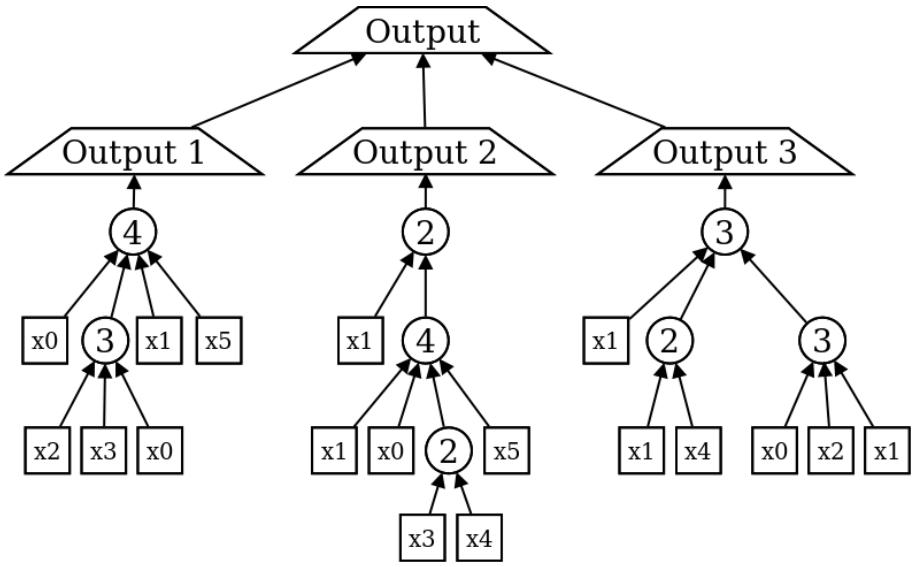

Figure 4.6: A typical MIMO-FNT.

The FNT model was also applied by Chen and others to predict TCP traffic data from a website (CHEN; YANG; MENG, 2012). In this application, the topology of the FNT was developed using genetic programming while PSO was utilized to otimize its weights.

In 2012, Yina Guo, Qinghua Wang, Shuhua Huang and Ajith Abraham successfully applied to recognize hand gestures (GUO et al., 2012). In this work, the FNTs were trained to interpret surface electromyography signals recorded of superficial muscles from the skin surface. The model was able to classify six different hand gestures in real time, with 97.46% of accuracy.

In 2013, Souhir Bouaziz, Habib Dhahri, Adel M. Alimi and Ajith Abraham proposed a different type of FNT that utilizes flexible neuron beta operators (BOUAZIZ et al., 2013). They called their method as "Flexible Beta Basis Function Neural Tree" (FBBFNT). The method utilizes extended genetic programming to evolve the structure whereas opposite-based PSO optimizes the weights.

4.4 Derivation of a modular neural network by softening the nodes of a decision tree

In 1998, Cezary Janikow investigated the softening of the nodes of a decision tree using fuzzy functions (JANIKOW, 1998). This way, he tried to achieve a fuzzy decision tree that could deal better with inexact and uncertain information and keep the usual

advantages of decision trees. After several tests, he proved that his fuzzy decision trees could even produce real-valued outputs (with gradual shifts), as seen in the figure 4.7.

Fuzzy decision trees were also investigated by Alberto Suárez and James F. Lutsko (SUÁREZ; LUTSKO, 1999). They superimposed a fuzzy structure over the skeleton of a decision tree that was built with the CART algorithm. Then, the authors proposed a learning algorithm to fix the parameters of the fuzzy nodes. The method proved to be useful on regression and classification problems.

Ö. Ciftcioglu, M.S. Bittermann and I.S. Sariyildiz proposed a fuzzy logic system that mixes Gaussian membership functions and neural trees, as seen in the figure 4.8. The Gaussian functions utilize the weights of the connections as parameters. Once the learning algorithm updates these weights, the format of each Gaussian function is changed. Two possible formats are shown in the figure 4.9. These "Gaussian neurons" transform the neural tree on a complete fuzzy system, where the top of the tree plays the role of defuzzification unit.

4.5 Design of Decision Trees with Embedded Neural Networks

Decision trees, neural networks and genetic algorithms were combined by Qiangfu Zhao to create what he called "neural network tree" (NNTree). Basically, a NNTree is a decision tree with each node being an expert neural network (ZHAO, 2001). The global structure of the tree is created using the traditional C4.5 algorithm. Each node is a fixed-topology neural network, that is trained with a genetic algorithm. The method generates a network like the one in figure 4.10. In the paper, the NNTree was tested against a conventional decision tree in a digit recognition problem. The results suggested that NNTrees may be more efficient than normal decision trees, since they can achieve higher accuracy with less nodes. However, the work cannot be considered totally conclusive, once its scope is limited to only two techniques and one data set.

As explained before, each node in a NNTree is an embedded neural network that deal with binary inputs. The node (actually, a neural network) produces an output value between 0 and 1. If the value is less than the threshold (fixed on 0.5), the decision algorithm visits the left child node. Otherwise, it goes to the right child node. The threshold is usually fixed on 0.5 because the algorithm expects that the embedded MLP

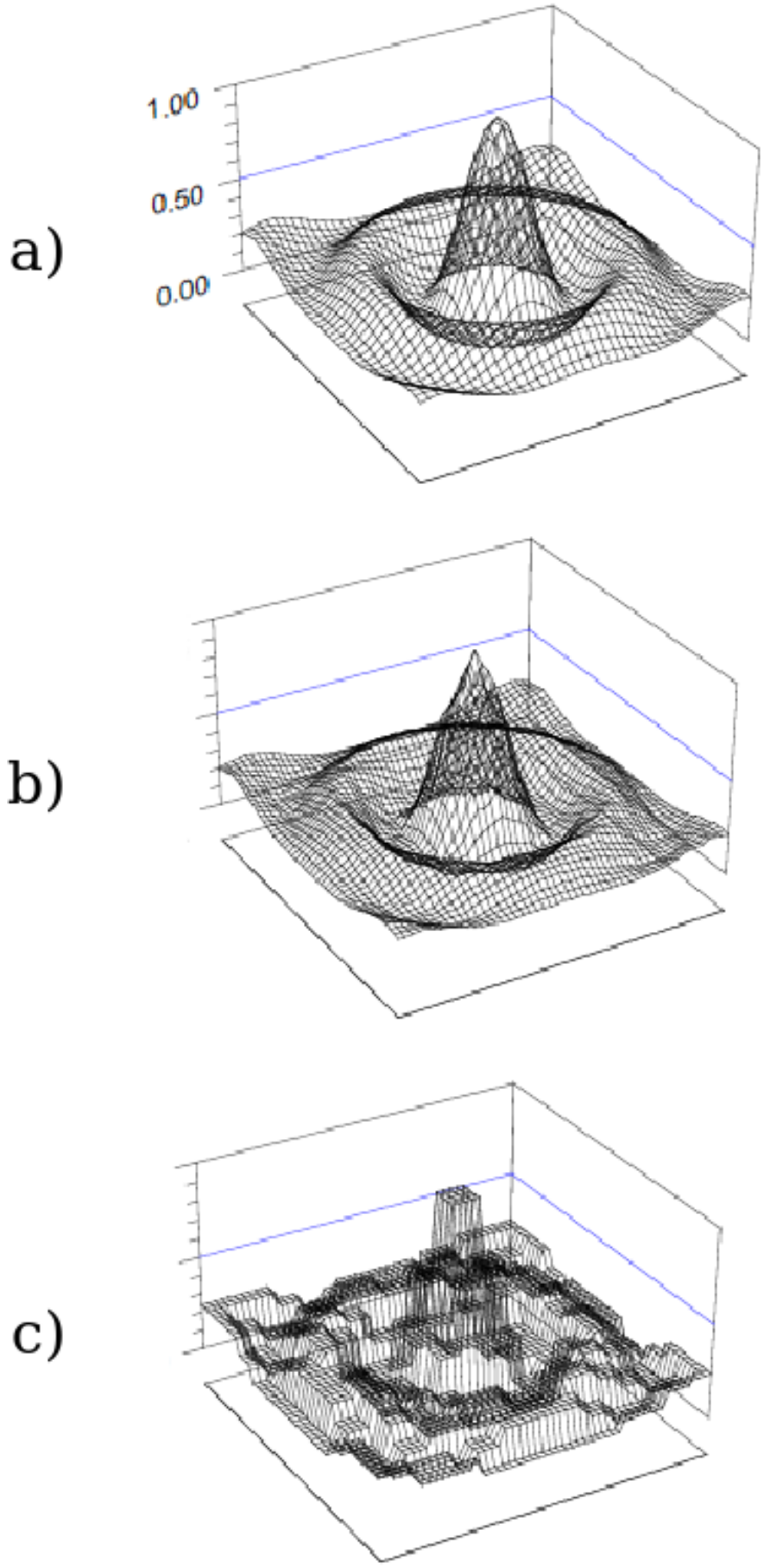

Figure 4.7: A original function (a) and the responses of two different fuzzy decision trees (b)(c).

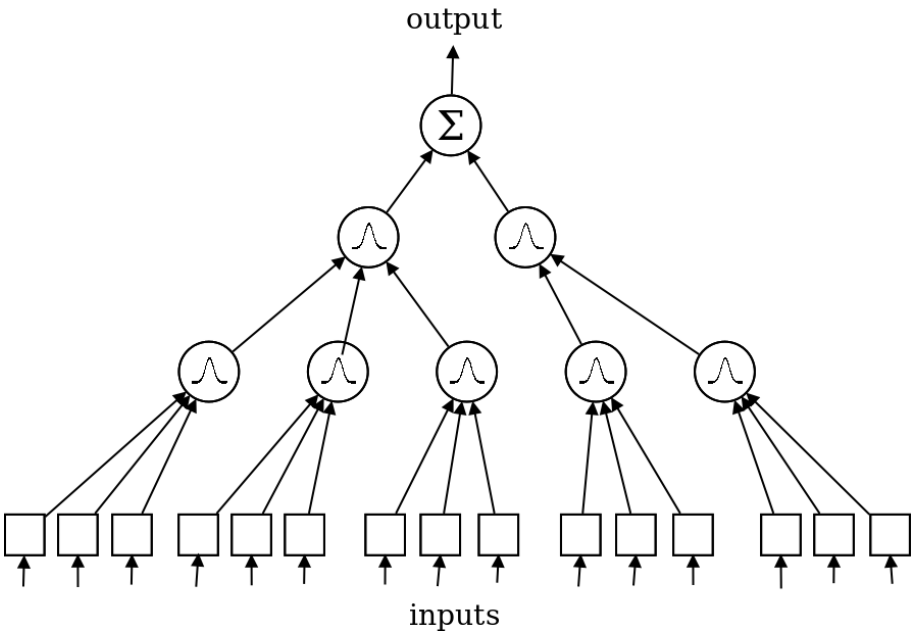

Figure 4.8: A neural tree with Gaussian membership functions.

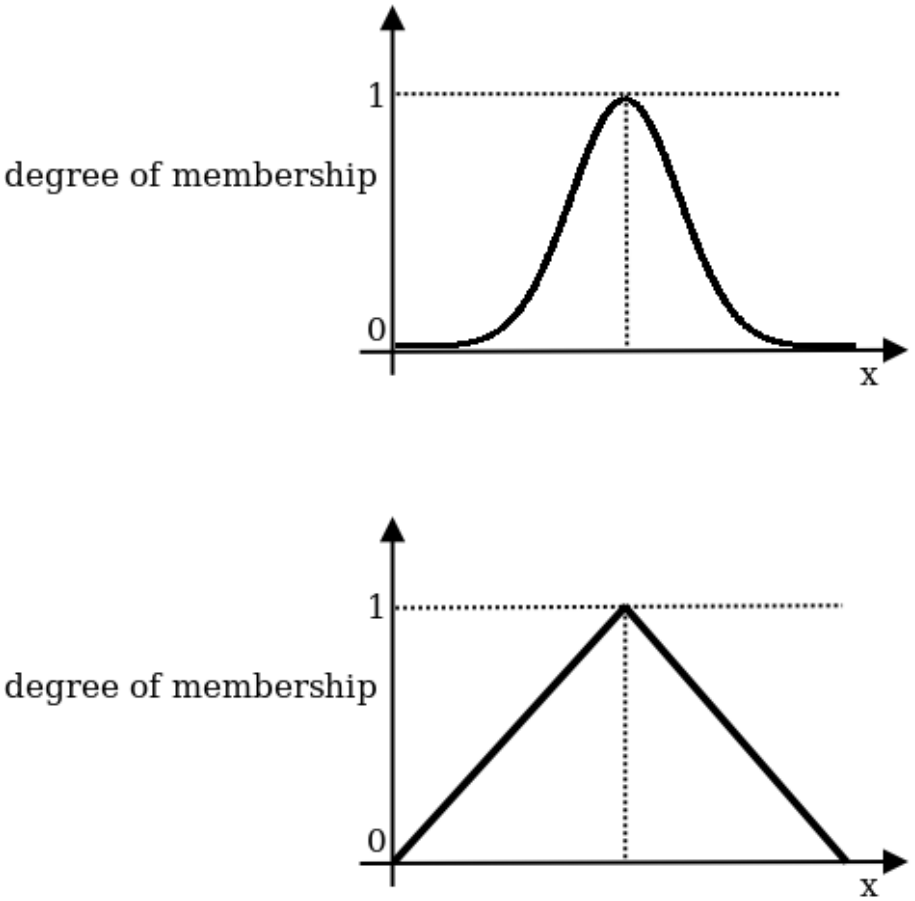

Figure 4.9: Two possible Gaussian membership functions.

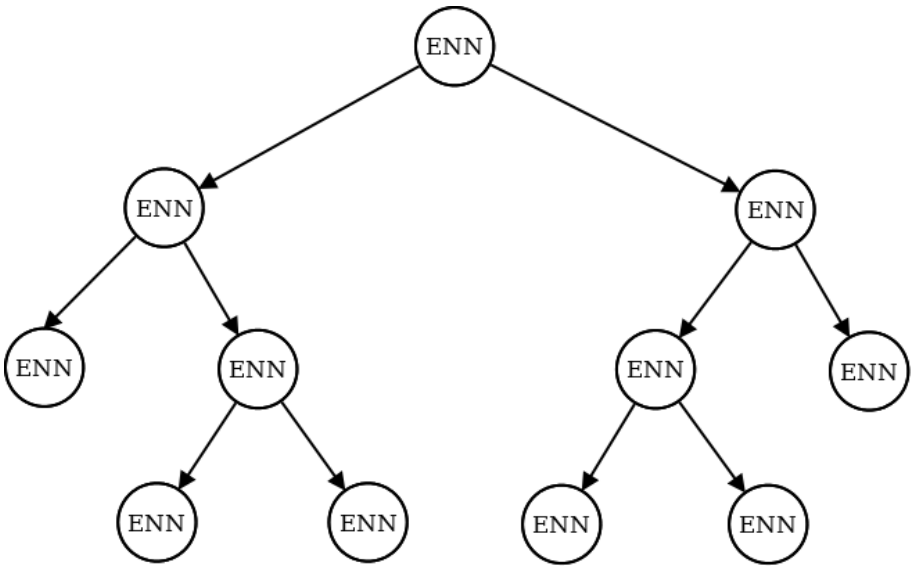

Figure 4.10: An example of a NNTree.

will project the data to the neighborhood of 0 or 1, as shown in case 1 of figure 4.11. However, the MLP can project outputs like case 2 or case 3. Hirotomo Hayashi and Qiangfu Zhao observed that if they tune this threshold value, it would be possible to obtain better classification rates. Their fine-tuning algorithm was described as follows (HAYASHI; ZHAO, 2009):

1. Obtain the output values $y_1, y_2, ..., y_n$ of the neural network for all data assigned to the current node.

2. Sort the data according to the output values $y_1, y_2, ..., y_n$.

3. Calculate the average values $a_k = (y_k + y_{k+1})/2$, for $k = 1, 2, ..., n-1$.

4. Calculate $IGR(a_k)$, which is the information gain ratio corresponding to a_k, for $k = 1, 2, ..., n-1$.

5. The desired threshold is given by $T = \arg\max_k IGR(a_k)$.

One of the great advantages of NNTrees in comparison to neural networks is their interpretability. The experiments suggested that NNTrees could be easily interpreted since each embedded neural network had a limited quantity of neurons. Takeda Takaharu and Qiangfu Zhao considered a two-stage training approach that could reduce the NNTrees size (TAKAHARU; ZHAO, 2002). Firstly, they trained the NNTree with, for example, only 10% of the entire data set. This first training generates the topology of the NNTree.

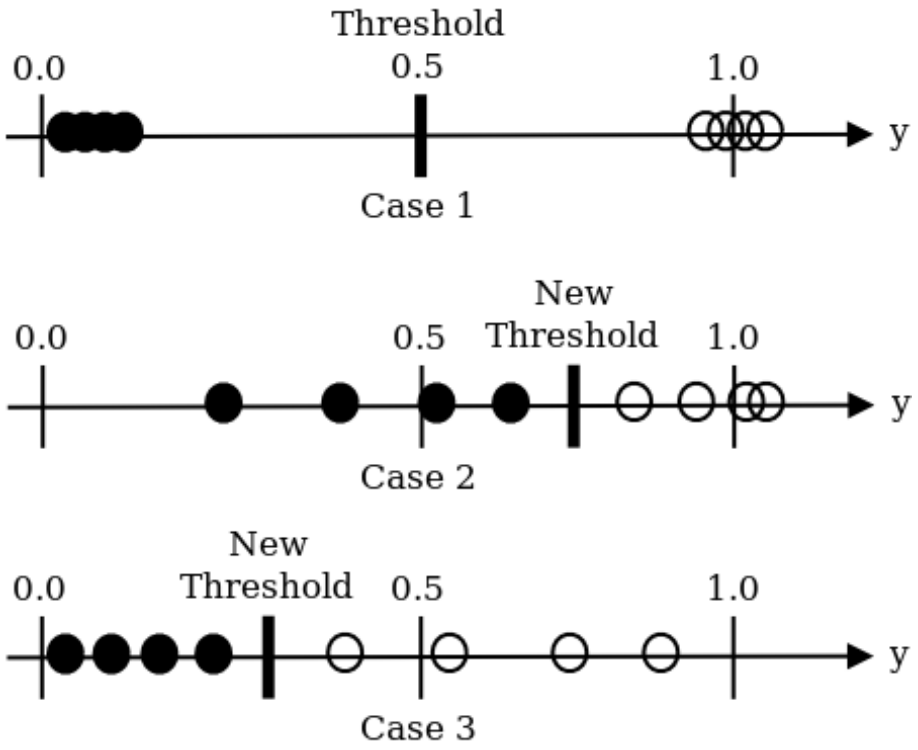

Figure 4.11: Fine tuning of threshold.

Once the size of the tree is approximately proportional to the quantity of examples, the algorithm creates a smaller model. Then, the entire data set is utilized to optimize only the weights of the NNTree. The method worked well for situations where the database is highly redundant. In other situations, the retraining of the small NNTree did not generate a good classifier.

Another way to deal with the NNTree size problem was proposed by Hirotomo Hayashi and Qiangfu Zhao (HAYASHI; ZHAO, 2009a). They investigated the dimensionality reduction using two different methods: PCA (*principal component analysis*) and LDA (*linear discriminant analysis*). The experiments confirmed that, in most cases, the LDA approach can succesfuly reduce the parameters and produce smaller NNTrees (less nodes) without degrading the performance. Nevertheless, the LDA has a high computational cost when applied to large databases. To solve, the same authors studied centroid based approaches for NNTrees induction (HAYASHI; ZHAO, 2009b). Their experiments indicated that these techniques are usually comparable to LDA approach.

However, when the input features are continuous, the quantity of neurons tends to grow up easily. It happens because NNTrees only deal with binary values. For example, if each continuous input is represented by a 16-bit binary number, the quantity of input neurons is multiplied by 16. Since the quantity of hidden neurons is directly proportional

to the quantity of input neurons, the network size easily grows up.

To deal with it, Qinzhen Xu, Qiangfu Zhao, Wenjiang Pei, Luxi Yang and Zhenya He proposed the quantization of the continuous inputs as a way to reduce the size of the embedded neural networks (XU et al., 2004). The main principle was to obtain the same performance with less neurons, which would increase the network interpretability since it is easier to interpret smaller nets. Their research indicated that usually less than 10 quantization points are necessary to map a continuous feature. This way, it can be represented with 3 or 4 binary inputs, representing a great reduction against the initial 16 bits. The NNTrees that were obtained with this process proved to be more interpretable, besides conserving the original performance.

Takeda Takaharu, Qiangfu Zhao and Yong Liu proposed two on-line learning algorithms for NNTrees (TAKAHARU; ZHAO; LIU, 2003), which they called as GUWL (*growing up with learning*) and SGU (*simple growing up*). With these modifications, the NNTree is updated if it cannot classify a pattern. The SGU creates an additional node to tree, while the GUWL tries to retrain the network before creating new nodes. Despite of the interesting motivation, these algorithms are not very useful since they are not faster than just recreating the entire NNTree with the updated database.

NNTrees were also used by Xu Qinzhen, Yang Luxi, Zhao Qiangfu, He Zhenya and Wenjiang Pei in intrusion detection systems (XU et al., 2006a)(XU et al., 2006b). Indentifying unauthorized access to a computer has become extremely important, because of the expansion of distributed systems. Once it is a NP-complete problem, many authors were stimulated to test several AI techniques to solve it, like hidden Markov models, neural networks, statical techniques and expert systems. In this specific domain, the authors verified that NNTrees could overcome a MLP trained with backpropagation, both on accuracy and learning time.

In 2002, Gian Luca Foresti and Christian Micheloni presented a new training rule that reevaluates the whole tree each time a new level is created, which they called as GNT (*generalized neural tree*) (FORESTI; MICHELONI, 2002). The weights correction strategy considers the entire tree. Each tree node is a local neural network. This way, the connected nets create a global neural network, with tree structure. It is interesting to observe that the authors implemented a normalization rule that fixes the output of each local network, as shown in figure 4.12. It guarantees that the sum of the outputs of a local network will be always equals to 1. In this way, the outputs can be interpreted as probabilities, which are used during weights updating. The GNT had a good classification

performance and was tested against training sets with complex distribution, as seen in figure 4.13.

Gentili, Bragato and Michelini applied neural trees in the automatic detection of earthquakes in Italy (GENTILI; BRAGATO, 2006) (GENTILI; MICHELINI, 2006). They utilized a tree where each node is a MLP, as shown in figure 4.14. This formalism was initially proposed by Stefania Gentili and called IUANT2 (GENTILI, 2003). This technique automatically generates the topology of the network.

In 2012, Micheloni, Rani, Kumar and Foresti proposed a new model that they called "balanced neural tree" (BNT) (MICHELONI et al., 2012). The BNTs follow the concept a decision tree structure where each node is, in fact, a perceptron. However, this method introduces two differences: perceptron substitution and pattern removal. The perceptron substitution involves the replacement of nodes that are not efficient with the objective to balance the structure of tree. On the other hand, the pattern removal intends to avoid overfitting by removing outliers from the training set.

Biswal, Jalaja and others have utilized the BNTs to classify different power signal disturbances (JALAJA; BISWAL, 2013) (BISWAL et al., 2014).

4.6 Other Methods

In 1989, Paul E. Utgoff proposed the perceptron trees, which are a hybrid combination of decision trees and LTU (*linear threshold units*) (UTGOFF, 1989). Basically, a perceptron tree can be defined as a tree where each node is a Rosenblatt's perceptron. An example is showed in figure 4.15. Utgoff's trees were one of the first attempts to mix neural networks and decision trees in most robust models. The perceptron trees proved to be smaller than decision trees, mainly because of the classification power of the LTUs. In other words, as LTU is a better model than a IF clause, they need less nodes. However, once a perceptron tree was still based on linear models, its classification performance was limited when compared to several non-linear approaches.

Wilson Wen, Andrew Jennings and Huan Li proposed a hierarchical clustering algorithm that creates the topology of a neural network using a decision tree algorithm (WEN; JENNINGS; LIU, 1992). They called it SGNT (*Self-Generating Neural Tree*) algorithm, as defined in algorithm 5. The method executes three steps: generation, optimization and prunning. These steps are shown in the figure 4.16.

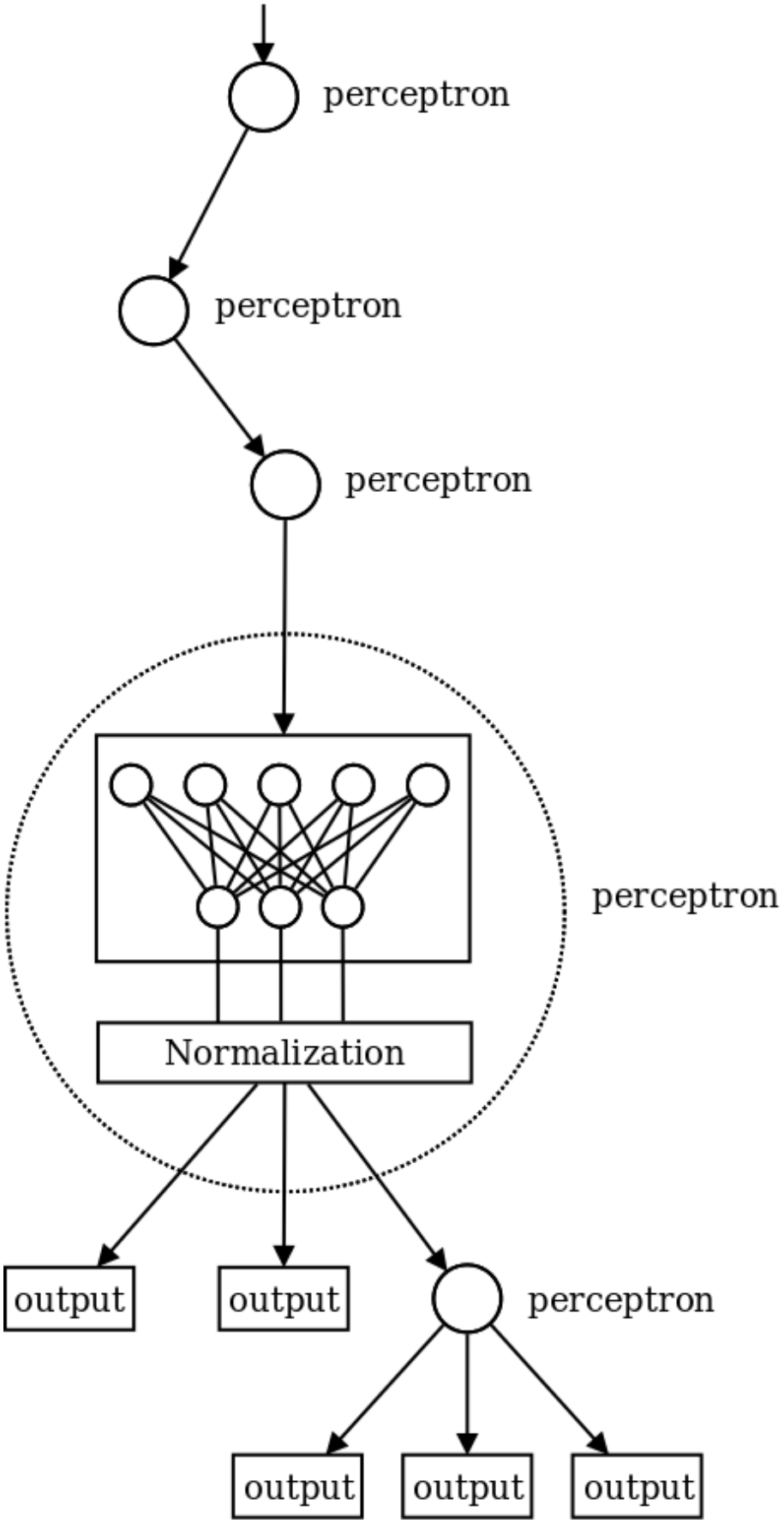

Figure 4.12: An example of a generalized neural tree.

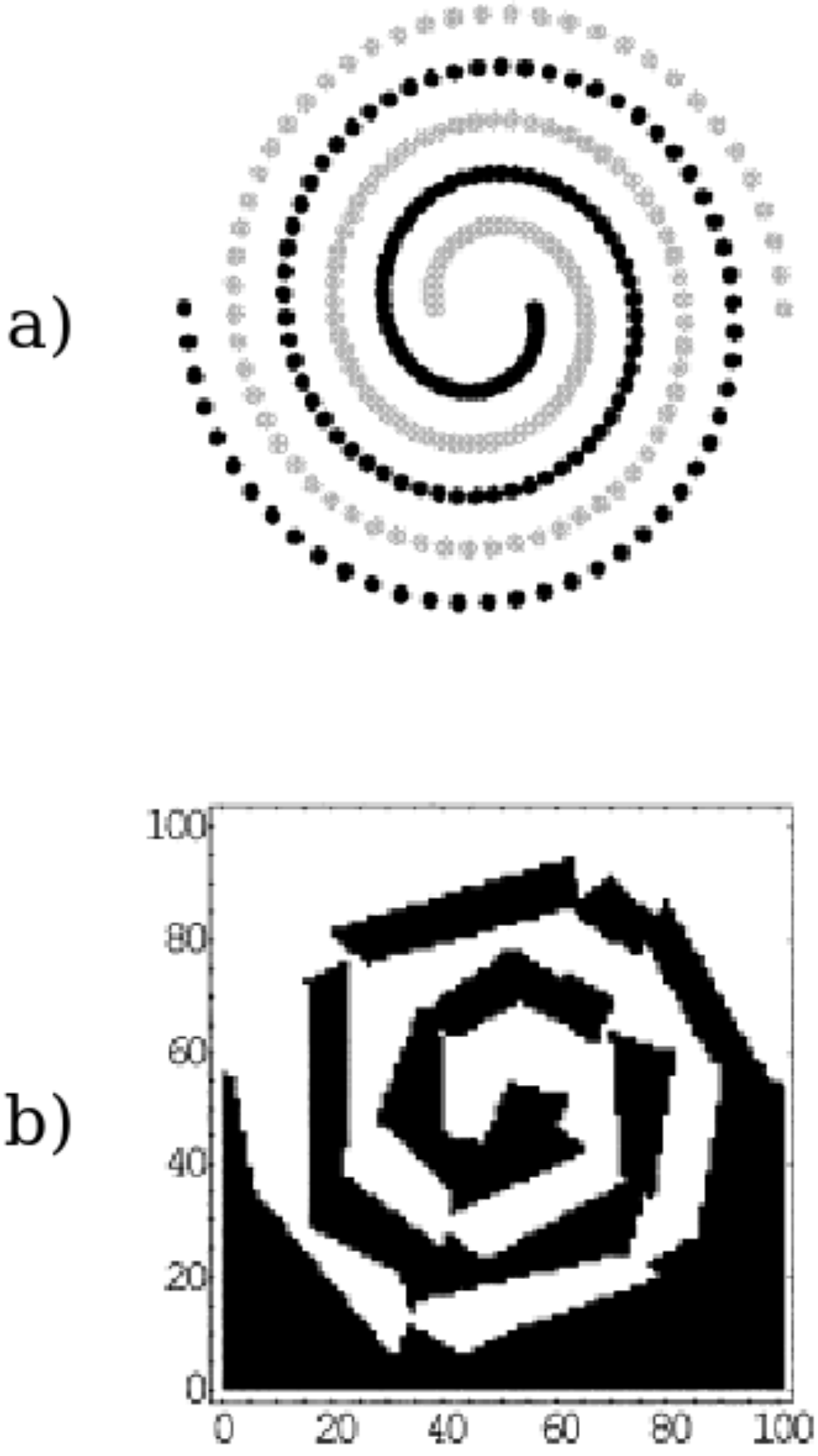

Figure 4.13: (a) Double spiral database. (b) Classification obtained by a GNT.

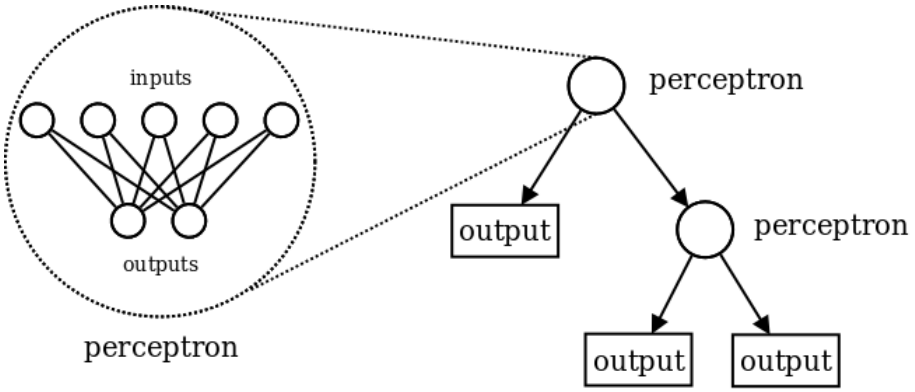

Figure 4.14: A simplified schematics of the neural trees that were utilized by Gentili, Bragato and Michelini to detect earthquakes.

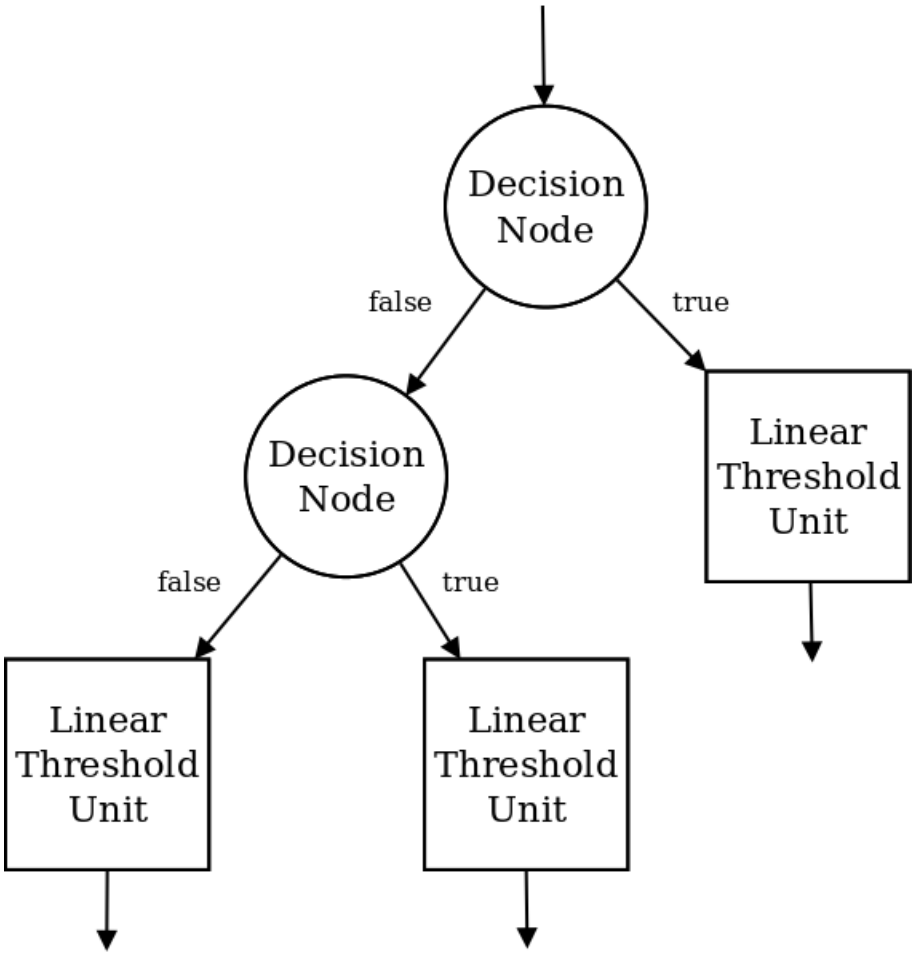

Figure 4.15: An example of an Utgoff's perceptron tree.

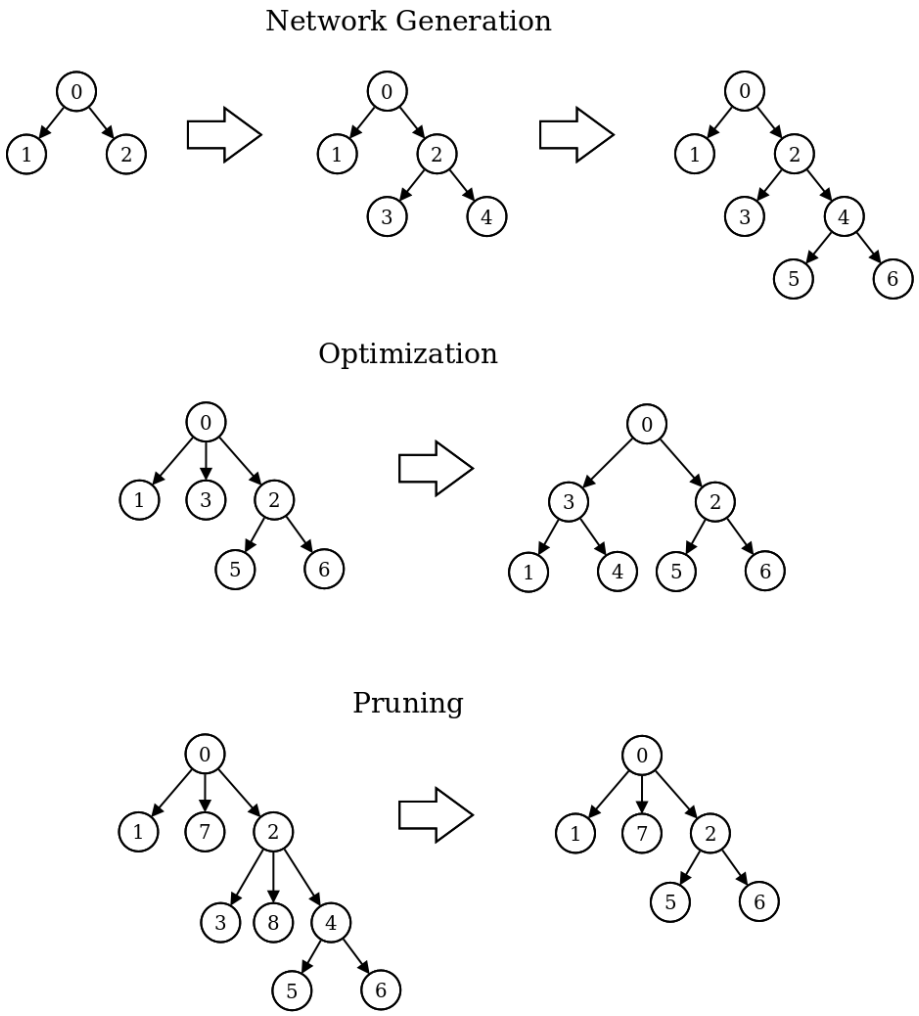

Figure 4.16: Steps involved in the creation of a Self-Generation Neural Tree.

Algorithm 5 Self-Generating Neural Tree algorithm

Require: A group of training examples $E = e_i, i = 1, ..., N$.
Require: A threshold $\xi \geq 0$.
Require: A distance measure for each attribute/weight in instances/neurons.
Ensure: An SGNT created from E.

```
Create a root neuron and copy the attributes/weights in the instance/neuron e_0 to root.
i ← 1
j ← 1
while i ≤ N do
  winner ← root
  minimumDistance ← distance between instance e and winner neuron
  if minimumDistance ≥ ξ then
    if neuronhas no children then
      Create a n_j neuron and copy the attributes/weights in the instance/neuron n_j
      to winner.
      Connect neuron n_j to winner neuron.
      j ← j + 1
    end if
    Create a n_j neuron and copy the attributes/weights in the instance/neuron n_j to
    e_i.
    Connect neuron n_j to winner neuron.
    j ← j + 1
  end if
  Update the weights vector of neuron n_i.
  i ← i + 1
end while
```

In a related paper, the authors pointed some attractive features of SGNTs in comparison to another AI techniques (WEN et al., 1992):

- The structure is adaptively determined.
- There is an efficient use of neurons and connections.
- There is no delay in the learning process once there are no redundant neurons.
- They can be used on supervised, unsupervised and hybrid learning tasks.

Shu-Hsun Lo proposed a new kind of neural tree, which was called QUANT (*quadratic-neuron-based neural tree*) (LO, 2007). The QUANT method utilizes a batch learning algorithm to grow a tree structured neural network formed by quadratic neurons. These neurons recursively split the features space into hyper-ellipsoidal regions, so the QUANT can be utilized on classification problems. An interesting characteristic of the method is its plasticity. Basically, the QUANT can be partially retrained, receiving new quadratic

neurons. This way, there is no need to rebuild the entire network. The success of QUANT in spam detection stimulated further studies (SU; LO; HSU, 2010).

In 2013, Jarmila Skrinárová, Ladislav Huraj and Vladimír Siládi proposed a neural tree architecture with probabilistic neurons (SKRINáROVá; HURAJ; SILáDI, 2013). These trees were utilized for classification of a large amount of computer grid resources. The tests demonstrated that the neural trees improved the performance of the classification in this task when compared to other methods.

4.7 Chapter Review

This chapter presented the most utilized methods involving neural networks, decision trees and/or evolutionary algorithms. The state of art was described in details. This analysis is essential, once the proposed method is a hybrid method by itself. By studying the past works, it is possible to detect the innovations that are being proposed here.

Using the taxonomy suggested by Quiangu Zhao (ZHAO, 2001), the proposed method would be classified as a mix of three classes:

- **Derivation of a neural network from a previously induced decision tree:**

  Technically, the proposed algorithm does not derive the network, since it does not create the decision tree. However, it is fully inspired on the C4.5 learning algorithm principles. So, it is reasonable to assume that it, at least, partially part of this class.

- **Design of a tree structured neural network using genetic programming:**

  Although Zhao uses the term "genetic programming", this class groups all hybrid methods that utilize some kind of evolutionary algorithm. The proposed method uses PSO to optimize the connections weights, so it is part of this class.

- **Design of decision trees with embedded neural networks:**

  The proposed method generates a complete neural network by incrementally adding perceptrons that performs different operations: "AND", "OR", "greater than", and others. Initially, each perceptron is separately trained. Thus, the final network can be also described as several connected perceptrons that perform each one a separated function. However, this is a partial phase of the process, once the

PSO evolves the weights and the perceptrons may "lose" their local functions in favor of the global network.

Based on the theory that was described, the next chapter will present the proposed algorithm. All the principles that lead to the method are fully described. By seeing the topology of the created network, it is possible to see its differences from the methods in the state of art.

Chapter 5

Proposal

The main theme of the present thesis is the creation of a constructive learning algorithm for artificial neural networks. On this purpose, three challenges must be faced:

1. Generally, easy learning techniques expect a fixed-topology neural network. However, one of the premisses of this research is not utilize a human-defined topology. This way, it is necessary to find some algorithm that automatically builds the network architecture.

2. By definition, easy learning techniques are based on the single-cell learning. To accomplish that, the learning algorithm demands inputs and outputs for each neuron. As the data sets do not have this information, an automatic creation method must be developed.

3. Finally, beyond the single-cell learning, this proposal aims to optimize the entire network. Normally, the backpropagation algorithm could solve that. However, as this project involves the creation of a constructive learning algorithm, it is preferable to find a parallel training solution.

Firstly, this chapter presents how to deal with each of the above challenges. Further, an algorithm that meets all these statements is described, as the experiments that may confirm its validity.

5.1 Network Topology

How could one automatically establish the topology of a neural network? Considering that there is no efficient method for determining an optimal network topology(PAREKH; YANG; HONAVAR, 2000), some kind of heuristic should be found. But what are heuristics?

> *"Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices." (PEARL, 1984)*

So, it is necessary to find a simple principle to guide the algorithm in the creation and interconnection of the neurons. Once this project aims to achieve a generic method, it is very important that this same principle could be utilized in many domains of application. To find a good heuristic, one has to detect an aspect of the context that could base it.

It is very interesting to observe that, despite the fact that artificial neural networks have their inspiration on biological neurons, the common topologies are not very "biological". For example, MLPs are usually composed by three or more single-row layers, where each neuron of a layer is interconnected to all the neurons of near layers. This "geometric" positioning cannot easily be seen in natural world. And it definitely does not look like a human brain. The same finding could be done regarding other methods, like self-organizing maps or adaptive resonance theory neural networks.

Based on this observation, it is reasonable to affirm that a more "biological" topology may be quite relevant. In nature, there are many structures that could provide inspiration to the topology, such as spider webs, trees, rivers, and others. The present study is particularly interested in the use of trees as inspiration in the construction of the network architecture. This preference is based on the following reasons:

- The "tree" structure is very organic and exists in many biologic organisms.
- A tree can be defined as a graph model, so it is well suited to be a knowledge model.
- Tree-like models are very easy to understand and interpret.
- Many authors developed several learning algorithms for decision trees. So, there are many heuristics to choose as the base of the topology-constructor algorithm.

Indeed, the existence of many tree learning algorithms is a great incentive to this choice. These algorithms are able to build a model that predicts an attribute based on other attributes. In other words, they create classification models. Once they build these models automatically from a collection of examples, one can say that they are a good solution for the problem of creating a network topology.

The literature describes several algorithms, such as ID3, C4.5, random forests, MARS, and others. This study will prefer the C4.5 (QUINLAN, 1993), which is an extension of ID3 algorithm (QUINLAN, 1986). The C4.5 was chosen because it can handle both categorical and numerical attributes. The heuristic is based on the concept of information entropy. It constructs the topology by iteratively choosing the attributes with better normalized information gain. In each interation, the algorithm chooses the attribute that better splits the data. The pseudocode of the C4.5 algorithm is (KOTSIANTIS, 2007):

- Check for base cases;
- For each attribute a:
  - Find the normalized information gain from splitting on a;
- Let a-best be the attribute with the highest normalized information gain;
- Create a decision node that splits on a-best;
- Recurse on the sublists obtained by splitting on a-best, and add those nodes as children of node.

The pseudocode algorithm could be the base of the proposed topology creator algorithm. However, it should be changed to create a neural network instead of a decision tree. The classical “golf” problem can be utilized to illustrate this nee. The “golf” problem is about the creation of a model that rules whether to play a game of golf or not. Considering that the table 5.1 contains the training data of the problem, the application of C4.5 algorithm creates the decision tree showed in the figure 5.1.

As seen in the created model, the C4.5 generates a perfect tree-like model that starts with one root that splits in two branches, and so successively. The tree stops at the leaves (that contain the classes), and you can have many leaves representing the same class. This architecture represents well a decision tree, but not a neural network. Basically, a neural network cannot start with just one root node. Indeed, all the attributes have to be

Table 5.1: “Golf” problem: training data.

Outlook	Temperature	Humidity	Windy	Play (positive) / Don’t Play (negative)
sunny	85	85	false	Don’t Play
sunny	80	90	true	Don’t Play
overcast	83	78	false	Play
rain	70	96	false	Play
rain	68	80	false	Play
rain	65	70	true	Don’t Play
overcast	64	65	true	Play
sunny	72	95	false	Don’t Play
sunny	69	70	false	Play
rain	75	80	false	Play
sunny	75	70	true	Play
overcast	72	90	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don’t Play

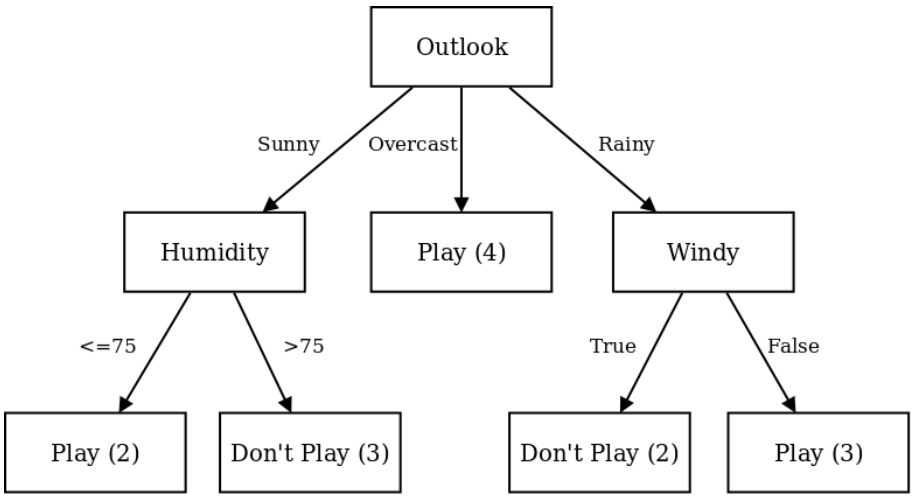

Figure 5.1: “Golf” problem: decision tree.

input neurons. Also, each class must be mapped to just one output neuron. These two differences affect completely the topology of the tree (mainly, the start and the end).

Based on that perception, the algorithm should be slightly changed so it can build an adequate network topology. On this purpose, four crucial questions may be answered:

- How to deal with numerical inputs?
- How to deal with categorical inputs?
- How to deal with logical inputs?
- How to guarantee that each class is mapped to just one output neuron?

The main concern about numerical inputs is how to present them to the network. Basically, an input is processed in a "if" clause, where the node checks if it is greater, equal or lesser than a reference value. For example, in the "golf" example, the model checks if the humidity is greater than 75%. This operation can be accomplished by a single neuron. However, the scale must be observed. Normally, a neuron is a processing unit that is modelled to work in a unitary scale (for example, -1 to 1). Then, it is mandatory to normalize the numerical attribute before presenting it to the network.

Categorical inputs are more complicated. It is not possible to map all the possible values of a categorical attribute to a single input neuron. The better option is to create as many input neurons as existent categories. Using the "outlook" attribute of the "golf" as example, three input neurons would be created: "sunny", "overcast" and "rain". This way, one would activate just the respective input neuron to determine the value of that attribute.

The logical inputs can assume two values: true or false. Initially, a single neuron could represent it. However, it is important to reflect about how decision trees concatenate the clauses. It can be illustrated by a complete logical rule from the "golf" problem, which is "IF outlook = sunny AND humidity $>$ 75 THEN do not play". These "AND" logical operations can be performed with additional neurons, positioned between the numerical and categorical clauses.

Finally, each class must be mapped to a single output neuron. To do this, the algorithm needs to remember the class of each created output neuron, so it can create a new connection instead of a new neuron. It is important to observe that the output neurons will perform an "OR" logical operation. In other words, an output neuron will be active if one or more of its connections are active.

Because of this, it is better to use two input neurons to represent a logical input. This way, it is possible to have the input attribute and its negative version, standardizing the "AND" and "OR" logical operations.

This way, the new topology-builder algorithm will be (in pseudocode):

1. Create one output neuron for each class.
2. For each attribute a:
   (a) Find the normalized information gain from splitting on a.
3. Let a_{best} be the attribute with the highest normalized information gain.
4. If a_{best} is a categorical attribute:
   (a) For each category of a_{best}:
      i. If there is no input neuron for this category, create it.
      ii. Else:
         A. Create an "AND" neuron.
         B. Connect the "AND" neuron to the input neuron of this category.
         C. Connect the "AND" neuron to the neuron that invoked this recursion.
      iii. Remove the training examples where a_{best} is equal to this category.
      iv. If possible, recurse on the subsets.
      v. Else, connect the created neuron to the respective output neuron.
      vi. Put back the removed training examples.
5. If a_{best} is a logical attribute:
   (a) If there is no input neuron when a_{best} is true, create one input neuron.
   (b) Else:
      i. Create an "AND" neuron.
      ii. Connect the "AND" neuron to the input neuron of this logical attribute.
      iii. Connect the "AND" neuron to the neuron that invoked this recursion.
   (c) Remove the training examples where a_{best} is true.
   (d) If possible, recurse on the subsets.
   (e) Else, connect the created neuron to the respective output neuron.

(f) Put back the removed training examples.

(g) Repeat the same steps, considering a_{best} as false.

6. If a_{best} is a numerical attribute:

(a) If there is no input neuron when a_{best} is true, create one input neuron.

(b) Find a threshold value to split the data set.

(c) Create a hidden neuron and connect it to the input neuron.

(d) Remove the training examples where $a_{best} <= threshold$.

(e) If possible, recurse on the subsets.

(f) Else, connect the created neuron to the respective output neuron.

(g) Put back the removed training examples.

(h) Repeat the same steps, considering $a_{best} > threshold$.

5.2 Single-Cell Learning

Another topic point in this thesis lies in the single-cell learning, which is a main characteristic of easy learning algorithms. Essentialy, it is mandatory to find a way to automatically create the inputs and outputs of each cell in the network.

To achieve that, the same C4.5 algorithm may be useful as inspiration. Indeed, the decision tree learning method generates the architecture of the model and the "IF-THEN" clauses of each node. So, each "IF-THEN" clause can be utilized to generate a small set of training examples to its respective neuron. Thus, it will be possible to apply, for example, the Pocket learning algorithm to train that single unit.

Previously, it was described that the network must deal with categorical, logical and numerical parameters. Because of that, there are two categories of neurons:

- Comparational: a neuron that compares the input value against a threshold value, doing a "greater than or equal to" or a "less than" operation.
- Logical: a neuron that performs "AND" or "OR" logical operations.

A generator of examples (inputs and outputs) should be specified for the training of comparational neurons. The algorithm (in pseudocode) would be:

Table 5.2: “AND” truth table.

Input 1	Input 2	Output
Inactive	Inactive	Inactive
Inactive	Active	Inactive
Active	Inactive	Inactive
Active	Active	Active

- threshold := reference value
- mode := “greater than or equal to” or “lower than”
- activeState := 1
- inactiveState := -1
- range := absoluteValue(activeState) - absoluteValue(inactiveState)
- examplesGroup := a desirable quantity of examples
- examplesCount := the quantity of examples
- step := range / examplesCount
- input := inactiveState
- For each example in examplesGroup:
  - If mode is “greater than or equal to”, then:
    - If input is greater than or equal to threshold, then the output is active
    - Else, the output is inactive
  - Else, if mode is “lower than”, then:
    - If input is lower than threshold, then the output is active
    - Else, the output is inactive

On the other hand, logical neurons do not need a specific algorithm, since the truth table of “AND” and “OR” logical operations are very known, being showed in tables 5.2 and 5.3.

Table 5.3: "OR" truth table.

Input 1	Input 2	Output
Inactive	Inactive	Inactive
Inactive	Active	Active
Active	Inactive	Active
Active	Active	Active

5.3 Weights Optimization

After the previous two steps (topology construction and single-cell learning), the network is ready to be utilized. At this point, its accuracy may be similar to an equivalent decision tree. However, it is not enough to make the proposed method an attractive choice. So, ways to improve the final accuracy of the generated network should be considered. In other words, an optimization method is necessary.

A first choice would be the backpropagation learning algorithm. It has already proved its efficiency in the training of MLPs. Nevertheless, backpropagation does not fit the concept of this project. Indeed, it is the complete opposite of the proposed constructive method. The proposed algorithm builds the network in the reverse orientation of backprogation method.

A parallel approach may be more compatible to these needs, since it is not against the adopted heuristic. In previous chapters, it was described that many evolutionary algorithms could perform parallel optimization. This work is particularly interested in particle swarm optimization (PSO), once it can easily perform the optimization of neural network weights.

James Kennedy and Russel Eberhart had proposed the PSO in 1995 (KENNEDY; EBERHART, 1995). The technique was created as a simplified social model of flocks or fish shoals looking for food (VESTERSTROEM; RIGET, 2002). Although the method was first intended for simulating social behaviour, it was discovered that it could perform optimization.

By having a population of candidate solutions (or particles) that are moving around the search-space, the PSO can optimize a problem. The particle movement is based on a simple mathematical formulae which guide them using the best local and global positions.

Differently from classical optimization algorithms, PSO can be applied to any kind of problem. It happens because it does not use gradients during the process, so the optimized

problem does not need to be differentiable. This way, PSO can be defined as an efficient, robust and simple non-deterministic optimization algorithm (BISCAIA-JR.; SCHWAAB; PINTO, 2004).

As commented before, PSO concept can be explained using a "flock metaphor":

1. Let be a flock looking for food.
2. Initially, the birds do not know where the food is.
3. So, each bird starts flying without any pattern.
4. Eventually, they start to follow those birds that are near to food, which creates several small flocks.
5. Finally, one bird finds the food and attracts the others, creating an unique large flock.

This "flock metaphor" can be translated to an algorithm like this:

1. Let be a fitness function $f(x)$ that must be minimized and x_i a candidate solution in the form of a vector of real numbers. So, $f(x_i)$ will output a real number that is the fitness of x_i. The best global particle (the one with the lowest fitness) is represented as g.
2. Produce a population of n particles (candidate solutions) with positions x_i and velocities v_i, where $0 <= i < n$.
3. Initialize the position of each particle as a uniformly distributed random vector.
4. Initialize the velocity of each particle as a uniformly distributed random vector.
5. Repeat until a termination criterion is met[1]:
   (a) Update the velocity of each particle.
   (b) Use each velocity to update its respective particle's position.
   (c) If $f(x_i) < f(g)$, then $g \rightarrow x_i$.
6. The best solution is g.

[1]Normally, a maximum number of iterations or a fitness threshold.

Previous works already proved that PSO can optimize the weights of a neural network (BECKERT-NETO et al., 2010). Indeed, these studies suggested that it can surpass the backpropagation algorithm in many cases. In this kind of application, random network weights are arranged as vectors, which are optimized. The fitness function calculates the total error of a candidate classifying the training examples of a data set. A disadvantage of the method is that it requires a previous defined network topology.

Particularly in this project, the network architecture will be defined by the early described topology-builder algorithm. Additionaly, the single-cell learning process provides an already trained neural network. Thus, instead of creating random candidates, the method will mutate the trained neural network to produce the initial population. The process will be:

1. To arrange the trained network weights as an array.
2. To clone the arrays, creating a population.
3. To mutate randomly some arrays.
4. To apply the PSO algorithm.

After this process, it is expected that the network can overcome the initial accuracy of the equivalent decision tree.

5.4 Pruning

The trained network can be optimized with a pruning routine. Basically, the neurons that present a constant activation can be interpreted as bias neurons. This way, their input connections could be removed from the network, since they do not affect the outputs. Additionally, the neurons could be also removed, because their activations can be added to the bias of the output neurons. This pruning routine can be designed as the algorithm 6.

5.5 An Example

In this section, the proposed learning algorithm is manually executed to produce a neural network that models the “golf” database, that was previously described. This

Algorithm 6 Pruning algorithm

Require: A neural network $NN = n_i, i = 1, ..., T$, where n_i is a neuron, T is the total of neurons, c_{ij} is a connection from neuron n_i to neuron n_j, a_i is the medium activation of neuron n_i, sd_i is the the standard deviation of the activations of neuron n_i and b_i is the bias of the neuron n_i.
Require: A threshold t.
Ensure: A prunned neural network created from NN.

```
i ← 1
while i ≤ T do
  if sd_i ≤ t then
    Remove all input connections from n_i.
    j ← 1
    while j ≤ T do
      if existsc_ij then
        b_j ← b_j + a_i
        Remove c_ij.
      end if
    end while
  end if
  i ← i + 1
end while
Remove all neurons without any output connection.
```

Figure 5.2: Creation of the output neurons.

execution still considers the 14 training examples listed on the table 5.1.

Based on the number of classes, the algorithm creates two output nodes ("play" and "don't play") as shown in the figure 5.2. So, it checks all base cases and calculates the normalized information gain from splitting the database on each attribute. The results are:

- Outlook: 0.2467;
- Temperature: 0.0000;
- Humidity: 0.0000;
- Windy: 0.0481.

The algorithm selects the "outlook" attribute, because it has the highest normalized information gain (0.2467). Since it is a categorical attribute, three input neurons are created ("sunny", "overcast" and "rainy"), as showed in the figure 5.3.

The "outlook" attribute splits the data set on three subsets. Thus, the algorithm has to be executed for each subset. The first subset is composed by 5 training examples where outlook is "sunny". The normalized information gain is calculated for the remaining attributes, as follows:

- Temperature: 0.0000;
- Humidity: 0.9710;
- Windy: 0.0200;

The "humidity" attribute is selected (0.9710). Because it is a numerical attribute, just one input neuron is created, as showed in the figure 5.4. The algorithm utilizes the C4.5 formula to establish a threshold for the attribute, which is 75% in this case[2]. Then, the algorithm creates two hidden neurons that are activated by the "humidity" input neuron. These hidden neurons represent the threshold. One neuron represents "$humidity \leq 75$" and the other represents "$humidity > 75$", as displayed on the figure 5.5.

Using the threshold, the algorithm creates a database to train the first neuron. The training examples are shown in the table 5.4[3]. The Pocket algorithm with ratchet is utilized to train the neuron.

Then, the algorithm creates another data set to train the second hidden neuron, as listed in the table 5.5. The Pocket algorithm with ratchet is utilized again.

The algorithm concatenates the "outlook" and "humidity" attributes with two additional hidden neurons, as showed in the figure 5.6. These neurons represent the "AND" logical operation. They are trained with the Pocket algorithm with ratchet. A simplified version of the utilized training examples is presented on the table 5.6.

Considering the "$humidity \leq 75$" neuron, the algorithm calculates the normalized information gain for the remaining attributes:

- Temperature: 0.0000;

[2]To facilitate the understanding, the normalization step will be ignored in this explanation. However, it is relevant to observe that all numerical values must be normalized in the unary range, so they can be calculated by the neural network.

[3]The size of the data set is limited to facilitate the demonstration.

Figure 5.3: Creation of “outlook” input neurons.

Figure 5.4: Creation of “humidity” input neuron.

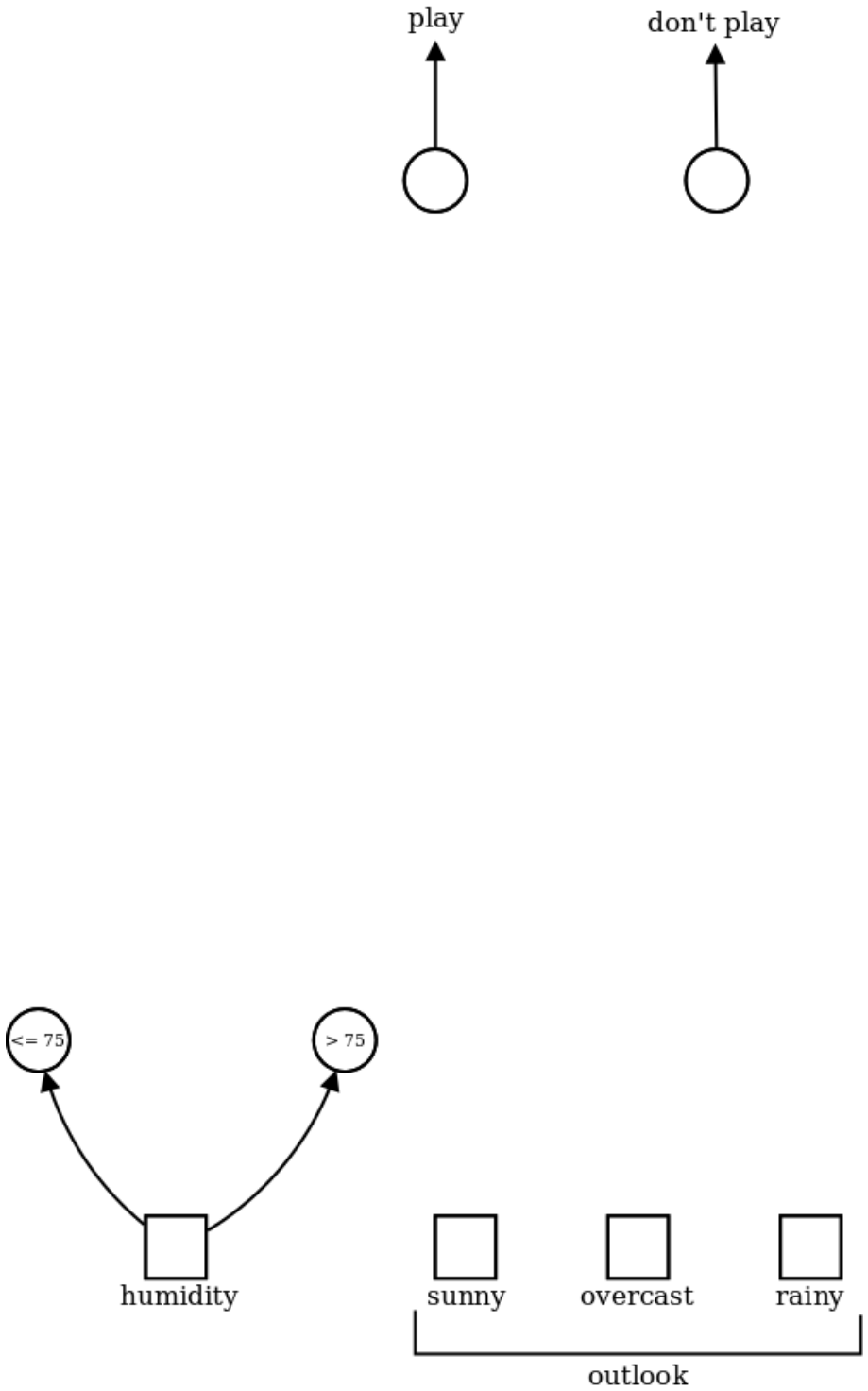

Figure 5.5: Creation of threshold neurons of “humidity” input neuron.

Table 5.4: Training examples for the "*humidity* ≤ 75" neuron.

Training Example	(Bias, x1)	c
X1	(+1, 55)	+1
X2	(+1, 60)	+1
X3	(+1, 65)	+1
X4	(+1, 70)	+1
X5	(+1, 75)	+1
X6	(+1, 80)	-1
X7	(+1, 85)	-1
X8	(+1, 90)	-1
X9	(+1, 95)	-1
X10	(+1, 100)	-1

Table 5.5: Training examples for the "*humidity* > 75" neuron.

Training Example	(Bias, x1)	c
X1	(+1, 55)	-1
X2	(+1, 60)	-1
X3	(+1, 65)	-1
X4	(+1, 70)	-1
X5	(+1, 75)	-1
X6	(+1, 80)	+1
X7	(+1, 85)	+1
X8	(+1, 90)	+1
X9	(+1, 95)	+1
X10	(+1, 100)	+1

Table 5.6: Training examples of "AND" logical operation (simplified version).

Training Example	(Bias, x1, x2)	c
X1	(+1, -1, -1)	-1
X2	(+1, -1, +1)	-1
X3	(+1, +1, -1)	-1
X4	(+1, +1, +1)	+1

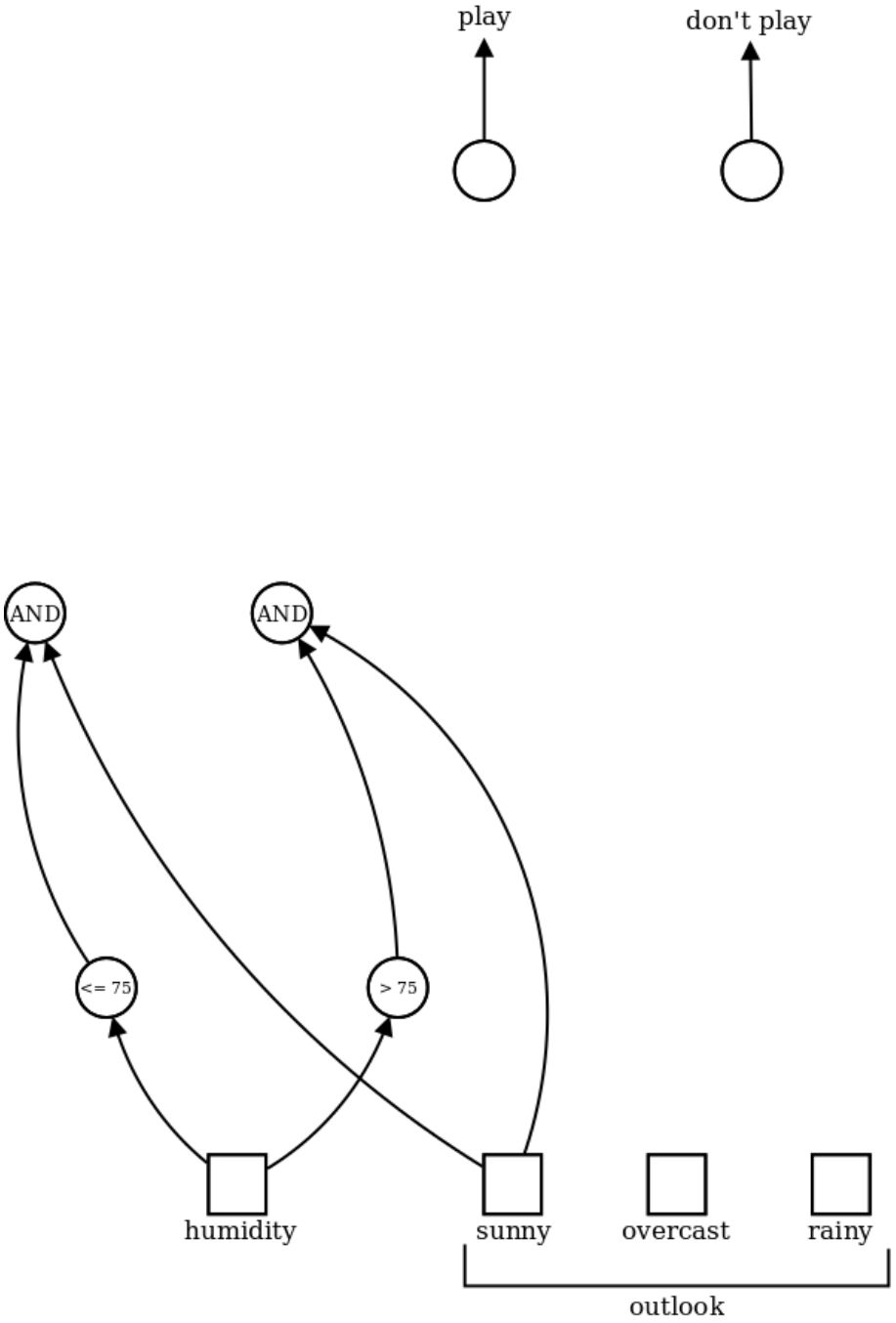

Figure 5.6: Concatenation of “outlook” and “humidity” neurons.

- Windy: 0.0000.

Because the result is zero for the both attributes, the algorithm stops the recursive execution on this node. It connects the "$humidity \leq 75$" neuron to its respective output neuron ("play" class), as shown in the figure 5.7.

Then, the algorithm calculates the normalized information gain for the resting attributes of the "$humidity > 75$" neuron:

- Temperature: 0.0000;
- Windy: 0.0000.

These values describe a final node, so this neuron is connected to its output neuron ("don't play" class), as shown in the figure 5.8.

The algorithm goes to the "overcast" neuron, which has 4 training examples. It calculates the normalized information gain for the remaining attributes, as follows:

- Temperature: 0.0000;
- Humidity: 0.0000;
- Windy: 0.0000.

Since there is no way to keep splitting the subset, the "overcast" neuron is connected to its output neuron ("play" class), as shown in the figure 5.9.

After, the algorithm calculates the normalized information gain considering the 5 resting training examples:

- Temperature: 0.0000;
- Humidity: 0.0000;
- Windy: 0.9710.

Based on these results, the algorithm selects "windy" as the best attribute. Since it is a logical attribute, two input neurons are created: "windy is TRUE" and "windy is FALSE". They are presented in figure 5.10. Two "AND" neurons are utilized to concatenate these inputs to the "rainy" neuron, as shown in figure 5.11.

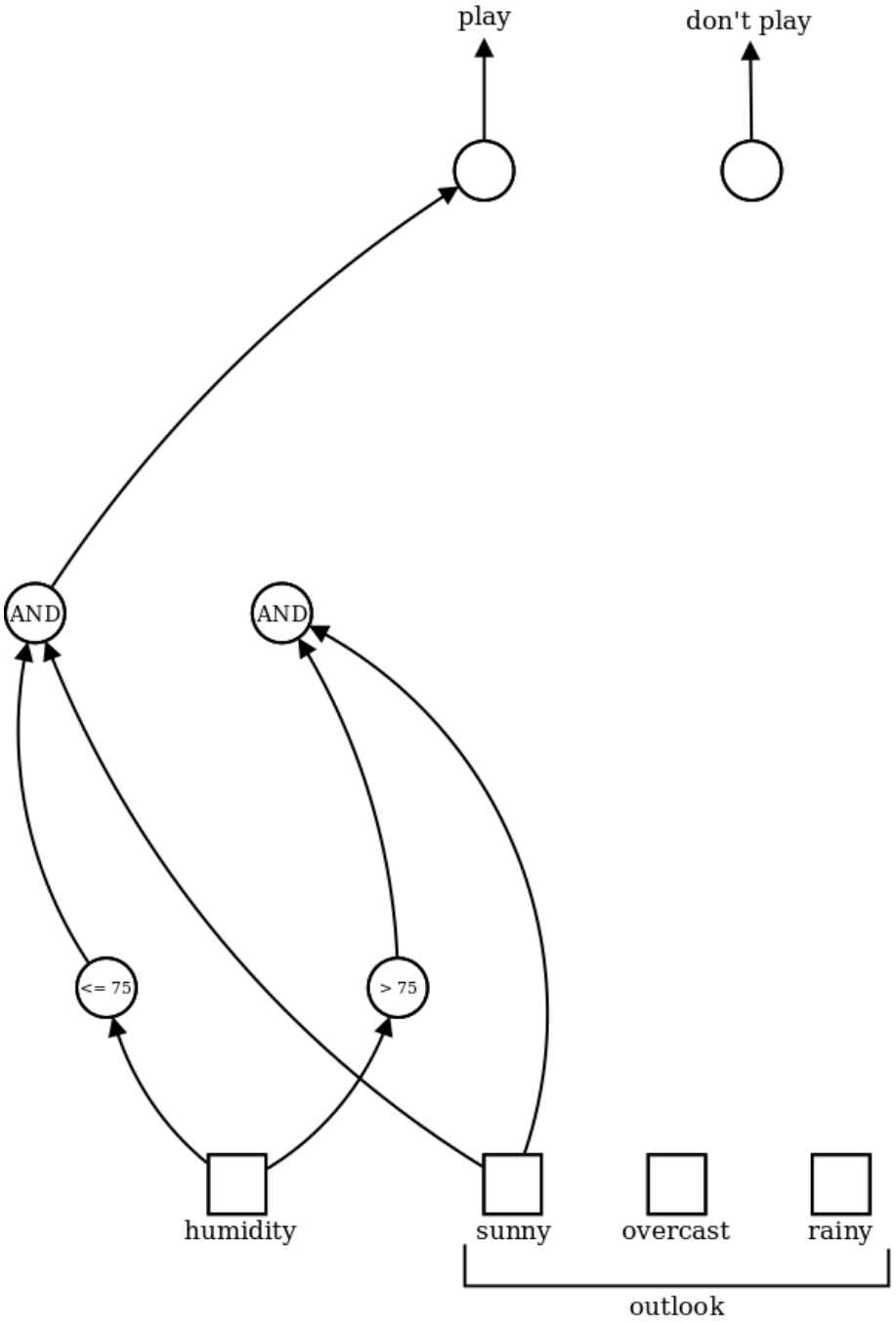

Figure 5.7: Connection of the “humidity” and “play” output neurons.

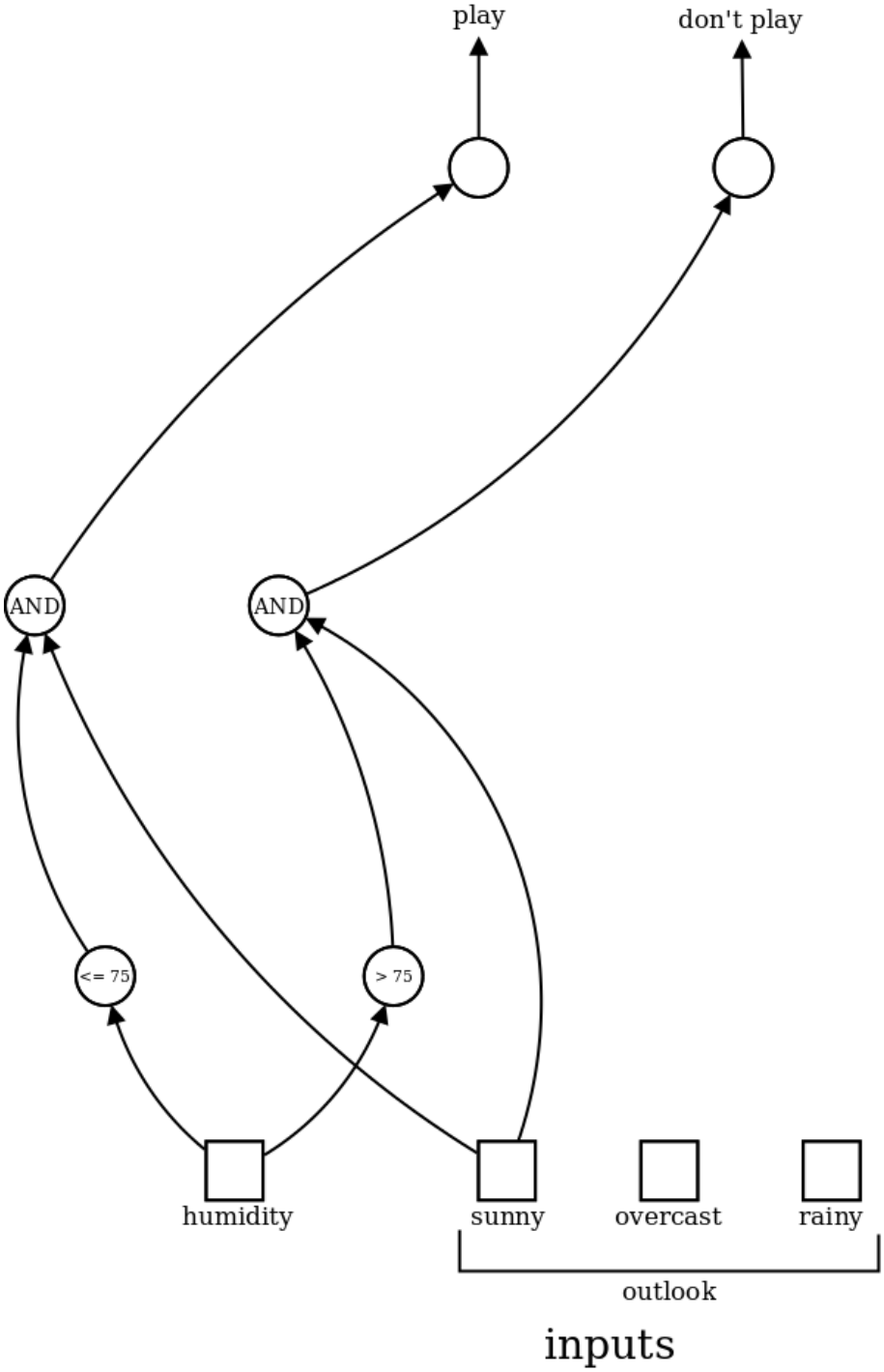

Figure 5.8: Connection of the “humidity” and “don’t play” output neurons.

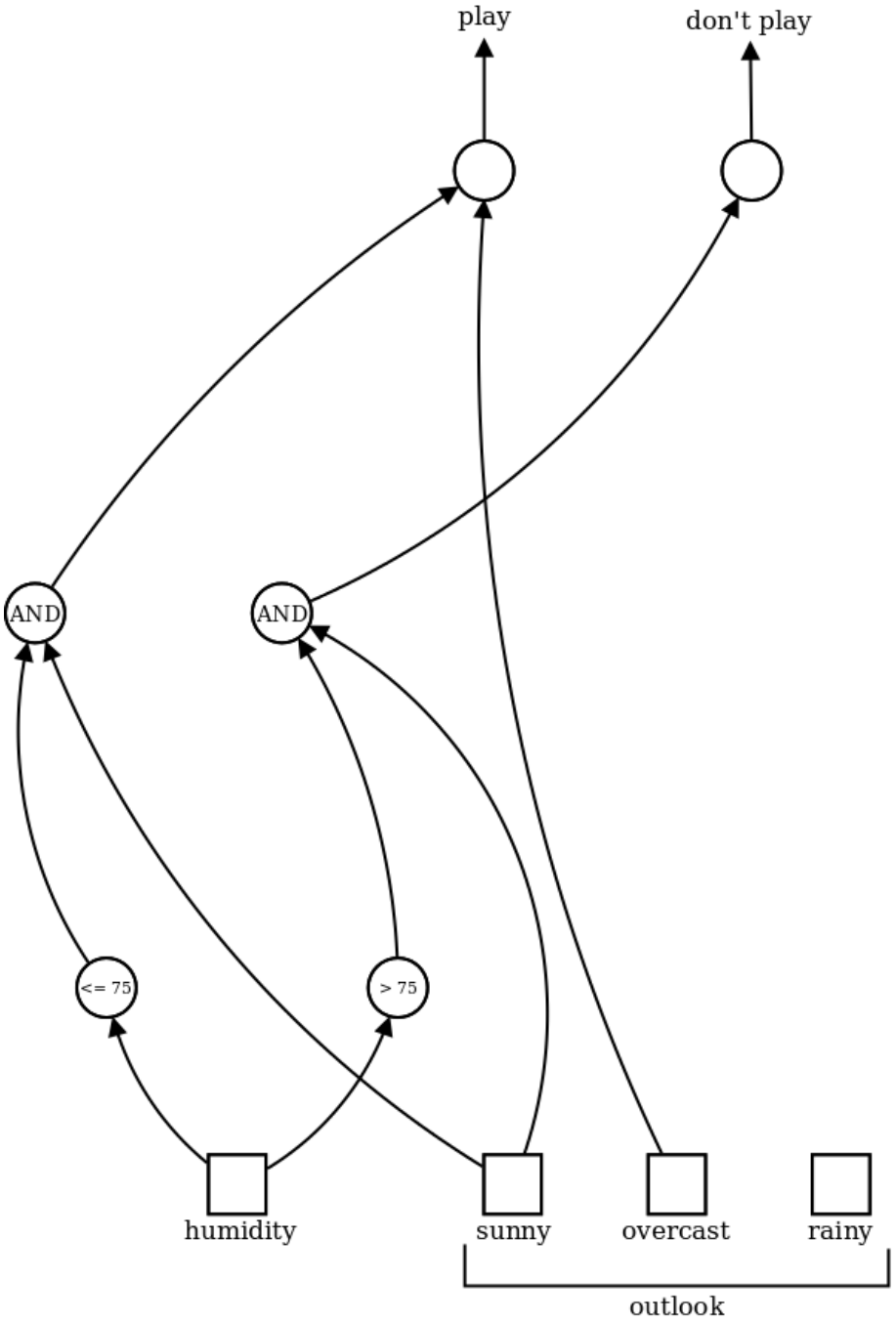

Figure 5.9: Connection of the “overcast” and “play” output neurons.

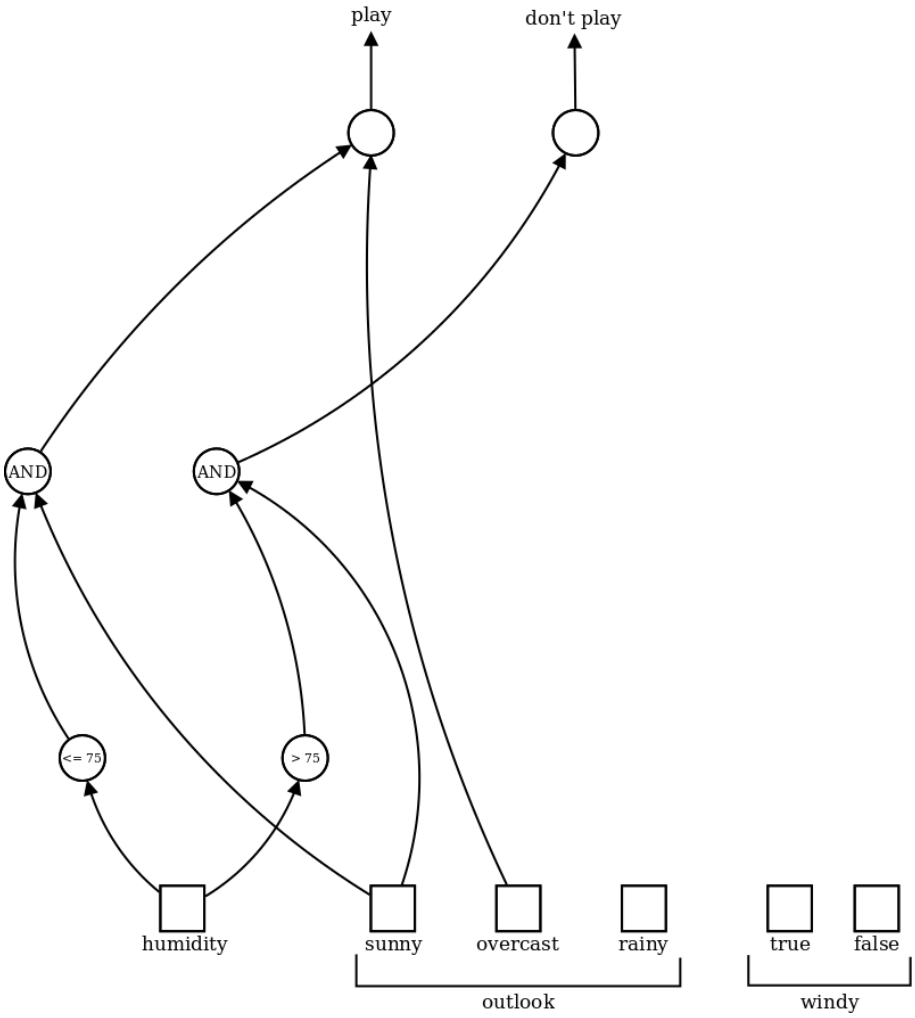

Figure 5.10: Creation of the “windy” input nodes.

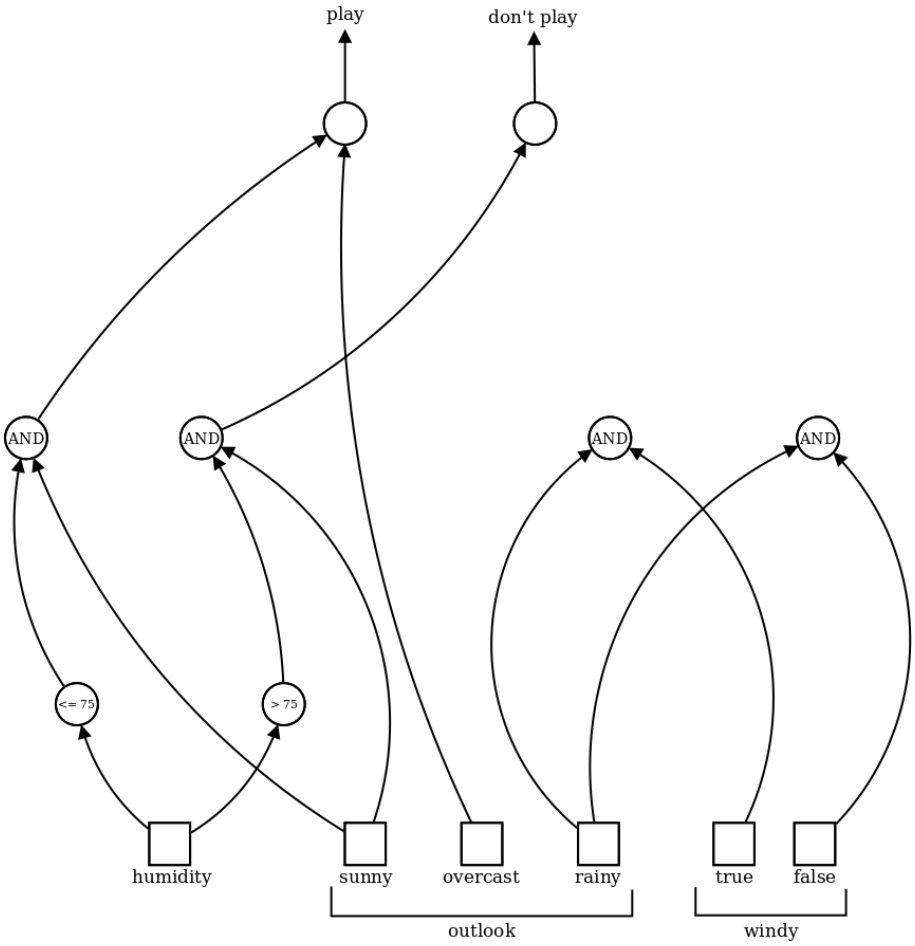

Figure 5.11: Concatenation of the “rainy” and “windy” neurons.

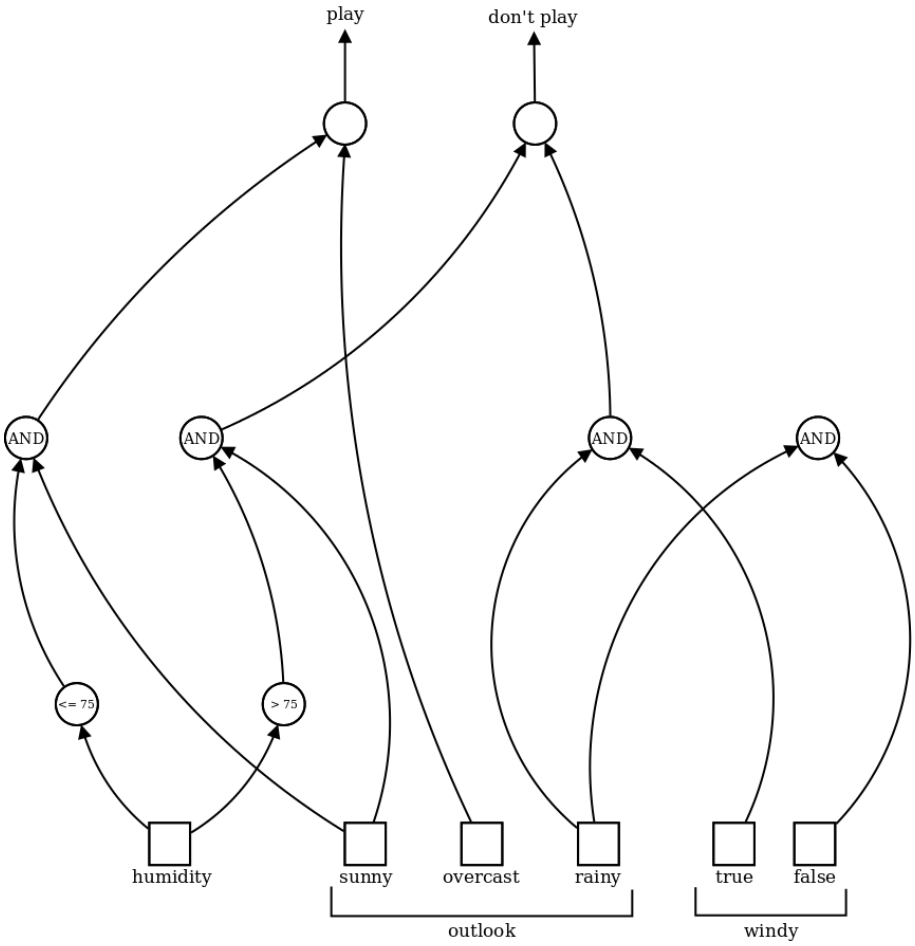

Figure 5.12: Connection of the “windy” and “don’t play” output neurons.

Considering the “windy is TRUE” neuron, the algorithm calculates the normalized information gain for the resting attributes:

- Temperature: 0.0000;
- Humidity: 0.0000;

These results ends the recursive mode, so the “outlook = rainy AND windy is TRUE” neuron is connected to the “don’t play” output neuron, as displayed in the figure 5.12.

Next, the algorithm tries the resting attributes of the “windy is FALSE” neuron:

- Temperature: 0.0000;
- Humidity: 0.0000;

In a similar way, it connects the “overcast = rainy AND windy is FALSE” neuron to the “play” output neuron, as presented on the figure 5.13.

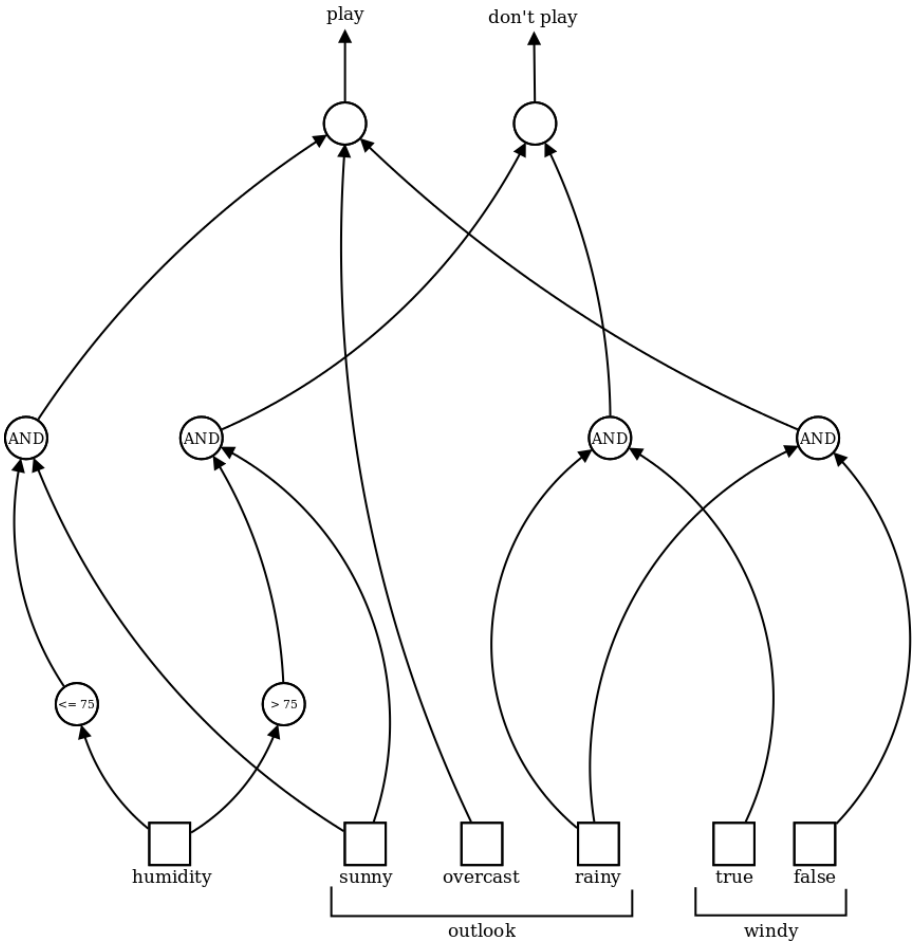

Figure 5.13: Connection of the “windy” and “play” output neurons.

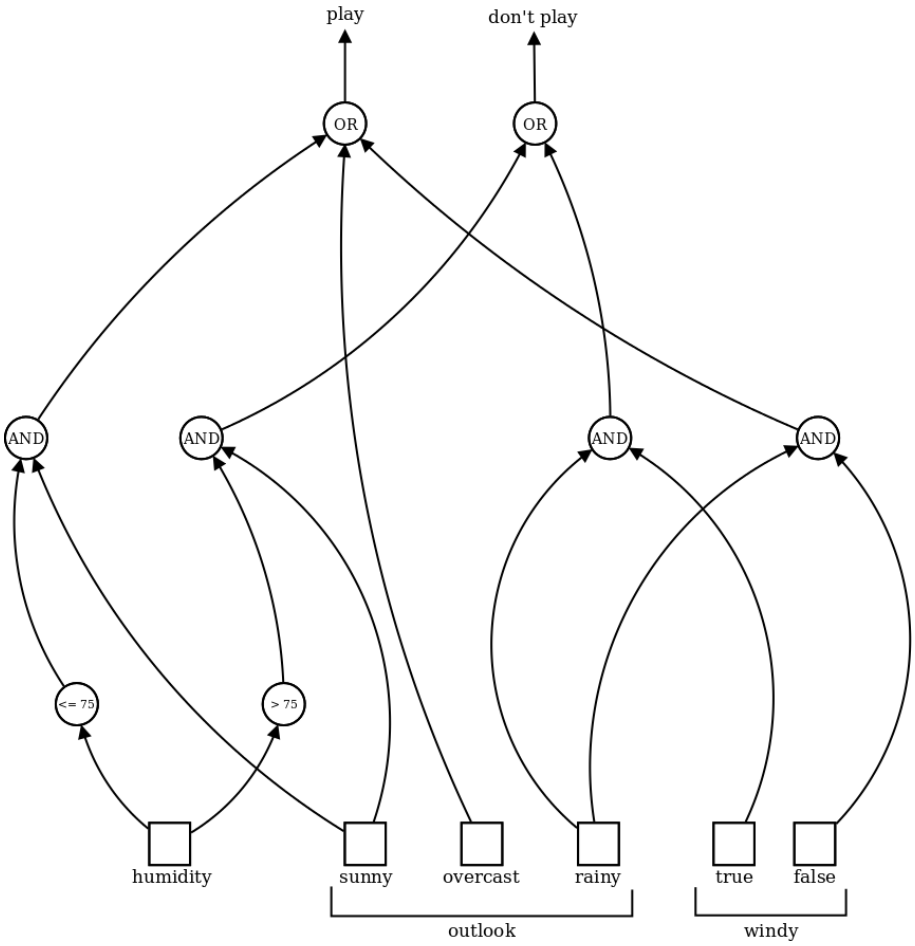

Figure 5.14: Training of the output nodes.

This way, the topology of the neural network is complete. It is relevant to cite that the "temperature" was excluded from the network inputs, since it did not present significant normalized information gains. In a certain way, it is possible to say that the algorithm tries to customize the number of attributes and, consequently, the network size.

The next step is to individually train each output neuron to perform the "OR" logical operation, as shown in the figure 5.14. Again, they are trained with the Pocket algorithm with ratchet. A simplified version of the utilized training examples is presented on the table 5.7.

At this stage, the network is already a fully-functional classifier, with a classification rate equivalent to a similar decision tree. The algorithm orders the network weights in an array. Then, this array is mutated to generate an initial population of related arrays. This population is optimized with the PSO algorithm. The fitness function tries to maximize the accuracy and minimize the error rate. During the execution of the PSO algorithm, new training examples are presented to the potential networks. The best classifier is

Table 5.7: Training examples of "OR" logical operation (simplified version).

Training Example	(Bias, x1, x2)	c
X1	(+1, -1, -1)	-1
X2	(+1, -1, +1)	+1
X3	(+1, +1, -1)	+1
X4	(+1, +1, +1)	+1

selected and the training is complete.

The final network conserves the topology, but it is not possible to determine the logical operations or the numeric thresholds, as shown in the figure 5.15. It happens because the weights are changed during the optimization process.

5.6 Chapter Review

The proposed algorithm was described in this chapter. It is based on three main principles:

- It automatically builds the network topology, inspired on a tree structure.
- It trains each neuron individually.
- It optimizates all the weights using PSO.

The algorithm was manually executed on the "golf" database, a restricted data set that is much utilized on examples involving decision trees. The generated network is very different from a usual MLP and also differs from a decision tree. It can be described as a sparsely connected neural network, where the hidden neurons are not disposed in layers.

Nevertheless, this single example is not solid enough to validate the algorithm. Therefore, the next chapter will describe the methodology that may fully test the raised hypothesis. It describes the experiments, the data sets and the control groups. Additionaly, it presents the metrics that will be utilized to measure the behaviour of the method.

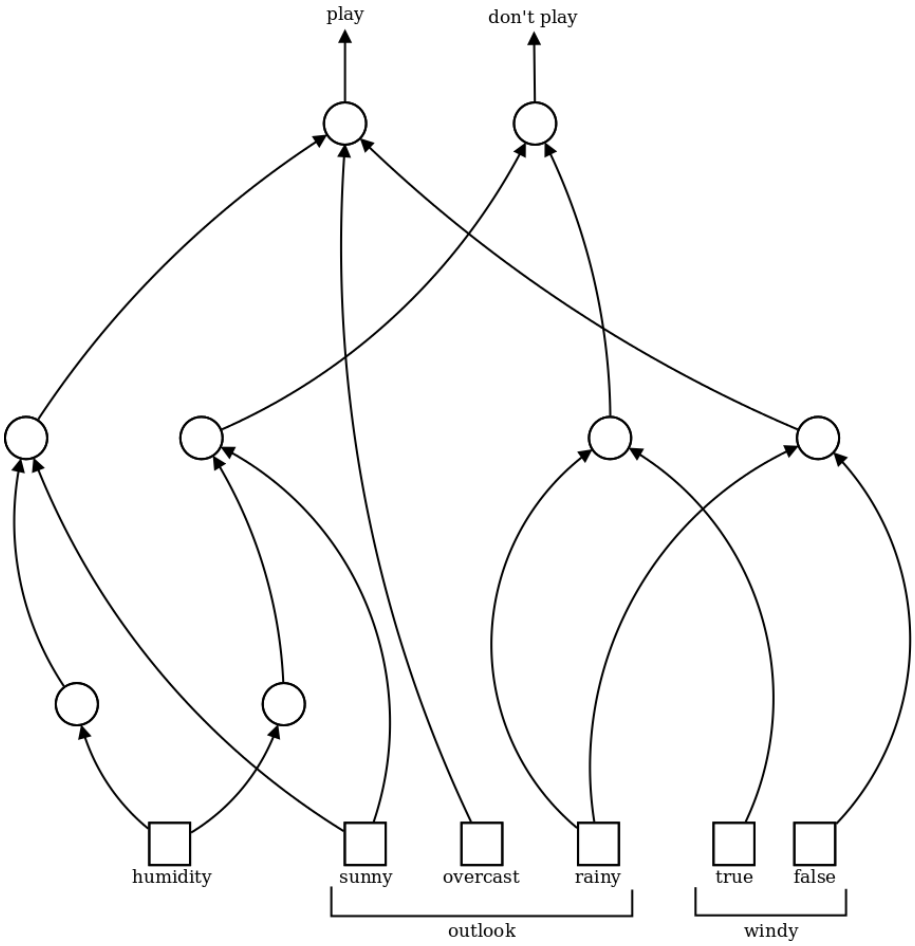

Figure 5.15: Optimized network.

Chapter 6

Experiments

Galileo Galilei established the experimenting as the basis of the scientific method. To prove a hypothesis or a theory, one must establish experiments that reveal the necessary evidences. This "rule" applies to the several domains of science. Particularly, in a machine learning thesis (mainly in classification algorithms), the experiments should answer some questions:

- Does the method work?
- How well does it work?
- When it works?
- How better (or worser) is it when compared to other methods?

To have these answers, it is necessary to:

- Select some data sets to be utilized in the experiments.
- Choose some other ML techniques that will be compared to the proposed method.
- Define a experimenting procedure that will be followed during the tests, in the same way for the proposed method and for the compared techniques.
- Establish some metrics that will measure the performance of the technique during the tests.

The following sections will answer all these questions and establish the experiments that will validate the hypothesis.

6.1 Data Sets

Thirteen (13) databases were selected from the UCI Machine Learning Repository (FRANK; ASUNCION, 2010) to be utilized in the experiments. This process tried to:

- Select data sets with few instances.
- Select data sets with many instances.
- Select data sets with few attributes.
- Select data sets with many attributes.
- Select data sets with only categorical attributes.
- Select data sets with only numerical attributes.
- Select data sets with both categorical and numerical attributes.

The main criterion was to try the proposed method against many types of databases and verify its performance in these situations. The selected data sets are compared in the table 6.1 and listed below:

- Balance Scale;
- Breast Cancer;
- Breast Cancer – Wisconsin;
- Horse Colic;
- Pima Indians Diabetes;
- Heart Disease – Cleveland;
- Heart Disease – Hungarian;
- Hepatitis Domain;
- Iris;
- Mushroom;
- Postoperative Patient Data;

Table 6.1: Data sets.

Name	Attribute Types	# Instances	# Attributes
Balance Scale	Categorical	625	4
Breast Cancer	Categorical	286	9
Breast Cancer - Wisconsin	Categorical	699	10
Horse Colic	Categorical and Numerical	368	27
Pima Indians Diabetes	Numerical	768	8
Heart Disease - Cleveland	Categorical and Numerical	303	14
Heart Disease - Hungarian	Categorical and Numerical	294	14
Hepatitis Domain	Categorical and Numerical	155	19
Iris	Numerical	150	4
Mushroom	Categorical	8,124	22
Post-Operative Patient	Categorical and Numerical	90	8
Lymphography	Categorical	148	18
Lung Cancer	Categorical	32	56

- Lymphography;
- Lung Cancer.

6.1.1 Balance Scale Database

The Balance Scale database was generated to model results obtained from psychological experiments, reported by R.S. Siegler in 1976. The examples are classified as having the balance scale tip to the right, tip to the left, or be balanced.

6.1.2 Breast Cancer Database

The Breast Cancer database was created on the Institute of Oncology of the University Medical Centre (Ljubljana, Yugoslavia) and provided by the physicians Matjaz Zwitter and Milan Soklic. The task involves predicting whether the patients will have recurrence events or not. The data set has 286 instances composed by 9 categorical attributes:

- Age: 10-19, 20-29, 30-39, 40-49, 50-59, 60-69, 70-79, 80-89 or 90-99;
- Menopause: less than 40 years, greater than 40 years or pre-menopause;
- Tumor size: 0-4, 5-9, 10-14, 15-19, 20-24, 25-29, 30-34, 35-39, 40-44, 45-49, 50-54 or 55-59;

- Inv nodes: 0-2, 3-5, 6-8, 9-11, 12-14, 15-17, 18-20, 21-23, 24-26, 27-29, 30-32, 33-35 or 36-39;
- Node caps: yes or no;
- Degree of malignancy: 1, 2 or 3;
- Breast: left or right;
- Breast quadrant: left-up, left-low, right-up, right-low or central;
- Irradiation: yes or no.

6.1.3 Breast Cancer Wisconsin Database

The Breast Cancer Wisconsin database compiles several samples reported by Dr. William H. Wolberg, from University of Wisconsin Hospitals. There are 699 instances composed by 10 categorical atributes:

- Sample code number: id number;
- Clump Thickness: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Uniformity of Cell Size: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Uniformity of Cell Shape: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Marginal Adhesion: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Single Epithelial Cell Size: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Bare Nuclei: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Bland Chromatin: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Normal Nucleoli: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Mitoses: 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10;
- Class: “2” for benign and “4” for malignant.

6.1.4 Horse Colic Database

The Horse Colic database has 27 attributes and 368 instances. It contains both categorical and numerical attributes. It is relevant to remember that 30% of values are missing, which represents a great challenge for the proposed algorithm.

6.1.5 Pima Indians Diabetes Database

The Pima Indians Diabetes Database contains only numerical attributes. There are some missing values, which are encoded as "0" and may represent a challenge during training. All 768 instances come from female patients at least 21 years old of Pima Indian heritage. The attributes are:

- Number of times pregnant;
- Plasma glucose concentration a 2 hours in an oral glucose tolerance test;
- Diastolic blood pressure;
- Triceps skin fold thickness;
- 2-Hour serum insulin;
- Body mass index;
- Diabetes pedigree function;
- Age;
- Class variable.

6.1.6 Heart Disease Databases

Two databases regarding heart disease are utilized in this thesis: Cleveland and Hungarian. Both contains 14 numerical and categorical attributes. The "class" attribute refers to the presence or not of angiographic disease in the patient.

6.1.7 Hepatitis Domain Database

The Hepatitis Domain database has 155 instances and 19 attributes. There are numerical and categorical attributes, with missing values. The "class" attribute refers to the survival or not of the patient with hepatitis disease. The attributes are:

- Class: "die" or "live";
- Age: continuous;
- Sex: "male" or "female";
- Steroid: "yes" or "no";
- Antivirals: "yes" or "no";
- Fatigue: "yes" or "no";
- Malaise: "yes" or "no";
- Anorexia: "yes" or "no";
- Liver big: "yes" or "no";
- Liver firm: "yes" or "no";
- Spleen palpable: "yes" or "no";
- Spiders: "yes" or "no";
- Ascites: "yes" or "no";
- Varices: "yes" or "no";
- Bilirubin: continuous;
- Alk phosphate: continuous;
- Sgot: continuous;
- Albumin: continuous;
- Protime: continuous;
- Histology: "yes" or "no";

6.1.8 Iris Database

The Iris database is one of the most present databases in the machine learning literature. It contains 3 classes (50 instances each), where each class is a type of iris plant. Only one class may be linearly separated from other two. All the 4 attributes are numerical.

6.1.9 Mushroom Database

The Mushroom Database includes 8,124 descriptions of hypothetical samples of 23 gilled mushrooms species in the Agaricus and Lepiota Family. The samples are classified as poisonous or edible. This data set was based on records from The Audubon Society Field Guide to North American Mushrooms (LINCOFF, 1981). It is relevant to notice that the guide affirms there is no simple rule to determine the edibility of a mushroom, what reinforces the difficulty of this task. The data set has 22 categorical attributes, as follows:

- Cap shape: bell, conical, convex, flat, knobbed or sunken;
- Cap surface: fibrous, grooves, scaly or smooth;
- Cap color: brown, buff, cinnamon, gray, green, pink, purple, red, white or yellow;
- Bruises: yes or not;
- Odor: almond, anise, creosote, fishy, foul, musty, none, pungent or spicy;
- Gill attachment: attached, descending, free or notched;
- Gill spacing: close, crowded or distant;
- Gill size: broad or narrow;
- Gill color: black, brown, buff, chocolate, gray, green, orange, pink, purple, red, white or yellow;
- Stalk shape: enlarging or tapering;
- Stalk root: bulbous, club, cup, equal, rhizomorphs, rooted or missing;
- Stalk surface above ring: fibrous, scaly, silky or smooth;
- Stalk surface below ring: fibrous, scaly, silky or smooth;

- Stalk color above ring: brown, buff, cinnamon, gray, orange, pink, red, white or yellow;
- Stalk color below ring: brown, buff, cinnamon, gray, orange, pink, red, white or yellow;
- Veil type: partial or universal;
- Veil color: brown, orange, white or yellow;
- Ring number: none, one or two;
- Ring type: cobwebby, evanescent, flaring, large, none, pendant, sheathing or zone;
- Spore print color: black, brown, buff, chocolate, green, orange, purple, white or yellow;
- Population: abundant, clustered, numerous, scattered, several or solitary;
- Habitat: grasses, leaves, meadows, paths, urban, waste or woods;

6.1.10 Post-Operative Patient Database

The Post-Operative Patient Database was created by Sharon Summers (School of Nursing, University of Kansas) and Linda Woolery (School of Nursing, University of Missouri). It is a small data set that contains only 90 instances describing 8 vital signals of each patient:

- Patient's internal temperature (°C): high (higher than 37°C), mid (between 36°C and 37°C) or low (below 36°C).
- Patient's surface temperature (°C): high (higher than 36.5°C), mid (between 35°C and 36.5°C) or low (below 35°C).
- Oxygen saturation (%): excellent (higher than 98%), good (between 90% and 98%), fair (between 80% and 90%) or poor (below 80%).
- Last measurement of blood pressure (mmHg): high (higher than 130/90 mmHg), mid (between 90/70 mmHg and 130/90 mmHg) or poor (below 90/70 mmHg).
- Stability of patient's surface temperature: stable, mod-stable or unstable.

- Stability of patient's core temperature: stable, mod-stable or unstable.
- Stability of patient's blood pressure: stable, mod-stable or unstable.
- Patient's perceived comfort at discharge: an integer between 0 and 20.

The task is to determine where patients in a post-operative recovery area should be sent: intensive care unit (2 instances), general hospital floor (64 instances) or home (24 instances). It is important to cite that 3 instances have a missing attribute (patient's perceived comfort at discharge).

6.1.11 Lymphography Database

The Lymphography database has 18 categorical attributes and 148 instances. There are no missing values. The "class" attributes refers to the diagnosis of lymph cancer: normal find, metastases, malign lymph or fibrosis.

6.1.12 Lung Cancer Database

The Lung Cancer database describes 3 types of pathological lung cancers. All 56 attributes are categorical and there are missing values. It is a very small data set with only 32 instances. The small size of data set and the absence of some values may provide a great challenge to the proposed method.

6.2 Compared Techniques

It is very important to compare a new machine learning method with other approaches in order to verify its viability. In this thesis, the proposed method is contrasted with two classic AI approaches: decision trees and multilayer perceptrons. It is an ontological choice, since this work proposes a constructive algorithm for creation of neural networks that is heavily inspired in decision trees.

An open-source Java implementation of C4.5 algorithm, called J48 algorithn, was selected to represent decision trees. It is part of the Weka package (GROUP, 2010), which is a collection of machine learning algorithms for data mining tasks. The Weka's implementation was prefered because it also implements the backpropagation learning

algorithm for MLPs. This way, the same application can be utilized to run the tests with both techniques.

6.3 Metrics

6.3.1 Statistical Metrics

Four statistical metrics are utilized in this work to compare the performance of the algorithm against decision trees and MLPs. They are: accuracy, sensitivity, specificity and precision. To define these metrics, a confusion matrix will be utilized as example. Essentialy, it is a matrix that presents the predicted classifications against the actual classifications. The size of matrix is defined by the quantity of possible classes. For example, for a 2-classes problem, that is the confusion matrix:

$$\begin{bmatrix} Actual/Predicted & ClassI & ClassII \\ ClassI & a & b \\ ClassII & c & d \end{bmatrix}$$

Accuracy may be defined as the rate of correct predictions made by the model over the validation data set. On the above example, it would be $accuracy = (a+d)/(a+b+c+d)$.

Sensitivity is also called recall or true positive rate. This metric must be measured for each class. It measures the rate of correct predictions of a class. For example, the sensitivity of the class II can be calculated by $sensitivity = d/(c+d)$.

Specificity, also known as true negative rate, is also a specific class metric. It measures the proportion of negatives that are correctly predicted. The specificity of class II, for example, would be defined by $specificity = a/(a+b)$.

Precision has to be calculated for each class, too. It measures the number of true positives against the total elements of that class. For example, the precision of class II would be $precision = d/(b+d)$.

6.3.2 Computational Cost

As the first four metrics are statistical tools, they can be easily described by mathematical formulas. On the other hand, computational cost is harder to describe. Indeed, what is computational cost? And, more important, how to measure it? Many answers

are correct, but in the context of the present project, computational cost is defined as the "effort" that a computer must take to solve a problem.

In other words, it establishes how "fast" an algorithm creates the model from the training data set. This way, computational cost could be categorized as a temporal metric. However, this "duration time" cannot be considered as an absolute measure, since it will change according to many variables, such as hardware, software, and others. To overcome this problem, the "computational cost" must be turned in a relative metric:

1. Considering a fix platform:
   (a) Hardware:
      i. Processor: Intel Core 2 Duo T5550 (1.83 GHz, 667 Mhz FSB, 2 MB L2 cache);
      ii. Memory: 2 GB DDR2;
      iii. Hard drive: 320 GB 7.200 RPM;
   (b) Software:
      i. Operational System: Ubuntu Linux 10.04 64 bits (kernel: 2.6.32-23-generic);
      ii. Software: Weka (version 3.6.0).
2. Measure (in milliseconds) how much time each classifier takes to create a model from the same training data set.
3. Scale the results in 0-100% range, considering that the larger delay is 100%.
4. Computational cost is the scaled value. Smaller values are better.

Computational cost is a relative metric that is only useful to compare classifiers under a fixed platform. If you change anything in the platform (the processor, for example) all the tests must be executed again. Also, the values may slightly vary, in different executions. Therefore, it is very important to run the same test more times and use the average value. Despite of the fact that computational cost does not provide exact values, it may be very useful to contrast the differences in "speed" among the classifiers under consideration.

6.3.3 Comprehensibility

One of the key motivations of this thesis is to generate more comprehensible neural networks. Indeed, the proposed methodology tries to blend the principles of a "white-

box" technique (decision trees) with the "black-box" neural networks. This way, it is expected to obtain "gray-box" neural networks. However, main questions rise over these assumptions:

- What is comprehensibility?
- How could one measure the comprehensibility of a neural network?
- How more comprehensible the proposed neural networks are?
- How to compare the comprehensibility of different models (decision trees, neural networks and the proposed method)?

Zhi-Hua Zhou defines comprehensibility as the ability of a data mining algorithm to produce patterns understandable to human beings (ZHOU, 2005). It is a very interesting definition, because it highlights a crucial aspect of comprehensible models: human beings may understand the knowledge that is embedded in the model. Without this understanding, the model is just a "black-box", which only provides answers without further explanations.

Comprehensibility is an essential feature in ML algorithms. In fact, Ryszard S. Michalski reinforces its importance on his comprehensibility principle:

> *"The results of computer induction should be symbolic descriptions of given entities, semantically and structurally similar to those a human expert might produce observing the same entities. Components of these descriptions should be comprehensible as single 'chunks' of information, directly interpretable in natural language, and should relate quantitative and qualitative concepts in an integrated fashion." (MICHALSKI, 1983)*

Additionaly, Mark William Craven and Jude W. Shavlik listed several reasons that justify the importance of this criterion (CRAVEN; SHAVLIK, 1995):

- Validation: The designers and end-users must know how a system arrives at its decisions, so they can rely on its performance.
- Discovery: If the representations created by the system are comprehensible, then human beings can review it to make new scientific discoveries.

- Explanation: In some domains, it is desirable to explain the classifications of individual input patterns.
- Improving generalization: Representations that can be understood and analyzed may provide ideas to the creation of better models.
- Refinement: Intelligent systems can be used by researchers to refine approximately-correct domain theories, so it is important that these models can be expressed in a comprehensible manner.

Despite its importance, comprehensibility is a subjective criterion. No author has established a definitive way to quantify how comprehensible an intelligent system is. A shortcut approach was proposed by Tuve Löfström, Ulf Johansson and Lars Niklasson (LÖFSTRÖM; JOHANSSON; NIKLASSON, 2004). Based on the sense that comprehensibility tries to capture if a model is easy enough to understand, they suggested that the complexity can be seen as its opposite. This way, it is possible to indirectly quantify the comprehensibility of a system by quantifying its complexity. In other words, less complex systems tend to be more comprehensible.

This is a very interesting observation, because complexity is easier to measure. For example, the complexity of a decision tree can be determined by the number of internal nodes and utilized symbols (LÖFSTRÖM; JOHANSSON; NIKLASSON, 2004). In a similar way, the complexity of a neural network could be calculated from the number of hidden neurons and connections.

Nevertheless, even if it is possible to measure the complexity of a model, how to compare measurements between differente models? The complexity of decision trees, traditional MLPs and the proposed neural networks is measured with different parameters and formulas. So, they cannot be utilized as a comparation standard. It is necessary to obtain a more generic formula.

Andrey Nikolaevich Kolmogorov, a notorious russian mathematician, developed a theory about how to measure the computational complexity of objects (Kolmogorov, A. N., 1965) [1]. Indeed, the so-called "Kolgomorov complexity" (a.k.a. "algorithmic complexity") offers the theoric bases to measure the computational resources needed to specify an object, such as a piece of text or a string. Volker Nannen has summarized

[1]Besides Andrey Kolmogorov, the concept was individually developed (with different motivations) by Ray Solomonoff (SOLOMONOFF, 1964)(SOLOMONOFF, 1964b) and Gregory Chaitin (CHAITIN, 1966)(CHAITIN, 1969). However, Kolmogorov was the main responsible by the consolidation of the theory and by the study of computational complexity.

the Kolgomorov complexity $C(s)$ of any binary string $s \in \{0,1\}^n$ as the length of the shortest computer program s^* that can produce this string on the UTM (*Universal Turing Machine*) and then halt (NANNEN, 2010). It means that $C(s)$ bits are necessary to encode the string s on UTM [2].

Lance Fortnow presented a very interesting example of use of Kolgomorov complexity (FORTNOW, 2001). Consider three strings:

010101010101010101010101
110100110010110100101100
100111011101011100100110

Despite all being 24-bit binary strings, their complexities are very different. The first string can be described as a 24-bit binary string with a 1 in position n if n is odd. On the other hand, the second string can be defined as a 24-bit binary string with a 1 in position n if the binary representation of n has an odd quantity of 1's. However, the third string cannot be described succinctly, so it is necessary to list its contents verbatim.

This simple example illustrates well the power of the theory. Any object that can be digitally represented (e.g. strings, images, programs) could have its complexity measured in an absolute units (bits). Nonetheless, the Kolgomorov complexity is just theorical and cannot be computed. It happens because no algorithm can predict if every program will halt or not, what causes that it is impossible to predict the output of every program (TURING, 1936). Even if there is a known short program that produces the string, there are always another shorter and unknown programs which are impossible to predict the output of if they will even halt.

Although it is not computable, the theory provides the foundations for practical metrics. This way, it is possible to establish some heuristic and estimate the complexity of a program. The results are not absolute, because of the above explanation. Anyway, they are useful because they provide hints about the complexity criterion.

In this thesis, the complexity of decision trees and neural networks will be measured in the following way. A common grammar was defined to represent both models:

model := *rule* [**else** *rule*]
rule := **if** *expression* **then** *output*

[2] It is important to comment that the UTM is a theoric computer, that does not need to be defined. Once any UTM can be implemented on another UTM, the size of a program would only change by the addition of the implementation of the host UTM (a constant).

expression := *relational_expression* [*logical_operator expression*]
relational_expression := *term relational_operator* {*number*|*logical_state*}*
relational_operator := =|<=|>
logical_operator := **and**|**or**
term := *input*|*activation_function*
activation_function := **act**(*sum*)
sum := *product*[+ *sum*]
product := *value* * *value*
value := *input*|*number*|*activation_function*
input := *letter*{*letter*|*digit*|*punctuation*}
number := {*digit*}*[.{*digit*}*]
logical_state := **true**|**false**
output := **class is** *class_name*
class_name := *letter*{*letter*|*digit*|*punctuation*}
letter := {**a**|**b**|**c**|**d**|**e**|**f**|**g**|**h**|**i**|**j**|**k**|**l**|**m**|**n**|**o**|**p**|**q**|**r**|**s**|**t**|**u**|**v**|**x**|**y**|**z**}*
digit := {**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**}*
punctuation := {_}*

Using the above grammar, the decision tree previously created for the "golf" database would be:

if outlook = sunny and humidity <= 75 or outlook = overcast or outlook = rainy and windy = false then class is play else if outlook = sunny and humidity > 75 or outlook = rainy and windy = true then class is dont_play

And the neural network previously created for the same database would be[3]:

if act(act(act(humidity * W) * W + sunny * W) * W + overcast * W + act(rainy * W + windy_false * W) * W) > 0.5 then class is play else if act(act(act(humidity * W) * W + sunny * W) * W + act(rainy * W + windy_true * W) * W) > 0.5 then class is dont_play

A normal 3-layers MLP, like the one in figure 6.1, would be:

[3]In these examples, W means the numerical value of a connection weight.

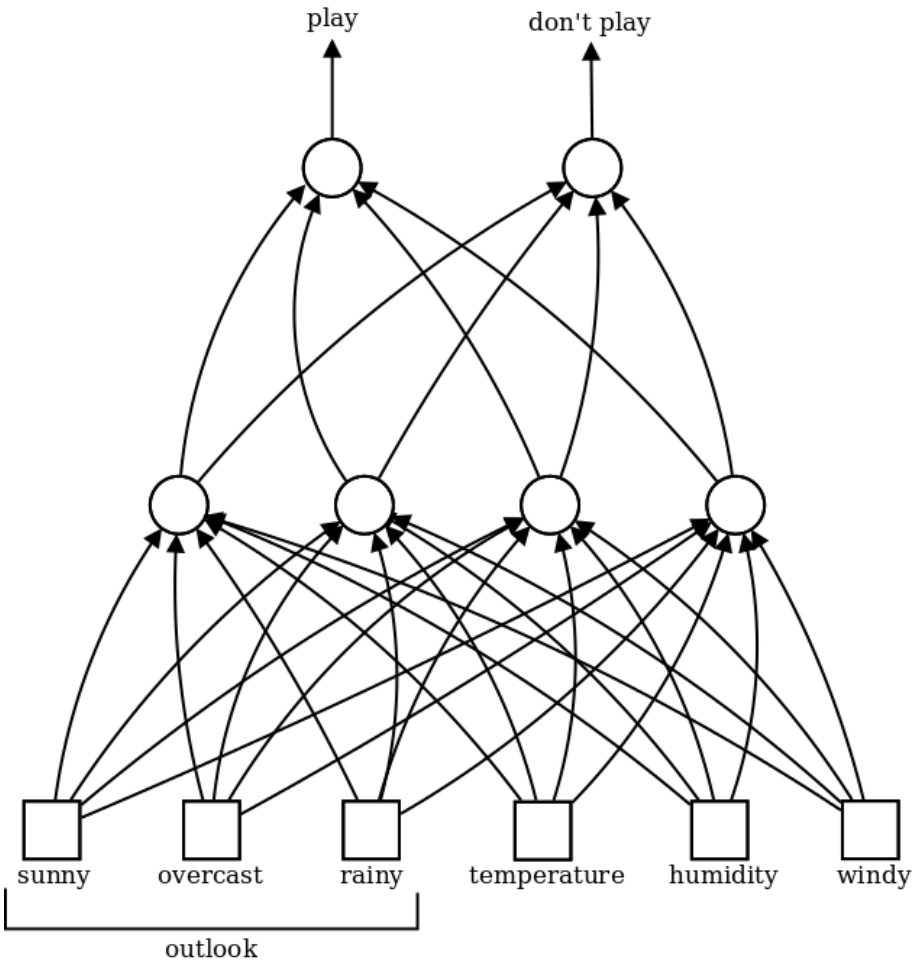

Figure 6.1: A multilayer perceptron that can solve the "golf" problem.

if act(act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W) + act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W) + act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W) + act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W)) > 0.5 then class is play else if act(act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W) + act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W) + act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W) + act(sunny * W + overcast * W + rainy * W + temperature * W + humidity * W + windy * W)) > 0.5 then class is dont_play

The complexity of a model will be defined by the absolute number of elements that are utilized to describe it. This way, the complexities of the above models are:

- Decision tree: 45;
- Proposed neural network: 69;
- Traditional MLP: 215.

However, the complexity is not the final objective of this measure. The key point is to establish the comprehensibility of the models. In this thesis, it is considered that the comprehensibility is inversely proportional to the complexity:

$$comprehensibility = -10 * \ln complexity + 110.986122887$$

Thus, the comprehensibility of a model it is a fraction of 100. A very simple model (e.g. a rule with an unique class[4]) will be 100% comprehensible. More complicated models will have less comprehensibility. This metric follows the *lex parsimoniae* and favors simpler models.

Using the above formula, the comprehensibility of the previously cited models would be:

- Decision tree: 73%;
- Proposed neural network: 69%;
- Traditional MLP: 57%.

6.4 Experiments

The basic structure for each experiment is:

1. Let D be a database under consideration.
2. Split D in 2 random stratified subsets named K_{train} and K_{test}, where K_{train} has 66.67% of all instances and $|K_i \cap K_j| = \emptyset$.
3. For each classifier under consideration, repeat 10 times:

[4]A rule with an unique class is represented by 3 elements: "*class is X*".

(a) Build a different instance of the classifier using K_{train} as the training set.

(b) Test the classifier in the K_{test} set, using the established metrics.

6.5 Chapter Review

This chapter described the methodology that will be conducted to confirm the raised hypothesis and the validity of the proposed algorithm. It described the experiments that will be executed in several databases to check the behaviour of this new learning algorithm in different contexts.

The results will be always compared to decision trees and MLPs, once these both techniques are extremely involved to the present work. The performance of the method will be measured by six metrics, consisting of four statistical tools (accuracy, sensitivity, specificity and precision) and two relative metrics that were created for this project (computational cost and comprehensibility).

Chapter 7

Results and Discussion

The present chapter presents the results of this work. All the results are analyzed in quantitative and qualitative ways. Section 1 presents how the proposed algorithm was implemented and the main aspects of the final software, including how to use it in further experiments.

Section 2 lists statistical results of the classifier in several data sets. As control, the same data sets were classified with other traditional techniques. These values are then compared in both quantitative and qualitative ways.

Section 3 benchmarks the performance of the final software in a controlled environment. Again, the values are compared to traditional algorithms under the same general conditions.

Section 4 analyzes the comprehensibility of the generated models and how better (or worser) they are when compared to other methods.

7.1 Implementation

The algorithm proposed in the previous chapters was implemented as a module of the Weka platform, using the Java programming language. The source code is freely available in the appendix A, under the terms of the Apache Commons license.

Figure 7.1 shows the Weka data mining tool running a test with the proposed classifier. As the algorithm was developed as a normal module of this platform, it can be utilized in the same way of other common tools (as J48, for example).

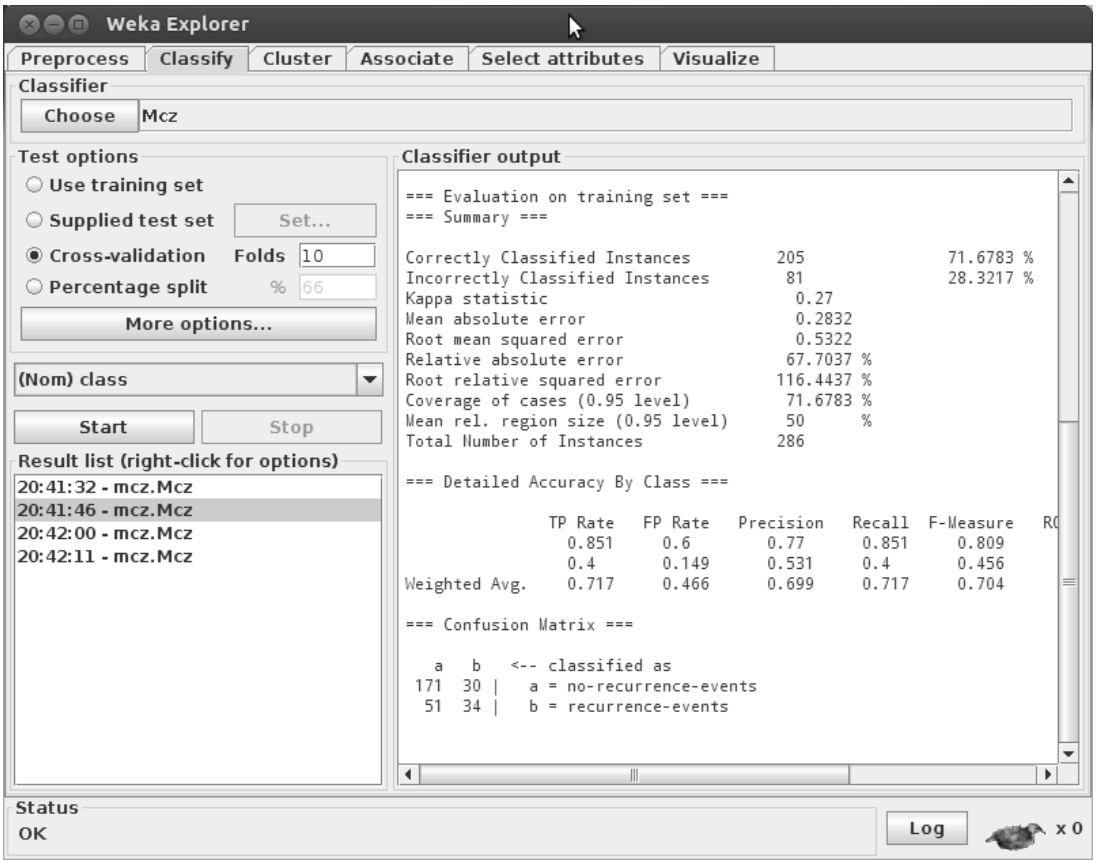

Figure 7.1: User interface of the implemented classifier during a test.

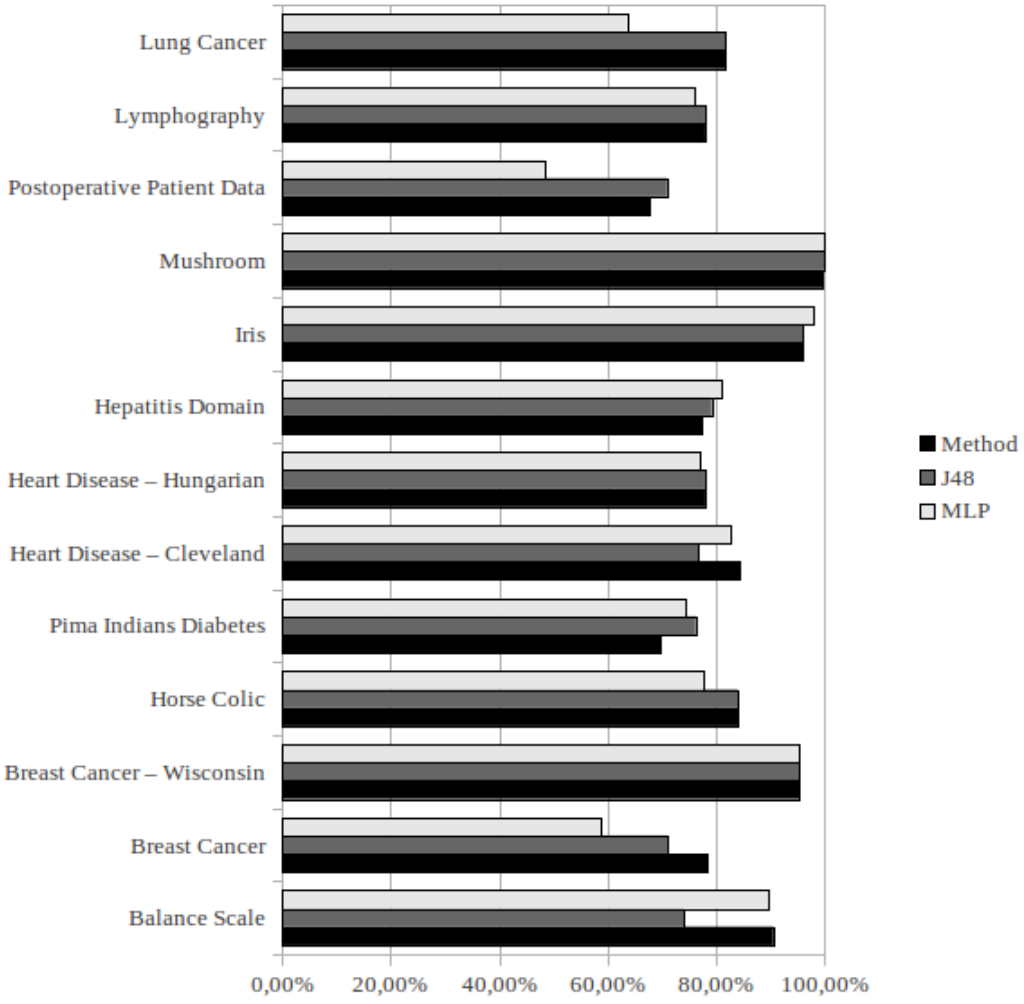

Figure 7.2: Accuracy of the proposed method compared to other techniques.

7.2 Statistical Results

Four statistical metrics were utilized in this work: accuracy, sensitivity, specificity and precision. They were utilized to quantify the classification performance of the proposed algorithm. As control, the same metrics were calculated with two traditional classifiers. The compared results are listed in the table 7.1. Furthermore, the same information is presented as a comparative chart in the figures 7.2, 7.3, 7.4 and 7.5.

The obtained results indicate that the proposed algorithm is an effective classifier, as shown in table 7.2. The T-test was applied over the results to confirm if the performance of the proposed classifier is statistically similar to those obtained with the control classifiers.

Table 7.1: Statistical results of the proposed method compared to traditional classifiers.

Classifier	Database	Accuracy	Sensitivity	Specificity	Precision
Method	Balance Scale	90.56%	90.60%	91.70%	84.60%
	Breast Cancer	78.35%	42.30%	78.40%	75.60%
	Breast Cancer - Wisconsin	95.38%	92.70%	95.40%	95.50%
	Horse Colic	84.00%	78.00%	84.00%	84.00%
	Pima Indians Diabetes	69.73%	51.80%	69.70%	67.60%
	Heart Disease - Cleveland	84.47%	84.70%	84.50%	84.70%
	Heart Disease - Hungarian	78.00%	71.30%	78.00%	79.40%
	Hepatitis Domain	77.36%	50.70%	77.40%	75.40%
	Iris	96.08%	97.90%	96.10%	96.50%
	Mushroom	99.78%	99.80%	99.80%	99.80%
	Postoperative Patient Data	67.74%	34.30%	67.70%	60.40%
	Lymphography	78.00%	79.10%	78.00%	74.90%
	Lung Cancer	81.82%	51.50%	81.80%	85.50%
J48	Balance Scale	74.06%	83.90%	74.10%	73.80%
	Breast Cancer	71.13%	37.10%	71.10%	67.40%
	Breast Cancer - Wisconsin	95.38%	95.40%	95.40%	95.40%
	Horse Colic	84.00%	78.00%	84.00%	84.00%
	Pima Indians Diabetes	76.25%	65.80%	76.20%	75.60%
	Heart Disease - Cleveland	76.70%	76.80%	76.70%	77.10%
	Heart Disease - Hungarian	78.00%	72.80%	78.00%	78.00%
	Hepatitis Domain	79.25%	42.70%	79.20%	78.30%
	Iris	96.00%	98.00%	96.00%	96.00%
	Mushroom	100.00%	0.00%	100.00%	100.00%
	Postoperative Patient Data	70.97%	29.00%	71.00%	50.40%
	Lymphography	78.00%	83.40%	78.00%	80.00%
	Lung Cancer	81.82%	51.50%	81.80%	85.50%
MLP	Balance Scale	89.62%	92.40%	89.60%	89.00%
	Breast Cancer	58.76%	42.20%	58.80%	64.20%
	Breast Cancer - Wisconsin	95.38%	94.90%	95.40%	95.40%
	Horse Colic	77.60%	72.40%	77.60%	77.30%
	Pima Indians Diabetes	74.33%	70.70%	74.30%	75.60%
	Heart Disease - Cleveland	82.52%	82.60%	82.50%	83.00%
	Heart Disease - Hungarian	77.00%	70.50%	77.00%	77.40%
	Hepatitis Domain	81.13%	69.60%	81.10%	84.60%
	Iris	98.04%	98.80%	98.00%	98.10%
	Mushroom	100.00%	0.00%	100.00%	100.00%
	Postoperative Patient Data	48.39%	36.90%	48.40%	51.50%
	Lymphography	76.00%	80.30%	76.00%	75.10%
	Lung Cancer	63.64%	44.70%	63.60%	63.60%

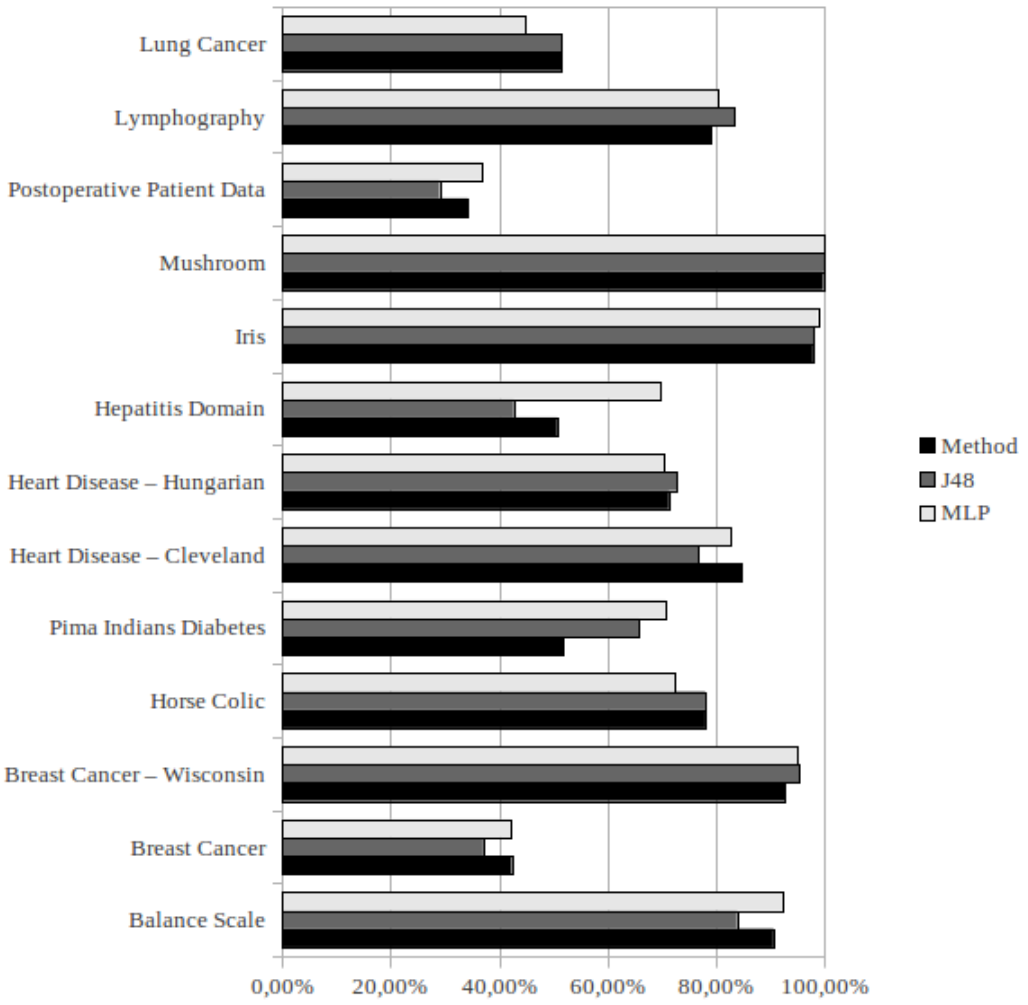

Figure 7.3: Sensitivity of the proposed method compared to other techniques.

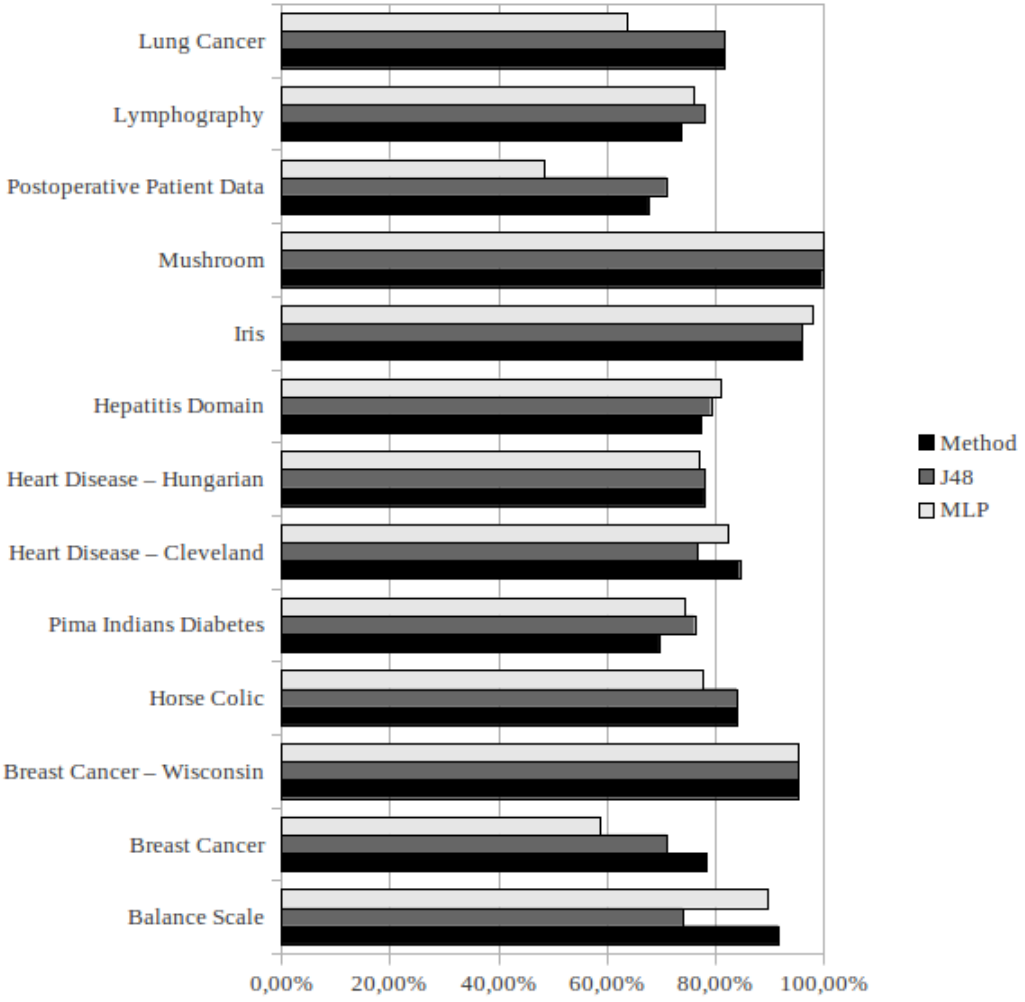

Figure 7.4: Specificity of the proposed method compared to other techniques.

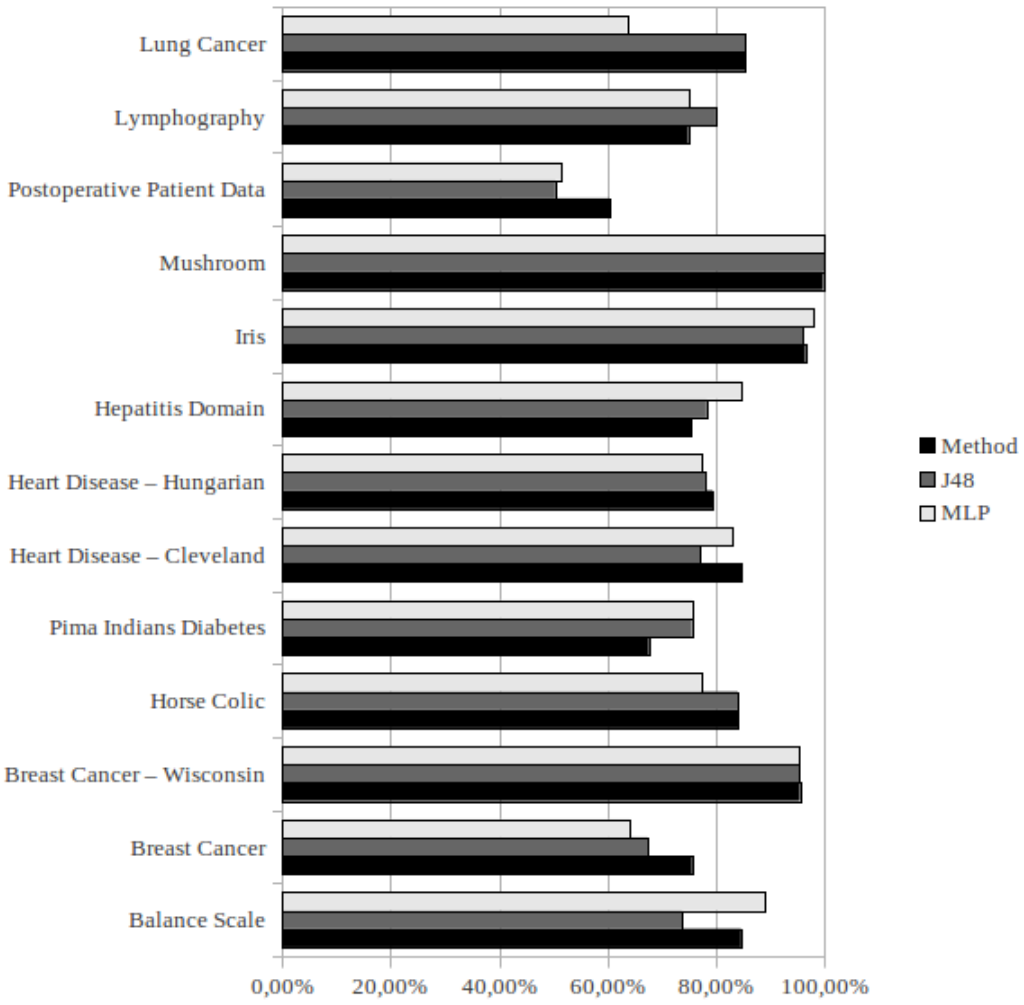

Figure 7.5: Precision of the proposed method compared to other techniques.

Table 7.2: Average results of the compared methods.

Metric	Proposed Method	J48	MLP
Accuracy	83.17%	81.66%	78.65%
Sensitivity	71.13%	70.34%	73.54%
Specificity	82.95%	81.65%	78.64%
Precision	81.84%	80.12%	79.60%

Table 7.3: Application of T-test over the statistical results.

Metric	T-test for J48 and Proposed Method	T-test for MLP and Proposed Method
Accuracy	0.396	0.894
Sensitivity	0.088	0.280
Specificity	0.332	0.843
Precision	0.359	0.441

The results of this test are shown in table 7.3. Considering that the test was done with 24 degrees of freedom and with 95% of confidence, it is possible to assume that the results are statistically similar. It happens because all values are inferior to 1.711, which is the default T-value for these conditions.

This is a very important observation, because it demonstrates that the proposed approach can generate effective classifiers, reinforcing the importance of the raised hypothesis. The great majority of learning algorithms for neural networks are based in evolving weights (e.g. backpropagation, self-organizing maps, Hebbian networks, and others). The neural networks generated by these approaches are fully connected, so each neuron is connected to several nodes. The main topology of the networks is usually replicated in several cases and is not considered a strong aspect of the model.

The results demonstrate that sparse neural networks can present the same performance with less complexity. These results reinforce the importance of topology construction in neural networks. It proves that refined topologies designed over the data instances can compensate fewer connections.

7.3 Computational Cost

An important aspect of any classifier is its computational performance. In other words, can the classifier be applied in a feasible time window? Naturally, this is a very tricky question, since this kind of benchmark depends on several factors, such as hardware,

operational system, programming language, implementation strategy, and others.

In this work, a simple strategy is applied to solve this question. All the experiments were conducted in the same environment: hardware, operational system, programming language and platform (Weka). With this approach, the duration of each test could be measured in a fair way. Thus, it is possible to compare the results of the proposed algorithm against those of the control classifiers. The duration (in seconds) of each test is listed in the table 7.4.

It is important to reinforce that these results depend of the environment, which means that they will change with different hardware or software. However, when the results are observed inside a same context, it is possible to determine if the analyzed algorithm presents a reasonable computational performance.

The results demonstrate that the proposed algorithm performed worser than the other classifiers. This kind of result was expected, since the implementation of the algorithm was not done aiming this kind of performance. Also, this is the first implementation, without optimization of the code or its performance. On the other hand, the control classifiers were implemented in a more sophisticated way, since they are usually utilized in the field.

Anyway, despite of that, the results prove that the proposed classifier can generate a model in a feasible time, which is very important. The implemented code can be strongly optimized in future, generating even better performance.

7.4 Comprehensibility

One of the main motivations of this work was to generate more comprehensible neural networks. To achieve that, the proposed algorithm aimed the creation of sparse nets, where each neuron has few connections to other nodes. It was expected that less connections could simplify the models.

As stated in previous chapters, the comprehensibility is a subjective criterion, which makes more difficult to compare different approachs or models. To overcome that, a new metric was proposed. It is based in the assumption that compreensibility is inversely proportional to complexity, which means that less complex models are generally more comprehensible.

Table 7.5 lists the comprehensibility analysis of each generated model. The proposed

Table 7.4: Computational cost of the proposed method compared to traditional classifiers.

Classifier	Database	Computational Cost
Method	Balance Scale	528.71s
	Breast Cancer	762.96s
	Breast Cancer - Wisconsin	981.34s
	Horse Colic	954.71s
	Pima Indians Diabetes	7061.52s
	Heart Disease - Cleveland	954.98s
	Heart Disease - Hungarian	760.11s
	Hepatitis Domain	1480.55s
	Iris	623.83s
	Mushroom	7065.94s
	Postoperative Patient Data	627.14s
	Lymphography	255.64s
	Lung Cancer	10.3s
J48	Balance Scale	0,04s
	Breast Cancer	0,01s
	Breast Cancer - Wisconsin	0,01s
	Horse Colic	0,01s
	Pima Indians Diabetes	0,02s
	Heart Disease - Cleveland	0,01s
	Heart Disease - Hungarian	0,01s
	Hepatitis Domain	0,01s
	Iris	0,00s
	Mushroom	0,12s
	Postoperative Patient Data	0,17s
	Lymphography	0,01s
	Lung Cancer	0,00s
MLP	Balance Scale	0,46s
	Breast Cancer	4,49s
	Breast Cancer - Wisconsin	0,73s
	Horse Colic	8,57s
	Pima Indians Diabetes	0,85s
	Heart Disease - Cleveland	1,50s
	Heart Disease - Hungarian	1,28s
	Hepatitis Domain	0,43s
	Iris	0,12s
	Mushroom	421,95s
	Postoperative Patient Data	0,34s
	Lymphography	1,22s
	Lung Cancer	2,56s

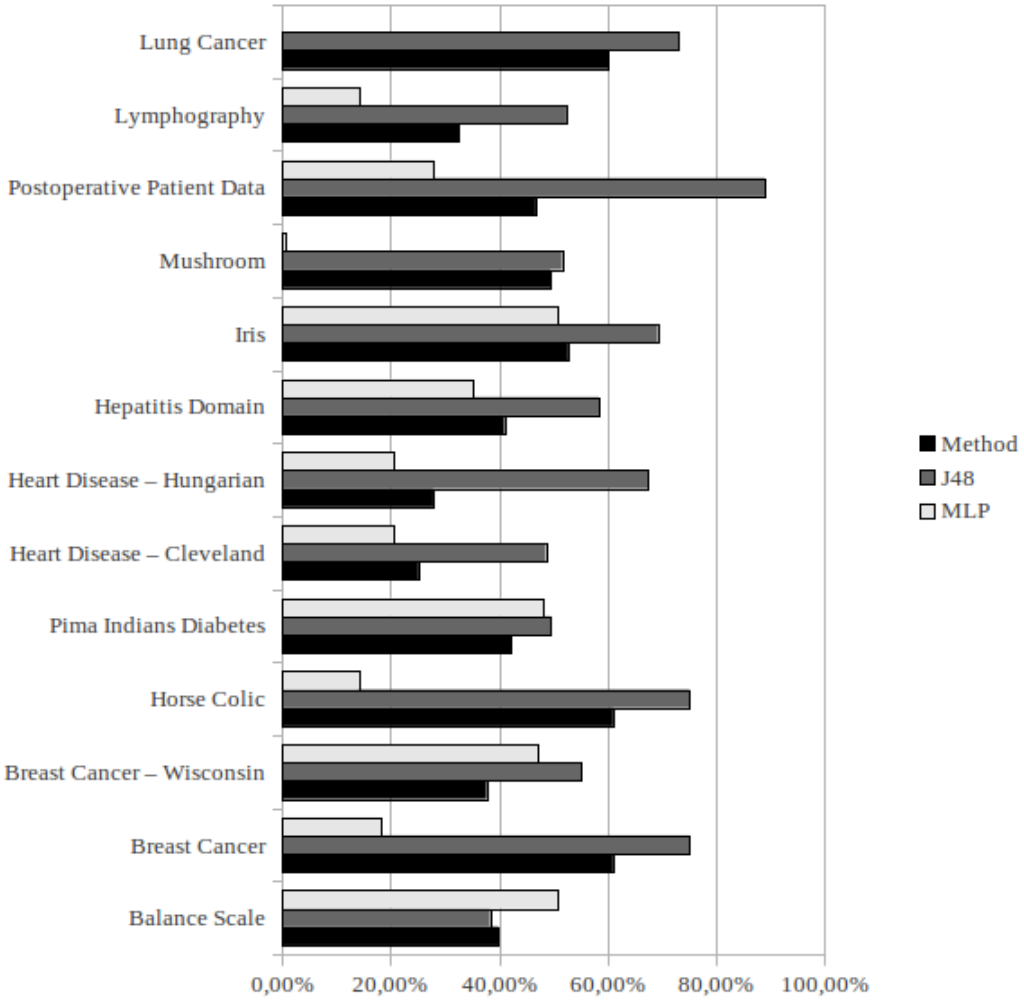

Figure 7.6: Comprehensibility of the proposed method compared to other techniques.

algorithm is compared to the control classifiers. Thus, it is possible to check its performance in this particular aspect. The same results are showed as a comparative chart in the figure 7.6.

The results indicate that the proposed algorithm achieved the expected goal. It generates more comprehensible neural networks, when compared to the traditional multilayer perceptrons.

The analysis of the measures demonstrates that the proposed method generates models with intermediate comprehensibility, when compared to decision trees and multilayer perceptrons. This characteristic was expected, because of the hybrid nature of this approach. The average comprehensibility for each classifier is:

Table 7.5: Comprehensibility of the proposed method compared to traditional classifiers.

Classifier	Database	Comprehensibility
Method	Balance Scale	39,84%
	Breast Cancer	61,01%
	Breast Cancer - Wisconsin	37,68%
	Horse Colic	61,01%
	Pima Indians Diabetes	42,20%
	Heart Disease - Cleveland	25,12%
	Heart Disease - Hungarian	27,81%
	Hepatitis Domain	40,97%
	Iris	52,70%
	Mushroom	49,46%
	Postoperative Patient Data	46,75%
	Lymphography	32,51%
	Lung Cancer	60,24%
J48	Balance Scale	38,34%
	Breast Cancer	75,15%
	Breast Cancer - Wisconsin	55,11%
	Horse Colic	75,15%
	Pima Indians Diabetes	49,57%
	Heart Disease - Cleveland	48,62%
	Heart Disease - Hungarian	67,42%
	Hepatitis Domain	58,41%
	Iris	69,24%
	Mushroom	51,64%
	Postoperative Patient Data	89,01%
	Lymphography	52,41%
	Lung Cancer	73,14%
MLP	Balance Scale	50,87%
	Breast Cancer	18,25%
	Breast Cancer - Wisconsin	47,12%
	Horse Colic	14,07%
	Pima Indians Diabetes	48,11%
	Heart Disease - Cleveland	20,60%
	Heart Disease - Hungarian	20,60%
	Hepatitis Domain	35,16%
	Iris	50,87%
	Mushroom	0,70%
	Postoperative Patient Data	27,96%
	Lymphography	14,35%
	Lung Cancer	0,00%

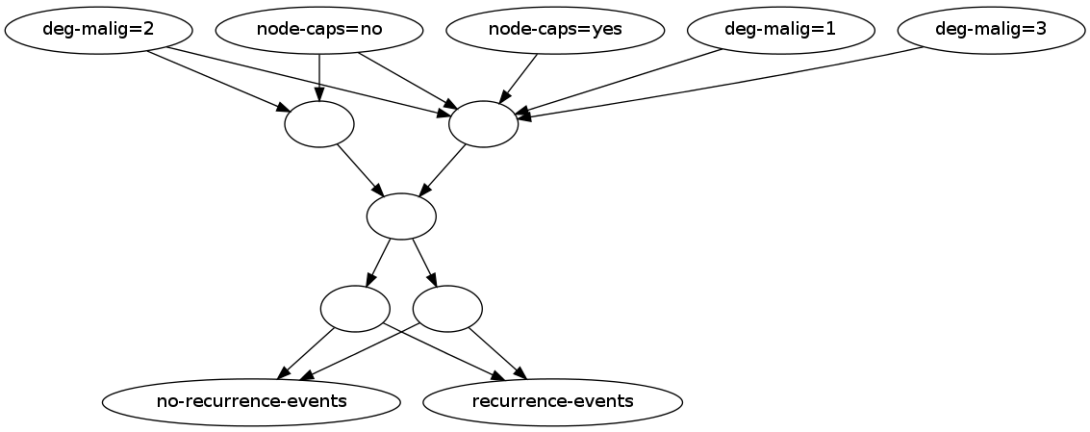

Figure 7.7: The neural network created by the proposed method for the Breast Cancer database.

- Proposed method: 44.41%;
- J48: 61.79%;
- MLP: 26.82%.

The T-test was applied to verify if the comprehensibility of proposed method is statistically superior to MLP and inferior to J48. The results (3.346 and 2.900 for J48 and MLP, respectively) indicate that this inference is correct, with 99.5% of confidence.

Figure 7.7 shows a neural network created by the proposed algorithm for the "Breast Cancer" dataset. This network has 12 neurons and 15 neural connections. On the other hand, a traditional multilayer perceptron with 3 layers would have 55 input neurons, 26 hidden neurons and 2 output neurons. It means that this network would have 1,404 neural connections to represent the same database model.

It is possible to observe that the sparse neural network is a simpler model, i.e. it is a more comprehensible net. Because there are fewer connections, it becomes clear how each neuron affects (or not) other nodes. Indeed, if a grayscale gradient is utilized to "paint" each node according to its activation (where "0" is white and "1" is black), it becomes possible to visualize the internal processing of the network with more plainness. This way, one could say that this kind of neural network is a "gray-box" model unlike the traditonal nets, which are "black-box" models. This interesting aspect can be observed in figure 7.8, which shows a neural network identifying different instances of "Breast Cancer" dataset.

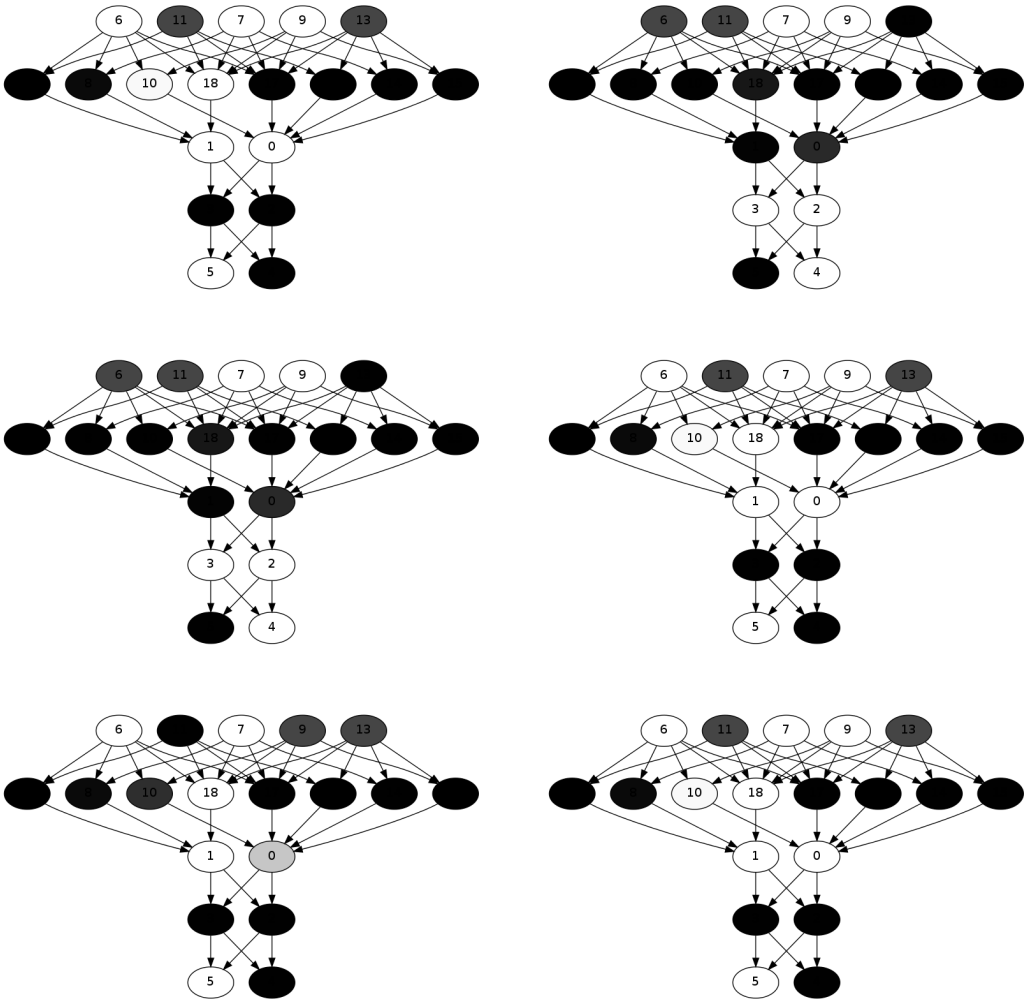

Figure 7.8: The internal activations of a neural network during the classification of different instances of "Breast Cancer" dataset.

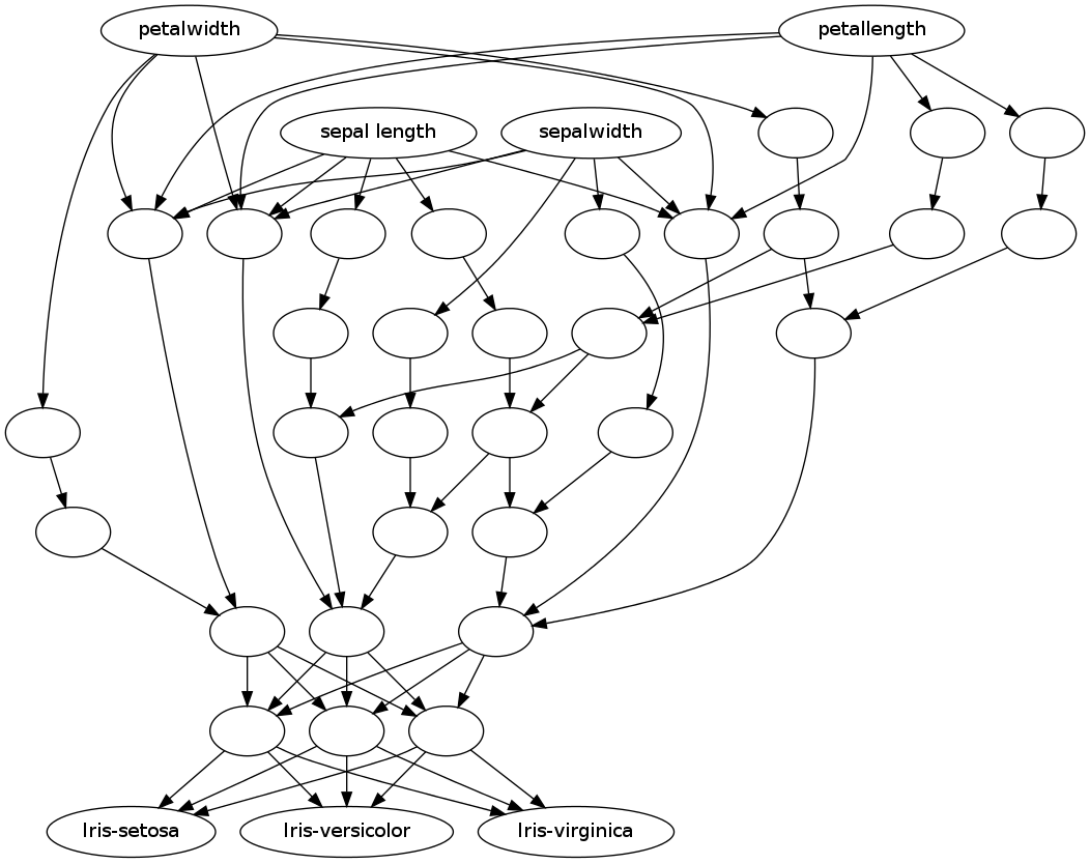

Figure 7.9: A neural network generated for the "Iris" database, before the pruning process.

7.4.1 Pruning

Figure 7.10 presents the effect of the pruning algorithm over the neural network presented in figure 7.9. It is possible to observe that several neurons and connections where removed, simplifying the complete model. Obviously, the pruning also affects the comprehensibility of the network. The same experiments were reproduced with this algorithm and the results are presented in table 7.6.

The results indicate that the pruning process had a positive effect over the comprehensibility of the generated models. This kind of effect was expected, since pruning removes complexity by definition. It is relevant to notice that the upgrade was stronger in complex models, e.g. post-operative patient database. The figure 7.11 highlights the differences.

A reasonable explanation for that could be find as an intrinsic deffect in the proposed algorithm. Apparently, the algorithm generates some unnecessary nodes during the creation of topology. These nodes are removed by the pruning process. Probably, the quantity of "extra" nodes is proportional to the size of the network. In other words,

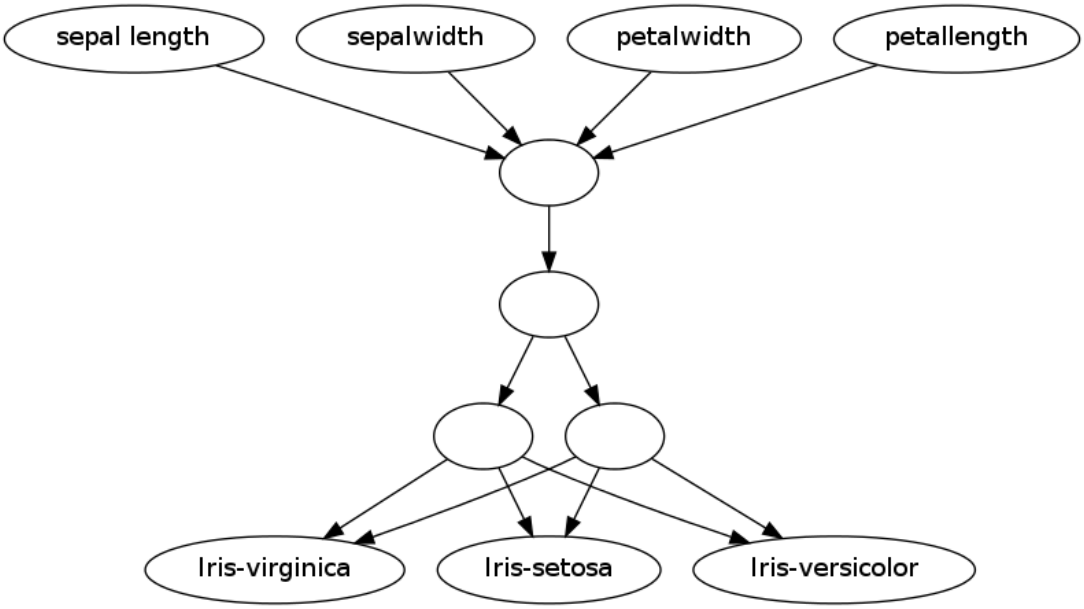

Figure 7.10: A neural network generated for the "Iris" database, after the pruning process.

Table 7.6: Comprehensibility of the proposed method after the pruning process.

Database	Before Pruning	After Pruning
Balance Scale	39.84%	63.98%
Breast Cancer	61.01%	69.24%
Breast Cancer - Wisconsin	37.68%	41.63%
Horse Colic	61.01%	64.54%
Pima Indians Diabetes	42.20%	61.64%
Heart Disease - Cleveland	25.12%	64.26%
Heart Disease - Hungarian	27.81%	69.56%
Hepatitis Domain	40.97%	46.00%
Iris	52.70%	71.10%
Mushroom	49.46%	49.46%
Postoperative Patient Data	46.75%	86.14%
Lymphography	32.51%	59.06%
Lung Cancer	60.24%	68.08%

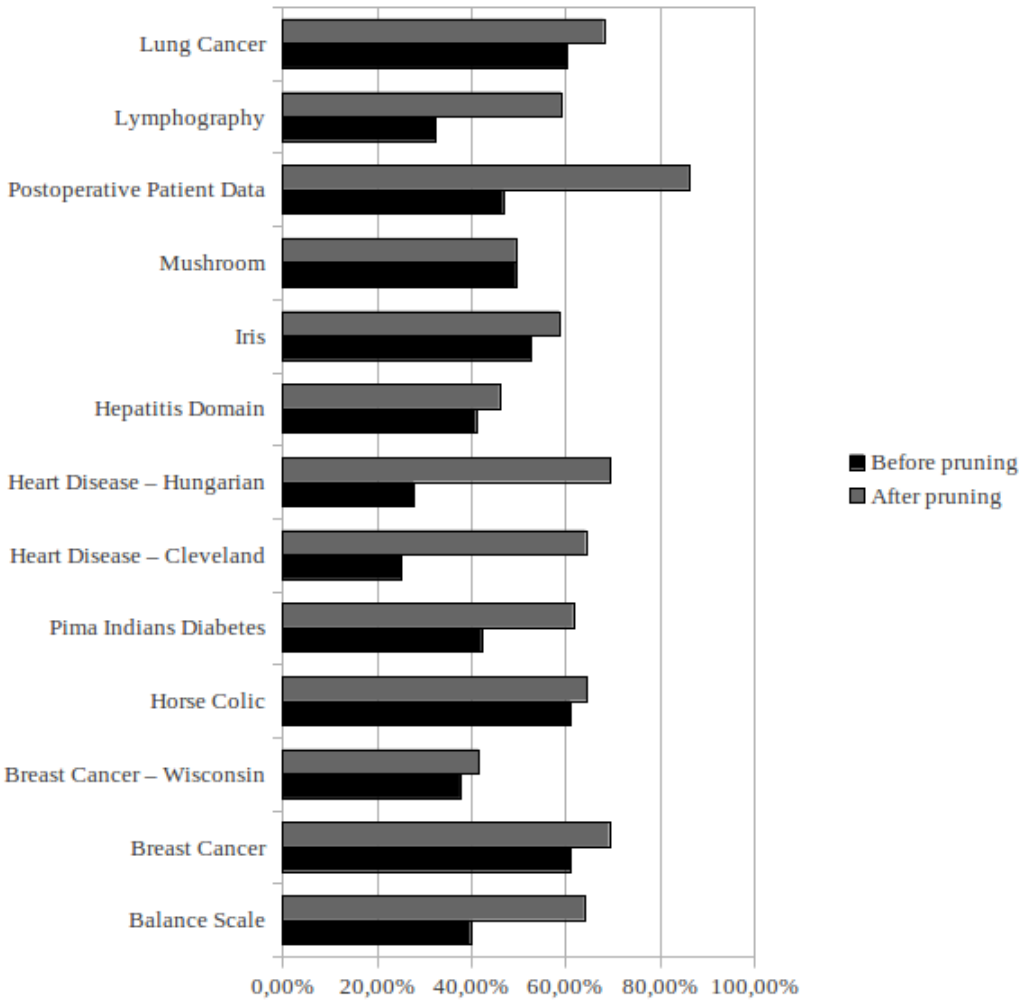

Figure 7.11: Pruning effects on comprehensibility of the generated models.

the larger networks, which are more complex and less comprehensible, tend to have more unnecessary neurons. This way, the pruning process is more radical in these models.

Another interesting point about the proposed learning algorithm appeared during the analysis of the network activations. Normally, a neural network is a "black-box" model, which means that it is not clear to understand the internal processes that determine its outputs. However, the networks created with the proposed algorithm are sparsed, with few connections between the neurons. This characteristic allows a better understading of the internal processing of the networks, once it is possible to visualize which neurons and connections are activated during each iteration.

It is important to reinforce that the pruning process does not affect the accuracy of the model. The pruning only affects the topology of the model, once the unnecessary nodes and connections are replaced by changes in the bias. This way, only the representation is affected, without any change in the mathematical model.

Chapter 8

Conclusion

This work investigated a new approach in neural networks field. It was proposed that the topology of a neural network is so important than the quantity of neurons, connections and weights. It is a very important point, because most neural algorithms do not focus this aspect during the learning process. In fact, it is possible to say that most methods are based in weights optimization.

Unlike traditional methods that focus mainly updating the weights, the proposed algorithm tries to create neural topologies according to the nature of the analyzed data. To achieve that, it uses the information gain to select the most important attributes and create the neural architecture. Because the information gain varies according data set examples, each neural network has a different structure.

Created neural networks share an important characteristic: they are sparsely connected. It means that each neuron has few connections. This is a key difference to conventional multilayer perceptrons, which are fully connected. This aspect is very important when comprehensibility is analyzed. The results showed that the sparse neural networks are sistematically more comprehensible thant the fully connected nets.

This can be observed in both quantitative and qualitative ways. The sparse neural networks tend to have less neurons and connections, which means they can be represented with less symbols, resulting in simpler models. In this work, a metric was proposed to measure the comprehensibility as a numeric value. The results reinforce this analysis.

Furthermore, the subjective review of networks also supports this conclusion. Because there are less connections between nodes, it is more feasible to notice which neurons are activated during each iteration. Thereby, it becomes possible to observe the internal behaviour of the network during the classification. In other words, one could visualize the

different activation patterns that are generated by the network when classifying different instances.

The statistical results showed that the proposed classifier had a good performance. This means that it achieved good classification rates in feasible time frames. Different databases were utilized during the tests to ensure the method validity. Also, all the results were compared to multilayer perceptrons and decision trees. These two traditional methods were utilized as control standards. The tests proved that the proposed method is statiscally similar to those obtained by decision trees and multilayer perceptrons.

Bibliography

ALMEIDA, M. *Introdução ao Estudo das Redes Neurais Artificiais*. 2006. Disponível em: <http://twiki.im.ufba.br/pub/MAT054/TodoMaterial/RNA.PDF>. Acesso em: May 22th, 2006.

BANERJEE, A. Initializing neural networks using decision trees. In: *Proceedings of the International Workshop on Computational Learning and Natural Learning Systems*. [S.l.]: MIT Press, 1994. p. 3–15.

BECKERT-NETO, A. et al. Automatic detection of cardiac arrhythmias using wavelets, neural networks and particle swarm optimization. In: *Proceedings of the 8th International Conference on Industrial Technology - IEEE/ICIT 2010*. Vina del Mar, Valparaiso, Chile: IEEE Press, 2010. v. 1, p. 1–5.

BISCAIA-JR., E.; SCHWAAB, M.; PINTO, J. Um novo enfoque do método do enxame de partículas. In: *Anais do NanoBio2004 - I Workshop em Nanotecnologia e Computação Inspirada na Biologia*. Rio de Janeiro, RJ, Brasil: [s.n.], 2004. p. 1–7.

BISWAL, B. et al. Automatic classification of power quality events using balanced neural tree. *Industrial Electronics, IEEE Transactions on*, v. 61, n. 1, p. 521–530, 2014. ISSN 0278-0046.

BLUM, A. L.; RIVEST, R. L. *Training A 3-Node Neural Network Is NP-Complete*. 1992.

BOUAZIZ, S. et al. A hybrid learning algorithm for evolving flexible beta basis function neural tree model. *Neurocomputing*, v. 117, n. 0, p. 107 – 117, 2013. ISSN 0925-2312. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0925231213001975>.

BRYSON-JR, A. E.; HO, Y.-C. *Applied optimal control: optimization, estimation, and control*. Waltham, Massachussets, USA: Blaisdell Pub. Co., 1969. 481 p.

CHAITIN, G. J. On the length of programs for computing finite binary sequences. *Journal of the ACM*, v. 13, p. 547–569, 1966.

CHAITIN, G. J. On the length of programs for computing finite binary sequences: Statistical considerations. *Journal of the ACM*, v. 13, p. 547–569, 1969.

CHEN, Y.; ABRAHAM, A. Feature selection and intrusion detection using hybrid flexible neural tree. In: *International Symposium on Neural Networks (ISNN 03*. [S.l.]: Lecture, 2005. p. 439–444.

CHEN, Y.; ABRAHAM, A.; YANG, B. Hybrid flexible neural-tree-based intrusion detection systems. *Int. J. Intell. Syst.*, v. 22, n. 4, p. 337–352, 2007.

CHEN, Y.; ABRAHAM, A.; ZHANG, Y. Ensemble of flexible neural trees for breast cancer detection. *The International Journal of Information Technology and Intelligent Computing*, v. 1, n. 1, p. 187–201, 2006.

CHEN, Y.; CHEN, F.; YANG, J. Y. Evolving mimo flexible neural trees for nonlinear system identification. In: ARABNIA, H. R.; YANG, M. Q.; YANG, J. Y. (Ed.). *Proceedings of the 2007 International Conference on Artificial Intelligence*. Las Vegas, Nevada, USA: CSREA Press, 2007. v. 1, p. 373–377.

CHEN, Y.; PENG, L.; ABRAHAM, A. Exchange rate forecasting using flexible neural trees. In: WANG, J. et al. (Ed.). *Advances in Neural Networks - ISNN 2006*. [S.l.]: Springer Berlin / Heidelberg, 2006, (Lecture Notes in Computer Science, v. 3973). p. 518–523.

CHEN, Y.; PENG, L.; ABRAHAM, A. Gene expression profiling using flexible neural trees. In: CORCHADO, E. et al. (Ed.). *7th International Conference of Intelligent Data Engineering and Automated Learning - IDEAL*. Burgos, Spain: Springer, 2006. (Lecture Notes in Computer Science, v. 4224), p. 1121–1128.

CHEN, Y.; YANG, B.; ABRAHAM, A. Flexible neural trees ensemble for stock index modeling. *Neurocomputing*, v. 70, n. 4-6, p. 697 – 703, 2007. ISSN 0925-2312. Advanced Neurocomputing Theory and Methodology - Selected papers from the International Conference on Intelligent Computing 2005 (ICIC 2005), International Conference on Intelligent Computing 2005. Disponível em: <http://www.sciencedirect.com/science/article/B6V10-4M4002C-3/2/5be44674b8842f82af3f140d9d6c8226>.

CHEN, Y.; YANG, B.; DONG, J. Evolving flexible neural networks using ant programming and pso algorithm. In: YIN, F.; WANG, J.; GUO, C. (Ed.). *Advances in Neural Networks - ISNN 2004*. [S.l.]: Springer Berlin / Heidelberg, 2004, (Lecture Notes in Computer Science, v. 3173). p. 211–216.

CHEN, Y. et al. Time-series forecasting using flexible neural tree model. *Information Sciences*, v. 174, n. 3-4, p. 219 – 235, 2005. ISSN 0020-0255. Disponível em: <http://www.sciencedirect.com/science/article/B6V0C-4DV19Y4-1/2/921daa2db2ad13ceee8a656f83189250>.

CHEN, Y.; YANG, B.; MENG, Q. Small-time scale network traffic prediction based on flexible neural tree. *Applied Soft Computing*, v. 12, n. 1, p. 274 – 279, 2012. ISSN 1568-4946. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1568494611003280>.

CRAVEN, M. W. *Extracting comprehensible models from trained neural networks*. Tese (Doutorado) — The University of Wisconsin - Madison, 1996. Supervisor-Shavlik, Jude W.

CRAVEN, M. W.; SHAVLIK, J. W. Extracting comprehensible concept representations from trained neural networks. In: *In: Working Notes on the IJCAIâ95 Workshop on Comprehensibility in Machine Learning*. [S.l.: s.n.], 1995. p. 61–75.

DANCEY, D.; MCLEAN, D.; BANDAR, Z. Decision tree extraction from trained neural networks. In: BARR, V.; MARKOV, Z. (Ed.). *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference*. Miami Beach, Florida, USA: AAAI Press, 2004.

FAHLMAN, S. E.; LEBIERE, C. The cascade-correlation learning architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 524–532, 1990.

FAUSSET, L. *Fundamentals of Neural Neworks: Architectures, Algorithms and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, 1994. 461 p.

FORESTI, G. L.; MICHELONI, C. Generalized neural trees for pattern recognition. *IEEE Transactions on Neural Networks*, v. 13, n. 6, p. 1540–1547, November 2002.

FORTNOW, L. Kolmogorov complexity. In: DOWNEY, R.; HIRSCHFELDT, D. (Ed.). *Aspects of Complexity*. Kaikoura, New Zealand: De Gruyter, 2001. v. 4.

FRANK, A.; ASUNCION, A. *UCI Machine Learning Repository*. 2010. Disponível em: <http://archive.ics.uci.edu/ml>. Acesso em: July 11th, 2010.

FREAN, M. R. *Small Nets and Short Paths: Optimizing Neural Computation*. Tese (PhD thesis) — University of Edinburgh, Edinburgh, Scotland, 1990.

FREAN, M. R. The upstart algorithm: a method for constructing and training feedforward neural networks. *Neural Comput.*, MIT Press, Cambridge, MA, USA, v. 2, n. 2, p. 198–209, 1990. ISSN 0899-7667.

FU, L. Rule learning by searching on adapted nets. In: *The 9th National Conference on Artificial Intelligence*. Anaheim, Canada: [s.n.], 1991. p. 590–595.

GALLANT, S. I. Three constructive algorithms for network learning. In: *Eighth Annual Conference of the Cognitive Science Society*. Miami Beach, FL, USA: [s.n.], 1985. p. 313–319.

GALLANT, S. I. Perceptron-based learning algorithms. *Neural Networks, IEEE Transactions on*, v. 1, n. 2, p. 179–191, 1990.

GALLANT, S. I. *Neural Network Learning and Expert Systems*. Cambridge, Massachussets, USA: The MIT Press, 1994. 365 p.

GENTILI, S. A new method for information update in supervised neural structures. *Neurocomputing*, v. 51, p. 61 – 74, 2003. ISSN 0925-2312. Disponível em: <http://www.sciencedirect.com/science/article/B6V10-461XK41-2/2/af6fa2cc8810635fc213ef140a2cd68b>.

GENTILI, S.; BRAGATO, P. A neural-tree-based system for automatic location of earthquakes in northeastern italy. *Journal of Seismology*, Springer Netherlands, v. 10, p. 73–89, 2006. ISSN 1383-4649. 10.1007/s10950-005-9001-z. Disponível em: <http://dx.doi.org/10.1007/s10950-005-9001-z>.

GENTILI, S.; MICHELINI, A. Automatic picking of p and s phases using a neural tree. *Journal of Seismology*, Springer Netherlands, v. 10, p. 39–63, 2006. ISSN 1383-4649. 10.1007/s10950-006-2296-6. Disponível em: <http://dx.doi.org/10.1007/s10950-006-2296-6>.

GROSSBERG, S. How neural networks learn from experience. *Contour enhancement, short term memory, and constancies in reverberating neural networks*, v. 52, n. 3, p. 213–157, 1937.

GROUP, W. M. L. *WEKA*. 2010. Disponível em: <http://mloss.org/software/view/16/>. Acesso em: December 1st, 2010.

GUO, Y. et al. "flexible neural trees for online hand gesture recognition using surface electromyography". *Journal of Computers*, v. 7, n. 5, 2012. Disponível em: <http://www.ojs.academypublisher.com/index.php/jcp/article/view/jcp070510991103>.

GURNEY, K. *An Introduction to Neural Networks*. New York, NY, USA: Cambridge University Press, 1997. 234 p.

HARI, V. *Empirical Investigation of CART and Decision Tree Extraction from Neural Networks*. 2009. 142 p.

HAYASHI, H.; ZHAO, Q. Improvement of the neural network trees through fine-tuning of the threshold of each internal node. In: *ICONIP '09: Proceedings of the 16th International Conference on Neural Information Processing*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 657–666. ISBN 978-3-642-10676-7.

HAYASHI, H.; ZHAO, Q. Induction of compact neural network trees through centroid based dimensionality reduction. In: *Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics*. San Antonio, TX, USA: IEEE Press, 2009. p. 948–953.

HAYASHI, H.; ZHAO, Q. Model reduction of neural network trees based on dimensionality reduction. In: *IJCNN'09: Proceedings of the 2009 international joint conference on Neural Networks*. Piscataway, NJ, USA: IEEE Press, 2009. p. 1119–1124. ISBN 978-1-4244-3549-4.

HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. New York, NY, USA: Macmillan, 1994.

HEATON, J. *Introduction to Neural Networks with Java*. Chesterfield, MO, USA: Heaton Research, Inc., 2005. 380 p.

HEBB, D. O. *The Organization of Behavior: A Neuropsychological Theory*. New edition. New York: Wiley, 1949. Hardcover.

HINTON, G.; SEJNOWSKI, T.; ACKLEY, D. *Boltzmann Machines: Constraint Satisfaction Networks that Learn*. [S.l.], maio 1984.

HINTON, G. E. How neural networks learn from experience. *Scientific American*, v. 267, n. 3, p. 144–151, 1992.

JALAJA, R.; BISWAL, B. Power quality event classification using hilbert huang transform. In: SATAPATHY, S. C.; UDGATA, S. K.; BISWAL, B. N. (Ed.). *Proceedings of the International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA)*. Springer Berlin Heidelberg, 2013, (Advances in Intelligent Systems and Computing, v. 199). p. 153–161. ISBN 978-3-642-35313-0. Disponível em: <http://dx.doi.org/10.1007/978-3-642-35314-7$_1$8¿.

JANIKOW, C. Z. Fuzzy decision trees: Issues and methods. *IEEE Transactions on Systems, Man, and Cybernetics*, v. 28, n. 1, p. 1–14, 1998.

KENNEDY, J.; EBERHART, R. C. Particle swarm optimization. In: *Proceedings of the IEEE International Conference on Neural Networks*. [S.l.: s.n.], 1995. p. 1942–1948.

Kolmogorov, A. N. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, v. 1, n. 1, p. 1–7, 1965.

KOTSIANTIS, S. B. Supervised machine learning: A review of classification techniques. *Informatica*, v. 31, n. 3, p. 249–268, 2007.

LINCOFF, G. H. *The Audubon Society Field Guide to North American Mushrooms*. New York, NY, USA: Alfred A. Knop, 1981.

LO, S.-H. *A Neural Tree with Partial Incremental Learning Capability and Its Application in Spam Filtering*. 84 p. Tese (MsC thesis) — Computer Science and Information Engineering, National Central University, Taiwan, 2007.

LÖFSTRÖM, T.; JOHANSSON, U.; NIKLASSON, L. Rule extraction by seeing through the model. In: PAL, N. R. et al. (Ed.). *Neural Information Processing*. [S.l.]: Springer Berlin / Heidelberg, 2004, (Lecture Notes in Computer Science, v. 3316). p. 555–560. 10.1007/978-3-540-30499-9$_8$5.

MCCULLOCH, W.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, v. 5, n. 4, p. 115–133, December 1943.

MÉZARD, M.; NADAL, J.-P. Learning in feedforward layered networks: the tiling algorithm. *Journal of Physics A: Mathematical and General*, v. 22, n. 12, p. 2191, 1989.

MICHALSKI, R. S. A theory and methodology of inductive learning. In: _____. *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, California, USA: TIOGA Publishing Co., 1983. p. 111–161.

MICHELONI, C. et al. A balanced neural tree for pattern classification. *Neural Networks*, v. 27, n. 0, p. 81 – 90, 2012. ISSN 0893-6080. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0893608011002693>.

MINSKY, M.; PAPERT, S. *Perceptrons*. [S.l.]: Cambridge, MA: MIT Press, 1969.

NADAL, J.-P. Study of a growth algorithm for a feedforward network. *Int. J. Neural Syst.*, v. 1, n. 1, p. 55–59, 1989.

NANNEN, V. A short introduction to kolmogorov complexity. *CoRR*, abs/1005.2400, 2010.

NICOLETTI, M. do C.; BERTINI-JR., J. R. An empirical evaluation of constructive neural network algorithms in classification tasks. *Int. J. Innov. Comput. Appl.*, v. 1, n. 1, p. 2–13, 2007.

PAREKH, R.; YANG, J.; HONAVAR, V. *Constructive Neural Network Learning Algorithms for Multi- Category Real-Valued Pattern Classification*. [S.l.], February 1997. 50 p.

PAREKH, R.; YANG, J.; HONAVAR, V. Constructive neural-network learning algorithms for pattern classification. *IEEE Transactions on Neural Networks*, v. 11, n. 2, p. 436–451, 2000.

D.B. Parker. *Learning Logic: Invention Report*. 1982.

PEARL, J. *Heuristics: intelligent search strategies for computer problem solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN 0-201-05594-5.

QUINLAN, J. R. Induction of decision trees. In: *Machine Learning*. [S.l.: s.n.], 1986. p. 81–106.

QUINLAN, J. R. *C.4.5: Programs for machine learning*. Los Altos, CA, USA: Morgan Kauffman Publishers Inc., 1993. 302 p.

REMEIKIS, N.; SKUcAS, I.; MELNINKAITė, V. Text categorization using neural networks initialized with decision trees. *Informatica*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 15, n. 4, p. 551–564, 2004. ISSN 0868-4952.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, v. 65, n. 6, p. 386–408, 1958.

ROSENBLATT, F. *Principles of Neurodynamics; Perceptrons and the Theory of Brain Mechanisms*. Michigan, USA: Spartan Books, 1962. 616 p.

ROUNTREE, N. *Initialising Neural Networks with Prior*. Tese (PhD thesis) — University of Otago, Dunedin, New Zealand, 2006.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning internal representations by error propagation. MIT Press, Cambridge, MA, USA, p. 318–362, 1986.

SHOU-NING, Q. et al. Modeling of fluid industry based on flexible neural tree. *International Journal of Computer Theory and Engineering*, v. 1, n. 1, p. 97–101, 2009.

SKRINáROVá, J.; HURAJ, L.; SILáDI, V. A neural tree model for classification of computing grid resources using pso tasks scheduling. *Neural Network World*, v. 23, n. 3, p. 223–241, 2013.

SOLOMONOFF, R. J. A formal theory of inductive inference, part 1. *Information and Control*, v. 7, p. 1–22, 1964.

SOLOMONOFF, R. J. A formal theory of inductive inference, part 2. *Information and Control*, v. 7, p. 224–254, 1964b.

SU, M.-C.; LO, H.-H.; HSU, F.-H. A neural tree and its application to spam e-mail detection. *Expert Syst. Appl.*, Pergamon Press, Inc., Tarrytown, NY, USA, v. 37, n. 12, p. 7976–7985, 2010. ISSN 0957-4174.

SUÁREZ, A.; LUTSKO, J. F. Globally optimal fuzzy decision trees for classification and regression. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 21, n. 12, p. 1297–1311, 1999. ISSN 0162-8828.

TAKAHARU, T.; ZHAO, Q. Size reduction of neural network trees through retraining. *IEIC Technical Report (Institute of Electronics, Information and Communication Engineers)*, v. 102, n. 382, p. 17–22, 2002.

TAKAHARU, T.; ZHAO, Q.; LIU, Y. A study on on-line learning of nntrees. In: *Proceedings of the International Joint Conference on Neural Networks*. [S.l.]: IEEE Press, 2003. v. 4, p. 2540–2545.

TURING, A. M. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, v. 2, n. 42, p. 230–265, 1936.

UTGOFF, P. E. Perceptron trees: A case study in hybrid concept representations. *Connection Science*, v. 1, p. 377–391, 1989.

VESTERSTROEM, J.; RIGET, J. *Particle Swarms: Extensions for Improved Local, Multimodal, Dynamic Search in Numerical Optimization*. 2002.

WEN, W. X.; JENNINGS, A.; LIU, H. Learning a neural tree. In: *Proceedings International Joint Conference on Neural Networks*. [S.l.: s.n.], 1992. p. 2709–2715.

WEN, W. X. et al. Some performance comparisons for self-generating neural tree. In: *Proceedings International Joint Conference on Neural Networks*. [S.l.: s.n.], 1992.

WERBOS, P. J. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Tese (PhD thesis) — Harvard University, Cambridge, Massachussets, USA, 1974.

WU, P.; CHEN, Y. Grammar guided genetic programming for flexible neural trees optimization. In: *PAKDD'07: Proceedings of the 11th Pacific-Asia conference on Advances in knowledge discovery and data mining*. Berlin, Heidelberg: Springer-Verlag, 2007. p. 964–971. ISBN 978-3-540-71700-3.

XU, Q. et al. A intrusion detection approach based on understandable neural network trees. *JCSNS International Journal of Computer Science and Network Security*, v. 6, n. 11, p. 229–234, November 2006.

XU, Q. et al. A novel intrusion detection mode based on understandable neural network trees. *Journal of Electronics (China)*, Science Press, co-published with Springer-Verlag GmbH, v. 23, p. 574–579, 2006. ISSN 0217-9822.

XU, Q. et al. Interpretable Neural Network Tree for Continuous-Feature Data Sets. *Neural Information Processing*, v. 3, n. 3, p. 77–84, June 2004.

ZHANG, B. tak; OHM, P.; MÜHLENBEIN, H. Evolutionary induction of sparse neural trees. *Evolutionary Computation*, v. 5, p. 213–236, 1997.

ZHAO, Q. Evolutionary design of neural network tree - integration of decision tree, neural network and ga. In: *Proc. IEEE Congress on Evolutionary Computation*. [S.l.: s.n.], 2001. p. 240–244.

ZHOU, Z.-H. Comprehensibility of data mining algorithms. In: WANG, J. (Ed.). *Encyclopedia of Data Warehousing and Mining*. Nanjing University, China: IGI Global, 2005. p. 190–195.

Appendix A

Source Code

A.1 File: Mcz.java

```
1  package weka.classifiers.mcz;
2
3  import java.io.File;
4  import java.io.PrintWriter;
5  import java.io.Serializable;
6  import java.util.ArrayList;
7  import java.util.HashMap;
8  import java.util.Iterator;
9  import java.util.List;
10 import java.util.Map;
11 import java.util.StringTokenizer;
12 import net.sourceforge.jswarm_pso.FitnessFunction;
13 import net.sourceforge.jswarm_pso.Neighborhood;
14 import net.sourceforge.jswarm_pso.Neighborhood1D;
15 import net.sourceforge.jswarm_pso.Swarm;
16 import weka.attributeSelection.ASEvaluation;
17 import weka.attributeSelection.AttributeEvaluator;
18 import weka.attributeSelection.GainRatioAttributeEval;
19 import weka.classifiers.Classifier;
20 import weka.classifiers.mcz.singlelearning.Perceptron;
21 import weka.classifiers.mcz.singlelearning.SingleLearner;
```

```
22 import weka.classifiers.trees.j48.C45Split;
23 import weka.core.Attribute;
24 import weka.core.Capabilities;
25 import weka.core.CapabilitiesHandler;
26 import weka.core.Instance;
27 import weka.core.Instances;
28 import weka.core.Summarizable;
29 import weka.core.Utils;
30
31 public class Mcz extends FitnessFunction implements
32         Classifier, CapabilitiesHandler, Summarizable,
              Serializable, Cloneable {
33
34     private AttributeEvaluator infoGain;
35     private Map<String, Neuron> net;
36     private List<String> neuronsOrder;
37     private ActivationFunction function;
38     private List<String> axons;
39     private List<String> dendrites;
40     private Instances data;
41     private List<String> inputLayer;
42     private double[] bestPosition;
43     private boolean isPruningActivated;
44
45     public Mcz() {
46         infoGain = new GainRatioAttributeEval();
47         net = new HashMap<String, Neuron>();
48         neuronsOrder = new ArrayList<String>();
49         axons = new ArrayList<String>();
50         dendrites = new ArrayList<String>();
51         function = new SigmoidFunction();
52         inputLayer = new ArrayList<String>();
53     }
54
55     @Override
56     public void buildClassifier(Instances data) throws
```

```
        Exception {
57      this.data = data;
58      // calculates the info gain for each attribute
59      ((ASEvaluation) infoGain).buildEvaluator(data);
60
61      // creates the output layer
62      for (int i = 0; i < data.numClasses(); i++) {
63          Neuron output = new Neuron();
64          output.setName(data.classAttribute().value(i));
65          output.setBias(-0.56047);
66          output.setFunction(function);
67          output.setId(net.size());
68          net.put(output.getName(), output);
69          neuronsOrder.add(output.getName());
70      }
71
72      // creates the output layer
73      for (int i = 0; i < data.numClasses(); i++) {
74          Neuron output = new Neuron();
75          output.setName(data.classAttribute().value(i) +
                "_extra");
76          output.setBias(0);
77          output.setFunction(function);
78          output.setId(net.size());
79          for (int j = 0; j < data.numClasses(); j++) {
80              output.setInput(net.get(data.classAttribute().value(j)),
                    0);
81          }
82          net.put(output.getName(), output);
83          neuronsOrder.add(output.getName());
84      }
85
86      // creates the output layer
87      for (int i = 0; i < data.numClasses(); i++) {
88          Neuron output = new Neuron();
89          output.setName(data.classAttribute().value(i) +
```

```
                "_extra2");
90              output.setBias(0);
91              output.setFunction(function);
92              output.setId(net.size());
93              for (int j = 0; j < data.numClasses(); j++) {
94                  output.setInput(
95                          net.get(data.classAttribute().value(j)
                                + "_extra"), 0);
96              }
97              net.put(output.getName(), output);
98              neuronsOrder.add(output.getName());
99          }
100
101         // builds the network topology, in a recursive way
102         buildTopology(data, null, 0, new
                ArrayList<Attribute>());
103
104         // adds some "jocker" neurons, to increase complexity
                of model
105         for (int i = 0; i < data.numClasses(); i++) {
106             Neuron output =
                    net.get(data.classAttribute().value(i));
107
108             for (int k = 0; k < 1; k++) {
109                 Neuron jocker = new Neuron();
110                 jocker.setName("jocker_" + k + "_" +
111                         data.classAttribute().value(i));
112                 jocker.setId(net.size());
113                 jocker.setBias(0);
114                 jocker.setFunction(function);
115
116                 for (int j = 0; j < inputLayer.size(); j++) {
117                     Neuron input = net.get(inputLayer.get(j));
118                     jocker.setInput(input, 0);
119                 }
120
```

```
121                 output.setInput(jocker, 0);
122
123                 net.put(jocker.getName(), jocker);
124                 neuronsOrder.add(jocker.getName());
125             }
126
127             // sets the OR neurons
128             Map<Neuron, Double> connections =
                    output.getInputs();
129             double[] orWeights =
                    SingleLearner.learnOr(connections.size());
130             Object[] neurons = connections.keySet().toArray();
131             for (int j = 0; j < neurons.length; j++) {
132                 output.setInput((Neuron) neurons[j],
                        orWeights[0]);
133             }
134             output.setBias(orWeights[orWeights.length - 1]);
135         }
136
137         Iterator<Neuron> neurons = net.values().iterator();
138         while (neurons.hasNext()) {
139             Neuron n = neurons.next();
140             axons.add(n.getName());
141             dendrites.add(n.getName());
142             Iterator<Neuron> inputNeurons =
                    n.getInputs().keySet().iterator();
143             while (inputNeurons.hasNext()) {
144                 Neuron input = inputNeurons.next();
145                 if (input == null) {
146                     continue;
147                 }
148                 axons.add(input.getName());
149                 dendrites.add(n.getName());
150             }
151
152             n.fix();
```

```
153         }
154
155         Swarm swarm = null;
156         setMaximize(false);
157
158         double lastBestFitness = Double.MAX_VALUE;
159         double maxPosition = 21.0;
160         double minPosition = -21.0;
161         double minMaxVelocity = 0.1;
162         double inertia = 1.2;
163         double particleIncrement = 3;
164         double globalIncrement = 0.9;
165
166         for (int i = 0; i < 1; i++) {
167             System.out.println("Iteration " + i);
168
169             swarm = new Swarm(70,
170                     new NetworkParticle(axons.size()), this);
171             Neighborhood neigh = new Neighborhood1D(
172                     Swarm.DEFAULT_NUMBER_OF_PARTICLES / 5,
                            true);
173             swarm.setNeighborhood(neigh);
174             swarm.setMaxPosition(maxPosition);
175             swarm.setMinPosition(minPosition);
176             swarm.setMaxMinVelocity(minMaxVelocity);
177             swarm.setInertia(inertia);
178             swarm.setParticleIncrement(particleIncrement);
179             swarm.setGlobalIncrement(globalIncrement);
180
181             swarm.init();
182             for (int j = 0; j < 1000; j++) {
183                 swarm.evolve();
184                 System.out.println("Evolution " + i + ":" + j +
                        " _ "
185                             + swarm.getBestFitness());
186
```

```
187                 if (j > 0 && j % 200 == 0) {
188                     inertia -= 0.1;
189                     if (inertia < 0.6) {
190                         inertia = 0.6;
191                     }
192                     swarm.setInertia(inertia);
193                 }
194             }
195 
196             if (swarm.getBestFitness() < lastBestFitness) {
197                 lastBestFitness = swarm.getBestFitness();
198                 bestPosition = swarm.getBestPosition();
199                 setWeights(bestPosition);
200             }
201         }
202 
203         neurons = net.values().iterator();
204         while (neurons.hasNext()) {
205             neurons.next().restore();
206         }
207 
208         if (isPruningActivated) {
209             prune(data, 0.1);
210         }
211     }
212 
213     public boolean buildTopology(Instances data, Neuron
        inputNeuron,
214             int depth, List<Attribute> attributes) throws
                Exception {
215         if (data.numInstances() == 0) {
216             return false;
217         }
218 
219         ((ASEvaluation) infoGain).buildEvaluator(data);
220 
```

```
221         double tempGain = 0;
222         double maxGain = 0;
223         int maxIndex = 0;
224         for (int i = 0; i < data.numAttributes(); i++) {
225             if (data.classIndex() == i ||
226                     attributes.contains(data.attribute(i))) {
227                 continue;
228             }
229
230             C45Split split = new C45Split(i, 0,
                    data.sumOfWeights(), true);
231             split.buildClassifier(data);
232             if (split.gainRatio() > maxGain) {
233                 tempGain = split.gainRatio();
234                 maxGain = tempGain;
235                 maxIndex = i;
236             }
237         }
238
239         if (maxGain == 0) {
240             int[] distribution = new
                    int[data.classAttribute().numValues()];
241             for (int i = 0; i < data.numInstances(); i++) {
242                 distribution[(int) data.instance(i).value(
243                         data.classAttribute())]++;
244             }
245             int classIndex = 0;
246             int maxDistribution = -1;
247             for (int i = 0; i < distribution.length; i++) {
248                 if (distribution[i] > maxDistribution) {
249                     classIndex = i;
250                     maxDistribution = distribution[i];
251                 }
252             }
253
254             Neuron outputNeuron = net.get(
```

```
255                     data.classAttribute().value(classIndex));
256             double weight = 1.68228;
257             outputNeuron.setInput(inputNeuron, weight);
258             return true;
259         }
260 
261         Attribute attribute = data.attribute(maxIndex);
262         if (attribute.equals(data.classAttribute())) {
263             return false;
264         }
265 
266         attributes.add(attribute);
267 
268         if (attribute.isNominal()) {
269             int categories = attribute.numValues();
270             for (int i = 0; i < categories; i++) {
271                 String nodeName = attribute.name() + "=" +
                        attribute.value(i);
272 
273                 // checks if this category is representative
274                 boolean isRepresentative = false;
275                 for (int j = 0; j < data.numInstances(); j++) {
276                     if
                            (data.instance(j).stringValue(attribute).equals(
277                                 attribute.value(i))) {
278                         isRepresentative = true;
279                     }
280                 }
281                 if (!isRepresentative) {
282                     continue;
283                 }
284 
285 
286                 Neuron neuron = null;
287                 if (net.containsKey(nodeName)) {
288                     neuron = net.get(nodeName);
```

```
289                 } else {
290                     neuron = new Neuron();
291                     neuron.setName(nodeName);
292                     neuron.setId(net.size());
293                     neuron.setBias(0);
294                     neuron.setFunction(function);
295                     net.put(nodeName, neuron);
296                     neuronsOrder.add(nodeName);
297                     inputLayer.add(nodeName);
298                 }
299                 if (inputNeuron != null) {
300                     double[] andWeights = Perceptron.learnAnd();
301 
302                     Neuron andNeuron = new Neuron();
303                     andNeuron.setId(net.size());
304                     andNeuron.setName("AND" +
                            andNeuron.getId());
305                     andNeuron.setBias(0);
306                     andNeuron.setFunction(function);
307                     net.put(andNeuron.getName(), andNeuron);
308                     neuronsOrder.add(andNeuron.getName());
309 
310                     andNeuron.setInput(inputNeuron,
                            andWeights[0]);
311                     andNeuron.setInput(neuron, andWeights[1]);
312                     andNeuron.setBias(andWeights[andWeights.length
                            - 1]);
313                     neuron = andNeuron;
314                 }
315                 Instances dataSubset = new Instances(data);
316                 for (int j = 0; j < dataSubset.numInstances();
                        j++) {
317                     if
                            (!dataSubset.instance(j).stringValue(attribute).equal
318                                 attribute.value(i))) {
319                         dataSubset.remove(j);
```

```
320                         j--;
321                     }
322                 }
323                 if (!buildTopology(dataSubset, neuron, depth +
                        1, attributes)) {
324                     Neuron outputNeuron = net.get(
325                             data.classAttribute().value(
326                             (int)
                                    data.instance(0).classValue()));
327                     double weight = 1.68228;
328                     outputNeuron.setInput(neuron, weight);
329                 }
330             }
331
332             attributes.remove(attribute);
333             return true;
334         } else if (attribute.isNumeric()) {
335             String nodeName = attribute.name();
336             Neuron neuron = null;
337             if (net.containsKey(nodeName)) {
338                 neuron = net.get(nodeName);
339             } else {
340                 neuron = new Neuron();
341                 neuron.setName(nodeName);
342                 neuron.setId(net.size());
343                 neuron.setBias(0);
344                 neuron.setFunction(function);
345                 net.put(nodeName, neuron);
346                 neuronsOrder.add(nodeName);
347                 inputLayer.add(nodeName);
348             }
349
350             C45Split split = new C45Split(
351                     attribute.index(), 0, data.sumOfWeights(),
                            true);
352             split.buildClassifier(data);
```

```
353             double threshold = split.splitPoint();
354
355             Neuron lowerNeuron1 = new Neuron();
356             lowerNeuron1.setName(nodeName + "<=" + threshold +
                    "_1");
357             lowerNeuron1.setId(net.size());
358             lowerNeuron1.setFunction(new SigmoidFunction());
359             lowerNeuron1.setBias(
360                     0.20884620618871286 + (-47.169443613047356
                            * threshold));
361             lowerNeuron1.setInput(neuron, 47.447919015473474);
362             net.put(lowerNeuron1.getName(), lowerNeuron1);
363             neuronsOrder.add(lowerNeuron1.getName());
364
365             Neuron lowerNeuron = new Neuron();
366             lowerNeuron.setName(nodeName + "<=" + threshold);
367             lowerNeuron.setId(net.size());
368             lowerNeuron.setFunction(new LinearFunction());
369             lowerNeuron.setBias(0.9954430924071158);
370             lowerNeuron.setInput(lowerNeuron1,
                    -1.987600737950668);
371             net.put(lowerNeuron.getName(), lowerNeuron);
372             neuronsOrder.add(lowerNeuron.getName());
373
374             if (inputNeuron != null) {
375                 double[] andWeights = Perceptron.learnAnd();
376
377                 Neuron andNeuron = new Neuron();
378                 andNeuron.setId(net.size());
379                 andNeuron.setName("AND" + andNeuron.getId());
380                 andNeuron.setBias(0);
381                 andNeuron.setFunction(function);
382                 net.put(andNeuron.getName(), andNeuron);
383                 neuronsOrder.add(andNeuron.getName());
384
385                 andNeuron.setInput(inputNeuron, andWeights[0]);
```

```
386                 andNeuron.setInput(lowerNeuron, andWeights[1]);
387                 andNeuron.setBias(andWeights[andWeights.length
                        - 1]);
388                 lowerNeuron = andNeuron;
389             }
390
391             Instances dataSubset = new Instances(data);
392             for (int j = 0; j < dataSubset.numInstances(); j++)
                    {
393                 if (dataSubset.instance(j).value(attribute) >
                        threshold) {
394                     dataSubset.remove(j);
395                     j--;
396                 }
397             }
398             if (!buildTopology(
399                     dataSubset, lowerNeuron, depth + 1,
                            attributes)) {
400                 Neuron outputNeuron = net.get(
401                         data.classAttribute().value(
402                         (int) data.instance(0).classValue()));
403                 double weight = 1.68228;
404                 outputNeuron.setInput(lowerNeuron, weight);
405             }
406
407             Neuron higherNeuron1 = new Neuron();
408             higherNeuron1.setName(nodeName + ">" + threshold +
                    "_1");
409             higherNeuron1.setId(net.size());
410             higherNeuron1.setFunction(new SigmoidFunction());
411             higherNeuron1.setBias(
412                     -0.09314142234358645 + (-44.149995579850916
                            * threshold));
413             higherNeuron1.setInput(neuron, 44.826409240030856);
414             net.put(higherNeuron1.getName(), higherNeuron1);
415             neuronsOrder.add(higherNeuron1.getName());
```

```
416
417             Neuron higherNeuron = new Neuron();
418             higherNeuron.setName(nodeName + ">" + threshold);
419             higherNeuron.setId(net.size());
420             higherNeuron.setFunction(new LinearFunction());
421             higherNeuron.setBias(-1.0060884523270537);
422             higherNeuron.setInput(higherNeuron1,
                    2.006073420423026);
423             net.put(higherNeuron.getName(), higherNeuron);
424             neuronsOrder.add(higherNeuron.getName());
425
426             if (inputNeuron != null) {
427                 double[] andWeights = Perceptron.learnAnd();
428
429                 Neuron andNeuron = new Neuron();
430                 andNeuron.setId(net.size());
431                 andNeuron.setName("AND" + andNeuron.getId());
432                 andNeuron.setBias(0);
433                 andNeuron.setFunction(function);
434                 net.put(andNeuron.getName(), andNeuron);
435                 neuronsOrder.add(andNeuron.getName());
436
437                 andNeuron.setInput(inputNeuron, andWeights[0]);
438                 andNeuron.setInput(higherNeuron, andWeights[1]);
439                 andNeuron.setBias(andWeights[andWeights.length
                        - 1]);
440                 higherNeuron = andNeuron;
441             }
442
443             dataSubset = new Instances(data);
444             for (int j = 0; j < dataSubset.numInstances(); j++)
                    {
445                 if (dataSubset.instance(j).value(attribute) <=
                        threshold) {
446                     dataSubset.remove(j);
447                     j--;
```

```
448                 }
449             }
450             if (!buildTopology(
451                     dataSubset, higherNeuron, depth + 1,
                            attributes)) {
452                 Neuron outputNeuron = net.get(
453                         data.classAttribute().value(
454                         (int) data.instance(0).classValue()));
455                 double weight = 1.68228;
456                 outputNeuron.setInput(higherNeuron, weight);
457             }
458
459             attributes.remove(attribute);
460             return true;
461         }
462
463         attributes.remove(attribute);
464         return false;
465     }
466
467     @Override
468     public double classifyInstance(Instance instance) throws
            Exception {
469         Iterator<Neuron> neurons = net.values().iterator();
470         while (neurons.hasNext()) {
471             neurons.next().reset();
472         }
473
474         Neuron input = null;
475         Attribute attribute = null;
476         for (int i = 0; i < instance.numAttributes(); i++) {
477             attribute = instance.attribute(i);
478             if (attribute.index() == instance.classIndex()) {
479                 continue;
480             }
481             if (attribute.isNumeric()) {
```

```
482                 input = net.get(attribute.name());
483                 if (input == null) {
484                     continue;
485                 }
486                 input.setOutput(
487                         SigmoidFunction.calc(instance.value(attribute)));
488             } else {
489                 input = net.get(attribute.name() + "=")
490                         + attribute.value((int)
                                instance.value(attribute)));
491                 if (input == null) {
492                     continue;
493                 }
494                 input.setOutput(SigmoidFunction.calc(1));
495             }
496         }
497 
498         Neuron output = null;
499         int winner = 0;
500         double winnerOutput = Double.MIN_VALUE;
501         for (int i = 0; i < instance.numClasses(); i++) {
502             output = net.get(instance.classAttribute().value(i)
                    + "_extra2");
503             if (output.activate() > winnerOutput) {
504                 winnerOutput = output.getOutput();
505                 winner = i;
506             }
507         }
508 
509         return winner;
510     }
511 
512     @Override
513     public String toSummaryString() {
514         return "Classifier";
515     }
```

```
516
517     public void setWeights(double[] weigths) {
518         Iterator<Neuron> neurons = net.values().iterator();
519         while (neurons.hasNext()) {
520             neurons.next().restore();
521         }
522
523         for (int i = 0; i < weigths.length; i++) {
524             Neuron input = net.get(axons.get(i));
525             Neuron output = net.get(dendrites.get(i));
526             if (input.equals(output)) {
527                 output.setBias(output.getBias() + weigths[i]);
528             } else {
529                 try {
530                     output.setInput(input,
                                output.getInputs().get(input)
531                                     + weigths[i]);
532                 } catch (Exception e) {
533                 }
534             }
535         }
536     }
537
538     public double[] distributionForInstance(Instance instance)
539             throws Exception {
540         double[] dist = new double[instance.numClasses()];
541         switch (instance.classAttribute().type()) {
542             case Attribute.NOMINAL:
543                 double classification =
                            classifyInstance(instance);
544                 if (Utils.isMissingValue(classification)) {
545                     return dist;
546                 } else {
547                     dist[(int) classification] = 1.0;
548                 }
549                 return dist;
```

```
550             case Attribute.NUMERIC:
551                 dist[0] = classifyInstance(instance);
552                 return dist;
553             default:
554                 return dist;
555         }
556     }
557 
558     public Capabilities getCapabilities() {
559         Capabilities result = new Capabilities(this);
560         result.enableAll();
561         return result;
562     }
563 
564     @Override
565     public double evaluate(double[] position) {
566         setWeights(position);
567 
568         double fitness = 0;
569 
570         for (int i = 0; i < data.numInstances(); i++) {
571             try {
572                 fitness +=
                        Math.abs(learnInstance(data.instance(i)));
573             } catch (Exception e) {
574                 e.printStackTrace();
575             }
576         }
577 
578         return fitness;
579     }
580 
581     public double learnInstance(Instance instance) throws
        Exception {
582         Iterator<Neuron> neurons = net.values().iterator();
583         while (neurons.hasNext()) {
```

```
584             neurons.next().reset();
585         }
586
587         Neuron input = null;
588         Attribute attribute = null;
589         for (int i = 0; i < instance.numAttributes(); i++) {
590             attribute = instance.attribute(i);
591             if (attribute.index() == instance.classIndex()) {
592                 continue;
593             }
594             if (attribute.isNumeric()) {
595                 input = net.get(attribute.name());
596                 if (input == null) {
597                     continue;
598                 }
599                 input.setOutput(
600                         SigmoidFunction.calc(instance.value(attribute)));
601             } else {
602                 input = net.get(attribute.name() + "=" +
603                         attribute.value((int)
                                instance.value(attribute)));
604                 if (input == null) {
605                     continue;
606                 }
607                 input.setOutput(SigmoidFunction.calc(1));
608             }
609         }
610
611         int realClass = (int)
                instance.value(data.classAttribute());
612         double[] realOutputs = new
                double[instance.numClasses()];
613         realOutputs[realClass] = 1.0;
614
615         double[] errors = new double[instance.numClasses()];
616
```

```
617         Neuron output = null;
618         for (int i = 0; i < instance.numClasses(); i++) {
619             output = net.get(instance.classAttribute().value(i)
                    + "_extra2");
620             output.activate();
621             errors[i] += Math.abs(Math.abs(output.getOutput()) -
622                     Math.abs(realOutputs[i]));
623         }
624
625         double averageError = 0;
626         for (int j = 0; j < errors.length; j++) {
627             averageError += errors[j];
628         }
629         return (double) (averageError / errors.length);
630     }
631
632     public double getComprehensibility(Instances data) {
633         double comprehensibility = 0;
634
635         if (data == null) {
636             return comprehensibility;
637         }
638
639         StringBuffer network = new StringBuffer();
640         for (int i = 0; i < data.numClasses(); i++) {
641             if (i > 0) {
642                 network.append("else ");
643             }
644             network.append("if act ( ");
645             Neuron output =
                    net.get(data.classAttribute().value(i));
646             if (output == null) {
647                 continue;
648             }
649             System.out.println("hey: " +
                    data.classAttribute().value(i));
```

```
650             reccur(network, output);
651             network.append(
652                     ") > 0.5 then class is " +
                            data.classAttribute().value(i)
653                     + " ");
654         }
655
656         StringTokenizer tok = new
                StringTokenizer(network.toString());
657         int tokens = tok.countTokens();
658         comprehensibility = -10 * Math.log1p(tokens) +
                110.986122887;
659
660         System.out.println("==========================");
661         System.out.println("Comprehensibility: " +
                comprehensibility);
662         System.out.println("==========================");
663         System.out.println(network.toString());
664         System.out.println("==========================");
665
666         return comprehensibility;
667     }
668
669     public void reccur(StringBuffer network, Neuron neuron) {
670         Map<Neuron, Double> inputs = neuron.getInputs();
671         int i = 0;
672         for (Iterator<Neuron> it = inputs.keySet().iterator();
673                 it.hasNext(); i++) {
674             if (i > 0) {
675                 network.append("+ ");
676             }
677             Neuron aNeuron = it.next();
678             if (aNeuron.getInputs().size() > 0) {
679                 network.append("act ( ");
680                 reccur(network, aNeuron);
681                 network.append(") * W ");
```

```
682             } else {
683                 network.append(aNeuron.getName() + " * W ");
684             }
685         }
686     }
687
688     @Override
689     public String toString() {
690         double comprehensibility = getComprehensibility(data);
691         return "\n\nComprehensibility: " + comprehensibility +
                " %\n\n";
692     }
693
694     void prune(Instances data, double threshold) throws
            Exception {
695         StatisticUtils stat = new StatisticUtils();
696         Iterator<Neuron> allNeurons = null;
697         for (int i = 0; i < data.size(); i++) {
698             classifyInstance(data.instance(i));
699             allNeurons = net.values().iterator();
700             while (allNeurons.hasNext()) {
701                 Neuron aNeuron = allNeurons.next();
702                 stat.add(aNeuron.getId(), aNeuron.getOutput());
703             }
704         }
705
706         List<String> prunedNeurons = new ArrayList<String>();
707         allNeurons = net.values().iterator();
708         while (allNeurons.hasNext()) {
709             Neuron aNeuron = allNeurons.next();
710             double standardDeviation =
711                     stat.calculateStandardDeviation(aNeuron.getId());
712             if (standardDeviation < threshold &&
713                     !aNeuron.getName().contains("_extra2")) {
714                 prunedNeurons.add(aNeuron.getName());
715             }
```

```
716         }
717
718         for (int i = 0; i < axons.size(); i++) {
719             if (prunedNeurons.contains(axons.get(i))
720                     ||
                        prunedNeurons.contains(dendrites.get(i)))
                        {
721                 axons.remove(i);
722                 dendrites.remove(i);
723                 i--;
724             }
725         }
726
727         for (int i = 0; i < prunedNeurons.size(); i++) {
728             net.remove(prunedNeurons.get(i));
729         }
730
731         allNeurons = net.values().iterator();
732         while (allNeurons.hasNext()) {
733             Neuron aNeuron = allNeurons.next();
734             for (int i = 0; i < prunedNeurons.size(); i++) {
735                 aNeuron.getInputs().remove(prunedNeurons.get(i));
736             }
737         }
738     }
739
740     public void printNets(Instances data, String dir) throws
        Exception {
741         for (int i = 0; i < data.size(); i++) {
742             classifyInstance(data.instance(i));
743             File file = new File(dir + i + ".dot");
744             PrintWriter writer = new PrintWriter(file);
745             writer.println("digraph G {");
746             writer.println("node [style=filled];");
747             writer.flush();
748             Iterator<Neuron> allNeurons =
```

```
                net.values().iterator();
749             while (allNeurons.hasNext()) {
750                 Neuron aNeuron = allNeurons.next();
751                 double activation = aNeuron.getOutput();
752                 int intActivation = (int) (0xFF * activation);
753                 if (intActivation > 0xFF) {
754                     intActivation = 0xFF;
755                 }
756                 if (intActivation < 0) {
757                     intActivation = 0;
758                 }
759                 intActivation = 0xFF - intActivation;
760                 String colorLevel =
                        Integer.toHexString(intActivation);
761                 if (intActivation <= 0xF) {
762                     colorLevel = "0" + colorLevel;
763                 }
764                 writer.println(aNeuron.getId() + " [fillcolor =
                        \"#" +
765                         colorLevel + colorLevel + colorLevel +
                            "\"];");
766             }
767             for (int j = 0; j < axons.size(); j++) {
768                 if (!axons.get(j).equals(dendrites.get(j))) {
769                     writer.println(net.get(axons.get(j)).getId()
                            + " -> " +
770                             net.get(dendrites.get(j)).getId() +
                                ";");
771                     writer.flush();
772                 }
773             }
774             writer.println("}");
775             writer.flush();
776             writer.close();
777         }
778     }
```

```
779 }
```

A.2 File: Neuron.java

```
1  package weka.classifiers.mcz;
2
3  import java.io.Serializable;
4  import java.util.HashMap;
5  import java.util.Map;
6
7  public class Neuron implements Serializable {
8
9      private int id;
10     private String name;
11     private double bias;
12     private Map<Neuron, Double> inputs;
13     private double output;
14     private ActivationFunction function;
15     private boolean wasActivated;
16
17
18     private double fixedBias;
19     private Map<Neuron, Double> fixedInputs;
20
21     public Neuron() {
22         inputs = new HashMap<Neuron, Double>();
23         wasActivated = false;
24     }
25
26     public double getBias() {
27         return bias;
28     }
29
30     public ActivationFunction getFunction() {
31         return function;
32     }
```

```
33
34      public int getId() {
35          return id;
36      }
37
38      public String getName() {
39          return name;
40      }
41
42      public double getOutput() {
43          if(!wasActivated) {
44              activate();
45          }
46          return output;
47      }
48
49      public Map<Neuron, Double> getInputs() {
50          return inputs;
51      }
52
53      public void setBias(double bias) {
54          this.bias = bias;
55      }
56
57      public void setFunction(ActivationFunction function) {
58          this.function = function;
59      }
60
61      public void setId(int id) {
62          this.id = id;
63      }
64
65      public void setName(String name) {
66          this.name = name;
67      }
68
```

```
69      public void setOutput(double output) {
70          this.output = output;
71          wasActivated = true;
72      }
73
74      public void setInput(Neuron inputNeuron, double weight) {
75          if(inputNeuron == null) {
76              return;
77          }
78          if(inputs.containsKey(inputNeuron)) {
79              inputs.remove(inputNeuron);
80          }
81          inputs.put(inputNeuron, weight);
82      }
83
84      public double activate() {
85          wasActivated = true;
86          output = function.activate(this);
87          return output;
88      }
89
90      public void reset() {
91          wasActivated = false;
92          output = 0;
93      }
94
95      @Override
96      public boolean equals(Object obj) {
97          if (obj == null) {
98              return false;
99          }
100         if (obj instanceof String && name != null) {
101             return name.equals(obj);
102         }
103         if (getClass() != obj.getClass()) {
104             return false;
```

```
105         }
106         final Neuron other = (Neuron) obj;
107         if (this.id != other.id) {
108             return false;
109         }
110         if ((this.name == null) ? (other.name != null) :
                !this.name.equals(other.name)) {
111             return false;
112         }
113         return true;
114     }
115
116     @Override
117     public int hashCode() {
118         return (this.name != null ? this.name.hashCode() : 0);
119     }
120
121     public void fix() {
122         fixedBias = bias;
123         fixedInputs = new HashMap<Neuron, Double>();
124         fixedInputs.putAll(inputs);
125     }
126
127     public void restore() {
128         bias = fixedBias;
129         inputs.clear();
130         inputs.putAll(fixedInputs);
131     }
132
133 }
```

A.3 File: ActivationFunction.java

```
1 package weka.classifiers.mcz;
2
3 import java.io.Serializable;
```

```
4
5  public interface ActivationFunction extends Serializable {
6
7      double activate(Neuron neuron);
8
9  }
```

A.4 File: LinearFunction.java

```
1  package weka.classifiers.mcz;
2
3  import java.util.Iterator;
4  import java.util.Map;
5
6  public class LinearFunction implements ActivationFunction {
7
8      public double activate(Neuron neuron) {
9          Map<Neuron, Double> inputs = neuron.getInputs();
10         Iterator<Neuron> inputNeurons =
               inputs.keySet().iterator();
11         double sum = neuron.getBias();
12         while(inputNeurons.hasNext()) {
13             Neuron input = inputNeurons.next();
14             sum += input.getOutput() * inputs.get(input);
15         }
16
17         if(Double.isNaN(sum)) {
18             sum = 0;
19         }
20
21         if(sum>0) {
22             return sum;
23         }
24
25         return 0;
26     }
```

```
27
28      static public double calc(double input) {
29          if(input>0) {
30              return input;
31          }
32          return 0;
33      }
34  }
```

A.5 File: SigmoidFunction.java

```
1  package weka.classifiers.mcz;
2
3  import java.util.Iterator;
4  import java.util.Map;
5
6  public class SigmoidFunction implements ActivationFunction {
7
8      public double activate(Neuron neuron) {
9          Map<Neuron, Double> inputs = neuron.getInputs();
10         Iterator<Neuron> inputNeurons =
               inputs.keySet().iterator();
11         double sum = neuron.getBias();
12         while(inputNeurons.hasNext()) {
13             Neuron input = inputNeurons.next();
14             sum += input.getOutput() * inputs.get(input);
15         }
16
17         if(Double.isNaN(sum)) {
18             sum = 0;
19         }
20
21         if (sum<-45) {
22             return 0;
23         } else if (sum>45) {
24             return 1;
```

```
25          }
26          double result = 1 / (1 + Math.exp(-sum));
27          return result;
28      }
29
30      static public double calc(double input) {
31          if (input<-45) {
32              return 0;
33          } else if (input>45) {
34              return 1;
35          }
36          return 1 / (1 + Math.exp(-input));
37      }
38  }
```

A.6 File: NetworkParticle.java

```
1  package weka.classifiers.mcz;
2
3  import net.sourceforge.jswarm_pso.Particle;
4
5  public class NetworkParticle extends Particle {
6
7      static private int dimension = 10;
8
9      public NetworkParticle() {
10         super(dimension);
11     }
12
13     public NetworkParticle(int dimension) {
14         super(dimension);
15         NetworkParticle.dimension = dimension;
16     }
17
18 }
```

A.7 File: StatisticUtils.java

```
1  package weka.classifiers.mcz;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.List;
6
7  public class StatisticUtils {
8
9      private HashMap<Integer, List<Double>> activations;
10
11     public StatisticUtils() {
12         activations = new HashMap<Integer, List<Double>>();
13     }
14
15     public void add(int id, double output) {
16         if(activations.containsKey(id)) {
17             List<Double> aList = activations.get(id);
18             aList.add(output);
19         } else {
20             List<Double> aList = new ArrayList<Double>();
21             aList.add(output);
22             activations.put(id, aList);
23         }
24     }
25
26     public double calculateStandardDeviation(int id) {
27         List<Double> aList = activations.get(id);
28         double average = 0;
29         for(int i=0; i<aList.size(); i++) {
30             average += aList.get(i);
31         }
32         average = (double) (average / aList.size());
33         double standardDeviation = 0;
34         for(int i=0; i<aList.size(); i++) {
```

```
35              standardDeviation += Math.pow(average-aList.get(i),
                    2);
36          }
37          standardDeviation = Math.sqrt(standardDeviation);
38          return standardDeviation;
39      }
40  }
```