Learning Clasiffier Systems with Hebbian Learning for Autonomus Behaviors

Marco Ramos[1](✉), Vianney Muñoz-Jiménez[1], and Félix F. Ramos[2]

[1] Universidad del Estado de México, Toluca, Mexico
{maramosc,vmunozj}@uaemex.mx
[2] CINVESTAV del IPN, Guadalajara, Jalisco, Mexico
framos@gdl.cinvestav.mx

Abstract. One of the main characteristics of multi-agent systems is the ability to solve problems achieving objectives. This is possible because of the learning mechanisms that are embedded in the systems and go from neural networks up to vector support machines. Agent-based systems stand out for their autonomy and adaptation of dynamic conditions of the environment. This article presents the Hebbian theory, which is one of the learning methods from the neuroscience field. A particularity presented by the Hebbian theory from the computer since field perspective is the primary mechanism of synaptic plasticity where the value of a synaptic connection increases if neurons on both sides of a said synapse are activated repeatedly, creating a new one simultaneously. This mechanism is integrated into the Learning Classifier Systems (LCS) to validate its effectiveness in the solution task, and can be used in multi-agent systems.

Keywords: Hebbian theory · Learning · LCS · Multi-agents

1 Introduction

One of the main tasks of artificial intelligence is to be able to create artefacts that show a very similar conscience to that of their creators [11]. It talks about the human being per se. In this sense, it is currently looking for the autonomous generation behavior rather than the optimization of systems. Multi-agent systems allow solving problems using different approaches of artificial intelligence, such as neural networks, fast learning, machine learning, and others. Autonomy and adaptation to the environment presented by the multi-agent systems is possible because of the architecture that makes them up, through the generation mechanism of new knowledge and negotiation protocols that lead them to solve problems and achieve objectives [15].

An important feature of these agents is the way they react immediately to environment changes due to the sensors that are constantly monitoring. This activity is possible because of the learning system that is embedded in the system and because of the way it processes information from the environment [10,15]. Current techniques solve problems for which they were created but do not respond efficiently to behaviors or changes in beliefs to help in human beings' tasks.

This article deals basically with Learning Classifier Systems (*LCS*) [7] because of their features. This means, *LCS* learn and adapt from the environment. These qualities make them highly reactive, and in the event the systems are not able to find an answer within the environment, they do not get blocked, allowing more information to converge towards the best solution.

The most advanced *LCS* have a genetic algorithm that allows them to be evolutionary and a learning algorithm that adds new knowledge to the database. The modularity *LCS* allow modifying the type of learning, and it is in this sense that the Hebbian learning was implemented in *LCS* [5]. This learning comes from the neuroscience and is based on the plasticity of the connections of neurons [6,13].

2 Learning Classifier Systems (*LCS*)

John H. Holland [7], describes the *LCS* as the framework that uses a genetic algorithm to study learning-based systems in rules and the "condition/action" pair.

The set of *rules* of action is built using the following notation: # Symbol of *"don't care"* or *"doesn't matter"*, ? is the symbol *"fits all"*, 0 = false, 1 = true. The symbols # and ?, are used interchangeably for purposes of the implementation in the *LCS*, because in both cases, they act as wild cards that accept any allowed value in the environment [7,8].

In Fig. 1, the general architecture of *LCS* can be observed. Considering this architecture as the base, various adjustments and improvements have been made, and authors such as Martin Butz, Wilson, Stolzmann, and Goldberg, focus on the learning reward processes or the error prediction [8].

This article presents an adaptation of the *LCS* architecture that incorporates the Hebbian learning.

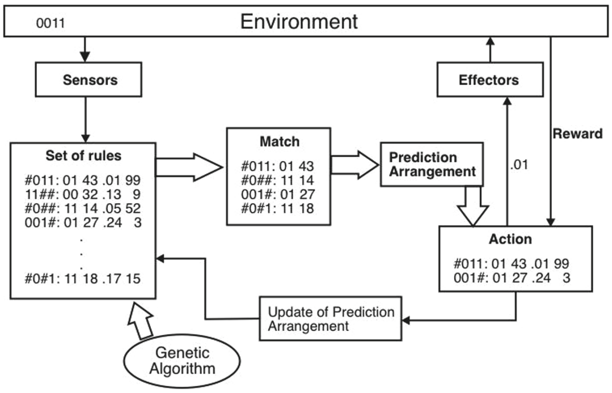

Fig. 1. General architecture of *LCS*

3 Hebbian Learning

Hebbian learning comes from the Donald O. Hebb' postulate [3] studying the neuronal function of the human brain and cognition. Currently, Hebbian learning is applied to neuronal networks and other areas of technology.

Hebb's postulate states: "When an axon of a cell ***A*** is close enough to excite a cell ***B*** and persistently takes part in its activation, some growth process takes place in one or both cells, so that the efficiency of ***A***, as one of the cells activating ***B***, is increased" [3].

Hebb postulated that the weight of the synaptic connections adjusts by the correlation between the activation values of the two connected neurons. Neurons are activated by thoughts, decisions and experiences from external stimulation, and the brain checks if there has already been a similar or not stimulus and takes the characteristics of the input data generating an association between the inputs and outputs of the system.

The connection of neurons generates a synaptic weight. If a weight contributes to the activation of a neuron, then the weight increases but if the weight is inhibited, then it decreases. This way, the strength of the synapse in the brain changes proportionally according to the firing of the input and output of neurons. This process remodels the old Hebbian networks (those that are not active) or creates new networks generating neuroplasticity. The input neuron is known as pre-synaptic and the output as post-synaptic [5]. The scheme of the neuroplasticity generation is described in Fig. 2.

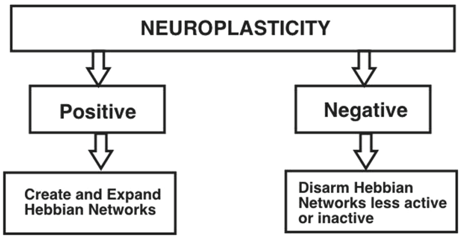
NEUROPLASTICITY
Positive
Negative
Create and Expand Hebbian Networks
Disarm Hebbian Networks less active or inactive

Fig. 2. Hebbian learning, based on neuroplasticity

Hebbian learning uses the rules of synaptic plasticity. A rule of synaptic plasticity can be seen as a set of differential equations that describes the average of tests of the synaptic weights, the role of pre-synaptic activities, post-synaptic, and other factors. In some plasticity models, the activity of each neuron is described as a continuous variable that can take positive and negative values. The value of the synaptic weight increases if the input and output values are positive or negative but will decrease if one is positive and the other negative. A variable activity that takes both values, the positive as well as the negative one, can be interpreted as the difference between the average of tests and a modified background rate, or that they are among the firing rates of two neurons that are treated as a unit [3,13].

There are several models of the Hebbian learning, though, for this research, we only considered those that are somehow normalized and can be implemented. For example, the multiplicative rule, also known as the *Oja* rule, was created for unsupervised learning. The *Oja* rule introduces the restriction dynamically, where the square of the synaptic weights is added. The use of normalization has the effect of introducing competition between neuron synapses over limited resources, which is essential for stabilization from a mathematical point of view. The Eq. (1) describes a convenient form of normalization [12].

$$w_{ij}(q) = w_{ij}(q-1) + \alpha.y_i(q).[x_j(q) - y_i(q).w_{ij}(q-1)] \tag{1}$$

where:
q is the input/output pair that is given a weight
x contains the input values
y contains the output values
α is the learning rate with values between [0,1]
y_i is the output of the neuron i
x_j is the input of the neuron j
w_{ij} is the connection weight of neuron j to neuron i

If the value of α approximates 1, the network will have a small weight but will remember little of what it learned in previous cycles. The term $\alpha y(n)x_i(n)$ represents the Hebbian modifications to the synaptic weight w_i and n the discrete-time. The equation $-y_i(q)w_{ij}(q-1)$ is responsible for the stabilization and is related to the forgetting factor or degradation rate that is generally used in learning rules. In this case, the degradation rate takes a higher value in response to a stronger response. The choice of a degradation rate has a great effect on the rapidity of convergence of the algorithm; particularly, it cannot be too large. Otherwise, the algorithm would become unstable. In practice, a good strategy is to use a relatively large value of α at the beginning to ensure an initial convergence, and gradually make α decrease until the desired precision is achieved [12]. The magnitude of the weights restricted to 0, which corresponds to having no weight, and 1, which corresponds to the input neuron with any weight; this done so that the weight vector is 1 [9].

The ***instar*** rule is an improvement to the Hebb rule with degradation [4] that prevents weights from degrading when a stimulus is not found. *Instar* is considered a neuron with a vector of inputs, and it is the simplest network capable of having pattern recognition. *Instar* is active as long as the internal product of the vector of weights and the input is greater than or equal to $-b$, where b is considered as the "trend" and must have a value between 0 and 1. This rule belongs to unsupervised learning and works locally. Unlike Hebb's rule with degradation, it only allows the degradation of weight when the neuron *instar* is active. That is, when $a \neq 0$. To achieve this, a term that avoids forgetting is added, which is proportional to $y_i(q)$ [14]. This is seen in the following Eq. (2).

$$w_{ij}(q) = w_{ij}(q-1) + \alpha.y_i(q).(x(q) - w_{ij}(q-1)) \tag{2}$$

Where:
$x(q)$ contains all the values of the output vector. α takes values between 0 and 1 to ensure that weights values increase, if α reaches 0 it becomes unstable and, on the contrary, if it gets close to 1, the network begins to forget the old entries quickly and will only remember the most recent patterns allowing to have a limited growth of the weight matrix. The maximum value in weight w_{ij}^{max} is determined by α, which happens when y_i and x_j have values = 1 for all q, which maximizes learning.

Another rule that literature classifies as stable is the ***outstar***, which has a scalar input and a vector output. This can improve the call pattern by associating a stimulus with the response vector. The *outstar* rule, unlike to the *instar*, makes the degradation of the weight proportional to the x_j network entry. The Eq. (3) shows the rule of *outstar* [4]:

$$w_{ij}(q) = w_{ij}(q-1) + \alpha.(y(q) - w_{ij}(q-1)).x_j(q) \tag{3}$$

Where: $y(q)$ contains all the values of the output vector.

The rule *outstar* belongs to the supervised learning. In it, learning occurs whenever $x_j \neq 0$. If the rule has learned something, the column w_{ij} approaches the output vector, that is, when some weights are equal to those of the desired output [2].

In this first research work, standardized Hebbian learning is used to evaluate the feasibility of implementation in a *LCS*, and the tests are carried out in environments that the classifiers community uses.

4 Hebbian Learning Implementation in a *LCS*

The base architecture used is the GXCS classifier [10]. The modules that were added for the incorporation of the Hebbian learning are shown (shaded) in Fig. 3. These specific modules allow the classifier to calculate a vector of weights, composed of environmental conditions, as well as to evaluate each of these conditions to learn about the dynamics of the environment, contrary to the array of predictions used in traditional learning [10]. A module for the values of x and y has also been integrated, which is directly assigned to the conditions of the environment that are embedded in each of the competing rules to perform a specific action. The values that x and y receive, can be scalar or vector according to a Hebbian condition such as *Oja*, *instar* and *outstar*, which are stable rules. That is, they do not generate undesirable states in the [1] environment. A degradation rate is used to evaluate the permanence or erasure of the rules that are within the set of actions by controlling the number of rules produced by the interaction of the environment, thus avoiding over-exploitation of rules that could slow down the system.

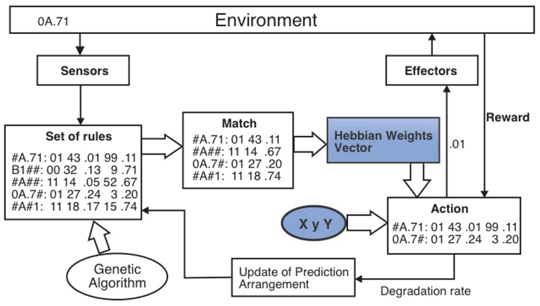

Fig. 3. GXCS-H architecture

The initialization of the synaptic weights vector, as well as the introduction of the Hebbian learning equation represented with $H(e, a)$. The degradation rate is the parameter of deletion or preservation of any specific rule.

The assigned value for the degradation rate is 0.5 [1]. If values outside the range of $(0, 1)$ are used in the area of neural networks, the stability of the Hebbian weights gets broken. To avoid this, Hebbian rules that have these parameters are used, so that they stabilize the equation of the basic rule of Hebb [1,5] and help convergence within the classifiers. These parameters allow the values to be in the range between 0 and 1, values that can be manipulated in the equations, and above all, can be used and read to generate an expected output. The stable rules are: *Multiplicative normalization rule or* ***Oja***, ***Instar*** *rule* and ***Outstar*** *rule.*

The ***instar***, ***outstar*** **and multiplicative** (***Oja***) rules contain a decay parameter that controls the degradation of weights. This parameter prevents the elimination of rules that are useful in future environmental conditions and allows the elimination of those that have not participated or that have an aptitude with a value close to zero, thus avoiding saturation of memory.

4.1 Multiplicative Normalization Rule

This rule takes the information from the synapse that is being modified, preventing Hebbian weights from growing indefinitely. The product of the value of the output, associated with the rule and the Hebbian weight of each rule, is responsible for the stabilization of the weights. This product is known as a forgetting factor or degradation rate and it allows eliminating only the rules that have not been used or have low values, close to zero or even zero [5].

A fragment of the GXCS architecture is shown in Fig. 4 with the integration of the Hebbian learning where the vector of weights that have been assigned to the rules, can be observed. The vector will be in the set of actions and are taken for recalculating the values of inputs and outputs of each of them. The value associated with the entering environment rule is stored in a x scalar. The same applies to the output y that takes its value randomly. Both values fluctuate between 0 and

1. The degradation rate ***decay*** is also calculated with random values that are between 0 and 1 and is assigned to an element of the set **A** of rules and actions. If necessary, the genetic algorithm used by the *LCS*, included in this version, is invoked. Once the values of the weights **w** (the degradation rate ***decay***, the inputs **x** and the outputs **y**) are obtained, the value of the following weight is calculated using equation (2) of the multiplicative normalization (or of *Oja*).

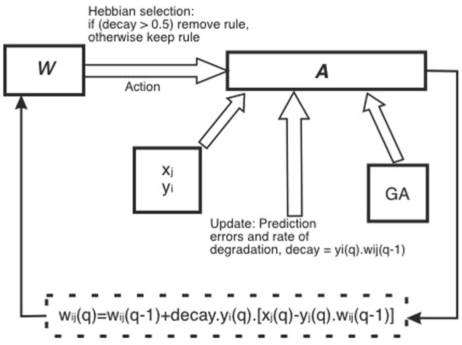
Hebbian selection: if (decay > 0.5) remove rule, otherwise keep rule
W
A
Action
xj
yi
GA
Update: Prediction errors and rate of degradation, decay = yi(q).wij(q-1)
wij(q)=wij(q-1)+decay.yi(q).[xj(q)-yi(q).wij(q-1)]

Fig. 4. Standardization of architecture

4.2 Rule of *Instar*

This rule prevents weights from degrading when there is not enough stimulus when calculating the values of inputs and outputs. It allows degradation only if the rule is active; that is, that the output has a value higher than 0.5. The weight values are learned at the same time that the old values of the inputs and outputs are degraded, as well as the Hebbian weight. All of them are assigned to each set of rules and actions within the classifier. The vector of weights has similar values to those of the vector of entries when the rule has an associated value to the aptitude greater than the average. In this case, if the degradation rate is equal to 0, the current weight is equal to the new weight assigned to the rule. If the degradation rate gets a value of 1, then the new Hebbian weight has its maximum value; if the degradation rate has a value of 0.5, then the value of the vector of entries is added to the Hebbian weight. The process is very similar to that of the *Oja* rule. The main difference is that the value of y of the outputs, associated with the actions taken from a set of values that belong to an output vector, takes the most suitable one. The second difference concerning the *Oja* rule is that the *decay* degradation rate is not taken directly from the equation, but is calculated pseudo-randomly with values that fluctuate between 0 and 1.

4.3 Rule of *Outstar*

In this case, the input is a scalar value and the output a vector. The value of the Hebbian weight gets close to the values found in the output vector when the ruler has learned something, acquiring the same value as the output. The weight changes when the ruler has a greater or equal aptitude than the average. The difference concerning the *instar* rule is that the value of x of the entries associated with the rules are taken from a set of values belonging to a vector of entries, from where it takes the fittest. The calculation of the degradation rate *decay* is done in the same way as in the *instar* rule.

5 Experimentation

The tests performed at the GXCS, are based on the environments proposed by the *LCS* community. These are **multiplexer** and **woods** since both allow obtaining an easy to interpret solution. The multiplexer environment is known as static and woods as dynamic. The woods environment is used to test the *Oja* rule with supervised learning and *outstar*. In the multiplexer environment, the multiplicative normalization rule (*Oja*) was tested for unsupervised learning and *instar*. The **woods** environment can be seen as a maze where there are obstacles, and an agent that runs through it to reach a goal and the goal itself. Figure 5 illustrates an example of how a maze can be developed. The agent is shown as a person who will travel through the maze to reach the final goal, which, in this case, is exemplified with a gift. The person has to go through the shortest path avoiding all the obstacles in it.

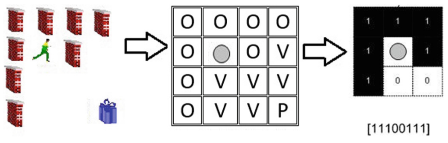

Fig. 5. Woods environment

This environment, computationally speaking, can be seen as an array with values, where each obstacle is assigned with a value of 1 and the cell that has no obstacle with the value of 0. A chain of zeros and ones is arranged by position. Each position in the chain indicates if there is an obstacle or if the way is free. For example, the first position of the chain can indicate the position above the agent, where a number one indicates that there is an obstacle. In the example, the cell that is just above the agent was taken as the first to be added to the rule. From there on, values are taken clockwise. As a result of this example, the generated rule is 11100111, and it can be observed that there are obstacles just above the agent, in the upper right corner and on the right side of it. Then two zeros can be seen, which indicates that there are no obstacles in the lower right corner and precisely below it, and after, there are more obstacles in the next three positions that are at its left side.

The **multiplexer** environment can be illustrated using logic gates where there are several inputs, in this case six, and only one output. It can be observed in Fig. 6, that the entries are associated with the variables x_i where $i = 0, 1, 2, ..., 5$. These inputs enter a circuit with accommodated logic gates similar to the Boolean multiplexer, and the values get mixed with logical operators of AND and OR, achieving a single output F_6.

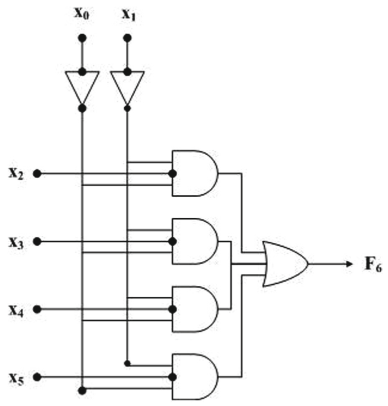

Fig. 6. Multiplexer environment

For a LCS, the multiplexer inputs are part of the rule to be evaluated, where each input represents a particular parameter of that rule, and the output is the action associated with the entire rule. This data is entered into the multiplexer, which evaluates as if it had logic gates and allows the values to be assigned to the corresponding rule/action pair.

Within the first tests, comparisons were made between a GXCS and a GXSC-H. The latter is the one that incorporates the Hebbian learning. The first environment was the multiplexer. After 20,000 rules with the three Hebbian slopes, as seen in Figs. 7, 8 and 9, we analyzed, the performance in the first instance. This one was calculated from the values obtained from the weight vector, the reward, and the aptitude of each rule. The degradation or decay rate allowed us to eliminate the rules that had a lower weight and therefore were not candidates to be taken into account for future calculations. At the implementation level, we generated an output file to store the test results: performance, the number of tests performed, a margin of error in the prediction, and the number of elements for each iteration. It was observed, as shown in the graphs in Figs. 7, 8, and 9 that there is a higher performance. However, a more significant number of tests is necessary to validate these results.

The first tests were done with an initial weight very close to zero, with pseudo-random values between 0.5 and 0.6. Still, it is remarkable that better results were obtained with a wider range, then it was decided to use the range from 0.1 to 0.6, improving the performance. This calculation was used for the three tested Hebbian slopes since better values were found in the weights and the performance of the classifier.

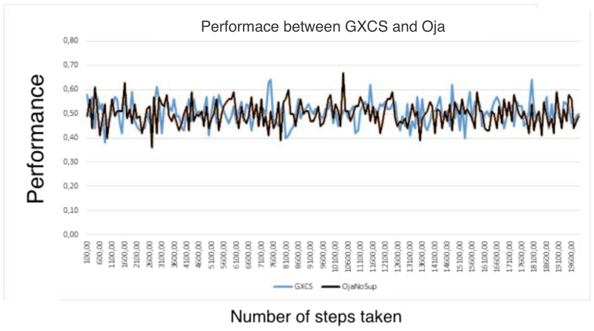

Fig. 7. Comparison between GXCS and GXCS-H with *Oja*

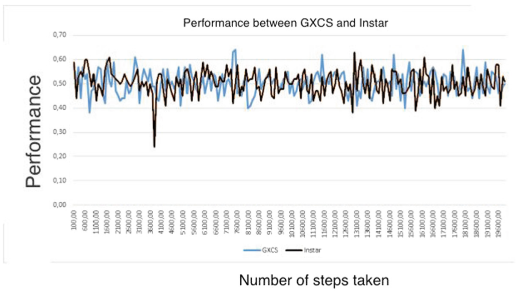

Fig. 8. Comparison between GXCS and GXCS-H with *Instar*

The variable that was used for the degradation rate started with values close to zero. The used range was between 0.1 and 0.7, and then it was applied to erase the rules. Initially, the rules that had a degradation rate greater than 0.5 were deleted. However, this caused the system not to erase enough rules, since there were very few with a rate with those characteristics, and therefore saturated the population as a whole. It was decided to delete rules with degradation values of 0.25. This was applied to the three Hebbian rules that we tested in the system.

The observed behaviors between GXCS and GXCS-H with the rule of *Oja* (Fig. 7), are perceived similarly, especially at the beginning of the iterations. But gradually, GXCS-H begins to stabilize better than the GXCS, increasing the number of steps. More rules are added through the genetic algorithm, and those with a degradation rate greater than 0.5, are eliminated. This indicates that the capability and the Hebbian weights increase and stabilize, which means that more rules are apt to solve a problem through the actions assigned to it.

Figure 8 points out that the *instar* rule has slightly smaller behaviors than GXCS. In this example, it can be seen that the *instar* rule presents a fall in performance in step 3500. This is because a new rule was created that had not been tested and, therefore, it was given Hebbian weights and minimal rewards at the beginning. Later it can be observed that the GXCS-H with the *instar*

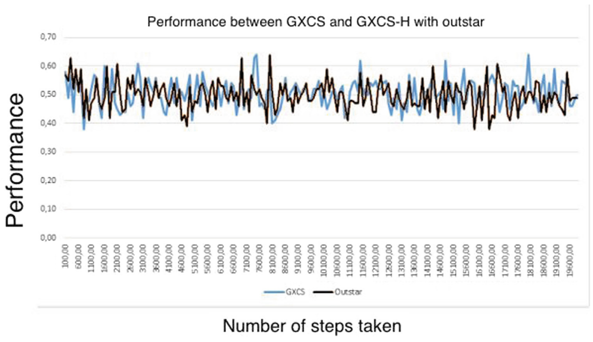

Fig. 9. Comparison between GXCS and GXCS-H with *outstar*

rule is retrieved and the rules are slightly more stable. In general, the *instar* rule generates slightly higher returns than those generated by the GXCS. The same can be seen for the *outstart* rule in Fig. 9.

The results from the incorporation of the Hebbian learning compared to the results of the GXCS show an improvement in capability, reward, and punishment for each condition-action pair. The integration of this learning of the *LCS* gives us the possibility of having an easier adaptation to environmental conditions.

On the other hand, the quantitative part in the use of LCS with traditional learning or with Hebbian learning is a subject of study in future implementations. Where we can carry out simulations with agents, for the moment, we are comparing the convergence and stability of the classifier so that it does not overflow with the generation of rules or their sub-generalization.

6 Conclusion and Future Work

One of the difficulties found within some aspects of the Hebbian learning is that the variation in the number of uncontrolled rules quickly causes instability in the values of the weights when there are too many rules or weights with negative or substantial values. The rules of multiplicative normalization (also called *Oja*, *instar* and *outstar*) are used to solve the problem of instability allowing more stable rules. The integration of the Hebbian learning in a *LCS* is possible, and allows a much better learning of the conditions of the surroundings without losing the re-activity of the system in terms of uncertainty. The use of a degradation variable allows the *LCS* to have only rules that are apt for giving a solution, and the sub-generalization of rules is reduced. The results, in the suggested environments by the community, are satisfactory. However, it is necessary to test other types of environments that have a greater complexity to corroborate results.

References

1. Brodeur, S., Rouat, J.: Regulation toward self-organized criticality in a recurrent spiking neural reservoir. In: Villa, A.E.P., Duch, W., Érdi, P., Masulli, F., Palm, G. (eds.) ICANN 2012. LNCS, vol. 7552, pp. 547–554. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33269-2_69
2. Carpenter, G.A.: A distributed outstar network for spatial pattern learning. Neural Networks **7**(1), 159–168 (1994)
3. Dayan, P., Abbott, L.: Theoretical neuroscience: computational and mathematical modeling of neural systems. J. Cogn. Neurosci. **15**(1), 154–155 (2003)
4. Demuth, H.B., Beale, M.H., De Jess, O., Hagan, M.T.: Neural Network Design. Martin Hagan, Stillwater (2014)
5. Grabner, C.A.L.: Neuroplasticidad y redes hebbianas: las bases del aprendizaje. Asociación Educar, Buenos Aires (Argentina) (2011)
6. Hebb, D.O.: The Organization of Behavior: A Neuropsychological Theory. Psychology Press, Hove (2005)
7. Holland, J.H., et al.: What is a learning classifier system? In: Lanzi, P.L., Stolzmann, W., Wilson, S.W. (eds.) IWLCS 1999. LNCS (LNAI), vol. 1813, pp. 3–32. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45027-0_1
8. Lanzi, P.L., Stolzmann, W., Wilson, S.W. (eds.): IWLCS 1999. LNCS (LNAI), vol. 1813. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45027-0
9. Oja, E.: Simplified neuron model as a principal component analyzer. J. Math. Biol. **15**(3), 267–273 (1982). https://doi.org/10.1007/BF00275687
10. Ramos Corchado, M.A.: Etude et proposition d'un système comportemental autonome anticipatif. Thèse de doctorat, Université de Toulouse, Toulouse, France, (décembre 2007)
11. Rennard, J.P.: Vie artificielle. Où la biologie rencontre l'informatique. Vuibert informatique (2002)
12. Rodríguez-Piñero, P.T.: Redes neuronales artificiales para el análisis de componentes principales. la red de oja. Vniversitat de Valéncia (2000)
13. Trappenberg, T.P.: Fundamentals of Computational Neuroscience. Oxford Press, Oxford (2010)
14. Wilamowski, B.: Neural networks and fuzzy systems. In: The Microelectronic Handbook (2002)
15. Wooldridge, M.: An Introduction to Multi-agent Systems. Wiley, Hoboken (2002)