Classifier Systems that Compute Action Mappings

Pier Luca Lanzi[*†] and Daniele Loiacono[*]
[*]Artificial Intelligence and Robotics Laboratory (AIRLab), Politecnico di Milano, I-20133, Milano, Italy
[†]Illinois Genetic Algorithm Laboratory (IlliGAL)
University of Illinois at Urbana Champaign, Urbana, IL 61801, USA
pierluca.lanzi@polimi.it, loiacono@elet.polimi.it

ABSTRACT

The learning in a niche based learning classifier system depends both on the complexity of the problem space and on the number of available actions. In this paper, we introduce a version of XCS with computed actions, briefly XCSCA, that can be applied to problems involving a large number of actions. We report experimental results showing that XCSCA can evolve accurate and compact representations of binary functions which would be challenging for typical learning classifier system models.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics Based Machine Learning, Learning Classifier Systems

General Terms

Algorithms

Keywords

LCS, XCS, Action Mappingss.

1. INTRODUCTION

Learning Classifier Systems combine reinforcement learning and genetic algorithms to solve problems. They maintain a population of condition-action-prediction rules called classifiers which represents the current solution to the target problem. Each classifier represents a small portion of the overall solution. The condition c identifies a part of the problem domain. The action a represents a decision on the subproblem identified by the condition c. The prediction p provides an estimate of how valuable action a is in terms of problem solution on the subproblem identified by condition c. In learning classifier systems, the genetic component works on classifier conditions searching for an adequate decomposition of the target problem into a set of subproblems; the reinforcement component works on classifier predictions to estimate the action values in each subproblem.

Research in knowledge representation for learning classifier systems has been usually focused on the classifier conditions and on the development of new representations which could improve the problem decomposition (e.g., [24, 8, 15]). Recently, classifier prediction has also received attention and the idea of *computed prediction* has been introduced to improve the estimation of classifiers in terms of problem solution [25]. Until now however only few works has considered the action side of classifier representation (see e.g. [1, 26]). This probably because in the vast majority of applications the number of actions is usually limited between two (in typical Boolean problems) and eight (in typical maze problems). However, the learning performance of a learning classifier system depends on the number of available actions. More actions means more options to be explored for finding the solution and also a higher learning time. For instance, Butz et al. [6] proved that in XCS the population size required to support all the subsolutions to a problem grows linearly in the number of actions while the time to convergence is *exponential* in the number of available actions [5]. These bounds suggest that XCS has an inductive bias which is not well suited for problem involving a large number of actions.

In this paper we investigate the application of learning classifier systems to problem involving a large number of *discrete* actions (for continuous actions, we refer the reader to [26]). In particular, we consider the learning of binary functions, which maps bitstrings of size m into bitstrings of size n, which require 2^n discrete actions, and introduce a version of XCS [21] designed to tackle such tasks. We call our version of XCS, XCS with computed action or briefly XCSCA. Our approach is inspired by the sUpervised Classifier System (UCS) developed by Bernado et al. [2] and by previous work on computed prediction [25, 14]. From UCS [2], XCSCA borrows the idea of focusing only on the best possible action without developing a complete knowledge about the problem, i.e., without building a complete mapping as done in XCS [21]. From Wilson's computed prediction [25], XCSCA borrows the idea of replacing one classifier parameter with a parametrized function: the prediction in [25], the classifier action in our case. XCSCA dramatically reduces the search space both because (i) it focuses only on the best action for each subproblem (as done in UCS), and because (ii) it introduces generalization over the action space which reduces the search space even further.

2. XCS WITH COMPUTED ACTION

XCS with Computed Action (XCSCA) extends the typical XCS structure to tackle problems involving a large number of discrete actions. It is inspired to UCS [2], the extension of XCS to supervised learning, and to XCSF [25], the extension of XCS with computed prediction. XCSCA borrows from UCS the idea of having the correct action (the correct output) as a part of the system input, from XCSF the idea of replacing a classifier parameter with a parametrized function. Accordingly, XCSCA does not learn a complete mapping of the target problem but, as UCS, it focuses on the correct output which in XCSCA is computed from the current input.

Classifiers. In XCSCA, classifiers consist of a condition, which specifies the set of inputs that the classifier matches, and four parameters: the vector $\mathbf{w}$, used to compute a *discrete* classifier action; the error ϵ, that estimates the error affecting the computed action; the fitness F that estimates the accuracy of the computed action; the numerosity *num*, a counter used to represent different copies of the same classifier. In XCSCA, classifiers have no action. The classifier action is computed using a function $a_f(x, \mathbf{w})$, parametrized by a vector $\mathbf{w}$, that computes the classifier *discrete* action based on the current input x and the parameter vector $\mathbf{w}$ associated to each classifier. In XCSCA, classifiers have no prediction since, as in UCS, there is no incoming reward.

Performance Component. XCSCA works similarly to UCS [2]. At each time step t, XCSCA receives as input the current input example x_t and the associated output y_t. XCSCA builds a *match set* [M] containing the classifiers in the population [P] whose condition matches the current input x_t. If [M] is empty *covering* takes place and a new classifier that matches the current input is inserted in the population. The covering classifier is generated as follows: the classifier condition is created as in XCS [2, 21], the parameter vector $\mathbf{w}$ is initialized with zero values, while all the other parameters are initialized as in XCS [21, 10].

At this point, the behavior of XCSCA is different depending on whether the system is working in *learning mode* or in *testing mode*. During learning, XCSCA exploits the incoming information about the desired output y_t to update the classifiers in [M] following the procedure described below. During testing, for each classifier *cl* in [M], XCSCA computes the current (*discrete*) action $a_f(x_t, cl.\mathbf{w})$. Then, for each action a computed from the classifiers in the match set [M], XCSCA computes the *classification accuracy* of action a for the input x_t, $C(x_t, a)$, as the average fitness of the classifiers in [M] that advocate action a, i.e.,

$$C(x_t, a) = \frac{\sum_{[M](x_t,a)} cl.F}{|[M](x_t, a)|} \qquad (1)$$

where $[M](x_t, a)$ is the set of classifiers in [M] that for the input x_t advocate action a. Finally, XCSCA selects the action with the highest classification accuracy.

Classifier Update. XCSCA works in a supervised fashion as UCS [2] and thus it has no incoming reward. In UCS, the incoming correct output y_t is used to build the set [C] of correct classifiers and the set [!C] of incorrect classifiers. In XCSCA, the same information is exploited to train the classifiers in the match set [M]. The classifier error is updated first, then the prediction, and finally the classifier fitness.

To update classifier error, we define an error function $\varepsilon_f(x_t, y_t, a)$ that measures the accuracy of the classifier action. The error function $\varepsilon_f(x_t, y_t, a)$ takes as input the current example $\langle x_t, y_t\rangle$ and the proposed output a and returns a measure of the error affecting the action prediction a with respect to the target output value y_t. In all of the experiments we performed we used a very simple error function that returns 0 if the classification is correct (i.e., $y_t = a$) and 1000 if the classification is incorrect (i.e., $y_t \neq a$). This error function has the same range as the classifier error in XCS and thus it allows us to use for XCSCA the typical XCS settings for ϵ_0. However, more complex definitions of $\varepsilon_f(x_t, y_t, a)$ may be used in more complex tasks. Given the error $\varepsilon_f(x_t, y_t, a)$, the error ϵ of the classifiers in [M] is updated as, "$cl.\varepsilon \leftarrow cl.\varepsilon + \beta(\varepsilon_f(x_t, y_t, a) - cl.\varepsilon)$." Then weight vectors of the classifiers in [M] are updated according the correct incoming action. This training phase depends on the action function $a_f(\mathbf{x_t}, cl.\mathbf{w})$ used and it is illustrated in details in Section 4. Finally, classifier fitness is updated from the classifier error as in XCS [21].

Discovery Component. The genetic algorithm works as in XCS [24]. On a regular basis depending on the parameter θ_{ga}, the genetic algorithm is applied to classifiers in [M]. It selects two classifiers with probability proportional to their fitness, copies them, and with probability χ performs crossover on the copies; then, with probability μ it mutates each allele. Crossover and mutation work as in XCS [22] except for the parameter vector $\mathbf{w}$. The parameter vectors $\mathbf{w}$ of offspring classifiers can be either copied from the parents (as done in the experiments presented here) or alternatively they may be obtained by recombining the parents vectors. The resulting offspring are inserted into the population and two classifiers are deleted to keep the population size constant.

3. RELATED WORK

To our knowledge, the first notion of computed action, although not called with this name, was given by Ahluwalia and Bull [1] who extended ZCS [20] with actions represented using GP expressions and applied the new model to a well-known classification task. In [1] action are computed in that the GP expression that represents the classifier action is then evaluated over the current inputs. With respect to our work, the approach is limited to a two value action and its scalability of multiple action is unknown; in addition, the function is fully evolved, there is no training of the action function as in our approach. Later, O'hara and Bull [17] introduced the neural classifier system X-NCS which has no condition-action structure but both the classifier condition and the classifier action are represented by a neural network whose weights are evolved by a genetic algorithm. Also in this case, the classifier action is computed based on the current input, though it is not learned but it is evolved through the evolution of the network weights. This because X-NCS is not restricted to supervised learning problems as XCSCA or UCS but it may also be applied to typical multistep (grid) problems. As in [1], also X-NCS has been usually applied to problems involving a small number of actions 2 or 8 depending on the type of problem considered so the scalability to more outputs is not clear. We refer the reader to the group main page (`http://www.csm.uwe.ac.uk/lcsg`) and to the many published papers available online for further details.

The concept of having the action information provided by the environment in a supervised learning fashion was first introduced by Bernado et al. [2]. In UCS the classifier action is created during covering using the incoming correct action, then classifier actions can be modified through mutation. In contrast, in XCSCA, the classifier action is trained everytime that the classifier enters in the match set so that it does not have the concept of *correct set* [C] and *incorrect set* [!C] [2]. UCS also uses a different definition of classifier accuracy and classifier fitness. In UCS, the classifier accuracy is computed as an average over the input-output examples that the classifier experienced.

Essentially, XCSCA uses the same principle adopted in computed prediction [23, 25]. In fact computed action is basically obtained by restricting the choice of the parametrized function to a range adequate for discrete classifier actions. Recently, Wilson [26] has extended classifier actions to realm of continuous values. He proposed three architectures inspired to his previous work on computed prediction [23]: one based on interpolation, one based on an actor-critic paradigm, and one, apparently the most promising, on treating the action as a continuous variable homogeneous with the input.

4. ACTION FUNCTIONS

Several functions may be used to compute discrete classifiers actions depending on the problem being solved. With Boolean functions [21], any function returning two values is feasible. With binary functions, when more actions are involved, more elaborate solutions need to be considered.

Constant Action Function. In the simplest case, the classifier action can be represented by a parameter which is set using the desired output y_t. The vector $\mathbf{w}$ thus consists of just one element, w_0, which represents the classifier action, i.e., $a_f(x_t, \mathbf{w}) = w_0$, while the training of a classifier cl simply sets the value of the classifier action to the desired output y_t. XCSCA with a constant action function works similarly to UCS [2], but while in UCS [2] the classifier action is set during covering and it can be modified through mutation, in XCSCA the classifier action is changed everytime the classifier enters the match set during training.

The Perceptron takes the current input $\mathbf{x_t}$ and outputs 0 or 1 through a two stage process: first the linear combination $\mathbf{wx_t}$ of the inputs $\mathbf{x_t}$ and of the weight vector $\mathbf{w}$ is calculated, then the perceptron outputs 1 if $\mathbf{wx_t}$ is greater than zero, 0 otherwise. The original binary input $\mathbf{x_t}$ is enriched with the usual constant input x_0 and, since zero values for inputs must be generally avoided [11], binary inputs are mapped into integer values x_i by replacing the zeros and ones in $\mathbf{x_t}$ with -5 and +5 respectively. Given the current input $\mathbf{x_t}$ and the desired output y_t, the weight w_i associated to the integer input x_i is updated as [19]:

$$w_i \leftarrow w_i + \eta(y_t - o_t)x_i \tag{2}$$

where o_t is the perceptron output for input $\mathbf{x_t}$, and η is the usual learning rate.

The Sigmoid is the obvious extension of the perceptron and it is one of the most typical activation functions used in neural networks [11]. The action function in this case is defined as:

$$a_f(\mathbf{x_t}, \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{wx_t}}} \tag{3}$$

Classifier weights are updated through gradient descent as follows,

$$w_i \leftarrow w_i + \eta(y_t - a_f(\mathbf{x_t}, \mathbf{w}))[\frac{\partial f(z)}{\partial z} + \epsilon]x_i$$

where $z = \mathbf{w_t x_t}$ and the term "$\partial f(z)/\partial z$" is the usual adjustment factor to avoid flat spots [11].

Neural Networks. When the problem involves many actions, we can either employ an array of simple Boolean functions, such as for instance an array of sigmoids, or a more adequate solution such as a neural network. The action function $a_f(\mathbf{x_t}, \mathbf{w})$ is now computed by neural network with n inputs one for each component of $\mathbf{x_t}$, h hidden nodes, and as many outputs as required by the problem. The activation functions for both hidden and output nodes are the usual sigmoid [11]. Given the current input $\mathbf{x_t}$ and the desired output $\mathbf{y_t}$, the network weights are updated using online backpropagation [11].

5. DESIGN OF EXPERIMENTS

Each experiment consists of a number of problems that the system must solve. Each problem is either a *learning* problem or a *test* problem. During *learning* problems, the system exploits all the incoming information, the input $\mathbf{x_t}$ and the desired output $\mathbf{y_t}$, to train the classifiers in the match set, thus no action is performed. During *test* problems, the system always selects the action with highest classification accuracy for the input $\mathbf{x_t}$ and no update is performed. The genetic algorithm is enabled only during *learning* problems, and it is turned off during *test* problems. The covering operator is always enabled, but operates only if needed. Learning problems and test problems alternate. The performance is computed as the percentage of correct answers during the last 100 test problems. All the reported statistics are averages over 20 experiments.

6. BOOLEAN FUNCTIONS

In the first set of experiments, we applied XCS with computed actions to the learning of Boolean functions, a typical testbed for learning classifier systems. For this purpose, we selected a typical functions from the literature, the Boolean multiplexer [21, 22].

Boolean Multiplexer. These are defined over binary strings of length n where $n = k + 2^k$; the first k bits, $x_0, \ldots, x_{k-1}$, represent an address which indexes the remaining 2^k bits, $y_0, \ldots, y_{2^k-1}$; the function returns the value of the indexed bit. For instance, in the 6-multiplexer function, mp_6, we have that $mp_6(100010) = 1$ while $mp_6(000111) = 0$.

Experiments. At first, we compared XCS and XCSCA on the 20-multiplexer; for XCS and XCSCA the same parameter setting was used [9]: $N = 2000$, $\beta = 0.2$; $\alpha = 0.1$; $\epsilon_0 = 10$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $\theta_{del} = 50$; $\theta_{GA} = 50$; $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off. Figure 1a compares the performance of XCS with that of XCSCA with the constant action function, the perceptron, and the sigmoid. XCSCA generally reaches optimal performance faster than XCS: with the constant action function such a difference is slight but when more powerful action functions, namely the perceptron and the sigmoid, are used the difference is more visible. The 20-multiplexer is a very simple problem and the sigmoid does not provide an advantage over the percetron: in fact, the perceptron converges faster due to its simpler update procedure. As expected, computed action allows better generalization: all the three versions of XCSCA converge faster to smaller solutions than XCS. XCSCA with the perceptron generalizes faster than XCSCA with the sigmoid because the perceptron update is faster than the sigmoid update and the two function are basically equivalent on this simple problem. In fact, the differences in Figure 1 are actually not statistically significant.

To understand how XCSCA evolve more compact solutions we analyze one of the evolved solutions. For this purpose we report in Table 1 classifiers from one of the population evolved by XCSCA with the perceptron in the first experiment on the 20-multiplexer; column "c" reports the classifier condition; column "**xw**" reports the argument of the perceptron (if $\mathbf{xw} > 0$ then the perceptron output is 1, otherwise it is 0); column "ϵ" reports the error, column "F" reports the fitness, column "*num*" reports the numerosity; variable X_i represents the multiplexer variable x_i after it has been mapped into -5/5 values; variable Y_i represents the multiplexer variable y_i after it has been mapped into -5/5 values. The population mainly consists of classifiers whose conditions have only four specific bits instead of the usual five specific bits that XCS would require to solve the same problem. Such classifiers are characterized by small weights corresponding to each input variable except for the one corresponding to the bit identified by the address bits. For instance let us consider the column "**wx**" of the first classifier in Table 1. We can distinguish three different contributions: (i) a constant term due to the first constant weight and to the variables inputs X_i, that is overall equal to $2.6 - 0.5 \cdot 5 + 0.5 \cdot 5 + 0.4 \cdot (-5) + 0.2 \cdot (-5) = -0.4$; (ii) the term due to the inputs corresponding to the bits identified by the address, i.e. $2.6Y_{12}$; (iii) a sort of *noise* that is due to all the inputs Y_i with $i \neq 12$, that ranges between -8.5 and $+8.5$, depending on Y_i values. Overall we have $\mathbf{wx} = -0.4 \pm 8.5 + 2.6Y_{12})$ that is $\mathbf{wx} \geq 4.1$ when $y_{12} = 1$ (i.e. $Y_{12} = 5$) and $\mathbf{wx} \leq -4.9$ when $y_{12} = 0$ (i.e. $Y_{12} = -5$). This results in an action equal to 1 ($a_f(\mathbf{x}) = 1$) when $y_{12} = 1$ and action equal to 0 ($a_f(\mathbf{x}) = 0$) otherwise, i.e the classifier is able to compute the correct action.

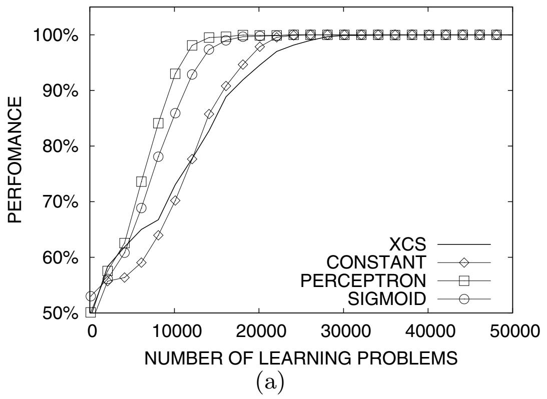
7. BINARY FUNCTIONS

We now move to problems involving many actions. For this purpose we consider binary functions which map bitstrings of size m into bitstrings of size n.

7.1 Binary Shift

We begin with a very simple binary function which can become easily challenging for a niche based learning classifier system like XCS, i.e., the binary shift. The binary shift of size m, shortly $\mathtt{shift}_m$, takes as input a binary string $\mathbf{x} = \langle x_1, \ldots, x_m \rangle$ and returns the binary string $\mathbf{y}$ of size m obtained by shifting the m input bits to the right, i.e., $\mathbf{y} = \langle 0, x_1, \ldots, x_{m-1} \rangle$. For example, if $m = 8$ and $\mathbf{x} =$11011011, then $\mathtt{shift}_8$(11011011) returns 01101101.

The binary shift is challenging both because the number of possible actions is exponential in the number of inputs (there are in fact 2^{m-1} actions) and because the usual ternary representation of classifier conditions allows only generalizations in the position m of classifier conditions. Thus, XCS requires a population of $2^{2(m-1)}$ classifiers to represent the



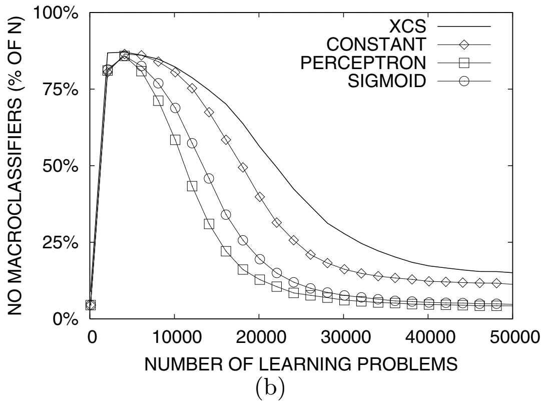


Figure 1: XCS and XCSCA applied to the 20-multiplexer: (a) performance; (b) number of macroclassifiers.

solution. However, it is rather intuitive to design an action function to solve the binary shift. The most straightforward solution consists of an array of two-valued action functions (e.g., the perceptron or the sigmoid) each one computing a separate action bit based on the current inputs. For instance, to solve the binary shift of size 8, $\mathtt{shift}_8$, the action function would consist of an array of eight perceptrons (one for each output).

Figure 2 compares (a) the performance and (b) the population size of XCS and XCSCA using a constant action, an array of perceptrons, and an array of sigmoids. For all the four models the parameters were set as follows: $\beta = 0.2$; $\alpha = 0.1$; $\epsilon_0 = 10$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $\theta_{del} = 50$; $\theta_{GA} = 50$; $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off. For XCS the population size was set to 20000 while for XCSCA was set to 2000. As the plot shows, with an array of simple two-valued action functions, XCSCA quickly reaches optimality and evolves the minimal representation possible, one classifier with a condition that matches all the possible inputs; the action function is implemented by an array of perceptrons (or sigmoids) which transfer the input bits in their final shifted position. Learning the $\mathtt{shift}_8$ with a constant action function is much simpler than learning it with XCS. XCSCA with a constant action function needs only 2^7 classifiers to represent the optimal solution. Accordingly, it slowly reaches optimality while the population slowly decreases. Although 20000

c	**xw**	ϵ	F	num
1100################	$2.6 - 0.5X_0 + 0.5X_1 + 0.4X_2 + 0.2X_3 + 0.2Y_0 + 0.1Y_1$ $+0.1Y_2 - 0.1Y_3 - 0.0Y_4 + 0.1Y_5 + 0.2Y_6 - 0.1Y_7 - 0.1Y_8 - 0.3Y_9$ $-0.1Y_{10} - 0.0Y_{11} + 2.6Y_{12} - 0.0Y_{13} - 0.1Y_{14} + 0.2Y_{15}$	0.0	1.0	147.0
1101################	$0.2 - 0.1X_0 + 0.0X_1 - 0.0X_2 + 0.0X_3 - 0.0Y_0 - 0.0Y_1$ $-0.0Y_2 + 0.1Y_3 - 0.1Y_4 - 0.1Y_5 - 0.0Y_6 + 0.1Y_7 + 0.1Y_8 + 0.0Y_9$ $-0.0Y_{10} - 0.1Y_{11} + 0.1Y_{12} + 1.9Y_{13} + 0.1Y_{14} - 0.1Y_{15}$	0.0	1.0	137.0
...	...	...	...	
0011################	$0.6 - 0.1X_0 - 0.2X_1 + 0.1X_2 - 0.5X_3 - 0.0Y_0 + 0.0Y_1$ $+0.1Y_2 + 1.7Y_3 - 0.0Y_4 - 0.0Y_5 - 0.0Y_6 - 0.0Y_7 + 0.1Y_8 + 0.0Y_9$ $-0.0Y_{10} - 0.0Y_{11} - 0.1Y_{12} + 0.0Y_{13} - 0.1Y_{14} - 0.0Y_{15}$	0.0	1.0	108.0

Table 1: Classifiers from a population evolved by XCSCA when the action function is implemented using a perceptron: c is the condition; xw is the argument of the perceptron, if xw > 0 then the action is 1, otherwise it is 0; ϵ is the error, F is the fitness, *num* is the numerosity; X_i and Y_i represent the multiplexer variables x_i and y_i after they have been mapped into -5/5 values.

classifiers are theoretically enough to represent the solution for the shift$_8$, XCS cannot reach optimality. However, it should be noted that while XCSCA is actually learning only the optimal actions, XCS evolves a complete representation of the problem solution. Thus, its learning burden is actually much higher than that of XCSCA.

7.2 Binary Sum

The binary shift involves many actions, however it is a rather easy problem in that each action bit can be independently determined. We now move to a more difficult function, the binary sum, in which the action bits are correlated. The binary sum of size m, shortly sum$_m$, takes as input a binary string of size $2m$, representing two binary numbers of size m, x and y, and returns a binary string s of size $m+1$ obtained by the sum $x+y$. For example, suppose that $k = 4$, $x =$1001, and $y =$1011, then sum$_4$(10011011) returns 10100. The circuit that computes sum$_4$ is depicted in Figure 3. As can be noted, while the least significant output bit s_0 is solely determined by the least significant input bits (x_0 and y_0), the subsequent action bits s_j depend both on the input bits x_j and y_j and the incoming carry bit (c_{j-1}) determined by the sum of the preceding bits. The sum$_4$ function is rather difficult. Accordingly, when we apply a feedforward neural network to learn sum$_4$ we need at least 20 hidden nodes to reach optimal performance (Figure 4).

Figure 5a reports the performance of XCSCA on the sum$_4$ function; XCS is not reported since it cannot learn the function even with a large population. The parameters are set as in the previous experiments with the shift$_8$. As expected, sum$_4$ is more difficult. In fact, XCSCA quickly reaches optimal performance when actions are computed using an array of perceptrons, an array of sigmoids, or a neural network with 10 hidden nodes. In contrast, the learning is extremely slow when a constant action function is involved. If we compare the performance of XCSCA (Figure 5a) with that of single neural networks (Figure 4), we note that, in terms of number of learning problems, XCSCA is slightly faster than neural networks (although the difference is not statistically significant). However, in terms of computation effort neural networks are obviously cheaper since XCSCA maintains a population of networks. When we consider the amount of generalization achieved (Figure 5b) we note that XCSCA with neural networks require less classifiers, while the array of perceptrons and the array of sigmoids provide similar generalizations. Finally, it is interesting to analyze the generalizations that XCSCA evolves in this problem. Table 2 shows

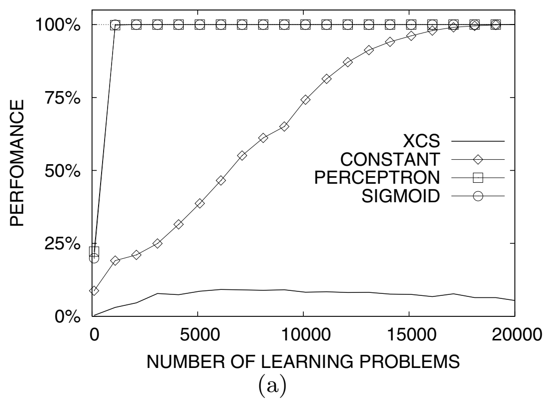
c	ϵ	F	num
###0####	0.0	1.0	971.0
###1####	0.0	1.0	967.0
#######1	0.0	0.1	56.0
##0###0#	0.0	0.0	4.0
###1##0#	0.0	0.0	1.0
#011##1#	0.0	0.0	1.0

Table 2: Population evolved by XCSCA for the sum$_4$ when the action function is computed using a neural network.

a typical population evolved by XCSCA for sum$_4$ when the action function is computed using a neural network with 10 hidden nodes (which in principle cannot solve the problem); action functions are not reported since they would require too much space. As it can be noted, XCSCA tends to evolve few very general and overlapping classifiers which partition the problem space based on the value of the least significant input bits. For instance, the first classifier in Table 2 is applied only when the first number (specified by the initial four bits) is even, i.e., when the sum won't result in a carry from the first bit. The next two classifiers are activated when a carry is probable, i.e., when one of the least significant bit is set to 1.

7.3 Anticipatory Function

In the last set of experiments, we applied XCSCA to a problem that is well-known to the learning classifier systems community, i.e., the learning of anticipatory behavior [7]. Anticipatory classifier systems [7] extend the typical classifier structure by adding the prediction of the next state that the system will encounter after the classifier action is performed. In our case, we did not extend XCSCA with anticipations but simply we applied XCSCA to learn a model of the environment which may be used to implement anticipations. For "*real*" anticipatory classifier systems we refer the reader to [7] and to [18] for a neural based implementation of anticipatory classifier systems. We focused on typical multistep environments, namely the *woods* environments, that have been already used both with XCS [21] and ACS [7]. We applied XCS to Woods1 [21] and Maze5 [13] and we traced the state transitions that XCS performed during the learning steps. Each transition records the current state s_t, the performed action a_t, and the next state s_{t+1} encountered af-



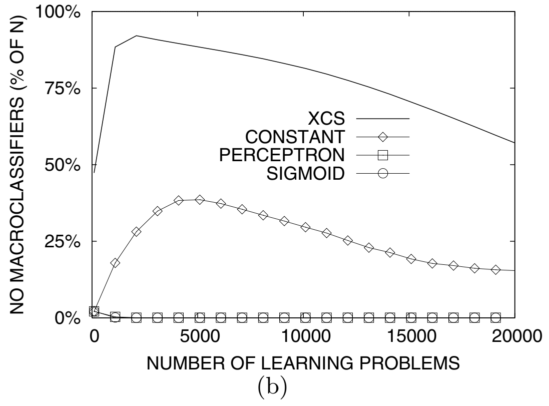
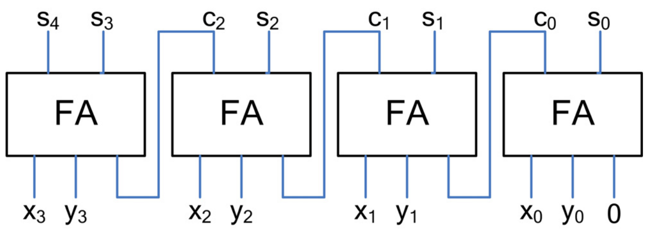


Figure 2: XCSCA applied to the shift$_8$: (a) performance (b) number of macroclassifiers.



s_4 s_3 c_2 s_2 c_1 s_1 c_0 s_0
FA
FA
FA
FA
x_3 y_3 x_2 y_2 x_1 y_1 x_0 y_0 0

Figure 3: The circuit that implements the function sum$_4$. The blocks "FA" are *full adder* that implement the sum of two bits including an incoming carry.

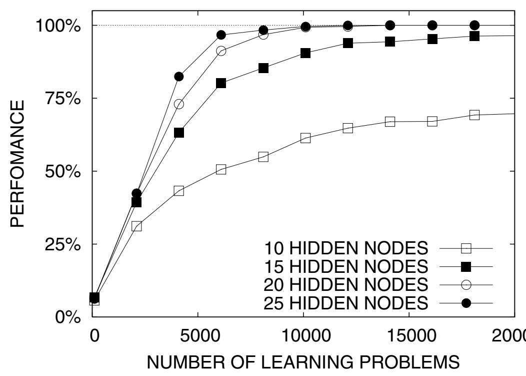


Figure 4: The performance of neural networks on sum$_4$ using different numbers of hidden nodes.

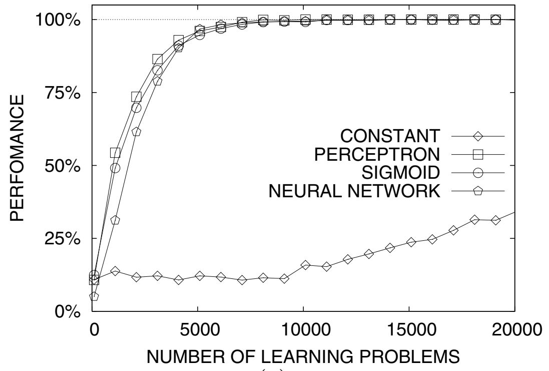
ter action a_t was performed in s_{t+1}. For each environment, we generated ten data sets containing 100000 transitions. The sequence of transitions contained in each data set was fed to XCSCA which had to learn a mapping from state-action pairs (represented as strings of 19 bits) into the next states (represented as strings of 16 bits). Since the transitions were obtained during learning, each data set contain many copies of the same transitions and the sequences are highly correlated.

We applied different versions of XCSCA to learn the model of `Woods1` (Figure 6) and `Maze5` (Figure 7); all the parameters were set as in the previous experiments except for the population size N that was set to 5000 classifiers. Figure 6a reports the performance and the population size for XCSCA in `Woods1` using an array of perceptrons, an array of sigmoids, and neural networks with ten hidden nodes. The problem is simple and all the three versions of XCSCA perform almost the same and they also reach the same level of generalization. Figure 6b compares the performance of XCSCA with neural networks with the performance of feedforward neural networks with different numbers of hidden nodes. As can be noted, a neural network with 40 hidden nodes quickly reaches a near optimal performance, though it never goes stably to 100%. To learn a model of `Woods1` with a 100% accuracy requires more than 40 hidden nodes. In contrast, XCSCA using neural networks with 10 hidden nodes can learn as fast as neural networks with 30 hidden nodes (which on the other hand does not reach fully optimality). Figure 7 provides the same comparison for `Maze5`. In this case, XCSCA with neural networks with 20 hidden nodes is slightly faster than XCSCA using arrays of perceptrons and sigmoids (Figure 7a). However, the difference is not statistically significant. Feedforward neural networks with 40 hidden nodes cannot learn a model of `Maze5` whereas XCSCA using neural networks with 20 hidden nodes easily converges to optimal performance (Figure 7b).

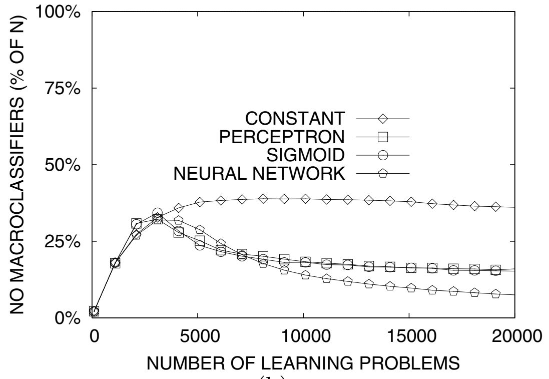
Table 3 shows examples of classifiers from populations evolved by XCSCA using neural networks for (a) `Woods1` and (b) `Maze5`. Action functions are not reported because too complex. The `Woods1` environment is simple to model, in fact most of the evolved classifiers are fully general except for the bits corresponding to the last three action bits. I.e., the problem of predicting the next state can be partitioned based on the performed action and using the state bits for computation. `Maze5` is more difficult (as the performance of single feedforward neural networks suggest). In fact, the evolved classifiers require more specific bits than those evolved for `Woods1`. This suggest that to predict the next state in `Maze5`, XCSCA also needs to partition the problem based on the current state, not only on the performed action.

8. CONCLUSIONS

Problems involving many actions can be challenging for a niche based learning classifier system like XCS. In this paper, we introduced a version of XCS in which computed actions are exploited to learn functions involving a large number of discrete actions. Previous theoretical results suggest that XCS has an inductive bias which is not well suited for dealing

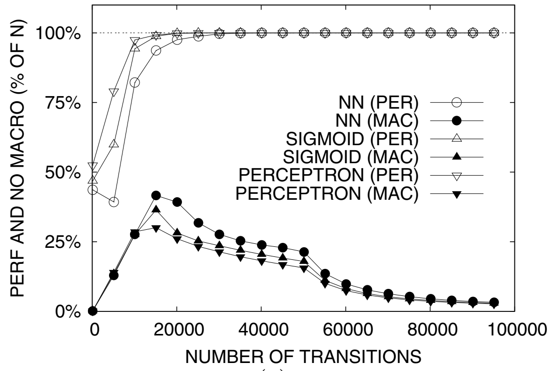


(a)

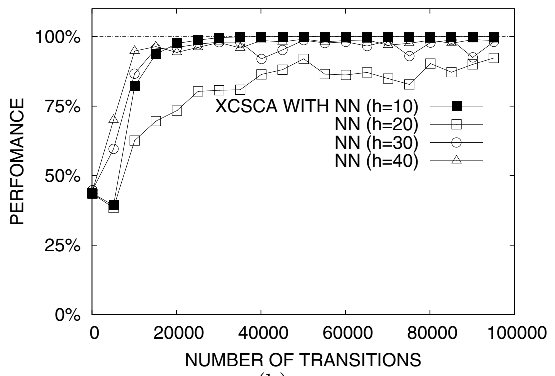


(b)

Figure 5: XCSCA applied to sum$_4$: (a) performance and (b) number of macroclassifiers.



(a)



(b)

Figure 6: (a) XCSCA applied to the learning of the model for Woods1: performance (empty dots) and population size (solid dots); (b) Comparison with neural networks.

with many actions [6, 5]. However, the XCSF idea [25], i.e., partitioning plus the use of mapping functions, seems to have a much less restricted inductive bias (with some functions better than others, as illustrated). In fact, by applying the mapping functions to compute actions, XCSCA can evolve maximally accurate and compact representations of binary functions that cannot be solved with XCS [21].

In XCSCA, each classifier implements a very simple supervised learning model that is trained from online experience; the population is an ensemble of models that are initially filtered through the matching process and later aggregated through an accuracy weighted average so as to provide a prediction of the target output. In this perspective, XCSCA (more than other classifier system models) resembles many similarities to aggregated predictors such as bagging [3], random forests [4], and other approaches developed in the area of Machine Learning [16]. Accordingly, future research directions include the application of XCSCA to supervised classification problems to compare the performance of XCSCA with that of aggregated approaches [3, 4].

9. REFERENCES

[1] Manu Ahluwalia and Larry Bull. A genetic programming-based classifier system. In Wolfgang Banzhaf et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, pages 11–18. Morgan Kaufmann, 1999.

[2] Ester Bernadoó-Mansilla and J.M. Garrell. Accuracy-based learning classifier systems: Models, analysis and applications to classification tasks. *Evolutionary Computation*, 11:209–238, 2003.

[3] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

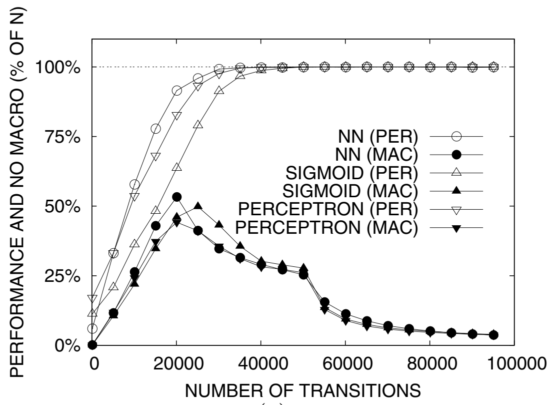
[4] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] Martin Butz, David G. Goldberg, and Pier Luca Lanzi. Bounding learning time in XCS. In *Genetic and Evolutionary Computation – GECCO-2004*, LNCS, 2004. Springer-Verlag.

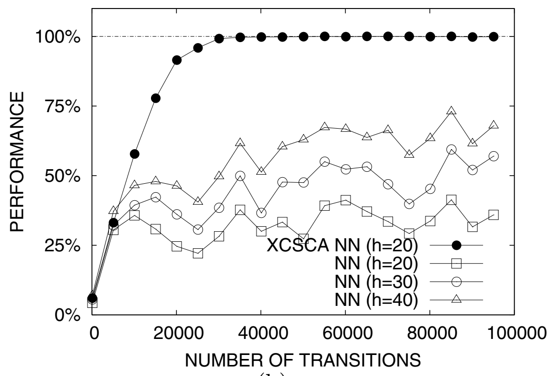
[6] Martin Butz, David G. Goldberg, Pier Luca Lanzi, and Kumara Sastry. Bounding the population size to ensure niche support in XCS. Technical Report 2004033, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, IL 61801, February 2004.

[7] Martin V. Butz. *Anticipatory Learning Classifier Systems*, volume 4 of *Genetic Algorithms and Evolutionary Computation*. Springer-Verlag, 2000.

[8] Martin V. Butz. Kernel-based, ellipsoidal conditions in the real-valued XCS classifier system. In Beyer H.G. et al., editor, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1835–1842, 2005. ACM Press.



(a)



(b)

Figure 7: (a) XCSCA applied to the learning of the model for `Maze5`: performance (empty dots) and population size (solid dots); (b) Comparison with neural networks.

c	ϵ	F	num
`#################011`	0.0	1.0	557.0
`#################000`	0.0	0.9	496.0
`#################111`	0.0	0.8	451.0
`#################101`	0.0	0.8	436.0
...	...	...	

(a)

c	ϵ	F	num
`#####0#0#####0#0#111`	0.0	0.9	126.0
`1####0#########1#00#`	0.0	0.9	118.0
`####0##########0#010`	0.0	0.8	113.0
`####0#######01###100`	0.0	0.8	112.0
...	...	...	

(b)

Table 3: Classifiers from two populations evolved by XCSCA for (a) `Woods1` and (b) `Maze5`.

[9] Martin V. Butz, Kumara Sastry, and David E. Goldberg. Strong, stable, and reliable fitness pressure in XCS due to tournament selection. *Genetic Programming and Evolvable Machines*, 6:53–77, 2005.

[10] Martin V. Butz and Stewart W. Wilson. An algorithmic description of xcs. *Journal of Soft Computing*, 6(3–4):144–153, 2002.

[11] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1998.

[12] J.H. Holland. Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems. In R.S. Michalski et al., editors, *Machine Learning, An Artificial Intelligence Approach*, volume 2, chapter 20, pages 593–623. Morgan Kaufmann, Los Altos, CA, 1986.

[13] Pier Luca Lanzi. An Analysis of Generalization in the XCS Classifier System. *Evolutionary Computation Journal*, 7(2):125–149, 1999.

[14] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. XCS with computed prediction for the learning of boolean functions. In *Proceedings of the IEEE Congress on Evolutionary Computation – CEC-2005*, pages 588–595, 2005. IEEE.

[15] Drew Mellor. A first order logic classifier system. In Hans-Georg Beyer and Una-May O'Reilly, editors, *GECCO*, pages 1819–1826. ACM, 2005.

[16] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[17] Toby O'Hara and Larry Bull. Accuracy-based neuro and neuro-fuzzy classifier systems. Technical Report UWELCSG02-001, University of West England., 2002.

[18] Toby O'Hara and Larry Bull. Building anticipations in an accuracy-based learning classifier system by use of an artificial neural network. In IEEE Press, editor, *IEEE Congress on Evolutionary Computation*, pages 2046–2052, 2005.

[19] F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, New York, 1962.

[20] Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994. http://prediction-dynamics.com/.

[21] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995. http://prediction-dynamics.com/.

[22] Stewart W. Wilson. Generalization in the XCS classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[23] Stewart W. Wilson. Function approximation with a classifier system. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 974–981, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[24] Stewart W. Wilson. Mining Oblique Data with XCS. volume 1996 of *Lecture notes in Computer Science*, pages 158–174. Springer-Verlag, April 2001.

[25] Stewart W. Wilson. Classifiers that approximate functions. *Journal of Natural Computating*, 1(2-3):211–234, 2002.

[26] Stewart W. Wilson. Three architectures for continuous action. Technical Report 2006019, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2006.