XCS with Computed Prediction for the Learning of Boolean Functions

Pier Luca Lanzi[†*], Daniele Loiacono[†], Stewart W. Wilson[‡*], David E. Goldberg[*]
[†]Artificial Intelligence Laboratory, Dip. di Elettronica e Informazione
Politecnico di Milano – Milano 20133, Italy
[*]Illinois Genetic Algorithms Laboratory, Dept. of General Engineering
University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
[‡]Prediction Dynamics, Concord, MA 01742, USA
lanzi@elet.polimi.it, loiacono@elet.polimi.it, wilson@prediction-dynamics.com, deg@illigal.ge.uiuc.edu

Abstract- Computed prediction represents a major shift in learning classifier system research. XCS with computed prediction, based on linear approximators, has been applied so far to function approximation, to single step problems involving continuous payoff functions, and to multi step problems. In this paper we take this new approach in a different direction and apply it to the learning of Boolean functions – a domain characterized by highly discontinuous 0/1000 payoff functions. We also extend it to the case of computed prediction based on functions, borrowed from neural networks, that may be more suitable for 0/1000 payoff problems: the perceptron and the sigmoid. The results we present show that XCSF with linear prediction performs optimally in typical Boolean domains and it allows more compact solutions evolving classifiers that are more general compared with XCS. In addition, perceptron based and sigmoid based prediction can converge slightly faster than linear prediction while producing slightly more compact solutions.

1 Introduction

With the introduction of XCSF [18] Wilson has recently extended the traditional idea of learning classifier systems through the concept of a computed classifier prediction. In typical learning classifier systems the prediction (or the strength [8]) associated to each classifier is memorized as a parameter. In XCSF [18], classifier prediction is computed as a linear combination of the current input and a weight vector associated to each classifier. Originally, XCSF was conceived as a pure function approximator [18]: classifiers did not have an action and computed classifier prediction was used to produce piecewise linear approximations of target functions. The initial results [18] shows that XCSF can evolve populations of classifiers that represent accurate piecewise linear approximations of sections of the target function. XCSF achieves such results through the balance of two main forces: (i) an evolutionary pressure, the same as XCS [1], pushing toward accurate maximally general classifiers so to break down the problem space into subspaces were accurate generalizations are possible; (ii) a learning mechanism based on typical linear approximators (Widrow-Hoff in [18], but others can be used) to adapt classifier weights with respect to the problem subspace. Recently, XCSF has been successfully applied to problems involving discrete actions [19]) and delayed rewards [10].

In this paper, we take XCS with computed prediction in a different direction. We start from XCSF with actions and linear prediction (XCS-LP in [19]) which, from now on we will simply call XCSF to abstract the concept of computed prediction (first introduced with XCSF [18]) from the more specific implementation with action and linear prediction (XCS-LP in [19]). We apply XCSF with Boolean actions and linear prediction to well-known Boolean problems which involve discontinuous 0/1000 payoff functions. The results we report show that XCS with linear computed prediction applied to Boolean functions can perform optimally and it converges slightly faster than XCS. But, most important, XCSF can evolve solutions that are on the average more compact than those evolved by XCS since the use of a computed prediction allows more general solutions. Then, we extend XCSF with alternative ways to compute classifier prediction that may be more suitable in 0/1000 payoff landscapes, borrowed from neural networks, namely the perceptron [13] and the sigmoid [7]. We apply these new versions of XCSF to the same Boolean problems and show that XCSF with perceptron based and sigmoid based prediction can converge slightly better than the version with linear prediction, while producing solutions that are also slightly more compact.

2 The XCSF Classifier System

XCSF differs from XCS in three respects: (i) classifiers conditions are extended for numerical inputs, as done in XCSI [17]; (ii) classifiers are extended with a vector of weights w, that are used to compute classifier's prediction; finally, (iii) the original update of classifier prediction must be modified so that the weights are updated instead of the classifier prediction. These three modifications result in a version of XCS, XCSF [18, 19], that maps numerical inputs into actions with an associated calculated prediction. In the original paper [18] classifiers have no action and assumes that XCSF outputs the estimated prediction, instead of the action itself. In this paper, we consider the version of XCSF with actions and linear prediction (named XCS-LP [19]) in which more than one action is available. As said before, throughout the paper we do not keep the (rather historical) distinction between XCSF and XCS-LP since the two systems are basically identical except for the use of actions in the latter case.

In XCSF, classifiers consist of a condition, an action, and four main parameters. The condition specifies which input states the classifier matches; as in XCSI [17], it is rep-

resented by a concatenation of interval predicates, $int_i = (l_i, u_i)$, where l_i ("lower") and u_i ("upper") are integers, though they might be also real. The action specifies the action for which the payoff is predicted. The four parameters are: the weight vector w, used to compute the classifier prediction as a function of the current input; the prediction error ε, that estimates the error affecting classifier prediction; the fitness F that estimates the accuracy of the classifier prediction; the numerosity *num*, a counter used to represent different copies of the same classifier. Note that the size of the weight vector w depends on the type of approximation. In the case of piecewise-linear approximation, considered in this paper, the weight vector w has one weight w_i for each possible input, and an additional weight w_0 corresponding to a constant input x_0, that is set as a parameter of XCSF.

Performance Component. XCSF works as XCS. At each time step t, XCSF builds a *match set* [M] containing the classifiers in the population [P] whose condition matches the current sensory input s_t; if [M] contains less than θ_{mna} actions, *covering* takes place and creates a new classifier that matches the current inputs and has a random action. Each interval predicate $int_i = (l_i, u_i)$ in the condition of a covering classifier is generated as $l_i = s_t(i) - \text{rand}(r_0)$, and $u_i = s_t(i) + \text{rand}(r_0)$, where $s_t(i)$ is the input value of state s_t matched by the interval predicated int_i, and the function rand(r_0) generates a random integer in the interval $[0, r_0]$ with r_0 fixed integer. The weight vector w of covering classifiers is randomly initialized with values from [-1,1]; all the other parameters are initialized as in XCS (see [4]).

For each action a_i in [M], XCSF computes the *system prediction* which estimates the payoff that XCSF expects when action a_i is performed. As in XCS, in XCSF the *system prediction* of action a is computed by the fitness-weighted average of all matching classifiers that specify action a. However, in contrast with XCS, in XCSF classifier prediction is computed as a function of the current state s_t and the classifier vector weight w. Accordingly, in XCSF system prediction is a function of both the current state s and the action a. Following a notation similar to [2], the system prediction for action a in state s_t, $P(s_t, a)$, is defined as:

$$P(s_t, a) = \frac{\sum_{cl \in [M]|_a} cl.p(s_t) \times cl.F}{\sum_{cl \in [M]|_a} cl.F} \qquad (1)$$

where *cl* is a classifier, $[M]|_a$ represents the subset of classifiers in [M] with action a, *cl.F* is the fitness of *cl*; $cl.p(s_t)$ is the prediction of *cl* computed in the state s_t. In particular, when piecewise-linear approximation is considered, $cl.p(s_t)$ is computed as:

$$cl.p(s_t) = cl.w_0 \times x_0 + \sum_{i>0} cl.w_i \times s_t(i)$$

where $cl.w_i$ is the weight w_i of *cl* and x_0 is a constant input. The values of $P(s_t, a)$ form the *prediction array*. Next, XCSF selects an action to perform. The classifiers in [M] that advocate the selected action are put in the current *action set* [A]; the selected action is sent to the environment and a reward P is returned to the system.

Algorithm 1 XCSF: Weights update with the modified delta rule.

```
1:  procedure UPDATE_PREDICTION(cl, s, P)
2:      error ← P − cl.p(s)
3:      norm ← x_0^2;                          ▷ Compute |x|^2
4:      for i ∈ {1, ..., |s|} do
5:          norm ← norm + s_i^2
6:      end for
7:      correction ← (η / norm) × error   ▷ Compute the overall
            correction
8:      cl.w_0 ← x_0 × correction          ▷ Update the weights
            according to the correction
9:      for i ∈ {1, ..., |s|} do
10:         cl.w_i ← cl.w_i + s_i × correction
11:     end for
12: end procedure
```

Reinforcement Component. XCSF uses the incoming reward P to update the parameters of classifiers in action set [A]. The weight vector w of the classifiers in [A] is updated using a *modified delta rule* [15]. For each classifier $cl \in [A]$, each weight $cl.w_i$ is adjusted by a quantity Δw_i computed as:

$$\Delta w_i = \frac{\eta}{|s_t(i)|^2}(P - cl.p(s_t))s_t(i) \qquad (2)$$

where η is the correction rate and $|s_t|^2$ is the norm the input vector s_t, (see [18] for details). Equation 2 is usually referred to as the "*normalized*" Widrow-Hoff update or "*modified* delta rule", because of the presence of the term $|\vec{s}_t(i)|^2$ [6]. The values Δw_i are used to update the weights of classifier *cl* as:

$$cl.w_i \leftarrow cl.w_i + \Delta w_i \qquad (3)$$

Then the prediction error ε is updated as:

$$cl.\varepsilon \leftarrow cl.\varepsilon + \beta(|P - cl.p(s_t)| - cl.\varepsilon)$$

Finally, classifier fitness is updated as in XCS.

Discovery Component. The genetic algorithm and subsumption deletion in XCSF work as in XCSI [17].

3 XCSF for Boolean Functions

XCSF can be applied to the learning of Boolean functions. For this purpose, we consider a version of XCSF in which the integer-based conditions originally used in [18] are replaced by the ternary representation [16]; there are two actions, 0 and 1, which represent the output of the Boolean function; while matching, covering, crossover, and mutation work *exactly* as in the original XCS [16]. To update the classifier weights, during the update process Boolean inputs are mapped into integer values by replacing zeros with -5 and ones with +5. We need to do this since with linear approximators zero values for inputs must be generally avoided [7]. The procedure to update the weights of one classifier is reported, following the notation of [4], as Algorithm 2.

Algorithm 2 XCSF: Weights update for Boolean inputs.

procedure UPDATE_PREDICTION(*cl*, s, P)
  for $i \in \{1, \ldots, |s|\}$ **do**
    if $s_i = 0$ **then** ▷ Map Boolean inputs to integers
      $y_i \leftarrow -5$; ▷ Input 0 is changed to -5
    else
      $y_i \leftarrow +5$; ▷ Input 1 is changed to +5
    end if
  end for
  error $\leftarrow P - cl.p(y)$
  norm $\leftarrow x_0^2$; ▷ Compute $|x|^2$
  for $i \in \{1, \ldots, |s|\}$ **do**
    norm $\leftarrow$ norm $+ y_i^2$
  end for
  correction $\leftarrow \frac{\eta}{norm} \times$ error
  $cl.w_0 \leftarrow x_0 \times$ correction
  for $i \in \{1, \ldots, |s|\}$ **do**
    $cl.w_i \leftarrow cl.w_i + y_i \times$ correction
  end for
end procedure

4 Design of Experiments

To apply XCSF to the learning of Boolean functions, we follow the standard settings used in the literature [16]. Each experiment consists of a number of problems that the system must solve. Each problem is either a *learning* problem or a *test* problem. In *learning* problems, the system selects actions randomly from those represented in the match set. In *test* problems, the system always selects the action with highest prediction. 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 reward policy we use is the usual one for Boolean functions [16]: when XCSF solves the problem correctly, it receives a constant reward of 1000; otherwise it receives a zero reward. The performance is computed as the percentage of correct answers during the last 100 test problems. All the reported statistics are averages over 50 experiments. In this paper, we consider two types of functions: Boolean multiplexer and hidden parity.

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 $mp_6(000111) = 0$.

Hidden Parity. This class of Boolean functions has been first used with XCS in [9] to relate the problem difficulty to the number of accurate maximally general classifiers needed by XCS to solve the problem. They are defined over binary strings of length n in which only k bits are relevant; the hidden parity function ($\mathsf{HP}_{n,k}$) returns the value of the parity function applied to the k relevant bits, that are *hidden* among the n inputs. For instance, given the hidden parity function $\mathsf{HP}_{6,4}$ defined over inputs of six bits ($n = 6$), in which only the first four bits are relevant ($k = 4$), then we have that $\mathsf{HP}_{6,3}(110111) = 1$ while $\mathsf{HP}_{6,3}(000111) = 1$.

5 Linear Prediction

We apply the version of XCSF for Boolean problems to the 11-multiplexer, with the following parameters setting: $N = 1000$, $\beta = 0.2$; $\alpha = 0.1$; $\epsilon_0 = 10$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $\theta_{del} = 20$; $\theta_{GA} = 25$; $\delta = 0.1$; GA-subsumption is on with $\theta_{GAsub} = 20$; while action-set subsumption is on with $\theta_{ASsub} = 400$; we use tournament selection [3] with size 0.4; the don't care probability for ternary conditions is $P_{\#} = 0.3$. Figure 1a compares the performance and the population size of the original XCS [16] with those of XCSF; curves are averages over 50 runs. As can be noted, XCSF reaches optimal performance slightly faster and also evolves solutions that on the average are slightly more compact. We apply the same version of XCSF to the 20-multiplexer with $N = 2000$, $P_{\#} = 0.5$, and $\theta_{ASsub} = 800$; all the other parameters are set as in the previous experiment. Figure 1b compares the performance and the population size of the original XCS [16] with those of XCSF; curves are averages over 50 runs. The results confirm what found in the 11-multiplexer, XCSF converges to optimal performance slightly faster, evolving solutions that are also slightly more compact. In fact, XCS evolves solutions that on the average contain 250 classifiers (the 12.5% of N), while XCSF evolves solutions that on the average contain 172 classifiers (the 8.6% of N). Finally, we apply XCSF to the $\mathsf{HP}_{20,5}$ problem with the parameter settings taken from [11]: $N = 2000$, $\beta = 0.2$; $\alpha = 0.1$; $\epsilon_0 = 1$; $\nu = 5$; $\chi = 1.0$, $\mu = 0.04$, $\theta_{del} = 20$; $\theta_{GA} = 25$; $\delta = 0.1$; GA-subsumption is on with $\theta_{GAsub} = 20$; while action-set subsumption is off; the don't care probability for ternary conditions is $P_{\#} = 1.0$; the other XCSF parameters are set as usual. Figure 1c compares the performance and the population size of the original XCS [16] with those of XCSF; curves are averages over 50 runs. Also these results confirm what priorly found, though in this problem the difference in convergence between XCSF and XCS is more evident.

To understand why XCSF can evolve solutions that are generally more compact, we need to analyze the evolved solutions; for this purpose we consider the first experiment on the 11-multiplexer. Table 1 reports the most numerous classifiers in one of the populations evolved by XCSF; column `id` reports a unique classifier identifier, column `Condition:Action` reports the classifier condition and action; column $cl.p(\vec{x})$ reports the equation used to compute classifier prediction, for the sake of brevity zero and near zero weights which do not contribute to the overall prediction are not reported; columns $cl.\varepsilon$, *cl.F*, *cl.exp*, *cl.num* indicate respectively classifier error, fitness, experience, and numerosity. As can be noted, the population contains classifiers whose conditions have only three specific bits (e.g., classifier 0,1,2,4, and 5) instead of the usual four specific bits that XCS would require to solve the same problem. Such classifiers are characterized by (i) small weights corresponding to the address bits of the multiplexer, (ii) a large weight for the the bit that should be specified in the solu-

tion evolved by XCS but it is actually generalized by XCSF, (iii) all the weights associated to other inputs have zero or near zero values. For instance, in classifier number 2 in Table 1, the specific bits and the constant term (corresponding to $w_0 x_0$) provide a constant start value (equal to 500.1) from which the classifier prediction is computed based on the value of the target input bit x_8: if the Boolean value of input 8 is 1 (and thus the actual input x_8 is 5), then the classifier prediction is $500.1 + 100.0 \times 5$, that is 1000.1; if the Boolean value of input 8 is 0 (and thus the actual input x_8 is -5), then the classifier prediction is $500.1 + 100.0 \times (-5)$, that is 0.1. Basically, XCSF can allow more general conditions, with more "#", by exploiting the numerical value of the inputs matched by a "#", to produce an accurate value of the classifier prediction. Thus on the average XCSF can evolve more compact solutions since it allows more general conditions. This may also simplify the search space and improve the convergence speed in problems, such as the hidden parity, in which the number of specific bits highly influences the learning process.

6 The Perceptron

In the previous section, we showed that XCSF can exploit linear prediction to solve Boolean multiplexer and hidden parity apparently faster than XCS. Another approach, related to neural networks, to tackle Boolean problems involves the perceptron model [12]. The perceptron takes as input a vector of real values $\vec{x} = \langle x_0, x_1 \ldots x_n \rangle$ (where x_0 is the usual constant input) and outputs either -1 either 1 through a two stage process: first the linear combination $\vec{w}\vec{x}$ of the real inputs $\vec{x}$ and of the weight vector $\vec{w}$ is calculated, then the perceptron outputs 1 if $\vec{w}\vec{x} > 0$, -1 otherwise. The perceptron is trained from a set of examples $\{\langle \vec{x}_k, y_k \rangle\}$, where y_k is the desired output for input $\vec{x}_k$, by adjusting the weights. Given the input vector $\vec{x}$ and the desired output y, the weight w_i associated to input x_i is updated according to the perceptron training rule as follows [13]:

$$w_i \leftarrow w_i + \eta(y - o)x_i \tag{4}$$

where o is the perceptron output for input $\vec{x}$, and η is the usual learning rate.

Note that the perceptron training rule in Equation 4 is very similar to the weight update used in XCSF for linear prediction (Equation 2). There are only two differences. The former one is in the way the current output o is computed: in XCSF, $o = \vec{w}\vec{x}$, while in the perceptron $o = sgn(\vec{w}\vec{x})$, with sgn the sign function. The latter is in the normalization factor $|\vec{x}|^2$ used in the delta rule update (Equation 2) which is missing from Equation 4. It is thus straightforward to modify XCSF by replacing the usual linear approximation with a perceptron like approximator to compute classifier prediction. While XCSF exploits linear approximation to learn a 0/1000 payoff function, with a perceptron based approximator XCSF directly learns a two value function to approximate the same payoff. In the version of XCSF modified with the perceptron approximation classifier prediction

Algorithm 3 XCSF: Weights update with perceptron.

```
1: procedure UPDATE_PREDICTION(cl, s, P)
2:     error ← P − cl.p(s)
3:     correction ← η × error
4:     cl.w_0 ← x_0 × correction
5:     for i ∈ {1, ..., |s|} do
6:         cl.w_i ← cl.w_i + s_i × correction
7:     end for
8: end procedure
```

is computed as,

$$cl.p(\vec{x}) = \begin{cases} 1000 & \text{if } \vec{x} \times cl.w > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

while the update of classifier weights is performed according to Algorithm 3.

We apply the version of XCSF with perceptron based classifier prediction to all the problems previously considered, with the same parameter settings. Figure 2 compares the performance and population size of XCSFwith linear prediction and XCSF with perceptron based prediction in (a) the 11-multiplexer, (b) the 20-multiplexer, and (c) $\mathsf{HP}_{20,5}$. As the plots show, XCSF modified with the perceptron based prediction converges slightly faster than XCSF with linear approximation; however, while in the Boolean multiplexer there is almost no difference between the size of the solutions evolved by linear and perceptron prediction, perceptron based prediction evolves slightly larger solutions in the hidden parity problem.

When analyzing the solutions evolved by this version of XCSF we discover that the generalization produced are similar since, also with perceptron, XCSF allows classifiers with fewer specific bits. However, with perceptron based prediction the weights are used in a completely different way. In the case of linear prediction, XCSF needs to evolve weight values that allow an accurate estimate of the classifier prediction. In contrast, in the case of the perceptron, XCSF must guarantee that the weight values are such that the argument of the sign function ($cl.\vec{w} \times \vec{x}$) produces the correct 0/1000 payoff value. This implies that with the perceptron there are less constraints on the values that the classifier weights can take. Generally, it is convenient for XCSF with perceptron to evolve classifiers for which the argument of the sign function ($cl.\vec{w} \times \vec{x}$) has an high probability of being far from zero. In fact, a near zero value of $cl.\vec{w} \times \vec{x}$ might be source of sudden changes in the classifier prediction which would make the classifier inaccurate. As an example, let us consider the classifier 2 in Table 1), for the same classifier XCSF with perceptron has evolved a weight vector corresponding to the following value of $cl.\vec{w} \times \vec{x}$:

$$\begin{aligned} cl.\vec{w} \times \vec{x} \;=\; & 0.0 - 234.1x_1 + 234.1x_5 + 234.1x_6 + \\ & +6764.1x_8 - 2530.1x_{10} + 2000.0x_{11} \end{aligned}$$

which has zero prediction error and a classifier fitness of 0.956. Note that only x_1 among the address bits has a non zero weight, and, apart from x_8 which determines the correct prediction, there are also other inputs with a non zero

id	Condition:Action	*cl.p*($\vec{x}$)	*cl.*ε	*cl.F*	*cl.exp*	*cl.num*
0	101######## : 0	$420.5 + 9.7x_1 + 6.2x_2 + 12.4x_3 - 100.0x_9$	0.000	1.000	5904	45
1	010######## : 0	$486.2 + 3.4x_1 + 7.5x_2 + 1.3x_3 - 100.0x_6$	0.000	1.000	6050	43
2	100######## : 1	$337.6 + 16.9x_1 - 16.9x_2 + 1.3x_3 + 100.0x_8$	0.000	0.994	5969	42
3	111#######1 : 0	$58.2 - 0.2x_1 - 4.8x_2 - 4.1x_3 - 2.5x_{11}$	0.000	0.999	1394	42
4	011######## : 0	$438.6 + 5.8x_1 + 4.9x_2 + 13.1x_3 - 100.0x_7$	0.000	1.000	6061	42
5	110######## : 1	$312.5 + 12.1x_1 + 12.3x_2 - 13.2x_3 + 100.0x_{10}$	0.000	0.999	5655	40
6	010##1##### : 1	$984.6 - 21.5x_1 + 32.4x_2 - 23.6x_3 - 74.4x_6$	0.000	1.000	4199	39
7	111#######1 : 1	$501.9 + 24.5x_1 + 18.3x_2 + 31.5x_3 + 25.3x_{11}$	0.000	0.927	4242	36
8	110######0# : 0	$530.7 + 5.9x_1 + 35.7x_2 - 10.6x_3 - 41.7x_{10}$	0.000	0.933	4425	36
9	100######## : 0	$566.4 + 3.6x_1 + 3.3x_2 + 13.6x_3 - 100.0x_8$	0.000	0.919	6133	35
10	101#####1## : 1	$510.6 + 25.5x_1 - 20.4x_2 + 25.5x_3 + 26.4x_9$	0.000	0.988	4473	35
11	111#######0 : 1	$137.3 + 6.9x_1 - 48.1x_2 + 6.9x_3 - 6.9x_{11}$	0.000	0.828	1522	33
12	0001####### : 0	$99.3 + 17.5x_1 - 5.0x_2 + 2.1x_3 - 5.2x_4$	0.000	0.912	1273	33
13	001######## : 0	$286.9 - 13.9x_1 - 14.3x_2 + 14.3x_3 - 100.0x_5$	0.000	0.676	6150	32
14	011###1#### : 1	$573.8 - 23.4x_1 + 27.7x_2 + 9.7x_3 + 24.4x_7$	0.000	0.903	4530	31
15	110######1# : 0	$104.5 - 1.4x_1 - 22.2x_2 - 4.0x_3 - 1.4x_{10}$	0.000	0.920	1282	31
16	101#####0## : 1	$244.6 + 12.2x_1 - 12.2x_2 - 3.2x_3 + 70.2x_9$	0.000	0.869	1400	30
17	111#######0 : 0	$637.3 + 31.9x_1 - 23.1x_2 + 31.9x_3 - 31.9x_{11}$	0.000	0.949	3865	30
18	011###0#### : 1	0.0	0.000	0.936	1464	29
...	...	...	...	...	...	...

Table 1: Example of classifiers evolved by XCSF for the 11-multiplexer.

contribution to the value of $cl.\vec{w} \times \vec{x}$. However, the contribution to $cl.\vec{w} \times \vec{x}$ of the relevant inputs (x_1 and x_8) alone is so high (-32650 when $x_8 = -5$, +34991 when $x_8 = 5$) to make the other contributions irrelevant with respect to the perceptron output.

7 The Sigmoid

The sigmoid is the obvious extension of the perceptron and it is one of the most typical activation functions used in neural networks [7]. It is defined as:

$$f(z) = \frac{k}{1 + e^{-z}} \tag{6}$$

where z is $\vec{w}\vec{x}$ and k is a scale factor, specified by the user, which defines an upper limit to function $f(z)$. We now extend XCSF with sigmoid based computation of classifier prediction; since in this paper we focus on 0/1000 payoffs, we set $k = 1000$ so that the prediction of a classifier *cl* is now defined as:

$$cl.p(\vec{x}) = \frac{1000}{1 + e^{-cl.\vec{w}\vec{x}}},$$

which maps the input vector into a prediction value between 0 and 1000. The update of classifier weights is done according to a simple gradient descent [7] on the prediction error as follows:

$$w_i \leftarrow w_i + \eta(P - cl.p(z))\frac{\partial f(z)}{\partial z}x_i \tag{7}$$

where $z = cl.\vec{w} \times \vec{x}$. Unfortunately this update suffers from the well-known problem of "flat spots" [7]: when $|z| > 3$ weights are adjusted very slowly and the derivative contribution in Equation 7 tends to be zero very easily. To overcome this problem, we adopt the most elementary technique used in the literature [5] to limit flat spots: a constant ϵ (empirically determined based on the problem domain) is added to the derivative contribution. The weight update is now,

$$w_i \leftarrow w_i + \eta(P - cl.p(z))[\frac{\partial f(z)}{\partial z} + \epsilon]x_i$$

In all the experiments discussed here, we set the contribution ϵ to 25 which we determined from some preliminary experiments as the 10% of the average contribution of the central slope in our problem domain. The update of classifier weights for sigmoid based prediction is reported as Algorithm 4.

Algorithm 4 XCSF: Weights update with sigmoid.

```
1: procedure UPDATE_PREDICTION(cl, s, P)
2:     error ← P − cl.p(s);                    ▷ Compute the error
3:     z = cl.w_0 × x_0;                       ▷ Compute z
4:     for i = 1 to |s| do
5:         z ← z + cl.w_i × s(i);
6:     end for
7:     gr ← 1000/(1 + e^{−z}) × [1 − 1/(1 + e^{−z})) + ε];
8:     correction ← η× error × gr;
9:     cl.w_0 ← x_0 × correction;
10:    for i = 1 to |s| do
11:        cl.w_i ← cl.w_i + s_i × correction;
12:    end for
13: end procedure
```

We apply XCSF with sigmoid prediction to the same problems considered for the previous versions with the same experimental settings. Figure 3 compares the performance and population size of XCSF with linear and sigmoid based prediction in (a) the 11-multiplexer, (b) the 20-multiplexer, and (c) $\mathsf{HP}_{20,5}$ respectively. As can be noted from the plots, in Boolean multiplexer XCSF with sigmoid converges slightly slower than linear prediction showing a performance curve very similar to that of the perceptron (Figure 2a and Figure 2b); in the hidden parity problem, XCSF with sigmoid prediction converges slightly faster than linear prediction. With respect to the size of the solutions evolved, XCSF with sigmoid prediction behaves basically as XCSF with perceptron prediction: it performs basically the same as linear prediction in Boolean multiplexer, but produces bigger solutions in the hidden parity problem.

8 Conclusions

We applied XCS with computed prediction, XCSF [18, 19] to the learning of Boolean functions. In particular, we considered three versions of XCSF: one involving classifier prediction based on linear approximation [18, 19]; one involving classifier prediction based on the perceptron [13]; and finally, one involving classifier prediction based on the sigmoid [7]. We applied these three versions of XCSF and XCS [16] to the 11-multiplexer, the 20-multiplexer, and the hidden parity problem ($\mathsf{HP}_{20,5}$).

Our results show that the use of computed prediction can result in solutions that are more compact compared with those evolved by XCS. In fact, XCSF can allow more general conditions by exploiting the input values corresponding to don't care positions to produce accurate estimates of the target payoff values.

As a general result, we note that all the three versions of XCSF converge (slightly) faster than XCS; such difference becomes more evident in problems (such as $\mathsf{HP}_{20,5}$) where the number of specific bits needed to reach a solution has more influence on the number of problems required to converge. The approach involving the sigmoid has a small drawback: to avoid *flat spots* [7] it might require adhoc parameter settings for the weight update.

We also note that all the three versions of XCSF evolve solutions that are on the average at least as compact as those evolved by XCS. Nevertheless, from our preliminary results, it appears that XCSF with linear prediction evolves populations that are on the average more compact that those evolved by XCSF with perceptron and sigmoid based predictions.

Finally, we wish to note that the work presented here is also related to the work of Smith and Cribbs [14] where a simple analogy between learning classifier systems and neural networks was discussed. Note however that in this work, classifiers are enriched with the simplest form of neural computation (a perceptron) that is used to compute solely classifier prediction, while in [14] classifiers represent hidden nodes of one competitive neural network. Nevertheless, given the many similarities of this work to [14], a deeper comparison between XCSF, the analogy proposed in [14], and typical neural approaches is currently planned.

We believe that XCS with computed prediction represents a major shift in learning classifier system research and a very promising research direction for the future development in this area. In [18], Wilson has shown that XCSF can solve problems that cannot be tackled by traditional models. The results presented in this paper suggest that XCSF can also perform better than traditional models (better than XCS) in problems were traditional models are usually applied. As a matter of fact, XCSF is a proper generalization of XCS and we can actually implement XCS with XCSF by considering a very simple function to compute classifier prediction, i.e., $cl.p(\vec{x}) = w_0$, and by using the basic Widrow-Hoff update to adjust w_0. On the other hand we can move the idea of computed classifier prediction beyond the perceptron and the sigmoid considered here so as to allow any type of approximation, depending on the problem considered. We might as well allow XCSF to evolve the most adequate prediction function for providing an accurate approximation of payoff values in each problem subspace.

Acknowledgments

This work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF (F49620-03-1-0129), and by the Technology Research, Education, and Commercialization Center (TRECC), at University of Illinois at Urbana-Champaign, administered by the National Center for SupercomputingApplications (NCSA) and funded by the Office of Naval Research (N00014-01-1-0175). The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

Bibliography

[1] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. Toward a theory of generalization and learning in xcs. *IEEE Transaction on Evolutionary Computation*, 8(1):28–46, Feb. 2004.

[2] M. V. Butz and M. Pelikan. Analyzing the evolutionary pressures in xcs. In L. S. et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 935–942, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[3] M. V. Butz, K. Sastry, and D. E. Goldberg. Strong, stable, and reliable fitness pressure in xcs due to tournament selection. *Genetic Programming and Evolvable Machines*, 2005. in press.

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

[5] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, 1995. available online at http://neuron.eng.wayne.edu/tarek/MITbook/.

[6] S. Haykin. *Adaptive Filter Theory*. Prentice-Hall, 1986.

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

[8] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975. Republished by the MIT press, 1992.

[9] T. Kovacs and M. Kerber. What makes a problem hard for XCS? In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*, pages 80–99. Springer–Verlag, Berlin, 2001.

[10] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. Xcs with computable prediction in multistep environments. Technical Report 2005008, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2005. accepted at GECCO2005.

[11] Martin V. Butz and David E. Goldberg, K. Tharakunnel. Analysis and Improvement of Fitness Exploitation in XCS: Bounding Models, Tournament Selection, and Bilateral Accuracy. *Evolutionary Computation*, 11(4):239–277, 2003.

[12] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1959.

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

[14] R. E. Smith and H. B. Cribbs. Is a Learning Classifier System a Type of Neural Network? *Evolutionary Computation*, 2(1):19–36, 1994.

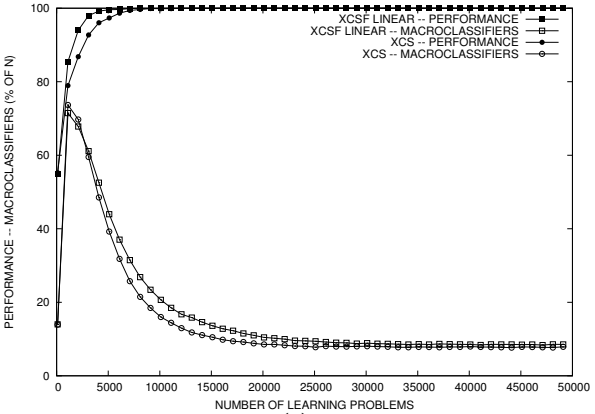
[15] B. Widrow and M. E. Hoff. *Adaptive Switching Circuits*, chapter Neurocomputing: Foundation of Research, pages 126–134. The MIT Press, Cambridge, 1988.

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

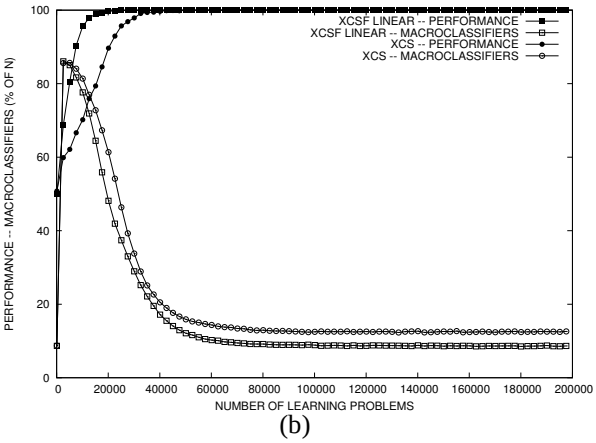
[17] S. W. Wilson. Mining Oblique Data with XCS. volume 1996 of *Lecture notes in Computer Science*, pages 158–174. Springer-Verlag, Apr. 2001.

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

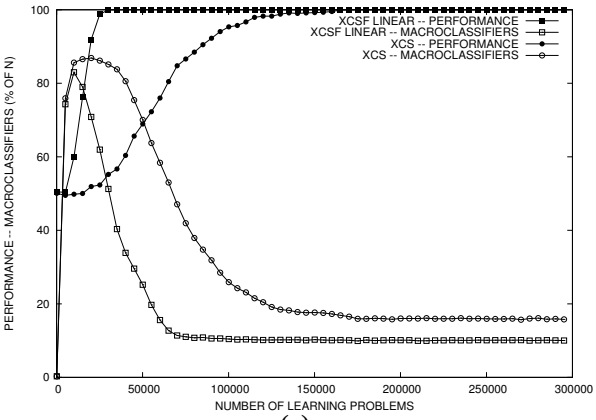
[19] S. W. Wilson. Classifier systems for continuous payoff environments. In K. D. et al., editor, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 824–835, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.



(a)

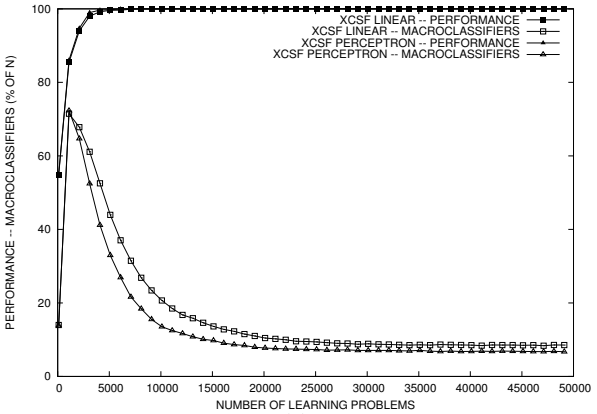


(b)

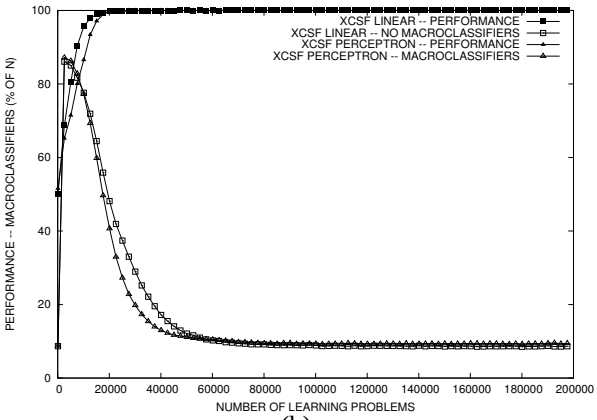


(c)

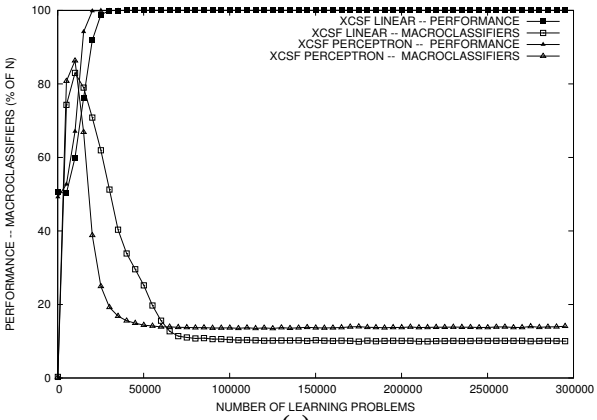
Figure 1: XCS and XCSF in (a) the 11-multiplexer, (b) the 20-multiplexer, and (c) $\mathsf{HP}_{20,5}$: performance (solid dots) and number of macroclassifiers (empty dots).



(a)

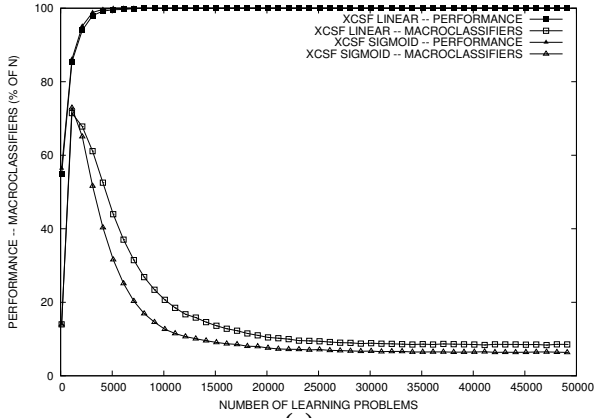


(b)

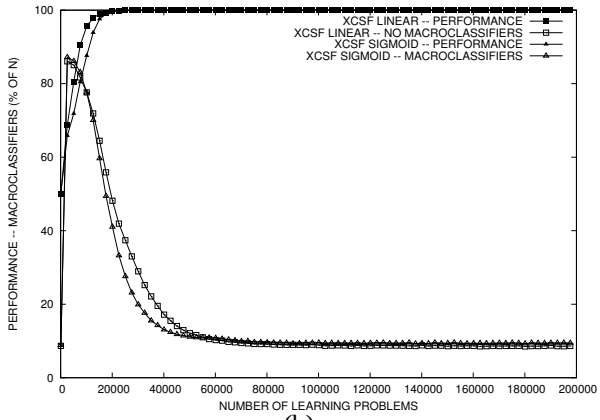


(c)

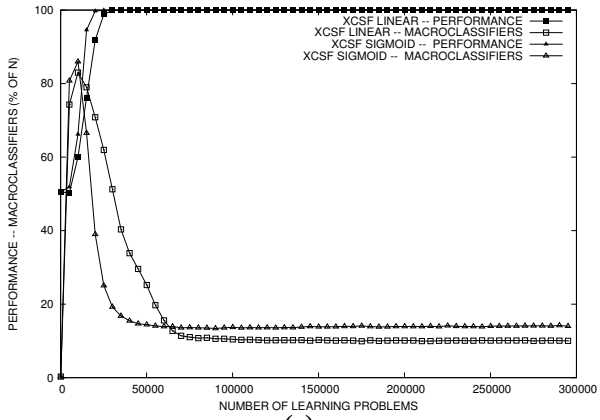
Figure 2: Linear and perceptron prediction in the (a) 11-multiplexer, (b) 20-multiplexer, and (c) $\mathsf{HP}_{20,5}$: performance (solid dots) and number of macroclassifiers (empty dots).



(a)



(b)



(c)

Figure 3: Linear and sigmoid prediction in (a) the 11-multiplexer, (b) the 20-multiplexer, and (c) $\mathsf{HP}_{20,5}$: performance (solid dots) and number of macroclassifiers (empty dots).