A Spiking Neural Repesentation for XCSF

David Howard and Larry Bull

Technical report UWELCSG09-003

Learning Classifier Systems Group

UWE, Bristol, U.K.

gerard2.howard@uwe.ac.uk

17 July 2009

` **ABSTRACT**

This paper presents a Learning Classifier System (LCS) where each classifier condition is represented by a spiking neural network. Adaptive behavior is realized through the use of self-adaptive parameters and neural constructivism, providing the system with a flexible knowledge representation. The approach allows for the evolution of networks of appropriate complexity to solve a continuous maze environment, here using discrete-valued actions. It is shown that the neural LCS is capable of developing optimal solutions to the reinforcement learning task presented in this paper.

1. INTRODUCTION

Agent navigation tasks serve as a well-established test bed for learning systems. Typical tasks involve an agent, which is initially situated randomly within a maze environment, using sensory readings to navigate to a goal state; arrival at the goal state triggers a reward. These kinds of navigation tasks can be broadly defined as either discrete (e.g., [26]) or continuous (e.g., [4]). In this paper we describe a learning system that operates in continuous environments.

Our chosen learning system is the Learning Classifier System (LCS) [17] – an online evolutionary reinforcement-based machine learning system that evolves a population of rules (classifiers) using a Genetic Algorithm (GA) [16]. XCS [36] is a type of LCS where the fitness of a classifier is related to the accuracy of its prediction of rewards – this leads to a system where all areas (not just high reward areas) of the problem landscape are covered by classifiers that predict expected rewards with a high degree of accuracy. XCS has been extended to include function approximation, creating XCSF [37]. XCSF computes predicted rewards – classifier prediction is not a constant value, but rather is calculated as the product of the sensory input and a weight vector, which allows the same classifier to generalize by predicting different reward values in different areas of the environment. In this paper, following [5], we replace each classifier condition with a neural network – in this case a spiking neural network (see [14] for an overview). We base our LCS implementation on the XCSF classifier system.

We employ self-adaptive mechanisms to allow the rate of evolution (self-adaptive mutation) of the agent and both the minor (connection selection – to alter inter-neural connectivity patterns) and major (neural constructivism – to add or remove entire hidden layer nodes) topological complexity of the neural network rules to be altered automatically by the agent in response to repeated interactions with its environment.

2. REPRESENTATIONS IN LEARNING CLASSIFIER SYSTEMS

Due to their ability to easily integrate many biologically-inspired and machine-learning algorithms within their learning architectures, a number of hybrid LCS have been developed which share similar motifs to our system. A recent survey into the current state of the field, including areas of successful LCS application is given by [32].

Because of the modular nature of LCS, a variety of classifier representations have been implemented. For example, [33] introduce a relational representation scheme for use with XCSF. Traditionally, classifier membership is inclusive – that is, the classifier can only express if it matches the input state or not. Four operators are introduced; IN, CLOSE, OUT and FAR – which give a more succinct covering of problem space, increasing generalization. This more powerful representation shows various relationships between classifier and state, meaning less classifiers are needed to cover the entire problem space.

Related work in the area of representations in LCS includes that of integer conditions for function approximation, the task to which XCSF was originally applied [37] and the initial version of neural XCS [5]. Fuzzy logic was first used in traditional LCS in [34], later in [3] and more recently in XCS [8]. Ahluwalia and Bull [1] used LISP S-expressions for designing LCS as a feature pre-processor. More recently [7] demonstrates an XCSF-derivative to control a robot arm using a real interval representation. Current arm posture (represented as three joint angles) is used as state input, and prediction input encodes the change in these angles, and predicts the change in hand location. Bonarini et al. [3] introduce fuzzy interval coding for real-valued problems. They consider the effect of fuzzy granularity on performance, testing their system on an autonomous robotic agent.

Also of interest is the work of Dam and Abbas [9], who detail the implementation of a neural network ensemble into a supervised, accuracy-based classifier system known as UCS. Further neural

representations include the work on which this paper is based [19,20], where a self-adaptive, constructive neural XCS and XCSF are used to solve both discrete and continuous-valued maze environments.

Dynamical GP conditions are implemented in [28], whereby each input state is run a number of times through a GP graph which may be updated either synchronously or asynchronously. The power of the representation is the ability to implicitly model temporal dynamics, although the authors only implement a discrete input state, rather than the continuous one demonstrated here.

3. SPIKING MODELS

Spiking neural models are several degrees more complex than MLP models, and can be generally defined in terms of being either time-based or rate-based. They are covered in detail in [14]. Two well-known formal implementations are the Integrate and Fire (IAF) model and the Spike Response Model (SRM). Both allow for interesting behavioural dynamics, which potentially allows for more degrees of freedom with regards to control options. Motivation for the inclusion of spiking networks as opposed to simpler network forms (e.g. MLP networks [31]) is based on the assumption that simpler network forms can constrain the types of solutions evolved by the system, possibly in a manner that is detrimental to performance.

The IAF model allows neurons to be stimulated either by an external current or by connections from presynaptic neurons. A spike is emitted from the neuron when its membrane potential reaches a threshold, whereupon the potential is reset.

The SRM is a generalization of the IAF model, in which the state of a given neuron i is captured in a single variable, u_i. This variable is initially said to be 0, or "at rest". Any incoming spikes to the neuron i increase the value of u_i, which then slowly returns to 0. In this way, it can be seen that several spikes within a given temporal windows will be required to trigger a postsynaptic action potential. A

spike again occurs when the membrane potential surpasses some threshold, which is then set to be extremely positive to prevent further immediate firing, before slowly decreasing back to some more attainable dynamic value. Korkin et al. [25] apply an "integrate-and-fire" spiking neuron model to evolve networks that produce time-dependant patterns such as sinusoids. Federici [10] uses a similar spiking model and applies it to a simple robotic navigation task, concluding that the inherent dynamics of a spiking network may provide further degrees of freedom to a given problem solution. An alternative model, presented for large-scale (e.g. 1000-neuron networks) is presented in [23]. The author proves his network model has more degrees of freedom than traditional IAF approaches, and thus is capable of modeling several neural activation patterns found in the cortex, including tonic bursting and chaotic behavior.

Spiking networks have previously been applied to robot control; Indiveri et al [22] apply spiking circuits to model abstractions of biological retina to extract motion information, and apply their system to a robotic platform where the output of the networks is used to control the robots motion in response to visual stimuli. Floreano, Durr and Mattussi [12] survey various methods for evolving both weights and architectures in neural networks, highlighting the complementary nature of coupling the computing power of neural networks with the adaptability of evolutionary approaches.

Applications of spiking networks utilizing the neuro-evolutionary paradigm have been prevalent in recent years. Particularly relevant is the work of Quinn (e.g. [29]), who uses abstracted spiking models to control a swarm of robots. Further neuro-evolutionary spiking implementations include [13], who argue that temporal patterns of sensory-motor events could potentially be exploited more effectively by a network type that includes temporal functionality, when compared to, for example, MLP networks. The authors also highlight the benefits of network evolution, pointing out that hand-tuned networks may be particularly difficult to design given the non-linear dynamics of spiking networks.

Floreano and Mattiusi [11] evolve SRM spiking networks for vision-based robot navigation, applied to a simple task where the fitness of an agent is proportional to the amount of forward motion in a given time frame. Floreano et al. [13] apply an IAF spiking model, again for the goal of evolving agent navigation.

4. NEURAL XCSF

In XCSF, a classifier's prediction is computed. Here, the prediction is the reward a classifier *expects* to gain from executing its action based on the current input state. This allows the same classifier to predict different payoff values in different regions of the environment, increasing the generalization capability of the system. For a complete algorithmic description we refer the reader to [37].

XCSF evolves a population of classifiers, [P], to cover a problem space. Each classifier consists of a condition and an action. In our case, a classifier is a spiking neural network which represents both condition and (calculated) action. At each time-step, XCSF builds a match set, [M], from [P] consisting of all classifiers whose conditions match the current input state *st*. In this case, *st* consists of the x and y components of the agent's current location within the environment. Both the x and y position of the agent are subject to noise; +/- [0%-5%] of the agents' true position.

Each classifier condition/action pair is represented by a spiking network object, itself containing a number of node and connection objects, which can be used to create an individual network. Each weight in this condition vector is uniformly initialized randomly in the range [0, 1]. For the agent navigation task considered here, each network comprises 2 input neurons (representing the x and y location of the agent), a number of hidden layer neurons under evolutionary control (see section 5.1 for more details), and 3 output neurons. The first two output neurons represent the strength of action passed to the left and right motors of the robot respectively (via the number of spikes received at the output neuron in a given window of time). The third output neuron is a "don't-match" neuron, that excludes the classifier from the

match set if it has spikes in more than half of its sampling window. This is necessary as the action of the classifier must be re-calculated for each state the classifier encounters, so each classifier is exposed to each state transition as it occurs.

The outputs at the other two neurons (spike trains) are mapped to a single discrete movement in one of four compass directions (North, East, South, West). This takes place in a way similar to [6], except here there are four possible directions and only two ranges of discrete output are possible: $0.0 \leq x < 0.5$ (low), and $0.5 \leq x \leq 1.0$ (high). A neuron is said to be highly activated if it has spikes in over half of the positions in the sampling window; otherwise it is said to have low activation – see section 6 for more precise implementation details. The combined actions of the two neurons translate to a discrete movement according to the two motor output strengths – (high, high) = North, (high, low) = East, (low, high) = South, and (low, low) = West. Covering is achieved by repeatedly generating random networks until the networks action matches the desired output for a given input state, i.e., until the "don't match" neuron has low activation and the appropriate output values are produced on the other two nodes for the given input. In this paper, in the discrete action case, each action must be present in each [M].

At the start of each experiment, each classifier is initialized with a weight vector, *w*, which is used to compute the classifiers prediction. This weight vector has one element for each input (2 in this case), plus an additional element w_0 which corresponds to x_0, a constant input that is set as a parameter of XCSF. Each weight vector element is initialized to 0. Once [M] is formed, a prediction array is created. In XCSF, each classifier prediction (*cl.p*) is calculated as a product of the environmental input (or state, *st*) and the weight vector (*w*) associated with each classifier, specifically:

$$cl.p(st) = cl.w_0 * x_0 + \sum_{i>0} cl.w_i * st(i) \qquad (1)$$

The prediction array is the fitness weighted-average of the calculated predictions for each possible action. These predictions are summed to form the prediction array as a fitness-weighted average of all classifiers in the match set that specify a given action. The prediction array is then used as in regular XCSF to decide on an action to take (deterministic during an exploit trial and random during an explore trial). The action is then performed and reward possibly returned from the environment. During reinforcement, the weight vector of each classifier in the action set is updated using a version of the delta rule, rather than updating the classifiers' prediction value (equation 2). Here, the vector x is the state st augmented by the parameter x_0.

$$\Delta w_i = \eta / |x_{t-1}|^2 (r - cl.p(s_{t-1})) x_{t-1}(i) \quad (2)$$

Each weight is then updated (equation 3) and prediction error is calculated (equation 4).

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

$$\varepsilon \leftarrow \varepsilon + \beta (|r - cl.p(s_{t-1})| - \varepsilon) \quad (4)$$

Once an action is selected, all classifiers that advocate the selected action form the action set [A]. The action is taken and, if the goal state is reached, a reward is returned from the environment that is used to update the parameters of the classifiers in [A]. A discounted reward is propagated to the previous action set $[A_{-1}]$ if it exists. Further details of the update procedure used in XCSF can be found in [37]. This cycle of $[P] \rightarrow ([M] \rightarrow [A]) \rightarrow reward$ is called a trial. That is to say, a trial begins when the agent is

randomly initialized in the environment, and ends when the agent reaches the goal state, with varying numbers of set formations (*[M]*→*[A]*steps) taking place during the trial. Each experiment consists of 20,000 trials. Each trial consists is either in exploration mode (roulette wheel action selection) or exploitation mode (deterministic action selection). See figure 1 for a graphical representation of the XCSF classifier system.

The GA may then fire based on the usual XCSF mechanism. Our GA cycle is modified to be a two-stage process. Stage 1 (section 4.1) controls the rates of mutation and constructivism/connection selection that occur within the system, with stage 2 (sections 5.1 and 5.2) controlling the evolution of neural architecture in terms of both neurons and connections.

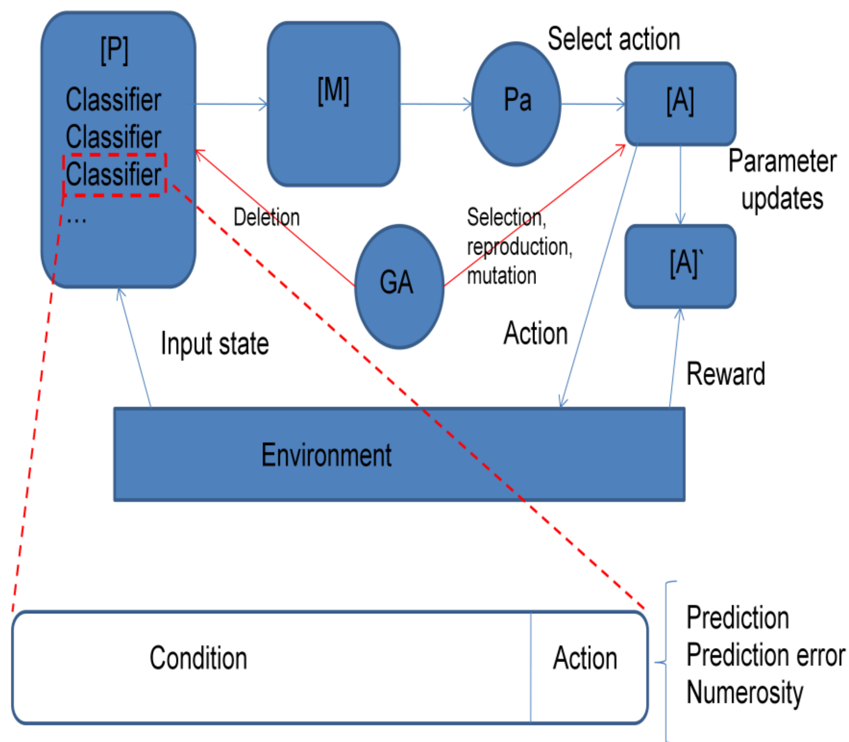


Figure 1. The XCSF classifier system

4.1 SELF-ADAPTATION

A GA is periodically triggered in [A] to evolve fitter classifiers in an environmental niche. We apply self-adaptation as in [6], to dynamically control the amount of genetic search (the frequency of mutation events) taking place within the niche. This potentially provides stability to parts of the problem space that are already "solved" as the mutation rate for a niche is typically directly proportional to its distance from the goal state during learning; generalization learning, along with the value function learning, occurs faster nearer the goal state [6]. Here, the μ value (rate of mutation per allele) of each classifier is initialized uniformly randomly in the range [0,1]. During a GA cycle, a parent's μ value is modified as in equation 5. The result is clipped within the range [0,1]. The offspring then adopts this new μ, and mutates its' condition by this value, before being inserted into the population.

$$\mu \leftarrow \mu * e^{N(0,1)} \qquad (5)$$

5. TOPOLOGY MECHANISMS

In addition to self-adaptive mutation, we apply two evolutionary topology morphology schemes to allow the modification of the spiking network conditions in two regards; by adding/removing nodes, and adding/removing inter-neural connections. The effect of this self-adaptive, constructivist framework is to tailor the evolution of the classifier to the complexity of the environment, allowing each classifier to control its own knowledge representation autonomously.

- Self-adaptive parameters alter the amount of mutation that takes place in a given niche at a given time [6].

- Neural constructivism adapts the hidden layer topology of the neural networks to reflect the predominant complexity of the problem space considered by the network [5].

- Connection selection automatically deactivates non-salient or disruptive network connections to potentially produce more parsimonious solutions containing only those features necessary to function optimally [18]. It can also be thought of the equivalent of the traditional ternary "don't care" # symbol, applied over the deactivated connection by essentially zeroing the contribution of that connection to the output of the network.

5.1 NEURAL CONSTRUCTIVISM

The scenario for constructivist learning (e.g., [29]) is that, rather than start with a large neural network, development begins with a small network. Learning then adds appropriate structure, particularly through growing/pruning dendritic connectivity, until some satisfactory level of utility is reached. Suitable specialized neural structures are not defined *a priori*; the representation of the problem space is flexible and tailored by the learner's interaction with it (see also [15]).

Implementation of constructivism in this system is based on the aforementioned work in neural LCS [5], plus Hurst and Bulls' implementation for use on a real robot [21]. Each rule has a varying number of hidden layer neurons (initially 1, and always > 0), with additional neurons being added or removed from the hidden layer depending on the constructivism element of the system. Constructivism takes place during a GA cycle, after mutation. Two new self-adaptive parameters, ψ and ω, are added. Here, ψ represents the probability of performing a constructivism event and ω is the probability of adding a neuron, with removal occurring with probability 1- ω. As with self-adaptive mutation, both are initially randomly generated uniformly in the range [0,1], and offspring classifiers have their parents' ψ and ω values modified during reproduction as with μ. Nodes created during constructivism are initially excitory with 50% probability, otherwise they are inhibitory.

5.2 CONNECTION SELECTION

Feature selection (e.g., [2]) is a method of streamlining the data input process, e.g., to remove noisy features. This can be done manually (by a human with relevant domain knowledge), although this process can be error-prone, costly and, of course, requires an expert. A popular alternative in machine learning is to automate feature selection, e.g., through a "wrapper" approach [24]. The search space of possible input combinations can be explored via a GA, which probabilistically flips the connections from enabled to disabled (and vice versa), or other heuristics, and fitness/utility of the chosen feature set is determined by running the given learner over the task using only those features. Especially pertinent to the current work is the implementation of feature selection within the NEAT framework (FS-NEAT) [35], who demonstrate the ability to solve a double pole balancing task with 256 inputs.

Connection selection is implemented in our system as follows: Each connection in a classifiers condition is stored in an unordered vector that details every connection in the network. During a GA cycle, and based on a new self-adaptive parameter τ (which is initialized and self-adapted in the same manner as the other parameters), a connection can be created (enabled) or removed (disabled). If a connection is disabled, the connection is removed from the networks connection vector. A newly enabled connection has its connection weight randomly initialised uniformly in the range [0,1]. All connections are initially set to "enabled" for newly initialised classifiers and classifiers created via cover. During a node addition event, all possible connections from the new node to all other nodes (except input nodes) are evaluated, and enabled with 50% probability.

As our spiking networks are implemented in an object-oriented manner, disabled connections are removed from the networks connection vector, giving a performance bonus to sparsely-connected networks. This is in comparison to our previous work (e.g. [18]), whereby a connection is kept in memory whether it is enabled or disabled.

6. SPIKING REPRESENTATION

In contrast to previous work, our spiking representation is implemented in an object-oriented manner. Each classifier condition is represented by a "network" object, which contains a vector of both "node" and "connection" objects. During runtime, the network generates a number of "spike" objects, which are stored in a separate vector. Each type of object details pertinent features (e.g. Nodes have a type – inhibitory or excitory, and keep track of their own membrane potentials and spike refactory values. Connections have a weight (adopting the type of their originating neuron) and input and output neuron indices. Spikes have a time delay and target neuron, and adopt the weight of the connection they travel down (and therefore implicitly also adopt the sign of the neuron that originally sent the spike).

Our representation is based on the IAF model for spiking neurons, which can be loosely conceptualised as a Continuous Time Recurrent Neural Network (CTRNN) [38] coupled to a pulse generator which emits a spike once the relevant node has reached a certain value of membrane potential.

Each neuron has an internal state (or membrane potential, *v*). As spikes are received by the neuron, *v* increases until it surpasses a threshold, *vthresh*. At this point, the neuron sends a spike to all connected neurons, whose strength is relative to the connection weight between those two neurons. At time t, the membrane potential of a neuron *n* is given as:

$$V(t+1) = v(t) + (I + a - bv(t)) \tag{6}$$

With the simple reset equation:

$$If(v > vthresh)\ v = c \tag{7}$$

Here, $v(t)$ is the membrane potential at time t, I is the input current to the neuron, a is a positive constant, b is degradation constant and c is the reset membrane potential of the neuron.

We make some changes to the standard IAF model to allow it to better fit our system. Firstly, we introduce a noisy reset by introducing uniform noise to the reset parameter c. Instead of resetting to a constant $c = 0$, we reset to c = [0.0,0.01]. This enhancement is designed simply to prevent a network from becoming stuck in a regular activation pattern loop.

Secondly, we introduce simple temporal delays to our model. Connections from input neurons to hidden layer neurons are given a delay of 1 (meaning they will be executed at the next cycle), and likewise for all connections from hidden layer neurons to output neurons. Within the hidden layer, delays are calculated by an artificial distance metric based on the difference between neuron indices. For example, a spike sent between neuron with index 1 and a neuron with index 2 has a delay of 1, whereas a spike sent between two neurons with indices 2 and 6 will have a delay of 4. These temporal delays should potentially allow the networks to evolve more complex behaviours.

Normally, spiking neurons have their initial membrane potentials v set to the membrane reset value c. We employ a boot-strapping mechanism to encourage quicker attainment of highly activated neurons (which are required to produce movements to the North (high, high), East (high, low) and South (low, high)). In our system, the initial membrane potential of every network is set to c_ini; in the following experiments $c_ini = 0.5$. This allows the networks to spike quicker, giving less "setup" cycles as the networks begin to respond more strongly to the inputs and thus allowing us to use smaller window sizes at the output nodes (a performance-enhancing measure; see later), as well as a faster general emergence of temporally-sensitive activation patterns within the networks. Note that after an initial spike, neurons are reset to c and not c_ini, boot-strapping only affects the time to first spike of each neuron in the network.

On classifier creation, initial construction of the networks proceeds as follows:

- Create 2 input and 3 output nodes (always excitory)
- Create 3 hidden layer nodes (50% chance of excitory, else inhibitory)
- Fully connect the input to hidden, hidden to hidden (recurrent connections are allowed, no duplicate connections), and hidden to output. Input nodes must connect to hidden nodes, no inter-input or inter-output connections are permitted. Connections cannot have their outputs as an input node, nor their input as an output node. Randomly weight connections in the range [0-1].
- Set all initial *v* to *c_ini*.

Two main changes were made to the execution cycle to accommodate spiking networks, in action calculation and GA activity.

Action calculation

The following cycle is repeated five times, each time feeding the environmental input into the input neurons.

- Use the environmental x and y values as input to the two input neurons. If spikes are generated at the input neurons, add them to the spike vector with "time to activation" = 1.
- Decrease the "time to activation" of all spikes in the spike vector by 1.
- If any spikes have time=0, remove them from the spike vector and update the target neuron of the spike. Find the contribution of the spike to the target neurons membrane potential by multiplying the weight of the connection the spike travels along by the sign of the originating neuron (+1 for excitory neurons, -1 for inhibitory neurons).
- Then update each hidden layer neuron in turn, generating spikes and altering membrane potentials as necessary. If an update causes a spike at an arbitrary neuron, find all nodes that are connected

to that neuron, where the spike-generating neuron is the input and a given arbitrary node (or nodes) are the output and add those spikes (with their calculated delays) to the spike vector.

- Record whether or not each output neuron generated a spike, save the resulting behaviour in a window.

Finally, calculate the strength of action at the output neurons by examining the spike outputs of the five cycles. If more than two of the five window elements contain spikes, activation = high. Otherwise, activation = low.

GA activity

As the sign of a connection (and hence effect of a spike) depends on the type of neuron generating the spike, we extend the regular GA to include neuron sign mutation. On satisfaction of μ, a neurons type can be changed from excitory to inhibitory (or vice versa). We also constrain connection weights to be >0, so that an excitory node always has excitory connections, and an inhibitory node always has inhibitory connections, regardless of weight mutation.

5. EXPERIMENTATION

5.1 ENVIRONMENT

The test environment for our system is the 2D continuous Grid world [4]. The environment extends from 0 to 1 in both *x* and *y* directions. The agent starts randomly within the maze, in any state except the goal state, and must navigate to the goal state (where the agents x and y position ≥ 1.00) in the fewest moves possible.

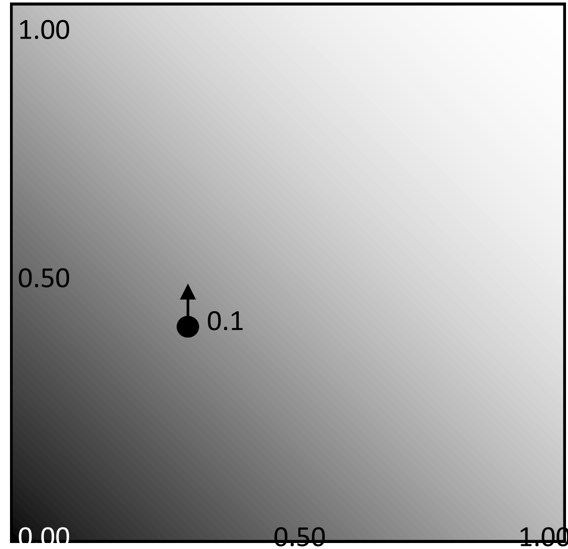
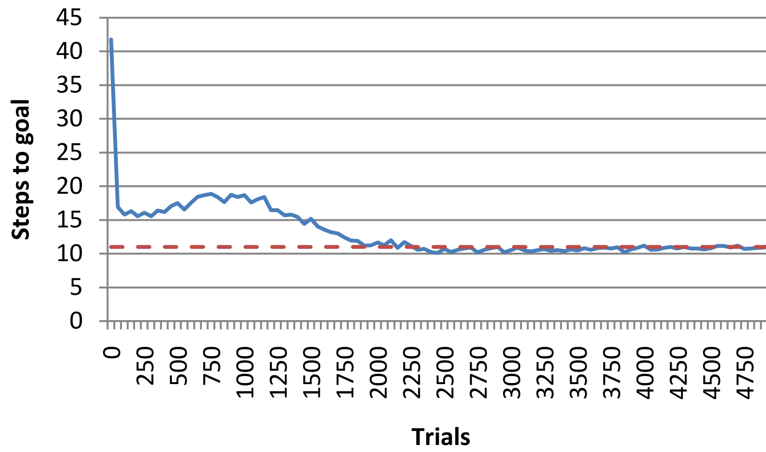


Figure 2. The continuous 2D Grid environment, and a proposed agent movement

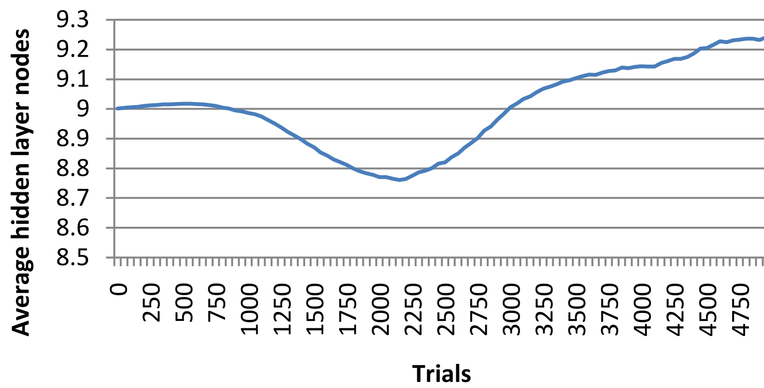
Upon reaching the goal state, the agent receives a reward of 1000. Using a step size of 0.1 (the size of a single agent movement), the average optimal number of steps to the goal state (the chief indicator of performance in such environments) is 11 [27]. The environmental discount rate γ=0.95. Related work in using XCSF-based systems to solve the Grid environment can be found in [20,27].

5.2 RESULTS

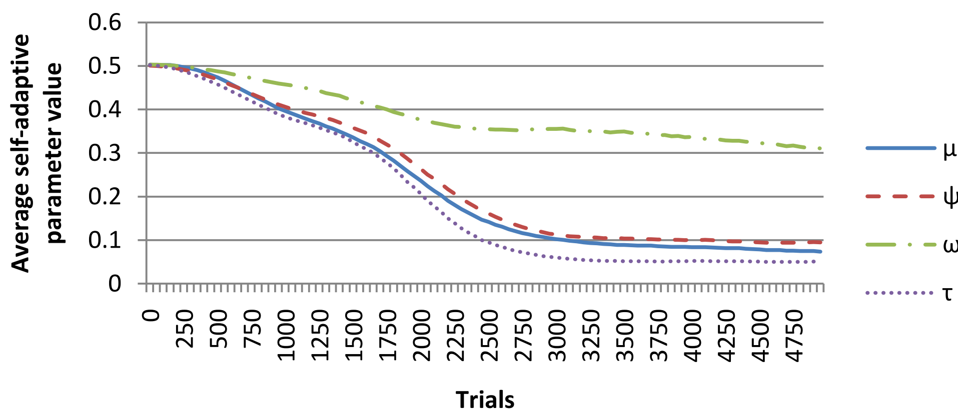
The following experiments are parameterized as follows, following [37]: N=20000, β=0.2, ε_0= 0.005, v=5, θ_{GA}=50, θ_{DEL}=50. Additionally the XCSF x_0 parameter is set to 1.0 and the correction rate η to 0.2. Spiking parameters include a=0.3,b=0.05, c=0.0 and $vthresh$ = 1.0. Each experiment lasts for 5000 trials. Every 50 trials, the system statistically analyses the current state of the system – this information is used to create the graphs shown below.



(a)



(b)



(c)

Figure 3 (a) Steps to goal (b) average hidden layer nodes (c) self-adaptive parameter values in Grid(0.1)

As the graphs show, our spiking classifier system is able to solve the grid (0.1) environment rapidly (after around 2000 trials on average). Figure 3(b) shows little increase in the initial number of hidden layer neurons, with a final figure of 9.23 connected hidden layer neurons.

6. DISCUSSION

In this paper we have introduced what is to our knowledge the first LCS to utilise a spiking neural condition/action representation; and therefore also the first spiking XCSF. The performance of the system has been shown to be high on the continuous maze environment presented. It is important to note that, although the move from a MLP representation to a spiking representation may allow the system to tackle a more difficult class of problems (or solve existing problems in a more elegant manner/ with more degrees of freedom), the run-time of the system is many times greater than many other classifier condition representations.

7. REFERENCES

1. Ahluwalia, M. & Bull, L. 1999. *A Genetic Programming Classifier System*. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela & R.E. Smith (Eds.) Proceedings of the Genetic and Evolutionary Computation Conference – GECCO-99. San Mateo, CA: Morgan Kaufmann, pp11-18.

2. Belue, L.M. & Bauer Jr., K.W. 1995. *Determining input features for multilayer perceptrons*. Neurocomputing 7:111-121.

3. Bonarini, A., Bonacina, C., & Matteucci, M. 2000. *Fuzzy and crisp representations of real-valued input for learning classifier systems*. In P.L. Lanzi, W. Stolzmann, & S. W. Wilson. (Eds.)

Learning Classifier Systems: From Foundations to Applications, volume 1813 of LNAI, Berlin, Springer-Verlag, pp107-124.

4. Boyan, J.A. & Moore, A.W. 1995. *Generalization in reinforcement learning: Safely approximating the value function*. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, Advances in Neural Information Processing Systems 7, the MIT Press, Cambridge, MA , pp369-376.

5. Bull, L. 2002. *On Using Constructivism in Neural Classifier Systems*. In Merelo, J, Adamidis, P., Beyer, H-G., Fernandez-Villacanas, J-L., & Schwefel, H-P. (Eds.) Parallel Problem Solving from Nature – PPSN VII. Springer Verlag, pp558-567.

6. Bull, L., Hurst, J., & Tomlinson, A. 2000. *Self-Adaptive Mutation in Classifier System Controllers*. In J-A. Meyer, A. Berthoz, D. Floreano, H. Roitblatt & S.W. Wilson (Eds.) From Animals to Animats 6 – The Sixth International Conference on the Simulation of Adaptive Behaviour, MIT Press.

7. Butz, M. V. & Herbort, O. 2008. *Context-dependent predictions and cognitive arm control with XCSF*. In Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (Atlanta, GA, USA, July 12 - 16, 2008). M. Keijzer, Ed. GECCO '08. ACM, New York, NY, 1365-1372.

8. Casillas, J. Carse, B. & Bull, L. 2007. *Fuzzy XCS: A Michigan Genetic Fuzzy System*. IEEE Transactions on Fuzzy Systems 15(4): 536-550.

9. Dam, H.H., Abbass, H.A., Lokan, C. & Yao, X. 2008. *Neural-Based Learning Classifier Systems*. IEEE Trans. on Knowl. and Data Eng. 20, 1 (Jan. 2008), 26-39.

10. Federici, D. 2005. *Evolving developing spiking neural networks*. In: Proceedings of CEC 2005 IEEE congress on evolutionary computation.

11. Floreano, D. & Mattiusi, C. 2001. *Evolution of Spiking Neural Controllers for Autonomous Vision-Based Robots*. In Gont, T., ed., Evolutionary Robotics, From Intelligent Robotics to Artificial Life. Tokyo, Springer Verlag.

12. Floreano, D., Dürr, P. & Mattiussi, C. 2008. *Neuroevolution: from architectures to learning*. Evolutionary Intelligence, 1(1) pp. 47-62.

13. Floreano, D., Schoeni, N., Caprari, G., and Blynel, J. 2003. *Evolutionary bits'n'spikes*. In Proceedings of the Eighth international Conference on Artificial Life (December 09 - 13, 2002). R. K. Standish, M. A. Bedau, and H. A. Abbass, Eds. MIT Press, Cambridge, MA, 335-344.

14. Gerstner , W. & Kistler, W.M. 2002. *Spiking Neuron Models - Single Neurons, Populations, Plasticity*, Cambridge Univ. Press.

15. Harvey, I., Husbands, P. & Cliff, D. 1994. *Seeing the Light: Artificial Evolution, Real Vision*. In D. Cliff, P. Husbands, J-A. Meyer & S.W. Wilson (eds) From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour, Cambridge, MA: MIT Press, pp392-401.

16. Holland, J.H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

17. Holland, J.H. 1976. *Adaptation.* In R. Rosen & F.M. Snell (Eds.) Progress in Theoretical Biology 4. New York: Academic Press, pp263-293.

18. Howard, D. & Bull, L. 2008. *On the Effects of Node Duplication and Connection-Orientated Constructivism in Neural XCSF*, In M. Keijzer et al. (eds) *GECCO-2008: Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, pp. 1977-1984

19. Howard, D., Bull, L. & Lanzi, P-L. 2008. *Self-Adaptive Constructivism in Neural XCS and XCSF*. In M. Keijzer et al. (eds) GECCO-2008: Proceedings of the Genetic and Evolutionary Computation Conference. ACM Press.

20. Howard, D., Bull, L. & Lanzi, P-L. 2009. *Continuous Actions in Continuous Space and Time using Self-Adaptive Constructivism in Neural XCSF*. In *GECCO-2009: Proceedings of the Genetic and Evolutionary Computation Conference.* ACM Press.

21. Hurst, J. & Bull, L. 2006. *A Neural Learning Classifier System with Self-Adaptive Constructivism for Mobile Robot Control*. Artificial Life 12(3): 353 – 380.

22. Indiveri, G. 2001. *A Neuromorphic VLSI device for implementing 2D selective attention systems*. IEEE Transactions on Neural Networks. In press.

23. Izhikevich E.M. 2003. *Simple Model of Spiking Neurons.* IEEE Transactions on Neural Networks, 14:1569- 1572.

24. Kohavi, R. & John, G. 1997. *Wrappers for Feature Subset Selection*. Artificial Intelligence 1: 273-324.

25. Korkin, M., Nawa, N.E., de Garis H. 1998 *A 'spike interval information coding' representation for ATR's CAM-brainmachine (CBM)* In: Proceedings of the 2nd international conference on evolvable systems: from biology to hardware(ICES'98). Springer, Heidelberg.

26. Lanzi, P.L. 1999. *An Analysis of Generalization in the XCS Classifier System.* Evolutionary Computation 7(2): 125-149.

27. Lanzi, P.L., Loiacono, D., Wilson, S.W. & Goldberg, D.E. 2005. *XCS with computed prediction in continuous multistep environments*. In Proceedings of the IEEE Congress on Evolutionary Computation CEC-2005, IEEE, Edinburgh, UK, pp2032-2039.

28. Preen, R. & Bull, L. 2009. *Discrete Dynamical Genetic Programming in XCS*. In GECCO-2009: Proceedings of the Genetic and Evolutionary Computation Conference. ACM Press.

29. Quartz, S.R. & Sejnowski, T.J. 1997. *The Neural Basis of Cognitive Development: A Constructionist Manifesto*. Behavioural and Brain Sciences 20(4): 537-596.

30. Quinn, M. 2001. A comparison of approaches to the evolution of homogeneous multi-robot teams. In Proc. Congr. Evolutionary Computation, 128-135. Seoul, South Korea: IEEE Press.

31. Rumelhart, D.E. & McClelland, J.L. 1986. Parallel Distributed Processing. Cambridge, MA: MIT Press.

32. Sigaud, O. & Wilson, S.W. 2007. *Learning classifier systems: a survey*. Soft Comput. 11(11): 1065-1078.

33. Tabarzad, M.A., Lucas C., & Hamzeh, A. 2006. *Relational Representation in XCSF*. 14th International Conference on Computational and Information Sciences, Prague, Czeck Republic, August 25- 27.

34. Valenzuela-Rendón, M. 1991. *The Fuzzy Classifier System: a Classifier System for Continuously Varying Variables*. In Proceedings of the 4th International Conference on Genetic Algorithms (ICGA91) pp346-353.

35. Whiteson, S., Stone, P., Stanley, K.O., Miikkulainen, R. & Kohl,N. 2005. *Automatic Feature Selection in Neuroevolution,* Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington DC, USA.

36. Wilson, S.W. 1995. *Classifier Fitness Based on Accuracy*. Evolutionary Computation, 3(2):149-175.

37. Wilson, S.W. 2001. *Function Approximation with a Classifier System*. In Spector, L., D., G. E., Wu, A., Langdon, W.B., Voight, H. M., and Gen, M., (Eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 01) Morgan Kaufmann. pp974-981.

38. Zipser. D. 1989. *A learning algorithm for continually running fully recurrent neural networks*. Neural computation, MIT Press.