

# Autoencoding with a Classifier System

Richard J. Preen<sup>id</sup>, Stewart W. Wilson<sup>id</sup>, and Larry Bull<sup>id</sup>

**Abstract**—Autoencoders are data-specific compression algorithms learned automatically from examples. The predominant approach has been to construct single large global models that cover the domain. However, training and evaluating models of increasing size comes at the price of additional time and computational cost. Conditional computation, sparsity, and model pruning techniques can reduce these costs while maintaining performance. Learning classifier systems (LCS) are a framework for adaptively subdividing input spaces into an ensemble of simpler local approximations that together cover the domain. LCS perform conditional computation through the use of a population of individual gating/guarding components, each associated with a local approximation. This article explores the use of an LCS to adaptively decompose the input domain into a collection of small autoencoders where local solutions of different complexity may emerge. In addition to benefits in convergence time and computational cost, it is shown possible to reduce code size as well as the resulting decoder computational cost when compared with the global model equivalent.

**Index Terms**—Autoencoder, evolutionary algorithm, learning classifier system, neural network, self-adaptation, stochastic gradient descent, XCSF.

## I. INTRODUCTION

EVOLUTIONARY algorithms (EAs) combined in some form with stochastic gradient descent have experienced a resurgence in their use for optimising large neural networks [1]. These techniques seek to construct a single large global network that covers the entire feature space. While very large networks have consistently attained state-of-the-art performance in a wide range of domains, training and evaluating networks of increasing size comes at the expense of additional time and computation. Consequently, a variety of techniques to reduce the cost while maintaining performance have been proposed, e.g., pruning and sparsity [2]–[4].

Dynamic neural networks [5] encompass a range of techniques to adaptively modify the structure or parameters of a neural network in response to the input and have a number of advantages in terms of accuracy and computational efficiency. Conditional computation [6] is one such technique that selectively activates parts of the network on a per instance basis. For example, a separate routing network has been used successfully to form layers of sparsely-gated Mixture-of-Experts in which thousands of feed-forward sub-networks may be activated [7].

Manuscript received August 5, 2020; revised November 2, 2020 and February 25, 2021; accepted May 1, 2021. Date of current version May 11, 2021. (Corresponding author: Richard J. Preen.)

R. J. Preen and L. Bull are with the Department of Computer Science and Creative Technologies, University of the West of England, Bristol BS16 1QY, U.K. (e-mail: richard2.preen@uwe.ac.uk; larry.bull@uwe.ac.uk).

S. W. Wilson is with Prediction Dynamics, Concord, MA 01742 USA (e-mail: wilson@prediction-dynamics.com)

Digital Object Identifier 10.1109/TEVC.2021.3079320

Similar to most machine learning techniques, the learning classifier system (LCS) XCSF [8] attempts to find solutions that are accurate and maximally general over the global input space. However, it maintains the additional power to adaptively subdivide the input space into an ensemble of simpler local approximations that together cover the domain.

XCSF performs conditional computation through the use of individual gating/guarding components associated with each local approximation. Reward is therefore allocated directly to the sub-solutions. This is in contrast with the traditional EA approach where the individual being rewarded (or reinforced) represents the overall solution to the problem, and credit is therefore much less direct in terms of rewarding the components actually responsible for the decision. Consequently, XCSF may experience faster convergence and reduced computational cost.

Autoencoders are data-specific compression algorithms learned automatically from examples. They form a core component of many learning systems [9] and have significantly contributed to improvements in the current state-of-the-art for speech recognition, computer vision, and natural language processing. Autoencoders are commonly used to perform dimensionality reduction, data denoising, imputing missing data, and anomaly detection. They may be combined with a predictive component and further refined under a supervised scheme.

Encoders and decoders of different computational complexity are required depending on the specific application. For example, a server with large amounts of available computational power can afford to spend more time encoding data, but when sent to a low power mobile device a smaller decoder may be needed.

Usually, autoencoders are trained as single large networks using standard backpropagation techniques [10]. However, an ensemble of autoencoders used for image compression has recently demonstrated superior performance than JPEG-2000 [11]. In this approach, the most suitable autoencoder is used for compression and an additional byte then added to the coding cost in order to identify which autoencoder within the ensemble was used.

While LCS have been extensively applied to reinforcement learning and supervised learning tasks, their use for unsupervised learning remains almost unexplored. Following on very initial work with a simple LCS [12], we suggest that an XCSF-like system might be capable of building an emergent ensemble of heterogeneous autoencoders with possible advantages in performance and efficiency over global model building techniques.

For example, for a specified distortion/error, a collection of LCS individuals, each smaller than the equivalent global solution, could potentially be used in a similar approach to [11],

thereby reducing byte size and reducing the cost of decoding.

In particular, this article makes the following contributions.

- 1) The XCSF classifier system is adapted for the autoencoder problem and tested on numerous data sets for the first time.
- 2) The performance of neural networks is explored where the number of neurons as well as the connectivity are evolved, i.e., heterogeneous niched encoders may emerge.
- 3) A self-adaptive scheme is introduced wherein each layer adapts to a local rate of gradient descent.
- 4) A means is provided by which a target error is specified and the system automatically designs maximally compressed networks with the desired reconstruction error. This is in contrast with the traditional approach of manually specifying the network architecture and imposing predefined penalty functions and sparsity constraints.
- 5) The hypothesis that LCS adaptive niching can provide improvements in performance is empirically tested by comparison with the global model building equivalent.

The remainder of this article is organised as follows. Section II describes the XCSF classifier system, and presents an overview of the related work on neural classifiers and autoencoders. Section III describes the neural classifier representation and learning scheme adopted, along with the experimental method applied. Section IV presents the results from experimentation on a range of publicly available data sets. Section V presents our conclusions.

## II. BACKGROUND

### A. XCSF Classifier System

XCSF is an accuracy-based online evolutionary machine learning system with locally approximating functions that compute classifier payoff prediction directly from the input state. XCSF can be seen as a generalisation of XCS [13] where the prediction is a scalar value.

XCSF is rule-based and maintains a population of classifiers, where each classifier  $cl$  consists of (i) a condition component  $cl.C$  that determines whether the rule matches input  $\vec{x}$  (ii) an action component  $cl.A$  that selects an action  $a$  to be performed for a given  $\vec{x}$  (iii) a prediction component  $cl.P$  that computes the expected payoff for performing  $a$  upon receipt of  $\vec{x}$ . XCSF thus generates rules of the general form: IF  $matches \leftarrow cl.C(\vec{x})$  THEN perform action  $a \leftarrow cl.A(\vec{x})$  and EXPECT payoff  $\vec{p} \leftarrow cl.P(\vec{x})$ .

For each step within a learning trial, XCSF constructs a match set  $[M]$  composed of classifiers in the population set  $[P]$  whose  $cl.C$  matches  $\vec{x}$ . If  $[M]$  contains fewer than  $\theta_{\text{mna}}$  actions, a covering mechanism generates classifiers with matching  $cl.C$  and random  $cl.A$ . For each possible action  $a_k$  in  $[M]$ , XCSF estimates the expected payoff by computing the fitness-weighted average as a system prediction  $P(a_k)$ . That is, for each action  $a_k$  and classifier prediction  $p_j$  in  $[M]$ , the system prediction  $P_k = \sum_j F_j p_j / \sum_j F_j$ . A system action is then randomly or probabilistically selected during exploration, and the highest payoff action  $\arg \max P_k$  used during exploitation. Classifiers in  $[M]$  advocating the chosen

action are subsequently used to construct an action set  $[A]$ . The action is then performed and a scalar reward  $r \in \mathbb{R}$  received, along with the next sensory input.

Upon reaching a terminal state within the environment (as is always the case in single-step problems), each classifier  $cl_j \in [A]$  has its experience  $exp$  incremented and fitness  $F$ , error  $\epsilon$ , and set size  $as$  updated using the Widrow-Hoff delta rule with learning rate  $\beta \in [0, 1]$  as follows.

- ▷ Error:  $\epsilon_j \leftarrow \epsilon_j + \beta(|r - p_j| - \epsilon_j)$
- ▷ Accuracy:  $\kappa_j = \begin{cases} 1 & \text{if } \epsilon_j < \epsilon_0 \\ \alpha(\epsilon_j/\epsilon_0)^{-\nu} & \text{otherwise.} \end{cases}$

With target error threshold  $\epsilon_0$  and accuracy fall-off rate  $\alpha \in [0, 1]$ ,  $\nu \in \mathbb{N}_{>0}$ .

- ▷ Relative accuracy:  $\kappa'_j = \kappa_j \cdot num_j / \sum_j \kappa_j \cdot num_j$

Where classifier numerosity initialised  $num = 1$ .

- ▷ Fitness:  $F_j \leftarrow F_j + \beta(\kappa'_j - F_j)$
- ▷ Set size estimate:  $as_j \leftarrow as_j + \beta(|[A]| - as_j)$

Thereafter,  $cl.C$ ,  $cl.A$ , and  $cl.P$  are updated according to the representation adopted.

The EA is applied to classifiers within  $[A]$  if the average set time since its previous execution exceeds  $\theta_{\text{EA}}$ . Upon invocation, the time stamp  $ts$  of each classifier is updated. Two parents are chosen based on their fitness via roulette wheel (or tournament) selection and  $\lambda$  number of offspring are created via crossover with probability  $\chi$  and mutation with probability  $\mu$ . Offspring parameters are initialised by setting the error and fitness to the parental average, and discounted by reduction parameters for error  $\epsilon_R$  and fitness  $F_R$ . Offspring  $exp$  and  $num$  are set to 1.

If subsumption is enabled and the offspring are subsumed by either parent with sufficient accuracy ( $\epsilon_j < \epsilon_0$ ) and experience ( $exp_j > \theta_{\text{sub}}$ ), it is not included in  $[P]$ ; instead the parents' micro-classifier numerosity  $num$  is incremented. The resulting offspring are added to  $[P]$  and the maximum (micro-classifier) population size  $N$  is enforced by removing classifiers selected via roulette wheel with the deletion vote.

The deletion vote for each  $cl \in [P]$  is set proportionally to the action set size estimate  $as$ . However, the vote is increased by a factor  $\bar{F}/F_j$  for classifiers that are sufficiently experienced ( $exp_j > \theta_{\text{del}}$ ) and with small fitness  $F_j < \delta \bar{F}$ ; where  $\bar{F}$  is the  $[P]$  mean fitness, and typically  $\delta = 0.1$ . Classifiers selected for deletion have their (micro-classifier)  $num$  decremented, and in the event that  $num < 1$  are removed from  $[P]$ .

In multi-step problems, the previous action set  $[A]_{-1}$  is updated after each step with a  $\gamma \in [0, 1]$  discounted reward, similar to  $Q$ -learning, and the EA may be run therein. For regression problems, a single (dummy) action is performed such that  $[A] = [M]$  and the system prediction is made directly accessible to the environment. See schematic in Fig. 1.

A number of interacting pressures have been identified within XCS [14]. A set pressure provides more frequent reproduction opportunities for more general rules. In opposition is a fitness pressure which represses the reproduction of inaccurate and over-general rules. See [15] for an overview of LCS and [16] for a detailed introduction to XCSF.

Many forms of  $cl.C$ ,  $cl.A$ , and  $cl.P$  have been used for classifier knowledge since the original ternary conditions, integer actions, and scalar predictions. Notable examples

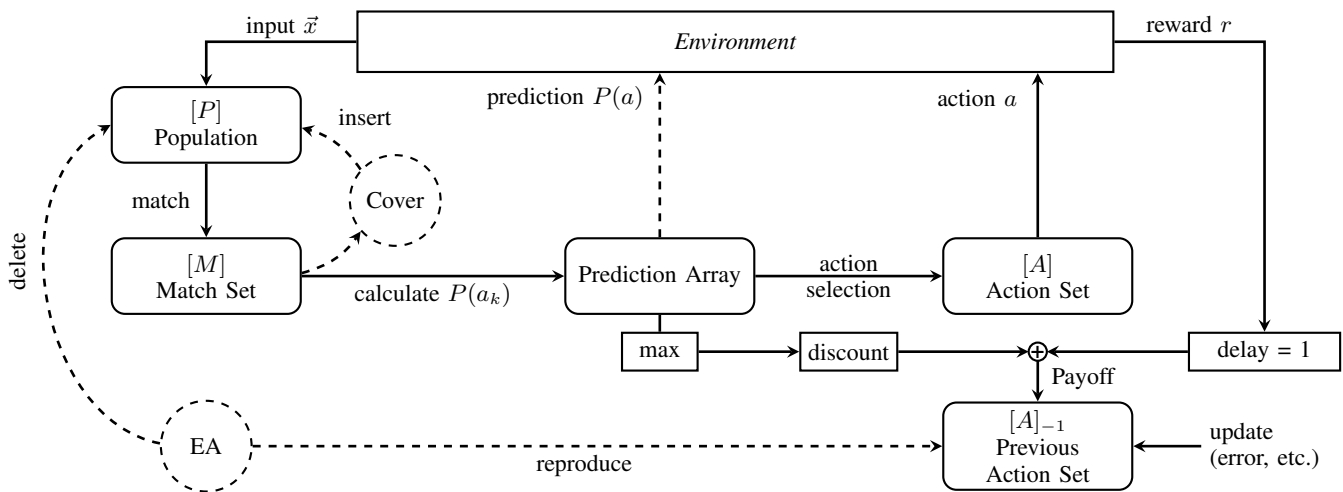


Fig. 1: XCSF schematic illustration. For supervised learning, a single (dummy) action is performed such that  $[A] = [M]$  and the system prediction is made directly accessible to the environment; classifier updates and the EA thus performed in  $[M]$ .

include, hyperrectangle [17], symbolic tree [18], and Haar-like feature conditions [19]; least squares [8] and support vector predictions [20]; hyperellipsoidal conditions and recursive least squares predictions [21]; fuzzy logic [22]; temporally dynamic graphs [23], and neural networks [24].

Perhaps somewhat surprisingly, there had been no previous use of XCS for extracting structure within unlabelled data until the work of [25] on clustering. They showed how the XCS generalisation mechanisms can be used to identify clusters, both their number and description.

### B. Evolving Neural Classifiers

A long history of searching neural network topologies can be traced back to the origins of computing. Evolutionary, Bayesian, and reinforcement learning methods are currently widely used approaches [1], [26]. A recent survey of evolutionary neural architecture search may be found in [27] wherein a large number of proposed approaches are analysed in terms of encoding and search spaces, encoding and architecture strategies, evolutionary operators and selection strategies. These approaches typically evolve large global networks whose structure and parameters remains static across different inputs and are represented as directed graphs where each node is a module representing an entire layer or sub-network, e.g., [28].

While EAs usually combine stochastic gradient descent with an evolutionary search of the architecture, it has been suggested that EAs are competitive with stochastic gradient descent on high-dimensional problems, particularly in the case of reinforcement learning [29]. It has also recently been shown that evolving the network architecture without explicit weight training can produce similar results to fixed architectures where all weights are adapted [30].

EAs are able to optimise neural networks even when there is no gradient information available. Moreover, several approaches exist wherein they may be combined with gradient descent techniques. Under a Lamarckian scheme, the learned weights remain as part of the genetic code for evolutionary operators to act upon [31]. In contrast, with Baldwinian evolution, lifetime

learning is not directly reflected within the genome, but still influences selection [32].

Adaptive gradient descent methods such as AdaGrad, RMSPprop, and Adam have become increasingly popular. These scale the magnitude of update for each individual parameter based on various moments of the gradient. However, they frequently require some form of annealing (or warm-up schedule) to maintain early stability. These warm-up parameters typically require tuning for a specific problem and model; and the benefits over simple stochastic gradient descent with an appropriate learning rate remain controversial [33].

There has also been a long history of comparison between LCS and neural networks. For example, [34] compared classifiers with the hidden neurons of a single neural network. [35] used an EA with fitness sharing to perform layer-by-layer training of a neural network. In their approach, each individual represents a hidden neuron and the number is allowed to vary within each layer. Neurons are partitioned into sets that perform similar functions and a representative from each set is chosen to form the layer. Layers are added after a fixed number of search generations. Fitness sharing encourages the formation of different feature detectors (hidden neurons) within the population.

[24] was the first to represent LCS classifiers as neural networks: both *cl.C* and *cl.A* were performed within a single network rule. Self-adaptive mutation [36] and stochastic gradient descent [37] were subsequently applied. In the latter, local search was performed by adapting the weights of the least fit networks in  $[A]$  towards the fittest rule in the set.

[38] used neural classifiers for function approximation where gradient descent was used to update the *cl.P* weights using the target outputs—there single networks performed *cl.C* and *cl.P*. With the inclusion of an additional classifier network to predict the next state input, [39] extended the approach for anticipatory LCS. More recently, [40] have explored the more biologically plausible spiking neural networks within LCS, adapting both the number of neurons and connections to perform temporal reinforcement learning.

Since the  $cl.C$  is an individual component of a classifier, different knowledge representations may be paired with neural networks used for  $cl.P$  or  $cl.A$ . For example, [41] used hyperrectangle  $cl.C$  and neural network  $cl.P$  within XCSF where the EA adapted the network topology but not the weights. This separation of concerns enable different encodings, e.g., symbolic trees or graphs, to be easily interchanged depending on the problem.

Furthermore,  $cl.C$  such as hyperrectangles can be used effectively as simple gating functions. Whereas shared representations forming a single network rule [24] enable the  $cl.C$  to not only perform matching functionality, but also transform the inputs processed by  $cl.P$ . Cyclic graphs within XCSF have previously been used in this way to exploit the memory of processing each input, evolving gated recurrent networks [42].

Recently, [43] have investigated an LCS where the EA performs feature selection using bitstring conditions and a selection of convolutional neural network actions are used. LCS therefore are not mutually exclusive of novel and application-specific evolutionary NAS representations and operators, but provide a general framework within which they may be incorporated to gain additional benefits from adaptive niching and problem decomposition.

### C. Autoencoding

Autoencoders are composed of an encoder and decoder, which are jointly trained to minimise the discrepancy between the original input data and its reconstruction. To capture useful structure, the encoder must be prevented from simply learning an identity function. Constraining the size of the encoder, applying regularisation techniques, penalty terms and sparsity constraints are all widely used for this purpose [10].

Autoencoders have a wide range of applications even without any labelled data. For example, imputing missing data values and anomaly detection [44]. They are frequently used in computer vision and image editing, e.g., colourising black-and-white images [45], denoising images, inpainting missing regions, removing watermarks, and sharpening images [46]. Of particular use with categorical data, the trained encoder can be used to visualise data within the latent space, e.g., finding the nearest neighbours within the compressed space rather than the original input features. Multi-modal learning can be performed by jointly training an autoencoder to reconstruct multiple data modalities such as vision and language [47]. Recurrent neural network autoencoders can be used for sequence learning [48] and video compression.

A general overview of representation learning may be found in [10], and EA approaches to feature selection in [49]. EAs have frequently been used to design autoencoders. For example, [50] used an EA to design the topology of compositional pattern producing networks (CPPNs) where the outputs were taken as the weights of a neural network autoencoder. The autoencoder was subsequently refined via gradient descent and the resulting gradients used to update the CPPN weights. Other examples include the EA design of denoising autoencoders for transfer learning by treating previously optimised solutions as corrupted solutions for newly encountered problems [51].

Additionally, EAs have been used to design deep sparse autoencoders via differential evolution [52] and with multi-objective optimisation [53]. Particle swarm optimisation has also been used to design the architecture of deep convolutional autoencoders [54]. Furthermore, in [55] evolutionary search is simultaneously performed on multiple tasks through the use of a denoising autoencoder. EAs have also recently been used to evolve deep neural networks for classification tasks with autoencoding components [56], [57].

Similar to training neural networks for supervised learning tasks, the predominant approach to constructing autoencoders has been to train single large global networks. However, [11] found superior compression results when compared with JPEG-2000 by training multiple hand-crafted convolutional networks. Each autoencoder in the ensemble was optimised for a particular rate distortion trade-off. The autoencoder producing the smallest distortion was then chosen for each image and bit rate, and an extra byte added to identify the chosen autoencoder.

Autoencoding via a single neural network has recently been used with XCS [58]. The feature inputs were initially passed through a pretrained encoder to reduce the dimensionality before performing XCS classification with an interval encoding.

## III. METHODOLOGY

Here, we use a derivative of XCSF to explore the effects of adaptive niching in the automatic design of an ensemble of multi-layer perceptron autoencoders. That is, each classifier is trained to reproduce its inputs via a much smaller (encoding) hidden layer. Each  $cl.C$  and  $cl.P$  is a separate fully-connected neural network, as illustrated in Fig. 2. Each network is composed of hidden scaled exponential linear units (SELUs) [59], and logistic outputs. The  $cl.C$  output layer contains a single neuron that determines whether the rule matches a given input. The  $cl.P$  (decoding) output layer contains as many output neurons as inputs.

A population of  $N = 500$  classifiers are initialised randomly and undertake Lamarckian learning. That is, after the application of evolutionary operators to both  $cl.C$  and  $cl.P$  during reproduction, stochastic gradient descent updates  $cl.P$  during reinforcement. The resulting  $cl.P$  weights are copied to offspring upon parental selection.

During instantiation of  $[P]$  the weights of each network are initialised with small random values sampled from a Gaussian normal distribution with mean  $m = 0$  and standard deviation  $\sigma = 0.1$ . Biases are zero initialised. Should covering be triggered at any stage, networks with random weights and biases are generated by the same method until the network matches the current input, however using a larger  $\sigma = 1$ . Upon receipt of  $\vec{x}$ ,  $[M]$  is formed by adding all  $cl \in [P]$  whose  $cl.C$  outputs a value greater than 0.5.

Classifier reinforcement and the EA take place within  $[M]$ . The  $[M]$  fitness-weighted average prediction is also used for system output as is usual in XCSF. However, here learning consists of updating the matching error, which is derived from the mean squared error (MSE) with respect to  $\vec{x}$  and the corresponding values on each output neuron  $\vec{O}$  of a rule

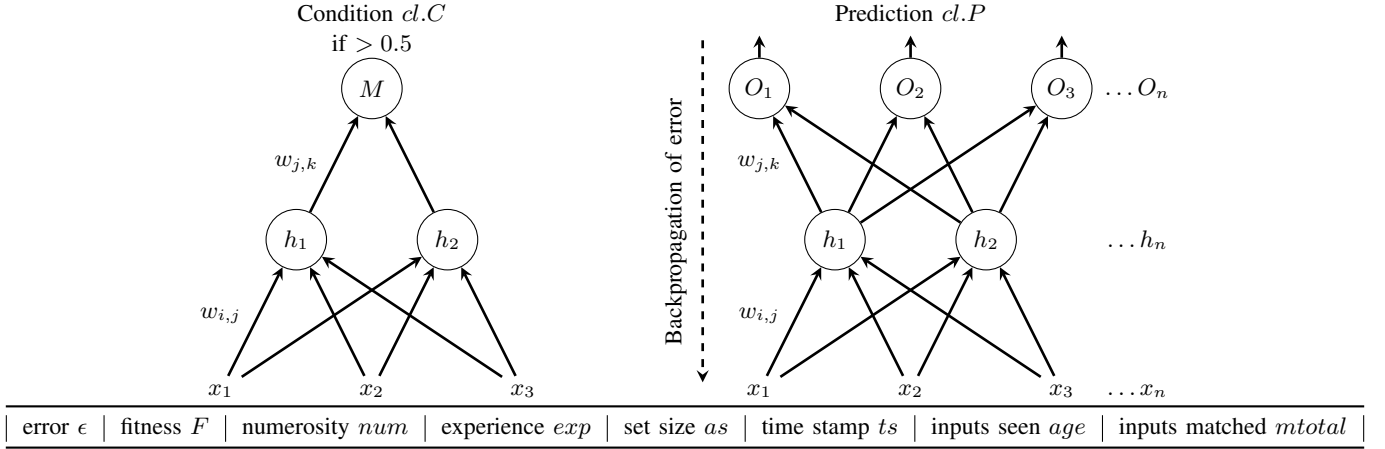


Fig. 2: Neural classifier knowledge representation. Separate fully-connected feed-forward networks calculate classifier matching and prediction. Each layer is encoded as a vector of weights (and biases), along with a binary vector indicating whether each connection is active, an activation function, and gradient descent rate. Each layer maintains its own vector of mutation rates  $\vec{\mu}$ .

in the current  $[M]$  using the modified Widrow-Hoff delta rule with learning rate  $\beta$ :

$$\epsilon_j \leftarrow \epsilon_j + \beta \left[ \frac{1}{n} \sum_{i=1}^n (x_i - O_i)^2 - \epsilon_j \right] \quad (1)$$

Subsequently, each  $cl.P$  within  $[M]$  is updated using simple stochastic gradient descent [60] with a layer-specific learning rate  $\eta \in \mathbb{R}_{>0}$  and momentum  $\omega \in [0, 1]$ . That is, the chain rule is applied at match time  $t$  to compute the partial derivative of the error with respect to each weight  $\partial\epsilon/\partial w$ , and the weight change:

$$\Delta w_t = -\eta \partial\epsilon/\partial w_t + \omega \Delta w_{t-1} \quad (2)$$

Gradient descent is not applied to  $cl.C$ .

Following [36] crossover is omitted and self-adaptive mutation used. However, here each layer within each classifier maintains a vector of mutation rates initially seeded randomly from a uniform distribution  $\vec{\mu} \sim U[\mu_{\min}, 1]$ . These parameters are passed from parent to offspring. The offspring then applies each of these mutation rates to itself using a Gaussian distribution, i.e.,  $\mu'_i = \mu_i e^{\mathcal{N}(0,1)}$ , before mutating the rest of the rule at the resulting rate. This is similar to the approach used in evolution strategies (ES) where the mutation rate is a locally evolving entity in itself, i.e., it adapts during the search process. Self-adaptive mutation not only reduces the number of hand-tunable parameters of the EA, it has also been shown to improve performance. Here, four types of mutation are explored such that for each layer:

- Weights and biases are adapted through the use of a single self-adaptive mutation rate, which controls the  $\sigma$  of a random Gaussian added to each weight and bias. This is also similar to the approach used in ES.
- A second self-adaptive rate controls the number of hidden neurons to add or remove. This value is discretised into the range  $[-h_M, h_M]$  with  $h_M$  determining the maximum number of neurons that may be added or removed per mutation event. Pressure to evolve minimally sized networks is achieved by altering the population size

enforcement mechanism as follows. Each time a classifier must be removed, two classifiers are selected via roulette wheel with the deletion vote as described above and then the rule with the most hidden layer nodes is deleted.

- To adapt the rate of gradient descent, each layer maintains its own  $\eta$ . These values are constrained  $[10^{-4}, 0.01]$  and seeded uniformly random. A third self-adaptive mutation rate controls the  $\sigma$  of a random Gaussian added to each  $\eta$ , similar to weight adaptation. [61] have previously shown how the self-adaptation of local search parameters can speed learning within XCS. Here it enables the learning rate to continually adapt to the parameters throughout the search process, e.g., potentially performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features.
- A fourth self-adaptive rate controls the probability of enabling or disabling each connection within the layer. This may encourage a more efficient sparse representation within the networks. Networks are always initialised fully-connected. When a connection is disabled, the corresponding weight value is set to zero and is excluded from mutation and gradient descent updates. Upon activation, the weight is set to a small random value  $\sim \mathcal{N}(0, 0.1)$ . When connection mutation is enabled, the connections of newly added neurons are activated with 50% probability. See outline in Algorithm 1.

Since the possibility exists that a  $cl.C$  (not generated through covering) may never match any inputs, any classifiers that have not matched any inputs within 10000 trials since creation are selected for removal during population deletion.

As a measure of generalisation we report the fraction of inputs matched by the single best rule  $cl_{\text{mfrac}}^*$ . This rule is determined as follows. If no classifier has an error below  $\epsilon_0$ , the classifier with the lowest error is chosen. If more than one classifier has an error below  $\epsilon_0$ , the classifier that matches the largest number of inputs is used.

To test the hypothesis that LCS adaptive niching can improve

**Algorithm 1: Offspring Creation.**

```

1  $\{c1p, c2p\} \in [M] \leftarrow \text{SelectParentsRouletteFitness}()$ 
2 for  $\lambda/2$  number of times do
3    $\{c1, c2\} \leftarrow \text{Copy}(c1p, c2p)$ 
4   for  $cl \in \{c1, c2\}$  do
5     for layer  $\ell \in \{cl.C, cl.P\}$  do
6       // self-adapt mutation rates
7        $\ell.\mu_i \leftarrow \ell.\mu_i e^{\mathcal{N}(0,1)}$ 
8       // mutate gradient descent rate
9        $\ell.\eta \leftarrow \ell.\eta + \mathcal{N}(0, \ell.\mu_1)$ 
10      // mutate number of neurons
11       $\ell.h \leftarrow \ell.h + \text{round}((2 \times \ell.\mu_2 - 1) \times h_M)$ 
12      // mutate connectivity
13      for  $w_i \in \ell.\vec{w}$  do
14        if  $U[0, 1] < \ell.\mu_3$  then
15          if  $a_i$  is enabled then
16             $w_i, a_i \leftarrow 0$ , disabled
17          else
18             $w_i, a_i \leftarrow \mathcal{N}(0, 0.1)$ , enabled
19      // mutate weight magnitudes
20      for  $w_i \in \ell.\vec{w}$  do
21        if  $a_i$  is enabled then
22           $w_i \leftarrow w_i + \mathcal{N}(0, \ell.\mu_4)$ 

```

TABLE I  
LEARNING PARAMETERS

Description	Parameter	Value
Maximum population size (in micro-classifiers)	$N$	500
Population initialised with random classifiers	$P_{\text{mit}}$	true
Target error, under which accuracy is set to 1	$\epsilon_0$	0.01
Update rate for fitness, error, and set size	$\beta$	0.1
Accuracy offset (1=disabled)	$\alpha$	1
Accuracy slope	$\nu$	10
Fraction of classifiers to increase deletion vote	$\delta$	0.1
Classifier deletion threshold	$\theta_{\text{del}}$	20
Classifier initial fitness	$F_I$	0.01
Classifier initial error	$\epsilon_I$	0
Offspring fitness reduction (1=disabled)	$F_R$	0.1
Offspring error reduction (1=disabled)	$\epsilon_R$	1
Minimum number of actions in $[M]$	$\theta_{\text{mna}}$	1
EA invocation frequency	$\theta_{\text{EA}}$	50
Number of offspring per EA invocation	$\lambda$	2
Crossover probability	$\chi$	0
Minimum self-adaptive mutation value	$\mu_{\text{min}}$	$10^{-4}$
Stochastic gradient descent momentum	$\omega$	0.9
Initial number of hidden neurons	$h_I$	1
Max. neurons added or removed per mut.	$h_M$	{1,2,5}
Whether EA subsumption is performed	$EA_{\text{Subsume}}$	false
Whether set subsumption is performed	$Set_{\text{Subsume}}$	false

performance, we compare results with the same system, where however  $cl.C$  always match  $\vec{x}$ . Classifier updates and the EA are thus performed within  $[P]$ , and single networks that cover the entire state-space are designed. This configuration operates as a more traditional EA, and we henceforth refer to this system as the EA. The EA acts as a control with which we can make direct comparison since it operates with the same evolutionary and gradient descent operators, and uses the same parameters. Any future changes or improvements made to the EA may be incorporated within XCSF.

When comparing XCSF and the EA on a single data set,

we use the Wilcoxon ranked-sum test, with the null hypothesis that all observed results come from the same distribution. To measure the performance across the first 100000 trials, we also present the area under the curve (AUC) results using a composite Simpson's rule applied to the mean errors.

When making comparisons across multiple data sets we follow the recommendations of [62] and use the Wilcoxon signed-rank test with the null hypothesis that taken across all data sets there is no difference in performance. All tests are applied with a 95% confidence interval.

The following publicly available data sets are used for initial evaluation from <https://www.openml.org>

- 1) USPS DIGITS: 256 features; 10 classes; 9298 instances. OpenML ID: 41082.
- 2) MNIST DIGITS: 784 features; 10 classes; 70000 instances. OpenML ID: 554.
- 3) MNIST FASHION: 784 features; 10 classes; 70000 instances. OpenML ID: 40996.
- 4) CIFAR10: 3072 features; 10 classes; 60000 instances. OpenML ID: 40927.

Table I lists the parameters used. Most values are typical defaults (e.g., [21]), with the primary exception of a larger  $\nu$  increasing fitness pressure. This was found to speed convergence of XCSF and the EA similarly during initial testing. All graphs presented depict mean  $[P]$  values over 10 runs for 100000 trials. Inputs are scaled  $[0, 1]$  and instances are drawn at random. 90% of the sample instances are used for training and 10% reserved for testing.

## IV. RESULTS

### A. Neuron Growth Rates

The performance of XCSF and the EA without connection mutation on the MNIST data sets is shown in Fig. 3. As can be seen, across the 100000 trials with a maximum growth rate of  $h_M = 1$ , XCSF achieves a smaller error than the EA. On both MNIST DIGITS and MNIST FASHION, XCSF has a smaller AUC (2287.95, 2004.95) than the EA (3184.24, 2339.69).

From Table II it can be seen that when comparing early learning performance at 20000 trials on MNIST DIGITS with  $h_M = 1$ , the XCSF mean error is significantly smaller than the EA. Similarly on MNIST FASHION after 20000 trials, the XCSF mean error is significantly smaller than the EA.

On MNIST DIGITS with  $h_M = 1$ , the mean XCSF error reaches  $\epsilon_0$  after 97000 trials, whereas the EA does not do so within the 100000 trials run. Comparing performance at 97000 trials, the XCSF mean error is significantly smaller than the EA. While neither XCSF nor the EA with  $h_M = 1$  were able to reach the target error within 100000 trials on MNIST FASHION, XCSF has a significantly smaller error after 100000 trials than the EA.

Increasing the growth rates clearly results in faster error convergence for both the EA and XCSF, with significantly smaller errors observed when compared with  $h_M = 1$  after 100000 trials. On MNIST DIGITS with  $h_M = 2$ , the mean XCSF error reaches  $\epsilon_0$  after 43000 trials, and the EA does so after 47000 trials. The  $h_M = 2$  XCSF AUC = 1415.97 and EA AUC = 1801.79, showing again that XCSF is faster than the EA



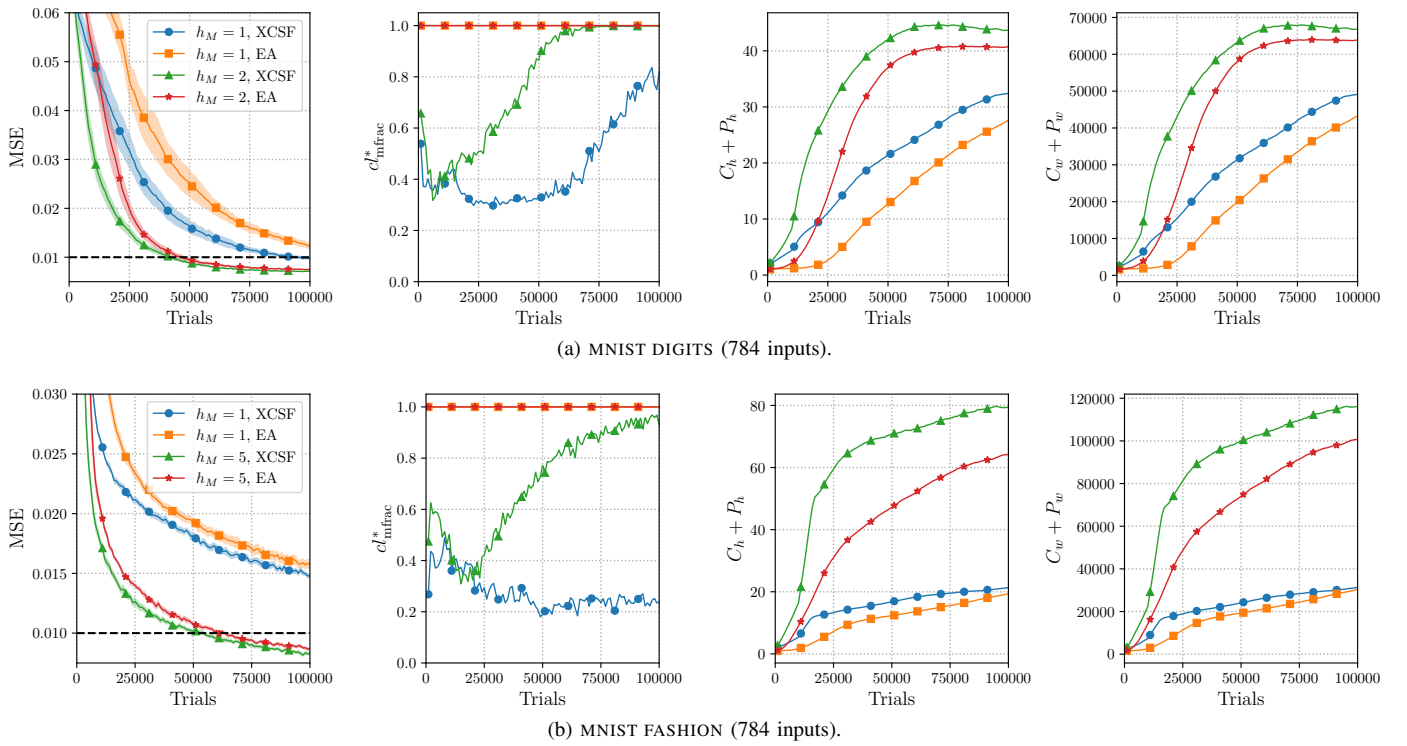


Fig. 3: The effect of maximum growth rates on the MNIST data sets. Shown are the mean squared error (MSE), fraction of inputs matched by the best rule ( $cl_{\text{frac}}^*$ ), condition hidden neurons ( $C_h$ ), and prediction hidden neurons ( $P_h$ ) for the EA and XCSF with different maximum neuron growth/removal per mutation event  $h_M$ .

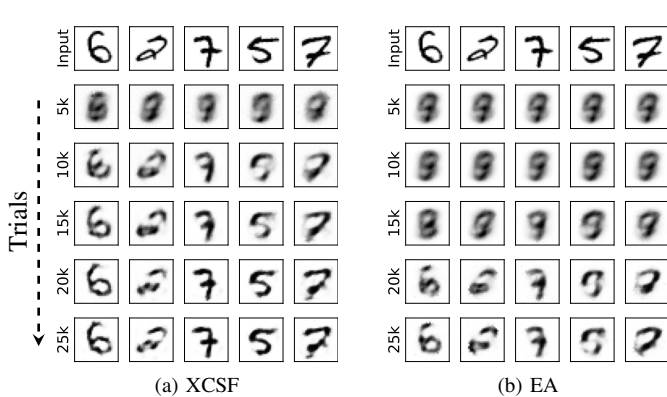


Fig. 4: MNIST DIGITS test set reconstruction. Single run with of the EA and XCSF for 25000 trials; no connection mutation;  $h_M = 2$ . XCSF MSE = 0.0122 and  $P_h = 28.2$ ,  $cl_{\text{frac}}^* = 1$ ; EA MSE = 0.0253 and  $P_h = 6.1$ .

and that  $h_M = 2$  results in a smaller error across the whole 100000 trials.

XCSF early convergence with  $h_M = 2$  on MNIST DIGITS is again faster than the EA after 20000 trials. This difference in early learning performance can be observed qualitatively in Fig. 4, which shows the XCSF and EA reconstructions of a sample of images from the MNIST DIGITS test set over the first 25000 trials.

While the mean XCSF error after 100000 trials with  $h_M = 2$  on MNIST DIGITS is not significantly different than the EA,

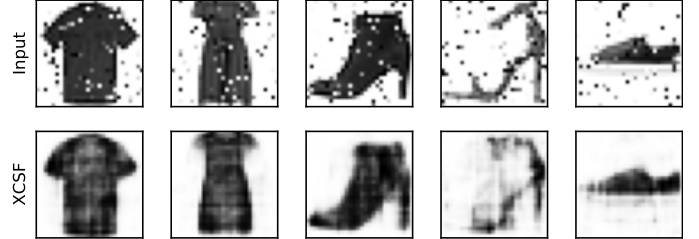


Fig. 5: XCSF reconstruction of samples from MNIST FASHION test set with 10% salt and pepper noise after 100000 trials. Training MSE = 0.0082 and  $P_h = 67.2$ .

the XCSF mean, min and median are all smaller.

With  $h_M = 5$  on MNIST FASHION, XCSF has a smaller AUC (1212.36) than the EA (1381.6) and early convergence is again faster with XCSF. After 20000 trials, XCSF has a significantly smaller error than the EA. Furthermore, XCSF reaches  $\epsilon_0$  after only 55000 trials, whereas the EA reaches the threshold after 64000 trials. Comparing performance after 55000 trials, shows that XCSF has a significantly smaller error.

Fig. 5 and Fig. 6 show the XCSF reconstruction after 100000 trials on MNIST FASHION where 10% and 20% salt and pepper noise has been added to the test images presented as input. As can be seen, an efficient representation has been learned, which can be used to effectively denoise the data. Fig. 7 shows the XCSF reconstruction where random cutout has been applied to the test images, showing how the learned representation can be used to impute missing values.

TABLE II

AUTOENCODING ERRORS. STATISTICALLY SIGNIFICANT VALUES IN BOLDFACE. SHADED ROWS WITH SMALLEST MEAN. TRIALS IN 1000S. MAXIMUM NEURON GROWTH RATE ( $h_M$ ); MAXIMUM NUMBER OF NEURONS ( $h_{\max}$ ); WITH CONNECTION MUTATION ENABLED (CMUT).

Algorithm	Trials	$h_M$	MSE $\pm$ SE	Min	Median
<b>MNIST DIGITS</b>					
XCSF	20	1	<b>0.0373<math>\pm</math>0.0043</b>	0.0218	0.0316
EA	20	1	0.0570 $\pm$ 0.0036	0.0307	0.0619
XCSF	97	1	<b>0.0097<math>\pm</math>0.0003</b>	0.0082	0.0098
EA	97	1	0.0128 $\pm$ 0.0036	0.0097	0.0126
XCSF	20	2	<b>0.0182<math>\pm</math>0.0015</b>	0.0128	0.0174
EA	20	2	0.0285 $\pm$ 0.0044	0.0150	0.0259
XCSF	43	2	0.0098 $\pm$ 0.0004	0.0080	0.0101
EA	43	2	0.0108 $\pm$ 0.0006	0.0083	0.0106
<b>MNIST FASHION</b>					
XCSF	20	1	<b>0.0222<math>\pm</math>0.0003</b>	0.0211	0.0221
EA	20	1	0.0252 $\pm$ 0.0006	0.0220	0.0251
XCSF	100	1	<b>0.0148<math>\pm</math>0.0002</b>	0.0136	0.0147
EA	100	1	0.0157 $\pm$ 0.0004	0.0133	0.0156
XCSF	20	5	<b>0.0137<math>\pm</math>0.0004</b>	0.0119	0.0134
EA	20	5	0.0149 $\pm$ 0.0002	0.0140	0.0149
XCSF	55	5	<b>0.0099<math>\pm</math>0.0002</b>	0.0091	0.0099
EA	55	5	0.0105 $\pm$ 0.0002	0.0099	0.0103
<b>USPS DIGITS</b>					
XCSF	16	1	<b>0.0097<math>\pm</math>0.0005</b>	0.0071	0.0092
EA	16	1	0.0192 $\pm$ 0.0046	0.0091	0.0112
XCSF-cmut	16	1	<b>0.0115<math>\pm</math>0.0006</b>	0.0069	0.0120
EA-cmut	16	1	0.0327 $\pm$ 0.0038	0.0147	0.0354
XCSF $h_{\max}=12$	100	1	<b>0.0108<math>\pm</math>0.0002</b>	0.0099	0.0107
EA $h_{\max}=12$	100	1	0.0127 $\pm$ 0.0002	0.0115	0.0128
<b>CIFAR10</b>					
XCSF	20	5	<b>0.0406<math>\pm</math>0.0044</b>	0.0240	0.0374
EA	20	5	0.0591 $\pm$ 0.0007	0.0546	0.0595
XCSF-cmut	20	5	<b>0.0179<math>\pm</math>0.0012</b>	0.0134	0.0171
EA-cmut	20	5	0.0302 $\pm$ 0.0036	0.0184	0.0265
XCSF	100	5	<b>0.0135<math>\pm</math>0.0006</b>	0.0107	0.0140
EA	100	5	0.0163 $\pm$ 0.0009	0.0131	0.0151
XCSF-cmut	100	5	0.0099 $\pm$ 0.0002	0.0090	0.0099
EA-cmut	100	5	0.0101 $\pm$ 0.0002	0.0094	0.0102

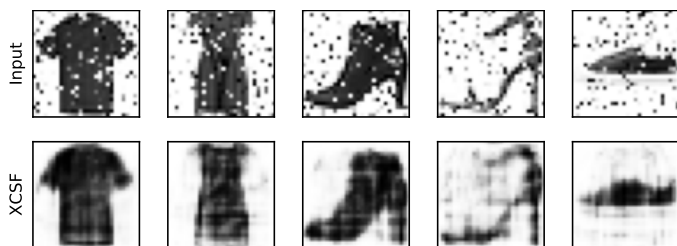


Fig. 6: XCSF reconstruction of samples from MNIST FASHION test set with 20% salt and pepper noise after 100000 trials. Training MSE = 0.0082 and  $P_h = 67.2$ .

### B. Feature Selection

The performance of XCSF and the EA with and without connection mutation on USPS DIGITS and CIFAR10 is shown in Fig. 8 and a summary of error scores in Table II. As can be seen, across the 100000 trials, XCSF achieves a smaller error than the EA. On both data sets without connection mutation, XCSF has a smaller AUC (893.56, 2568.85) than the EA (1108.87, 3722.06). Similarly with connection mutation enabled, XCSF has a smaller AUC (1148.17, 1607.72) than the EA (1439.88, 2053.15). While the AUCs with connection mutation are larger



Fig. 7: XCSF reconstruction of samples from MNIST FASHION test set with random cut out after 100000 trials. Training MSE = 0.0082 and  $P_h = 67.2$ .

on USPS DIGITS, they are smaller on CIFAR10, showing that connection mutation can be beneficial to learning. Moreover, with connection mutation enabled the number of neurons grows to a larger number, whilst the number of non-zero weights is smaller, showing that a more sparse representation is learned.

Without connection mutation on USPS DIGITS, XCSF with  $h_M = 1$  reaches  $\epsilon_0$  after 16000 trials, compared with the EA which reaches the threshold after 19000 trials. When connection mutation is enabled, XCSF reaches  $\epsilon_0$  after 33000 trials, and the EA after 28000 trials. However, comparing performance after 16000 trials (i.e., when  $\epsilon_0$  is reached without connection mutation), the XCSF error without connection mutation is not significantly different than with connection mutation enabled.

Early MSE on USPS DIGITS with XCSF both with and without connection mutation is significantly smaller than the EA. For example, after 16000 trials without connection mutation, the XCSF error is significantly smaller than the EA. Similarly at the same number of trials with connection mutation, the XCSF error is significantly smaller than the EA.

On CIFAR10 with  $h_M = 5$ , neither XCSF nor the EA reach  $\epsilon_0$  after 100000 trials without connection mutation. When connection mutation is used, XCSF reaches  $\epsilon_0$  after 100000 trials, whereas the EA does not. Comparing errors after 100000 trials shows that XCSF without connection mutation has a significantly larger error than XCSF with connection mutation. Furthermore, XCSF without connection mutation has a significantly smaller error than the EA without connection mutation. Finally, while there is no significant difference when comparing XCSF and the EA with connection mutation, the XCSF mean, min and median are all smaller.

### C. Heterogeneous Niche Ensembles

The performance of XCSF and the EA on USPS DIGITS when the maximum number of hidden neurons  $h_{\max} = 12$  is fixed below that which  $\epsilon_0$  can be attained with a global model is shown in Fig. 9 and Table II; connection mutation is disabled and  $h_M = 1$ . After 100000 trials, XCSF attains a significantly smaller training error than the EA. The XCSF AUC = 1374.65 and EA = 1700.94, confirming that XCSF is able to partition the input space and achieve a smaller error than possible with a global solution.

Applying the evolved autoencoders to a simple denoising task shows that this improvement in XCSF training error translates to a smaller test error. After adding 10% salt and pepper noise



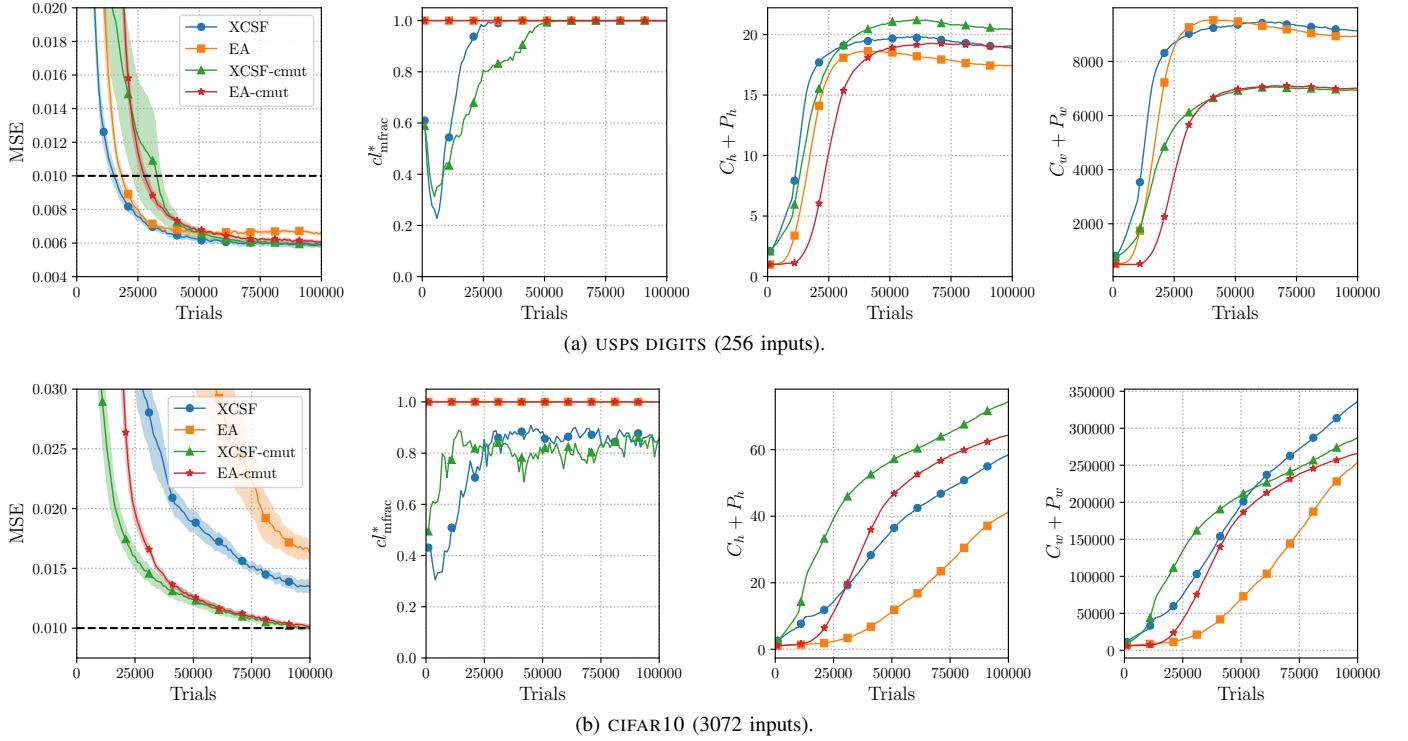


Fig. 8: The effect of feature selection on USPS DIGITS and CIFAR10. Shown are the mean squared error (MSE), fraction of inputs matched by the best rule ( $c_{\text{mfrac}}^*$ ), total number of condition ( $C_h$ ) and prediction hidden neurons ( $P_h$ ), and the total number of non-zero condition ( $C_w$ ) and prediction weights ( $P_w$ ) for the EA with (star) and without connection mutation (square), as well as XCSF with (triangle) and without connection mutation (circle). For USPS,  $h_M = 1$  and for CIFAR10,  $h_M = 5$ .

to the test set inputs and comparing the reconstructions with the original clean inputs, the XCSF MSE ( $0.017 \pm 0.001$ ) is significantly smaller than the EA ( $0.021 \pm 0.001$ ).

#### D. Summary

Across all data sets, XCSF provides faster convergence over the first 100000 trials than the EA (AUC metric). Furthermore, across all data sets, XCSF can be seen to achieve faster early convergence, and for finding a global solution is always at least as fast as the EA in number of trials to  $\epsilon_0$ . While connection mutation was found to slow convergence on the simple USPS DIGITS data set, it was found beneficial to the search process on CIFAR10, which contains a large number of inputs and highly correlated features (i.e., RGB channels).

Moreover, when the number of hidden neurons is restricted below which a global solution can reach the target error, XCSF has been shown capable of subdividing the input domain to achieve a smaller reconstruction error than is possible with the EA. For a given maximum allowable distortion  $\epsilon_0$ , the resulting decoders can therefore be made smaller and require less computation to perform decoding, which may be beneficial for low powered devices, etc.

Grouping the mean errors at 100000 trials for all data sets by algorithm, and applying the Wilcoxon signed-rank test shows that XCSF has a significantly smaller error than the EA,  $p \leq 0.0077$ . A summary of the discovered autoencoding architectures can be seen in Table III.

TABLE III  
SUMMARY OF DISCOVERED AUTOENCODING ARCHITECTURES AND TIME TO  $\epsilon_0$  AFTER 100000 TRIALS. MEAN VALUES REPORTED.

Algorithm	$C_h$	$P_h$	$C_w$	$P_w$	$\epsilon_0$	Trials
<u>MNIST DIGITS</u>						
XCSF $h_M=1$	2.1	30.3	1662	47496		97000
EA $h_M=1$	n/a	27.6	n/a	43274		n/a
XCSF $h_M=2$	2.1	41.5	1651	65110		43000
EA $h_M=2$	n/a	40.7	n/a	63801		47000
<u>MNIST FASHION</u>						
XCSF $h_M=1$	2.7	18.6	2150	29201		n/a
EA $h_M=1$	n/a	19.3	n/a	30294		n/a
XCSF $h_M=5$	10.8	68.7	8455	107671		55000
EA $h_M=5$	n/a	64.2	n/a	100667		64000
<u>USPS DIGITS</u>						
XCSF $h_M=1$	2.1	16.8	534	8593		16000
EA $h_M=1$	n/a	17.4	n/a	8940		19000
XCSF-cmut $h_M=1$	1.9	18.6	242	6689		33000
EA-cmut $h_M=1$	n/a	19.0	n/a	7008		28000
<u>CIFAR10</u>						
XCSF $h_M=5$	7.6	50.9	23374	312956		n/a
EA $h_M=5$	n/a	41.4	n/a	253995		n/a
XCSF-cmut $h_M=5$	7.2	67.3	11245	276113		100000
EA-cmut $h_M=5$	n/a	64.3	n/a	265566		n/a

For an identical number of trials, XCSF and the EA are presented with the same number of training samples. However, it should be noted that forward and backward propagation of  $cl.P$  networks are performed for all members of  $[P]$  with the

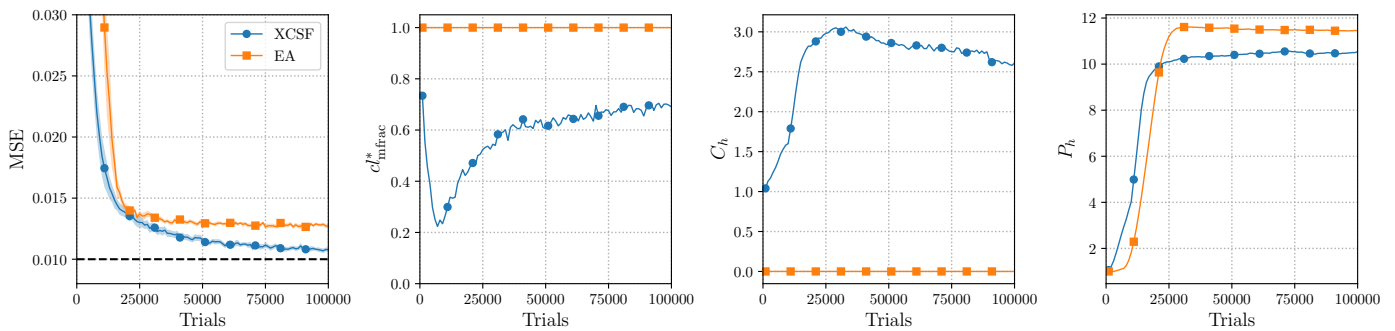


Fig. 9: The performance of the EA and XCSF with maximum number of hidden neurons  $h_{max} = 12$  on USPS DIGITS. Shown are the mean squared error (MSE), fraction of inputs matched by the best rule ( $cl_{mfrac}^*$ ), condition hidden neurons ( $C_h$ ), and prediction hidden neurons ( $P_h$ ), for the EA (square) and XCSF (circle). Connection mutation not applied;  $h_M = 1$ .

EA, whereas only those in  $[M]$  are propagated with XCSF. In this regard, XCSF performs conditional computation and can be seen as a form of dynamic pruning based on gating functions [5], as well as sharing characteristics with sparse autoencoders [52].

## V. CONCLUSION

Autoencoding is a key component of many learning systems and this article has presented the first results from using a variant of XCSF to perform such dimensionality reduction. The traditional approach to autoencoding involves manually specifying the number of neurons and using predefined constraints. The approach outlined here automatically identifies the minimal number of neurons required to reach a target error  $\epsilon_0$ —under this threshold, the system focuses on increasing the generality of solutions and pruning neurons.

LCS enable the emergence of an ensemble of structurally heterogeneous solutions to cover the problem space. In this case, when the number of neurons in the autoencoders is allowed to evolve, networks of differing complexity are typically seen to cover different areas of the problem space. Furthermore, the scheme introduced here entirely self-adapts the search process: both the gradient-free mutation of weights and their local refinement where gradient information is available. Not only does this potentially reduce the number of hand-tunable parameters, it may provide further benefits in network analysis, use in non-stationary and online domains, etc. Moreover, the LCS ensemble may reveal input categories more clearly than are seen in a global network solution. Given their basis in EAs, LCS do not require the existence of helpful gradients within the weight space, although gradient-based search can speed learning, as here.

XCSF adaptively subdivides the input domain into local approximations that are simpler than a global neural network solution, enabling a more efficient allocation of reward. Given this difference in credit allocation, further improvement in convergence might result from certain architectural modifications to the classifiers. For instance, in the present system, both condition and prediction networks use a similar architecture and are subject to the same offspring creation algorithm. It seems possible that using separate algorithms allowing wider relative complexities could reveal higher rates of convergence.

Current work is therefore exploring additional layers of autoencoding and alternative layer architectures such as those involving local connectivity and weight sharing (e.g., convolutional layers) as well as those with recurrent connections (e.g., long short-term memory layers). A future examination of alternative classifier condition representations, e.g., symbolic trees or Haar-like features, and enhancements to the evolutionary operators may yield further improvements for specific applications. Future application to supervised and reinforcement learning tasks, e.g., under a pretraining scheme, will determine the effectiveness of the suggested dimensionality reduction shown here on various data sets.

## REFERENCES

- [1] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Mach. Intell.*, vol. 1, no. 1, pp. 24–35, Jan. 2019.
- [2] W. Hua, Y. Zhou, C. M. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Proc. NeurIPS*, H. Wallach *et al.*, Eds., vol. 32. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [3] N. Lee, T. Ajanthan, S. Gould, and P. H. Torr, "A signal propagation perspective for pruning neural networks at initialization," in *Proc. ICLR*, A. Rush, Ed., May 2020.
- [4] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *arXiv*, vol. 2102.00554, Jan. 2021.
- [5] Y. Han *et al.*, "Dynamic neural networks: A survey," *arXiv*, vol. 2102.04906, Feb. 2021.
- [6] E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup, "Conditional computation in neural networks for faster models," in *Proc. ICLR*, Y. Bengio and Y. LeCun, Eds., May 2016.
- [7] N. Shazeer *et al.*, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," in *Proc. ICLR*, Y. Bengio and Y. LeCun, Eds., May 2017.
- [8] S. W. Wilson, "Function approximation with a classifier system," in *Proc. GECCO*, L. Spector *et al.*, Eds. San Francisco, CA, USA: Morgan Kaufmann, 2001, pp. 974–981.
- [9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.
- [10] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.
- [11] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy image compression with compressive autoencoders," in *Proc. ICLR*, Y. Bengio and Y. LeCun, Eds., May 2017.
- [12] L. Bull, "Autoencoding with a learning classifier system: Initial results," *arXiv*, vol. 1907.11554, Jul. 2019.
- [13] S. W. Wilson, "Classifier fitness based on accuracy," *Evol. Comput.*, vol. 3, no. 2, pp. 149–175, Summer 1995.
- [14] M. V. Butz, T. Kovacs, P.-L. Lanzi, and S. W. Wilson, "Toward a theory of generalization and learning in XCS," *IEEE Trans. Evol. Comput.*, vol. 8, no. 1, pp. 28–46, Feb. 2004.

- [15] L. Bull, "A brief history of learning classifier systems: From CS-1 to XCS and its variants," *Evol. Intell.*, vol. 8, no. 2–3, pp. 55–70, Sep. 2015.
- [16] M. V. Butz, *Rule-Based Evolutionary Online Learning Systems*, 1st ed., ser. STUDFUZZ. Berlin, Germany: Springer, 2006, vol. 191.
- [17] C. Stone and L. Bull, "For real! XCS with continuous-valued inputs," *Evol. Comput.*, vol. 11, no. 3, pp. 299–336, Fall 2003.
- [18] M. Iqbal, W. N. Browne, and M. Zhang, "Reusing building blocks of extracted knowledge to solve complex, large-scale Boolean problems," *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 465–480, Aug. 2014.
- [19] T. Ebadi, I. Kukenys, W. N. Browne, and M. Zhang, "Human-interpretable feature pattern classification system using learning classifier systems," *Evol. Comput.*, vol. 22, no. 4, pp. 629–650, Winter 2014.
- [20] D. Loiacono, A. Marelli, and P.-L. Lanzi, "Support vector regression for classifier prediction," in *Proc. GECCO*, D. Thierens *et al.*, Eds. New York, NY, USA: ACM, 2007, pp. 1806–1813.
- [21] M. V. Butz, P.-L. Lanzi, and S. W. Wilson, "Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction," *IEEE Trans. Evol. Comput.*, vol. 12, no. 3, pp. 355–376, Jun. 2008.
- [22] J. Casillas, B. Carse, and L. Bull, "Fuzzy-XCS: A Michigan genetic fuzzy system," *IEEE Trans. Fuzzy Syst.*, vol. 15, no. 4, pp. 536–550, Aug. 2007.
- [23] R. J. Preen and L. Bull, "Dynamical genetic programming in XCSF," *Evol. Comput.*, vol. 21, no. 3, pp. 361–387, Fall 2013.
- [24] L. Bull, "On using constructivism in neural classifier systems," in *Proc. PPSN VII*, ser. LNCS, J. J. M. Guervós *et al.*, Eds., vol. 2439. Berlin, Germany: Springer, Oct. 2002, pp. 558–567.
- [25] K. Tamee, L. Bull, and O. Pinnigern, "Towards clustering with XCS," in *Proc. GECCO*, D. Thierens *et al.*, Eds. New York, NY, USA: ACM, 2007, pp. 1854–1860.
- [26] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 55, pp. 1–21, Mar. 2019.
- [27] Y. Liu *et al.*, "A survey on evolutionary neural architecture search," *arXiv*, vol. 2008.10937, Aug. 2020.
- [28] Z. Lu *et al.*, "Multi-objective evolutionary design of deep convolutional neural networks for image classification," *IEEE Trans. Evol. Comput.*, in press, doi: 10.1109/TEVC.2020.3024708.
- [29] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv*, vol. 1703.03864, Mar. 2017.
- [30] A. Gaier and D. Ha, "Weight agnostic neural networks," in *Proc. NeurIPS*, H. Wallach *et al.*, Eds., vol. 32. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [31] F. Gruau and D. Whitley, "Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect," *Evol. Comput.*, vol. 1, no. 3, pp. 213–233, Fall 1993.
- [32] G. E. Hinton and S. J. Nowlan, "How learning can guide evolution," *Complex Syst.*, vol. 1, no. 3, pp. 495–502, 1987.
- [33] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," in *Proc. NeurIPS*, I. Guyon *et al.*, Eds., vol. 30. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 4148–4158.
- [34] R. E. Smith and H. B. Cribbs, "Is a learning classifier system a type of neural network?" *Evol. Comput.*, vol. 2, no. 1, pp. 19–36, Spring 1994.
- [35] H. C. Andersen and A. C. Tsoi, "A constructive algorithm for the training of a multilayer perceptron based on the genetic algorithm," *Complex Syst.*, vol. 7, no. 4, pp. 249–268, 1993.
- [36] L. Bull and J. Hurst, "A neural learning classifier system with self-adaptive constructivism," in *Proc. IEEE Congr. Evol. Comput.*, R. Sarker *et al.*, Eds., vol. 2. Piscataway, NJ, USA: IEEE Press, 2003, pp. 991–997.
- [37] T. O'Hara and L. Bull, "A memetic accuracy-based neural learning classifier system," in *Proc. IEEE Congr. Evol. Comput.*, D. Corne *et al.*, Eds., vol. 3. Piscataway, NJ, USA: IEEE Press, Sep. 2005, pp. 2040–2045.
- [38] —, "Backpropagation in accuracy-based neural learning classifier systems," in *Learning Classifier Systems*, ser. LNCS, T. Kovacs *et al.*, Eds., vol. 4399. Berlin, Germany: Springer, 2007, pp. 25–39.
- [39] —, "Building anticipations in an accuracy-based learning classifier system by use of an artificial neural network," in *Proc. IEEE Congr. Evol. Comput.*, D. Corne *et al.*, Eds., vol. 3. Piscataway, NJ, USA: IEEE Press, Sep. 2005, pp. 2046–2052.
- [40] D. Howard, L. Bull, and P.-L. Lanzi, "A cognitive architecture based on a learning classifier system with spiking classifiers," *Neural Process. Lett.*, vol. 44, no. 1, pp. 125–147, Aug. 2016.
- [41] P.-L. Lanzi and D. Loiacono, "XCSF with neural prediction," in *Proc. IEEE Congr. Evol. Comput.*, G. G. Yen *et al.*, Eds. Piscataway, NJ, USA: IEEE Press, Jul. 2006, pp. 2270–2276.
- [42] R. J. Preen and L. Bull, "Discrete and fuzzy dynamical genetic programming in the XCSF learning classifier system," *Soft Comput.*, vol. 18, no. 1, pp. 153–167, Jan. 2014.
- [43] J.-Y. Kim and S.-B. Cho, "Exploiting deep convolutional neural networks for a neural-based learning classifier system," *Neurocomputing*, vol. 354, pp. 61–70, Aug. 2019.
- [44] V. L. Cao, M. Nicolau, and J. McDermott, "Learning neural representations for network anomaly detection," *IEEE Trans. Cybern.*, vol. 49, no. 8, pp. 3074–3087, Aug. 2019.
- [45] R. Zhang, P. Isola, and A. A. Efros, "Colorful image colorization," in *Proc. Euro. Conf. Computer Vision*, ser. LNCS, B. Leibe *et al.*, Eds., vol. 9907. Berlin, Germany: Springer, 2016, pp. 649–666.
- [46] S. Li *et al.*, "Multi-channel and multi-model-based autoencoding prior for grayscale image restoration," *IEEE Trans. Image Process.*, vol. 29, pp. 142–156, Jul. 2019.
- [47] Y. Shi, N. Siddharth, B. Paige, and P. H. S. Torr, "Variational mixture-of-experts autoencoders for multi-modal deep generative models," in *Proc. NeurIPS*, H. Wallach *et al.*, Eds., vol. 32. Red Hook, NY, USA: Curran Associates Inc., 2019, pp. 15 692–15 703.
- [48] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NeurIPS*, Z. Ghahramani *et al.*, Eds., vol. 27. Red Hook, NY, USA: Curran Associates Inc., 2014, pp. 3104–3112.
- [49] B. Xue, M. Zhang, W. N. Browne, and X. Yao, "A survey on evolutionary computation approaches to feature selection," *IEEE Trans. Evol. Comput.*, vol. 20, no. 4, pp. 606–626, Aug. 2016.
- [50] C. Fernando *et al.*, "Convolution by evolution: Differentiable pattern producing networks," in *Proc. GECCO*, T. Friedrich *et al.*, Eds. New York, NY, USA: ACM, 2016, pp. 109–116.
- [51] L. Feng, Y.-S. Ong, S. Jiang, and A. Gupta, "Autoencoding evolutionary search with learning across heterogeneous problems," *IEEE Trans. Evol. Comput.*, vol. 21, no. 5, pp. 760–772, Oct. 2017.
- [52] M. Gong, J. Liu, H. Li, Q. Cai, and L. Su, "A multiobjective sparse feature learning model for deep neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 12, pp. 3263–3277, Dec. 2015.
- [53] H. Cheng, Z. Wang, Z. Wei, L. Ma, and X. Liu, "On adaptive learning framework for deep weighted sparse autoencoder: A multi-objective evolutionary algorithm," *IEEE Trans. Cybern.*, in press, doi: 10.1109/TCYB.2020.3009582.
- [54] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "A particle swarm optimization-based flexible convolutional autoencoder for image classification," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 8, pp. 2295–2309, Aug. 2019.
- [55] L. Feng *et al.*, "Evolutionary multitasking via explicit autoencoding," *IEEE Trans. Cybern.*, vol. 49, no. 9, pp. 3457–3470, Sep. 2019.
- [56] Y. Sun, G. G. Yen, and Z. Yi, "Evolving unsupervised deep neural networks for learning meaningful representations," *IEEE Trans. Evol. Comput.*, vol. 23, no. 1, pp. 89–103, Feb. 2019.
- [57] X. Chen, Y. Sun, M. Zhang, and D. Peng, "Evolving deep convolutional variational autoencoders for image classification," *IEEE Trans. Evol. Comput.*, in press, doi: 10.1109/TEVC.2020.3047220.
- [58] K. Matsumoto, T. Tatsumi, H. Sato, T. Kovacs, and K. Takadama, "XCSR learning from compressed data acquired by deep neural network," *J. Adv. Comput. Intell. Inform.*, vol. 21, no. 5, pp. 856–867, Sep. 2017.
- [59] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Proc. NeurIPS*, I. Guyon *et al.*, Eds., vol. 30. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 972–981.
- [60] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.
- [61] D. Wyatt and L. Bull, "A memetic learning classifier system for describing continuous-valued problem spaces," in *Recent Advances in Memetic Algorithms*, ser. STUDFUZZ, W. E. Hart, J. E. Smith, and N. Krasnogor, Eds., vol. 166. Berlin, Germany: Springer, 2005, pp. 355–395.
- [62] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, no. 1, pp. 1–30, Jan. 2006.



**Richard J. Preen** received the B.Sc. (Hons.) and M.Sc. degrees in computer science, and the Ph.D. degree in artificial intelligence from the University of the West of England, Bristol, U.K., in 2004, 2008, and 2011, respectively.

He is currently a Research Fellow with the Department of Computer Science and Creative Technologies at the University of the West of England.



**Larry Bull** received both the B.Sc. (Hons.) degree in computing for real-time systems and the Ph.D. degree from the University of the West of England, Bristol, U.K., in 1992 and 1995, respectively.

He is a Professor of artificial intelligence with the Department of Computer Science and Creative Technologies, University of the West of England. Prof. Bull was the founding Editor-in-Chief of the journal *Evolutionary Intelligence* and has published widely on nature inspired computation, including the monograph *The Evolution of Complexity* (Springer,

2020).



**Stewart W. Wilson** is an independent researcher in classifier systems and evolutionary computation. He received his S.B. (physics) and S.M. and Ph.D. (electrical engineering) from the Massachusetts Institute of Technology. His Ph.D. thesis under Edwin Land investigated what would happen if a child could ask questions and receive excellent spoken answers from a machine that connected him to recordings made by a scientist such as Carl Sagan. The results showed long, thoughtful lines of inquiry for this student-driven style of learning. Later, at Polaroid

Corporation, Wilson developed a first practical implementation.

In 1981 he became interested in John Holland's classifier system idea, contacted Holland and his incipient group, and took up this machine learning approach. The results, eventually, were the algorithms XCS and XCFS which made Holland's idea practical and stimulated applications and further research worldwide. From his interest in vision, Wilson did research that connected the stroboscopic imagery of Gray Walter to the structure of the retino-cortical mapping, suggesting the latter contains a scanning mechanism. In recent years, Wilson developed a system based on genetic algorithms for automatic investment portfolio definition. The concept found application with SumGrowth Strategies, LLC in its implementation of Temporal Portfolio Theory.

Dr. Wilson is a co-founder of *Adaptive Behavior* and the Simulation of Adaptive Behavior (SAB) Conferences.