Topological Evolution of Spiking Neural Networks

Sam Slade
Department of Computer and Information Sciences
Northumbria University
Newcastle, England
Email: samslade_13@hotmail.com

Li Zhang
Department of Computer and Information Sciences
Northumbria University
Newcastle, England
Email: li.zhang@northumbria.ac.uk

Abstract—**Neuro-evolution is often used to generate the parameters, topology, and rules of artificial neural networks. This technique allows for automatic configuration of a neural network. In this paper we propose a method to generate Spiking Neural Networks (SNNs) automatically called NENG (Neuro-Evolutionary Network Generation). The aim was to help alleviate the manual construction and optimization of neural network implementations. The results show the algorithm is successful at generating and improving the design of SNNs for a Classification task. After 812 generations with a population size of 20 the algorithm converges to model the Xor gate with 100% accuracy. The results show improvements to the algorithm execution time and number of neurons over time.**

I. Introduction

Neural networks have been a significant area of research in the past decade and are applied to many machine learning tasks in computer vision, audio event recognition and recommender systems. Their topology is typically designed by hand on a per case basis. Finding an optimum network topology is often a matter of applying previous experience and following the structure of previously successful neural networks then testing them. To address this problem, it would be ideal to automate the generation of a network's topology. This paper aims to provide a method for automatically optimizing network topologies by applying a Genetic Algorithm (GA) to select the nodes and connections of a spiking neural network to enable Xor Classification. The GA was chosen for it's power to provide solutions from a large search space and it's biological heritage which aligns with the biological modeling focus that drives Spiking Neural Network (SNN) development. This is achieved by implementing SNNs with a Partially Observable Markov Decision Process (POMDP) for reinforcement learning treating each neuron node as an independent agent. The entire topology of the SNNs is generated by a GA and is evaluated based on Classification metrics. Binary Classification is performed by counting spike output from the network after feeding it one of the typical Xor patterns.

This paper will start by surveying some related work regarding GAs, SNNs and Classification. The paper is organized as follows. Section II discusses related work on GAs, SNNs and Classification methods. In section III NENG will be presented and followed by the experimental results and analysis in Section IV. Conclusions will be drawn in Section V.

II. Related Work

NENG involves three areas of scientific knowledge, GA, SNNs and Classification methods. Here we will cover some related work regarding each area.

A. Genetic Algorithm

To create systems that are intended to replicate human processes it seems reasonable to take note of natures methods of doing so. GA's, much like an SNN's are inspired by modeling biological processes. Generating a neural network topology using a GA is a form of Neuro-evolution [1]. Using Neuro-evolution for optimizing network topologies is essentially a search space problem and has been used to solve many optimization problems [2] [3] [4] in the past along with other meta-heuristics such as Bee Colony Optimization [5] [6] [7] [5] [8], Particle Swarm Optimization [9] [10] [11] [12] [13] [14] and Simulated Annealing [15] [16] [17] [18]. The fundamental structure of a GA consists of three basic operations:

- *Fitness evaluation for individuals in the population*
- *Creation of a gene pool through selection*
- *Recombination through crossover and mutation*

In addition to these operations the genetic code must be defined to represent the way a network topology is formed.

Optimizing a neural network topology with GAs has been shown in [19] where direct binary encoding is used to define the genes and thus the search space of the potential topological configurations. This entails using binary values to directly represent the connections between the nodes. Selection is performed using a roulette wheel approach where probabilities are assigned to each individual and a random number is generated. If the random number lands in a specific probability range, then the individual associated with that range is selected. The probabilities are weighted such that individuals with a higher fitness have a larger probability of selection. Crossover is then performed between two individuals on the string representations of their binary encoded genes by randomly selecting a point on the string (loci) with equal probabilities and switching the remaining bits of both strings. Mutation is achieved by splitting the string into 3 key components which represent certain aspects of the topology and connection weighting of the network. Each section has a unique probability of mutation. This algorithm also employs use of the Simplex algorithm to optimize the local search of the GA. Other examples of neuro-evolution can be seen in [20] where a steady state GA is used to optimize the weights and topology of SNN's tasked with controlling a simulated Khepera II robot.

Further work utilizes a variant of the standard GA called the nondominated sorting GA II (NSGA-II) to optimize the amount of material used to satisfy a given shape within load and boundary constraints. As with other multi objective optimization algorithms, NSGA-II is concerned with finding the Pareto-optimal solution. This involves finding the set of solutions that are non-dominated. The NSGA-II algorithm is an improvement on the standard NSGA and is used to address problems of high computational complexity, lack of elitism and need for specifying a sharing parameter [21]. For selection the solutions are ranked in the order of non-dominance. The first rank consists of completely non-dominated solutions. The second rank consists of solutions which are non-dominated when ignoring the solutions in first rank and the third rank is those solutions which are non-dominated ignoring the first and second ranks ad infinitum. A crowding distance measure is used to order solutions with the same rank such that more distinct/isolated ones are favored over those which are similar. This allows for better coverage of the search space.

B. SNNs

Spiking neurons are more biologically accurate models of the neuron modelling specific details such as membrane potential and action potentials. One of the first models proposed was the Integrate and Fire model, often attributed to Louis Lapicque in a paper he wrote in 1907 where he framed the neuron membrane in terms of an electrical circuit consisting of a capacitor and a resistor in parallel, although he did not specifically mention Integrate and Fire models [22]. A paper by A.V. Hill in 1935 shows some of the first formulations of the model. He details how membrane potential changes over time to cause spiking and how it is reset to a resting potential by an adaptive factor [23]. This initial formulation did not account for the slow decay of the membrane potential over time which reduced its utility. To fix this a leak term was introduced. Equation 1 describes a leaky Integrate and Fire model derived from the Hodgkin Huxley model [24] [25] [26] with the neuron regarded as a single point in space undergoing changes in membrane potential $v(t)$ over time [24].

$$C_m \frac{dv(t)}{dt} = I_{leak}(t) + I_s(t) + I_{inj}(t) \tag{1}$$

Where Cm is the membrane capacitance, I_{leak} is the current leaked from the membrane, I_s is the current from the synapse and $I_{inj}(t)$ is the current artificially injected using a cathode; often used for experimentation. The leak term is defined in Equation 2 where V_0 is the resting potential and τ_m is the passive membrane time constant.

$$I_{leak}(t) = \frac{-C_m}{\tau_m}\left[v(t) - V_0\right] \tag{2}$$

The function describing a spike in terms of current at time t is show in Equation 3.

$$I_{spk}(t) = C_m \left[\frac{dv(t)}{dt}\right]^{-1}_{v=V_{th}} (V_{rset} - V_{th})\,\delta\left[v(t) - V_{th}\right] \tag{3}$$

Where V_{th} is the threshold at which a spike is formed, V_{rset} is the value of the membrane potential after firing and δ is the Dirac Delta function. The current at the synapse I_s can be modeled regarding conductance or current. The current model can be used when input summation is linear and the conductance model facilitates nonlinear summation where the amplitude of the synaptic inputs is dependent upon the difference between the membrane potential and reversal potential.

The Hodgkin Huxley model is a general description of many of the processes relating to propagation of the action potential and spike firing. It encompasses many of the chemical processes in mathematical form and can be used as the base description from which multiple specialized and simplified cases can be derived. Using the Hodgkin Huxley model and combining it with a Leaky Integrate and fire neuron can lead to an improvement of computation speed and simulation accuracy [27] [28]. One such model was created by Izhikevich [27] [29] [28], leading to a quadratic leaky integrate and fire neuron model. This model simulates some dynamic behaviors of biological neurons such as bursting and adaption. Recently there has been research into increasing the scale of SNN architectures so they can enjoy the same treatment received by ANN's regarding parallel GPU computing. These papers still regard the Izhikevich model as relevant for computational efficiency and biological accuracy [27] [30] [31] [32] [33]. Using Bifurcation methods, the Hodgkin Huxley model was reduced to two ordinary differential equations where $'$ is the derivative with respect to time.

$$v^{'} = 0.04v^2 + 5v + 140 - u + I \tag{4}$$

$$u^{'} = a(bv - u) \tag{5}$$

v is variable representing the membrane potential and u is membrane recovery variable akin to the I_{leak} variable in the leaky Integrate and Fire model (Equation 2). I is the synaptic or injected current fed into the system. a adjusts the time scale of u and b defines the sensitivity of u. The following equation is used to reset the membrane potential and recovery variable after a spike.

$$\text{if } v \geq 30_{mV}, \text{ then} \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{6}$$

Where c is the reset value for the membrane potential and d is the adjustment value for resetting the recovery variable. By setting the values a, b, c and d it is possible to produce various spiking behaviors like those found in biological neurons (see Figure 1).

These neurons can be networked together by taking note of spikes generated by presynaptic neurons within a time cycle, summing over the synaptic weight factored with the input and

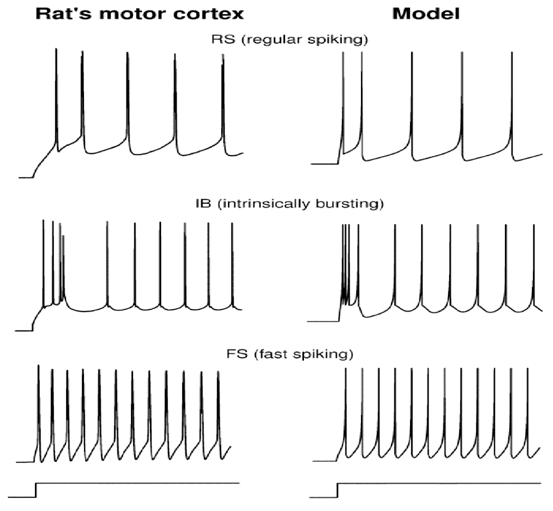

Fig. 1. Each graph shows the voltage of a single neuron over time which alludes to the voltage spiking behaviors that occur when a neuron is excited by external stimulation. On the left are recordings taken from neurons in an actual rat brain and on the right are graphs of comparable spike patterns generated by the Izhikevich model [29].

injecting it into the input of the postsynaptic neuron. This facilitates all the same kind of network topologies available to ANNs.

Unlike perceptron style artificial neural networks, an SNN transfers signals as trains of spikes traveling from one neuron to the next. This spiking behavior breaks the continuity of the transfer function which is used to calculate the error contribution of each neuron. This prevents the use of back-propagation. Some attempts have been made to work around this problem by treating the spike discontinuities as noise [34]. Other learning methods have also been applied to overcome this problem such as direct re-enforcement Learning.

One SNN algorithm uses partially observable markov decision processes(POMPD) to apply direct re-enforcement learning [35]. In effect this approach introduces Hebbian synaptic plasticity to the neuron model. This means when a neuron fires the connection between the presynaptic neurons involved in ‘exciting’ it are strengthened, providing a method of learning without backpropagation. Essentially each input synapse connected to a neuron consists of a weight, spike input and an observing factor z. The weight is factored with a spike input of 0 or 1 giving a value s. z slowly decays at each time step by an amount defined by the term βz_t. The $(s_t - \sigma(v_t))s_{t-\tau}$ term adjusts z according to the spiking behavior and weight of the synaptic connection. This is shown in Equation 7, where Equation 8 shows the a squashing function used on the previous membrane voltage v_t. Other implementations of MPD’s for reinforcement learning have also been conducted recently [36] [37].

$$z_{t+\tau} = \beta z_t + (s_t - \sigma(v_t))s_{t-\tau} \tag{7}$$

$$\sigma(x) = \frac{1}{(1+e^{-x})} \tag{8}$$

A new weight is then calculated as in Equation 9 with z and a reinforcement value r being used to update the synaptic weight. γ functions similarly to a learning rate in classic artificial neural networks.

$$w_{t+\tau} = w_t + \gamma r_{t+\tau} z_{t+\tau} \tag{9}$$

The weight affects the contribution that a received spike has on the membrane voltage of a receiving neuron.

C. Classification

Statistical Classification is a heavily researched area of machine learning which involves organizing new observations into a category based on a set of training observations with known categories. A large body of research is aimed at image classification where an image or some aspect of an image is put into a category such as ‘person’, ‘table’ and ‘road’ etc. Many of these image classifiers are based on Convolutional Neural Networks (CNNs) [38]. CNNs work by taking a matrix of inputs on which a window is systematically moved across the dimensions of the matrix. These windows contain a series of values that when compared with the values in the input matrix trigger activations resulting in an activation map per window. These activation maps are then processed with a rectilinear function so that negative values are removed. From here more convolutional layers (sliding windows) are applied as desired. A pooling layer is often applied at the end of these convolutions which reduces the dimensions of the resultant convolutional matrices. One such technique is down sampling where a window of some size is convolved over an input matrix such that the maximum value of the window is stored in the resultant matrix. After a max pooling layer the resultant matrix is wired to a fully connected layer which links each matrix value to a set of output value representing each of the desired Classification labels. Training this kind of network requires the use of various loss functions to calculate an error from values fed forward through the network from some image with a known set of categories. These loss functions calculate the error between the output values that occurred and the desired values for correct categorization of that image. The error is then passed back through the network to adjust weights so that the network provides a less error-some set of output values in the future. The loss function is dependent upon the type of Classification desired. Binary and multilabel Classification may use a binary cross entropy loss function. In the case of multilabel Classification each class label is treated as a single binary Classification. Multiclass Classification might use a categorical cross entropy loss function. The first examples of this type of network were shown in AlexNet [39] which was a CNN classifier applied to the ImageNet dataset. More modern implementations can be seen in ResNet [40], Google’s inception [41] and the YOLOv2 algorithm [42].

III. NENG: The Proposed System

Neuro-Evolutionary Network Generation (NENG) utilizes a GA to select the best performing topologies for SNNs via the direct use of re-enforcement learning. The SNNs are tasked

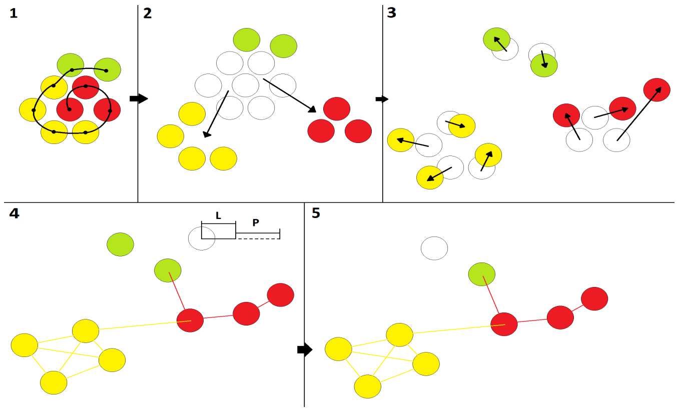

Fig. 2. The generation of the SNN topology as described in Algorithm 2. Here 3 sets of neuron genes are used to create 3 distinct types of neurons, highlighted as red, green and yellow. **(1)** The neurons are generated in a spiral pattern. **(2)** The neurons migrate. **(3)** The neurons scatter randomly. **(4)** Connections are formed within a minimum limit (L) with a probability of forming longer connections (P) with a gaussian distribution. **(5)** Neurons which are not connected are pruned.

with classifying data representing an Xor Gate. The Xor pattern was chosen because it has simple known solutions which can be compared with solutions generated by the algorithm thus allowing for identification of redundant complexity in the generated topologies and minimizes execution time for testing. The ideal test results would be fast algorithmic execution time with minimal extraneous topological structures. As such we would expect to see something akin to a 2 layer feed forward neural network In the ideal case.

A. GA

The intuition behind the GA is based on the idea that neurons have cell fates which determine their final position and the kind of neuron they will become. We emulate simplified versions of neuronal migration, synaptogenesis and pruning that occur within real organisms. A gene encoding for a chromosome is carefully crafted to guide the generation of the SNN topology in combination with Algorithm 2.

Each individual consists of two chromosomes, one from each parent. The gene sequence of a chromosome is represented by a fixed length string with specialized delimiters to separate the functional purposes of the values (see Figure 3).

This chromosome has ten sets of values which can be encoded differently. Each set is encompassed in square brackets '[]'. This allows for neurons with different behaviors to be produced. At the bottom of the chromosome is a term relating to migration time.

The neurons from each neuronal set are generated in a spherical pattern, with each neuron occupying a distinct space in spherical co-ordinate space (see Figure 2 box 1 and the Neurogenisis Stage of Algorithm 2). The number of generated neurons for the each set is colored in cyan. After this a genetically determined direction (red) and speed (blue) is used to regulate the migration of the neurons (see Migration Stage of Algorithm 2 and box 2 of Figure 2). The direction vector is encoded with integers 0, 1 and 2 representing negative, stationary and positive directions respectively. For the first neuronal

[0,2,1,00003,00023,00002,1,1,1,00014,00002]
[2,2,1,00003,00002,00004,1,0,0,00019,00006]
[2,2,1,00001,00000,00004,0,0,0,00003,00015]
[0,2,0,00004,00001,00000,0,1,1,00009,00011]
[2,0,1,00000,00003,00001,0,1,1,00017,00015]
[0,2,1,00003,00002,00004,0,0,0,00016,00012]
[1,0,1,00002,00000,00004,0,1,1,00005,00008]
[2,0,1,00001,00002,00000,1,0,1,00013,00012]
[0,0,2,00003,00004,00002,0,0,0,00010,00016]
[0,0,0,00000,00000,00000,0,0,0,00000,00000]
;0000000004;

Fig. 3. A colour coded breakdown of the gene encoding used by the GA. A unique set of values for neuron generation is contained between each set of square brackets. The final value between the semicolons is a migration time factor affecting the generation of all the neurons.

set the direction vector is x=-1, y=1 and z=0. These values are combined with the migration speed (blue). Each neuron in the first set will migrate by -3, 3, 0 in the x, y, z directions respectively. This occurs for the length of the migration time delimited by the semicolons ';'. For this chromosome it is 4. This means the neurons in the first set will migrate a total of -12,12,0 in the x, y and z directions. Once the neurons have migrated they will disperse in the x, y, z directions by the amount colored in black factored with a random normally distributed gaussian value (see Figure 2 box 3 and line 20 in Algorithm 2). Afterwards synapses are formed between the neurons if they are within the minimum distance (highlighted in grey) factored with a random normalized gaussian value (see Figure 2 box 4 and the Synaptogenisis Stage in Algorithm 2). Any neurons with no connections at the end of this process are deleted. The gene encoding also contains Boolean binary values for determining if a neuron generates its own signal (green), whether it is excitory or inhibitory (magenta) and if it forms a loopback connection with itself (purple). Finally, the neurons base type is defined by an index value (dark orange). The neurons base type defines the kind of spike pattern it emits when firing. These base types are defined by predetermined constants for a, b, c, d and I that are shown to produce specific kinds of spiking behavior as shown by Izhikevich [29]

The full algorithm executes as described in Figure 4. The initial population is generated with random values which are constrained within predefined boundaries to reduce the search space within sensible ranges (see Table I). These can be adjusted to handle larger problems. Selection is performed by choosing the top 50% of the fittest individuals. The next generation is created from a random combination of the previously selected individuals. The parents individual chromosomes undergo a digital implementation of meiosis using their own two parent chromosomes. The chromosomes are copied so that there is a duplicate of each chromosome. Two of the four chromosomes are selected at random and then crossover is performed by selecting a random point along the genetic string (the loci) and switching them. The gene sequences are then stored and used as the new input alongside the unchanged chromosomes. The process is repeated for a

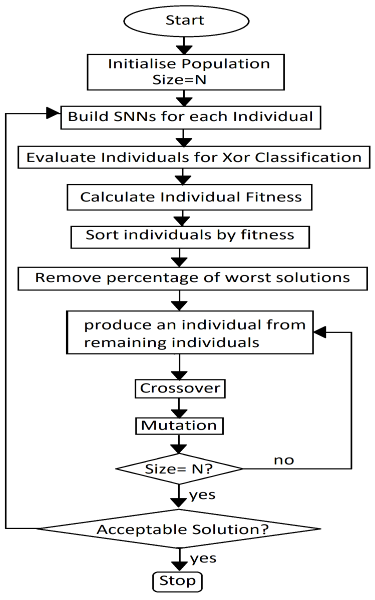
Start
Initialise Population Size=N
Build SNNs for each Individual
Evaluate Individuals for Xor Classification
Calculate Individual Fitness
Sort individuals by fitness
Remove percentage of worst solutions
produce an individual from remaining individuals
Crossover
Mutation
Size= N?
no
yes
Acceptable Solution?
yes
Stop

Fig. 4. The process of the GA

TABLE I
THE BOUNDARIES FOR THE RANDOM GENERATION OF CERTAIN VARIABLES

Value	Min	Max
Migration Amount	0	4
Dispersal Amount	0	4
Synapse Forming Limit	0	4
Neuron Number	0	19

random number of crossovers between 1 and 20.

Mutation is set as $\frac{1}{currentBest - previousBest}$ thus mutation becomes larger when fitness stagnates. Mutation is handled by generating a random real number between 0.0 and 1.0 and if the number is less than the mutation rate then a genome is mutated. This process is repeated until a number higher than the mutation rate is generated. The max mutation rate is set to 0.99999 to prevent infinite and unreasonably long mutation times. Each genetic variable has a equal likelihood of being mutated. Mutated variables produce a new random number between their boundary conditions with the exceptions of migration speed, migration dispersal, synapse forming limit and the number of neurons to create. These exceptions will either increment of decrement upon mutation.

The SNNs generated by the above GA model were used to perform a classification task. The fitness/objective function is calculated by summing the outputs of the classifier as in Equation 10 with an additional term for fitness based on execution time t. A is Accuracy, P_M is Macro Precision, P_μ is the micro precision, R_M is Macro Recall and R_μ is the micro recall. These measures are calculated as defined in a paper by Marina Sokolova [43].

$$\text{Fitness} = A + P_M + P_\mu + R_M + R_\mu + \frac{1}{t} \tag{10}$$

B. SNN

An SNN algorithm was devised by combining the Izhikevich neuron model (eqns 4, 5 and 6) with the Hebbian re-enforcement learning proposed by Bartlett (Equations 7, 8 and 9). The result is shown in Algorithm 1. The Izhikevich neuron model was used for its ability to model many aspects of the spiking behaviors provided by the Hodgkin Huxley model without the associated computational overhead. Learning is implemented with a POMDP with each non-input neuron behaving as an independent agent in a re-enforcement learning problem [35].

Algorithm 1 SNN execution algorithm

```
Require:
    set decay rate $\beta \in [0, 1)$.
    set learning rate rate $\gamma$.
    initialise synaptic weights $w^i_{j,t=0}$.
    initialise spike inputs $s^{i,j}_{t=0}$.
    set time step $\tau$
 1: for $t = \tau, 2\tau, 3\tau...$ do
 2:     set $s^j_t$ for input neurons.
 3:     for Non-input neurons $i$ do
 4:         $v_{sum} = \sum_j w^i_{j,t} s^{i,j}_t$.
 5:         $v^i_{t+\tau} = v_{sum} + \tau + 0.04 * {v_{sum}}^2 + 5v_{sum} + 140 - u^i_t + I$.
 6:         $u^i_{t+\tau} = u^i_t + \tau a^i (b^i v^i_{t+\tau} - u^i_t)$.
 7:     for Non-input neurons $i$ do
 8:         if $v^i_{t+\tau} > 30$ then
 9:             $s^i_{t+\tau} = 1$.
10:             $v^i_{t+\tau} = c^i$.
11:             $u^i_{t+\tau} = u^i_{t+\tau} + d^i$.
12:         else
13:             $s^i_{t+\tau} = 0$.
14:     if learning then
15:         set reward $r_{t+\tau}$ from network outputs.
16:         for Non-input neurons $i$ do
17:             $z^i_{j,t+\tau} = \beta z^i_{j,t} + (s^i_t - \sigma(v^i_t)) s^j_{t-\tau}$.
18:             $w^i_{j,t+\tau} = w^i_{j,t} + \gamma r_{t+\tau} Z^i_{j,t+\tau}$.
```

Algorithm 1 shows the SNN algorithm. At each time step $t + \tau$ the membrane potential $v^i_{t+\tau}$ and reset values $u^i_{t+\tau}$ are updated for each neuron i shown in lines 5 and 6 of Algorithm 1. $s^{i,j}_t$ is the previous spike input from each presynaptic neuron j, I is the injected current ($I = 0$ when no current is injected). When $v^i_{t+\tau}$ and $u^i_{t+\tau}$ have been updated each neuron is checked to see if it reaches a 30_{mv} threshold which causes a spike to be propagated to subsequent neurons; $s^i_{t+\tau} = 1$. If a spike occurs, $v^i_{t+\tau}$ and $u^i_{t+\tau}$ are adjusted as in line 10 and 11 in Algorithm 1.

When the SNN is learning it takes a reinforcement signal such that $r \in \{-1, 0, 1\}$. 0 ensures no change in the weights, 1 ensures the weights increase by the corresponding z value and -1 decreases it. γ is a learning parameter similar to the learning rate in classic ANNs. $z^i_{j,t+\tau}$ in line 17 of Algorithm 1 is an adaptation of Equation 7 that indicates the update for

Algorithm 2 SNN generation algorithm

```
Require:
    set number of snn inputs $I = 2$ and ouputs $O = 1$.
    initialise braincell array $bca$.
1:  for gene g in neuronGenes Ng do                    ▷ Neurogenisis Stage
2:      gID = 0;
3:      for int $i = 0; i <$ g.NumberToCreate; $i$++ do
4:          $bca$.Add (new bc( gID,
                g.BaseNeuronType, bool(g.IsExcitor),
                bool(g.IsSignalGenerator), bool(g.IsExcitor),
                g.BaseNeuronType, bool(g.IsLoopback)));
            gID++;
5:      Shuffle ($bca$);
6:      $\rho = 1,\ \theta = 2,\ \phi = 2$
7:      for bc in $bca$ do
8:          $bc.setCoords(\rho\sin(\theta\pi)\cos(\phi\pi),$ ▷$x$
                $\rho\sin(\theta\pi)\sin(\phi\pi),$ ▷$y$
                $\rho\cos(\theta\pi));$ ▷$z$
9:          if $\theta >= 4$ then
10:             $\theta = 2,\ \phi = \phi + \frac{0.5}{\rho}$
11:         if $\phi >= 4$ then
12:             $\phi = 2,\ \rho = \rho + 1$
13:         $\theta = \theta + \frac{0.5}{\rho};$
14: $bca$.Sort();
15: for bc in $bca$ do                                 ▷ Migration Stage
16:     for gene g in neuronGenes Ng do
17:         if bc.BaseNeuronType == g.BaseNeuronType then
18:             for int $i = 0; i <$ Ng.MigrationTime; $i$++ do
19:                 bc.setCoords (
                        $bc.x +$ (g.MigrateLeft $- 1)\times$
                            g.MigrateDist,
                        $bc.y +$ (g.MigrateUp $- 1)\times$
                            g.MigrateDist,
                        $bc.z +$ (g.MigrateForward $- 1)\times$
                            g.MigrateDist);
20:             bc.setCoords (
                    $bc.x +$ g.DisperseDist $\times$ RandGauss(),
                    $bc.y +$ g.DisperseDist $\times$ RandGauss(),
                    $bc.z +$ g.DisperseDist $\times$ RandGauss());
21:             break;
22: for bc in $bca$ do                                 ▷ Synaptogenisis Stage
23:     for gene g in neuronGenes Ng do
24:         if bc.BaseNeuronType == g.BaseNeuronType then
25:             intialise connections array $ca$
26:             for bc2 in $bca$ do
27:                 $\Delta x = (bc.x - bc2.x)^2$
28:                 $\Delta y = (bc.y - bc2.y)^2$
29:                 $\Delta z = (bc.z - bc2.z)^2$
30:                 $d = \sqrt{\Delta x + \Delta y + \Delta z}$
31:                 if $d$ <= g.SynapseLimit $\times$ RandGauss() then
32:                     if bc.Id != bc2.Id then
33:                         $ca$.Add ($bc2.Id$);
34:                     else if bc.IsLoopback then
35:                         $ca$.Add ($bc2.Id$);
36:                 break;
```

each synaptic weight for each neuron. Once the z values have been updated the actual weights of the synapse connections can be updated as in line 18 of Algorithm 1.

C. Classification

In order to classify the output of the SNN a few deviations from the normal approach should be observed. After an SNN topology is generated the algorithm automatically selects the left-most neurons as the input neurons and the right-most neurons as the outputs. The algorithm will select as many inputs and outputs as needed to classify the given data. Using a simple Xor data set (see Table II) we can see we need 2 input neurons and 1 output, thus the algorithm will select the 2 left-most neurons as inputs and 1 right-most neuron as an output (see Figure 5).

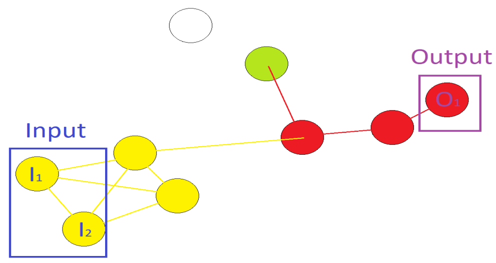

Fig. 5. The input and output neuron selection when using the Xor dataset in table II applied to the SNN generation example shown in Figure 2

1) Training: Input values are presented to the network for a number of time cycles while the SNN attempts to learn the pattern. At each time cycle (τ) the output neuron values are compared to the ground truth values in the data set. If the ground truth is $>$ the output neurons value then $r = 1$. Since r is shared between all neurons, this causes neurons which receive input to increase their synaptic strength. After a number of cycles the weights for the synaptic connections will increase causing subsequent neurons to fire (see Figure 6). This makes the signal cascade through the network eventually causing the output neuron to fire. When the output neuron fires and the ground truth value is 1 then they will be equal, at this point $r = 0$ preventing any further adjustments to the weights. In the opposite case where the ground truth value is 0 and the output is 1 then $r = -1$ causing a decrease in weights until the output neuron no longer responds to the input signal. After one signal has been presented to the network several times the proceeding signals are presented until an entire epoch of the data has been shown. This can be repeated for as many times as desired. This setup works using a single neuron for binary Classification but requires adaptation to work with multi-label and multi-class setups.

TABLE II
A SIMPLE XOR DATA SET

Input	Output
0,0	0
1,1	0
0,1	1
1,0	1

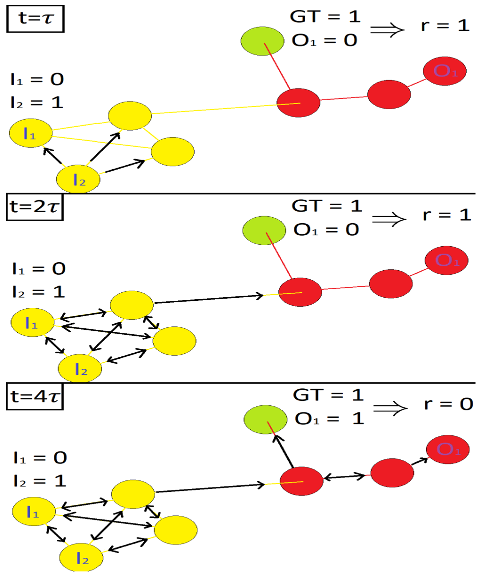

Fig. 6. The signal propagation through a few time cycles, GT = ground truth

2) Testing: To check the network had learned the Xor training set learning was disabled and the network was presented with each signal once with the network being run for a predetermined number of cycles. If the output neuron produced a spike it was recorded and checked against the ground truth. If a spike was not received, then the algorithm would cycle for the predetermined time and then be checked against the ground truth. This allowed for Classification accuracy to be tested.

3) Confusion Matrix: After executing all the training and testing the results were compiled into a confusion matrix from which various measures could be calculated. The confusion matrix has an unusual adaptation to accommodate for multi-class/label setups. It is possible for multiple spikes to occur within a single test. To accommodate for this an extra column is added to capture all unknown spiking counts. For instance, in a multi-class setup with 3 classes using one neuron with each class corresponding to a spike value of 0, 1 or 2, it is possible that 4 spikes would be recorded. This would be classified into an unknown class. This means the confusion matrix always has an additional column.

4) Classification measures: The following equations were used to implement the Classification measures used for the fitness function.

$$\text{Average Accuracy} = \frac{\sum_{i=1}^{l} \frac{\text{tp}_i+\text{tn}_i}{\text{tp}_i+\text{fn}_i+\text{fp}_i+\text{tn}_i}}{l} \tag{11}$$

$$\text{Precision}_M = \frac{\sum_{i=1}^{l} \frac{\text{tp}_i}{\text{tp}_i+\text{fp}_i}}{l} \tag{12}$$

$$\text{Precision}_\mu = \frac{\sum_{i=1}^{l} \text{tp}_i}{\sum_{i=1}^{l} (\text{tp}_i + \text{fp}_i)} \tag{13}$$

$$\text{Recall}_M = \frac{\sum_{i=1}^{l} \frac{\text{tp}_i}{\text{tp}_i+\text{fn}_i}}{l} \tag{14}$$

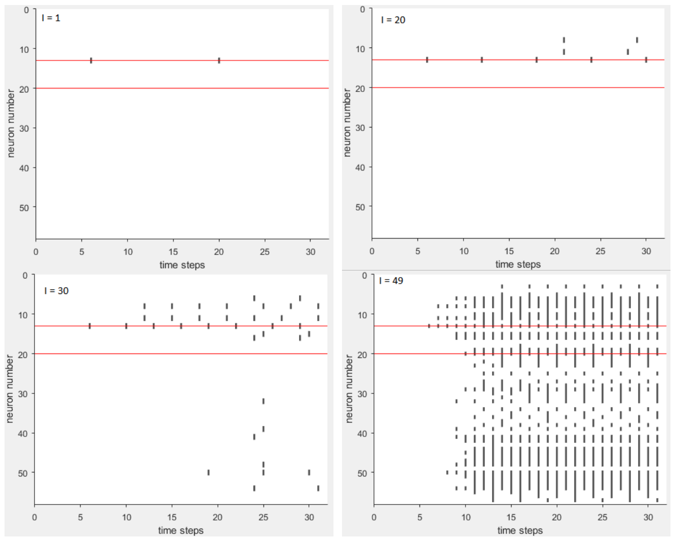

Fig. 7. Spike raster plot of all the neurons for the best performing SNN at 4 different iterations; 1, 20, 30 and 49. Here the SNN is learning the {1,0} data point. The red lines indicate the input neurons. Each image shows the full 31 time cycles.

$$\text{Recall}_\mu = \frac{\sum_{i=1}^{l} \text{tp}_i}{\sum_{i=1}^{l} (\text{tp}_i + \text{fn}_i)} \tag{15}$$

IV. Evaluation

The Algorithm was run on 64 bit windows 10 Professional with 32GB of ram, an i7-5930K CPU @ 3.50GHz using Unity 3D. The population was initialized at 20 with $\beta = 0.5$ and $\gamma = 0.1$. The SNN cycle time step was set as $\tau = 0.2$ with each SNN executing 31 complete time cycles for training and testing. During training the pattern was shown to the network 50 times for each data point and was run for 1 epoch. Figure 7 displays the best solution as it learns the $1, 0$ data point. Table III shows the significant solutions generated by NENG. Total runtime was 2:13:46 (h:mm:ss).

In III The first major improvement from solution 19-25 is better recall and precision. Solution 611 shows progress in accuracy along with precision and recall, showing the correct classification of 75% of the data points. From solution 611 to 4689 improvements are made such as reducing the number of neurons and optimizing topology and types of neurons. Some of the later finesses are actually worse than the preceding ones. This is likely due to background processes running on the computer slowing down the evaluation, since time is a factor for the fitness evaluation this seems like a reasonable cause. In general, the execution time and neuron numbers were reduced until solution 8133. This solution showed complete correct Classification of the data set with accuracy 1, $P_\mu = 1$ and $R_\mu = 1$. Note the macro versions of precision and accuracy are limited to 0.667 due to the extra unknown class to catch any odd spiking behavior exhibited by the SNNs such ass producing multiple spikes. During the testing there were no noted additional spikes and this extra class could be removed. The

TABLE III
THE SIGNIFICANT BEST SOLUTIONS GENERATED BY THE PROPOSED ALGORITHM

ID	Original Generation	Accuracy	P_M	R_M	P_μ	R_μ	Run Time	Neurons	Fitness
19	0	0.667	0.166	0.333	0.5	0.5	239ms	4	0.3618085
25	1	0.667	0.333	0.333	0.5	0.5	893ms	16	0.3890755
611	60	0.833	0.555	0.5	0.75	0.75	7191ms	71	0.564838
753	74	0.833	0.555	0.5	0.75	0.75	5708ms	68	0.564844
1027	101	0.833	0.555	0.5	0.75	0.75	2481ms	56	0.564882
1107	109	0.833	0.555	0.5	0.75	0.75	2329ms	56	0.564886
1201	119	0.833	0.555	0.5	0.75	0.75	2598ms	57	0.564879
1224	121	0.833	0.555	0.5	0.75	0.75	2247ms	57	0.564889
1393	138	0.833	0.555	0.5	0.75	0.75	2322ms	61	0.564887
1595	158	0.833	0.555	0.5	0.75	0.75	2021ms	61	0.564897
1638	163	0.833	0.555	0.5	0.75	0.75	1575ms	49	0.564921
1690	168	0.833	0.555	0.5	0.75	0.75	1716ms	49	0.564912
1851	184	0.833	0.555	0.5	0.75	0.75	1274ms	49	0.564946
1961	195	0.833	0.555	0.5	0.75	0.75	1273ms	47	0.564928
2090	208	0.833	0.555	0.5	0.75	0.75	1260ms	46	0.564947
4564	445	0.833	0.555	0.5	0.75	0.75	1019ms	23	0.564978
4683	467	0.833	0.555	0.5	0.75	0.75	947ms	23	0.564991
8133	812	1	0.667	0.667	1	1	4218ms	57	0.722261
8866	885	1	0.667	0.667	1	1	2594ms	57	0.722286

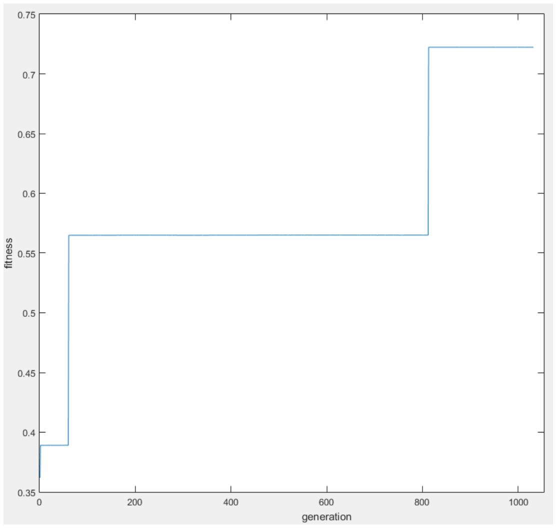

Fig. 8. The fitness of the best solutions per generation, some variation in the fitness values is too small to see, this variation is caused by small differences in the evaluation times of the SNN's

last solution (8866) shows an improvement which increases the execution time by around 1000ms. Such a significant jump is likely due to topological improvements rather than variances in available compute power being restricted by background processes. It is important to mention that the execution times are notably increased by a significant amount because the output of each iteration and time cycle of every neuron is recorded during the execution and then written to a file The full selection of best solutions over time can be seen in Figure 8.

Figure 9 shows an example of the test phase for solution 8866. We can see that the input neurons stimulate a pattern that is similar for all the data points except data point $\{0, 0\}$ where no input is provided and a Classification of 0 was the goal. Data point $\{1, 0\}$ has a spike at $\tau = 30$ correctly classifying the input data with a 1. The same is true of data point $\{0, 1\}$

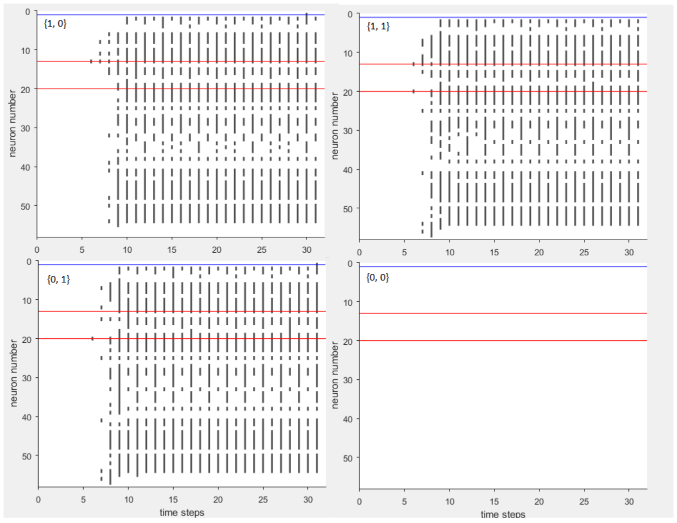

Fig. 9. The raster plots for each time cycle when testing a trained SNN on the 4 Xor data points. The blue line is the output neuron, the red lines are the input neurons. Note that for points $\{0, 1\}$ and $\{1, 0\}$ an ouput spike is generated, for $\{0, 0\}$ no network activity is generated and for $\{1, 1\}$ network activity is similar to the patterns for $\{1, 0\}$ and $\{0, 1\}$ but without producing an output spike.

except that the spike occurs in time cycle 31.

V. CONCLUSION

This paper proposes a novel neuro-evolutionary algorithm for generating valid solutions for classifying the Xor gate. It is able to automatically select an appropriate SNN to classify non-linear functions. The algorithm uses a GA model with indirect gene encoding to generate SNN topology using a POMDP to enable re-enforcement learning to train the classifier. Validation was performed using accuracy, macro precision, micro precision, macro recall, micro recall and execution time. The algorithm was shown to find a successful solution and provide further improvement as time progresses.

The algorithm requires further work to improve evaluation times and enable Classification on larger datasets. Currently the total number of time cycles for SNN training and testing is predetermined. This could be automated by implementing something akin to Dijkstra's algorithm to determine the number of required cycles as a function of the shortest path through the network.

References

[1] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[2] S. C. Neoh, W. Srisukkham, L. Zhang, S. Todryk, B. Greystoke, C. P. Lim, M. A. Hossain, and N. Aslam, "An intelligent decision support system for leukaemia diagnosis using microscopic blood images," *Scientific reports*, vol. 5, p. 14938, 2015.

[3] D. V. Vargas and J. Murata, "Spectrum-diverse neuroevolution with unified neural models," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 8, pp. 1759–1773, 2017.

[4] D. Cheong, Y. M. Kim, H. W. Byun, K. J. Oh, and T. Y. Kim, "Using genetic algorithm to support clustering-based portfolio optimization by investor information," *Applied Soft Computing*, vol. 61, pp. 593–602, 2017.

[5] S. Ghambari and A. Rahati, "An improved artificial bee colony algorithm and its application to reliability optimization problems," *Applied Soft Computing*, 2017.

[6] J. Luo, Q. Liu, Y. Yang, X. Li, M.-r. Chen, and W. Cao, "An artificial bee colony algorithm for multi-objective optimisation," *Applied Soft Computing*, vol. 50, pp. 235–251, 2017.

[7] P. Rakshit, A. Konar, and A. K. Nagar, "Learning automata induced artificial bee colony for noisy optimization," in *Evolutionary Computation (CEC), 2017 IEEE Congress on*. IEEE, 2017, pp. 984–991.

[8] A. Saad, S. A. Khan, and A. Mahmood, "A multi-objective evolutionary artificial bee colony algorithm for optimizing network topology design," *Swarm and Evolutionary Computation*, 2017.

[9] K. Mistry, L. Zhang, S. C. Neoh, C. P. Lim, and B. Fielding, "A micro-ga embedded pso feature selection approach to intelligent facial emotion recognition," *IEEE transactions on cybernetics*, vol. 47, no. 6, pp. 1496–1509, 2017.

[10] W. Srisukkham, L. Zhang, S. C. Neoh, S. Todryk, and C. P. Lim, "Intelligent leukaemia diagnosis with bare-bones pso based feature optimization," *Applied Soft Computing*, vol. 56, pp. 405–419, 2017.

[11] L. Zhang, K. Mistry, S. C. Neoh, and C. P. Lim, "Intelligent facial emotion recognition using moth-firefly optimization," *Knowledge-Based Systems*, vol. 111, pp. 248–267, 2016.

[12] P. Liu and J. Liu, "Multi-leader pso (mlpso): A new pso variant for solving global optimization problems," *Applied Soft Computing*, vol. 61, pp. 256–263, 2017.

[13] N. Lynn, M. Z. Ali, and P. N. Suganthan, "Population topologies for particle swarm optimization and differential evolution," *Swarm and Evolutionary Computation*, 2017.

[14] W. Ye, W. Feng, and S. Fan, "A novel multi-swarm particle swarm optimization with dynamic learning strategy," *Applied Soft Computing*, vol. 61, pp. 832–843, 2017.

[15] F. Javidrad and M. Nazari, "A new hybrid particle swarm and simulated annealing stochastic optimization method," *Applied Soft Computing*, vol. 60, pp. 634–654, 2017.

[16] F. Y. Vincent, A. P. Redi, Y. A. Hidayat, and O. J. Wibowo, "A simulated annealing heuristic for the hybrid vehicle routing problem," *Applied Soft Computing*, vol. 53, pp. 119–132, 2017.

[17] J. Lin, Y. Zhong, E. Li, X. Lin, and H. Zhang, "Multi-agent simulated annealing algorithm with parallel adaptive multiple sampling for protein structure prediction in ab off-lattice model," *Applied Soft Computing*, 2017.

[18] K. Geng and V. Z. Marmarelis, "Methodology of recurrent laguerre-volterra network for modeling nonlinear dynamic systems," *IEEE transactions on neural networks and learning systems*, 2017.

[19] V. Maniezzo, "Genetic evolution of the topology and weight distribution of neural networks," *IEEE Transactions on neural networks*, vol. 5, no. 1, pp. 39–53, 1994.

[20] G. Howard, L. Bull, B. de Lacy Costello, E. Gale, and A. Adamatzky, "Evolving spiking networks with variable resistive memories," *Evolutionary computation*, vol. 22, no. 1, pp. 79–103, 2014.

[21] D. Lobiyal, S. Prasad *et al.*, "An elitist nondominated sorting genetic algorithm for qos multicast routing in wireless networks," *Swarm and Evolutionary Computation*, vol. 33, pp. 85–92, 2017.

[22] N. Brunel and M. C. Van Rossum, "Lapicques 1907 paper: from frogs to integrate-and-fire," *Biological cybernetics*, vol. 97, no. 5-6, pp. 337–339, 2007.

[23] A. Hill, "Excitation and accommodation in nerve," *Proceedings of the Royal Society of London. Series B, Biological Sciences*, vol. 119, no. 814, pp. 305–355, 1936.

[24] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological cybernetics*, vol. 95, no. 1, pp. 1–19, 2006.

[25] A. Hodgkin and A. Huxley, "The components of membrane conductance in the giant axon of loligo," *The Journal of physiology*, vol. 116, no. 4, pp. 473–496, 1952.

[26] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[27] E. M. Izhikevich, *Dynamical systems in neuroscience*. MIT press, 2007.

[28] G. Chatzikonstantis, D. Rodopoulos, C. Strydis, C. I. De Zeeuw, and D. Soudris, "Optimizing extended hodgkin-huxley neuron model simulations for a xeon/xeon phi node," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[29] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[30] K. Minkovich, C. M. Thibeault, M. J. O'Brien, A. Nogin, Y. Cho, and N. Srinivasa, "Hrlsim: a high performance spiking neural network simulator for gpgpu clusters," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 2, pp. 316–331, 2014.

[31] F. Naveros, N. R. Luque, J. A. Garrido, R. R. Carrillo, M. Anguita, and E. Ros, "A spiking neural simulator integrating event-driven and time-driven computation schemes using parallel cpu-gpu co-processing: a case study," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 7, pp. 1567–1574, 2015.

[32] U. B. Rongala, A. Mazzoni, and C. M. Oddo, "Neuromorphic artificial touch for categorization of naturalistic textures," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 4, pp. 819–829, 2017.

[33] C. Liu, J. Wang, H. Li, C. Fietkiewicz, and K. A. Loparo, "Modeling and analysis of beta oscillations in the basal ganglia," *IEEE Transactions on Neural Networks and Learning Systems*, 2017.

[34] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in neuroscience*, vol. 10, 2016.

[35] P. L. Bartlett and J. Baxter, "Hebbian synaptic modifications in spiking neurons that learn," *Research School of Information Sciences and Engineering*, 1999.

[36] M. J. Islam, M. M. Islam, and A. A. Al Islam, "Intelligent dynamic spectrum access using hybrid genetic operators," *Swarm and Evolutionary Computation*, 2017.

[37] X. Xu, Z. Huang, L. Zuo, and H. He, "Manifold-based reinforcement learning via locally linear reconstruction," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 4, pp. 934–947, 2017.

[38] P. Kinghorn, L. Zhang, and L. Shao, "A region-based image caption generator with refined descriptions," *Neurocomputing*, vol. 272, pp. 416–424, 2018.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[42] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[43] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.