Neuroevolution of Spiking Neural P Systems

Leonardo Lucio Custode[0000−0002−1652−1690], Hyunho Mo[0000−0002−6497−2250], and Giovanni Iacca[0000−0001−9723−1830]

Department of Information Engineering and Computer Science, University of Trento, Italy

Abstract. Membrane computing is a discipline that aims to perform computation by mimicking nature at the cellular level. Spiking Neural P (in short, SN P) systems are a subset of membrane computing methodologies that combine spiking neurons with membrane computing techniques, where "P" means that the system is intrinsically parallel. While these methodologies are very powerful, being able to simulate a Turing machine with only few neurons, their design is time-consuming and it can only be handled by experts in the field, that have an in-depth knowledge of such systems. In this work, we use the Neuroevolution of Augmenting Topologies (NEAT) algorithm, usually employed to evolve multi-layer perceptrons and recurrent neural networks, to evolve SN P systems. Unlike existing approaches for the automatic design of SN P systems, NEAT provides high flexibility in the type of SN P systems, removing the need to specify a great part of the system. To test the proposed method, we evolve Spiking Neural P systems as policies for two classic control tasks from OpenAI Gym. The experimental results show that our method is able to generate efficient (yet extremely simple) Spiking Neural P systems that can solve the two tasks. A further analysis shows that the evolved systems act on the environment by performing a kind of "if-then-else" reasoning.

Keywords: Neuroevolution · NEAT · Membrane Computing · Spiking P Systems · OpenAI Gym

1 Introduction

Membrane computing is a branch of natural computing initiated by Păun in 1998 [1]. The goal of membrane computing is to perform computations by emulating nature at the *cellular* level. In the area of membrane computing, membrane systems (also called P systems) indicate models that have parallel and distributed computation capability. Spiking Neural P (in short, SN P) systems [2, 3] incorporate the idea of spiking neurons (and spike trains) into P systems. SN P systems, differently from other combinations of neural models and P systems [4], use *time* as a source of information in the computation, similarly to what happens in biological brains. Moreover, it has been proved that a SN P system can simulate a Turing machine, given a sufficient number of neurons [5–7].

While SN P systems have proven to be applicable to a wide variety of problems, their design is (almost always) currently done manually by an expert. Very

few attempts have been made at automating this step [8, 9]. The lack of automatic design methodologies, of course, represents a *bottleneck* in the development of the field, as a lot of work (and time) is needed to design such systems.

Here, we employ a well-known neuroevolutionary algorithm, namely the Neuroevolution of Augmenting Topologies (NEAT) [10], to automatically design SN P systems for a given task. More specifically, we modify the original NEAT algorithm to handle the parameters of a specific type of SN P systems by increasing the number of parameters contained in the genotype and adapting them to the parameters of this type of neurons.

To the best of our knowledge, our work represents the first attempt to use neuroevolution to *fully* design SN P systems. In fact, while other approaches for the automatic design of SN P systems do exist, they limit the parameters that can be optimized, e.g., by fixing the topology [8] or, by fixing the rules [9]. Our approach, instead, allows to optimize all the SN P system's parameters (except for the number of rules) simultaneously. In this sense, it reduces the need of experts for designing such systems, which in turn may foster a broader applicability of the SN P systems.

In a nutshell, the goal of this paper is to address two main research questions:

- Is it possible to evolve SN P systems by using (a modified version of) the NEAT algorithm?
- How do SN P systems evolved with NEAT compare to other methodologies for classic control tasks?

Our results on two classic control tasks, namely MountainCar-v0 and CartPole-v1 from OpenAI Gym [11], show that our approach is able to produce SN P systems of good quality that are competitive with the state of the art.

The rest of the paper is structured as follows. The next section introduces the background concepts on SN P systems. Section 2 summarizes the related work, followed by Section 4, which describes the proposed method to evolve SN P systems. Section 5 presents the numerical results and their analysis. Finally, Section 6 draws the conclusions of this work and suggests future works.

2 Background

In the neurophysiological behavior of biological neurons, a neuron transmits an electric pulse, a spike, via its synapses. In particular, the spiking neurons considered in our study carry information by means of the number and the timing of the spikes rather than the size and the shape of each spike, assuming that all the spikes of a spiking neuron are identical.

A P system is a computing device based on the progression of objects in a membrane structure initialized with a specific number of objects in each membrane. The system then operates using the rules present in the membrane until its computation is finished. After finishing the computation, the result is provided as the number of objects in each membrane.

A neural-like P system is a P system that has its compartments structured as in a neural net. The behavior of a neural-like P system is based on the state of its neurons and their interactions. Finally, SN P systems are a class of neural-like P systems that apply the idea of spiking neurons.

In the following, we describe the formal definition of a SN P system with the same notions of regular languages used in [2]. A standard SN P system of degree $m \geq 1$, is formally defined as follows:

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, I_{in}, I_{out}) \tag{1}$$

where:

- $O = a$ is a singleton alphabet, where a represents a spike;
- $\sigma_1, \ldots, \sigma_m$ are neurons, each one defined as: $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, being n_i the number of spikes initially present in σ_i and R_i a finite set of rules in σ_i, respectively;
- syn is the set of synapses, where each synapse is defined in $\{1, \ldots, m\} \times \{1, \ldots, m\}$;
- I_{in} and I_{out} indicate, respectively, the sets of input and output neurons, with each of them being a mutually exclusive subset of $\{\sigma_1, \ldots, \sigma_m\}$.

At each timestep, the state of the system is updated based on the number of spikes and the set of rules in each neuron.

The first type of rule $(E/a^c \rightarrow a^p; d)$ is called *spiking rule* (or *firing rule*). In the formula, E is a regular language over O; c and $p < c$ denote the number of spikes consumed and the number of spikes generated, respectively, when the spiking rule is applicable; d is the "refractory" period that forces the neuron to wait d timesteps between two consecutive spikes. Denoting with g the number of spikes contained in a neuron σ_i, the spiking rule above can be interpreted as follows: the rule is applicable only if the number of spikes g is greater than or equal to the number of spikes to be consumed c (i.e., $g \geq c$). At a certain timestep, if the number of spikes g contained in σ_i is above the threshold c, then σ_i consumes c spikes to fire the neuron and $g - c$ spikes remain in the neuron. After immediately emitting p spikes, the neuron cannot fire for the following d timesteps.

The second type of rule $(E/a^f \rightarrow \lambda)$ is referred to as *forgetting rule*. In the formula, f denotes the exact number of spikes needed to apply the forgetting rule, and λ represents an empty string. At a certain timestep, if a neuron σ_i contains *exactly* f spikes and the spiking rule is not applicable, then the neuron consumes f spikes without producing any spikes.

Concerning the set of synapses, syn, its elements have the form of $(j, i, w_{j,i})$ where $1 \leq j, i \leq m$, $j \neq i$ denote the neuron indexes and the weight on synapse (j, i), denoted by $w_{j,i}$, is an integer. Thus, syn describes the topology of the connections among neurons, and their weights. Figure 1 shows the graphical representation of a SN P system used in the rest of the paper. Each node in the graph represents a neuron σ_i. Each neuron is either an input or an output neuron: $I_{in} = \{\sigma_1, \sigma_2\}$ and $I_{out} = \{\sigma_3\}$. The text in the input nodes, t_{v1} or

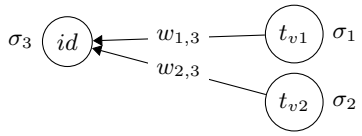


Fig. 1: Graphical representation of an example SN P system.

t_{v2}, indicates the number of spikes for the corresponding input variables obtained from the environment of the given task. Each output neuron, which has a given *id*, produces a response to the environment by emitting spikes. Lastly, the edges between neurons show the synapses *syn*, and the weight of each synapse is specified on each edge.

One additional note is that a neuron can use only one rule at each timestep. If there are more than two applicable rules for a neuron σ_i at any timestep, one of them is chosen non-deterministically with the same probability.

3 Related work

Spiking Neural P systems One of the earliest applications of SN P systems was to use the spike trains as language generators: in [3], a binary language generating device was introduced, based on the distances between spikes. Then, several extensions of this idea were investigated in [12], to get a language over an alphabet with as many symbols as the number of concurrently generated spikes.

SN P systems have also been used to solve several computationally hard problems. The baseline idea, i.e., to activate the exponentially large number of inactive neurons in polynomial time, was proposed by [13], to solve a SAT problem. This idea was further extended in [14, 15], by starting with an exponentially large precomputed workspace instead of producing an exponential workspace in polynomial time. After that, a variant of SN P systems called Optimization Spiking Neural P systems (OSNP systems) was introduced in [16], to obtain an analytic solution for the knapsack problem, which is known to be NP-complete. This variant introduces a "guider" that adaptively adjusts rule probabilities to solve combinatorial optimization problems. OSNP systems were further improved in [17] to address the Travelling Salesman Problem (TSP), which is known to be NP-hard. The major difference w.r.t. [16] is that they employ a genetic algorithm (GA) to adjust the rule probabilities instead of following the guide algorithm specified in [16].

In [18], Ionescu et al. applied SN P systems for simulating logical gates. They encoded the Boolean values, 0 and 1 respectively, into one and two spikes, as inputs given to one neuron. This study inspired the application of SN P systems for designing the arithmetic logic unit used in CPUs. For instance, a variant of SN P systems using anti-spikes was proposed in [19], to perform arithmetic operations such as addition and subtraction, as well as logic operations such as AND, OR and NOT. Further improvements to this approach were made in

[20], with the introduction of asynchronous parallelism. XOR and NAND gate operations were later simulated by Song et al. in [21], using SN P systems with astrocyte-like control.

Another important area of application of SN P systems is fuzzy reasoning, particularly in the area of fault diagnosis in power systems. In [22], Peng et al. proposed a fuzzy reasoning Spiking Neural P system (FRSNP system) to handle fuzzy diagnosis knowledge and reasoning, which are indispensable for some fault diagnosis applications. Their proposed method includes several mechanisms such as fuzzy logic and a new firing mechanism, and was tested on the fault diagnosis of a transformer. Such FRSNP system was further improved in [23] and [24]. In particular, the method proposed in [23], called adaptive FRSNP system (AFRSNP system), is able to adjust the weights in the fault diagnosis model automatically. In [24], the efficiency of the AFRSNP system was further improved by optimizing the learning algorithm by means of particle swarm optimization (PSO).

Other works have used SN P systems and their variants to perform pattern recognition tasks. In this area, [25, 26] used SN P systems to implement a parallel method for image skeletonizing. The proposed method, based on SN P systems with weights that are associated with the synapses, was used as a thinning algorithm for skeletonizing binary images. SN P systems have been used also for fingerprint recognition. For this task, [27] proposed a double-layer self-organized SN P system that can adaptively create and delete neurons present in the different layers. Finally, Song et al. [28] introduced a variant called SN P system with Hebbian learning function to recognize English letters. The use of the learning function, which enables a dynamic update of the neuron connections during the computation, allowed the proposed method to obtain promising results compared to traditional neural networks based optimized by back-propagation.

While previous attempts to the automatic design of SN P systems do exist [8, 9], they still require an expert to specify the hyperparameters of these algorithms. Our approach, instead, removes almost all the hyperparameters, leaving only the number of rules (inside each neuron) as parameter that the user has to specify (besides the hyperparameters for NEAT).

Neuroevolution Neuroevolution, that is the application of evolutionary algorithms to optimize neural networks, is a growing field in Computational Intelligence. In this area, Neuroevolution of Augmenting Topologies (NEAT) [10] is a well-established technique that is capable to optimize both the parameters and topology of neural networks. The earliest applications of NEAT concerned evolutionary learning in control problems, such as pole balancing [10] and pole chasing [29].

Later, NEAT has been widely used in games, in particular to obtain optimal strategies to decide which action to take given as input a description of the current state of the game. In [30], NEAT was used to play Go by evaluating where the next stone should be placed. Taylor et al. [31] employed NEAT for the same task, but their proposed method accelerates learning by using transfer learning. A real-time version of NEAT (rtNEAT) was introduced in [32] to evolve

neural networks in real time, so that the proposed method makes agents improve their behavior while the game is being played. This method was tested on a game called neuroevolving robotic operatives (NERO), to improve the competitiveness of a virtual robot team in real time.

More recently, NEAT has been applied to the automatic design of deep neural networks (DNNs). In 2019, Miikkulainen et al. proposed a further extension of NEAT, called CoDeepNEAT [33], that can be applied to DNNs to perform coevolutionary optimization of topology, network components, and hyperparameters. They evaluated their method on various tasks: object recognition, language modeling and automated image captioning. CoDeepNEAT achieved promising results comparable to human-designed networks.

NEAT and its variants discussed above are restricted to traditional (connectivist) neural networks, including DNNs. In contrast, [34, 35] applied NEAT to spiking neural networks (SNNs). In particular, in [34] a powerful neuromorphic hardware called SpiNNaker was used for evolving neural controllers based on SNNs through NEAT. In [35], NEAT was used to develop a recurrent spiking controller that can solve nonlinear control problems in continuous domains. The proposed method was evaluated on a pole balancing task, demonstrating that the learning speed of the evolved spiking controller is significantly faster than that of a traditional neural network that makes use of a sigmoidal activation function.

Finally, we should note that while NEAT is one of the most popular neuroevolutionary algorithms, it is not the only one. In fact, several other methods have been recently proposed for evolutionary neural architecture search [36–38]. However, most of these methods evolve neural networks by composing high-level blocks, while the aim of our work is to evolve neural networks by optimizing them at the level of single neurons.

4 Method

In order to validate our approach, we consider here the automatic design of basic SN P systems (i.e., not the advanced variants discussed in Section 3). This choice was made to ensure that NEAT is able to deal at least with the simplest SN P systems. Moreover, for simplicity we consider neurons that have exclusively one spiking rule and one forgetting rule, and we set the initial number of spikes to $n_i = 0$. These assumptions make these neurons a specialization of the more general type of neurons that can be used in SN P systems. While these neurons may be significantly less expressive than the general ones, they allow us to adapt, with minimal effort, the NEAT algorithm. However, the reduced expressiveness of SN P systems with only two rules per neuron may reduce their performance in some tasks where higher expressiveness (for each neuron) is required.

Another assumption we make regards the weights: in fact, we evolve SN P systems whose connections may have a (either positive or negative) weight, as in [6]. The weight acts as a multiplier for the number of tokens that are produced by the neurons, i.e., given t_o tokens in output from neuron o and a connection

from o to p with weight $w_{o,p}$, the total number of spikes given in input to the neuron p is $t_p = t_o \cdot \ w_{o,p}$.

While, ideally, given an input we want to wait that the SN P system has finished the computation (i.e., no more spikes are produced in the system), this may significantly slow down the evolution. To speed up the evolutionary process, we constrain the computation in the SN P system to 100 timesteps.

4.1 NEAT

The NEAT algorithm starts the evolutionary process from minimal networks that, as the evolutionary process goes on, are complexified by means of mutation and crossover. Each network is encoded through two lists: the list of nodes and the list of connections. Moreover, the algorithm allows to perform efficient neuroevolution by allowing the detection of *similar* individuals, and preserves diversity by using niching. More details on these aspects of the algorithm can be found in [10].

4.2 Genotype

We adapt the original NEAT algorithm[1] to our purpose by including into the genotype the following parameters of every neuron σ_i:

- $c_i \in \mathbb{N} \setminus \{0\}$: no. of spikes needed to fire the neuron;
- $p_i \in \mathbb{N} \setminus \{0\}$: no. of output spikes;
- $d_i \in \mathbb{N}$: minimum delay between two subsequent spikes for the neuron;
- $f_i \in \mathbb{N}$: no. of spikes needed to activate the forgetting rule;
- $w_{j,i} \in \mathbb{Z}$: weight of synapse (j, i) for each synapse in input to the neuron.

Moreover, we also optimize the set of connections *syn*.

4.3 Phenotype

When translating a genotype into a phenotype (i.e., an instance of SN P systems), the following constraints are handled:

- $p_i \leq c_i$: if $p_i > c_i$ then it is set to $p_i = c_i$;
- $f_i < c_i$: if $f_i \geq c_i$ then it is set to $f_i = c_i - 1$.

Moreover, since during the initialization of the genotypes the values for the parameters are sampled from Gaussian distributions, to create the SN P systems the parameters are cast into integers by taking the floor of the number.

Each resulting phenotype consists then of a SN P system containing k neurons, where k is the number of non-hidden nodes from the genotype. Subsequently, a connection is created for each enabled connection gene, connecting the created previously neurons.

[1] Our code, which is based on the `neat-python` package [39], is publicly available online at https://github.com/leocus/snps.

4.4 Input features

The tasks considered in our experimentation use real-valued input data, which are incompatible with the data type employed in SN P systems. To mitigate this problem, we perform a very simple conversion from floating-point encoding to integer encoding. The conversion consists in normalizing each input in its range of variation:

$$\bar{x_k} = \frac{x_k - min(x_k)}{max(x_k) - min(x_k)} \tag{2}$$

Then, we convert the normalized input into a number of input spikes as follows:

$$t_k = \lfloor k \cdot \bar{x_k} \rfloor \tag{3}$$

In our experiments, we empirically set k to 20. Finally, the input spikes $\{t_0, \ldots, t_n\}$, where n is the number of inputs, are fed into the SN P system.

4.5 Fitness evaluation

After the genotype-phenotype mapping is applied, each SN P system is evaluated in n_{ep} episodes, where an episode consists in our case in a simulation of the control task at hand. At each timestep of a simulation, we transform the raw inputs coming from the simulator into integer features, using the procedure described in the previous subsection; then, we feed the features into the SN P system, which produces an output vector containing the number of spikes for each output neuron (with size equal to the number of actions). The action performed by the SN P system is then the argmax of the output vector.

When the n_{ep} episodes have been completed, the phenotype is assigned a fitness equal to the mean score across the n_{ep} episodes.

Note that, when evaluating the SN P system, each neuron cannot contain a negative number of spikes. So, when $s_i < 0$, the number of spikes inside that neuron is reset to $s_i = 0$.

5 Results

To test the capabilities of our approach, we evolve SN P systems on two classic control tasks, namely MountainCar-v0 and CartPole-v1, taken from the OpenAI Gym library [11]. While these two tasks may seem trivial for testing modern neuro-evolutionary approaches, it must be noted that, to our knowledge, this is the first application of SN P systems as controller for control tasks, thus their ability to work in these scenarios is yet to be determined. The MountainCar-v0 task is a "driving" task, where there is a car, initially in a valley, that has to reach the rightmost hill. To do so, the agent must learn how to build momentum by swinging between the two hills. In this case, the agent takes in input both the horizontal position and velocity of the car and must produce a decision $a \in \{0, 1, 2\}$ that corresponds to: accelerate to the left, do not accelerate, and accelerate to the right, respectively. In this task, each timestep gives a reward of

-1 points, so the quicker the agent solves the task, the higher the score. The task is considered solved if the agent reaches a score $s > -110$. On the other hand, the CartPole-v1 environment consists in a classic pole-balancing task. In this task, the agent takes the following inputs: horizontal position (x), horizontal velocity (y), angle of the pole (θ), angular velocity of the pole (ω), and must produce a binary decision $a \in \{0, 1\}$ that consists in moving the cart to the left or to the right, respectively. The reward given to the agent at each timestep consists in 1 point. If the pole falls, the simulation is terminated, If the maximum score is reached (500), the simulation is terminated. The task is considered as solved if the agent reaches a score $s > 475$. Table 1 shows the parameters used to evolve SN P systems with the NEAT algorithm. The same parameters were used on both the MountainCar-v0 and the CartPole-v1 tasks.

Table 1: Parameters used for the NEAT algorithm.

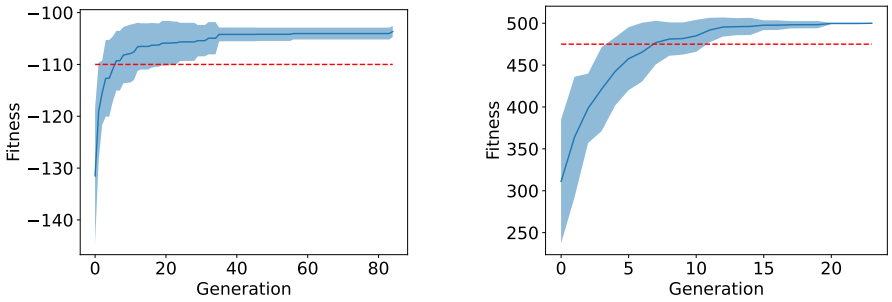
Parameter	**Value**	**Parameter**	**Value**
Population size	300	Generations	300
Init c_i	$\sim \mathcal{N}(30, 10)$	Init p_i	$\sim \mathcal{N}(30, 10)$
Init d_i	$\sim \mathcal{N}(30, 10)$	Init f_i	$\sim \mathcal{N}(30, 10)$
Init weight	$\sim \mathcal{N}(0, 3)$	c_i range	[1, 100]
p_i range	[1, 100]	d_i range	[0, 100]
f_i range	[1, 100]	Weight range	[-10, 10]
Mutation power	$\sim \mathcal{N}(0, 3)$	Mutation rate	0.2
Replacement rate	0.1	Add connection rate	0.5
Remove connection rate	0.5	Add node rate	0.5
Remove node rate	0.5	Toggle “enable” rate	0.1
Max stagnation period	20		

Table 2 shows the results obtained in 10 independent runs on the two tasks. Note that, to speed up the computation, we set a stopping criterion based on the score: if the agent achieves a mean score that is greater or equal than a threshold, the evolution is stopped. For the CartPole-v1 task, the maximum score is 500, while the minimum score required for solving the task is 475. In this case, we set the threshold to 499. On the other hand, for the MountainCar-v0 task there is no maximum score, and the minimum score required is -110. Here, we set the threshold to -105. While this stopping criterion may hinder reaching the global maximum, it significantly speeds up the evolution process. As shown in the table, our approach is able to solve the task in 100% of the cases (computed on 10 independent runs) for both tasks.

Figures 2a and 2b show the fitness trends averaged over the 10 independent runs for each task, where it can be observed that the proposed method is fairly robust across multiple runs, and converges quite quickly (after about 40 generations in the case of MountainCar-v0, 20 in the case of CartPole-v1).

Table 2: Descriptive statistics of the results obtained by the proposed method in 10 independent runs on the two tested tasks.

Task	**Min**	**Mean**	**Median**	σ	**Max**	**Solved**
MountainCar-v0	-105.18	-104.56	-104.37	0.34	-104.14	10/10
CartPole-v1	491.52	497.49	498.05	2.86	500.00	10/10



(a) MountainCar-v0 (b) CartPole-v1

Fig. 2: Fitness trend (mean ± std. dev. across 10 independent runs) for the best individuals found during the evolutionary process on the two tested tasks. The dashed line represents the "solved" threshold.

5.1 Analysis of the solutions

In the following, we analyze the best SN P system evolved for each task, trying to gain insights on the evolved policies.

MountainCar-v0 Figure 3 shows the diagram of the best SN P system obtained for this task, where t_x and t_v represent the number of tokens (i.e., spikes) obtained from the real-valued inputs coming from the simulator (x and v stand for position and velocity of the car, respectively) by means of the "translation" process described in Section 4.4. Each circle with a number inside represents a spiking P neuron, where the number represents its *id*. Each circle with a symbol inside represents an input neuron, where the symbol identifies the spikes generated from the corresponding input. Ignoring Neuron 0 and 1 (that do not contribute to the control of the agent), the evolved parameters are:

– Neuron 2 (Accelerate to the right):
$c_i = 43$, $p_i = 31$, $d_i = 31$, $f_i = 14$

The policy shown in Figure 3 acts as an "if-then-else" policy. In fact, since we choose the action based on the argmax, two scenarios may happen:

– Neuron 2 does not produce any spike: in this case, the action taken is 0 (Accelerate to the left). This happens because the argmax function returns the first index that contains the max.

– Neuron 2 produces one or more spikes: in this case, the argmax would be 2 and the agent will accelerate to the right.

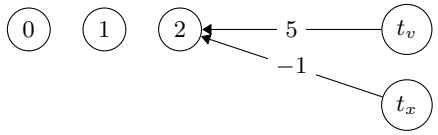


Fig. 3: Best SN P system evolved for the MountainCar-v0 task.

CartPole-v1 Figure 4 shows the best SN P system obtained for this task, which follows the same representation of Figure 3. In this case, the evolved parameters are:

– Neuron 0 (Accelerate to the left):
  $c_0 = 31$, $p_0 = 2$, $d_0 = 22$, $f_0 = 26$
– Neuron 1 (Accelerate to the right):
  $c_1 = 47$, $p_1 = 36$, $d_1 = 25$, $f_1 = 8$

This policy seems more complex than the one produced for the MountainCar-v0 task. In fact, here we observe that all the input variables (t_x, t_v, t_ω and t_θ) contribute (positively) to the output of Neuron 1, while only two variables, namely t_x and t_θ, contribute to the output of Neuron 0, and t_θ contributes negatively to it.

However, by inspecting how Neuron 0 works, we can conclude that also this policy can be reduced to an "if-then-else". In fact, for each spike in t_x 2 spikes will be added to this neuron, and, for each spike in t_θ, 5 spikes will be removed from it. Since a neuron cannot contain a negative number of spikes, the number of spikes that Neuron 0 can contain is:

$$s_{0,in} = max(0, 2(t_x - t_\theta)) \tag{4}$$

This means that, after all the spikes in t_θ have been consumed (assuming $t_x > t_\theta$) 2 spikes will be added to $s_{0,in}$ at each timestep. Then, three scenarios can happen:

– $t_x \leq t_\theta \Rightarrow s_{0,in} = 0$;
– $t_x - t_\theta \in [0, 13) \Rightarrow s_{0,in}$ will be too small to trigger any rule;
– $t_x - t_\theta \in [13, 20] \Rightarrow s_{0,in}$ will be equal to 26 at a certain point, so the *forgetting* rule will be applied. The number of spikes that can be added in the following steps will be insufficient to trigger any rule.

This means that Neuron 0 will never fire any spike and, for this reason, also this SN P system represents an "if-then-else" policy, similarly to what we obtained for the MountainCar-v0 task.

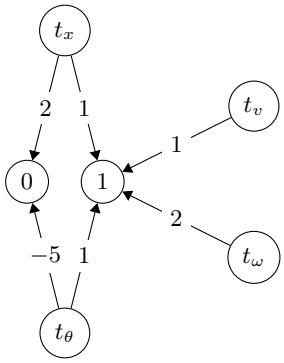


Fig. 4: Best SN P system evolved for the CartPole-v1 task.

5.2 Comparison with state of the art

Tables 3 and 4 compare our approach to the state of the art on the two tasks (from either the OpenAI Gym Leaderboard[2] or the literature). We observe that our approach is competitive with the state of the art. Moreover, the fact that we used a stopping criterion based on the fitness may have hindered the discovery of a better performing method on the MountainCar-v0 task.

Table 3: Comparison of our approach to the state of the art on the MountainCar-v0 task. The boldface indicates the best values known so far.

Source	**Method**	**Score**
Zhiqing Xiao[3]	Closed-form policy	-102.61
Keavnn[4]	Soft Q Networks [40]	-104.58
Harshit Singh[5]	Deep Q Network	-108.85
Colin M[6]	Double Deep Q Network	-107.83
Amit[7]	SARSA	-105.99
Anas Mohamed[8]	SARSA	-109.41
Custode & Iacca [41]	Decision Tree	**-101.72**
Ours	SN P system	-104.14

[2] github.com/openai/gym/wiki/Leaderboard
[3] github.com/ZhiqingXiao/OpenAIGymSolution
[4] github.com/StepNeverStop/RLs, accessed: 3 aug 2021.
[5] github.com/harshitandro/Deep-Q-Network
[6] github.com/CM-Data/Noisy-Dueling-Double-DQN-MountainCar
[7] github.com/amitkvikram/rl-agent
[8] github.com/amohamed11/OpenAIGym-Solutions

Table 4: Comparison of our approach to the state of the art on the CartPole-v1 task. The boldface indicates the best values known so far.

Source	**Method**	**Score**
Meng et al. [42]	Deep Q Network	327.30
Meng et al. [42]	Tree-Backup(λ)	494.70
Meng et al. [42]	Importance-Sampling	498.70
Meng et al. [42]	Qπ	489.90
Meng et al. [42]	Retrace(λ)	461.10
Meng et al. [42]	Policy discrepancy w/ β	499.90
Meng et al. [42]	Policy discrepancy w/ η	493.20
Meng et al. [42]	Watkins's Q(λ)	484.30
Meng et al. [42]	Policy discrepancy w/ β	494.90
Meng et al. [42]	Policy discrepancy w/ η	493.30
Meng et al. [42]	Peng & Williams's Q(λ)	496.70
Meng et al. [42]	Policy discrepancy w/ β	**500.00**
Meng et al. [42]	Policy discrepancy w/ η	499.40
Meng et al. [42]	General Q(λ)	499.90
Meng et al. [42]	Policy discrepancy w/ β	**500.00**
Meng et al. [42]	Policy discrepancy w/ η	**500.00**
Xuan et al. [43]	Deep Q Network	98.33
Xuan et al. [43]	Bayesian Deep RL	113.52
Xuan et al. [43]	Bayesian Deep RL weighted	136.75
Beltiukov [44]	K-FAC	321.00
Custode & Iacca [41]	Decision Tree	**500.00**
Ours	SN P system	**500.00**

6 Conclusions

Spiking Neural P (in short, SN P) systems are a computational tool from the field of membrane computing that gained a lot of attention in recent years. Several variants of these systems have been proposed for different applications. However, until now, these systems have almost always been manually derived from experts. The approach presented here aims to automate the design step, so that SN P systems can be automatically produced for a given task with no (or little) supervision from an expert. We tested our method on two classic control tasks, and found that our approach is able to solve both tasks in all the runs. Moreover, it produces controllers whose performances are competitive with the state of the art.

The most important limitation of this work is that we assumed rather simplified SN P systems. In particular, we limited the number of rules that each neuron can employ. In future work, we expect to remove this limitation. Moreover, we considered only a basic version of SN P systems, while it would be interesting to combine our approach with more advanced variants such as OSNP systems [16], in order to perform both "architectural search" and real-time parameter optimization.

Other relevant future directions include, for instance: 1) testing our approach on different (and more challenging) tasks; 2) applying techniques to allow SN P systems to work with non-integer inputs; 3) evolving SN P systems that, instead of using the number of spikes for each neuron, use the difference between two spikes as the output, as proposed in [2]; 4) extend our approach to evolve also the number of timesteps allowed in the computation; and 5) test novel approaches for the neuro-evolution of SN P systems, such as Cartesian genetic programming [45].

References

1. Gheorghe Păun: Computing with Membranes. Journal of Computer and System Sciences **61**(1) (2000) 108–143
2. Ionescu, Mihai and Păun, Gheorghe and Yokomori, Takashi: Spiking neural P systems. Fundamenta informaticae **71**(2, 3) (2006) 279–308
3. Păun, Gheorghe and Pérez-Jiménez, Mario J and Rozenberg, Grzegorz: Spike trains in spiking neural P systems. International Journal of Foundations of Computer Science **17**(04) (2006) 975–1002
4. Martín-Vide, C. and Pazos, J. and Păun, G. and Rodríguez-Patón, A.: A New Class of Symbolic Abstract Neural Nets: Tissue P Systems. In Ibarra, Oscar H. and Zhang, Louxin, ed.: International Computing and Combinatorics Conference (COCOON), Berlin, Heidelberg, Springer Berlin Heidelberg (2002) 290–299
5. Pan, Linqiang and Zeng, Xiangxiang: A note on small universal spiking neural P systems. In: International Workshop on Membrane Computing (WMC), Springer (2009) 436–447
6. Wang, Jun and Hoogeboom, Hendrik Jan and Pan, Linqiang and Păun, Gheorghe and Pérez-Jiménez, Mario J: Spiking neural P systems with weights. Neural Computation **22**(10) (2010) 2615–2646
7. Wang, Xun and Song, Tao and Gong, Faming and Zheng, Pan: On the computational power of spiking neural P systems with self-organization. Scientific reports **6**(1) (2016) 1–16
8. Dong, Jianping and Stachowicz, Michael and Zhang, Gexiang and Cavaliere, Matteo and Rong, Haina and Paul, Prithwineel: Automatic Design of Spiking Neural P Systems Based on Genetic Algorithms. International Journal of Unconventional Computing **16** (2021)
9. Casauay, Lovely Joy P and Cabarle, Francis George C and Macababayao, Ivan Cedric H and Adorna, Henry N and Zeng, Xiangxiang and MartÍNez-Del-Amor, Miguel Ángel and others: A Framework for Evolving Spiking Neural P Systems. International Journal of Unconventional Computing **16** (2021)
10. Stanley, Kenneth O and Miikkulainen, Risto: Evolving neural networks through augmenting topologies. Evolutionary computation **10**(2) (2002) 99–127
11. Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba: OpenAI Gym (2016)
12. Chen, Haiming and Ishdorj, Tseren-Onolt and Paun, Gheorghe and Pérez Jiménez, Mario de Jesús: Spiking neural P systems with extended rules. In: 4h Brainstorming Week on Membrane Computing (BWMC). Volume I., Sevilla, ETS de Ingeniería Informática, 30 de Enero-3 de Febrero, Fénix Editora (2006) 241–265
13. Chen, Haiming and Ionescu, Mihai and Ishdorj, Tseren-Onolt: On the efficiency of spiking neural P systems. In: 4h Brainstorming Week on Membrane Computing

(BWMC). Volume I., Sevilla, ETS de Ingeniería Informática, 30 de Enero-3 de Febrero (2006) 195–206
14. Ishdorj, Tseren-Onolt and Leporati, Alberto: Uniform solutions to SAT and 3-SAT by spiking neural P systems with pre-computed resources. Natural Computing **7**(4) (2008) 519–534
15. Leporati, Alberto and Gutiérrez-Naranjo, Miguel A: Solving Subset Sum by spiking neural P systems with pre-computed resources. Fundamenta Informaticae **87**(1) (2008) 61–77
16. Zhang, Gexiang and Rong, Haina and Neri, Ferrante and Pérez-Jiménez, Mario J: An optimization spiking neural P system for approximately solving combinatorial optimization problems. International Journal of Neural Systems **24**(05) (2014) 1440006
17. Qi, Feng and Liu, Mengmeng: Optimization spiking neural P system for solving TSP. In: International Conference on Machine Learning and Intelligent Communications (MLICOM), Springer (2017) 668–676
18. Ionescu, Mihai and others: Some applications of spiking neural P systems. Computing and Informatics **27**(3+) (2008) 515–528
19. Peng, Xian Wu and Fan, Xiao Ping and Liu, Jian Xun: Performing balanced ternary logic and arithmetic operations with spiking neural P systems with anti-spikes. Advanced Materials Research **505** (2012) 378–385
20. Hamabe, Ryoma and Fujiwara, Akihiro: Asynchronous SN P systems for logical and arithmetic operations. In: International Conference on Foundations of Computer Science (FCS), The Steering Committee of The World Congress in Computer Science (2012) 1
21. Tao Song and Pan Zheng and M.L. Dennis Wong and Xun Wang: Design of logic gates using spiking neural P systems with homogeneous neurons and astrocytes-like control. Information Sciences **372** (2016) 380–391
22. Peng, Hong and Wang, Jun and Pérez-Jiménez, Mario J and Wang, Hao and Shao, Jie and Wang, Tao: Fuzzy reasoning spiking neural P system for fault diagnosis. Information Sciences **235** (2013) 106–116
23. Tu, M. and Wang, Jindo and Peng, Hong and Shi, P.: Application of Adaptive Fuzzy Spiking Neural P Systems in Fault Diagnosis of Power Systems. Chinese Journal of Electronics **23** (2014) 87–92
24. Wang, Jun and Peng, Hong and Tu, Min and Pérez-Jiménez, J Mario and Shi, Peng: A fault diagnosis method of power systems based on an improved adaptive fuzzy spiking neural P systems and PSO algorithms. Chinese Journal of Electronics **25**(2) (2016) 320–327
25. Díaz-Pernil, Daniel and Peña-Cantillana, Francisco and Gutiérrez-Naranjo, Miguel A: A parallel algorithm for skeletonizing images by using spiking neural P systems. Neurocomputing **115** (2013) 81–91
26. Song, Tao and Pang, Shanchen and Hao, Shaohua and Rodríguez-Patón, Alfonso and Zheng, Pan: A parallel image skeletonizing method using spiking neural P systems with weights. Neural Processing Letters **50**(2) (2019) 1485–1502
27. Ma, Tongmao and Hao, Shaohua and Wang, Xun and Rodríguez-Patón, Alfonso Alfonso and Wang, Shudong and Song, Tao: Double Layers Self-Organized Spiking Neural P Systems With Anti-Spikes for Fingerprint Recognition. IEEE Access **7** (2019) 177562–177570
28. Song, Tao and Pan, Linqiang and Wu, Tingfang and Zheng, Pan and Wong, ML Dennis and Rodríguez-Patón, Alfonso: Spiking neural P systems with learning functions. IEEE Transactions on Nanobioscience **18**(2) (2019) 176–190

29. Pardoe, David and Ryoo, Michael and Miikkulainen, Risto: Evolving neural network ensembles for control problems. In: Genetic and Evolutionary Computation Conference. (2005) 1379–1384
30. Stanley, Kenneth O and Miikkulainen, Risto: Evolving a roving eye for go. In: Genetic and Evolutionary Computation Conference, Springer (2004) 1226–1238
31. Taylor, Matthew E and Whiteson, Shimon and Stone, Peter: Transfer via inter-task mappings in policy search reinforcement learning. In: International joint conference on Autonomous agents and multiagent systems (AAMAS). (2007) 1–8
32. Stanley, K.O. and Bryant, B.D. and Miikkulainen, R.: Real-time neuroevolution in the NERO video game. IEEE Transactions on Evolutionary Computation **9**(6) (2005) 653–668
33. Miikkulainen, Risto and Liang, Jason and Meyerson, Elliot and Rawal, Aditya and Fink, Daniel and Francon, Olivier and Raju, Bala and Shahrzad, Hormoz and Navruzyan, Arshak and Duffy, Nigel and others: Evolving deep neural networks. In: Artificial intelligence in the age of neural networks and brain computing. Elsevier (2019) 293–312
34. Vandesompele, Alexander and Walter, Florian and Röhrbein, Florian: Neuroevolution of spiking neural networks on SpiNNaker neuromorphic hardware. In: Symposium Series on Computational Intelligence (SSCI), IEEE (2016) 1–6
35. Qiu, Huanneng and Garratt, Matthew and Howard, David and Anavatti, Sreenatha: Evolving spiking neural networks for nonlinear control problems. In: Symposium Series on Computational Intelligence (SSCI), IEEE (2018) 1367–1373
36. Suganuma, Masanori and Shirakawa, Shinichi and Nagao, Tomoharu: A genetic programming approach to designing convolutional neural network architectures. In: Genetic and Evolutionary Computation Conference. (2017) 497–504
37. Assunção, Filipe and Lourenço, Nuno and Machado, Penousal and Ribeiro, Bernardete: DENSER: deep evolutionary network structured representation. Genetic Programming and Evolvable Machines **20**(1) (2019) 5–35
38. Lu, Zhichao and Whalen, Ian and Boddeti, Vishnu and Dhebar, Yashesh and Deb, Kalyanmoy and Goodman, Erik and Banzhaf, Wolfgang: NSGA-net: neural architecture search using multi-objective genetic algorithm. In: Genetic and Evolutionary Computation Conference. (2019) 419–427
39. McIntyre, A., Kallada, M., Miguel, C.G., da Silva, C.F.: neat-python. `https://github.com/CodeReclaimers/neat-python`
40. Jingbin Liu and Xinyang Gu and Shuai Liu and Dexiang Zhang: Soft Q-network. arXiv:1912.10891 [cs] (2019)
41. Leonardo Lucio Custode and Giovanni Iacca: Evolutionary learning of interpretable decision trees (2021)
42. Meng, Wenjia and Zheng, Qian and Yang, Long and Li, Pengfei and Pan, Gang: Qualitative Measurements of Policy Discrepancy for Return-Based Deep Q-Network. IEEE Transactions on Neural Networks and Learning Systems (2019) 1–7
43. Xuan, Junyu and Lu, Jie and Yan, Zheng and Zhang, Guangquan: Bayesian Deep Reinforcement Learning via Deep Kernel Learning. International Journal of Computational Intelligence Systems **12**(1) (2018) 164–171
44. Beltiukov, Roman: Optimizing Q-Learning with K-FAC Algorithm. In: International Conference on Analysis of Images, Social Networks and Texts (AIST). Springer, Cham (2020) 3–8
45. Miller, Julian Francis and Harding, Simon L: Cartesian genetic programming. In: Genetic and Evolutionary Computation Conference - Companion. (2008) 2701–2726