Spiking P System simulator for image recognition

This document aims to formally and operationally describe the code available at this link. As this is an ongoing work, the features described herein refer to the commit dated 18/08/2025. The model has been developed within a collaboration aimed at implementing Spiking Neural P Systems with a high degree of realism. The simulator is capable of handling P Systems under various aspects: it can generate them, load them from input, simulate them, modify their weights and rules, compute their energy cost, and analyze both their structure and performance.

Contents

1 Possible configurations

The Spiking Neural P system model presented in [1] supports multiple modes of operation. Likewise, the simulator can adopt different execution and halting behaviors depending on how input and output are handled. In the model, each neuron can be assigned one of the following types: type 0 (input neuron), type 1 (intermediate neuron), or type 2 (output neuron).

Input

- **None**: there are no neurons of type 0. The system operates without any external spike injection.

- **Spike train**: each input neuron receives a predefined spike train, with one spike value provided at each computation step. One or more input neurons should be present, and in this configuration, they all receive the same spike sequence.
- **Images**: a set of type 0 neurons is associated with the pixels of a binarized input image. Each neuron fires a spike if and only if its corresponding pixel is active (i.e., carries a value of 1 in the binarized image).

Output

- **Halting**: the network has no output neurons and works in the acceptor mode. The result is determined by whether the system halts. If it reaches a configuration where no rules are applicable and no neuron is in a refractory state, the input is *accepted*; if the computation continues indefinitely (and is thus stopped by the maximum step limit), the input is *rejected*.
- **Generative**: typically, there is a single output neuron. The output is given by the number of computation steps between the first and second firing of this neuron. The computation halts immediately after the second spike. Note that this is the standard convention, but other SN P systems may define halting conditions differently.
- **Prediction**: the last layer of the network contains one output neuron for each class in the dataset. Each output neuron accumulates spikes proportionally to the likelihood that the input belongs to its associated class.
- **Images**: There is a large number of output neurons, one for each pixel, ordered numerically. To reconstruct the image generated by the network, the opposite process of the input is applied: a pixel is assigned to the image if the corresponding neuron has accumulated a charge.

Determinism

- **Deterministic**: the same input will always produce the same output. Rule application follows a predefined priority scheme, equivalent to an intrinsic ordering of rules in the CSV file.
- **Non-deterministic**: the set of applicable rules is randomly shuffled at each step, potentially leading to different outcomes for the same input.

Examples

- SNPS generating all even numbers. Input: none, output: generative, non-deterministic. See Fig.3 of [1].
- SNPS determining whether a number is divisible by 3. Input: spike train, output: halting, deterministic. See example 9 of [2].
- SNPS for edge detection. Input: images, output: images, deterministic. See Chapter 3.
- SNPS for image classification. Input: images, output: prediction, deterministic. See Chapter 4.

2 Rules codification and application

How the simulator executes and encodes the rules is of fundamental importance in the project. A network is entirely defined by a CSV file, where each row describes a single neuron. Each neuron in the file has a list of values called id, $initialCharge$, $outputTargets$, $neuronType$, $rules$, and the rules are theoretically unlimited in number, defined as tuples of the form $[div, mod, source, target, delay]$. Let's examine the elements.

- **Regex definition** div and mod make it possible to describe the regular expression present in the formula, which must match the number of spikes available. These spikes form an alphabet consisting solely of a's (for example, a neuron with 6 spikes has the alphabet aaaaaa). A rule in such an alphabet can always be written as $a^{mod}(a^{div})^*$, where the exponents are the values from the tuple used to describe it. For example, the regex $a(aa)^+a$ can be rewritten as $aaaa(aa)^*$ both through mathematical transformation and because both expressions represent the same concept: accept all even numbers greater than 2.

- **Rule application** To be described, a rule also needs a value that defines the number of spikes that will be consumed upon its application, and thus required for firing. To apply a rule, you need the values $mod, div, source$ of the tuple $[div, mod, source, target, delay]$, where $charge$ is the internal charge of the neuron, which must match the regex. The code to evaluate the regex is:

```
def check(self, charge):
    if charge > 0 and charge >= self.mod:
        if self.div > 0:
            return charge >= self.source and (charge - self.
                mod) % self.div == 0
        if self.div == 0:
            return charge >= self.source and charge == self.
                mod
    return False
```

Listing 1: `check` Method

The charge is checked to ensure it is not negative, because a neuron with no spikes will never fire. It is also checked whether it is greater than mod, because no formula of the type $a^{mod}(a^{div})^*$ can be matched by an expression of the form a^{charge} if $charge < mod$. With this formula, it is therefore possible to apply all such regexes.

Implementation workarounds: for now, if a rule contains 0 in the source value, this is interpreted as 'consume all spikes, therefore set to 0'. This was implemented as a practical necessity during the first firing phase of rules and would require an extension of the formalism allowing regular expressions over consumed charges, rather than just fixed values. This problem arises only in a fast network analyzing one image per step, which is currently the test model, but may be changed. To solve this issue, it is sufficient to define a forgetting function that accepts values below the limit and deletes all present spikes. One of the next development steps will certainly involve ensuring that all rules are fully realistic and compliant with the described model, avoiding ad-hoc programming shortcuts.

- **Target** The fourth value, *target*, can be 0 or 1. 0 indicates a forgetting rule, while 1 indicates a firing rule.
- **Delay** The last value of the tuple is the number of steps required for the neuron to activate the rule, during which the neuron remains inactive.

3 Edge detection SNPS

The implemented Spiking Neural P System performs edge detection on grayscale images, inspired by the neuromorphic photonic processing approach described in [3]. To describe its structure, reference is made to images from the BloodMNIST [4] database, measuring 28×28, but the code is ready to adapt to any type of image. The architecture is composed of three layers:

First layer: binarized pixel encoding

The first layer consists of 28×28 input neurons (type 0), each corresponding to a pixel in the binarized input image. Each neuron fires if the associated pixel contains a spike, and remains silent otherwise. The output spikes from these neurons are simultaneously transmitted to six parallel subnetworks in the second layer, each subnetwork implementing a distinct 2×2 convolution kernel.

Second layer: kernel-based feature extraction

The second layer is composed of six independent subnetworks, each with a 27×27 neuron grid (type 1), corresponding to the valid convolution area for a 2×2 kernel applied to the input image.

Each subnetwork uses a different kernel matrix with values $\{1, -1\}$, where:

- 1 indicates that a spike from the corresponding input pixel should be propagated to the target neuron.
- -1 indicates that an anti-spike (inhibitory signal) should be propagated instead.

The six kernels are:

$$\begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}, \quad \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}, \quad \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

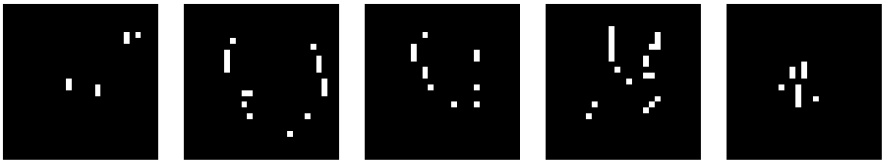
Each kernel is tuned to detect edges at a specific orientation: two vertical, two horizontal, and two diagonal configurations.

Unlike the work in [3], where the image consisted of 1 and -1 and the activation threshold was 4, the neurons in this layer fire only when the total charge equals 2, which corresponds to the matching condition for the kernel.

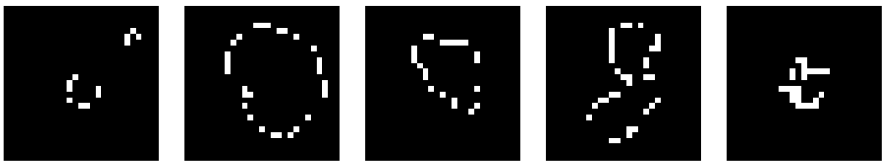
Third layer: edge aggregation

The third and final layer consists of 27×27 output neurons (type 2). Each neuron receives inputs from the corresponding position in all six second-layer subnetworks, summing their activations. The result is a spike map highlighting the edges in the input image, as can be seen in Fig.1.

(a) Input images



(b) Edge detection with the first 2 kernels (vertical edges)



(c) Final edge detection

Figure 1: Process of edge detection in the first 5 images of BloodMNIST (MedMNIST [4]) database

4 Image classification SNPS

One of the main objectives of this collaboration has been to establish the groundwork and initiate the exploration toward a realistic implementation of mechanisms intrinsic to the human brain. To this end, it was necessary to design a P System capable of classifying, albeit in a basic manner, a dataset of images divided into classes. The following section analyzes the structure of this model, followed by a description of its two separate training phases and the corresponding energy costs.

4.1 Model structure

The Spiking Neural P System is composed of a three-layer architecture designed to process visual input through spiking activity. The input image is first split into its three color channels (Red, Green, and Blue), and each channel is independently processed through an identical but separate P System. Each of those P Systems has two train phases, the first related to the rules and the second to the synapses.

First layer: Binarized pixel encoding

In the first layer, each neuron is assigned to a single pixel. This results in a one-to-one mapping between input pixels and spiking neurons. Before reaching this layer, pixel values are binarized (0 or 1) based on a threshold, effectively producing a binary spiking pattern representing the presence or absence of activity in each pixel.

Each color channel is handled independently, so the network is effectively triplicated at this stage, with one subnetwork per color channel. These subnetworks will be trained separately, allowing them to learn channel-specific characteristics.

Second layer: Configurational blocks

The second layer is a downsampled representation of the first layer, where each neuron in this layer receives inputs from a square receptive field (window) of neurons in the previous layer. A typical configuration uses a 4×4 window, meaning each second-layer neuron integrates inputs from 16 first-layer neurons.

For example, an input image of size 28×28 would yield a second layer of size 7×7, since the spatial resolution is reduced by a factor of the window size in both dimensions. This creates a compact representation that retains local spatial patterns. Note that those are example numbers; in the simulator, all the parameters can be easily changed.

Initially, second-layer neurons fire if they receive at least one spike (i.e., the firing threshold is 1). During the first training phase, these thresholds are adapted based on firing frequency: neurons that fire too often will increase their threshold, while those that rarely fire will lower it. This calibration step ensures a more balanced activation across neurons and avoids bias towards centrally located features.

Third layer: Classification

The third layer consists of a number of neurons equal to the number of output classes in the dataset. This layer is fully connected to the second layer, meaning each second-layer neuron connects to every output neuron.

During the second training phase, synaptic connections between the second and third layers are refined. Synapses that consistently lead to incorrect classifications are either pruned or converted into inhibitory connections, based on variables defined in the input. Conversely, synapses that reliably lead to correct classifications are preserved. This selective connectivity helps the network sharpen its decision boundaries and improve classification performance.

Training phases

The training process is divided into two distinct stages:

- **Threshold adaptation:** Second-layer neuron firing thresholds are adjusted proportionally to their activity levels. For example, a neuron firing 50% of the time might adjust its threshold to 8 (in a 4×4 window), while one that fires almost always may approach a threshold of 15 or 16.
- **Synaptic pruning and inhibition:** Synapses are evaluated based on their contribution to correct or incorrect classifications. Those associated with wrong predictions are removed or inhibited, while beneficial ones are retained.

4.2 Train phases

4.2.1 Rules train phase

The first training phase aims to calibrate the firing thresholds of the second-layer neurons. Initially, each neuron fires upon receiving only a single spike, which often leads to excessive firing. To regulate this, the system monitors firing frequency and adjusts neuron rules based on their observed activation behavior.

Firing count logging: Each time a second-layer neuron fires, its activity is logged in a firing count array. This allows the system to track how frequently each neuron is activated:

```
def fire(self, rule):
    if Config.NEURONS_LAYER1 <= self.nid < Config.
       NEURONS_LAYER1_2:
        self.snp_system.layer_2_firing_counts[self.nid - Config.
           NEURONS_LAYER1] += 1
```

Listing 2: Logging firings for second-layer neurons

Only neurons in the second layer, based on their ID, are tracked for this training step.

Normalization of firing counts: After the full training dataset has been processed, the firing counts are normalized based on the number of images. This generates a new threshold matrix which determines the future firing behavior of each neuron:

```
def normalize_rules(firing_counts, imgs_number):
    min_threshold = 1
    max_threshold = Config.BLOCK_SHAPE**2
    norm = firing_counts / imgs_number
    threshold_matrix = norm * (max_threshold - min_threshold) +
       min_threshold
    threshold_matrix = np.round(threshold_matrix).astype(int)
    SNPS_csv(threshold_matrix)
```

Listing 3: Normalization and rule tuning

Neurons that fire more frequently receive a higher threshold to reduce their future activations. This ensures that the activity across the layer is balanced, preventing dominant neurons from overshadowing the rest.

Thresholds assignment: Finally, the thresholds calculated in the previous step are used to define new firing rules for each second-layer neuron. These are written into the configuration that governs the network behavior:

```
for neuron_id in range(Config.NEURONS_LAYER1, Config.
   NEURONS_LAYER1_2):
    firing_threshold = threshold_array[neuron_id - Config.
       NEURONS_LAYER1]
    firing_rule = f"[1,{firing_threshold},0,1,0]"
    writer.writerow([
        neuron_id,              # id
        0,                      # initial_charge
        output_targets,         # output_targets
        1,                      # neuron_type
        firing_rule,            # firing rule based on input matrix
        "[1,1,1,0,0]"           # forgetting rule if didn't fire
    ])
```

Listing 4: Assigning updated firing rules

Through this mechanism, each neuron adapts to its individual activity level, encouraging a more even contribution from all neurons in the layer during inference.

4.2.2 Synapses train phase

The second phase of training is applied to the synaptic connections between the second and third layers. After the firing thresholds are calibrated, the goal of this phase is to strengthen the synapses that correctly contribute to classification, inhibit misleading ones, and prune the rest.

Synaptic reinforcement: During each training step, the system compares the current firing counts with those from the previous step to identify which neurons in the second layer have fired. For every active neuron, its connections to the correct output class are reinforced, while its connections to incorrect classes are weakened:

```
fired_diff = self.layer_2_firing_counts - self.
    old_layer_2_firing_counts
fired_indices = np.where(fired_diff > 0)[0]  # index of firing
    neurons
if fired_indices.size > 0:
    label = self.labels[self.t_step - 2]  # class label of
        current input
    for idx in fired_indices:
        self.layer_2_synapses[label][idx] += Config.
            POSITIVE_REINFORCE
        for wrong_label in range(Config.CLASSES):
            if wrong_label != label:
                self.layer_2_synapses[wrong_label][idx] -= Config
                    .NEGATIVE_PENALIZATION
    self.old_layer_2_firing_counts = self.layer_2_firing_counts.
        copy()
```

Listing 5: Updating synaptic weights based on firing activity

In this way, the network accumulates a synaptic weight matrix indicating the relevance of each second-layer neuron to each output class.

Pruning and inhibition: Once the training iterations are complete, the system prunes the synaptic matrix based on the accumulated weights. Only a fraction of the strongest connections (as defined by a pruning percentage in the `Config` file) are preserved as excitatory, while the weakest are converted into inhibitory synapses. All remaining connections are discarded:

```
def prune_matrix(synapses):
    keep_matrix = np.zeros_like(synapses, dtype=int)  # start
        with all 0
    for class_idx in range(synapses.shape[0]):
        weights = synapses[class_idx]
        num_excite = int((1 - Config.PRUNING_PERC - Config.
            INHIBIT_PERC) * len(weights))
        num_inhibit = int(Config.INHIBIT_PERC * len(weights))

        excite_indices = np.argsort(weights)[-num_excite:]
        inhibit_indices = np.argsort(weights)[:num_inhibit]
        keep_matrix[class_idx, excite_indices] = 1
        keep_matrix[class_idx, inhibit_indices] = -1
    return keep_matrix
```

Listing 6: Synaptic pruning and inhibition

The resulting synaptic structure is no longer fully connected. Instead, each output neuron is connected only to a selected subset of second-layer neurons. Some connections are excitatory (positive weights), others inhibitory (negative weights), and the rest are removed entirely, leading to a more efficient and selective network architecture.

4.3 Energy costs

To obtain a meaningful estimate of the network's energy cost, the type and number of operations required to process an image are calculated. Unlike ANNs and SNNs, which typically perform operations in floating point, this model uses only boolean values (at the synaptic level) and integers (for spikes within neurons). This significantly reduces computational costs. However, one must consider the energy cost of evaluating the regular expression associated with each neuron—a cost not present in the two models mentioned above. Table 1 provides an estimate of operating costs based on CMOS logic. It is worth noting that SNNs can obtain much higher energy-efficiency on neuromorphic hardware platform. These energy costs lead to Table 2, which shows an estimate of the costs for the operations present in the Spiking Neural P Systems simulator.

Operation	**Energy cost**	**Reference**
Switching	∼0.0021-0.047 fJ	See Fig.1 in [5]
A single logic gate (AND, OR, NOT, ...) on 1 bit	∼0.1-1 fJ	See [6], that cite [7] and [8].
Simple regular expressions (==, less then, %2)	∼1-10 fJ	See [8]
INT4	∼250-1600 fJ	See Fig.1 in [5]
FP32	∼4.8-63 pJ OR ∼0.9-3.7 pJ	See Fig.1 in [5] OR table IV in [9]

Table 1: Joule costs of some operations in CMOS logic

Model Component	**Estimated Cost**	**Operation**
Boolean spike sum	∼0.1–1 fJ	A single logic gate for each incoming spike
Simple neuron REGEX (e.g. $\geq N$ or ==N)	∼1–10 fJ	Simple regular expression
Complex neuron REGEX (e.g., "multiple of 4 but not 6")	∼10–100 fJ	Small logic network: combination of multiple gates and conditions

Table 2: Energy estimates for basic computational operations in the model, based on typical CMOS logic gate consumption.

Worst-case estimate

- Each synapse fires at every step, consuming approximately 1 fJ per spike.
- Each neuron evaluates complex regular expressions at every step, with an estimated cost of 100 fJ per rule.
- Assuming the entire model contains nr rules to be checked at each step, the worst-case cost is:

$$E_{\text{worst}} = (N_{\text{synapses}} + 100 \cdot nr)\,\text{fJ}$$

Case-specific estimate (realistic scenario)

- We can choose to maintain the photoresistive encoding, or invert it. If the dataset contains mostly 1s, after training, the synaptic matrix and rule thresholds can be inverted, allowing inversion of the input data as well. This inversion ensures that no more than 50% of synapses are activated during input image processing. In deeper layers, this assumption may not hold, but rule thresholds are calibrated to maintain appropriate firing rates. Thus, assuming that roughly 50% of synapses are active, and each active synapse costs 0.5 fJ:

$$E_{\text{synapses}} = 0.5 \cdot N_{\text{synapses}} \cdot 0.5\,\text{fJ}$$

- The rules in the current P system are simple threshold checks, costing about 10 fJ each. However, as noted in [10] that *'In SNPSs, neurons only become active when they accumulate enough spikes to trigger a rule, which allows many neurons to remain inactive, further reducing energy consumption. This sparse and asynchronous activity is a defining characteristic that enhances the efficiency of SNPSs"*. Therefore, only neurons that actually fire contribute to the energy cost. Assuming about 50% of neurons are active at each step:

$$E_{\text{neurons}} = N_{\text{neurons}} \cdot 0.5 \cdot 10\,\text{fJ}$$

  And this leads to a formula of:

$$E_{\text{total}} = (0.5 \cdot N_{\text{synapses}} \cdot 0.5 \; + 0.5 \cdot N_{\text{neurons}} \cdot 10\,)\,\text{fJ}$$

- In our specific case, a 4×4 convolutional window is applied to a 28×28 input, using a stride of 4, resulting in a 7×7 feature map (i.e., 49 neurons). These are then connected to 8 output classes. The estimated value turns out to be:

$$E_{\text{total}} = (0.5 \cdot 1176 \cdot 0.5 \; + 0.5 \cdot 841 \cdot 10\,)\,\text{fJ} = 1500\text{fJ} = 1.5\text{pJ}$$

  Note that, in the code, we have the exact number of spikes generated and rules applied, and we can distinguish between firing rules and forgetting rules, so the value in joules obtained from the code will be more accurate.

ANN comparison

- By examining the work in [9], particularly Table IV, we observe that a 32-bit multiply-and-accumulate (MAC) operation consumes approximately 4.6 pJ of energy. Based on this, we can estimate the energy required by the model if it were implemented as a fully connected network:

$$E_{\text{ANN}} = (N_{\text{layer1}} \cdot N_{\text{layer2}} + N_{\text{layer2}} \cdot N_{\text{classes}}) \cdot 4.6\,\text{pJ}$$

- However, our current code instead uses a convolutional layer as the first layer, followed by a fully connected (FC) layer as the second. Therefore, the energy cost becomes:

$$E_{\text{CNN+FC}} = (N_{\text{layer1}} + N_{\text{layer2}} \cdot N_{\text{classes}}) \cdot 4.6\,\text{pJ}$$

The specific case considered and described above led to an estimated energy cost of:

$$E_{\text{CNN+FC}} = (28 \cdot 28 + 7 \cdot 7 \cdot 8) \cdot 4.6\,\text{pJ} = 5400\text{pJ}$$

Therefore, looking solely at these values, there is an advantage of 3 orders of magnitude. It should be noted that this is a very rough estimate based on the CMOS component. In order to work better on this issue, variables relating to the number of firing and forgetting rules applied, as well as the number of spikes emitted and inhibited, have been included in the code.

5 Towards realism

Thanks to the collaboration with Matteo Balzerani, several important realistic mechanisms in image analysis have been identified. A summary of the main ones is presented here, together with a description of possible implementations in the developed code.

Mechanism 1: Cones, rods and parallel pathways

- **Functioning:** Human photoreceptors are of two types: **cones**, which transmit chromatic information, and **rods**, which encode only brightness. Their signals follow two parallel pathways: the magnocellular (rod-based, low resolution, no color, "where" pathway) and the parvocellular (cone-based, high resolution, with color, "what" pathway). This division allows the brain to process spatial aspects without requiring detailed chromatic information.

- **Purpose:** Separating color and non-color processing reduces the computational load, while still achieving accurate results. This is essential since neural resources are finite.

- **Possible application:** In addition to the standard RGB channels, one could add a fourth input layer representing a grayscale version of the image. Parallel subnetworks could then analyze color information or monochrome orientation separately, similar to biological pathways.

- **Translation:** This mechanism has been applied in the edge detection network described in Section 3. By introducing a fourth black-and-white channel, edges are extracted directly, avoiding the costly process of separating and recombining information across the three color channels. Currently, grayscale is computed with a weightless average, but this could be replaced by the luminosity method, where the average is weighted according to the eye's sensitivity to different wavelengths. Currently, six 2×2 matrices are used for edge detection, but the use of ganglion cells is being considered for this task.

Mechanism 2: Anti-spikes

- **Functioning:** In our brain, presynaptic signals can be excitatory or inhibitory. The effect depends on the receptors of the postsynaptic cell, not on the neurotransmitter itself: the same spike can thus be excitatory for one neuron and inhibitory for another. Correct integration is obtained through the algebraic sum of all received signals.

- **Purpose:** The aim is to reproduce realistic integration by including inhibitory signaling. In the human brain, inhibition is crucial for balanced coding: excessive excitation, for example, is linked to pathological states such as anxiety. Therefore, proper coding requires not only spikes, but also opposite signals, called anti-spikes.

- **Possible application:** Instead of pruning synapses during training, the model could reverse their effect by predicting anti-spikes. At any time t, the effective input to a postsynaptic neuron could be defined as:

$$input\_spike_{post} = output\_spike_{pre} - output\_antispike_{pre} \tag{1}$$

  This would allow the network to exploit not only positive contributions, but also explicit inhibitory signals, similar to how biological systems refine information processing.

- **Translation:** This feature was implemented in a relatively simple way. In the neuron description, where the IDs of connected neurons are listed, it is possible to insert a negative value. For example, in the *outputTargets* one might find $[10, 15, -18]$. This means that the neuron sends spikes to neurons 10 and 15, but removes charge from neuron 18 through an anti-spike. Negative values are added during the second training phase, related to synapses, described in Section 4.2.2.

Mechanism 3: Graduated phototransduction

- **Functioning:** Photoreceptors encode light intensity through graded neurotransmitter release, unlike the binary spiking used in visual processing.

- **Purpose:** Preserve information on brightness levels, avoiding loss from simple input binarization.

- **Possible application:** Replace ON/OFF input encoding with discretized intensity values. Each RGB channel of a pixel could be mapped to n spikes proportional to brightness (e.g., dividing $[0, 255]$ into subranges).

- **Translation:** This functionality has not yet been implemented, as it substantially changes how the input is interpreted. Moving from a Boolean input to a range of values requires either more neurons per pixel or higher maximum charges for input neurons. These neurons would then need to handle higher charges with rules that inform the network about the charge peak. However, all rules are constrained to firing only a single spike, making them indistinguishable to the output neuron that receives them. An effective implementation of this mechanism may therefore require a more complex input management strategy, or even a modification of the simulated model itself, introducing features not present in classical Spiking Neural P Systems. In reality, photoreceptors fire more in darkness, implying an inverted

encoding. Both direct and inverted mappings could be tested, possibly adapting the choice to average image brightness.

References

[1] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. "Spiking neural P systems". In: *Fundamenta informaticae* 71.2-3 (2006), pp. 279–308.

[2] Sergey Verlan, Rudolf Freund, Artiom Alhazov, Sergiu Ivanov, and Linqiang Pan. "A formal framework for spiking neural P systems". In: *Journal of Membrane Computing* 2.4 (2020), pp. 355–368.

[3] Joshua Robertson, Paul Kirkland, Juan Arturo Alanis, Matěj Hejda, Julián Bueno, Gaetano Di Caterina, and Antonio Hurtado. "Ultrafast neuromorphic photonic image processing with a VCSEL neuron". In: *Scientific reports* 12.1 (2022), p. 4874.

[4] Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. "Medmnist v2-a large-scale lightweight benchmark for 2d and 3d biomedical image classification". In: *Scientific Data* 10.1 (2023), p. 41.

[5] Sadasivan Shankar and Albert Reuther. "Trends in energy estimates for computing in ai/machine learning accelerators, supercomputers, and compute-intensive applications". In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2022, pp. 1–8.

[6] Reza Maram, James van Howe, Deming Kong, Francesco Da Ros, Pengyu Guan, Michael Galili, Roberto Morandotti, Leif Katsuo Oxenløwe, and José Azaña. "Frequency-domain ultrafast passive logic: NOT and XNOR gates". In: *Nature Communications* 11.1 (2020), p. 5839.

[7] Rodney S Tucker and Kerry Hinton. "Energy consumption and energy density in optical and electronic signal processing". In: *IEEE Photonics Journal* 3.5 (2011), pp. 821–833.

[8] David AB Miller. "Attojoule optoelectronics for low-energy information processing and communications". In: *Journal of Lightwave Technology* 35.3 (2017), pp. 346–396.

[9] Youngeun Kim, Joshua Chough, and Priyadarshini Panda. "Beyond classification: Directly training spiking neural networks for semantic segmentation". In: *Neuromorphic Computing and Engineering* 2.4 (2022), p. 044015.

[10] Claudio Zandron. "An Overview on Applications of Spiking Neural Networks and Spiking Neural P Systems". In: *Languages of Cooperation and Communication: Essays Dedicated to Erzsébet Csuhaj-Varjú to Celebrate Her Scientific Career* (2025), pp. 267–278.