MATLAB MODEL FOR SPIKING NEURAL NETWORKS

IVAN BOGDANOV RADU MIRSU VIRGIL TIPONUT
Applied Electronics
"POLITEHNICA" University of Timisoara
Timisoara, Str. Vasile Parvan, Nr.2
ROMANIA
ivan.bogdanov@etc.upt.ro radu.mirsu@etc.upt.ro virgil.tiponut@etc.upt.ro

Abstract: - Spiking Neural Networks are the most realistic model compared to its biological counterpart. This paper introduces a MATLAB toolbox that is specifically designed for simulating spiking neural networks. The toolbox includes a set of functions that are useful for: creating and organizing the desired architecture; updating stimuli signals, adapting synapses and simulating the network; extracting and visualizing the simulation results.

Key-Words: - spiking neural networks, neural modeling, MATLAB modeling, neural synchronism

1 Introduction

Spiking neural networks are of the last generation. Compared to more traditional models, spiking models have spike emitting outputs rather than continuously varying outputs. This change comes as a generalization of the coding techniques and allows precise spike timing to be utilized as information carrier. The older generations of networks assumed that all the information is coded within the rate of the spike train and therefore time averaging was allowed without any information loss. This averaging reduced all spike signals to continuous variables.

2 Problem Formulation

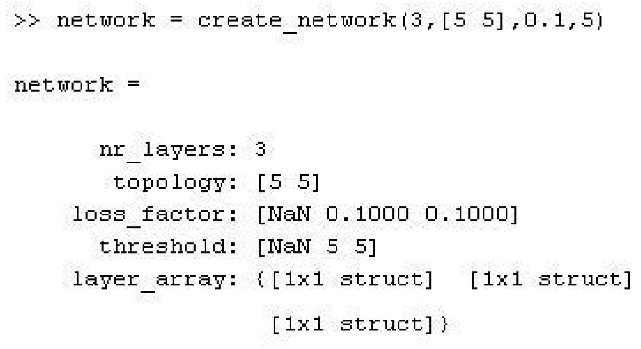
Proper investigation of coding methods and network functionality requires a simulation environment accompanied by appropriate tools useful for results processing, interpretation and visualization. Such an environment is MATLAB, which preferred by most scientists due to its vast library of functions and toolboxes which are oriented towards scientific modeling and experimentation. Unfortunately, MATLAB does not have an existing toolbox that is directly suited for simulation of spiking neural networks. This paper proposes to introduce a MATLAB toolbox specifically designed to simulate spiking neural networks. It also introduces a few functions that are useful for visualizing results.

3 Model Objects

The model uses structures as functional objects. This approach allows a direct association between software modules and actual parts of the network architecture, easing the task of extracting and interpreting simulation results. It also facilitates to combine fragments of code that were written independently.

3.1. Network Object

The main object, which comprises the entire architecture, is the "network object". Figure 1 describes the creation of such an object together with all its internal variables. The network architecture is volumetric being organized on layers, each of them having a bi-dimensional topology. In this example the dimensionality of the network is 3x5x5.



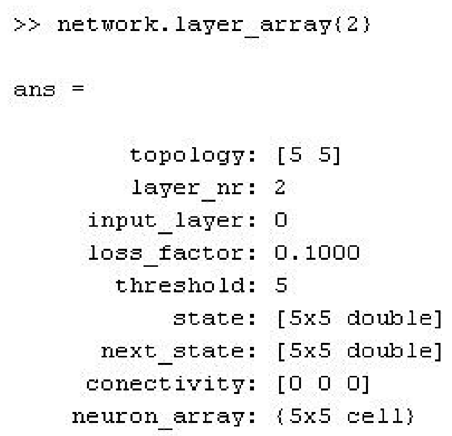
```
>> network = create_network(3,[5 5],0.1,5)

network =

      nr_layers: 3
       topology: [5 5]
    loss_factor: [NaN 0.1000 0.1000]
      threshold: [NaN 5 5]
    layer_array: {[1x1 struct]  [1x1 struct]
                  [1x1 struct]}
```

Figure 1. Network Object

An important variable of the network object is the layer array. This array has a number of entries equal to the number of layers in the network, each entry being occupied by one layer object. Other variables memorize network constants such as the loss factors and thresholds.

3.2. Layer Object

The content of the layer object is illustrated in figure 2. Variables like topology, loss factor and threshold are redundantly stored and are the same as specified by the network object. Theoretically, every neuron of all layers can be connected to any neuron of any layer. In practice, however, only some of these connections are valid and most are zero. The connectivity vector holds Boolean variables that show to which layers is the current layer connected to. This is useful because the simulator will examine this vector and will skip propagating spikes between layers with null connections, decreasing simulation time significantly. In this example the connectivity vector holds only zeros as the network has not been initialized.



```
>> network.layer_array{2}

ans =

        topology: [5 5]
        layer_nr: 2
     input_layer: 0
     loss_factor: 0.1000
       threshold: 5
           state: [5x5 double]
      next_state: [5x5 double]
     conectivity: [0 0 0]
    neuron_array: {5x5 cell}
```

Figure 2. Layer object.

The simulator, as it will be described in paragraph 4, is time based and has as output the spiking activity of the network. Because the spikes will only hold information in their timing or rate and not in their amplitude the spike activity of one neuron can be represented by a Boolean time-trace with ‘1’ denoting a spike. Variable “state” of the layer object is a matrix of the same size as the layer topology and holds the spike activity of the layer at the current time instant. Every entry in this matrix corresponds to the output of one individual neuron. This means that building the spike trace of one neuron is done by recording the value of one matrix entry at every time instant of the simulation.

The layer object contains a secondary state variable named “next_state”. The reason why two state variables are needed comes from the functional difference that exists between a biological neural system and a computer system. All the neurons of the biological system operate in parallel while the computer code is executed sequentially. This means that any change in the output of one neuron will affect the simulation of the next neuron. This is incorrect since normally the two neurons are executing in parallel. In order to preserve the parallel functionality of the network two state variables are used. All neuron read signals from the “state” variable while their outputs write signals into the “next_state” variable. Until all the neurons are executed the simulator considers time to remain unchanged. When the “next_state” variable is complete the simulator uses it to overwrite the “state” variable and time is advanced. This technique is illustrated in figure 3.

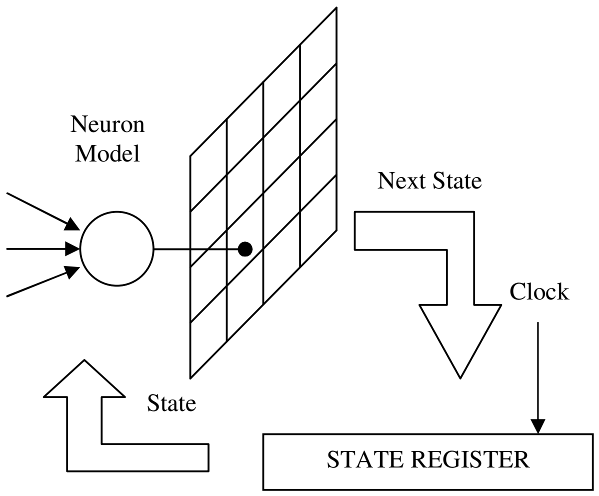


Figure 3. Updating Network State

Variable “input_layer” of the layer object denotes whether this is the first layer of the network or not. Input layers are different from the other layers because all of their variables are null except for the “state” variable which is used store the input signals. The input signals can either remain constant or be changed during the simulation.

The “neuron_array” variable is an array with the same topology as the layer object. This array holds all the neuron objects.

3.3. Neuron Object

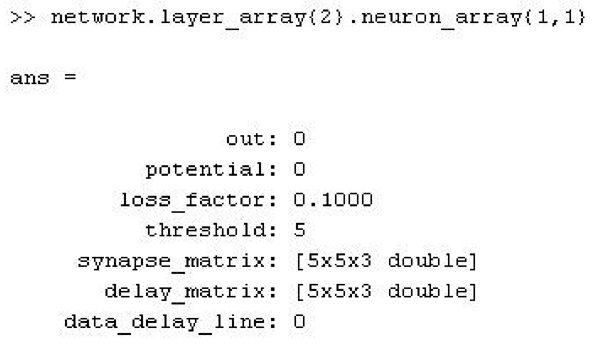
Each individual neuron uses and integrate and fire model. This means that neuron i integrates all incoming spikes as membrane potential $p_i(t)$. The integrator is lossy with factor *k_loss*. When the potential reaches threshold Th it is discharged to zero and a spike of unitary amplitude is emitted at the output $u_i(t)$. Spikes coming from neuron k to neuron i cross a synapse which produces gain G_{ik} and delay D_{ik}. This type of behavior is modeled by equation (1).

$$p_i(t) = \int_{ti}^{t} \sum_{k=1}^{K} G_{ik} * u_i(t - D_{ik}) - k\_loss * p_i(t) \qquad p_i(t) \leq Th \qquad (1)$$

$$p_i(t) = 0, \quad ti = t \qquad p_i(t) \geq Th$$

Additional information on integrate-and-fire models can be found in [1], [2] and [3]. The neuron object is illustrated in figure 4. The "potential" variable holds the neuron potential that is computed according to (1). The values for G_{ik} and D_{ik} are stored in the "synapse_matrix" and "delay_matrix" variables.

Spikes propagate between neurons with different delays. This means that in order to compute spike influence on present potentials a history of the spike activity needs to be recorded. This history needs to be at least as long as the largest delay value. The model presented in this paper is organized such that each neuron keeps track of all the spikes that will affect its potential at some time in the future. This is done by placing a "data_delay_line" vector in each neuron object. When a neuron object is executed its "synapse_matrix" is multiplied with the activity matrix at that time instant. The activity matrix is obtained by concatenating all the "state" variables found in the layer objects. This operation produces a matrix that contains all the spikes affected by the appropriate gains. The spikes are then placed inside the "data_delay_line" at a position according to its associated delay. At each time step the delay line is shifted and the first entry is used for computing the new neuron potential.



```
>> network.layer_array{2}.neuron_array{1,1}

ans =

             out: 0
       potential: 0
     loss_factor: 0.1000
       threshold: 5
  synapse_matrix: [5x5x3 double]
    delay_matrix: [5x5x3 double]
 data_delay_line: 0
```

Figure 4. Neuron Object

Variable "out" is Boolean and represents the emission of a spike. This variable is directly mapped to the "next_state" variable of the layer object.

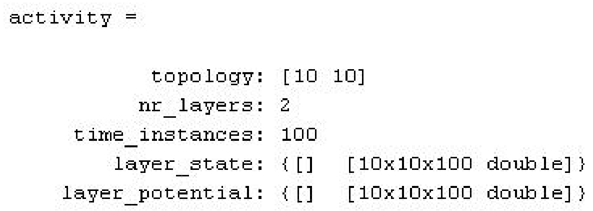
4. Model Functions

4.1. Simulation Functions

Simulating a network is done by calling function "simulate_network". The function takes as input parameters the network object that is to be simulated and the simulation duration in seconds. The simulation is performed by calling repetitively subroutines like "advance_time", update_neuron" and "compute_next_state" and finally returns two objects as output.

One output object is the post-simulation network. Therefore, a comparison between the internal state of the initial network object and the final network object can show the influence that the external stimuli has had on the network within the time span of the simulation.

The second output object is an activity object. During the simulation, time traces of the neural outputs and membrane potentials are recorded. These traces are organized in multidimensional vectors that are stored inside the activity object. The activity object is very useful because it holds data that completely characterizes the behavior of the network during the simulation. An instance of such an object is illustrated in figure 5. The neural activity has the same topology as the network that generated it.



```
activity =

         topology: [10 10]
        nr_layers: 2
   time_instances: 100
      layer_state: {[]  [10x10x100 double]}
  layer_potential: {[]  [10x10x100 double]}
```

Figure 5. Activity Object

Several other functions are offered for creating and initializing new objects and also for uploading stimuli and adapting synapses.

4.2. Visualization Functions

The activity object contains the time traces recorded from the membrane potentials and neural outputs. Most often, the easiest way to interpret this data is by visualization.

4.2.1. Visualizing Neural Time Traces

The most straightforward way to visualize is by plotting the actual time traces. For this purpose function "display_activity" can be used. The function accepts several variables which influence the display mode. The first variable for this function is the activity object that supplies the data. The second variable will specify the number of the neural layer that will be plotted. If this variable is omitted all layers will be plotted in several distinct windows. The third variable is a string that will select between displaying the neural spiking output activity or the membrane potentials. Figure 6 shows the output of this function for both cases. The fourth variable is optional and allows visualization of a sub-region of the neural array.

Visualization of the time traces is not very useful when it comes to the interpretation of the data. However, the function described above can be very useful during the debugging period of a project. Subtle effects created by different network parameters can be spotted on the time traces and so several problems can be avoided or fixed.

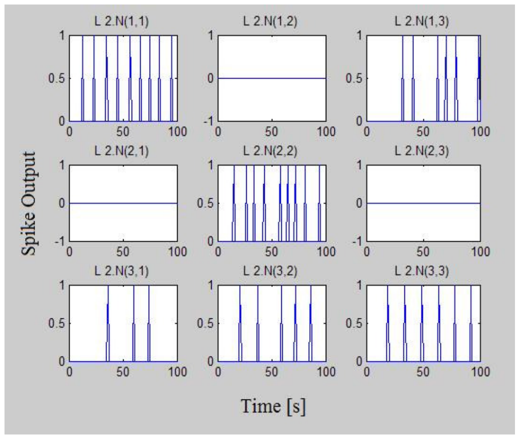


Figure 6a. Layer Activity. Spike Output

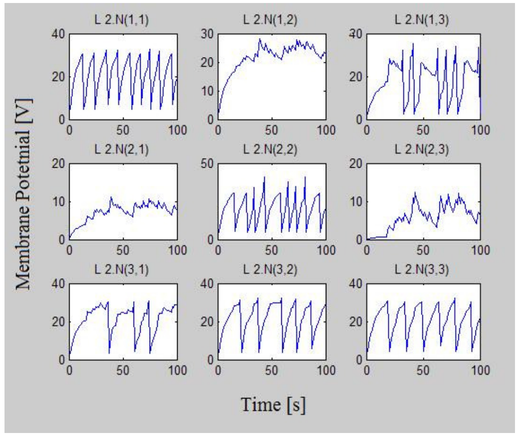
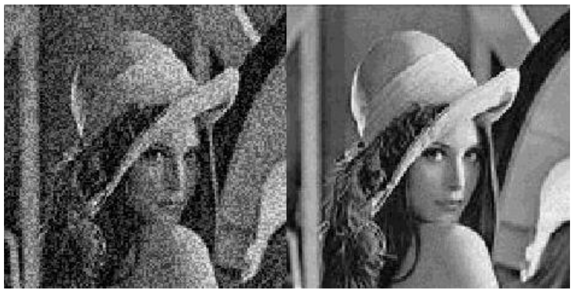


Figure 6b. Layer Activity. Membrane Potential

4.2.2. Visualizing Neural Spike Rates

When the neural array is large visualizing by plotting time traces can be very difficult or even impossible. For this purpose the function "display_rate" was developed. This function computes the rate of the spike train for each neuron and then maps these rates onto a black-white image. This permits easy visualization of large arrays. The function's input variables allow selecting the time at which the average rate is computed and also the size of the averaging window. Figure 7 illustrates the output of this function. Figure 7a displays the image that is fed to the network as input stimuli. The image only presents a snapshot of the input stimuli which will continuously change during the simulation as an effect of the time-varying white noise.



7a. Noisy Image 7b. Average spike rate

Figure 7. Visualizing activity rate

Figure 7b illustrates the output of the neural rate function. Because the noise is time-varying it is rejected by the filtering effect of the integrate-and-fire neurons.

4.2.3 Visualizing Neural Synchrony

Another important aspect in neural activity analysis is neural synchrony. For example, at image processing and shape recognition, neural synchrony can be used in the segmentation stage. Observations among biological systems have led to the idea that neurons processing pixels belonging to the same object tend to fire at the similar rates and also in synchrony. With a design like the one presented in the previous example, having point-to-point synaptic connections between pairs of neurons and pixels, neurons belonging to the same object will fire at the close rates. This is based on the assumption that the two pixels that are sourcing the neurons will have similar values. However, due to different initial conditions or system noise, these neurons will fall out of phase. Synchrony can still be achieved by using lateral connections in the proximity of each neuron. This way spikes generated by one neuron can force neurons that are almost ready to fire to generate a spike ahead of time and thus inducing synchrony. Additional information on neural synchrony can be found in [5]. For the purpose of visualizing neural synchrony function "display_synchrony" was developed. The function expects three variables as input. The first one is the neural activity object that is supposed to be analyzed. The second one is the time value at which synchrony is evaluated. The third variable is a synchrony threshold that will be explained shortly. The function uses a fraction variable to denote how well two neurons are synchronized with 0 meaning completely out of phase and 1 meaning fully synchronized. Assuming that the neurons are firing at the same rate full de-synchronization occurs when the time distance between spikes is half of the period. The function builds a map of synchrony between each neuron and its neighbors. Then, by comparing the synchrony levels with the synchrony threshold, decides whether the two neurons are sufficiently synchronous to be considered as belonging to the same object. This way segmentation is performed. Lastly, the function maps groups of synchronized neurons to different colors and plots the result. Figure 8a, 8b and 8c illustrates the output of this function at different times along an activity object. The activity object was obtained by simulating a network model similar to the one described above that was sourced with a grey scale image comprising of three objects, each at a different grey level.

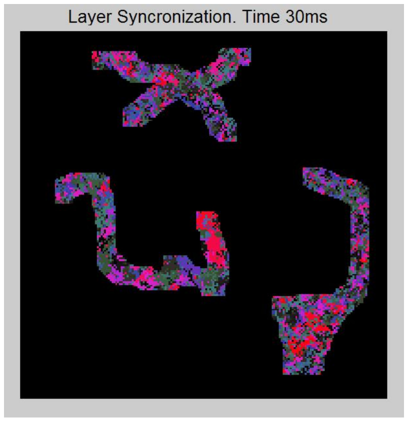


Figure 8 a). Layer Synchrony at t = 30 ms.

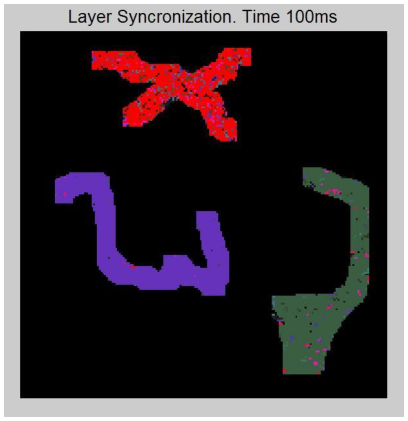


Figure 8 a). Layer Synchrony at t = 100 ms

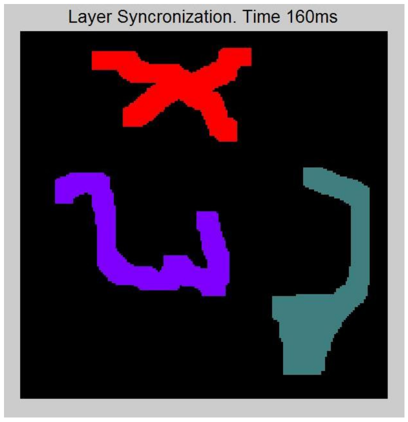


Figure 8 a). Layer Synchrony at t = 160 ms
Synchrony Threshold = 0.8

It can be observed that initially the neurons are unsynchronized and so the image is segmented into large number of objects. At 100ms large groups of neurons become synchronized. After 160ms all neurons of the same object are fully synchronized (above the synchrony threshold which in this case was 0.8).

5. Conclusions

This paper presents a MATLAB toolbox for spiking neural networks. It starts with the mathematical model for an individual neuron, continues with the organization and functionality of the network architecture and finalizes with the description of some simulation and visualization functions. Future work will be oriented towards developing additional functions for data visualization and analysis. Also, upgrading the model such that it can be run on a multiprocessor platform is of interest as it would allow simulating significantly larger networks.

References:

[1] Wulfram Gerstner, Werner M. Kistler, *Spiking Neuron Models*, Cambridge University Press, 2002.

[2] Eugene M. Izhikevich, Simple Model of Spiking Neurons, *IEEE Transactions on Neural Networks*, Vol.14, No.6, 2003, pp. 1569-1572.

[3] Sebastian A. Wills, *Computation with Spiking Neurons*, PhD Dissertation, University of Cambridge, 2004.

[4] Duane Hanselman, Bruce Littlefield, *MATLAB The Language of Technical Computing*, Prentice Hall, 2001.

[5] Talia Konkle, *Image Segmentation Using Neural Oscillators.*