Practical Assignment 3
Spiking Neural Networks

Wouter Kruijne

November 29, 2017

- Complete the exercises below *individually*.
- Write your answers down in a pleasantly formatted document
- Some of the exercises may require you to do some coding
- Some questions are marked *BONUS* **for a reason**. I suggest that you skip them on your first pass!

Hand in your assignment as a zip-file that contains your report, and for some assignments also python code or other relevant files. Name your zipfile `[LASTNAME]_[studentnr]_assignment1.zip`, and hand it in via Canvas.

The deadline for this practical is **Friday the 15th of December, 23:59** .

Introduction

For the first part of the practical, you are going to explore some of the properties of LIF- and Izhikevich neurons, and their synapses.

The second part is an 'open' assignment: here, you are free to construct your own network with spiking neurons and have this network solve a problem or simulate a psychological process.

As mentioned in class: the somewhat late deadline is intended to give you time to 'get creative' for the second part of this assignment. I strongly encourage you to finish the questions in part 1 earlier on, and make sure that the assignment does not interfere with your study time!

Overview of the files

In the previous assignments, the source files were made to be 'standalone' programs. Here, the files are organized as follows:

- `neurons.py` is a file that specifies classes that can be used to construct two types of neurons: the LIF neuron, and the Izhikevich neuron, both discussed in the lecture. Both neurons have 2 functions that define their behavior: `step()` simulates a single simulated timestep according to the neurons definition; `get_V()` and `spike()` are functions that can be used to determine the neuron's membrane potential, and whether it is currently spiking (True/False).

  Both neurons make use of an abstract parent class `Neuron`, which is a "template" class: it defines some functions that are shared among its children (Izh_neuron and LIF_neuron), but does not implement a functioning neuron by itself.

- `synapses.py` defines different types of input a neuron might have. It similarly has a parent class, with some core functions, and child classes to model different types of input. That is:
  - `Synapse` is the template class.
  - `Continuous_synapse` simulates experimentally clamped input, i.e. $I_{\text{ext}}(t)$ in the equations of all neuron models.
  - `Neuronal_synapse` simulates the connection between 2 neurons. It will transmit a current, only when the pre-synaptic neuron emits a spike.
  - `Poisson_synapse` A common way to simulate 'noise', without modeling presynaptic neurons. Simulates, stochastically, spikes arriving with a certain predefined rate.
- `neuron_tester.py` is a file that is predominantly used in the first half of the assignment, to explore the behavior of these model neurons in detail.
- `neuron_tester.py` is a file that is predominantly used in the first half of the assignment, to explore the behavior of these model neurons in detail.
- `network.py` is a file predominantly used in part 2: it reads your own file `networkfile.py` to define a network of neurons and synapses, and uses it to simulate a single run with the defined network.
- For the simulations in part 2 of the assignment, you may want to use `network_simulator.py`. This class implements some 'smart' ways of communicating with the `Network`-class, and allows you, for example, to run multiple 'trials', store the results in a csv-table or read previous data from a csv-table

Make sure that these source files are downloaded into the same directory, and that this directory is your *working directory*!

Part 1: Introductory Questions

Question 1 – Testing Neurons

Have a look at the function `two_neurons()` in the file `neuron_tester.py`. It simulates 2 neurons, run for 800ms, simulating 1ms at each time step. Both neurons are connected to the same clamp input, that injects a constant current into the cell.

Before you run it and look at the output plot, make sure that you understand each step taken in the code at least conceptually.

a) Run the function `two_neurons()` and put the resulting graph in your report. Which of the traces belongs to the Izh-neuron, and which to the LIF neuron? How can you tell from the shape of the traces?

b) Change the weight (`w=...`) of the clamp-synapse to 0.1, 0.25, 0.5, 0.75 and 1.5. Can you say something about the 'sensitivity' of both neuron types, i.e. to what extent is their spiking rate dependent on the magnitude of the input?

c) Spiking neuron models are usually defined using differential equations. Mathematically, these equations describe how the change of the system depends on its current state and on other inputs. In the simulations, we evaluate the equation at every time tep, and then add the outcome to the previous state of the neuron. This method of evaluating differential equations is called *Euler integration*. In the 'step' function of both neurons (in `neurons.py`), you can find, for both neurons, the code that updates the membrane potential. Find those lines, for both neuron types, and put the code in your report.

d) Why is it important that the right-hand side of this implementation is multiplied by `dt`? (Hint: what happens if `dt` is, for example, 0.5) ?

Now, move on to the next function: `izh_tester`. This function defines 1 Izh-neuron. You can pass its 'type' as an argument by passing the letters $[A--F]$. (Try them out!)

e) Which of the letters corresponds to the 'phasic burster' ? Run the neuron and plot the resulting spike trace. What characteristics of the spike trace make this neuron a 'phasic burster' (as opposed to being a 'tonic burster' or a 'phasic spiker')?

Neurons usually do not have constant clamped input currents. Typically, neurons are connected to many other neurons, which continuously affect the membrane potential in a somewhat unpredictable manner.

f) Copy the code from `two_neurons()` into the body of the function `two_neurons_poiss()`, and change the `clamp` input into a `poiss` input to simulate stochastic background noise. You should construct a poisson input as follows:

```
poiss = synapses.Poisson_synapse(
    firing_rate = 0.5, w = 0.75, onset = 200, offset = 650)
```

Make sure to (1) connect the synapse to each neuron; (2) connect the synapse to both neurons; (3) make sure to update the synapse at each timestep that you update the neurons. Put the resulting spike trace in your report.

g) now, decrease the input strength `w` of this synapse to `w=0.2`. How many spikes are emitted by the Izh-neuron? (approximately – it may vary across simulations)

h) The sluggish membrane only moves to the threshold (and spikes) when multiple inputs reach the neuron within a short time period. Explain how the sluggish membrane can act as a filter, to separate signals from noise in a neurons input.

Finally, look at the function `connecting_neurons()`. Again, this is the same setup as before, but now an Izh-neuron receives stochastic (poisson) input, and is connected to the LIF-neuron, using a 'neuronal-synapse' object:

i) Run the code and describe what happens to the LIF-neuron when the Izhikevich-neuron fires.

j) Run the code again, but using `izh_type = 'C'` in the initialization of the Izhikevich-neuron. Describe what happens now, and the difference with the previous question.

k) Implement the same effect (as in j) by having three, rather than just one Izh-neurons (type 'A') that all project to the same LIF-neuron. Do this by adapting the code in `connecting_neurons()`, and include the new code in your zip file.

   To do this, you will have to:

   1. Define three new neurons (e.g. 'nrna','nrnb','nrnc')
   2. Connect these neurons to the same poisson input (as was done for nrn1)
   3. Connect the LIF neuron (nrn2) to the three new neurons.
   4. Make sure all new neurons and synapses are updated during the for-loop.

Part 2: Making a Network

In this part of the assignment, we will first use the `Network` class to see whether we can mimic the behavior of a 'decision maker' (comparable to the LBA) with a spiking network; after that, it is up to you to get creative, and build a network that performs any task you like.

We will mimic the LBA using the class in `network.py` and the `networkfile.py` file. The code you put in the latter file determines what happens within the network see Figure 1.

But first, have a look at the constructor of the `Network` class:

1. First (l. 32-48), 2 input synapses are constructed. For now, we leave `rand_input` set to False, and the assumed stimulus is always presented to input number 0. This is done by turning both input synapses at into Poisson synapses, onset at 300ms, of which `in0` has firing rate 0.75, and `in1` has firing rate 0.

2. Then (l. 51-52), two output nodes are constructed. Whenever any of those neurons exceeds a spike rate of 1sp/10ms, averaged over the last 300 time steps, the neuron is said to have made a decision, and the simulation will end. Otherwise, it will simulate up to 5000ms, and it will note that no decision was made.

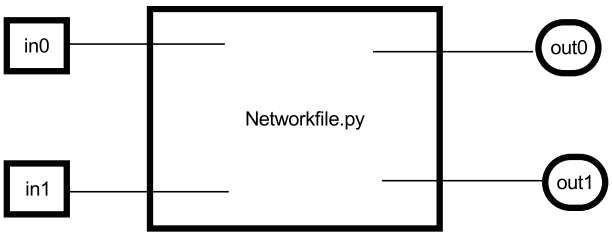

Figure 1: Wiring of in- and outputs to the network file

3. In lines 54-72; the class tries to first compile (i.e. translate into machine-language) the code in the `networkfile.py`-file, and then run it. If you make a coding mistake in your networkfile, the program will crash at this point.

In `networkfile.py`, you can construct a simple network, as long as you keep the following in mind:

- At the end of your script, make sure the newly generated nodes and synapses of your networks are added to the `nodes`- and `synapses`-lists respectively. Otherwise, they will be completely ignored by the `network`-class
- Make sure you always *append* things to this list, and that it is never overwritten; otherwise the input synapses and/or the output nodes might get lost.
- Whenever you generate a node or synapse object, make sure you use the constructor to generate a *new* one. Only unique nodes and synapses are allowed (so e.g. `nodes += [my_neuron]*100` will get you only one copy of my_neuron!)
- the objects `in0, in1, out0, out1` are accessible, so you are free to connect them to each other or to any new node you create.
- You may create as many nodes and synapses as you wish, but keep in mind that a complex network will be more time-consuming to run, as many objects are updated every time step.
- To explore what neurons and synapses are doing, you may create them with the additional arguments `set_record=True` and `name = 'NiceName'`. Set record will, after the simulation, plot the potential V_m of the requested neuron (or the current I of the requested synapse).

To make things somewhat harder (but more realistic!) for you, every node you add will inherently have a Poisson-synapse in itself, randomly firing to simulate background noise. You may not delete this additional input!

You will first explore how to use the networkfile in two examples. To start, in `networkfile.py`, uncomment the first example. Make sure you understand

what the resulting network looks like, how many neurons there are, and how they are connected (perhaps make a drawing for yourself). Run the example, either by running `network.py`, or (probably better!) by opening an IPython-shell and running the following code step-by-step:

```
from network import Network
# reading the networkfile:
with open('networkfile.py') as nwsfile:
    nws = nwsfile.read()
# generate a Network-object
net = Network(nws)

# Run the simulation:
desc, rt = net.simulate()
net.make_plots(trace=True, im=False, tmax=rt)
```

Question 2 – Exploring the networkfile

a) Both neurons fire in response to the random background noise input, and Neuron B sometimes fires in response to neuron A due to their synaptic link. However, no decision was made. Can you explain why no decision was made? If you don't understand this question, make a drawing of all the neurons and synapses that are implemented in the network, taking Figure 1 as a starting point.

b) comment Example nr 1, and uncomment Example nr 2; run the network again using the statements above, but for making the plots now set: `net.make_plots(trace=False, im=True, tmax=rt)`. You should get a very colorful, yet noisy plot of what all the nodes (how many are there?) in your network have been doing – the x-axis marks the time in milliseconds.

   Zoom in at the interval 0-600ms, and explain how the plot should be interpreted. Also, describe the transition at 300ms, and why that is.

c) We can introduce some neuronal synapses and have our network make a decision, by adding the following statement to the for-loop in example 2:

```
s = Neuronal_synapse(w = 3.5, pre = n, post = out0)
```

   Run this a couple of times, and output `desc` and `rt`. The resulting RT does not vary very much. Why?

Question 3 – Making a decision maker

N.B. : From this point on, I really don't expect everyone to get equally far with these questions. If you get really stuck, or completely lose interest in modeling the decision making task, feel free to move on to the next question. What's

most important, in that case is that you manage to write down what you've done so far and why.

The last question left you at a point that our network made a decision after the stimulus was presented, but this did not yield a reasonable 'spread' in reaction times. The goal in this assignment is to create a network that does this. See whether you can 'slow down' the network a bit, have it gradually accumulate evidence, and include enough noise in the input to mimic the characteristic 'skewed' distribution of decision times.

To get a distribution of RTs you need to run the network multiple times, and save the results. The file `network_simulator.py` can help you do this. Read this class, and see how you can use it (especially its `simulate()`-function) to get insight in the distribution. Carefully read its example usage illustrated in the 'main'-code at the bottom and see to it that you understand what each statement does. The commented statements at the very bottom illustrate how you can save your results to a csv-table, which you can read at a later moment.

Some questions to get you started:

1. First, try to get a spread of 'RTs' of several hundreds of ms, containing several dozens of trials. Include the resulting 'rug' plot in your report, and if possible, also the csv-file of resulting trials.

2. See whether you can now design your network so that it makes a decision regardless of which stimulus was on and which was off. Can you include enough noise that you sometimes end up with the incorrect decision?

3. See whether you can 'ground' different parameters of the LBA into neurobiology. For example, what happens when you raise the number of neurons connecting the input and the output? What happens when you introduce more and stronger background noise?

Question 4 – Get creative!

In this question, anything goes. See whether you can design networks that can perform any these tasks, a different task/model that you like, or see whether you can improve my code to make the simulations more closely resemble neurobiology.

You may design whatever you wish, but some possibilities are given below. Some of those might have you alter the code in `network.py` or other source files; if you feel unconfident about some of the changes you'd like to make (or you feel you've completely broken something) just email/ask me! As always, I don't want you to get stuck on the programming-aspect of this assignment

Here are some possibilities:

1. Design a spiking multilayer network that makes an XOR decision (you have to alter the input signals in `network.py`)

2. Design a network that classifies whether your favorite letter from the braille alphabet was presented.

3. Design a network that classifies whether a vowel was presented, using the braille alphabet.

   N.B. For suggestions 2 and 3, you'll have to change the number of inputs into the network, and add the new input synapses to the `synapses`-array in the `_network_spec_header` (see lines 13–18 in `network.py`).

4. Alter `neurons.py` to add a LIF-neuron with a more intelligent type of after-spike hyperpolarization. Before using your neuron in a networkfile, explore its behavior using clamped input, e.g. by using the test-code in `test_code.py`, and make sure its membrane potential fluctuates plausibly (e.g. between -80 and +40mV).

5. Implement a network with short term memory using sustained firing. Whenever a stimulus is presented, then disappears, make it so that a pool of neurons remains active for at least 1500ms.

6. Create a neuronal synapse with spike-timing dependent plasticity (STDP, that is LTP or LTD) cf. the rules defined in FCNS (p. 100) and discussed in class on Friday. Perhaps you can have it 'learn' something, or simulate the experiments presented in section 4.2.1.

7. Create a neuronal synapse with synaptic potentiation, and show that it can be used to temporarily store something (Mongillo, 2008; discussed in class on Friday).

Good luck!