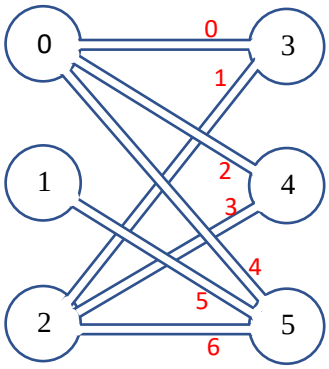
Lab #3 (Computing weighted sums)

In this lab, we finally move beyond single cells. We'll use BITSEY to run a simulation of multiple cells connected by gap junctions. The goal will be to compute weighted sums, with each gap junction determining how much weight to give an input cell.

We will build the following network.



The input layer (cells #0 to #2) is drawn on the left; the output layer (cells #3 to #5) is drawn on the right. Seven gap junctions (numbered 0-6 in red) interconnect the two layers. The goal is that cells #4 through #6 each compute a particular weighted sum of the input cells:

cell #3: $V_{mem,3} = .5V_{mem,0} + .5V_{mem,2}$

cell #4: $V_{mem,4} = \frac{5}{6}V_{mem,0} + \frac{1}{6}V_{mem,2}$

cell #5: $V_{mem,5} = \frac{5}{7}V_{mem,0} + \frac{1}{7}V_{mem,1} + \frac{1}{7}V_{mem,2}$

You can use the same BITSEY files as last week. This time, we'll be using and modifying the *setup_lab_QSS_weighted_sum*() function. You have a skeleton version of the function, for which you will fill in the details. The existing code

- instantiates the cells and the gap junctions
- sets ion-channel conductances to create V_{mem} for the input-layer cells
- programs gap junctions #0 and #1 to build a weighted sum in cell #3

Your code must then program gap junctions #2-#6 to build weighted sums in cells #4 and #5. You should keep the GJ conductances roughly the same magnitude as those driving cell #3.

You'll note the variable *GJ_scale*. Once you have the code working, this will let you easily scale all of the gap junctions to have larger or smaller conductance, and see how this affects the workings of our little "neural network."

Note also that we've set *sim.Dm_array*[:,3]= 0, thus setting all of the ion-channel conductances to 0 in cell #3, essentially saying that the output cells have no ion channels. We did this because gap junctions are bidirectional – the output-layer cells are just as capable of affecting the input-layer cells as vice versa! Having information flow in "reverse" is not our goal; one simple means of avoiding it is to not put any ion channels in the output-layer cells, which essentially prevents the output from driving the inputs.

Finally, note that Bitsey has a graphing facility called *pretty_plot*(), which makes graphical plots of a network such as this one. We will use it for the lab's results. It is already in the main code, but commented out; you should uncomment it.

You should perform three runs. First, with *GJ_scale*=1. Then a second run with *GJ_scale*=0.1, and a third with *GJ_scale*=10. Run each simulation for as long as it takes to reach quasi steady state (QSS), based on looking at graphs of V_{mem} vs. time (i.e., when all V_{mem} plots become essentially flat). Then, for each simulation, use *pretty_plot*() to make a picture of the final, end-of-sim cell V_{mem} values. Turn in the three pretty plots as part of the report, and one *main.py*.

Also, please include the following information into your report. First, just the results. For each value of *GJ_scale*:

1. How close did your cells come to computing the correct answer (i.e., V_{mem})?
2. How long did it take to do the computation (i.e., for V_{mem} to settle to roughly its final QSS value)? Again, you can judge this roughly by eye.
3. Did the computation affect the input V_{mem} values? I.e., was the final V_{mem} for cells #0-2 affected by your computation? Note that if we had only build cells #0-2, their final V_{mem} values would have been 44mV, 0mV and -66mV respectively.

Next, the *why*. I.e., explain the results that you just reported. Why did varying *GJ_scale* have the effects that you just noted?

- You should use an equivalent-circuit model similar to what we just discussed in class: represent each input cell with a single battery and resistor, and represent each GJ with one resistor (remember: the reason that the GJs don't need an internal current source is that all of our cells have roughly the same ion concentrations, and so there is no diffusion through GJs). Since the output cells have no ion channels or ion pumps, you can represent each of them as a simple junction.
- To explain what the final voltages in the output cells should be, you should talk about voltage dividers and do a small amount of algebra to compute the output voltage of our equivalent circuit (noting how resistors in series behave).
- To explain why V_{mem} of the input cells changed, you should mention voltage drop across the resistors in the equivalent circuit.

Please turn in two files: your main.py and the report (with the plots and the answers to the questions).

Optional part 2

In this lab so far, we've managed to build a weighted sum. While that's a big part of an artificial neural network, it's not everything. The next steps are building an activation function and building multiple layers. Part 2 (which is optional and is due two weeks after the main lab) should achieve both of these.

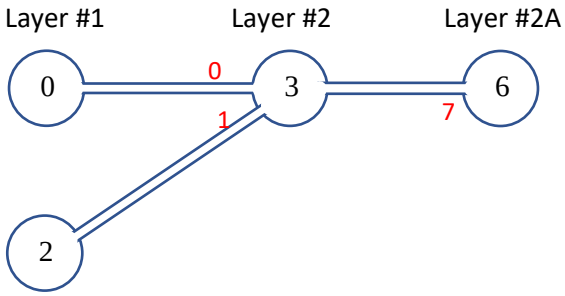
First, the activation function. Remember that this takes the weighted sum z and computes $a=f(z)$ for some function f, where f is a useful nonlinear function (e.g., $f(z)=1$ if $z>0.5$ else $f(z)=0$). How might we compute this?

Let's define our input layer (cells #0-#2) as layer #1, and our output layer (cells #3-#5) as layer #2. We've already built layer #2's V_{mem} to be a weighted sum z of layer-1 cells. If we want to compute some $f(z)$,

there are a few obvious choices. For example, we might use layer #2's V_{mem} to gate ion channels or to gate GJs.

How might we use gated ion channels? Could we somehow use them to have layer #2's V_{mem} compute $f(z)$? This seems quite challenging, since we're essentially building a function where layer #2's V_{mem} is both the input and the output of f! This seems unlikely to work well.

How about using gated GJs? We would have a new layer 2A, connected by GJs to layer #2:



GJ #7 would be gated by cell #3's V_{mem}; it would work to let ions flow from cell #6 to cell #7 (and inadvertently, to some extent, in the reverse direction). But how exactly? One way is as follows.

Let the layer-2 cells have the code similar to

```
A = sim.ion_i['A'];
sim.decay_cells[A] = 2          # 1/s
sim.gen_cells[A,3:6] = 10       # moles/m^3 per s
```

What does this mean? "*A*" is a new ion. We will assume it to be a protein; all proteins decay away over time. We've set *A* to decay at the rate given by $\frac{d[A]}{dt} = 2[A]$. We have further set that cells #3 through 5 (i.e., the cells in layer #2) create *A* at a constant rate of 2 moles/(m^3·s). If these cells were not connected by GJs to other cells, then their [*A*] would eventually stabilize to 5 moles/m^3 (i.e., the point where generation and decay are both 10 moles/(m^3·s), and hence balance each other).

If GJ #7 is off, then cell #6 will indeed be cut off from its neighbors. Since there is no generation of *A* in cell #6, any *A* in cell #6 will eventually decay, resulting in [*A*]=0. If, however, GJ #7 is on, then [*A*] in cell #6 will stabilize to some nonzero value, balancing the influx (generation in cell #3 and traveling through GJ #7) against decay. We can then give cell #6 ion channels whose conductance depends on its [*A*], which will effectively make $V_{mem,6}$ a function of $V_{mem,3}$ as desired (the last half dozen or so slides in the slide set on worms, 4_bioelec_4_worm.pptx, describe how to do this).

Going still further

If you really like devising ways to implement neural networks with biology, you can take this still further and turn it into a final project. Here's a completely different way to implement a neural network using biological parts. It doesn't use bioelectricity at all, but rather uses diffusion of a charge-neutral ion to implement the weighted sum and also the activation function. It is thus much slower than the method we've built. On the other hand, it is arguably substantially simpler.

In this method, each cell will represent a number not by its V_{mem}, but by [*B*] for some charge-neutral ion *B*. Just as before, we will build an input layer #1, and connect it to a layer #2 that computes a weighted sum. This time, however, the weighted sum is simply created by diffusion (and moving appreciable

quantities of *B* by diffusion takes time). Since *B* is not charged, V_{mem} does not affect it. Great – we have a weighted sum.

Next comes the activation function. We will use a gene-regulatory-network (GRN) buffer to both implement *f*() and to drive a new neutral ion *A*. So [*A*] will represent the output *f*(*B*) in layer 2A. That's it – you are now free to repeat the process with as many new layers as you like.