The Hopfield-ish energy



Note in Scellier’s implementation, he treats i and j as though they’re layer numbers (because that implementation assuming layers only talk to other layers), but for a more general implementation, Wij should be a full weight matrix for every node to every other node

Flow of ideas:

Start with a neuron’s behavior, for instance a leaky-integrator:

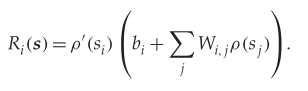


(or the continuous version)



* (ignore eta above, it’s noise which we can dismiss for now)
* Here, what’s said is that the change in state for neuron *h* is equal to some external-driving-force R(s) input to that neuron which depends on all the other neurons (*s)*

Now, according to this local update rule, the neuron states *s* will evolve into some kind of configuration where they become stable. But what is that configuration? Well, if we assume, say, that the driving force for each neuron R\_i(s) is just a bias term plus the weighted sum of outputs (***rho(s\_j)***) from its neighbors ***j***:



* Ignore rho’(s) above, it’s a not necessary for this discussion
* R(s) really is just the input to a given neuron, but remember that in this model the neuron state itself tracks towards that input value!
* rho(s) is really just the output from all the neuron’s connected neighbors

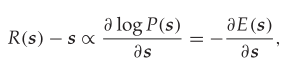
We will find that the configurations that *s* settles into correspond to the lowest-energy points of some as-yet-undetermined energy function *which is entirely determined by the neuron states* ***s*** *and how the neurons talk to each other, i.e.* ***R(s)***. In particular, if we can write that energy function s.t. the probability of those states are



then by definition



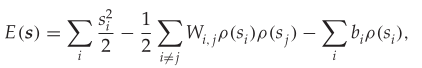
In particular, if we can write that energy function s.t.



Then the stationary states (= the final configuration = the fixed points = when R(s) == s) of ***s*** will correspond to local minima in the energy function and be stable there. We’re going to go a step farther here, and assume that finding these low-energy states is actually useful (without explanation!)

The energy function

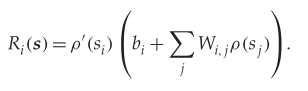
If we just whip up the following energy function out of thin air, we see that it meets the criteria above:



Namely, it obeys:

Where R(s) is the behavior we specified initially for the neuron:



Modifying the energy function

In Scellier & Bengio, they modify the energy function and call it “weakly clamping”, but that left me confused. Their modified form is of the energy is F:



Where C is defined by the difference between the outputs *y* and the target outputs *d*.

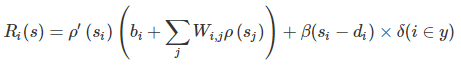


This confused me, because doesn’t that mean it modifies the way the neurons are expected to operate? Isn’t all of this tied together? Well, the answer is yes, it kind of does – but it modifies the way the neurons operate by just what they said. It just acts as an external input signal into the output neurons, as though you had.

Following the equation for R(s)



But with F instead of E, we get



Summary: If neuron *i* is not one of the output neurons (i.e. it’s on *y*), it behaves normally. If it is one of the output neurons, however, it receives external input proportional to the error between its current value and the target value.

Next steps

Implementation plan

* Create code to simulate leaky-integrator neurons (the n/v/a test maybe?)
* Run the simulation to see how long it takes to get to the free point
* Calculate the energy E for the leaky-integrator neurons at the free point
* Clamp the output neurons

Simulating leaky-integrator neurons using “Extending the framework…” to update the state of the hidden units *h* :

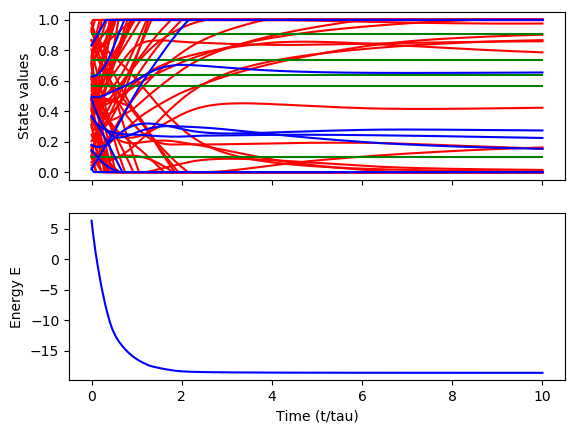
 



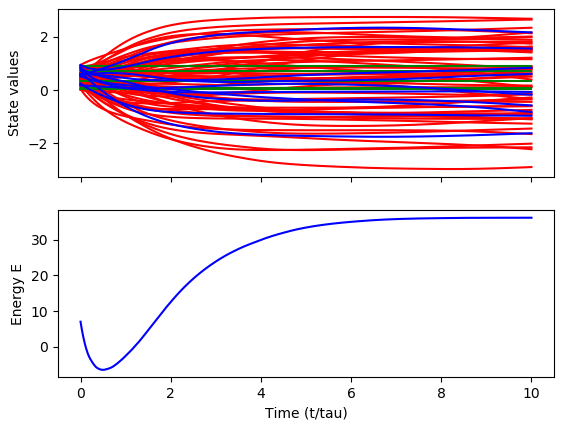
By approximating updates in the Euler fashion:

 (sign error here, should be + eps\*mu)

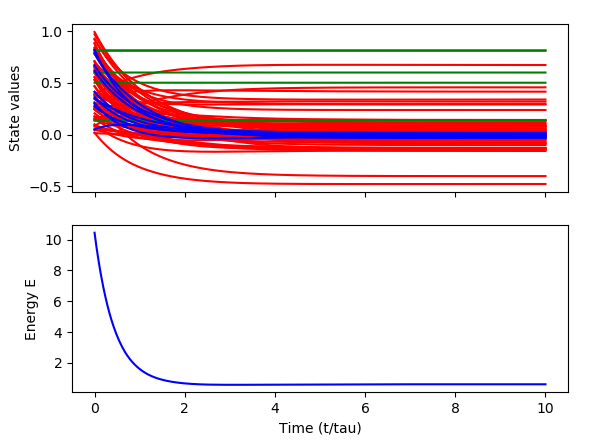
Produces the following state values and energy:



Strangely, though, if I don’t clip the state s to the bounds (0,1), I end up with this (below).



However, I realized that’s because the equation for dE/dS ( = R(s) – s) includes the rho’(s) term. Without the rho’(s) term, which clips the state to between zero and one, the energy minimization property (the fact that we’re descending dE/ds withou our neuron updates) isn’t true. And sure enough, if for example I reduce all the weights by a factor of 10 so I don’t get near the clipping edges – but keeping disabled the clipping of the state between zero and one -- we see that the energy function does get minimized:



In short: You can either make sure you write out R(s) faithfully—including the rhoprime(s) term—or you can ignore the rhoprime(s) term and clip the state to (0,1) which accomplishes the same thing (and is numerically faster)