CISC 481 (Bonus) Programming 3

November 15, 2019

In this extra credit assignment, you'll be training some simple neural networks to be two bit adders. A two bit adder can be looked at as a function that takes two inputs which are the bits to be added, and outputs a pair of values - a bit representing the sum of the two input bits (sans overflow), as well as a bit representing whether or not there was an overflow (the carry bit). Since their are only four possible combinations of values for our input bits, this function is easy to represent in tabular form:

x_1	x_2	y_1 (carry)	y_2 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

My recommendation on building up to a network capable of learning to be a two bit adder is as follows:

1. Come up with a graph datastructure that lets you link nodes of neurons (units) in arbitrary ways as a directed graph. Keep in mind that an edge going from one unit i to another j means that i's output will be given as input to j, so both i and j will need access to a data structure which stores such input/output. One very simple way to do this in Lisp is to associate with each edge a pair `(w . x)` where `w`, the `car`, is the weight associated with that edge and `x`, the `cdr` is where the unit from which the edge comes will place its output and the unit to which the edge goes will read its input.
2. Implement the simple threshold activation function (where we output a 0 if the weighted sum of our inputs is negative, and a 1 otherwise), and the simple weight update function (where we take the difference between what was expected and what we got, multiply it by the input, and add that to the current weight to get the new weight).
3. Using the above, implement a `classify` function which takes a network and some number of inputs and then "runs" the network on those inputs to obtain some outputs. If you store your hidden units in a list (or array) such that units earlier in the sequence will only have output edges to units later in the sequence, and then store your output units (the ones at the "end" of the network - that is, they don't have any edges going out to any other units) in a separate list (array), you can set the inputs passed to `classify` on your input nodes, and then run through the lists of units in order (first the hidden list, then the output list), for each unit u collecting up the inputs and weights associated with each edge u and passing them to the activation function, and then taking the output x of that and setting the input/output value on each edge leaving u to x. Once you've run through all the units in the networks, you can read the values computed by the output units and those will be the return values for `classify`.
4. Implement a simple `update-weights` function which takes a network and some number of expected outputs and, for each output and corresponding output unit, uses the weight update function you implemented in part 2 to adjust the weights for the output unit. One

easy way to do this is to just match up the first expected output with the first unit in the output list, the second expected output with the second unit, and so on and so forth.

5. Build the simple one layer network shown in Figure 1. This is the network we manually trained on the board in class for carry: Keep in mind the dummy inputs denoted by $w_{0,3}$

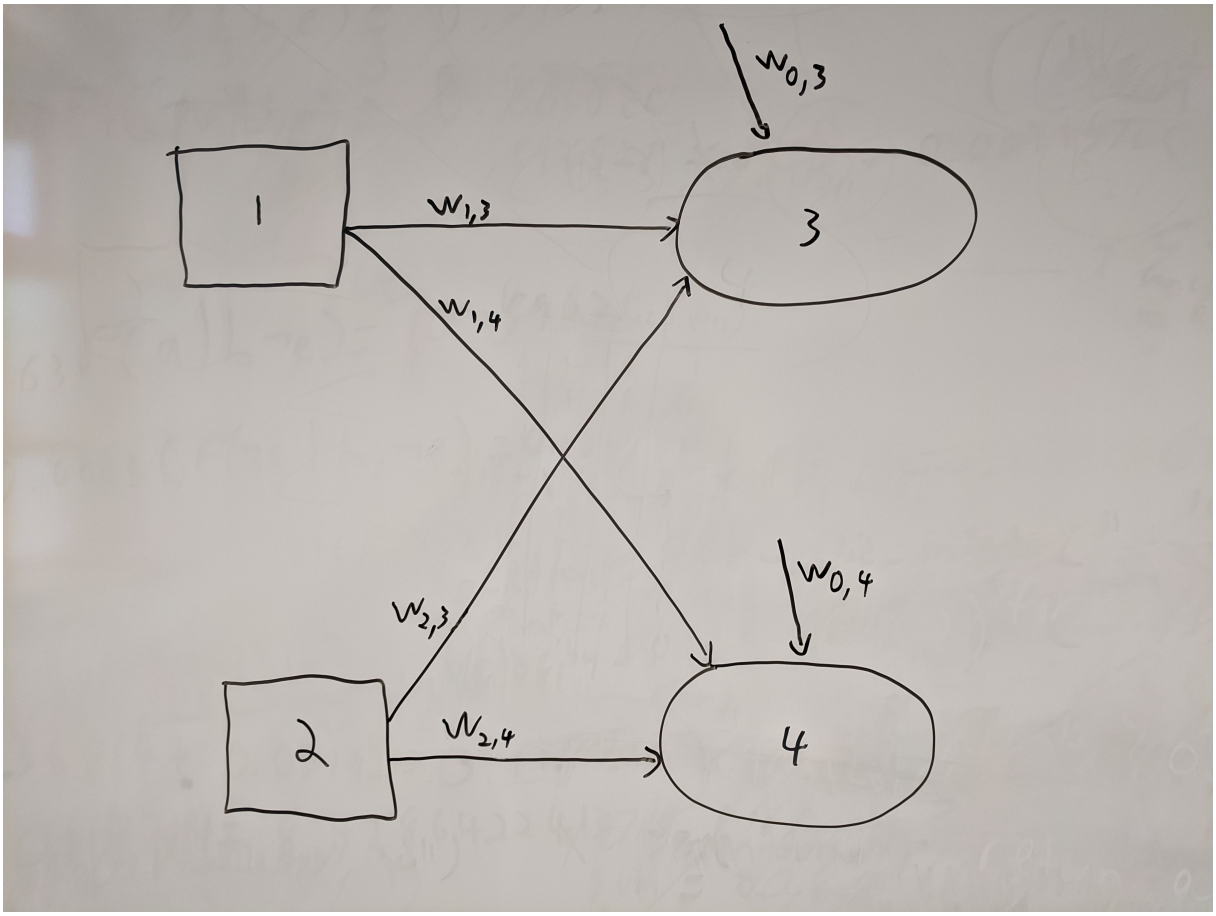

Figure 1: Simple 2 neuron, single layer feed-forward nueral network

and $w_{0,4}$ in the figure. You can represent these as pair in your input list with the cdr set to 1. Since this edge is coming from nowhere (that is, it's not in the output list of any other neuron), that 1 will never get changed, which is the behavior we're after.

6. Now, train your network on the input table at the top of this write-up. For example, if you defined a network `*test-net2*`, you might do this:

```
(dotimes (i 5)
  (classify *test-net2* 0 0)
  (update-weights *test-net2* 0 0)
  (classify *test-net2* 0 1)
  (update-weights *test-net2* 0 1)
  (classify *test-net2* 1 0)
  (update-weights *test-net2* 0 1)
  (classify *test-net2* 1 1)
  (update-weights *test-net2* 1 0))
```

Your network should come to the configuration of weights for unit 3 as we arrived at on the board. I believe ended up with $w_{1,3} = 1$, $w_{2,3} = 2$, and $w_{0,3} = -3$. This will serve as a "sanity check" on all of the code you've written up to this point.

7. Now implement a second activation function using the *Logit* function, and a corresponding weight update function (essentially the same as the first, but with that extra term corresponding to the derivative of *Logit*).
8. Modify your `update-weights` to handle back propagation (whereas the original suggestion was just to update the weights of the output units, now you'll have to go through the list of hidden units *in reverse* and update their weights via the backpropagation algorithm).
9. Build the 2 layer network shown in Figure 2.

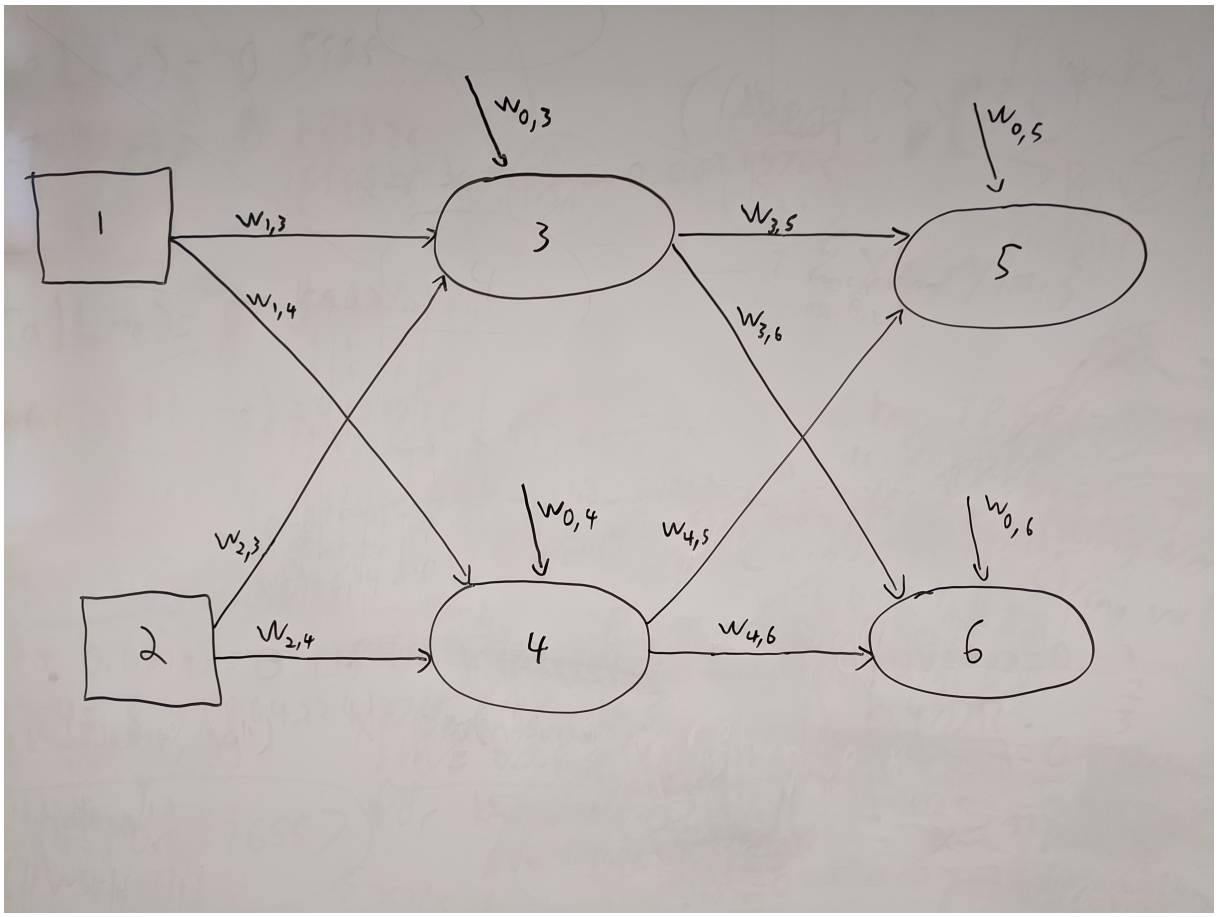

Figure 2: Simple 4 neuron, 2 layer feed-forward nueral network

Try training this network using both the simple threshold activation (and corresponsing weight update), and again (remembering to either reset all the weights to 0 or just rebuild it from scratch) with the logit activation and weight update.

Before, we saw that unit 4 is unable to learn sum. The idea here is that unit 3 will still easily learn carry, and since unit 6 receives inputs from both 3 and 4, that 4 should be able to learn *or*, and 6 can use that information plus whether or not a carry happened to decide whether to output a 0 or a 1. 5 should simply be able to pass on what it got as input from 3 as its own output.

To get the extra credit for this assignment, you should submit all of your code along with instructions on how to run it. If your simple single layer network will successfully learn carry, you'll get 1 percentage point added on to your final grade. If you manage to get the the 2 layer network to learn both sum *and* carry, you'll get 2 additional perecentage points.