HSL622: Assignment 2

Sumeet Agarwal

April 6, 2024

1 Problem statement

This assignment is about familiarising ourselves with the basic ideas that underlie the working of connectionist or neural network models in AI. We first implement a simple network with one intermediate or hidden layer between input and output and fixed (hardcoded) weights, and then experiment with a simple form of weight learning for just a single artificial neuron with two inputs.

2 What to do

2.1 An XNOR network

[5 marks] In class you have seen the XOR network as the canonical example of a connectionist model which needs an intermediate layer between inputs and output. Now consider the logical complement of XOR, *i.e.*, XNOR:

x_1	x_2	x_1 XNOR x_2
0	0	1
0	1	0
1	0	0
1	1	1

Design a connectionist network to compute this function, using the same kinds of artificial neurons as seen in class, which have weights on the incoming connections/synapses and an activation threshold as their parameters. First you should draw the network visually, depicting all the nodes and links and parameters (this diagram will go in your report). Then write a simple piece of code in your chosen programming language, which implements the network. The code should be clearly structured and commented so that the processing corresponding to each individual neuron in the network is apparent. It should take the two input values (x_1 and x_2) from the user, and print out the output value from the final neuron. Check that your code correctly outputs x_1 XNOR x_2, for all 4 possible input configurations as in the above truth table.

2.2 A simple form of weight learning

[10 marks] Now, we would like to play with a basic random process for how the weights in such connectionist networks can be learnt from experience (which is what gives these models much of their power). No claims are made as to the biological plausibility or efficiency of the process to be implemented here, but it is just meant to provide a quick feel for how such a process might work in practice.

For this part, just implement a single artificial neuron with two Boolean inputs, called x_1 and x_2. However, you now have to allow for the corresponding weight values w_1 and w_2 to be variable parameters which we will code up a search procedure over, so that we can find settings for them such that the neuron computes the function x_1 AND x_2. The activation threshold θ can be kept fixed at 0.5, you need not vary that. For w_1 and w_2, initialise them both to some random number[1] between 0 and 1. Then, implement the following kind of search process (as a loop which will run until the termination condition is reached):

1. First, check the output of the current network for all 4 possible input configurations, and compare them to the desired outputs for the AND function. In case all 4 outputs match the desired ones, then you are done, and you can exit the loop.

2. Otherwise, we need to check if the mismatches are in the positive or negative direction. For simplicity, randomly take just any *one* of the input configurations for which the network output did not match the desired output (note that the {0,0} input will never give an error, as its output is always 0 irrespective of the weights). If the network output was 1 and the desired output is 0, we call this a positive error. If vice versa, it's a negative error.

3. Now, the 'learning' part: we will try to adjust the weights so as to move in the direction of fixing the error (we can think of this as a kind of error signal or feedback driving the learning). If you got a positive error in the previous step, it means some weights are too high and need to be lowered. But note that for a given input configuration, *only* the weights corresponding to inputs which were 1 are relevant; for inputs which are 0, the weight just gets multiplied by 0 and hence its value doesn't matter. So generate a random number between 0 and 0.1[2] and *subtract* it from the weights corresponding to those inputs which were 1 in the chosen input configuration[3] Similarly, if you got a negative error, generate a random

[1] Using a random number generator, which will be needed at multiple places in this program. All programming languages provide this functionality; in case of any difficulty with figuring out how to use it, feel free to ask the instructor or TA for help.

[2] This magnitude of how much we change the weights by in a single iteration is sometimes known as the *learning rate*; while a fixed range has been provided to you here, you may want to try varying it for further exploration and seeing how that affects the time taken to converge.

[3] To spell it out: If the chosen input configuration with an error is {1,0} you would only modify w_1. If it is {0,1}, only w_2. If it is {1,1}, both w_1 and w_2.

number between 0 and 0.1 and *add* it to the weights corresponding to the inputs which were 1.

4. Now with the weights updated, go back to step 1 and repeat the loop until the exit condition is reached.

Finally, when you exit the loop, your program should print out the final learnt weights, and also the number of iterations for which you had to run the loop in order to reach them.

Run your program 5 times and report the above outputs for each run in the form of a table.

3 What to submit

- Your code, which should just be a single code file for each of the two parts. These should be in a programming language of your choice, with clear structuring and commenting as mentioned. The program inputs and outputs for each part should be as mentioned (for the second part, there are no inputs to be taken).
- A short report (common for both parts), where you document in text/diagram form the things you have been asked to for each part above. If you wish you can use scans/photos of hand-drawn diagrams/tables.

These files will need to be uploaded on Moodle; the opening of the submissions will be announced in due course. Please aim to complete this assignment by the end of **Saturday 27th April**.