Computation and Mind: ***Lab 2 - Simple Forms of Memory***

In the previous lab, we explored how processing in simple neural-like processing elements (receptive field neurons) could help explain how the visual system is able to process images to do interesting things like edge detection, low pass spatial filtering, and support hyper-acuity. In this lab, we extend the same idea of simple neural computation to support learning and memory processes.

One of the most basic forms of neural learning is known as Hebbian learning (something discussed in our reading of Chapter 3). According to Hebbian learning, neurons that fire together become increasingly linked so that the firing of either neuron will increasingly cause both of them to fire. In this lab, we will explore a memory system that operates based on this simple principal and explore some of the computational characteristics associated with this system.

The ideas presented in this lab should seem familiar to you given our readings from Edelman's book (particularly page 160-172). In particular, in our lecture over those sections we played with a simple system that could learn to memorize particular patterns and recall them given incomplete or noisy cues. In this lab we are going to try to actually build a simulator that works in a similar way.

The Principal of Hebbian Learning

Donald Hebb first noticed a form of neuronal learning in 1949. In his experiments, he found that when two neurons repeatedly fire together the association between them seems to strengthen such that activating one of the pair would often cause the other to fire. In effect the paired firing of two neurons appears to "link" them together. In his words, "When neuron A repeatedly participates in firing neuron B, the strength of the action of A onto B increases".

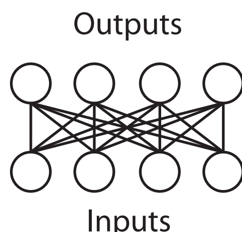


Fig. 1: An Auto-associative network

In this lab we will investigate this principal using a neural network called an auto-associator network. The basic idea is that there are a set of input neurons and a set of output neurons. Association weights connect each input unit to each output unit. During learning, changes are made to the strength of the weights. The goal of the network is to re-create its input on the output. While this might seem a pointless task, we will show how this type of system can create a relatively powerful memory.

We assume that the input units are a single list of values. Neurons can either be ON (1) or OFF (-1). Output units are activated by the input units according to the following equation:

$$y_j = \sum_i w_{ij} \cdot x_i$$

which simply says the output of neuron 'j' is equal to the sum of all the input units (x_i) times the value of the weight connecting input using (x_i) to output unit (y_j).

Learning takes the form of the Hebbian association rule:

$$\Delta W_{ij}(t) = \eta \cdot y_j \cdot x_i$$

Which says the change to the weight connecting input unit I and output unit j is equal to the value of the input pattern unit (x_i) and output unit (y_j) times a learning rate or step-size parameter (η).

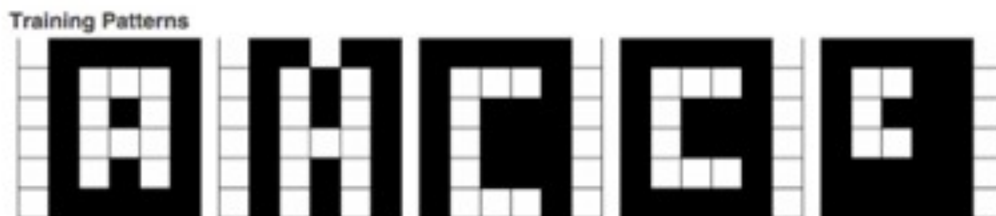
In class, we will discuss the operation of the memory.py script which is available online. This script will form the basis of our experimentation.

Step 1

What happens when you make two of the input patterns very similar to one another? What happens when you make all of the patterns in a different section of the screen (so there is little overlap)?

Step 2

Set PATSIZE=6 (so each input/output pattern is a 6x6 grid). Then input the following training patterns.



Note the column of "on" pixels along either the left or right edge. You can think of this as the "category label". Here we have two categories, A and C. Train your network for ~40 blocks. Now perform the following tests:

1. Input the large c (middle pattern) but turn OFF the column along the right (i.e., remove the category label). Test the pattern with the network? What happens?
2. Input the smaller "A" (first pattern) also without its "category label" or name. What

happens here?

3. Now input only the category label (i.e. vertical column on left or right) with no pattern. What does the network do?
4. Now create 5 new input patterns. Put two patterns in category 'A' (associated with the left column all being set to ON) and three in category 'B' (associated with the right column all begin set to ON). However, make it so that each pattern doesn't overlap with each other either within or between categories (patterns can be simple lines or something). Train the network as above, and test the "categorization performance" of the network. Does the network "fill-in" the category label when it is missing?

Step 3

One advantage of models like this is that they can be resistant to damage (called fault-tolerance). Add a new button to the program called "Lesion" which, when clicked, will randomly select ~1% of the memory weights in the model and reset them to zero (imagine this is the same as inducing "lesions" in a rat, but obviously much less painful!!). The basic outline of where you should write this function has been provided as a function called "lesion". Repeat your experiment above. However, after testing (to be sure the network learned the training patterns), press the lesion button. Describe how well the model does. Keep pressing the lesion button until performance drops considerably (i.e., the output pattern looks nothing like the input pattern). How fault-tolerant is the network in your opinion?

Step 4

At the top of the script you see that input is encoded such that OFF=-1, ON = 1. What happens if you change OFF =0 instead? Consider the equation above first, then try it in the code by making the change.

Step 5

What are some limitations to this simple model? Besides the situation implied by the last question, can you think of an example set of pictures that the model will perform poorly on but humans are likely to do well? One way to think about this problem is with respect to the last example. Why does the model do poorly in this situation? Show a picture of the patterns you chose and describe what happens.