Hebbian Pattern Association

Let us consider one of the most basic methods of pattern association in the literature: **Hebbian learning**. The Hebb rule itself was described in **LAB 4**. To repeat, it is a learning rule which adjusts weights according to the following rule: At every time step, we change each weight by the product of a learning rate epsilon, the source activation, and the target activation. *"Neurons that fire together wire together."*

Simple Heteroassociative Network

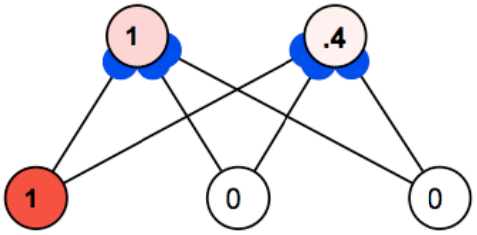
Suppose we want to train a network to perform the following association task:

Source	Target
(1,0,0)	(1, 0.4)
(0,1,0)	(0.8, 0.2)
(0,0,1)	(0.5, 0.7)

That is, we want a network which implements a vector valued function from a three-dimensional vector space to a two dimensional vector space.

Teaching the network this task isn't too hard:

- Create the network shown to the right
- Make all neurons clamped
- Set all weights to **0** (**W** then **C**)
- Make the weights **Hebbian** and set the learning rate to **1**.



Now set the activations of the source nodes to correspond to the first pattern, and those of the target nodes to correspond to the second pattern, as follows:

Now clamp the neurons by pressing the **Clamp Neurons** button on the network toolbar. Now **iterate** the workspace. The weights will be updated according to the Hebb rule, and our first association has been formed. You will notice the synapses change color and size. As an exercise, try to say what the new weights values are.

(Note that you must only iterate the network once. If you keep doing it the weights will keep increasing or decreasing in magnitude until they reach their maximum or minimum value. This sensitivity is one of the problematic features of Hebbian learning.)

We now repeat the process for the other two source / target associations. This completes the **training phase** of this task. Now we enter a **testing phase**. We want to test the network to see how well it can recall these associations. Will it recall the right target pattern given a source pattern? Has it properly implemented the vector-valued function above?

To test the network, we need to do two things. First, we must clamp all the synapses; we must turn learning off. To do this press the **Clamp Weights** button in the network toolbar. Second, we need to unclamp the neurons, by toggling the **Clamp Neurons** button to be off.

Now we are ready to test. For the first input pattern, set the input nodes to **(1, 0, 0),** and **iterate** the workspace. The output neurons should produce the correct pattern, **(1, 0.4).** Similarly for the other inputs.

Note also that the network generalizes fairly well. For example, try the pattern **(1, 0.1, 0)** it produces more or less the same output as **(1, 0, 0).** A related point is that the network does ok with noisy inputs, and degrades somewhat gracefully (though given the size of the network its degradation under loss of weights is not great).

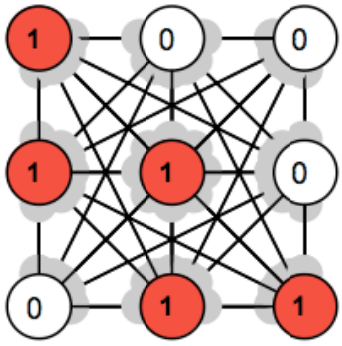
Qualities of the input space

The effectiveness of Hebbian pattern association depends on the nature of the input patterns being learned. If the input patterns are too similar, they will cause the same weights to be trained, and this will interfere with recall later on. This is sometimes called **cross-talk**.

The best case is the case where all the input vectors are **orthogonal** (for us, this means vectors whose dot product is **0**). Why does this help the pattern associator learn? Let's focus on the case of binary vectors. Roughly speaking, orthogonal input vectors don't step on each other's toes, and this is especially apparent with orthogonal vectors which are binary. Consider: binary input vectors produce an activity of **1** in some input nodes and **0**'s everywhere else. If they are orthogonal, they produce **1**'s on different nodes. That means that the various input vectors give rise to learning on different weights. So there is no cross-talk.

The thing to remember is that Hebbian pattern associators can learn associations when all the input vectors are orthogonal. Otherwise there will be cross-talk and perfect learning is not guaranteed.

Simple Auto-associative network



We now consider a recurrent, auto-associative network trained using the Hebb law. Recall that an auto-associative network is trained to associate patterns with themselves. What is attractive about these networks is that they provide a biologically realistic model for the formation of memories. They also do well at such tasks as pattern completion.

In this section we study very simple networks of neurons using linear activation functions with slope **1** (the identity function), which lack self-connections but are otherwise **fully interconnected**. The synapses start off at **0**, and we assume a learning

rate of **1**. (Such network are also called "**linear auto-associators**" and are related to "brain-state-in-a-box" models).

- Now load the network **AutoAssoc9.xml** via **File** --> **Open Network**.

To **train** the network, create a pattern in its nodes, as above. Be sure the neurons are clamped by using the **Clamp neurons button** and that the weights are not clamped. Now select some nodes (three or more) and press the **up button** to add an activation of **1**. **Iterate** the network a single time, and you will see that co-active neurons "wire together" in that the connections between them turn red (what will their value be after this one iteration?).

Note that when creating patterns the **wand tool** helps. To start over quickly press **"W"** and then **"C"** (to select all weights then clear them). It is also helpful to press **"N"** (to select all neurons) and then to press the **up** and **down buttons** to quickly set the network state. To test the network for recall, **clamp the weights** and **unclamp** the neurons. Now add part of the input of the pattern you trained it on, and **iterate** the network a few times. It should reproduce the pattern you trained it on. It's easy to see why: all the synapses which were trained form a subset of the network's nodes connected by positive weights. They have been trained to fire together, as an ensemble. Also, note that this network will perform fairly well even if some synapses are removed (**graceful degradation**) or if you add noise to the input; in fact the partial input can be thought of as a noisy input.

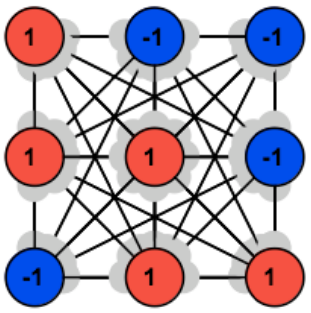
You can also look at this network in terms of a **dynamical systems theory**. If you put the network in to random activation states (by selecting all neurons and pressing the randomize button) and iterate, you will note that it either settles into the pattern you created or to its **"dual"**, a pattern of **-1**'s on the same neurons.

You can visualize this by opening a **projection plot**

- **Click**  and then Projection Plot.
- Now couple your network to this plot by clicking **Couple --> Projection1**.

Any point in activation space will evolve to one of these two attracting fixed points. Thus you have, via learning, created two **attractors** corresponding to this pattern and its dual.

We just now trained this network on a single binary input vector. What if we try to train it on another pattern? Various interesting things can be observed. If the second pattern overlaps the first (in which case the two input vectors are not orthogonal), then during recall any partial version of either pattern will produce the two patterns together. Why this happens makes sense: in the training the network on the second vector you essentially added on to the network of positively connected neurons.



To address this problem, we can use bipolar patterns. The pattern above in bipolar form looks like this:

Do the same procedure as above, training the network on a few patterns. You will notice that this time, after training, all the non-excitatory weights become inhibitory. This time, you can allow the training patterns to overlap a bit, and in some cases it will still remember them. In other cases, however, the last pattern you trained the network on will dominate.

Since bipolar patterns are being used, when patterns are recalled, not only is the whole pattern activated but other nodes are inhibited, they take on negative values. This makes it possible to store overlapping patterns. If you store two patterns with just one overlapping unit, for example, you will notice that both can be independently recalled. The reason is that activating one pattern simultaneously **represses** the other.

You will also notice new patterns, which are, as it were, byproducts of the first two: these are sometimes called **spurious memories**. In the case of a network trained on **1** bipolar pattern, these are easy to predict: they correspond to the compliment of the trained pattern (that is, the pattern formed by **-1**'s rather than **1**'s).

These techniques can be scaled up to networks with more neurons. (One experiment you can try is creating a network with 25 neurons and then teaching it various letters of the alphabet.)

Another problem we run into with these networks is oscillations. The network does not always settle down into a stable fixed point. In terms of dynamical systems theory, we can observe **periodic orbits** in activation space. This second problem can be solved by changing the way we update neurons. One way this is done is using **"Hopfield networks"**. We will not cover them in this tutorial, though.