NOTES ON NEURAL NETWORKS

Neural networks are abstract mathematical models of interconnected neuronal behavior. Perhaps the first such model was that of the McCullough-Pitts artificial neuron in 1949. This postulates a unit of processing (acell body and an axon) with characteristics indicated:

```
        --------
  a1---|Wj1 |   |
  a2---|Wj2 |   |
  a3---|Wj3 |Hj |--------------- aj
  ...  |... |   |
  an---|Wjn |   |
        --------
         unit j
```

Here the ai are incoming signals (0 or 1) that arrive at the processing unit j, where each is multiplied by a weight Wij and the result summed: SUM = aiWj1 + a2Wj2 + ... + anWjn. This sum is then compared to the threshhold value Hj for the unit. Finally, the output signal aj (which is sometimes called the activation level of the unit) is defined as

```
         { 1 if SUM >= Hj
    aj = {
         { 0 otherwise
```

More precisely,

```
              { 1 if SUM >= Hj
    aj(t+1) = {
              { 0 otherwise
```

where SUM(t) = ai(t)Wj1 + a2(t)Wj2 + ... + an(t)Wjn.

A neat mathematical notation for this is

```
               _
    aj(t+1) = (-)(SUM(t) - Hj))
```

```
       _
where (-)(x) = 1 if x >= 0, and 0 otherwise (the so-called Heaviside
function).
```

That is, the signals ai vary dynamically at each time step, and the output signal is recomputed and sent one time-step after the incoming signals arrive. The weights Wij can be any real numbers; a negative weight can reduce the overall incoming contribution to SUM, and corresponds to an inhibitory synapse.

By a network is meant a collection of such units that are connected to one another via signal wires. An output wire such as aj above can split and connect to many units. Note that in principle aj(t) can be an input to unit j and thus can influence aj(t+1). A network that allows this is called "recurrent"; one prominent example is the "complete" network in which *every* pair of units is connected by a wire.

Here is a simple example of a (recurrent) network:

```
           --------
  -->-----| -1 | 0 |-->-
 |         --------     |
 |                      |
  --------<-------------
```

This has just one unit which feeds back to itself. It "blinks" since whatever the incoming signal is on the left (say on, or 1), it produces the opposite (eg off, or 0) on the right one step later, but then that (eg 0) becomes the new incoming signal, etc. So the output signal perpetually changes back and forth: 0,1,0,1,0,...

In addition to recurrent networks, there are so-called feedforward networks, in which each neuron is used only once in a given computation, as signals pass through it (from left to right) and then on to another "layer" of neurons:

```
layer 0   1   2   3
      |   |   |   |
      V   V   V   V

  ----o---o
       \   \
        \   \
         \   \
  ----o---o---o---o----
         /
        /
       /
  ----o---o---o---o----
       \
```

```
       \
        \
 ----o---o---o---o----

        ^^^^^^
 input  hidden   output
 layer  layers   layer
```

In the diagram, layer 0 is the input layer, layers 1 and 2 are "hidden" layers, and layer 3 is the output layer.

Here is a feedforward network that acts like an AND-gate:

```
      -------
p----|   |   |        The only way for the input contribution to
     | 1 |   |        reach the threshhold of 2 is for both p and
     |___| 2 |____    q to be "on", ie to be 1's.  If we interpet
     |   |   |        1 as true (and 0 as false) then this is an
     | 1 |   |        AND gate: it fires (produces an output of 1)
q----|   |   |        iff both inputs are true.
      -------
```

Notice that there are no hidden layers here: just an input layer (p and q) and an output layer, for a total of three units. Also one can easily and similarly create OR-gates, NOT-gates, and many other logic gates.

But it turns out that to make an XOR-gate (true iff exactly one of itw two inputs is true) one must have at least one hidden layer, as in:

```
       -------
  p----|   |   |                 __________
       | 1 |   |                |     |    |
       |___| 1 |________________|     |    |
       |   |   |                |  1  |    |
       |-1 |   |                |     |    |
  q----|   |   |                |     |    |
       -------                  |-----|    |
       -------                  |     |    |
  p----|   |   |                |     |    |
       |-1 |   |                |     |    |
       |___| 1 |________________|  1  |    |
       |   |   |                |     |    |
       | 1 |   |                |     |    |
  q----|   |   |                |     |    |
       -------                  |-----| 1  |------ P XOR q
```

```
      -------                  |     |    |
p----|   |   |                 |     |    |
     | 1 |   |                 |     |    |
     |___| 2 |_________________| -1  |    |
     |   |   |                 |     |    |
     | 1 |   |                 |     |    |
q----|   |   |                 |     |    |
      -------                  |-----|    |
      -------                  |     |    |
p----|   |   |                 |     |    |
     |-1 |   |                 | -1  |    |
     |___| 0 |_________________|     |    |
     |   |   |                 |     |    |
     |-1 |   |                 |_____|____|
q----|   |   |
      -------
```

Here there are only two input layer units, p and q, as before, but for ease of drawing I have shown their connectione to all four hidden units without showing the crossing wires that would be needed if they are all lying on the same plane. The top two hidden units fire if either p is true and q false (top unit) or vice versa (second unit). The bottom two fire if either both are true (third unit) or neither (last unit). This exactly one of the four hidden units can fire for any given input values. [In fact, as one student pointed out, the bottom two units and their connections to the output unit can be removed altogether and the simpler network (only two hidden units) still computes XOR.]

With enough hidden units in a feedfoward netwrok, it is possible to compute any computable function.

For feedforward networks there is a famous and much-used algorithm, "error backpropagation", that allows the networks synaptic weights Wij to be adjusted to fit with 'training data" so that the outputs are the desired ones for given inputs; and after that the network often tends to exhibit the desired input-ouput relationship even for new data. For instance, suppose a feedforward network is trained on 100 instances of the handwritten letter "E" and another 100 that are not E's. Let us further suppose that there are, say, 625 input units (corresponding to a 25x25 grid of pixels) and two output units (corresponding to "E and "not-E"). Once its weights are adjusted by repeated runs in the error-backpropagation learning algorithm so that it correctly categorizes the 200 inputs, it can then be used with no further adjustment to distinguish new handwritten letters (as E or not-E). A more sophisticated network might be trained to distinguish all 26 letters, and upper and lower case as well. The rough idea behind backpropagation is to compare the actual output with the desired output (for a given input) and calculate a set of weight-alterations that will bring the ouput closer to what is wanted. The actual algorithm uses derivatives and requires a change in the basic formula for aj(t+1) so that instead of the Heaviside function, a differentiable substitute is used.

Recurrent networks can also be trained, and have found useful application when matching a new item with a set of stored ones to see which is the closest match. In this paradigm, there are no layers, and the entire network is used for input, as well as for output. Input values can be specified as activation values (0 or 1) on the connecting wires; and output values are also so specified, after the ensuing computations settle into a stable (unchanging) state.

An example is face-recognition: a complete network is trained on, say, 100 faces (photographs) so that, given any one of them as input (eg pixel data) it it simply remains in that state. Then a new photograph is presented, and the network goes through stages of processing as activation levels change until eventually each wire reaches a final activation (0 or 1) that no longer changes. It turns out that (if the training was done properly) the final state is exactly one of the original 100 stored images, and (typically) the closest match to the new image.

While feedforward nets can also be used for this purpose, complete networks can store far more information due to the very large number of connections. Viewed as storage devices, complete networks are said to implement a kind of "associative memory."