1. DEFINITIONS

A neural network is specified by a number of *nodes*, which may be designated as network input elements or computation elements. Every node not designated as an input element receives input from other nodes in the network.

For any node k, denote by S_k the set of *successor* nodes, which receive input from node k. Also, denote by P_k the set of *predecessor* nodes, from which which node k receives input. Finally, denote by Ω the set of nodes which compute the network's final output. If the node j is an element of S_i, then the connection between node i and node j is governed by a numerical *weight*, denoted w_{ij}.

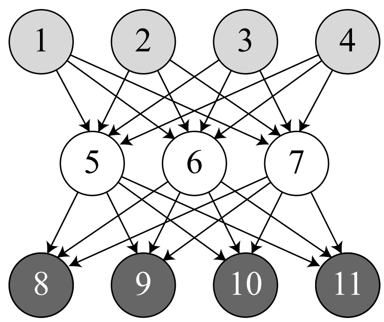

Figure 1: A simple neural network. Lightly shaded nodes (*top*) are network inputs. Dark shaded nodes (*bottom*) are outputs.

In the figure above, nodes 1-4 are all network input nodes, and nodes and the set of network outputs is $\Omega = \{8, 9, 10, 11\}$. The predecessors of node 6 constitute the set $P_6 = \{1, 2, 3, 4\}$. The successors of node 6 constitute the set $S_6 = \{8, 9, 10, 11\}$.

A network need not be configured in fully connected layers, as in the figure above; however, connections between nodes must be *acyclic*: that is, there should be no path along connections between nodes thad leads from a certain node back to that node itself.

2. FEEDFORWARD COMPUTATION

Each node k relays a numerical output y_k to its successor nodes. If i, is an input node, y_i is simply the value input. For a non-input node j, the output $y_j = f(x_j)$, where

$$x_j = \sum_{i \in P_j} w_{ij}\, y_i$$

and $f(x)$ is a smooth non-linear function which maps the entire real line to a small domain. One popular choice for $f(x)$ is the hyperbolic tangent function $\tanh(x)$:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

which asymptotically reaches $y = \pm 1$ at $x = \pm\infty$, respectively.

To compute the output of the network for a given input, begin by setting y_i to the input value for each input node. Then, iteratively compute y_j for each non-input node j whose inputs have all been determined until all of the nodes in the network have been set.

Bias nodes. Typically, each non-input node j in the network receives some input from a *bias* node i, a special type of input node whose value is always $y_i = 1$. This allows the network to learn a greater range of functions.

3. BACKPROPAGATION OF ERROR

In order to train the network to achieve a particular output for a given input, we will use gradient descent. Denote by t_k the desired output for node $k \in \Omega$, given a certain set of inputs.

We will define the *error* of the network to be the sum of squared residuals:

$$E = \frac{1}{2} \sum_{k \in \Omega} (t_k - y_k)^2$$

Ultimitely, for each weight w_{ij}, we wish to compute

$$g_{ij} = -\frac{\partial E}{\partial w_{ij}}$$

taken together, the set of g_{ij} form the (negative) gradient of the error with respect of the weights, the direction of steepest descent of network error.

We can use the chain rule to compute each g_{ij}. We define

$$\delta_k = -\frac{\partial E}{\partial x_k}$$

For an output node $k \in \Omega$,

$$\begin{aligned}
\forall k \in \Omega, \quad \delta_k &= -\frac{\partial E}{\partial x_k} \\
&= -\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial x_k} \\
&= (t_k - y_k) f'(x_k)
\end{aligned}$$

For a non-output node $i \notin \Omega$, we have

$$\begin{aligned}
\forall i \notin \Omega, \quad \delta_i &= -\frac{\partial E}{\partial x_i} \\
&= \sum_{j \in S_i} -\frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial x_i} \\
&= \sum_{j \in S_i} -\frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial y_i}\frac{\partial y_i}{\partial x_i} \\
&= \sum_{j \in S_i} \delta_j w_{ij} f'(x_i) \\
&= f'(x_i) \sum_{j \in S_i} \delta_j w_{ij}
\end{aligned}$$

Then, to compute any particular g_{ij}, we have

$$\begin{aligned}
g_{ij} &= -\frac{\partial E}{\partial w_i j} \\
&= -\frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} \\
&= \delta_j y_i
\end{aligned}$$

The process of computing the g_{ij} values is known as backpropagation because, unlike computing the feed-forward network outputs which happens in a top-town manner, computing the error gradient happens in a bottom up manner.

4. NETWORK TRAINING

To train a network, begin by initializing all of the weights w_{ij} to small random values (typically in the range [-1,1]). Then, until performance is adequate, repeat the following steps:

- For every weight w_{ij}, initialize $\Delta_{ij} \leftarrow 0$
- Present the network with N sets of inputs and outputs randomly selected from a training set. For each input/output pair, do the following:
  - Compute and store x_k, y_k, and $f'(x_k)$ for each node k using the feedforward process.
  - Compute and store δ_k for each node k and g_{ij} for each weight w_{ij}, using back-propagation.
  - For each weight w_{ij}, set $\Delta_{ij} \leftarrow \Delta_{ij} + g_{ij}$.
- After N training examples, for each weight w_{ij}, set $w_{ij} \leftarrow w_{ij} + \alpha\Delta_{ij}$, where α is a small step size greater than zero.

Generally $N = 10$ or so works well across a variety of network sizes. The step size parameter α is fairly sensitive to network size. For networks with a small number of weights (up to 10 or so), $\alpha = 0.1$ works fairly well. For networks with large numbers of weights, you will have to decrease the step size significantly. A reasonable starting point is to set $\alpha \approx 1/W$, where W is the number of weights.

If α is too large, either the network weights will rapidly oscillate between useless values, or the weights will be driven to ever-increasing values. In either case, the error will fail to decrease over time. If this happens, try reducing α by an order of magnitude and starting over.