Backpropagation

Fall 2011

This is a document describing the backpropagation algorithm for Neural Networks.

1 Notation

A neural network is an acyclic directed graph of nodes. Each node is numbered and produces an associated value, the value produced by node j is written z_j. The sources (nodes with no incoming arcs) are *input nodes*, the sinks (nodes with no outgoing arcs, often there is only a single sink) are *output nodes*, and the other nodes are called *hidden nodes* since their outputs are not directly observed outside of the network. An arc (i, j) indicates that value produced by node j depends on the value produced at node i. Neural networks are usually drawn with the arcs going up, see Figure 1.

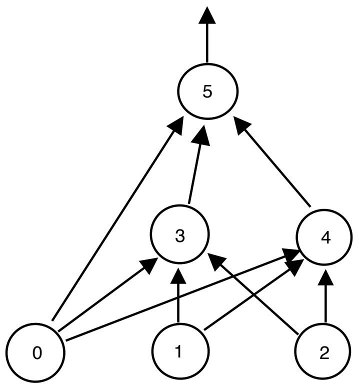

Figure 1: An example neural network with 6 nodes.

Most input nodes correspond to features of the example (the inputs to the neural network) and produce the value of the corresponding feature. However, there is also a special input node (usually numbered 0) that produces the constant value 1 so that biases can be easily represented.

Let $\mathcal{I}_j$ be the set of input values to node j, so $\mathcal{I}_j$ is the set of nodes $\{i : (i, j) \text{ is an arc in the graph}\}$. Similarly, let $\mathcal{U}_j$ be the set of nodes $\{k : (j, k) \text{ is an arc in the graph}\}$, so $\mathcal{U}_j$ is the set of nodes using the value produced by node j. For example, in the network of Figure 1, $\mathcal{I}_3 = \{0, 1, 2\}$ and $\mathcal{U}_3 = \{5\}$.

The value produced by a hidden node j is calculated in two steps. First, the *activation* a_j is calculated and then the activation is transformed to get the z_j value produced by node j. The activation a_j is a weighted sum of the input values to node j. Let w_{ji} be the weight of the arc from i to j (note that the ji order on the subscript may seem backwards – it indicates how much node j weights the value produced by node i). Using this notation the activation a_j can be expressed as:

$$a_j = \sum_{i \in \mathcal{I}_j} w_{ji} z_i \quad .$$

Neural networks typically use a sigmoidal ("S"-shaped) function at the hidden nodes, like the tanh()

or logistic sigmoid. We consider the logistic sigmoid function $\sigma()$ here, so if j is a hidden node then

$$z_j = \sigma(a_j) = \frac{1}{1 + e^{-a_j}} \quad .$$

For example, consider the net in the figure and assume that $z_0 = 1$, $z_1 = 1$, and $z_2 = 2$. Recall that node 0 produces the constant value 1, so this corresponds to the feature vector (1,2). If the weights for node 3 are: $w_{30} = 1$, $w_{31} = 2$, and $w_{32} = -1$ then the activation at node 3, $a_3 = 1 + 2 - 2 = 1$ and the output $z_3 = \frac{1}{1+e^{-1}} \approx 0.73$.

Note that the $\sigma()$ function has some nice properties: it is increasing in the activation, $\sigma(-\infty) = 0$, and $\sigma(\infty) = 1$. Also its derivative is

$$\frac{d\sigma(a_j)}{da_j} = \sigma(a_j)(1 - \sigma(a_j)) = z_j(1 - z_j) \quad . \tag{1}$$

Output nodes have a transformation function appropriate to the problem. If they are supposed to output a probability, then the $\sigma()$ function is appropriate. If they are supposed to match arbitrary real numbers, then the identity function is often used. Here we will assume that the transformation function on the output node is the identity function, so (for the net in the figure) $z_5 = a_5 = w_{50} \cdot 1 + w_{53} \cdot z_3 + w_{54} \cdot z_4$.

2 Stochastic Gradient Descent

Neural networks are generally trained using gradient descent on a function measuring the error between the networks outputs (z_5 in the example) and the required outputs. In stochastic gradient descent the training set is given to the network one example at a time and the weights are adjusted based on each example in turn. The training process cycles through the training set multiple times until the weights converge. Here we show how to update the weights based on a single example, using gradient descent to reduce the the squared error $\frac{1}{2}(\text{output} - \text{desired value})^2$. (Note that the factor of $\frac{1}{2}$ is to simplify the derivative.)

To perform a gradient descent step based on an example $((x_1, x_2), t)$, we first feed the instance (x_1, x_2) into the network and calculate the a_j and z_j values for each node j based on the current weights. This is a "forward" evaluation where values flow in the direction of the arcs.

Once we have the output (z_5 for the example network) we can compute the squared error, $E = \frac{1}{2}(z_5 - t)^2$. We now have to compute, for each node j and each of its weights w_{ji}, $\frac{\partial E}{\partial w_{ji}}$ and then update $w_{ji} := w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}$. The back-propagation algorithm is an efficient way to compute these partial derivatives.

For node $j = 5$, the output node, we have by the chain rule:

$$\frac{\partial E}{\partial w_{5,i}} = \frac{\partial E}{\partial a_5} \frac{\partial a_5}{\partial w_{5,i}} \quad .$$

Since $a_5 = z_5$, we have that $\frac{\partial E}{\partial a_5} = (a_5 - t)$. Since $a_5 = \sum_{i \in \mathcal{I}_5} w_{5,i} z_i$ we also see that $\frac{\partial a_5}{\partial w_{5,i}} = z_i$ (for each $i \in \mathcal{I}_5$).

The quantities $\frac{\partial E}{\partial a_j}$ will be very useful, and were called δ_j in the lecture. These are the values that are "back-propagated" down through the network (in the opposite direction of the arcs).

Consider now an arbitrary hidden node j (like 3 or 4 in the example), and let's examine $\frac{\partial E}{\partial w_{ji}}$ for some node i whose z_i value is used by the hidden node j.

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} z_i \tag{2}$$

since $a_j = \sum_{\ell \in \mathcal{I}_j} w_{j\ell} z_\ell$.

We now concentrate on the $\frac{\partial E}{\partial a_j}$ term. Note that node j's activation a_j is only used to compute its z_j value. The value z_j is then used by each node k in $\mathcal{U}_j$. It might help to think of writing out the computation of upper part of the neural network in terms of z_j and the other z-values produced by nodes at the same level as node j. The derivative of the final output with respect to z_j has to account for everywhere z_j appears in this expression. Therefore, using the chain rule again,

$$\begin{aligned}
\frac{\partial E}{\partial a_j} &= \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j} && (3)\\
&= \left(\sum_{k \in U_j} \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial z_j}\right)\frac{\partial z_j}{\partial a_j} && (4)\\
&= \left(\sum_{k \in U_j} \frac{\partial E}{\partial a_k} w_{kj}\right) z_j(1-z_j) && (5)
\end{aligned}$$

where the last step used the facts that $a_k = \sum_{\ell \in \mathcal{I}_k} w_{k\ell} z_\ell$ and $z_j = \sigma(a_j)$ (as well as Equation (1)).

Combining this with Equation (2) shows that

$$\frac{\partial E}{\partial w_{ji}} = \left(\sum_{k \in U_j} \frac{\partial E}{\partial a_k} w_{kj}\right) z_j(1-z_j) z_i \tag{6}$$

Therefore, if only we knew the $\frac{\partial E}{\partial a_k}$ values for the nodes "above" node j in the network then we could calculate the needed $\frac{\partial E}{\partial w_{ji}}$ values to update node j's weights. Fortunately, Equation (5) gives us a recursive way to calculate the $\frac{\partial E}{\partial a_k}$ values starting at the output nodes (top) and working backwards down through the network. This is the well known back-propagation algorithm for updating the weights based on a single example and the old weights:

1. Use the example and old weights to compute all the a_j and z_j values in the network.
2. Directly compute the $\frac{\partial E}{\partial a_k}$ values at the output node(s).
3. Use Equation (5) to compute the other $\frac{\partial E}{\partial a_j}$, working backwards through the network
4. Use Equation (6) to compute the derivative of the error with respect to each weight in the network
5. Finally, update each w_{ji} weight to $w_{ji} - \eta\frac{\partial E}{\partial w_{ji}}$ where η is the learning rate. (We take a step in the negative gradient direction since we are attempting to decrease, rather than increase, the error.)

Note that only step 2 depends on the activation function at the output nodes. Also, although I have described stochastic gradient descent here, it is also possible to perform a batch-style gradient descent, where one adds together corresponding $\frac{\partial E}{\partial w_{ji}}$ values from all of the examples before changing any weights and then does a single combined update.