Note on Back-propagation

What is Back-propagation?

Back-propagation is a recursive algorithm, by virtue of which the computation (evaluation) of gradients for parameters of **Neural Networks** is reduced in computational complexity from $O(W^2)$ to $O(W)$.

The Back-propagation algorithm is essentially a clever way to calculate derivatives of specific forms of function compositions (e.g. **feed-forward Neural Networks**) exploiting structure in the resulting analytical expressions when applying the chain-rule of calculus.

Derivation

Note that Backpropagation as derived here, concerns **Neural Networks** with a **feed forward** topology (No feedback loops or generally, cycles, in the computation graphs).

The derivation here holds in generality for any such **Neural Network**.

We use the following notation:

- a_j refers to the **input** for node j (to emphasize generality, nodes are not specifically indexed by **layer**, we only care about **connections**, however given a **feedforward** topology)
- z_j is the **activation** of **hidden node** j.
- y_k is the **activation** of **output node** k.
- E is the **error function** applied to the **activations** of the **output nodes**
- t_k is the **target value** for

Note that we have, a_j and z_j and y_k given in the following form, respectively,

$$
\begin{aligned}
a_j &\equiv \sum_i w_{ji} z_i \\
z_j &\equiv g(a_j) \\
y_k &\equiv g(a_k)
\end{aligned}
$$

Where the functions $g(.)$ are called **activation functions**. These are generally **nonlinear transformations** applied to the **inputs** for the respective nodes.

Now, we are interested in calculating derivates w.r.t. the parameters of the network, which are given as the connection weights w_{ij} $\forall i, j \in Network(Edges)$.

$$\frac{\partial E}{\partial w_{ji}}$$

Which take (by simple application of the *chain rule of calculus*) the form,

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

We will make each of the terms on the right hand side precise now, distinguishing between **ouput nodes** denoted by subscripts k and **hidden nodes** denoted by subscripts j.

First, consider the right-most term, from our definition of a_j (above), we get that,

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

Second we denote in general,

$$\delta_j \equiv \frac{\partial E}{\partial a_j}$$

Se we can rewrite,

$$\frac{\partial E}{\partial w_{ji}} = \delta_j \frac{\partial a_j}{\partial w_{ji}}$$

Now we investigate the δ_j terms.

For **output nodes** we get,

$$\delta_k \equiv \frac{\partial E}{\partial a_k} = g^{'}(a_k)\frac{\partial E}{\partial y_k}$$

Where $\frac{\partial E}{\partial y_k}$ depends on the specific error function used.

For **hidden nodes** we get,

$$
\begin{aligned}
\delta_j \equiv \frac{\partial E}{\partial a_j} &= \sum_m \frac{\partial E}{\partial a_m}\frac{\partial a_m}{\partial a_j} = \sum_k \delta_m \frac{\partial a_m}{\partial a_j} \\
&= \sum_m \delta_m \frac{\partial \sum_i w_{mi} z_i}{\partial a_j} \\
&= \sum_m \delta_m \frac{\partial \sum_i w_{mi} g(a_i)}{\partial a_j} \\
&= \sum_m \delta_m w_{mj} g^{'}(a_j) \\
&= g^{'}(a_j) \sum_m \delta_m w_{mj}
\end{aligned}
$$

Where, the the sums range over all units m, to which our unit j sends informations (in other words, there exists a connection between node j and node m).

The equation,

$$
\delta_j = g^{'}(a_j) \sum_m \delta_m w_{mj}
$$

is known as the **back-propagation formula**.

Note, the recursive character of the equation. To evaluate δ_j, we need to have access to δ_m, for all $m \in M$, where $M = \{m \in Network(Nodes) : j \rightarrow m \in Network(Edges)\}$. The implication is that we need to evaluate the δ_j terms, starting from the output nodes backwards (hence *back* propagation). This works because for the output nodes we can evaluate the corresponding equations directly, hence we are able to start the recursion.

Remarks

1. The algorithm provided is fully general for feed-forward topologies. The only requirement is that the corresponding derivatives can be evaluated
2. The above derivation abstracts away the actual application case of running through many training examples when evaluating the gradients. However, this is immaterial, because, under the usual assumption that the data are *i.i.d* from an underlying distribution ρ, the following **total error** gradient is justified $\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^n}{\partial w_{ji}}$. Hence, nothing changes in our derivation of the gradients, we simply add together the computed gradients evaluated at the respective training examples.

3. To see the value of the *algorithm*, it may be useful to contrast what is written below, with a naive application of the chain-rule to calculate derivatives with respect to each parameter in a neural network. Even simpler, just consider numerically evaluating those derivates. Doing so entails applying a small perturbation to each weight in the weight-vector of the neural network, one by one, and evaluating the corresponding change in the *objective function*. Each of those forward propagations, has computational cost $O(W)$ hence such a procedure would cost us on the order of $O(W^2)$.