Supervised Training

While one could create an artificial neural network by hand, it is very difficult to set the weights of a network appropriately. Instead, we want to use a training method that learns the weights from a training set consisting of labeled examples.

- Training set: a set of labeled data containing examples of all classes the network needs to see.
- Testing set: a similarly labeled data set on which the network never trains, but which is used to evaluate the network’s performance.

The most common training method for feedforward neural networks is called backpropagation. It is a gradient descent algorithm that modifies the weights to improve the performance of the network on the training set. Gradient descent algorithms are greedy iterative algorithms that take one step towards the solution each iteration, making the change that maximally minimizes an error metric.

The process of training using a single pattern is as follows:

- Apply the training pattern to the network (i.e. forward propagate a data point as input)
- Calculate the output of the network and compare it to the desired output (the known class).
- Calculate the change to the weights for the output layer to make the actual and desired outputs closer.
- Move back through the network, calculating how to change the weights at each layer.

Applying all of the training patterns to the network is called an epoch. There are three algorithms for an epoch of training

1. On-line:
   For each training pattern (input),
   - Apply the training patterns as above
   - Apply the weight changes to the network, modified by a learning rate

2. Stochastic: This is the same as on-line, but randomly permutes the order in which the training patterns are applied.
3. Batch:

- Apply all of the training patterns as above and store (but don’t apply) the weight changes
- Apply some function of the stored the weight changes to the network, modified by a
  learning rate. This function could be the sum, mean, or median.

Training can often take hundreds or thousands of epochs.

You can visualize the error surface of the network as a set of valleys, hills, and plateaus. Each point on the error surface represents a set of values for the network weights. The best network, given the training and testing data, is the one at the bottom of the deepest valley, the lowest point on the error surface.

Backpropagation training generally begins with the weights of the network set to small, non-zero random values. Imagine the network starting at some randomly selected point on the error surface. By calculating the impact on the error rate of changing each weight in a specific manner, the backpropagation algorithm can determine which changes correspond to the optimal step down the error surface. By iterating that process, the network moves down the error surface towards the nearest valley.

Problems can arise with any greedy gradient-descent algorithm when the error surface is bumpy or has lots of valleys other than the deepest one. Since the network moves towards the closest valley, where the network starts can determine whether it can learn the problem optimally. The situation is similar to K-means clustering, where the initial cluster means end up determining how the data set is divided.

As with K-means clustering, one method of finding the global optimum is to train the network multiple times, starting it in different locations on the error surface each time. Other methods modify the backpropagation algorithm by introducing terms into the optimization that allow it to skip over or bounce out of local minima (valleys).

Algorithm for a single training pattern

1. Execute a forward pass on a training pattern to calculate the output of the network $O$.
2. Calculate the mean squared error between the output $O$ and the desired output $T$.

\[
E = \frac{1}{N_O} \sum_h (o_h - t_h)^2
\]

where $N_O$ is the number of output nodes and $h$ takes on the values of their indices.

3. Calculate the weight changes required to move $O$ closer to $T$

- Weight changes are calculated using the **generalized delta rule**

\[
\Delta w_{ji} = \eta \delta_i o_j
\]

- $w_{ji}$ is the weight going from node $j$ to node $i$
- $\Delta w_{ji}$ is the change that should be made to $w_{ji}$
- $\eta$ is the learning constant, and governs the magnitude of the weight changes
– $\delta_i$ relates the input of node $i$ to its influence on the output
– $o_j$ is the output of node $j$

- The weight change ought to be proportional to the amount of change in the overall error realized by changing the weight. In other words, we need to know the derivative of the error with respect to changes in the weight.

$$\Delta w_{ji} \propto -\frac{\partial E}{\partial w_{ji}}$$  \hspace{1cm} (3)

- To figure out this derivative, we need to have a complete expression connecting the inputs and outputs of a node. The overall input to a node is a weighted sum of the outputs of all connected nodes.

$$I_i = \sum_k w_{ki}o_k$$  \hspace{1cm} (4)

where $k$ takes on the values of the indices of all of the nodes that send output to node $i$.

- Given the expression for the overall input to a node, the output of a node is simply

$$o_j = \Phi(I_j) = \frac{1}{1 + e^{-I_j}}$$  \hspace{1cm} (5)

- The derivative of the sigmoid function is simple to calculate.

$$\Phi'(x) = \Phi(x)(1 - \Phi(x))$$  \hspace{1cm} (6)

- Now we can divide the derivative from above into two parts, reflecting the change in the error with respect to the input of a node and the change in the input to a node with respect to a single weight.

$$-\frac{\partial E}{\partial w_{ji}} = -\frac{\partial E}{\partial I_i} \frac{\partial I_i}{\partial w_{ji}}$$  \hspace{1cm} (7)

- The second term in (7) is the derivative of the input equation (4)

$$\frac{\partial I_i}{\partial w_{ji}} = \sum_k w_{ki}o_k = o_j$$  \hspace{1cm} (8)

and explains the $o_j$ in the generalized delta rule. (Note about the math: $j = w$ for one term and that is the term for which the derivative is non-zero.)

- The $\delta$ term in the delta rule corresponds to the remaining partial derivative, which we can again expand.

$$-\frac{\partial E}{\partial I_i} = -\frac{\partial E}{\partial o_i} \frac{\partial o_i}{\partial I_i} = -\frac{\partial E}{\partial o_i} \Phi'(I_i)$$  \hspace{1cm} (9)

- The second term of (9) is given by $\Phi'(I_j)$. The first term is different for output units and hidden units.

- For an output node, the differential of the error term with respect to its output is given by (10).

$$\frac{\partial E}{\partial o_i} = \frac{\partial}{\partial o_i} \left( \frac{1}{N_O} \sum_h (o_h - t_h)^2 \right) = -(t_i - o_i)$$  \hspace{1cm} (10)

In other words, the weight change for a link coming into an output node depends upon how different the output is from the target. (Note about the math: $h = i$ for one term and that is the term for which the derivative is non-zero.)
• Combining (10) and (9), the delta value for an output node is given by

$$\delta_i = (t_i - o_i)\Phi'(I_i)$$  \hspace{1cm} (11)$$

• Computing the delta value for a hidden node is more challenging, because it must take into account all of the ways in which a single weight can affect the different between the network outputs and the target. If a hidden node connects to multiple output nodes, then changing a weight of a connection coming into a hidden node changes all of the output node values.

If there are \( H \) output nodes, we can use the chain rule to identify the relationships.

$$\frac{\partial E}{\partial o_i} = \sum_h \frac{\partial E}{\partial I_h} \cdot \frac{\partial I_h}{\partial o_i} = \sum_{h=1}^H \frac{\partial E}{\partial I_h} \cdot \sum_k w_{kh} o_k = \sum_h \frac{\partial E}{\partial I_h} w_{ih} = - \sum_h \delta_h w_{ih}$$  \hspace{1cm} (12)$$

The way to interpret (12) is that the \( \delta \) for a weight going into the hidden node is the sum of the \( \delta \) values of all the connections leaving the node multiplied by their respective weights.

• The generalized delta rule for a hidden layer is, therefore, as follows.

$$\delta_i = \Phi'(I_i) \sum_h \delta_h w_{ih}$$  \hspace{1cm} (13)$$

• The update for a weight connected to an output node is given in (14).

$$\Delta w_{ji} = \eta(t_i - o_i)\Phi'(I_i) o_j$$  \hspace{1cm} (14)$$

• The update for a weight connected to a hidden node is given in (15).

$$\Delta w_{ji} = \eta \left( \Phi'(I_i) \sum_h \delta_h w_{ih} \right) o_j$$  \hspace{1cm} (15)$$

Intuitively, the update for a weight \( w_{ij} \) connecting node \( i \) to hidden node \( j \) is proportional to \( o_i \) modified by the impact of any change in the weight on all of the \( K \) output nodes it affects.

One thing to note is that in both cases if \( o_i = 0 \) then there will be no weight adjustment. Therefore, that suggests avoiding training examples where the inputs are zero. For example, the network learns much better when the training samples for the XOR problem avoid exact zero as an input, instead using numbers close to zero.

Likewise, trying to train a network to achieve a true 0.0 or a true 1.0 is difficult, and often leads to overtraining because it pushes the outputs far into the saturation zone of the sigmoid function. Instead, having training examples where the maximum desired output is 0.9, for example, rather than 1.0, tends to result in faster training and a more generalized network.

4. The weight update step is accomplished by

$$w_{ji} = w_{ji} + \Delta w_{ji}$$  \hspace{1cm} (16)$$
Notes from Stephanie

The above notes were written by Bruce Maxwell in 2012 and modified by Stephanie Taylor in 2013.

A link I found helpful:

http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Also, you can read the Wikipedia entries on multilayer perceptrons and backpropagation.