6.858/18.428 Machine Learning	Lecture 15: November 7, 1994
Lecturer: Ron Rivest	*Scribe: Jerry Pratt*

15.1 Outline

- Neural Networks
- Back propagation

15.2 Neural Networks

Neural networks are used to learn arbitrary target functions which map a vector of real numbers to another vector of real numbers,

$$t : X \subset \Re^m \rightarrow Y \subset \Re^n$$

The neural network is trained on a set, S, of m examples $(\vec{x}, \vec{y})$ such that $\vec{y} = t(\vec{x})$. By training the net to do well on the given examples, we hope that it will do well later in predicting the value of new examples.

The most popular method of training neural networks is to use gradient descent to minimize the Mean Square Error of the training sample. Back propagation is a computationally clever means of doing this. These concepts are explained below.

15.2.1 Mean Squared Error (MSE)

Suppose we have a neural network which computes $f_w(\vec{x})$ but is trying to learn the target function $t(\vec{x})$. (Note that the function which the neural net computes depends on the architecture of the network, and on the weights of the network.) Given a fixed sample S, with m elements, we can define the Mean Squared Error (MSE) of the neural network on sample S to be:

$$E(\vec{w}) = \frac{1}{m} \sum_{\vec{x} \in S} (f_w(\vec{x}) - t(\vec{x}))^2$$

Since the target function and sample are fixed, the MSE is only a function of the weights of the network.

The goal of our training routine is thus to minimize the MSE. The MSE is a useful cost function for several reasons. It can be used for arbitrary functions, is well defined, and is easily computed. Most importantly, however, is that it has a computable, continuous derivative (as long as $f_w(\vec{x})$ has a computable, continuous derivative). This will play an important role in the training of the neural network. The major disadvantage of using the MSE is that "outliers" have a large effect on it.

For boolean functions, minimizing the MSE is equivalent to minimizing errors. This is not true in general however.

15.2.2 Gradient Descent

To minimize the MSE cost function, we use gradient descent, also known as "hill climbing". Gradient descent is done by taking small steps in the direction which reduces the error the most. In general, one starts with an initial weight vector $\vec{w}_o \in \Re^k$. At each iteration, the weight vector is updated:

$$\vec{w}_{i+1} \leftarrow \vec{w}_i + \epsilon \bigtriangleup \vec{w}_i$$

where ϵ is the learning rate parameter and $\bigtriangleup \vec{w}_i$ is the direction to move in. Essentially, we are taking a walk in weight space, always going down hill. The iteration is stopped after converging to a minima to within a given margin.

The direction which reduces the error the fastest is opposite the gradient ∇E, where

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_k} \right)$$

so that the update is

$$\vec{w}_{i+1} \leftarrow \vec{w}_i - \epsilon \nabla E$$

There are several important issues in using gradient descent which are discussed below.

Initial weight vector

The choice of the initial weight vector, $\vec{w}_o$, is important in determining whether gradient descent converges to a local minima or to the true global minima. There are several methods for choosing the initial vector. These include random generation, having a fixed vector, or choosing the initial vector. Random generation is most popular in neural networks. In general, there is no known way to tell, given an initial weight vector, whether gradient descent will converge to a global or local minima. In practice, all that can be done is determine when a local minima is hit (when no more progress is being made) and start over with a new initial weight vector.

Learning parameter

The choice of a learning parameter ϵ determines whether or not gradient descent will converge to a minima, and how long it will take. If ϵ is too small, gradient descent will progress too slowly. If it is too large, gradient descent will overshoot the minima and hence possibly not converge. There is no systematic method for picking the "right" ϵ. However, there are adaptive methods and also methods for dynamically adjusting it during learning.

Batch vs. Incremental processing

Given a sample set, S, gradient descent can either be done on a point-by-point basis or by batch processing. If batch processing is done, the gradient is computed as follows:

$$\nabla E(S) = \frac{1}{m}\left(\nabla E(\{\vec{x}_1\}) + \nabla E(\{\vec{x}_2\}) + \ldots + \nabla E(\{\vec{x}_m\})\right)$$

Momentum

Gradient descent gets in trouble when it encounters narrow, slowly sloping valleys. In such a situation, it will tend to oscillate between the sides of the valley while slowly making progress down it. The short term oscillations can be cancelled out by maintaining some of the "momentum" of the previous updates. This is done by computing the direction to move in in the following way:

$$\triangle \vec{w}_{i+1} = \alpha \triangle \vec{w}_i - (1-\alpha)(\nabla E)$$

where $0 < \alpha \leq 1$. This effectively is a low pass filter. If $\alpha = 0$, no filtering occurs.

Convergence time

Convergence time may be decreased if the hessian is used instead of the gradient. The hessian is more accurate because it retains second order terms in addition to the first order terms of the gradient.

Using a dynamically adjusted variable step size may also decrease the convergence time.

When to stop training

Training is usually stopped when the MSE seems to have converged to a (local) minimum. However, in some cases it may be best to stop training early. Prediction of samples not contained in the training set may be more accurate if training is stopped short.

15.2.3 Application: Linear Functions

The following example shows how to use gradient descent to minimize the Mean Square Error when learning the linear function

$$f_w(\vec{x}) = \vec{w} \cdot \vec{x}$$

where $\vec{w}, \vec{x} \in \Re^k$. This function is similar to a perceptron except that there is no threshold. Using batch processing, the MSE and gradient can be calculated and the weight update made:

$$E(\vec{w}) = \frac{1}{m} \sum_i (\vec{w} \cdot \vec{x}_i - y_i)^2$$

$$\nabla E = \frac{\delta E(\vec{w})}{\delta \vec{w}} = \frac{2}{m} \sum_i (\vec{w} \cdot \vec{x}_i - y_i) \vec{x}_i$$

$$\vec{w} \leftarrow \vec{w} - \epsilon \nabla E$$

If single point processing is used, we have:

$$\vec{w} \leftarrow \vec{w} - \epsilon (\vec{w} \cdot \vec{x} - y) \vec{x}$$

This last update equation is very similar to the perceptron training rule except now the update is proportional to the error.

15.3 Back Propagation

By using the chain rule of differentiation, we can show that the derivative of the MSE with respect to the weight vector of a given node ($\frac{\delta E}{\delta \vec{w}_i}$) can be computed with access only to information found in the node and information in the nodes directly connected to the output of the given node. Therefore gradient information can be propagated backward through the network in order to update the weights. Hence the name back propagation. In order to propagate gradient information back through the threshold functions, it is important that they have a continuous derivative which is easily computed. The following sigmoid functions have such properties.

15.3.1 Sigmoid Functions

Sigmoid functions are similar to the threshold function except that they are smooth and have a computable, continuous derivative. Other properties of a sigmoid function is that its range is the real numbers, its domain is a bounded interval, and it is monotonic. Two popular sigmoid functions are the hyperbolic tangent and the logistic function.

Hyperbolic tangent

The hyperbolic tangent sigmoid function is

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

Its derivative is

$$\frac{\delta z}{\delta x} = \mathrm{sech}^2(x) = \frac{4}{(e^x + e^{-x})^2}$$

The domain of the hyperbolic tangent is $\Re$ and the range is the interval $[-1, 1]$. See Figure 15.1 for a graph of the hyperbolic tangent (solid) and its derivative (dotted).

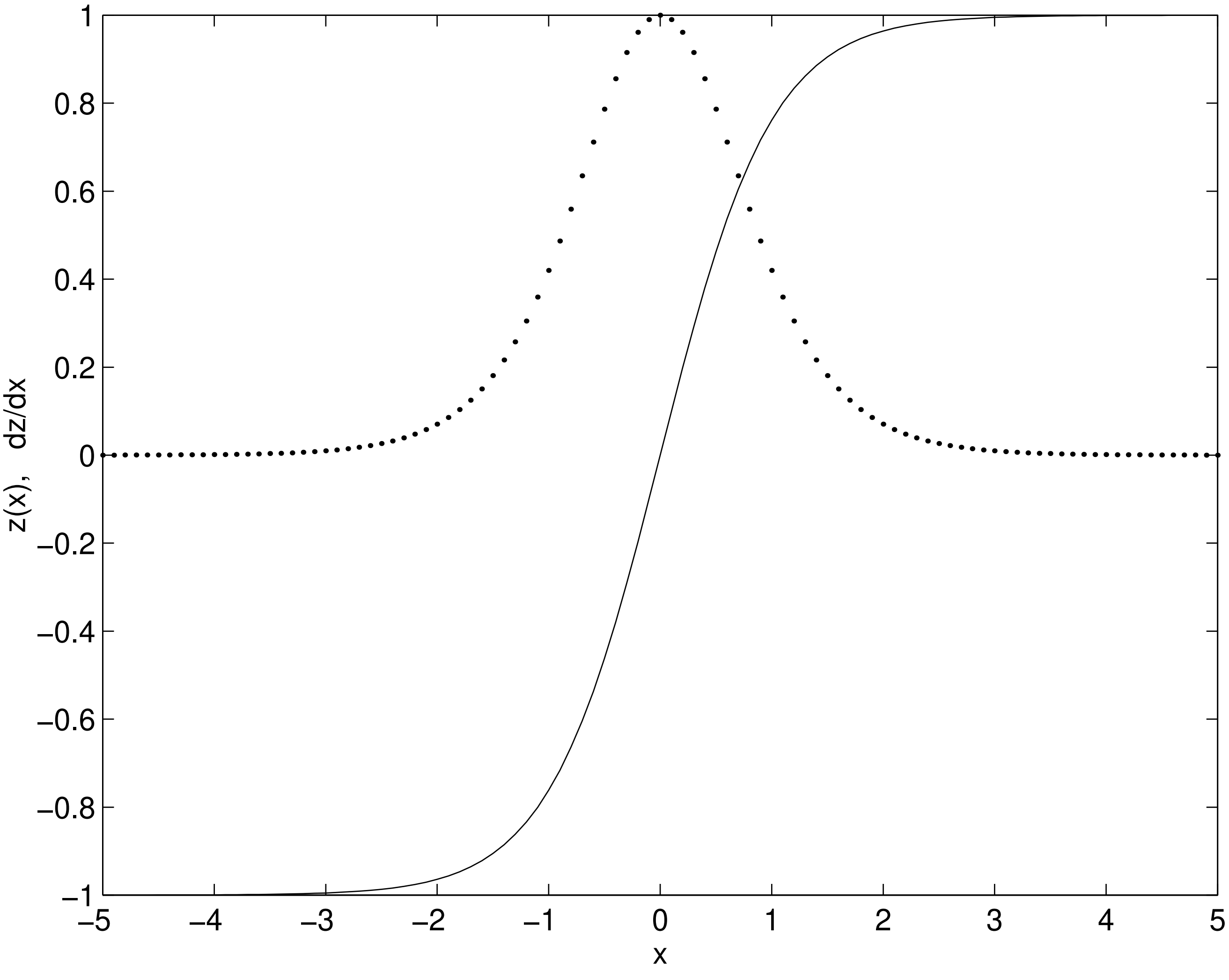

Figure 15.1: Hyperbolic tangent sigmoid function (solid line) and derivative (dotted line).

Logistic function

The logistic sigmoid function is

$$z = \frac{1}{1 + e^{-x}}$$

Its derivative is

$$\frac{\delta z}{\delta x} = z(1 - z)$$

The domain of the logistic function is $\Re$ and the range is the interval $[0, 1]$. See Figure 15.2 for a graph of the logistic function (solid) and its derivative (dotted). The logistic function is popular because its derivative can be written in terms of its output. Since its output is computed in order to compute the output of the network, not much more computation is necessary in computing the derivative.

15.3.2 Chain Rule

Let y be a function of x, $y = f_w(x)$. The chain rule states that

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y}\frac{\delta y}{\delta x}$$

For the following, lets use the notation $\delta_x = \frac{\partial E}{\partial x}$. Then the chain rule states $\delta_x = \delta_y \frac{\delta y}{\delta x}$. It is conceptually easier to see how back propagation works, via the chain rule, if a node of the network is redrawn in its expanded form, as in Figure 15.3.

Figure 15.4 contains all of the elementary blocks which are used in building a neural network. Using the chain rule we can calculate how the partial derivatives are propagated backward through each block:

a) $y = x^2$

$$\delta_x = 2\delta_y x$$

b) $y = \sigma(x)$

logistic:

$$\delta_x = \delta_y y(1 - y)$$

hyperbolic tangent:

$$\delta_x = \delta_y \mathrm{sech}^2(x)$$

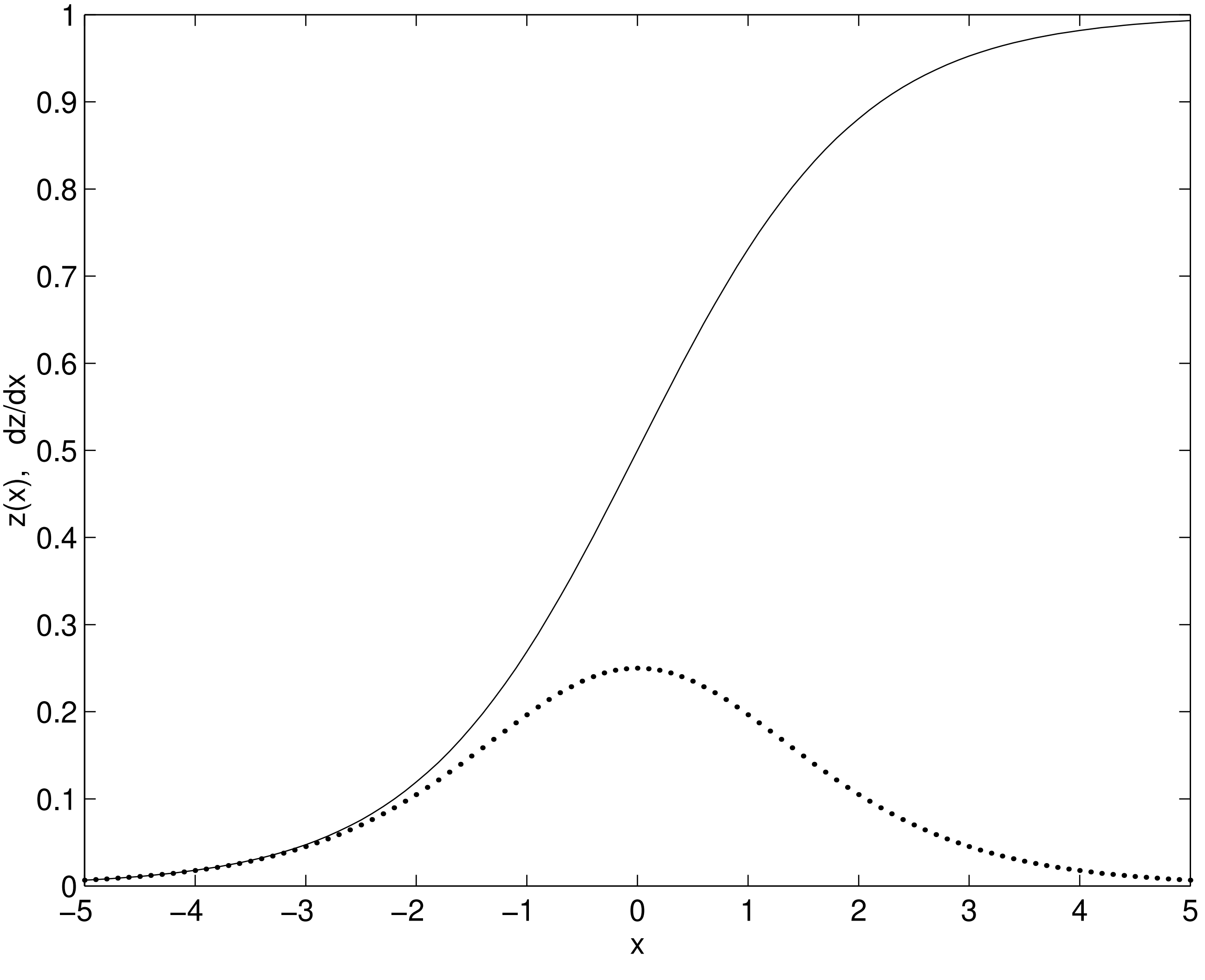

Figure 15.2: Logistic sigmoid function (solid line) and derivative (dotted line).

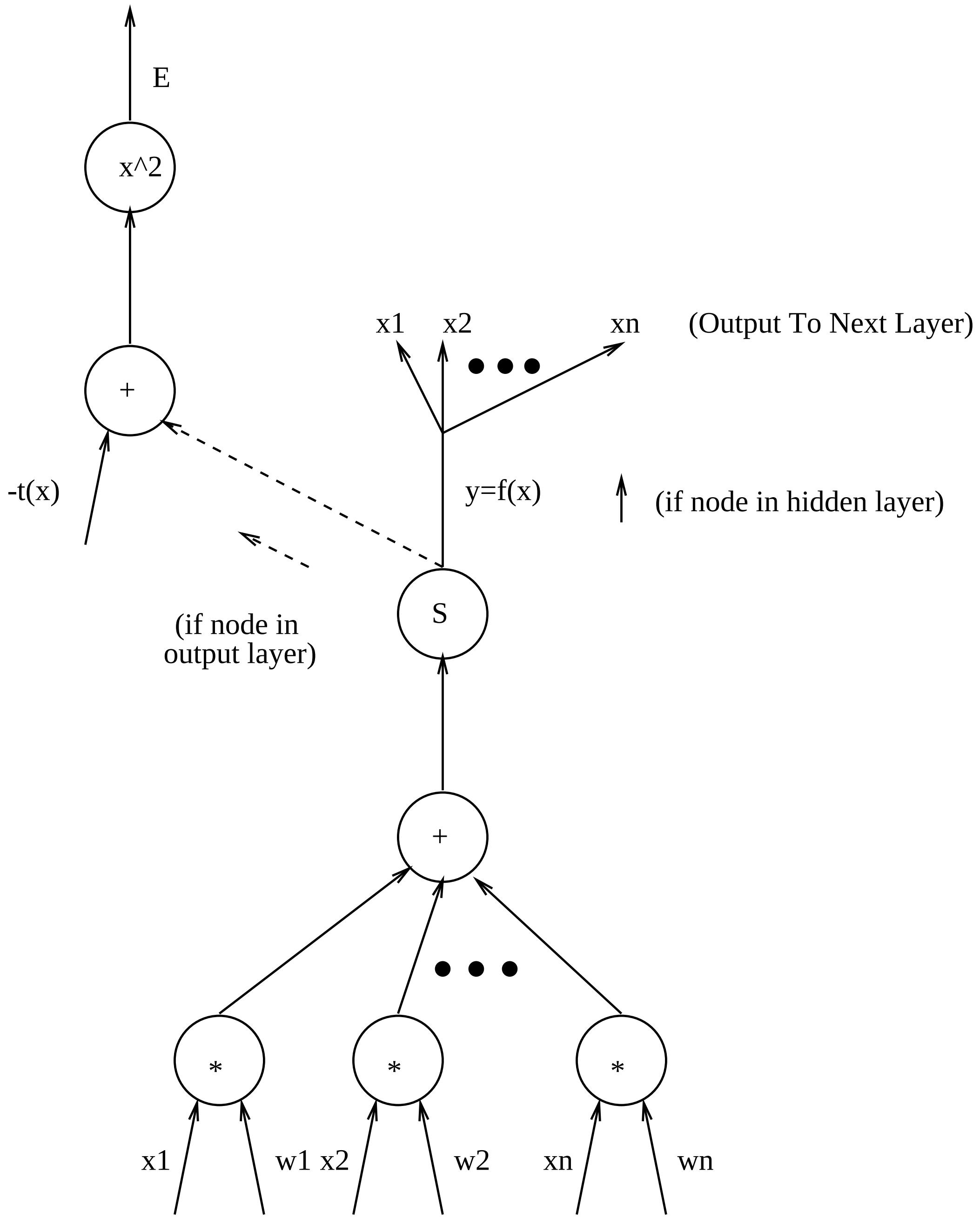

Figure 15.3: Expanded node of network for visualization of back propagation. If the node is an output node then we can make the network compute the mean squared error, by subtracting the target value from the value output at the node, and squaring this value.

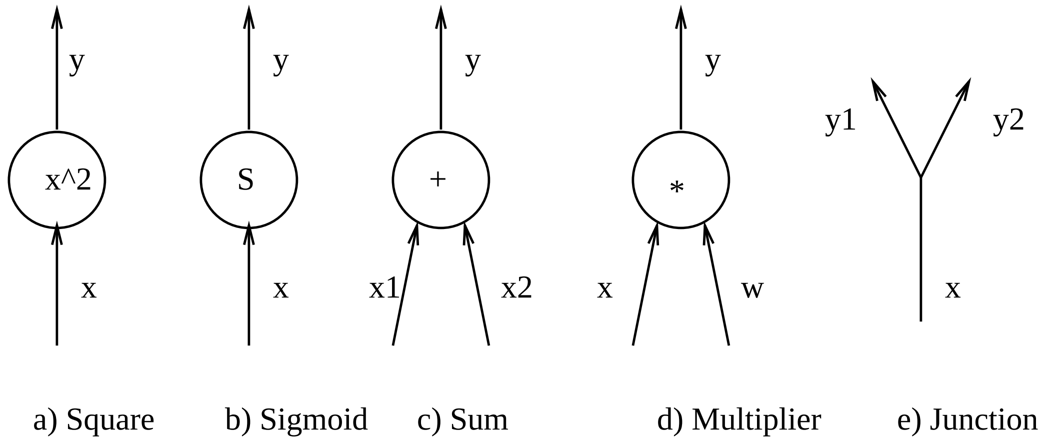

Figure 15.4: Five different blocks found in neural networks.

c) $y = x_1 + x_2$

$$\delta_{x_1} = \delta_y$$
$$\delta_{x_2} = \delta_y$$

d) $y = wx$

$$\delta_x = \delta_y w$$
$$\delta_w = \delta_y x$$

e) $y_1 = x, y_2 = x$

$$\delta_x = \delta_{y_1} + \delta_{y_2}$$

By combining the blocks, as in Figure 15.3, we can now compute how the partial derivatives propagate backward through an entire node:

If we are using a logistic sigmoid function we have:

$$\delta_{x_i} = \delta_y y(1-y)w_i$$
$$\delta_{w_i} = \delta_y y(1-y)x_i$$

With a hyperbolic tangent sigmoid function, we have:

$$\delta_{x_i} = \delta_y \text{sech}^2(x_i)w_i$$
$$\delta_{w_i} = \delta_y \text{sech}^2(x_i)x_i$$

For hidden nodes, we have:

$$\delta_y = \sum \delta_{x_i}$$

where the sum is over the x_i of the nodes in the next higher layer that y branches out to.

For the output nodes, we have

$$\delta_y = 2(y_d - y) = 2(t(\vec{x}) - f_w(\vec{x}))$$

15.3.3 Back propagation Algorithm

We can now write the pseudocode for a back propagation algorithm. The following uses the logistic sigmoid function, a constant learning parameter, ϵ, and batch processing.

Until progress is stopped
{
    For each input $\vec{x}$
    {
        Compute the output, y, of each node in a forward manner.
        For each node, in a backward manner,
        {
            Compute $\delta_y(\vec{x})$ using $\delta_y(\vec{x}) = 2(t(\vec{x}) - f_w(\vec{x}))$ if node in output layer or
                $\delta_y(\vec{x}) = \sum \delta_{x_i}(\vec{x})$ if node in hidden layer.
            For each input x_i to the node, compute $\delta_{x_i}(\vec{x}) = \delta_y(\vec{x})y(1-y)w_i$
            For each weight w_i on edges to the node, compute $\delta_{w_i}(\vec{x}) = \delta_y(\vec{x})y(1-y)x_i$
        }
    }
    Add up the $\delta_w(\vec{x})$ for each input $\vec{x}$ and update the weights using
    $w_i \leftarrow w_i - \frac{1}{m}\epsilon \sum_{\vec{x} \in S} \delta_{w_i}(\vec{x})$
}

References

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. *Parallel Distributed Processing - Explorations in the Microstructure of Cognition*, pages 318-362, MIT Press, 1986.