Hebbian Learning and Gradient Descent Learning

Introduction to Neural Networks : Lecture 5

1. Hebbian Learning
2. Learning by Error Minimisation
3. Computing Gradients and Derivatives
4. Gradient Descent Learning
5. Deriving the Delta Rule
6. Delta Rule vs. Perceptron Learning Rule

Hebbian Learning

In 1949 neuropsychologist Donald Hebb postulated how biological neurons learn:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place on one or both cells such that A's efficiency as one of the cells firing B, is increased."

In other words:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.

This rule is often supplemented by:

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

so that chance coincidences do not build up connection strengths.

Hebbian versus Perceptron Learning

In the notation used for Perceptrons, the ***Hebbian learning*** weight update rule is:

$$\Delta w_{ij} = \eta \,.\, out_j \,.\, in_i$$

There is strong physiological evidence that this type of learning does take place in the region of the brain known as the *hippocampus*.

Recall that the ***Perceptron learning*** weight update rule we derived was:

$$\Delta w_{ij} = \eta .\, (targ_j - out_j) \,.\, in_i$$

There is some similarity, but it is clear that Hebbian learning is not going to get our Perceptron to learn a set of training data.

The are variations of Hebbian learning that do provide powerful learning techniques for biologically plausible networks, such as ***Contrastive Hebbian Learning***, but we shall adopt another approach for formulating learning algorithms for our networks.

Learning by Error Minimisation

The Perceptron Learning Rule is an algorithm for adjusting the network weights w_{ij} to minimise the difference between the actual outputs out_j and the desired outputs $targ_j$.

We can define an ***Error Function*** to quantify this difference:

$$E(w_{ij}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j - out_j\right)^2$$

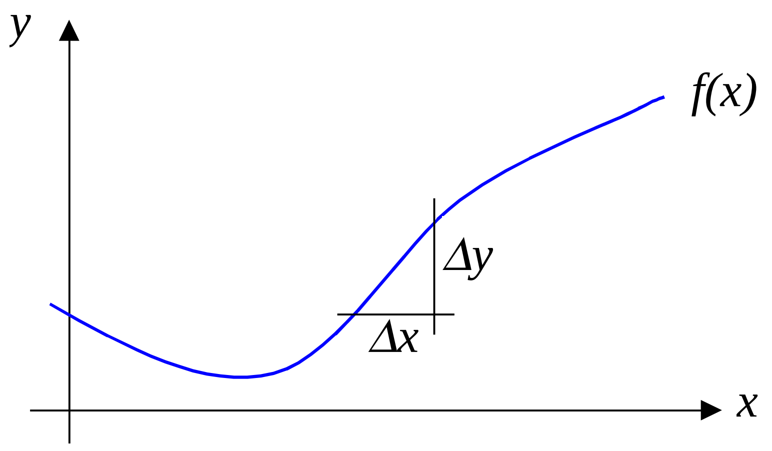
For obvious reasons this is known as the ***Sum Squared Error*** function. It is the total squared error summed over all output units j and all training patterns p.

The aim of ***learning*** is to minimise this error by adjusting the weights w_{ij}. Typically we make a series of small adjustments to the weights $w_{ij} \rightarrow w_{ij} + \Delta w_{ij}$ until the error $E(w_{ij})$ is 'small enough'.

A systematic procedure for doing this requires the knowledge of how the error $E(w_{ij})$ varies as we change the weights w_{ij}, i.e. the ***gradient*** of E with respect to w_{ij}.

Computing Gradients and Derivatives

There is a whole branch of mathematics concerned with computing gradients – it is known as ***Differential Calculus***. The basic idea is simple. Consider a function $y = f(x)$



The gradient, or rate of change, of $f(x)$ at a particular value of x, as we change x can be approximated by $\Delta y/\Delta x$. Or we can write it exactly as

$$\frac{\partial f(x)}{\partial x} = \operatorname*{Lim}_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \operatorname*{Lim}_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

which is known as the ***partial derivative*** of $f(x)$ with respect to x.

Examples of Computing Derivatives Analytically

Some simple examples should make this clearer:

$$f(x) = a.x + b \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{[a.(x+\Delta x)+b]-[a.x+b]}{\Delta x} = a$$

$$f(x) = a.x^2 \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{\left[a.(x+\Delta x)^2\right]-\left[a.x^2\right]}{\Delta x} = 2ax$$

$$f(x) = g(x) + h(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{\left(g(x+\Delta x)+h(x+\Delta x)\right)-\left(g(x)+h(x)\right)}{\Delta x} = \frac{\partial g(x)}{\partial x} + \frac{\partial h(x)}{\partial x}$$

Other derivatives can be computed in the same way. Some useful ones are:

$$f(x) = a.x^n \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = nax^{n-1} \qquad\qquad f(x) = \log_e(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \frac{1}{x}$$

$$f(x) = e^{ax} \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = ae^{ax} \qquad\qquad f(x) = \sin(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \cos(x)$$

Gradient Descent Minimisation

Suppose we have a function *f(x)* and we want to change the value of *x* to minimise *f(x)*. What we need to do depends on the gradient of *f(x)*. There are three cases to consider:

If	$\frac{\partial f}{\partial x} > 0$	then	*f(x)* increases as *x* increases	so	we should decrease *x*
If	$\frac{\partial f}{\partial x} < 0$	then	*f(x)* decreases as *x* increases	so	we should increase *x*
If	$\frac{\partial f}{\partial x} = 0$	then	*f(x)* is at a maximum or minimum	so	we should not change *x*

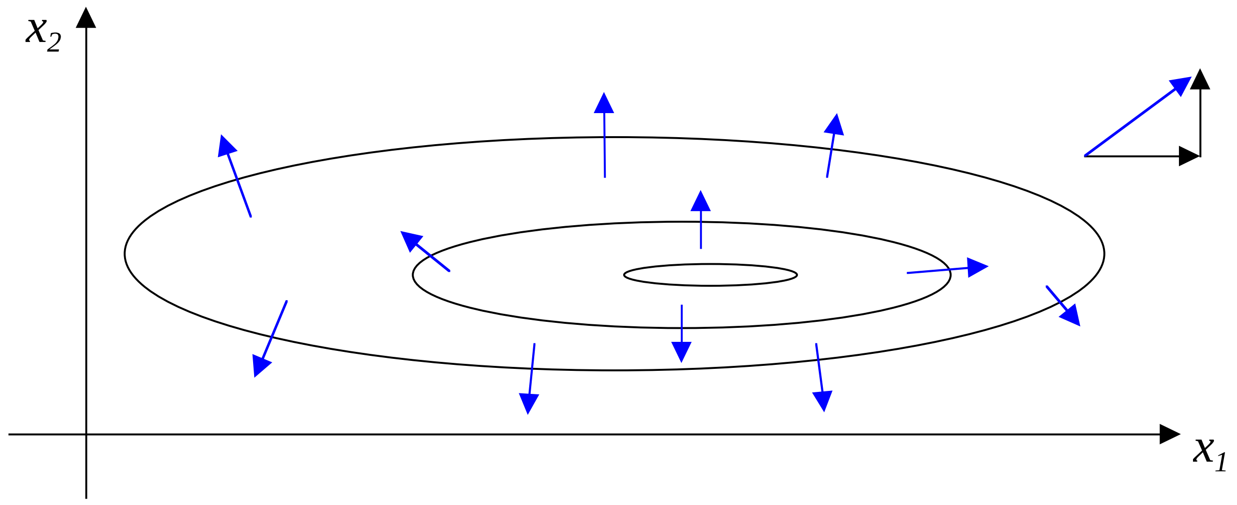
In summary, we can decrease *f(x)* by changing *x* by the amount:

$$\Delta x = x_{new} - x_{old} = -\eta \frac{\partial f}{\partial x}$$

where η is a small positive constant specifying how much we change *x* by, and the derivative $\partial f/\partial x$ tells us which direction to go in. If we repeatedly use this equation, *f(x)* will (assuming η is sufficiently small) keep descending towards its minimum, and hence this procedure is known as ***gradient descent minimisation***.

Gradients in More Than One Dimension

Is it obvious that we need the gradient/derivative itself in the weight update equation, rather than just the sign of the gradient? Consider the two dimensional function shown as a ***contour plot*** with its minimum inside the smallest ellipse:



A few representative gradient vectors are shown. By definition, they will always be perpendicular to the contours, and the closer the contours, the larger the vectors. It is now clear that we need to take the relative magnitudes of the x_1 and x_2 components of the gradient vectors into account if we are to head towards the minimum efficiently.

Gradient Descent Error Minimisation

Remember that we want to train our neural networks by adjusting their weights w_{ij} in order to minimise the error function:

$$E(w_{ij}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j - out_j\right)^2$$

We now see it makes sense to do this by a series of gradient descent weight updates:

$$\Delta w_{kl} = -\eta \frac{\partial E(w_{ij})}{\partial w_{kl}}$$

If the transfer function for the output neurons is *f(x)*, and the activations of the previous layer of neurons are in_i, then the outputs are $out_j = f(\sum_i in_i w_{ij})$, and

$$\Delta w_{kl} = -\eta \frac{\partial}{\partial w_{kl}}\left[\tfrac{1}{2}\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij})\right)^2\right]$$

Dealing with equations like this is easy if we use the chain rules for derivatives.

Chain Rules for Computing Derivatives

Computing complex derivatives can be done in stages. First, suppose *f(x) = g(x).h(x)*

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{g(x+\Delta x).h(x+\Delta x) - g(x).h(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{\left(g(x) + \frac{\partial g(x)}{\partial x}\Delta x\right).\left(h(x) + \frac{\partial h(x)}{\partial x}\Delta x\right) - g(x).h(x)}{\Delta x}$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial g(x)}{\partial x} h(x) + g(x)\frac{\partial h(x)}{\partial x}$$

We can similarly deal with nested functions. Suppose *f(x) = g(h(x))*

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{g(h(x+\Delta x)) - g(h(x))}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(h(x) + \frac{\partial h(x)}{\partial x}\Delta x) - g(h(x))}{\Delta x}$$

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{g(h(x)) + \frac{\partial g(x)}{\partial h(x)}\Delta h(x) - g(h(x))}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(h(x)) + \frac{\partial g(x)}{\partial h(x)}\left(\frac{\partial h(x)}{\partial x}\Delta x\right) - g(h(x))}{\Delta x}$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial g(h(x))}{\partial h(x)} \cdot \frac{\partial h(x)}{\partial x}$$

Using the Chain Rule on our Weight Update Equation

The algebra gets rather messy, but after repeated application of the chain rule, and some tidying up, we end up with a very simple weight update equation:

$$\Delta w_{kl} = -\eta \frac{\partial}{\partial w_{kl}} \left[\tfrac{1}{2} \sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right)^2 \right]$$

$$\Delta w_{kl} = -\eta \left[\tfrac{1}{2} \sum_p \sum_j \frac{\partial}{\partial w_{kl}} \left(targ_j - f(\sum_i in_i w_{ij}) \right)^2 \right]$$

$$\Delta w_{kl} = -\eta \left[\tfrac{1}{2} \sum_p \sum_j 2 \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(-\frac{\partial}{\partial w_{kl}} f(\sum_m in_m w_{mj}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{nj}) \frac{\partial}{\partial w_{kl}} (\sum_m in_m w_{mj}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{nj})(\sum_m in_m \frac{\partial w_{mj}}{\partial w_{kl}}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{ij})(\sum_m in_m \delta_{mk} \delta_{jl}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{nj})(in_k \delta_{jl}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \left(targ_l - f(\sum_i in_i w_{il}) \right) \left(f'(\sum_n in_n w_{nl})(in_k) \right) \right]$$

$$\Delta w_{kl} = \eta \sum_p \left(targ_l - out_l \right) . f'(\sum_n in_n w_{nl}) . in_k$$

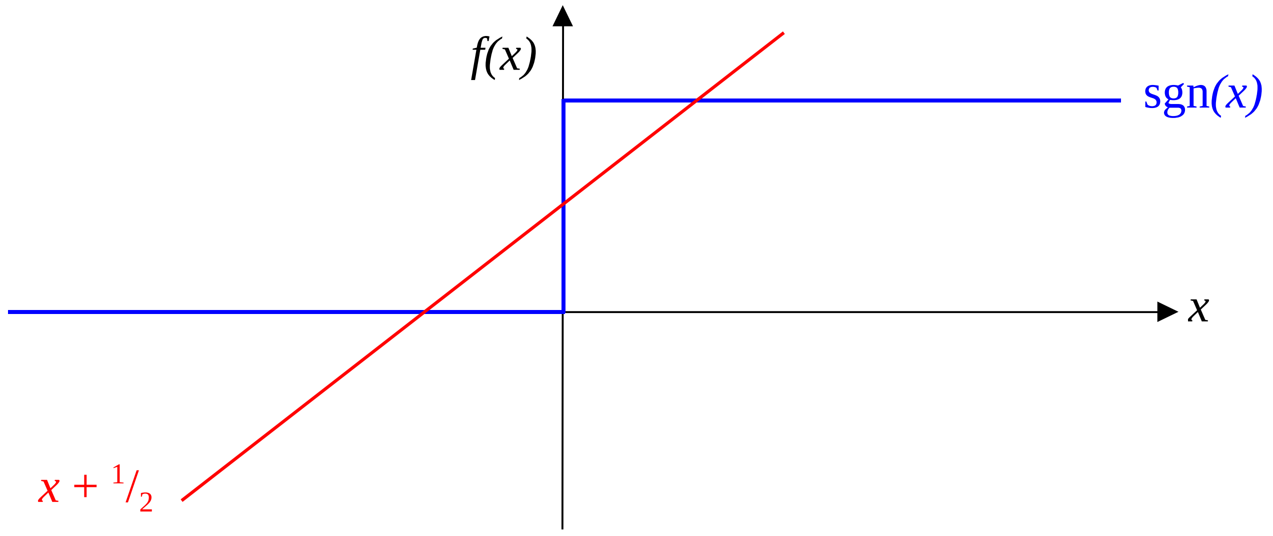
The *prime notation* is defined such that f' is the derivative of f. We have also used the *Kronecker Delta* symbol δ_{ij} defined such that $\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ when $i \neq j$.

The Delta Rule

We now have the basic gradient descent learning algorithm for single layer networks:

$$\Delta w_{kl} = \eta \sum_p \left(targ_l - out_l\right).f'\left(\sum_i in_i w_{il}\right).in_k$$

Notice that it still involves the derivative of the transfer function *f(x)*. This is clearly problematic for the simple Perceptron that uses the step function sgn(x) as its threshold function, because this has zero derivative everywhere except at $x = 0$ where it is infinite.



Fortunately, there is a ***clever trick*** we can use that will be apparent from the above graph. Suppose we had the transfer $f(x) = x + {}^1/_2$, then when the target is 1 the network will learn $x = {}^1/_2$, and when the target is 0 it will learn $x = -{}^1/_2$. It is clear that these values will also result in the right values of sgn(x), and so the Perceptron will work properly.

In other words, we can use the gradient descent learning algorithm with $f(x) = x + {}^1/_2$ to get our Perceptron to learn the right weights. In this case $f'(x) = 1$ and so the weight update equation becomes:

$$\Delta w_{kl} = \eta \sum_p \left(targ_l - out_l \right) . in_k$$

This is known as the ***Delta Rule*** because it depends on the discrepancy

$$\delta_l = targ_l - out_l$$

NOTE: We need to be very careful when using tricks like this. We are using one output function to learn the weights, i.e. $f(x) = x + {}^1/_2$, and a totally different one to produce the required binary outputs of the perceptron, i.e. $f(x) = \mathrm{sgn}(x)$. It is easy to get confused!

Delta Rule vs. Perceptron Learning Rule

We can see that the Delta Rule and the Perceptron Learning Rule for training Single Layer Perceptrons have exactly the same weight update equation:

$$\Delta w_{kl} = \eta \sum_p \left(targ_l - f\left(\sum_i in_i w_{il} \right) \right) . in_k$$

However, there are significant underlying differences. The Perceptron Learning Rule uses the actual activation function $f(x) = \mathrm{sgn}(x)$, whereas the Delta Rule uses the linear function $f(x) = x + {}^1/_2$. The two algorithms were also obtained from very different theoretical starting points. The Perceptron Learning Rule was derived from a consideration of how we should shift around the decision hyper-planes, while the Delta Rule emerged from a gradient descent minimisation of the Sum Squared Error.

The Perceptron Learning Rule will converge to zero error and no weight changes in a finite number of steps if the problem is linearly separable, but otherwise the weights will keep oscillating. On the other hand, the Delta Rule will (for sufficiently small η) always converge to a set of weights for which the error is a minimum, though the convergence to the precise values of $x = \pm{}^1/_2$ will generally proceed at an ever decreasing rate.

Overview and Reading

1. We began with a brief look at Hebbian Learning.
2. We then saw how neural network weight learning could be put into the form of minimising an appropriate output error function.
3. We then learnt how to compute the gradients/derivatives that would enable us to formulate efficient error minimisation algorithms.
4. Finally, we saw how gradient descent minimisation procedures could be used to derive the Delta Rule for training Simple Perceptrons, and compared it with the Perceptron Learning Rule.

Reading

1. Gurney: Sections 5.1, 5.2, 5.3
2. Beale & Jackson: Section 4.4
3. Callan: Sections 2.1, 2.2
4. Haykin: Sections 2.2, 2.4, 3.3
5. Bishop: Sections 3.1, 3.2, 3.3, 3.4, 3.5