Hebbian Learning and Gradient Descent Learning

Neural Computation : Lecture 5

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.

In the notation used for Perceptrons, this ***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*.

Modified Hebbian Learning

An obvious problem with this rule is that it is unstable – chance coincidences will build up the connection strengths, and all the weights will tend to increase indefinitely. Consequently, the basic learning rule (1) 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.

Another way to stop the weights increasing indefinitely involves normalizing them so they are constrained to lie between 0 and 1. This is preserved by the weight update

$$\Delta w_{ij} = \frac{w_{ij} + \eta.out_j.in_i}{\left(\sum_k (w_{kj} + \eta.out_j.in_k)^2\right)^{1/2}} - w_{ij}$$

which, using a small η and linear neuron approximation, leads to ***Oja's Learning Rule***

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

which is a useful stable form of Hebbian Learning.

Hebbian versus Perceptron Learning

It is instructive to compare the Hebbian and Oja learning rules with the ***Perceptron learning*** weight update rule we derived previously, namely:

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

There is clearly some similarity, but the absence of the target outputs $targ_j$ means that Hebbian learning is never going to get a Perceptron to learn a set of training data.

There exist variations of Hebbian learning, such as ***Contrastive Hebbian Learning***, that do provide powerful supervised learning for biologically plausible networks.

However, it has been show that, for many relevant cases, much simpler non-biologically plausible algorithms end up producing the same functionality as these biologically plausible Hebbian-type learning algorithms.

For the purposes of this module, we shall therefore pursue simpler non-Hebbian approaches for formulating learning algorithms for our artificial neural networks.

Learning by Error Minimisation

The Perceptron Learning Rule is an algorithm that adjusts 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*** E to quantify this difference, for example:

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

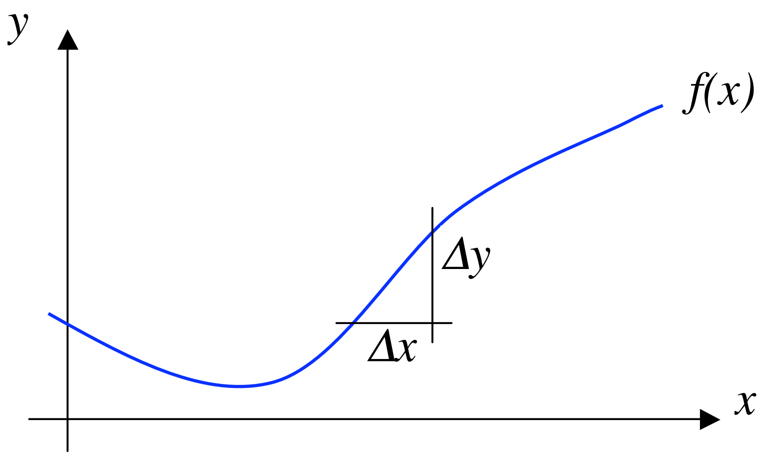
For obvious reasons this is known as the ***Sum Squared Error*** (SSE) function. It is the total squared error summed over all output units j and all training patterns p.

The aim of ***learning*** is to minimise such an error measure 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

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

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 particularly 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

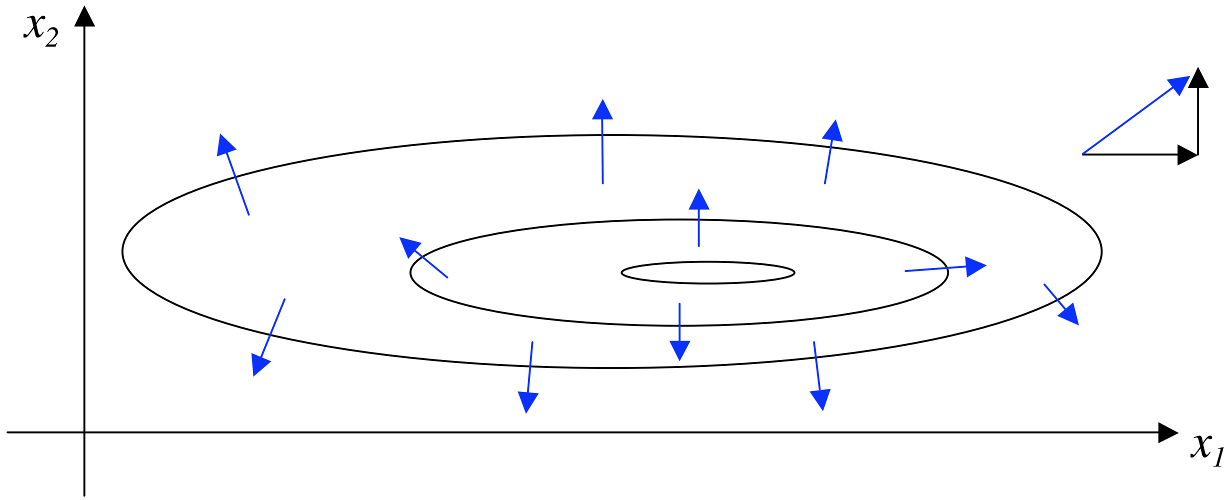
In summary, we can decrease $f(x)$ by changing x by the amount:

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

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.

Training a Single Layer Feed-forward Network

Now we understand how gradient descent weight update rules can lead to minimisation of a neural network's output errors, it is straightforward to train any network:

1. Take the set of training patterns you wish the network to learn

   $\{in_i^p, out_j^p : i = 1 \dots ninputs, j = 1 \dots noutputs, p = 1 \dots npatterns\}$
2. Set up the network with *ninputs* input units fully connected to *noutputs* output units via connections with weights w_{ij}
3. Generate random initial weights, e.g. from the range $[-smwt, +smwt]$
4. Select an appropriate error function $E(w_{ij})$ and learning rate η
5. Apply the weight update $\Delta w_{ij} = -\eta \, \partial E(w_{ij})/\partial w_{ij}$ to each weight w_{ij} for each training pattern p. One set of updates of all the weights for all the training patterns is called one ***epoch*** of training.
6. Repeat step 5 until the network error function is 'small enough'.

You thus end up with a trained neural network. But steps 4 and 5 can still be difficult…

Gradient Descent Error Minimisation

We will look at how to choose the error function E next lecture. Suppose, for now, that we want to train a neural network by adjusting its weights w_{ij} to minimise the SSE:

$$E(w_{ij}) = \tfrac{1}{2}\sum_{p}\sum_{j}\left(targ_j - out_j\right)^2$$

We have seen that we can do this by making 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 gradient descent learning algorithm for single layer SSE networks:

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

Notice that the updates also involve the derivative *f′(x)* of the transfer function *f(x)*. This is clearly problematic for a 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. For a simple linear activation function $f(x) = x$ we have

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

This is often known as the ***Delta Rule*** because the weight updates are simply proportional to the output discrepancy

$$delta_l = targ_l - out_l$$

This is exactly the same as the Perceptron Learning Rule we saw earlier.

Delta Rule vs. Perceptron Learning Rule

We have seen that the Delta Rule and the Perceptron Learning Rule for training Single Layer Perceptrons have exactly the same weight update equation.

However, the two algorithms were 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 for sign function outputs, while the Delta Rule emerged from a gradient descent minimisation of the Sum Squared Error for a linear output activation function.

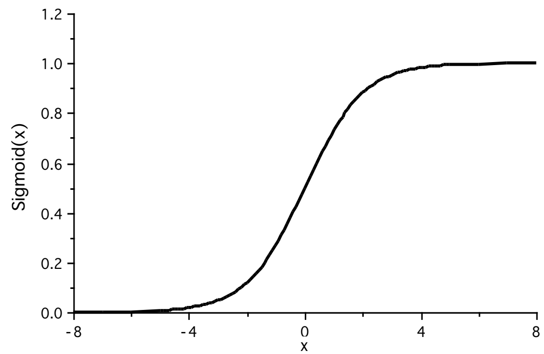
The Perceptron Learning Rule will converge to zero error and no weight changes in a finite number of steps if the problem is linearly separable, but otherwise the weights will keep oscillating. On the other hand, the Delta Rule will (for sufficiently small η) always converge to a set of weights for which the error is a minimum, though the convergence to the precise target values will generally proceed at an ever decreasing rate proportional to the output discrepancies $delta_l$.

Choosing the Activation Function

We have seen that the activation function f must be differentiable for gradient descent learning to work, and that a simple linear activation function with the SSE cost function leads to the simple delta rule, but is that the best choice of activation function?

It seems natural to proceed by looking for simple smooth (i.e. differentiable) versions of the sign (i.e. step) threshold function we used for the Simple Perceptron. The standard sigmoid (a.k.a. logistic function) is a particularly convenient smooth replacement:

$$f(x) = \text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$



One can attempt to use this with the SSE cost function, but we shall see next lecture that there is a better cost function to use with it. The most appropriate choice of activation function and cost function actually depends on what type of problem is being studied.

Overview and Reading

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

Reading

1. Bishop: Sections 3.1, 3.2, 3.3, 3.4, 3.5
2. Gurney: Sections 5.1, 5.2, 5.3
3. Beale & Jackson: Section 4.4
4. Callan: Sections 2.1, 2.2
5. Haykin-1999: Sections 2.2, 2.4, 3.3