Gradient Descent Learning

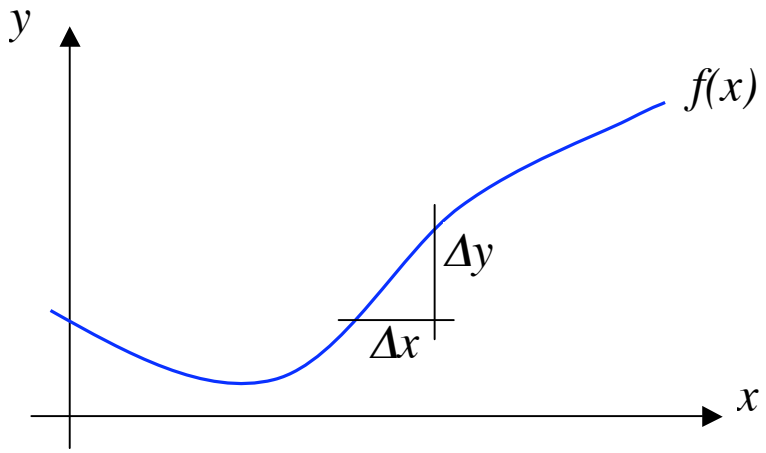
We have already seen how iterative weight updates work in Hebbian learning and the Perceptron Learning rule. The aim now is to develop a ***learning algorithm*** that minimises a cost function (such as Sum Squared Error) by making appropriate iterative adjustments to the weights w_{ij}. The idea is to apply a series of small updates to the weights $w_{ij} \rightarrow w_{ij} + \Delta w_{ij}$ until the cost $E(w_{ij})$ is "small enough".

For the Perceptron Learning Rule, we determined the direction that each weight needed to change to bring the output closer to the right side of the decision boundary, and then updated the weight by a small step in that direction.

Now we want to determine the direction that the ***weight vector*** needs to change to best reduce the chosen cost function. A systematic procedure for doing that requires knowledge of how the cost $E(w_{ij})$ varies as the weights w_{ij} change, i.e. the ***gradient*** of E with respect to w_{ij}. Then, if we repeatedly adjust the weights by small steps against the gradient, we will move through ***weight space***, descending along the gradients towards a minimum of the cost function.

Computing Gradients and Derivatives

The branch of mathematics concerned with computing gradients is called ***Differential Calculus***. The relevant general idea is straightforward. Consider a function $y = f(x)$:



The gradient of $f(x)$, at a particular value of x, is the rate of change of $f(x)$ as we change x, and that can be approximated by $\Delta y/\Delta x$ for small Δx. It can be written 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 illustrate how derivatives can be computed:

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

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

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

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

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

$$f(x) = \log_e(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \frac{1}{x}$$

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

$$f(x) = \sin(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \cos(x)$$

Gradient Descent Minimisation

If we want to change the value of x to minimise a function $f(x)$, what we need to do depends on the gradient of $f(x)$ at the current value of x. There are three cases:

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

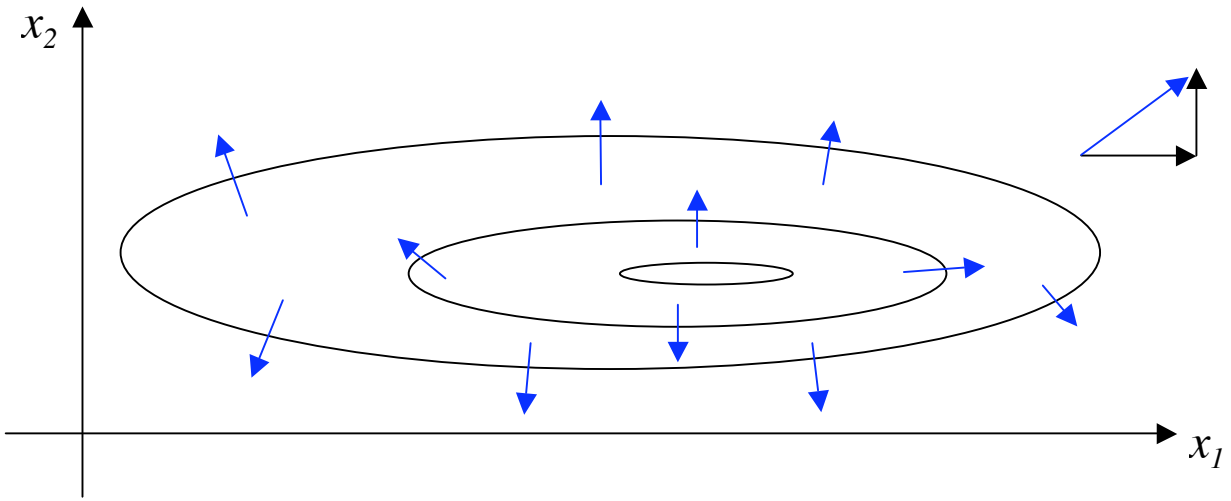
In summary, we can decrease $f(x)$ by changing x by the amount:

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

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 a minimum, and hence this procedure is known as ***gradient descent minimisation***.

Gradients in More Than One Dimension

It might not be obvious that one needs the gradient/derivative itself in the weight update equation, rather than just the sign of the gradient. So, 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 know how gradient descent weight update rules can lead to minimisation of a neural network's output errors, it is straightforward to train any single layer network:

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

   $\{in_i^p, targ_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\}$

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".

This will produce 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 single layer 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 the 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:

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

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

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

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

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

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

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

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

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

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 thus have the gradient descent learning algorithm for single layer SSE networks:

$$\Delta w_{kl} = \eta \sum_{p} (targ_l - out_l).f'(\sum_{n} in_n w_{nl}).in_k$$

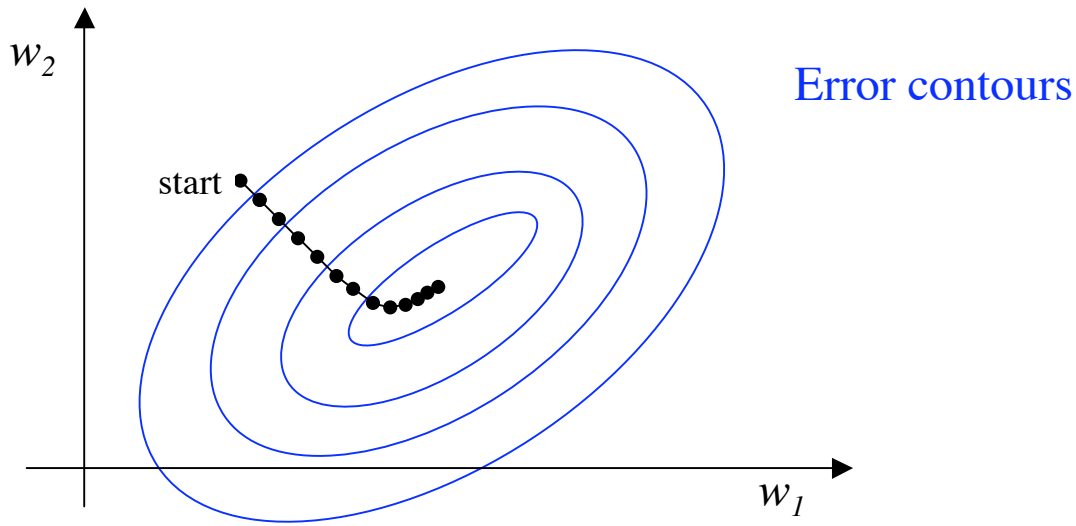
Notice that these weight updates involve the derivative $f'(x)$, so the activation function $f(x)$ must be differentiable for gradient descent learning to work. This is clearly no good for simple classification Perceptrons which use the step function step(x) as their threshold function, because that has zero derivative everywhere except at $x = 0$ where it is infinite. We will need to use a smoothed version of a step function, like a sigmoid. However, for a linear activation function $f(x) = x$, appropriate for regression, we have weight updates

$$\Delta w_{kl} = \eta \sum_{p} (targ_l - out_l).in_k$$

which is often known as the ***Delta Rule*** because each update Δw_{kl} is simply proportional to the relevant input in_k and the corresponding output discrepancy $delta_l = targ_l - out_l$. This update rule is exactly the same as the Perceptron Learning Rule we derived earlier.

Visualising Learning

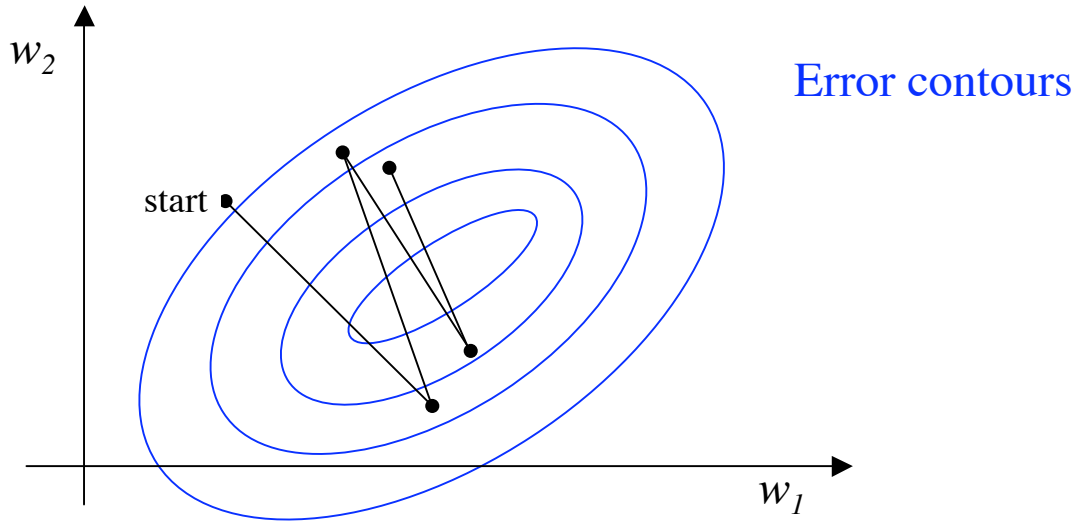
Visualising neural network learning is difficult because there are so many weights being updated at once. However, one can plot error function contours for pairs of weights to get some idea of what is happening. The weight update equations will produce a series of steps in weight space from the starting position to an error minimum:



True gradient descent produces a smooth curve perpendicular to the contours. Weight updates with a small step size η will result in a good approximation to it as shown.

Step Size Too Large

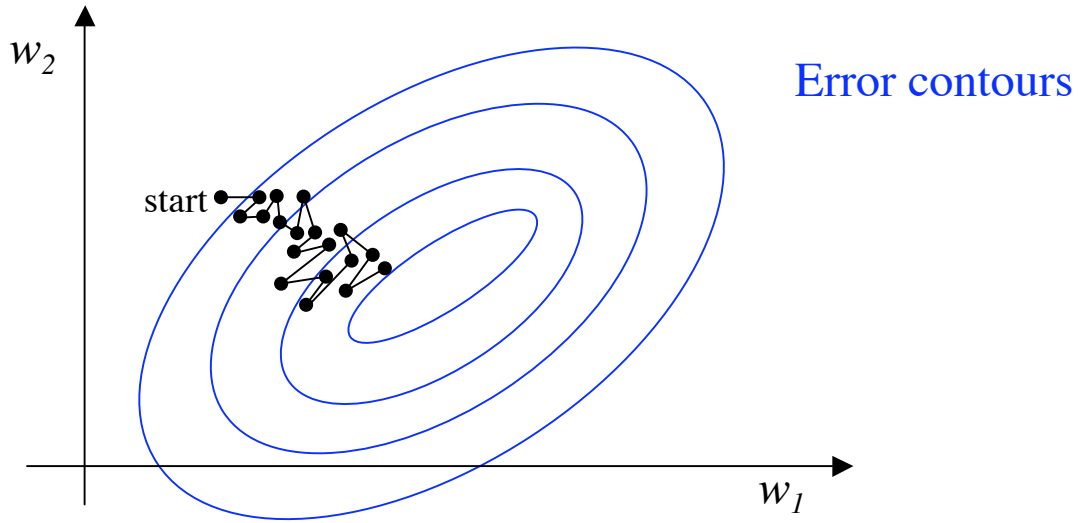
If the step size (i.e. learning rate η) is set too large, the approximation to true gradient descent will be poor, and this will result in overshoots, or even divergence:



In practice, there is no need to plot error contours to see if this is happening. It is obvious that, if the error function is fluctuating, it would be wise to reduce the step size. It is also worth checking individual weights and output activations as well. On the other hand, if everything is smooth, it is worth trying to increase η and seeing if it stays smooth.

On-line Learning

If the weights are updated after each training pattern, rather than adding up the weight changes for all the patterns before applying them, the learning algorithm is no longer true gradient descent, and the weight changes will not be perpendicular to the contours:



If the step sizes are kept small enough, the erratic behaviour of the weight updates will not be too much of a problem, and the increased number of weight changes will still get us to the minimum quicker than true gradient descent (i.e. batch learning).

Learning with Momentum

A compromise that will smooth out the erratic behaviour of on-line updates, without slowing down the learning too much, is to update the weights with the ***moving average*** of the individual weight changes corresponding to single training patterns.

If everything is labelled by the time t (which can conveniently be measured in weight update steps), then implementing a moving average is easy:

$$\Delta w_{hl}^{(n)}(t) = \eta.delta_l^{(n)}(t).out_h^{(n-1)}(t) + \alpha.\Delta w_{hl}^{(n)}(t-1)$$

One simply adds a ***momentum*** term $\alpha.\Delta w_{hl}^{(n)}(t-1)$ which is the weight change of the previous step times a momentum parameter α. If α is zero, then we have the standard on-line training algorithm used before. As α is increased towards one, each step includes increasing contributions from the previous training patterns. Obviously it makes no sense to have α less than zero, or greater than one. Good sizes of α depend on the size of the training data set and how variable it is. Usually, we will need to decrease η as we increase α, so that the total step sizes don't get too large.

Learning with Line Searches

Learning algorithms which work by taking a sequence of steps through weight space all have two basic components: the ***step size*** $size(t)$ and the ***direction*** $dir_{hl}^{(n)}(t)$ such that

$$\Delta w_{hl}^{(n)}(t) = size(t).dir_{hl}^{(n)}(t)$$

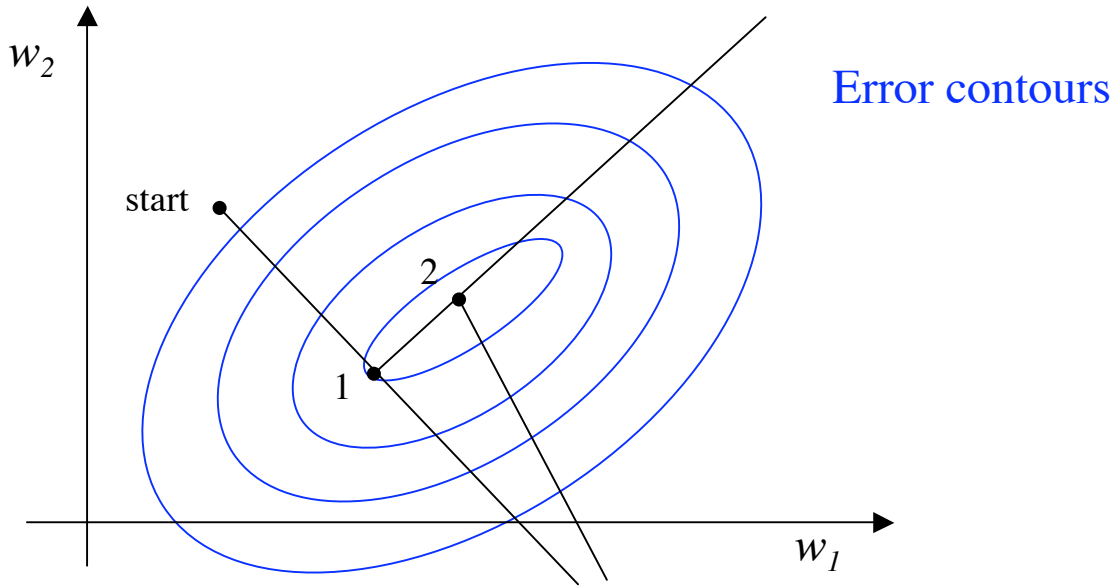
For gradient descent algorithms, such as Back-Propagation, the chosen direction is given by the partial derivatives of the error function $dir_{hl}^{(n)}(t) = -\partial E(w_{jk}^{(m)}) \big/ \partial w_{hl}^{(n)}$, and the step size is simply the small constant learning rate parameter $size(t)=\eta$.

A better procedure might be to carry on along in the chosen direction until the error starts rising again. This involves performing a ***line search*** to determine the step size.

The simplest procedure for performing a line search would be to take a series of small steps along the chosen direction until the error increases, and then go back one step. However, that is not likely to be any more efficient than standard gradient descent. Fortunately, there exist better procedures for performing line searches.

Determining the Step Size

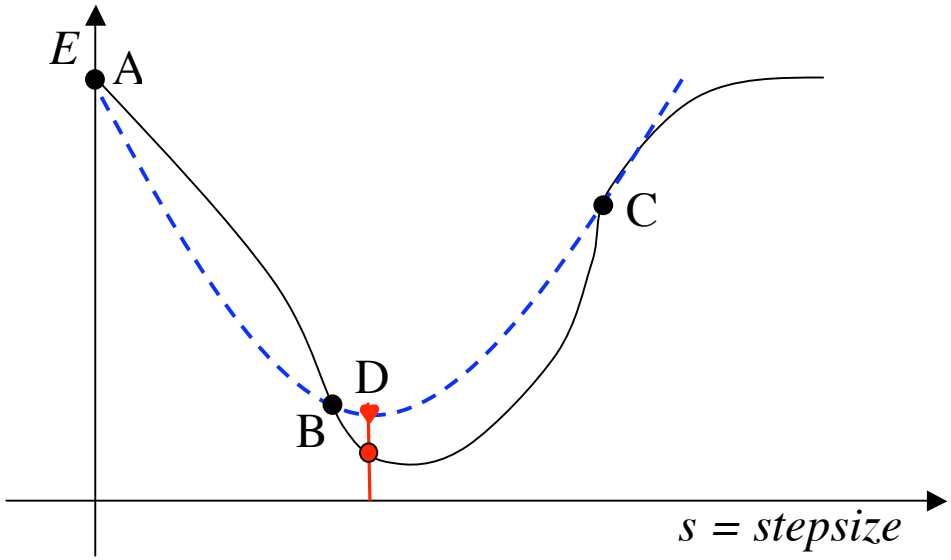
It is possible to see how to proceed with choosing an optimal step size by looking again at the error contours plot of a typical learning process:



At each stage (start, 1, 2, ...) the gradient line defines a one dimensional section through the error surface. There are well known iterative procedures for finding the minimum of such a section – a particularly good one is called ***Parabolic Interpolation***.

Parabolic Interpolation

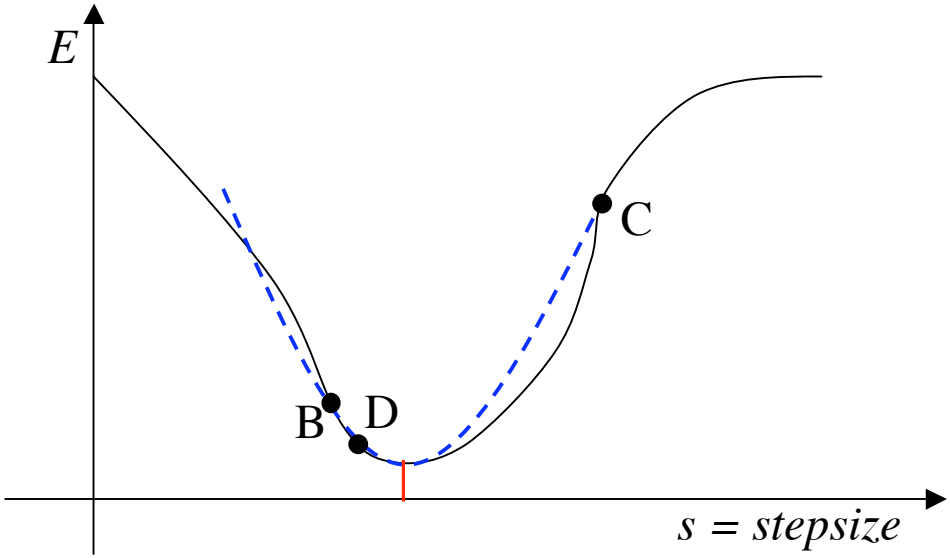
One can plot the variation of the error function $E(s)$ as the step s taken along the chosen direction is increased. Suppose we can find three points A, B, C such that $E(\mathrm{A}) > E(\mathrm{B})$ and $E(\mathrm{B}) < E(\mathrm{C})$. If A is where we are, and C is far away, this should not be difficult.



The three points are sufficient to define the parabola that passes through them, and we can then easily compute the minimum point D of that parabola. The step size corresponding to that point D is a good guess for the appropriate step size for the actual error function.

Repeated Parabolic Interpolation

The process of parabolic interpolation can easily be repeated. We take the guess of the appropriate step size given by point D, together with the two original points of lowest error (i.e. B and C) to make up a new set of three points, and repeat the procedure:



Each iteration of this process brings us closer to the minimum. However, the gradients change with each step, so it is not computationally efficient to find each step size too accurately. Usually it is better to get it roughly right and move on to the next direction.

Properties of the Gradient Descent Directions

We have seen how we can determine appropriate step sizes by doing line searches, but it is not obvious that using the gradient descent direction is really the best thing to do.

Since the step size $s(t-1)$ is chosen to minimise the new error $E(w_{ij}(t))$, the gradient of that error with respect to that step size must be zero at the new weight $w_{ij}(t)$, i.e.

$$\frac{\partial E(w_{ij}(t))}{\partial s(t-1)} = 0$$

Also, using the standard chain rule for derivatives, we know:

$$\frac{\partial E(w_{ij}(t))}{\partial s(t-1)} = \sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} \cdot \frac{\partial w_{ij}(t)}{\partial s(t-1)}$$

so the gradient descent directions $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ are seen to have the property:

$$\sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} \cdot \frac{\partial w_{ij}(t)}{\partial s(t-1)} = 0$$

Problems using Gradient Descent with Line Search

To see what that implies, we need to remember that the new weights $w_{ij}(t)$ are

$$w_{ij}(t) = w_{ij}(t-1) - s(t-1)\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}$$

and hence the partial derivatives of those with respect to the step size $s(t-1)$ are

$$\frac{\partial w_{ij}(t)}{\partial s(t-1)} = -\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}$$

Substituting that into the above gradient descent direction property reveals that:

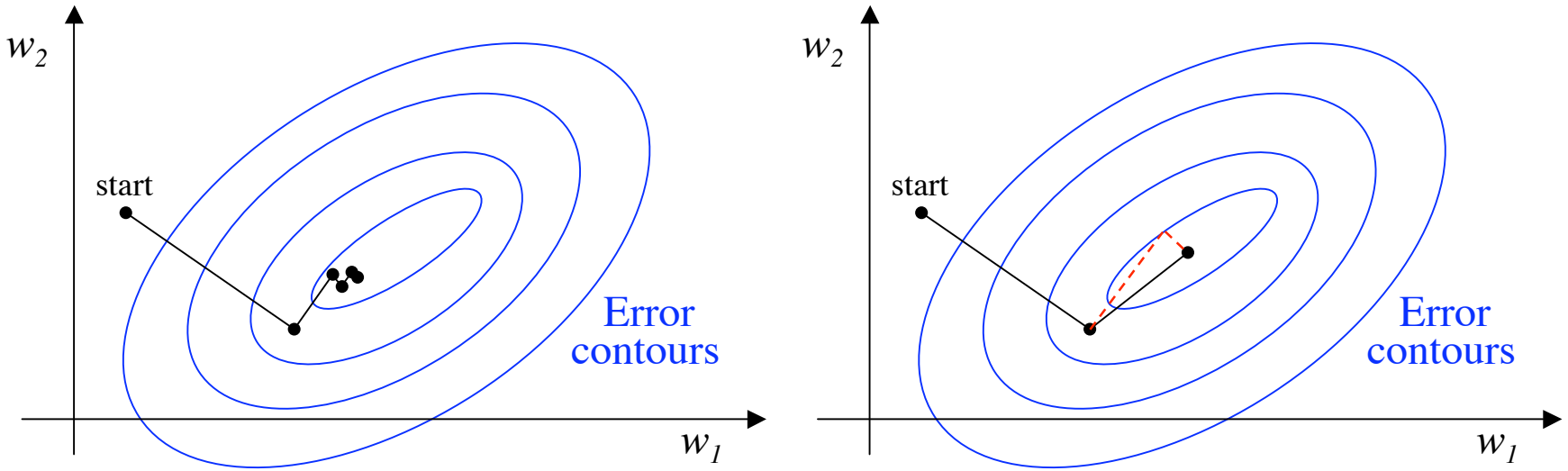
$$\sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} \cdot \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} = 0$$

The scalar product of the old direction $-\partial E(w_{ij}(t-1))/\partial w_{ij}(t-1)$ and the new direction $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ is zero, which means the directions are orthogonal (perpendicular). This will result in a less than optimal zig-zag path through weight space.

Finding a Better Search Direction

An obvious approach to avoid the zig-zagging that occurs when we use line searches and gradient directions is to make the new step direction $dir_{ij}(t)$ a compromise between the new gradient direction $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ and the previous step direction $dir_{ij}(t-1)$:

$$dir_{ij}(t) = -\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} + \beta . dir_{ij}(t-1)$$



Conjugate Gradient Learning

The basis of ***Conjugate Gradient Learning*** is to find a value for β in the last equation so that each new search direction spoils as little as possible the minimisation achieved by the previous one. We thus want to find the new direction $dir_{ij}(t)$ such that the gradient $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ at the new point $w_{ij}(t) + s.dir_{ij}(t)$ in the old direction is zero, i.e.

$$\sum_{i,j} dir_{ij}(t-1).\frac{\partial E(w_{ij}(t) + s.dir_{ij}(t))}{\partial w_{ij}(t)} = 0$$

The appropriate value of β that satisfies this is given by the ***Polak-Ribiere rule***

$$\beta = \frac{\sum_{i,j}\left(\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} - \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}\right).\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}}{\sum_{i,j}\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}.\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}}$$

For most practical applications this is the fastest way to train a neural network.

Approximating the Error Surface

The idea of a quadratic (parabolic) approximation along a line search can be extended to the whole error surface. We can perform a ***Taylor Expansion*** of the cost function around the current point $w_i(t)$ in weight space:

$$E(w_i(t)+\Delta w_i(t)) = E(w_i(t)) + \sum_j \left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} \Delta w_j(t)$$

$$+\frac{1}{2}\sum_k\sum_j \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)} \Delta w_j(t)\Delta w_k(t) + 0(\Delta w(t)^3)$$

The first derivative is the ***Gradient Vector*** G_j that we have been using before, and the second derivative is known as the local ***Hessian Matrix*** H_{jk}

$$G_j = \left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} \quad , \quad H_{jk} = \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)}$$

This approximation can be used as a basis of powerful learning algorithms.

Newton Method for Weight Updating

From the Taylor expansion, the local approximation for the error gradient is

$$\frac{E(w_i(t)+\Delta w_i(t)) - E(w_i(t))}{\Delta w_j(t)} = \left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} + \sum_k \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)} \Delta w_k(t) + 0(\Delta w(t)^2)$$

The first term is what we use for the standard gradient descent learning algorithm. If we use the Hessian term too, we can expect a better learning algorithm. If we ignore the higher order terms in Δw and note that the error gradient at the minimum is zero, then

$$\left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} + \sum_k \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)} \Delta w_k(t) = 0$$

If the Hessian is invertible, we can solve this to give the ***Newton Method*** weight update

$$\Delta w_k(t) = -\sum_j G_j H_{jk}^{-1}$$

Quasi-Newton Method for Weight Updating

Unfortunately, the Newton Method is usually computationally prohibitive, requiring $0(NW^2)$ operations to compute the Hessian and $0(W^3)$ operations to invert it (where W is the number of weights and N is the number of training patterns).

Quasi-Newton Methods use second order (curvature) information about the error surface without requiring computation of the Hessian itself. They build up an approximation to the inverse Hessian H^{-1} or weight update vector GH^{-1} over a number of steps, using successive iterations of the weights $w(t)$ and gradients $G(t)$.

There are numerous approaches in the literature for doing this. Which approach is best will depend on how well the error surface approximates a quadratic, and how many weights the network has.

Like the Conjugate Gradient method, these methods can find the minimum of a quadratic error surface in at most W steps with an overall computational cost of $0(NW^2)$.

8

Taylor Series Expansion

$$F(x) = F(x^*) + \frac{d}{dx}F(x)\Big|_{x=x^*}(x-x^*)$$

$$+ \frac{1}{2}\frac{d^2}{dx^2}F(x)\Big|_{x=x^*}(x-x^*)^2 + \cdots$$

$$+ \frac{1}{n!}\frac{d^n}{dx^n}F(x)\Big|_{x=x^*}(x-x^*)^n + \cdots$$

Example

$$F(x) = e^{-x}$$

Taylor series of $F(x)$ about $x^* = 0$:

$$F(x) = e^{-x} = e^{-0} - e^{-0}(x-0) + \frac{1}{2}e^{-0}(x-0)^2 - \frac{1}{6}e^{-0}(x-0)^3 + \ldots$$

$$F(x) = 1 - x + \frac{1}{2}x^2 - \frac{1}{6}x^3 + \ldots$$

Taylor series approximations:

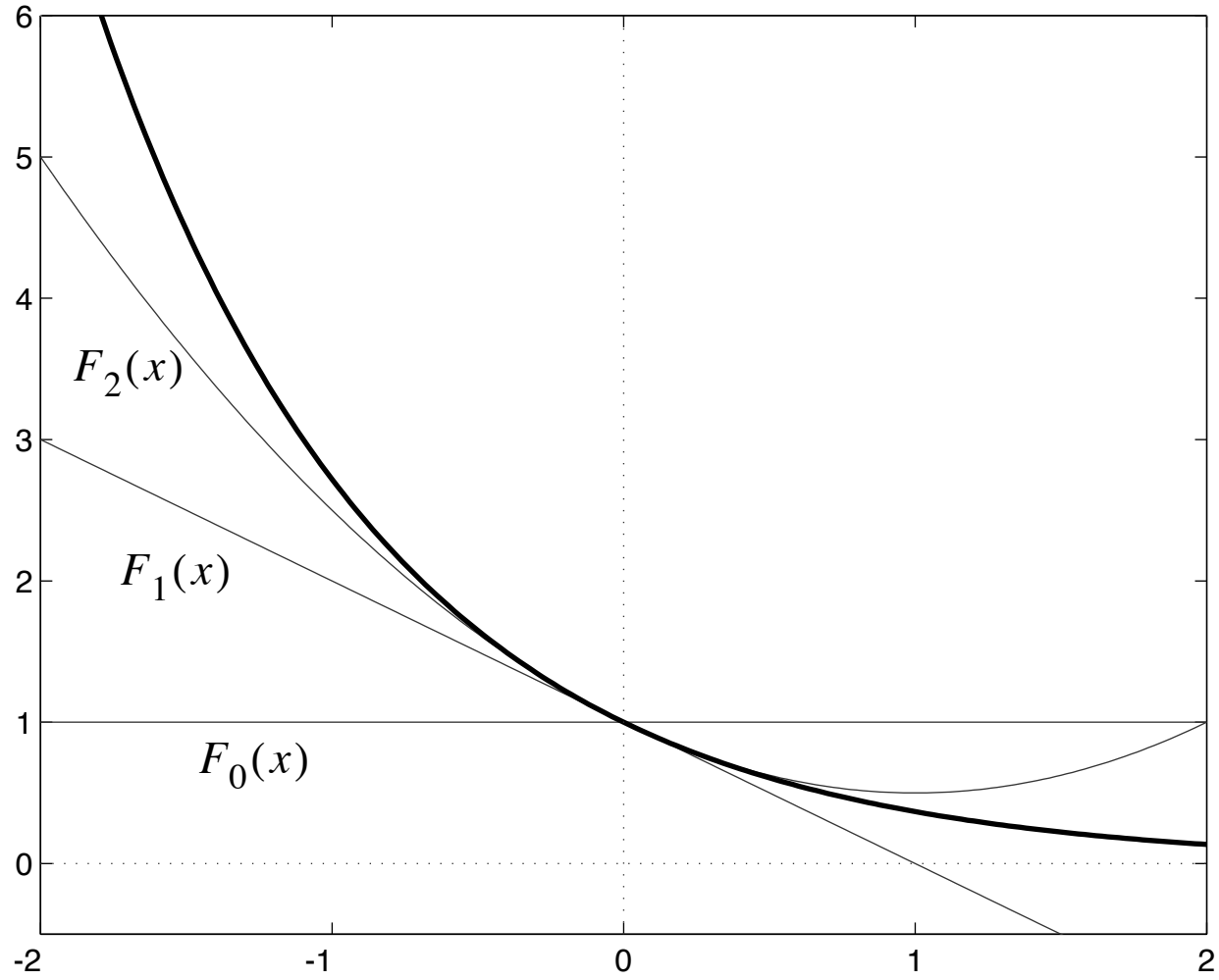
$$F(x) \approx F_0(x) = 1$$

$$F(x) \approx F_1(x) = 1 - x$$

$$F(x) \approx F_2(x) = 1 - x + \frac{1}{2}x^2$$

8

Plot of Approximations



Vector Case

$$F(\mathbf{x}) = F(x_1, x_2, \ldots, x_n)$$

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \frac{\partial}{\partial x_1} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*) + \frac{\partial}{\partial x_2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_2 - x_2^*)$$

$$+ \cdots + \frac{\partial}{\partial x_n} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_n - x_n^*) + \frac{1}{2}\frac{\partial^2}{\partial x_1^2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*)^2$$

$$+ \frac{1}{2}\frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*)(x_2 - x_2^*) + \cdots$$

Matrix Form

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T \Big|_{\mathbf{x}=\mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*)$$

$$+ \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x}=\mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \cdots$$

Gradient

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

Hessian

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} F(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_n^2} F(\mathbf{x}) \end{bmatrix}$$

8 Directional Derivatives

First derivative (slope) of $F(\mathbf{x})$ along x_i axis: $\partial F(\mathbf{x})/\partial x_i$

(ith element of gradient)

Second derivative (curvature) of $F(\mathbf{x})$ along x_i axis: $\partial^2 F(\mathbf{x})/\partial x_i^2$

(i,i element of Hessian)

First derivative (slope) of $F(\mathbf{x})$ along vector $\mathbf{p}$: $$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|}$$

Second derivative (curvature) of $F(\mathbf{x})$ along vector $\mathbf{p}$: $$\frac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2}$$

Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2$$

$$\mathbf{x}^* = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \qquad \mathbf{p} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix}\Bigg|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} 2x_1 + 2x_2 \\ 2x_1 + 4x_2 \end{bmatrix}\Bigg|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|} = \frac{\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}}{\left\| \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\|} = \frac{\begin{bmatrix} 0 \end{bmatrix}}{\sqrt{2}} = 0$$

Second-Order Condition

If the first-order condition is satisfied (zero gradient), then

$$F(\mathbf{x}^* + \Delta\mathbf{x}) = F(\mathbf{x}^*) + \frac{1}{2}\Delta\mathbf{x}^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} \Delta\mathbf{x} + \cdots$$

A strong minimum will exist at $\mathbf{x}^*$ if $\Delta\mathbf{x}^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} \Delta\mathbf{x} > 0$ for any $\Delta\mathbf{x} \neq \mathbf{0}$.

Therefore the Hessian matrix must be positive definite. A matrix $\mathbf{A}$ is positive definite if:

$$\mathbf{z}^T \mathbf{A}\mathbf{z} > 0 \quad \text{for any } \mathbf{z} \neq 0.$$

This is a **sufficient** condition for optimality.

A **necessary** condition is that the Hessian matrix be positive semidefinite. A matrix A is positive semidefinite if:

$$\mathbf{z}^T \mathbf{A}\mathbf{z} \geq 0 \quad \text{for any } \mathbf{z}.$$

Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} = \mathbf{0} \quad \Rightarrow \quad \mathbf{x}^* = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$$

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}$$

(Not a function of **x** in this case.)

To test the definiteness, check the eigenvalues of the Hessian. If the eigenvalues are all greater than zero, the Hessian is positive definite.

$$\left|\nabla^2 F(\mathbf{x}) - \lambda \mathbf{I}\right| = \left|\begin{bmatrix} 2-\lambda & 2 \\ 2 & 4-\lambda \end{bmatrix}\right| = \lambda^2 - 6\lambda + 4 = (\lambda - 0.76)(\lambda - 5.24)$$

$\lambda = 0.76, 5.24$ Both eigenvalues are positive, therefore strong minimum.

Quadratic Functions

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c \qquad \text{(Symmetric } \mathbf{A}\text{)}$$

Gradient and Hessian:

Useful properties of gradients:

$$\nabla(\mathbf{h}^T\mathbf{x}) = \nabla(\mathbf{x}^T\mathbf{h}) = \mathbf{h}$$

$$\nabla\mathbf{x}^T\mathbf{Q}\mathbf{x} = \mathbf{Q}\mathbf{x} + \mathbf{Q}^T\mathbf{x} = 2\mathbf{Q}\mathbf{x} \quad \text{(for symmetric } \mathbf{Q}\text{)}$$

Gradient of Quadratic Function:

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}$$

Hessian of Quadratic Function:

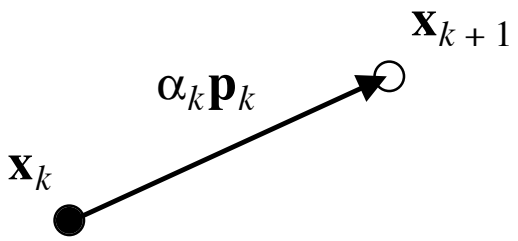
$$\nabla^2 F(\mathbf{x}) = \mathbf{A}$$

9 Basic Optimization Algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

or

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$$



$\mathbf{p}_k$ - Search Direction

α_k - Learning Rate

Steepest Descent

Choose the next step so that the function decreases:

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$$

For small changes in $\mathbf{x}$ we can approximate $F(\mathbf{x})$:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k$$

where

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}$$

If we want the function to decrease:

$$\mathbf{g}_k^T \Delta\mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0$$

We can maximize the decrease by choosing:

$$\mathbf{p}_k = -\mathbf{g}_k$$

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k}$$

Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \alpha = 0.1$$

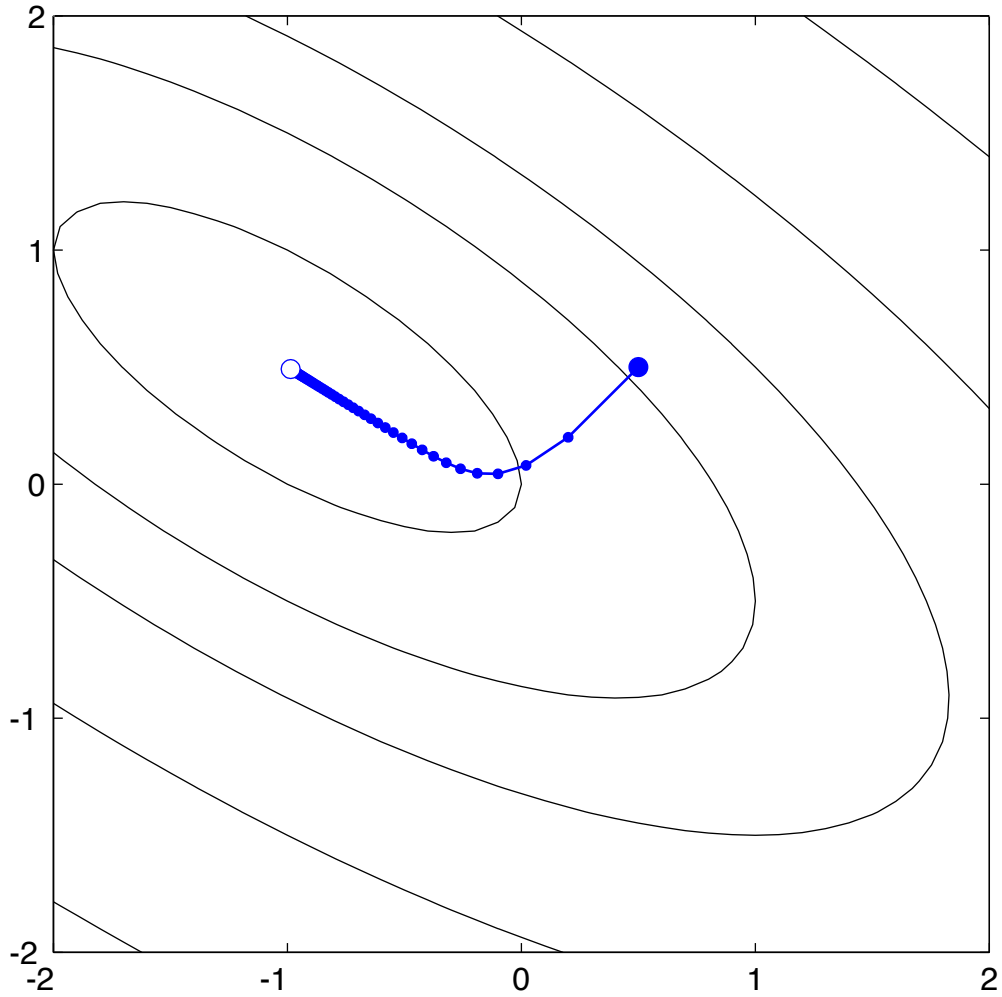
$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

9

Plot



9 Stable Learning Rates (Quadratic)

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c$$

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\mathbf{g}_k = \mathbf{x}_k - \alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}) \Longrightarrow \mathbf{x}_{k+1} = \underbrace{[\mathbf{I} - \alpha\mathbf{A}]}\mathbf{x}_k - \alpha\mathbf{d}$$

Stability is determined by the eigenvalues of this matrix.

$$[\mathbf{I} - \alpha\mathbf{A}]\mathbf{z}_i = \mathbf{z}_i - \alpha\mathbf{A}\mathbf{z}_i = \mathbf{z}_i - \alpha\lambda_i\mathbf{z}_i = \underbrace{(1 - \alpha\lambda_i)}\mathbf{z}_i$$

(λ_i - eigenvalue of $\mathbf{A}$)

Eigenvalues of [$\mathbf{I}$ - $\alpha\mathbf{A}$].

Stability Requirement:

$$\left|(1 - \alpha\lambda_i)\right| < 1 \qquad \alpha < \frac{2}{\lambda_i}$$

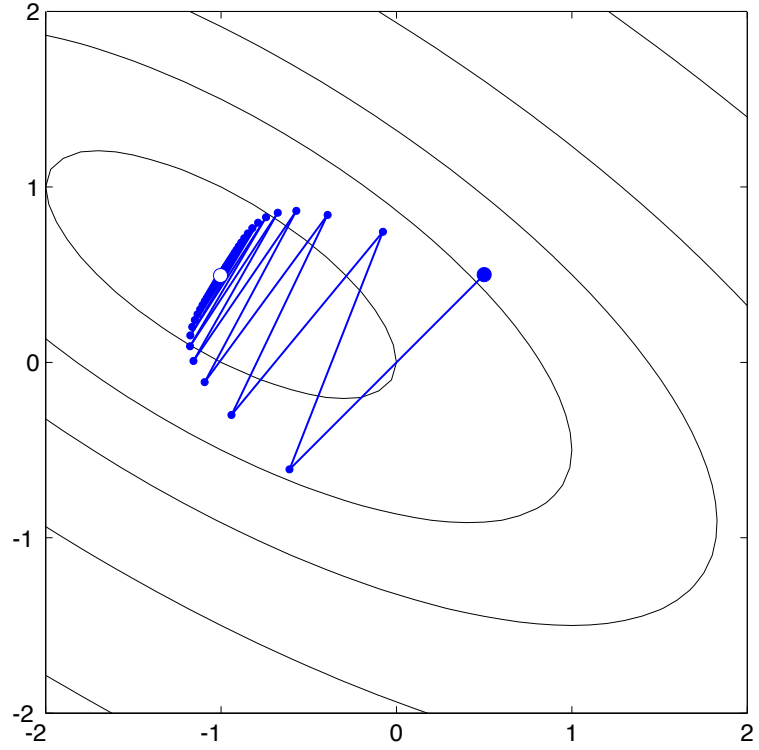
$$\alpha < \frac{2}{\lambda_{max}}$$

Example

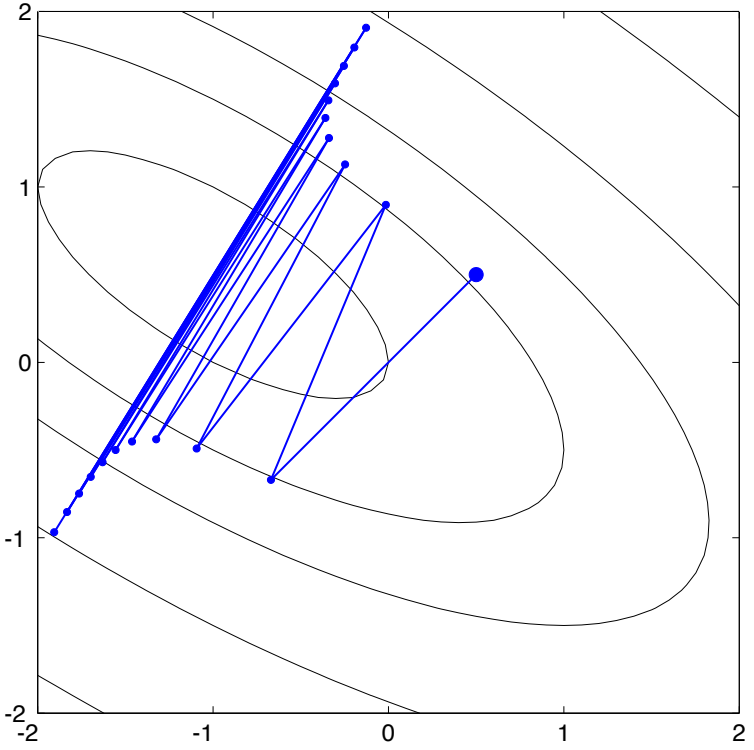
$$\mathbf{A} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \qquad \left\{(\lambda_1 = 0.764), \left(\mathbf{z}_1 = \begin{bmatrix} 0.851 \\ -0.526 \end{bmatrix}\right)\right\}, \left\{\lambda_2 = 5.24, \left(\mathbf{z}_2 = \begin{bmatrix} 0.526 \\ 0.851 \end{bmatrix}\right)\right\}$$

$$\alpha < \frac{2}{\lambda_{max}} = \frac{2}{5.24} = 0.38$$

$\alpha = 0.37$



$\alpha = 0.39$



Minimizing Along a Line

Choose α_k to minimize $F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$

$$\frac{d}{d\alpha_k}(F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)) = \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k + \alpha_k \mathbf{p}_k^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k$$

$$\alpha_k = -\frac{\nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$

where

$$\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}$$

Example

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}2 & 2\\ 2 & 4\end{bmatrix}\mathbf{x} + \begin{bmatrix}1 & 0\end{bmatrix}\mathbf{x} \qquad \mathbf{x}_0 = \begin{bmatrix}0.5\\ 0.5\end{bmatrix}$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix}\frac{\partial}{\partial x_1}F(\mathbf{x})\\ \frac{\partial}{\partial x_2}F(\mathbf{x})\end{bmatrix} = \begin{bmatrix}2x_1 + 2x_2 + 1\\ 2x_1 + 4x_2\end{bmatrix} \qquad \mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix}-3\\ -3\end{bmatrix}$$

$$\alpha_0 = -\frac{\begin{bmatrix}3 & 3\end{bmatrix}\begin{bmatrix}-3\\ -3\end{bmatrix}}{\begin{bmatrix}-3 & -3\end{bmatrix}\begin{bmatrix}2 & 2\\ 2 & 4\end{bmatrix}\begin{bmatrix}-3\\ -3\end{bmatrix}} = 0.2 \qquad \mathbf{x}_1 = \mathbf{x}_0 - \alpha_0\mathbf{g}_0 = \begin{bmatrix}0.5\\ 0.5\end{bmatrix} - 0.2\begin{bmatrix}3\\ 3\end{bmatrix} = \begin{bmatrix}-0.1\\ -0.1\end{bmatrix}$$