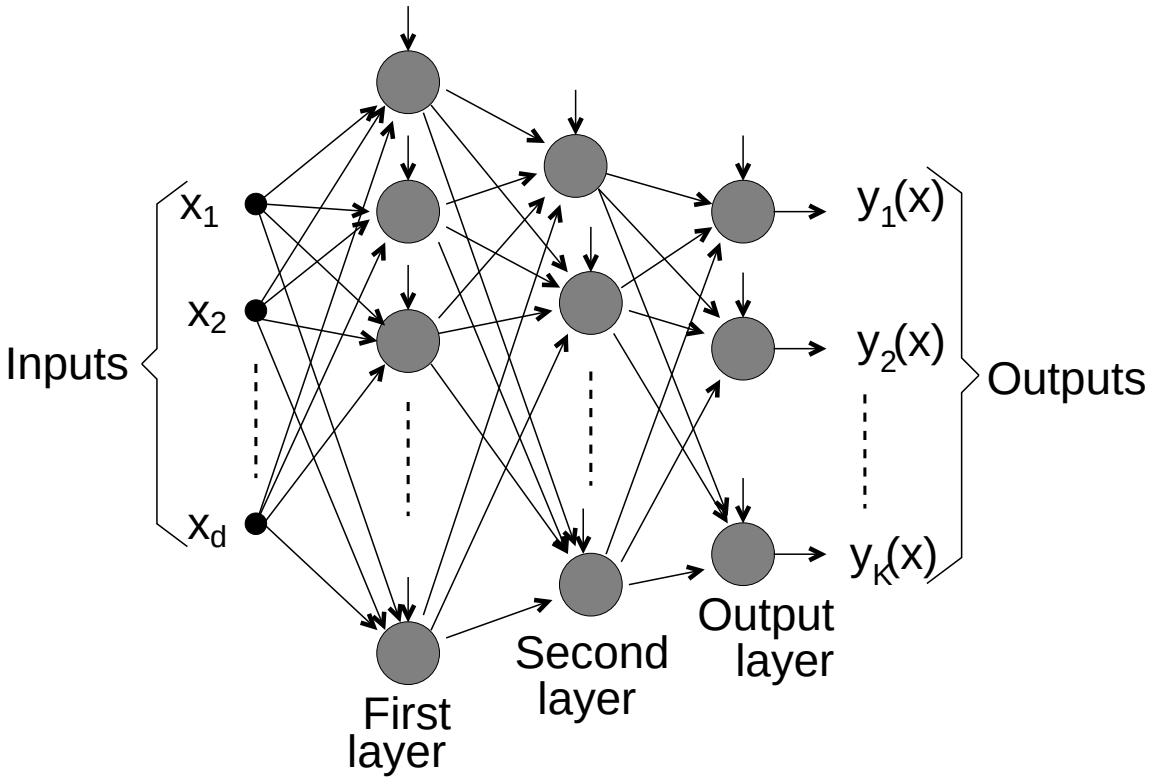
University of Cambridge
Engineering Part IIB

Module 4F10: Statistical Pattern Processing

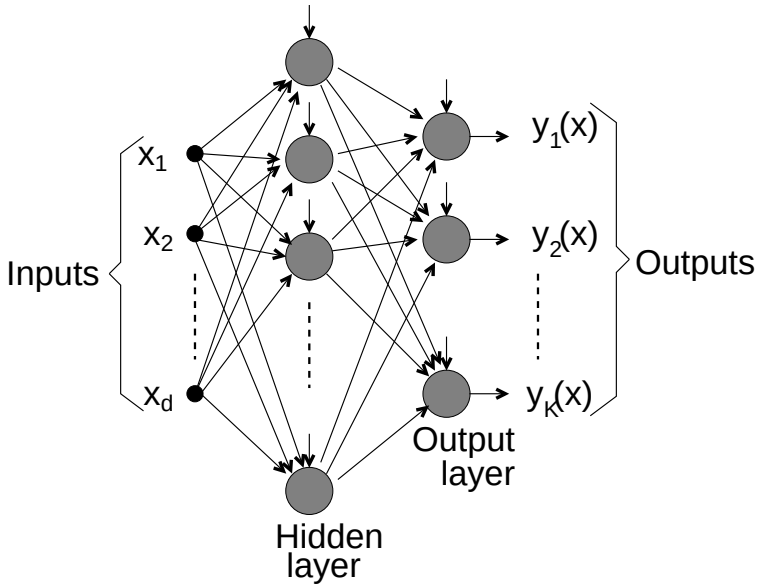
Handout 7: Multi-Layer Perceptrons II



Phil Woodland
pcw@eng.cam.ac.uk
Lent 2007

Error Back Propagation

Interest in multi-layer perceptrons (MLPs) resurfaced with the development of the *error back propagation* algorithm. This allows multi-layer perceptons to be simply trained.



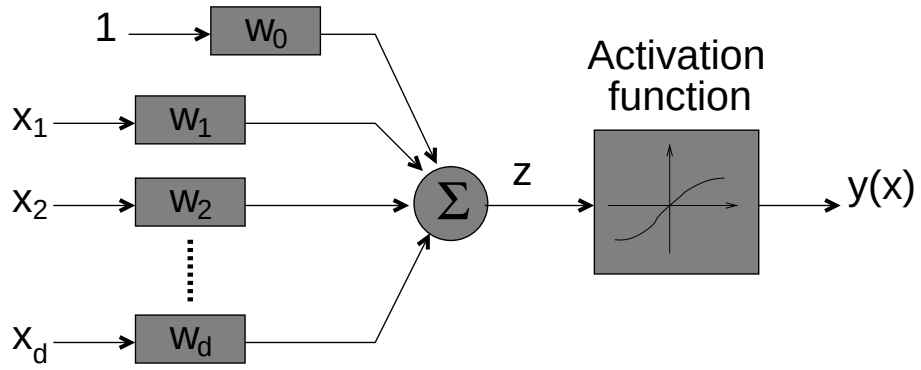
A single hidden layer network is shown above. As previously mentioned with sigmoidal activation functions arbitrary decision boundaries may be obtained with this network topology.

The error back propagation algorithm is based on *gradient descent*. Hence the activation function must be differentiable. Thus *threshold* and *step* units will not be considered. We need to be able to compute the derivative of the error function with respect to the weights of *all* layers.

All gradients in the next few slides are evaluated at the *current* model parameters.

Single Layer Perceptron

Rather than examine the multi-layer case instantly, consider the following single layer perceptron.



We would like to minimise (for example) the square error between the target of the output, $t(\mathbf{x}_p)$, and the current output value $y(\mathbf{x}_p)$. Assume that the activation function is known to be a sigmoid function. The cost function may be written as

$$E = \frac{1}{2}\sum_{p=1}^{n}(y(\mathbf{x}_p) - t(\mathbf{x}_p)'(y(\mathbf{x}_p) - t(\mathbf{x}_p)) = \sum_{p=1}^{n} E^{(p)}$$

To simplify notation, we will only consider a single observation $\mathbf{x}$ with associated target values $t(\mathbf{x})$ and current output from the network $y(\mathbf{x})$. The error with this single observation is denoted E.

The first question is how does the error change as we alter $y(\mathbf{x})$.

$$\frac{\partial E}{\partial y(\mathbf{x})} = y(\mathbf{x}) - t(\mathbf{x})$$

But we are not interested in $y(\mathbf{x})$ - how do we find the effect of varying the weights?

SLP Training (cont)

Calculate effect of changing z on the error using the chain rule

$$\frac{\partial E}{\partial z} = \left(\frac{\partial E}{\partial y(\mathbf{x})}\right)\left(\frac{\partial y(\mathbf{x})}{\partial z}\right)$$

However what we really want is the change of the error with respect to the weights (the parameters that we want to learn).

$$\frac{\partial E}{\partial w_i} = \left(\frac{\partial E}{\partial z}\right)\left(\frac{\partial z}{\partial w_i}\right)$$

The error function therefore depends on the weight as

$$\frac{\partial E}{\partial w_i} = \left(\frac{\partial E}{\partial y(\mathbf{x})}\right)\left(\frac{\partial y(\mathbf{x})}{\partial z}\right)\left(\frac{\partial z}{\partial w_i}\right)$$

Noting that

$$\frac{\partial y(\mathbf{x})}{\partial z} = y(\mathbf{x})(1 - y(\mathbf{x}))$$

$$\frac{\partial E}{\partial w_i} = (y(\mathbf{x}) - t(\mathbf{x}))y(\mathbf{x})(1 - y(\mathbf{x}))x_i$$

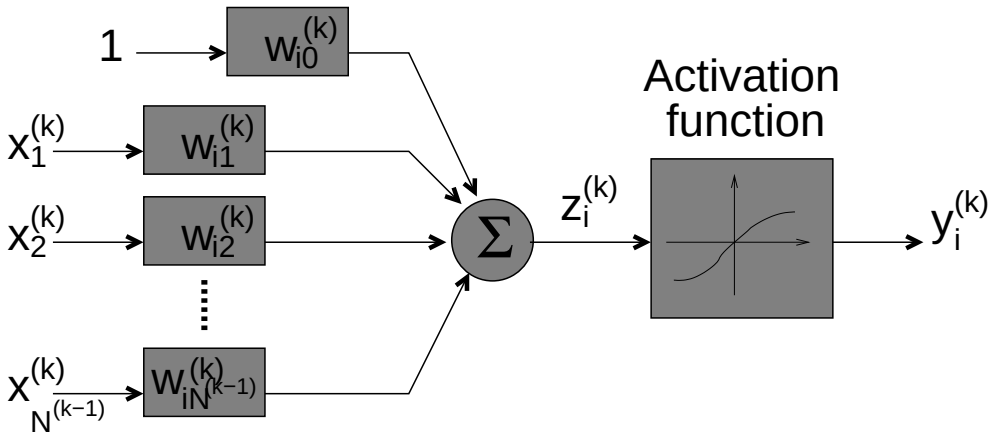
This has been computed for a single observation. We are interested in terms of the complete training set. We know that the total errors is the sum of the individual errors, so

$$\boldsymbol{\nabla} E = \sum_{p=1}^{n} (y(\mathbf{x}_p) - t(\mathbf{x}_p))y(\mathbf{x}_p)(1 - y(\mathbf{x}_p))\tilde{\mathbf{x}}_p$$

So for a single layer we can use gradient descent schemes to find the "best" weight values. We can also apply the above to compute the derivatives wrt the weights for the final hidden to output layer for an MLP.

Error Back Propagation Algorithm

Now consider a particular node, i, of hidden layer k. Using the previously defined notation, the input to the node is $\tilde{\mathbf{x}}^{(k)}$ and the output $y_i^{(k)}$.



From the previous section we can simply derive the rate of change of the error function with the weights of the output layer. We need to now examine the rate of change with the k^{th} hidden layer weights.

A general error criterion, E, will be used, although it is also assumed that the output nodes can be treated independently (not true for softmax ...) We will assume that a sigmoid activation function is used (as above). In terms of the derivations given the output layer will be considered as the $L + 1^{th}$ layer.

The training observations are assumed independent and so

$$E = \sum_{p=1}^{n} E^{(p)}$$

where $E^{(p)}$ is the error cost for the p observation and the observations are $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

Error Back Propagation Algorithm (cont)

We are required to calculate $\frac{\partial E}{\partial \tilde{w}_{ij}^{(k)}}$ for all layers, k, and all rows and columns of $\tilde{\mathbf{W}}^{(k)}$. Applying the chain rule

$$\frac{\partial E}{\partial \tilde{w}_{ij}^{(k)}} = \frac{\partial E}{\partial z_i^{(k)}} \frac{\partial z_i^{(k)}}{\partial \tilde{w}_{ij}^{(k)}} = \delta_i^{(k)} \tilde{x}_j^{(k)}$$

where

$$\frac{\partial E}{\partial z_i^{(k)}} = \delta_i^{(k)}$$

and the δ's are sometimes known as the indiviual "errors" (that are back-propagated).

For the output nodes the evaluation of δ_i is straightforward as we saw for the single layer perceptron.

To evaluate the δ_i's for hidden layers

$$\delta_i^{(k)} = \sum_m \left[\frac{\partial E}{\partial z_m^{(k+1)}} \frac{\partial z_m^{(k+1)}}{\partial z_i^{(k)}} \right]$$

where it is assumed that only the units in layer $k+1$ are connected to units in layer k, or

$$\delta_i^{(k)} = y_i^{(k)}(1 - y_i^{(k)}) \sum_m \tilde{w}_{mi}^{(k+1)} \delta_m^{(k+1)}$$

Note that all that is being done here is evaluating the differentials of the error at the output with respect to the weights throughout the network by using the chain rule for partial derivatives.

Error Back Propagation

To calculate $\nabla E^{(p)}\big|_{\boldsymbol{\theta}_{[\tau]}}$ ($\boldsymbol{\theta}[\tau]$ is the set of "current" (training epoch τ) values of the weights) we use the following algorithm.

1. Apply the input vector $\mathbf{x}_p$ to the network and use the feed forward matrix equations to propagate the input forward through the network. For all layers this yields $\mathbf{y}^{(k)}$ and $\mathbf{z}^{(k)}$.
2. Compute $\frac{\partial E}{\partial \mathbf{y}(\mathbf{x})}\big|_{\boldsymbol{\theta}_{[\tau]}}$ (the gradient at the output layer).
3. Using the back-propagation formulae back-propagate the δs back through the network, layer by layer and hence the partial derivatives for each weight.

Having obtained the derivatives of the error function with respect to the weights of the network, we need a scheme to optimise the value of the weights.

The obvious choice is **gradient descent**

Gradient Descent

Having found an expression for the gradient, gradient descent may be used to find the values of the weights.

Initially consider a **batch** update rule. Here

$$\tilde{\mathbf{w}}_i^{(k)}[\tau+1] = \tilde{\mathbf{w}}_i^{(k)}[\tau] - \eta \frac{\partial E}{\partial \tilde{\mathbf{w}}_i^{(k)}}\bigg|_{\boldsymbol{\theta}[\tau]}$$

where $\boldsymbol{\theta}[\tau] = \{\tilde{\mathbf{W}}^{(1)}[\tau], \ldots, \tilde{\mathbf{W}}^{(L+1)}[\tau]\}$, $\tilde{\mathbf{w}}_i^{(k)}[\tau]$ is the i^{th} row of $\tilde{\mathbf{W}}^{(k)}$ at training **epoch** τ and

$$\frac{\partial E}{\partial \tilde{\mathbf{w}}_i^{(k)}}\bigg|_{\boldsymbol{\theta}[\tau]} = \sum_{p=1}^{n} \frac{\partial E^{(p)}}{\partial \tilde{\mathbf{w}}_i^{(k)}}\bigg|_{\boldsymbol{\theta}[\tau]}$$

If the total number of weights in the system is N then all N derivatives may be calculated in $\mathcal{O}(N)$ operations with memory requirements $\mathcal{O}(N)$.

However in common with other gradient descent schemes there are problems as:

- we need a value of η that achieves a stable, fast descent;
- the error surface may have local *minima*, *maxima* and *saddle points*.

This has lead to refinements of gradient descent.

Training Schemes

On the previous slide the weights were updated after all n training examples have been seen. This is not the only scheme that may be used.

- **Batch** update: the weights are updated after all the training examples have been seen. Thus

$$\tilde{\mathbf{w}}_i^{(k)}[\tau+1] = \tilde{\mathbf{w}}_i^{(k)}[\tau] - \eta \left(\sum_{p=1}^{n} \left. \frac{\partial E^{(p)}}{\partial \tilde{\mathbf{w}}_i^{(k)}} \right|_{\boldsymbol{\theta}_{[\tau]}} \right)$$

- **Sequential** update: the weights are updated after every sample. Now

$$\tilde{\mathbf{w}}_i^{(k)}[\tau+1] = \tilde{\mathbf{w}}_i^{(k)}[\tau] - \eta \left. \frac{\partial E^{(p)}}{\partial \tilde{\mathbf{w}}_i^{(k)}} \right|_{\boldsymbol{\theta}_{[\tau]}}$$

  and we cycle around the training vectors.
  There are some advantages of this form of update.

  - It is not necessary to store the whole training database. Samples may be used only once if desired.
  - They may be used for *online* learning
  - In dynamic systems the values of the weights can be updated to "track" the system.

In practice forms of batch training or an intermediate between batch and sequential training are often used.

Refining Gradient Descent

There are some simple techniques to refine standard gradient descent. First consider the learning rate η. We can make this vary with each iteration. One of the simplest rules is to use

$$\eta[\tau+1] = \begin{cases} 1.1\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) < E(\boldsymbol{\theta}[\tau-1]) \\ 0.5\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) > E(\boldsymbol{\theta}[\tau-1]) \end{cases}$$

In words: if the previous value of $\eta[\tau]$ decreased the value of the cost function, then increase $\eta[\tau]$. If the previous value of $\eta[\tau]$ increased the cost function ($\eta[\tau]$ too large) then decrease $\eta[\tau]$.

It is also possible to add a momentum term to the optimisation (common in MLP estimation). The update formula is

$$\tilde{\mathbf{w}}_i^{(k)}[\tau+1] = \tilde{\mathbf{w}}_i^{(k)}[\tau] + \boldsymbol{\Delta}\tilde{\mathbf{w}}_i^{(k)}[\tau]$$

where

$$\boldsymbol{\Delta}\tilde{\mathbf{w}}_i^{(k)}[\tau] = -\eta[\tau+1] \left.\frac{\partial E}{\partial \tilde{\mathbf{w}}_i^{(k)}}\right|_{\boldsymbol{\theta}_{[\tau]}} + \alpha[\tau]\boldsymbol{\Delta}\tilde{\mathbf{w}}_i^{(k)}[\tau-1]$$

The use of the momentum term, $\alpha[\tau]$:

- smooths successive updates;
- helps avoid small local minima.

Unfortunately it introduces an additional tunable parameter to set. Also if we are lucky and hit the minimum solution we will overshoot.

Quadratic Approximation

Gradient descent makes use of first-order derivatives of the error function. What about higher order techniques?

Consider the vector form of the Taylor series

$$\begin{aligned} E(\boldsymbol{\theta}) &= E(\boldsymbol{\theta}[\tau]) + (\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])'\mathbf{g} \\ &\quad + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])'\mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau]) + \mathcal{O}(3) \end{aligned}$$

where

$$\mathbf{g} = \boldsymbol{\nabla} E(\boldsymbol{\theta})|_{\boldsymbol{\theta}[\tau]}$$

and

$$(\mathbf{H})_{ij} = h_{ij} = \left.\frac{\partial^2 E(\boldsymbol{\theta})}{\partial w_i \partial w_j}\right|_{\boldsymbol{\theta}[\tau]}$$

Ignoring higher order terms we find

$$\boldsymbol{\nabla} E(\boldsymbol{\theta}) = \mathbf{g} + \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])$$

Equating this to zero we find that the value of $\boldsymbol{\theta}$ at this point $\boldsymbol{\theta}[\tau + 1]$ is

$$\boldsymbol{\theta}[\tau + 1] = \boldsymbol{\theta}[\tau] - \mathbf{H}^{-1}\mathbf{g}$$

This gives us a simple update rule. The direction $\mathbf{H}^{-1}\mathbf{g}$ is known as the *Newton direction*.

Problems with the Hessian

In practice the use of the Hessian is limited.

1. The evaluation of the Hessian may be computationally expensive as $\mathcal{O}(N^2)$ parameters must be accumulated for each of the n training samples.
2. The Hessian must be inverted to find the direction, $\mathcal{O}(N^3)$. This gets very expensive as N gets large.
3. The direction given need not head towards a minimum - it could head towards a *maximum* or *saddle point*. This occurs if the Hessian is not *positive-definite* i.e.

   $$\mathbf{v}'\mathbf{H}\mathbf{v} > 0$$

   for all $\mathbf{v}$.
4. If the surface is highly non-quadratic the step sizes may be too large and the optimisation becomes unstable.

Approximations to the Hessian are commonly used.

The simplest approximation is to assume that the Hessian is diagonal. This ensures that the Hessian is invertible and only requires N parameters.

The Hessian may be made positive definite using

$$\tilde{\mathbf{H}} = \mathbf{H} + \lambda\mathbf{I}$$

If λ is large enough then $\tilde{\mathbf{H}}$ is positive definite.

Improved Learning Rates

Rather than having a single learning rate for *all* weights in the system, weight specific rates may be used without using the Hessian. All schemes will make use of

$$g_{ij}^{(k)}[\tau] = \left.\frac{\partial E}{\partial \tilde{w}_{ij}^{(k)}}\right|_{\boldsymbol{\theta}_{[\tau]}}$$

- **Delta-delta**: we might want to increase the learning rate when two consecutive *gradients* have the same sign. This may be implemented as

  $$\Delta\eta_{ij}^{(k)}[\tau] = \gamma g_{ij}^{(k)}[\tau] g_{ij}^{(k)}[\tau-1]$$

  where $\gamma > 0$. Unfortunately this can take the learning rate negative (depending on the value of γ)!

- **Delta-bar-delta**: refines delta-delta so that

  $$\Delta\eta_{ij}^{(k)}[\tau] = \begin{cases} \kappa, & \text{if } \overline{g}_{ij}^{(k)}[\tau-1] g_{ij}^{(k)}[\tau] > 0 \\ -\gamma\eta_{ij}^{(k)}[\tau-1], & \text{if } \overline{g}_{ij}^{(k)}[\tau-1] g_{ij}^{(k)}[\tau] < 0 \end{cases}$$

  where

  $$\overline{g}_{ij}^{(k)}[\tau] = (1-\beta) g_{ij}^{(k)}[\tau] + \beta \overline{g}_{ij}^{(k)}[\tau-1]$$

  One of the drawbacks with this scheme is that three parameters, γ,κ and β must be selected.

- **Quickprop**. Here

  $$\Delta\tilde{w}_{ij}^{(k)}[\tau+1] = \frac{g_{ij}^{(k)}[\tau]}{g_{ij}^{(k)}[\tau-1] - g_{ij}^{(k)}[\tau]} \Delta\tilde{w}_{ij}^{(k)}[\tau]$$

Line Search

Rather than determining the direction and the step size together as suggested above, another approach is to determine these separately.

Hence, if the *direction* of an update is given then we can explicitly find how *far* to move in that direction by a process of line search.

Assume found a direction for the update $\mathbf{d}[\tau]$. Now need to find the parameter λ such that we minimise

$$E(\boldsymbol{\theta}[\tau + 1]) = E\left(\boldsymbol{\theta}[\tau] + \lambda \mathbf{d}[\tau]\right)$$

The value of λ can be found approximately and efficiently by a line-search procedure that assuming the error surface is smooth and can be approximated by a quadratic function takes several measurements of the error function E (each involving a forward pass of the data) to bound the minimum and refine the minimum estimate.

Conjugate Directions

Assume that we have optimised in one direction, $\mathbf{d}[\tau]$ i.e. we have taken a step in this direction of the optimal size to minimise E.

What direction should we now optimise in?

We know that

$$\frac{\partial}{\partial \lambda} E(\boldsymbol{\theta}[\tau] + \lambda \mathbf{d}[\tau]) = 0$$

If we work out the gradient at this new point $\boldsymbol{\theta}[\tau+1]$ we know that

$$\boldsymbol{\nabla} E(\boldsymbol{\theta}[\tau+1])' \mathbf{d}[\tau] = 0$$

Is this the best direction? **No**.

What we really want is that as we move off in the new direction , $\mathbf{d}[\tau+1]$, we would like to maintain the gradient in the previous direction, $\mathbf{d}[\tau]$, being zero. In other words

$$\boldsymbol{\nabla} E(\boldsymbol{\theta}[\tau+1] + \lambda \mathbf{d}[\tau+1])' \mathbf{d}[\tau] = 0$$

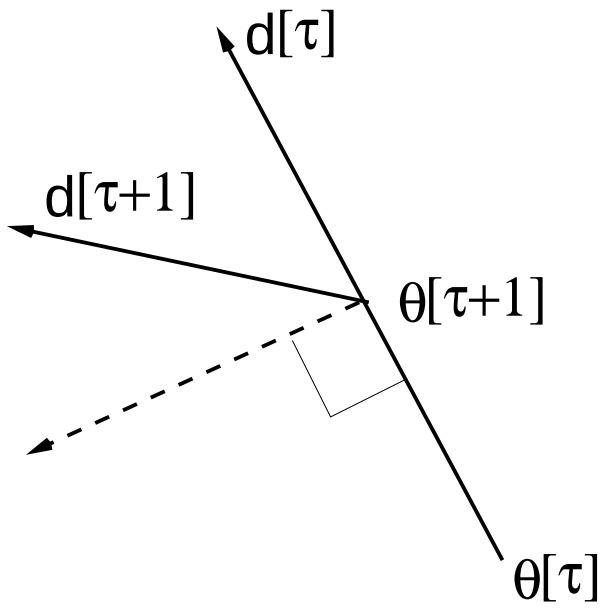
Using a first order Taylor series expansion

$$\boldsymbol{\nabla} \left(E(\boldsymbol{\theta}[\tau+1]) + \lambda \mathbf{d}[\tau+1]' \boldsymbol{\nabla} E(\boldsymbol{\theta}[\tau+1]) \right)' \mathbf{d}[\tau] = 0$$

Hence the following constraint is satisfied for a conjugate gradient

$$\mathbf{d}[\tau+1]' \mathbf{H} \mathbf{d}[\tau] = 0$$

Conjugate Gradient (cont)



Fortunately the conjugate direction can be calculated without explicitly computing the Hessian. This leads to the *conjugate gradient descent algorithm*: see Bishop (1995) pp. 274-276 for details in which a sequence of steps are taken which are in conjugate directions to all previous steps.

Input Transformations

If the input to the network are not normalised the training time may become very large. It is advantageous to *normalise* the input data by applying a transformation:

$$\overline{x}_{pi} = \frac{x_{pi} - \mu_i}{\sigma_i}$$

where

$$\mu_i = \frac{1}{n} \sum_{p=1}^{n} x_{pi}$$

and

$$\sigma_i^2 = \frac{1}{n} \sum_{p=1}^{n} (x_{pi} - \mu_i)^2$$

The transformed data has zero mean and variance 1.

This transformation may be generalised to *whitening*. Here the covariance matrix of the original data is calculated. The data is then *decorrelated* and the mean subtracted. This results in data with zero mean and an identity matrix covariance matrix.

Regularisation

One of the major issues with training neural networks is how to ensure *generalisation*. One commonly used technique is weight decay. A *regulariser* is used. Here

$$\Omega = \frac{1}{2}\sum_{i=1}^{N} w_i^2$$

where N is the total number of weights in the network. A new error function is defined

$$\tilde{E} = E + \nu\Omega$$

Using gradient descent on this gives

$$\boldsymbol{\nabla}\tilde{E} = \boldsymbol{\nabla}E + \nu\mathbf{w}$$

The effect of this regularisation term Ω penalises very large weight terms. From empirical results this has resulted in improved performance.

Rather than using an explicit regularisation term, the "complexity" of the network can be controlled by *training with noise*.

For batch training we replicate each of the samples multiple times and add a different noise vector to each of the samples. If we use least squares training with a zero mean noise source (equal variance ν in all the dimensions) the error function may be shown to have the form

$$\tilde{E} = E + \nu\Omega$$

This is a another form of regularisation.