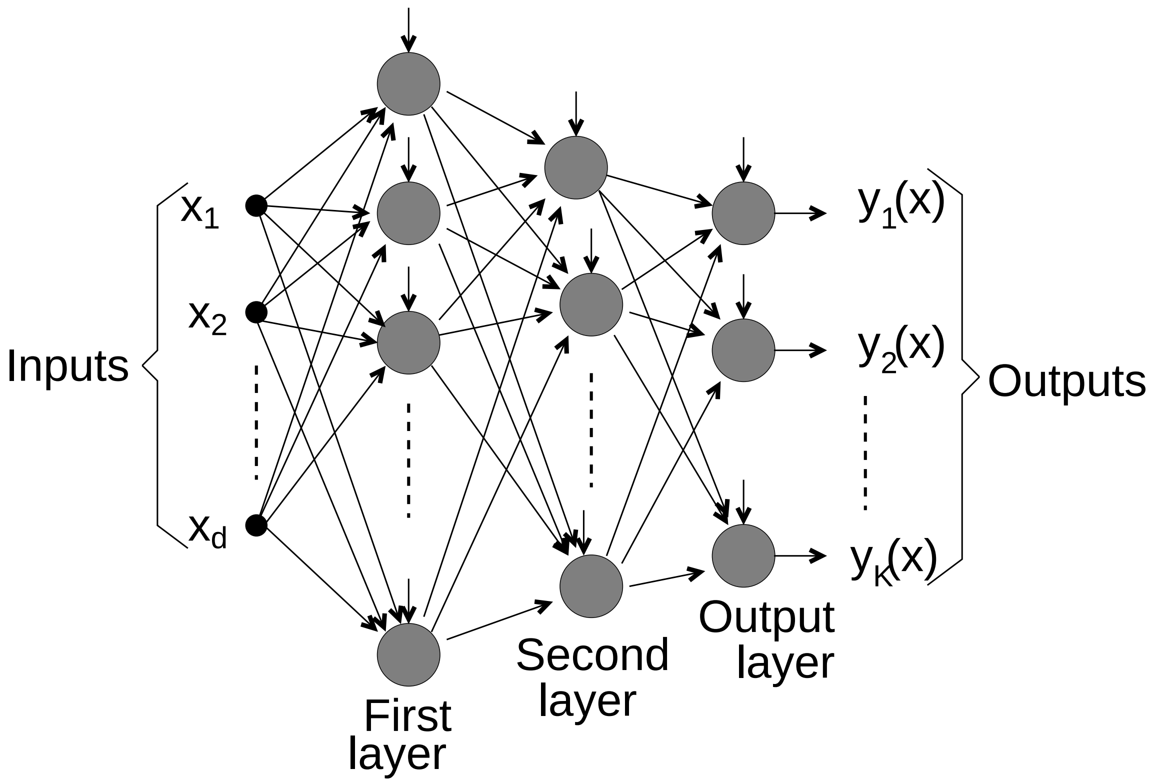
University of Cambridge
Engineering Part IIB & EIST Part II

Paper I10: Advanced Pattern Processing

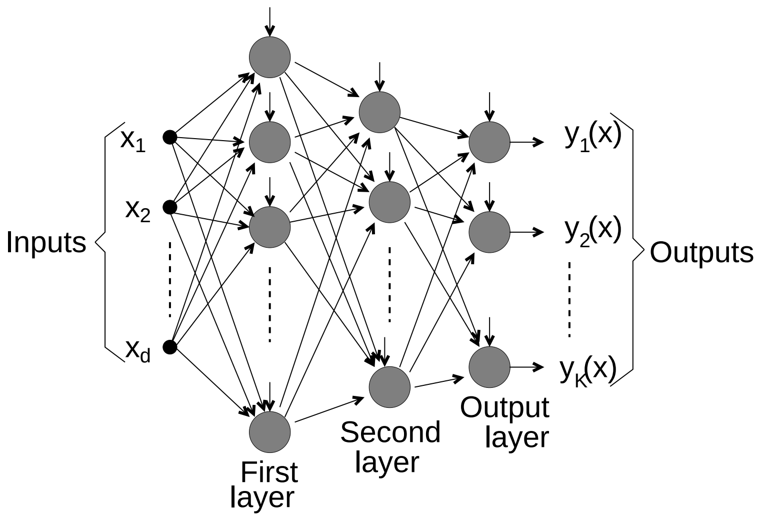
Handouts 4 & 5: Multi-Layer Perceptron: Introduction and Training



Mark Gales
mjfg@eng.cam.ac.uk
October 2001

Multi-Layer Perceptron

From the previous lecture we need a multi-layer perceptron to handle the XOR problem. More generally multi-layer perceptrons allow a neural network to perform arbitrary mappings.

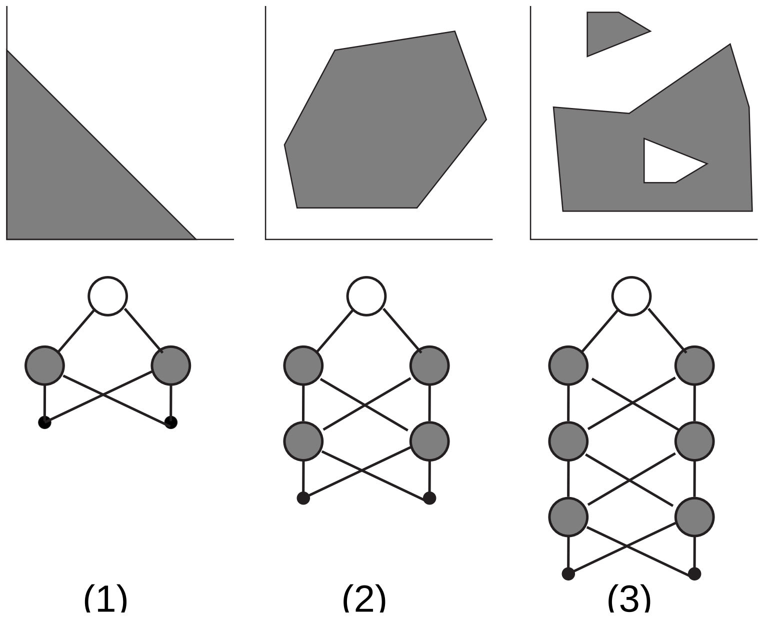


A 2-hidden layer neural network is shown above. The aim is to map an input vector $\mathbf{x}$ into an output $\mathbf{y}(\mathbf{x})$. The layers may be described as:

- **Input** layer: accepts the data vector or pattern;
- **Hidden** layers: one or more layers. They accept the output from the previous layer, weight them, and pass through a, normally, non-linear activation function.
- **Output** layer: takes the output from the final hidden layer weights them, and possibly pass through an output non-linearity, to produce the target values.

Possible Decision Boundaries

The nature of the decision boundaries that may be produced varies with the network topology. Here only threshold (see the single layer perceptron) activation functions are used.



There are three situations to consider

1. **Single layer**: this is able to position a hyperplane in the input space.
2. **Two layers** (one hidden layer): this is able to describe a decision boundary which surrounds a single convex region of the input space.
3. **Three layers** (two hidden layers): this is able to to generate arbitrary decision boundaries

Note: any decision boundary can be approximated arbitrarily closely by a two layer network having sigmoidal activation functions.

Number of Hidden Units

From the previous slide we can see that the number of hidden layers determines the decision boundaries that can be generated. In choosing the number of layers the following considerations are made.

- Multi-layer networks are harder to train than single layer networks.
- A two layer network (one hidden) with sigmoidal activation functions can model any decision boundary.

Two layer networks are most commonly used in pattern recognition (the hidden layer having sigmoidal activation functions).

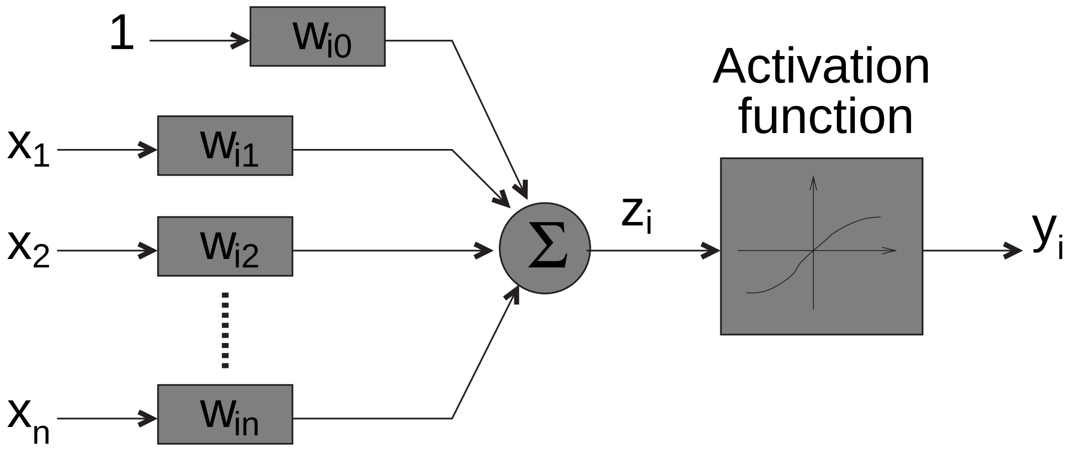
How many units to have in each layer?

- The number of output units is determined by the number of output classes.
- The number of inputs is determined by the number of input dimensions
- The number of hidden units is a design issue. The problems are:
  - too few, the network will not model complex decision boundaries;
  - too many, the network will have poor *generalisation*.

Hidden Layer Perceptron

The form of the hidden, and the output, layer perceptron is a generalisation of the single layer perceptron from the previous lecture. Now the weighted input is passed to a general *activation function*, rather than a threshold function.

Consider a single perceptron. Assume that there are n units at the previous level.



The output from the perceptron, y_i may be written as

$$y_i = \phi(z_i) = \phi(w_{i0} + \sum_{j=1}^{d} w_{ij}x_j)$$

where $\phi()$ is the activation function.

We have already seen one example of an activation function the threshold function. Other forms are also used in multi-layer perceptrons.

Note: the activation function is not necessarily non-linear. However, if linear activation functions are used much of the power of the network is lost.

Activation Functions

There are a variety of non-linear activation functions that may be used. Consider the general form

$$y_j = \phi(z_j)$$

and there are n units, perceptrons, for the *current* level.

- **Heaviside** (or step) function:

  $$y_j = \begin{cases} 0, & z_j < 0 \\ 1, & z_j \geq 0 \end{cases}$$

  These are sometimes used in *threshold* units, the output is binary.

- **Sigmoid** (or logistic regression) function:

  $$y_j = \frac{1}{1 + \exp(-z_j)}$$

  The output is continuous, $0 \leq y_j \leq 1$.

- **Softmax** (or normalised exponential or generalised logistic) function:

  $$y_j = \frac{\exp(z_j)}{\Sigma_{i=1}^{n} \exp(z_i)}$$

  The output is positive and the sum of all the outputs at the current level is 1, $0 \leq y_j \leq 1$.

- **Hyperbolic tan** (or tanh) function:

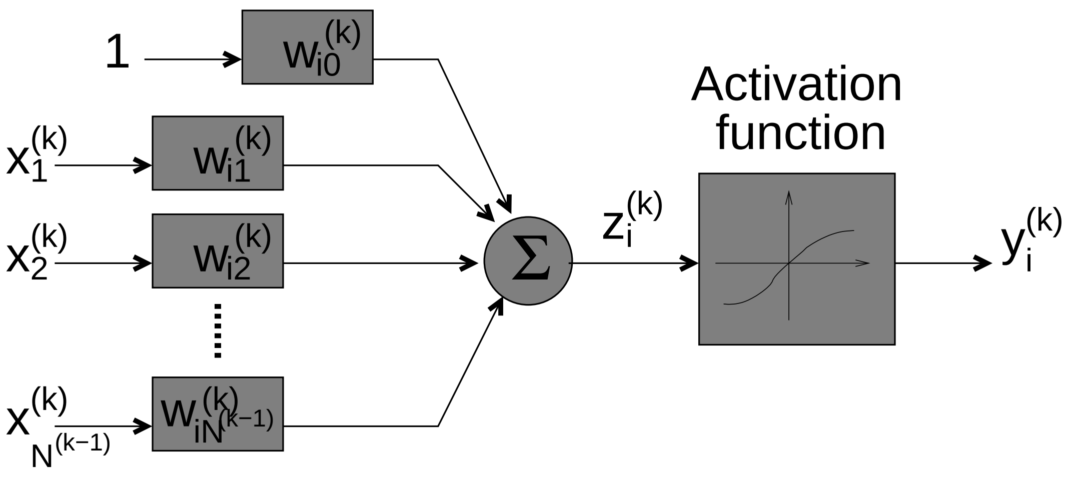
  $$y_j = \frac{\exp(z_j) - \exp(-z_j)}{\exp(z_j) + \exp(-z_j)}$$

  The output is continuous, $-1 \leq y_j \leq 1$.

Notation Used

Consider a multi-layer perceptron with:

- d-dimensional input data;
- L hidden layers ($L+1$ layer including the output layer);
- $N^{(k)}$ units in the k^{th} level;
- K-dimensional output.



The following notation will be used

- $\mathbf{x}^{(k)}$ is the input to the k^{th} layer
- $\tilde{\mathbf{x}}^{(k)}$ is the extended input to the k^{th} layer

$$\tilde{\mathbf{x}}^{(k)} = \begin{bmatrix} 1 \\ \mathbf{x}^{(k)} \end{bmatrix}$$

- $\mathbf{W}^{(k)}$ is the weight matrix of the k^{th} layer. By definition this is a $N^{(k)} \times N^{(k-1)}$ matrix.

Notation (cont)

- $\tilde{\mathbf{W}}^{(k)}$ is the weight matrix including the bias weight of the k^{th} layer. By definition this is a $N^{(k)} \times (N^{(k-1)}+1)$ matrix.
$$\tilde{\mathbf{W}}^{(k)} = \left[\begin{array}{cc} \mathbf{w}_0^{(k)} & \mathbf{W}^{(k)} \end{array} \right]$$
- $\mathbf{z}^{(k)}$ is the $N^{(k)}$-dimensional vector defined as
$$z_j^{(k)} = \tilde{\mathbf{w}}_j^{(k)} \tilde{\mathbf{x}}^{(k)}$$
- $\mathbf{y}^{(k)}$ is the output from the k^{th} layer, so
$$y_j^{(k)} = \phi(z_j^{(k)})$$

All the hidden layer activation functions are assumed to be the same $\phi()$. Initially we shall also assume that the output activation function is also $\phi()$.

The following matrix notation feed forward equations may then used for a multi-layer perceptron with input $\mathbf{x}$ and output $\mathbf{y}(\mathbf{x})$.

$$\begin{aligned} \mathbf{x}^{(1)} &= \mathbf{x} \\ \mathbf{x}^{(k)} &= \mathbf{y}^{(k-1)} \\ \mathbf{z}^{(k)} &= \tilde{\mathbf{W}}^{(k)} \tilde{\mathbf{x}}^{(k)} \\ \mathbf{y}^{(k)} &= \boldsymbol{\phi}(\mathbf{z}^{(k)}) \\ \mathbf{y}(\mathbf{x}) &= \mathbf{y}^{(L+1)} \end{aligned}$$

where $1 \leq k \leq L+1$.

The target values for the training of the networks will be denoted as $\mathbf{t}(\mathbf{x})$ for training example $\mathbf{x}$.

Training Criteria

A variety of training criteria may be used. Assuming we have supervised training examples

$$\{\{\mathbf{x}_1, \mathbf{t}(\mathbf{x}_1)\} \ldots, \{\mathbf{x}_n, \mathbf{t}(\mathbf{x}_n)\}\}$$

Some standard examples are:

- **Least squares error**: one of the most common training criteria.

$$\begin{aligned} E &= \frac{1}{2}\sum_{p=1}^{n} ||\mathbf{y}(\mathbf{x}_p) - \mathbf{t}(\mathbf{x}_p)||^2 \\ &= \frac{1}{2}\sum_{p=1}^{n}\sum_{i=1}^{K} (y_i(\mathbf{x}_p) - t_i(\mathbf{x}_p))^2 \end{aligned}$$

  This may be derived from considering the targets as being corrupted by zero-mean Gaussian distributed noise.

- **Cross-Entropy for two classes**: consider the case when $t(\mathbf{x})$ is binary (and softmax output). The expression is

$$E = -\sum_{p=1}^{n} (t(\mathbf{x}_p)\log(y(\mathbf{x}_p)) + (1 - t(\mathbf{x}_p))\log(1 - y(\mathbf{x}_p)))$$

  This expression goes to zero with the "perfect" mapping.

- **Cross-Entropy for multiple classes**: the above expression becomes (again softmax output)

$$E = -\sum_{p=1}^{n}\sum_{i=1}^{K} t_i(\mathbf{x}_p)\log(y_i(\mathbf{x}_p))$$

  The minimum value is now non-zero, it represents the *entropy* of the target values.

Network Interpretation

We would like to be able to interpret the output of the network. Consider the case where a least squares error criterion is used. The training criterion is

$$E = \frac{1}{2}\sum_{p=1}^{n}\sum_{i=1}^{K}\left(y_i(\mathbf{x}_p) - t_i(\mathbf{x}_p)\right)^2$$

In the case of an infinite amount of training data, $n \rightarrow \infty$,

$$\begin{aligned} E &= \frac{1}{2}\sum_{i=1}^{K}\int\int (y_i(\mathbf{x}) - t_i(\mathbf{x}))^2 p(t_i(\mathbf{x}), \mathbf{x}) dt_i(\mathbf{x}) d\mathbf{x} \\ &= \frac{1}{2}\sum_{i=1}^{K}\int\left[\int (y_i(\mathbf{x}) - t_i(\mathbf{x}))^2 p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x})\right] p(\mathbf{x}) d\mathbf{x} \end{aligned}$$

Examining the term inside the square braces

$$\begin{aligned} &\int (y_i(\mathbf{x}) - t_i(\mathbf{x}))^2 p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x}) \\ &\quad = \int (y_i(\mathbf{x}) - \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\} + \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\} - t_i(\mathbf{x}))^2 p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x}) \\ &\quad = \int (y_i(\mathbf{x}) - \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\})^2 + (\mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\} - t_i(\mathbf{x}))^2 p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x}) \\ &\qquad\quad + \int 2(y_i(\mathbf{x}) - \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\})(\mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\} - t_i(\mathbf{x})) p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x}) \end{aligned}$$

where

$$\mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\} = \int t_i(\mathbf{x}) p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x})$$

We can write the cost function as

$$\begin{aligned} E &= \frac{1}{2}\sum_{i=1}^{K}\int (y_i(\mathbf{x}) - \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\})^2 p(\mathbf{x}) d\mathbf{x} \\ &\quad + \frac{1}{2}\sum_{i=1}^{K}\int \left(\mathcal{E}\{t_i(\mathbf{x})^2|\mathbf{x}\} - (\mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\})^2\right) p(\mathbf{x}) d\mathbf{x} \end{aligned}$$

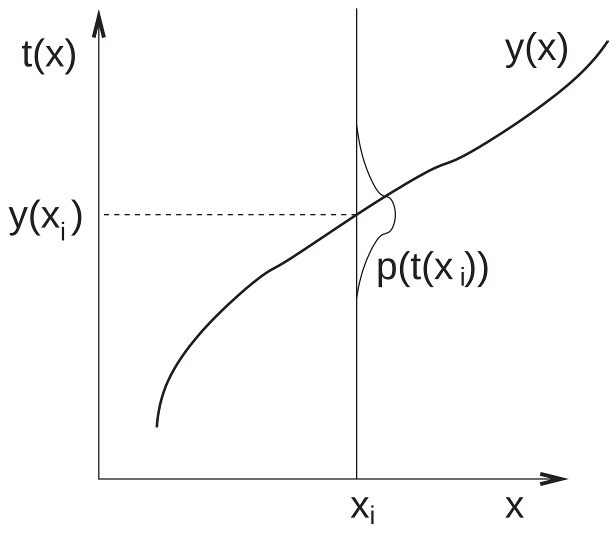
The second term is not dependent on the weights, so is not affected by the optimisation scheme.

Network Interpretation (cont)

The first term in the previous expression is minimised when it equates to zero. This occurs when

$$y_i(\mathbf{x}) = \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\}$$

The output of the network is the conditional average of the target data. This is the *regression* of $t_i(\mathbf{x})$ conditioned on x. So



the network models the posterior independent of the topology, but in practice require:

- an infinite amount of training data, or knowledge of correct distribution for x (i.e. $p(\mathbf{x})$ is known or derivable from the training data);
- the topology of the network is "complex" enough that final error is small;
- the training algorithm used to optimise the network is good - it finds the global maximum.

Posterior Probabilities

Consider the multi-class classification training problem with

- d-dimensional feature vectors: $\mathbf{x}$;
- K-dimensional output from network: $\mathbf{y}(\mathbf{x})$;
- K-dimensional target: $\mathbf{t}(\mathbf{x})$.

We would like the output of the network, $\mathbf{y}(\mathbf{x})$, to approximate the posterior distribution of the set of K classes. So

$$y_i(\mathbf{x}) \approx P(\omega_i|\mathbf{x})$$

Consider training a network with:

- means squared error estimation;
- 1-out-of$K$ coding, i.e.

$$t_i(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \omega_i \\ 0 & \text{if } \mathbf{x} \notin \omega_i \end{cases}$$

The network will act as a d-dimensional to K-dimensional mapping.

Can we interpret the output of the network?

Posterior Probabilities (cont)

From the previous regression network interpretation we know that

$$\begin{aligned} y_i(\mathbf{x}) &= \mathcal{E}\{t_i(\mathbf{x})|\mathbf{x}\} \\ &= \int t_i(\mathbf{x}) p(t_i(\mathbf{x})|\mathbf{x}) dt_i(\mathbf{x}) \end{aligned}$$

As we are using the 1-out-of-K coding

$$p(t_i(\mathbf{x})|\mathbf{x}) = \sum_{j=1}^{K} \delta(t_i(\mathbf{x}) - \delta_{ij}) P(\omega_j|\mathbf{x})$$

where

$$\delta_{ij} = \begin{cases} 1, & (i = j) \\ 0, & \text{otherwise} \end{cases}$$

This results in

$$y_i(\mathbf{x}) = P(\omega_i|\mathbf{x})$$

as required.

The same limitations are placed on this proof as the interpretation of the network for regression.

Compensating for Different Priors

The standard approach to described at the start of the course was to use Bayes' law to obtain the posterior probability

$$P(\omega_j|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_j)P(\omega_j)}{p(\mathbf{x})}$$

where the priors class priors, $P(\omega_j)$, and class conditional densities, $p(\mathbf{x}|\omega_j)$, are estimated separately. For some tasks the two use different training data (for example for speech recognition, the language model and the acoustic model).

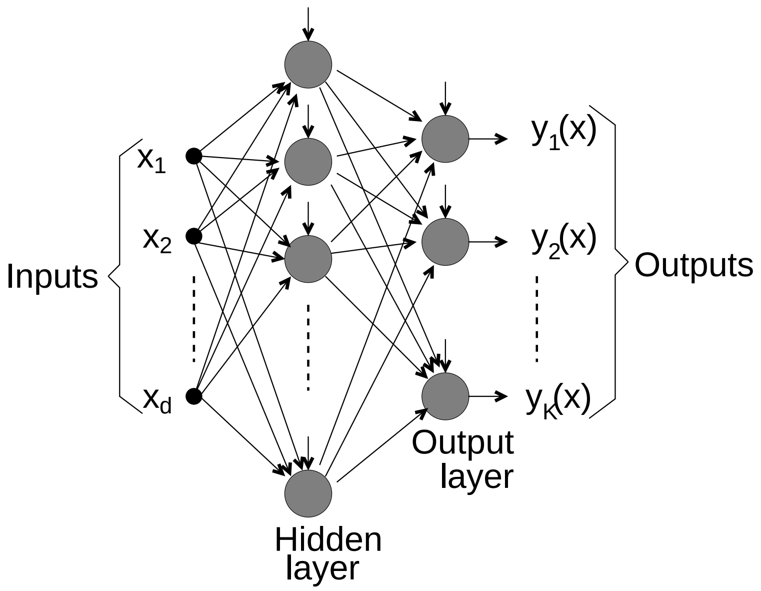
How can this difference in priors from the training and the test conditions be built into the neural network framework where the posterior probability is directly calculated? Again using Bayes' law

$$p(\mathbf{x}|\omega_j) \propto \frac{P(\omega_j|\mathbf{x})}{P(\omega_j)}$$

Thus if posterior is divided by the *training data prior* a value proportional to the class-conditional probability can be obtained. The standard form of Bayes' rule may now be used with the appropriate, different, prior.

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.



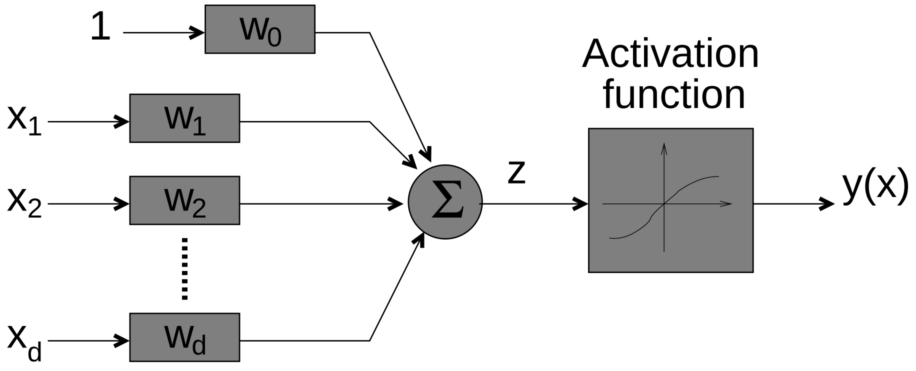
A single hidden layer network is shown above. As previously mentioned with sigmoidal activation functions arbitrary decision boundaries may be obtained with this network topology.

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)

We can calculate the effect that a change in z has 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 rate with 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

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

All these expressions are known so we can write

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

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

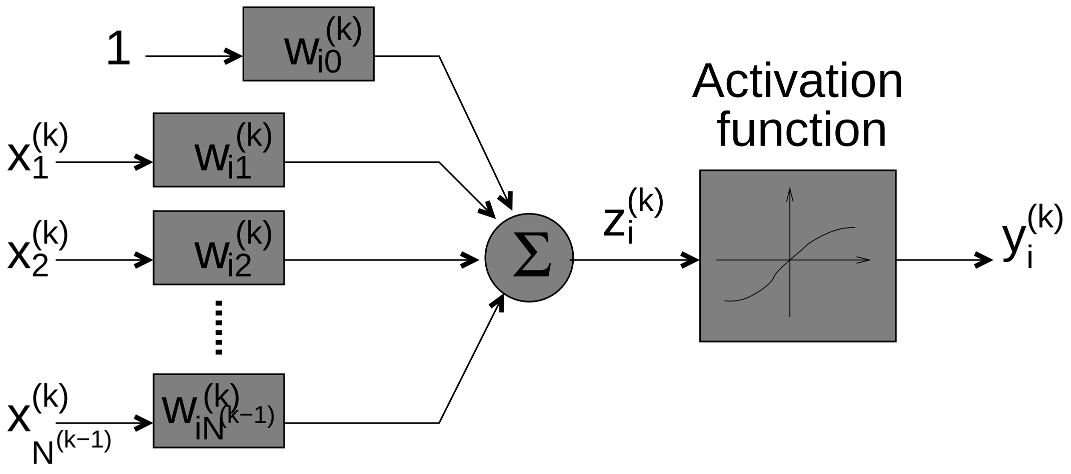
$$\boldsymbol{\nabla} E = \sum_{p=1}^{n}(y(\mathbf{x}_p) - t(\mathbf{x}_p))y(\mathbf{x}_p)(1 - y(\mathbf{x}_p))\tilde{\mathbf{x}}_p$$

So for a single layer we can use gradient descent schemes to find the “best” weight values.

However we want to train multi-layer perceptrons!

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. Furthermore we will not assume that $y_j^{(k)}$ only depends on the $z_j^{(k)}$. For example a softmax function may be used. 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

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

In matrix notation we can write

$$\frac{\partial E}{\partial \tilde{\mathbf{W}}^{(k)}} = \boldsymbol{\delta}^{(k)} \tilde{\mathbf{x}}^{(k)\prime}$$

We need to find a recursion for $\boldsymbol{\delta}^{(k)}$.

$$\begin{aligned}
\boldsymbol{\delta}^{(k)} &= \left(\frac{\partial E}{\partial \mathbf{z}^{(k)}}\right) \\
&= \left(\frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{z}^{(k)}}\right) \left(\frac{\partial E}{\partial \mathbf{z}^{(k+1)}}\right) \\
&= \left(\frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{z}^{(k)}}\right) \left(\frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{y}^{(k)}}\right) \boldsymbol{\delta}^{(k+1)}
\end{aligned}$$

But we know from the forward recursions

$$\frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{y}^{(k)}} = \frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{x}^{(k+1)}} = \mathbf{W}^{(k+1)\prime}$$

This yields the recursion

$$\boldsymbol{\delta}^{(k)} = \boldsymbol{\Lambda}^{(k)} \mathbf{W}^{(k+1)\prime} \boldsymbol{\delta}^{(k+1)}$$

Backward Error Recursion

where we define the *activation derivative matrix* for layer k as

$$\boldsymbol{\Lambda}^{(k)} = \frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{z}^{(k)}} = \begin{bmatrix} \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} & \frac{\partial y_2^{(k)}}{\partial z_1^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_1^{(k)}} \\ \frac{\partial y_1^{(k)}}{\partial z_2^{(k)}} & \frac{\partial y_2^{(k)}}{\partial z_2^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_2^{(k)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1^{(k)}}{\partial z_{N^{(k)}}^{(k)}} & \frac{\partial y_2^{(k)}}{\partial z_{N^{(k)}}^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_{N^{(k)}}^{(k)}} \end{bmatrix}$$

This has given a matrix form of the *backward recursion* for the error back propagation algorithm.

We need to have an initialisation of the backward recursion. This will be from the output layer (layer $L+1$)

$$\begin{aligned} \boldsymbol{\delta}^{(L+1)} &= \frac{\partial E}{\partial \mathbf{z}^{(L+1)}} \\ &= \left(\frac{\partial \mathbf{y}^{(L+1)}}{\partial \mathbf{z}^{(L+1)}}\right)\left(\frac{\partial E}{\partial \mathbf{y}^{(L+1)}}\right) \\ &= \boldsymbol{\Lambda}^{(L+1)}\left(\frac{\partial E}{\partial \mathbf{y}(\mathbf{x})}\right) \end{aligned}$$

$\boldsymbol{\Lambda}^{(L+1)}$ is the activation derivative matrix for the output layer.

Error Back Propagation

To calculate $\boldsymbol{\nabla} E^{(p)}\big|_{\boldsymbol{\theta}[\tau]}$ ($\boldsymbol{\theta}[\tau]$ 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 the set of derivative matrices $\boldsymbol{\Lambda}^{(k)}$ for all layers.
3. Compute $\frac{\partial E}{\partial \mathbf{y}(\mathbf{x})}\big|_{\boldsymbol{\theta}[\tau]}$ (the gradient at the output layer).
4. Using the back-propagation formulae back propagate the derivatives through the network.

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

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

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

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

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 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 maxima.

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 if first-order terms 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

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

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

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

  where

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

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

- **Quickprop**. Here

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

Conjugate Directions

Assume that we have optimised in one direction, $\mathbf{d}[\tau]$.

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

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

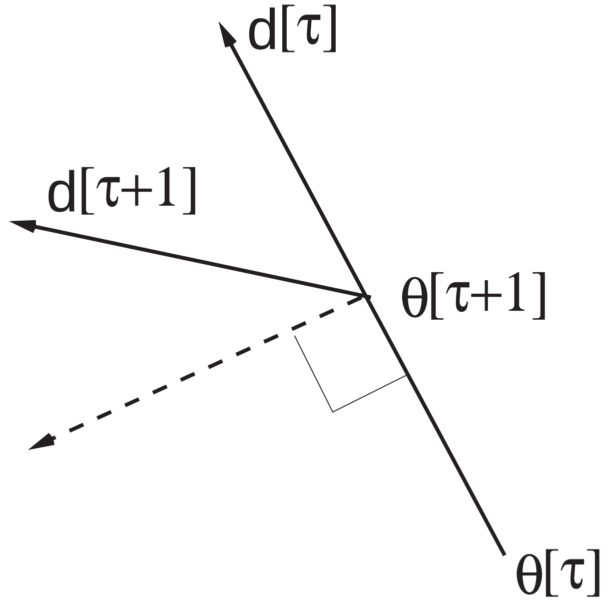
Using a first order Taylor series expansion

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

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 book by Bishop for details).

Input Transformations

If the input to the network are not normalised the training time may become very large. The data is therefore *normalised*. Here the data is transformed to

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

where

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

and

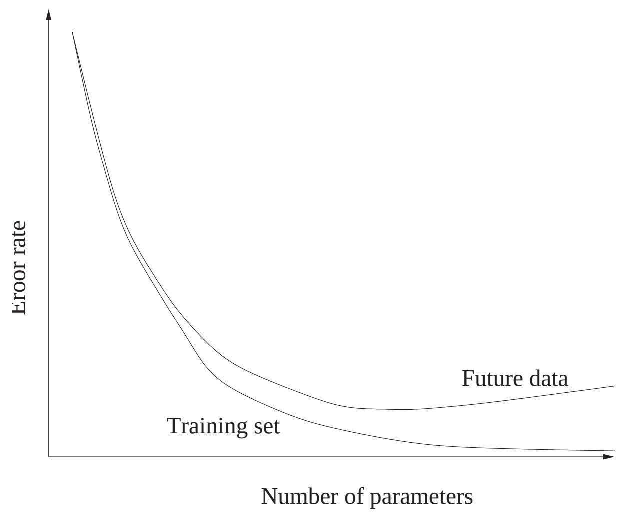
$$\sigma_i^2 = \frac{1}{n} \sum_{p=1}^{n} (x_{pi} - \mu_i)^2$$

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.

Generalisation

In the vast majority of pattern processing tasks, the aim is not to get good performance on the training data (in supervised training we know the classes!), but to get good performance on some previously unseen test data.



Typically the performance actually goes as above. As the number of model parameters, N, increase the training data likelihood is *guaranteed* to increase (provided the parameter optimisation scheme is "sensible"). The associated classification error rate *usually* decreases. However the test (future) set performance has a maximum in the likelihood and associated minimum in error rate. Then the likelihood decreases and the error rate increases.

The objective in any pattern classification task is to have the minimum test set (future data) error rate.

Regularisation

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