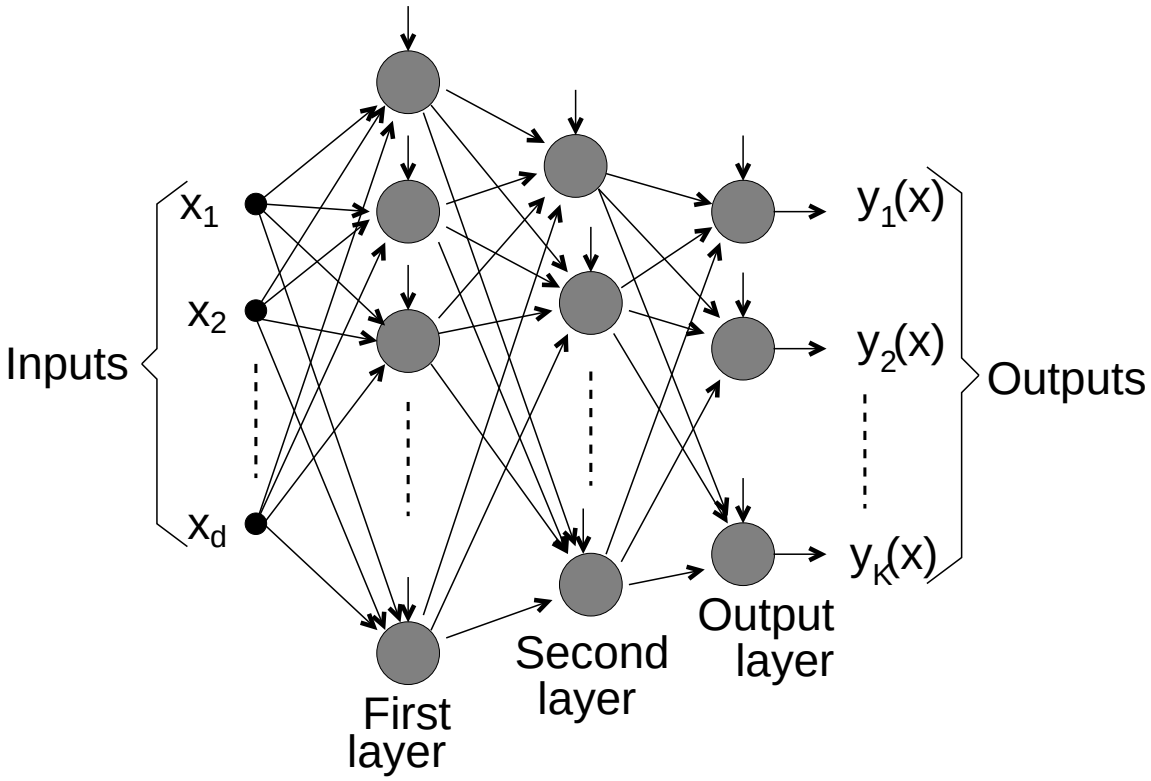
University of Cambridge
Engineering Part IIB

Module 4F10: Statistical Pattern Processing

Handout 6: Multi-Layer Perceptrons I



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

Introduction

In these lectures we will look at Multi-Layer Perceptrons (MLPs) which are more powerful than the Single-Layer models which construct linear decision boundaries.

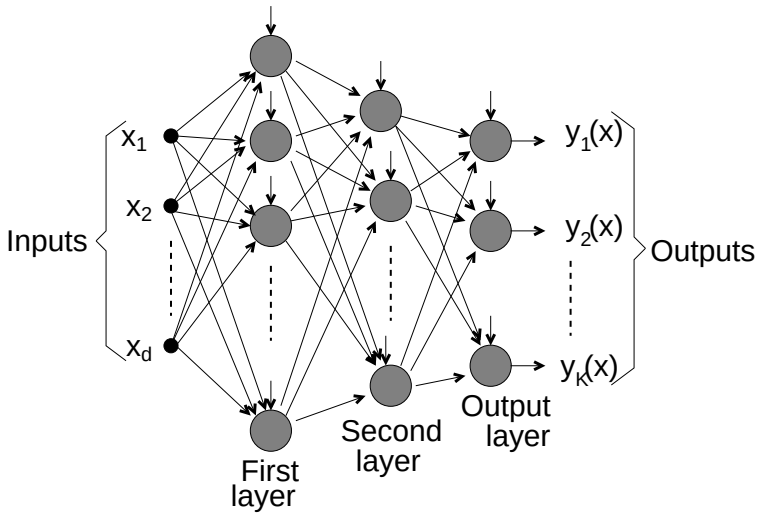
MLPs are classified as a type of Artificial Neural Network: the computation is performed using a set of (many) simple units with weighted connections between them. Furthermore there are learning algorithms to set the values of the weights and the same basic structure (with different weight values) is able to perform many tasks.

In this and the following lecture we will consider

- Overall structure of multi-layer perceptrons
- Decision boundaries that they can form
- Training Criteria
- Networks as posterior probability estimators
- Basic Error back-propagation training algorithm
- Improved training methods

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.

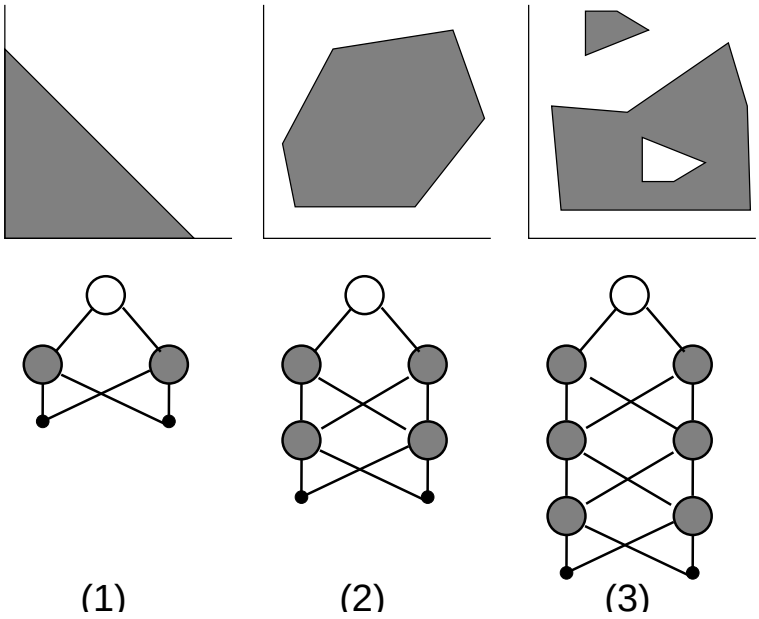


A 2-hidden layer neural network is shown above. The aim is to map an input vector $\mathbf{x}$ into an output $\mathbf{y}(\mathbf{x})$. The layers may be described as:

- **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 if there are enough hidden units.

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).

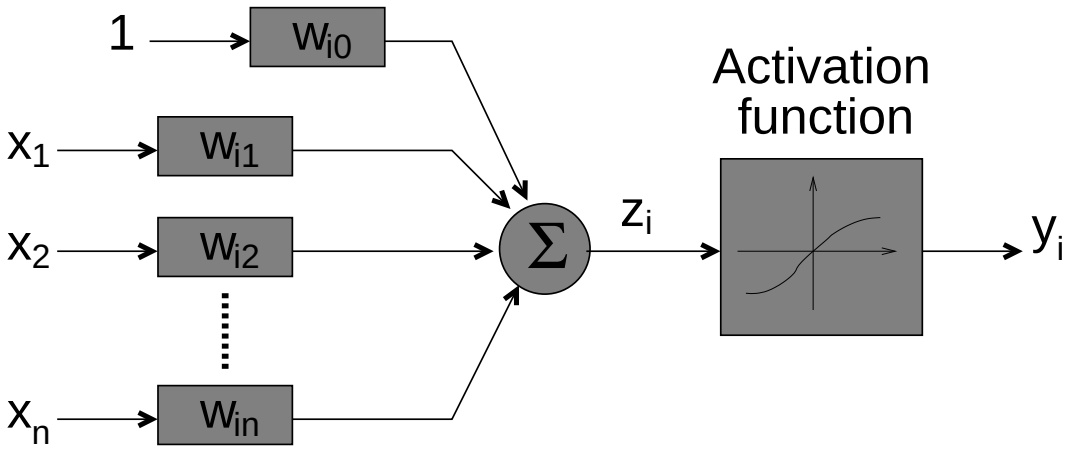
How many units to have in each layer?

- 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:

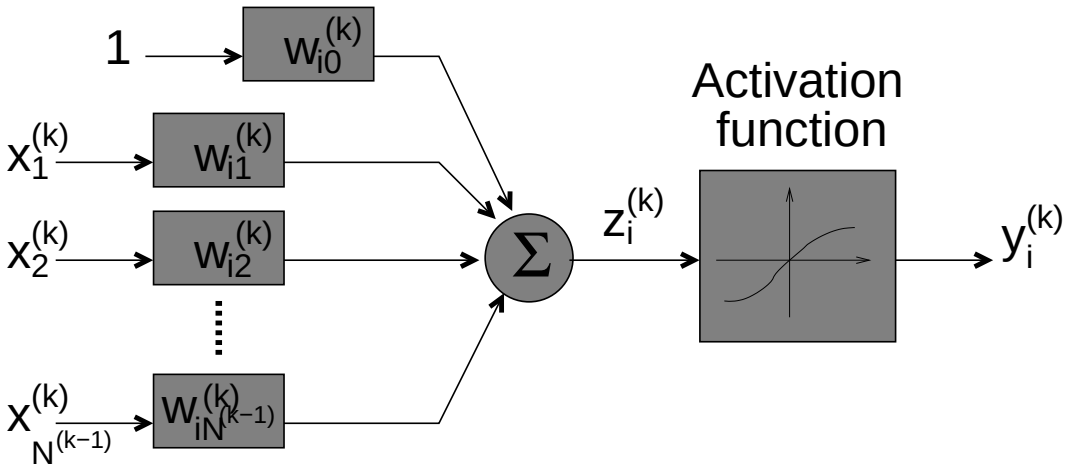
$$y_j = \frac{\exp(z_j) - \exp(-z_j)}{\exp(z_j) + \exp(-z_j)}$$

  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

$$\mathbf{z}^{(k)} = \tilde{\mathbf{W}}^{(k)} \tilde{\mathbf{x}}^{(k)}$$

- $\mathbf{y}^{(k)}$ 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

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

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

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

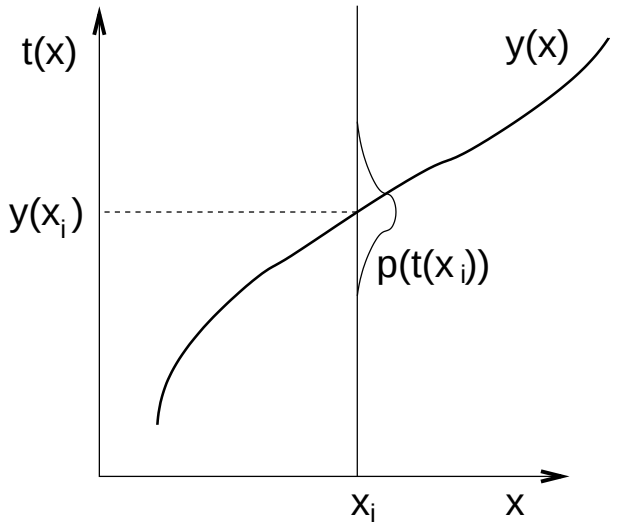
The second term is not dependent on the weights, so is not affected by the optimisation scheme.

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 $\mathbf{x}$.



So the network models the regression of the targets independent of the topology (& this can be the class posterior probability, see next slides), but in practice require:

- an infinite amount of training data, or knowledge of correct distribution for $\mathbf{x}$ (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.