Learning from Data
Layered Neural Networks

Course lecturer: Amos Storkey
a.storkey@ed.ac.uk
Course page : http://www.anc.ed.ac.uk/∼amos/lfd/

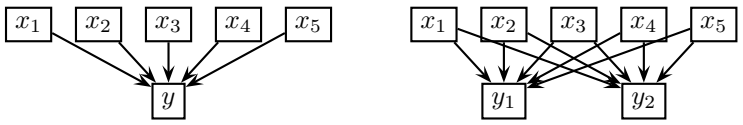


Figure 1: (Left) A simple perceptron. We use square boxes to emphasise the deterministic nature of the network. (Right) We can use two perceptrons with weights $\mathbf{w}_1$ and $\mathbf{w}_2$ to model a mapping $(x_1, x_2, x_3, x_4, x_5) \to (y_1, y_2)$

1 Sequential Layered Processing

In natural systems, information processing is often found to occur in stages. For example, in human vision, the light falling on the retina is transformed first in a non-linear logarithmic fashion. Local parts of the image are then "recognised" by neurons specialised to respond to particular local image patterns. The information from these "feature" extraction neurons is then fed into subsequent layers which correspond to higher cognitive functions. Artificial layered networks mimic such sequential processing.

In this chapter, we shall consider that each "neuron" or processing unit computes a deterministic function of its input. In this sense :*Neural Networks are graphical representations of functions.*

2 The Perceptron

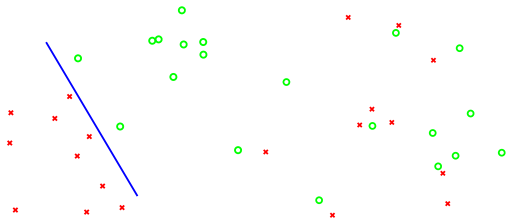
The perceptron is essentially just a single neuron like unit that computes a non-linear function y of its inputs x,

$$y = g\left(\sum_j w_j x_j + \mu\right) = g\left(\mathbf{w}^T\mathbf{x} + \mu\right) \tag{2.1}$$

where the weights $\mathbf{w}$ encode the mapping that this neuron performs. Graphically, this is represented in fig(1). We can consider the case of several outputs as follows:

$$y_i = g\left(\sum_j w_{ij} x_j + \mu_i\right)$$

and can be used to model an input-output mapping $\mathbf{x} \to \mathbf{y}$, see fig(1)(right). Coupled with an algorithm for finding suitable weights, we can use a perceptron for regression. Of course, the possible mappings the perceptron encodes is rather restricted, so we cannot hope to model all kinds of complex input-output mappings successfully. For example, consider the case in which $g(x) = \Theta(x)$ – that is, the output is a binary valued function ($\Theta(x) = 1$ if $x \geq 0$, $\Theta(x) = 0$ if $x < 0$) . In this case, we can use the perceptron for binary classification. With a single output we can then classify an input $\mathbf{x}$ as belonging to one of two possible classes. Looking at the perceptron, equation (2.1), we see that we will classify the input as being in class 1 if $\sum_j w_j x_j + \mu \geq 0$, and as in the other class if $\sum_j w_j x_j + \mu < 0$. Mathematically speaking, the decision boundary then forms a hyperplane in the $\mathbf{x}$ space, and which class we associate with a datapoint $\mathbf{x}$ depends on which side of the hyperplane this datapoint lies, see fig(2).



(a) A linearly separable problem

(b) A non-linearly separable problem

Figure 2: Linear separability: The data in (a) can be classified correctly using a hyperplane classifier such as the simple perceptron, and the data is termed linearly separable. This is not the case in (b) so that a simple perceptron cannot correctly learn to classify this data without error.

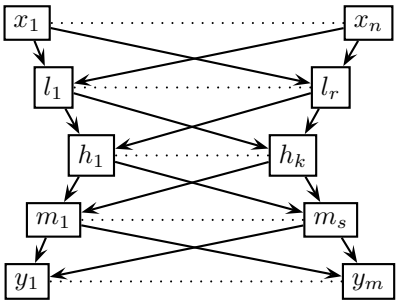


Figure 3: A multilayer perceptron (MLP) with multiple hidden layers, modeling the input output mapping $\mathbf{x} \rightarrow \mathbf{y}$. This is a more powerful model than the single hidden layer, simple perceptron. We used here boxes to denote the fact that the nodes compute a deterministic function of their inputs.

3 Multilayer Perceptrons

If the data that we are modeling is not linearly separable, we have a problem since we certainly cannot model this mapping using the simple perceptron. Similarly, for the case of regression, the class of function mappings that our perceptron forms is rather limited, and only the simplest regression input-output mappings will be able to be modelled correctly with a simple perceptron. These observations were pointed out in 1969 by Minsky and Papert and depressed research in this area for several years. A solution to this perceived problem was eventually found which included "hidden" layers in the perceptron, thus increasing the complexity of the mapping.

Transfer Functions

Each hidden node computes a non-linear function of a weighted linear sum of its inputs. The specific non-linearity used is called the *transfer function*. In principle, this can be any function, and different for each node. However, it is most common to use an S-shaped (sigmoidal) function of the form $\sigma(x) = 1/(1 + e^{-x})$. This particular choice is mathematically convenient since it has the nice derivative property $d\sigma(x)/dx = \sigma(x)(1-\sigma(x))$. Another popular choice is the sigmoidal function $\tanh(x)$. Less "biological" transfer functions include the Gaussian, $e^{-\frac{1}{2}x^2}$, see fig(4). For example, in fig(3), we

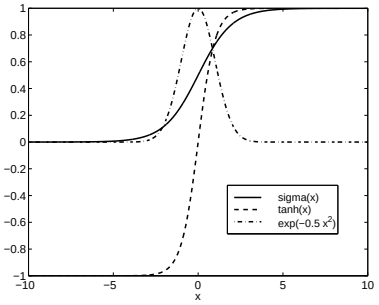


Figure 4: Common types of transfer functions for neural networks.

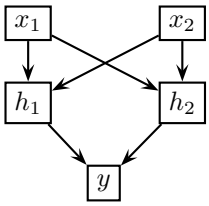


Figure 5: A MLP with one hidden layer.

plot a simple single hidden layer function,

$$h_1 = \sigma\left(\mathbf{w}_1^T\mathbf{x} + b_1\right), h_2 = \sigma\left(\mathbf{w}_2^T\mathbf{x} + b_2\right), y = r(\mathbf{v}^T\mathbf{h} + b_3) \tag{3.1}$$

where the adaptable parameters are $\boldsymbol{\theta} = \{\mathbf{w}_1, \mathbf{w}_2, \mathbf{v}, b_1, b_2, b_3\}$. Note that the output function $r(\cdot)$ in the final layer is usually taken as the idendity function $r(x) = x$ in the case of regression – for classification models, we use a sigmoidal function. The biases, b_1, b_2 are important in shifting the position of the "bend" in the sigmoid function, and b_3 shifts the bias in the output.

Generally, the more layers that there are in this process, the more complex becomes the class of functions that such MLPs can model. One such example is given in fig(3), in which the inputs are mapped by a non-linear function into the first layer outputs. In turn, these are then fed into subsequent layers, effectively forming new inputs for the layers below. However, it can be shown that, provided that there are sufficiently many units, a single hidden layer MLP can model an arbitrarily complex input-output regression function. This may not necessarily give rise to the most efficient way to represent a function, but motivates why we concentrate mainly on single hidden layer networks here.

3.1 Understanding Neural Networks

There are a great number of software packages that automatically set up and train the networks on provided data. However, following our general belief that our predictions are only as good as our assumptions, if we really want to have some faith in our model, we need to have some insight into what kinds of functions neural networks are.

The central idea of neural networks is that each neuron computes some

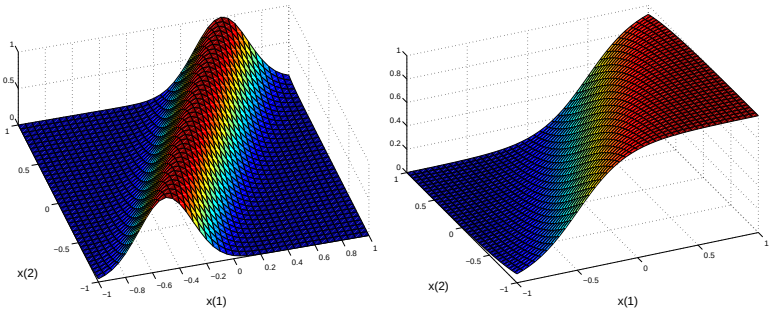


Figure 6: The output for a single neuron, $\mathbf{w} = (-2.5, 5)^T$, $b = 0$. Left: The network output using the transfer function $\exp(-0.5x^2)$. Right: using the transfer function $\sigma(x)$. Note how the network output is the same along the direction perpendicular (orthogonal) to $\mathbf{w}$, namely $\mathbf{w}^\perp = \lambda(2, 1)^T$.

function of a linear combination of its inputs:

$$h(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b) \tag{3.2}$$

where g is the transfer function, usually taken to be some non-linear function. Alternatively, we can write

$$h(\mathbf{x}) = g(a(\mathbf{x})) \tag{3.3}$$

where we define the *activation* $a(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$. The parameters for the neuron are the weight vector $\mathbf{w}$ and bias b. Each neuron in the network has its own weight and bias, and in principle, its own transfer function. Consider a vector $\mathbf{w}^\perp$ defined to be orthogonal to $\mathbf{w}$, that is, $\mathbf{w}^T\mathbf{w}^\perp = 0$. Then

$$a(\mathbf{x} + \mathbf{w}^\perp) = \left(\mathbf{x} + \mathbf{w}^\perp\right)^T \mathbf{w} + b \tag{3.4}$$

$$= \mathbf{x}^T\mathbf{w} + b + \underbrace{\left(\mathbf{w}^\perp\right)^T \mathbf{w}}_{0} \tag{3.5}$$

$$= a(\mathbf{x}) \tag{3.6}$$

Since the output of the neuron is only a function of the activation a, this means that any neuron has the same output along directions $\mathbf{x}$ which are orthogonal to $\mathbf{w}$. Such an effect is given in fig(6), where we see that the output of the neuron does not change along directions perpendicular to $\mathbf{w}$. This kind of effect is general, and for any transfer function, we will always see a ridge type effect. This is why a single neuron cannot achieve much on its own – essentially, there is only one direction in which the function changes (I mean that unless you go in a direction which has a contribution in the $\mathbf{w}$ direction, the function remains the same). If the input is very high dimensional, we only see variation in one direction.

Combining Neurons

In fig(7) we plot the output of a network of two neurons in a single hidden layer. The ridges intersect to produce more complex functions than single neurons alone can produce. Since we have now two neurons, the function will not change if we go in a direction which is simultaneously orthogonal to both $\mathbf{w}_1$ and $\mathbf{w}_2$. In this case, $\mathbf{x}$ is only two dimensional, so there is no

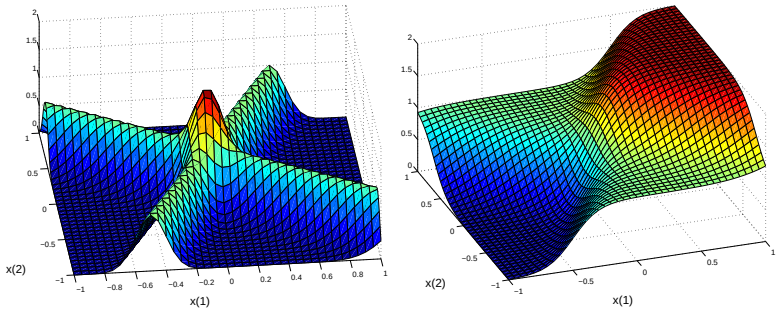


Figure 7: The combined output for two neurons, $\mathbf{w}_1 = (-5, 10)^T$, $b_2 = 0$, $\mathbf{w}_2 = (7, 5)^T$, $b_2 = 0.5$. The final output is linear, with weights $\mathbf{v} = (1, 1)^T$ and zero bias. Left: The network output using the transfer functions $\exp(-0.5x^2)$. Right: using the transfer function $\sigma(x)$ – this is exactly the function in equation (3.1) with r the identity function.

direction we can go along that will be orthogonal to both neuron weights. However, if $\mathbf{x}$ where higher dimensional, this would be possible. Hence, we now have variation along essentially two directions.

In general, if we had K neurons interacting in a single hidden layer in this way, we would essentially have a function which can vary along K independent directions in the input space.

4 Training multi-layered perceptrons

To a statistician, neural networks are a class of non-linear (adaptive basis function) models. Let us consider, for convenience, only a single output variable y. Given a set of input-output pairs, $D = \{(\mathbf{x}^\mu, y^\mu), \mu = 1, \ldots, P\}$, how can we find appropriate "weights" $\boldsymbol{\theta}$ that minimise the error that the network makes in fitting this function? In neural-network terminology, we would define an "energy" function that measures the errors that the network makes, and then try to minimise this function with respect to $\boldsymbol{\theta}$.

Regression

For example, a suitable choice of energy or error function for regression might be

$$E_{train}(\boldsymbol{\theta}) = \sum_\mu \left(y^\mu - f(\mathbf{x}^\mu, \boldsymbol{\theta})\right)^2 \tag{4.1}$$

where $f(x^\mu, \boldsymbol{\theta})$ is the output of the network for input $\mathbf{x}^\mu$, given that the parameters describing the network are $\boldsymbol{\theta}$. We can train this network by any standard (non-linear) optimisation algorithm, such as conjugate gradient descent.

Classification

A suitable choice of energy or error function to minimise for classification is the negative log likelihood (if $y^\mu \in \{0, 1\}$)

$$E_{train}(\boldsymbol{\theta}) = -\sum_\mu \left(y^\mu \log f^\mu + (1 - y^\mu) \log(1 - f^\mu)\right) \tag{4.2}$$

where $f^\mu = f(\mathbf{x}^\mu, \boldsymbol{\theta})$. In this case, we would need that the final output $r(x)$ is bounded between 0 and 1 in order that it represents a probability. The

case of more than two classes is handled in a similar way using the so-called soft-max function (see Bishops book for references).

Regularisation

In principle, the problem of training neural networks is equivalent to the general statistical problem of fitting models to data. One of the main problems when fitting complex non-linear models to data is how to prevent "over-fitting", or, more generally, how to select the model that not only fits the data, but also generalises well to new data. We have already discussed this issue in some generality, and found that one approach is to use a penalty term which encourages smoother functions. In the case of MLPs, smoother functions can be encouraged if we penalise large weight values. The reason for this is that the larger the weights $\mathbf{w}^i$ are, the more rapidly the function can change as $\mathbf{x}$ changes (since we could flip from close to one near saturated region of the sigmoid to the other with only a small change in $\mathbf{x}$).

A term which penalises large weights,

$$E_{regtrain}(\boldsymbol{\theta}) = E_{train}(\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}^T \boldsymbol{\theta} \tag{4.3}$$

We can set λ as usual by using a validation set.

4.1 Single Hidden Layer

A MLP with a single hidden layer is

$$f(\mathbf{x}, \boldsymbol{\theta}) = r\left(\sum_{i=1}^{K} v_i g(\mathbf{w}_i \cdot \mathbf{x} + b_i) + b\right) \tag{4.4}$$

an example of which is given in fig(3).

Regression

In the case of regression, we would use an output function r to be the identity, and the squared output to form the error[1]:

$$E(\boldsymbol{\theta}) = \sum_{\mu=1}^{P} (f(\mathbf{x}^\mu, \boldsymbol{\theta}) - y^\mu)^2 + \lambda \sum_{k=1}^{K} (\mathbf{w}_k)^T \mathbf{w}_k \tag{4.5}$$

To use the conjugate gradients algorithm to optimise this objective function, we need to know the derivatives with respect to all the parameters $\partial E / \partial \theta_i$.

$$\frac{\partial E}{\partial \theta_i} = 2 \sum_{\mu=1}^{P} (f(\mathbf{x}^\mu, \boldsymbol{\theta}) - y^\mu) \frac{\partial f(\mathbf{x}^\mu, \boldsymbol{\theta})}{\partial \theta_i} + 2 \sum_{k} \sum_{j=1}^{dim(\mathbf{w}_k)} w_{j,k} \frac{\partial w_{j,k}}{\partial \theta_i} \tag{4.6}$$

The final term is zero unless we are differentiating with respect to a parameter that is included in the regularisation term. If θ_i is included in the regularisation term, then the final term simply is $2\theta_i$. All that is required then is to calculate the derivatives of f with respect to the parameters. This is a straightforward exercise in calculus, and we leave it to the reader to show that, for example,

$$\frac{\partial f(\mathbf{x}^\mu, \boldsymbol{\theta})}{\partial v_1} = g(\mathbf{w}_1^T \mathbf{x}^\mu + b_1) \tag{4.7}$$

[1] There is no need to penalise the biases, since they only really affect a translation of the functions, and don't affect how bumpy the functions are.

and

$$\frac{\partial f(\mathbf{x}^\mu, \boldsymbol{\theta})}{\partial w_{1,2}} = v_2 g(\mathbf{w}_2^T \mathbf{x}^\mu + b_1) x_1^\mu \tag{4.8}$$

Example code for regression using a single hidden layer is given below. It is straightforward to adapt this for classification. This code is not fully vectorised for clarity, and also uses the `scg.m` function, part of the NETLAB (see `http://www.ncrg.aston.ac.uk`) package which implements many of the methods in these chapters.

```
% Single hidden layer Neural Network to do regression
% Note : this code assumes that each datapoint is a row vector
% Also required is the NETLAB scg.m, available from http://www.ncrg.aston.ac.uk

% training data:
x=randn(20,1); % one dimensional inputs
%x=randn(20,2); % two dimensional inputs
y = sin(4*sum(x,2)); % one dimensional ouputs
n = size(x,2);
K = 10; % number of hidden units
% initial value for the parameters:
w = 0.01*randn(n,K); b = randn(1,K); v = randn(K,1); b0=1

nw = prod(size(w)); % number of weight parameters
th_init=[reshape(w,1,nw),b,v',b0];  % initial parameter vector
lambda = 0.1; % regularisation

% now use Scaled Conjugate Gradients to find optimal parameters:
options = zeros(1,18); options(9)=1; options(1)=1;options(14)=200;
[th_opt]=scg('E',th_init,options,'grad_E',x,y,nw,K,lambda);

% plot the result :
plot(x,y,'x'); hold on; xplot = [-3:0.01:3]';
plot(xplot,nnfn(xplot,th_opt,nw,K))

function E = E(th,x,y,nw,K,lambda)
 [f,w,b,v,v0] = nnfn(x,th,nw,K);
 E = sum((y-f).^2) + lambda*sum(sum(w.*w));

function gE = grad_E(th,x,y,nw,K,lambda)
  [f,w,b,v,b0] = nnfn(x,th,nw,K);
  gf = grad_nnfn(x,th,nw,K);
  gE = 2*sum(repmat(f-y,1,length(th)).*gf);
  grad_reg=zeros(size(gE));
  grad_reg(1:K*size(x,2))=2*lambda.*reshape(w,1,prod(size(w))); % regularisation
  gE = gE + grad_reg;

function [f,w,b,v,b0] = nnfn(x,th,nw,K)
% single hidden layer NN for regression
  [p,n]= size(x);
  % get the parameters from vector th
  w=reshape(th(1:nw),n,K);  b=th(nw+1:nw+K);
  v=th(nw+K+1:nw+2*K)';  b0 =th(nw+2*K+1);
  a = x*w + repmat(b,p,1); % hidden unit activations
%  h = 1./(1+exp(-a)); % sigmoidal hidden units
   h = exp(-0.5*a.^2); % Gaussian hidden units
  a0 = h*v + repmat(b0,p,1); % output activation
  f = a0; % output transfer function is the identity

function g = grad_nnfn(x,th,nw,K)
% single hidden layer NN for regression
  [p,n]= size(x);  [f,w,b,v,b0] =nnfn(x,th,nw,K);

  a = x*w + repmat(b,p,1); % hidden unit activion
  %  h = 1./(1+exp(-a)); % sigmoidal hidden units
  h = exp(-0.5*a.^2); % Gaussian hidden units

%  gb = repmat(v',p,1).*h.*(1-h); % sigmoidal hidden units
   gb = -repmat(v',p,1).*a.*h;  % Gaussian hidden units

   for mu =1:p % done using loop for clarity
     gw(mu,:) = reshape(x(mu,:)'*gb(mu,:),1,prod(size(w)));
   end
  gv=h;   gb0=ones(p,1);  g = [gw gb gv gb0];
```

4.2 Back Propagation

In computing the gradient of the error function, naively it appears that we need of the order of PW^2 operations (if W is the number of parameters in the model and P is the number of training patterns), since computing the output of the network involves roughly W summations for each of the P patterns, and the gradient is a W-dimensional vector. The essence of the backpropagation procedure is that the gradient can instead by computed in order PW operations. If the training set is very large, standard computation of the gradient over all training patterns is both time-consuming and sensitive to round-off errors. In that case, "on-line learning", with weight updates based on the gradient for individual patterns, offers an alternative. Back propagation is most useful in cases where there are more than one hidden layer in the network. In this case, the gradient can be computed more efficiently, and time saved therefore to find the optimal parameters.

4.3 Training ensembles of networks

A problem with neural networks is that they are difficult to train. This is because the surface of the error function $E(\boldsymbol{\theta})$ is very complicated and typically riddled with local minima. *No algorithm can guarantee to find the global optimum of the error surface.* Indeed, depending on the initial conditions that we use, the parameters found by the optimisation routine will in general be different. How are we to interpret these different solutions? Perhaps the simplest thing to do is to see which of the solutions has the best error on an independent validation set. Many algorithms have been proposed on how to combine the results of the separate networks into a single answer and for computing error bars that indicate the reliability of this answer. Imagine that we have used optimisation several times, and found the different solutions $\boldsymbol{\theta}^i, i = 1, \ldots, M$. One simple approach (for regression) is to combine the outputs of each of the trained models,

$$\bar{f}(\mathbf{x}) = \frac{1}{M} \sum_{i=1}^{M} f(\mathbf{x}, \boldsymbol{\theta}^i) \tag{4.9}$$

This is also useful since we can make an estimate of the variance in the predictions at a given point,

$$\mathrm{var}(f(\mathbf{x}) = \frac{1}{M} \sum_{i=1}^{M} \left(f(\mathbf{x}, \boldsymbol{\theta}^i) - \bar{f}\right)^2 \tag{4.10}$$

This can then be used to form error bars $\bar{f}(\mathbf{x}) \pm \sqrt{\mathrm{var}(f(\mathbf{x}))}$.

5 Neural Network Applications

5.1 General considerations

Most neural network applications derive from their learning capabilities. Mainly used are standard feedforward neural networks, either for classification or for regression. In these applications, the neural network serves as an alternative to standard statistical solutions. The underlying idea is that due their inherent nonlinearity, neural networks are better in modeling complex

relationships than classical statistical methods. In the end, this need not always be the case. One of the reasons for the popularity of neural networks is that, using standard tools, a reasonable solution can be obtained in a reasonable amount of time. This gave neural networks the image of "second best" solutions.

This viewpoint is understandable when we consider standard neural packages, i.e., stand-alone software for horizontal applications across different domains. As explained before, a (feedforward) neural network is not principally different from standard statistical models. However, many of the available software packages do not treat neural networks as such. On the contrary, the thoroughness that surrounds classical statistical models (tools for computing confidence intervals, model and feature selection, outlier detection, and so on) is replaced with sloppiness under the presumption that neural networks are so powerful that they can do without. Furthermore, the standard packages are often not flexible enough to handle all peculiarities of the problem to be solved. So, indeed, it is relatively straightforward to build simple neural applications with standard packages, but these are easily outperformed when more effort is put in.

An alternative, rapidly gaining ground, are neural toolboxes for statistical packages like SPSS, SAS, and Matlab. The threshold for usage is somewhat higher, the user has to buy and become acquainted with the statistical engine supporting the neural algorithms. The advantages over stand-alone neural packages are increasing flexibility and easy integration with statistical tools. This makes them very well suited for building tailored neural applications, examples of which will be given below. In these tailored applications, the neural machinery often only takes up a small but essential piece of the total solution.

5.2 Example applications

Applications of neural networks can be found in any domain where data is available to support decisions. The general term in this context is "data mining", also referred to as "knowledge discovery in databases". Neural networks are often quoted as *the* technology to be used, others being machine learning, clustering, statistics, and visualization techniques. Here we will give a short overview of neural data mining applications.

Marketing. : Customer credit, billing, and purchases were some of the first business transactions to be automated with computers, yielding huge amounts of data available for mining in search for knowledge that can improve marketing results or lower marketing costs. A typical example is direct mailing. A test mailing is made to a small subset of customer. A feedforward neural network is used to model the response as a function of the characteristics of the customer. This model can then be used to determine who should be included in the subsequent mass mailing and which offers should be included. Other examples can be found in customer relationship management (enhance the revenues of existing customers by tuning marketing messages) and preventing customer retention (identifying customers who are likely to switch to competitors).

Retail and logistics. : Neural networks are used for demand forecasting. In principle, these

are standard time-series prediction problems in which neural networks have to compete with standard tools such as Box-Jenkins and ARMA. A nontrivial application has been developed for the prediction of single-copy newspaper sales. In this setting, predictions are needed on a daily basis for a huge set of individual outlets. By combining all outlets in a single neural network, the outlets can "learn from each other", e.g., by extracting typical demand features. In this setting, the neural architecture yields a clear benefit over standard approaches that treat all outlets individually.

Finance. : Finance is *the* domain for success stories of the type "neural networks predict stock returns". Despite the fact that there may be successful solutions that temporarily work, including neural ones, the general feeling is that anything can be lucky. There are other problems in the financial domain that are better suited for a neural approach. Examples are the detection of fraudulent transactions with credit cards, portfolio optimization, predicting bankruptcies, and credit risk assessment. In most of these applications, neural networks are either used for time-series prediction or classification, their benefit over other tools being the capability of dealing with nonlinearities.

Manufacturing. : The quality of a manufactured product often depends on the settings of many parameters. The exact relationship between these settings and the quality are often not well understood and too complex to describe with a physical or chemical model. Trained on examples yielding good and bad qualities, neural networks can provide a solution. Other applications of neural networks are in job shop scheduling and automatic inspection. In these control applications, neural networks are mainly used for function fitting to model (part of) the process one needs to control.

Health and medicine. : Some of the applications in health and medicine resemble those in marketing and finance: detection of fraudulent insurance claims, risk assessment of clients, and so on. Other application relate to automatic diagnosis of diseases. It should be mentioned that in many medical applications, the surplus value of using neural networks over standard tools such as Cox survival analysis is often rather small. In many cases the databases are too small and too noisy to provide evidence of very complex relationships that would benefit from a neural approach.

Energy and utility. : Prediction of energy demand is very relevant, both for large consumers who are often charged based on their peak energy usage, and for providers that have to anticipate upon extreme demands. In this context neural networks are used as nonlinear time-series predictors. A quite different kind of application in this area involves the detection of likely sites for gas and oil deposits. Based on all kinds of measurements at test drilling sites, neural networks are used to predict changes in the strata of rock, which relates to the presence of mineral deposits.

Summarising, there are indeed many (potential) applications of neural networks. Often it is not so much the technique that matters, but more the insight that appropriate use of available data can help to solve the problem. The exact technique becomes important for large-scale problems, where

small improvements have major consequences. In general, neural machinery is most promising if there are nonlinear relationships between explanatory and response variables and sufficient data to find these. Especially the latter condition need not always be fulfilled, in which case the surplus value of neural networks over simpler techniques is limited. The trends are vertical applications, embedding in other statistical techniques, and combination of knowledge about the domain and available data. Most applications are based on classical frequentist statistics, although those using Bayesian methodology are increasingly common.