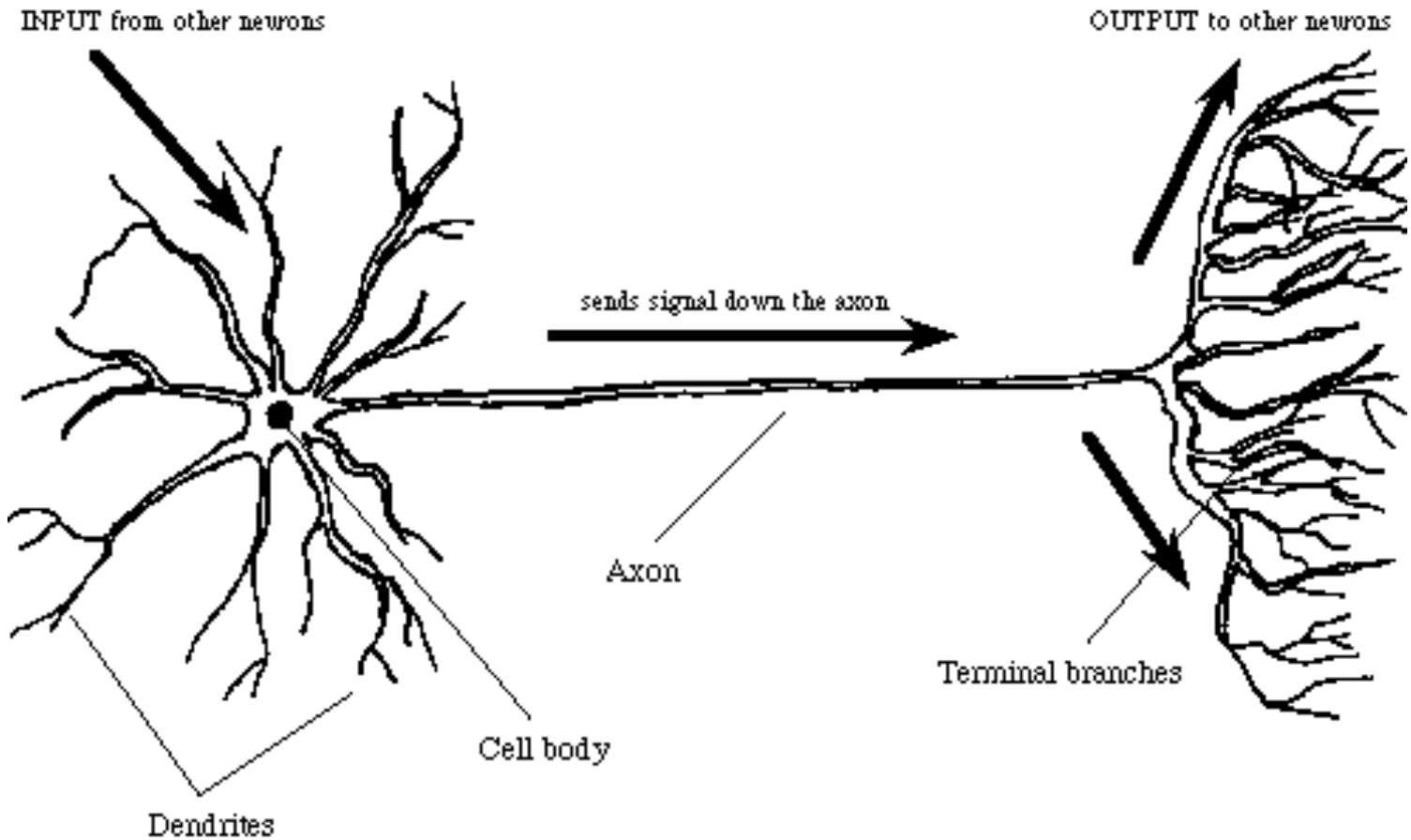
Ch.6 Feed-forward neural network models [Book, Chap.4]

The human brain is a massive network of about 10^{11} interconnecting neural cells called neurons, performing highly parallel computing.



The brain is exceedingly robust and fault tolerant.

A neuron is a very simple processor – its "clockspeed" is of the order of a millisecond, about a million times slower than that of a computer – yet the human brain beats the computer on many tasks involving vision, motor control, common sense, etc.

Hence, the power of the brain lies in its massive network structure.

What computational capability is offered by a massive network of interconnected neurons has led to the development of the field of *neural networks* (NN).

Scientists from many disciplines are interested in *artificial* neural networks, i.e. how to borrow ideas from neural network structures to

develop better techniques in computing, artificial intelligence, data analysis, modelling and prediction.

The most widely used type of NN is the *feed-forward* neural network, where the signal in the model only proceeds forward from the inputs through any intermediate layers to the outputs without any feedback.

6.1 McCulloch and Pitts model [Book, Sect. 4.1]

First NN model of significance is the McCulloch and Pitts (1943) model.
A neuron receives stimulus (signals) from its neighbours, and if the total stimulus exceeds some threshold, the neuron becomes activated and fires off (outputs) a signal.

Their model neuron is a binary threshold unit, i.e. it receives a weighted sum of its inputs from other units, and outputs either 1 or 0 depending on whether the sum exceeds a threshold.

For a neuron, if x_i denotes the input signal from the ith neighbour, which is weighted by a *weight parameter* w_i, the output of the neuron y is given by

$$y = H\left(\sum_i w_i x_i + b\right), \tag{1}$$

where b is called an *offset* or *bias* parameter, and H is the Heaviside step function,

$$H(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0\,. \end{cases} \tag{2}$$

By adjusting b, the threshold level for the firing of the neuron can be changed.

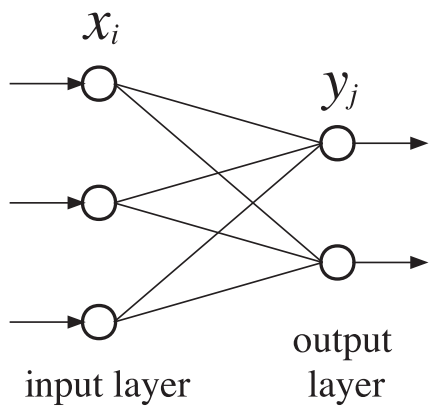
McCulloch and Pitts proved that networks made up of such neurons are capable of performing any computation a digital computer can, though there is no provision that such an NN computer is necessarily faster or easier.

In the McCulloch and Pitts model, the neurons are very similar to conventional logical gates, and there is no algorithm for finding the appropriate weight and offset parameters for a particular problem.

6.2 Perceptrons [Book, Sect. 4.2]

Next major advance is the *perceptron* model of Rosenblatt (1958, 1962) (and similar work by Widrow and Hoff (1960)). The

perceptron model consists of an input layer of neurons connected to an output layer of neurons:



Neurons are also referred to as *nodes* or *units* in the NN literature.

The key advance is the introduction of a *learning algorithm*, which finds the weight and offset parameters of the network for a particular problem.

An output neuron

$$y_j = f\left(\sum_i w_{ji} x_i + b_j\right), \tag{3}$$

where x_i denotes an input, f a specified transfer function known as an *activation function*, w_{ji} the weight parameter connecting the ith input neuron to the jth output neuron, and b_j the offset or bias parameter of the jth output neuron, and $-b_j$ is also called the *threshold parameter*.

A more compact notation eliminates the distinction between weight and offset parameters by expressing $\sum_i w_i x_i + b$ as
$\sum_i w_i x_i + w_0 = \sum_i w_i x_i + w_0 1$,
i.e. b can be regarded as simply the weight w_0 of an extra constant input $x_0 = 1$, and (3) can be written as

$$y_j = f\left(\sum_i w_{ji} x_i\right), \tag{4}$$

with the summation starting from $i = 0$.

The step function was originally used as the activation function, but other continuous functions can also be used.

Given data of the inputs and outputs, one can train the network so that the model output values y_j derived from the inputs using (3) are as closed as possible to the data $y_{\mathrm{d}j}$ (also called the *target*), by finding the appropriate weight and offset parameters in (3).

With the parameters known, (3) gives the output variable y_j as a function of the input variables. Details of the training process will be given later.

In situations where the output are not binary variables, a commonly used activation function is the *logistic sigmoidal function*, or simply the logistic function, where 'sigmoidal' means an S-shaped function:

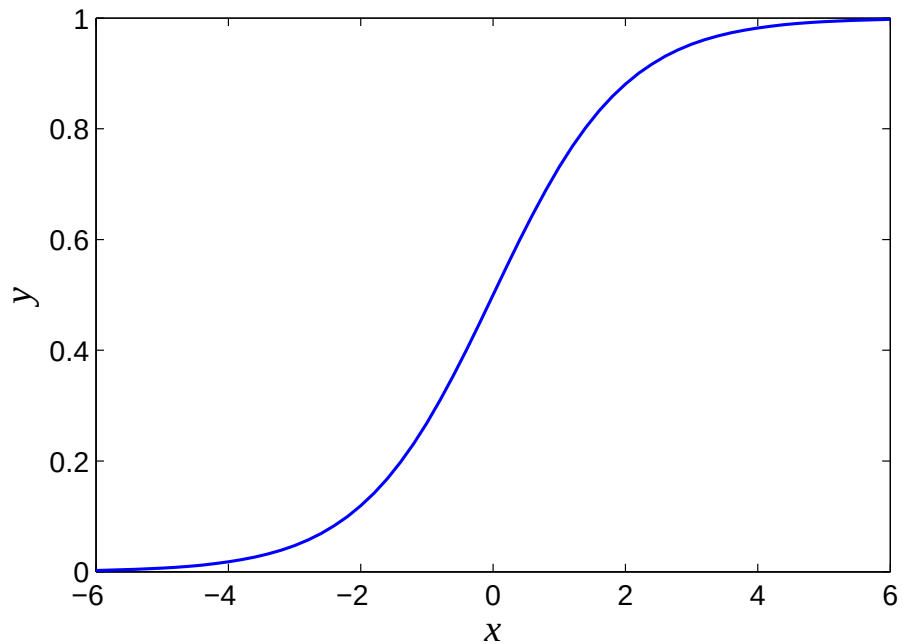
$$f(x) = \frac{1}{1 + e^{-x}} \quad . \tag{5}$$

This function has an asymptotic value of 0 when $x \to -\infty$, and rises smoothly as x increases, approaching the asymptotic value of 1 as $x \to +\infty$.

The logistic function is used because it is nonlinear & differentiable.

The role of the weight and offset parameters in $f\left(\sum_i w_i x_i + b\right)$ can be readily seen in the univariate form $f(wx + b)$, where a large w gives a steeper transition from 0 to 1 ($w \to \infty$ approaches the

Heaviside function), while increasing b slides the logistic curve to the left along the x-axis.



The advent of the perceptron model led to great excitement; however, the serious limitations of the perceptron model was soon recognized (Minsky and Papert, 1969).

Simple examples are provided by the use of perceptrons to model the Boolean logical operators AND and XOR (the exclusive OR):

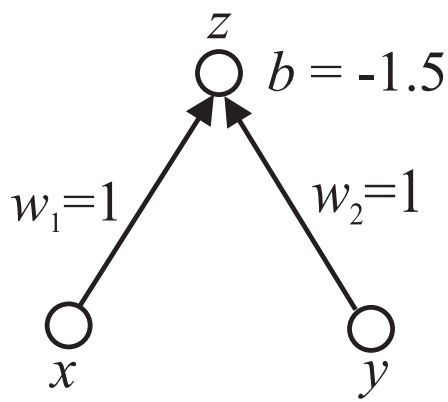
For $z = x.\mathrm{AND}.y$, z is TRUE only when both x and y are TRUE.

For $z = x.\mathrm{XOR}.y$, z is TRUE only when exactly one of x or y is TRUE.

Let 0 denotes FALSE and 1 denotes TRUE, and the activation function used is the step function .

The simple perceptron model which represents $z = x.\mathrm{AND}.y$, maps from (x, y) to z in the following manner:

$$\begin{aligned}(0,0) &\rightarrow 0\\ (0,1) &\rightarrow 0\\ (1,0) &\rightarrow 0\\ (1,1) &\rightarrow 1\,.\end{aligned}$$



However, a perceptron model for $z = x.\mathrm{XOR}.y$ does not exist! One cannot find a perceptron model which will map:

$$\begin{aligned} (0,0) &\rightarrow 0 \\ (0,1) &\rightarrow 1 \\ (1,0) &\rightarrow 1 \\ (1,1) &\rightarrow 0\,. \end{aligned}$$

The difference between the two problems is shown in Fig.:

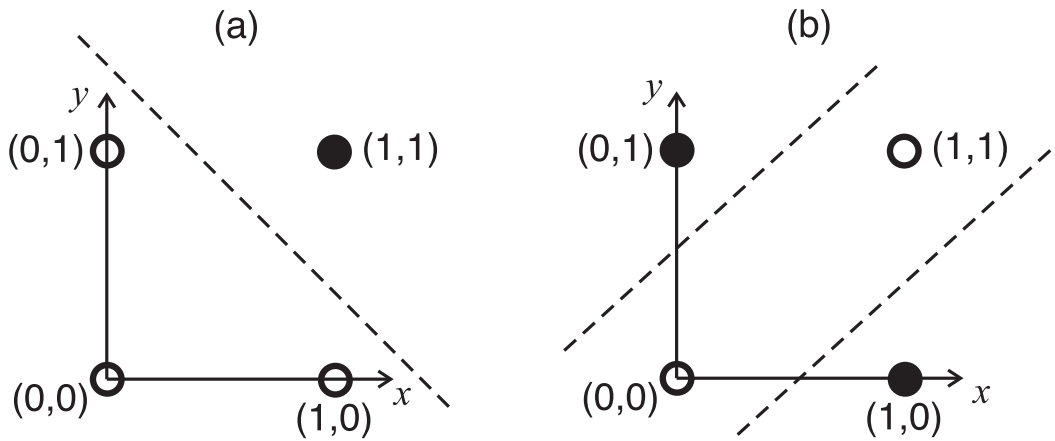


Figure : The classification of the input data (x, y) by the Boolean logical operator (a) AND, and (b) XOR (exclusive OR). In (a), the decision boundary separating the TRUE domain (black circle) from the FALSE domain (white circles) can be represented by a straight (dashed) line, hence the problem is *linearly separable*; whereas in (b), two lines are needed, rendering the problem not linearly separable.

It is easy to see why the perceptron model is limited to a linearly separable problem. If the activation function $f(z)$ has a decision boundary at $z = c$, (3) implies that the decision boundary for the jth output neuron is given by $\sum_i w_{ji} x_i + b_j = c$, which is the equation of a straight line in the input **x**-space.

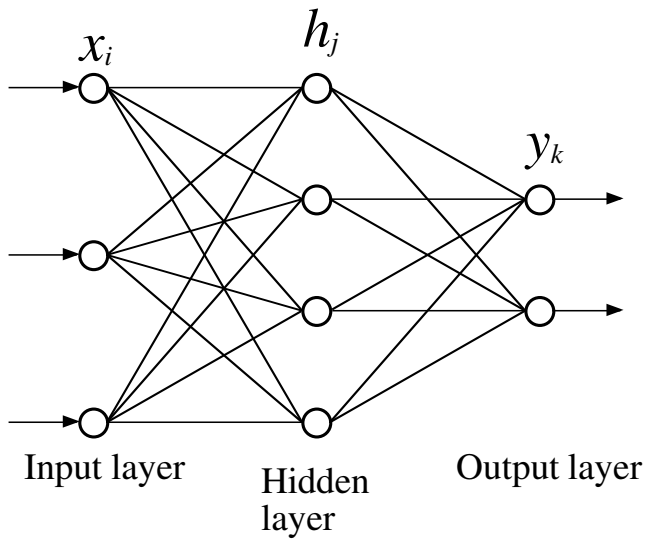
For an input **x**-space with dimension $n = 2$, there are 16 possible Boolean functions (among them AND and XOR), and 14 of the 16 are linearly separable.
When $n = 3$, 104 out of 256 Boolean functions are linearly separable.
When $n = 4$, the fraction of Boolean functions which are linearly separable drops further— only 1882 out of 65536 are linearly separable (Rojas, 1996).
As n gets large, the set of linearly separable functions forms a very tiny subset of the total set (Bishop, 1995).

Interests in NN research waned following the realization that the perceptron model is restricted to linearly separable problems.

6.3 Multi-layer perceptrons (MLP) [Book, Sect. 4.3]

When the limitations of the perceptron model was realized, it was felt that the NN might have greater power if additional '*hidden*' layers of neurons were placed between the input layer and the output layer— but there was then no algorithm which would solve for the parameters of the *multi-layer perceptrons* (MLP) .

Interests in NN revived in the mid 1980s when a way was found to solve for the MLP parameters.



The input signals x_i are mapped to the hidden layer of neurons h_j by

$$h_j = f\left(\sum_i w_{ji} x_i + b_j\right), \tag{6}$$

and then onto the output y_k,

$$y_k = g\left(\sum_j \tilde{w}_{kj} h_j + \tilde{b}_k\right), \tag{7}$$

where f and g are activation functions, w_{ji} and $\tilde{w}_{kj}$ weight parameter matrices, and b_j and $\tilde{b}_k$ are offset parameters.

To train the NN to learn from a dataset (the target), we minimize the *objective function* J (also referred to as the *cost function*, *error*

function, or *loss function*), defined here to be one half the *mean squared error* (MSE) between the model output and the target,

$$J = \frac{1}{N} \sum_{n=1}^{N} \left\{ \frac{1}{2} \sum_{k} \left[y_k^{(n)} - y_{\mathrm{d}k}^{(n)} \right]^2 \right\} \tag{8}$$

where $y_{\mathrm{d}k}$ is the target data, and there are N observations.

An optimization algorithm is needed to find the weight and offset parameter values which minimize the objective function, hence the MSE between the model output and the target.

The MSE is the most common form used for the objective function in nonlinear regression problems, [as minimizing the MSE is equivalent to maximizing the likelihood function assuming Gaussian error distribution].

Details on how to perform the nonlinear optimization will be presented later, as there are many choices of optimization schemes.

In general, nonlinear optimization is difficult, and convergence can be drastically slowed or numerically inaccurate if the input variables are poorly scaled. Hence it is important to *standardize* the input data before applying the NN model.

Besides the logistic function, another commonly used sigmoidal activation function is the *hyperbolic tangent function*

$$f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} . \tag{9}$$

This function has an asymptotic value of -1 when $x \to -\infty$, and rises smoothly as x increases, approaching the asymptotic value of 1 as $x \to +\infty$.

The tanh function can be viewed as a scaled version of the logistic function (Eq. 5):

Q1: Show that

$$\tanh(x) = 2\ \mathrm{logistic}(2x) - 1\,. \tag{10}$$

While the range of the logistic function is $(0, 1)$, the range of the tanh function is $(-1, 1)$. Hence the output of a tanh function does not have the positive systematic bias found in the output of a logistic function, which if input to the next layer of neurons, is

somewhat difficult for the network to handle, resulting in slower convergence during NN training.

Hence the tanh activation function is to be preferred over the logistic activation function in the hidden layer(s).

Since the range of the tanh function is $(-1, 1)$, the range of the network output will be similarly bounded if tanh is used for g in (7).

This is useful if the NN tries to classify the output into one of two classes, but may cause a problem if the output variables are unbounded.

Even if the output range is bounded within $[-1, 1]$, values such as -1 and 1 can only be represented by the tanh function at its asymptotic limit.

One possible solution is to scale the output variables, so they all lie within the range $(-0.9, 0.9)$— the range $(-0.9, 0.9)$ is preferred to $(-1, 1)$ as it avoids using the asymptotic range of the tanh function, resulting in faster convergence during NN training.

When the output is not restricted to a bounded interval, the identity activation function is commonly used for g, i.e. the output is simply a linear combination of the hidden neurons in the layer before,

$$y_k = \sum_j \tilde{w}_{kj} h_j + \tilde{b}_k \,. \tag{11}$$

For the 1-hidden layer NN, this means the output is just a linear combination of sigmoidal shaped functions.

Some confusion on how to count the number of layers:

The most common convention is to count the number of layers of mapping functions, which is equivalent to the number of layers of neurons excluding the input layer. The 1-hidden-layer NN will be referred to as a 2-layer NN.

[Some researchers count the total number of layers of neurons, and refer to the 1 hidden layer NN as a 3-layer NN].

Useful shorthand notation for describing the number of inputs, hidden and output neurons: a 3-4-2 network denotes a 1-hidden layer NN with 3 input, 4 hidden and 2 output neurons.

The total number of (weight and offset) parameters in an m_1-m_2-m_3 network is $N_{\mathrm{p}} = (m_1 + 1)m_2 + (m_2 + 1)m_3$, of which $m_1 m_2 + m_2 m_3 = m_2(m_1 + m_3)$ are weight parameters, and $m_2 + m_3$ are offset parameters.

In multiple linear regression problems with m predictors and 1 response variable, i.e. $y = a_0 + a_1 x_1 + \ldots + a_m x_m$, there are $m + 1$ parameters.

For corresponding nonlinear regression with an m-m_2-1 MLP network, there will be $N_{\mathrm{p}} = (m + 1)m_2 + (m_2 + 1)$ parameters, usually greatly exceeding the number of parameters in the multiple linear regression model.
Unlike the parameters in a multiple linear regression model, the parameters of an NN model are in general extremely difficult to interpret.
In a 1-hidden layer MLP, if the activation functions at both the hidden and output layers are linear, then it is easily shown that the outputs are simply linear combinations of the inputs.

The hidden layer is therefore redundant and can be deleted altogether. The MLP model simply reduces to multiple linear regression.

Hence, the presence of a nonlinear activation function at the hidden layer is essential for the MLP model to have nonlinear modelling capability.

While there can be exceptions, MLP are usually employed with $N_{\mathrm{p}} < N$, N being the number of observations in the dataset.

Ideally, one would like to have $N_{\mathrm{p}} \ll N$, but in many environmental problems, this is unattainable.

Q2: Given 50 observations in time, you want to use a 1-hidden layer MLP to model y as a function of $\mathbf{x}$, where $\mathbf{x}$ contains 6 predictors. How many hidden neurons can you use without violating $N_\mathrm{p} < N$?

In most climate problems, decent climate data have been available only after World War II.

Another problem is the number of predictors can be very large, although there can be strong correlations among predictors. PCA is commonly applied to the predictor variables, and the leading PCs served as inputs to the MLP network, to greatly reduce the number of input neurons and therefore N_p.

If there are multiple outputs, one has two choices: Build a single NN with multiple outputs, or build multiple NN models each with a single output.

If the output variables are correlated among themselves, then the single NN approach often leads to higher skills, since training separate networks for individual outputs does not take into account the relations between the output variables.

If the output variables are uncorrelated (e.g. the outputs are principal components), then training separate networks often leads to slightly higher skills, as this approach focuses the single-output NN (with fewer parameters than the multiple-output NN) on one output variable without the distraction from the other uncorrelated output variables.

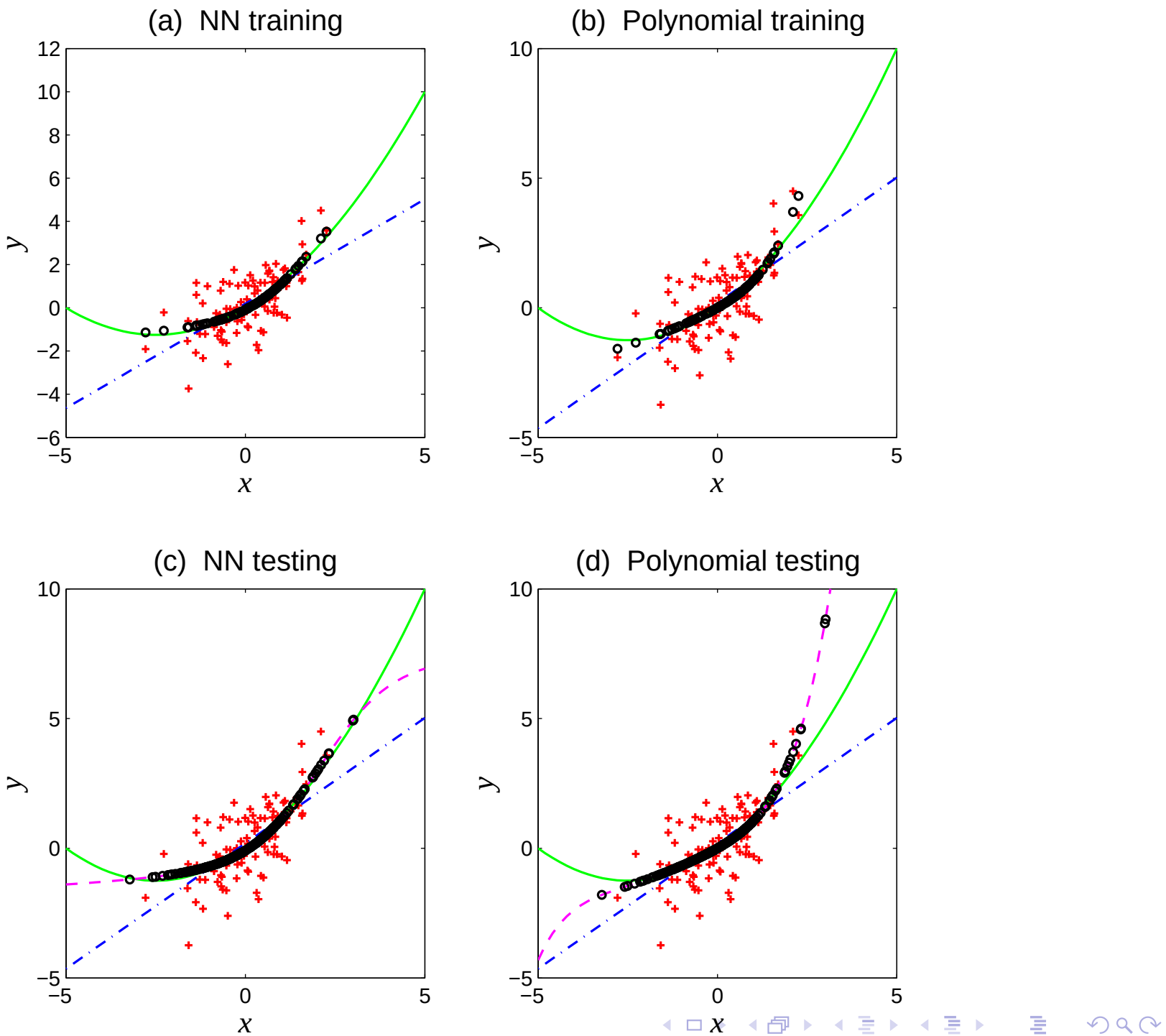
To model a nonlinear relation $y = f(\mathbf{x})$, why not use Taylor series

$$y = a_0 + \sum_{i_1=1}^{m} a_{i_1} x_{i_1} + \sum_{i_1=1}^{m} \sum_{i_2=1}^{m} a_{i_1 i_2} x_{i_1} x_{i_2} + \sum_{i_1=1}^{m} \sum_{i_2=1}^{m} \sum_{i_3=1}^{m} a_{i_1 i_2 i_3} x_{i_1} x_{i_2} x_{i_3} + \cdots . \tag{12}$$

In practice, only terms up to order k are kept, i.e. y is approximated by a kth order polynomial, and there are m input variables. The number of adjustable regression coefficients (a's) is $\mathrm{O}(m^k)$, i.e. of the order of m^k (Bishop, 1995).

In practice, m^k means the number of model parameters rise at an unacceptable rate as m increases.
In MLP, the number of parameters typically grows at $\mathrm{O}(m)$.
However, nonlinear optimization is needed to find the parameters.

Polynomials can do crazy extrapolations.


(a) NN training
(b) Polynomial training
(c) NN testing
(d) Polynomial testing
x
y

6.4 Back-propagation [Book, Sect. 4.4]

How to find the optimal weight and offset parameters which would minimize the objective function J?

To minimize J, one needs to know the gradient of J with respect to the parameters. The *back-propagation* algorithm gives the gradient of J through the backward propagation of the model errors.

The MLP problem could not be solved until the introduction of the back-propagation algorithm by Rumelhart et al. (1986), though the algorithm had actually been discovered in the Ph.D. thesis of Werbos (1974).

In general, the weights (i.e. parameters) are randomly initialized at the start of the optimization process. The inputs are mapped forward by the network, and the output model errors are obtained.

The back-propagation algorithm is composed of two parts: The first part computes the gradient of J by the backward propagation of the model errors.
The second part descents along the gradient towards the minimum of J.
This descent method is called *gradient descent* or *steepest descent*, and is very inefficient.

This process of mapping the inputs forward and then backpropagating the error is iterated until J satisfies some convergence criterion.

Nowadays, the term 'back-propagation' is used somewhat ambiguously— it could mean the original back-propagation algorithm, or it could mean using only the first part involving the backward error propagation to compute the gradient of J, to be followed by a much more efficient descent algorithm, such as the *conjugate gradient* algorithm, resulting in much faster convergence.

The objective function convergence criterion is a subtle issue. Many MLP applications do not train until convergence to the global minimum. The reason is that data contain both signal and noise. Given enough hidden neurons, an MLP can have enough parameters to fit the training data to arbitrary accuracy, which means it is also fitting to the noise in the data, an undesirable condition known as *overfitting*. When one obtains an overfitted solution, it will not fit new data well.

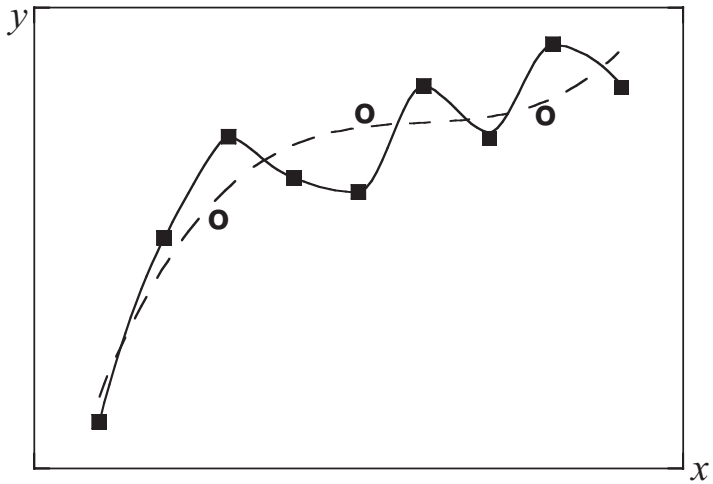
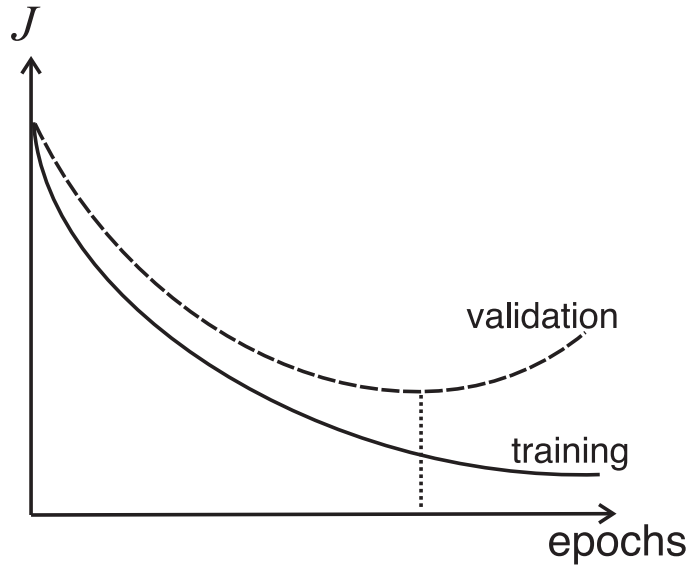


Figure : The dashed curve illustrates a good fit to noisy data (indicated by the squares), while the solid curve illustrates overfitting — where the fit is perfect on the training data (squares), but is poor on the validation data (circles). Often the NN model begins by fitting the training data as the dashed curve, but with further iterations, ends up overfitting as the solid curve.

One is interested not in using NNs to fit a given dataset to arbitrary accuracy, but in using NN to learn the underlying relationship in the given data, i.e. be able to *generalize* from a given dataset, so that the extracted relationship even fits new data not used in training the NN model.

To prevent overfitting, the dataset is often divided into two parts, one for training, the other for *validation*.
As the number of training iterations (i.e. epochs) increases, the objective function evaluated over the training data decreases.
However, the objective function evaluated over the validation data will drop but eventually increase as training epochs increase, indicating that the training dataset is already overfitted.
The minimum in the objective function evaluated over the validation data indicates when training should be stopped to avoid overfitting (as marked by the vertical dotted line).



This very common approach is called the *early stopping* (a.k.a. stopped training) method.

Similarly, J evaluated over the training data generally drops as the *number of hidden neurons* increases. Again, the objective function evaluated over a validation set will drop initially but eventually increase due to overfitting from excessive number of hidden neurons.

Hence the minimum of the objective function over the validation data may indicate how many hidden neurons to use.

A further complication is the common presence of multiple local minima in the objective function.

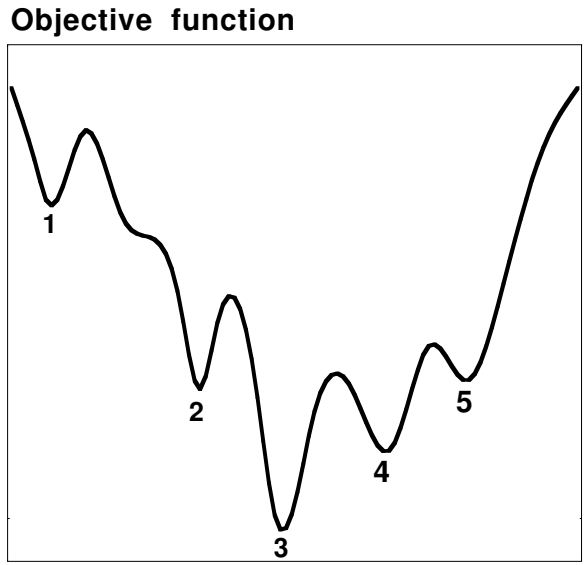


Figure : A schematic diagram illustrating the objective function surface, where depending on the starting condition, the search algorithm often gets trapped in one of the numerous deep local minima. The local

minima labelled 2, 4 and 5 are likely to be reasonable local minima, while the minimum labelled 1 is likely to be a bad one (in that the data was not well fitted at all). The minimum labelled 3 is the global minimum, which could correspond to an overfitted solution (i.e. fitted closely to the noise in the data), and may in fact be a poorer solution than the minima labelled 2, 4 and 5.

6.5 Hidden neurons [Book, Sect. 4.5]

So far, we have only described an NN with 1 hidden layer. How many layers of hidden neurons does one need? And how many neurons in each hidden layer?

Studies (Cybenko, 1989; Hornik et al., 1989; Hornik, 1991) have shown that given enough hidden neurons in an MLP with 1 hidden

layer, the network can approximate arbitrarily well any continuous function $y = f(\mathbf{x})$.

Thus even with 1 hidden layer, the MLP has become a successful universal function approximator. There is however no guidance as to how many hidden neurons are needed.

In real world applications, one may encounter some very complicated nonlinear relations where a very large number of hidden neurons are needed if a single hidden layer is used, whereas if two hidden layers are used, more modest number of hidden neurons suffices, with greater accuracy.

Hidden neurons are intermediate variables needed to carry out the computation from the inputs to the outputs, and are generally not easy to interpret.

If the number of hidden neurons are few, then they might be viewed as a low-dimensional phase space describing the state of the system.

E.g., Hsieh and Tang (1998) considered a simple MLP network for forecasting the tropical Pacific wind stress field.

The input consists of the first 11 PCs from a singular spectrum analysis of the wind stress field, plus a sine and cosine function of annual period to indicate the phase of the annual cycle, as the El Niño-Southern Oscillation (ENSO) fluctuations are often phase locked to the annual cycle.

The single hidden layer has 3 neurons, and the output layer the same 11 PC time series one month later.

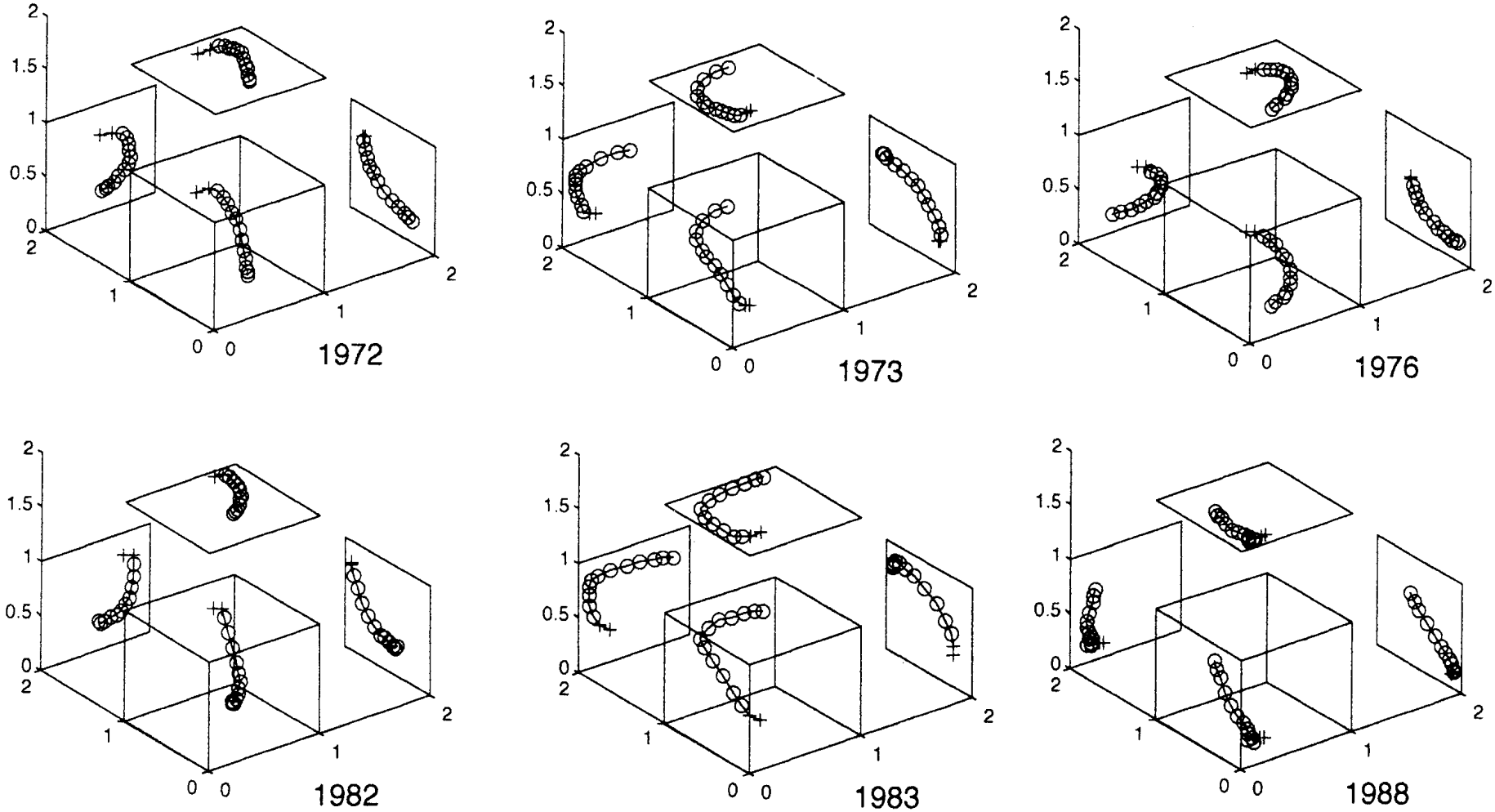


Figure : The values of the 3 hidden neurons plotted in 3-D space for the years 1972, 1973, 1976, 1982, 1983 and 1988. Projections onto 2-D planes are also shown. The small circles are for the months from January

to December, and the two "+" signs for January and February of the following year. El Niño warm episodes occurred during 1972, 1976 and 1982, while a cool episode occurred in 1988. In 1973 and 1983, the tropics returned to cooler conditions from an El Niño. Notice the similarity between the trajectories during 1972, 1976 and 1982, and during 1973, 1983 and 1988. In years with neither warm nor cold episodes, the trajectories oscillate randomly near the centre of the cube. From these trajectories, one can identify the precursor phase regions for warm episodes and cold episodes; and when the system enters one of these precursor phase regions, a forecast for either a warm or a cool episode can be issued.

We can interpret the NN as a projection from the input space onto a 3-D phase space, as spanned by the neurons in the hidden layer. The state of the system in the phase space then allows a projection onto the output space.

For most NN applications, it is not worth spending time to find interpretations for the hidden neurons, especially when there are many hidden neurons in the network.

References:

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Clarendon Pr., Oxford.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:252–257.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366.

Hsieh, W. W. and Tang, B. (1998). Applying neural network models to prediction and data analysis in meteorology and oceanography. *Bulletin of the American Meteorological Society*, 79:1855–1870.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in neural nets. *Bulletin of Mathematical Biophysics*, 5:115–137.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

Rojas, R. (1996). *Neural Networks– A Systematic Introduction*. Springer, New York.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan, New York.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D., McClelland, J., and Group, P. R., editors, *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, Cambridge, MA.

Werbos, P. J. (1974). *Beyond regression: new tools for prediction and analysis in the behavioural sciences*. Ph.D. thesis, Harvard University.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, volume 4, pages 96–104, New York.