Artificial Intelligence I

Dr Sean Holden

Notes on *machine learning using neural networks*

Did you heed the DIRE WARNING?

At the beginning of the course I suggested making sure you can answer the following two questions:

1. Let

$$f(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i^2$$

where the a_i are constants. Compute $\partial f / \partial x_j$ where $1 \leq j \leq n$?

Answer: As

$$f(x_1, \dots, x_n) = a_1 x_1^2 + \cdots + a_j x_j^2 + \cdots + a_n x_n^2$$

only one term in the sum depends on x_j, so all the other terms differentiate to give 0 and

$$\frac{\partial f}{\partial x_j} = 2 a_j x_j$$

Did you heed the DIRE WARNING?

2. Let $f(x_1, \dots, x_n)$ be a function. Now assume $x_i = g_i(y_1, \dots, y_m)$ for each x_i and some collection of functions g_i. Assuming all requirements for differentiability and so on are met, can you write down an expression for $\partial f / \partial y_j$ where $1 \leq j \leq m$?

   Answer: this is just the *chain rule* for partial differentiation

$$\frac{\partial f}{\partial y_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial y_j}$$

Supervised learning with neural networks

We now look at how an agent might *learn* to solve a general problem by seeing *examples*.

Aims:

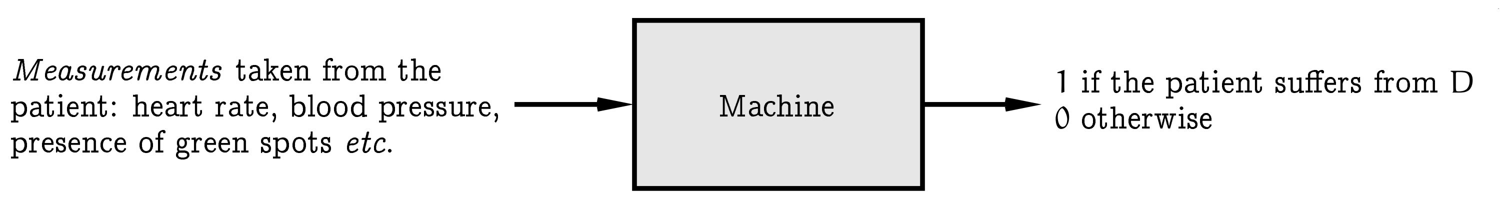
- To present an outline of *supervised learning* as part of AI.
- To introduce much of the notation and terminology used.
- To introduce the classical *perceptron*.
- To introduce *multilayer perceptrons* and the *backpropagation algorithm* for training them.

Reading: Russell and Norvig chapter 20.

An example

A common source of problems in AI is *medical diagnosis*.

Imagine that we want to automate the diagnosis of an Embarrassing Disease (call it D) by constructing a machine:



Measurements taken from the patient: heart rate, blood pressure, presence of green spots *etc.*

Machine

1 if the patient suffers from D
0 otherwise

Could we do this by *explicitly writing a program* that examines the measurements and outputs a diagnosis?

Experience suggests that this is unlikely.

An example, continued...

An alternative approach: each collection of measurements can be written as a vector,

$$\mathbf{x}^T = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}$$

where,

$$\begin{aligned} x_1 &= \text{heart rate} \\ x_2 &= \text{blood pressure} \\ x_3 &= \text{1 if the patient has green spots} \\ & \text{0 otherwise} \\ &\vdots \\ & \text{and so on} \end{aligned}$$

(*Note*: it's a common convention that vectors are *column vectors* by default. This is why the above is written as a *transpose*.)

An example, continued...

A vector of this kind contains all the measurements for a single patient and is called a *feature vector* or *instance*.

The measurements are *attributes* or *features*.

Attributes or features generally appear as one of three basic types:

- *Continuous*: $x_i \in [x_{\text{min}}, x_{\text{max}}]$ where $x_{\text{min}}, x_{\text{max}} \in \mathbb{R}$.
- *Binary*: $x_i \in \{0, 1\}$ or $x_i \in \{-1, +1\}$.
- *Discrete*: x_i can take one of a finite number of values, say $x_i \in \{X_1, \ldots, X_p\}$.

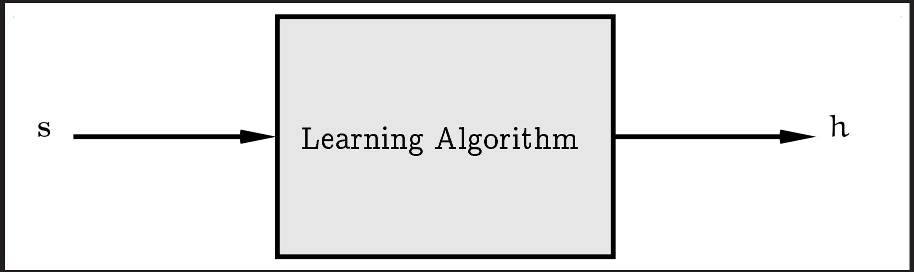
An example, continued...

Now imagine that we have a large collection of patient histories (m in total) and for each of these we know whether or not the patient suffered from D.

- The ith patient history gives us an instance $\mathbf{x}_i$.
- This can be paired with a single bit—0 or 1—denoting whether or not the ith patient suffers from D. The resulting pair is called an *example* or a *labelled example*.
- Collecting all the examples together we obtain a *training sequence*
$$\mathbf{s} = ((\mathbf{x}_1, 0), (\mathbf{x}_2, 0), \ldots, (\mathbf{x}_m, 0))$$

An example, continued...

In supervised machine learning we aim to design a *learning algorithm* which takes s and produces a *hypothesis* h.



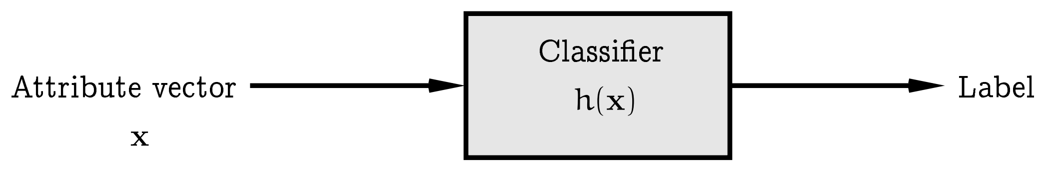
Intuitively, a hypothesis is something that lets us diagnose *new* patients.

This is *IMPORTANT*: we want to diagnose patients that *the system has never seen*.

The ability to do this successfully is called *generalisation*.

An example, continued...

In fact, a hypothesis is just a *function* that maps *instances* to *labels*.



As h is a *function* it assigns a label to *any* $\mathbf{x}$ and *not just the ones that were in the training sequence*.

What we mean by a *label* here depends on whether we're doing *classification* or *regression*.

Supervised learning: classification

In *classification* we're assigning $\mathbf{x}$ to one of a set $\{\omega_1, \ldots, \omega_c\}$ of c *classes*.

For example, if $\mathbf{x}$ contains measurements taken from a patient then there might be three classes:

$$\begin{aligned}\omega_1 &= \text{patient has disease} \\ \omega_2 &= \text{patient doesn't have disease} \\ \omega_3 &= \text{don't ask me buddy, I'm just a computer!}\end{aligned}$$

The *binary* case above also fits into this framework, and we'll often specialise to the case of two classes, denoted C_1 and C_2.

Supervised learning: regression

In *regression* we're assigning $\mathbf{x}$ to a *real number* $h(\mathbf{x}) \in \mathbb{R}$.

For example, if $\mathbf{x}$ contains measurements taken regarding today's weather then we might have

$$h(\mathbf{x}) = \text{estimate of amount of rainfall expected tomorrow}$$

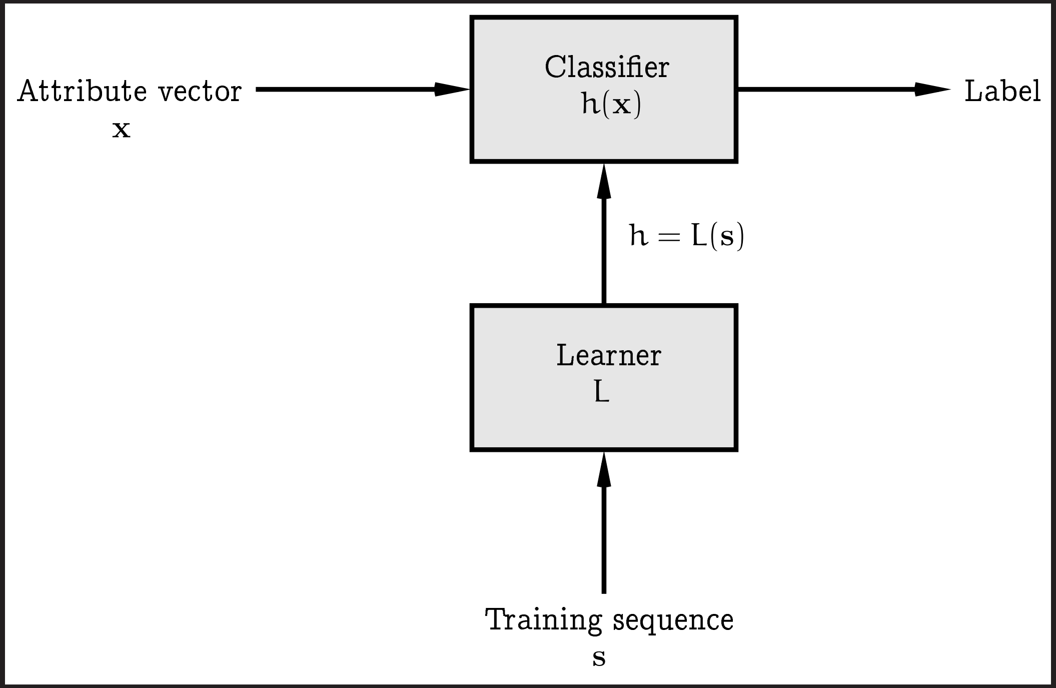
For the two-class classification problem we will also refer to a situation somewhat between the two, where

$$h(\mathbf{x}) = \Pr(\mathbf{x} \text{ is in } C_1)$$

and so we would typically assign $\mathbf{x}$ to class C_1 if $h(\mathbf{x}) > 1/2$.

Summary

We don't want to design h explicitly.



So we use a *learner* L to infer it on the basis of a sequence $\mathbf{s}$ of *training examples*.

Neural networks

There is generally a set $\mathcal{H}$ of hypotheses from which L is allowed to select h

$$\mathsf{L}(\mathbf{s}) = \mathsf{h} \in \mathcal{H}$$

$\mathcal{H}$ is called the *hypothesis space*.

The learner can output a hypothesis explicitly or—as in the case of a *neural network*—it can output a vector

$$\mathbf{w}^\top = \begin{pmatrix} w_1 & w_2 & \cdots & w_W \end{pmatrix}$$

of *weights* which in turn specify h

$$\mathsf{h}(\mathbf{x}) = \mathsf{f}(\mathbf{w}; \mathbf{x})$$

where $\mathbf{w} = \mathsf{L}(\mathbf{s})$.

Types of learning

The form of machine learning described is called *supervised learning*.

This introduction will concentrate on this kind of learning. In particular, the literature also discusses:

1. *Unsupervised learning*.
2. Learning using *membership queries* and *equivalence queries*.
3. *Reinforcement learning*.

Some of this further material will be covered in AI 2.

Some further examples

- *Speech recognition*.
- Deciding *whether or not to give credit*.
- Detecting *credit card fraud*.
- Deciding whether to *buy or sell a stock option*.
- Deciding whether a *tumour is benign*.
- *Data mining*: extracting interesting but hidden knowledge from existing, large databases. For example, databases containing *financial transactions* or *loan applications*.
- Deciding whether *driving conditions are dangerous*.
- *Automatic driving*. (See Pomerleau, 1989, in which a car is driven for 90 miles at 70 miles per hour, on a public road with other cars present, but with no assistance from humans.)

This is very similar to curve fitting

This process is in fact very similar to *curve fitting*.

Think of the process as follows:

- Nature picks an $h' \in \mathcal{H}$ but doesn't reveal it to us.
- Nature then shows us a training sequence $\mathbf{s}$ where each $\mathbf{x}_i$ is labelled as $h'(\mathbf{x}_i) + \epsilon_i$ where ϵ_i is noise of some kind.

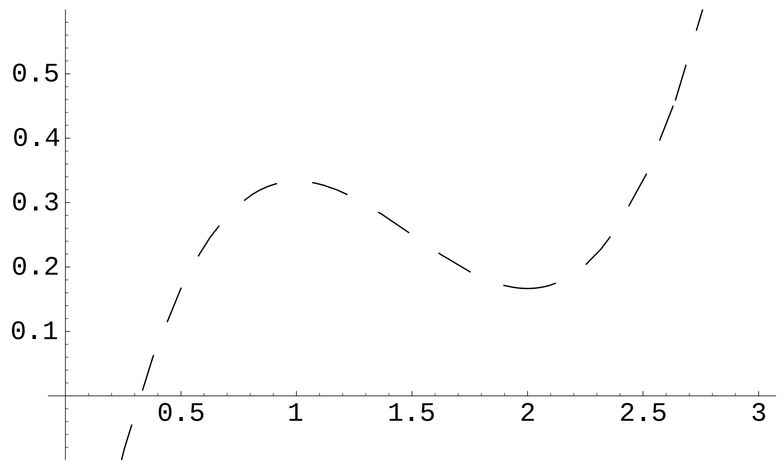
Our job is to try to infer what h' is *on the basis of* $\mathbf{s}$ *only*.

This is easy to visualise in one dimension: *it's just fitting a curve to some points*.

Curve fitting

Example: if $\mathcal{H}$ is the set of all polynomials of degree 3 then nature might pick

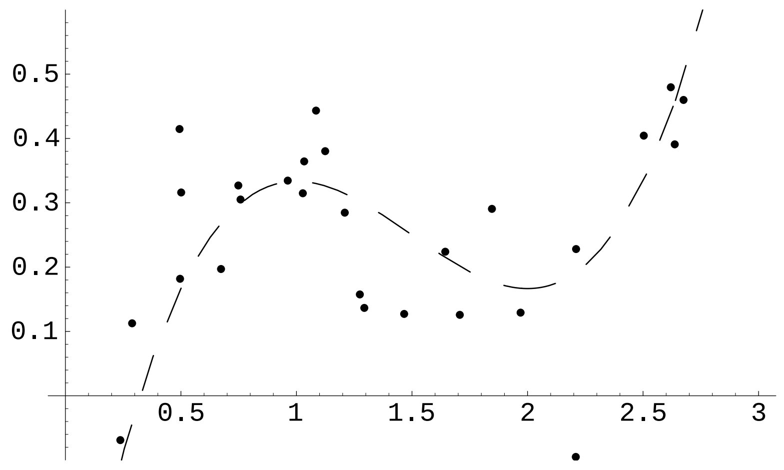
$$h'(x) = \frac{1}{3}x^3 - \frac{3}{2}x^2 + 2x - \frac{1}{2}$$



The line is dashed to emphasise the fact that *we don't get to see it*.

Curve fitting

We can now use h' to obtain a training sequence s in the manner suggested..



Here we have,

$$s^T = ((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m))$$

where each x_i and y_i is a real number.

Curve fitting

We'll use a *learning algorithm* L that operates in a reasonable-looking way: it picks an $h \in \mathcal{H}$ minimising the following quantity,

$$E = \sum_{i=1}^{m} (h(x_i) - y_i)^2$$

In other words

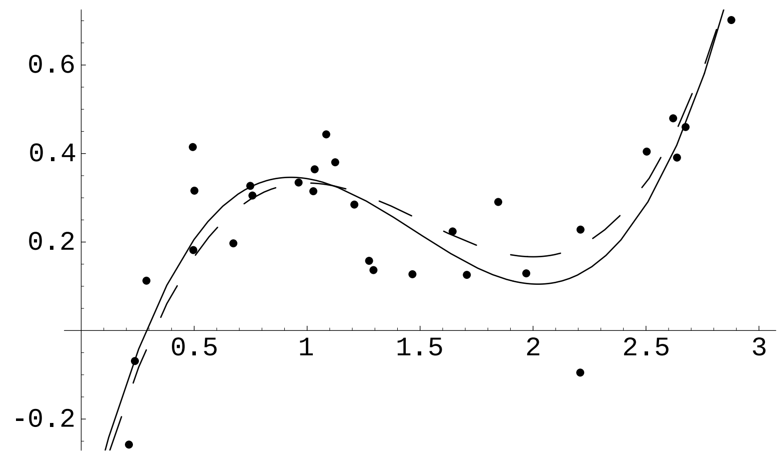
$$h = L(s) = \operatorname*{argmin}_{h \in \mathcal{H}} \sum_{i=1}^{m} (h(x_i) - y_i)^2$$

Why is this sensible?

1. Each term in the sum is 0 if $h(x_i)$ is *exactly* y_i.
2. Each term *increases* as the difference between $h(x_i)$ and y_i increases.
3. We add the terms for all examples.

Curve fitting

If we pick h using this method then we get:



The chosen h is close to the target h', even though it was chosen *using only a small number of noisy examples*.

It is not quite identical to the target concept.

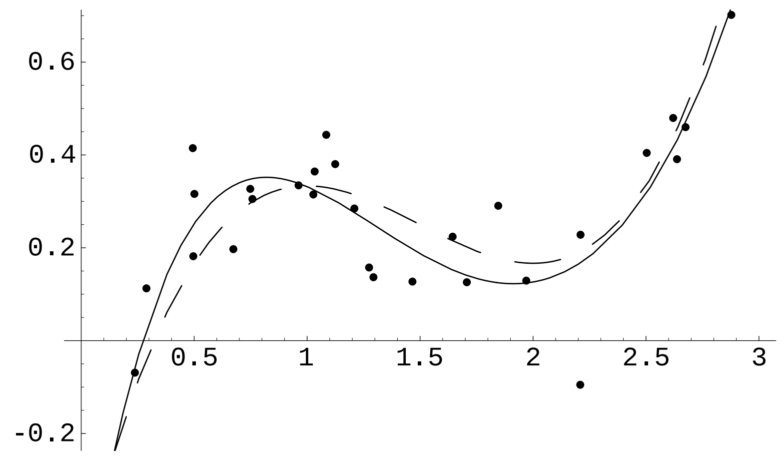
However if we were given a new point x' and asked to guess the value $h'(x')$ then guessing $h(x')$ might be expected to do quite well.

Curve fitting

Problem: we don't know *what $\mathcal{H}$ nature is using*. What if the one we choose doesn't match? We can make *our $\mathcal{H}$* 'bigger' by defining it as

$$\mathcal{H} = \{\mathrm{h} : \mathrm{h} \text{ is a polynomial of degree at most } 5\}$$

If we use the same learning algorithm then we get:



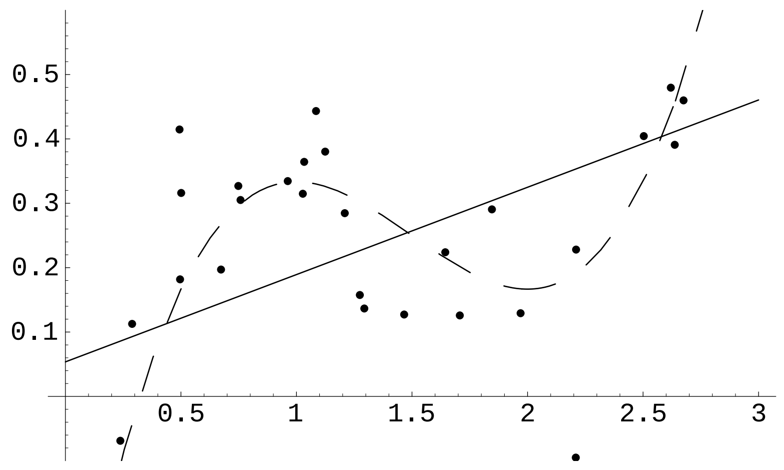
The result in this case is similar to the previous one: h is again quite close to h', but not quite identical.

Curve fitting

So what's the problem? Repeating the process with,

$$\mathcal{H} = \{h : h \text{ is a polynomial of degree at most } 1\}$$

gives the following:



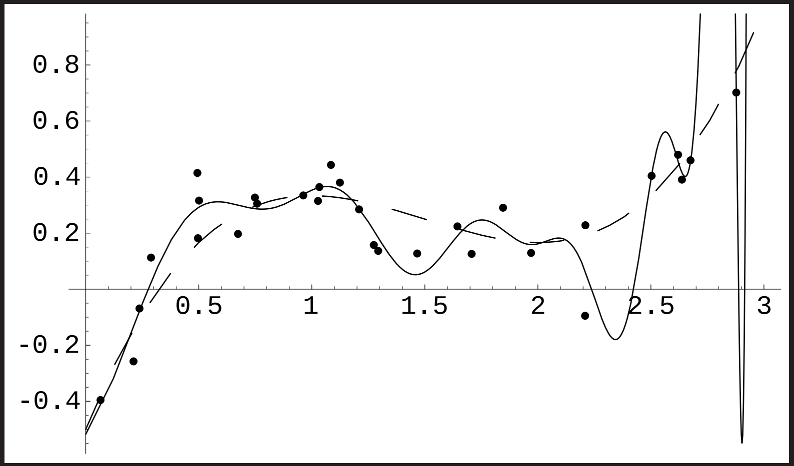
In effect, we have made *our* $\mathcal{H}$ too 'small'. It does not in fact contain any hypothesis similar to h'.

Curve fitting

So we have to make $\mathcal{H}$ huge, right? WRONG!!! With

$$\mathcal{H} = \{\mathrm{h} : \mathrm{h} \text{ is a polynomial of degree at most } 25\}$$

we get:



BEWARE!!! This is known as *overfitting*.

Curve fitting

An experiment to gain some further insight: using

$$h'(x) = \frac{1}{10}x^{10} - \frac{1}{12}x^{8} + \frac{1}{15}x^{6} + \frac{1}{3}x^{3} - \frac{3}{2}x^{2} + 2x - \frac{1}{2}.$$

as the unknown underlying function.

We can look at how *the degree of the polynomial the training algorithm can output affects the generalisation ability of the resulting* h.

We use the same training algorithm, and we train using

$$\mathcal{H} = \{h : h \text{ is a polynomial of degree at most } d\}$$

for values of d ranging from 1 to 30

Curve fitting

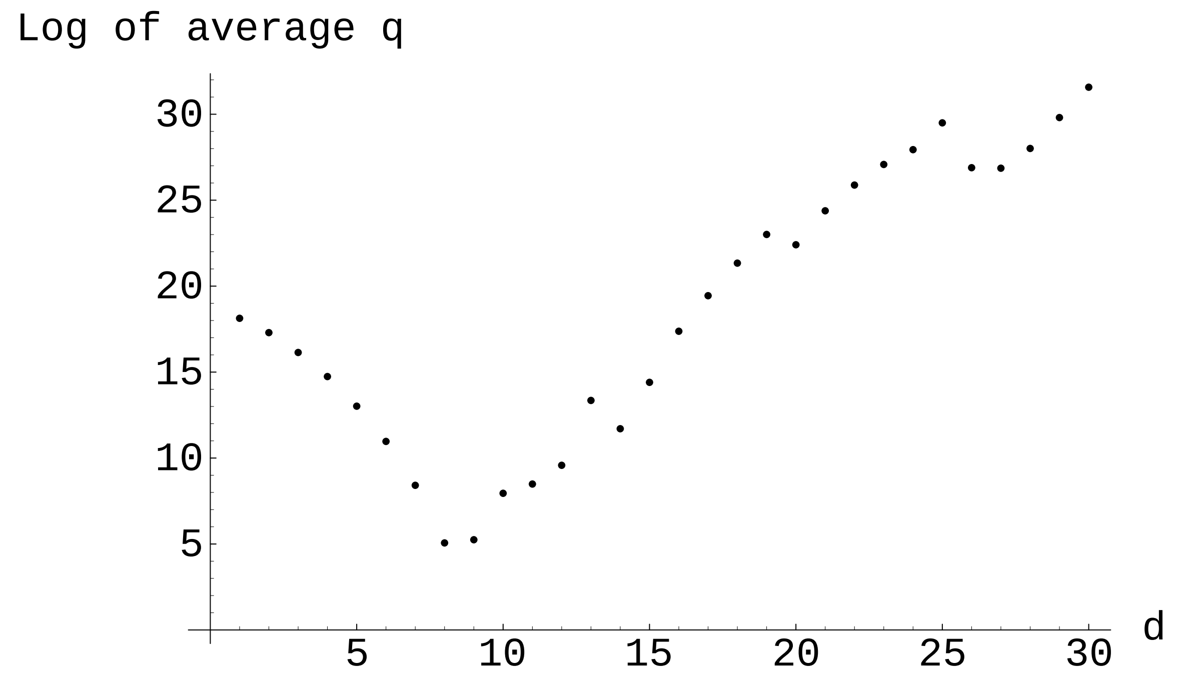
- Each time we obtain an h of a given degree—call it h_d—we assess its quality *using a further* 100 *inputs* $\mathbf{x}'_i$ *generated at random* and calculating

$$q(d) = \frac{1}{100} \sum_{i=1}^{100} (h'(\mathbf{x}'_i) - h_d(\mathbf{x}'_i))^2$$

- As the values $q(d)$ are found using inputs that are not necessarily included in the training sequence *they measure generalisation*.
- To smooth out the effects of the random selection of examples we repeat this process 100 times and average the values $q(d)$.

Curve fitting

Here is the result:



Clearly: we need to choose $\mathcal{H}$ sensibly if we want to obtain *good generalisation performance*.

The perceptron

The example just given illustrates much of what we want to do. However in practice we deal with *more than a single dimension*.

The simplest form of hypothesis used is the *linear discriminant*, also known as the *perceptron*. Here

$$h(\mathbf{w};\mathbf{x}) = \sigma\left(w_0 + \sum_{i=1}^{m} w_i x_i\right) = \sigma\left(w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n\right)$$

So: we have a *linear function* modified by the *activation function* σ.

The perceptron's influence continues to be felt in the recent and ongoing development of *support vector machines*.

The perceptron activation function I

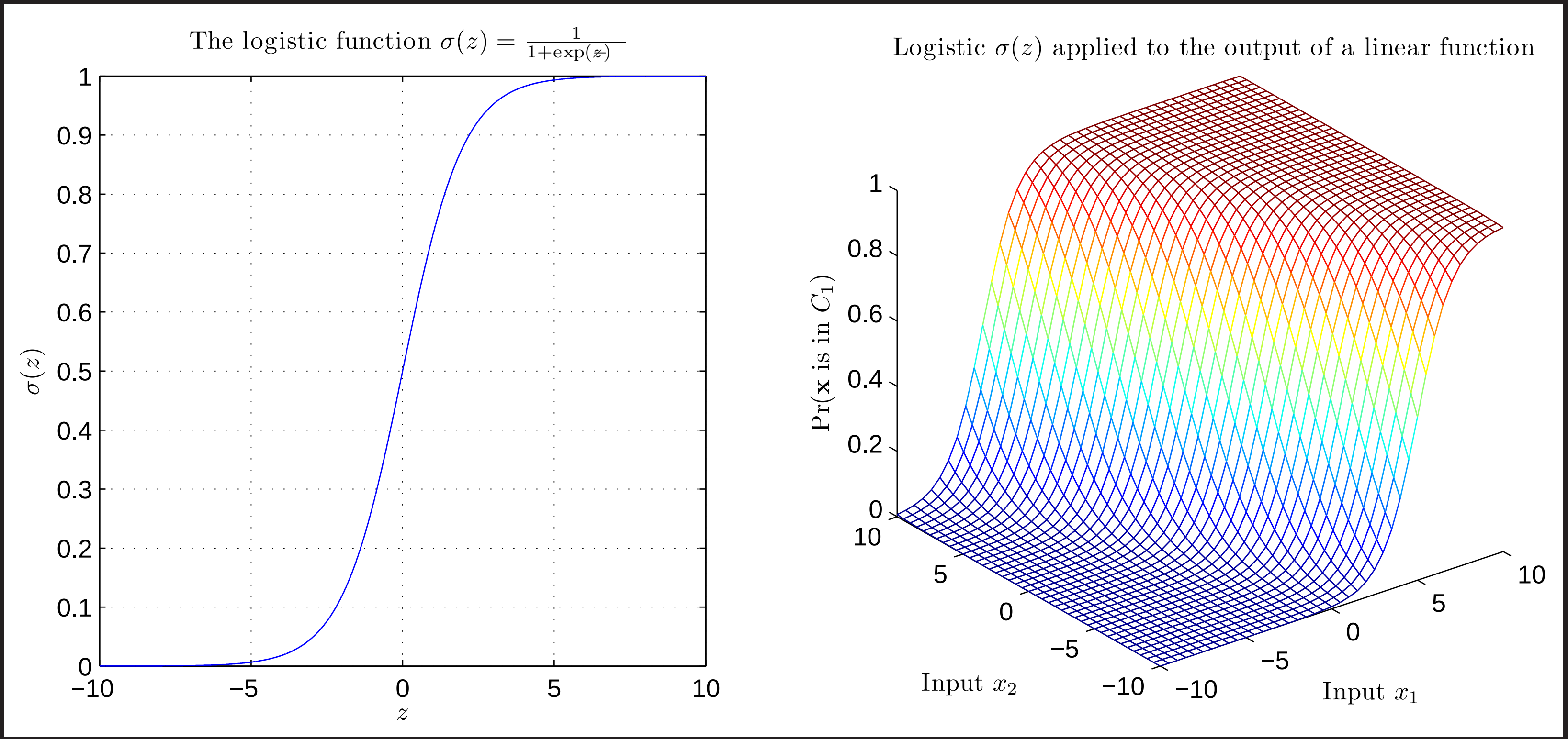
There are three standard forms for the activation function:

1. *Linear*: for *regression problems* we often use
$$\sigma(z) = z$$
2. *Step*: for *two-class classification problems* we often use
$$\sigma(z) = \begin{cases} C_1 & \text{if } z > 0 \\ C_2 & \text{otherwise.} \end{cases}$$
3. *Sigmoid/Logistic*: for *probabilistic classification* we often use
$$\Pr(\mathbf{x} \text{ is in } C_1) = \sigma(z) = \frac{1}{1 + \exp(-z)}.$$

The *step function* is important but the algorithms involved are somewhat different to those we'll be seeing. We won't consider it further.

The *sigmoid/logistic function* plays a major role in what follows.

The sigmoid/logistic function



Gradient descent

A method for *training a basic perceptron* works as follows. Assume we're dealing with a *regression problem* and using $\sigma(z) = z$.

We define a measure of *error* for a given collection of weights. For example

$$E(\mathbf{w}) = \sum_{i=1}^{m} (y_i - h(\mathbf{w}; \mathbf{x}_i))^2$$

Modifying our notation slightly so that

$$\begin{aligned} \mathbf{x}^T &= \begin{pmatrix} 1 & x_1 & x_2 & \cdots & x_n \end{pmatrix} \\ \mathbf{w}^T &= \begin{pmatrix} w_0 & w_1 & w_2 & \cdots & w_n \end{pmatrix} \end{aligned}$$

lets us write

$$E(\mathbf{w}) = \sum_{i=1}^{m} (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

Gradient descent

We want to *minimise* $E(\mathbf{w})$.

One way to approach this is to start with a random $\mathbf{w}_0$ and update it as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

where

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \left(\frac{\partial E(\mathbf{w})}{\partial w_0} \;\; \frac{\partial E(\mathbf{w})}{\partial w_1} \;\; \cdots \;\; \frac{\partial E(\mathbf{w})}{\partial w_n} \right)^T$$

and η is some small positive number.

The vector

$$-\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$$

tells us the *direction of the steepest decrease in* $E(\mathbf{w})$.

Gradient descent

With

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2$$

we have

$$\begin{aligned}
\frac{\partial E(\mathbf{w})}{\partial w_j} &= \frac{\partial}{\partial w_j}\left(\sum_{i=1}^{m}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2\right) \\
&= \sum_{i=1}^{m}\left(\frac{\partial}{\partial w_j}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2\right) \\
&= \sum_{i=1}^{m}\left(2(y_i - \mathbf{w}^\top \mathbf{x}_i)\frac{\partial}{\partial w_j}(-\mathbf{w}^\top \mathbf{x}_i)\right) \\
&= -2\mathbf{x}_i^{(j)}\sum_{i=1}^{m}(y_i - \mathbf{w}^\top \mathbf{x}_i)
\end{aligned}$$

where $\mathbf{x}_i^{(j)}$ is the jth element of $\mathbf{x}_i$.

Gradient descent

The method therefore gives the algorithm

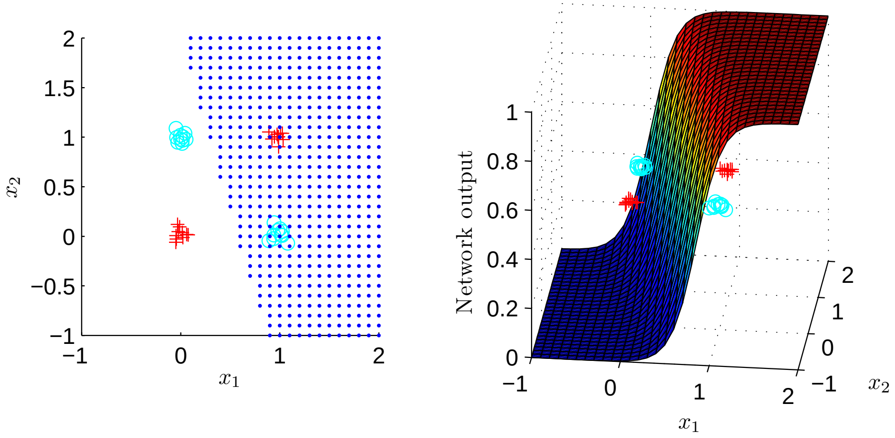
$$\mathbf{w}_{t+1} = \mathbf{w}_t + 2\eta \sum_{i=1}^{m} \left(y_i - \mathbf{w}_t^T \mathbf{x}_i\right) \mathbf{x}_i$$

Some things to note:

- In this case $E(\mathbf{w})$ is *parabolic* and has a *unique global minimum* and *no local minima* so this works well.
- *Gradient descent* in some form is a very common approach to this kind of problem.
- We can perform a similar calculation for *other activation functions* and for *other definitions for* $E(\mathbf{w})$.
- Such calculations lead to *different algorithms*.

Perceptrons aren't very powerful: the parity problem

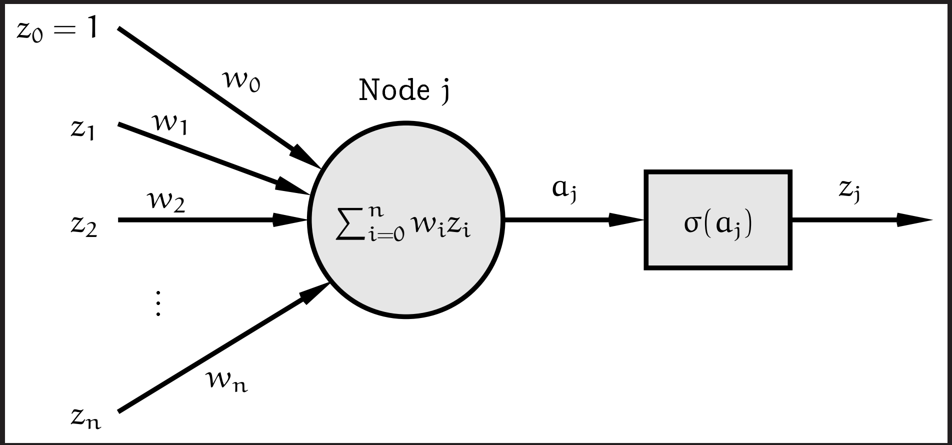
There are many problems a perceptron can't solve.



We need a network that computes *more interesting functions*.

The multilayer perceptron

Each *node* in the network is itself a perceptron:



- *Weights* w_i connect nodes together.
- a_j is the weighted sum or *activation* for node j.
- σ is the *activation function*.
- The *output* is $z_j = \sigma(a_j)$.

The multilayer perceptron

Reminder:

We'll continue to use the notation

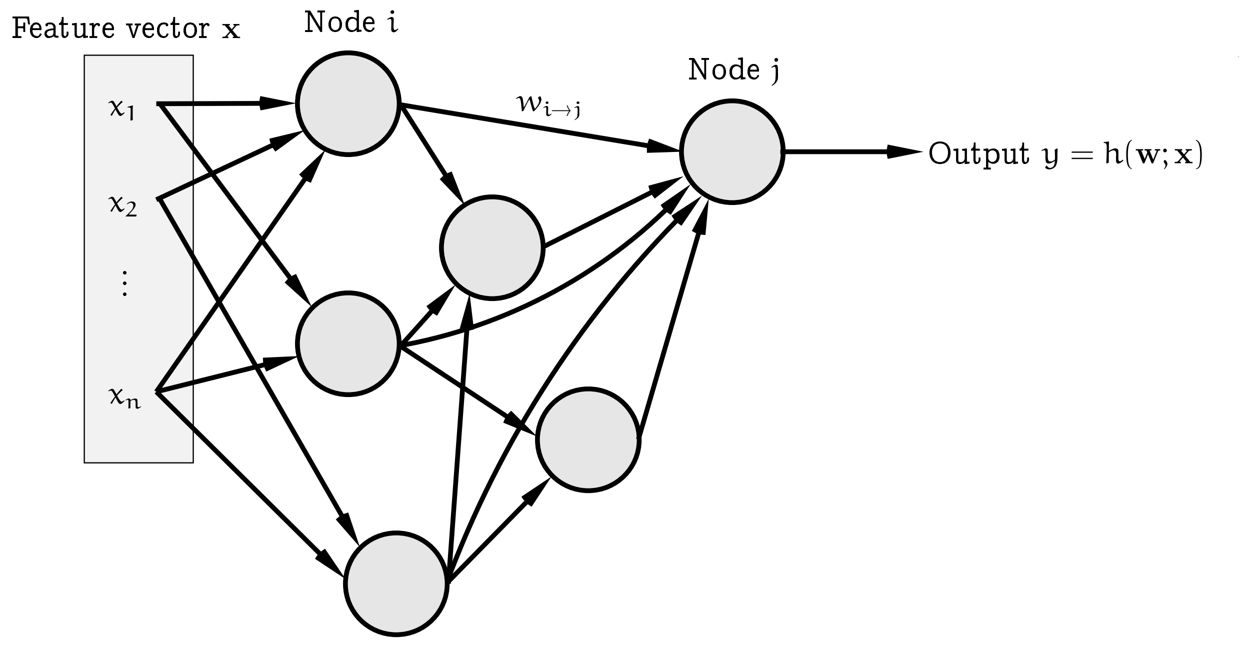
$$\mathbf{z}^{\mathsf{T}} = \begin{pmatrix} 1 & z_1 & z_2 & \cdots & z_n \end{pmatrix}$$
$$\mathbf{w}^{\mathsf{T}} = \begin{pmatrix} w_0 & w_1 & w_2 & \cdots & w_n \end{pmatrix}$$

So that

$$\begin{aligned} \sum_{i=0}^{n} w_i z_i &= w_0 + \sum_{i=1}^{n} w_i z_i \\ &= \mathbf{w}^{\mathsf{T}} \mathbf{z} \end{aligned}$$

The multilayer perceptron

In the general case we have a *completely unrestricted feedforward structure*:



Each node is a perceptron. *No specific layering* is assumed.

$w_{i \to j}$ connects node i to node j. w_0 for node j is denoted $w_{0 \to j}$.

Backpropagation

As usual we have:

- Instances $\mathbf{x}^T = (x_1, \ldots, x_n)$.
- A training sequence $\mathbf{s} = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$.

We also define a measure of training error

$$E(\mathbf{w}) = \text{measure of the error of the network on } \mathbf{s}$$

where $\mathbf{w}$ is the vector of *all the weights in the network*.

Our aim is to find a set of weights that *minimises* $E(\mathbf{w})$ using *gradient descent*.

Backpropagation: the general case

The *central task* is therefore to calculate

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$$

To do that we need to calculate the individual quantities

$$\frac{\partial E(\mathbf{w})}{\partial w_{i \rightarrow j}}$$

for *every weight* $w_{i \rightarrow j}$ *in the network*.

Often $E(\mathbf{w})$ is the sum of separate components, one for each example in $\mathbf{s}$

$$E(\mathbf{w}) = \sum_{p=1}^{m} E_p(\mathbf{w})$$

in which case

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \sum_{p=1}^{m} \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}$$

We can therefore consider examples individually.

Backpropagation: the general case

Place example p at the input and calculate a_j and z_j for *all nodes* including the output y. This is *forward propagation*.

We have

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to j}} = \frac{\partial E_p(\mathbf{w})}{\partial a_j} \frac{\partial a_j}{\partial w_{i\to j}}$$

where $a_j = \sum_k w_{k\to j} z_k$.

Here the sum is over *all the nodes connected to node* j. As

$$\frac{\partial a_j}{\partial w_{i\to j}} = \frac{\partial}{\partial w_{i\to j}} \left(\sum_k w_{k\to j} z_k \right) = z_i$$

we can write

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to j}} = \delta_j z_i$$

where we've defined

$$\delta_j = \frac{\partial E_p(\mathbf{w})}{\partial a_j}$$

Backpropagation: the general case

So we now need to calculate the values for δ_j...

When j is the *output node*—that is, the one producing the output $y = h(\mathbf{w}; \mathbf{x}_p)$ of the network—this is easy as $z_j = y$ and

$$\begin{aligned}
\delta_j &= \frac{\partial E_p(\mathbf{w})}{\partial a_j} \\
&= \frac{\partial E_p(\mathbf{w})}{\partial y} \frac{\partial y}{\partial a_j} \\
&= \frac{\partial E_p(\mathbf{w})}{\partial y} \sigma'(a_j)
\end{aligned}$$

using the fact that $y = \sigma(a_j)$.

Backpropagation: the general case

The first term is in general easy to calculate for a given E as the error is generally just a measure of the distance between y and the label in the training sequence.

Example: when

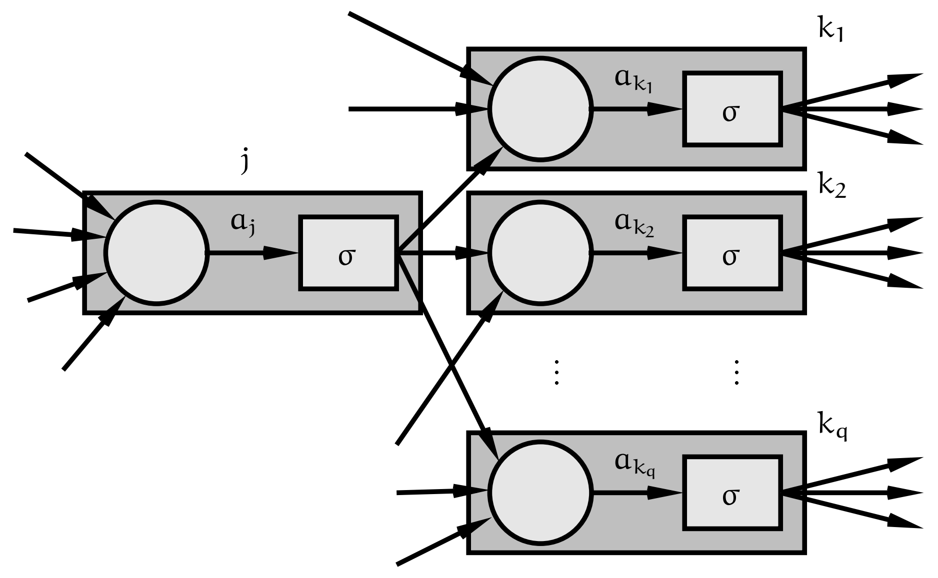
$$E_p(\mathbf{w}) = (y - y_p)^2$$

we have

$$\begin{aligned}\frac{\partial E_p(\mathbf{w})}{\partial y} &= 2(y - y_p) \\ &= 2(f(\mathbf{w}; \mathbf{x}_p) - y_p)\end{aligned}$$

Backpropagation: the general case

When j is *not an output node* we need something different:

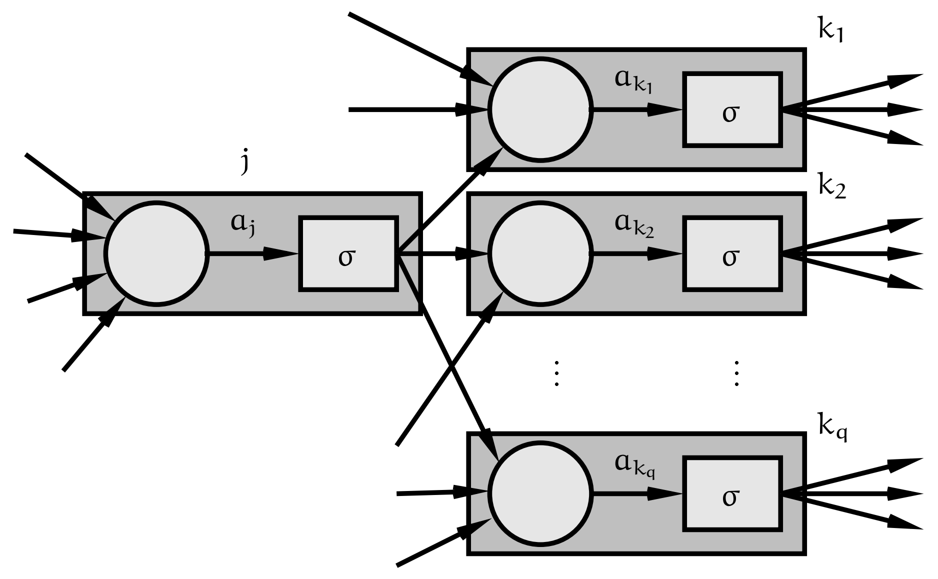


We're interested in

$$\delta_j = \frac{\partial E_p(\mathbf{w})}{\partial a_j}$$

Altering a_j can affect *several other nodes* $k_1, k_2, \ldots, k_q$ *each of which can in turn affect* $E_p(\mathbf{w})$.

Backpropagation: the general case

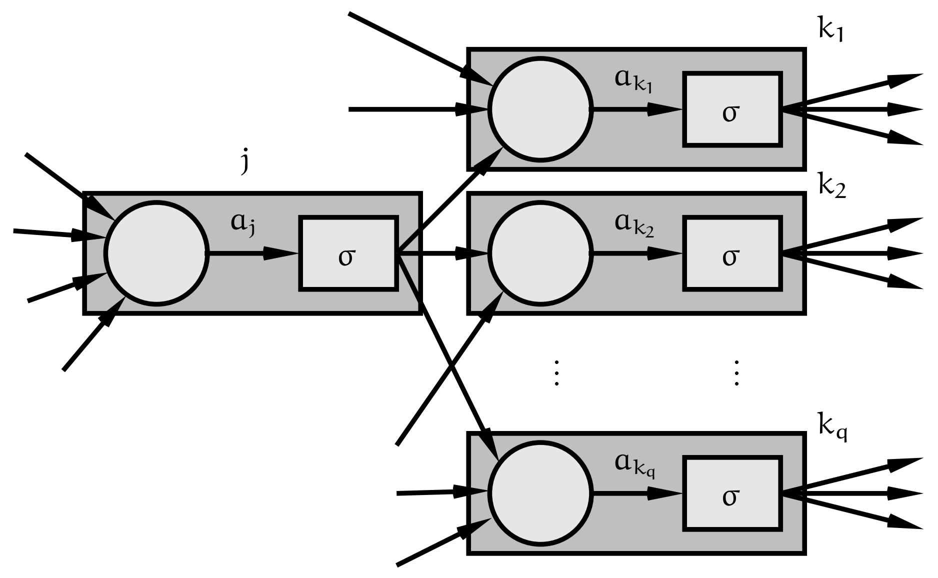


We have

$$\delta_j = \frac{\partial E_p(\mathbf{w})}{\partial a_j} = \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \frac{\partial E_p(\mathbf{w})}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k \frac{\partial a_k}{\partial a_j}$$

where $k_1, k_2, \ldots, k_q$ are the nodes to which node j sends a connection.

Backpropagation: the general case

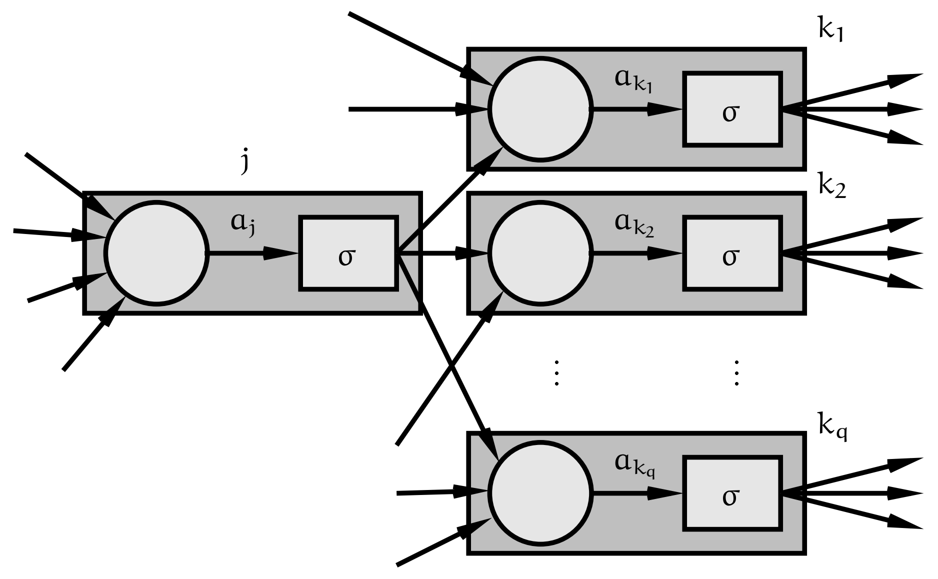


Because we know how to compute δ_j *for the output node* we can *work backwards* computing further δ values.

We will *always know all the values* δ_k *for nodes ahead of where we are*.

Hence the term *backpropagation*.

Backpropagation: the general case



$$\frac{\partial a_k}{\partial a_j} = \frac{\partial}{\partial a_j}\left(\sum_i w_{i\to k}\sigma(a_i)\right) = w_{j\to k}\sigma'(a_j)$$

and

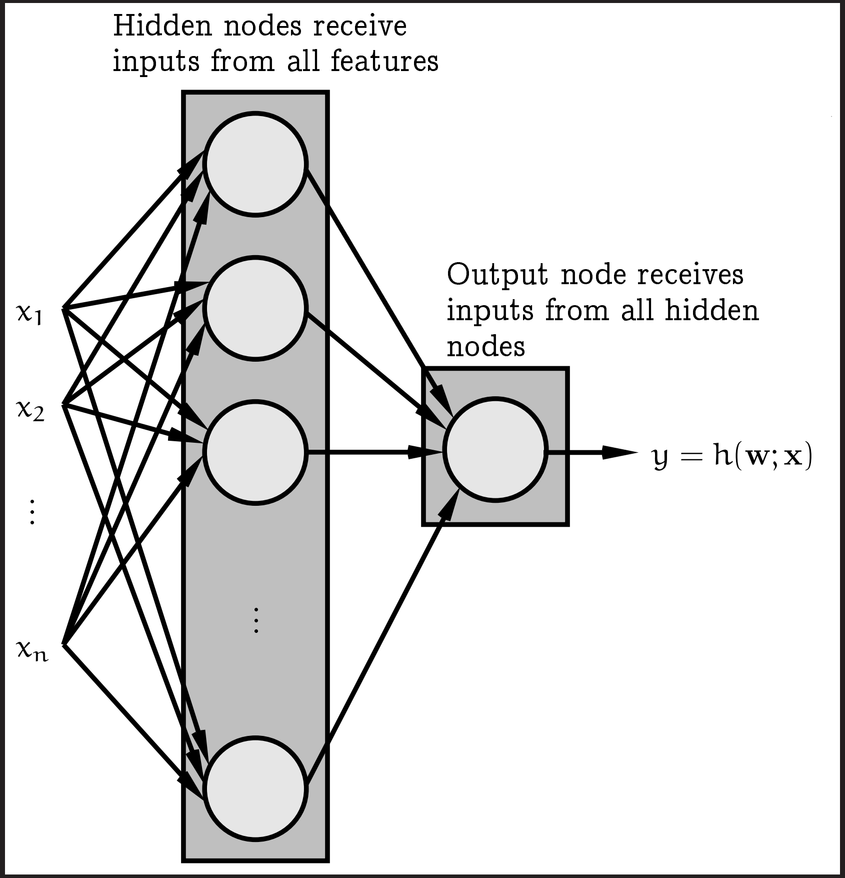
$$\delta_j = \sum_{k\in\{k_1,k_2,\ldots,k_q\}} \delta_k w_{j\to k}\sigma'(a_j) = \sigma'(a_j)\sum_{k\in\{k_1,k_2,\ldots,k_q\}} \delta_k w_{j\to k}$$

Backpropagation: the general case

Summary: to calculate $\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}$ for the pth pattern:

1. *Forward propagation*: apply $\mathbf{x}_p$ and calculate outputs *etc* for *all the nodes in the network*.
2. *Backpropagation 1*: for the *output* node
$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = z_i \delta_j = z_i \sigma'(a_j) \frac{\partial E_p(\mathbf{w})}{\partial y}$$
where $y = h(\mathbf{w}; \mathbf{x}_p)$.
3. *Backpropagation 2*: For other nodes
$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = z_i \sigma'(a_j) \sum_k \delta_k w_{j \to k}$$
where the δ_k were calculated at an earlier step.

Backpropagation: a specific example



For the output: $\sigma(a) = a$. For the hidden nodes $\sigma(a) = \frac{1}{1+\exp(-a)}$.

Backpropagation: a specific example

For the output: $\sigma(a) = a$ so $\sigma'(a) = 1$.

For the hidden nodes:

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

so

$$\sigma'(a) = \sigma(a)\,[1 - \sigma(a)]$$

We'll continue using the same definition for the error

$$E(\mathbf{w}) = \sum_{p=1}^{m} (y_p - h(\mathbf{w}; \mathbf{x}_p))^2$$

$$E_p(\mathbf{w}) = (y_p - h(\mathbf{w}; \mathbf{x}_p))^2$$

Backpropagation: a specific example

For the output: the equation is

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to\text{output}}} = z_i \delta_{\text{output}} = z_i \sigma'(a_{\text{output}}) \frac{\partial E_p(\mathbf{w})}{\partial y}$$

where $y = h(\mathbf{w}; \mathbf{x}_p)$. So as

$$\begin{aligned}\frac{\partial E_p(\mathbf{w})}{\partial y} &= \frac{\partial}{\partial y}\left((y_p - y)^2\right)\\ &= 2(y - y_p)\\ &= 2\left[h(\mathbf{w}; \mathbf{x}_p) - y_p\right]\end{aligned}$$

and $g'(a) = 1$ so

$$\delta_{\text{output}} = 2\left[h(\mathbf{w}; \mathbf{x}_p) - y_p\right]$$

and

$$\boxed{\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to\text{output}}} = 2z_i(h(\mathbf{w}; \mathbf{x}_p) - y_p)}$$

Backpropagation: a specific example

For the hidden nodes: the equation is

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to j}} = z_i \sigma'(a_j) \sum_k \delta_k w_{j\to k}$$

However *there is only one output* so

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to j}} = z_i \sigma(a_j) \left[1 - \sigma(a_j)\right] \delta_{\text{output}} w_{j\to\text{output}}$$

and we know that

$$\delta_{\text{output}} = 2\left[h(\mathbf{w};\mathbf{x}_p) - y_p\right]$$

so

$$\boxed{\begin{aligned}\frac{\partial E_p(\mathbf{w})}{\partial w_{i\to j}} &= 2z_i \sigma(a_j) \left[1 - \sigma(a_j)\right] \left[h(\mathbf{w};\mathbf{x}_p) - y_p\right] w_{j\to\text{output}} \\ &= 2x_i z_j (1 - z_j) \left[h(\mathbf{w};\mathbf{x}_p) - y_p\right] w_{j\to\text{output}}\end{aligned}}$$

Putting it all together

We can then use the derivatives in one of two basic ways:

Batch: (as described previously)

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \sum_{p=1}^{m} \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}$$

then

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

Sequential: using just one pattern at once

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left. \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

selecting patterns *in sequence or at random*.

Example: the parity problem revisited

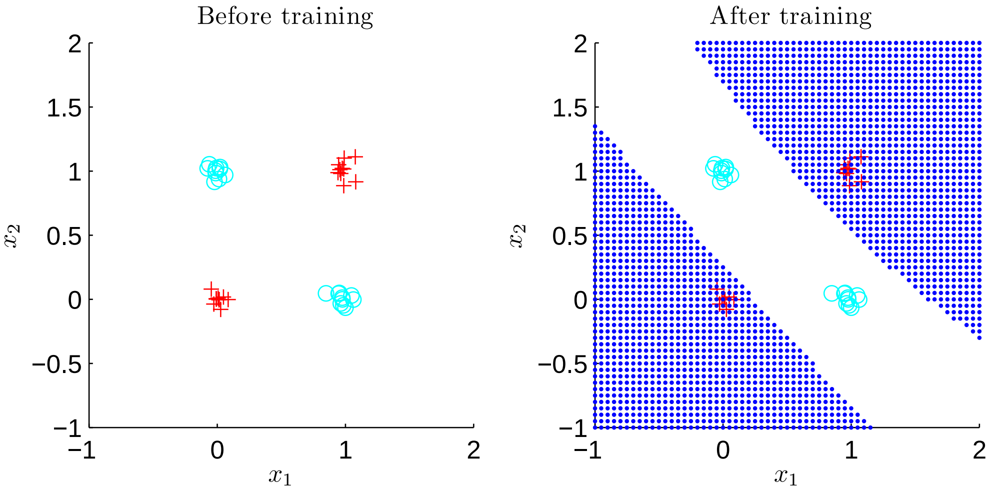
As an example we show the result of training a network with:

- Two inputs.
- One output.
- One hidden layer containing 5 units.
- $\eta = 0.01$.
- All other details as above.

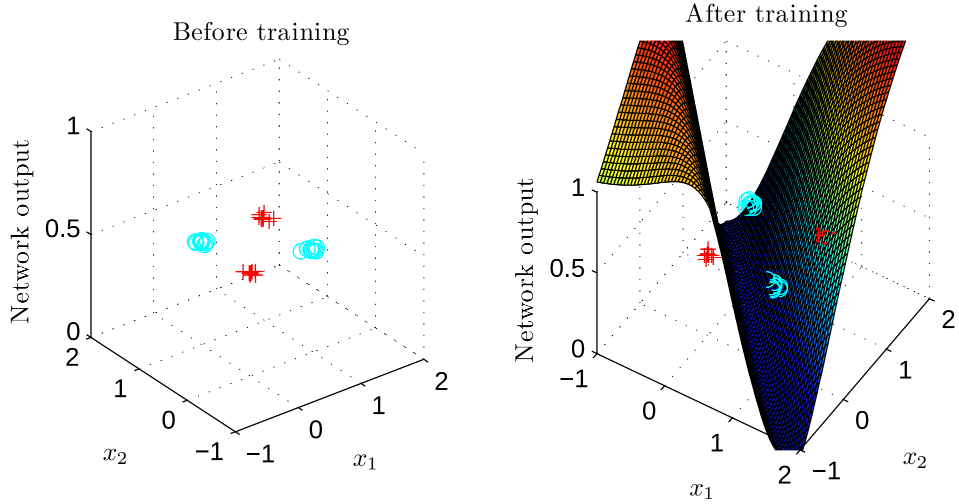
The problem is the parity problem. There are 40 noisy examples.

The sequential approach is used, with 1000 repetitions through the entire training sequence.

Example: the parity problem revisited



Example: the parity problem revisited



Example: the parity problem revisited