Supervised Learning with Neural Networks

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

Aims:

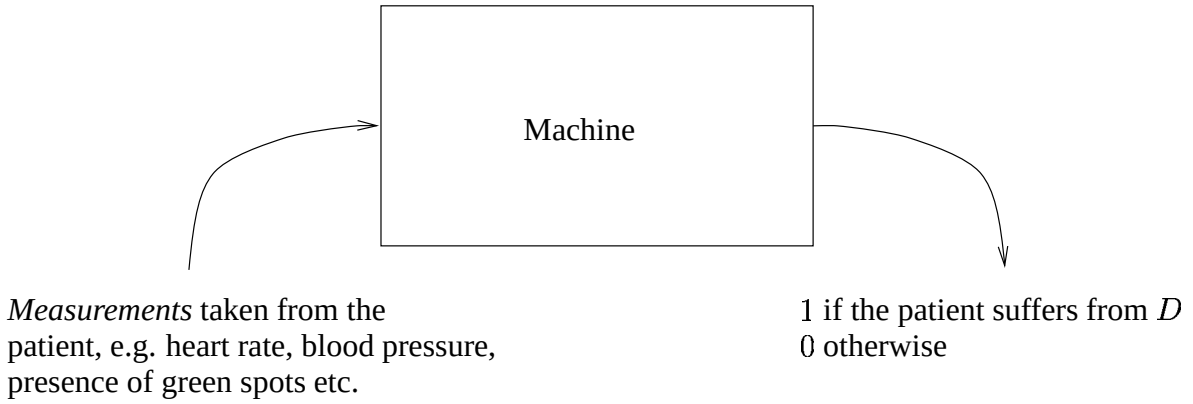
- to present an outline of *supervised learning* as part of AI;
- to introduce much of the notation and terminology used;
- to introduce the classical *perceptron*, and to show how it can be applied more generally using *kernels*;
- to introduce *multilayer perceptrons* and the *backpropagation algorithm* for training them.

Reading: Russell and Norvig, chapter 18 and 19.

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:



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...

Let's look at 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,

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

An example, continued...

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

The measurements are usually referred to as *attributes* or *features*. (Technically, there is a difference between an attribute and a feature - but we won't have time to explore it.)

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

- continuous: $x_i \in [a, b]$ where $a, b \in \mathbb{R}$;
- 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 $x_i \in \{v_1, \ldots, v_p\}$.

An example, continued...

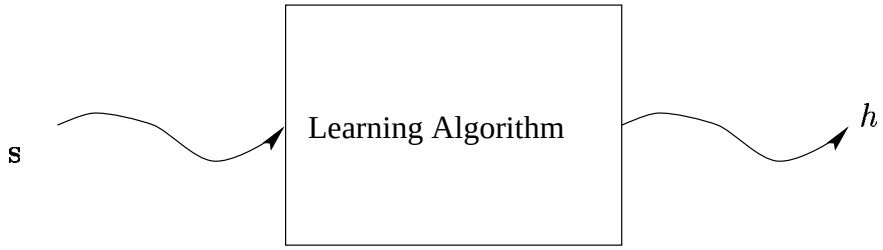
Now imagine that we have a large collection of patient histories (m in total) and for each of these we know whether or not the patient suffered from D.

- 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*,

$$\mathbf{s} = ((\mathbf{x}_1, 0), (\mathbf{x}_2, 0), (\mathbf{x}_3, 1), \ldots, (\mathbf{x}_m, 0))$$

An example, continued...

In the form of machine learning to be emphasised here, we aim to design a *learning algorithm* which takes $\mathbf{s}$ and produces a *hypothesis* h.



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

But what's a hypothesis?

What's a hypothesis?

- Denote by X the set of all possible instances.

$$X = \{\mathbf{x} | \mathbf{x} \text{ is a possible instance}\}$$

- Then a hypothesis h could simply be a *function* from X to $\{0, 1\}$.

$$h : X \to \{0, 1\}$$

- In other words, h can take *any* instance and produce a 0 or a 1, depending on whether (according to h) the patient the measurements were taken from is suffering from D.

But what's a hypothesis?

There are some important issues here, which are effectively right at the heart of machine learning. Most importantly: *the hypothesis h can assign a 0 or a 1 to **any** $\mathbf{x} \in X$*.

- This includes instances $\mathbf{x}$ that did *not* appear in the training sequence.
- The overall process therefore involves rather more than *memorising* the training sequence.
- Ideally, the aim is that h should be able to *generalize*. That is, it should be possible to use it to diagnose *new* patients.

But what's a hypothesis?

In fact we need a slightly more flexible definition of a hypothesis.

A hypothesis is a function from X to some suitable set Ω

$$h : X \to \Omega$$

because:

- there may be more than two classes:

  $$\Omega = \{\text{No disease}, \text{Disease } D_1, \text{Disease } D_2, \ldots, \text{Disease } D_c\}$$

- or, we might want h to indicated how *likely* it is that the patient has disease D

  $$\Omega = [0, 1]$$

  where 0 denotes 'definitely does have the disease' and 1 denotes 'definitely does not have it'. $h(\mathbf{x}) = 0.75$ might for example denote that the patient is reasonably certain to have the disease.

But what's a hypothesis?

- One way of thinking about the previous case is in terms of probabilities:

$$h(\mathbf{x}) = \Pr(\mathbf{x} \text{ is in class } 1)$$

- We may have $\Omega = \mathbb{R}$. For example if $\mathbf{x}$ contains several recent measurements of a currency exchange rate and we want $h(\mathbf{x})$ to be a prediction of what the rate will be in 10 minutes time. Such problems are generally known as *regression problems*.

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 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 - that is, 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!)
- Playing games. (For example, see Tesauro, 1992 and 1995, where a world class backgammon player is described.)

What have we got so far?

Extracting what we have so far, we get the following central ideas:

- A collection X of possible instances $\mathbf{x}$.
- A collection $\Omega = \{\omega_1, \ldots, \omega_c\}$ of classes to which any instance can belong. In some scenarios we might have $\Omega \subseteq \mathbb{R}$.
- A *training sequence* containing m labelled examples,

  $$\mathbf{s} = ((\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), (\mathbf{x}_3, y_3), \ldots, (\mathbf{x}_m, y_m))$$

  with $\mathbf{x}_i \in X$ and $y_i \in \Omega$ for $i = 1, \ldots, m$.
- A *learning algorithm* L which takes $\mathbf{s}$ and produces a *hypothesis* $h : X \to \Omega$. We can write,

  $$h = L(\mathbf{s}).$$

What have we got so far?

But we need some more ideas as well:

- We often need to state what kinds of hypotheses are available to L.
- The collection of available hypotheses is called the *hypothesis space* and denoted $\mathcal{H}$.

$$\mathcal{H} = \{h : h \text{ is available to } L\}$$

- Some learning algorithms do not always return the same $h \in \mathcal{H}$ for each run on a given sequence $\mathbf{s}$. In this case $L(\mathbf{s})$ is a *probability distribution* on $\mathcal{H}$.

What's the 'right' answer?

- We may sometimes assume that there is a 'correct' function that governs the relationship between the $\mathbf{x}$s and the labels.
- This is called the *target concept* and is denoted c. It can be regarded as the 'perfect' hypothesis, and so we have $c : X \to \Omega$.
- This is not always a sufficient way of thinking about the problem...

Generalization

The learning algorithm never gets to know exactly what the ‘correct’ relationship between instances and classes is - it only ever gets to see a finite number m of examples. So:

- generalization corresponds to the ability of L to pick a hypothesis h which is ‘close’ in some sense to the ‘best possible’;
- however we have to be careful about what ‘close’ means here.

For example, what if some instances are much more likely than others?

Generalization performance

How can generalization performance be assessed?

- Model the generation of training example using a probability distribution $\mathbb{P}$ on $X \times \Omega$.
- All examples are assumed to be independent and identically distributed (i.i.d.) according to $\mathbb{P}$.
- Given a hypothesis h and any example $(\mathbf{x}, y)$ we can introduce a measure $L(h, (\mathbf{x}, y))$ of the error that h makes in classifying that example.
- For example the following definitions for L might be appropriate:

$$L(h, (\mathbf{x}, y)) = I(h(\mathbf{x}) \neq y) \text{ for a classification problem}$$
$$L(h, (\mathbf{x}, y)) = (h(\mathbf{x}) - y)^2 \text{ when } \Omega \subseteq \mathbb{R}$$

Generalization performance

A reasonable definition of generalization performance is then

$$\text{er}(h) = \mathbb{E}_{(\mathbf{x},y)\in\mathbb{P}}\left(L(h,(\mathbf{x},y))\right)$$

In the case of the definition of L for classification problems given in the previous slide this gives

$$\begin{aligned}\text{er}(h) &= \mathbb{E}_{(\mathbf{x},y)\in\mathbb{P}}\left(I(h(\mathbf{x})\neq y)\right)\\ &= \mathbb{P}\left(h(\mathbf{x})\neq y\right)\end{aligned}$$

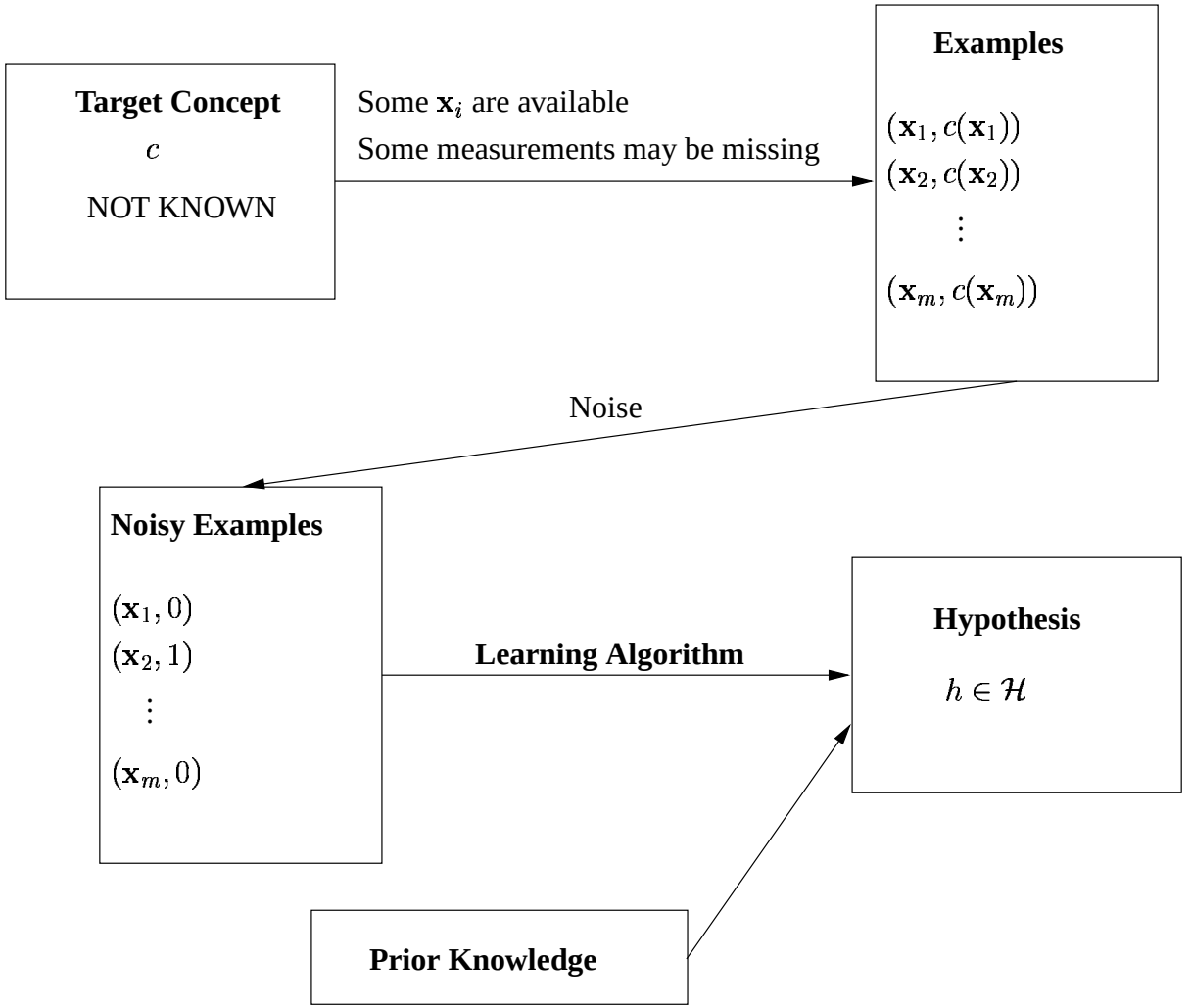
In the case of the definition for L given for regression problems, $\text{er}(h)$ is the expected square of the difference between true label and predicted label.

Problems encountered in practice

In practice there are various problems that can arise:

- Measurements may be missing from the $\mathbf{x}$ vectors.
- There may be noise present.
- Classifications in $\mathbf{s}$ may be incorrect.

The practical techniques to be presented have their own approaches to dealing with such problems. Similarly, problems arising in practice are addressed by the theory.


Target Concept
c
NOT KNOWN
Some $\mathbf{x}_i$ are available
Some measurements may be missing
Examples
$(\mathbf{x}_1, c(\mathbf{x}_1))$
$(\mathbf{x}_2, c(\mathbf{x}_2))$
$(\mathbf{x}_m, c(\mathbf{x}_m))$
Noise
Noisy Examples
$(\mathbf{x}_1, 0)$
$(\mathbf{x}_2, 1)$
$(\mathbf{x}_m, 0)$
Learning Algorithm
Hypothesis
$h \in \mathcal{H}$
Prior Knowledge

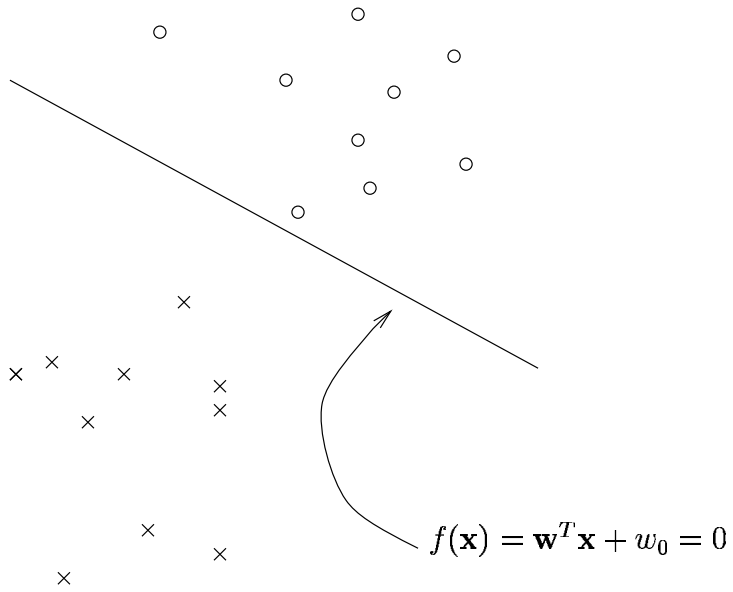
The perceptron: a review

The initial development of *linear discriminants* was carried out by Fisher in 1936, and since then they have been central to supervised learning.

Their influence continues to be felt in the recent and ongoing development of *support vector machines*, of which more later...

The perceptron: a review

We have a two-class classification problem in $\mathbb{R}^n$.



We output class 1 if $f(\mathbf{x}) \geq 0$, or class 2 if $f(\mathbf{x}) < 0$.

The perceptron: a review

So the hypothesis is

$$h(\mathbf{x}) = \text{sgn}(f(\mathbf{x})) = \text{sgn}(\mathbf{w}^T\mathbf{x} + w_0)$$

where

$$\text{sgn}(y) = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

The primal perceptron algorithm

$\eta \in \mathbb{R}^+, \mathbf{w}^{(0)} \leftarrow \mathbf{0}, w_0^{(0)} \leftarrow 0, k = 0, R = \max_i ||\mathbf{x}_i||.$

do
{
    for (each example in $\mathbf{s}$)
    {
        if ($y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \leq 0$)
        {
            $\mathbf{w} = \mathbf{w} + \eta y_i \mathbf{x}_i$
            $w_0 = w_0 + \eta y_i R^2$
            $k = k + 1$
        }
    }
}
while (mistakes are made in the for loop)
return $\mathbf{w}, w_0$.

Novikoff's theorem

The perceptron algorithm does *not* converge if $\mathbf{s}$ is not linearly separable. However Novikoff proved the following:

Theorem 1 *If $\mathbf{s}$ is non-trivial and linearly separable, where there exists a hyperplane $(\mathbf{w}_{optimum}, w_{optimum})$ with $||\mathbf{w}_{optimum}|| = 1$ and*

$$y_i(\mathbf{w}_{optimum}^T \mathbf{x}_i + w_{optimum}) \geq \gamma$$

for $i = 1, \ldots, m$, then the perceptron algorithm makes at most

$$\left(\frac{2R}{\gamma}\right)^2$$

mistakes.

Dual form of the perceptron algorithm

If we set $\eta = 1$ then the primal perceptron algorithm operates by adding and subtracting misclassified points $\mathbf{x}_i$ to an initial $\mathbf{w}$ at each step.

As a result, when it stops we can represent the final $\mathbf{w}$ as

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$$

Note:

- the values α_i are positive and proportional to the number of times $\mathbf{x}_i$ is misclassified;
- if $\mathbf{s}$ is fixed then the vector $\boldsymbol{\alpha}^T = \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_m \end{pmatrix}$ is an alternative representation of $\mathbf{w}$.

Dual form of the perceptron algorithm

Using these facts, the hypothesis can be re-written

$$
\begin{aligned}
h(\mathbf{x}) &= \operatorname{sgn}\left(\mathbf{w}^T\mathbf{x} + w_0\right) \\
&= \operatorname{sgn}\left(\left(\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i\right)^T \mathbf{x} + w_0\right) \\
&= \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i \left(\mathbf{x}_i^T \mathbf{x}\right) + w_0\right)
\end{aligned}
$$

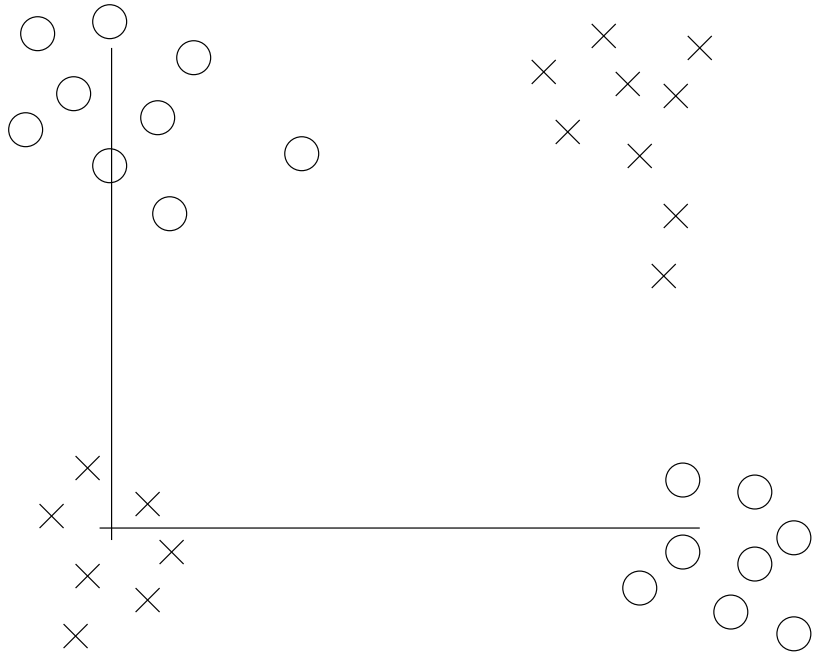
Dual form of the perceptron algorithm

$\boldsymbol{\alpha}^{(0)} \leftarrow \mathbf{0}, w_0^{(0)} \leftarrow 0, R = \max_i ||\mathbf{x}_i||.$

do
{
    for (each example in $\mathbf{s}$)
    {
        if ($y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \leq 0$)
        {
            $\alpha_i = \alpha_i + 1$
            $w_0 = w_0 + y_i R^2$
        }
    }
}
while (mistakes are made in the for loop)
return $\boldsymbol{\alpha}, w_0$.

Mapping to a bigger space

There are many problems a perceptron can’t solve.

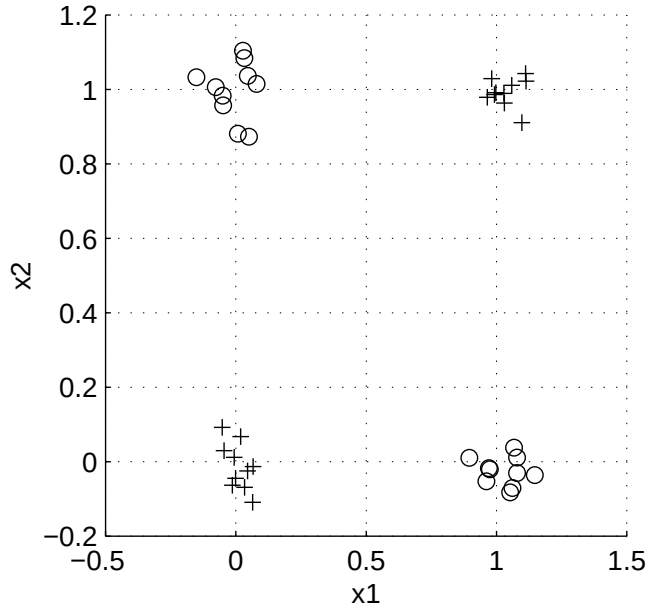


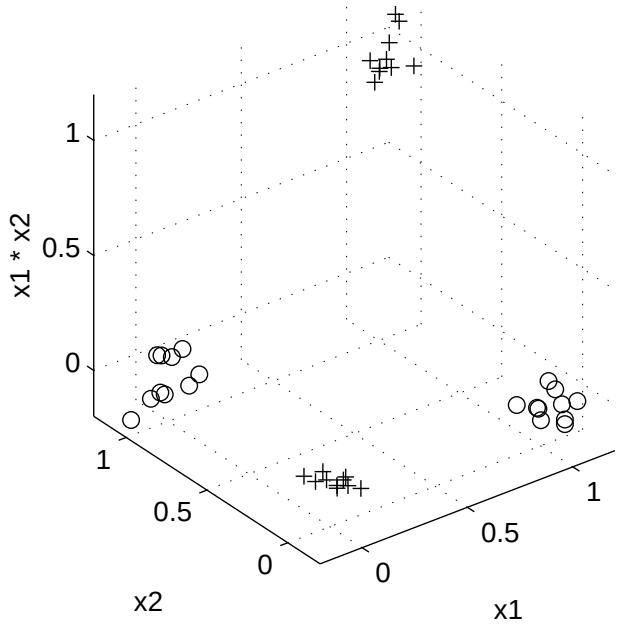
The classic example is the parity problem.

Mapping to a bigger space

But what happens if we add another element to $\mathbf{x}^T = (\ x_1\ \ x_2\ )$?

For example we could use $\mathbf{x}^T = (\ x_1\ \ x_2\ \ x_1 x_2\ )$.





In $\mathbb{R}^3$ a perceptron can easily solve this problem.

Mapping to a bigger space

$$
\begin{aligned}
h(\mathbf{x}) &= \mathrm{sgn}(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2) \\
&= \mathrm{sgn}(w_0 + w_1 \phi_1(\mathbf{x}) + w_2 \phi_2(\mathbf{x}) + w_3 \phi_3(\mathbf{x})) \\
&= \mathrm{sgn}(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}) + w_0)
\end{aligned}
$$

where

$$
\begin{aligned}
\phi_1(\mathbf{x}) &= x_1 \\
\phi_2(\mathbf{x}) &= x_2 \\
\phi_3(\mathbf{x}) &= x_1 x_2 \\
\mathbf{\Phi}(\mathbf{x})^T &= (\ \phi_1(\mathbf{x})\ \ \phi_2(\mathbf{x})\ \ \phi_3(\mathbf{x})\ ) \\
\mathbf{w}^T &= (\ w_1\ \ w_2\ \ w_3\ )
\end{aligned}
$$

Mapping to a bigger space

This is an old trick, and the functions ϕ_i can be anything we like.

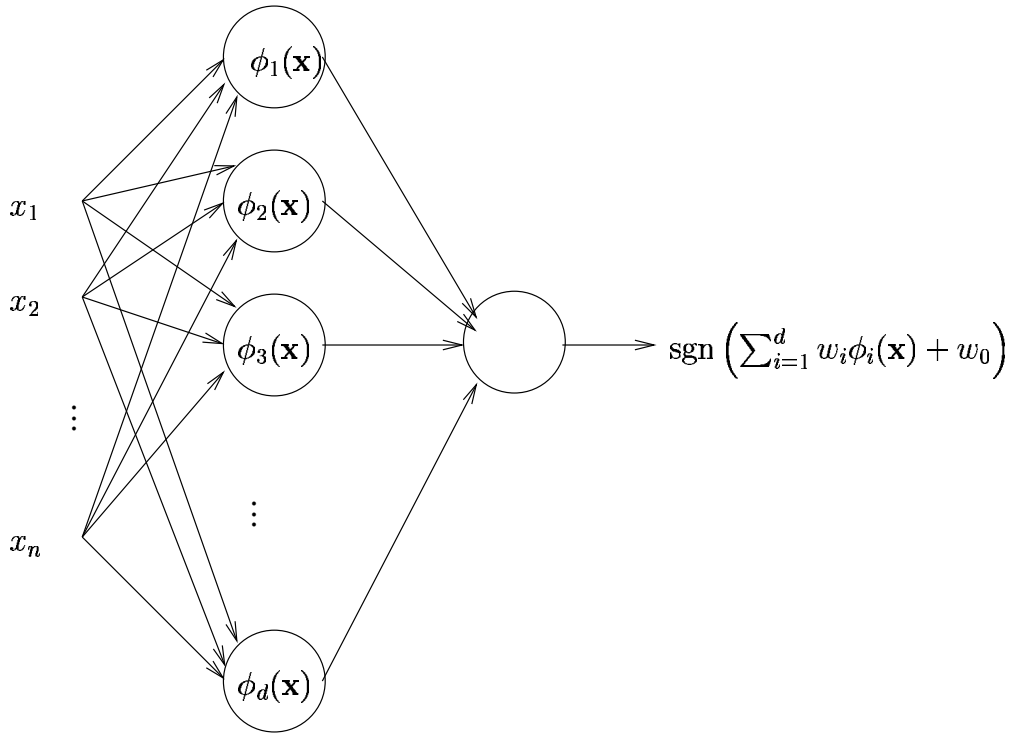
Example: In a multilayer perceptron

$$\phi_i(\mathbf{x}) = \frac{1}{1 + \exp(-(\mathbf{w}_i^T \mathbf{x} + w_{i0}))}$$

where $\mathbf{w}_i$ is the vector of weights and w_{i0} the bias associated with hidden node i.

Note however that for the time being the functions ϕ_i are *fixed*, whereas in a multilayer perceptron they are allowed to vary as a result of varying the $\mathbf{w}_i$ and w_{i0}.

Mapping to a bigger space



Mapping to a bigger space

We can now use a perceptron in the high-dimensional space, obtaining a hypothesis of the form

$$h(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{d} w_i \phi_i(\mathbf{x}) + w_0\right)$$

where the w_i are the weights from the hidden nodes to the output node.

So:

- start with $\mathbf{x}$;
- use the ϕ_i to map it to a bigger space;
- use a (linear) perceptron in the bigger space.

Mapping to a bigger space

What happens if we use the dual form of the perceptron algorithm in this process?

We end up with a hypothesis of the form

$$h(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i)^T \mathbf{\Phi}(\mathbf{x}) + w_0\right)$$

where $\mathbf{\Phi}(\mathbf{x})$ is the vector

$$\mathbf{\Phi}(\mathbf{x})^T = \begin{pmatrix} \phi_1(\mathbf{x}) & \phi_2(\mathbf{x}) & \cdots & \phi_d(\mathbf{x}) \end{pmatrix}$$

Notice that this introduces the possibility of a tradeoff of m and d.

The sum has (potentially) become smaller.

The cost associated with this is that we may have to calculate $\mathbf{\Phi}(\mathbf{x}_i)^T \mathbf{\Phi}(\mathbf{x})$ several times.

Kernels

This suggests that it might be useful to be able to calculate $\mathbf{\Phi}(\mathbf{x}_i)^T\mathbf{\Phi}(\mathbf{x})$ easily.

In fact such an observation has far-reaching consequences.

Definition 2 *A* kernel *is a function K such that for all vectors $\mathbf{x}$ and $\mathbf{y}$*

$$K(\mathbf{x}, \mathbf{y}) = \mathbf{\Phi}(\mathbf{x})^T\mathbf{\Phi}(\mathbf{y})$$

Note that a given kernel naturally corresponds to an underlying collection of functions ϕ_i. Ideally, we want to make sure that the value of d does not have a great effect on the calculation of $K(\mathbf{x}, \mathbf{y})$. If this is the case then

$$h(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + w_0\right)$$

is easy to evaluate.

Kernels: an example

We can use

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y})^p$$

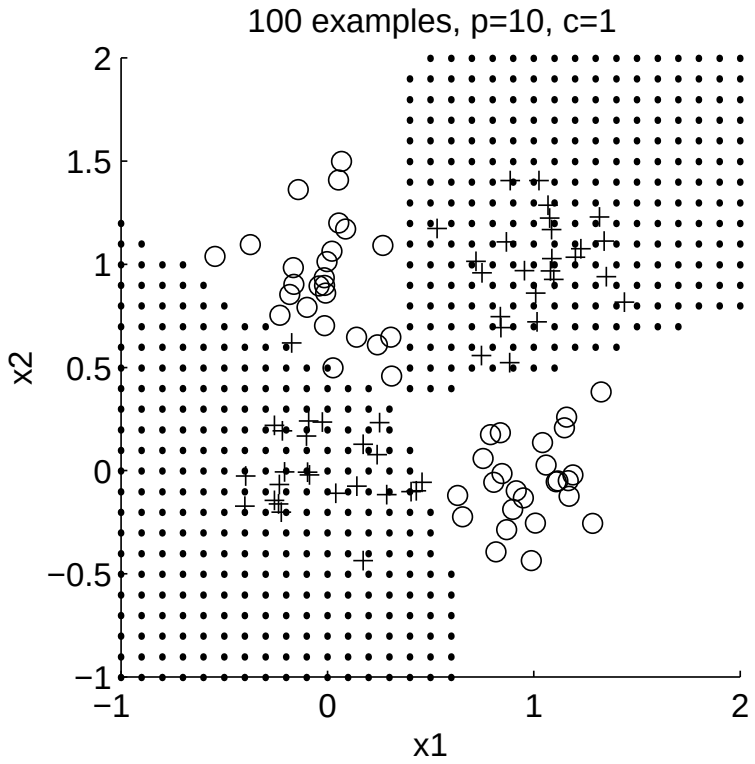
or

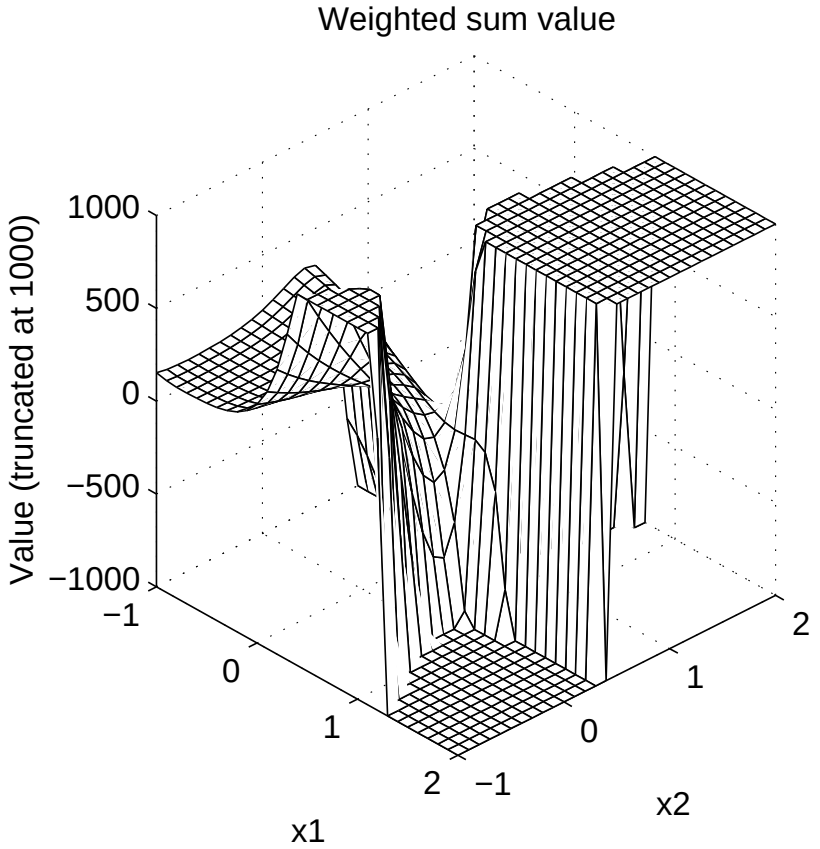
$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + c)^p$$

to obtain polynomial kernels.

In the latter case we have $\binom{n+p}{p}$ features that are monomials up to degree p, so the decision boundary obtained using K as described above will be a polynomial curve of degree p.

Kernels: an example





Gradient descent

An alternative method for training a basic perceptron works as follows. We define a measure of *error* for a given collection of weights. For example

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

where the sgn has not been used. Modifying our notation slightly so that

$$\mathbf{x}^T = (\ 1\ \ x_1\ \ x_2\ \ \cdots\ \ x_n\ )$$
$$\mathbf{w}^T = (\ w_0\ \ w_1\ \ w_2\ \ \cdots\ \ w_n\ )$$

gives

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

Gradient descent

$E(\mathbf{w})$ is parabolic and has a unique global minimum and no local minima. We therefore start with a random $\mathbf{w}$ and update it as follows:

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

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 $E(\mathbf{w})$.

Simple feedforward neural networks

- We continue using the same notation as previously.
- Usually, we think in terms of a training algorithm L finding a hypothesis h based on a training sequence $\mathbf{s}$

$$h = L(\mathbf{s})$$

  and then classifying new instances $\mathbf{x}$ by evaluating $h(\mathbf{x})$.
- Usually with neural networks the training algorithm provides a vector $\mathbf{w}$ of *weights*. We therefore have a hypothesis that depends on the weight vector. This is often made explicit by writing

$$\mathbf{w} = L(\mathbf{s})$$

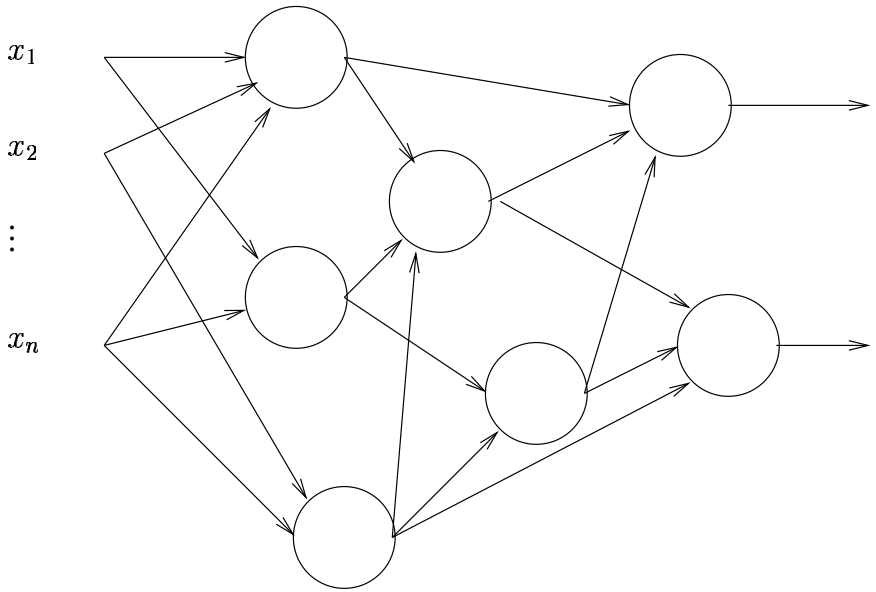
  and representing the hypothesis as a mapping depending on both $\mathbf{w}$ and the new instance $\mathbf{x}$, so

$$\text{classification of } \mathbf{x} = h(\mathbf{w}; \mathbf{x})$$

Backpropagation: the general case

First, let's look at the general case.

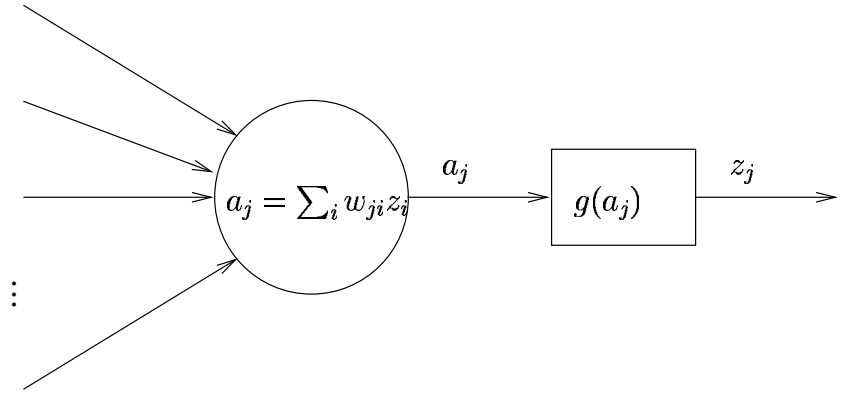
We have a completely unrestricted feedforward structure:



For the time being, there may be several outputs, and no specific layering is assumed.

Backpropagation: the general case

For each node:



- w_{ji} connects node i to node j.
- a_j is the weighted sum or *activation* for node j.
- g is the *activation function*.
- $z_j = g(a_j)$.

Backpropagation: the general case

In addition, there is often a *bias* input for each node, which is always set to 1.

This is not always included explicitly; sometimes the bias is included by writing the weighted summation as

$$a_j = \sum_i w_{ij} x_i + w_{j0}$$

where w_{j0} is the bias for node j.

Backpropagation: the general case

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* $E(\mathbf{w})$.

Backpropagation: the general case

How can we find a set of weights that minimises $E(\mathbf{w})$?

The approach used by the backpropagation algorithm is very simple:

1. begin at step 0 with a randomly chosen collection of weights $\mathbf{w}_0$;
2. at the ith step, calculate the gradient $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ of $E(\mathbf{w})$ at the point $\mathbf{w}_i$;
3. update the weight vector by taking a small step in the direction of the gradient
$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_i}$$
4. repeat this process until $E(\mathbf{w})$ is sufficiently small.

Backpropagation: the general case

In order to do this we have to calculate

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

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 inputs and calculate the values a_j and z_j for all the nodes. This is called *forward propagation*.

We have

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

where we've defined

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

and used the fact that

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}}\left(\sum_k z_k w_{jk}\right) = z_i$$

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

Backpropagation: the general case

When j is an output unit this is easy as

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

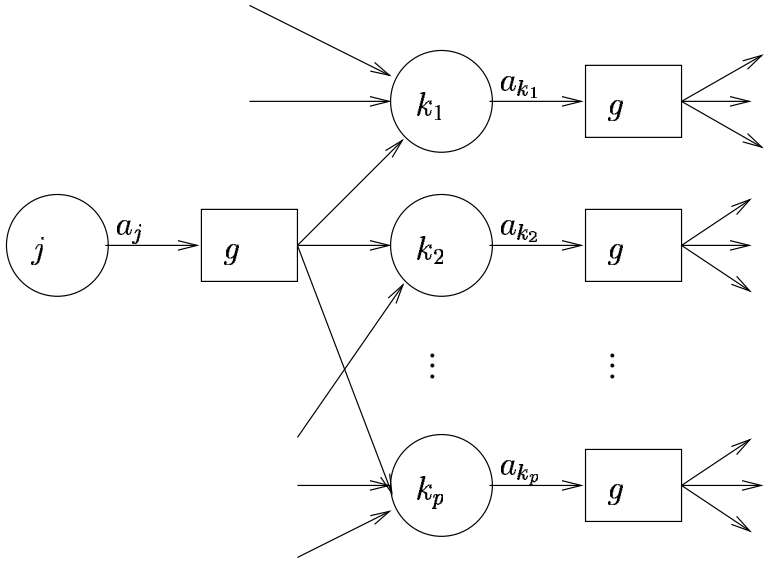
and the first term is in general easy to calculate for a given E.

Backpropagation: the general case

When j is not an output unit we have

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

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



Backpropagation: the general case

Then

$$\frac{\partial E_p(\mathbf{w})}{\partial a_k} = \delta_k$$

by definition, and

$$\begin{aligned}\frac{\partial a_k}{\partial a_j} &= \frac{\partial}{\partial a_j}\left(\sum_i w_{ki} g(a_i)\right) \\ &= w_{kj} g'(a_j)\end{aligned}$$

So

$$\begin{aligned}\delta_j &= \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k w_{kj} g'(a_j) \\ &= g'(a_j) \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k w_{kj}\end{aligned}$$

Backpropagation: the general case

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

1. Forward propagation: apply $\mathbf{x}_p$ and calculate outputs etc for *all* the nodes in the network.
2. Backpropagation 1: for *outputs* j

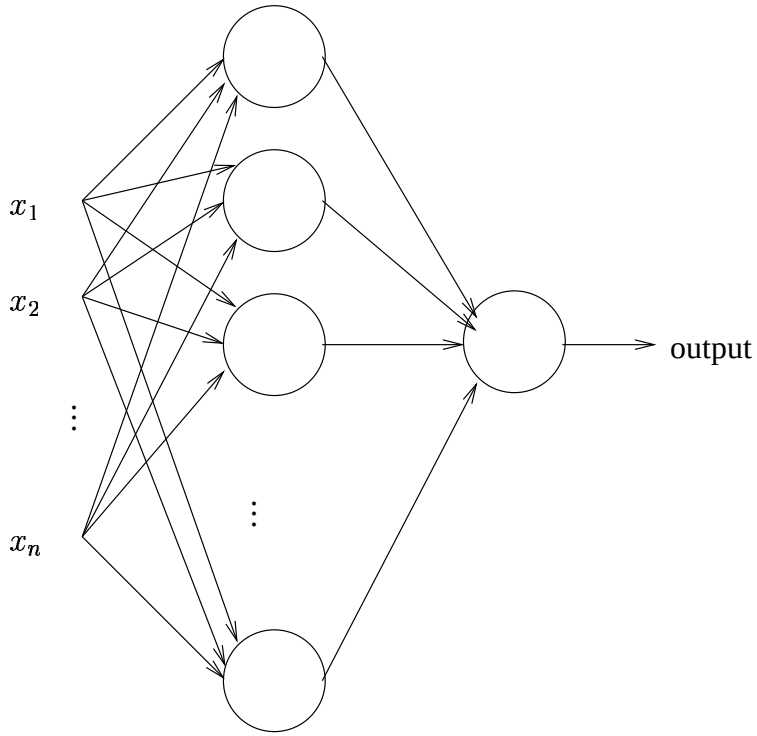
$$\frac{\partial E_p(\mathbf{w})}{\partial w_{ji}} = z_i \delta_j = z_i g'(a_j) \frac{\partial E_p(\mathbf{w})}{\partial z_j}$$

3. Backpropagation 2: For other nodes

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{ji}} = z_i g'(a_j) \sum_k \delta_k w_{kj}$$

where the δ_k were calculated at an earlier step.

Backpropagation: a specific example



Backpropagation: a specific example

For the output: $g(a) = a$.

For the other nodes:

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

so

$$g'(a) = g(a)(1 - g(a))$$

Also,

$$E_p(\mathbf{w}) = \frac{1}{2}(y_p - h(\mathbf{w}; \mathbf{x}_p))^2$$

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

Backpropagation: a specific example

For the output:

We have

$$\begin{aligned}\frac{\partial E_p(\mathbf{w})}{\partial z_{\text{output}}} &= \frac{\partial}{\partial z_{\text{output}}}\left(\frac{1}{2}(y_p - z_{\text{output}})^2\right) \\ &= z_{\text{output}} - y_p \\ &= h(\mathbf{w};\mathbf{x}_p) - y_p\end{aligned}$$

and

$$g'(a) = 1$$

so

$$\delta_{\text{output}} = h(\mathbf{w};\mathbf{x}_p) - y_p$$

and

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

Backpropagation: a specific example

For the hidden nodes:

We have

$$\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}_{ji}} = z_i g'(a_j) \sum_k \delta_k w_{kj}$$

but there is only one output so

$$\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}_{ji}} = z_i g(a_j)(1 - g(a_j)) \delta_{\text{output}} w_{\text{output}j}$$

and we have a value for δ_{output} so

$$\boxed{\begin{aligned}\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}_{ji}} &= z_i g(a_j)(1 - g(a_j))(h(\mathbf{w}; \mathbf{x}_p) - y_p) w_{\text{output}j} \\ &= x_i z_j (1 - z_j)(h(\mathbf{w}; \mathbf{x}_p) - y_p) w_{\text{output}j}\end{aligned}}$$

Putting it all together

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

Batch: (as described previously)

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

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_i}$$

Sequential: using just one pattern at once

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \left. \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_i}$$

selecting patterns in sequence or at random.

Example: the classical parity problem

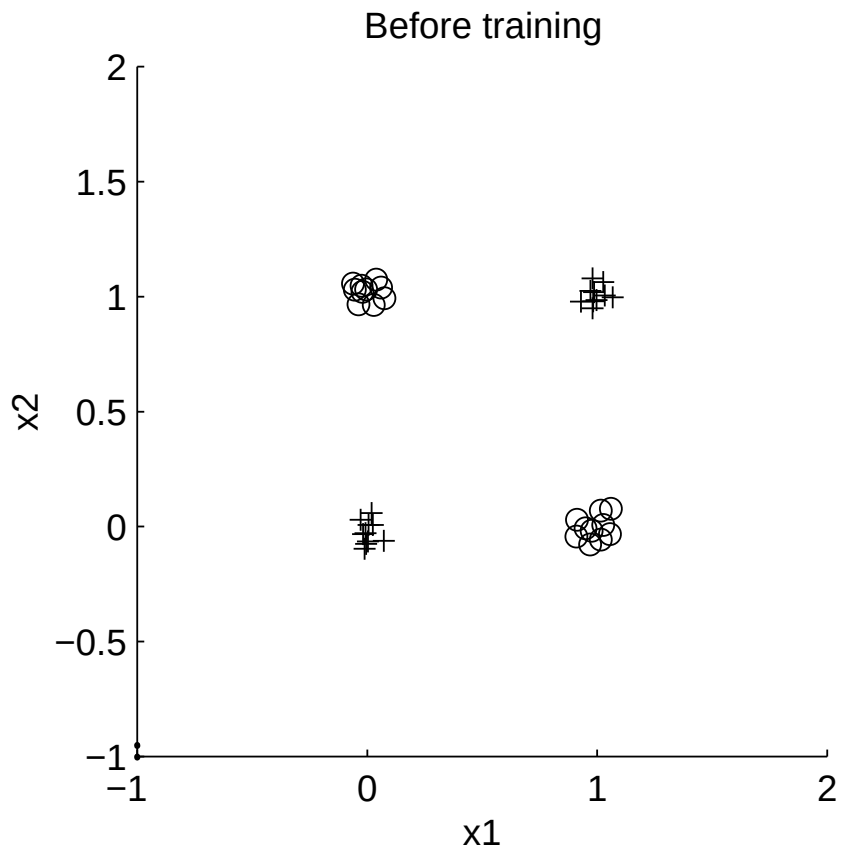
As an example we show the result of training a network with:

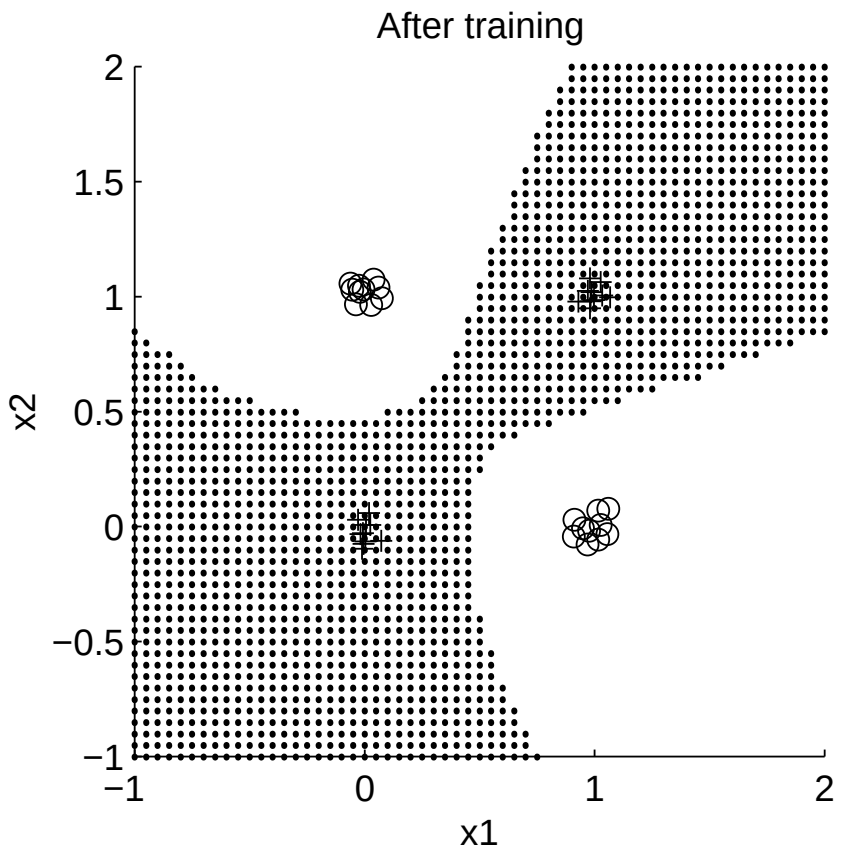
- two inputs;
- one output;
- one hidden layer containing 5 units;
- $\alpha = 0.01$;
- all other details as above.

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

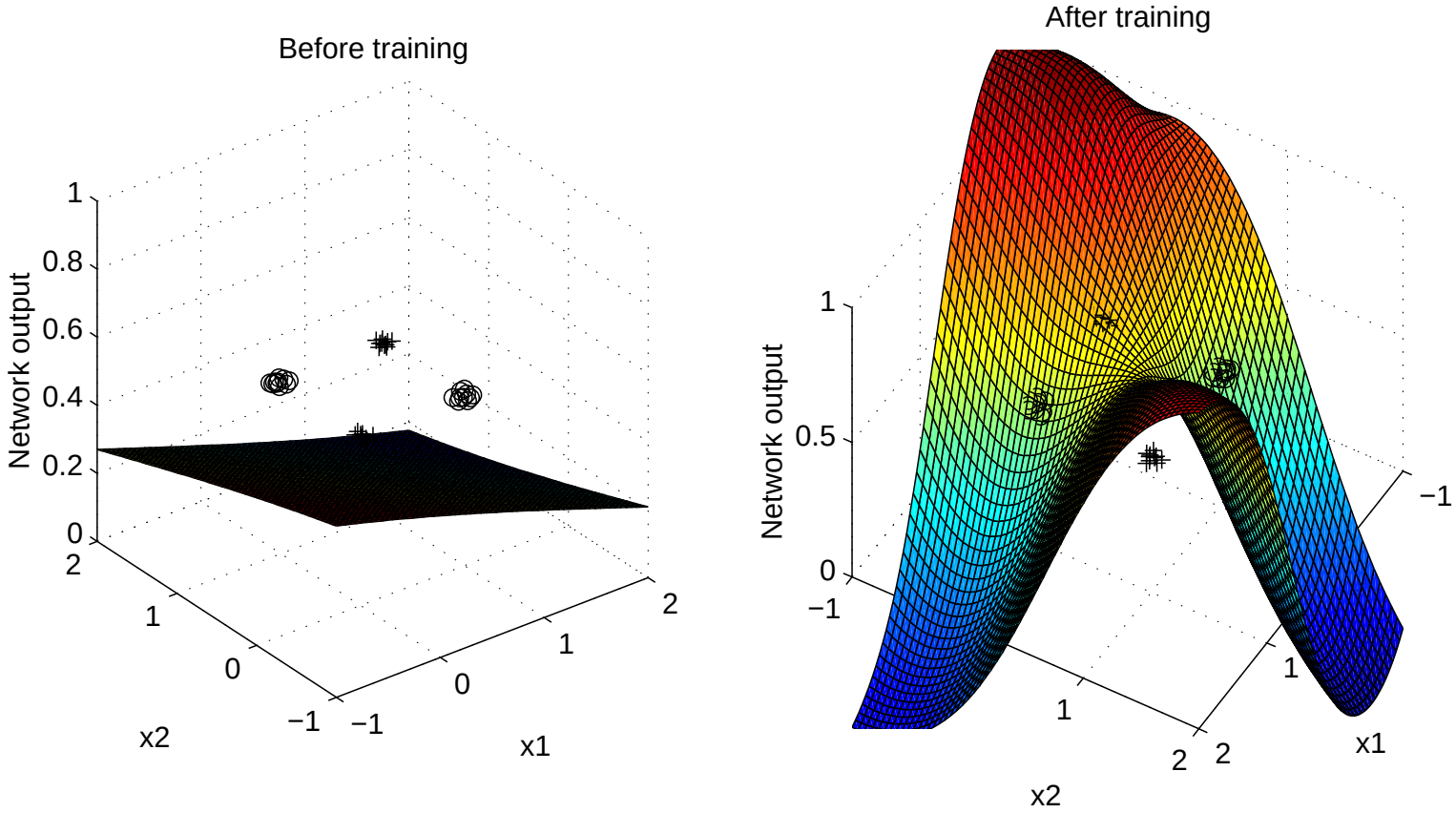
The sequential approach is used, with 1000 repetitions through the entire training sequence.

Example: the classical parity problem





Example: the classical parity problem



Example: the classical parity problem