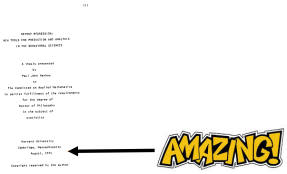
Part VI

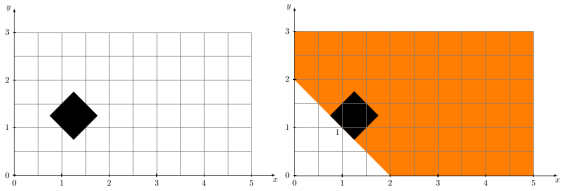
Feed-Forward Networks



Structure

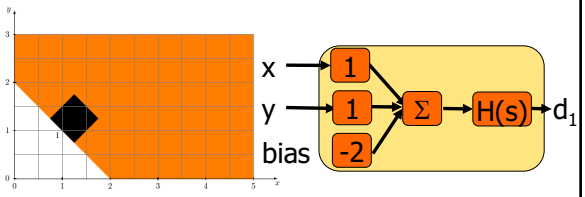
- Biological inspiration ✓
- History of Artificial Neural Networks ✓
- Two basic models of neural networks:
  - The perceptron ✓
  - The feed-forward network
- Learning in ANNs:
  - The backpropagation algorithm

Diamond (planar) problem

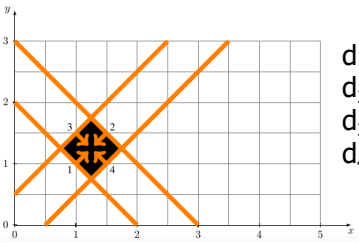


Not linearly separable

Neuron for orange half-space: $d_1 = H(x + y - 2)$

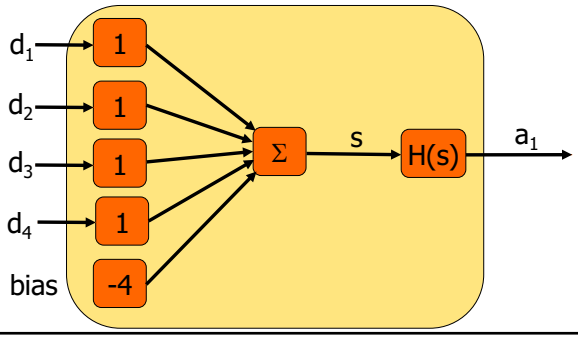


One neural for each half-space

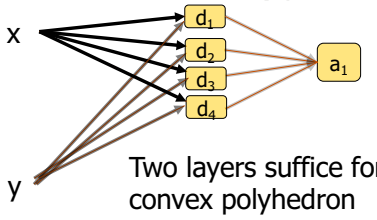


$d_1 = H(x+y - 2)$
$d_2 = H(-x-y + 3)$
$d_3 = H(x-y + 1)$
$d_4 = H(-x+y + 1)$

Another neuron for and: $a_1 = d_1 \wedge d_2 \wedge d_3 \wedge d_4$



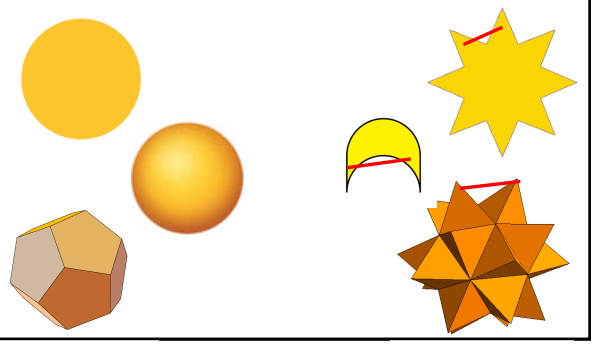
2-4-1 network



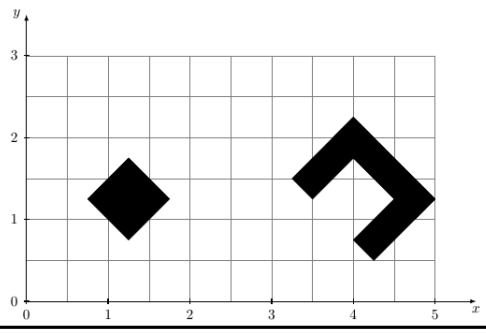
Two layers suffice for detecting a convex polyhedron

A set S is convex if for every pair of points x and y in S it follows that the line between x and y is also in S

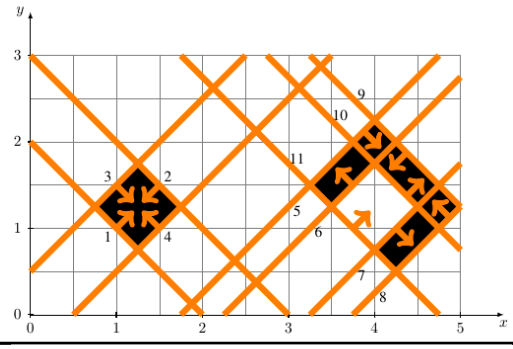
Convex and non-convex sets



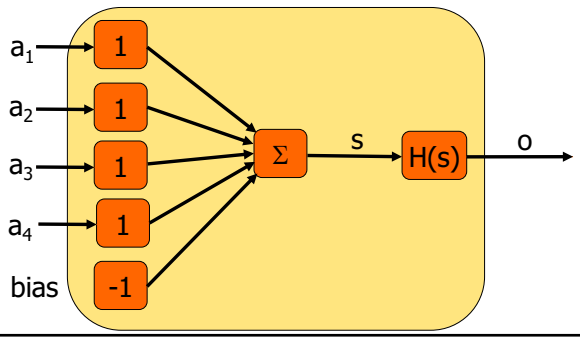
Non-convex (planar) problem



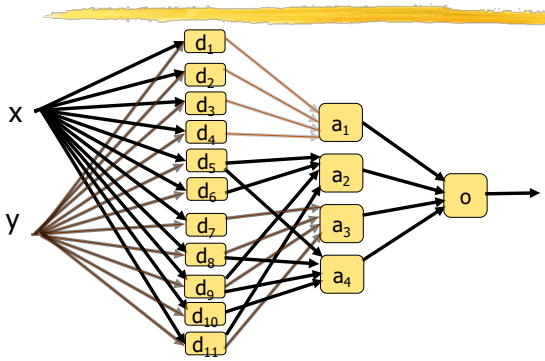
Three-layer problem (planar)



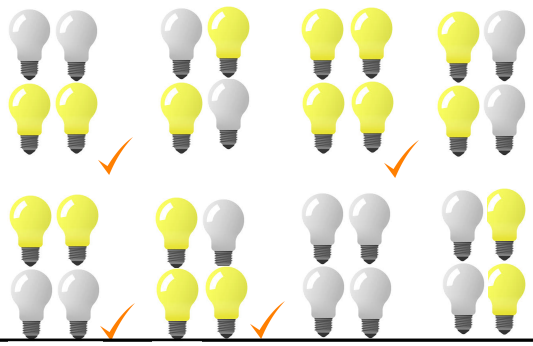
Neuron for logical or: $o = a_1 \vee a_2 \vee a_3 \vee a_4$



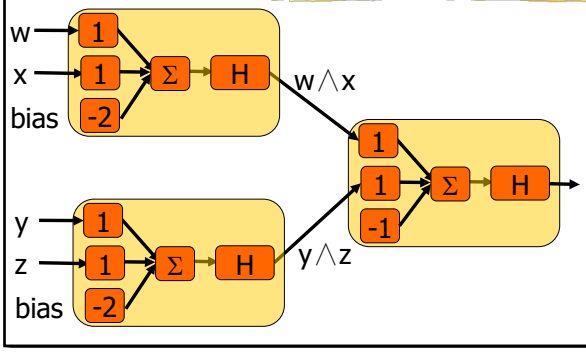
2-11-4-1 network



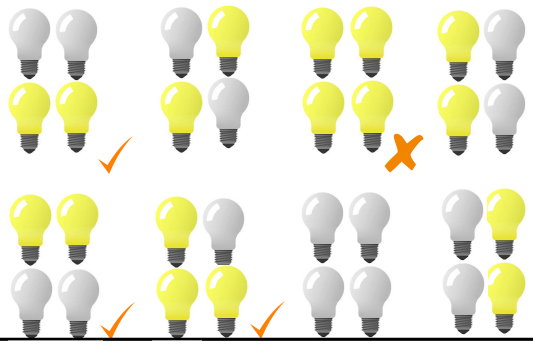
Lights in a row problem



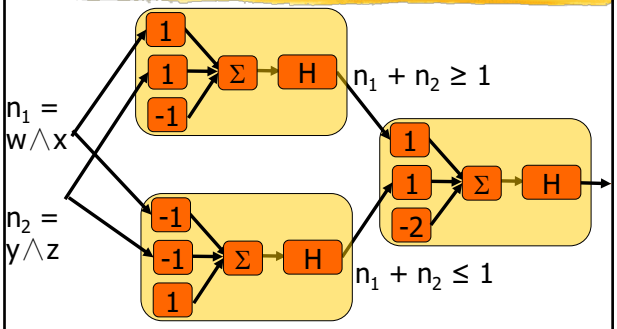
4-2-1 network for lights in a row where p = (w,x,y,z)



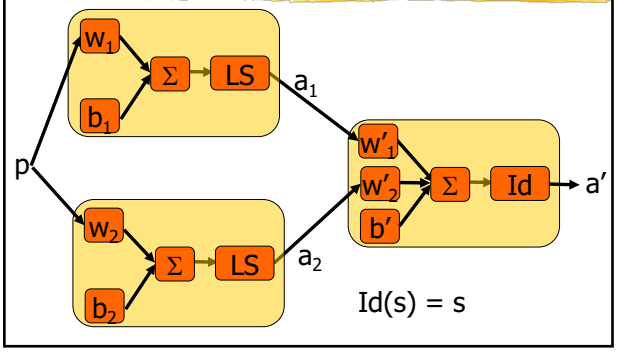
Reprise: lights in a (single) row problem



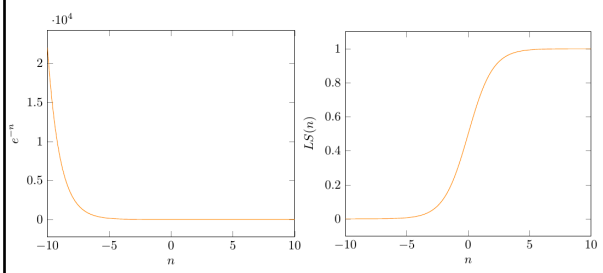
Reprise: 4-2-2-1 network for lights in a (single) row



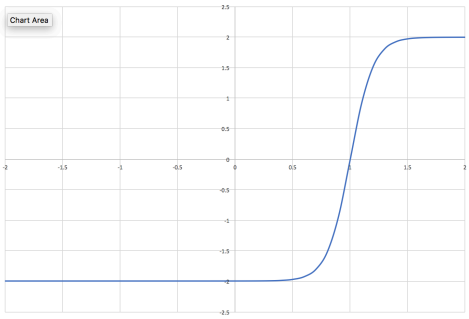
Varying parameters of a 1-2-1 network



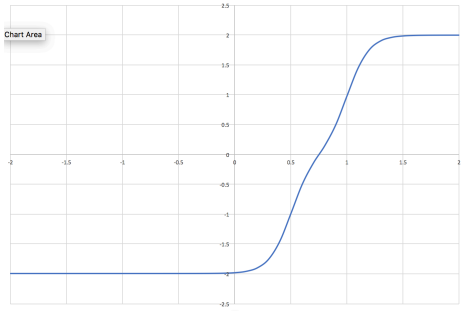
Recall: Log-Sigmoid function $LS(n) = 1/(1 + e^{-n})$



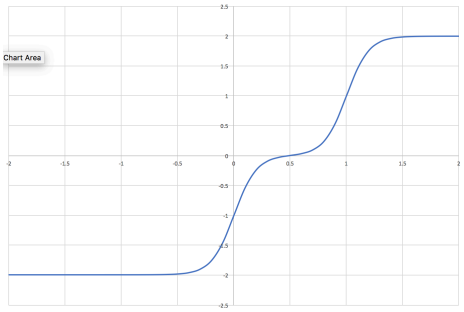
w_1= 10, b_1=-10, w_2=10, b_2=-10, w'_1=1, w'_2=1, b'= 0



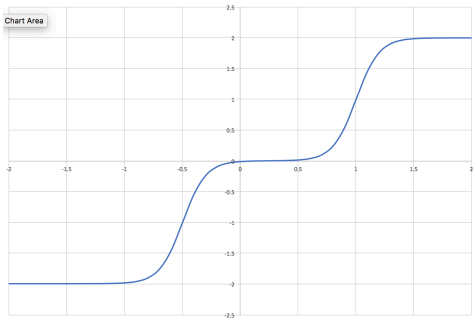
w_1= 10, b_1= -10, w_2=10, b_2=-5, w'_1=1, w'_2=1, b'=0



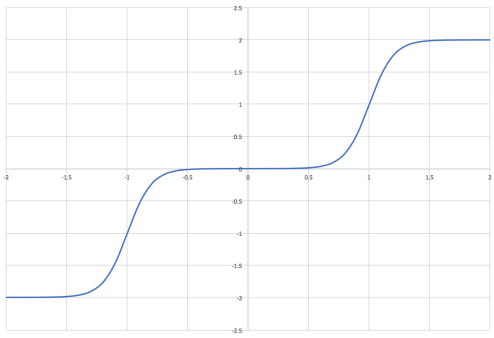
w_1=10, b_1=-10, w_2=10, b_2=0, w'_1=1, w'_2=1, b'=0



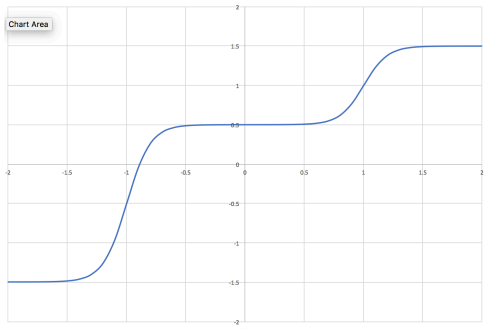
w_1=10, b_1=-10, w_2=10, b_2=5, w'_1=1, w'_2=1, b'=0



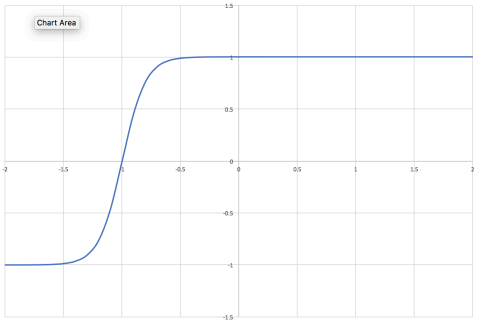
w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=1, w'_2=1, b'=0



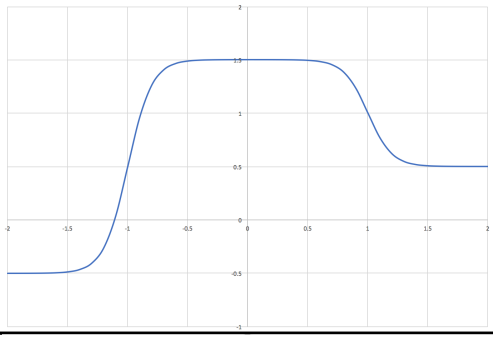
w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=0.5, w'_2=1, b'=0



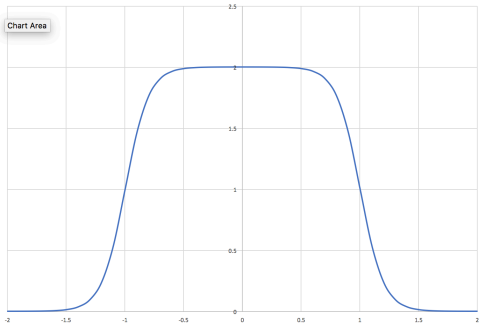
w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=0, w'_2=1, b'=0



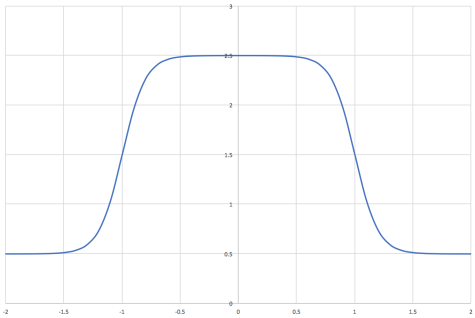
w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=-0.5, w'_2=1, b'=0



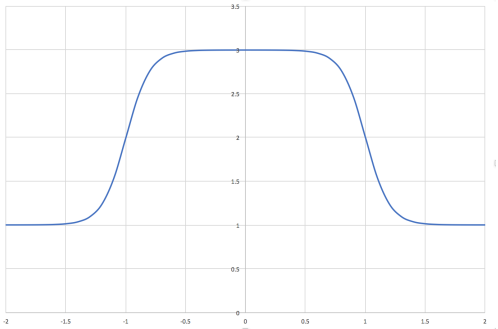
w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=-1, w'_2=1, b'=0



w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=-1, w'_2=1, b'=0.5



w_1=10, b_1=-10, w_2=10, b_2=10, w'_1=-1, w'_2=1, b'=1

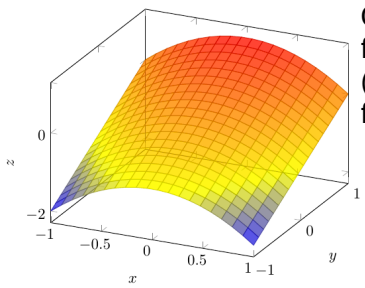


Hornik, Neural Networks, 2(5), 1989

- Showed "Multilayer feed-forward networks are universal approximators"
- Any Borel-integrable* function can be approximated to arbitrary precision given:
  - enough neurons in the hidden layer;
  - log-sigmoid functions in the hidden layer;
  - a linear function in the final layer
- Provides a guarantee for learning

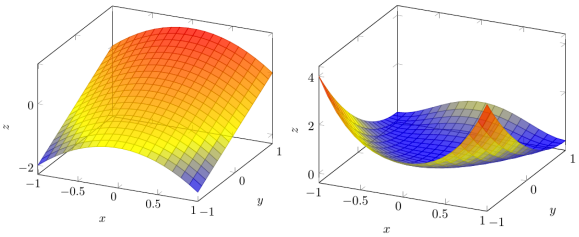
*can approximated as series of pieces

Steepest descent algorithm



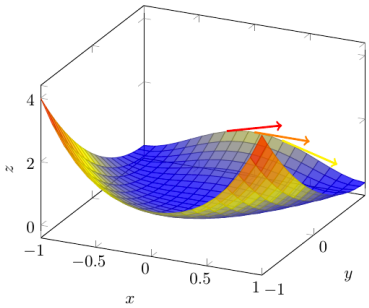
Given $f(x,y)=y - x^2$ find a coordinate (x, y) such that $f(x, y) = 0$

Illustrating $f = y - x^2$ and $f^2 = (y - x^2)^2$



$z=f(x,y)$ $z=f^2(x,y)$

Interpreting partial derivative $\partial f^2/\partial x(¼,1) = -15/16$



If we keep y at 1 but increase x by a tiny value Δ then $f(¼,1)$ decreases by $15\Delta/16$

If we keep y at 1 but increase x by a tiny value Δ then $f(-¼,1)$ increases by $15\Delta/16$

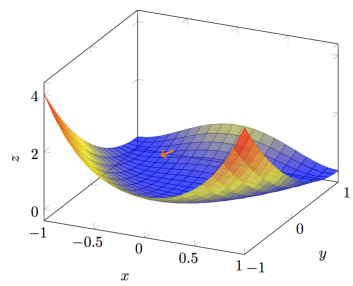
Minimise $f^2 = (y - x^2)^2 = y^2 - 2x^2y + x^4$

- Partial derivative $\partial f/\partial x = -4xy + 4x^3$
- Partial derivative $\partial f/\partial y = 2y - 2x^2$
- $f^2(¼, 1) = -15/16$
- $f^2(0, 1) = 1$
- $f^2(-¼, 1) = -15/16$
- $\partial f^2/\partial x(¼, 1) = -4(¼)1+4(¼)^3 = -15/16$
- $\partial f^2/\partial x(0, 1) = -4(0)1+4(0)^3 = 0$
- $\partial f^2/\partial x(-¼, 1) = -4(-¼)1+4(-¼)^3 = 15/16$

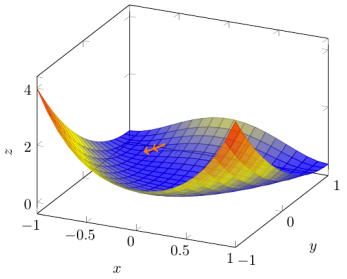
Steepest descent algorithm

- Guess initial (x_0, y_0)
- Update $x_{i+1} = x_i - \alpha\Delta x_i$
- Update $y_{i+1} = y_i - \alpha\Delta y_i$ where
  - $\alpha = 1/10$ (a small positive learning rate)
  - $\Delta x_i = \partial f/\partial x(x_i, y_i)$
  - $\Delta y_i = \partial f/\partial y(x_i, y_i)$
- Continue until $(x_{i+1}, y_{i+1}) \approx (x_i, y_i)$

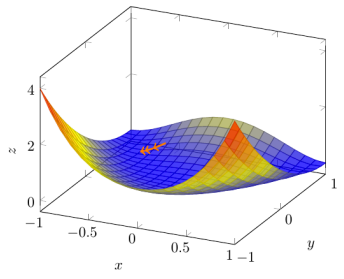
$(x_0, y_0) = (-0.5, 0.7)$ $(x_1, y_1) = (-0.59, 0.61)$



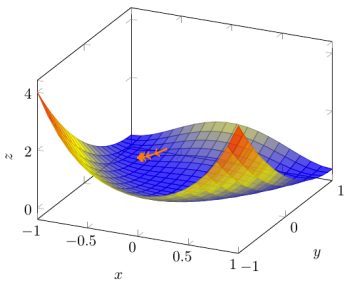
$(x_1, y_1) = (-0.59, 0.61)$
$(x_2, y_2) = (-0.6518, 0.5576)$



$(x_2, y_2) = (-0.6518, 0.5576)$
$(x_3, y_3) = (-0.6864, 0.5311)$



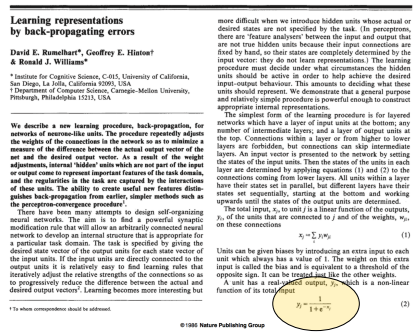
$(x_{13}, y_{13}) = (-0.7145, 0.5108)$
$(x_{14}, y_{14}) = (-0.7146, 0.5107)$



Sanity check:

$f(x_{14},y_{14})=y_{14} - x_{14}^2$
$= 0.5107 - 0.7146^2$
$= 0.5107 - 0.5106$
$= 0.0001$

Back-propagation errors



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

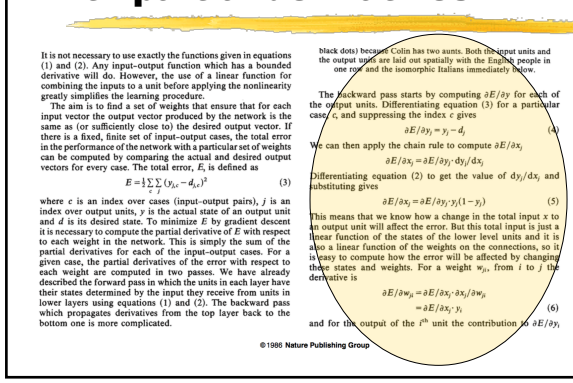
A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.

©1986 Nature Publishing Group

With partial derivatives



It is not necessary to use exactly the functions given in equations (1) and (2). Any input–output function which has a bounded derivative will do. However, the use of a linear function for combining the inputs to a unit before applying the nonlinearity greatly simplifies the learning procedure.

The aim is to find a set of weights that ensure that for each input vector the output vector produced by the network is the same as (or sufficiently close to) the desired output vector. If there is a fixed, finite set of input–output cases, the total error in the performance of the network with a particular set of weights can be computed by comparing the actual and desired output vectors for every case. The total error, E, is defined as

$$E = \tfrac{1}{2}\sum_c \sum_j (y_{j,c} - d_{j,c})^2 \quad (3)$$

where c is an index over cases (input–output pairs), j is an index over output units, y is the actual state of an output unit and d is its desired state. To minimize E by gradient descent it is necessary to compute the partial derivative of E with respect to each weight in the network. This is simply the sum of the partial derivatives for each of the input–output cases. For a given case, the partial derivatives of the error with respect to each weight are computed in two passes. We have already described the forward pass in which the units in each layer have their states determined by the input they receive from units in lower layers using equations (1) and (2). The backward pass which propagates derivatives from the top layer back to the bottom one is more complicated.

black dots) because Colin has two aunts. Both the input units and the output units are laid out spatially with the English people in one row and the isomorphic Italians immediately below.

The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \quad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot dy_j/dx_j$$

Differentiating equation (2) to get the value of dy_j/dx_j and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1 - y_j) \quad (5)$$

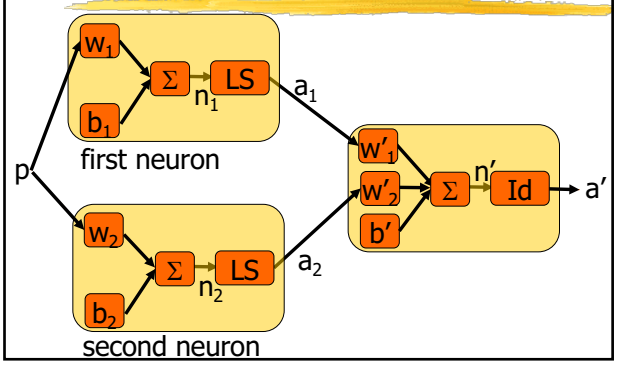
This means that we know how a change in the total input x to an output unit will affect the error. But this total input is just a linear function of the states of the lower level units and it is also a linear function of the weights on the connections, so it is easy to compute how the error will be affected by changing these states and weights. For a weight w_{ji}, from i to j the derivative is

$$\partial E/\partial w_{ji} = \partial E/\partial x_j \cdot \partial x_j/\partial w_{ji}$$
$$= \partial E/\partial x_j \cdot y_i \quad (6)$$

and for the output of the ith unit the contribution to $\partial E/\partial y_i$

©1986 Nature Publishing Group

1-2-1 network with
$LS(n) = 1/(1 + e^{-n})$



Batch training for 1-2-1

- Consider p_1, p_2 and p_3 with labels t_1, t_2, t_3
- Calculate new parameters w_1,b_1, w_2,b_2, w'_1,w'_2,b' to reduce error $f = (t_1 - a'_1)^2$ for input p_1 and output a'_1
- Calculate new parameters for p_2, a'_2 and t_2
- Calculate new parameters for p_3, a'_3 and t_3
- Update with mean parameters and iterate

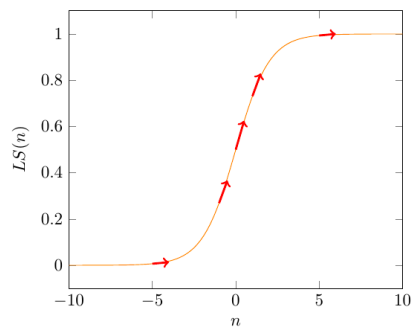
Steepest descent on 1-2-1

- Update second layer:
  - $w'_1 := w'_1 - \alpha\partial f/\partial w'_1$
  - $w'_2 := w'_2 - \alpha\partial f/\partial w'_2$
  - $b' := b' - \alpha\partial f/\partial b'$
- Update first neuron of first layer:
  - $w_1 := w_1 - \alpha\partial f/\partial w_1$
  - $b_1 := b_1 - \alpha\partial f/\partial b_1$
- Update second neuron of first layer:
  - $w_2 := w_2 - \alpha\partial f/\partial w_1$
  - $b_2 := b_2 - \alpha\partial f/\partial b_2$

Primer on chain rule

- Consider $\partial LS/\partial n$ for $LS = 1/(1 + e^{-n})$
- Put $w = (1 + e^{-n})$ so $LS = w^{-1}$
- Use $\partial LS/\partial n = (\partial LS/\partial w)(\partial w/\partial n)$
- Then $\partial LS/\partial w = -w^{-2} = -1/(1 + e^{-n})^2$
- Then $\partial w/\partial n = -e^{-n}$
- Thus $\partial LS/\partial n = e^{-n}/(1 + e^{-n})^2 = (1-LS)LS$ because $1 - LS = e^{-n}/(1 + e^{-n})$

Sanity checking $\partial LS(n)/\partial n = (1 - LS(n))LS(n)$



Introducing sensitivities

- Recall $n' = w'_1a_1 + w'_2a_2 + b'$
- So $\partial f/\partial w'_1 = (\partial f/\partial n')(\partial n'/\partial w'_1) = (\partial f/\partial n')a_1$
- Likewise $\partial f/\partial w'_2 = (\partial f/\partial n')a_2$
- So $\partial f/\partial b' = (\partial f/\partial n')(\partial n'/\partial b) = (\partial f/\partial n')1$
- Define sensitivity $s' = \partial f/\partial n'$
- Likewise $s_1 = \partial f/\partial n_1$ and $s_2 = \partial f/\partial n_2$

Refactored updates

- Update second layer:
  - $w'_1 := w'_1 - \alpha s'a_1$
  - $w'_2 := w'_2 - \alpha s'a_2$
  - $b' := b' - \alpha s'$
- Update first neuron of first layer:
  - $w_1 := w_1 - \alpha s_1 p$
  - $b_1 := b_1 - \alpha s_1$
- Update second neuron of first layer:
  - $w_2 := w_2 - \alpha s_2 p$
  - $b_2 := b_2 - \alpha s_2$

Bootstrapping s' = ∂f/∂n'

- Consider output a' and label t
- Recall a' = Id(n') so f = $(t - a')^2 = (t - n')^2$

- Put w = (t – n') so f = w^2
- Then ∂f/∂n' = (∂f/∂w)(∂w/∂n')
- But ∂f/∂w = 2w = 2(t – n')
- But ∂w/∂n' = -1
- So s' = ∂f/∂n' = (∂f/∂w)(∂w/∂n') = -2(t – n')

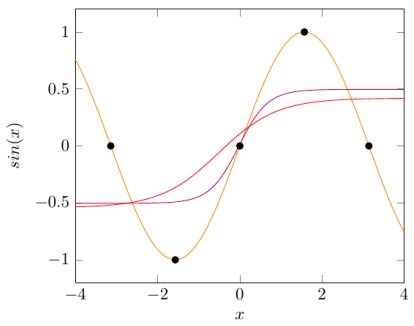
Back-propagating s_1 and s_2

- Recall $n' = w'_1 a_1 + w'_2 a_2 + b'$
- Hence $n' = w'_1 LS(n_1) + w'_2 LS(n_2) + b'$

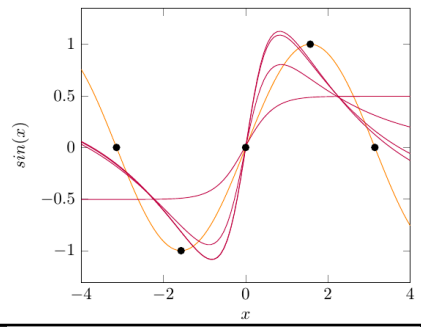
- Thus $s_1 = \partial f/\partial n_1 = (\partial f/\partial n')(\partial n'/\partial n_1)$
- But $\partial n'/\partial n_1 = w'_1 \partial LS(n_1)/\partial n_1$
- And recall s' = ∂f/∂n'
- So $s_1 = \partial f/\partial n_1 = s'w'_1(1 - LS(n_1))LS(n_1)$

- Likewise $s_2 = s'w'_2(1 - LS(n_2))LS(n_2)$

Orange = target; red = intermediate; purple = final



Different seed weights



Summary

- Multilayer nets are more expressive
- Training uses steepest descent; illustrates supervised learning
- Backpropagation of sensitivities
- But convergence can be slow and depends on seed weights