Connectionism: Unit 3

Backpropagation

Weight Adjustment using the Technique of Gradient Descent

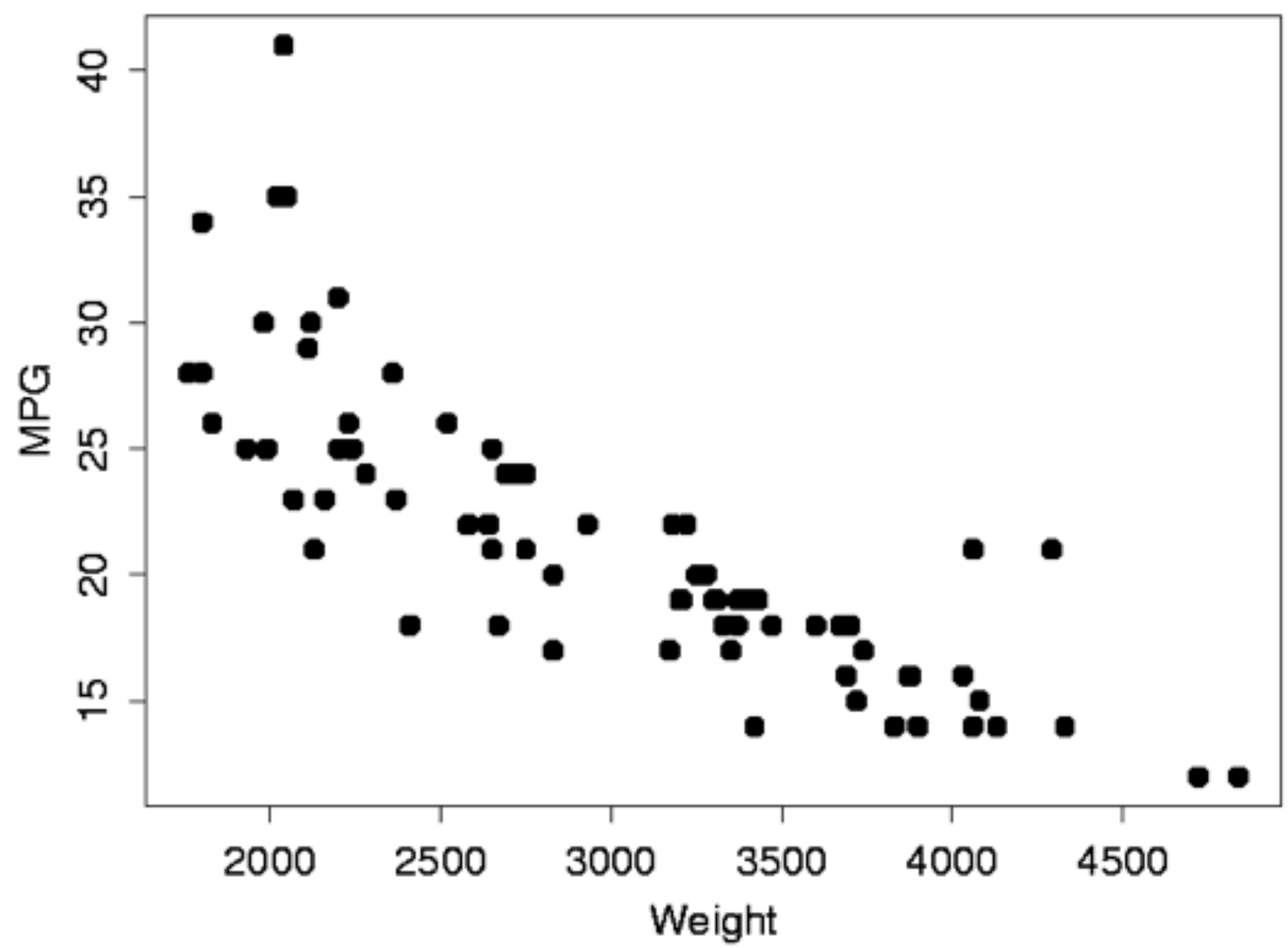
Fitting a model to data

There is a relationship evident between the two variables here (sadly, thoroughly non-cognitive).

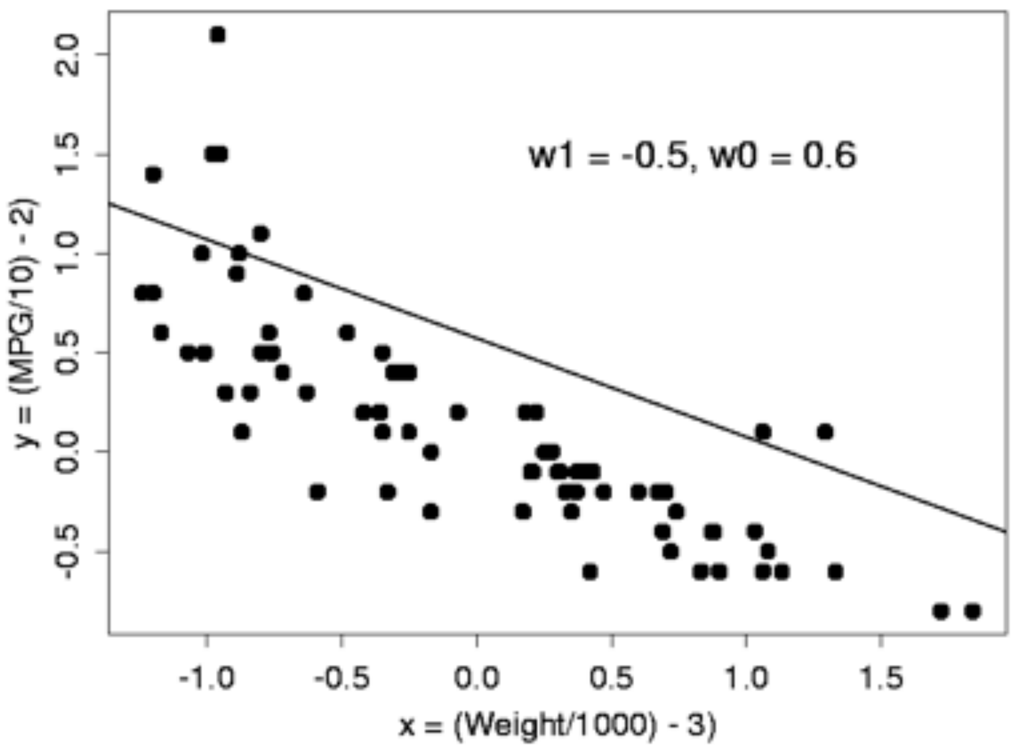
We could model that relationship using a linear regression:

$$y = w_1 x + w_0$$

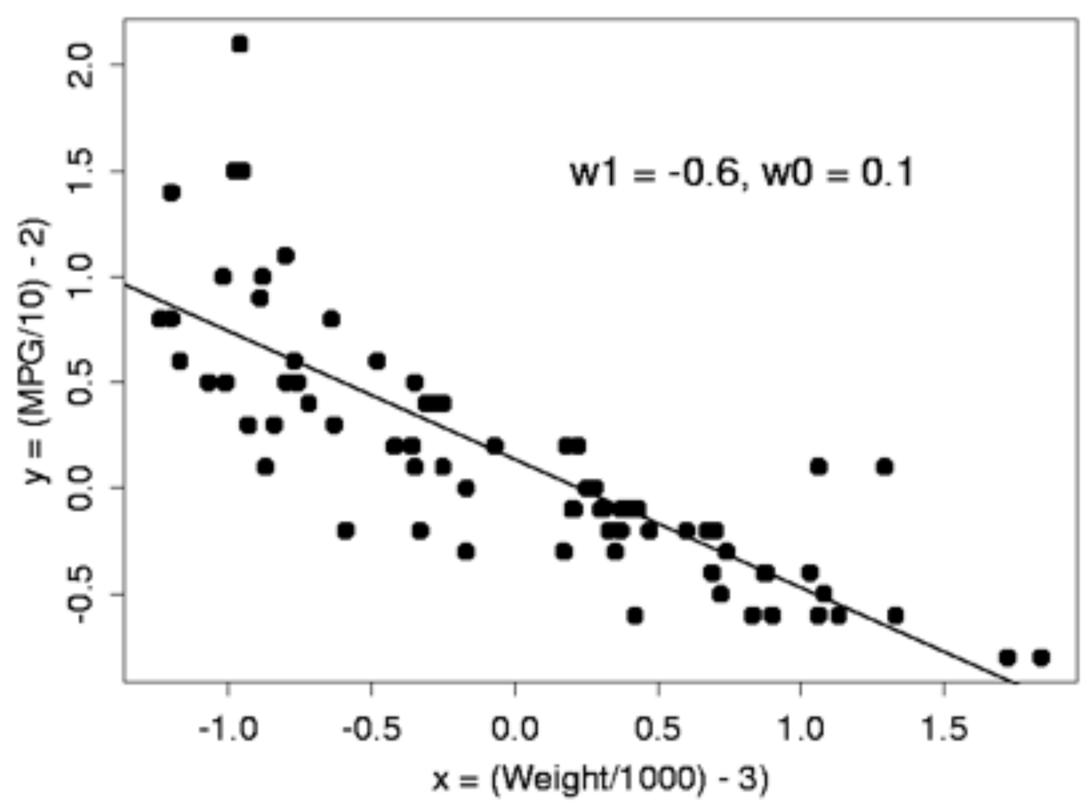
Which line is the best?



This?



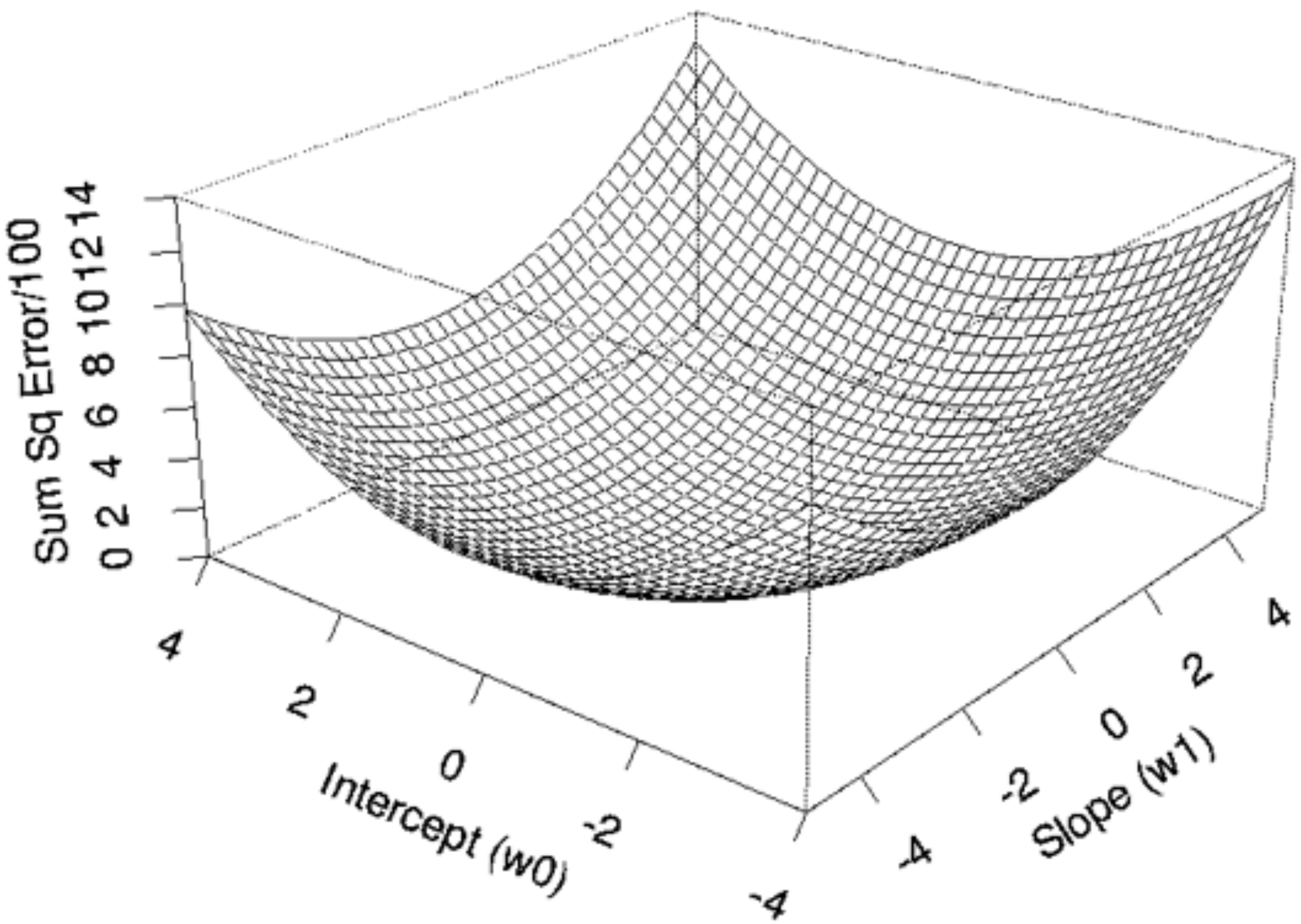
w1 = -0.5, w0 = 0.6
y = (MPG/10) - 2
x = (Weight/1000) - 3



w1 = -0.6, w0 = 0.1
y = (MPG/10) - 2
x = (Weight/1000) - 3

This?

Choosing among models in parameter space

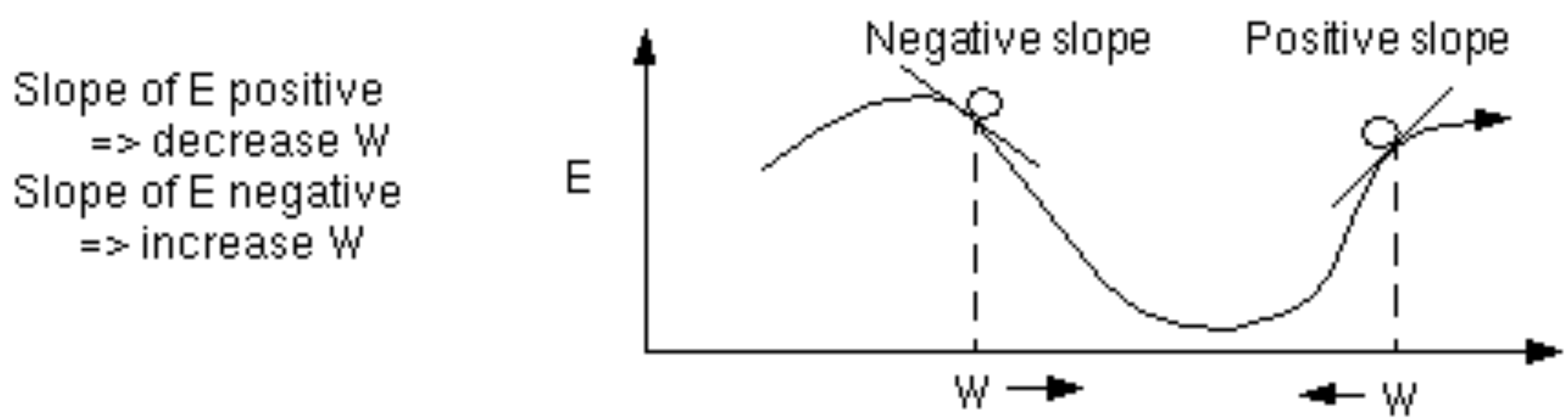


$$E = \frac{1}{2}\sum_{p}(t_p - y_p)^2$$

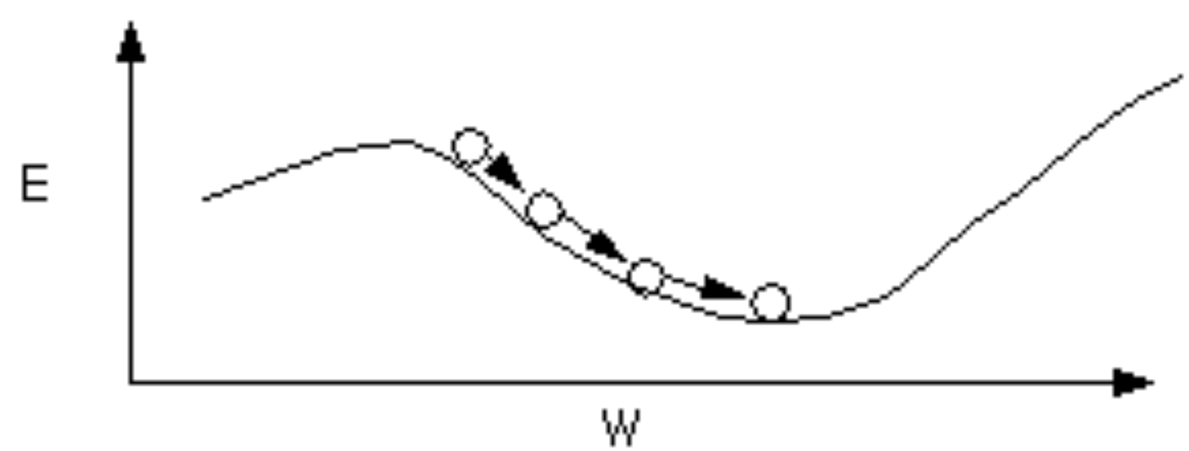
The sum over all points p in our data set of the squared difference between the target value t_p (here: actual fuel consumption) and the model's prediction y_p, calculated from the input value x_p (here: weight of the car) by equation 1.

Gradient Descent Algorithm

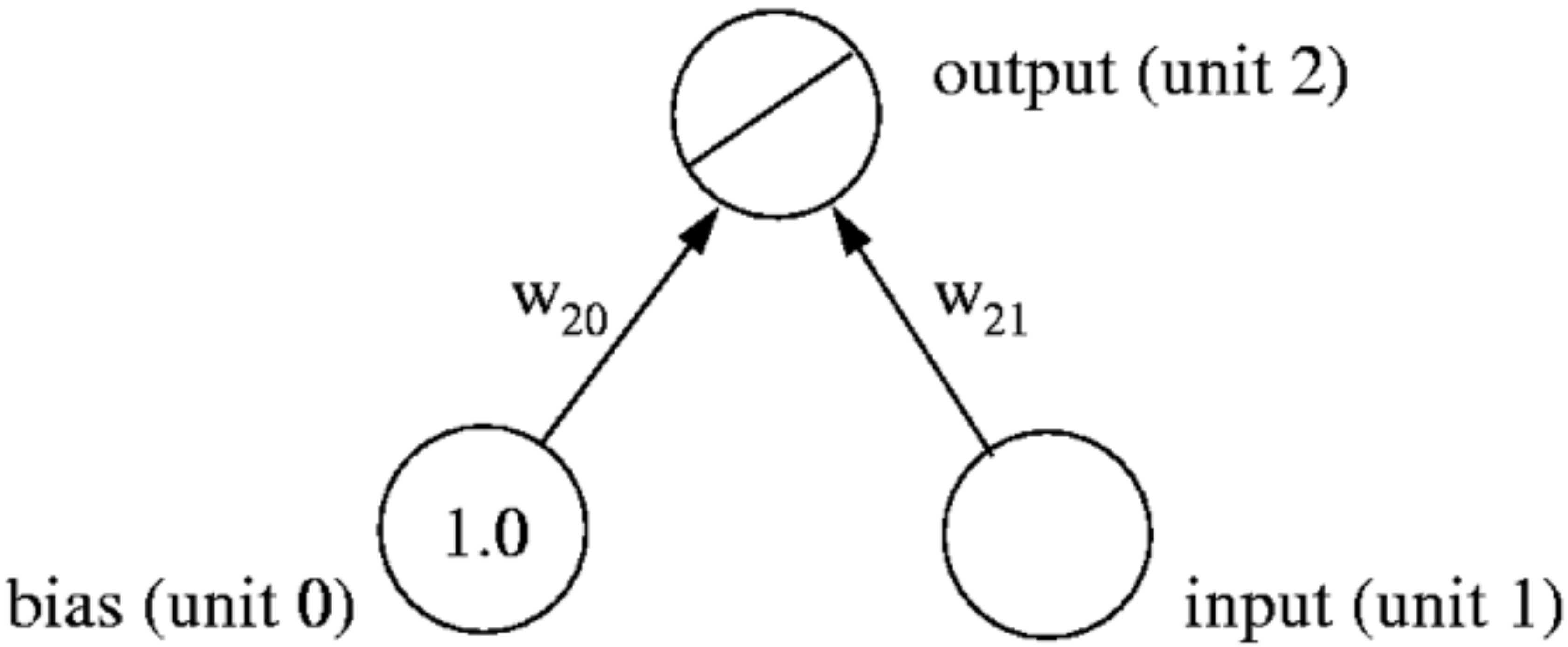
1. Choose some (random) initial values for the model parameters.
2. Calculate the gradient G of the error function with respect to each model parameter.
3. Change the model parameters so that we move a short distance in the direction of the greatest rate of decrease of the error, i.e., in the direction of −G.
4. Repeat steps 2 and 3 until G gets close to zero.



Gradient Descent contd.



It's a neural network!



$$y = w_{21}x + w_{20}$$

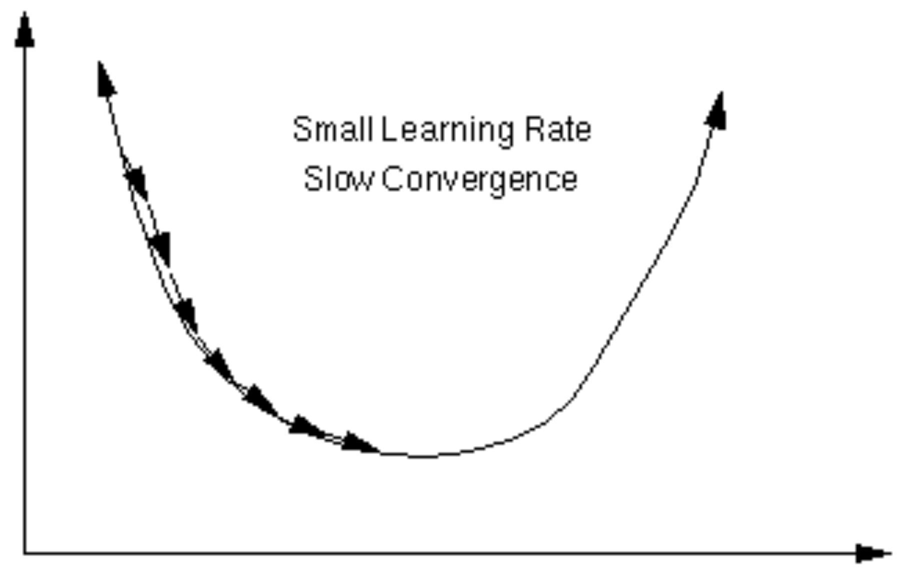
Gradient Descent for 2-layer linear networks

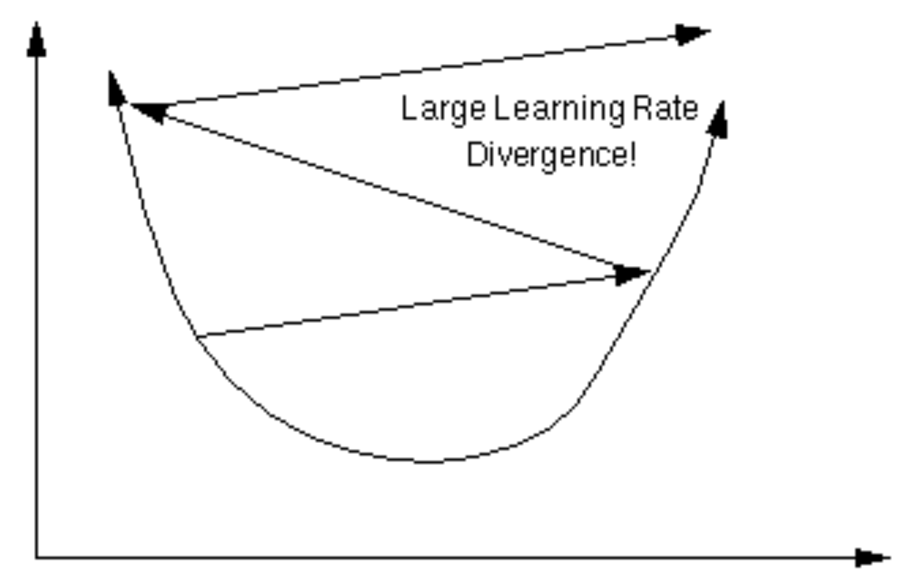
1. Initialize all weights to small random values.
2. REPEAT until done
    1. For each weight w_{ij} set $\Delta w_{ij}=0$
    2. For each data point $(x, t)^p$
        1. set input units to x
        2. compute output values

$$\Delta w_{ij} = \Delta w_{ij} + (t_i - y_i) y_j$$

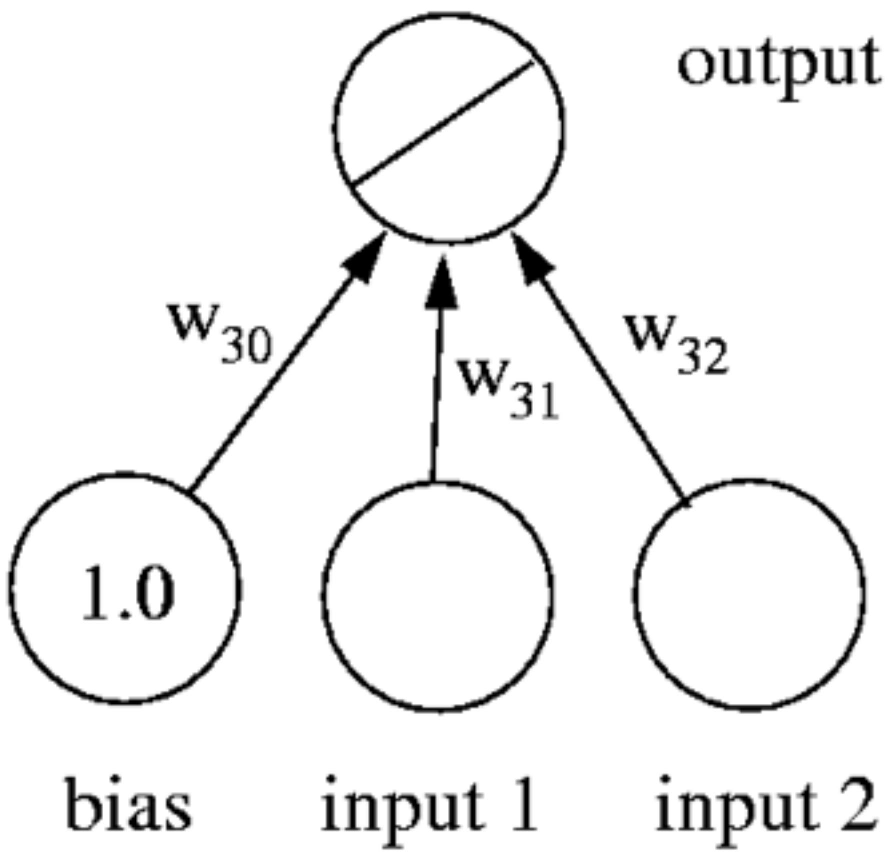
3. For each weight w_{ij} set $$w_{ij}(t) = w_{ij}(t) + \eta \Delta w_{ij}$$

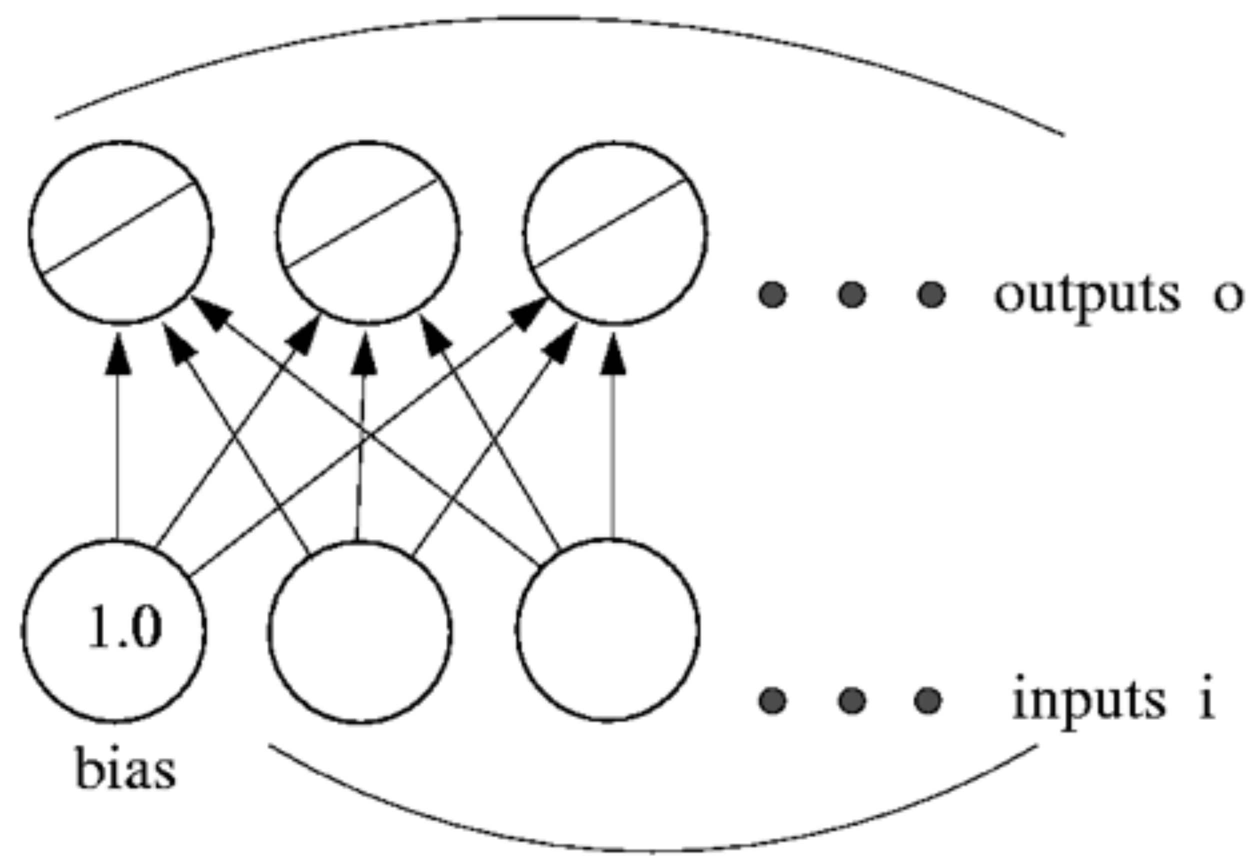
Learning rate





Multiple regression



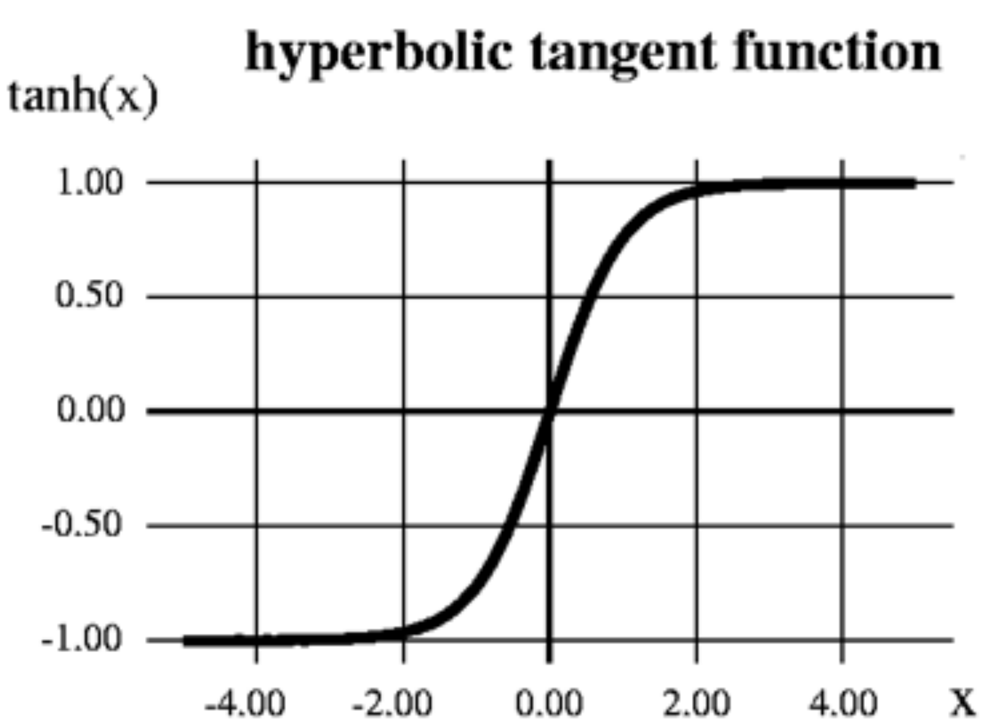


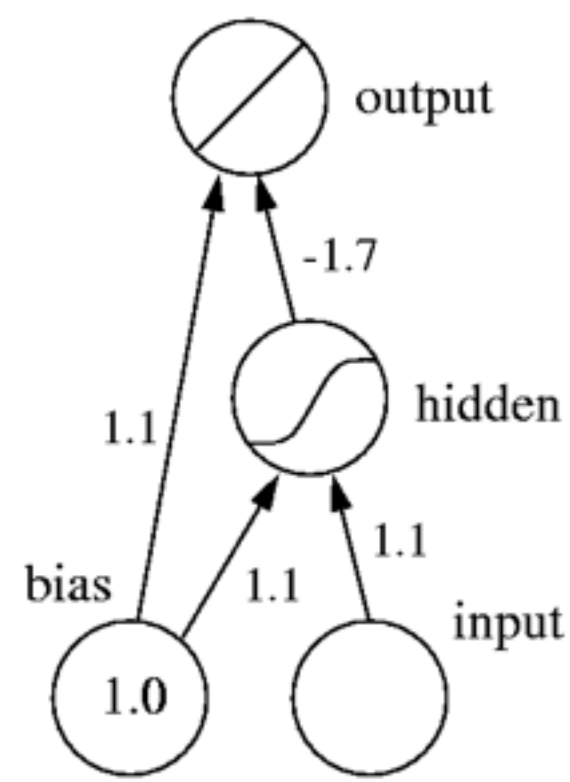
Activation functions

Networks do more than fit straight lines to data.

But to do that, we need to make the activation of a unit be some non-linear function of its net input.

Popular activation functions are the logistic function and the tanh function.

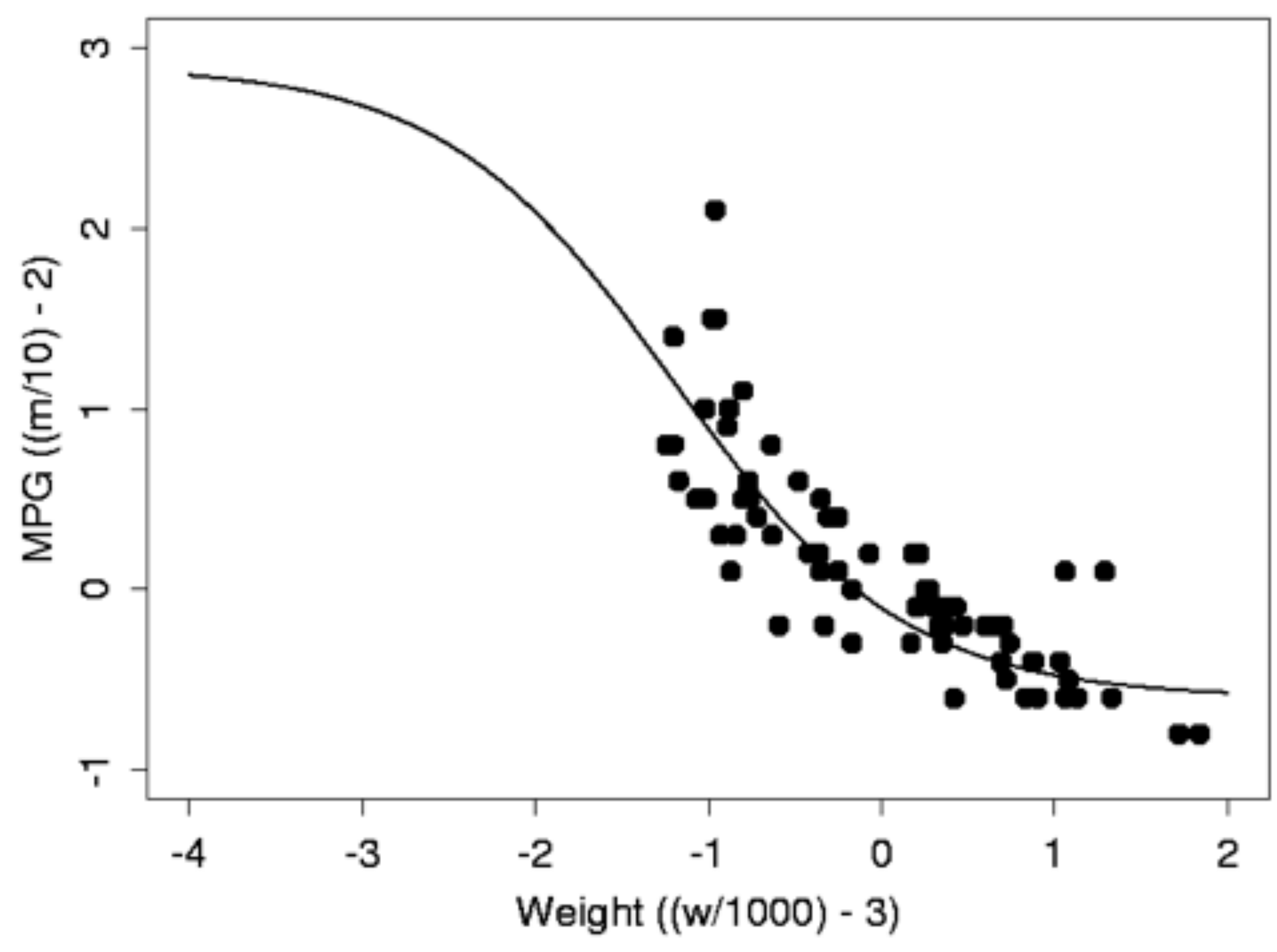




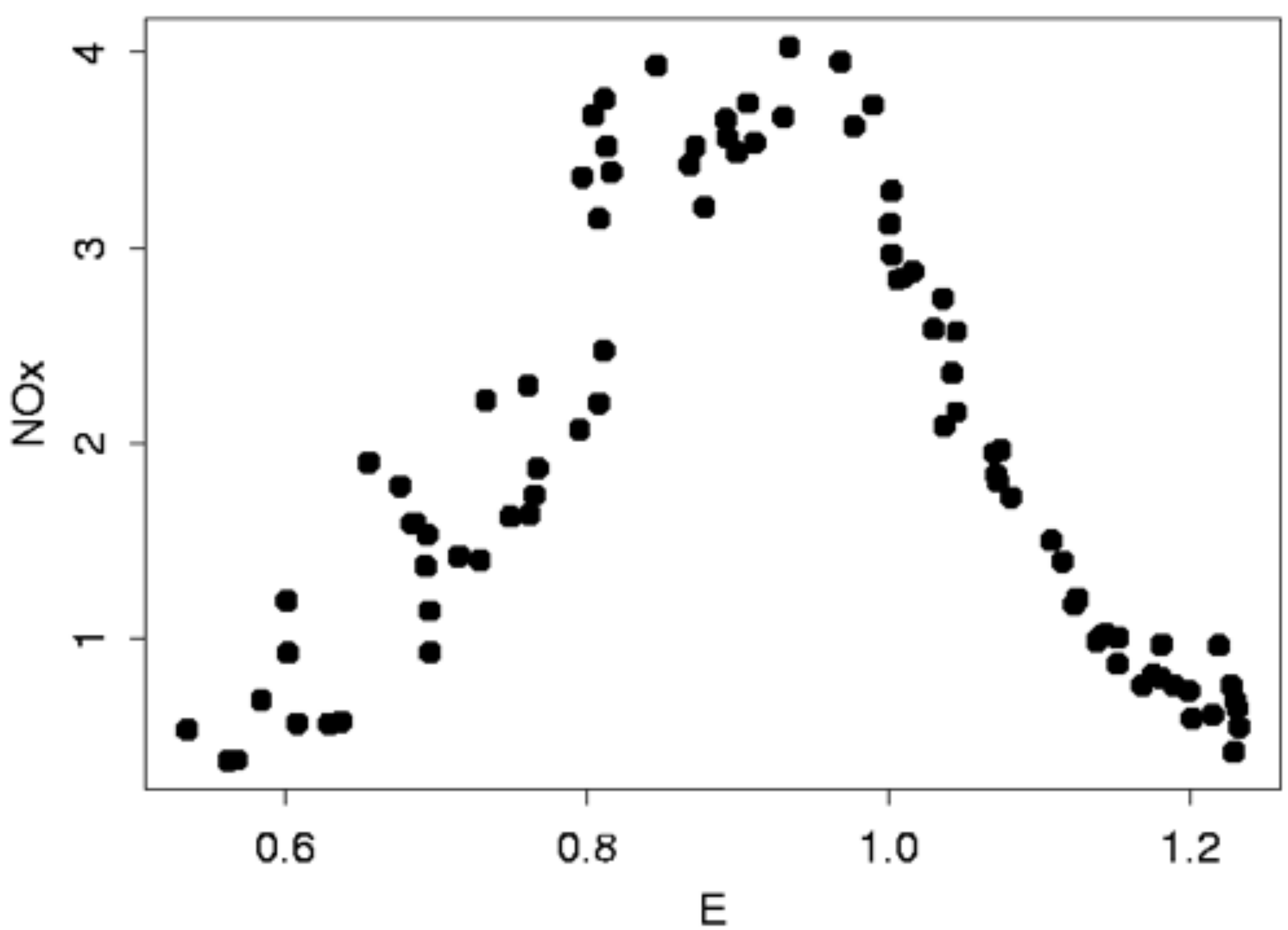
What is the activation of the output unit in this network?

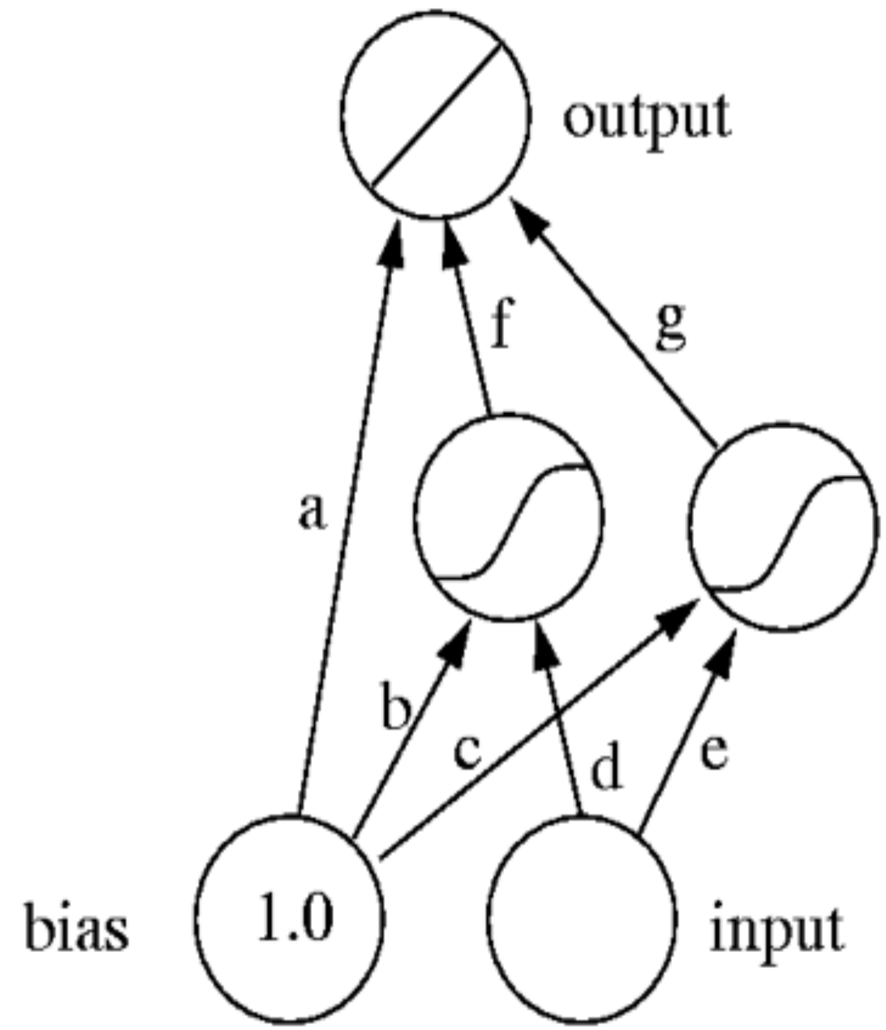
A better fit.....

$$y = w_{32}\tanh(w_{21}x + w_{20}) + w_{30}$$

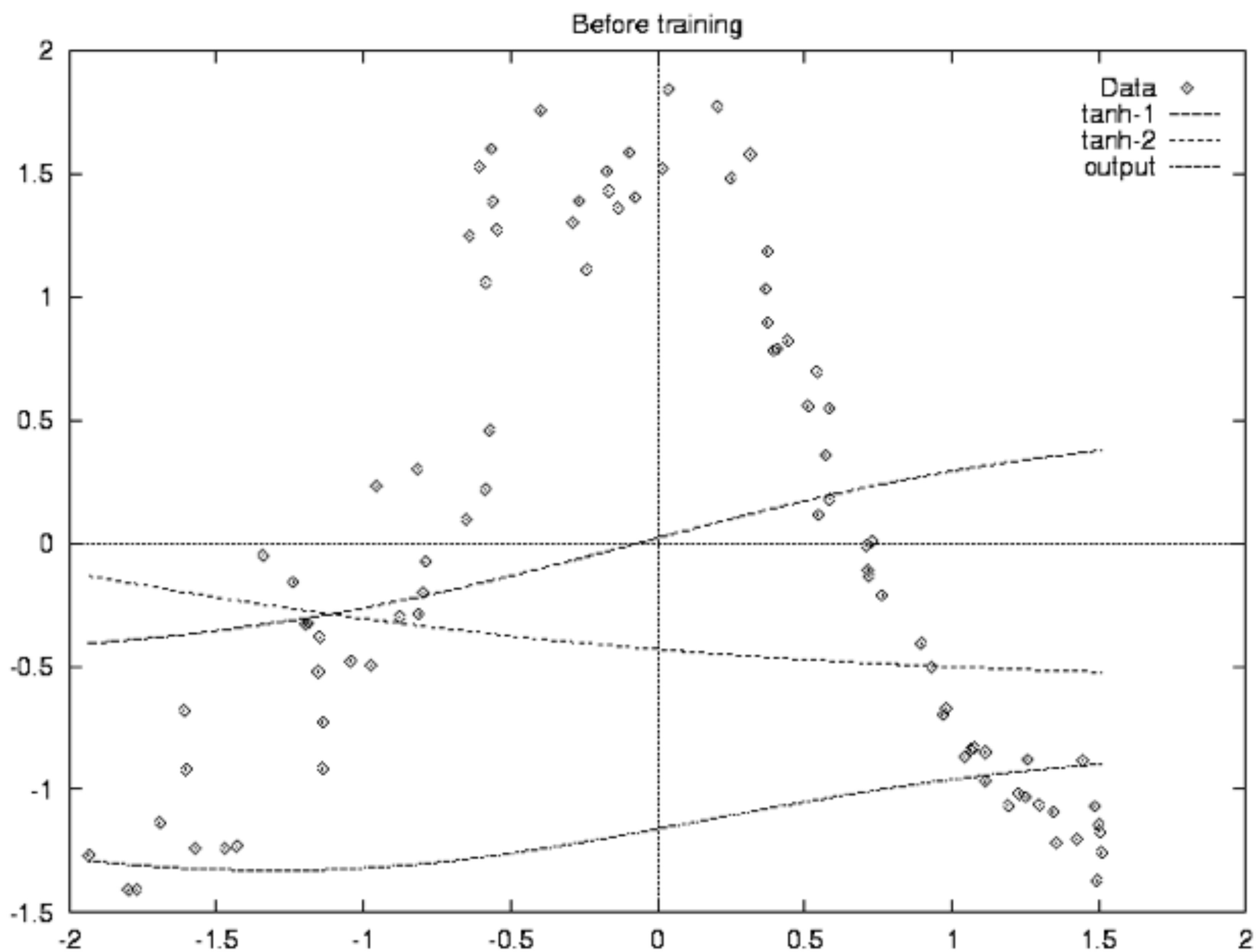


Another data set

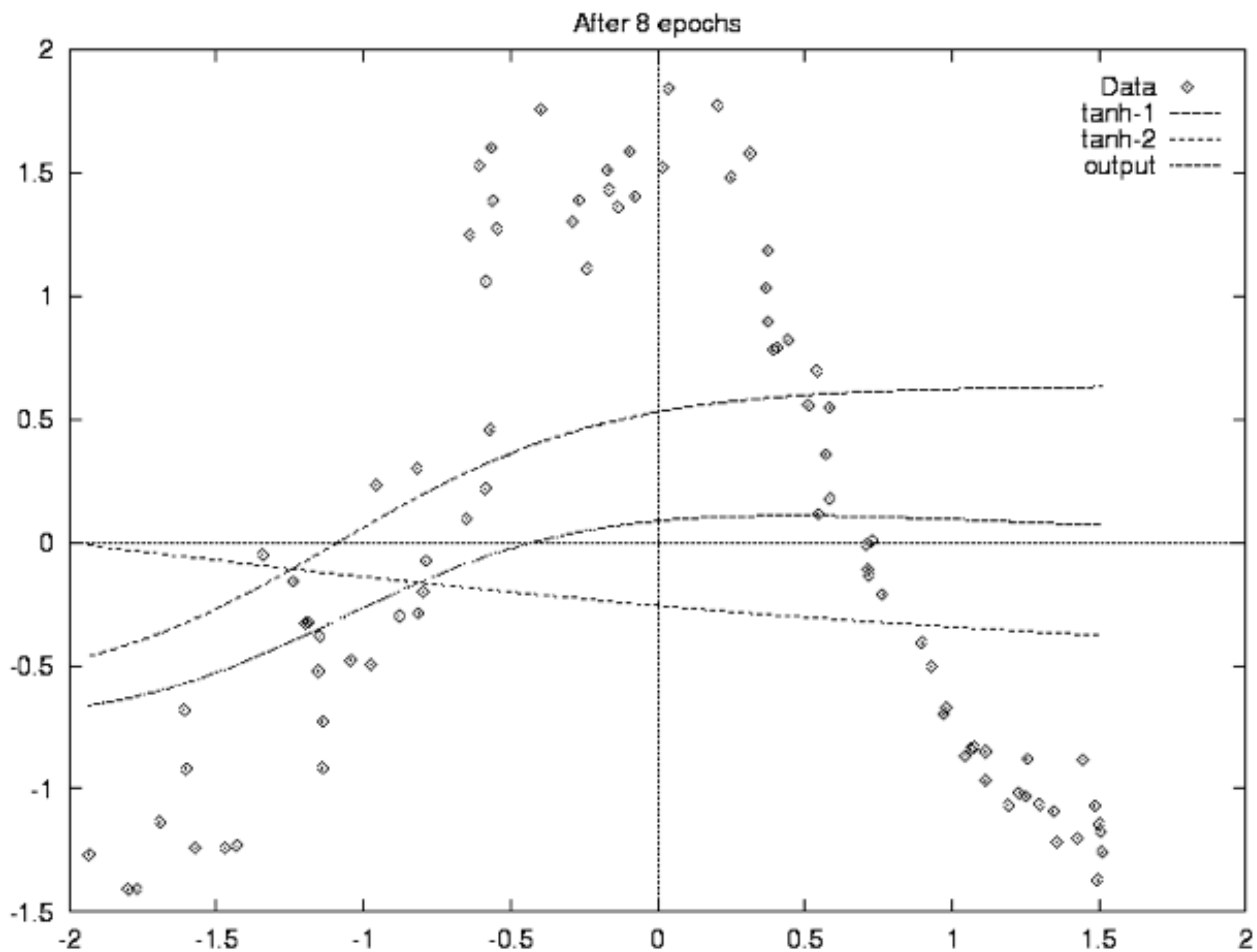




Before training

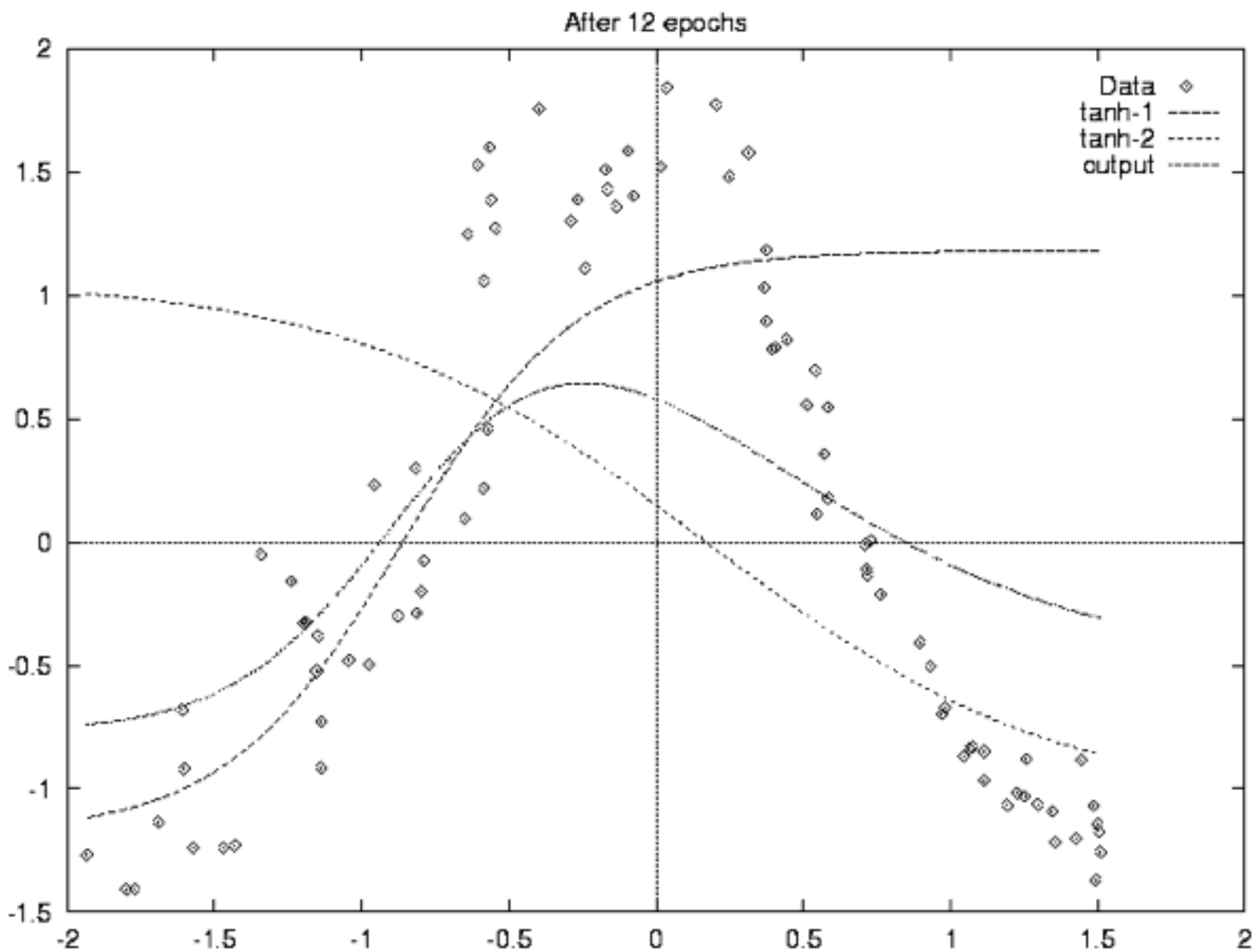


..then....



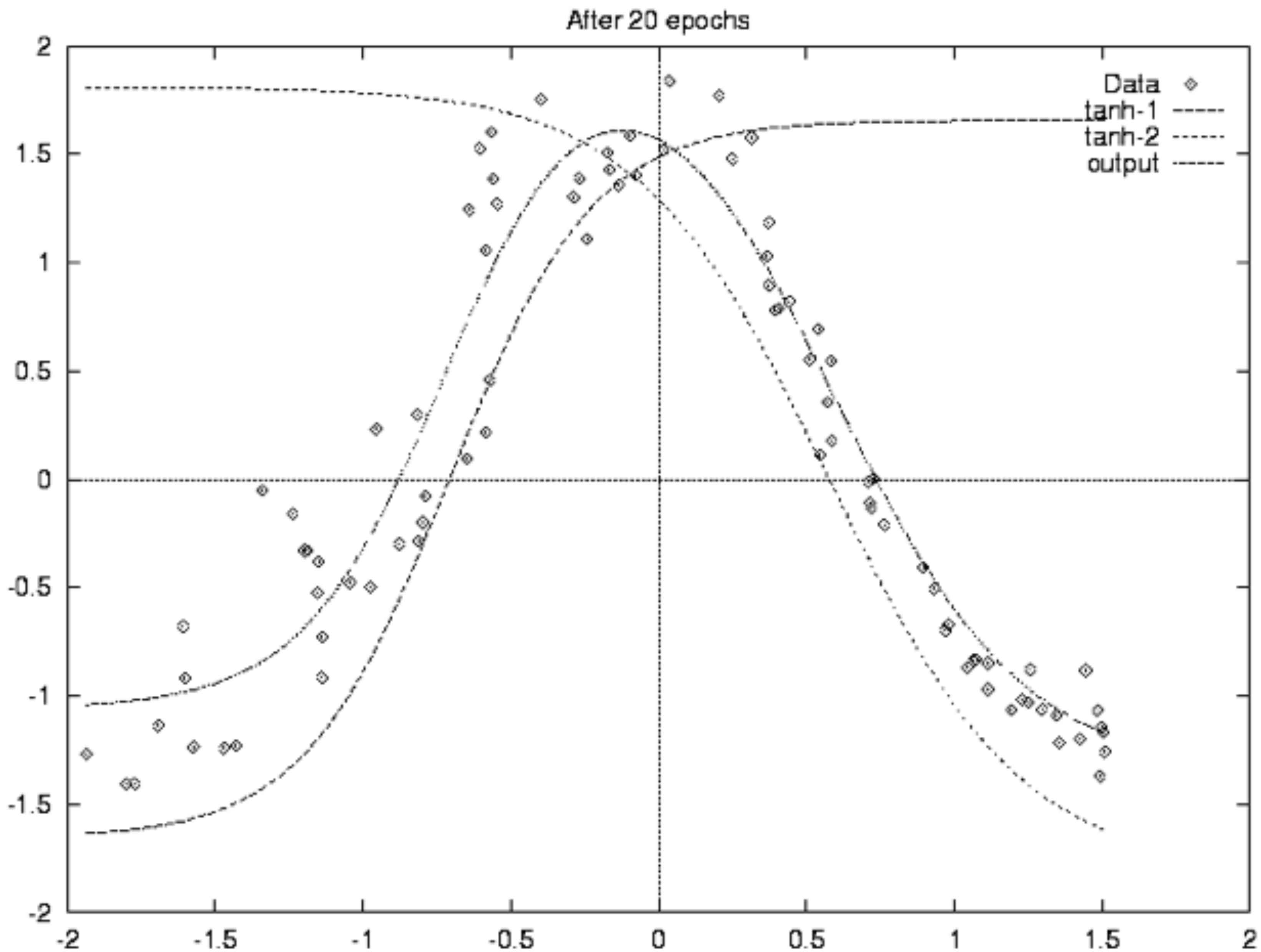
After 8 epochs
Data
tanh-1
tanh-2
output
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

...later....



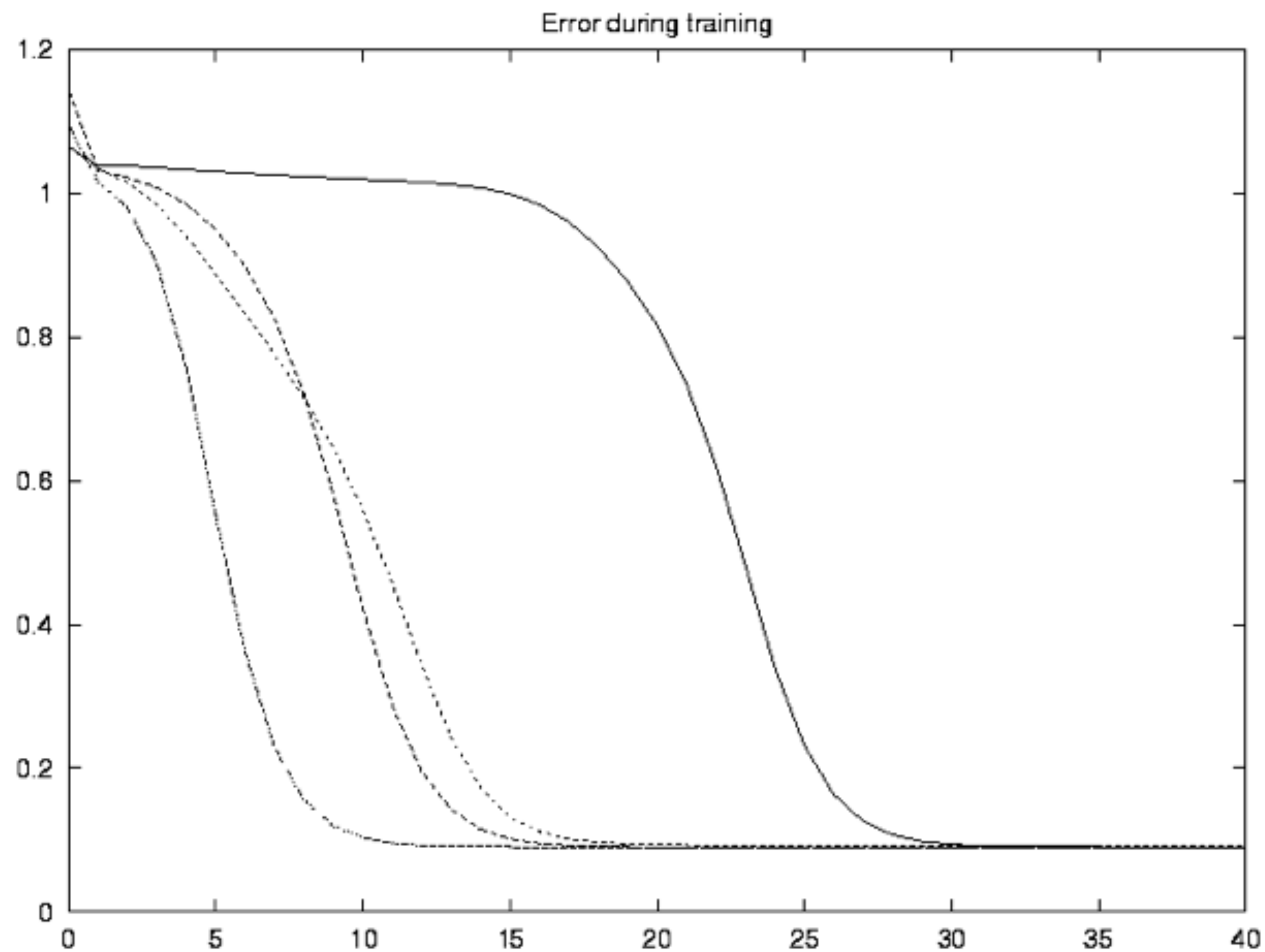
After 12 epochs
Data
tanh-1
tanh-2
output
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

...and finally....



After 20 epochs
Data
tanh-1
tanh-2
output
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

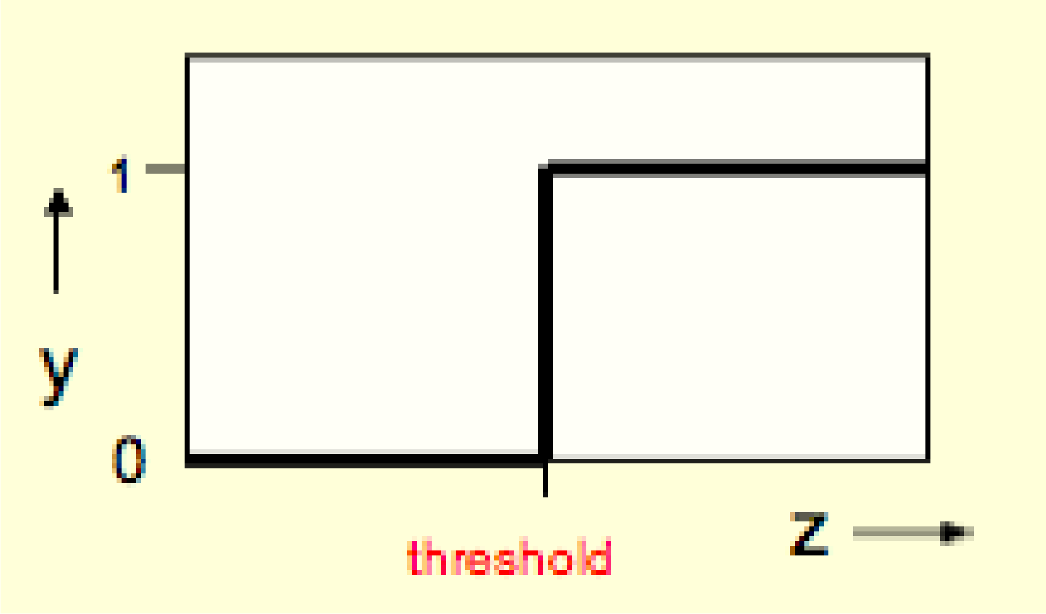
Error during training



Types of non-linear units

Binary Threshold Neurons (e.g. Perceptron output units)

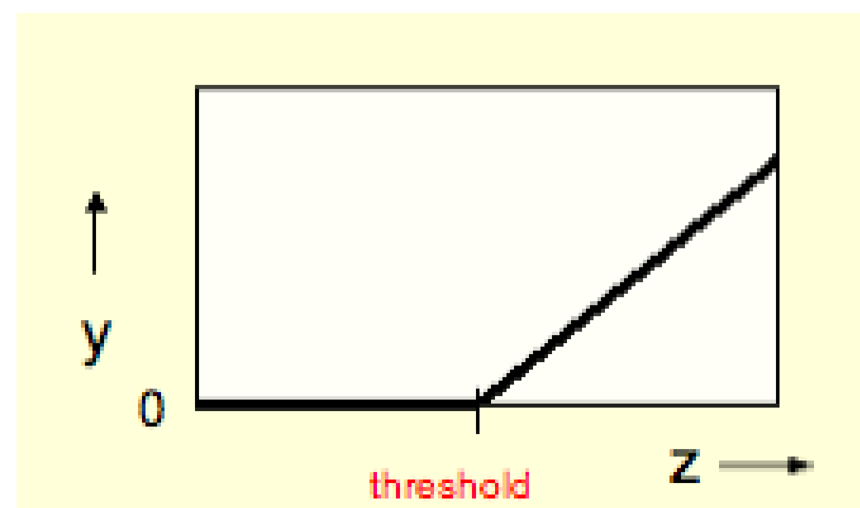
- Inspired by the 'all-or-nothing' character of neural firing
- Net input: $$net_i = \sum_i x_i w_i$$
- Output: $$y_i = \begin{cases} 1 & if z_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$



Rectified Linear Units

- Compute a *linear* weighted sum of inputs
- Output a *non-linear* function of the input

$$y_i = \begin{cases} z_i & if\, z_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

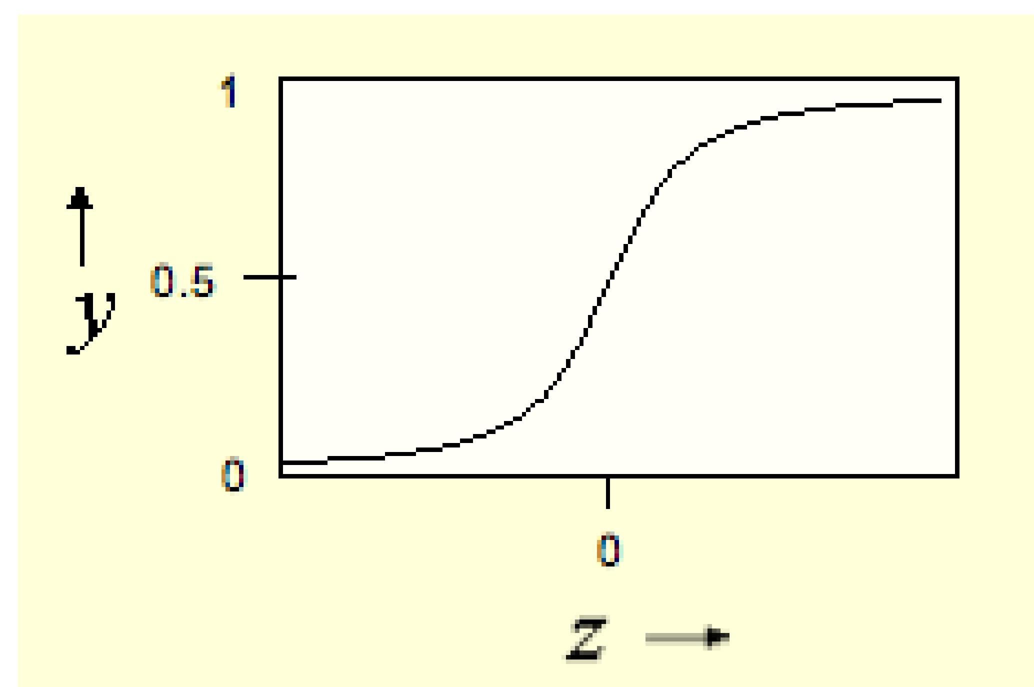


Surprisingly popular choice in modern deep learning networks

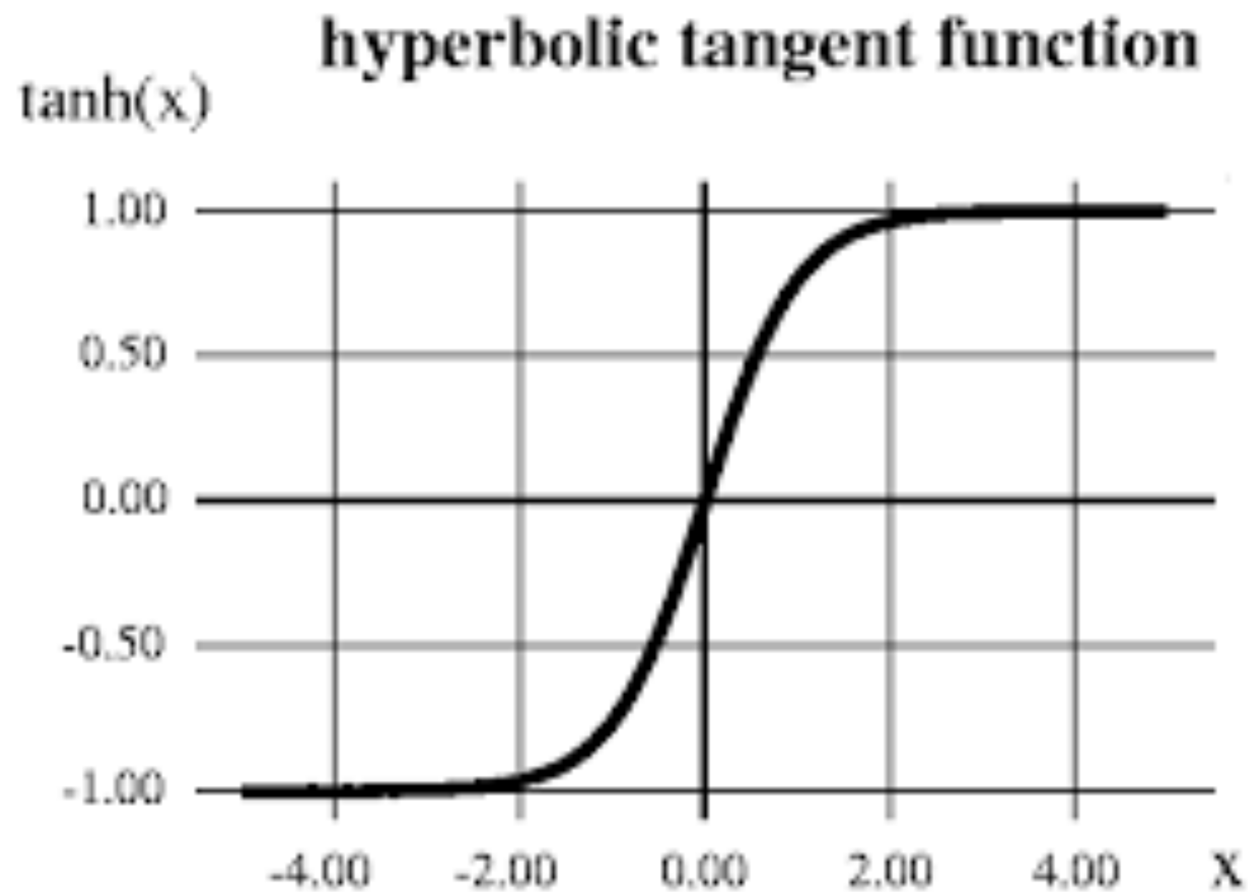
Logistic Sigmoidal Units

$$y_i = \frac{1}{1 + e^{-\mathrm{net}_i}}$$

- Real-valued output
- Smooth, continuous, bounded
- Nice derivatives
- Very (excessively?) common
- Range: 0..1



Tanh Units



- Similar to logistic units
- Range: -1..1 (symmetrical about 0)

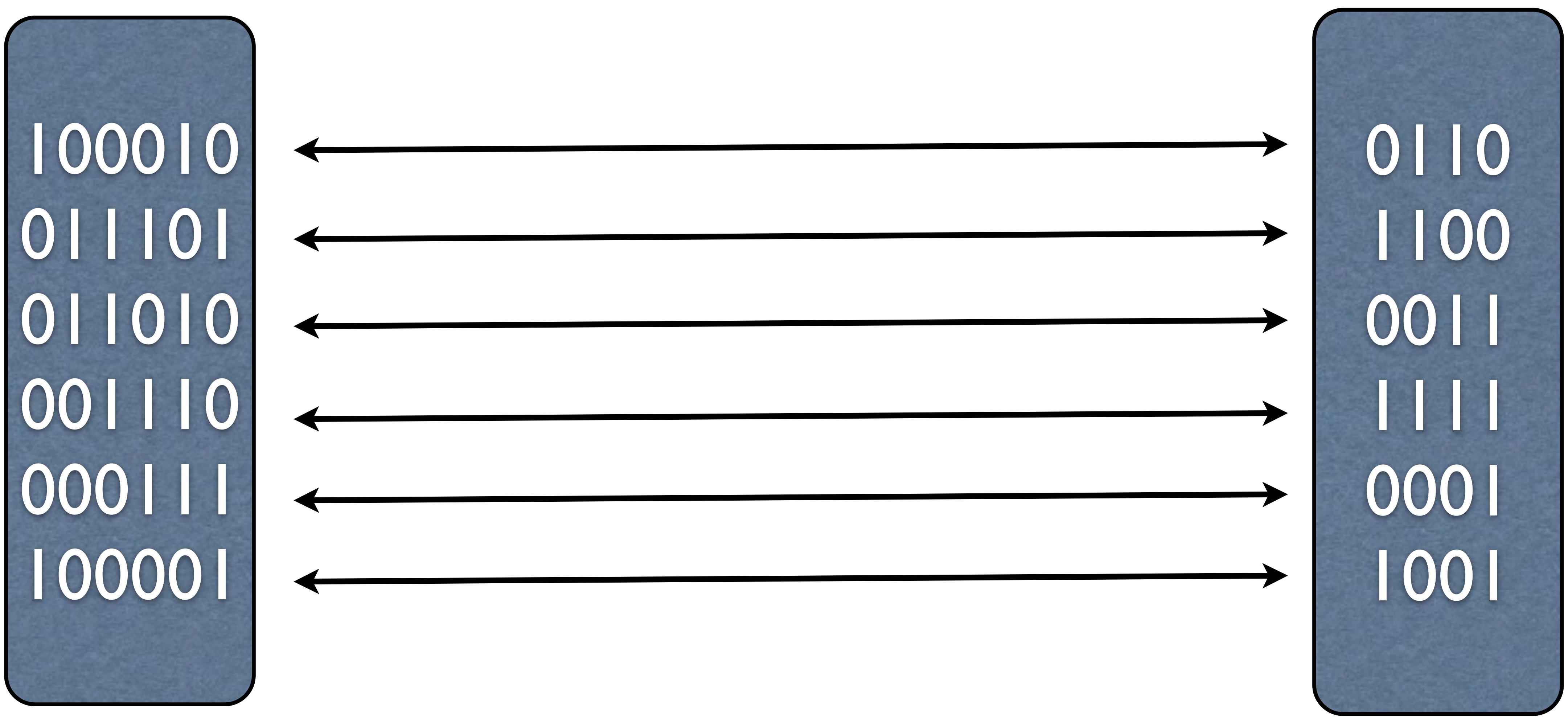
Back Propagation of Error

(Backprop)

A trained network has learnt a **mapping**

Input patterns

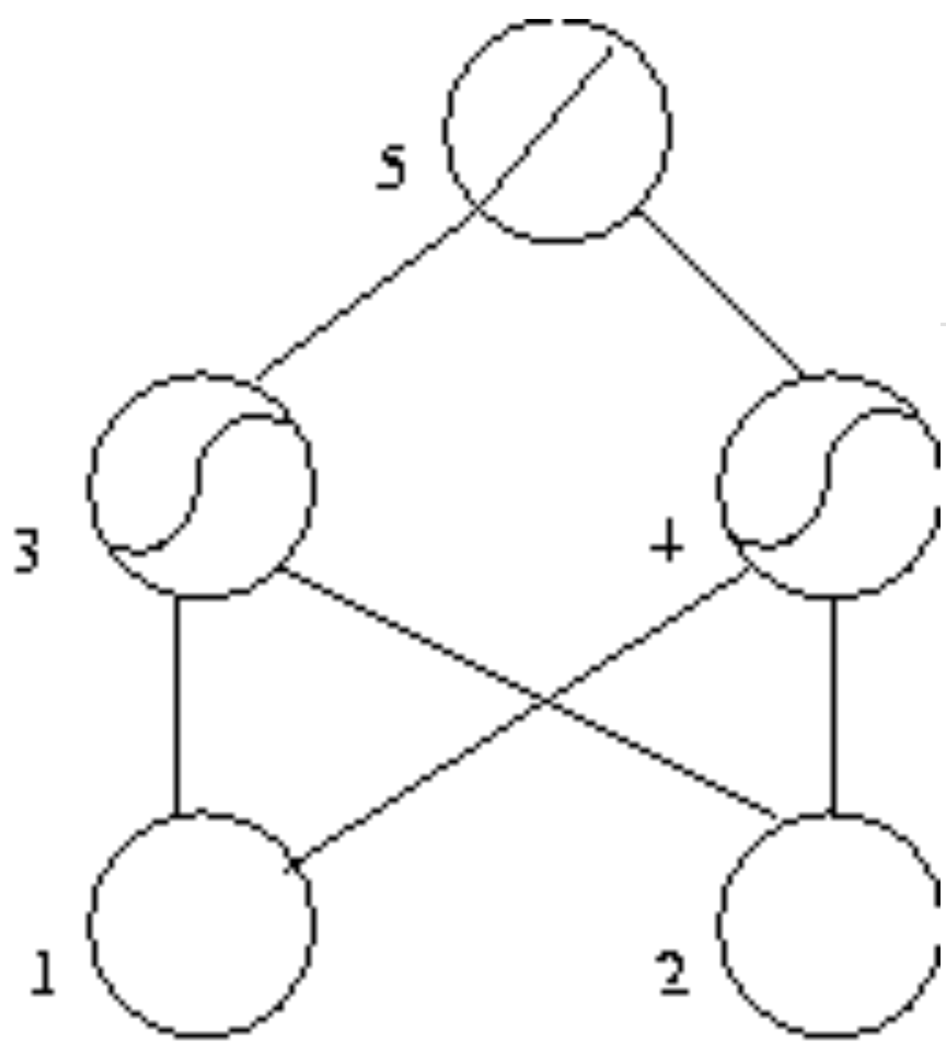
Output patterns



The **mapping** is a mathematical function, whose **parameters** are the **weights**

y = f(x)

$$
\begin{aligned}
y \;=\; & \tanh(x_2 w_{32} + x_1 w_{31} + w_{30}) w_{53} + \\
& \tanh(x_2 w_{42} + x_1 w_{41} + w_{40}) w_{54} + \\
& w_{50}
\end{aligned}
$$



The Problem: How do we find those weights?

Solution 1: Guess Them

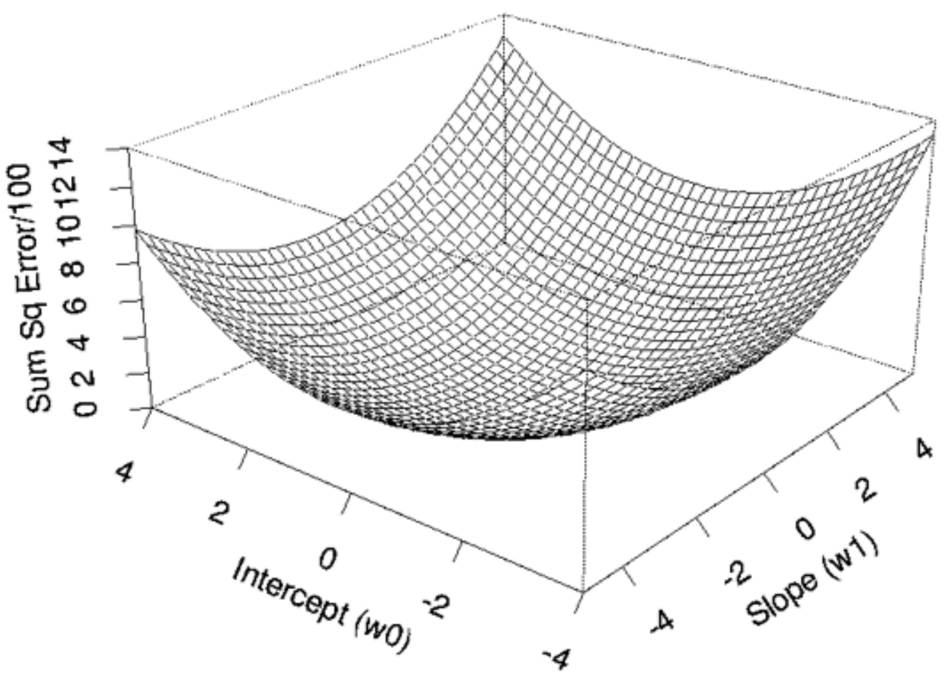
Not as daft as you might think!!

J. Schmidhuber, S. Hochreiter, Y. Bengio. Evaluating benchmark problems by random guessing. In S. C. Kremer and J. F. Kolen, eds., *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE press, 2001.

J. Schmidhuber and S. Hochreiter. Guessing can outperform many long time lag algorithms. Technical Note IDSIA-19-96, IDSIA, May 1996

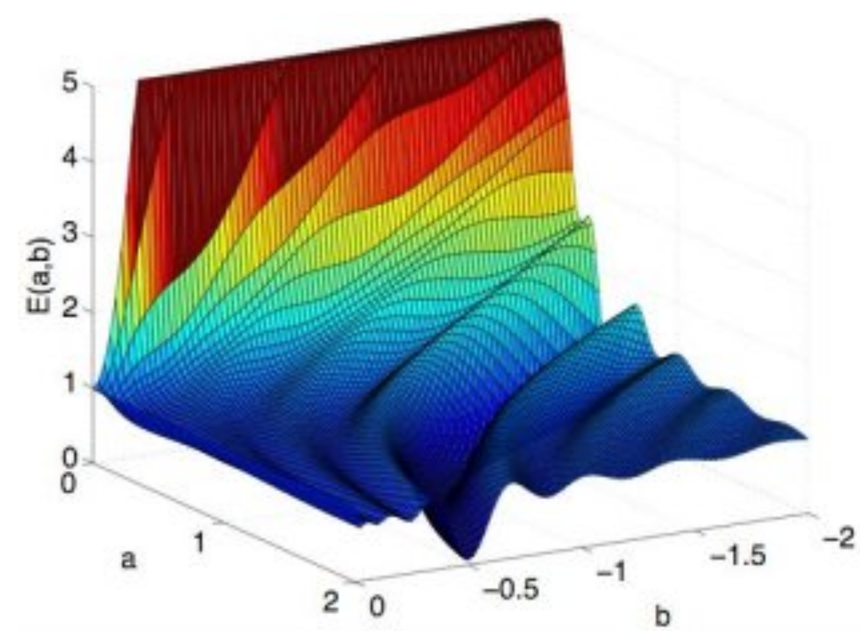
Requires fast machines and small problems.
Not a serious contender for serious problems.

Learning as Gradient Descent



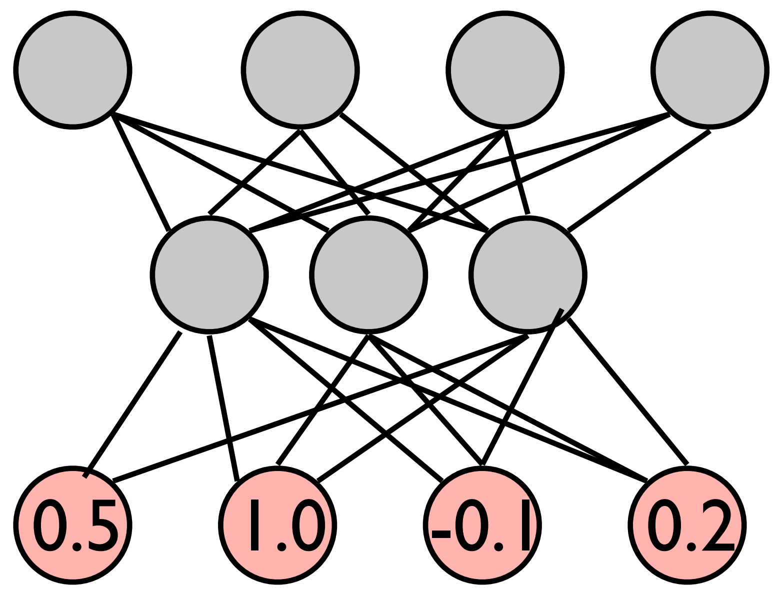
Error surface for a 2-wt, linear network

Complex error surface for hypothetical network training problem



Reminder

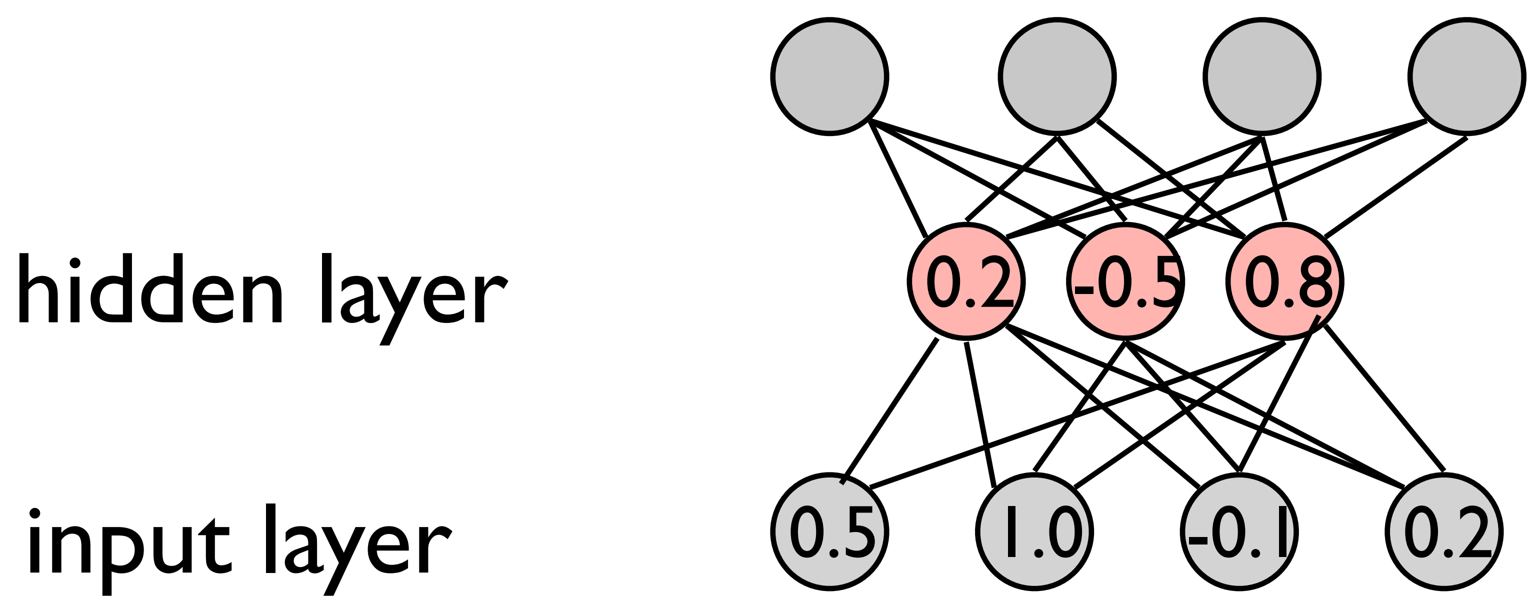
Mapping from input to output



input layer

Input pattern: <0.5, 1.0,-0.1,0.2>

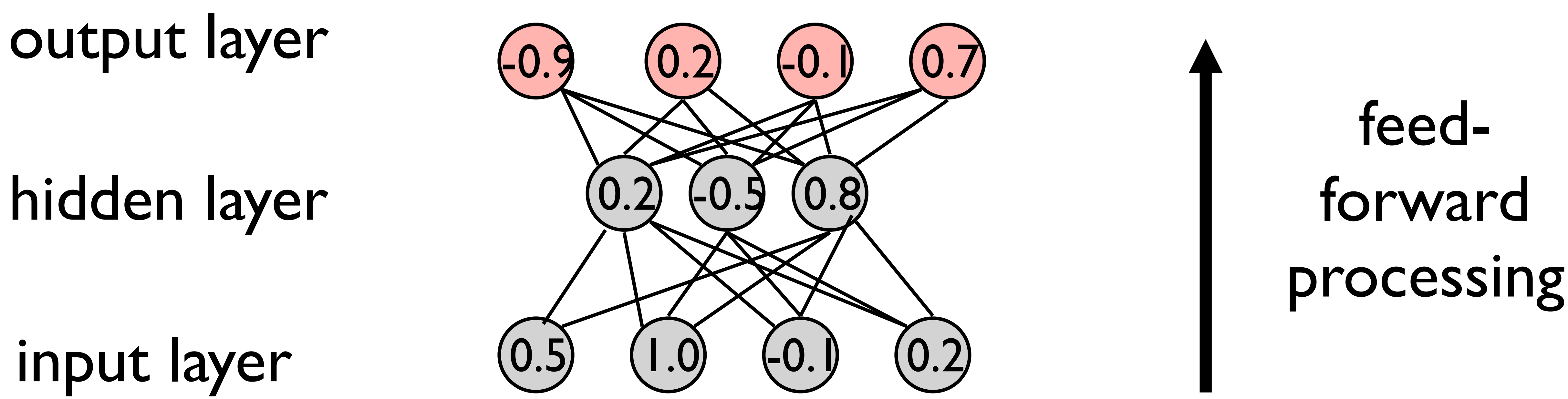
Mapping from input to output



Input pattern: <0.5, 1.0,-0.1,0.2>

Mapping from input to output

Output pattern: <-0.9, 0.2,-0.1,0.7>



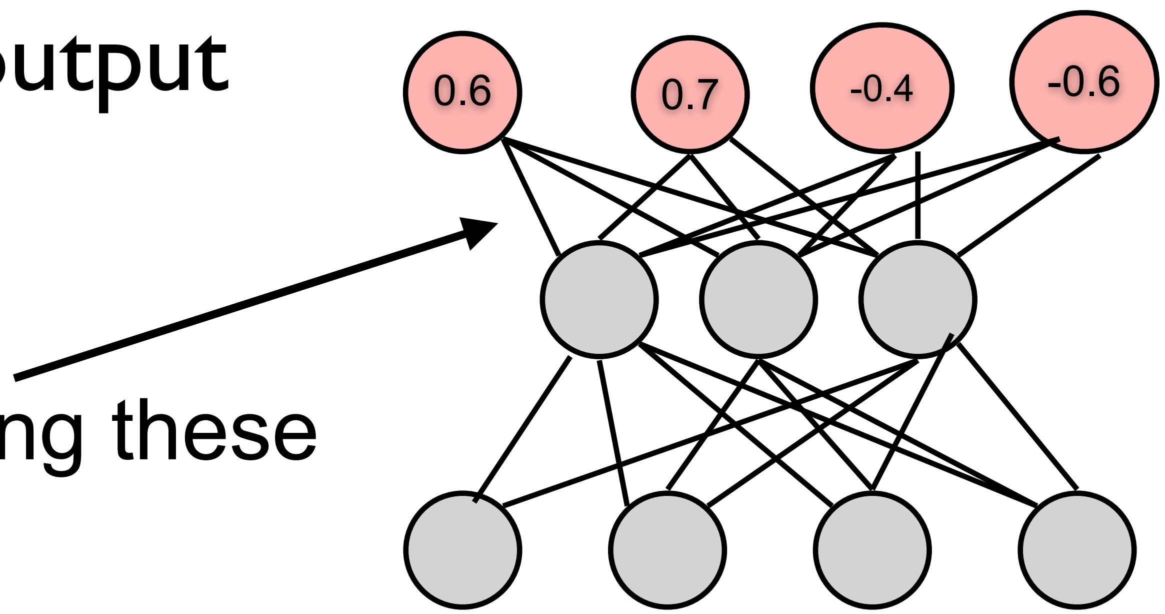
Input pattern: <0.5, 1.0,-0.1,0.2>

Calculating error

Target pattern: **<**-0.3, 0.9,-0.5,0.1**>**

Output pattern: <-0.9, 0.2, -0.1, 0.7>

errors at output



for changing these

Back Propagation of error

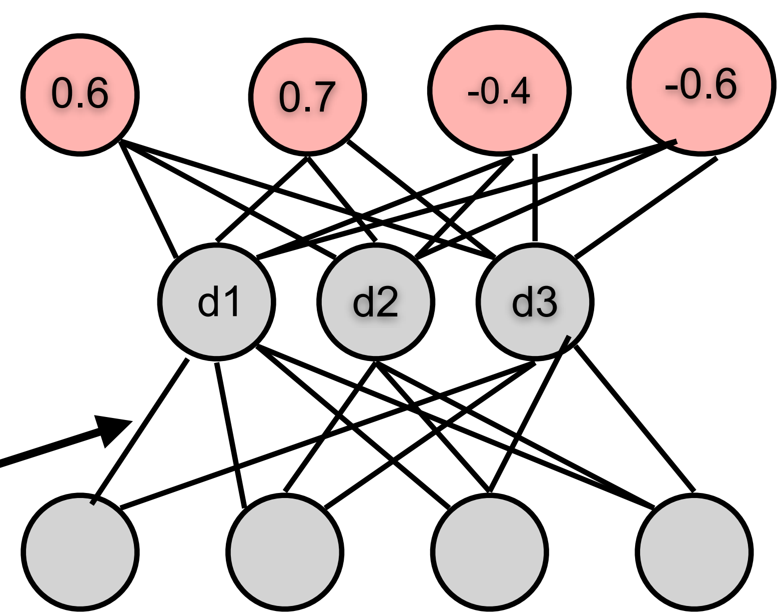
Output errors are used to compute changes for weights from hidden to outputs

Calculating error

Target pattern: <-0.3, 0.9,-0.5,0.1>

Output pattern: <-0.9, 0.2, -0.1, 0.7>

Generalised errors at hidden layer



for changing these

Back Propagation of error

Generalised errors are computed for hidden notes, so that we can compute changes for weights from input to hidden

An informal account of BackProp

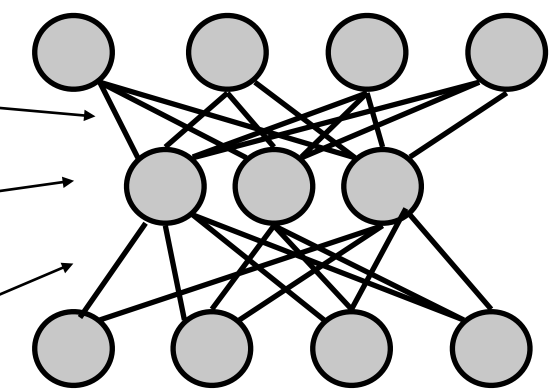
For each pattern in the training set:

Compute the error at the output nodes

Compute Δw for each wt in 2nd layer

Compute delta (generalized error expression) for hidden units

Compute Δw for each wt in 1st layer



After amassing Δw for all weights and all patterns, change each wt a little bit, as determined by the learning rate

$$\Delta w_{ij} = -\eta \delta_{ip} o_{jp}$$

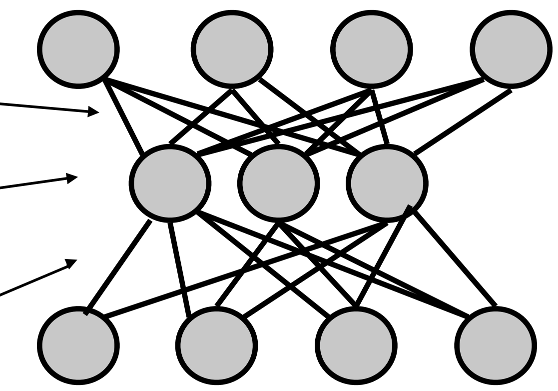
For each pattern in the training set:

Compute the error at the output nodes

Compute Δw for each wt in 2^{nd} layer

Compute delta (generalized error expression) for hidden units

Compute Δw for each wt in 1^{st} layer



Each pass through the whole set of patterns = 1 *epoch*

In classical backprop, weight changes are made at the end of the epoch

The Delta Rule

$$\Delta w_{ij} = -\eta \delta_{ip} o_{jp}$$

w_{ij} is the link to unit i from unit j (the feeding unit)

Δw_{ij}	This is the amount we would change w_{ij} based on the pattern we just presented
o_{jp}	This is the current activation of the feeding unit
δ_{ip}	This term depends on the contribution of w_{ij} to the error we are observing. Its form will depend on which weight this is, and what the activation of the
η	This is the learning rate. It scales the magnitude of the weight change, to ensure small steps.

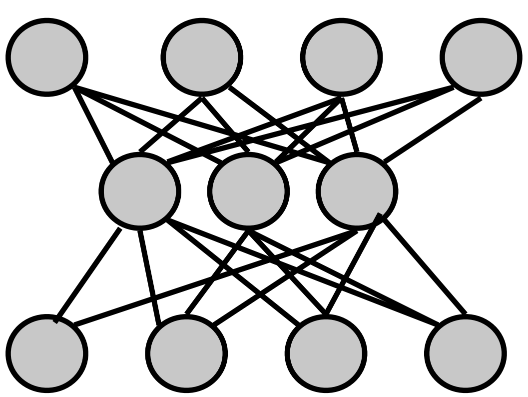
For the sake of simplicity, we will typically omit the subscript p which indexes the patterns.

Learning by Backpropagation of Error: I

We have presented a pattern *p* to the network, and it produced some (incorrect) output

-1 0 1 1 ⟵ Target

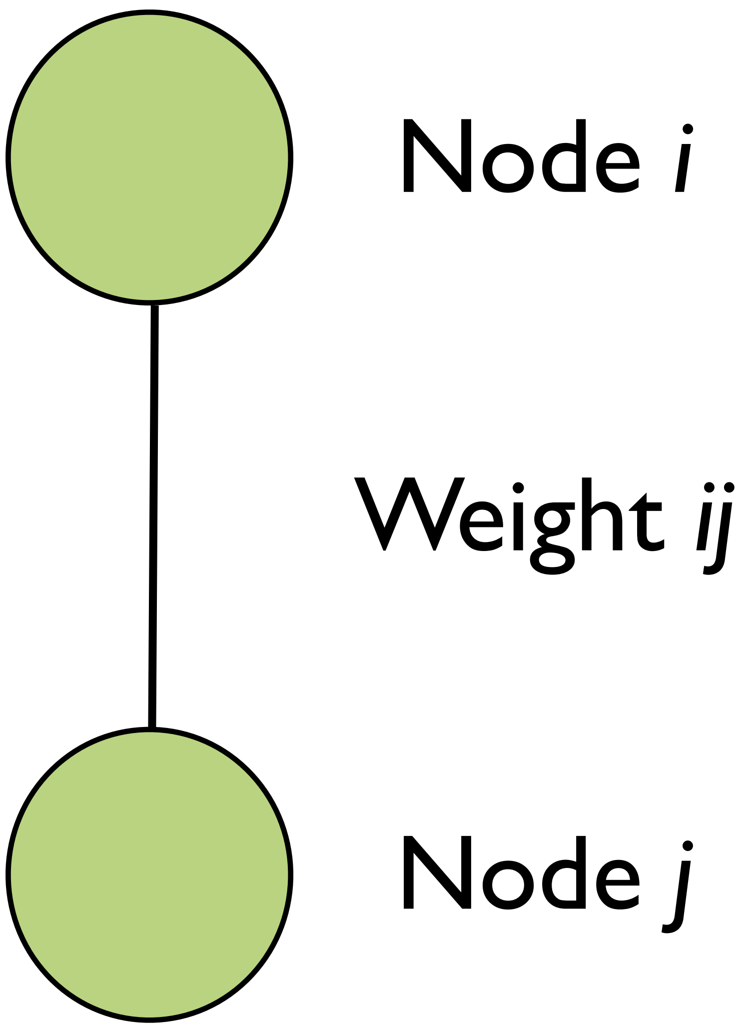
0.3 0.2 -0.4 0.2 ⟵ Output



What is the error at the *i*-th output unit?

$$e_i = t_i - o_i$$

Learning by Backpropagation of Error: II



We want to alter w_{ij} in proportion to its contribution to the overall error.

This is most straightforward, when Node *i* is an output node

Learning by Backpropagation of Error: III

Let *E* be the sum of the error at the output units, then

$$\frac{\partial E}{\partial w_{ij}}$$

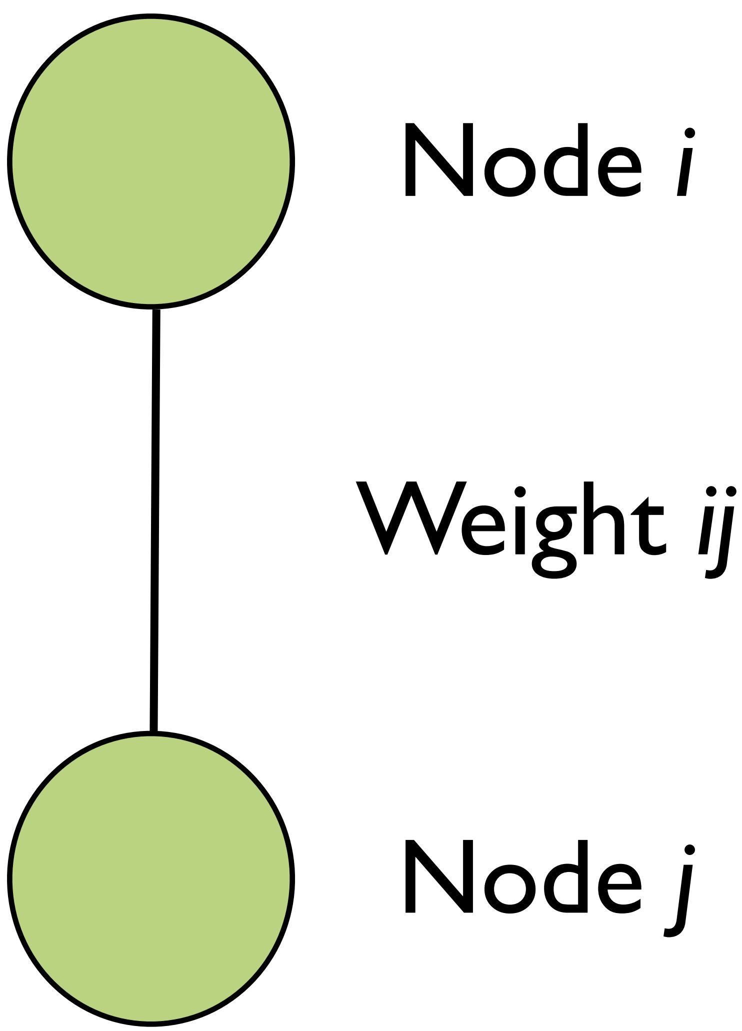
is the '*partial derivative of the error with respect to weight* w_{ij}'.

It provides a measure of how much *E* will change if we make a small change to w_{ij}.

$$\Delta w_{ij} = -\underbrace{\eta}_{\text{Learning Rate}} \frac{\partial E}{\partial w_{ij}}$$

Partial differentiation: assume all other variables are fixed, and just look at how one thing (E) changes as we wiggle another (w_{ij})

Learning by Backpropagation of Error: IV



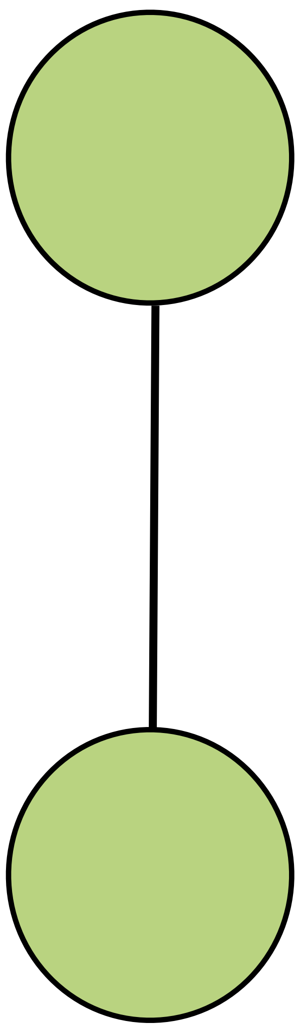
From that, we derive a general expression for the change to w_{ij}, based on the error at a given pattern:

$$\Delta w_{ij} = -\eta \delta_i o_j$$

'delta' is a generalized error term associated with Node *i*, and its meaning will differ, depending on whether *i* indexes an output node or not.

Learning by Backpropagation of Error:V

$$\Delta w_{ij} = -\eta \delta_i o_j$$



Node *i*

Weight *ij*

Node *j*

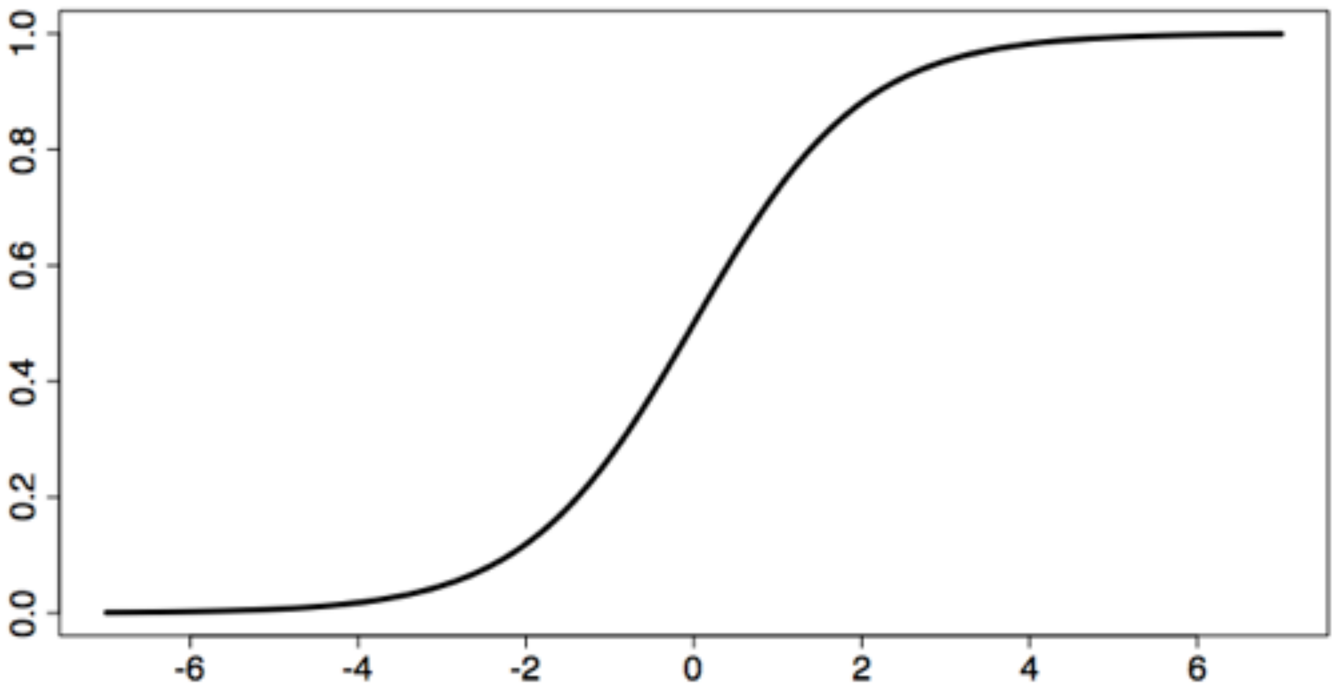
For a linear output unit:

$$\delta_i = (t_i - o_i)$$

For any output unit:

$$\delta_i = (t_i - o_i) f'(net_i)$$

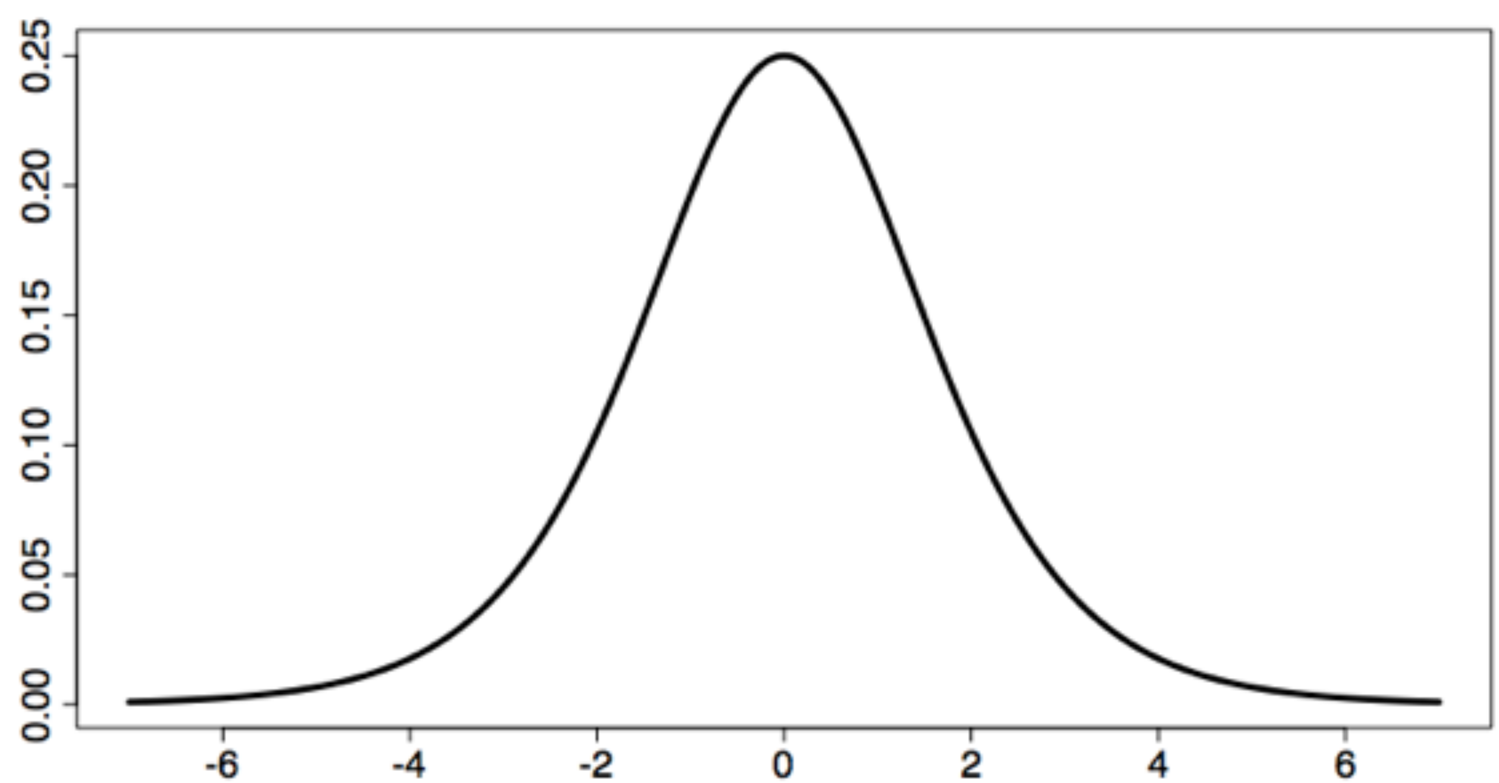
$f'(net_i)$ This is the derivative (rate of change) of the activation function

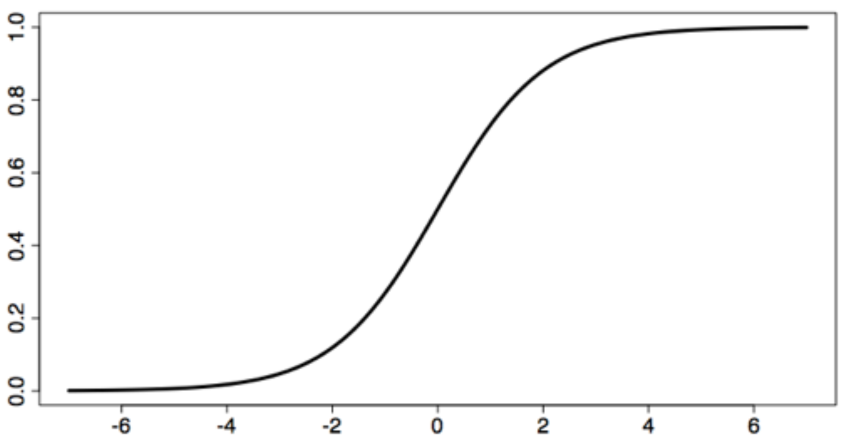


Logistic function

Derivative of logistic function

(= rate of change)



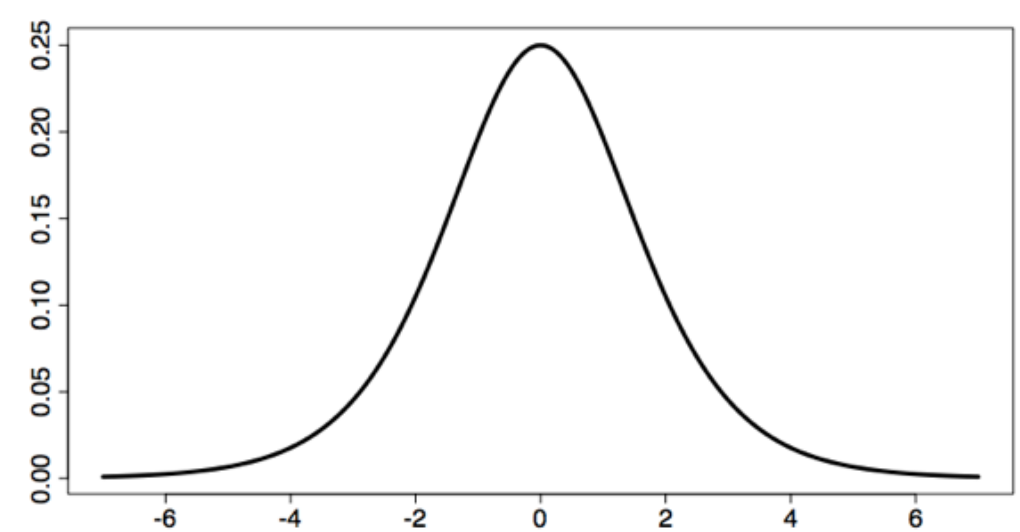


Logistic function

$$y = \frac{1}{1 + e^{-x}}$$

$$y' = y(x)(1 - y(x))$$

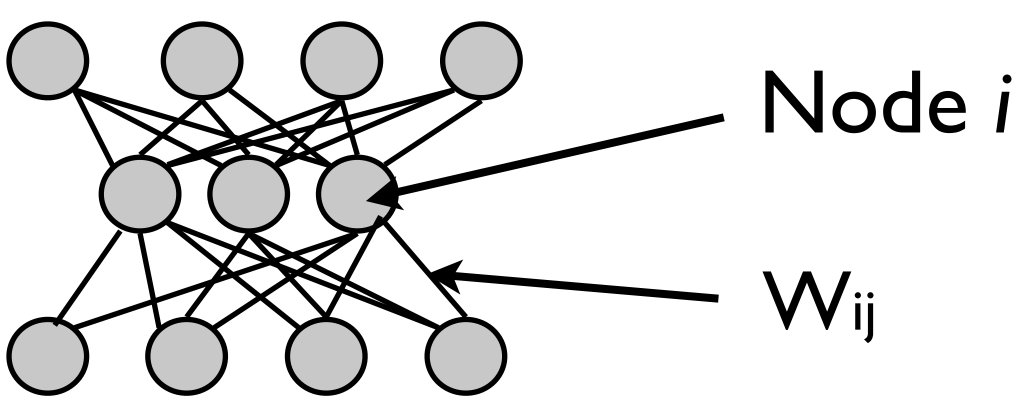
Derivative of logistic function



Learning by Backpropagation of Error: VI

$$\Delta w_{ij} = -\eta \delta_i o_j$$

When Node *i* is not an output node, delta, the error term, is based on a sum of all the errors to which this node contributes.



$$\delta_i = f'(net_i) \sum_k \delta_k w_{ki}$$

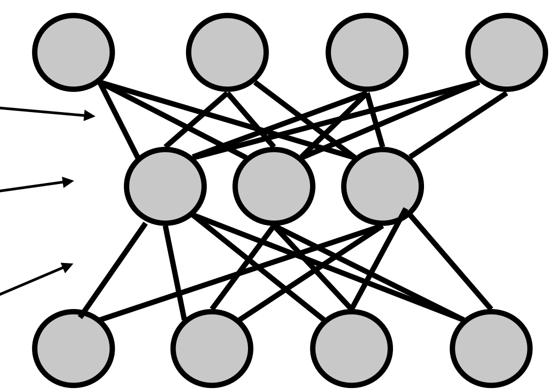
Recall..............

For each pattern in the training set:

Compute the error at the output nodes

Compute Δw for each wt in 2nd set of weights

Compute delta (generalized error expression) for hidden units



Compute Δw for each wt in 1st set of weights

After amassing Δw for all weights and all patterns, change each wt a little bit, as determined by the learning rate

$$\Delta w_{ij} = -\eta \delta_{ip} o_{jp}$$