Machine Learning
CS342

Lecture 14: Artificial Neural Networks (ANNs):
Multilayer Perceptrons (MLPs)

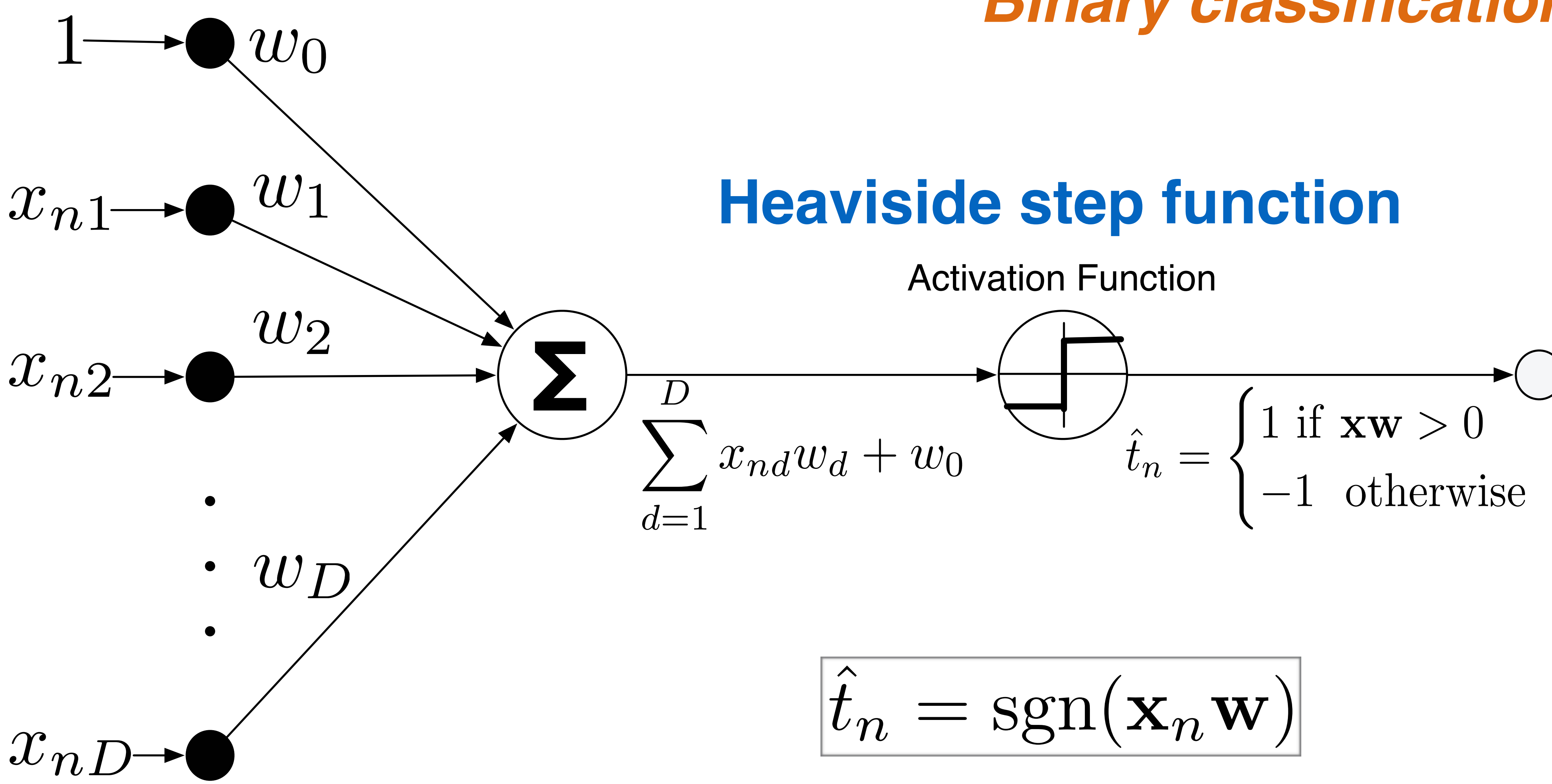
Dr. Theo Damoulas
T.Damoulas@warwick.ac.uk

Office hours: Mon & Fri 10-11am @ CS 307

Recap: Perceptron

Binary classification

Heaviside step function



$$\boxed{\hat{t}_n = \text{sgn}(\mathbf{x}_n \mathbf{w})}$$

Recap: Training a Perceptron

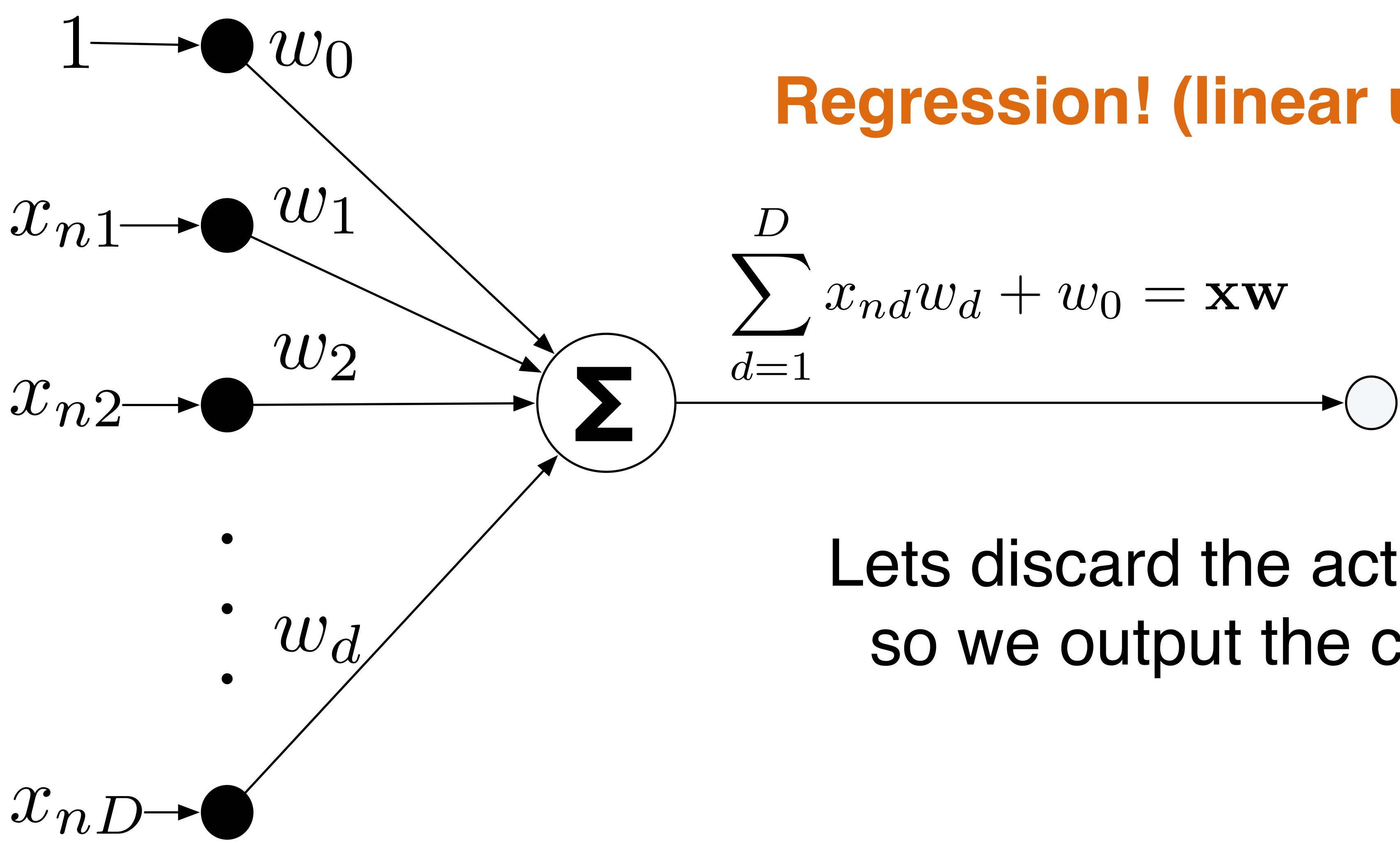
Binary classification

```
Initialise w randomly
eta = 0.1 (for example);
while there is a non-zero error
   for i = 1 to N (number of training examples)
      Choose ith training example x,t
      Compute dot product xw
      Compute error(i) = t-sign(xw)
      Update w += eta*error(i)*xT
```

If problem non-linearly separable this will not converge (Error not 0)

Recap: Linear unit perceptron & Gradient Descent

Perceptron error = (t_n - sgn($\mathbf{x}_n\mathbf{w}$)) and differentiating that wrt **w** is not nice



Regression! (linear unit perceptron)

$$\sum_{d=1}^{D} x_{nd}w_d + w_0 = \mathbf{xw}$$

Lets discard the activation function so we output the continuous **xw**

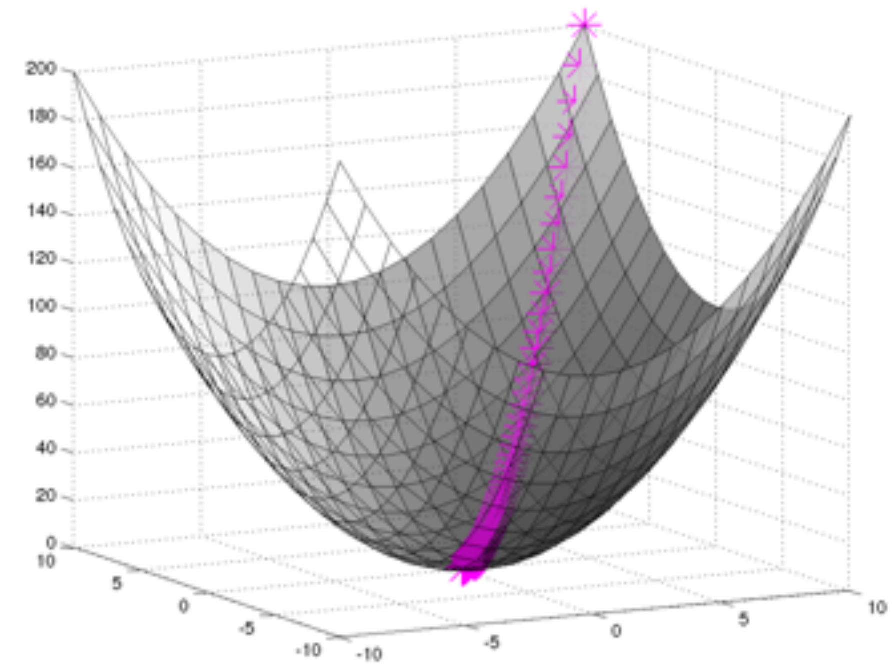
Today we will see an activation function that we can differentiate easily for classification with GD!

Recap: Gradient Descent on linear-unit perceptron

We are finding the OLS solution with a NN and gradient descent!

Batch-mode GD

```
Initialise w randomly
eta = 0.1;
while not converged
     Update w += eta*X^T(t - Xw)
```

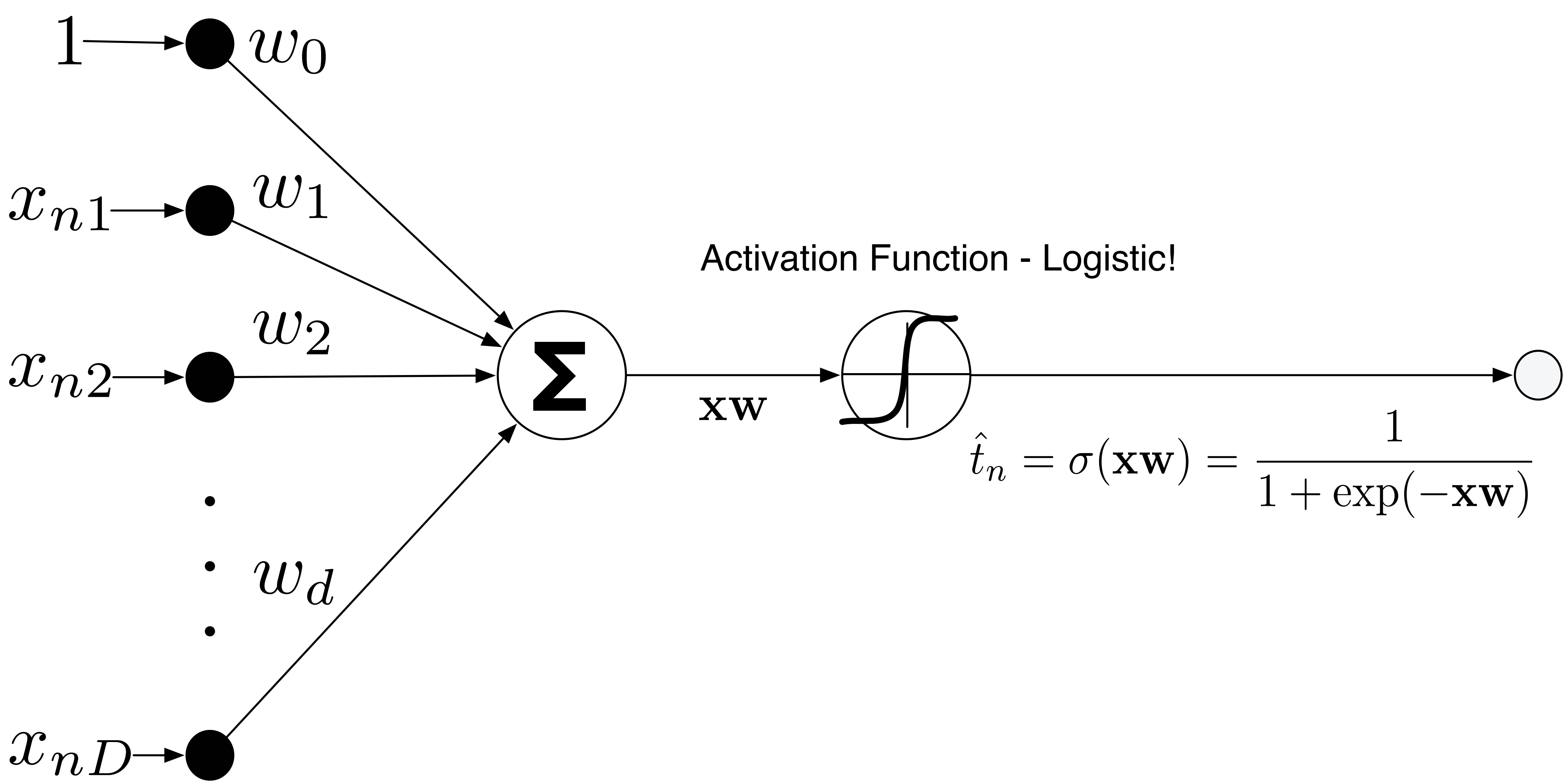


Stochastic GD

```
Initialise w randomly
eta = 0.01; (typically smaller then batch mode)
while not converged
  for i = 1 to N (number of training examples)
     Choose ith training example x,t
     Update w += eta(t - xw)x^T
```

Today: Logistic AF - MLPs - Backpropagation

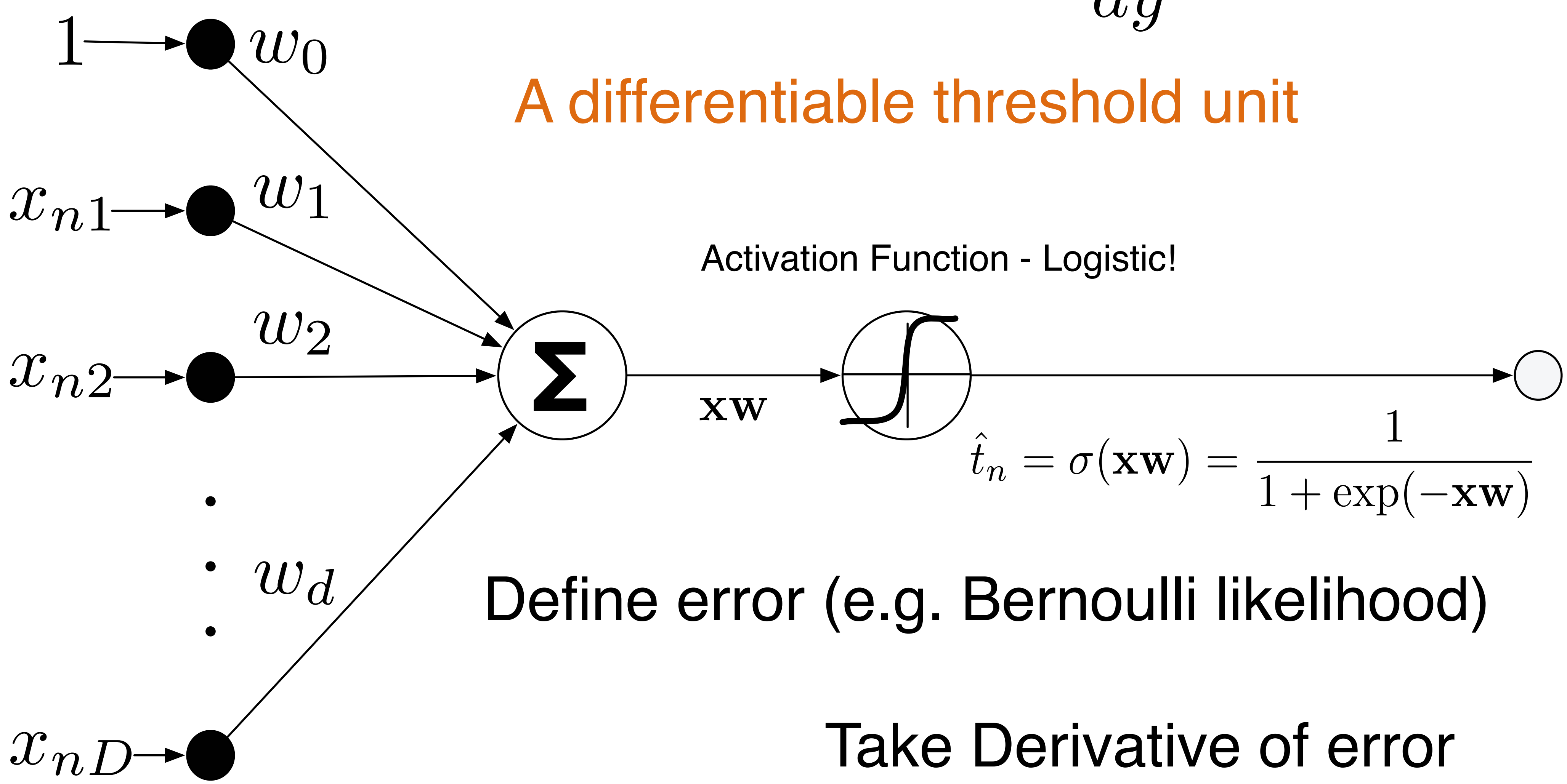
Binary classification:
We know another function apart from the step function that is continuous and has a nice derivative to use for GD: Logistic function!



$$\hat{t}_n = \sigma(\mathbf{xw}) = \frac{1}{1+\exp(-\mathbf{xw})}$$

Logistic regression with a ANN and GD

$$\frac{d\sigma(y)}{dy} = \sigma(y)(1 - \sigma(y))$$



A differentiable threshold unit

$$\hat{t}_n = \sigma(\mathbf{xw}) = \frac{1}{1 + \exp(-\mathbf{xw})}$$

Define error (e.g. Bernoulli likelihood)

Take Derivative of error
Perform Gradient descent
= (Maximum Likelihood on) Logistic Regression

So what's new then?

Original perceptron: Online, Step AF, Linearly separable

Linear unit perceptron = GD or SGD on OLS problem

Logistic AF on perceptron = GD or SGD on Logistic regression

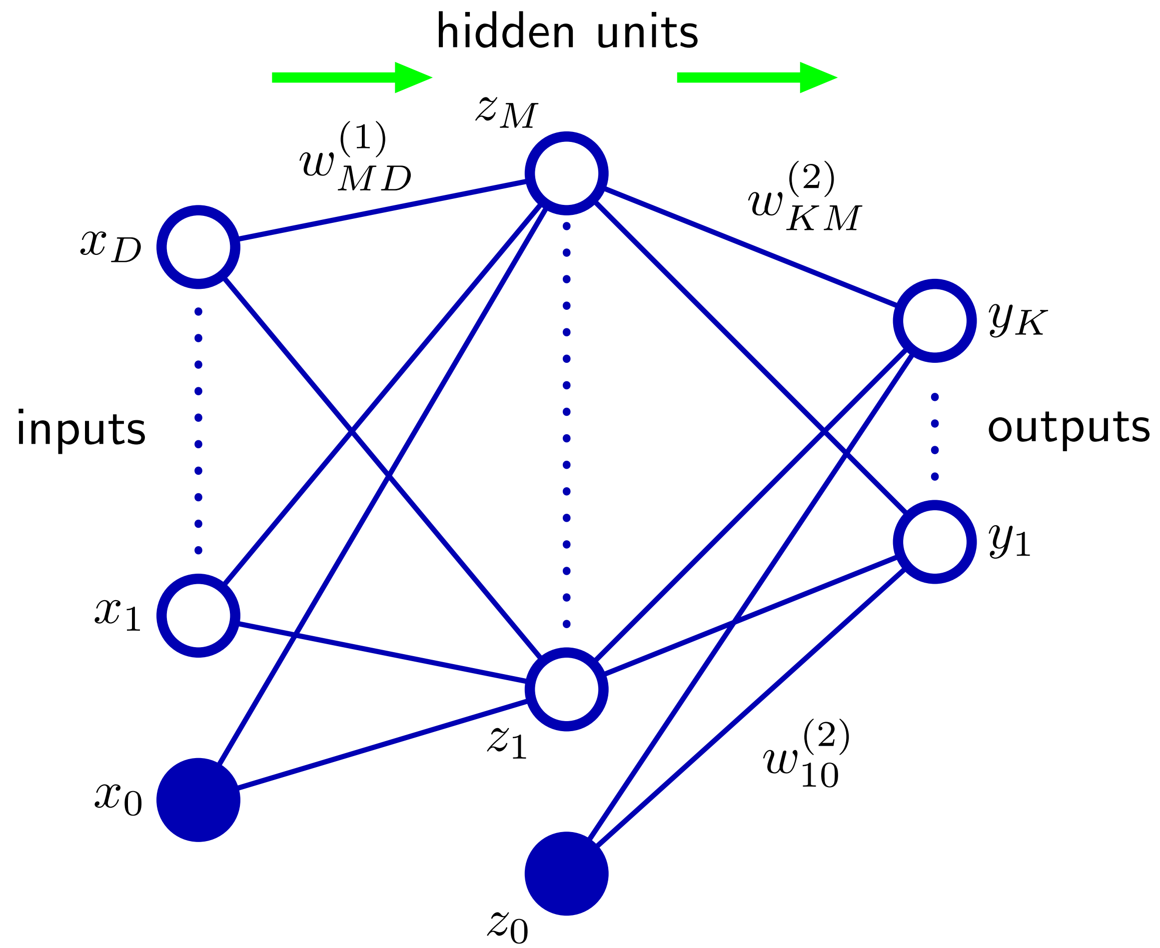
So really so far the novelty is:

a) online nature (SGD and perceptron)
b) excuse to describe GD and SGD
c) "visual" representation of dot products and squashing functions

In ANNs the real novelty is in combining multiple models: many "neurons":
Multilayer Perceptrons (MLPs)

Multilayer Perceptrons (MLPs)

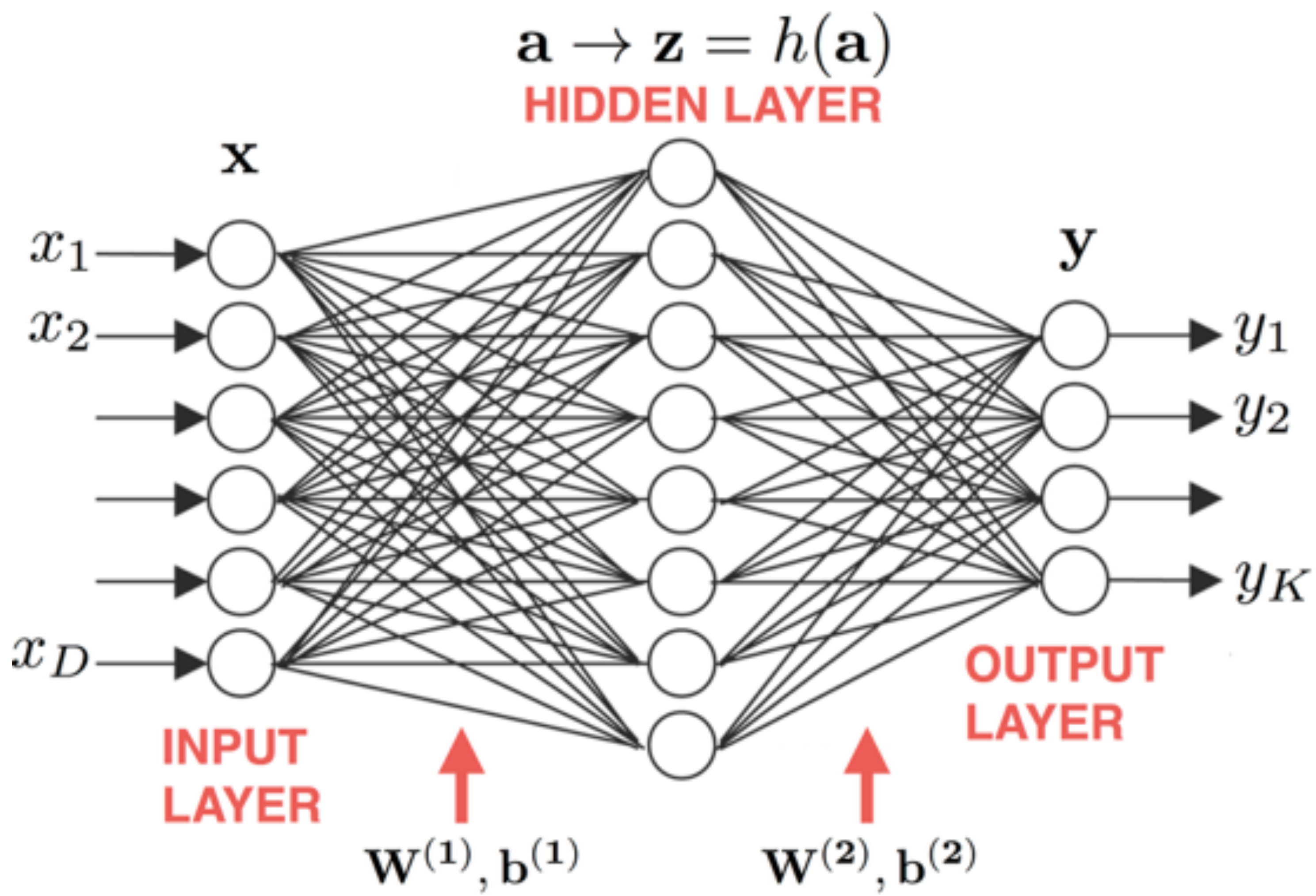
- So far one "computational unit" or neuron - how about **adding more**?
- Use logistic activation function or similar sigmoid functions (tanh)
- a.k.a Multilayer feedforward networks or MLPs



2-layer ANN
1 hidden layer

Deep Learning: A "version" of MLPs with many hidden layers and computational units

MLPs: Feed-forward ANNs



Fully connected

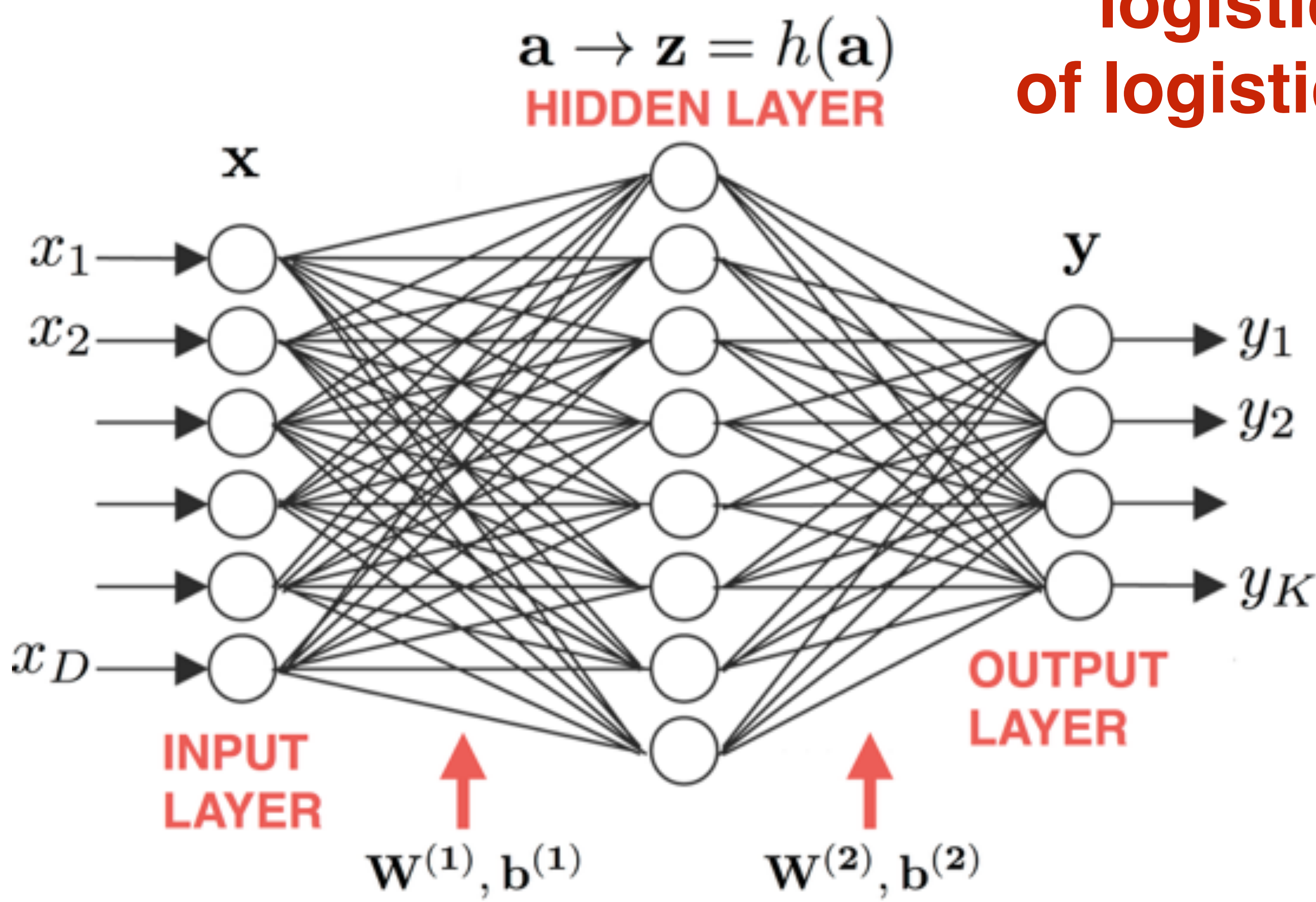
$W^{(1)}$ DxM matrix of connection weights (parameters) between input-hidden

$W^{(1)}$ MxK matrix of connection weights (parameters) between hidden-output

b “bias” term {x=1} like our intercept in LinReg

MLPs: Feed-forward ANNs

logistic regression of logistic regressions..



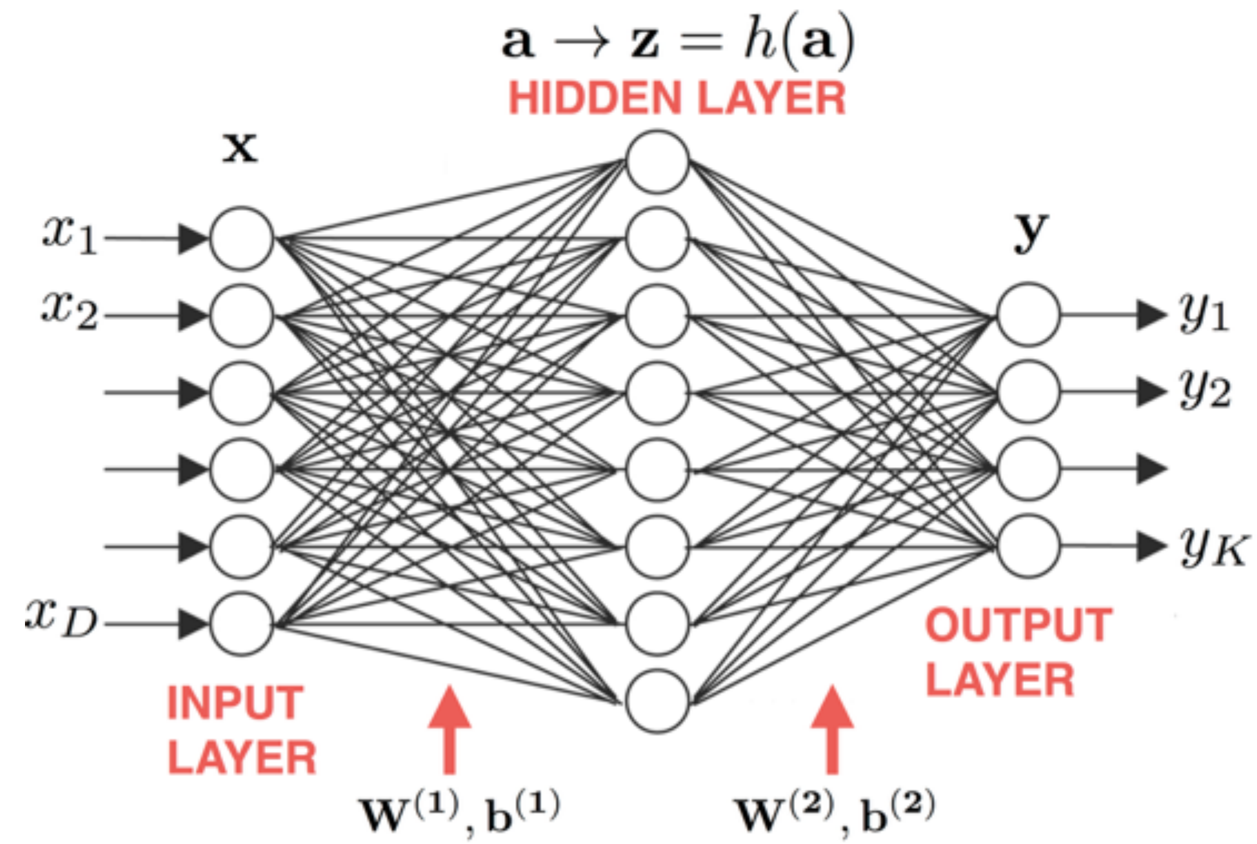
Input layer units is just where the attributes/data come in
Hidden layer units and output layer units are perceptrons with AFs
Typically **logistic** or **tanh** AF **h(a)** that are **differentiable**, non-linear squashing functions (sometimes even linear units).

Department of COMPUTER SCIENCE

MLPs: Feed-forward ANNs

1) Can I write the overall model down?

2) How do I learn (the parameters of) this model?



This network diagram (logistic AF) is equivalent to the model:

σ() and h() are sigmoid AF
e.g. logistic function

$$y_k = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

Same process as in perceptron: form error, take derivative and do GD…
Nasty derivative right?

Training MLPs

Universal Approximators

MLPs are said to be universal approximators. For example a two-layer network with linear units (regressors) and sufficient number of hidden units, can ***uniformly approximate any continuous function on a compact input domain to arbitrary accuracy***!

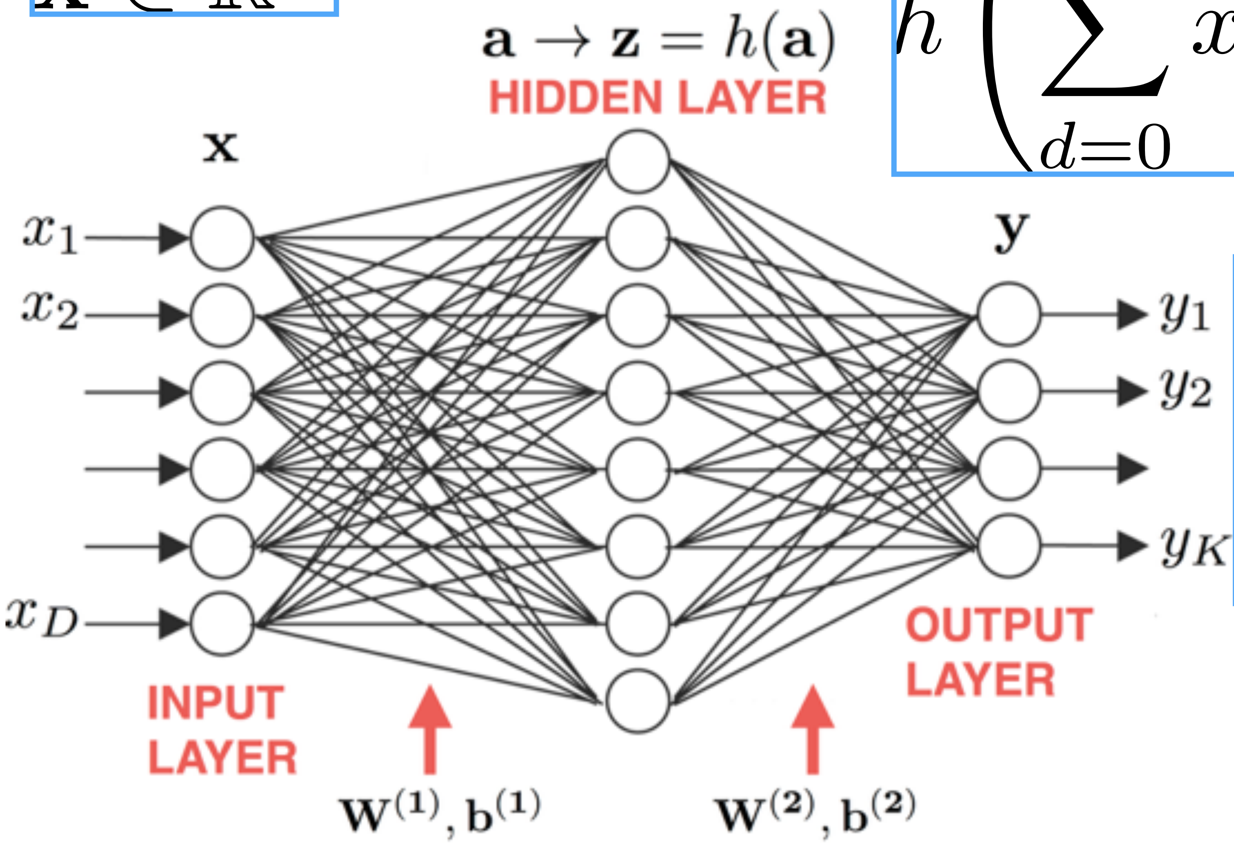
Can train MLPs with various inferential techniques you have already seen: Maximum Likelihood (GD/SGD/etc) & Bayesian approaches

The specific nature of this complex network requires a technique to associate a parameter w with some error component (so we can estimate the Error gradient wrt that parameter)

This technique of passing the error backwards to each unit/parameter is called (Error) back-propagation a.k.a *backprop*

Forward propagation

$$\mathbf{x} \in \mathbb{R}^D$$



$$h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)$$

I omitted a subscript n here as this is per observation n

$$y_k = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)\right)$$

Output layer above is logistic AF but could be a linear unit perceptron for regression

$$\sum_{d=0}^{D} x_d w_{dj}^{(1)}$$

$$\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)$$

Forward Propagation of Information

Department of COMPUTER SCIENCE

Forward propagation

Classifier: Output layer has logistic AF

$$\boxed{y_k = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)\right)}$$

Regression: Output layer has linear-unit (no AF)

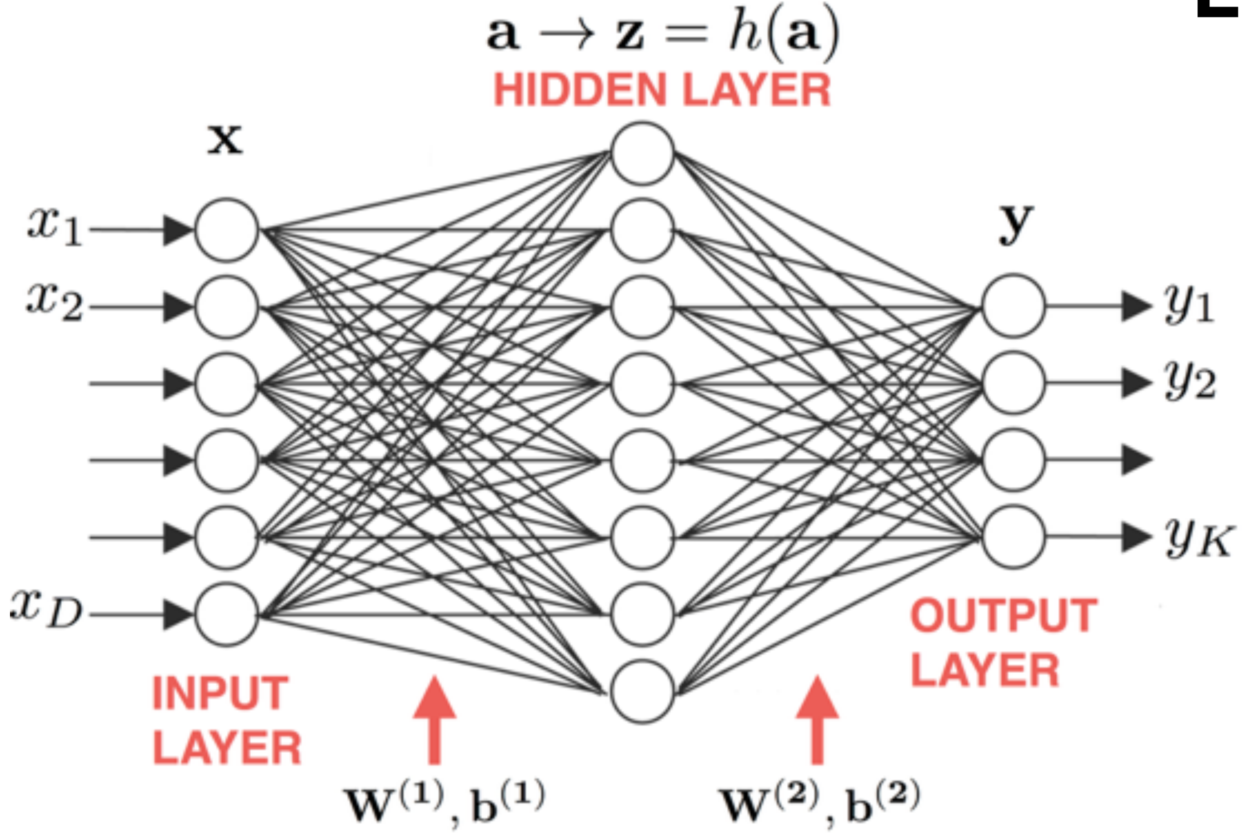
$$\boxed{y_k = \sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)}$$

Based on what we want to do we choose Error (e.g. SQE for regression) and take derivatives of the error with respect to parameters to form a (S)GD procedure

Department of COMPUTER SCIENCE

Error

Linear-unit! regression: SQE



The error of n^th observation

$$\boxed{E_n = \frac{1}{2} \sum_k (t_{nk} - y_{nk})^2}$$

Across all observations (and since its a function of w):

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

Notation convention in NNs $\hat{t}_{nk} = y_{nk}$

For linear-unit the output is:

$$y_{nk} = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)\right)$$

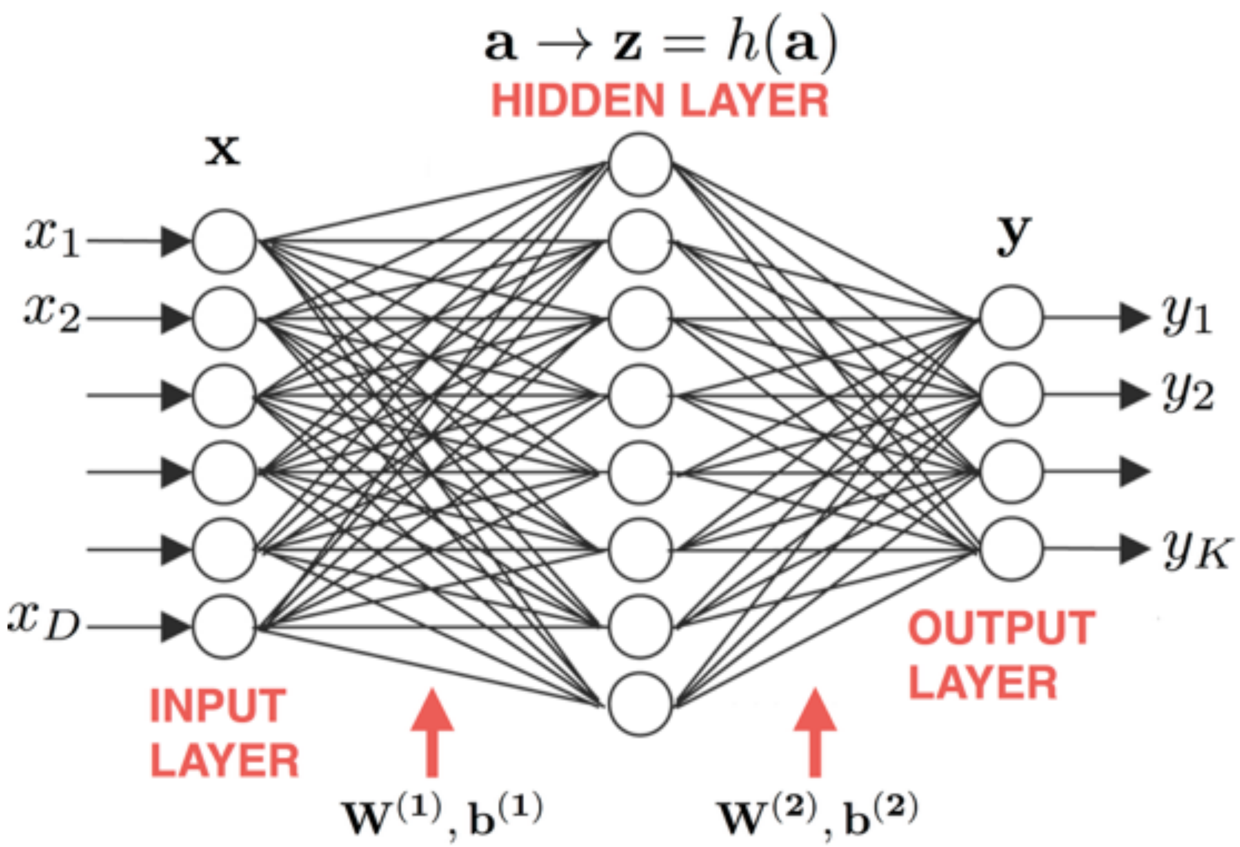
Error Backpropagation

General high level idea:

- As before we want derivatives of the error with respect to the parameters
- If we had these we could construct a SGD procedure to update them
- However the parameters now are "buried" inside complicated functions
- It turns out that we can compute the component of the error that each parameter/unit is responsible for by a message-passing procedure
- Start from the output layer and "assign" error back to hidden layer and from that to parameters.

Error Backpropagation

$$y_{nk} = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{d=0}^{D} x_d w_{dj}^{(1)}\right)\right)$$



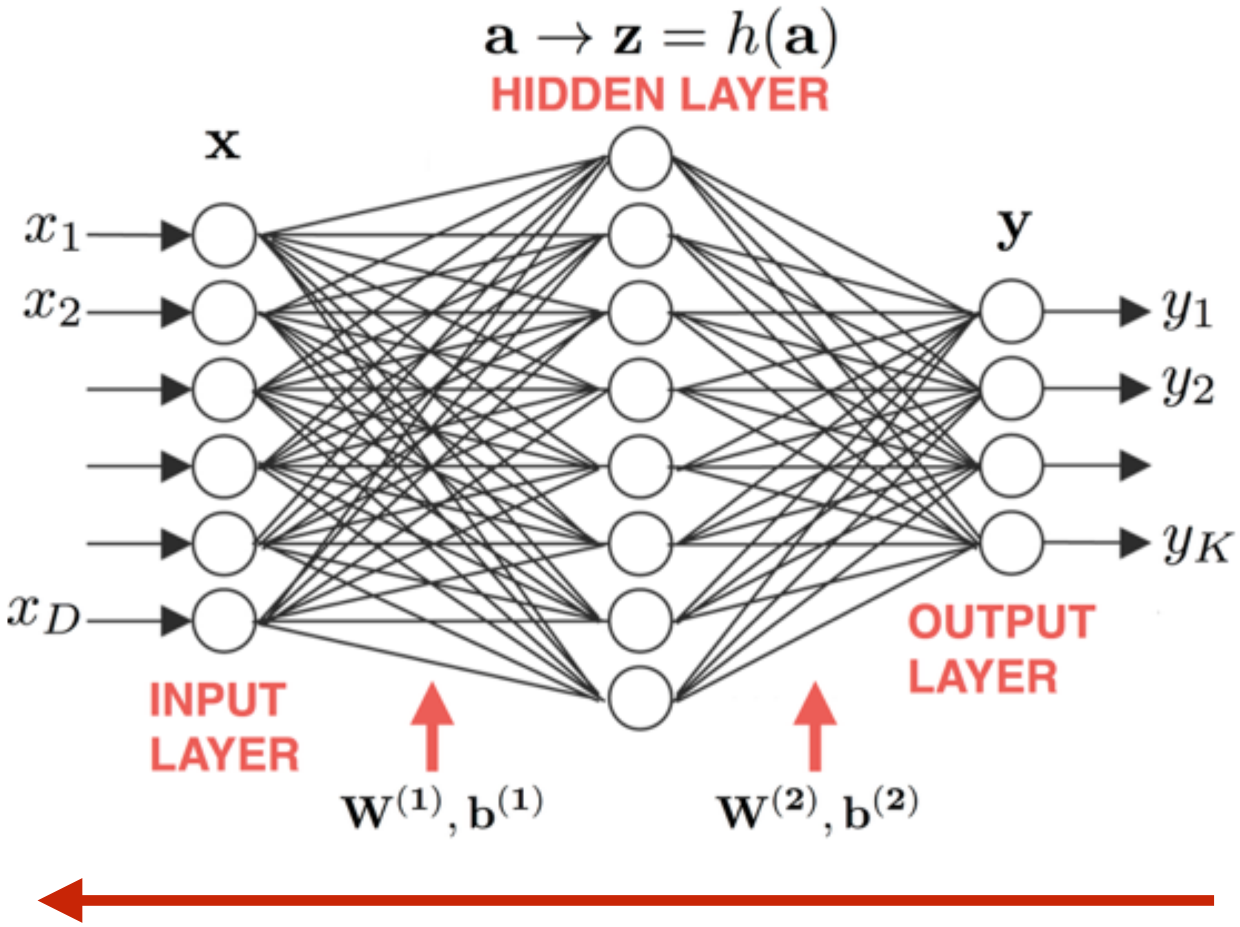
Chain rule!

$$\frac{\partial E_n}{\partial w_{dj}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{dj}}$$

$$\frac{\partial E_n}{\partial w_{dj}} = \delta_j z_d$$

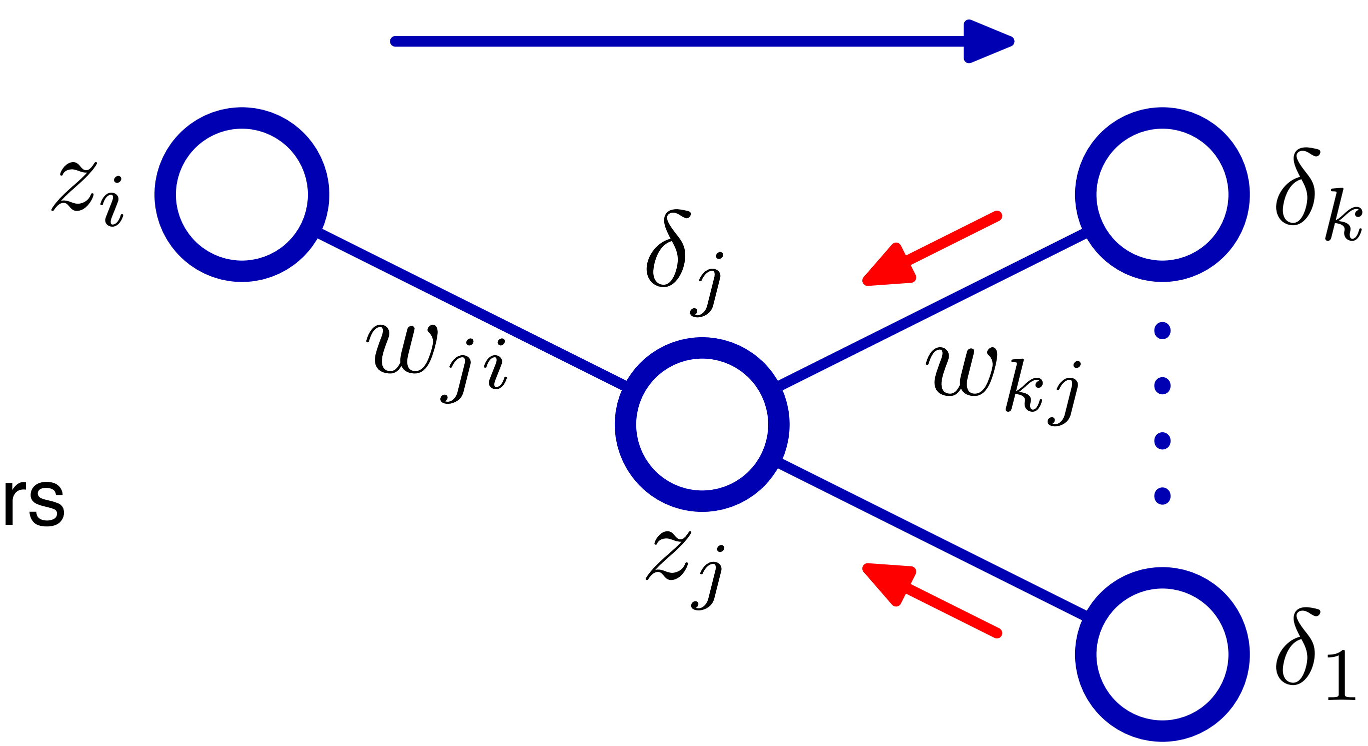
The derivative is obtained by multiplying the value of delta for the unit at the **output end of the weight** by the value z for the unit at the **input end of the weight**

Error Backpropagation



$$\frac{\partial E_n}{\partial w_{dj}} = \delta_j z_d$$

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$



"Message Passing" scheme
where we propagate the delta errors
back into the network

Error Backpropagation

- Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network to find the activations of all hidden and output units
- Evaluate deltas for all output units
- Backpropagate the deltas from output to obtain deltas at hidden units
- Estimate the required error derivatives

Stochastic Gradient Descent
Local minima but very successful!