Data-driven Intelligent Systems

Lecture 10

Classification with Multi-layer Neural Networks

KNOWLEDGE
TECHNOLOGY

http://www.informatik.uni-hamburg.de/WTM/

Overview

- Multilayer Perceptron
  - Error Function & Gradient
  - Gradient Descent Learning
  - Transfer Functions
  - Learning Capacity & Generalization
  - Testing

Recap: Linear Separability of a Perceptron

perceptron's output

inner activation

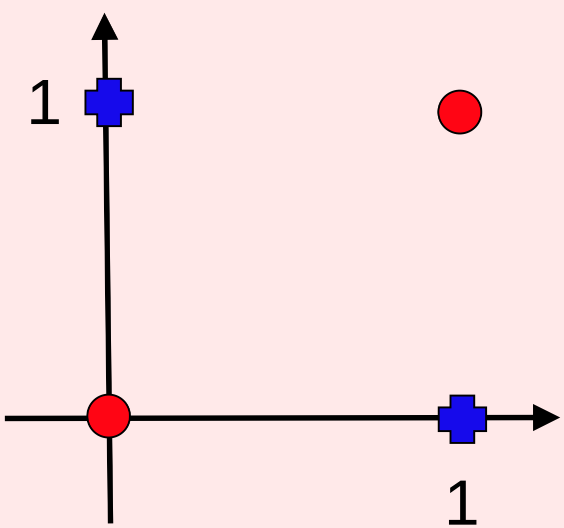
weights

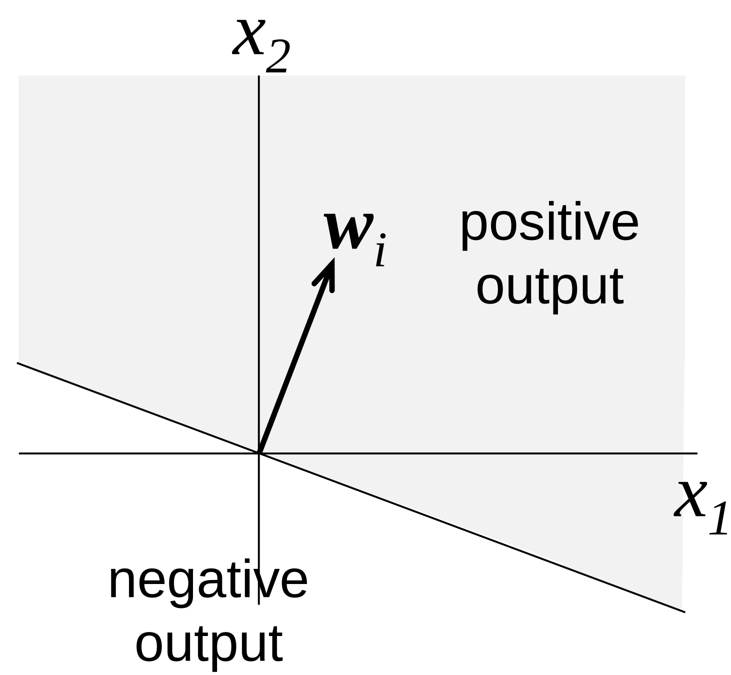
inputs

$$y_i = g(h_i), \text{ where } h_i = \sum_{j=1}^{n} w_{ij} x_j = |w_i| \cdot |x| \cdot \cos(w_i, x)$$

activation function $\quad g(h) = sign(h)$

Perceptron cannot solve XOR





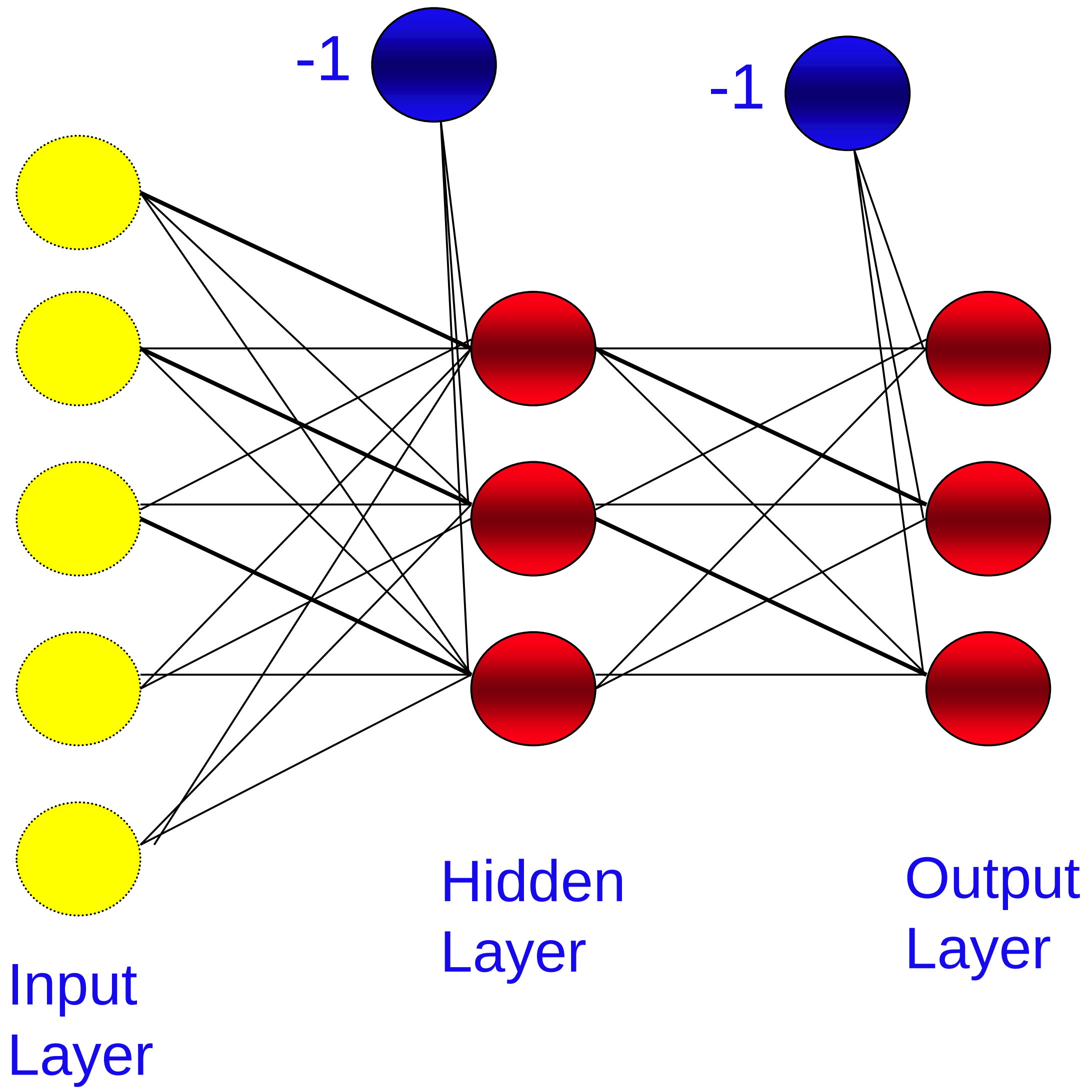
Perceptron

How can we make the perceptron more powerful?

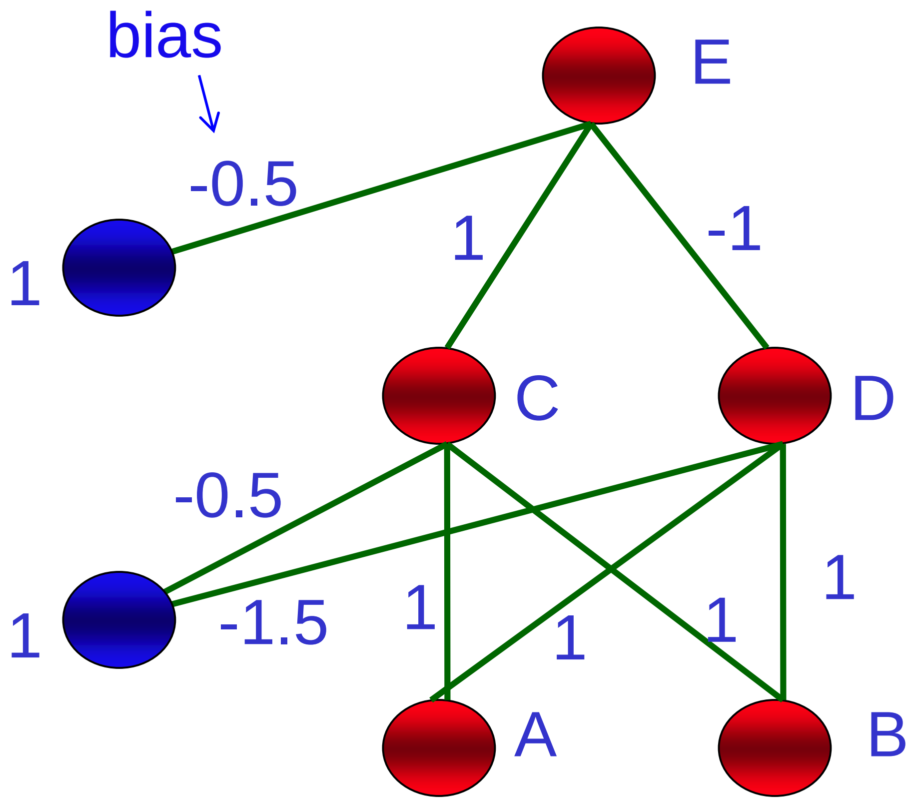
- More connections?
- More layers in the networks?

- Perceptron: one layer of weights
- Multi-layer perceptron: at least 2 layers of weights

The Multi-Layer Perceptron

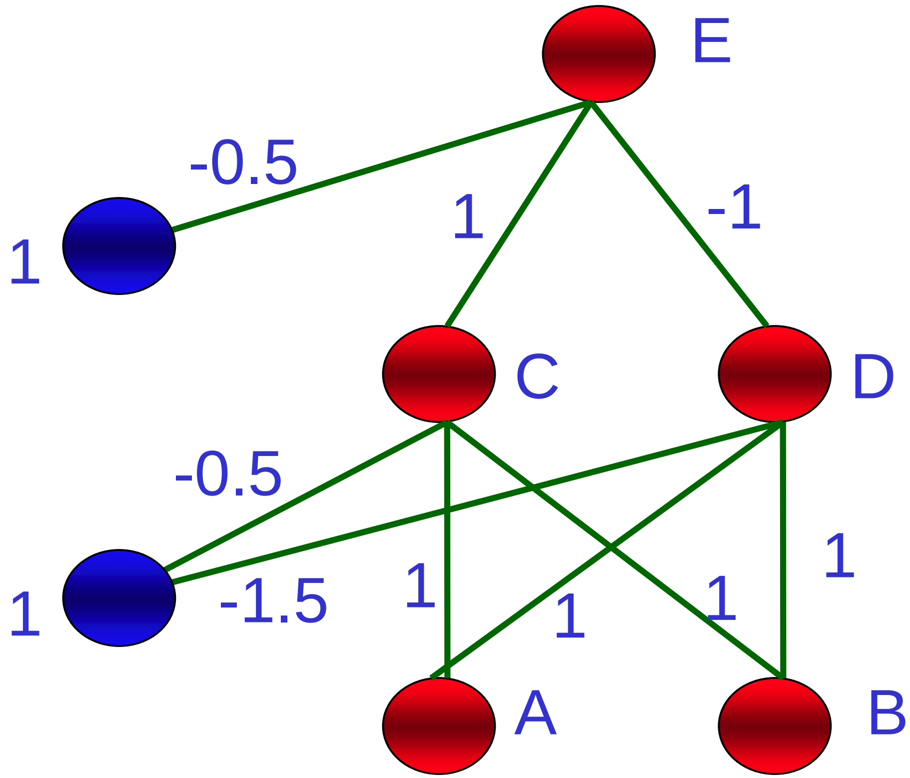


An MLP Can Solve the XOR Problem



A	B	E
0	0	0
0	1	1
1	0	1
1	1	0

XOR Solved

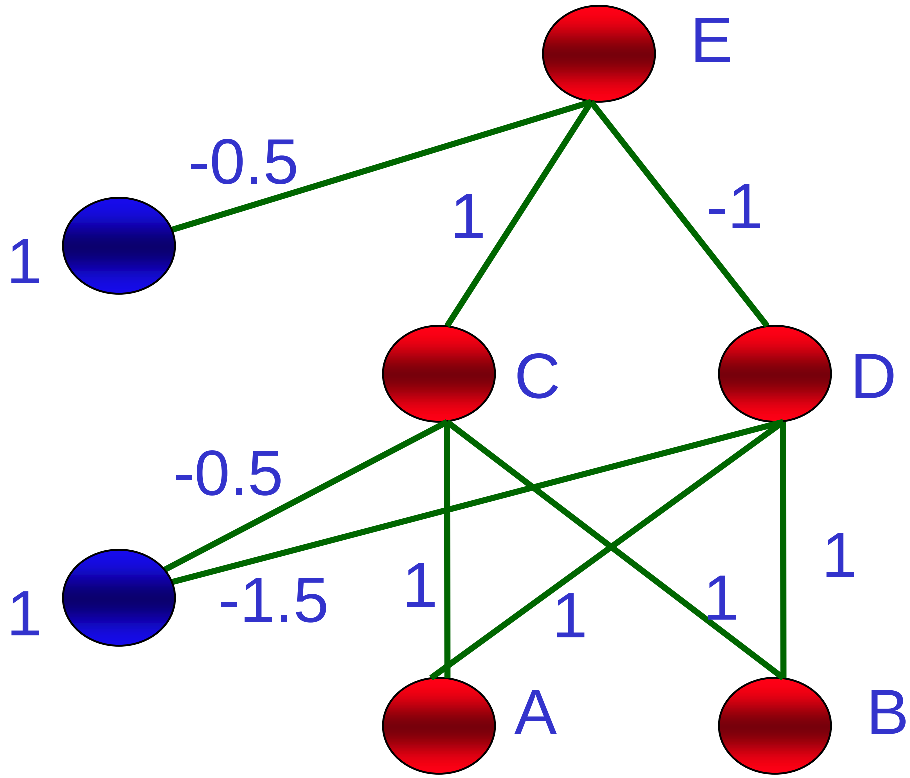


So E does not fire

C does not fire,
D does not fire

Apply input 0 0

XOR Solved

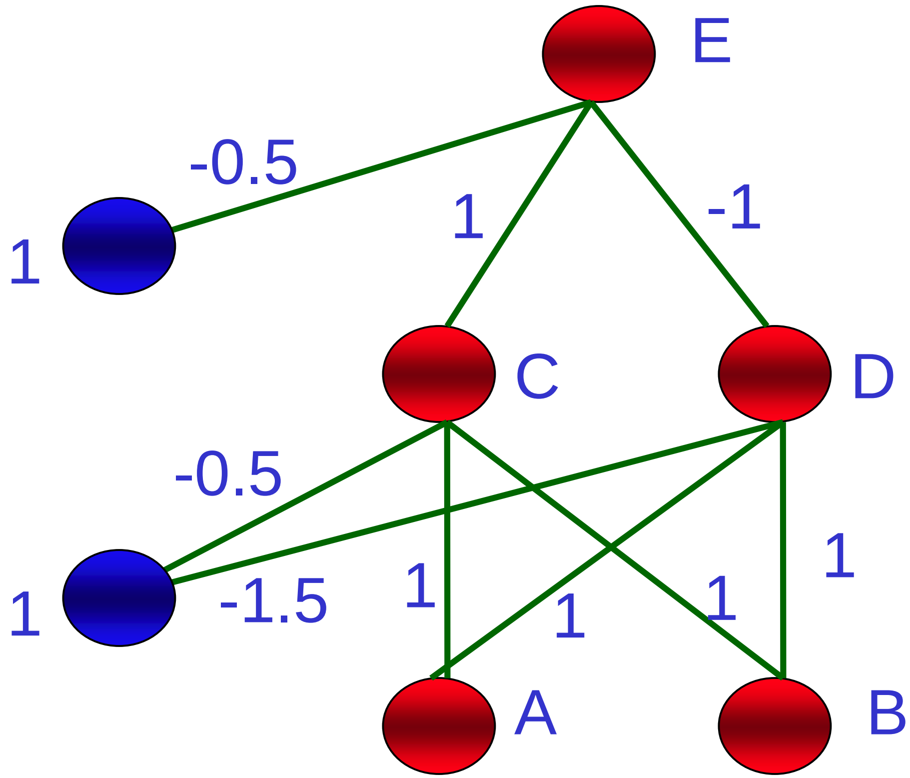


So overall E fires

C fires
D does not fire

Apply input 1 0
Same for input 0 1

XOR Solved

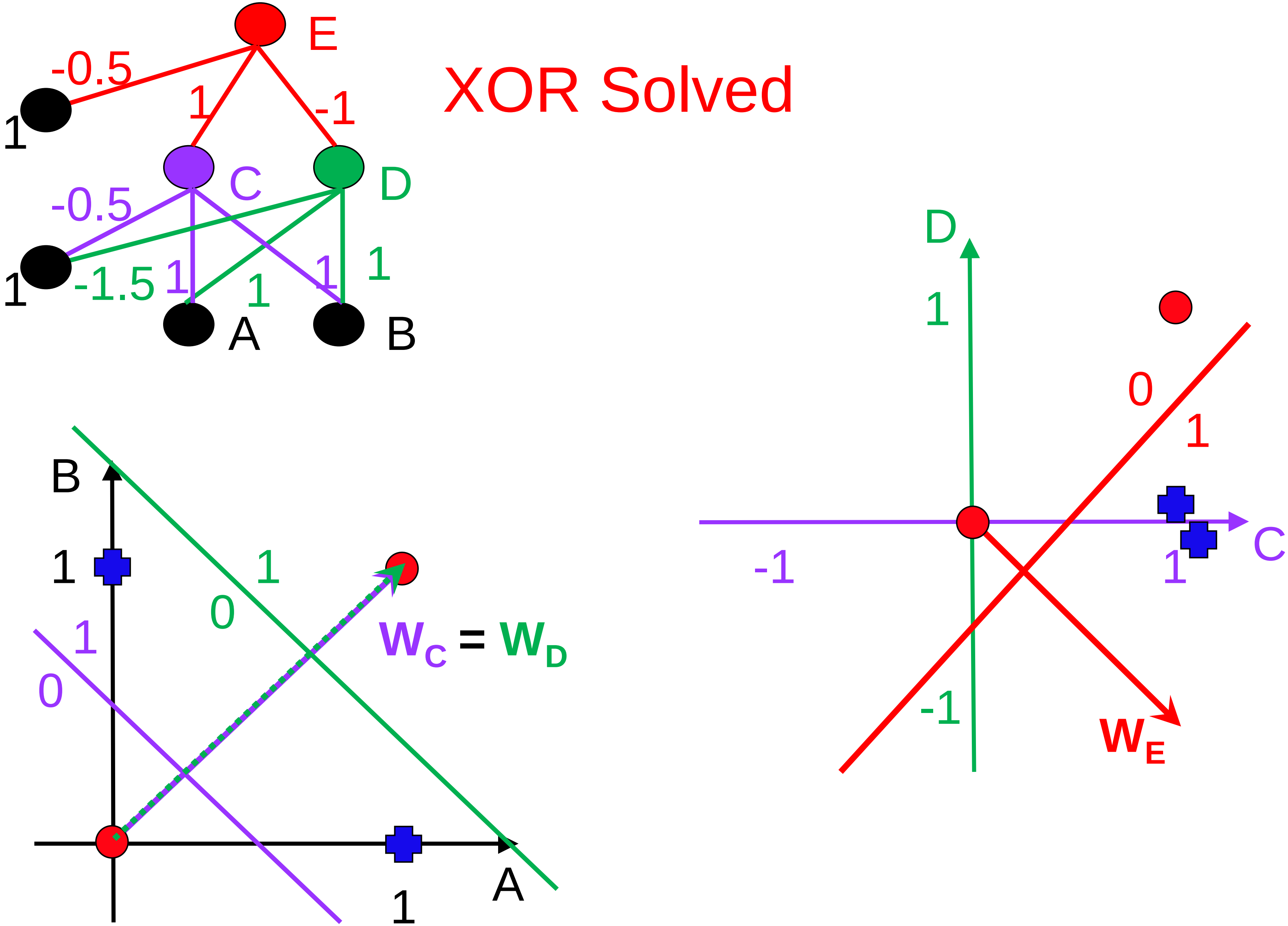


So E does not fire

C fires
D fires

Apply input 1 1

XOR Solved



Overview

- Multilayer Perceptron
  - Error Function & Gradient
  - Gradient Descent Learning
  - Transfer Functions
  - Learning Capacity & Generalization
  - Testing

How a Multi-Layer Neural Network Works

- The network is ***feed-forward***; there are no cycles; i.e. no weight feeds back to a unit in the same or a previous layer
  - The network ***inputs*** are the attributes of each training tuple
  - Inputs are fed simultaneously into the units of the ***input layer***
  - They are then weighted and fed simultaneously to a ***hidden layer***
  - The outputs of the last hidden layer are input to the ***output layer***, which emits the network's prediction

Decide on the Network Topology

- # of units in the *input layer* ←fixed through the application
  - One input unit per domain value
- # of units in the *output layer* ←fixed through the application
  - One output unit for each variable in regression
  - A single output unit for two-class classification
  - For more than two classes, one output unit per class
    - output values may be coupled by a softmax function
- # of *hidden layers* (if > 1)
  - Complex function – transformations? Hierarchical features?
- # of units in *each hidden layer*
  - Complex function – many features?
- *May repeat training with different network topologies*

Weight Initialisation & Data Preprocessing

- Initialize the network with small random weights
  - All-same weights would lead to symmetry-breaking problem: all hidden units will have same activations and learn the same
  - Large weights could lead to saturation of the transfer function
- Normalize the input values for each attribute measured in the training tuples, e.g.
  - shift & scale attribute values to be in the interval [0.0 .. 1.0], or
  - shift & scale them to have mean=0, variance=1;
  - done per attribute or over all attributes

    all attributes have same importance — *attributes keep their relative importance*

- *May repeat training with a different set of initial weights*

Gradient Descent

- The MLP ***can*** solve XOR
- How do we learn the weights?
- Harder than for the perceptron
  - Multiple layers
  - Which layer's weights are wrong?
    - Input-hidden or hidden-output?
- Use gradient descent learning
- Compute gradient ⇨ differentiation

An Error Function (recap)

- Sum-of-squares error

target output

$$E(\mathrm{w}) = \frac{1}{2} \sum_{data} \sum_{i} \left(t_i - y_i\right)^2$$

- Let's assume linear neurons: (no threshold function)

$$y_i = h_i = \sum_j w_{ij} x_j$$

- Rule for output unit's weights:

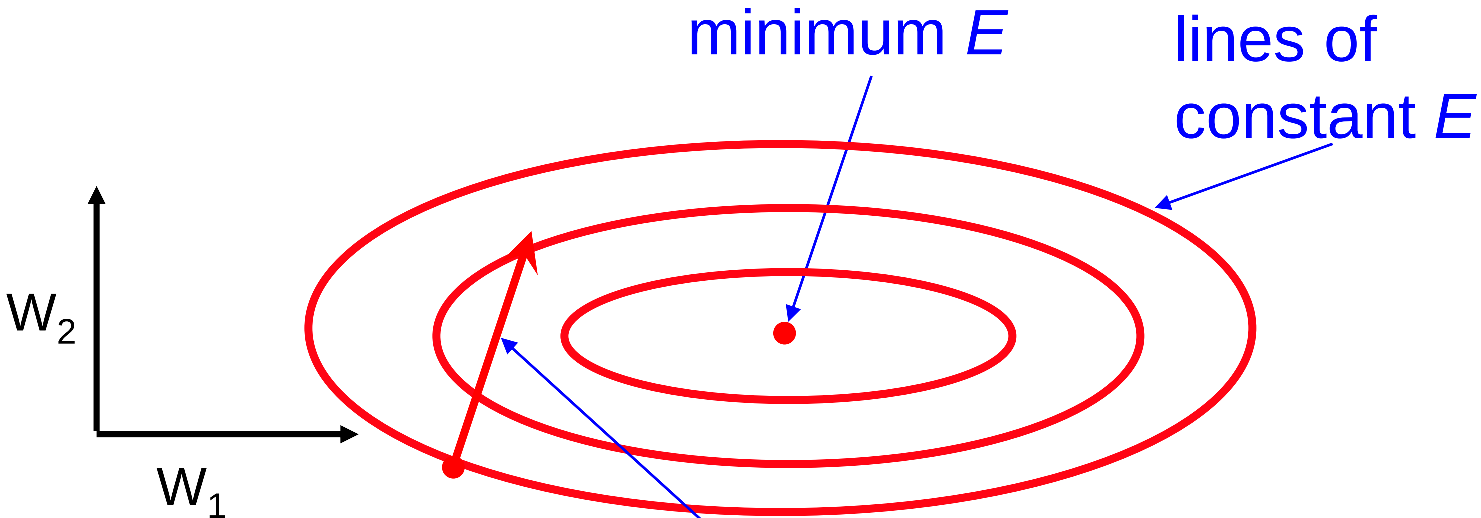
error on output unit

activation of input unit

$$\Rightarrow \qquad -\frac{\partial E}{\partial w_{ij}} = \sum_{data} \left(\overbrace{t_i - y_i}\right) \cdot x_j$$

Problem: error on hidden units is unknown
→ cannot find a rule!

Gradient Descent



$$\Delta w = -\eta \frac{\partial E}{\partial w} = -\Delta E$$

learning rate

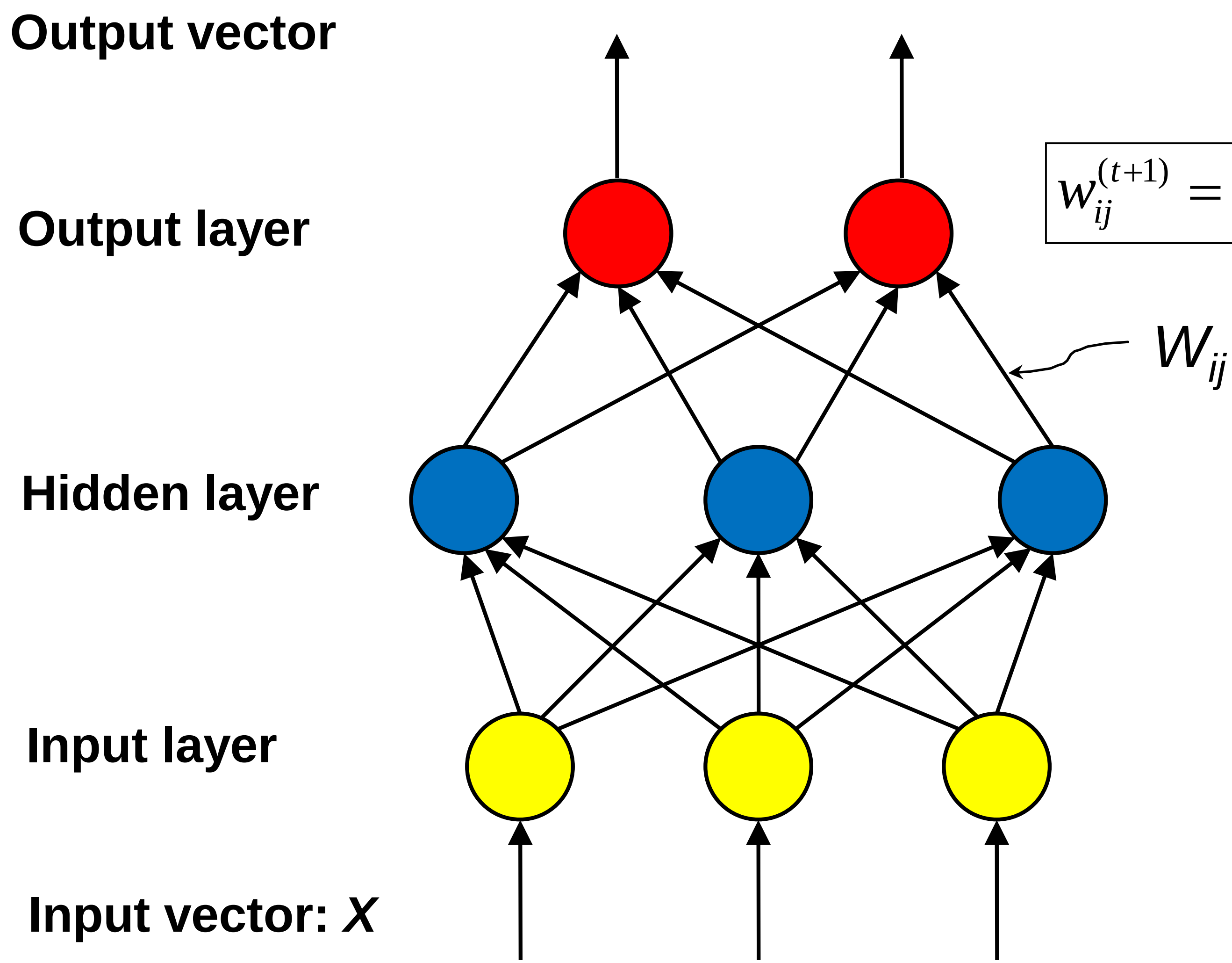
We differentiate error function *E* to get its negative gradient
→ good direction of change for parameters ***w***

Overview

- Multilayer Perceptron
  - Error Function & Gradient
  - Gradient Descent Learning
  - Transfer Functions
  - Learning Capacity & Generalization
  - Testing

A Multi-Layer Feed-Forward Neural Network

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \eta(t_i - y_i)x_j$$



From the Perceptron to the MLP

- Perceptron:

$$y_i = g\left(\sum_{j=1}^{n} w_{ij} x_j\right)$$

- MLP with one hidden layer (*L=1*), one output layer (*L=2*):

$$y_i^{L=2} = g^{L=2}\left(\sum_{j=1}^{n^{L=1}} w_{ij}^{L=2} y_j^{L=1}\right) = g^{L=2}\left(\sum_{j=1}^{n^{L=1}} w_{ij}^{L=2} g^{L=1}\left(\sum_{k=1}^{n^{L=0}} w_{jk}^{L=1} x_k\right)\right)$$

Let's write this in matrix notation …

From the Perceptron to the MLP

The activation function *g* is applied to every element of the vector ***h*** = W***x***

- Perceptron:

weight matrix

$$y = g\left(Wx\right)$$

output activation vector

input activation vector

- MLP with one hidden layer (*L=1*), one output layer (*L=2*):

$$y^{L=2} = g^{L=2}\left(W^{L=2}y^{L=1}\right) = g^{L=2}\left(W^{L=2}g^{L=1}\left(W^{L=1}x^{L=0}\right)\right)$$

let's use smooth, differentiable transfer functions *g*

- Derivation of the learning rule is done via the chain rule
  - for the MLP, the chain will be a little longer …
  - Good news: derivation yields recursive formulas for the deltas

Recursive Formulas for the Deltas

last layer (using index i):

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e_i}\;\frac{\partial e_i}{\partial y_i}\;\frac{\partial y_i}{\partial h_i}\;\frac{\partial h_i}{\partial w_{ij}} = \underbrace{e_i \quad -1 \quad g'(h_i)}_{\delta_i}\; y_j$$

top layer

2^{nd}-last layer (using index j):

$$\frac{\partial E}{\partial w_{jk}} = \overbrace{\sum_i \underbrace{\frac{\partial E}{\partial e_i}}_{e_i} \cdot \overbrace{\underbrace{\frac{\partial e_i}{\partial h_i}}_{-g_i'(h_i)} \; \underbrace{\frac{\partial h_i}{\partial y_j}}_{w_{ij}}}^{\frac{\partial e_i}{\partial y_j}}}^{\frac{\partial E}{\partial y_j}} \; \overbrace{\underbrace{\frac{\partial y_j}{\partial h_j}}_{g_j'(h_j)} \; \underbrace{\frac{\partial h_j}{\partial w_{jk}}}_{x_k}}^{\frac{\partial y_j}{\partial w_{jk}}}$$

no need to understand the details …
this can be done by auto-differentiation!

δ_j in lower layers are computed from δ_i in next-higher layer

δ_i (compare with above)

middle layer

δ_j (define as $\delta_j = g_j'(h_j) \sum_i \delta_i w_{ij}$)

perceptron rule

$\Rightarrow$ last layer: $\frac{\partial E}{\partial w_{ij}} = \delta_i y_j$

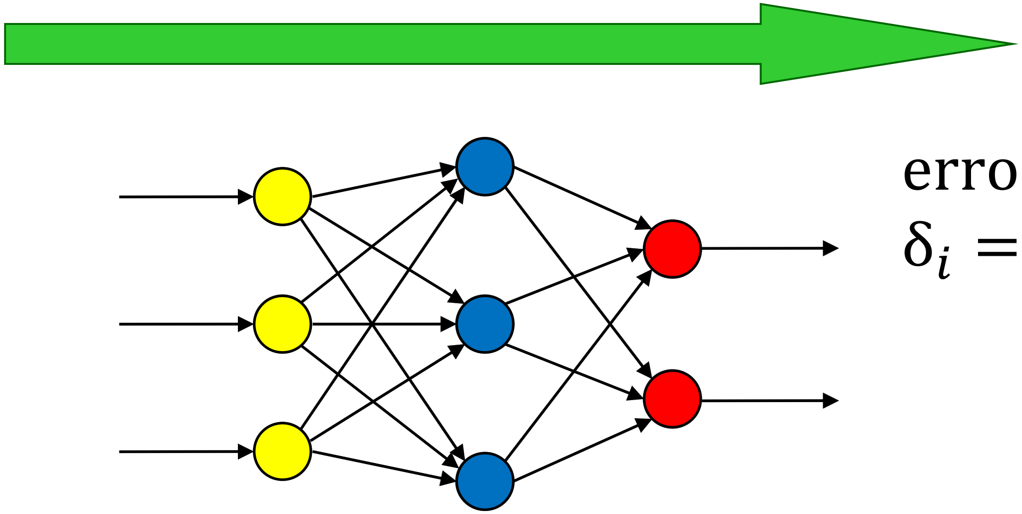
2nd-last layer: $\frac{\partial E}{\partial w_{jk}} = \delta_j x_k$

a similar rule

Training MLP

(1) Forward Pass

- Put the input values in the input layer
- Calculate the activations of the hidden nodes
- Calculate the activations of the output nodes
- Calculate the errors δ using the targets



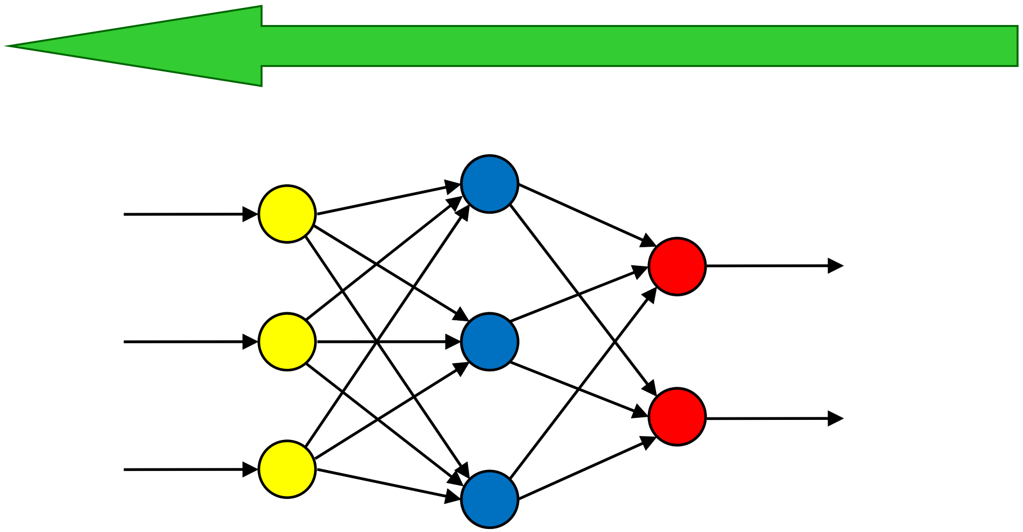
error

$$\delta_i = t_i - y_i$$

Training MLPs

(2) Backward Pass

- Using output errors, update last layer of weights
- Calculate hidden-layer errors, update hidden-layer weights
- Work backwards through the network
- Error is backpropagated through the network



Error Backpropagation - Summary

- Iteratively process training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to ***reduce*** the ***squared error*** between network's prediction and actual target value
  - This minimizes the ***mean*** square error over the entire data set
- Errors are computed "***backwards***": from the output layer, through each hidden layer down to the first hidden layer, hence "***backpropagation***"

Steps

- Initialize weights (to small random #s) and biases in the network
- For each data point:
  - Propagate inputs forward (apply activation function)
  - Propagate the error backwards (backpropagation)
  - Update weights and biases (using inputs and errors)
- Terminate when error small, test error increases, etc.

Overview

- Multilayer Perceptron
  - Error Function & Gradient
  - Gradient Descent Learning
  - Transfer Functions
  - Learning Capacity & Generalization
  - Testing

Activation Function

- In the analysis we ignored the activation function
  - The threshold function is not differentiable

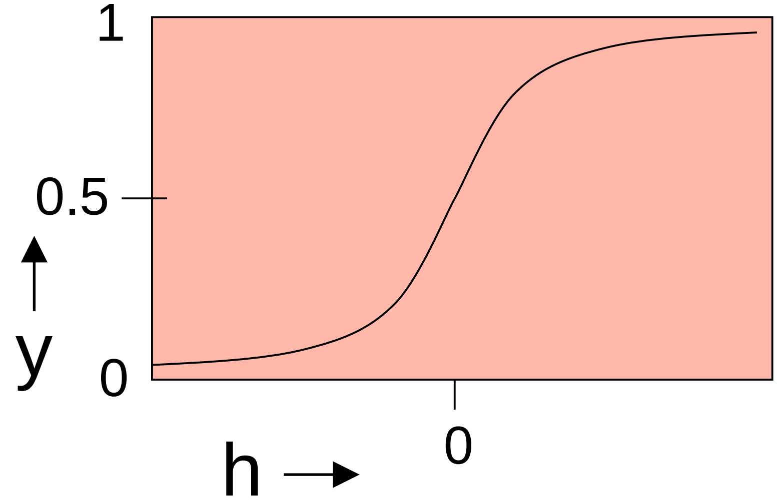
- What do we want in an activation function?
  - Differentiable
  - Should saturate (become constant at ends)
  - Change between saturation values quickly

Sigmoid Neurons

$$h = b + \sum_j x_j w_j$$

- Sigmoidal / logistic transfer function:
  - gives a real-valued, positive output
  - bounded in interval [0,1]
  - easily differentiable, positive derivative
  - output y can be interpreted as a ***probability*** of a binary output to be =1 (or of producing a spike) → stochastic binary neurons

$$\boxed{y = g(h) = \frac{1}{1+e^{-h}}}$$



Sigmoid Activation Function for a Neuron

Transfer function:

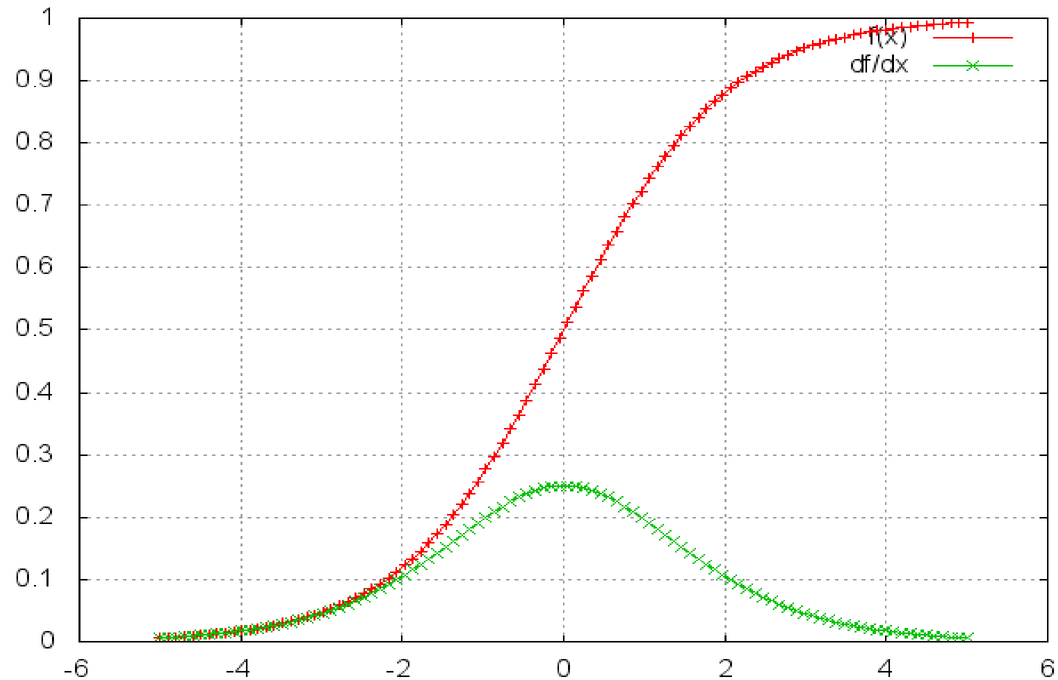
$$g(h) = \frac{1}{1+\exp(-h)}$$

Derivative:

$$g'(h) = \frac{\partial g(h)}{\partial h}$$

$$= \ldots$$

$$= g(h) \cdot (1 - g(h))$$



→ The derivative can be expressed as a function of the ***outputs***.

Overview of Transfer Functions

Transfer function:

Corresponding derivative:

- sigmoid

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

$$g'(h) = g(h) \cdot (1 - g(h))$$

- linear

$$g(h) = h$$

$$g'(h) = 1$$

- threshold function

$$g(h) = \begin{cases} 1 & h \geq \theta \\ 0 & h < \theta \end{cases}$$

no useful derivative

- sign

$$g(h) = \begin{cases} 1 & h \geq \theta \\ -1 & h < \theta \end{cases}$$

no useful derivative

Error Terms

- Need to differentiate the sigmoid function
- Gives us the following ***error terms*** (deltas)
  - For the outputs

$$\delta_i^{out} = (t_i - y_i) \cdot \underbrace{y_i (1 - y_i)}_{\text{derivative of sigmoid}}$$

  - For the hidden nodes (with activations y_j^{hid})

$$\delta_j^{hid} = \underbrace{y_j^{hid} (1 - y_j^{hid})}_{\text{derivative of sigmoid}} \sum_i w_{ij} \delta_i^{out}$$

Update Rules

- This gives us the necessary update rules
  - For the weights connected to the outputs:

$$w_{ij}^{out} \leftarrow w_{ij}^{out} + \eta \delta_i^{out} y_j^{\mathrm{hid}}$$

learning rate

  - For the weights connected to the hidden nodes:

$$w_{jk}^{hid} \leftarrow w_{jk}^{hid} + \eta \delta_j^{hid} x_k$$

Linear Transfer Function on Hidden Layer?

- MLP with one hidden layer (*L=1*), one output layer (*L=2*):

$$y^{L=2} = g^{L=2}\left(W^{L=2} \cdot g^{L=1}\left(W^{L=1} \cdot x^{L=0}\right)\right)$$

- Let's make the hidden layer linear: $g^{L=1}(h) = h$

$$y^{L=2} = g^{L=2}\left(\underbrace{W^{L=2} \cdot W^{L=1}}_{W^{eff}} \cdot x^{L=0}\right)$$

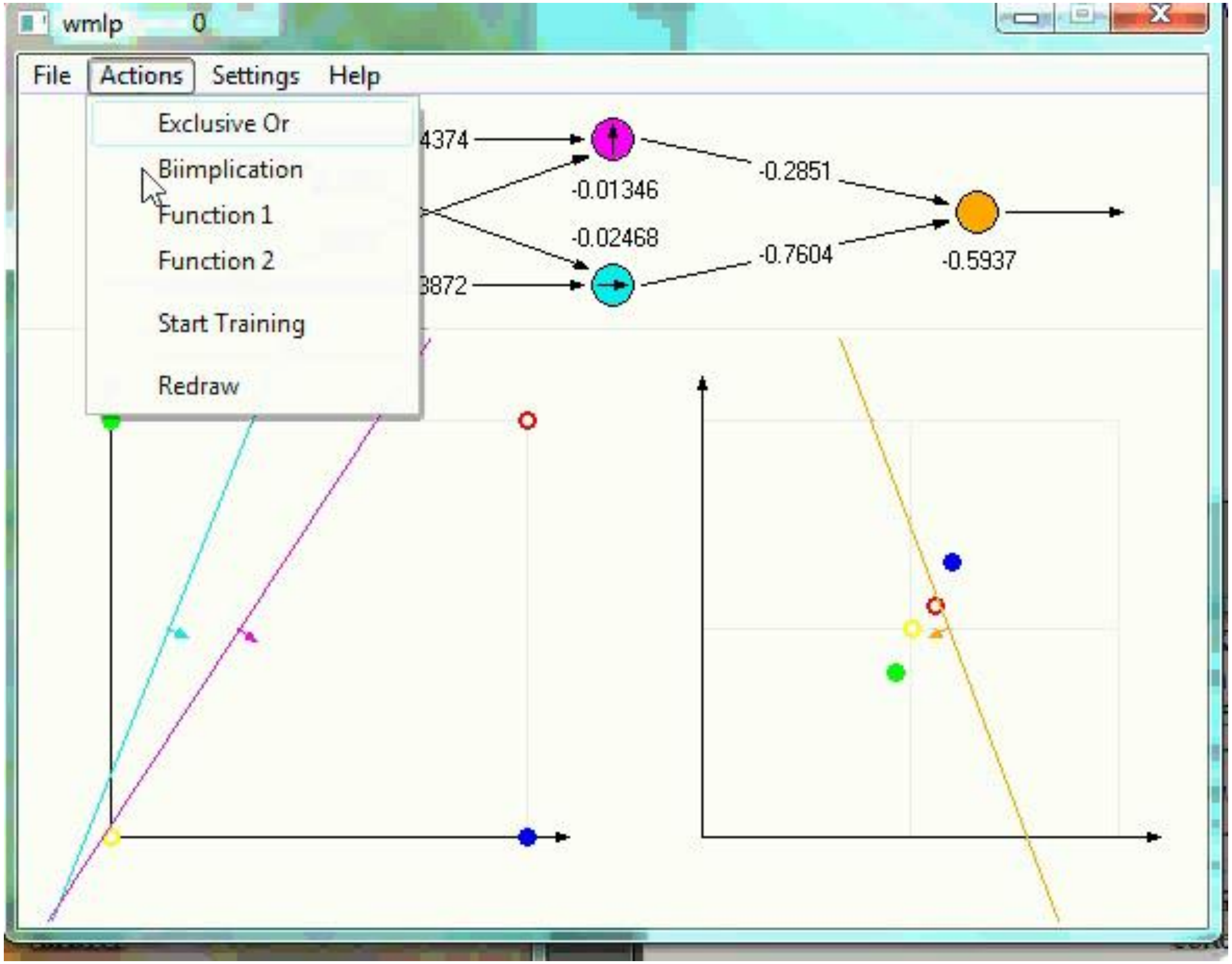
now both weight matrices multiply, becoming one effective matrix

- This yields a perceptron:

we have "lost" the hidden layer!

$$y^{L=2} = g^{L=2}\left(W^{eff} \cdot x^{L=0}\right)$$

MLP training a XOR problem

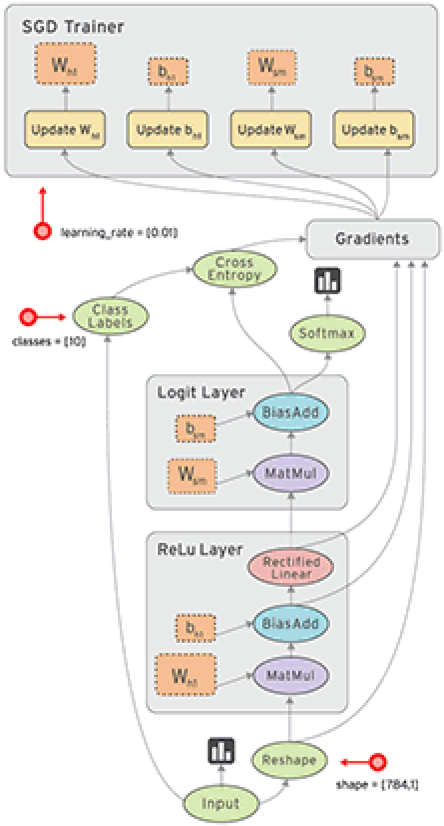


[http://www.borgelt.net/mlpd.html]

Tensorflow

- Open source package for deep MLP learning by google
- Given a network structure and cost function:

  → does automatic differentiation and learning
- Online demo for small networks:
  - http://playground.tensorflow.org



Overview

- Multilayer Perceptron
  - Error Function & Gradient
  - Gradient Descent Learning
  - Transfer Functions
  - Learning Capacity & Generalization
  - Testing

Network Topology

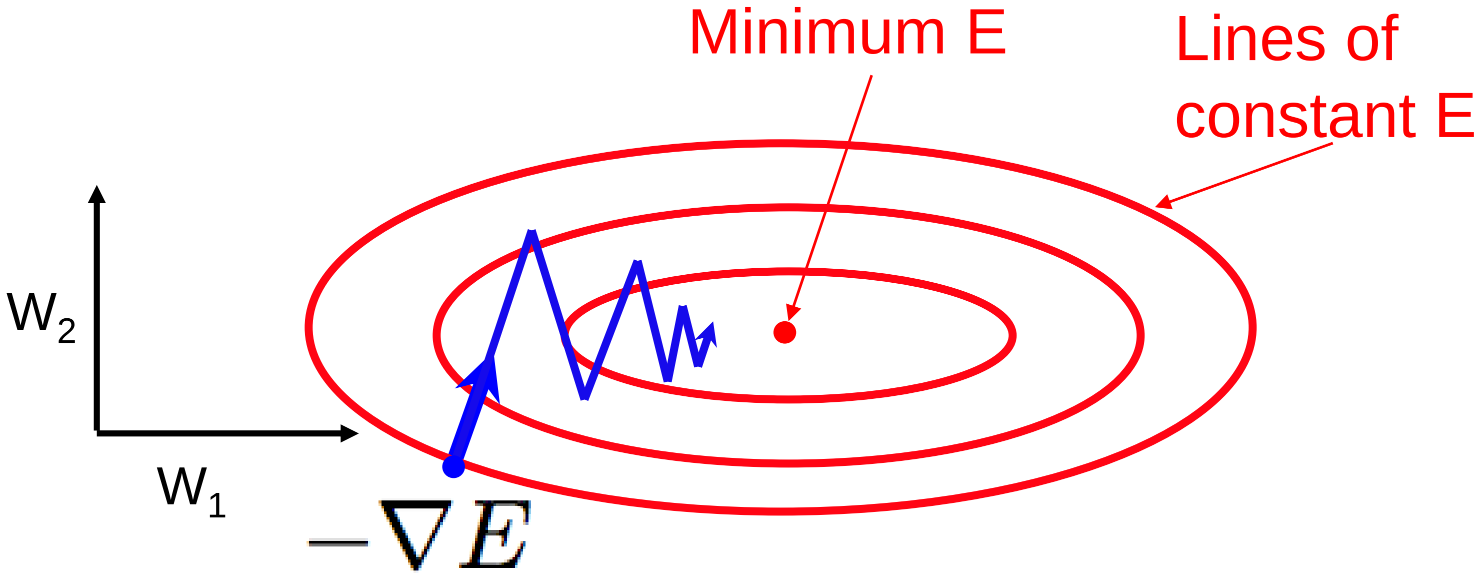
- How many layers?
- How many neurons per layer?
- Experiments
  - Often two or three hidden layers (but new research into deep learning networks…)
  - Determine size of layers (usually get smaller)
  - Test several different networks

Batch and Online Learning

- When should the weights be updated?
  - After all inputs seen (***batch,*** *(proper)* ***gradient descent***)
    - Converges systematically to the (local) minimum
    - Requires many epochs (passes through the whole dataset)
  - After each input is seen (***online, incremental, stochastic gradient descent***)
    - Simpler to program
    - Handles infinite amount of data (continual learning)
    - Noise may help escaping from saddle points in the energy landscape, or even from local minima
    - Pitfall: data distribution may drift.

      Remedy: randomize order of presentation

Gradient Descent Dynamics (I/II)

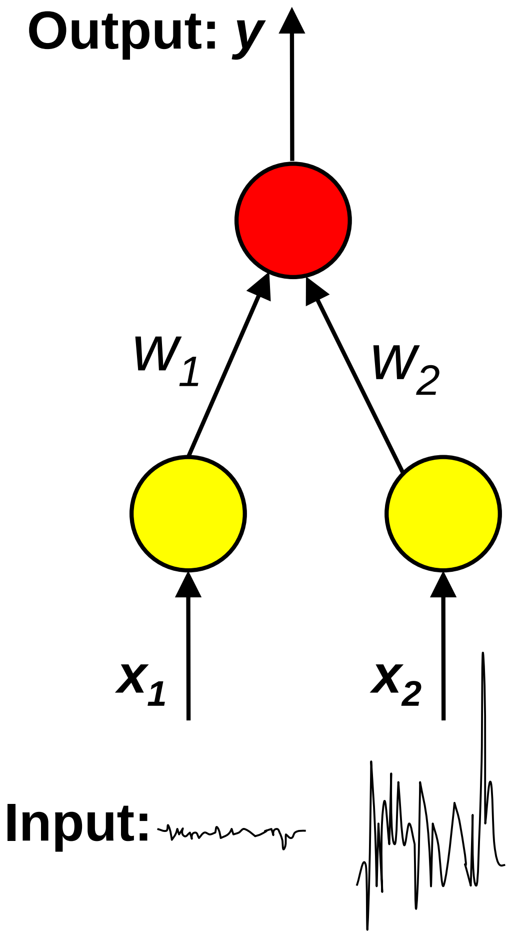


- Local gradient does not point towards minimum
- Gradient descent with large learning rate
  $\rightarrow$ oscillations
- Long learning time!

Gradient Descent Dynamics (II/II)

- Learning rule (for the simple case of the perceptron):

$$-\frac{\partial E}{\partial w_{ij}} = (t_i - y_i) \cdot x_j$$



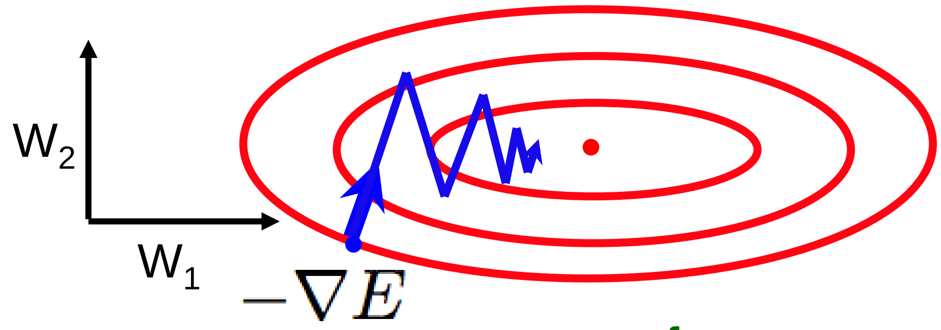
- Assume: inputs x_1 and x_2 are of similar importance for classification
- both have mean zero: $\mu(x_1) = \mu(x_2) = 0$
- variances of signals differ: $\sigma(x_1) < \sigma(x_2)$

$\rightarrow$ weights should be: $w_1 > w_2$

but average updates:

$|\Delta w_1| < |\Delta w_2|$

Momentum Term

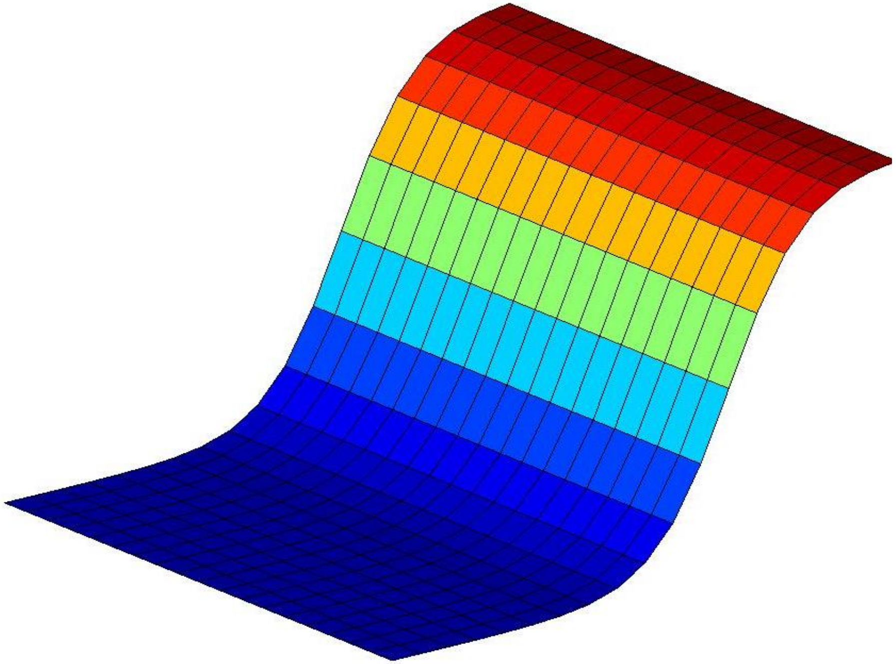


$$w_{ij}^{\tau} \leftarrow w_{ij}^{\tau-1} + \eta\delta_i y_j^{\text{hid}} + \underbrace{\overset{\text{free parameter}}{\alpha}\Delta w_{ij}^{\tau-1}}$$

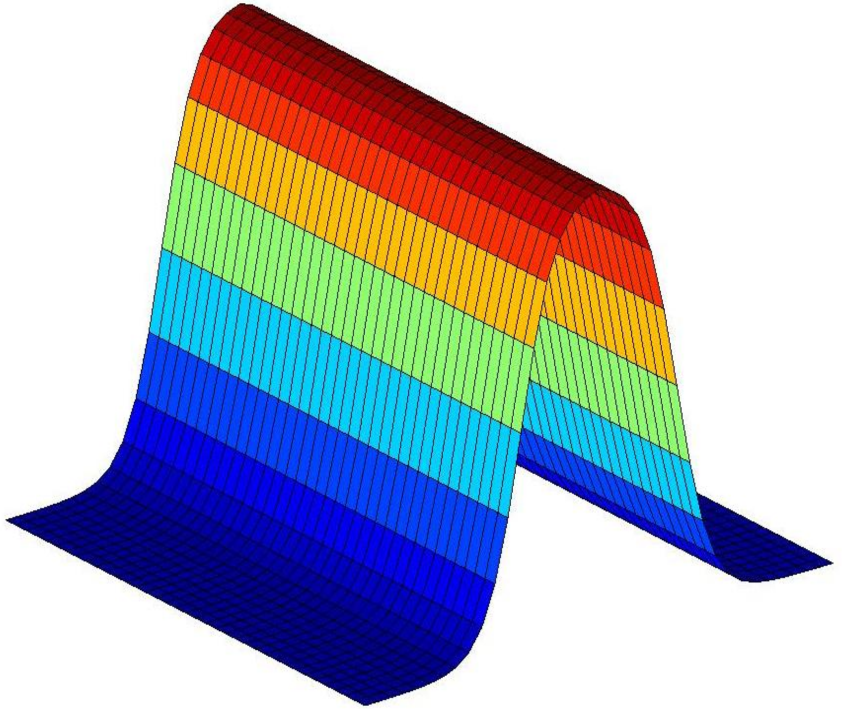
Add contribution from previous weight change (momentum)

- Counteracts oscillations, by averaging previous and current updates (relevant for batch learning)
- Averages out noise
  - relevant for on-line learning
- More stable, leads to faster learning

Learning Capacity



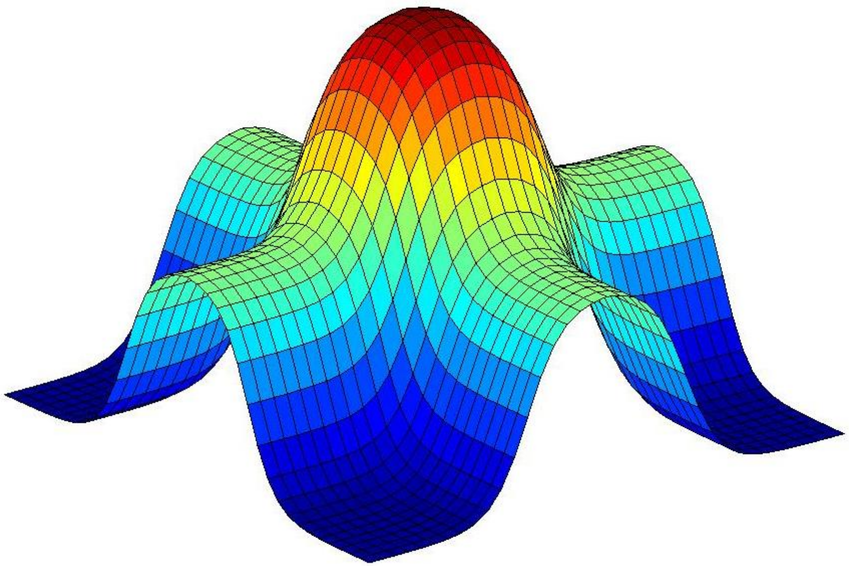
Output of one sigmoid

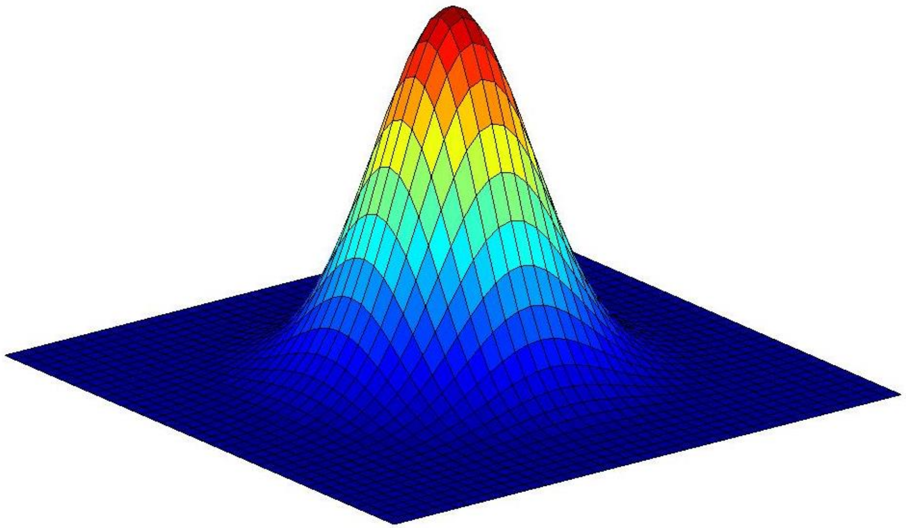


Output of two sigmoids

Learning Capacity

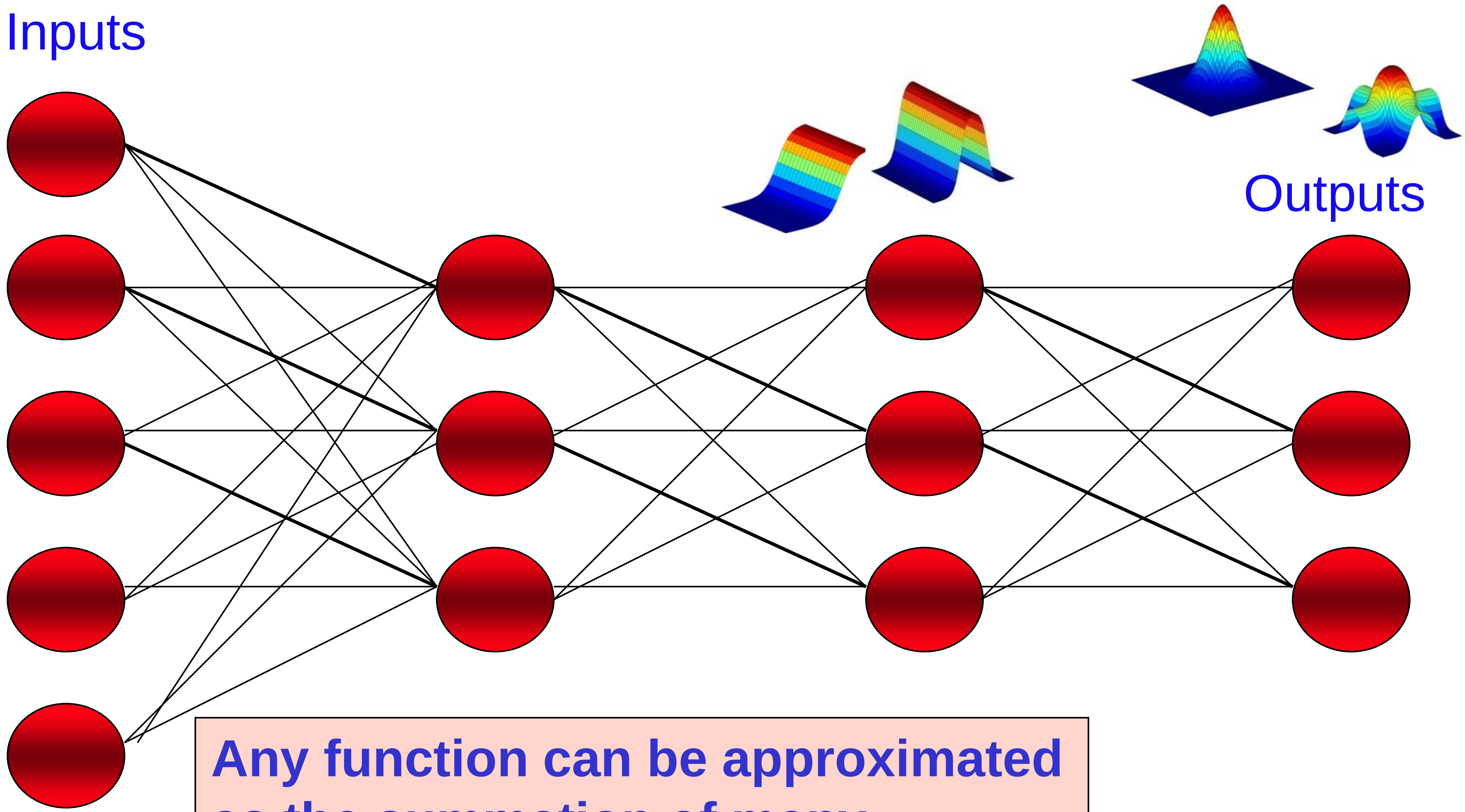
Addition of more ridges and transformation with another sigmoid → Localised response





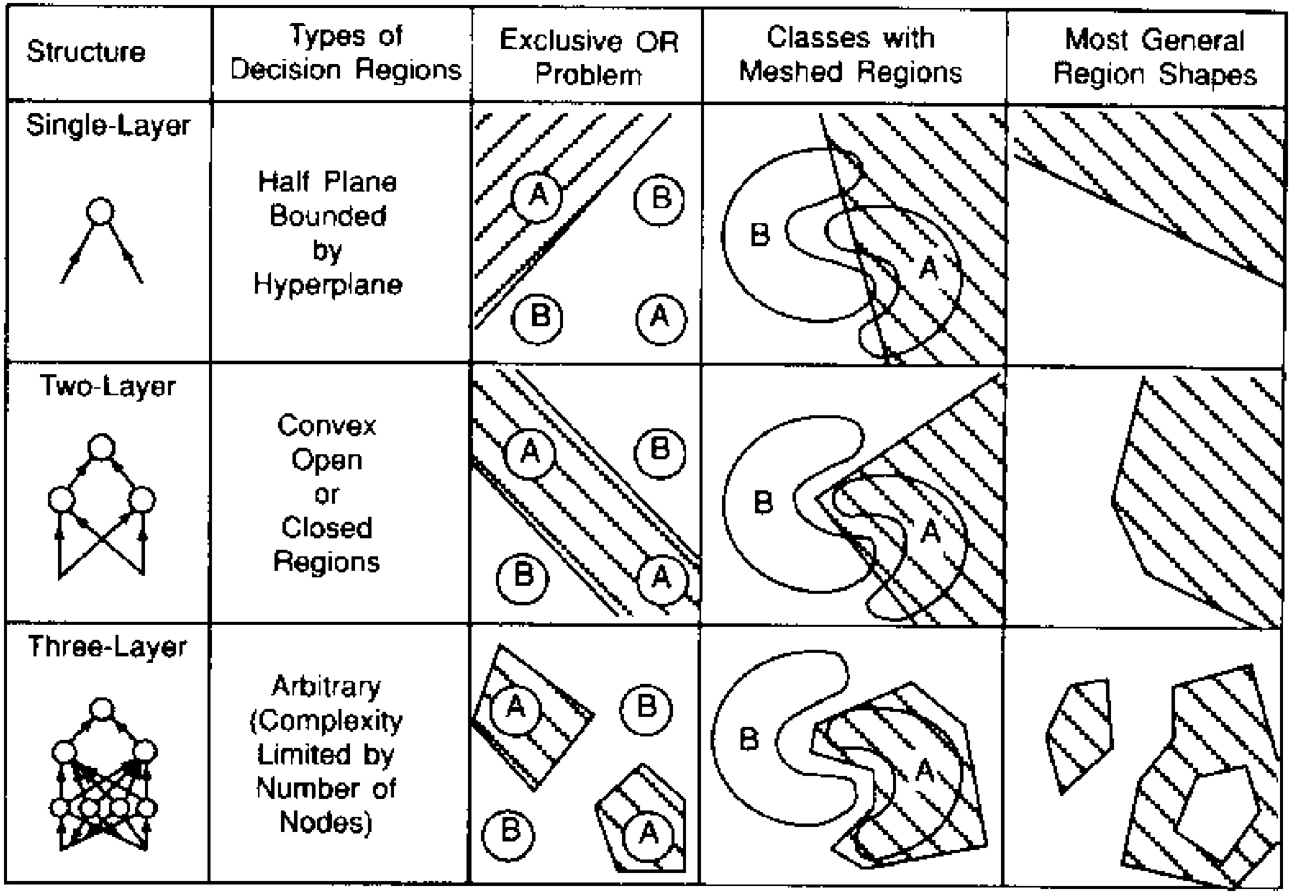
Addition of two ridges
Unique maximum

Learning Capacity



Any function can be approximated as the summation of many localised responses

Decision Boundaries (Lippmann)



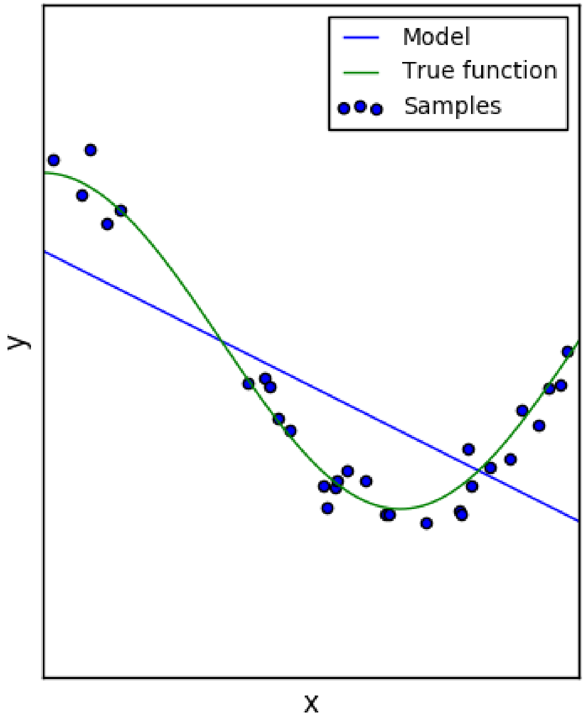
Structure	Types of Decision Regions	Exclusive OR Problem	Classes with Meshed Regions	Most General Region Shapes
Single-Layer	Half Plane Bounded by Hyperplane			
Two-Layer	Convex Open or Closed Regions			
Three-Layer	Arbitrary (Complexity Limited by Number of Nodes)			

Generalisation

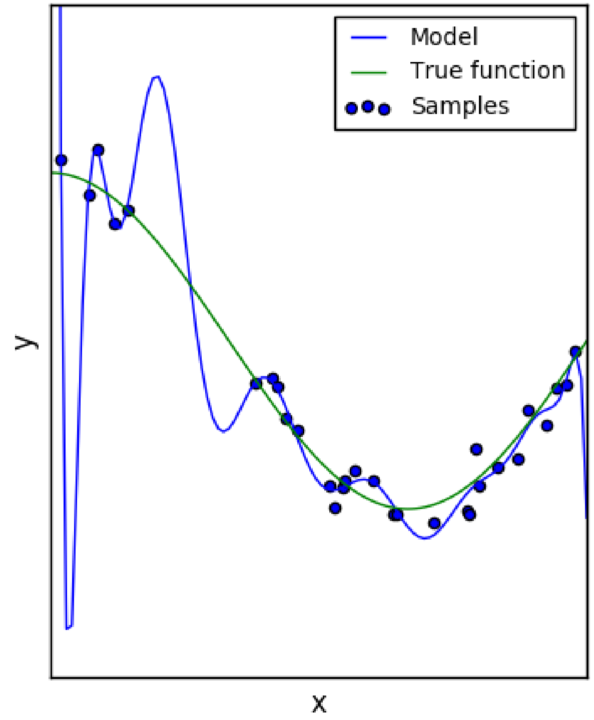
- Aim of neural network learning:

  Generalise from training examples to all possible inputs

- Undertraining is bad



- Overtraining is worse



Overview

- Multilayer Perceptron
  - Error Function & Gradient
  - Gradient Descent Learning
  - Transfer Functions
  - Learning Capacity & Generalization
  - Testing

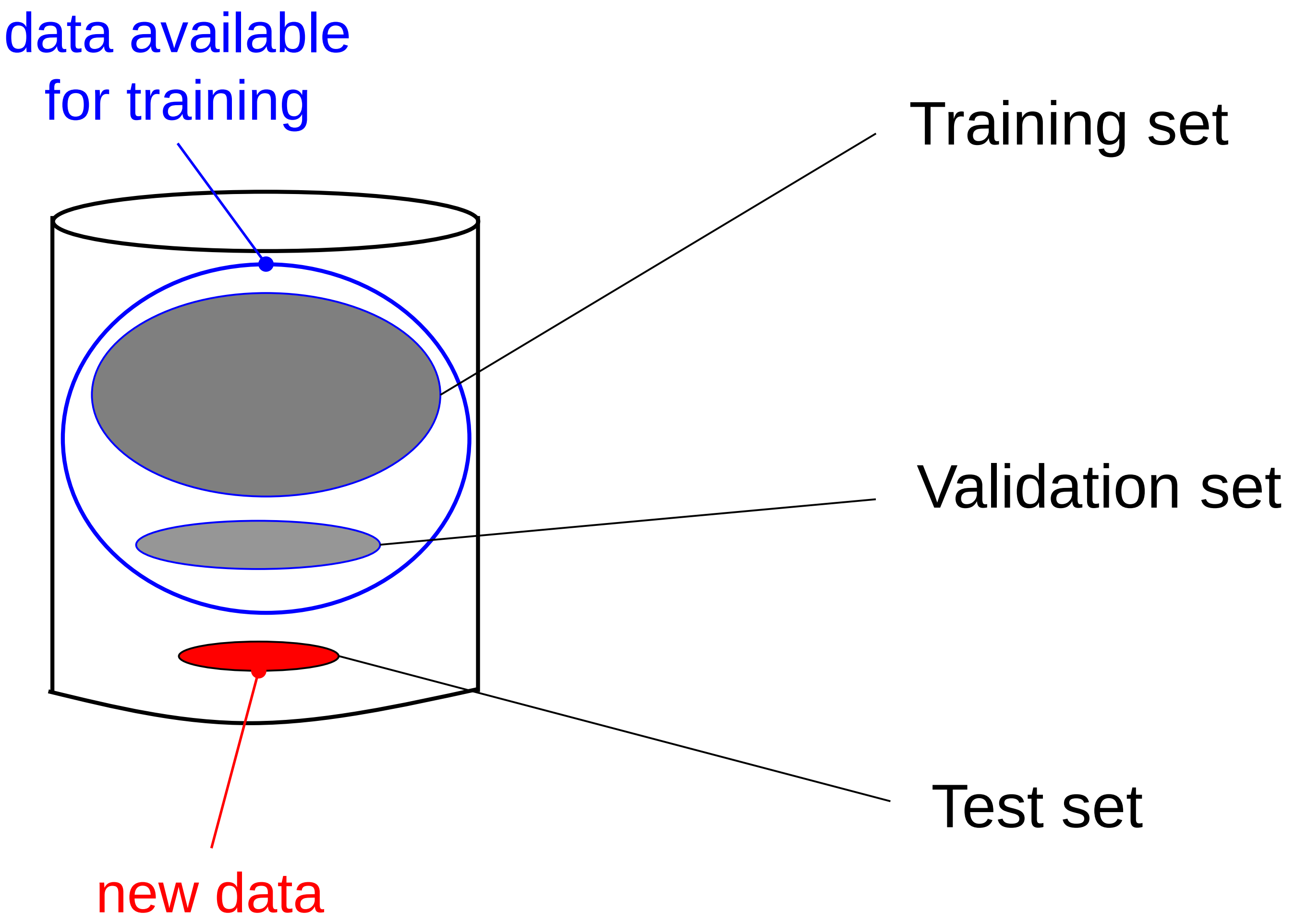
Validation and Testing

How do we evaluate our trained network?

- The error on the training data is biased and hides overfitting
- Validate on a separate validation set
  - evaluate periodically on this validation set during training (while training only on the training set)
  - indicator of overfitting: the validation error increases
- After training, test the final model on the test set

- This may come expensive on data!

Using Training, Validation and Test Data

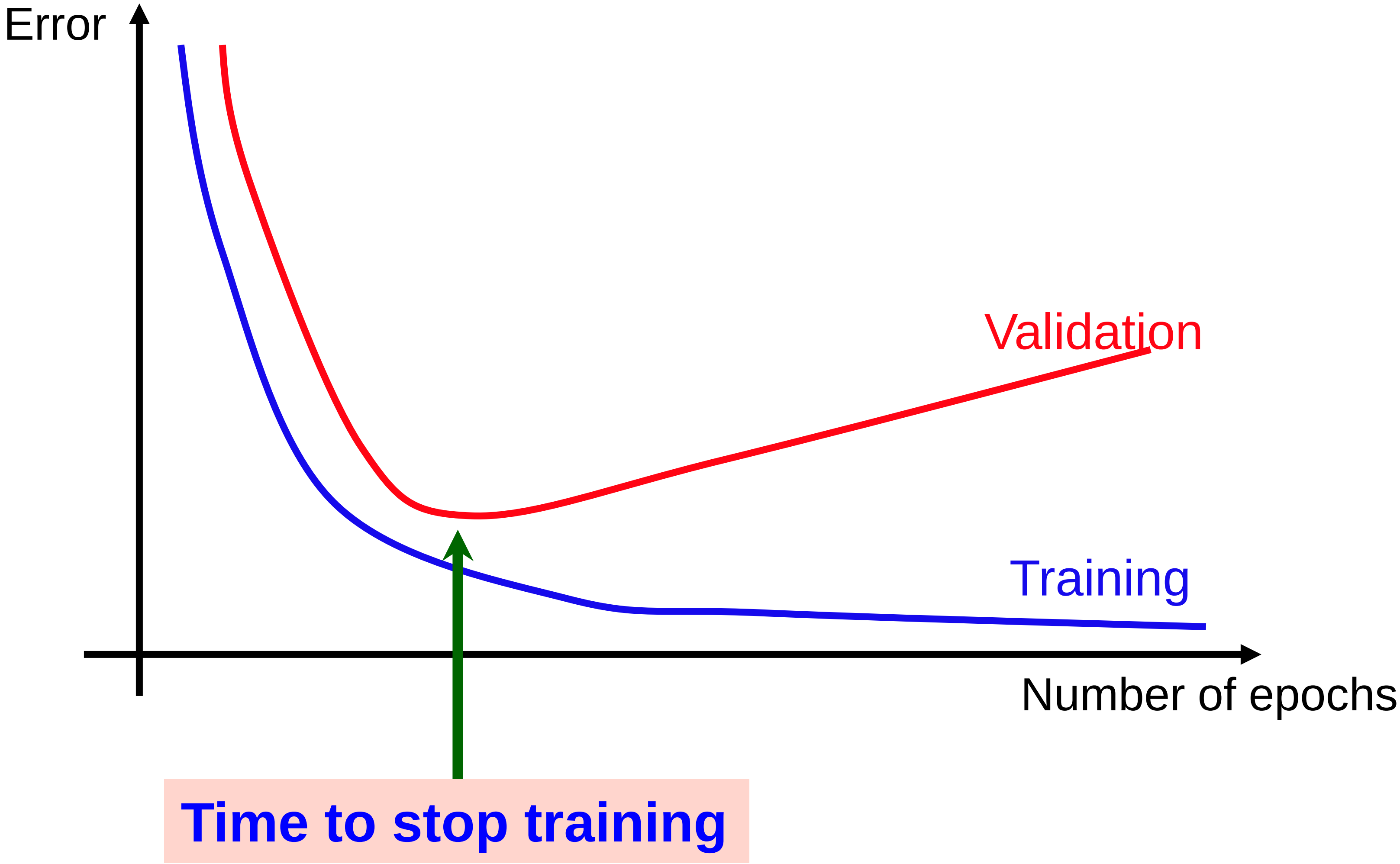


Early Stopping

When should we stop training?

- Could set a maximum training error
  - Danger of overfitting
- Could set a number of epochs
  - Danger of underfitting or overfitting
- Can use the validation set
  - Measure the error on the validation set during training
  - Idea: *validation error will get higher as network starts overfitting* to the training set

Early Stopping



Summary: Neural Networks as Classifiers

- Weaknesses
  - Several parameters have to be set empirically, e.g.
    - network topology, transfer functions, learning rate, etc.
  - Black box: hard to interpret hidden units and learned weights
  - Long training time
  - Cannot handle well missing values
- Strengths
  - Successful on a wide array of real-world data
    - Well-suited for continuous-valued inputs and outputs
    - High tolerance to noisy data
    - Generalisation ability: classify untrained patterns
  - Algorithms are inherently parallel
  - Neuron activations and weight vectors sometimes interpretable
  - Relationship to brain