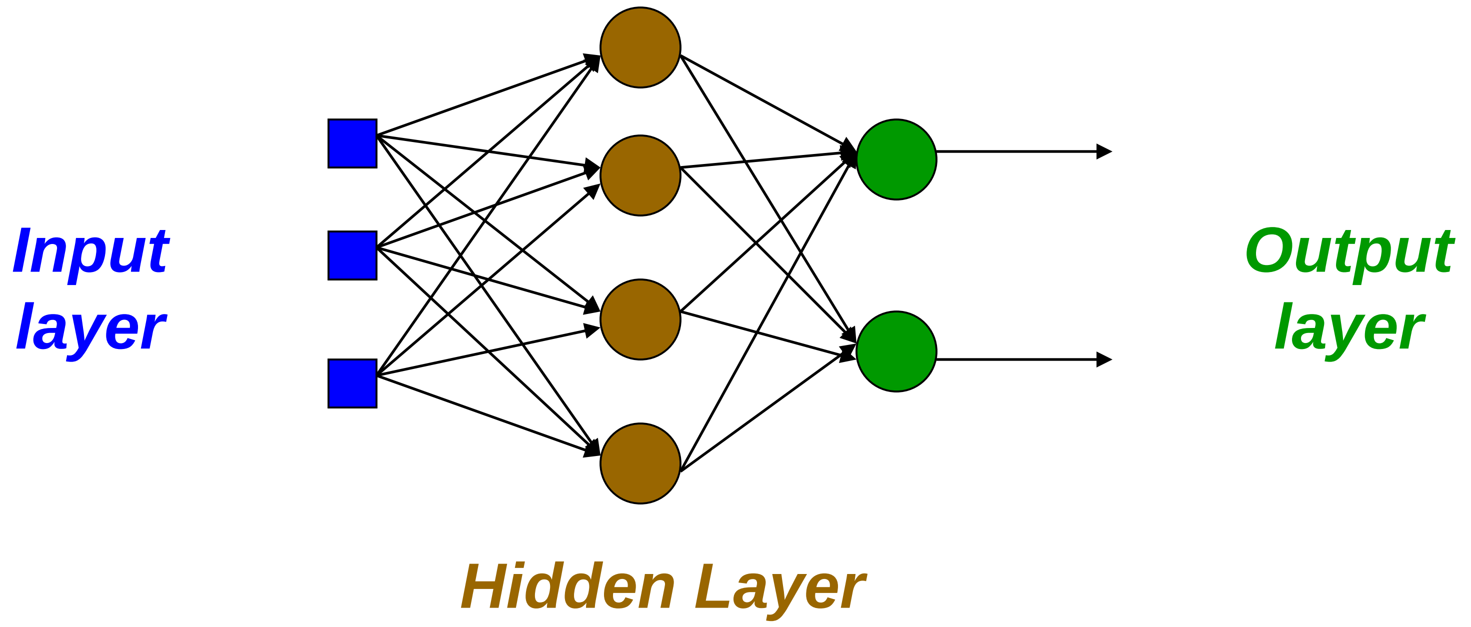
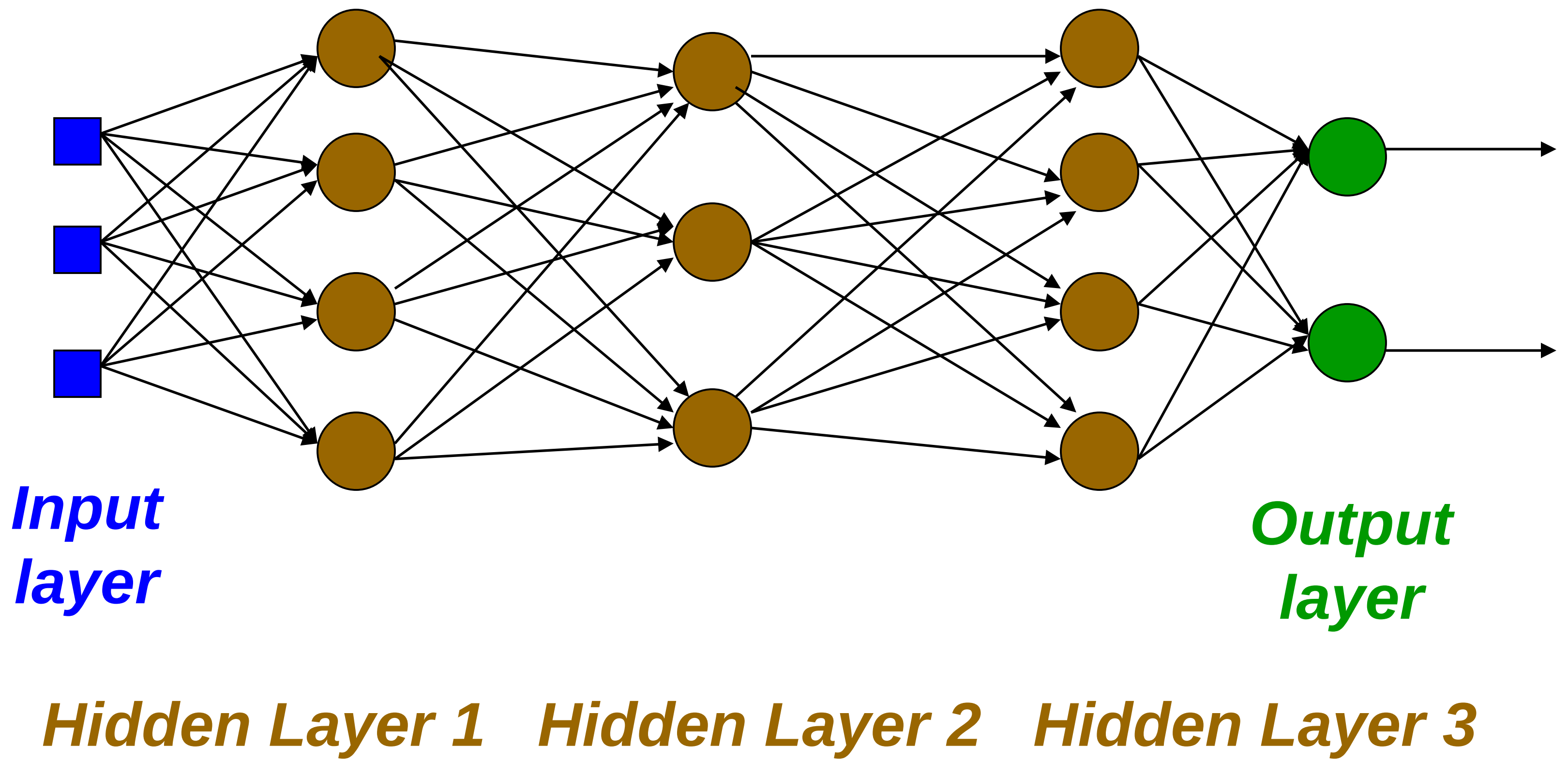
Multi-layer Perceptron (MLP) and Backpropagation

- Single perceptrons = linear decision boundary
- The XOR problem cannot be solved by a single perceptron
- MLPs overcome the limitation of single-layer perceptrons
- How to train them ? Backpropagation!

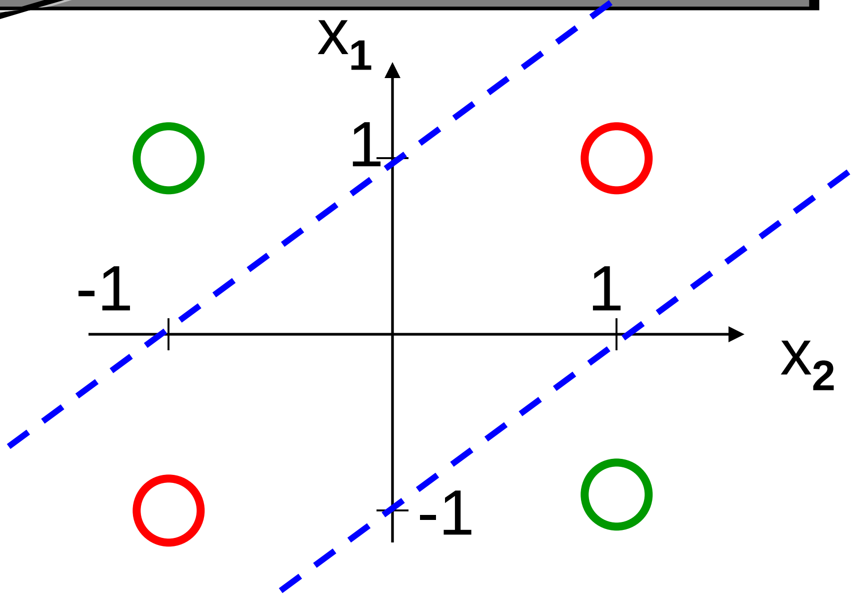


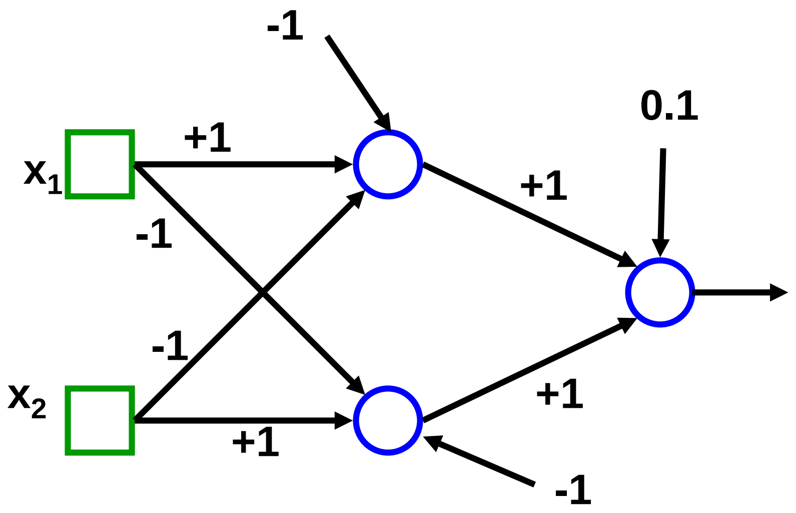
A 3:4:3:4:2 network



A solution for the XOR problem

x_1	x_2	x_1 xor x_2
-1	-1	**-1**
-1	1	**1**
1	-1	**1**
1	1	**-1**

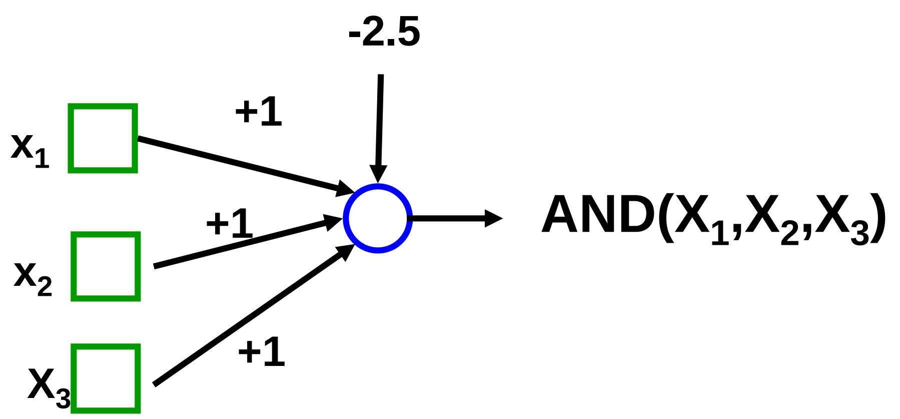


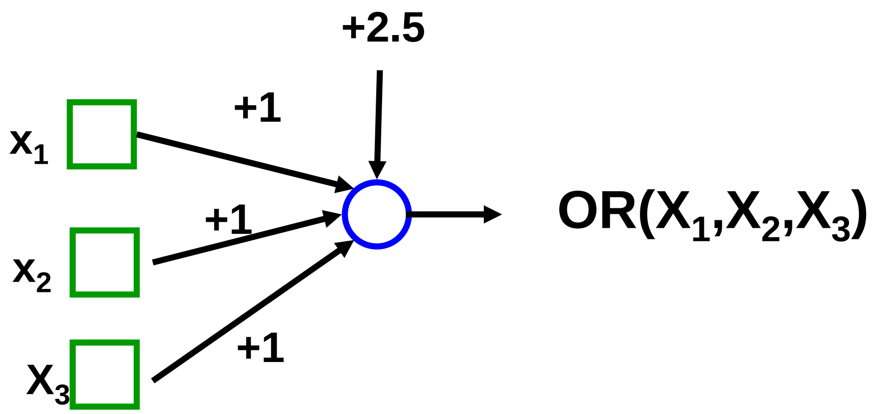


$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ -1 & \text{if } v \le 0 \end{cases}$$

φ is the sign function.

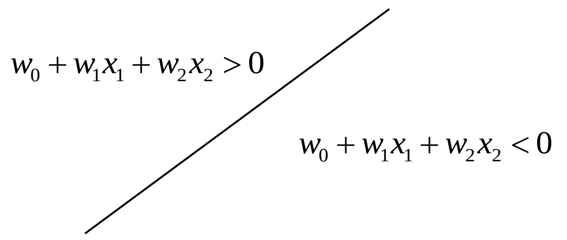
Generalized AND and OR

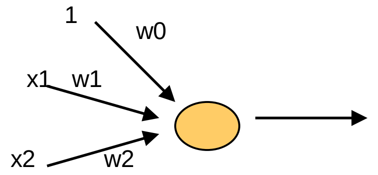




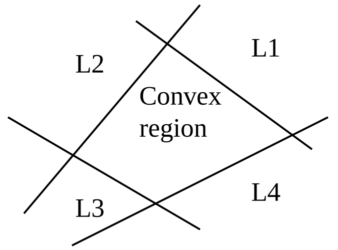
Generalized AND (OR) can be computed by a single perceptron!

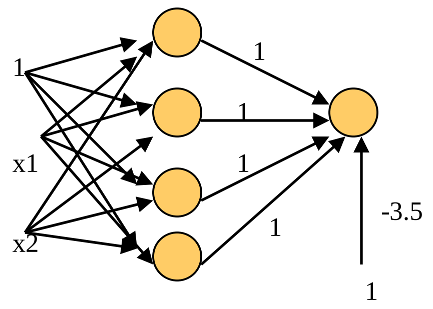
Types of decision regions



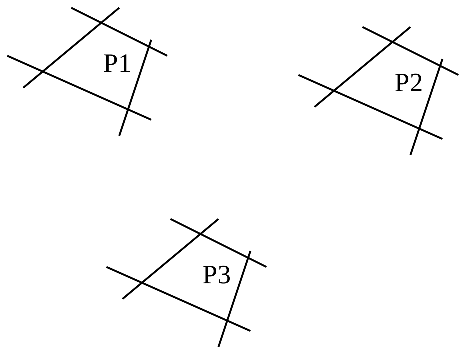


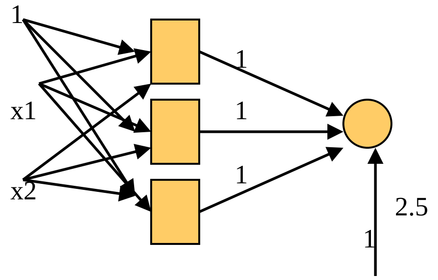
Network with a single node





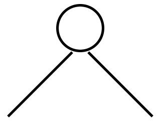
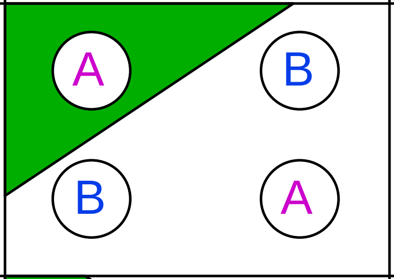
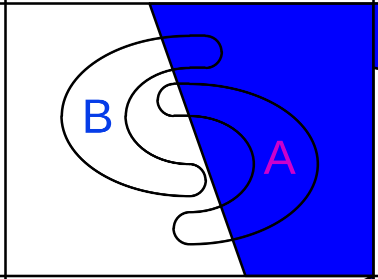
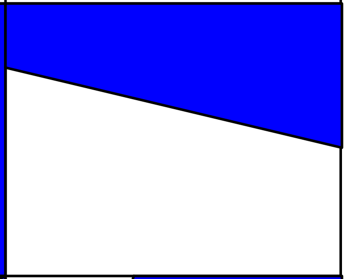
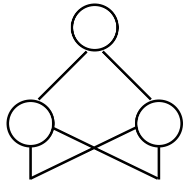
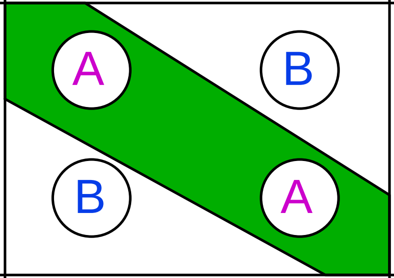
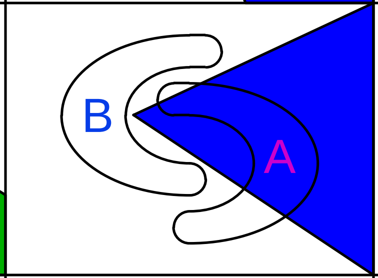
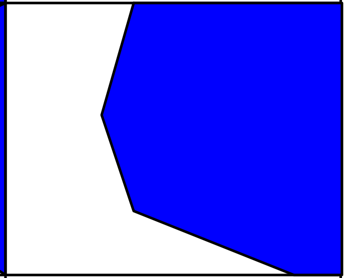
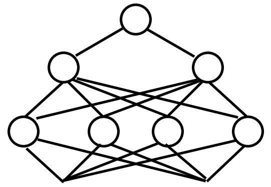
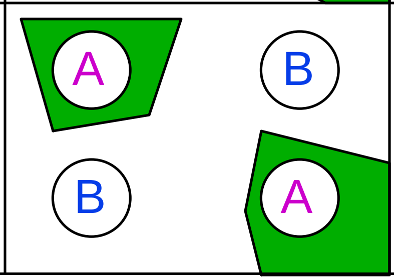
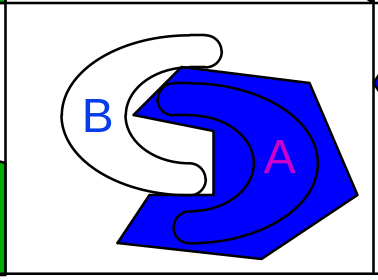
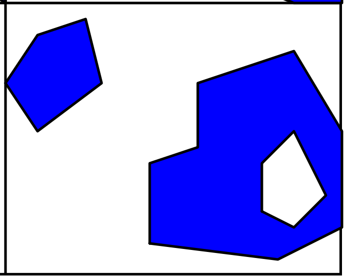
One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region





two-hidden layer network that realizes the union of three convex regions: each box represents a one hidden layer network realizing one convex region

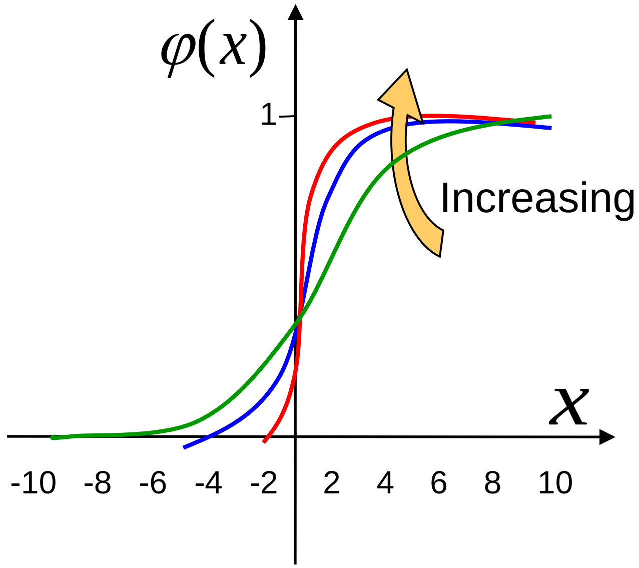
Different Non-Linearly Separable Problems

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 No. of Nodes)			

How to train multi-layer networks?

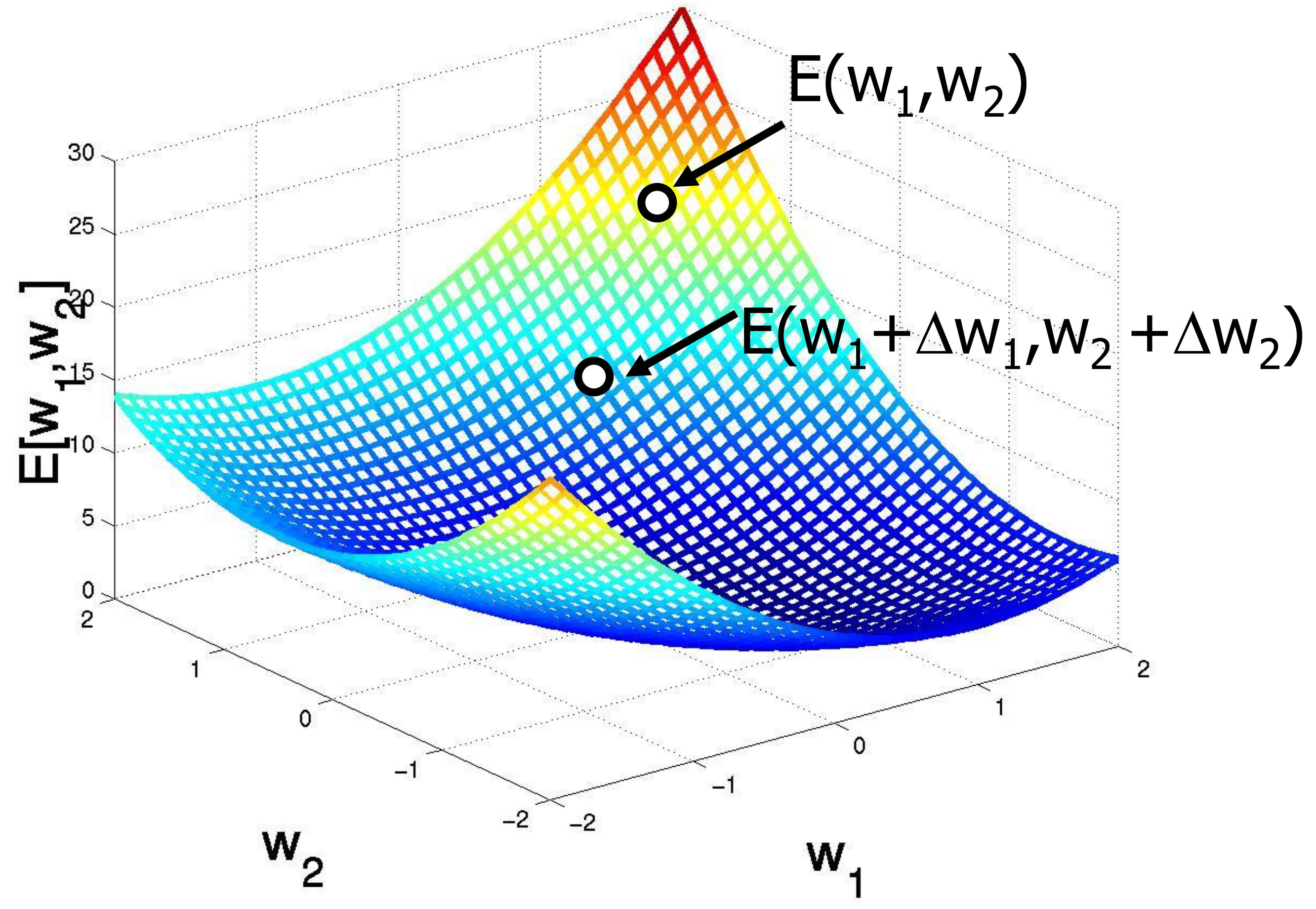
- Main Idea:
  Replace the sign function by its “smooth approximation” and use the gradient descent algorithm to find weights that minimize the error (as with Logistic Regression)

$$\varphi(\mathrm{x}) = \frac{1}{1+e^{-ax}} \quad \text{with } a > 0$$



- The function to be optimized is a complicated one, with many variables and nestings …
- FORTUNATELY: the final update rules are simple !

Gradient Descent



Weight Update Rule

gradient descent method: "walk" in the direction yielding the maximum decrease of the network error E. This direction is the opposite of the gradient of E.

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

direction (negative gradient)

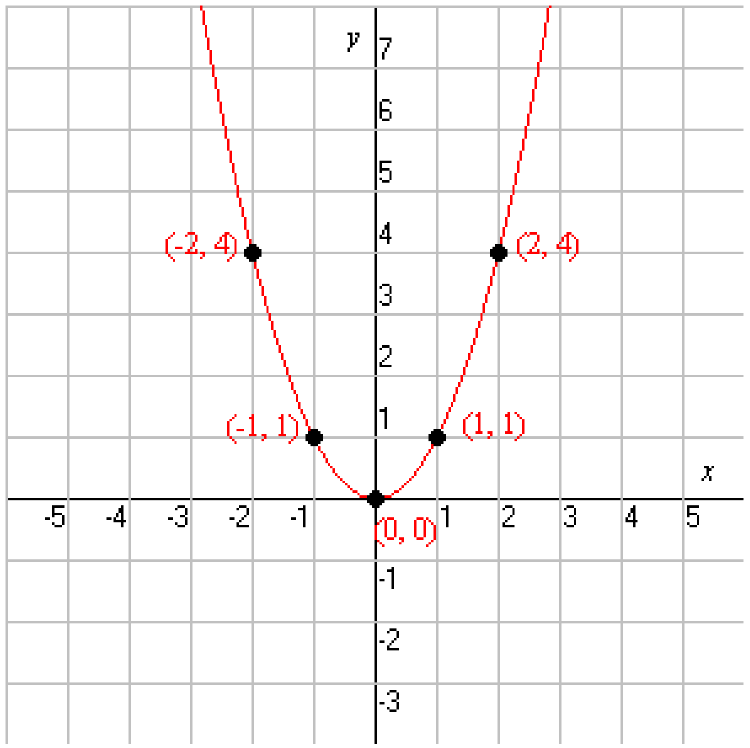
$$w_{ji} = w_{ji} + \Delta w_{ji}$$

update rule

$$w_{ji} = w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}$$

could be written in one rule!?
(in practice we first calculate all gradients and then update all weights)

Example: *find the minimum of $f(x)=x^2$*



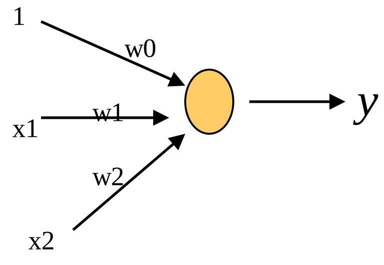
- Start at $x_0=1.0$ (or $x_0=2.0$)
- *$f'(x)=2x$, so the update rule:*

$$x_{n+1}= x_n- \eta 2x_n$$

- *Consider several scenario's:*
  - $\eta=1.5$
  - $\eta=1.0$
  - $\eta=0.75$
  - $\eta=0.5$
  - $\eta=0.25$

Delta Rule: one node

- Derive the Delta rule for the following network



$y = \varphi(net) = \varphi(w_0 + w_1 x_1 + w_2 x_2)$

$$E(w_0, w_1, w_2) = \frac{1}{2}(y-d)^2 = \frac{1}{2}(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ for } i \text{ in } \{0,1,2\}$$

- We need to find $\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$

Delta Rule: one node

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_1} = \frac{1}{2}\frac{\partial(\varphi(w_0 + w_1x_1 + w_2x_2) - d)^2}{\partial w_1}$$

$$= \frac{1}{2}2(\varphi(w_0 + w_1x_1 + w_2x_2) - d)\frac{\partial(\varphi(w_0 + w_1x_1 + w_2x_2) - d)}{\partial w_1}$$

$$= (\varphi(net) - d)\varphi'(net)\frac{\partial(w_0 + w_1x_1 + w_2x_2)}{\partial w_1}$$

$$= (\varphi(net) - d)\varphi'(net)x_1$$

- From similar calculations we get:

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_2} = (\varphi(net) - d)\varphi'(net)x_2$$

- and $$\frac{\partial E(w_0, w_1, w_2)}{\partial w_0} = (\varphi(net) - d)\varphi'(net)$$

Delta Rule: one node

Thus the update rules for the weights are:

$$\Delta w_0 = \eta(d - \varphi(net))\varphi'(net) \cdot 1$$

$$\Delta w_1 = \eta(d - \varphi(net))\varphi'(net) x_1$$

$$\Delta w_2 = \eta\underbrace{(d - \varphi(net))}_{\textit{delta}}\varphi'(net) \underbrace{x_2}_{\textit{input}}$$

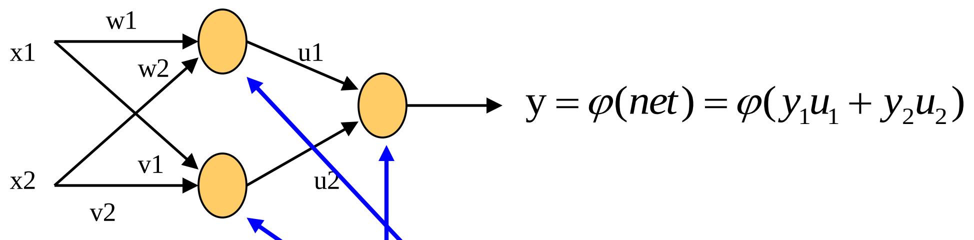
It’s handy to know that for the logistic sigmoid function:

φ(x)=*1/(1+exp(-x))* we have: φ′(x)=φ(x)(1-φ(x))=*output(1-output),*

so once we know φ(x) we also know φ′(x) *!*

Delta Rule: XOR-network

Consider the network (no biases):



$$net_1 = w_1 x_1 + w_2 x_2, \quad y_1 = \varphi(net_1)$$

$$net_2 = v_1 x_1 + v_2 x_2, \quad y_2 = \varphi(net_2)$$

$$net = y_1 u_1 + y_2 u_2, \quad y = \varphi(net)$$

$$= \varphi(y_1 u_1 + y_2 u_2) =$$

$$= \varphi(\varphi(net_1) u_1 + \varphi(net_2) u_2) =$$

$$= \varphi(\varphi(w_1 x_1 + w_2 x_2) u_1 + \varphi(v_1 x_1 + v_2 x_2) u_2)$$

Derivation of the Delta Rule

$$\frac{\partial E}{\partial u_1} = \frac{1}{2}\frac{\partial(\varphi(y_1u_1 + y_2u_2) - d)^2}{\partial u_1} = \frac{1}{2}2(\varphi(y_1u_1 + y_2u_2) - d)\frac{\partial(\varphi(y_1u_1 + y_2u_2) - d)}{\partial u_1}$$

$$= (y - d)\varphi'(net)\frac{\partial(y_1u_1 + y_2u_2)}{\partial u_1} = (y - d)\varphi'(net)y_1$$

From similar calculations we get: $\frac{\partial E}{\partial u_2} = (y - d)\varphi'(net)y_2$

$$\frac{\partial E}{\partial w_1} = \frac{1}{2}\frac{\partial(\varphi(y_1u_1 + y_2u_2) - d)^2}{\partial w_1} = \frac{1}{2}2(\varphi(y_1u_1 + y_2u_2) - d)\frac{\partial(\varphi(y_1u_1 + y_2u_2) - d)}{\partial w_1}$$

$$= (y - d)\varphi'(net)\frac{\partial(y_1u_1 + y_2u_2)}{\partial w_1}$$

Derivation of the Delta Rule

$$\frac{\partial(y_1u_1+y_2u_2)}{\partial w_1}=\frac{\partial(y_1u_1)}{\partial w_1}=u_1\frac{\partial y_1}{\partial w_1}=u_1\frac{\partial(\varphi(w_1x_1+w_2x_2))}{\partial w_1}$$

$$=u_1\varphi'(net_1)\frac{\partial(w_1x_1)}{\partial w_1}=u_1\varphi'(net_1)x_1$$

So we obtain:

$$\frac{\partial E}{\partial w_1}=(y-d)\varphi'(net)\frac{\partial(\varphi(y_1u_1+y_2u_2))}{\partial w_1}$$

$$=(y-d)\varphi'(net)u_1\varphi'(net_1)x_1$$

From similar calculations we get:

$$\frac{\partial E}{\partial w_2}=(y-d)\varphi'(net)u_1\varphi'(net_1)x_2$$

delta

$$\frac{\partial E}{\partial v_1}=(y-d)\varphi'(net)u_2\varphi'(net_2)x_1,\qquad \frac{\partial E}{\partial v_2}=(y-d)\varphi'(net)u_2\varphi'(net_2)x_2$$

Key idea behind Backpropagation

- The output of a MLP is a composition of two sorts of functions:
  - Linear (weighted) combinations of inputs
  - Activation functions (e.g., sigmoid, logistic, etc.)
  - *Additionally, a error function (loss function) at the output layer*

- Backpropagation is based on an observation that computing partial derivatives of the error function involves multiple application of the chain rule applied to simple functions (linear and sigmoid) which leads to a very efficient algorithm that involves 3 phases:

  - Computing the output of the network and the corresponding error,
  - Computing the contribution of each weight to the error,
  - Adjusting the weights accordingly (to the contribution to error).

General case: Generalized Delta Rule

$\Delta w_{ji} = \eta \delta_j y_i$, where:

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF j output node} \\ \varphi'(v_j) \sum\limits_{\text{k of next layer}} \delta_k w_{kj} & \text{IF j hidden node} \end{cases}$$

w_{ji} *: weight from node j to node i (moving from output to input!)*

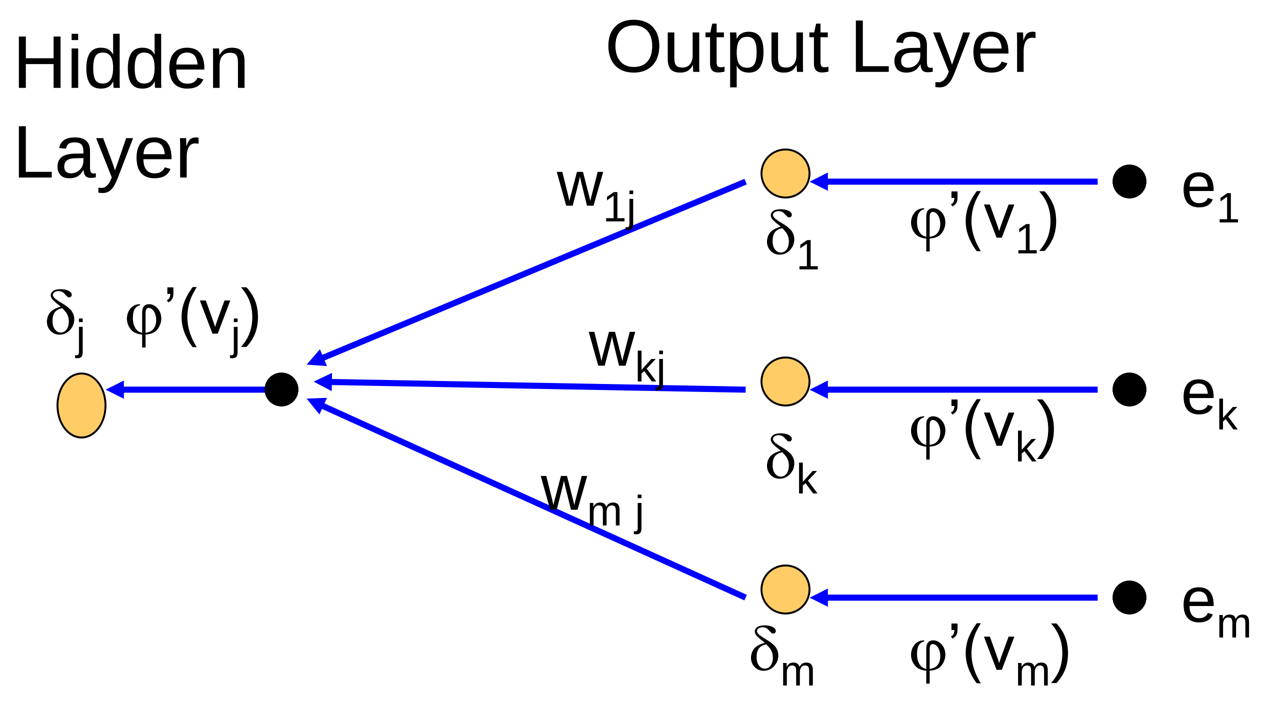
η *: learning rate (constant)*

y_j *: output of node j*

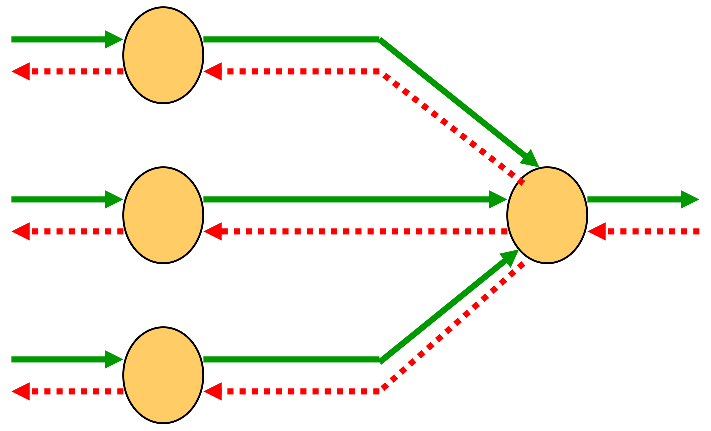
v_j *: activation of node j*

Error backpropagation

The flow-graph below illustrates how errors are back-propagated from ***m*** output nodes to the hidden neuron ***j***



Backpropagation: two phases



→ *Network activation*
Error computation
Forward Step

⇠ *Error propagation*
Backward Step

Backpropagation algorithm

- The Backprop algorithm searches for weight values that **minimize the total error of the network**

- Backprop consists of the **repeated** application of the following two passes:
  - **Forward pass**: in this step the network is activated on **one example** and the **error** of each neuron of the output layer is computed. Also **activations** of all hidden nodes are computed.

  - **Backward pass**: in this step the network error is used for updating the weights. Starting at the output layer, **the error is propagated backwards through the network, layer by layer** with help of the generalized delta rule. Finally, **all weights are updated**.

- Weights are initialized at random (how?)

- No guarantees of convergence (why?)
  (when learning rate too big or too small)

- In case of convergence: local (or global) minimum

Error function is the sum of errors over training examples, so the gradient of the error function is the sum of gradients over all training cases!

Three update strategies:

Full Batch mode (all inputs at once; conceptually "correct")
Weights are updated after all the inputs are processed

Batch mode (a small, random sample of inputs; "approximate")
Weights are updated after all a small random sample of inputs is processed (Stochastic Gradient Descent)

On-line mode (one input at a time)
Weights are updated after processing single inputs

Advantages Stochastic Gradient Descent

- **Additional randomness helps to avoid local minima**
- **Huge savings of the CPU-time**
- **Easy to execute on GPU cards**
- “Approximated gradient” works almost the same as “exact gradient” (almost the same convergence rate)

Stopping criteria

– total mean squared error change: Backprop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small

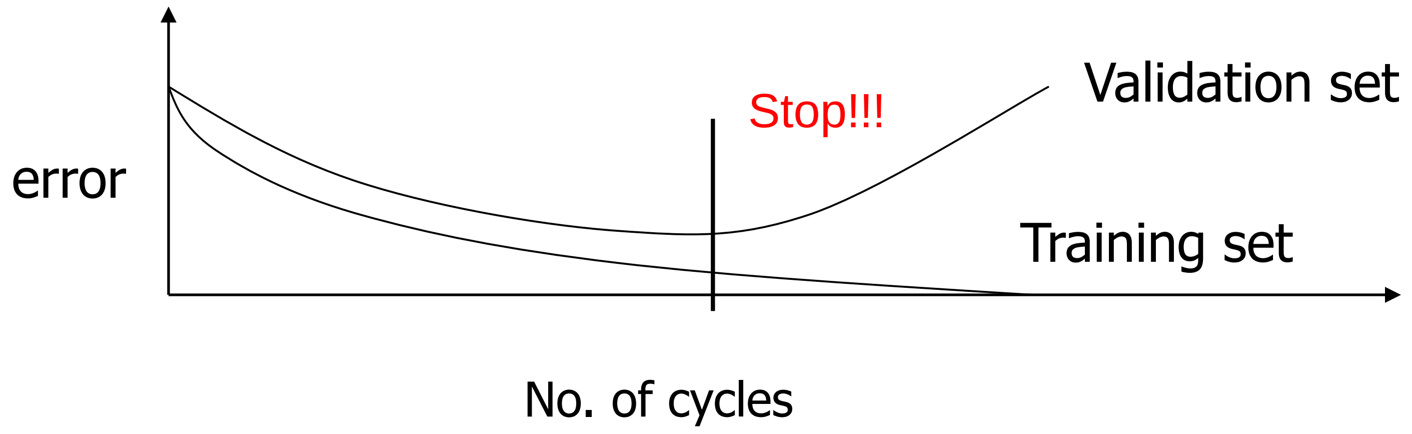
– generalization based criterion: After each epoch the NN is tested for generalization using a different set of examples (validation set). If the generalization performance is adequate then stop. (**Early Stopping**)

Early Stopping

Training network for too many cycles may lead to data overfitting (or “overtraining”)

Early Stopping:

stop training as soon at the error on the validation set increases



Early stopping and 3 sets:

- Training set – used for training
- Validation set – used to decide when to stop training by monitoring the error on it; not used for training!
- Test set – used for final estimation of the performance of the trained neural network

Model Selection by Cross-validation

- Too few hidden units may prevent the network from learning adequately the data and learning the concept.
- Too many hidden units leads to overfitting.
- A cross-validation scheme can be used to determine an appropriate number of hidden units by using the optimal test error to select the model with optimal number of hidden layers and nodes.
- N-fold cross-validation:
  1. split your data into N parts (equal size);
  2. develop N networks on all combinations of (N-1) parts;
  3. test each network on the remaining parts (test sets);
  4. average the error over these test sets.

In ”NN-practice” too expensive!

Expressive Power of MLP

Boolean functions:

- **Every boolean function** can be described by a network with a single hidden layer

  but it might require exponentially many neurons ...
  (in the number of inputs)

Continuous functions:

- **Any bounded continuous function** can be approximated with arbitrarily small error by a network with **two** hidden layers.
- **Stronger: Any bounded continuous function** can be approximated with arbitrarily small error by a network with **one** hidden layer.

Complexity of Learning

Blum, Rivest (1993):
"Training a 3-node neural network is NP-complete"

- "training" = "optimal training"
- only 3-nodes, but N k-dimensional input patterns
- "NP-complete" =
  probably "no polynomial-time algorithm exists"
- "in practice, the problem is not solvable"

Beyond the standard Backpropagation

- Momentum
- Nesterov Accelerated Gradient
- Adaptive Gradient Descent (ADAM)
- ….
- Dropout
- Batch normalization
- Heuristics for weight initialization

Will be discussed next week!

Alternative Error Functions

- Why Sum-Squared-Error?

- Modeling pdf:
  - regression: p(t, x)=> p(x), p(t | x), p(x), **<t | x>**
    (<t | x> is the expected value of t, given x)
  - classification:
    probability that input pattern x belongs to the class C_i, $P(C_i|x)$

- How should we define the error function of an MLP to get a probabilistic interpretation of outputs?

Regression

Assume that the training set (x, t) is given by:

f(x)+norm(0, σ)

(f(x) is a “deterministic” function).

Then one can show that the ML estimates of weights of a MLP that model the training set correspond to the result of training MLP with the linear output unit with Sum-Squared-Error measure.

Moreover, for each x, the output of the network approximates <t |x> (the assumption about normality of noise can be skipped).

Main Conclusion :
For regression problems use linear outputs
and the Sum-Squared-Error function

Binary Classification

If the output of the network, *output*, is interpreted as probability then for each input pattern **<x, t>** , where the class label t = 0 or 1, it should satisfy:

- **0< *output* <1**
- ***p(t|x)= output*** ^{t}***(1- output)*** $^{1-t}$

It turns out that ML estimates of weights of a MLP with a sigmoid output unit minimize the cross-entropy error function:

$$E=-\Sigma \text{target}_{k*}\log(\text{output}_k) + ((1-\text{target}_k)*\log(1-\text{output}_k))$$

Main Conclusion :
For binary classification problems use logistic output unit and minimize the cross-entropy function!

Multi-class Classification

In case of c classes and 1-of-c coding of the outputs, the error function that should be optimized is also the cross-entropy:

$$E=-\Sigma_k \Sigma_c \text{target}_{k*}\log(\text{output}_k)$$

But the output layer is activated by a <u>softmax</u> activation function:

$$y_k=\exp(a_k)/ \Sigma_{k'}(\exp(a_{k'}))$$

where a_k denotes the input to the k-th output node.

Main Conclusion :
For multi-class classification problems use softmax activation function and minimize the cross-entropy function!

Summary

The three common error functions and corresponding activation functions of the output layer are:

'linear‘ => SSE

'logistic’ => cross-entropy

'softmax‘ => cross-entropy + softmax

One can also add his/her own error function definitions
 (and the corresponding gradients).