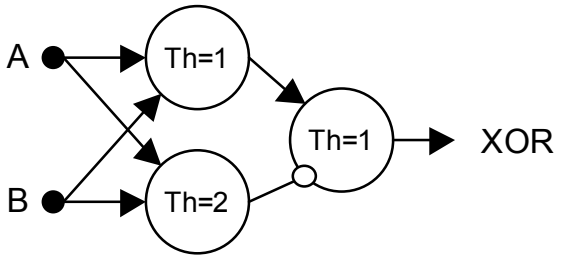
# LECTURE 18: Multi-layer perceptrons

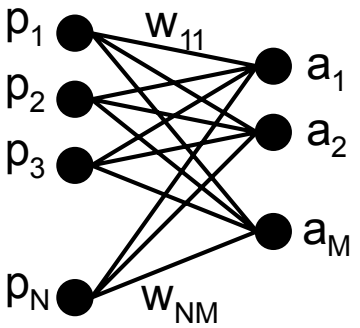
- The development of artificial neural networks
- The back propagation algorithm
- Enhancements to gradient descent
- Limiting network complexity
- Tricks of the trade

# A brief history of artificial neural networks (1)

- **McCulloch and Pitts, 1943**
  - The modern era of neural networks starts in the 1940's, when Warren McCulloch (a psychiatrist and neuroanatomist) and Walter Pitts (a mathematician) explored the computational capabilities of networks made of very simple neurons
    - A McCulloch-Pitts network fires if the sum of the excitatory inputs exceeds the threshold, as long as it does not receive an inhibitory input
    - Using a network of such neurons, they showed that it was possible to construct any logical function



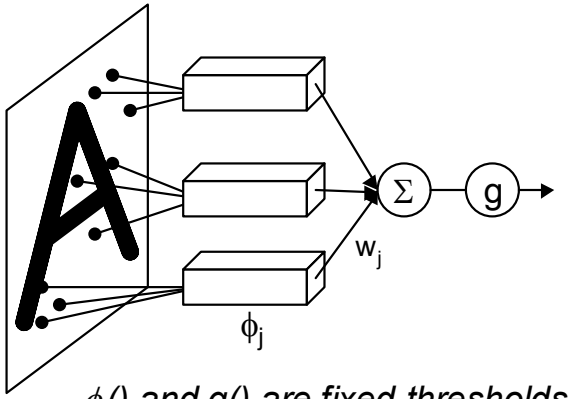

- **Hebb, 1949**
  - In his book "The organization of Behavior", Donald Hebb introduced his postulate of learning (a.k.a. Hebbian learning), which states that the effectiveness of a variable synapse between two neurons is increased by the repeated activation of one neuron by the other across that synapse
    - The Hebbian rule has a strong similarity to the biological process in which a neural pathway is strengthened each time it is used




$$w_{ij}^{new} = + w_{ij}^{old} + \alpha\, p_i a_j$$

# A brief history of artificial neural networks (2)

- **Rosenblatt, 1958**
  - Frank Rosenblatt introduced the perceptron, the simplest form of neural network
    - The perceptron is a single neuron with adjustable synaptic weights and a threshold activation function
    - Rosenblatt's original perceptron consisted of three layers (sensory, association and response)
      - Only one layer had variable weights, so his original perceptron is actually similar to a single neuron
  - Rosenblatt also developed an error-correction rule to adapt these weights (the perceptron learning rule)
  - He also proved that if the (two) classes were linearly separable, the algorithm would converge to a solution (the perceptron convergence theorem)




*$\phi_j()$ and g() are fixed thresholds*
*$w_j$ are variable weights*

- **Widrow and Hoff, 1960**
  - Bernard Widrow and Ted Hoff introduced the LMS algorithm and used it to train the Adaline (ADAptive Linear Neuron)
    - The Adaline was similar to the perceptron, except that it used a linear activation function instead of a threshold
    - The LMS algorithm is still heavily used in adaptive signal processing

# A brief history of artificial neural networks (3)

- **Minsky and Papert, 1969**
  - During the 1950s and 1960s two school of thought: Artificial Intelligence and Connectionism, competed for prestige and funding
    - The AI community was interested in the highest cognitive processes: logic, rational though and problem solving
    - The Connectionists' principal model was the perceptron
      - The perceptron was severely limited in that it could only learn linearly separable patterns
      - Connectionists struggled to find a learning algorithm for multi-layer perceptrons that would not appear until the mid 1980s
  - In their monograph "Perceptrons", Marvin Minsky and Seymour Papert mathematically proved the limitations of Rosenblatt's perceptron and conjectured that multi-layered perceptrons would suffer from the same limitations

> *The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension to multilayer systems is sterile.*
>
> M. Minsky and S. Papert, 1969

  - As a result of this book, research in ANNs was almost abandoned in the 1970s
    - Only a handful of researchers continued working on ANNs, mostly outside the US
  - The 1970s saw the emergence of SOMs [van der Malsburg, 1973], [Amari, 1977], [Grossberg, 1976] and associative memories: [Kohonen, 1972], [Anderson, 1972]

# A brief history of artificial neural networks (4)

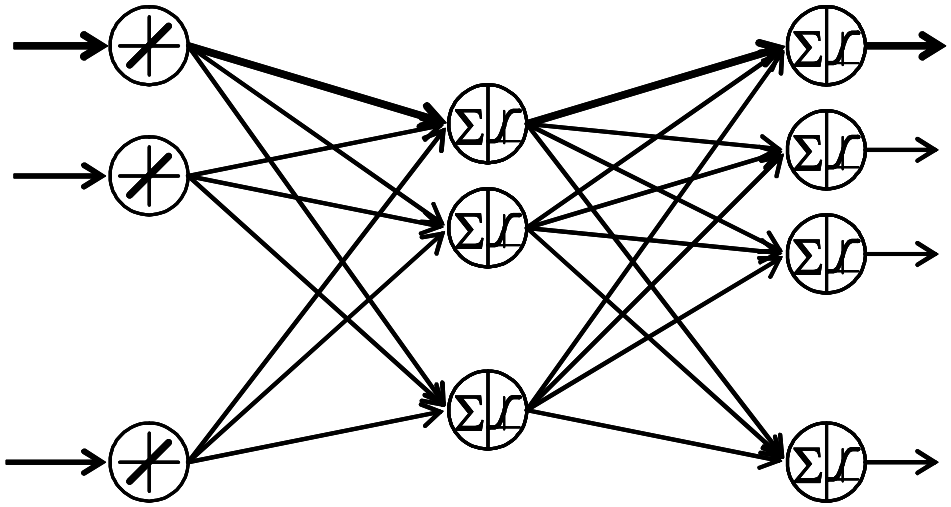
- **The resurgence of the early 1980s**
  - 1980: Steven **Grossberg**, one of the few researchers in the US that persevered despite the lack of support, establishes a new principle of self-organization called Adaptive Resonance Theory (ART) in collaboration with Gail Carpenter
  - 1982: John **Hopfield** explains the operation of a certain class of recurrent ANNs (Hopfield networks) as an associative memory using statistical mechanics. Along with backprop, Hopfield's work was most responsible for the re-birth of ANNs
  - 1982: Teuvo **Kohonen** presents SOMs using one- and two-dimensional lattices. Kohonen SOMs have received far more attention than van der Malsburg's work and have become the benchmark for innovations in self-organization
  - 1983: **Kirkpatrick, Gelatt and Vecchi** introduced Simulated Annealing for solving combinatorial optimization problems. The concept of Simulated Annealing was later used by Ackley, Hinton and Sejnowsky (1985) to develop the Boltzmann machine (the first successful realization of multi-layered ANNs)
  - 1983: **Barto, Sutton and Anderson** popularized reinforcement learning (Interestingly, it had been considered by Minsky in his 1954 PhD dissertation)
  - 1984: Valentino **Braitenberg** publishes his book "Vehicles" in which he advocates for a bottom-up approach to understand complex systems: start with very elementary mechanism and build up

# A brief history of artificial neural networks (5)

- **Rumelhart, Hinton and Williams, 1986**
  - In 1986, Rumelhart, Hinton and Williams announced the discovery of a method that allowed a network to learn to discriminate between not linearly separable classes. They called the method “backward propagation of errors”, a generalization of the LMS rule
  - Backprop provided the solution to the problem that had puzzled Connectionists for two decades
    - Backprop was in reality a multiple invention: David Parker (1982, 1985) and Yann LeCun (1986) published similar discoveries
    - However, the honor of discovering backprop goes to **Paul Werbos** who presented these techniques in his 1974 Ph.D. dissertation at Harvard University

# Multilayer Perceptrons (1)

- **MLPs are feed-forward networks of simple processing units with at least ONE "hidden" layer**
  - Each processing unit operates in a similar fashion as the perceptron, except for the threshold function is replaced by a differentiable non-linearity
    - A differentiable non-linearity is required to ensure that the gradient can be computed
  - The critical feature in MLPs is the non-linearity at the hidden layer
    - Note that if all neurons in an MLP had a linear activation function, the MLP could be replaced by a single layer of perceptrons, which can only solve linearly separable problems



# Multilayer Perceptrons (2)

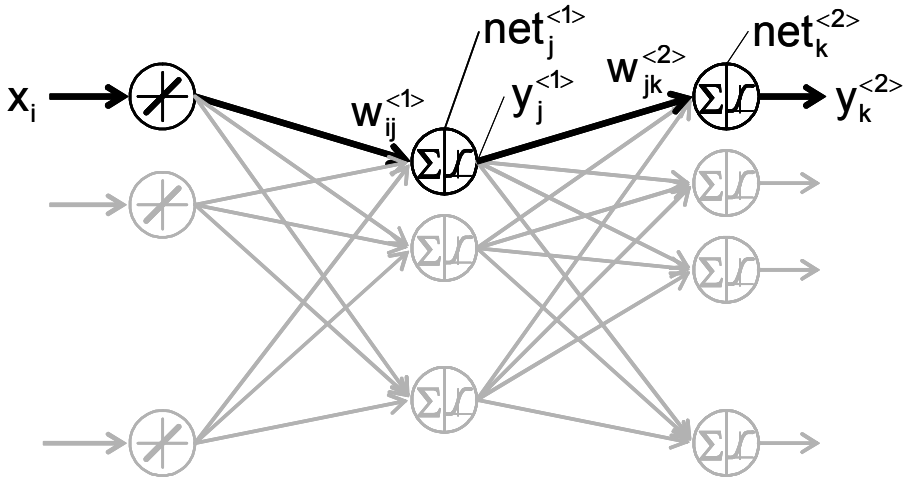
- **Notation**
  - $x_i$ is the $i^{th}$ input to the network
  - $w_{ij}^{<1>}$ is the weight connecting the $i^{th}$ input to the $j^{th}$ hidden neuron
  - $net_j^{<1>}$ is the dot product at the $j^{th}$ hidden neuron
  - $y_j^{<1>}$ is the output of the $j^{th}$ hidden neuron
  - $w_{jk}^{<2>}$ is the weight connecting the $k^{th}$ hidden neuron to the $j^{th}$ output
  - $net_k^{<2>}$ is the dot product at the $k^{th}$ output neuron
  - $y_k^{<2>}$ is the output of the $k^{th}$ output neuron
  - $t_k$ is the target (desired) output at the $k^{th}$ output neuron
  - For convenience we will treat biases as regular weights with an input of 1

$$net_j^{<1>} = \sum_{i=1}^{N_I} x_i w_{ij}^{<1>}$$

$$y_j^{<1>} = f\left(net_j^{<1>}\right) = \frac{1}{1+\exp\left(-net_j^{<1>}\right)}$$

$$net_k^{<2>} = \sum_{j=1}^{N_H} y_j^{<1>} w_{jk}^{<2>}$$

$$y_k^{<2>} = f\left(net_k^{<2>}\right) = \frac{1}{1+\exp\left(-net_k^{<2>}\right)}$$



# The back propagation algorithm (1)

- **The MLP learning problem is that of finding the weights W that capture the input/output mapping implicit in a dataset of examples**
  - We will use the sum-squared-error at the outputs as our objective function

$$J(W) = \sum_{n=1}^{N_{EX}} \sum_{k=1}^{N_O} \frac{1}{2} \left( t_k^{(n} - y_k^{<2>(n} \right)^2$$

    - where $t_k^{(n}$ is the desired target of the $k^{th}$ output neuron for the $n^{th}$ example
    - and $y_k^{<2>(n}$ is the output of the $k^{th}$ output neuron for the $n^{th}$ example
  - Back-prop learns the weights through gradient descent

$$w = w + \Delta w = w - \eta \frac{\partial J(W)}{\partial w}$$

    - The remaining part of this lecture will be concerned with finding an expression for $\partial J/\partial w$ (for each weight) in terms of what we know: the inputs $x_i$, the MLP outputs $y_k^{<2>}$ and the desired targets $t_k$
      - W (upper case) denotes the set of all weights in the MLP ($W=\{w_{ij}^{<1>}, w_{jk}^{<2>}\}$), whereas w (lower case) denotes a generic individual weight
  - For simplicity we will perform the derivation for one example ($N_{EX}=1$), allowing us to drop the outer summation

$$J(W) = \sum_{k=1}^{N_O} \frac{1}{2} \left( t_k - y_k^{<2>} \right)^2$$

# The back propagation algorithm (2)

- **Calculation of ∂J/∂w for hidden-to output weights**
  - Using the **chain rule**, the derivative of J(W) with respect to a HO weight is,

$$\frac{\partial J(W)}{\partial w_{jk}^{<2>}} = \frac{\partial J(W)}{\partial y_k^{<2>}} \frac{\partial y_k^{<2>}}{\partial net_k^{<2>}} \frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}}$$

  - We calculate each of these terms separately

$$\frac{\partial J(W)}{\partial y_k^{<2>}} = \frac{\partial}{\partial y_k^{<2>}} \left[ \sum_{n=1}^{N_O} \frac{1}{2} \left( y_n^{<2>} - t_n \right)^2 \right] = \left( y_k^{<2>} - t_k \right)$$

$$\frac{\partial y_k^{<2>}}{\partial net_k^{<2>}} = \frac{\partial}{\partial net_k^{<2>}} \left[ \frac{1}{1+\exp\left(-net_k^{<2>}\right)} \right] = \frac{\exp\left(-net_k^{<2>}\right)}{\left(1+\exp\left(-net_k^{<2>}\right)\right)^2} =$$

$$= \left[ \frac{1+\exp\left(-net_k^{<2>}\right)-1}{1+\exp\left(-net_k^{<2>}\right)} \right] \left[ \frac{1}{1+\exp\left(-net_k^{<2>}\right)} \right] = \left(1-y_k^{<2>}\right) y_k^{<2>}$$

$$\frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}} = \frac{\partial}{\partial w_{jk}^{<2>}} \left[ \sum_{n=1}^{N_H} w_{nk}^{<2>} y_n^{<1>} \right] = y_j^{<1>}$$

  - Merging all these derivatives yields

$$\boxed{\frac{\partial J(W)}{\partial w_{jk}^{<2>}} = \left( y_k^{<2>} - t_k \right) \left(1-y_k^{<2>}\right) y_k^{<2>} y_j^{<1>}}$$

    - For the bias weights, use $y_j^{<1>}=1$ in the expression above

# The back propagation algorithm (3)

- **Calculation of ∂J/∂w for input-to-hidden weights**
  - Using the chain rule, the derivative of J(W) with respect to a IH weight is

$$\frac{\partial J(W)}{\partial w_{ij}^{<1>}} = \frac{\partial J(W)}{\partial y_j^{<1>}} \frac{\partial y_j^{<1>}}{\partial net_j^{<1>}} \frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}}$$

  - The second and third terms are easy to calculate from our previous result

$$\frac{\partial y_j^{<1>}}{\partial net_j^{<1>}} = \left(1 - y_j^{<1>}\right) y_j^{<1>}$$

$$\frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}} = x_i$$

  - The first term, however, is not straightforward since we do not know what the outputs of the hidden neurons ought to be
  - This is known as the **credit assignment problem** [Minski,1961], which puzzled connectionists for two decades

# The back propagation algorithm (4)

- **The trick to solve this puzzle is to realize that hidden neurons do not make errors, they only contribute to the errors of the output nodes**
  - The derivative of the error with respect to a hidden node's output is therefore the sum of that hidden node's contribution to the errors of all the output neurons

$$\frac{\partial J(W)}{\partial y_j^{<1>}} = \sum_{n=1}^{N_O} \frac{\partial J(W)}{\partial y_n^{<2>}} \frac{\partial y_n^{<2>}}{\partial net_n^{<2>}} \frac{\partial net_n^{<2>}}{\partial y_j^{<1>}}$$

  - The first two terms in the summation are known from our earlier derivation

$$\frac{\partial J(W)}{\partial y_n^{<2>}} \frac{\partial y_n^{<2>}}{\partial net_n^{<2>}} = \left(y_n^{<2>} - t_n\right)\left(1 - y_n^{<2>}\right)y_n^{<2>} = p_n$$

  - The last term in the summation is

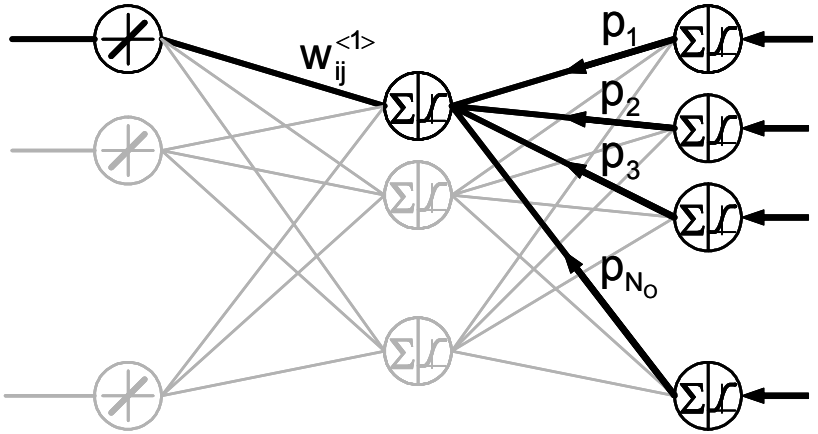
$$\frac{\partial net_n^{<2>}}{\partial y_j^{<1>}} = w_{jn}^{<2>}$$

# The back propagation algorithm (5)

- Merging these derivatives yields

$$\frac{\partial J(W)}{\partial y_j^{<1>}} = \sum_{n=1}^{N_O} \frac{\partial J(W)}{\partial y_n^{<2>}} \frac{\partial y_n^{<2>}}{\partial net_n^{<2>}} \frac{\partial net_n^{<2>}}{\partial y_j^{<1>}} = \sum_{n=1}^{N_O} \underbrace{\left(y_n^{<2>} - t_n\right)\left(1 - y_n^{<2>}\right)y_n^{<2>}}_{p_n} w_{jn}^{<2>}$$

  - Notice that what have actually doing is propagate the error term $p_n$ backwards trough the hidden-to-output weighs (hence the term backprop)



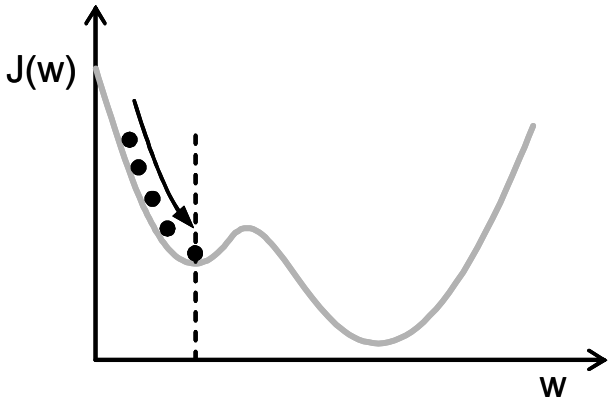

- And the final expression of $\partial J/\partial w$ for input-to-hidden weights is

$$\frac{\partial J(W)}{\partial w_{ij}^{<1>}} = \left[\sum_{n=1}^{N_O}\left(y_n^{<2>} - t_n\right)\left(1 - y_n^{<2>}\right)y_n^{<2>} w_{jn}^{<2>}\right]\left(1 - y_j^{<1>}\right)y_j^{<1>} x_i$$

  - For the bias weights, use $x_i$=1 in the expression above

# Enhancements to gradient descent: momentum (1)

- **One of the main limitations of back-prop (gradient descent) is local minima**
  - When the gradient descent algorithm reaches a local minimum, the gradient becomes zero and the weights converge to a sub-optimal solution



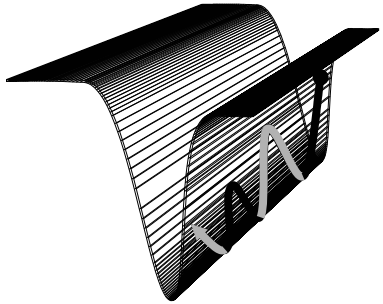

- **A very popular method to avoid local minima is to compute a temporal average direction in which the weights have been moving recently**
  - An easy way to implement this is by using an exponential average

$$\Delta w(n) = \mu[\Delta w(n-1)] + (1-\mu)\left[\eta \frac{\partial J(W)}{\partial w}\right]$$

  - The term $\mu$ is called the **momentum**
    - The momentum has a value between 0 and 1 (typically 0.9)
    - The closer to 1, the stronger the influence of the instantaneous steepest descent direction

# Enhancements to gradient descent: momentum (2)

- **The momentum term is also useful in spaces with long ravines characterized by sharp curvature across the ravine and a gently sloping floor**
  - Sharp curvature tends to cause divergent oscillations across the ravine
    - To avoid this problem, we could decrease the learning rate, but this is too slow



  - The momentum term filters out the high curvature and allows the effective weight steps to be bigger
  - It turns out that ravines are not uncommon in optimization problems, so the use of momentum can be helpful in many situations
- **However, a momentum term can hurt when the search is close to the minima (think of the error surface as a bowl)**
  - As the network approaches the bottom of the error surface, it builds enough momentum to propel the weights in the opposite direction, creating an undesirable oscillation that results in slower convergence

# More enhancements: adaptive learning rates (1)

- **Rather than having a unique learning rate for all weights in the MLP**
  - Let each weight to have its own learning rate
  - Let the learning rate adapt based on the weight's performance during training
    - If the direction in which the objective function decreases w.r.t. a weight is the same as the direction in which it has been decreasing recently, make the learning rate larger. Otherwise decrease the learning rate.
- **SOLUTION**
  - The direction $\delta(n)$ in which the error decreases at time $n$ is given by the sign of $\partial J/\partial w$
  - The direction $\bar{\delta}(n)$ in which the error has been decreasing "recently" is computed as the exponential average of $\delta(n)$

$$\bar{\delta}(n+1) = \theta\bar{\delta}(n) + [1-\theta]\delta(n)$$

  - To determine if the current direction and the recent direction coincide we compute the product between $\delta(n)$ and $\bar{\delta}(n)$
    - If the product is greater than 0, the signs are the same
    - If the product is less than 0, the signs are different

# More enhancements: adaptive learning rates (2)

- **Based on the dot product, the learning rates are adapted according to the following rule**

$$\eta(n) = \begin{cases} \eta(n-1) + \kappa & \text{if } \bar{\delta}(n) \cdot \delta(n) > 0 \\ \eta(n-1)\psi & \text{if } \bar{\delta}(n) \cdot \delta(n) \leq 0 \end{cases}$$

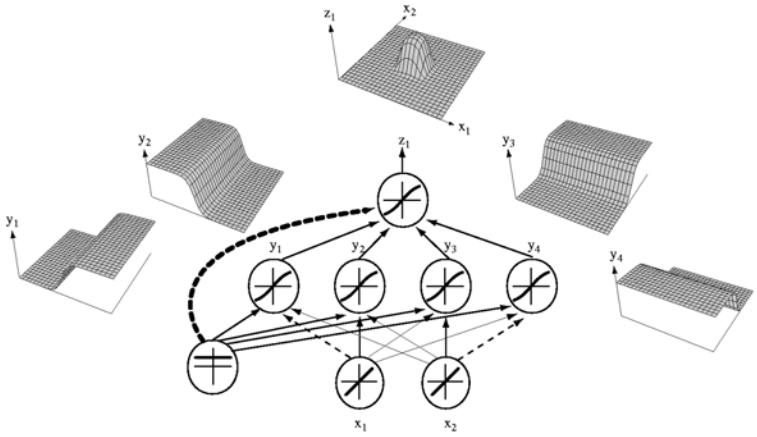
  - $\kappa$ is a constant that is added to the learning rate if the direction has not changed (the actual value of $\kappa$ is problem dependent)
  - $\Psi$ is a fraction that is multiplied to the learning rate if the direction has changed (typically 0.5)
- **Assuming also a momentum term, the final expression for the weight change becomes**

$$\Delta w(n) = \mu[\Delta w(n-1)] + (1-\mu)\left[\eta(n)\frac{\partial J(W)}{\partial w}\right]$$

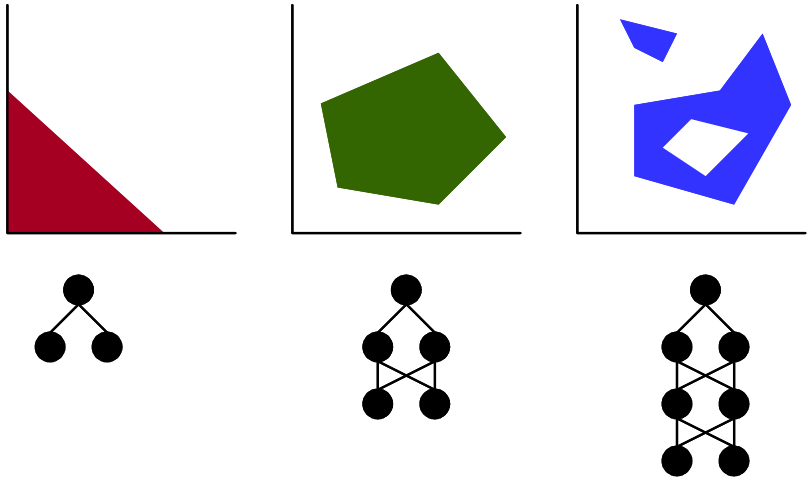
  - The adaptive nature of $\eta$ is made explicit by making it a function of time $\eta(n)$

# Limiting network complexity (1)

- **Number of hidden layers**
  - Given a sufficiently large number of hidden neurons, a two-layer MLP can approximate any continuous function arbitrarily well
    - The example below [Duda, Hart and Stork, 2001], shows that a combination of four hidden neurons can produce a “bump” at the output space of a two-layered MLP
    - A large number of these “bumps” can approximate any surface arbitrarily well
  - Nonetheless, the addition of extra hidden layers may allow the MLP to perform a more efficient approximation with fewer weights [Bishop, 1995]




*From [Duda, Hart and Stork, 2001],*



*From [Bishop, 1995]*

# Limiting network complexity (2)

- **Number of hidden neurons**
  - While the number of inputs and outputs are dictated by the problem, the number of hidden units is not related so explicitly to the application domain
    - The number of hidden units determines the degrees of freedom or expressive power of the model
      - A small number of hidden units may not be sufficient to model complex I/O mappings
      - A large number of hidden units may overfit the training data and prevent the network from generalizing to new examples
  - Despite a number of "rules of thumb" published in the literature, a-priori determination of an appropriate number of hidden units is an unsolved problem
    - The "optimal" number of hidden neurons depends on multiple factors, including number of examples, level of noise in the training set, complexity of the classification problem, number of inputs and outputs, activation functions and training algorithm.
    - In practice, several MLPs are trained and evaluated in order to determine an appropriate number of hidden units
  - A number of adaptive approaches have also been proposed
    - **Constructive** approaches start with a small network and incrementally add hidden neurons (e.g., cascade correlation),
    - **Pruning** approaches, which start with a relatively large network and incrementally remove weights (e.g., optimal brain damage)

# Limiting network complexity (3)

- **Weight decay**
  - To prevent the weights from growing too large (a sign of over-training) it is convenient to add a decay term of the form

$$w(n+1) = (1-\varepsilon)w(n)$$

    - Weights that are not needed eventually decay to zero, whereas necessary weights are continuously updated by back-prop
  - Weight decay is a simple form of regularization, which encourages smoother network mappings [Bishop, 1995]
- **Early stopping**
  - Early stopping can be used to prevent the MLP from over-fitting the training set
  - The stopping point may be determined by monitoring the sum-squared-error of the MLP on a validation set during training
- **Training with noise (jitter)**
  - Training with noise prevents the MLP from approximating the training set too closely, which leads to improved generalization

# Tricks of the trade (1)

- **Activation function**
  - An MLP trained with backprop will generally train faster if the activation function is anti-symmetric: *f(-x)=-f(x)* (e.g., the hyperbolic tangent)
- **Target values**
  - Obviously, it is important that the target values are within the dynamic range of the activation function, otherwise the neuron will be unable to produce them
  - In addition, it is recommended that the target values are not the asymptotic values of the activation function; otherwise backprop will tend to drive the neurons into saturation, slowing down the learning process
    - The slope of the activation function, which is proportional to $\Delta w$, becomes zero at $\pm\infty$
- **Input normalization**
  - Input variables should be preprocessed so that their mean value is zero or small compared to the variance
  - Input variables should be uncorrelated (use PCA to accomplish this)
  - Input variables should have the same variance (use Fukunaga's whitening transform)

# Tricks of the trade (2)

- **Initial weights**
  - Initial random weights should be small to avoid driving the neurons into saturation
    - Hidden-to-output (HO) weights should be made larger than input-to-hidden (IH) weights since they carry the back propagated error. If the initial HO weights are very small, the weight changes at the IH layer will initially be very small, slowing the training process
- **Weight updates**
  - The derivation of back-prop was based on one training example but, in practice, the data set contains a large number of examples
  - There are two basic approaches for updating the weights during training
    - On-line training: weights are updated after presentation of each example
    - Batch training: weights are updated after presentation of all the examples (we store the $\Delta w$ for each example, and add them up to the weight after all the examples have been presented)
  - Batch training is recommended
    - Batch training uses the TRUE steepest descent direction
    - On-line training achieves lower errors earlier in training but only because the weights are updated n (# examples) times faster than in batch mode
    - On-line training is sensitive to the ordering of examples