Backpropagation Learning Algorithm

EEL 4930/EEL 5840 ELEMENTS OF MACHINE INTELLIGENCE

FALL 2016

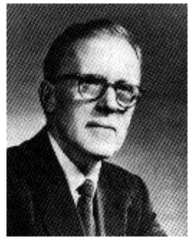
History

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.

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.



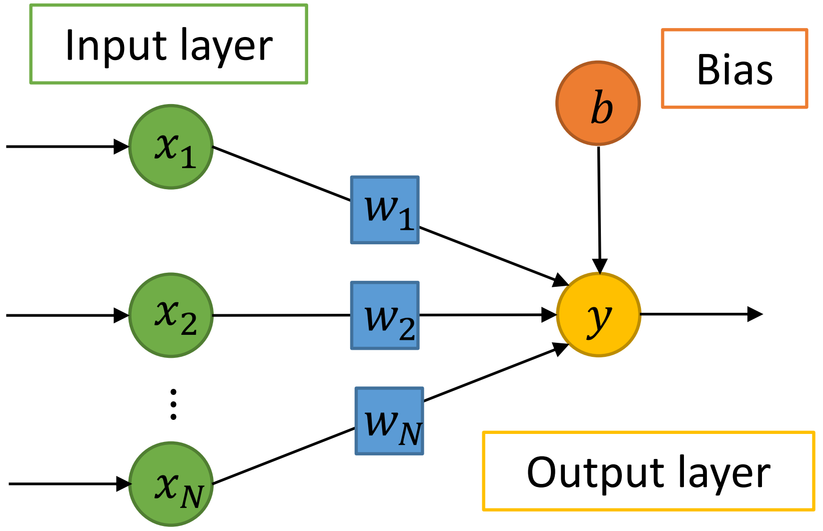
Rosenblatt, 1957

Frank Rosenblatt introduced the perceptron, the simplest form of a neural network.

Rosenblatt also developed an error-correction rule to adapt these weights (a.k.a. the perceptron learning rule), and proved that if the (two) classes were linearly separable, the algorithm would converge to a solution (a.k.a. the perceptron convergence theorem).

Single-layer Perceptron

A perceptron is a single layer **feed-forward network** based on a threshold transfer funciton. The perceptron is the simplest type of artificial neural networks and can only classify **linearly separable** cases with a binary target (0,1).



$$y = f(x;w) = \begin{cases} 1, & if \ \sum_{i=1}^{N} w_i x_i + b > \theta \\ 0, & otherwise \end{cases}$$

Perceptron Algorithm

The perceptron does not have *a priori* knowledge, so:

1. The initial weights, w, are assigned randomly.
2. The perceptron sums all the weighted inputs plus the bias, b, and if the sum is above the threshold (some determined value d), perceptron is said to be activated ($y = 1$).
3. The input values are presented to the perceptron, and if the predicted output is the same as the desired output, then the performance is considered satisfactory and no changes to the weights are made. However, if the output does not match the desired output, then the weights need to be changed to reduce the error.

Perceptron Weight Adjustments

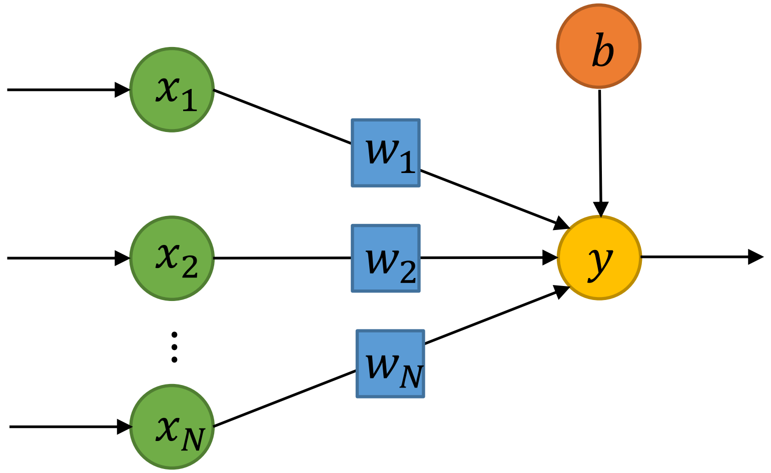
$$\Delta w = \eta \times d \times x$$

where

d is the predicted output, desired output

η is the learning rate, usually less than 1

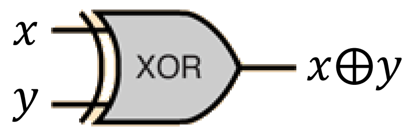
x is the input data



$$w_{new} = w + \Delta w$$

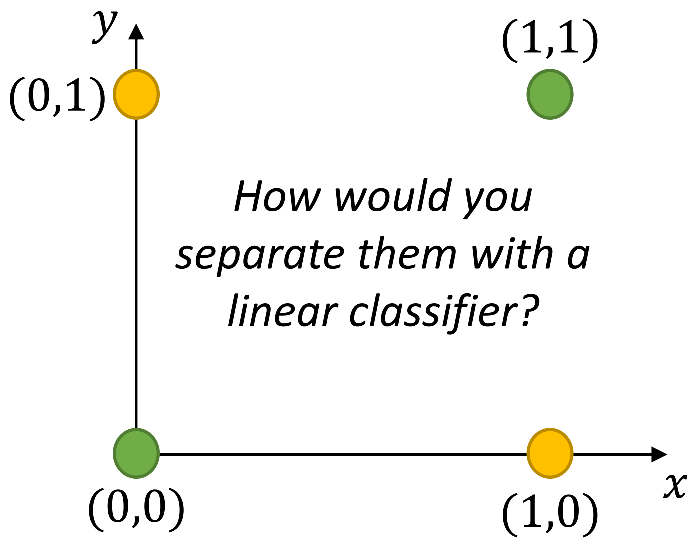
XOR Problem

Because the perceptron is a linear classifier and if the cases are not linearly separable, the learning process will never reach a point where all the cases are classified properly. The most famous example of the inability of perceptron to solve problems with linearly non-separable cases is the exclusive OR (XOR) problem.



$$x \oplus y = \bar{x}y + x\bar{y}$$

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0



History

Widrow and Hoff, 1960

At about the same time, Bernard Widrow and Ted Hoff introduced the Least Mean Square algorithm (a.k.a. LMS or Widrow-Hoff rule) and used it to train the Adaline (ADAptive Linear Neuron).

Minsky and Papert, 1969

In their monograph "Perceptrons", Marvin Minsky and Seymour Papert mathematically proved the limitations of Rosenblatt's perceptron and conjectured (incorrectly!) that multi-layered perceptrons would suffer from the same limitations.

As a result of this monograph, research in neural networks was almost abandoned in the 1970s.

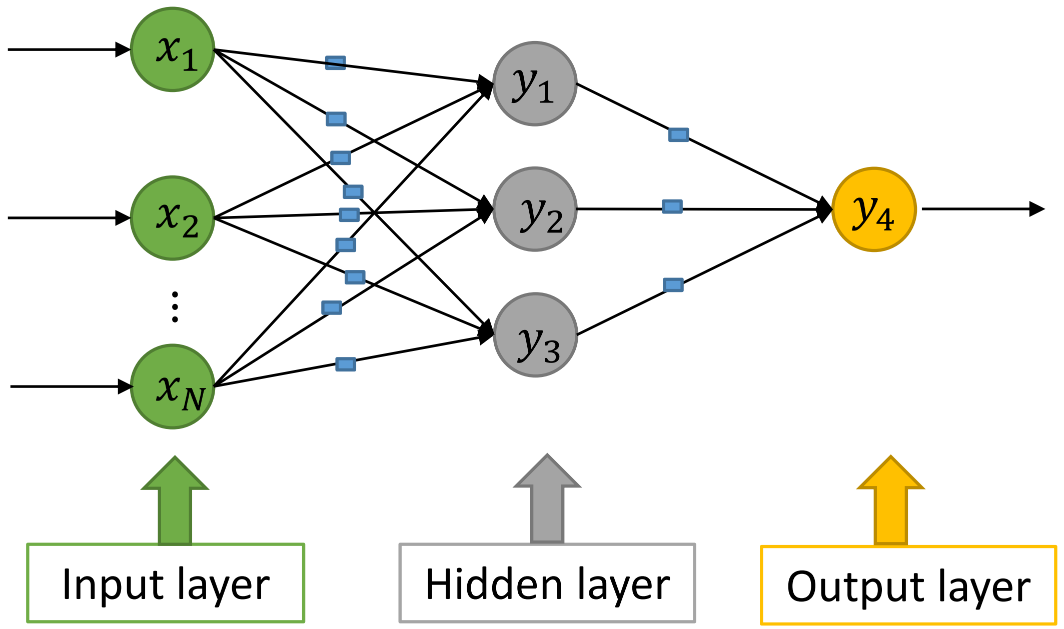
Rumelhart, Hinton and Williams, 1986

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.

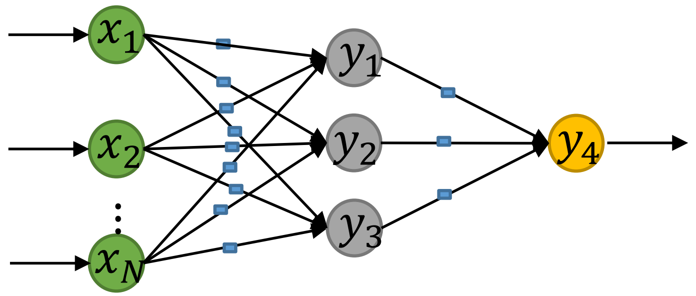
However, the honor of discovering backpropagation goes to **Paul Werbos** who presented these techniques in his **1974** Ph.D. dissertation at Harvard University.

Multi-layer Perceptron

A multi-layer perceptron (MLP) has the same structure of a single layer perceptron with one or more hidden layers. An example of an MLP with one hidden layer is as follows:



Properties of Architecture



- No connections withtin a layer.
- No direct connections between input and output layers.
- Fully connected between layers.
- The number of layers can vary.
- Number of output units need not to be equal to the number of input units.
- Number of hidden units per layer can be more or less than input or output units.
- The units do NOT know their “position” in space, i.e., they all DO THE SAME THING.

Each unit is a perceptron, often including a bias as an extra weight.

$$y_j = f\left(\sum_i w_{ji}x_i + b_j\right)$$

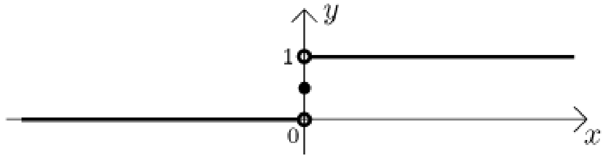
Where $f(x)$ is the activation function.

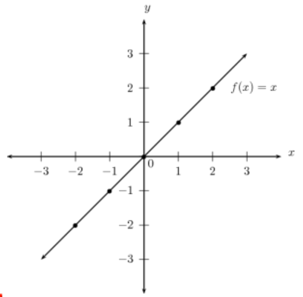
Activation Function

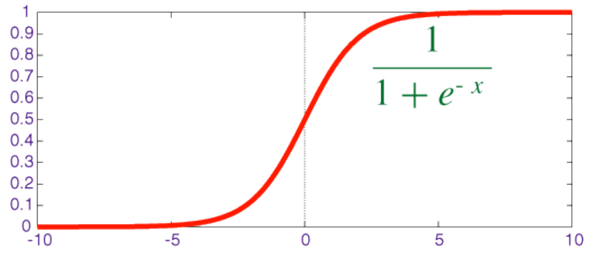
The weighted sum can be taken as a linear activation function or non-linear activation function. The *sigmoid function* is often utilized, that is, $f(x) = \frac{1}{1+e^{-x}}$. However, many other activation functions can be used. For example:

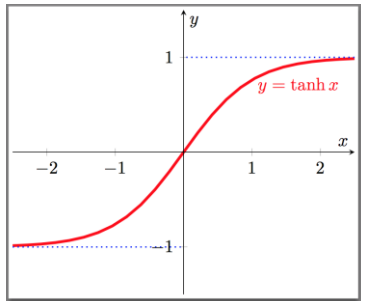
- Heaviside function: $\mathrm{f}(x) = \begin{cases} 0, & if\ x < 0 \\ 1, & if\ x \geq 0 \end{cases}$
- Linear function: $f(x) = x$
- Standard Sigmoidal function: $f(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic Tangent: $f(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$

And many others...



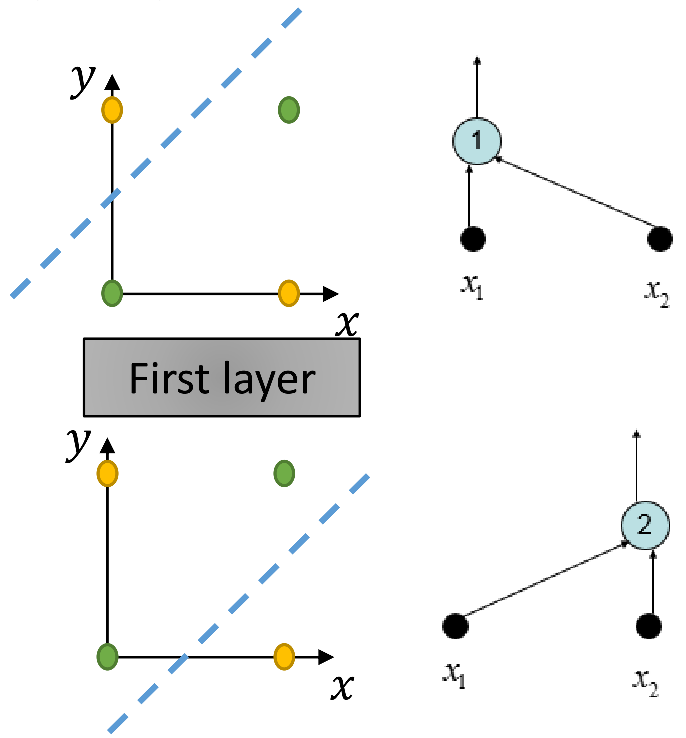


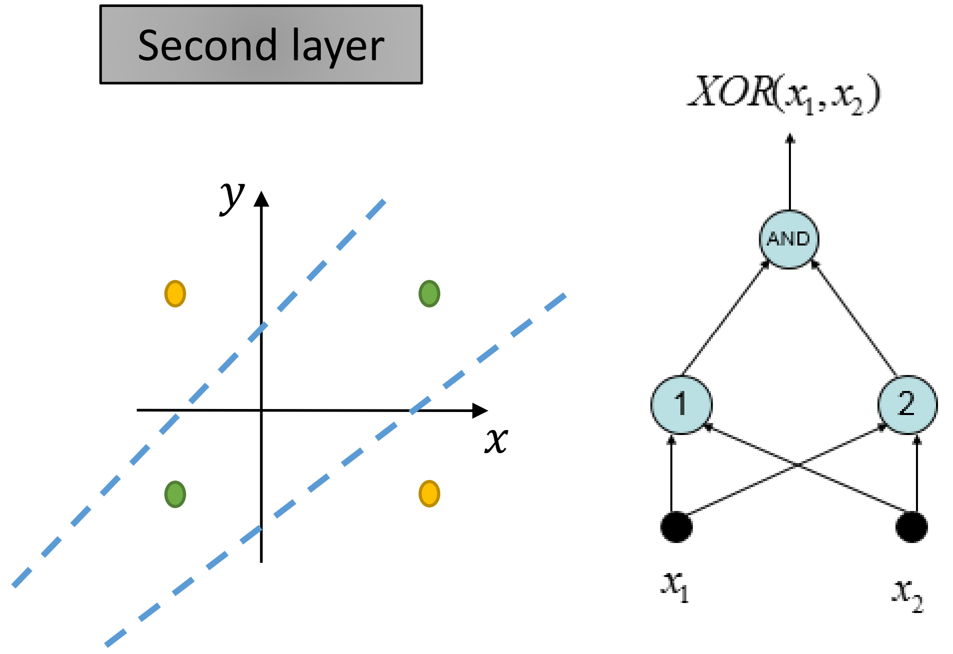




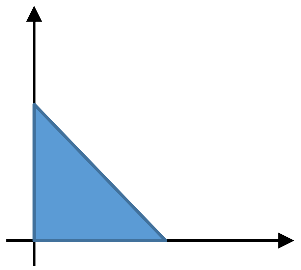
Solving XOR with an MLP

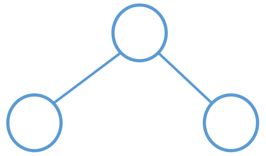
Minsky and Papert (1969) offered a solution to XOR problem by combining perceptron unit responses using a second layer of units. Piecewise linear classification using an MLP with threshold (perceptron) units and a linear activation function.



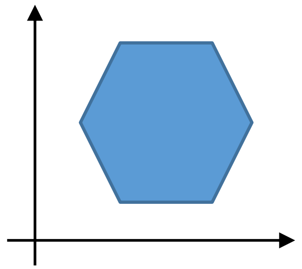


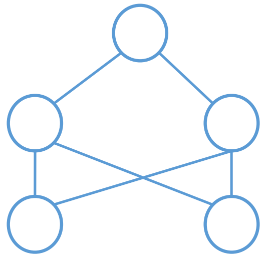
What do each of the layers do?



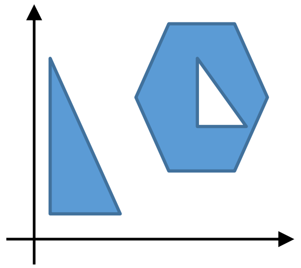


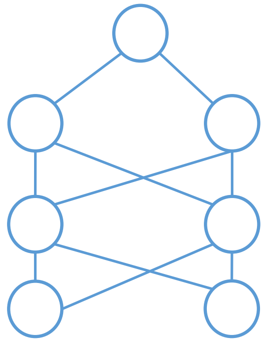
1st layer draws linear boundaries





2nd layer combines the boundaries





3rd layer can generate arbitrarily complex boundaries

Feedback ← Backpropagation

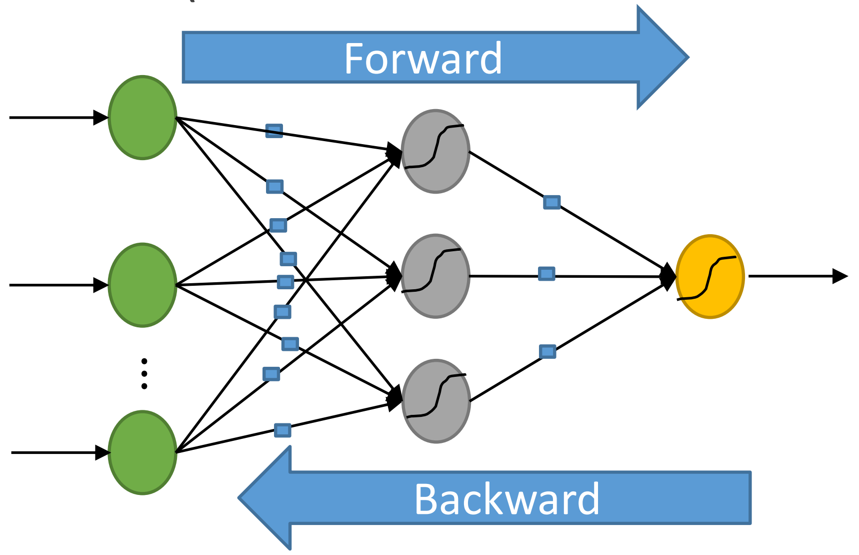
The learning procedure involves the presentation of a set of pairs of input and output patterns, $X = \{x_i\}_{i=1}^{N}$ and $\mathrm{Y} = \{y_j\}_{j=1}^{M}$. The system uses the input vector to produce its own output vector and then compares this with the *desire output*, or *target output*. If there is no difference, no **learning** takes place. Otherwise, the weights are changed to reduce the difference. This procedure is basically the perceptron learning algorithm.

This procedure can be **automated** by the machine itself, without any outside help, if we provide some **feedback** to the machine on how it is doing. The feedback comes in the form of the definition of an **error criterion** or cost function that must be **minimized**. For each training pattern we can define an error (ε_k) between the desired response (d_k) and the actual output (y_k). Note that when the error is zero, the machine output is equal to the desired response. This learning mechanism is called **backpropagation**.

Backpropagation Learning

The backpropagation (BP) algorithm consists of two phases:

- **Forward phase:** computes the “functional signal”, feed forward propagation of input pattern signals through the network.
- **Backward phase:** computes the “error signal”, *propagates* the error *backwards* through the network starting at the output units (where the error is the difference between desired and predicted output values).



Network Mapper

Neural networks (NN) map the input space, X, onto the output space, Y:

$$NN: X \to Y$$

A feed-forward MLP with one hidden layer and with sigmoidal activation function can approximate arbitrary closely any continous function.

The question is: how to set the parameters of the network?

Error Criterion

There are many possible definitions of the error, but commonly in neuro-computing one uses the error variance (or power), i.e. the sum of the squared difference between the desired response and the actual output. We already found this criterion in LMS/NLMS adaptation and called it the mean square error (MSE):

$$MSE = J(w) = \frac{1}{2N}\sum_k \varepsilon_k^2 = \frac{1}{2N}\sum_k (d_k - y_k)^2$$

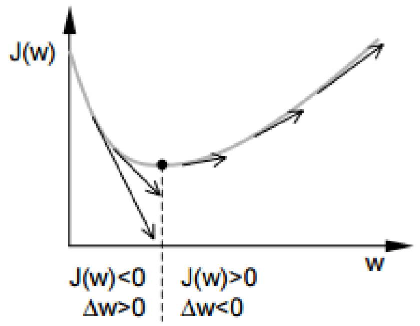
(The constant ½ appears here for simplification that we will see later.)

The goal of the classifier is to minimize this cost function by changing its free parameters. This search for the weights to meet a desired response or internal constraint is the essence of any connectionist computation. We already found this methodology in linear regression.

The Delta Rule

The minimum of the MSE cost function is found by following the opposite direction of the performance surface gradient estimated at each point. A systematic step by step procedure based on gradient descent (the LMS rule) is able to modify the weights such that the minimum of the performance surface was reached:

$$w(n+1) = w(n) + \eta\, \varepsilon(n)\, x(n)$$



Calculus reminder: Chain Rule

Basically the chain rule tells how to compute the partial derivative of a variable with respect to another when a functional form links the two. Let us assume that $y = f(x)$, and the goal is to compute ${}^{\delta y}/_{\delta x}$, the **sensitivity** of y with respect to x. As long as f is differentiable, then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial x}$$

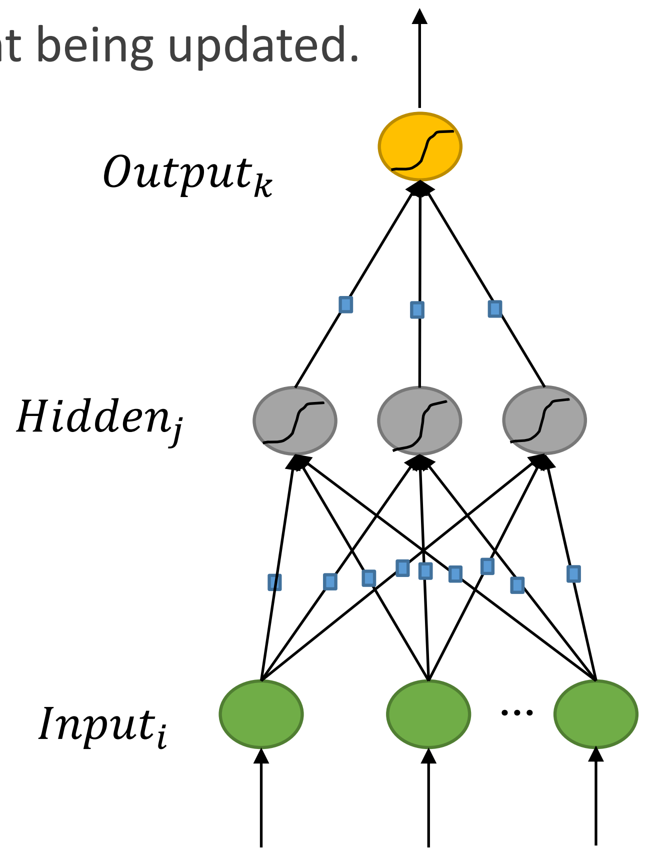
Notation

Notice that all the necessary components are locally related to the weight being updated.

This is one feature of backpropagation that seems biologically plausible.

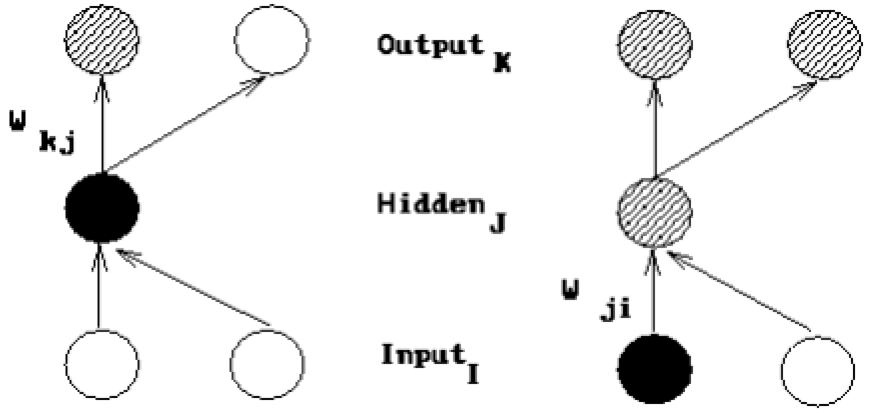
For the purpose of this derivation, we will use the following notation:

- The subscript k denotes the output layer.
- The subscript j denotes the hidden layer.
- The subscript i denotes the input layer.



Notation

- w_{kj} denotes a weight from the hidden to the output layer.
- w_{ji} denotes a weight from the input to the hidden layer
- y_k denotes the output value.
- net_j denotes the net input at layer j.
- d_k denotes the target value.



The total error in a network is given by the MSE:

$$MSE = J(w) = E = \frac{1}{2N}\sum_k \varepsilon_k^2 = \frac{1}{2N}\sum_k (d_k - y_k)^2$$

$$= \frac{1}{2N}\sum_k \left(d_k - f\left(\sum_j w_{kj}x_j + b_k\right)\right)^2 = \frac{1}{2N}\sum_k \left(d_k - f\left(\sum_j w_{kj}f(net_j) + b_k\right)\right)^2 = \cdots$$

Delta Rule – Local Gradient

We want to adjust the network's weights to reduce this overall error, that is

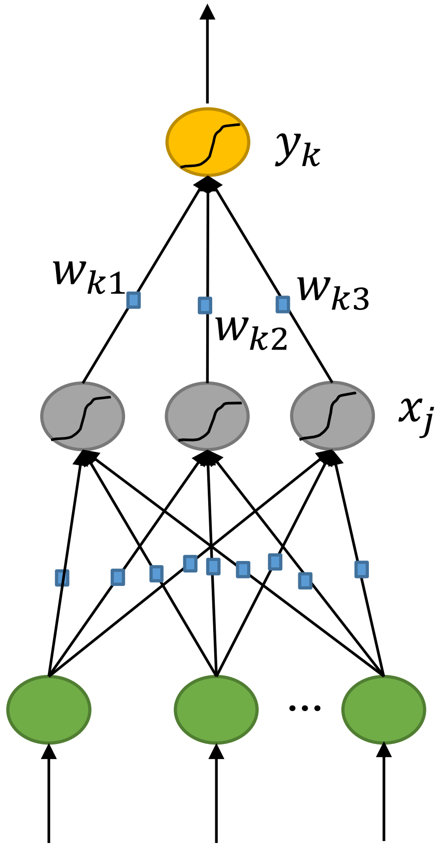
$$\Delta w \propto -\frac{\partial J(w)}{\partial w}$$

We will begin at the output layer with a particular weight.

$$\Delta w_{kj} \propto -\frac{\partial J(w)}{\partial w_{kj}}$$

However error is not directly a function of the weight. We expand this as follows:

$$\boxed{\Delta w_{kj} = -\eta \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}}}$$



Derivatives

- **Derivative of the error with respect to the output**:

$$\boxed{\frac{\partial E}{\partial y_k} = \frac{\partial\left(\frac{1}{2}(d_k - y_k)^2\right)}{\partial y_k} = -(d_k - y_k)}$$

(Now we see why the $^1/_2$ in the error term was useful.)

- **Derivative of the output with respect to the activation function:**

$$y_k = f\left(\sum_k w_{kj}x_j + b_k\right) = f(net_k) \qquad f(x) = \tanh(x) = \frac{1}{1+e^{-x}} = (1+e^{-x})^{-1}$$

$$\boxed{\frac{\partial y_k}{\partial net_k} = f'(net_k) = \frac{\partial\left((1+e^{-net_k})^{-1}\right)}{\partial net_k} = \frac{e^{-net_k}}{(1+e^{-net_k})^2}}$$

Derivatives

We’d like to be able to rewrite this result in terms of the activation function. Notice that:

$$f'_{sigmoid}(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\right)^2$$

Using this fact, we can rewrite the result of the partial derivative as:

$$f'_{sigmoid}(x) = f_{sigmoid}(x)(1 - f_{sigmoid}(x))$$

Also, for the hyperbolic tangent activation function:

$$f'_{tanh}(x) = 0.5\ (1 - f^2_{tanh}(x))$$

Derivatives

- **Derivative of the net input with respect to a weight:**

Note that only one term of the net summation will have a non-zero derivative: again the one associated with the particular weight we are considering.

$$\boxed{\frac{\partial net_k}{\partial w_{kj}} = \frac{\partial(w_{kj}x_j + b_k)}{w_{kj}} = \frac{\partial(w_{kj}net_j + b_k)}{w_{kj}} = net_j}$$

where, $net_j = \sum_i w_{ji}x_i + b_j$

Weight change rule for a hidden to output weight

$$\boxed{\Delta w_{kj} = -\eta \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}}}$$

$$\boxed{\frac{\partial E}{\partial y_k} = -(d_k - y_k)} \qquad \boxed{\frac{\partial y_k}{\partial net_k} = f(net_k)(1 - f(net_k))} \qquad \boxed{\frac{\partial net_k}{\partial w_{kj}} = net_j}$$

Now substituting these results back into our original equation we have:

$$\boxed{\Delta w_{kj} = \eta \overbrace{(d_k - y_k) f(net_k)(1 - f(net_k))}^{\delta_k} net_j}$$

$$\boxed{\Delta w_{kj} = \eta \delta_k net_j}$$

Weight change rule for an input to hidden weight

Now we have to determine the appropriate weight change for an input to hidden weight. This is more complicated because it depends on the error at all of the nodes this weighted connection can lead to.

$$\Delta w_{ji} \propto \left[\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial net_k}\frac{\partial net_k}{\partial y_j}\right]\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

$$= \eta\left[\sum_k \underbrace{(d_k - y_k)\, f(net_k)(1 - f(net_k))}_{\delta_k} w_{kj}\right] f(net_j)(1 - f(net_j))net_i = \eta \underbrace{\left[\sum_k \delta_k\, w_{kj}\right] f(net_j)(1 - f(net_j))}_{\delta_j} net_i$$

$$\boxed{\Delta w_{ji} = \eta\delta_j net_i}$$

(Note that $net_i = x_i$ are the input values.)

Recap

- Single-layer perceptrons can only classify linearly separable examples.
- Units in a neural network are not aware of they position in the network.
- MLPs can solve the XOR problem.
- MLPs can have one or more hidden layers.
- Feedback of the *local* errors is the fundamental concept of backpropagation.
- Bacpropagation has two stages: forward and backward stages.
- A feed-forward MLP with one hidden layer and with sigmoidal activation function can approximate arbitrary closely any continous functions.
- The weight updates are done from the output layer to the input layer.

Backpropagation Step 1/3

Step 1 (Forward propagation of activity): Initialize weights at random and choose a learning rate η. Do forward pass through network (with fixed weights) to produce output(s), i.e., in forward direction, layer by layer.

$$net_j = f\left(\sum_i w_{ji} x_i + b_j\right) = f\left(\sum_i w_{ji} f(net_i) + b_j\right)$$

Hidden Layer

$$y_k = f\left(\sum_j w_{kj}\, x_j + b_k\right) = f\left(\sum_j w_{kj}\, f(net_j) + b_k\right) = f\left(\sum_j w_{kj}\, f\left(\sum_i w_{ji} x_i + b_j\right) + b_k\right)$$

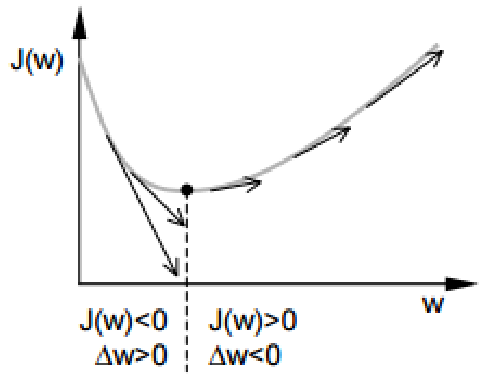
Output Layer

Backpropagation Step 2/3

Step 2 (Backward propagation of errors): Having calculated the output, one can now evaluate the error from the desire response and backpropagate the local gradient-based error at each layer.

$$\boxed{\Delta w_{kj} = \eta \delta_k net_j} \quad where\ \delta_k = (d_k - y_k) f(net_k)(1 - f(net_k))$$

$$\boxed{\Delta w_{ji} = \eta \delta_j net_i} \quad where\ \delta_j = \left[\sum_k \delta_k w_{kj}\right] f(net_j)(1 - f(net_j))$$



Backpropagation Step 3/3

Step 3 (Update weights and observe convergence): Update the weights and repeat step 2 and 3 until the MSE is minimized or (if) convergence is observed.

$$\boxed{w_{kj} = w_{kj} + \Delta w_{kj} = w_{kj} + \eta \delta_k net_j}$$

$$\boxed{w_{ji} = w_{ji} + \Delta w_{ji} = w_{ji} + \eta \delta_j net_i}$$

Non-Convexity of Performance Surface

- Several key aspects changed in the performance surface with the introduction of the nonlinearity. The performance depends on the topology of the network through the output error. So, when **nonlinear processing elements** are utilized to solve a given problem the relationship “performance - weights” becomes nonlinear and there is no guarantee of a single minimum. The performance surface may have several minima. The minimum that produces the smallest error in the search space is called the global minimum. The others are called local minima.
- Alternatively, we say that the performance surface is **non-convex**. This impacts the search scheme, because gradient descent uses local information to search the performance surface. In the immediate neighborhood, local minima are indistinguishable from the global minimum. So the gradient search algorithm may be caught in these sub-optimal performance points “thinking” it has reached the global minimum
- Backpropagation learning does not require normalization of input vectors; however, normalization could improve performance.

Normalization of Feature Space

Normalization of the feature space may improve performance.

As you recall, in PCA the feature that has the most variance is the dominant one. So **unit-variance normalization** can be applied to the feature space to make sure that the most variant feature does not dominate the network.

In some cases, subtracting the mean value eliminates any **bias** from the data.

Min-max normalization is another type of normalization.

But all of them have the same objective: provide relevant and discriminant information to the network.

Training of MLP with BP

- Training requires many iterations with many training examples or *epochs* (one epoch is entire presentation of complete training set).
- It can be slow!
- Note that computation in MLP is local (with respect to each neuron).
- Parallel computation implementation is also possible.
- There are other variants of BP, e.g. SuperSAB, QuickProp, Levenberg-Marquart algorithm.

Training and Test Sets

- How many examples?
  - The more the merrier!
- Disjoint training and testing data sets:
  - Learn from training data but evaluate performance (**generalization** ability) on unseen test data.
  - This also minimizes the effect of **overfitting**.
- **Aim**: minimize error on *test* data.
- **Optimization** will be addressed in the next lecture!

ANY QUESTIONS?