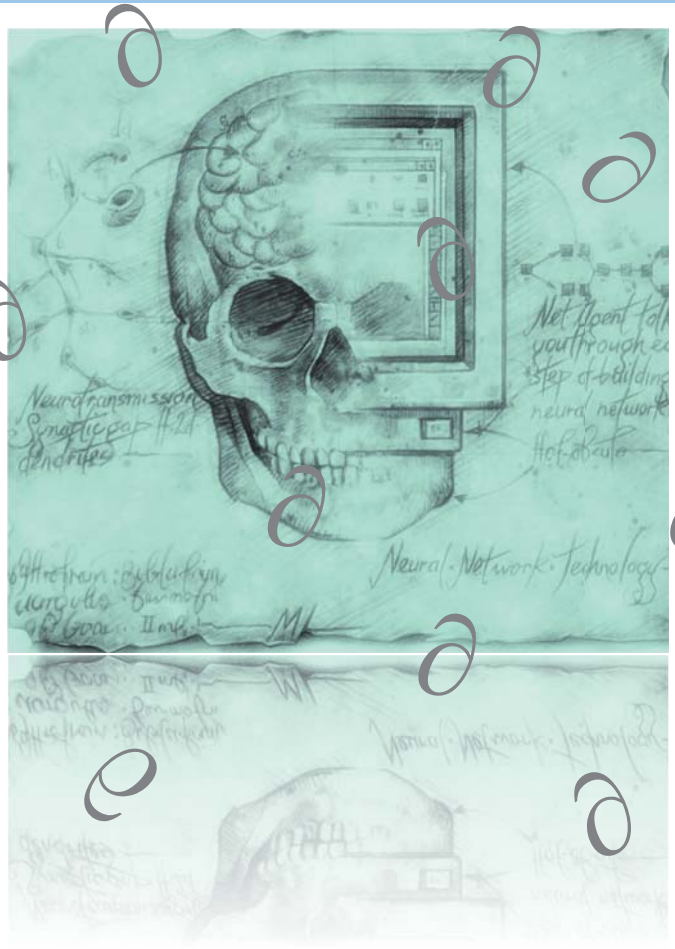
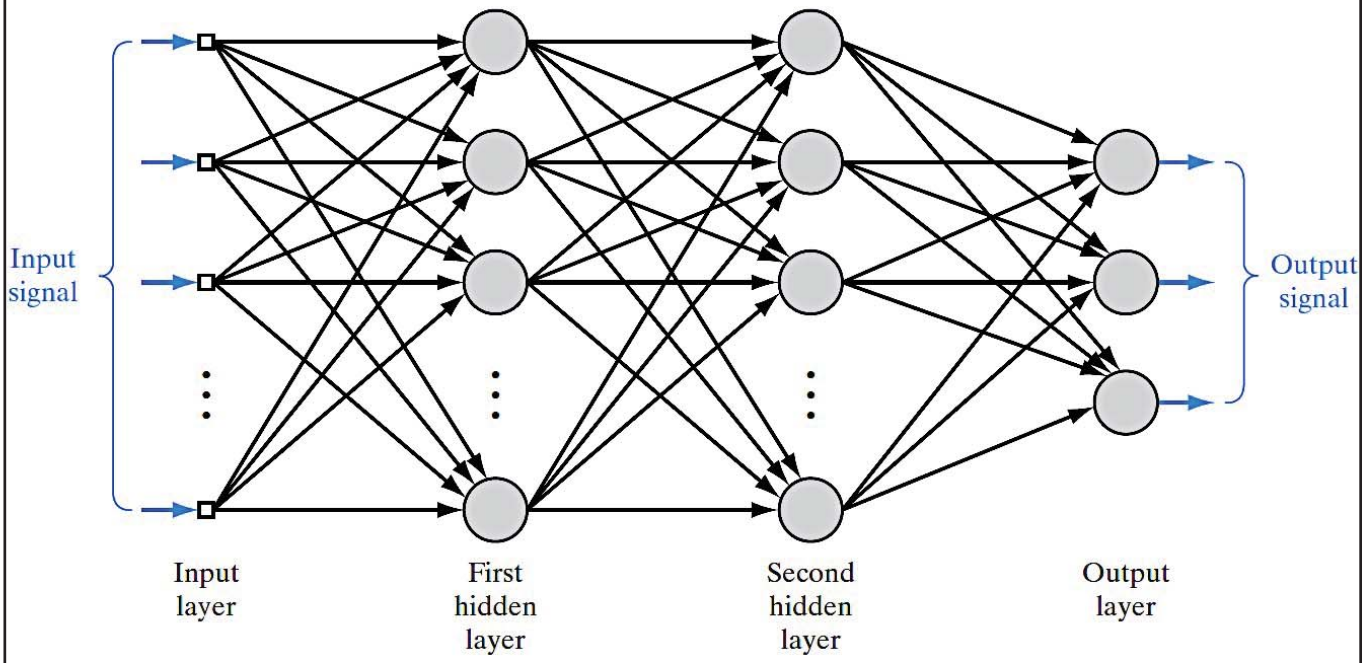
Computational Intelligence I: Neural Networks

Chapter 5: Multi Layer Perceptrons (MLPs)



Architecture of MLPs



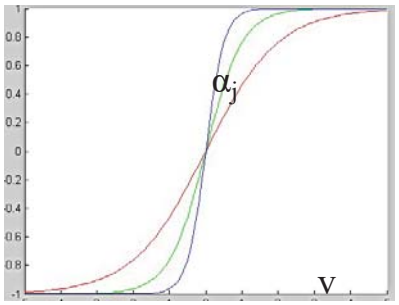
A general example (we also saw such architectures in Chapter 2)

Characteristics of MLPs

⌘ MLPs have certain characteristics which differentiate them from other networks:

i. Nonlinear activation functions which are smooth and differentiable (they cannot be the discontinuous Heaviside, for example). Examples include the logistic function which takes the ILF v_j (i.e., the weighted sum of all inputs to the j_{th} neuron and its bias) and transforms it to the output y_j:

$$y_j = \frac{1}{1+\exp\left(-\alpha_j v_j\right)}$$

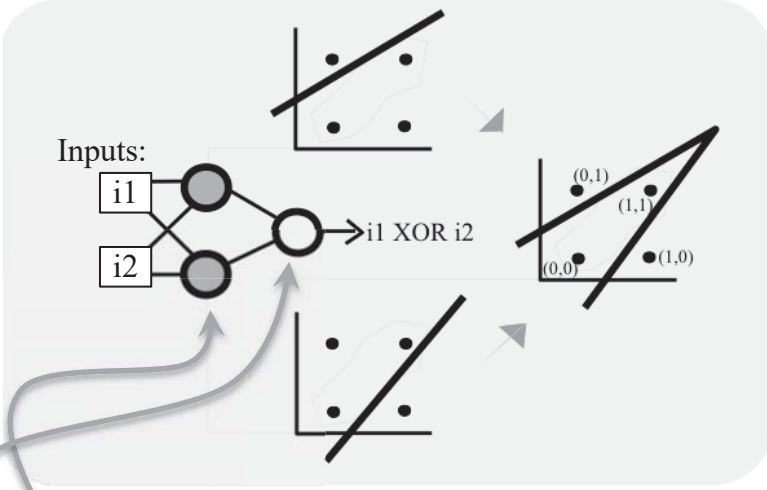


ii. Unlike SLPs, MLPs have one or more layers of neurons: the hidden layers. This allows the learning of more complex tasks, because each layer "generates" more complex features than its previous layer.

iii. They have high degrees of connectivity, since each node can receive input from any previous (only previous layer, to avoid feedback loops or cycles in the graphs).

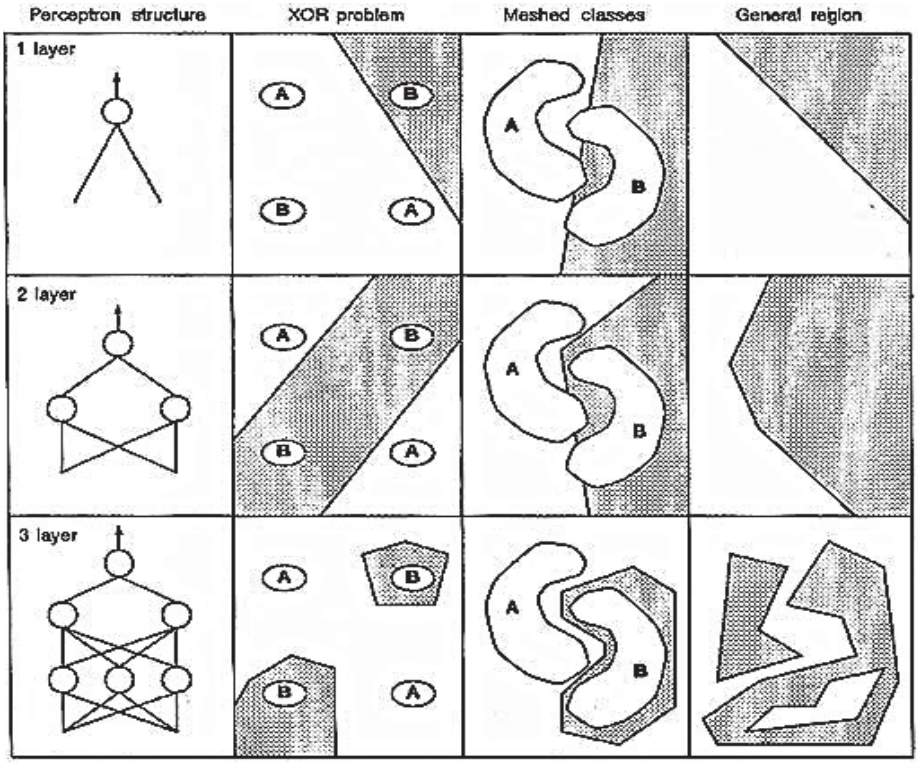
The function of hidden layers

- The presence of hidden layers can allow complex representations. Consider the XOR problem for instance, which is not a linearly separable one:



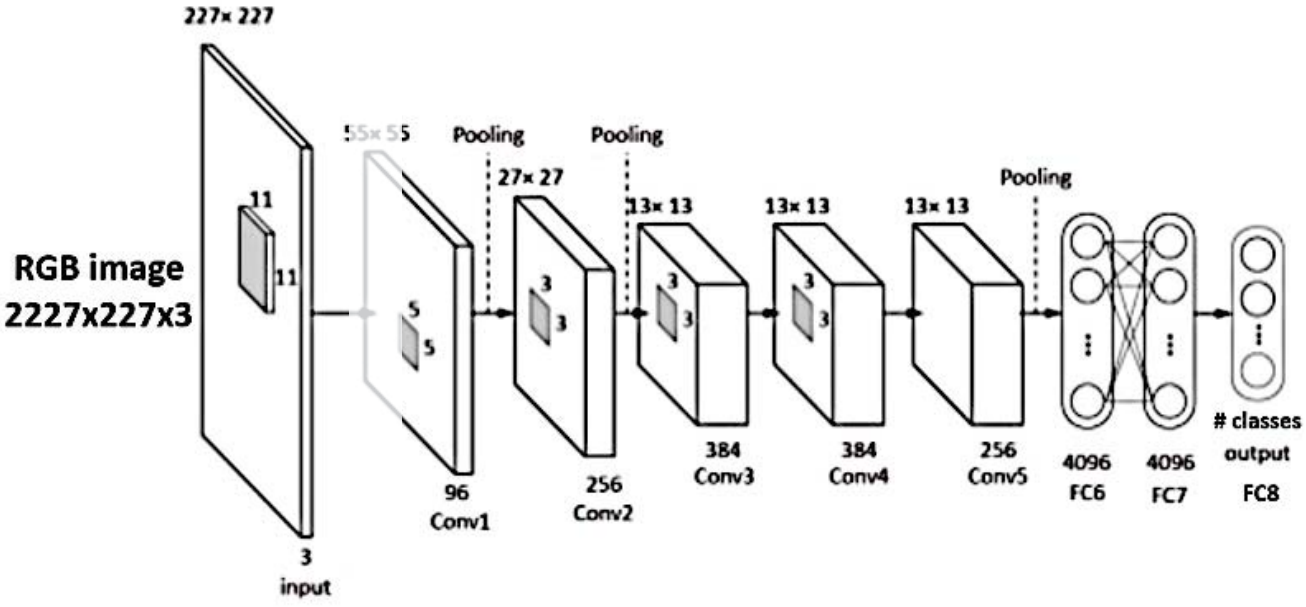
  - Each of the **hidden** nodes above, **finds a partial** solution to the problem.
  - Then, the **output** node **combines the two partial** solutions and makes the problem separable.

⌘ Any more layers? Of course! The **logic is the same**: more complex decision boundaries are progressively formed in the next layer:



⌘ State-of-the-art deep learning approaches use many hidden layers to learn complex patterns:

◎ Deep models, e.g. Alexnet, require a lot of training data and are computationally expensive (e.g., require the use of GPUs).



The Back-Propagation (BP) Algorithm: Derivation

- This is a very popular computationally efficient (but not unique) method for training MLPs. Although it is not optimal it discards the pessimism of some SLP pioneers about the uselessness of multi-layer learning!

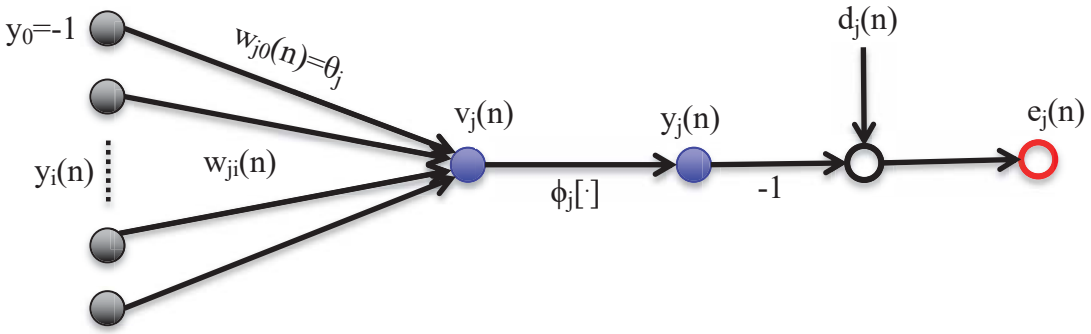
- The algorithm is as follows:

  - We define the error signals at the j^{th} output neuron level with target output d_j and at the network level as:

Output neuron error: $$e_j(n) = d_j(n) - y_j(n)$$

Output network error: $$E(n) = \frac{1}{2}\sum_{j \in \text{Output}} e_j^2(n)$$

Average net. output error: $$E_{avg} = \frac{1}{N}\sum_{n=1}^{N} E(n) \quad \text{(for N data patterns)}$$



  - We now define the ILF and neuron output as:

Induced local field: $$v_j(n) = \sum_{i=0}^{p} w_{ji}(n)\, y_i(n)$$

Neuron output: $$y_j(n) = \phi_j\left[v_j(n)\right]$$

⌘ We now wish to train the MLP using our available data patterns ($\mathbf{x}_i$,$\mathbf{d}_i$), i=1,…,N.

◎ But the goal is always the same:
  ⊙ Adapt each synaptic weight: $\Delta w_{ji}(n) \equiv w_{ji}(n+1) - w_{ji}(n)$

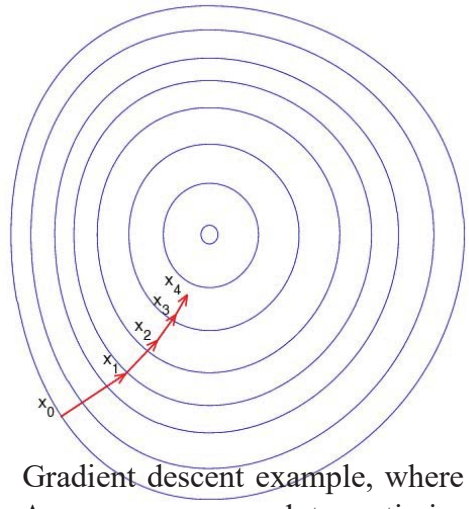
◎ First we calculate the derivative of the network error E(n) at time n with respect to the synaptic weight that needs to be adjusted to minimise this error:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \underbrace{\frac{\partial E(n)}{\partial e_j(n)}}_{e_j(n)} \underbrace{\frac{\partial e_j(n)}{\partial y_j(n)}}_{-1} \underbrace{\frac{\partial y_j(n)}{\partial v_j(n)}}_{\phi_j'[v_j(n)]} \underbrace{\frac{\partial v_j(n)}{\partial w_{ji}(n)}}_{y_i(n)} = \overbrace{-e_j(n)\phi_j'[v_j(n)]}^{\delta_j(n)} y_i(n) =$$

$$= -\delta_j(n) y_i(n)$$

◎ So, given a learning rate η, a simple gradient descent optimisation, would move in the weight space towards the direction of:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} = \eta \delta_j(n) y_i(n) \quad (1)$$



Gradient descent example, where **Δx** moves are used to optimise some function f(**x**).

◎ This direction which minimises E(n), is based on the newly introduced **local gradient** of the j[th] neuron, which is defined as the product of the error signal of that neuron and the derivative of its activation with respect to the ILF of the neuron:

$$\delta_j(n) \equiv e_j(n)\phi_j'[v_j(n)] \quad (A)$$

⌘ Thus, to calculate Δw_{ji} the error signal at the output of the j^{th} neuron is needed. But **not all** neurons have directly known output! This gives rise to two cases:

a) **j is an output neuron:** Here it is easy to calculate the weight change Δw_{ji}, since $e_j(n)=d_j(n)-y_j(n)$, and d_j is known by the training dataset.

b) **j is a hidden neuron:** Here the error is **not** directly known, **but** these neurons surely have a responsibility for the error observed in the output layer.

⌘ But how can we penalise or reward the hidden neuron when their errors $e_j(n)$ are not directly available because their desired responses $d_j(n)$ are not? This is an instance of the previously discussed ***Credit Assignment Problem***!

⌘ This is solved by "back-propagating" the signal error from the output layer through the previous hidden layers, as follows:

◎ Using the previous definitions, we can conceptually redefine the local gradient as:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} = \boxed{-\frac{\partial E(n)}{\partial y_j(n)}\phi_j'[v_j(n)]}^{2} = e_j(n)\phi'[v_j(n)]$$

◎ Now, we only(!) need to calculate $\partial E/\partial y_j$, which is explained in detail as follows.

⌘ Continued….

- For notational simplicity, we use subscript k to denote the k^{th} neuron following the hidden j^{th} neuron.
- The network output error was earlier defined as:

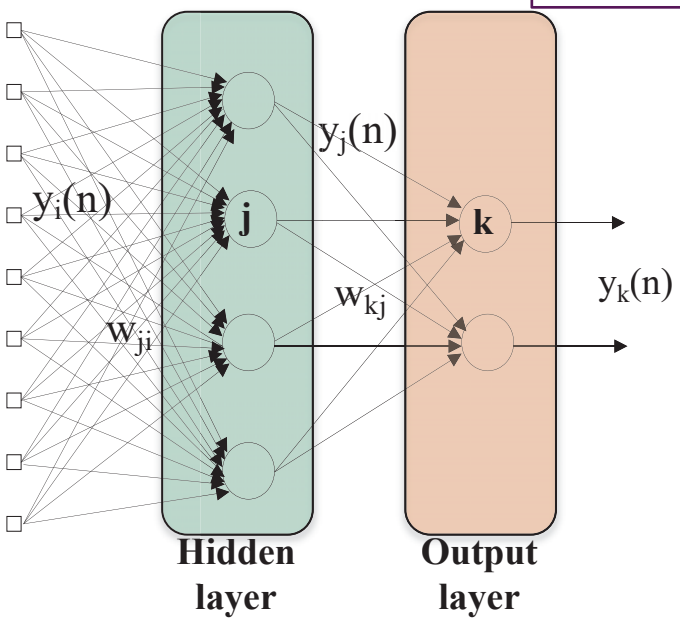
$$E(n) = \frac{1}{2}\sum_{k \in Output} e_k^2(n)$$

- Differentiating, expanding with the chain-rule and using the definition of the error $e_k(n)$ of an output neuron, we get:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)} = \sum_k e_k(n) \quad \frac{\partial e_k(n)}{\partial v_k(n)} \quad \frac{\partial v_k(n)}{\partial y_j(n)} \qquad (3)$$

$$\frac{\partial e_k(n)}{\partial v_k(n)} = \frac{\partial\left(d_k(n) - y_k(n)\right)}{\partial v_k(n)} = -\frac{\partial y_k(n)}{\partial v_k(n)} = -\phi_k'\left[v_k(n)\right]$$

substitute



- We know that the ILF of the k^{th} neuron with p inputs and bias w_{k0} is:

$$v_k(n) = \sum_{i=0}^{p} w_{ki}(n)\, y_i(n)$$

- Differentiating (as usual !), we get:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

substitute

So, with the two substitutions, Eq(3) becomes:

⌘ Continued….

- The two substitutions give this version of Eq(3):

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k \underbrace{e_k(n)\phi'_k\left[v_k(n)\right]}_{\delta_k(n)} w_{kj}(n)$$

- Finally, inserting the above into Eq(2), we get the back-propagation formula for the local gradient in the j^{th} hidden neuron:

$$\delta_j(n) = \phi'_j\left[v_j(n)\right]\sum_k \delta_k(n) w_{kj}(n) \quad \text{B}$$

- Eq(B) allows us to apply the weight change delta-rule of Eq(1) to hidden neurons. It can be seen that the δ_j of a hidden node can be calculated in terms of the δ_k of the k^{th} neurons connected (i.e., receiving signals) from neuron j.

- This means that Eq(B) is valid for j being a neuron in any hidden layer, whilst neuron k can be a neuron from the next hidden **or** the output layer.

- Overall, Eq(A) is used for the output layer, and Eq(B) for all the hidden ones.

The Back-Propagation (BP) Algorithm: the full summary

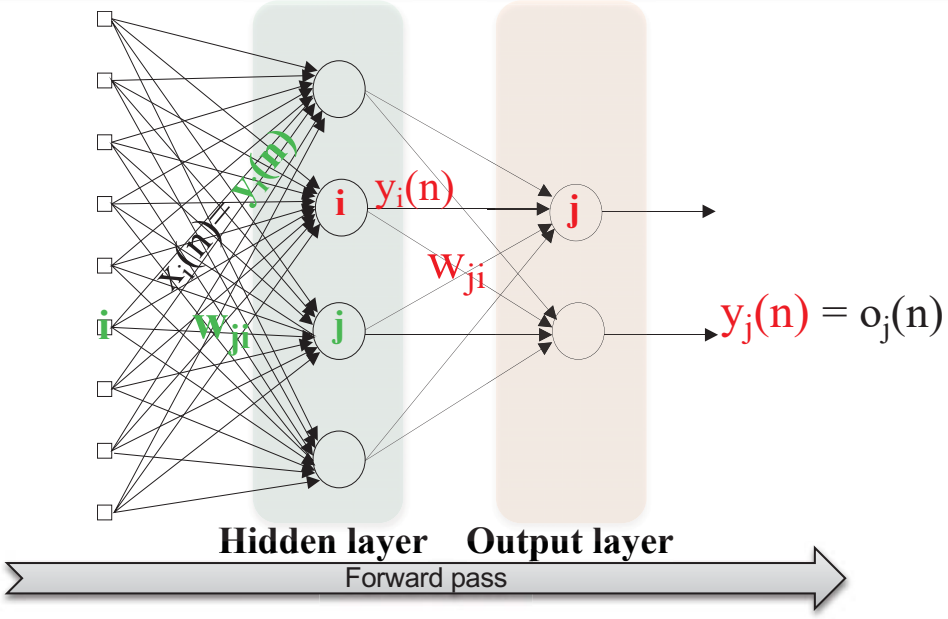
⌘ The BP algorithm for training MLPs consists of two passes of computation, which are alternatively applied.

Forward pass:

- All weights remain fixed.
- We start from the first hidden layer and for all its neurons, we calculate their outputs and ILFs as:

$$y_j(n) = \phi_j\left[v_j(n)\right], \qquad v_j(n) = \sum_{i=0}^{p} w_{ji}(n)\, y_i(n)$$

- If the above is applied to the first hidden layer, all $y_i(n)$ simply represent the network input signals $x_i(n)$.
- We repeatedly move to the next layer, until we reach the output layer where $y_j(n)$ is the j^{th} network output, typically also written as $o_j(n)$.
- If the above is applied to the 2nd, 3rd, etc. hidden layer or the output layer, $y_i(n)$ is output signal of the i^{th} node in the previous hidden layer.



⌘ Continued….

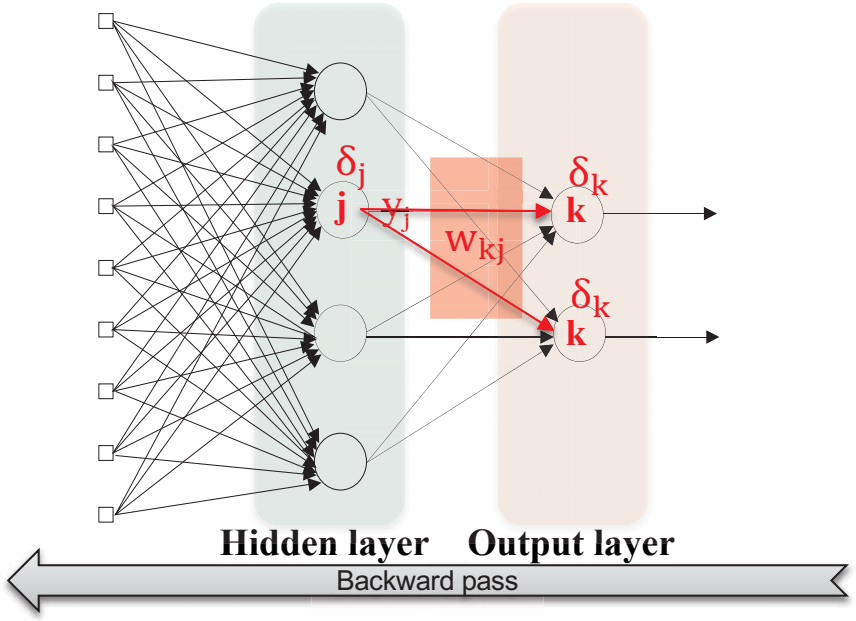
Backward pass:

- This pass starts from the last (the output layer) and works backwards in a layer-by-layer fashion, until it reaches the first hidden layer.
- All input signals to nodes are clamped during this pass.
- For each j^{th} node of the current layer it calculates the local gradient δ_j using:
  - Eq(A), i.e. $\delta_j(n) = e_j(n)\phi'_j[v_j(n)]$

    if the node is in the output layer, or
  - Eq(B), i.e. $\delta_j(n) = \phi'_j[v_j(n)]\sum_k \delta_k(n) w_{kj}(n)$

    if the node is in any hidden layer.
- All the local gradients are then used in Eq(1) to calculate Δw_{ji} and adapt the synaptic weights so that to reduce the network error:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} = \eta \delta_j(n) y_i(n)$$



BP activation functions

⌘ Different types of activations functions $y_j(n)=\phi_j(n)$ can be used in BP, assuming they are smooth nonlinear ones:

◎ The **Logistic function**, which has an easily expressed derivative. This is *simply* substituted within Eqs(A&B) of the local gradient δ_j of the output and hidden neurons:

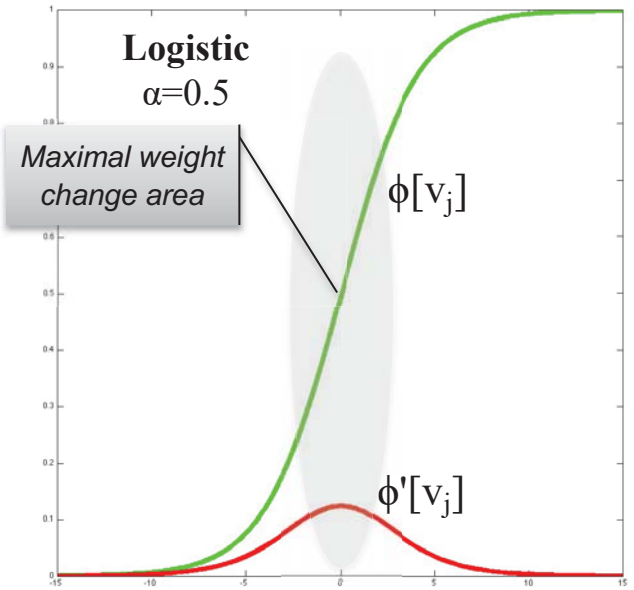
$$\phi_j\left[v_j(n)\right]=\frac{1}{1+\exp\left(-\alpha v_j(n)\right)}\equiv y_j(n)\in(0,1),\quad \alpha>0\Rightarrow$$

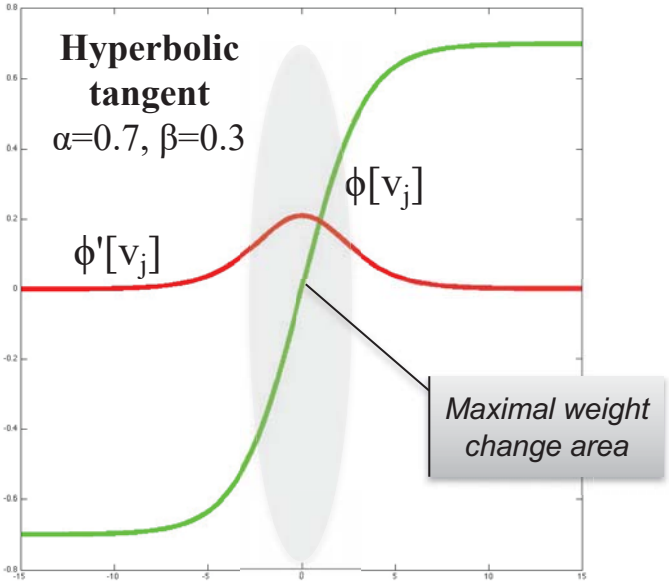
$$\phi_j'\left[v_j(n)\right]=\frac{\alpha\exp\left(-\alpha v_j(n)\right)}{\left[1+\exp\left(-\alpha v_j(n)\right)\right]^2}\Rightarrow \boxed{\phi_j'\left[v_j(n)\right]=\alpha y_j(n)\left[1-y_j(n)\right]}$$

◎ Another alternative is the **Hyperbolic tangent function** which also represent a sigmoid-shaped nonlinear mapping.

$$\phi_j\left[v_j(n)\right]=\alpha\tanh\left(\beta v_j(n)\right)\equiv y_j(n)\in(-\alpha,+\alpha),\quad a,\beta>0\Rightarrow$$

$$\phi_j'\left[v_j(n)\right]=\alpha\beta\left[1-\tanh\left(\beta v_j(n)\right)^2\right]\Rightarrow \boxed{\phi_j'\left[v_j(n)\right]=\frac{\beta}{\alpha}\left[\alpha^2-y_j^2(n)\right]}$$





BP in practice: training modes

⌘ How can we apply a training dataset to train an MLP:

- The training samples ($\mathbf{x}_i$,$\mathbf{d}_i$) i=1,…,N are presented to the network one by one. A complete presentation of all patterns is called an epoch.
- In the **Sequential training mode** (**STM**), the forward and backward passes are executed for each sample independently, and hence, weights are adjusted per pattern, until the last sample is processed and the current epoch is completed. The previous equations are used.
- In the **Batch training mode** (**BTM**), the weights are adapted only after all N samples are presented to the network. This mode is based on the error formulated over all patterns and network output signals:

$$E_{avg} = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in Output} e_j^2(n)$$

- In this case, proceeding as in BP derivation, we get the update:

$$\Delta w_{ji}(n) = -\eta\frac{\partial E_{av}}{\partial w_{ji}(n)} = -\frac{\eta}{N}\sum_{n=1}^{N} e_j(n)\frac{\partial e_j(n)}{\partial w_{ji}(n)}$$

Can be calculated as for the STM

- The above mode updates all weights after all patterns are processed.
- The STM requires less storage than the BTM and its stochastic pattern-by-pattern update can avoid local minima common in deep models.
- But the BTM is easier to mathematically analyse and parallelise.
- In the **Mini-batch training mode** (**MBTM**) the training data is divided into mini-batches and the weights are adapted after computing the gradient for each mini-batch.
- It is a compromise between sequential and batch training modes and the standard approach nowadays.
- It improves generalisation to new unobserved samples.

BP in practice: learning speed

⌘ Another issue commonly used in MLPs is the learning speed.

◎ So far in Eq(1) we saw how the **learning rate** parameter η regulates the pressure in weight update:

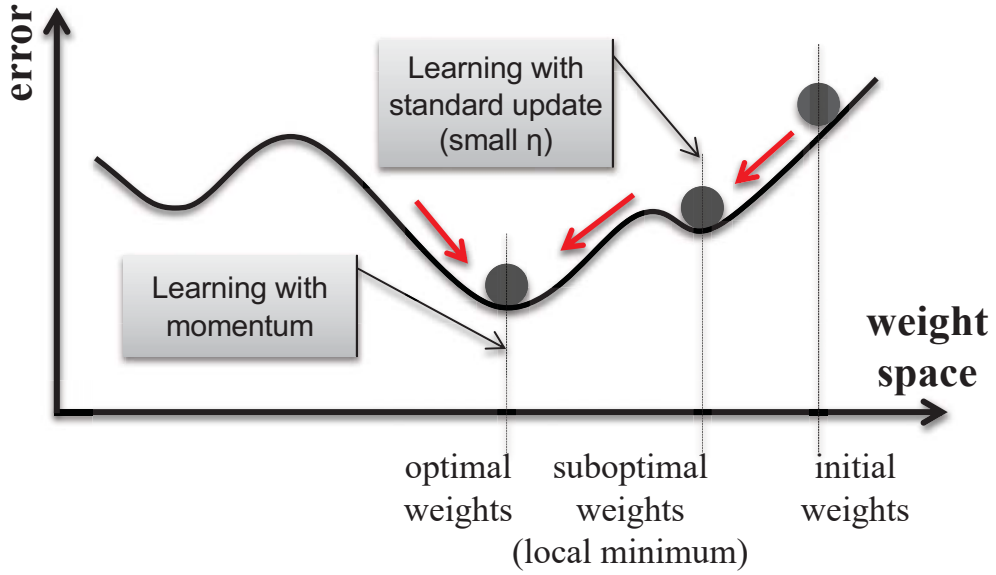
$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

◎ The smaller the η, the smaller the synaptic changes and the smoother the trajectory in the weight space. When η is large, it accelerates learning, but it may cause instabilities (oscillations).

◎ A very commonly used modification which increases learning speed but suppresses oscillatory behaviour is the **Momentum based learning** rule (also called the **Generalised Delta Rule**):

$$\Delta w_{ji}(n) = \underbrace{\eta \delta_j(n) y_i(n)}_{\text{standard BP update term}} + \underbrace{\mu \Delta w_{ji}(n-1)}_{\text{momentum term}}$$

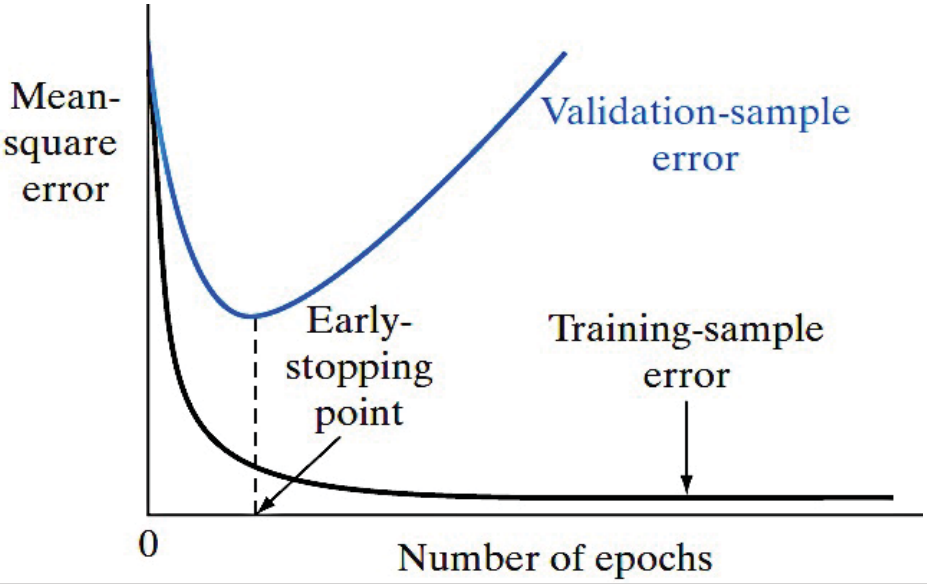
◎ μ is the momentum term and controls the momentum influence.



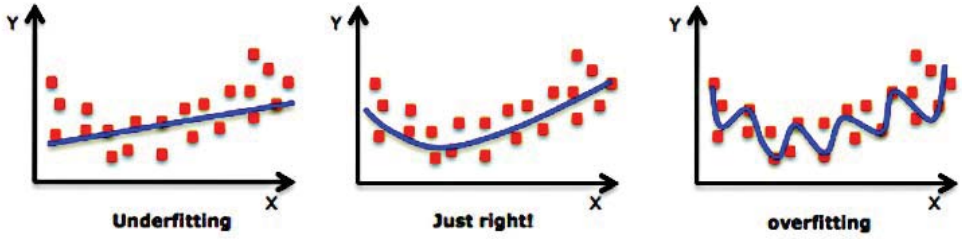
◎ There are also commonly used algorithms with adaptive learning rates and more complex heuristics, such as Adam, AdaGrad, RMSProp, etc.

BP in practice: controlling complexity

- The "hope" is that the MLP becomes "well" trained, so that it learns enough about the environment to generalise unseen data.
- But an MLP **may overfit** the training data and **not generalise** well to unseen data, especially when the MLP has a high-capacity (many layers/neurons = high complexity).
- We can divide the data set into three disjoint sets:
  - **Training set**: to execute BP and adapt model weights.
  - **Test set**: to assess the final model.
  - **Validation set**: to select model hyperparameters (network structure).
- The BP error typically starts off at a large value, decreases rapidly, and then continues to decrease slowly as the network makes its way to a local minimum on the error surface.
- When good generalisation is the goal, it is very difficult to figure out when it is best to stop training; so we can use **Early Stopping**:
  - Every a few epochs, the network error in the validation set is obtained.
  - If this seems to increase, then stop training!
  - Note: the training error appears to reduce and that we could do better. In reality, however, the network beyond this point is merely learning noise or isolated characteristics contained in the specific training patterns.



⌘ Apart from Early Stopping, there are other ways to avoid **overfitting**, and increase the ability to generalise over unseen patterns.



⌘ One way is to use **Regularisation theory**, and redefine the error to include: the standard network mean squared error E_{avg}, and also a smoothness error E_S which penalises complexity (controlled by a regularisation term λ). Both error terms depend on the weight **w**:

$$E(\mathbf{w}) = E_{avg}(\mathbf{w}) + \lambda E_S(\mathbf{w})$$

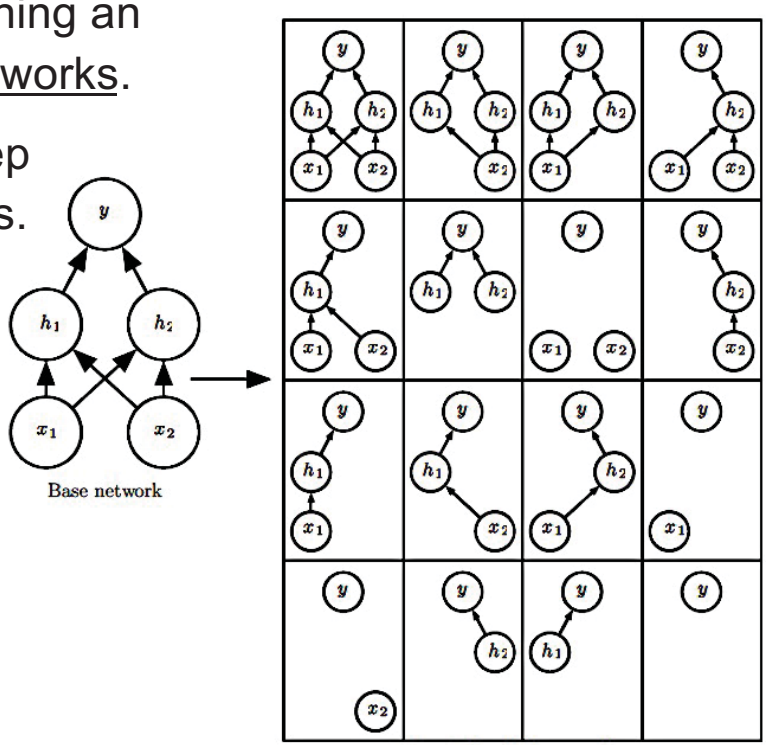
◎ **Penalisation of derivatives**: In general E_S can be expressed so that it minimises the k^{th} order derivatives of the model M(**w**,**x**) (this is our entire MLP function) within a region ξ(**x**) of the input space. k controls the degree of smoothness: the lower k is, the smoother (less complex) the function **M**(**w**,**x**):

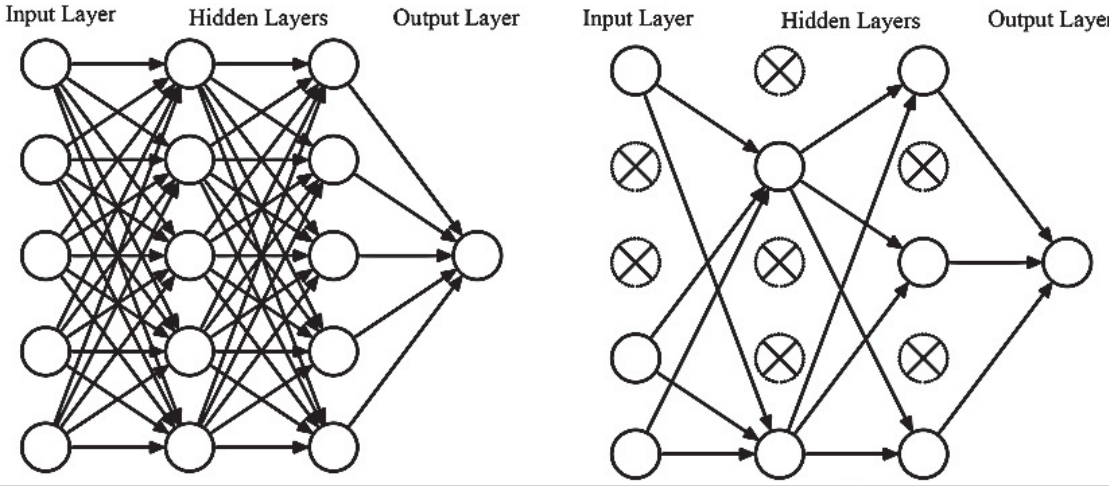
$$E_S(\mathbf{w}) = \frac{1}{2}\int \left\| \frac{\partial^k}{\partial \mathbf{x}^k} \mathbf{M}(\mathbf{w},\mathbf{x}) \right\|_2^2 \xi(\mathbf{x}) d\mathbf{x}$$

◎ **Weight Decay**: This simple method defines E_S in such a way so that it forces some *excess* weights which have no much influence in the network operation to assume very small magnitude (otherwise they would assume arbitrary high values that affect unnecessarily the MLP):

$$E_S(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \sum_i w_i^2$$

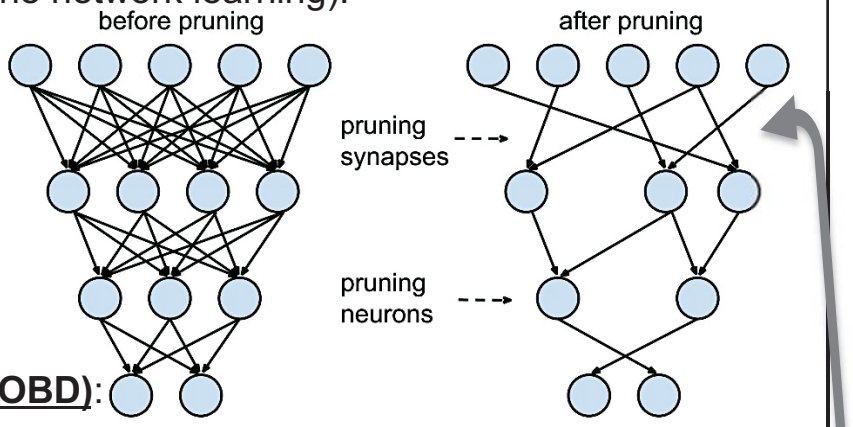
- The **Dropout** method:
  - At each training step, each node in the network (or in some hidden layers) is removed with a certain probability.
  - During testing, all nodes in the network are used but their output is multiplied by the probability of including this node.
  - We are actually training an ensemble of subnetworks.
  - Broadly used in deep learning approaches.





◎ **Weight Elimination**: This method employs a user-defined controller α. When $|w_i| \ll \alpha$, the penalty for the i^{th} weight becomes almost zero (this assumes this weight is unreliable and should be eliminated from the MLP). When $|w_i| \gg \alpha$, the penalty approaches unity (this assumes that the i^{th} weight is important for the network learning):

$$E_S(\mathbf{w}) = \sum_i \frac{(w_i/\alpha)^2}{1+(w_i/\alpha)^2}$$



◎ **Optimal Brain Damage (OBD)**:

◎ This method uses second-order differential information of the error surface $E_{avg}(\mathbf{w})$ to control complexity. It removes weights (**network pruning**) after training has been completed.

◎ It uses Taylor-series expansion of the error, with the gradient and Hessian **H** evaluated at **w**:

$$\left(\mathbf{H} \equiv \partial^2 E_{avg}(\mathbf{w}) / \partial \mathbf{w}^2\right)$$

$$E_{avg}(\mathbf{w}+\Delta\mathbf{w}) = E_{avg}(\mathbf{w}) + \nabla E_{avg}^T(\mathbf{w})\Delta\mathbf{w} + \tfrac{1}{2}\Delta\mathbf{w}^T\mathbf{H}(\mathbf{w})\Delta\mathbf{w} + \text{h.o.t.}$$

◎ Then assuming the error around the located minimum is quadratic (this rids the higher terms of the expansion), we get:

$$\Delta E_{avg} \cong \tfrac{1}{2}\Delta\mathbf{w}^T\mathbf{H}(\mathbf{w})\Delta\mathbf{w}$$

◎ Another assumption is that **H**(**w**) is diagonal to avoid calculation of the cross-weight terms and simplify the procedure. The **Optimal Brain Surgeon (OBS)** procedure does not have this assumption (full Hessian).

◎ The goal is to set one of the weights to zero, but with minimal increase in the error $E_{avg}(\mathbf{w})$. This requires to solve a two-level optimisation. The resulting $\Delta\mathbf{w}$ is used to update the trained weights accordingly and prune w_i.

$$\min_i \left\{ \min_{\Delta\mathbf{w}} \left\{ \tfrac{1}{2}\Delta\mathbf{w}^T\mathbf{H}(\mathbf{w})\Delta\mathbf{w} \;\middle|\; \text{s.t.} \begin{array}{l} \Delta\mathbf{w}_i + w_i = 0 \\ \text{i.e., } w_i^{(new)} = 0 \end{array} \right\} \right\}$$

<END of Chapter 5>