Multi Layer Perceptron

Dr. Kingshuk Srivastava

Contents

Contents

- MLP model details
- Back-propagation algorithm
- XOR Example
- Heuristics for Back-propagation
- Heuristics for learning rate
- Approximation of functions
- Generalisation
- Model selection through cross-validation
- Conguate-Gradient method for BP

Contents II

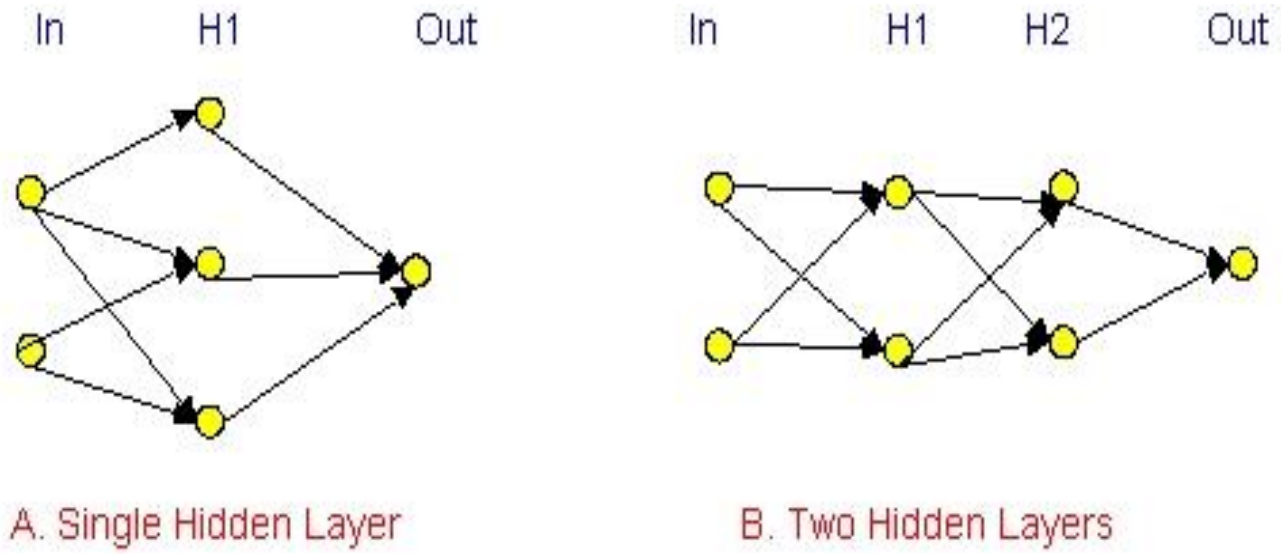
Contents

- Advantages and disadvantages of BP
- Types of problems for applying BP
- Conclusions

Multi Layer Perceptron

MLP Model

- "Neurons" are positioned in layers. There are Input, Hidden and Output Layers



Multi Layer Perceptron Output

MLP Model

- The output y is calculated by:

$$y_j(n) = \varphi_j(v_j(n)) = \varphi_j\left(\sum_{i=0}^{m} w_{ji}(n) y_i(n)\right)$$

Where $w_0(n)$ is the **bias**.

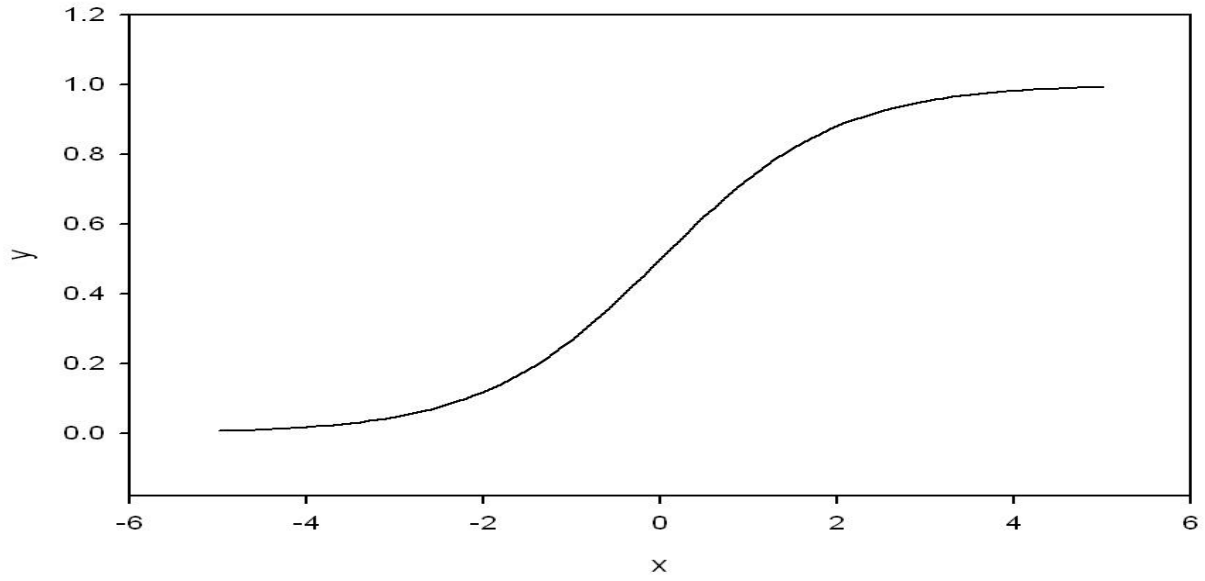
- The function $\phi_j(\bullet)$ is a *sigmoid* function. Typical examples are:

Transfer Functions

MLP Model

- The *logistic sigmoid*:

$$y = \frac{1}{1+\exp(-x)}$$

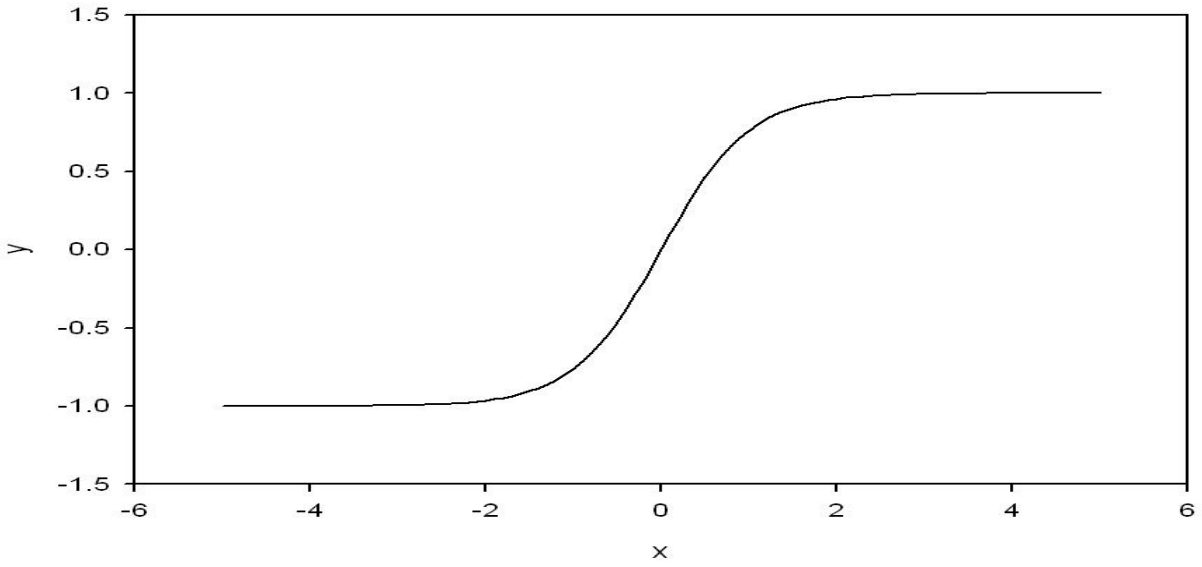


Transfer Functions II

MLP Model

- The *hyperbolic tangent sigmoid*:

$$y = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{\frac{(\exp(x) - \exp(-x))}{2}}{\frac{(\exp(x) + \exp(-x))}{2}}$$



Learning Algorithm

BP Algorithm

- Assume that a set of examples $\Im=\{\mathbf{x}(n),\mathbf{d}(n)\}$, n=1,…,N is given. **x**(n) is the *input vector* of dimension m_0 and d(n) is the *desired response* vector of dimension M
- Thus an *error signal*, $e_j(n)=d_j(n)-y_j(n)$ can be defined for the output neuron j.
- We can derive a learning algorithm for an MLP by assuming an optimisation approach which is based on the **steepest descent direction**, I.e.

$\Delta\mathbf{w}(n)=-\eta\mathbf{g}(n)$

Where **g**(n) is the gradient vector of the cost function and η is the *learning rate*.

Learning Algorithm II

BP Algorithm

- The algorithm that it is derived from the steepest descent direction is called **back-propagation**
- Assume that we define a SSE instantaneous cost function (I.e. per example) as follows:

$$\mathbf{E}(n) = \frac{1}{2}\sum_{j \in C} e_j{}^2(n)$$

Where C is the set of all *output neurons*.

- If we assume that there are N examples in the set $\Im$ then the *average squared error* is:

$$\mathbf{E}_{av} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{E}(n)$$

- We need to calculate the gradient wrt E_{av} or wrt to E(n). In the first case we calculate the gradient per *epoch* (i.e. in all patterns N) while in the second the gradient is calculated per *pattern*.
- In the case of E_{av} we have the **Batch** mode of the algorithm. In the case of E(n) we have the **Online** or **Stochastic** mode of the algorithm.
- Assume that we use the online mode for the rest of the calculation. The gradient is defined as:

$$\vec{g}(n) = \frac{\partial \mathrm{E}(n)}{\partial w_{ji}(n)}$$

Learning Algorithm IV

BP Algorithm

- Using the chain rule of calculus we can write:

$$\frac{\partial \mathrm{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathrm{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

- We calculate the different partial derivatives as follows:

$$\frac{\partial \mathrm{E}(n)}{\partial e_j(n)} = e_j(n)$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

Learning Algorithm V

BP Algorithm

- And,

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi_j'(v_j(n))$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

- Combining all the previous equations we get finally:

$$\Delta w_{ij}(n) = -\eta \frac{\partial \mathrm{E}(n)}{\partial w_{ji}(n)} = \eta e_j(n)\phi_j'(v_j(n))y_i(n)$$

Learning Algorithm VI

BP Algorithm

- The equation regarding the weight corrections can be written as:

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

Where $\delta_j(n)$ is defined as the *local gradient* and is given by:

$$\delta_j(n) = -\frac{\partial \mathrm{E}(n)}{\partial v_j(n)} = -\frac{\partial \mathrm{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = e_j(n) \phi_j'(v_j(n))$$

- We need to distinguish two cases:
  - j is an output neuron
  - j is a hidden neuron

Learning Algorithm VII

- Thus the Back-Propagation algorithm is an *error-correction* algorithm for supervised learning.

BP Algorithm

- If j is an output neuron, we have already a definition of $e_j(n)$, so, $\delta_j(n)$ is defined (after substitution) as:

$$\delta_j(n) = (d_j(n) - y_j(n))\phi_j'(v_j(n))$$

- If j is a hidden neuron then $\delta_j(n)$ is defined as:

$$\delta_j(n) = -\frac{\partial \mathrm{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial \mathrm{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n))$$

Learning Algorithm VIII

BP Algorithm

- To calculate the partial derivative of E(n) wrt to $y_j(n)$ we remember the definition of E(n) and we change the index for the output neuron to k, i.e.

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

- Then we have:

$$\frac{\partial \mathrm{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)}$$

BP Algorithm

- We use again the chain rule of differentiation to get the partial derivative of $e_k(n)$ wrt $y_j(n)$:

$$\frac{\partial \mathrm{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

- Remembering the definition of $e_k(n)$ we have:

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_v(n))$$

- Hence:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k{}'(v_k(n))$$

- The *local field* $v_k(n)$ is defined as:

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n)$$

Where m is the number of neurons (from the previous layer) which connect to neuron k. Thus we get:

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

- Hence:

$$\frac{\partial \mathrm{E}(n)}{\partial y_j(n)} = -\sum_{k \in C} e_k(n) \phi_k'(v_k(n)) w_{kj}(n)$$

$$= -\sum_{k \in C} \delta_k(n) w_{kj}(n)$$

Learning Algorithm XI

BP Algorithm

- Putting all together we find for the local gradient of a hidden neuron j the following formula:

$$\delta_j(n) = \phi_j'(v_j(n)) \sum_{k \in C} \delta_k(n) w_{kj}(n)$$

- It is useful to remember the special form of the derivatives for the logistic and hyperbolic tangent sigmoids:
  - $\phi_j'(v_j(n)) = y_j(n)[1-y_j(n)]$ (Logistic)
  - $\phi_j'(v_j(n)) = [1-y_j(n)][1+y_j(n)]$ (Hyp. Tangent)

Summary of BP Algorithm

BP Algorithm

1. *Initialisation:* Assuming that no prior infromation is available, pick the synaptic weights and thresholds from a uniform distribution whose mean is zero and whose variance is chosen to make the std of the local fields of the neurons lie at the transition between the linear and saturated parts of the sigmoid function
2. *Presentation of training examples:* Present the network with an epoch of training examples. For each example in the set, perform the sequence of the forward and backward computations described in points 3 & 4 below.

3. *Forward Computation:*
   - Let the training example in the epoch be denoted by (**x**(n),**d**(n)), where **x** is the input vector and **d** is the desired vector.
   - Compute the local fields by proceeding forward through the network layer by layer. The local field for neuron j at layer l is defined as:

$$v_j^{(l)}(n) = \sum_{i=0}^{m} w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

where m is the number of neurons which connect to j and $y_i^{(l-1)}(n)$ is the activation of neuron i at layer (l-1). $W_{ji}^{(l)}(n)$ is the weight

Summary of BP Algorithm III

BP Algorithm

which connects the neurons j and i.

- For i=0, we have $y_0^{(l-1)}(n)=+1$ and $w_{j0}^{(l)}(n)=b_j^{(l)}(n)$ is the bias of neuron j.
- Assuming a sigmoid function, the output signal of the neuron j is:

$$y_j^{(l)}(n) = \varphi_j(v_j^{(l)}(n))$$

- If j is in the input layer we simply set:

$$y_j^{(0)}(n) = x_j(n)$$

where $x_j(n)$ is the jth component of the input vector **x**.

Summary of BP Algorithm IV

BP Algorithm

- If j is in the output layer we have:

$$y_j^{(L)}(n) = o_j(n)$$

where $o_j(n)$ is the jth component of the output vector **o**. L is the total number of layers in the network.

- Compute the error signal:

$$e_j(n) = d_j(n) - o_j(n)$$

where $d_j(n)$ is the desired response for the jth element.

Summary of BP Algorithm V

4. *Backward Computation:*

- Compute the δs of the network defined by:

BP Algorithm

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \phi_j'(v_j^{(l)}(n))\sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases}$$

where $\phi_j(\bullet)$ is the derivative of function ϕ_j wrt the argument.

- Adjust the weights using the *generalised delta rule:*

$$\Delta w_{ji}^{(l)}(n) = \alpha \Delta w_{ji}^{(l)}(n-1) + \eta \delta_j^{(l)}(n) y_i^{(l-1)}(n)$$

where α is the **momentum constant**

Summary of BP Algorithm VI

5. *Iteration:* Iterate the forward and backward computations of steps 3 & 4 by presenting new epochs of training examples until the stopping criterion is met.

BP Algorithm

- The order of presentation of examples should be randomised from epoch to epoch
- The momentum and the learning rate parameters typically change (usually decreased) as the number of training iterations increases.

Stopping Criteria

BP Algorithm

- The BP algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.
- The BP is considered to have converged when the absolute value of the change in the average square error per epoch is sufficiently small

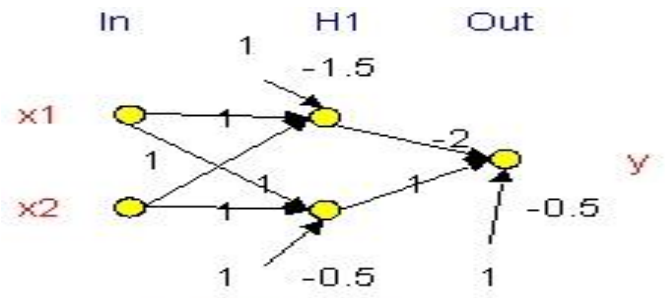
XOR Example

BP Algorithm

- The XOR problem is defined by the following truth table:

y	0	1	1	0
x1	0	1	0	1
x2	0	0	1	1

- The following network solves the problem. The perceptron could not do this. (We use Sgn func.)



XOR Network

Heuristics for Back-Propagation

BP Algorithm

- To speed the convergence of the back-propagation algorithm the following heuristics are applied:
  - H1: Use sequential (online) vs batch update
  - H2: Maximise information content
    - Use examples that produce largest error
    - Use example which very different from all the previous ones
  - H3: Use an antisymmetric activation function, such as the hyperbolic tangent. Antisymmetric means:

$$\phi(-x)=-\phi(x)$$

Heuristics for Back-Propagation II

BP Algorithm

- H4: Use different target values inside a smaller range, different from the asymptotic values of the sigmoid
- H5: Normalise the inputs:
  - Create zero-mean variables
  - Decorrelate the variables
  - Scale the variables to have covariances approximately equal
- H6: Initialise properly the weights. Use a zero mean distribution with variance of:

$$\sigma_w = \frac{1}{\sqrt{m}}$$

Heuristics for Back-Propagation III

where m is the number of connections arriving to a neuron

- H7: Learn from hints
- H8: Adapt the learning rates appropriately (see next section)

Heuristics for Learning Rate

BP Algorithm

- R1: Every adjustable parameter should have its own learning rate
- R2: Every learning rate should be allowed to adjust from one iteration to the next
- R3: When the derivative of the cost function wrt a weight has the same algebraic sign for several consecutive iterations of the algorithm, the learning rate for that particular weight should be increased.
- R4: When the algebraic sign of the derivative above alternates for several consecutive iterations of the algorithm the learning rate should be decreased.

Approximation of Functions

- Q: *What is the minimum number of hidden layers in a MLP that provides an approximate realisation of any continuous mapping?*

Approxim.

- A: **Universal Approximation Theorem**

Let $\phi(\bullet)$ be a nonconstant, bounded, and monotone increasing continuous function. Let I_{m0} denote the m_0-dimensional unit hypercube $[0,1]^{m0}$. The space of continuous functions on I_{m0} is denoted by $C(I_{m0})$. Then given any function $f \in C(I_{m0})$ and $\varepsilon > 0$, there exists an integer m_1 and sets of real constants a_i , b_i and w_{ij} where $i=1,\ldots, m_1$ and $j=1,\ldots, m_0$ such that we may

Approximation of Functions II

define:

$$F(x_1,\ldots,x_{m_0}) = \sum_{i=1}^{m_1} a_i \phi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right)$$

Approxim.

as an approximate realisation of function f(•); that is:

$$|F(x_1,\ldots,x_{m_0}) - f(x_1,\ldots,x_{m_0})| < \varepsilon$$

for all x_1, …, x_{m0} that lie in the input space.

Approximation of Functions III

Approxim.

- The Universal Approximation Theorem is directly applicable to MLPs. Specifically:
  - The sigmoid functions cover the requirements for function ϕ
  - The network has m_0 input nodes and a single hidden layer consisting of m_1 neurons; the inputs are denoted by $x_1, \ldots, x_{m0}$
  - Hidden neuron I has synaptic weights $w_{i1}, \ldots, w_{m0}$ and bias b_i
  - The network output is a linear combination of the outputs of the hidden neurons, with $a_1, \ldots, a_{m1}$ defining the synaptic weights of the output layer

Approximation of Functions IV

- The theorem is an *existence theorem*: It does not tell us exactly what is the number m_1; it just says that exists!!!

- The theorem states that a *single hidden layer is sufficient for an MLP to compute a uniform ε approximation to a given training set represented by the set of inputs x_1, …, x_{m0} and a desired output $f(x_1, \ldots, x_{m0})$.*

- The theorem **does not say** however that *a single hidden layer is optimum in the sense of the learning time, ease of implementation or generalisation.*

Approximation of Functions V

- Empirical knowledge shows that the number of data pairs that are needed in order to achieve a given error level ε is:

$$N = O\left(\frac{W}{\varepsilon}\right)$$

Approxim.

Where W is the total number of adjustable parameters of the model. There is mathematical support for this observation (but we will not analyse this further!)

- There is the **"curse of dimensionality"** for approximating functions in high-dimensional spaces.
- It is theoretically justified to use two hidden layers.

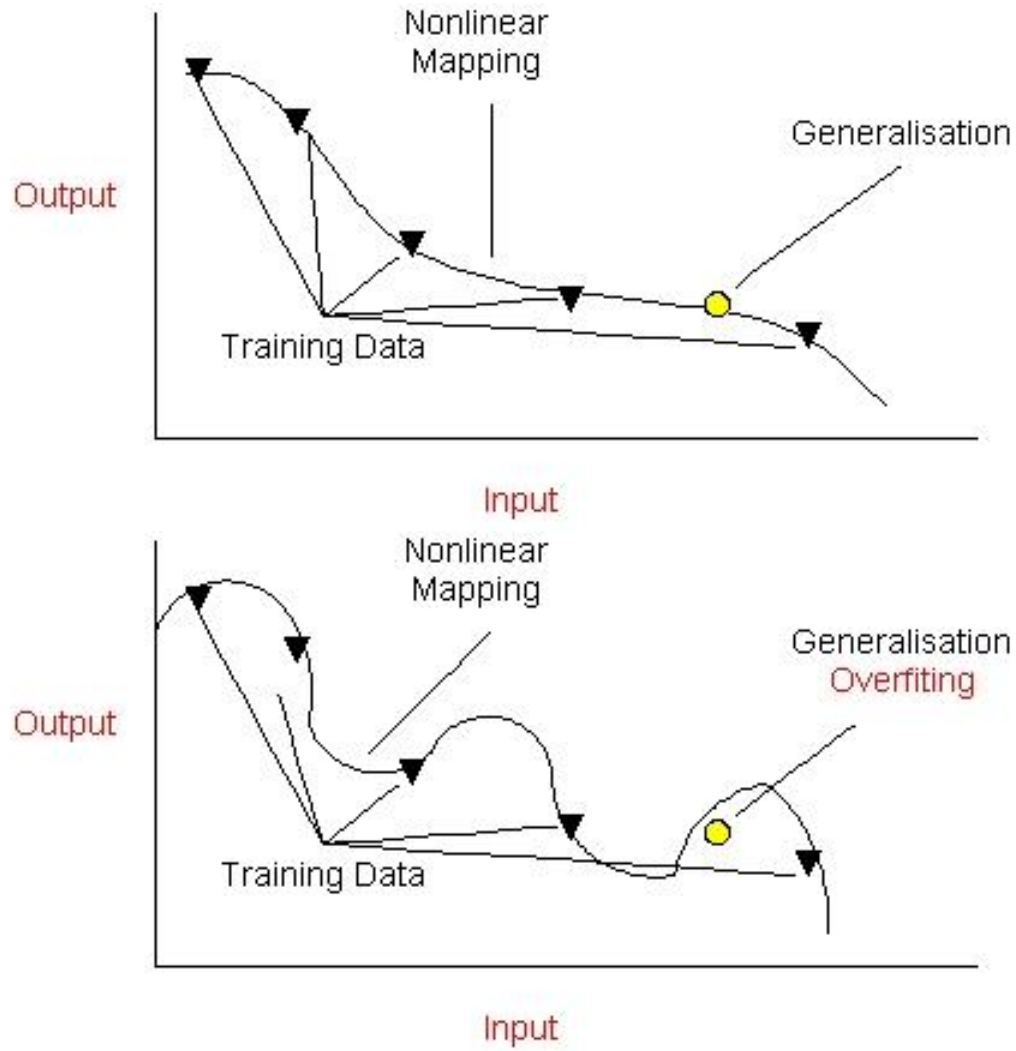
Generalisation

Def: A network *generalises* well when the input-output mapping computed by the network is correct (or nearly so) for test data never used in creating or training the network. It is assumed that the test data are drawn form the population used to generate the training data.

Model Selec.

- We should try to approximate the *true* mechanism that generates the data; not the specific structure of the data in order to achieve the generalisation. If we learn the specific structure of the data we have overfitting or overtraining.

Generalisation II



Model Selec.

Generalisation III

- To achieve good generalisation we need:
  - To have good data (see previous slides)
  - To impose smoothness constraints on the function
  - To add knowledge we have about the mechanism
  - Reduce / constrain model parameters:
    - Through cross-validation
    - Through regularisation (Pruning, AIC, BIC, etc)

Model Selec.

Cross Validation

- In cross validation method for model selection we split the training data to two sets:
  - *Estimation set*
  - *Validation set*
- We train our model in the estimation set.
- We evaluate the performance in the validation set.
- We select the model which performs "best" in the validation set.

Model Selec.

Cross Validation II

- There are variations of the method depending on the partition of the validation set. Typical variants are:
  - *Method of early stopping*
  - *Leave k-out*

Method of Early Stopping

- Apply the method of early stopping when the number of data pairs, N, is less than N<30W, where W is the number of free parameters in the network.
- Assume that r is the ratio of the training set which is allocated to the validation. It can be shown that the optimal value of this parameter is given by:

Model Selec.

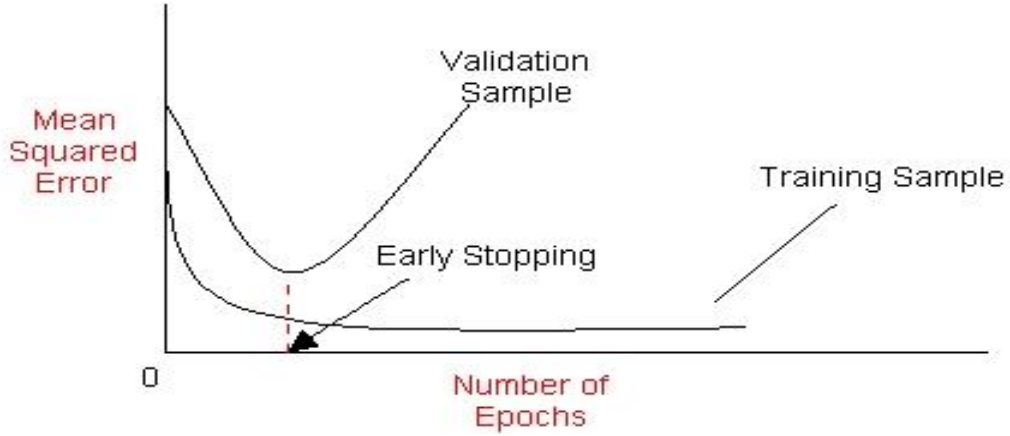
$$r_{opt} = 1 - \frac{\sqrt{2W - 1} - 1}{2(W - 1)}$$

- The method works as follows:
  - Train in the usual way the network using the data in the estimation set

Method of Early Stopping II

Model Selec.

- After a period of estimation, the weights and bias levels of MLP are all fixed and the network is operating in its forward mode only. The validation error is measured for each example present in the validation subset
- When the validation phase is completed, the estimation is resumed for another period (e.g. 10 epochs) and the process is repeated

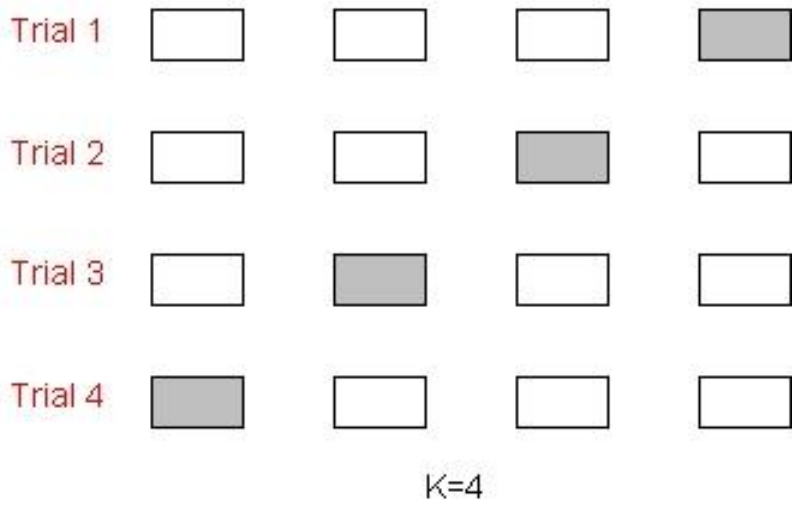


Leave k-out Validation

- We divide the set of available examples into K subsets
- The model is trained in all the subsets except for one and the validation error is measured by testing it on the subset left out
- The procedure is repeated for a total of K trials, each time using a different subset for validation
- The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment
- There is a limiting case for K=N in which case the method is called *leave-one-out*.

Leave k-out Validation II

- An example with K=4 is shown below



Network Pruning

- To solve real world problems we need to reduce the free parameters of the model. We can achieve this objective in one of two ways:
  - *Network growing:* in which case we start with a small MLP and then add a new neuron or layer of hidden neurons only when we are unable to achieve the performance level we want
  - *Network pruning:* in this case we start with a large MLP with an adequate performance for the problem at hand, and then we prune it by weakening or eliminating certain weights in a principled manner

Model Selec.

Network Pruning II

- Pruning can be implemented as a form of **regularisation**

Model Selec.

Regularisation

Model Selec.

- In model selection we need to balance two needs:
  - To achieve good performance, which usually leads to a complex model
  - To keep the complexity of the model manageable due to practical estimation difficulties and the overfitting phenomenon
- A principled approach to the counterbalance both needs is given by *regularisation theory*.
- In this theory we assume that the estimation of the model takes place using the usual cost function and a second term which is called *complexity penalty:*

Regularisation II

$R(\mathbf{w})=E_s(\mathbf{w})+\lambda E_c(\mathbf{w})$

Where R is the total cost function, E_s is the standard *performance measure*, E_c is the complexity penalty and $\lambda>0$ is a regularisation parameter

Model Selec.

- Typically one imposes *smoothness constraints* as a complexity term. I.e. we want to co-minimise the smoothing integral of the kth-order:

$$\mathrm{E}_c(\vec{w},k)=\frac{1}{2}\int \| \frac{\partial^k}{\partial \vec{x}^k} F(\vec{x},\vec{w}) \|^2 \ \mu(\vec{x}) d\vec{x}$$

Where F(**x**,**w**) is the function performed by the model and $\mu(\mathbf{x})$ is some weighting function which determines

Regularisation III

the region of the input space where the function $F(\mathbf{x},\mathbf{w})$ is required to be smooth.

Model Selec.

Regularisation IV

- Other complexity penalty options include:
  - *Weight Decay:*

$$E_c(\vec{w}) = \| \vec{w} \|^2 = \sum_{i=1}^{W} w_i^2$$

Where W is the total number of all free parameters in the model

Model Selec.

  - *Weight Elimination:*

$$E_c(\vec{w}) = \sum_{i=1}^{W} \frac{(w_i / w_0)^2}{1 + (w_i / w_0)^2}$$

Where w_0 is a pre-assigned parameter

Regularisation V

- There are other methods which base their decision on which weights to eliminate on the Hessian, **H**
- For example:
  - The *optimal brain damage* procedure (OBD)
  - The *optimal brain surgeon* procedure (OBS)
    - In this case a weight, w_i, is eliminated when:

      $E_{av} < S_i$

      Where S_i is defined as:

$$S_i = \frac{w_i^2}{2[H^{-1}]_{i,i}}$$

Model Selec.

Conjugate-Gradient Method

- The conjugate-gradient method is a 2nd order optimisation method, i.e. we assume that we can approximate the cost function up to second degree in the Taylor series:

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T A\vec{x} - \vec{b}^T\vec{x} + c$$

BP & Opt.

Where **A** and **b** are appropriate matrix and vector and **x** is a W-by-1 vector

- We can find the minimum point by solving the equations:

$x^* = \mathbf{A^{-1}b}$

Conjugate-Gradient Method II

- Given the matrix **A** we say that a set of nonzero vectors **s**(0), …, **s**(W-1) is **A**-conjugate if the following condition holds:

$$\mathbf{s}^T(n)\mathbf{A}\mathbf{s}(j)=0 , \quad \forall \text{ n and j, } n \neq j$$

- If **A** is the identity matrix, conjugacy is the same as orthogonality.

BP & Opt.

- **A**-conjugate vectors are **linearly independent**

Summary of the Conjugate-Gradient Method

1. *Initialisation:* Unless prior knowledge on the weight vector **w** is available, choose the initial value **w**(0) using a procedure similar to the ones which are used for the BP algorithm
2. *Computation:*
    1. For **w**(0), use the BP to compute the gradient vector **g**(0)
    2. Set **s**(0)=**r**(0)=-**g**(0)
    3. At time step n, use a line search to find $\eta(n)$ that minimises $E_{av}(n)$ sufficiently, representing the cost function E_{av} expressed as a function of η for fixed values of **w** and **s**

Summary of the Conjugate-Gradient Method II

4. Test to determine if the Euclidean norm of the residual **r**(n) has fallen below a specific value, that is, a small fraction of the initial value ||**r**(0)||
5. Update the weight vector:

   w(n+1)=**w**(n)+ η(n) **s**(n)
6. For **w**(n+1), use the BP to compute the updated gradient vector **g**(n+1)
7. Set **r**(n+1)=-**g**(n+1)
8. Use the Polak-Ribiere formula to calculate β(n+1):

$$\beta(n+1) = \max\left\{ \frac{\vec{r}^{\,T}(n+1)[\vec{r}(n+1) - \vec{r}(n)]}{\vec{r}^{\,T}(n)\vec{r}(n)} \quad , \quad 0 \right\}$$

Summary of the Conjugate-Gradient Method III

9. Update the direction vector:

$\mathbf{s}(n+1)=\mathbf{r}(n+1)+\beta(n+1)\mathbf{s}(n)$

10. Set n=n+1 and go to step 3

3. *Stopping Criterion:* Terminate the algorithm when the following condition is satisfied:

$||\mathbf{r}(n)|| \leq \varepsilon ||\mathbf{r}(0)||$

Where ε is a prescribed small number

Advantages & Disadvantages

Conclusions

- MLP and BP is used in Cognitive and Computational Neuroscience modelling but still the algorithm does not have real neuro-physiological support
- The algorithm can be used to make encoding / decoding and compression systems. Useful for data pre-processing operations
- The MLP with the BP algorithm is a universal approximator of functions
- The algorithm is computationally efficient as it has O(W) complexity to the model parameters
- The algorithm has "local" **robustness**
- The convergence of the BP can be very slow, especially in large problems, depending on the method

Advantages & Disadvantages II

- The BP algorithm suffers from the problem of local minima

Conclusions

Types of problems

- The BP algorithm is used in a great variety of problems:
  - Time series predictions
  - Credit risk assessment
  - Pattern recognition
  - Speech processing
  - Cognitive modelling
  - Image processing
  - Control
  - Etc
- BP is the **standard** algorithm against which all other NN algorithms are compared!!

Conclusions