Topic 8

Artificial neural networks:

Supervised learning part 2

Multilayer neural networks

Accelerated learning in multilayer neural networks

Multilayer neural networks

- A multilayer perceptron is a feedforward neural network with one or more hidden layers.
- The network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons.
- The input signals are propagated in a forward direction on a layer-by-layer basis.

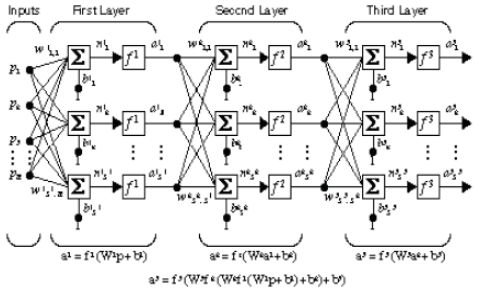
Multilayer neural networks

- Generally much more versatile than single neurons
- No linear separability requirement.
- Training is less obvious and potentially more time consuming.
- Several varieties, the most common of which is known as:

MLP (Multi-Level Perceptron)

Backpropagation Network (alluding to a common method of training these networks; other training methods could conceivably be used.)

Multilayer perceptron with two hidden layers



What does the middle layer hide?

- A hidden layer "hides" its desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network. There is no obvious way to know what the desired output of the hidden layer should be.
- Commercial ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Experimental neural networks may have five or even six layers, including three or four hidden layers, and utilise millions of neurons.

How to train a MLP?

With a single neuron, it is not too hard to see how to

adjust the weights based upon the error values.

With a multi-layer network, it is less obvious. For one thing, what is the "error" for the neurons in nonfinal layers? Without these, we don't know how to adjust.

This is called the "credit assignment" problem (maybe should be "blame assignment").

Backpropagation

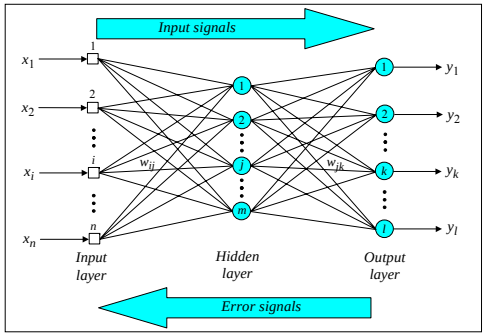
- Werbos, in his Harvard PhD thesis in 1974 found a method.
- Rumelhart and McClelland, in 1985 discovered the method, presumably independently, and popularized it under the current name.
- In mathematics, such methods are in the category of “optimization”.
- The technique is gradient descent, as for Adalines.
- However, the computation of the gradient is less clear.

Back-propagation neural network

- Learning in a multilayer network proceeds the same way as for a perceptron.
- A training set of input patterns is presented to the network.
- The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

- In a back-propagation neural network, the learning algorithm has two phases.
- First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

Three-layer back-propagation neural network



Backpropagation training cycle

- Forward propagation: Derive the activation values (the inputs to the activation functions) at each neuron, and the final output.
- Compute the error in the output.
- Backpropagate the error through the network to get “sensitivities” at each neuron. (The gradient approximation is derivable from the sensitivities.)
- Use the sensitivities to derive weight changes.
- Apply the weight changes.
- Backpropagate is mathematically a lot like forward propagate.
- Sensitivities are used instead of signal values.
- The sensitivities are the partial derivatives of the MSE with respect to the activation values.
- Basically both are iterated matrix multiplications.

Backpropagation

Given an input vector, can compute the outputs.

Given a sample, can compute the errors in output.

Knowing gradient, can adjust the weights.

- Big Question: How to compute the gradient?

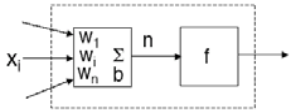
Recall that the gradient consists of components $\Delta J / \Delta w$

where J is the mean-squared error and w is some weight (or bias) in the network.

For the Adaline, already derived:

$\Delta J / \Delta w_i = -2 \varepsilon x_i f'(n)$, where xi is the input corresponding to weight wi, and n(net) is the weighted sum. This works as is for the multi-layer case at the output layer.

Inside one neuron



$\partial J/\partial w_i = (\partial J/\partial n)\ (\partial n/\partial w_i)$ chain rule

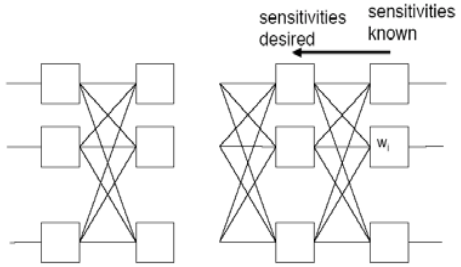
$= (\partial (d\text{-}f(n))^2/\partial n)\ (\partial n/\partial w_i)$

$= -2\,\varepsilon\, f'(n)\ x_i$

$= s\, x_i$

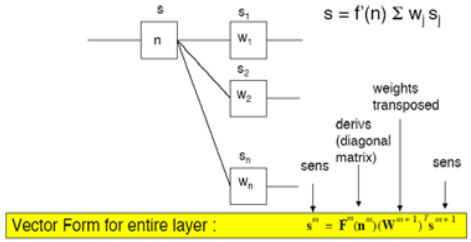
where $s = (\partial J/\partial n)$ is called the *sensitivity*

Backward propagation of sensitivity



Backward propagation of sensitivity

Express desired as a weighted sum of known:



The back-propagation training algorithm

Step 1: Initialisation

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range:

$$\left(-\frac{2.4}{F_i},\ +\frac{2.4}{F_i}\right)$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: Activation

Activate the back-propagation neural network by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired outputs $y_{d,1}(p), y_{d,2}(p), \ldots, y_{d,n}(p)$.

(*a*) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p)\cdot w_{ij}(p) - \theta_j\right]$$

where n is the number of inputs of neuron j in the hidden layer, and *sigmoid* is the *sigmoid* activation function.

Step 2: Activation (continued)

(*b*) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p)\cdot w_{jk}(p) - \theta_k\right]$$

where m is the number of inputs of neuron k in the output layer.

Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(*a*) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p)\cdot[1 - y_k(p)]\cdot e_k(p)$$

where $e_k(p) = y_{d,k}(p) - y_k(p)$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \cdot y_j(p)\cdot \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

Step 3: Weight training (continued)

(*b*) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p)\cdot[1 - y_j(p)]\cdot\sum_{k=1}^{l}\delta_k(p)\, w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \cdot x_i(p)\cdot \delta_j(p)$$

Update the weights at the hidden neurons:

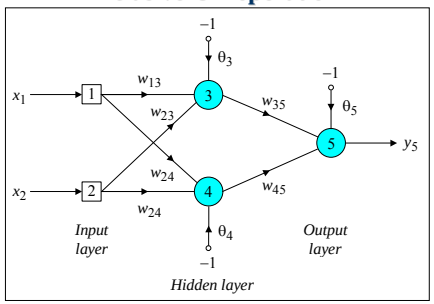
$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Step 4: Iteration

Increase iteration p by one, go back to *Step 2* and repeat the process until the selected error criterion is satisfied.

As an example, we may consider the three-layer back-propagation network. Suppose that the network is required to perform logical operation *Exclusive-OR*. Recall that a single-layer perceptron could not do this operation. Now we will apply the three-layer net.

Three-layer network for solving the Exclusive-OR operation



- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to −1.
- The initial weights and threshold levels are set randomly as follows:
  $w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$, $\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

- We consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/\left[1 + e^{-(1\cdot 0.5 + 1\cdot 0.4 - 1\cdot 0.8)}\right] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/\left[1 + e^{-(1\cdot 0.9 + 1\cdot 1.0 + 1\cdot 0.1)}\right] = 0.8808$$

- Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = sigmoid(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/\left[1 + e^{-(-0.5250\,1.2 + 0.8808\,1.1 - 1\cdot 0.3)}\right] = 0.5097$$

- Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

- The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, *e*, from the output layer backward to the input layer.
- First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5\,(1-y_5)\,e = 0.5097\cdot(1-0.5097)\cdot(-0.5097) = -0.1274$$

- Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha\cdot y_3\cdot\delta_5 = 0.1\cdot 0.5250\cdot(-0.1274) = -0.0067$$
$$\Delta w_{45} = \alpha\cdot y_4\cdot\delta_5 = 0.1\cdot 0.8808\cdot(-0.1274) = -0.0112$$
$$\Delta\theta_5 = \alpha\cdot(-1)\cdot\delta_5 = 0.1\cdot(-1)\cdot(-0.1274) = -0.0127$$

- Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1-y_3)\cdot\delta_5\cdot w_{35} = 0.5250\cdot(1-0.5250)\cdot(-0.1274)\cdot(-1.2) = 0.0381$$
$$\delta_4 = y_4(1-y_4)\cdot\delta_5\cdot w_{45} = 0.8808\cdot(1-0.8808)\cdot(-0.1274)\cdot 1.1 = -0.0147$$

- We then determine the weight corrections:

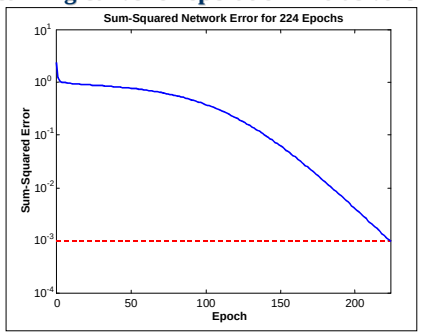
$$\Delta w_{13} = \alpha\cdot x_1\cdot\delta_3 = 0.1\cdot 1\cdot 0.0381 = 0.0038$$
$$\Delta w_{23} = \alpha\cdot x_2\cdot\delta_3 = 0.1\cdot 1\cdot 0.0381 = 0.0038$$
$$\Delta\theta_3 = \alpha\cdot(-1)\cdot\delta_3 = 0.1\cdot(-1)\cdot 0.0381 = -0.0038$$
$$\Delta w_{14} = \alpha\cdot x_1\cdot\delta_4 = 0.1\cdot 1\cdot(-0.0147) = -0.0015$$
$$\Delta w_{24} = \alpha\cdot x_2\cdot\delta_4 = 0.1\cdot 1\cdot(-0.0147) = -0.0015$$
$$\Delta\theta_4 = \alpha\cdot(-1)\cdot\delta_4 = 0.1\cdot(-1)\cdot(-0.0147) = 0.0015$$

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$
$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$
$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$
$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$
$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$
$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$
$$\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$$
$$\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$$
$$\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$$

- The training process is repeated until the sum of squared errors is less than 0.001.

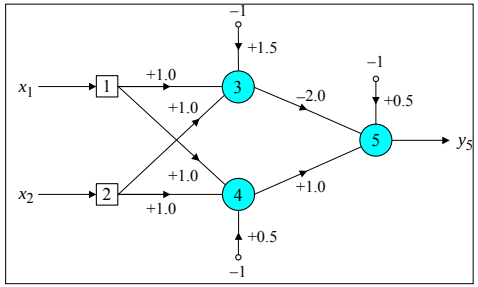
Learning curve for operation *Exclusive-OR*



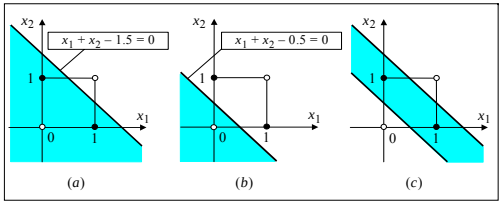
Final results of three-layer network learning

Inputs x_1	Inputs x_2	Desired output y_d	Actual output y_5	Error e	Sum of squared errors
1	1	0	0.0155	–0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	–0.0175	

Network represented by McCulloch-Pitts model for solving the *Exclusive-OR* operation



Decision boundaries



(*a*) Decision boundary constructed by hidden neuron 3;
(*b*) Decision boundary constructed by hidden neuron 4;
(*c*) Decision boundaries constructed by the complete three-layer network

Accelerated learning in multilayer neural networks

- A multilayer network learns much faster when the sigmoidal activation function is represented by a **hyperbolic tangent**:

$$Y^{tanh} = \frac{2a}{1+e^{-bX}} - a$$

where a and b are constants.

Suitable values for a and b are:
$a = 1.716$ and $b = 0.667$

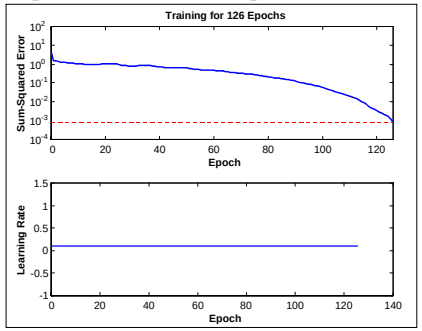
- We also can accelerate training by including a **momentum term** in the delta rule:

$$\Delta w_{jk}(p) = \beta \cdot \Delta w_{jk}(p-1) + \alpha \cdot y_j(p) \cdot \delta_k(p)$$

where β is a positive number ($0 \le \beta < 1$) called the **momentum constant**. Typically, the momentum constant is set to 0.95.

This equation is called the **generalised delta rule**.

Learning with momentum for operation *Exclusive-OR*



Learning with adaptive learning rate

To accelerate the convergence and yet avoid the danger of instability, we can apply two heuristics:

Heuristic 1

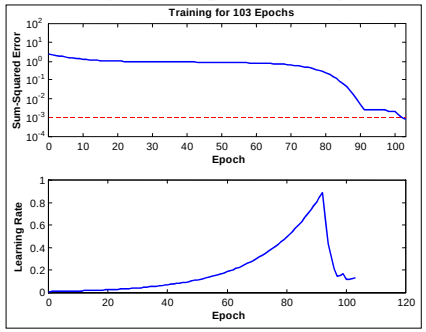
If the change of the sum of squared errors has the same algebraic sign for several consequent epochs, then the learning rate parameter, α, should be increased.

Heuristic 2

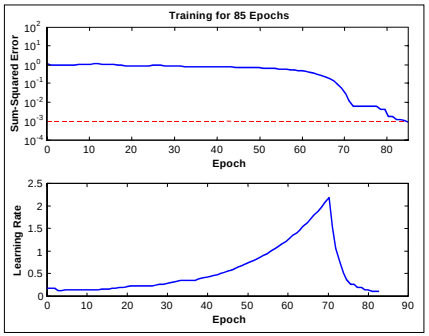
If the algebraic sign of the change of the sum of squared errors alternates for several consequent epochs, then the learning rate parameter, α, should be decreased.

- Adapting the learning rate requires some changes in the back-propagation algorithm.
- If the sum of squared errors at the current epoch exceeds the previous value by more than a predefined ratio (typically 1.04), the learning rate parameter is decreased (typically by multiplying by 0.7) and new weights and thresholds are calculated.
- If the error is less than the previous one, the learning rate is increased (typically by multiplying by 1.05).

Learning with adaptive learning rate



Learning with momentum and adaptive learning rate



BackProp Technique & Tricks (Some of these apply to General Neural Networks)

(Two References: Neural Networks Tricks of the Trade, Orr and Muller, eds.

http://www.dontveter.com/bpr/bpr.html)

- • Choose examples with maximum information content
  - – Shuffle the training set so that successive samples rarely belong to the same class.
  - – Present input examples that produce a large error more frequently than ones that produce a small error.

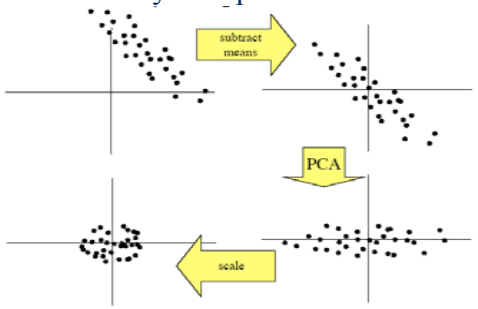
Technique and tricks

- • Normalize the inputs
  - – Better if mean of a particular variable is near 0.
    - • Then weight changes are less likely to be synchronized, since some will be positive, others negative.
    - • Therefore, **subtract the actual mean** from the variable before training.
  - – Better if the variables are scaled to have similar auto-covariances, defined as (sum-of-squares of variable)/(number of samples)
    - • Then the **weights will learn at similar rates**.
    - • Exception: When some variables are known in advance to be of less significance.

Technique and tricks

- • Decorrelate the inputs
  - – Better if no two input variables are correlated.
  - – Correlated inputs analogous to having linearly dependent variables in a linear system.
  - – A technique called PCA (Principal Components Analysis), aka Karhunen-Loeve Expansion, can be used to remove linear correlations.
  - – We will look at PCA later; PCA itself can be done by a PCA neural network.

Summary of input normalization



Technique and tricks

- • Prefer tansig (hyperbolic tangent) rather than logsig for inner layers.
  - – tansig output is symmetric about origin, logsig is not.
  - – tansig will more likely produce outputs close to 0 for the **next stage** of the network
- • Some recommend adding a small linear constant to the output of tansig to "avoid flat spots"
- Piecewise quadratic approximation to tanh

Choice of target values

- • Choosing target values of +1, -1 for a tansig causes the neuron to be driven toward the **saturation region**.
- • To get into this region, the weights are large and may become "stuck" because small gradient values will not change them sufficiently.
- • It may be better to **choose the targets offset** from these saturation values, or to scale the tansig to get the same effect, e.g. $f(x) = 1.7159 \tanh(2x/3)$, which has a maximum 2nd derivative where the function's value is +/- 1.

Weight initialization

- • Assuming that the training set has been normalized and the previous sigmoid is used,
- • Draw the initial weights from a distribution, such as a uniform distribution, with mean 0 and standard deviation 1/sqrt(m) where m is the **fan-in** (number of inputs to the node).
- • Increases likelihood that the input to the sigmoid will have a standard deviation of 1 (since the latter is the sqrt of the sum of the squares of the weights, for normalized input).

Learning rates

- • Ideally, each **weight** should have its own learning rate. See the Neural Networks Tricks of the Trade, Orr and Muller, eds., for how to choose learning rate based on 2nd derivatives.
- • As a substitute, each neuron, or each layer could have its own learning rate.
- • Learning rates should be proportional to the sqrt of the number of inputs to the neuron.
- • Weights in **earlier layers should be larger** than those in later layers, since the earlier layers tend to have a smaller 2nd derivative of the MSE.

Validation Technique ("Cross-Validation") & Early Stopping

- • Split the training set into **training** and **validation** subsets, e.g. 2:1 or 5:1 ratio.
- • Train only on the training subset; use the validation set for MSE, every so often (e.g. every 5 epochs).
- • **For early stopping:** Stop training **as soon as the**
- **validation error goes up**.
- • Use the weights **before** the error went up.
- • Rational: Even though a lower minimum might have been reached, the local minima tend to be fairly close in value in practice.

Over-fitting

- • It is possible for a network to *over-fit* the data, meaning that it learns small variations in the data which might actually be due to noise.
- • Another way of saying this is that the network does not generalize well; it is **too specialized**.
- • **Validation** is one technique used to help avoid over-fitting.
- • Over-fitting can result if the network has **too many neurons** at its disposal.

Sizing a network

- • Given a problem:
  - – How many layers?
  - – How many neurons per layer?
  - – What activation functions?
- • Theoretically, any function can be emulated over a given range by a network with just one hidden layer and one output layer (two layers total), with sufficient neurons in that layer.
- • Practically, 2-3 layers suffice for large families of problems, although more may be used, especially when special feature-selection layers are used, as in the zip-code recognition network.

Neurons

- • Choose number of neurons based on the assessed complexity within a layer (number of crests and valleys of a function, for example).
- • Two approaches for experimental determination:

– Start with a large number of neurons and prune.

– Start with a small number of neurons and build up.

- • Negligible weights can be eliminated (set to 0).
- • If all input weights to a node are 0, the node can be eliminated.
- • If all weights a node feeds are 0, the node itself can be eliminated.
- • Vary weights w to see whether $\Delta J / \Delta w$ is significant; if not, prune the weight.

Doubling

- • Start with a small number of neurons in the inner layer.
- • If at the conclusion of a training cycle, the MSE is inadequate, repeat with double the number of neurons.

Number of training samples

• Baum-Hausler rule (1989):

Necessary condition:

(number of samples) > W / (1-a)

where W is the number of weights in the network and a is the desired accuracy on the test set.

• **Sufficient condition**:

(number of samples) > log(N / (1-a)) * W/(1-a)

where N is the number of neurons.