Motivation/Rationale

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. We've already seen a couple of ways. With a multi-layer network, it is less obvious. For one thing, **what is the "error" for the neurons in non-final layers**? Without these, we don't know how to adjust. This is called the "credit assignment" problem.

Syllabus

Unit 4 – Learning Mechanisms **[__ hrs.]**

- 4.1. Supervised learning methods
  - 4.1.1. Back-propagation
  - 4.1.2. Conjugate Gradient method
  - 4.1.3. Levenberg-Marquardt (LM) method
  - 4.1.4. Madaline
  - 4.1.5. Radial-Basis Networks
  - 4.1.6. Cascade-Correlation Networks
  - 4.1.7. Polynomical Networks
  - 4.1.8. Recurrent Networks
    Time Series, back-propagation through time, finite impulse response (FIR) MLP, temporal differences method (TD)
- 4.2. Unsupervised learning methods
  - 4.2.1 Kohonen Self-Organizing Maps (SOMs)

LEARNING MECHANISMS

Learning algorithms can be divided into **supervised** and **unsupervised** methods. **Supervised learning** denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured. The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm. This kind of learning is also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors.

Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown. Assume, for example, that some points in two-dimensional space are to be classified into three clusters. For this task we can use a classifier network with three output lines, one for each class (see following figure). Each of the three computing units at the output must specialize by firing only for inputs corresponding to elements of each cluster. If one unit fires, the others must keep silent. In this case we do not know a priori which unit is going to specialize on which cluster. Generally we do not even know how many well-defined clusters are present. Since no "teacher" is available, the network must organize itself in order to be able to associate clusters with units.

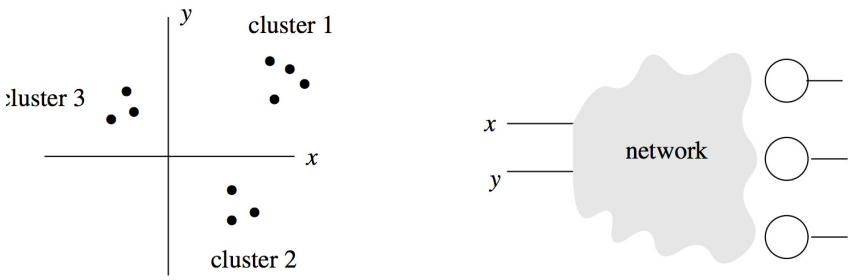


Fig. 1: Clusters and classifiers

Supervised learning is further divided into methods that use **reinforcement** or **error correction**. Reinforcement learning is used when after each presentation of an input-output

example we only know whether the network produces the desired result or not. The weights are updated based on this information (that is, the Boolean values true or false) so that only the input vector can be used for weight correction. In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights, and in many cases we try to eliminate the error in a single correction step.

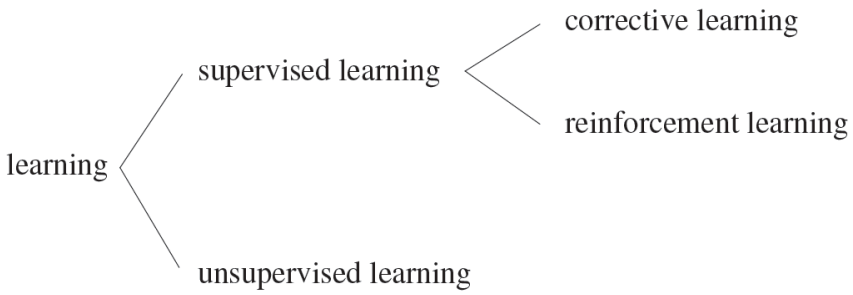


Fig. 2: Classes of learning algorithms

Back-propagation

The back-propagation algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. For this purpose, ***sigmoid*** function is used for activation.

$$s(x) = \frac{1}{1+e^{-x}}$$

and the derivative of sigmoid function;

$$\frac{ds(x)}{dx} = s(x).(1-s(x))$$

An alternative to sigmoid is symmetrical sigmoid S(x) defined as;

$$S(x) = 2s(x) - 1 = \frac{1-e^{-x}}{1+e^{-x}}$$, also known as hyperbolic tangent.

The range of output of sigmoid is (0, 1).

To sharpen the separation in input space;

$$s_c(x) = \frac{1}{1+e^{-cx}}$$, increase the value of *c*.

Since, the error is minimized iteratively, actually the training algorithm is performing optimization operation for the objective function of minimizing the MSE.

One step error function can be represented as;

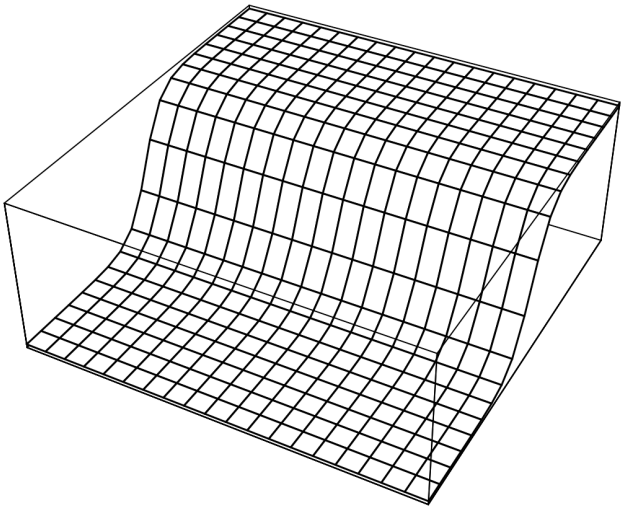


Fig. 3: One step error

A price has to be paid for all the positive features of the sigmoid as activation function. The most important problem is that, under some circumstances, local minima appear in the error function which would not be there if the step function had been used. Following figure shows an example of a local minimum with a higher error level than in other regions. The function was computed for a single unit with two weights, constant threshold, and four input-output patterns in the training set. There is a valley in the error function and if gradient descent is started there the algorithm will not converge to the global minimum.

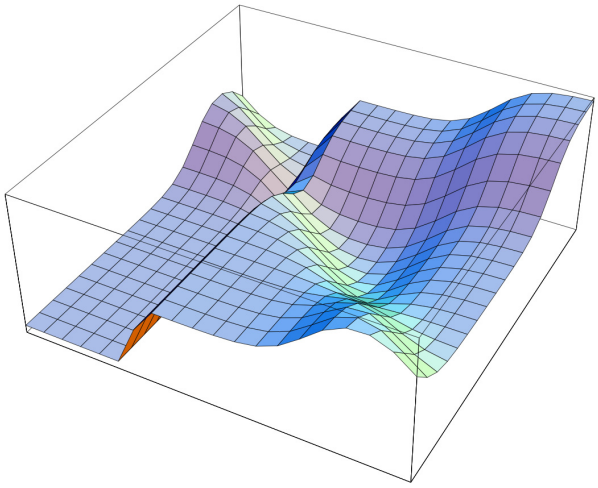


Fig. 4: Local minimum

Backpropagation Training Cycle

- **Forward propagation (feed forward)**: 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 and applications of the activation functions of the neurons.

Back-propagation-Steps

1. Feed forward

Let's consider, ***x*** is input to layer ***l*** and contains ***n*** no. of nodes.

l ranges from 0, ..., L

x_i^l is the output of layer 'l' with no. of nodes 'i'.

The functional output x_i^l is represented as;

$$x_i^l = f(u_i^l) = f\left(\sum_{j=1}^{n_{l-1}} w_{ji}^l . x_j^{l-1} + b_i^l\right), \quad \text{for all } l = 1 \text{ to L.}$$

Or, in simple terms;

If we have three layered network,

Let's consider,

- *X* is input vector,
- *H* is functional output of hidden layer,
- *HA* is activation output of hidden layer,
- *O* is functional output of output layer,
- *OA* is activation output of output layer
- W_{ih} is weights between input and hidden layer
- W_{ho} is weights between hidden and output layer

Then,

- $H = X \ . \ W_{ih}$
- $HA = logsig(H)$
- $O = HA \ . \ W_{ho}^T$
- $OA = logsig(O)$

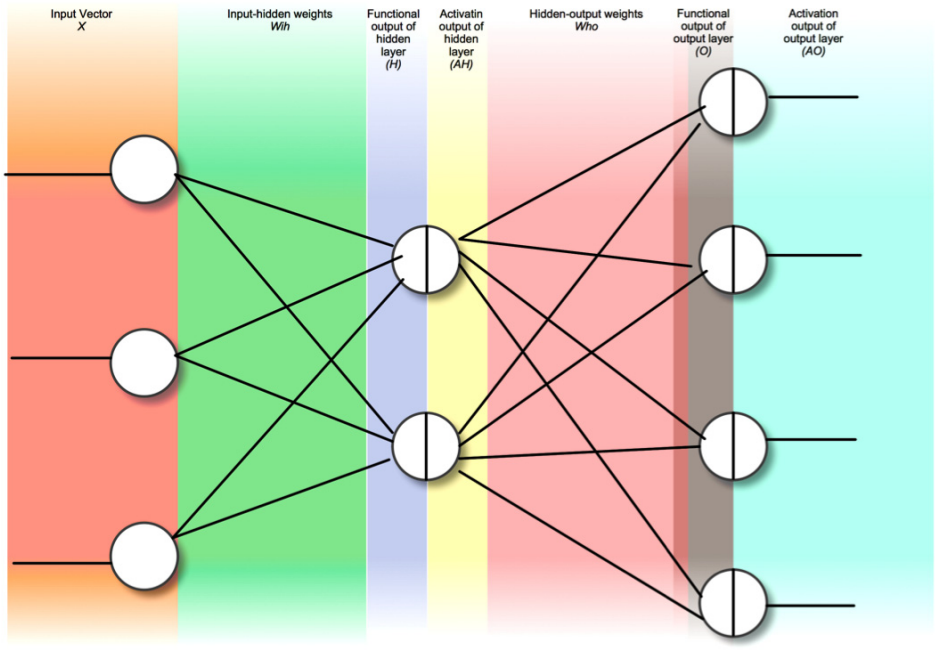


Fig. 5: Model of MLP

$x = [2 \quad 1 \quad 0.7]$

$$w_{ih} = \begin{bmatrix} 0.7 & 0.1 \\ 0.3 & 0.5 \\ 0.7 & 0.8 \end{bmatrix}$$

$h = x \,.\, w_{ih}$

$$hA = \frac{1}{1+e^{-h}}$$

$$w_{ho}{}^{T} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.7 \\ 0.5 & 0.6 & 0.9 & 0.4 \end{bmatrix}$$

$o = hA \,.\, w_{ho}{}^{T}$

$$oA = \frac{1}{1+e^{-o}}$$

2. Error computation
The difference between the desired output ***d*** and actual output ***xL*** is computed.

$\delta_i^L = f'(u_i^L).(d_i - x_i^L)$, note the derivative of activation function.

3. Backpropagation
The error signal at the output unit is propagated backwards through the entire network.

$\delta_j^{l-1} = f'(u_j^{l-1}).\sum_{i=1}^{n_l} \delta_i^l . W_{ij}^{\ l}$, from ***l*** = L to 1.

4. Learning updates
The synaptic weights and biases are updated using the results of the forward and backward passes;

$$\Delta W_{ij}^{\ l} = \eta . \delta_i^l . x_j^{l-1}$$

$$\Delta b_i^l = \eta . \delta_i^l$$, from ***l***=1 to L.

Gradient
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, for generalized Adaline;

$\frac{\partial J}{\partial W_i} = -2.\varepsilon.X_i.f'(n)$, where x_i is the input corresponding to weight w_i, and n (**net**) is the weighted sum. This works **as is** for the multi-layer case at the **output** layer.

PRACTICAL #7

MLP FeedForward Implementation

Objectives

To implement MLP for feedforward operation.

Description

__

The C program

In Matlab

```
# Feedforward demo for MLP

# input vector
x = [2 1 0.7]

# weight vector for input-to-hidden layer
wih [0.7 0.1; 0.3 0.5; 0.7 0.8]

# functional output of hidden layer
h = x * wih

# activation of hidden layer
hA = logsig(h)

# weight vector for hidden-to-output layer (Transposed)
whoT = [0.1 0.2 0.3 0.7; 0.5 0.6 0.9 0.4]

# functional output of output layer
o = hA * whoT

# activation of output layer
oA = logsig(o)
```

To do: Calculate back-propagation values using the Gradient method described.

Derivation of Gradient and Sensitivity in Back-propagation algorithm (including proof)

Back-propagation concerns:

- Gradient
- Chain rule
- Delta rule
- Sensitivity

The error vector at layer 'n' can be defined as;

$$e_j(n) = d_j(n) - y_j(n) \qquad (0)$$

From perceptrons, MSE is equal to;

$$MSE = J(w) = \frac{\sum (d - actual)^2}{N}$$

$$actual = \sum w_j . x_j$$

$$J(w) = \frac{\sum \left(d - \sum w_j . x_j\right)^2}{N}$$

for a single sample, gradient is;

$$J \approx (d - \sum w_j x_j)^2$$

now, the i[th] gradient component will be = $\frac{\partial J}{\partial w_i}$

$$= \frac{\partial (d - \sum w_j . x_j)^2}{\partial w_i}$$

$$= 2(d - \sum w_j . x_j) . \frac{\partial (d - \sum w_j . x_j)}{\partial w_i}$$

$$= 2\varepsilon . - x_j$$

In other words, with activation function '**f**', the i[th] gradient will be;

$$= \frac{\partial J}{\partial w_i}$$

$$= \frac{\partial \left[d - f\left(\sum w_j x_j\right)\right]^2}{\partial w_i}$$

$$= 2\left[d - f\left(\sum w_j x_j\right)\right] \frac{\partial \left[d - f\left(\sum w_j x_j\right)\right]}{\partial w_i}$$

$$= -2 \ \varepsilon \ f'\left(\sum w_j x_j\right) \frac{\partial f\left(\sum w_j x_j\right)}{\partial w_i}$$

The value of **f'** depends on activation function and its derivative.

$$Therefore \quad \Delta w = -2 \ \varepsilon \ f'\left(\sum w_j x_j\right) x_j$$

Now for gradient in back-propagation, we can use various methods;

- Chain rule
- Lagrange multiplier

The gradient of the network can be modeled as; $\frac{\partial J}{\partial w_i} = -2 \ \varepsilon \ x_i \ f'(n)$.

Let's start the derivation of the learning algorithm using gradient.

The error of the j[th] node in layer 'n' is;

$$e_j(n) = d_j(n) - y_j(n) \qquad -(0)$$

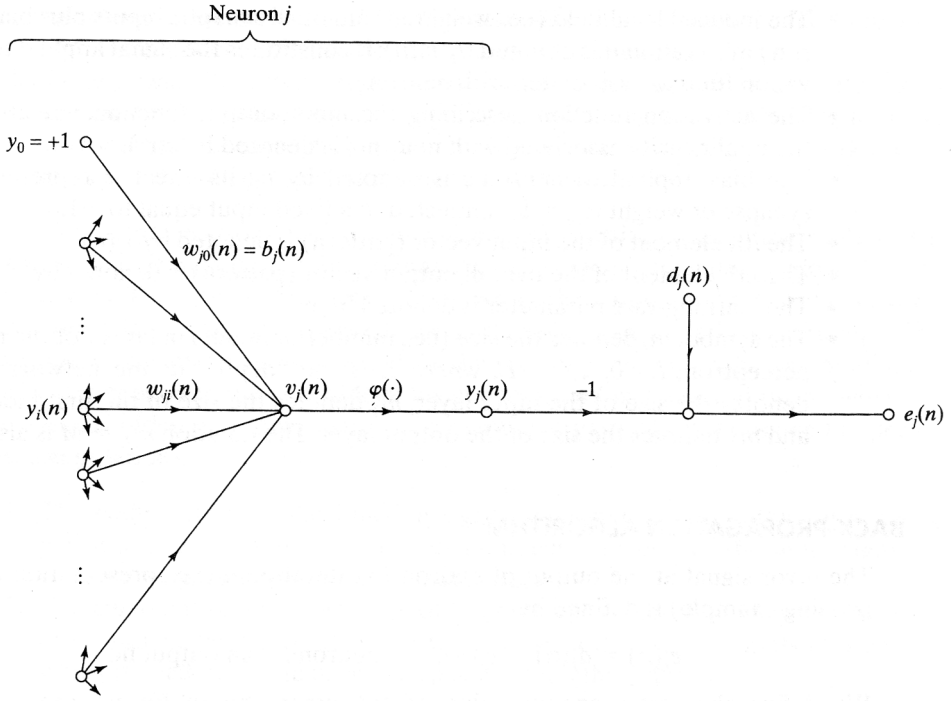
error energy for neuron 'j' = $\frac{1}{2} e_j^{\,2}(n) \qquad -(1)$

overall error of the network $\varepsilon(n) = \frac{1}{2}\sum e_j^{\,2}(n) \qquad -(2)$

average error $\varepsilon_{av} = \frac{1}{N}\sum_{n=1}^{N} \varepsilon(n) \qquad -(3)$

$\varepsilon(n)$ is instantaneous error and $\varepsilon_{av} \ \ is \ \ f(w_{ij}, b_j)$.

Now, let's consider following network.



If $\mathbf{v_j(n)}$ is functional output of a '**j**' neuron at **n[th]** layer.

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \qquad including \;\; bias \;\; -(4)$$

$$y_j(n) = \phi_j(v_j(n))$$

Now we consider that the change in the weight is equivalent to the gradient.

i.e. $\Delta w_{ji}(n) \approx \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)}$, which need **proof!!**

Let's use **Chain rule** to prove it.

The chain rule;

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = \frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \qquad -(6)$$

which illustrates that, the total error depends on error of individual neurons, error of individual neurons depends on activation output of the neuron, activation output of the neuron depends on functional output of the neuron, and functional output of the neuron depends on weights.

Differentiating both sides of equation (2), w.r.t. $e_j(n)$;

$$\frac{\partial \varepsilon(n)}{\partial e_j(n)} = e_j(n) \qquad -(7)$$

differentiating both sides of equation (0), w.r.t. $y_j(n)$;

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \qquad -(8)$$

differentiating equation (5) w.r.t. $v_j(n)$;

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi'_j(v_j(n)) \qquad -(9)$$

differentiating equation (4) w.r.t. $w_{ji}(n)$;

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_j(n) \qquad -(10)$$

Putting equations 6 – 10 together,

$$\frac{\delta \varepsilon(n)}{\delta w_{ji}(n)} = -e_j(n) \;\; \phi'_j(v_j(n)) \;\; y_i(n) \qquad -(11)$$

Now, the changes to the weights, i.e. $\Delta w_{ji}(n)$ to $w_{ji}(n)$ is defined by **delta-rule**.

$$\Delta wji(n) = -\eta \;\; \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} \qquad -(12)$$ -ve sign for gradient descent.

Now from equation (11) and (12),

$$\Delta w_{ji}(n) = \eta \;\; \delta_j(n) \;\; y_i(n) \qquad -(13)$$

and the local gradient will be;

$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial v_j(n)}$$

$$= \frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{dy_j(n)}{dv_j(n)}$$

$$= e_j(n) \; \phi'_j(v_j(n))$$

Sensitivity
Normally,

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial n} \frac{\partial n}{\partial w_j}$$

$$= -2 \; \varepsilon \; f'(n) \; x_j$$

$$= S \; x_j \qquad \text{where } S \text{ is sensitivity}$$

if, ε=0, no sensitivity i.e. no change is required in the weights if the error is 0.

The sensitivity is calculated at the final layer and back-propagated to the front layers.

Back-propagation of sensitivity;

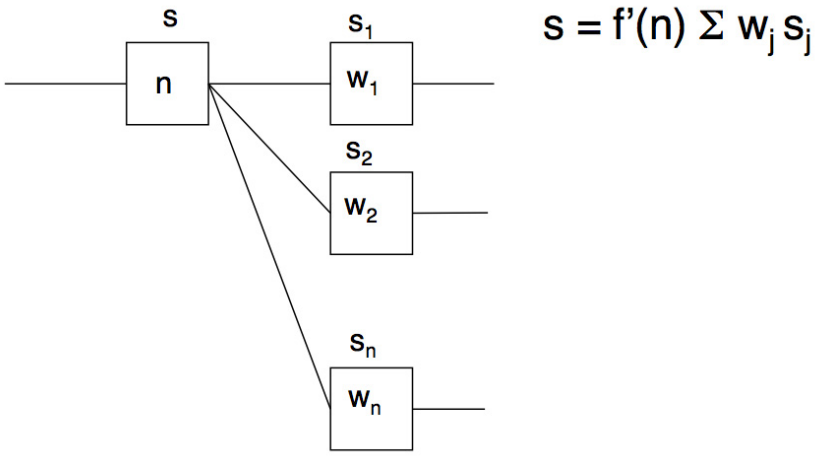


Fig. 5-2: Back-propagation of sensitivity

In vector form, sensitivity $S_j^m = \frac{\partial J}{\partial n_i^m}$

Gradient in the form of sensitivity $\frac{\partial J}{\partial w_{ij}^m} = S_j^m \; a_j^{m-1}$

Improvements over back-propagation learning mechanisms

Gradient Descent BP with Momentum (GDM)
Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum.

Momentum can be added to BP method learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the gradient descent BP rule. The magnitude of the effect that the last weight is allowed to have is mediated by a momentum constant, μ, which can be any number between 0 and 1.

When the momentum constant is 0 a weight change is based solely on the gradient. When the momentum constant is 1 the new weight change is set to equal the last weight change and the gradient is simply ignored. The new weight vector w_{k+1} is adjusted as;

$$w_{k+1} = w_k - \alpha . g_k + \mu . w_{k-1}$$

where; α is learning rage
g_k is gradient of error
μ is momentum constant (0 – 1)
W_{k-1} is previous weight

Variable learning rate BP with Momentum (GDX)
The learning rate parameter is used to determine how fast the BP method converges to the minimum solution. The larger the learning rate, the bigger the step and the faster the convergence. However, if the learning rate is made too large, the algorithm becomes unstable; saw-tooth effect/rattling. On the other hand, if the learning rate is set to too small, the algorithm will take a long time to converge. To speed up the convergence time, the variable learning rate gradient descent BP utilizes larger learning rate α when the network is far from the solution and smaller learning rate α when the neural net is near the solution. The new weight vector w_{k+1} is adjusted the same as in the gradient descent with momentum above but with a varying α_k. Typically, the new weight vector w_{k+1} is defined as;

$$w_{k+1} = w_k - \alpha_{k+1} . g_k + \mu . w_{k-1}$$
$$\alpha_{k+1} = \beta . \alpha_k$$
$$\beta = \begin{cases} 0.7 & if \quad new \quad error \quad > \quad 1.04 \\ 1.05 & if \quad new \quad error \quad < \quad 1.04 \end{cases}$$

What are conjugate gradients? (This portion is adapted from -
http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-5.html
Master link: http://www.faqs.org/faqs/ai-faq/neural-nets/)
There is no single best method for nonlinear optimization. You need to choose a method based on the characteristics of the problem to be solved. For objective functions with continuous second derivatives (which would include feedforward nets with the most popular differentiable activation functions and error functions), three general types of algorithms have been found to be effective for most practical purposes:

- For a small number of weights, stabilized Newton and Gauss-Newton algorithms, including various Levenberg-Marquardt and trust-region algorithms, are efficient. The memory required by these algorithms is proportional to the square of the number of weights.

- For a moderate number of weights, various quasi-Newton algorithms are efficient. The memory required by these algorithms is proportional to the square of the number of weights.
- For a large number of weights, various conjugate-gradient algorithms are efficient. The memory required by these algorithms is proportional to the number of weights.

Additional variations on the above methods, such as limited-memory quasi-Newton and double dogleg, can be found in textbooks such as Bertsekas (1995). Objective functions that are not continuously differentiable are more difficult to optimize. For continuous objective functions that lack derivatives on certain manifolds, such as ramp activation functions (which lack derivatives at the top and bottom of the ramp) and the least-absolute-value error function (which lacks derivatives for cases with zero error), subgradient methods can be used. For objective functions with discontinuities, such as threshold activation functions and the misclassification-count error function, Nelder-Mead simplex algorithm and various secant methods can be used. However, these methods may be very slow for large networks, and it is better to use continuously differentiable objective functions when possible.

All of the above methods find local optima--they are not guaranteed to find a global optimum. In practice, Levenberg-Marquardt often finds better optima for a variety of problems than do the other usual methods. I know of no theoretical explanation for this empirical finding.

For global optimization, there are also a variety of approaches. You can simply run any of the local optimization methods from numerous random starting points. Or you can try more complicated methods designed for global optimization such as simulated annealing or genetic algorithms (see Reeves 1993 and "What about Genetic Algorithms and Evolutionary Computation?"). Global optimization for neural nets is especially difficult because the number of distinct local optima can be astronomical.

Conjugate Gradient Method (CGP)

The basic BP algorithm adjusts the weights in the steepest descent direction. This is the direction in which the performance function is decreasing most rapidly. Although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In the conjugate gradient algorithms the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size which will minimize the performance function along that line.

The search direction at each iteration is determined by updating the weight vector as;

$$w_{k+1} = w_k + \alpha . p_k$$

$$where \;\; : \;\; p_k = -g_k + \beta_k . p_{k-1}$$

$$\beta_k = \frac{\Delta g_{k-1}^T . g_k}{g_{k-1}^T . g_{k-1}}$$

$$and \qquad \Delta g_{k-1}^T = g_k^T - g_{k-1}^T$$

Non-linear conjugate gradient training algorithm;

1. Choose W(0) using random number generator function
2. For W(0), calculate g(0)
3. Set S(0) = r(0) = -g(0)

4. At time step 'n', use a line search to find η(n) that minimizes $\varepsilon_{av}(n)$ sufficiently. $\varepsilon_{av}(n)$ is expressed as a function of η for fixed values of W and S.
5. When r(n) is sufficiently small, stop.
   a. W(n+1) = W(n) + η(n) . S(n)
   b. Set r(n+1) = -g(n+1)
   c. For $\beta(n+1) = \max\left\{\frac{r^T_{(n+1)}.(r_{(n+1)} - r_{(n)})}{r^T_{(n)}.r_{(n)}}, 0\right\}$
   d. Update the direction vector: $S_{(n+1)} = r_{(n+1)} + \beta_{(n+1)}.S_{(n)}$

Quasi-Newton BP (BFGS)

An alternative to the conjugate gradient methods for fast optimization. Newton's Method for minimizing quadratic approximation i.e. MSE. Uses quadratic approximation of the cost function $\varepsilon_{(w)}$ around the current point W(n). Minimize the error using second order Taylor series expansion of cost function $\varepsilon_{(w)}$.

$$\Delta\varepsilon_{(w(n))} = \varepsilon_{(w(n+1))} - \varepsilon_{(w(n))}$$

$$\approx g^T_{(n)}.\Delta w_{(n)} + \frac{1}{2}\Delta w^T_{(n)}.H_{(n)}.\Delta w_{(n)} \qquad -(i)$$

g(n) is m-by-1 gradient vector of cost function $\varepsilon_{(w)}$ at point W(n).

H(n) is m-by-m Hessian matrix of $\varepsilon_{(w)}$.

$$H = \nabla^2\varepsilon(w)$$

$$= \begin{bmatrix} \frac{\partial^2\varepsilon}{\partial w_1^2} & \frac{\partial^2\varepsilon}{\partial w_1\partial w_2} & \frac{\partial^2\varepsilon}{\partial w_1\partial w_m} \\ \frac{\partial^2\varepsilon}{\partial w_2\partial w_1} & \cdots & \frac{\partial^2\varepsilon}{\partial w_2\partial w_m} \\ \frac{\partial^2\varepsilon}{\partial w_m\partial w_1} & \cdots & \frac{\partial^2\varepsilon}{\partial w_m^2} \end{bmatrix} \qquad -(ii)$$

Differentiating equation(1) w.r.t. Δw,

$$g(n) + H(n)\Delta w(n) = 0$$

$$\Delta w(n) = -H^{-1}(n).g(n)$$

$$w(n+1) = w(n) - H^{-1}(n).g(n)$$

$$H = \frac{\partial^2\varepsilon_{av}(w)}{\partial w^2}$$

For calculation of H and H^{-1}, refer to Haykin P. 224-226 and Numerical Recipes in C, Chapter 10.7 : Variable metric methods in multi-dimensions.

PRACTICAL #11

Implementation of various faster convergence methods in Matlab.

Objectives

To implement

- Gradient descent with adaptive learning rate
- Gradient descent with momentum
- Conjugate gradient algorithms
- Quasi-Newton

Description

traingda

$$w_{k+1} = w_k - \alpha_{k+1}.g_k + \mu.w_{k-1}$$

$$\alpha_{k+1} = \beta.\alpha_k$$

$$\beta = \begin{cases} 0.7 & if \quad new \quad error \quad > \quad 1.04 \\ 1.05 & if \quad new \quad error \quad < \quad 1.04 \end{cases}$$

```
# input vector
p = [-1 -1 2 2; 0 5 0 5];

#target
t = [-1 -1 1 1];

# network with feed forward and gradient descent adaptation
net = newff(p, t, 3, {}, 'traingda');

# set learning rate to be 0.05
net.trainParam.lr = 0.05;

# set the ratio to increase learning rate
net.trainParam.lr_inc = 1.05;

#train the network
net = train(net, p, t);

#test the network
y = sim(net, p)
```

Default values of the parameters;

net.trainParam.lr	0.01	Learning rate
net.trainParam.lr_inc	1.05	Ratio to increase learning rate
net.trainParam.lr_dec	0.7	Ratio to decrease learning rate

traingdx – gd with momentum

```
# input vector
p = [-1 -1 2 2; 0 5 0 5];

#target
t = [-1 -1 1 1];
```

```
# network with feed forward and gradient descent adaptation
net = newff(p, t, 3, {}, 'traingdx');

# set learning rate to be 0.05
net.trainParam.lr = 0.05;

# set the ratio to increase learning rate
net.trainParam.lr_inc = 1.05;

# set the momentum
net.trainParam.mc = 0.7;

#train the network
net = train(net, p, t);

#test the network
y = sim(net, p)
```

Default values of the parameters

net.trainParam.lr	0.01	Learning rate
net.trainParam.lr_inc	1.05	Ratio to increase learning rate
net.trainParam.lr_dec	0.7	Ratio to decrease learning rate
net.trainParam.mc	0.9	Momentum constant

traincgp **–** conjugate gradient

```
net = newff(p, t, 3, {}, 'traincgp');
[net, tr] = train(net, p, t);
```

trainbfg **–** Quasi-Newton Algorithm, BFGS algorithm

```
net = newff(p, t, 3, {}, 'trainbfg');
[net, tr] = train(net, p, t);
```

trainlm **–** Levenberg-Marquardt algorithm

```
net = newff(p, t, 3, {}, 'trainlm');
[net, tr] = train(net, p, t);
```

Levenberg-Marquardt (LM) method

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as;

$$H = J^T J$$

and the gradient can be computed as;

$$g = J^T e$$

where J is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and **e** is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$w_{k+1} = w_k - \left[J^T J + \mu I\right]^{-1} J^T e$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

Madalines

Multiple Adalines.