On the Quasi-Newton Training Method for Feed-Forward Neural Networks

Nikolai Stanevski, Dimitar Tsvetkov

Abstract: The quasi-Newton training method is the most effective method for feed-forward neural networks with respect to the training precision. This method is well-known and popularly described in the neural networks literature. Nevertheless its implementation contains some difficulties because of the specific shape of the cost function and the large amount of variables. Here we give in sufficient details an example of a program implementation of the quasi-Newton method. This implementation (as a Borland Delphi application) seems to work well with various examples.

Keywords: feed forward neural networks, training methods, quasi-Newton method.

INTRODUCTION
Here we suppose that the reader is acquainted with the basis of the neural networks theory. Feed-forward (FF) neural network architecture consists of an input layer, output layer and several hidden layers. Suppose we are given \( L \) samples and each sample contains several input signals and several output signals. The problem is to train the network using given data that means to minimize some cost function which usually is the least square function. There are several second order optimization methods that may be used to train the network like conjugate-gradient method, Levenberg-Marquardt method and quasi-Newton method. Remember also the first order methods as steepest descent direction and its modification with momentum addend. All they avoid calculating of the Hessian matrix which calculation is very inconvenient for implementation. It is easy to verify that the quasi-Newton method is the best with respect to accuracy and therefore it is interesting and useful to investigate how it works and how it can be implemented in the practice. The quasi-Newton method is already well implemented in various applications, for example in MATLAB but all they have many hidden parameters and properties. The main aim of the present work is to offer in details a well working variant of quasi-Newton method as a computer implementation in Delphi application.

THE BACK PROPAGATION PROCEDURE
The Back Propagation procedure (BP) calculates gradient vector \( g(w) \) with respect to the weight vector \( w \). Here follows a brief presentation of BP. Let \( m = 0,1,\ldots,M \) is the layer index where index 0 stands for the input layer and \( M \) – for the output one. Denote by \( v[m,i], i = 0,1,\ldots,k[m], \) the signals of the \( m \)-th layer. It is convenient to use zero indices for the bias signal; then always we have \( v[m,0] = 1 \). The \( m \)-th layer consists of \( k[m] \) neurons and the \( i \)-th neuron is defined by its weighs \( w[m,i,j], i = 1,2,\ldots,k[m], j = 0,1,\ldots,k[m-1] \), and by its transfer function \( \varphi_m(u) \). The weight \( w[m,i,0] \) stands for the corresponding bias. In this way \( k[0] \) is the number of the inputs and \( k[M] \) is the number of the outputs. BP consists of several steps.

Consider the \( l \)-th sample. Then we have \( v[0,i] = \text{input}[l,i] \) for \( i = 1,2,\ldots,k[0] \), while the array \( \text{input}[l,i] \) is used for the set of inputs; for the outputs we use the array \( \text{output}[l,i], i = 1,2,\ldots,k[M] \).
BP1. Calculation of signals along the network (forward step)

\[ h[m,i] = \sum_{j=0}^{k[m-1]} w[m,i,j] \cdot v[m-1,j] \]  

(\text{for } m = 1,2,\ldots,M \text{ and } i = 1,2,\ldots,k[m]. \text{ Remember that we always have } v[m,0] = 1.)

Here we point out that the cost function for the \( l \)-th sample is

\[ \text{cf}_l(w) = \frac{1}{2} \sum_{i=1}^{k[M]} (v[M,i] - \text{output}[l,i])^2 \]  

(3)

BP2. Calculation of so-called deltas (backward step) starting from the output \( M \)-th layer for which we have

\[ \delta[M,i] = \varphi'_m(h[M,i]) \cdot (v[M,i] - \text{output}[l,i]), \]  

(4)

and for \( m = M, M-1, \ldots, 2 \), we calculate

\[ \delta[m-1,i] = \varphi'_m(h[m-1,i]) \sum_{j=1}^{k[m]} w[m,j,i] \cdot \delta[m,j] \]  

for \( i = 0,1,\ldots,k[m-1] \).

BP3. Calculation of the derivatives for \( \text{cf}_l(w) \) with respect to \( w \)

\[ g_l[m,i,j] = \delta[m,i] \cdot v[m-1,j], \]  

(6)

for \( m = 1,2,\ldots,M, \ i = 1,2,\ldots,k[m], \ j = 0,1,\ldots,k[m-1] \).

These derivatives are exactly the entries of the gradient

\[ g_i(w) = \frac{d}{dw} \text{cf}_i(w). \]  

(7)

Having once the values of \( g_i(w) \) we can update \( w \) with the well-known formula for incremental learning

\[ w_{new}[m,i,j] = w_{old}[m,i,j] - \eta g_i[m,i,j], \]  

(8)

where \( \eta \) is the learning rate parameter which usually takes value 0.05. In the process of the incremental learning, the steps above are repeated from the first sample to the last one that forms a separate epoch. A large number of epochs is needed to obtain a good network.

In the quasi-Newton method the learning mode is batch mode instead of the incremental mode and therefore the cost function is the sum of all the sample cost functions

\[ \text{cf}(w) = \sum_{l=1}^{L} \text{cf}_l(w). \]  

(9)

Then obviously

\[ g(w) = \frac{d}{dw} \text{cf}(w) = \sum_{l=1}^{L} g_i(w), \]  

(10)

and

\[ g[m,i,j] = \sum_{l=1}^{L} g_i[m,i,j]. \]  

(11)
THE QUASI-NEWTON METHOD – layer-by-layer implementation

Now we pay our attention to the line-search problem that means to solve the following optimization problem. Find \( t \geq 0 \) such that the scalar function

\[
\text{cf} \left( w - t g(w) \right)
\]

attains its minimum. This problem can be approximately solved in many ways. Fortunately the quasi-Newton method is not (too much) sensitive to the precision of the solution of the line-search problem. Here one can use straightforward calculations by varying \( t \) until some criterion is met (this simple strategy works well).

First of all it is important to know that the QN method is an abstract method which gives the exact solution of the minima problem in the case when the cost function is quadratic with a positively defined Hessian. In this sense QN looks better even than the pure Newton-Raphson method. Of course the cost function here is not quadratic but nevertheless QN converges well.

We will implement QN in a layer-by-layer mode. Consider the \( m \)-th layer (arbitrary chosen, \( m = 1, 2, \ldots, M \) ) and let \( w^m \) denotes the vector of the \( m \)-th layer weights. Suppose that we have the gradient \( g(w^m) \) in a vector form. Here arises a technical problem to transform the double-indexed array, as the gradient is obtained by \( \text{BP} \), to one-indexed and conversely. Denote by \( \omega \) the entry number of \( w^m \). Hereafter for brevity we will omit the upper index \( m \). Let \( w_0 \) be the starting value of the weight \( \omega \)-vector \( w \). (All the weights have to be initialized as small random numbers during the net construction.) Set a starting value of the \( \omega \times \omega \)-matrix \( H_0 = I \). QN method contains several stages.

QN1. Suppose we have already some current value \( w_n \). Next calculate \( g(w_n) \) and find the (approximate) solution \( t_n \) of the line-search problem

\[
\text{cf} \left( w_n - t H_n g(w_n) \right) \Rightarrow \min
\]

and set

\[
w_{n+1} = w_n - t_n H_n g(w_n).
\]

Calculate now \( g(w_{n+1}) \) and put

\[
\Delta w_n = w_{n+1} - w_n \quad \text{and} \quad \Delta g_n = g(w_{n+1}) - g(w_n).
\]

The values of the gradients are found by using of \( \text{BP} \). Now update the \( H \) matrix

\[
H_{n+1} = H_n + \frac{\Delta w_n \Delta w_n^T}{\Delta g_n^T \Delta g_n} - \frac{H_n \Delta g_n \Delta g_n^T H_n}{\Delta g_n^T H_n \Delta g_n}.
\]

QN2. Repeat steps QN1 (until a proper stop criterion is met). The number of repeats is at least \( \omega \) because the same number of repeats is necessary even in the case of a quadratic cost function. Our experience shows that it is sufficient to choose \( \omega \) as a number of these repeats.

QN3. Make a pass of steps QN1 and QN2 over the layers starting from the output layer to the input one. By analogy this pass can be called an epoch.

QN4. Repeat QN1-3 steps until the cost function becomes sufficiently small or stop changing itself (do some epochs).

All the \( H \) matrices occur positively defined and therefore the vector \( H_n g(w_n) \) has a positive projection on the gradient \( g(w_n) \). In fact \( H_n \) is an approximation to the inverse
Hessian matrix (in the quadratic case $H_\omega$ coincides with the inverse Hessian). The QN method looks straightforward and easy to program. The best fact here is that QN converges rapidly. The formulas in method described above corresponds to the so-called Davidon-Fletcher-Powell (DFP) method. There exist also other quasi-Newton methods. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method differs from DFP only in updating of $H$:

$$H_{n+1} = H_n + \frac{\Delta w_n \Delta w_n^T}{\Delta w_n^T \Delta g_n} - \frac{H_n \Delta g_n \Delta g_n^T H_n}{\Delta g_n^T H_n \Delta g_n} + \Delta g_n^T H_n \Delta g_n UU^T,$$

where

$$U = \frac{\Delta w_n}{\Delta w_n^T \Delta g_n} - \frac{H_n \Delta g_n}{\Delta g_n^T H_n \Delta g_n}.$$  \hspace{1cm} (17)

Separating the implementation of QN in the layer-by-layer mode obviously does not disturb the convergence. The numerical complexity of QN is $O(\omega^2)$ (in contrast with the other popular methods which complexity is $O(\omega)$) therefore QN becomes significant slower than the other methods in the case of very large neural networks. Nevertheless QN should be used when the learning precision is of the main importance.

EXAMPLE

In the following example we have 1 input, 2 outputs and 63 samples. Input is the uniform set of points in the segment $[-\pi, \pi]$ by step 0.1. First output is the values of the function $\cos(x)$ and the second output is the values of the function $\sin(x)$. The net has one hidden layer only with 5 neurons with a sigmoid transfer function. The initial error is 31.5. In the following graphics it is shown the error dynamics.

Figure 1. Plots of the output and simulated signals

<table>
<thead>
<tr>
<th>first output $\rightarrow \cos(x)$</th>
<th>second output $\rightarrow \sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 1 epoch, error=6.024</td>
<td></td>
</tr>
<tr>
<td>after 3 epochs, error=0.120</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>epoch number</th>
<th>error</th>
<th>epoch number</th>
<th>error</th>
<th>epoch number</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>31.500</td>
<td>6</td>
<td>0.090</td>
<td>11</td>
<td>0.079</td>
</tr>
<tr>
<td>1</td>
<td>6.024</td>
<td>7</td>
<td>0.087</td>
<td>12</td>
<td>0.076</td>
</tr>
<tr>
<td>2</td>
<td>4.128</td>
<td>8</td>
<td>0.086</td>
<td>13</td>
<td>0.075</td>
</tr>
<tr>
<td>3</td>
<td>0.120</td>
<td>9</td>
<td>0.084</td>
<td>14</td>
<td>0.073</td>
</tr>
<tr>
<td>4</td>
<td>0.093</td>
<td>10</td>
<td>0.082</td>
<td>15</td>
<td>0.070</td>
</tr>
<tr>
<td>5</td>
<td>0.091</td>
<td>11</td>
<td>0.081</td>
<td>16</td>
<td>0.066</td>
</tr>
</tbody>
</table>
By this example we see that the error decreases rapidly for the first epochs and then the decreasing becomes slowly. This common phenomenon is observed also for the other examples. Perhaps the reason for this slower error decreasing is hidden in the relatively imperfect solution of the line-search problem. On the other hand the more precise solution of the line-search problem makes the overall algorithm more slower which is the main disadvantage of the quasi-Newton method.

Let now see what is going on with the same example when the quasi-Newton updating of $H$ matrix is removed and the optimisation relies only on line-search. We need near 250 epochs to reach a visual satisfactory precision.

Figure 2. Plots of the output and simulated signals (only with line-search)

<table>
<thead>
<tr>
<th>first output $\rightarrow \cos(x)$</th>
<th>second output $\rightarrow \sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 250 epochs, error=0.573</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Error dynamics for 170 epochs (only with line-search)

<table>
<thead>
<tr>
<th>epoch number</th>
<th>error</th>
<th>epoch number</th>
<th>error</th>
<th>epoch number</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>31.500</td>
<td>60</td>
<td>2.596</td>
<td>120</td>
<td>1.958</td>
</tr>
<tr>
<td>10</td>
<td>4.510</td>
<td>70</td>
<td>2.516</td>
<td>130</td>
<td>1.813</td>
</tr>
<tr>
<td>20</td>
<td>3.288</td>
<td>80</td>
<td>2.439</td>
<td>140</td>
<td>1.691</td>
</tr>
<tr>
<td>30</td>
<td>3.006</td>
<td>90</td>
<td>2.338</td>
<td>150</td>
<td>1.581</td>
</tr>
<tr>
<td>40</td>
<td>2.835</td>
<td>100</td>
<td>2.248</td>
<td>160</td>
<td>1.464</td>
</tr>
<tr>
<td>50</td>
<td>2.685</td>
<td>110</td>
<td>2.141</td>
<td>170</td>
<td>1.339</td>
</tr>
</tbody>
</table>

Without updating of $H$, i.e. only with line-search strategy, the convergence becomes very slow (first order optimization method). The latter shows that the QN method is very self-consistent and does not admit arbitrary intuitive improvements.

CONCLUSIONS AND FUTURE WORK
Obviously the quasi-Newton method is easy to implement for small neural network architectures. The QN method can be explored intensively in many other optimisation problems of practical use. We are going to analyse the Levenberg-Marquardt method which is reported as a very good method both for the precision and the speed of learning process.

REFERENCES

ABOUT THE AUTHORS
Assoc. Prof. Dimiter Petkov Tsvetkov, PhD, Department of Mathematics, National Military University "Vassil Levski", Veliko Tarnovo, Phone: +359 62 600 210, E-mail: dimiter99@yahoo.com.
Nikolai Stoyanov Stanevski, PhD student, Cartographical Institute, Troyan, Phone: +359 670 2 58 59, E-mail: nstanevski@dir.bg.