A neural network implementation for data assimilation using MPI


1Laboratory for Computing and Applied Mathematics, Brazilian Institute for Space Research, Brazil
2Centre for Weather Forecast and Climate Studies, Brazilian Institute for Space Research, Brazil

Abstract

The meteorological data assimilation process can be described as a procedure that uses observational data to improve the weather forecast produced by means of a mathematical model. Traditional methods include deterministic and probabilistic approaches. Included in the latter class, Kalman filtering has the advantage of minimizing the error propagated during the data assimilation. However, this method demands a heavy computational power as it usually deals with meteorological parameters for a huge number of grid points. Recently, neural networks have been proposed as a new method for meteorological data assimilation by employing a Multilayer Perceptron network with backpropagation learning to emulate Kalman filtering at a lower computational cost. In order to further reduce that cost and to overcome performance bottlenecks of the training phase, code optimizations were performed, making use of classical, hardware-independent techniques and Fortran 95 intrinsic functions, thus eliminating inefficient routines. A former attempt to parallelize the code and run it in a 4-processor shared memory machine, made use of HPF (High Performance Fortran) directives imbedded in the optimized code. Performance was poor and therefore a new version of the Fortran 95 code was developed using Message Passing Interface library calls. This code was executed in a 16-node multicomputer showing good speed-up up to 4 processors.
1 Introduction

Data assimilation is a very important process in the numerical weather forecast. It allows the imbedding of observational data in the meteorological model. This data provides a feedback during the generation of the forecast in a real time fashion. However, the process of imbedding the observational data is not straightforward and it has to be done in a very smooth manner in order to minimize the propagation of errors in the forecast model. Usually, the assimilation process can be outlined as a two step iterative process [1]:

\[
\text{Forecast step:} \quad w^f_n = F[w^o_{n-1}] ; \\
\text{Analysis step:} \quad w^a_n = w^f_n + d_n ;
\]

where \(w_n\) represents model state variable at \(n\) time step, \(F[\cdot]\) is the mathematical (forecast) model, superscripts \(f\) and \(a\) denote forecast and analyzed values respectively, and \(d_n\) is the innovation of the observational data.

Several methods of data assimilation have been developed for air quality problems [2], numerical weather prediction [3], and numerical oceanic simulation [4]. In the case of atmospheric continuous data assimilation there are many deterministic and probabilistic methods [3]. Deterministic approaches include dynamic relaxation, variational methods and Laplace transform, whereas probabilistic methods include optimal interpolation and Kalman Filtering. Dynamic relaxation assumes the prediction model to be perfect, as does Laplace transform. Variational methods and optimal interpolation can be regarded as minimum–mean–square estimation of the atmosphere.

In the Kalman filtering, the analysis innovation \(d_n\) is computed as a linear function of the misfit between observation (superscript \(o\)) and forecast (superscript \(f\)):

\[
d_n = G_n (w^o_n - H_n w^f_n) ,
\]

where \(G_n\) is the weight (gain) matrix, \(w^o_n\) is the observed value of \(w_n\) and \(H_n\) is the observation matrix. An adaptive extended Kalman filter has been tested in strongly nonlinear dynamical systems for assimilation procedure, such as the Lorenz chaotic system. Kalman filtering has the advantage of minimizing the error in the assimilation plus propagating this minimized error from one data insertion to the next. However, this process involves a heavy computational load, in particular for large meteorological systems. A strategy to alleviate this load is the use of neural networks to emulate the accuracy of the Kalman filtering [5]. Neural networks can be efficiently applied to map two data sets according to Haykin [6], that also discusses several architectures proposed for neural networks. In particular, Multilayer Perceptron with backpropagation learning is presented.

In a recent paper, Gardner and Dorling [7] did a survey on applications of artificial neural networks (ANN) in meteorology, where a brief introduction about ANN and the backpropagation algorithm are shown. It also cites applications in
the atmospheric sciences aiming at: (i) prediction (air-quality: surface ozone concentration, sulfur dioxide concentrations; severe weather; Indian monsoon, Brazilian rainfall anomalies, solar radiation); (ii) function approximation (air-quality, modeling of non-linear transfer functions); and (iii) pattern classification (cloud classification; distinction between clouds and ice or snow; classification of atmospheric circulation patterns; land cover classification; classification of convergence lines from radar imagery; etc.). Although, the use of ANN for data assimilation can be understood as a case of function approximation, this application was not mentioned in Gardner and Dorling’s paper. The current work is based on an application of neural network with backpropagation learning for data assimilation. This paper shows the strategies used to optimize and parallelize the training phase. The training was applied on two physical models: chaotic Lorenz system [8] and Shallow Water model [9].

In order to optimize that code and overcome performance bottlenecks of the training phase, code optimizations were performed, making use of classical, hardware independent techniques and Fortran 95 intrinsic functions, thus eliminating inefficient routines. A former attempt to parallelize the code and run it in a 4-processor shared memory machine, made use of HPF - High Performance Fortran [10] directives imbedded in the optimized code. Run time measurements showed that data dependencies precluded possible parallelizations and execution time was not reduced in comparison to that of the sequential code. Therefore, a new version of the Fortran 95 code was developed using MPI – Message Passing Interface [11] library calls. This code was executed in a 16-node multicomputer showing good speed-up up to 4 processors.

The next section provides a brief description of the Multilayer Perceptron with Backpropagation Learning [6]. Further sections discuss the experimental results for the data assimilation test cases and present the strategies used to parallelize the code for the training phase. The final section assess the the performance that was obtained with different number of processors.

2 Multilayer perceptron neural networks with backpropagation learning

A ANN is an arrangement of units characterized by: (i) a large number of very simple neuron-like processing units; (ii) a large number of weight-biased connections between the units, where the knowledge of the network is stored; and (iii) a highly parallel, distributed control.

The processing element (unit) in an ANN combines linearly multiple weighted inputs that are forwarded to an activation function. There are several different architectures of ANN, most of which depend on the learning strategy adopted. The multilayer perceptron with backpropagation learning, or backpropagation neural network, is a feed-forward network composed of an input layer, an output layer, and a number of hidden layers for extracting high order statistics from the input data. Each of these layers that may contain one or more neurons. This is typically
known as supervised learning as both the input and the expected output are fed with data to train the network. Fig. 1 shows a backpropagation neural network with one hidden layer. Functions $g_i$ and $f_j$ provide the activation for the hidden layer and the output layer, respectively. In order to make the network more flexible to solve nonlinear problems, the activation functions for the hidden layer are sigmoid functions. Mathematically, a perceptron network simply maps input vectors of real values into output vectors of real values. The connections in the figure have associated weights that are adjusted during learning process, thus changing the performance of the network. There are two distinct phases in the usage of an ANN: the training phase (learning process) and the running phase (activation of the network). In the training phase, the weights are adjusted for the best performance of the network in mapping the many input-output vector pairs. In the activation phase, once the weights are established, new inputs are presented to the network in order to compute the corresponding outputs, based on what it has learned in the previous phase. The training phase of a multilayer perceptron is controlled by a supervised learning algorithm. The main difference between supervised and unsupervised learning is that the latter uses only information contained in the input data, whereas the former requires both input and output (desired) data, which allows the calculation of the network error as the difference between the calculated output and the desired vector. The Backpropagation Algorithm consists of the adjustment of the network weights by backpropagating such error through the network. The weight change rule is a development of the perceptron learning rule: weights are changed by an amount proportional to the error at that neuron unit times the output of the unit feeding into the weight. This is the essence of the so-called delta rule. The training phase can make use of two modes: batch mode and sequential mode [6]. The former deals with the whole input data whereas the latter carries out the
training based on each input pattern. The scope of this paper is restricted to batch mode in which all the input examples are taken at once and the learning procedure searches a set of weights $\theta$ and biases $\mu$ that minimizes the total squared error:

$$e_m = \sum_{k=1}^{N} \| F_{ANN}(X_k, \theta, \mu, m) - F(X_k) \|_2$$ (4)

where $N$ is the number of examples in the training set, $X_k$ is the input vector of example $k$, $\theta$ and $\mu$ are the weights and biases of the network, $F_{ANN}$ is the approximation and $F$ is the desired output value.

### 3 Experimental results

Tests for the data assimilation process were performed on the Lorenz system, given by following dynamical equations:

$$\begin{align*}
\frac{dX}{dt} & = -\sigma(X - Y) \\
\frac{dY}{dt} & = RX - Y - XZ \\
\frac{dZ}{dt} & = XY - bZ
\end{align*}$$ (5-7)

This system is integrated using the Euler predictor-corrector method adopting the following dimensionless quantities: $\Delta t = 0.001$, $\sigma = 10$, $b = 8/3$, $R = 28$, so that the system is in chaotic state, with initial conditions given by:

$$w = [X \quad Y \quad Z]^T = [1.508870 \quad -1.5312 \quad 25.46091]^T$$

The numerical experiment was made inserting observations every 12 time-steps. The observational data were the same as forecast data added to Gaussian deviations with zero mean, as in the test carried out by [12]. The data assimilation procedure was tested with 3 or 10 neurons in each of the two hidden layers. When the network was trained with 3 neurons, 259 iterations were necessary to determine weights and bias, whereas for 10 neurons, weights and bias were yielded in 152 iterations. However, the quality of assimilation was poor when using 3 neurons in the hidden layers, as shown in Fig. 2. On the other hand, a perfect assimilation was obtained with 10 neurons, as depicted in Fig. 3. Tests for data assimilation were also performed on the Shallow Water physical model equation [9]. Dyanami-
cal equations for this model are:

\[
\frac{\partial \zeta}{\partial t} + R_o \frac{\partial (u\zeta)}{\partial x} + \delta + R_\beta \nu = 0
\]  
(8)

\[
\frac{\partial \delta}{\partial t} + R_o \frac{\partial (u\delta)}{\partial x} - \zeta + R_\beta \nu + \frac{\partial^2 \phi}{\partial x^2} = 0
\]  
(9)

\[
\frac{\partial \phi}{\partial t} + R_o \frac{\partial (u\phi)}{\partial x} - R_o u_0 \nu + R_F \delta = 0
\]  
(10)

where \(u, \nu\) are zonal and meridian wind components; \(\phi\) is the geopotential; the term \(\delta = \partial u/\partial x\) is the divergence; \(\zeta = \partial \nu/\partial x\) is the vorticity; the quantities \(R_o = 0.10, R_F = 0.16, R_\beta = 10\), are dimensionless numbers that refer respectively to the Rossby number, the Froude number, and a number associated to the \(\beta\)-effect [9]. Hereafter, prognostic variables are grouped into a vector \(w = [\zeta, \delta, \phi]^T\). The system is discretized using forward and central finite difference method for time and space integration. The dimensionless quantities were derived from \(\Delta t = 1100\) seconds and \(N_x \Delta x = L = 10000\) km, where \(L\) is the total length of the channel and \(N_x = 32\) is the number of grid points. The numerical experiment was

Figure 2: Lorenz model using 3 neurons in the hidden layers.
made inserting observations every 11.1 hours. The observational data were the same as forecast data added to a Gaussian deviations with zero mean. For the data assimilation procedure tests were performed with 50 neurons in each of the two hidden layers. The weights and bias were generated after 179,519 iterations and the former, non-optimized, version of the program took several days to run. Fig. 4 shows the assimilation results for the geopotential $\phi$. Results for the $u$ and $v$ components were similar.

4 Performance analysis and optimization

The accuracy of the proposed assimilation method depends on the number of neurons in the intermediate layer as well as on the number of examples used to train the network. Therefore, good results cost a huge amount of processing time requiring code optimization. Initially, timing and profiling of the sequential code was done in order to identify performance bottlenecks [13]. This provided a path for code optimization and further parallelization. Classical, hardware-independent, optimizations were performed in the sequential code leading to a reduction of 40 % in the processing time. The original Fortran 77 code was translated from MATLAB backpropagation routines, which presented an inefficient code structure. This code was rewritten for the Fortran 95 language in order to use its array operations and
intrinsic functions, as in the case of the hyperbolic tangent. This helped to eliminate inefficient subroutines, taking advantage of the usually well optimized Fortran library functions.

An attempt to parallelize the optimized code was made using HPF – High Performance Fortran [10] directives, mainly the INDEPENDENT statement, specifying that there are no data dependencies between loop iterations and the FORALL statement, fully replacing the classical iteration-structured loop. This parallelization did not work as expected because the HPF compiler was not able to deal with the data dependencies that are inherent to neural network codes.

The current version, using MPI – Message Passing Interface [11] library calls was implemented achieving the intended parallelization. In the neural network code, the largest matrices have a number of columns equal to the number of examples (\( n_q \)) used to train the network. Another important issue is that in most cases, there are no data dependencies in the operations involving the elements of the \( n_q \) columns. Therefore, the matrices were partitioned in blocks of columns, being these chunks distributed among the processors. The number of lines of these matrices represents the number of neurons in each layer.

The training of the neural network is an iterative process and at each iteration it is calculated an error corresponding to the sum of the square differences between the output of the network and the target. This error is checked in order to update weights and biases. Convergence is achieved when this error is lower than a defined

**Figure 4**: Geopotential as function of time for the Shallow Water equation model using 50 neurons in the hidden layers.
acceptable tolerance. In the chosen parallelization strategy, the MPI directives were imbedded only in the main program, and the time-consuming routines are executed by each processor for its corresponding chunk of data. Each iteration demands communication in order to sum up the square differences and to update weights and biases for all processors, if this error is lower than the previous one. Some routines were merged, thus avoiding successive calls. This technique further reduced the processing time.

The Lorenz model test case was executed using the sequential version of the code in around 1 minute, which is not critical for practical purposes. This test case has \( n_q = 2000 \), but as the number of neurons is either 3 or 6, the ratio computation/communication would be too small. Therefore, the MPI version was tested for the Shallow Water model test case, with \( n_q = 500 \) being the number of neurons either 50, 60 or 120. This parallel code was executed on a distributed memory parallel machine, with 16 processing nodes. Processing times are shown in Tab. 1.

<table>
<thead>
<tr>
<th>no. processors</th>
<th>run time (h:min:sec)</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6:44:57</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2:22:13</td>
<td>2.8</td>
</tr>
<tr>
<td>4</td>
<td>2:05:50</td>
<td>3.2</td>
</tr>
<tr>
<td>5</td>
<td>2:25:39</td>
<td>2.8</td>
</tr>
<tr>
<td>10</td>
<td>3:40:07</td>
<td>1.8</td>
</tr>
</tbody>
</table>

5 Conclusions

The use of the proposed perceptron multilayer neural network for meteorological data assimilation is a feasible approach. In addition, the neural network codes are inherently parallel. Part of the optimization process was porting the code to Fortran 95 and removing inefficient routines. This provided a reduction of the processing time by a factor of 6. The Lorenz model test case was used to assess the proposed neural network methodology and the Shallow Water test case was used to check the efficiency of the parallel version of the optimized code. Communication requirements at every iteration precluded the efficient use of the HPF code. On the other hand, the MPI code showed good performance up to 4 processors. The use of more processors caused an excessive communication overhead for this particular problem. It was noticed a superlinear speed-up for the 2-processor run due to the particular data distribution significantly improving the memory access through the cache. Another issue is that the parallel execution has a numerical effect in the results, but without affecting the quality of the data assimilation. Particularly, the parallel execution accelerated the convergence by some hundreds of iterations.
It is difficult to provide a general rule for determining the appropriate number of intermediate layers as well as for choosing the number of neurons to be used in these layers. Therefore, these parameters must be obtained by numerical experimentation. Since these experiments consume a huge processing time, optimization and parallelization of the code are very much justified. Future work points to the use of other transfer functions such as logistic sigmoid, use of other neural network topologies such as radial base function and cascade correlation.

References