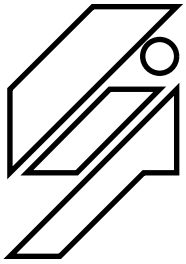
A mathematical model for feed-forward neural networks : theoretical description and parallel applications.

Cedric Gegout, Bernard Girau, Fabrice Rossi



Laboratoire de l'Informatique du Parallélisme

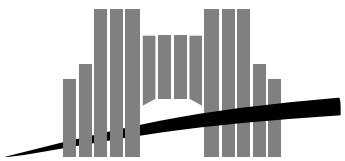
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

A Mathematical Model for Feed-Forward Neural Networks: Theoretical Description and Parallel Application

Cédric GÉGOUT
Bernard GIRAU
Fabrice ROSSI

September 1995

Research Report N^{O} 95-23



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

A Mathematical Model for Feed-Forward Neural Networks: Theoretical Description and Parallel Application

Cédric GÉGOUT
Bernard GIRAU
Fabrice ROSSI

September 1995

Abstract

We present a general model for differentiable feed-forward neural networks. Its general mathematical description includes the standard multi-layer perceptron as well as its common derivatives. These standard structures assume a strong relationship between the network links and the neuron weights. Our generalization takes advantage of the suppression of this assumption. Since our model is especially well-adapted to gradient-based learning algorithms, we present a direct and a backward algorithm that can be used to differentiate the output of the network. Theoretical computation times are estimated for both algorithms. We describe a direct application of this model: a parallelization method that uses the expression of our general backward differentiation to overlap the communication times.

Keywords: feed-forward neural networks, backpropagation, parallel implementation, network-partitioning

Résumé

Nous présentons un modèle général de réseaux de neurones différentiables non récurrents. Ce modèle inclut l'architecture standard du perceptron multicouche, aussi bien que ses dérivés classiques. Cette architecture standard se base sur une relation forte entre la notion de connexion du réseau et celle de poids des neurones. Notre généralisation tire profit de la suppression de cette relation. Le modèle présenté étant plus particulièrement adapté aux algorithmes d'apprentissage à base de gradient, nous présentons un algorithme direct ainsi qu'un algorithme rétropropagé pour le calcul de la différentielle de la sortie du réseau de neurones. Les temps de calcul des deux algorithmes sont estimés théoriquement. Enfin, nous présentons une des principales applications de ce modèle, basée sur l'algorithme généralisé de rétropropagation : une méthode de parallélisation qui se base sur une connaissance fine des calculs requis par cet algorithme afin d'introduire un recouvrement des communications par les calculs.

Mots-clés: réseaux de neurones non récurrents, rétropropagation, implémentation parallèle, partition de réseau

Introduction

The multi-layer perceptron (MLP) is now widely used as an efficient tool for classification and for function approximation. To enhance its performance and/or to reduce its training time, many authors have proposed modifications of its simple structure.

A MLP consists of several ordered layers of neurons. A neuron is a simple processing unit with several scalar inputs and one scalar output. It receives the output of each neuron of the previous layer with the help of a weighted connection. In order to compute its output, it multiplies each input by the corresponding connection weight, sums the resulting values, adds a threshold and then applies a transfer function to the sum. The simplest modification idea is to change the transfer function. The preprocessing computation may also be modified (see [10]). The neurons may be derived from a radial basis function (RBF) as in [9], or from a multidimensional wavelet ([11]).

Despite their differences, all these models share a common principle: they use an acyclic graph of simple units to compute a complex parametric vectorial function. Unfortunately, they are separately studied most of the time, especially in simulation softwares, where they are handled by totally independant objects, with totally different training algorithms.

In this paper, we extend a model proposed by Léon Bottou and Patrick Gallinari in [1]. The extended model allows to handle any feed-forward neural network model as a particular case of a general mathematical definition. The differential of the function calculated by a feed-forward network with respect to its parameters can be computed either with a direct method or with an extended back-propagation method. These methods are theoretically compared by means of a precise analysis of the operation amount they need. It shows that the back-propagation algorithm is not always faster than the direct method.

An efficient parallelization method of the back-propagation algorithm has been derived from our model. We take advantage of our precise study of the back-propagation principle to show that a neural network parallel mapping can be improved thanks to computation/communication overlapping. The efficiency of the derived parallel implementation is similar to the most advanced network-partitioning schemes, but it applies to *any* feed-forward network.

Chapter 1

The general model

In this chapter, we do not describe the exact mathematical model (e.g. the handled sets are ordered to exactly define the computation process, but this paper does not clearly define this ordering). It would be too long, and not really meaningful. The exact definitions, theorems and proofs can be found in a technical report [2][1]. We just provide the reader with the main ideas and results. Nevertheless, mathematical expressions remain when they are useful for the next chapter, or when they point out the cogency of the model. Their concrete meaning is always given.

1.1 Motivations

When creating this model, we had in mind our respective studies about MLP control, wavelet networks (WN) and MLP learning with genetic algorithms. The interest of a homogeneous approach of these different problems especially appeared when we wanted to use tools developed by one another. Despite its obvious significance, the model of [1] was not convenient for us. Our needs were:

- maximal generality, with regard to some given basic properties,
- a precise mathematical description of each handled object (and rigorous proofs),
- a clear distinction between these objects (and therefore, autonomous definitions),
- a strong relationship between the theoretical approach and the experimental requirements.

1.2 Feed-forward neural networks

The key ideas of the model are to generalize the notion of neuron and to allow an arbitrary feed-forward connection graph.

The main limitation of the standard MLP and of all its derivatives is the strong relationship between the connection graph and the network weights: one weight for each connection. Whereas the graph should only be a communication structure that allows the communication between the neurons, and the weights should be associated to the notion of controlling structure that allows a training algorithm to modify the local computation performed by each neuron.

Our generalization breaks this link: we use an arbitrary DAG (directed acyclic graph) for the communication structure and a control vector for each neuron. The relationship between the input of the neuron, its control vector and its output is only modeled by a vectorial function which can be of any form. All considered variables belong to **vectorial spaces**. This is useful for the generality of the model.

[1] available with the WWW at URL:
`http://www.ens-lyon.fr/~bgirau`

1.2.1 The neuron

In our model, a **neuron** N is a **differentiable** function. Its n **input** spaces are $I_1, \ldots, I_n$ (with $n \in \mathbb{N}^\star$). Its **output** space is O. Its control vector belongs to a **weight** space W. Its output is computed thanks to its input vectors and its weight vector.

Its partial differential with respect to its weight vector is called dN_w. Its partial differential with respect to its k-th input is called dN_{i_k}.

1.2.2 The neural net

The underlying structure of a neural net is a DAG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ ($\mathcal{N}$ is a set of **nodes**, $\mathcal{E}$ is a set of **edges**). A **feed-forward neural network** is a DAG, where the nodes are neurons that satisfy the following conditions. If N has no predecessor in the graph, then it is an **input neuron**, and it has only one input space. In is the set of the input neurons. If N has p predecessors, then it has exactly p input spaces, one for each predecessor in the graph (therefore the dimension of its k-th input space is equal to the dimension of the output of its k-th predecessor).

The input x of the neural network is the concatenation $(x_1, \ldots, x_{|In|})$ of the inputs of the input neurons. In the same way, its output is the concatenation of the outputs of the output neurons (no successor in the graph). The weight vector w of the whole network is the concatenation $(w_1, \ldots, w_{|\mathcal{N}|})$ of the weight vectors of all neurons in $\mathcal{N}$.

1.2.3 Computing the output

A rigorous mathematical approach requires an exhaustive definition of each handled function. Since there is no *a priori* knowledge about the graph, the only way to define the computed values is a recursive building. This method is correct only because the underlying graph is not cyclic.

To illustrate this, let us take the case of an intuitive result: the computation of the network output. Its mathematical description is as follows:

Let x and w be the input and weight vectors of $\mathcal{G}$. For any neuron N^l, its input E^l and its output o^l are recursively defined functions of x and w:

- $o^l(x, w) = N^l(E^l(x, w), w^l)$
- *If N^l is the k-th element of In, $E^l(x, w) = x_k$.*
- *If $N^l \notin In$, and if $P(l)_i$ is its i-th predecessor:*

$$E^l(x, w) = \left(o^{P(l)_1}(x, w), \ldots, o^{P(l)_{p^l}}(x, w)\right)$$

1.2.4 Input sharing

In a MLP, each input neuron uses the same input vector. In our model, this behavior is modeled by a differentiable input sharing function, that can be of any form. The simplest example is a replicating function which maps a vector x to a tuple $(x, \ldots, x)$ of size $|In|$. It allows all input neurons to share the same input, as in standard models (MLP, RBF networks or WNs).

The input sharing method does not change the differential of $\mathcal{G}$ with respect to its weight vector. Hence, it is not taken into account hereafter. The model also handles weight sharing.

1.3 Differentials within the general model

1.3.1 The direct computation method

As the neuron functions are differentiable, the function computed by the network is also differentiable (it is a composite function). The simplest method to compute its differentials is the standard chain rule. To simplify, if we consider the composite function $f(g(\ldots(h(x))))$, this method uses local differentiation at the

level of f, and global differentials inside the composite function. With very uncorrect notations, it might be written: $\frac{\partial f(\ldots x)}{\partial x} = \frac{\partial f}{\partial g(\ldots x)} \times \frac{\partial g(\ldots x)}{\partial x}$. The exact expression in the model is:

- *if* $N^l \neq N^k$*:*
  - *if* $N^l \notin In$ *(writing* E^l *for* $E^l(x,w)$*):*

$$\frac{\partial o^l}{\partial w^k}(x,w) = \sum_{j=1}^{p^l} dN^l_{i_j}\left(E^l, w^l\right) \frac{\partial o^{P(l)_j}}{\partial w^k}(x,w) \tag{1.1}$$

  - *if* $N^l \in In$*:*

$$\frac{\partial o^l}{\partial w^k}(x,w) = 0 \tag{1.2}$$

- *if* $N^l = N^k$*:*

$$\frac{\partial o^l}{\partial w^l}(x,w) = dN^l_w\left(E^l, w^l\right) \tag{1.3}$$

We have a similar property if we consider $\frac{\partial o^l}{\partial x_i}$, where x_i is the input of the i-th input neuron.

1.3.2 Back-propagation

The key idea of the back-propagation is to differentiate $f(g(\ldots(h(x))))$ by means of a local differential at the level of h and a global differential "above". With very uncorrect notations again, it might be written: $\frac{\partial f(\ldots x)}{\partial x} = \frac{\partial f(\ldots x)}{\partial h(x)} \times \frac{\partial h}{\partial x}$.

In the case of our model, we consider $o^k(x,w)$, the output of neuron N^k, as a function of $o^l(x,w)$, the output of another neuron N^l. We therefore define $o^{k\to l}(x,w,f^l)$ as the output of node N^k when o^l is set "free" from the network constraints (it can take arbitrary values, represented by f^l). It follows that $o^{k\to l}(x,w,o^l(x,w)) = o^k(x,w)$. But o^k and $o^{k\to l}$ are mathematically very different, and this equality is only satisfied for the constraint $f^l = o^l(x,w)$. The main mathematical difficulty is to prove that the intuitive differentiation is correct: the constraint does not add extra differentiation terms.

In the model, a first local equation states that o^k depends on w^l only through the output of N^l:

$$\frac{\partial o^k}{\partial w^l}(x,w) = \frac{\partial o^{k\to l}}{\partial o^l}\left(x,w,o^l(x,w)\right) dN^l_w\left(E^l, w^l\right) \tag{1.4}$$

A similar equation is fulfilled for $\frac{\partial o^l}{\partial x_i}$.

The back-propagation appears in the computation of $\frac{\partial o^{k\to l}}{\partial o^l}$. The main equation is, if there is a directed path from N^l to N^k in the graph:

$$\frac{\partial o^{k\to l}}{\partial o^l}\left(x,w,o^l(x,w)\right) = \sum_{\substack{N^j \\ \text{successor of } N^l}} \frac{\partial o^{k\to j}}{\partial o^j}\left(x,w,o^j(x,w)\right) dN^j_{i_{r(l,j)}}(E^j, w^j) \tag{1.5}$$

where N^l is the $r(l,j)$-th predecessor of N^j. If there is no directed path, the differential is null, and if $N^l = N^k$, it is the identity function.

A recursive method to compute $\frac{\partial o^{k\to l}}{\partial o^l}$ is given by formula 1.5: it needs $\frac{\partial o^{k\to j}}{\partial o^j}$ for every successor N^j of N^l. Therefore, $\frac{\partial o^{k\to l}}{\partial o^l}$ is computed from the last layer of the network to the input layer: this is a **backward** algorithm and therefore an extended back-propagation.

1.3.3 Error function

In order to train a neural network, we use an error function which estimates a distance between the output of the network and a desired output. This error $\mathcal{E}$ is considered as a function of the network weights. Gradient based training methods use its gradient $\nabla\mathcal{E}$. To compute $\nabla\mathcal{E}$, $\mathcal{E}$ can be handled as a composite function of the distance function and the neural output. In this case, the chain rule is applied to this composite function (the differential of the neural network output is therefore required: it can be computed either with the direct method or with the extended back-propagation).

The distance function can also be considered as a weightless final output neuron of the network, so that the back-propagation can be fully applied. If the output of this neuron is called $\mathcal{E}_{\mathcal{G}}(x,w)$, we can also define $\mathcal{E}_{\mathcal{G}}^{\rightarrow l}(x,w)$, when o^l is considered "free" from the network constraints. Then equations 1.4 and 1.5 become:

$$\frac{\partial \mathcal{E}_{\mathcal{G}}}{\partial w^l}(x,w) = \frac{\partial \mathcal{E}_{\mathcal{G}}^{\rightarrow l}}{\partial o^l}(x,w,o^l(x,w))dN_w^l(E^l,w^l)$$

$$\frac{\partial \mathcal{E}_{\mathcal{G}}^{\rightarrow l}}{\partial o^l}(x,w,o^l(x,w)) = \sum_{\substack{N^j \\ \text{successor of } N^l}} \frac{\partial \mathcal{E}_{\mathcal{G}}^{\rightarrow j}}{\partial o^j}(x,w,o^j(x,w))dN_{r(l,j)}^j(E^j,w^j) \tag{1.6}$$

In practice, a gradient based learning algorithm uses these equations. Since the output of $\mathcal{E}_{\mathcal{G}}$ is real, every back-propagated algebraic structure is therefore a gradient vector, wheras the direct method propagates matrices (so that the algebraic computations are intuitively more complex). Moreover, in the back-propagation algorithm, each non-local differential is the differential of the error with respect to a given neuron: we handle only one non-local differential per neuron. Whereas in the direct algorithm, for each neuron, we handle its differentials with respect to all its direct or indirect predecessors. It justifies some complexity results.

1.4 Complexity

The aim of this section is to compare the theoretical time needed by both differentiation algorithms. All proofs can be found in [2].

1.4.1 Notations and preliminary remarks

Both algorithms need to know the input, the output and the first-order differentials of each node. Therefore, the comparison will only focus on the cost of the algebraic operations required by both methods.

We introduce the following quantities:

- the main computation load is mostly due to the numerical operations needed for the algebraic operations, i.e. floating point number additions and multiplications. We will assume that the time needed to perform such an operation is 1. This is therefore the unit of our formulae.
- $m(i,j,k)$ is the time needed to multiply a (i,j)-matrix and a (j,k)-matrix (i.e approximatively $ik(2j-1)$) ;
- $s(i,j)$ is the time needed to sum to (i,j)-matrices (approximatively ij).

Let $|.|$ be the function which maps a vectorial space to its dimension (for instance, $|O^l|$ is the dimension of the output space of node N^l). The same notation will be used for the number of elements of a finite set (for instance $|\mathcal{N}| = n$).

Let $P^*(i)$ (resp. $S^*(i)$) be the set of the direct and indirect predecessors (resp. successors) of neuron N^i, including N^i. Let $P^+(i)$ (resp. $S^+(i)$) be the set of the direct and indirect strict predecessors (resp. successors) of neuron N^i, i.e. $P^+(i) = P^*(i) - \{N^i\}$ (resp. $S^+(i) = S^*(i) - \{N^i\}$).

1.4.2 Direct algorithm

Theorem 1 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, <)$ be a differentiable feedforward neural network. With the direct algorithm, computing the differential of G with respect to its parameter vector needs a time equal to:*

$$\sum_{N^j \not\in In} \sum_{N^l \in P^+(j)} \left((|P(j) \cap S^*(l)| - 1)\, s\left(\left|O^j\right|, \left|W^l\right|\right) + \sum_{N^k \in P(j) \cap S^*(l)} m\left(\left|O^j\right|, \left|O^k\right|, \left|W^l\right|\right) \right) \tag{1.7}$$

which is approximately equal to:

$$\sum_{N^j \notin In} \left|O^j\right| \sum_{N^l \in P^+(j)} \left|W^l\right| \left(2 \sum_{N^k \in P(j) \cap S^*(l)} \left|O^k\right| - 1 \right) \tag{1.8}$$

Corollary 1 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, <)$ be a differentiable feedforward neural network. Let F be an output function for $\mathcal{G}$, with O^F as output space. With the direct algorithm, computing the differential of F_G with respect to its parameter vector requires algebraic operations which total cost is:*

$$\sum_{N^j \notin In} \left|O^j\right| \sum_{N^l \in P^+(j)} \left|W^l\right| \left(2 \sum_{N^k \in P(j) \cap S^*(l)} \left|O^k\right| - 1 \right) + |O^F||W|(2|O| - 1) \tag{1.9}$$

1.4.3 Back-propagation

Theorem 2 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, <)$ be a feedforward neural network. With the back-propagation algorithm, computing the differential of G with respect to its parameter vector needs a time equal to:*

$$\begin{aligned} \sum_{k=1}^{out} \sum_{N^l \in P^+(Out_k)} \Bigg(& m(|O^{Out_k}|, |O^l|, |W^l|) + (|S(l) \cap P^*(Out_k)| - 1) s(|O^{Out_k}|, |O^l|) \\ & + \sum_{N^j \in S(l) \cap P^+(Out_k)} m(|O^{Out_k}|, |O^j|, |O^l|) \Bigg), \end{aligned} \tag{1.10}$$

approximately equal to:

$$\sum_{k=1}^{out} |O^{Out_k}| \left(|I^{Out_k}| + \sum_{N^l \in P^+(Out_k)} \left(|W^l|(2|O^l| - 1) + |O^l| \left(2 \sum_{N^j \in S(l) \cap P^+(Out_k)} |O^j| - 1 \right) \right) \right) \tag{1.11}$$

In the same way:

Theorem 3 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, <)$ be a differentiable feedforward neural network. Let F be an output function for $\mathcal{G}$, with O^F as output space. With the back-propagation algorithm, computing the differential of F_G with respect to its parameter vector requires algebraic operations which total cost is:*

$$|O^F| \left(\sum_{N^l} |W^l|(2|O^l| - 1) + \sum_{N^l \notin Out} |O^l| \left(2 \sum_{N^j \in S(l)} |O^j| - 1 \right) \right) \tag{1.12}$$

1.4.4 Comparison

In both cases (i.e. with or without error function), the cost formulae are not directly comparable: the theoretical costs must be computed to choose the fastest algorithm. Even for a standard MLP architecture, the number of neurons can be chosen so that the direct method is faster than the back-propagation to compute the differential of the network output.

Nevertheless computing the differential of the error is always faster with the back-propagation than with the direct method for the MLP and its derivatives (RBF networks and WNs). Another important result is that computing the differential of the error for a standard model within our mathematical model is as fast as doing it with the standard back-propagation algorithm.

1.5 Recapitulation

The proposed model generalizes the MLP model. It allows to separate the connections from the weights. Since it includes all standard feed-forward models, it allows to compare them. In order to use efficiently the standard gradient based learning algorithms, we developed several computation methods for the gradient of a neural network. Though it works well for MLPs, the back-propagation method is not the fastest algorithm for some other particular cases of the model.

Chapter 2

Parallelized Back-Propagation

2.1 Neural networks in parallel

A parallel implementation of neural computations is a possible solution for memory and time-consumming neural network applications (for instance real-time data processing). A survey of existing schemes to parallelize back-propagation can be found in [7]. The two main ideas are to distribute the patterns that are used for training, or to distribute the computation performed by the neural network (in this case, a pipeline may sometimes be used in addition).

Pattern-partitioning schemes require large pattern sets. They can handle any neural network, but they are not able to implement a stochastic gradient learning algorithm.

Network-partitioning schemes require large neural networks. An efficient solution is to use parallel implementations of the algebraic computation a MLP requires, but such a solution can not apply to other neural networks (RBF network, WN, sparse network). Another solution is to *map* the natural parallelism of a neural network onto the machine by means of a neuron partition among the processors. As it is shown in [5], a direct *mapping* of the neural network calculation leads to an unsatisfactory parallel efficiency.

The aim of this section is to show how our precise study of the back-propagation algorithm (paragraph 1.3.2, section 1) allows to design an efficient *mapping*.

2.2 Computation/communication overlapping (CCO)

The back-propagation algorithm is usually considered as a 2 steps calculation: computed outputs are first forwarded from the network input to its output, then the gradient is back-propagated from the output to the input. Considering equations 1.4, 1.5, and 1.6, each **local** computation can be divided into four steps, so that the time-consumming message transfers are overlapped by computation.

1. Neuron N^k computes its output thanks to the input $E^k(x, w)$ received from its predecessors.
2. N^k computes its local jacobian matrices (dN_i^k and dN_w^k), which only depend on the forwarded data. It sends **simultaneously** its computed output to its successors.
3. N^k computes the gradient of the error function with respect to its inputs ($\frac{\partial \mathcal{E}}{\partial o^{P(k)_j}}$, for all its predecessors $N^{P(k)_j}$), thanks to the backpropagated gradient ($\frac{\partial \mathcal{E}}{\partial o^k}$).
4. N^k computes the gradient of the error with respect to its weight vector ($\frac{\partial \mathcal{E}}{\partial w^k}$) and updates these weights thanks to a gradient descent. It sends **simultaneously** all $\frac{\partial \mathcal{E}}{\partial o^{P(k)_j}}$ to the corresponding predecessors.

In [6], a general description of the parallel implementation of any feed-forward neural network is given. We will only consider here the useful case of a multilayered network (MLP, RBF network or WN).

2.3 Implemented algorithm

Let L be the number of layers. Let n_l be the number of neurons in layer l. Consecutive layers are fully connected (standard *multilayered* structure).

The 4-step approach of the back-propagation can be used for any neural network parallel mapping ([6]). We consider here a *vertical sectioning* with p processors: each processor deals with n_l/p neurons in layer l. It is assumed that each n_l is a multiple of p. In practice, if a layer contains too few neurons, the whole layer should me mapped onto one processor.

If $\{N_{l,i}^{(p)}\}_{i=1..\frac{n_l}{p}}$ are the neurons of layer l mapped on processor p, each processor performs the following algorithm (hereafter called CCO algorithm):

for layer=1 to layer=L do
  step 1 performed for $\{N_{l,i}^{(p)}\}_{i=1..\frac{n_l}{p}}$
  step 2 for $\{N_{l,i}^{(p)}\}_{i=1..\frac{n_l}{p}}$ (the communication is a multinode broadcast)

for layer=L to layer=1 do
  step 3 for $\{N_{l,i}^{(p)}\}_{i=1..\frac{n_l}{p}}$ (except for layer 1)
  step 4 for $\{N_{l,i}^{(p)}\}_{i=1..\frac{n_l}{p}}$ (the communication is a multinode personnal reduction)

For a MLP, step 2 computation may be included in both step 3 and step 4. Therefore, step 2 only performs an all-to-all communication, whereas step 4 communication is still overlapped.

2.4 Algorithm efficiency

2.4.1 CCO versus other network partitions

The CCO principle can be applied to any neural network mapping, provided that the amount of computation allows the communication overlapping. Therefore, CCO may just be considered as an improving method for these parallel implementations, and it should be compared with the network-partitioning schemes that split each neuron computation onto several processors.

The first and main advantage of the CCO algorithm with respect to such algorithms is to **adapt to any feed-forward neural network**, and not only to MLP-like networks.

But for a numerical comparison, we limit ourselves to the case of MLPs. It is illustrated by figure 2.1, which uses the characteristics of an iPSC 860. Among numerous network-partitioning schemes, the *checkerboarding* method of [7] (hereafter called CB algorithm) may be taken as a reference, with regard to its high efficiency and scalability.

To simplify, the MLPs are assumed to have the same number n of neurons in each layer. The algorithms are implemented on a hypercube architecture of dimension d ($p = 2^d$) (hardware or simulated) to minimize the communication cost. The computation times of both CCO and CB algorithms are equal. If a communication time for N values is modelled as $\beta + N\tau$ for the standard case, and β for an overlapped communication, then the communication times are as follows:

- CCO: $(L-1)(2d\beta + kn\tau\frac{p-1}{p})$
- CB: $(L-1)(2d\beta + kn\tau\frac{2d}{\sqrt{p}})$

where k is the number of simultaneously handled patterns. Considering this model, the best algorithm is the CCO one if $d \leq 8$, i.e. $p \leq 256$, whereas the CB algorithm should be chosen when $d > 8$. Moreover, other current works show that if an optimal hybrid scheme (pattern- and network-partitioning are mixed with an optimal ratio to ensure the minimum learning convergence time) is considered, then the CCO algorithm should be chosen even with massively parallel computers ($p > 4096$ for instance).

It must be noticed that the CCO algorithm requires at least $\mathcal{O}(p)$ neurons per layer to allow the neuron partition and to obtain enough computation time to overlap the data transfers. Whereas the CB algorithm only requires $\mathcal{O}(\sqrt{p})$ neurons per layer.

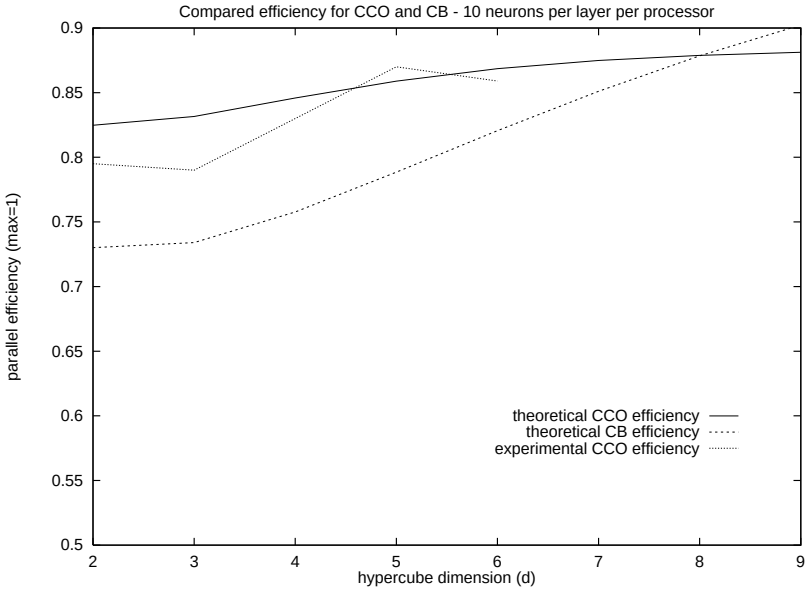


Figure 2.1: CCO versus CB (2 layers, $k = 5$)

Experiments on an iPSC 860 show that the efficiency formulae are reliable, though non-blocking communications do not exactly cost a constant time on this machine.

2.4.2 CCO versus pattern-partition

We also consider here the case of a pattern partition implemented on a hypercube architecture. We call it the PP algorithm.

Both CCO and PP may apply to any feed-forward neural network. As any network-partitioning scheme, the main advantage of the CCO algorithm is to **allow the parallelization of the stochastic gradient learning** (weight updating after each pattern presentation). Indeed, several experiments show that the number of required *epochs* (presentation of the whole training set) for learning convergence may be modelled as $A + kB$. For the PP algorithm, if each processor deals with b patterns, then $k = pb$ (whereas $k = b$ for CCO).

For a numerical comparison of the speedups (without considering the overall learning time), the communication times are:

- CCO: $2(L-1)d\beta$ except for a MLP (transfer time not overlapped within step 2, see above)
- PP: $d\beta + dW\tau$

where W is the number of neural network weights ($W = Ln(n+1)$ for a MLP, $W = 2Ln^2$ for a simple WN, $W = L(2n^2 + \frac{n^2(n-1)}{2})$ for some advanced versions of the WN model, ...). These communication times show that the PP algorithm should be chosen only for small neural networks. Figure 2.2 uses the parameters of an iPSC 860 (it contains only theoretically estimated efficiencies).

2.5 Recapitulation

Since the CCO algorithm is based on our general model, it adapts to any differentiable feed-forward neural network. Therefore its application domain is much larger than for the most efficient existing network-partitioning methods. Moreover, efficiencies are similar, though they are compared for the worst case of CCO, i.e. MLPs (forward communication not overlapped).

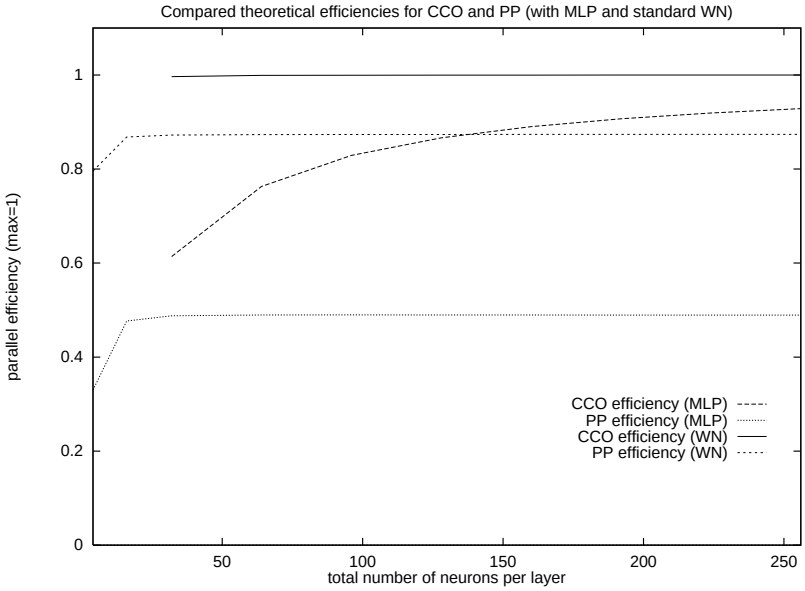


Figure 2.2: CCO versus PP (2 layers, $k = 160$, $p = 16$)

Since the CCO principle allows to improve a standard parallel mapping, it can still implement the stochastic gradient learning. Therefore it outperforms the pattern partitioning scheme for a training task, especially with numerous processors, provided that the handled neural network is large enough to allow the CCO implementation.

Conclusion

In this paper, we have presented a generalized model for feed-forward neural networks which includes and generalizes the MLP model and its derivatives. Gradient descent based training algorithms can be used in order to train any network which fulfills the general conditions. They will use the direct algorithm or the back-propagation algorithm in order to compute the gradient of the error made by the network on a specified training set: we have given theoretical cost formulae that allow to choose the fastest algorithm for a particular network architecture.

An efficient parallelization method has been designed, based on communication overlapping. This method derives from an analysis of our generalized back-propagation formulae. It can therefore apply to any feed-forward neural network.

Our model still extends: it handles second order differentiation methods, and it will soon deal with recurrent networks.

Bibliography

[1] L. Bottou and P. Gallinari. A Framework for the Cooperation of Learning Algorithms. In *Neural Information Processing Systems*, volume 3, pages 781–788. Morgan Kauffman, 1991.

[2] C. Gégout, B. Girau, and F. Rossi. A generic feed-forward neural network model. Technical report NC-TR-95-041, NeuroCOLT, Royal Holloway, University of London, 1995.

[3] C. Gégout, B. Girau, and F. Rossi. NSK, an Object-Oriented Simulator Kernel for Arbitrary Feed-forward Neural Networks. In *Int. Conf. on Tools with Artificial Intelligence*, pages 93–104, New Orleans (Louisiana), 1994. IEEE.

[4] L. Fuentes, J.F. Aldana, and J.M. Troya. Urano : an object-oriented artificial neural network simulation tool. In *Proc. Int. Workshop on Artificial Neural Networks*, volume 686, pages 364–369. Springer-Verlag, 1993.

[5] J. Ghosh and K. Hwang. Mapping neural networks onto message-passing multicomputers. *Journal of parallel and distributed computing*, 6:291–330, May 1989.

[6] B. Girau. Mapping neural network backpropagation onto parallel computers with computation/communication overlapping. In *Euro-Par'95 - Parallel Processing*, volume 966 of Lecture Notes in Computer Science, pages 513–524. Springer, 1995.

[7] V. Kumar, S. Shekhar, and M.B. Amin. A scalable parallel formulation of the back-propagation algorithm for hypercubes and related architectures. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1073–1090, 1994.

[8] A. Linden, Th. Sudbrack, Ch. Tietz, and F. Weber. An object-oriented framework for the simulation of neural nets. In *Advances in Neural Information Processing Systems*, volume V, pages 797–804. Morgan Kaufmann, 1992.

[9] T. Poggio and F. Girosi. Networks for approximation and learning. *Proc. IEEE*, 78(9):1481–1497, September 1990.

[10] F. Rossi and C. Gégout. Geometrical Initilization, Parametrization and Control of Multilayer Perceptrons: Application to Function Approximation. In *Int. Conf. on Neural Networks*, volume I, pages 546–550, 1994. IEEE.

[11] Q. Zhang and A. Benveniste. Wavelet networks. *IEEE Trans. On Neural Networks*, 3(6):889–898, November 1992.