ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0. Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

Parallelization of Deep Networks

Michele De Filippo De Grazia, Ivilin Stoianov, Marco Zorzi [*]

University of Padova - Dipartimento di Psicologia Generale
and Center for Cognitive Science - via Venezia, 8 Padova - Italy

Abstract. Learning multiple levels of feature detectors in Deep Belief Networks is a promising approach both for neuro-cognitive modeling and for practical applications, but it comes at the cost of high computational requirements. Here we propose a method for the parallelization of unsupervised generative learning in deep networks based on distributing training data among multiple computational nodes in a cluster. We show that this approach significantly reduces the training time with very limited cost on performance. We also show that a layerwise convergence stopping criterion yields faster training.

1 Introduction

A recent breakthrough in machine learning is the development of stochastic hierarchical generative models, implemented as neural networks with many hidden layers that learn increasingly more complex distributed nonlinear representations of the input data across layers without supervision [1]. These ”deep” networks outperform other machine learning algorithms on benchmark pattern recognition problems [2] and are extremely appealing for the purpose of neuro-cognitive modeling [3]. However, training of deep networks with millions of connections on large datasets poses a serious computational challenge. Here we show that it is possible to drastically reduce the training time by means of a parallel implementation on multi-core systems.

Deep Belief Networks (DBN) are neural networks composed of multiple layers of latent stochastic variables. A DBN can be viewed as a stack of learning modules, each of which is a Restricted Boltzmann Machine (RBM) [1]. A single RBM is a stochastic neural network that consists of one layer of visible units (encoding input data) and one layer of hidden units (feature detectors) connected by bidirectional and symmetric links. A RBM is trained to generate the data vector (i.e., maximizing the likelihood of recostructing the data) starting from a given state of the feature detectors and using the weights w_{ij} in a top-down direction. Contrastive-Divergence learning [2], given an input vector v_i^+, first actives the feature detectors h_j^+ (”positive” phase). Starting from stochastically selected binary state of the feature detectors (using their state h_j^+ as a probability to turn them on), it then infers an input vector v_i^- used in turn to reactivate the feature detectors h_i^- (”negative” phase). The weights w_{ij} are updated with a small learning fraction ϵ of the difference between input-output correlations measured in the positive phase and the negative phase:

$$\Delta w_{ij} = \epsilon \left(v_i^+ h_j^+ - v_i^- h_j^- \right)$$

*This work was supported by grant #210922 from the European Research Council to M.Z.

The RBM layers are trained in succession.

2 Parallel DBN

Here we propose a parallel implementation of DBN, based on non-proprietary software, on a heterogeneous cluster of computing nodes (network of multi-core boards) and examine its processing time and quality of learning.

Neural network training, inherently distributed, offers various levels of parallelization. For example, parallelization at the layer level implies that the calculations specific to subsets of feature detectors are distributed among the available processing nodes. In contrast, parallelization at the data level implies that the calculations relative to the entire network, but specific to subsets of data (single patterns or small batches of data) are distributed to the processing nodes [4, 5]. Parallelization at the layer level in neural networks with iterative activation or training requires relatively frequent synchronization of layer activation, which in a cluster of computers that communicate via network connections could prolong the computations due to traffic delays. This level of parallelization is more appropriate for specialized array processors, where data transfer time is deterministic and almost negligible.

For computer clusters, it is more appropriate to employ methods that require as little communication as possible. Training based on mini-batches allows such a very limited communication. The training data is divided into k batches $D_{j=1:k}$ that are equally distributed among the k available nodes. Each computational node j, on the basis of its data-set D_j, calculates in one training epoch an update vector ΔW_j of the common learner W. At the end of the epoch, when all nodes are ready, a master-node gathers all update vectors, and updates the common learner with the average of the single update-proposals: $\Delta W = 1/k \sum_{j=1:k} \Delta W_j$.

2.1 Test platform: 40-nodes MPI-controlled computing cluster

Message passing is the most common parallel programming paradigm for computer clusters. The key concept is exchange of messages between independent calculators (CPUs, or nodes). The messages transport data and synchronize the independent calculations. Message Passing Interface (MPI) specifies a language-independent communications protocol used to program parallel computational systems which execute independent processes that typically do not share common memory. MPI is the preferred technology due to high performance, scalability, and portability.

The experiments were executed on a HP distributed computing cluster. The cluster was composed of 5 nodes, each with two Quad-core processors and 32 GB of RAM. Overall, there were 40 cores. The nodes were interconnected with Infiniband-technology network [1]. The cluster was controlled by Linux (Mandrake 9.04 distribution), Octave 10.1, and Open-MPI library [2]. Open-MPI routines

[1]http://www.infinibandta.org

[2]http://www.open-mpi.org

were executed from Octave through the MPITB toolbox[3]. [6] showed that this toolbox outperforms all other MPI toolboxes for Octave/Matlab.

2.2 Implementation

We first adapted the original Matlab code of [2] for use in Octave, also generalizing it to arbitrary number of RBM-layers. In the parallel DBN implementation we added collective MPI routines [5, 7] that distribute mini-batches of data, execute training in parallel, gather proposed weight updates, and distribute an updated learner.

Table 1 illustrates the flowchart of the parallelization algorithm. The algorithm trains the overall DBN network sequentially, layer by layer, and every RBM layer in parallel.

(1*)		$W^0 = rand; i=0$ k=number-of-processors $D = D_1 \cup D_2 \cup \ldots \cup D_k$	
(2)	$\swarrow D_1, W^0 \ldots$	$\downarrow D_j, W^0$	$\ldots \searrow D_k, W^0$
(3)	$RBM_1^i \ldots$	RBM_j^i	$\ldots RBM_k^i$
(4)	$\searrow \Delta W_1^i \ldots$	$\downarrow \Delta W_j^i$	$\ldots \swarrow \Delta W_k^i$
(5*)		$\Delta W^i = \frac{1}{k} \sum_{j=1:k} \Delta W_j^i$ $W^{i+1} = W^i + \Delta W^i$	
(6)	$\swarrow W^{i+1} \ldots$	$\downarrow W^{i+1}$	$\ldots \searrow W^{i+1}$
(7)	*if not conv* $GOTO(3)\ldots$	*if not conv* $GOTO(3)$	*if not conv* $\ldots GOTO(3)$
(8)	$\searrow out_1^i \ldots$	$\downarrow out_j^i$	$\ldots \swarrow out_k^i$

(9*)		$D = \bigcup_h out_h^i$
		 if other layer $GOTO(1)$
		 else STOP	

Table 1: Flowchart of the proposed Parallel DBN. Rows and columns indicate sequential and parallel computations, respectively. Stars indicate operations at the master-node level.

The parallel training of each layer starts with an initialization step executed by a master node (step 1* of Table 1), which randomizes the learner' weights W^0 and randomly splits the training data D into k non overlapping subsets that contain a similar number of training samples: $D = D_1 \cup D_2 \cup \ldots \cup D_k$ (e.g.,[4, 5]). The training data of the first layer is simply the input data, while the training data of all other layers is the output of the immediately preceding layer. Then, the master node distributes to all nodes j the corresponding data-sets D_j and the weights of the learner (step 2).

At this point training starts in parallel on every node (step 3), each of which independently trains the same RBM on its portion of the training data. At completion of one learning iteration i, each RBM$_j$ has computed a candidate weight update ΔW_j^i. The master node collects then all weight updates (step

[3]`http://atc.ugr.es/javier-bin/mpitb`

4) in order to (step 5*) calculate a common update ΔW^i equal to the mean of the proposed updates [5, 8, 9, 10]. The common weight update ΔW^i is used to update the learner's weights: $W^{i+1} = W^i + \Delta W^i_j$, which are distributed back to all nodes (step 6) for a subsequent RBM training epoch.

Training continues with step 3 unless a convergence criterion is satisfied or a certain maximum number of epochs is reached, in which case the output of all networks is used as input data for the successive layer (steps 8 and 9*).

3 Empirical validation: learning handwritten digits

Learning time and performance of the parallel DBN were assessed on a classic handwritten digit recognition problem, similarly to [2]. We used the MNIST database, which contains handwritten digits encoded as 28x28-pixel grey-level images, size-normalized, mass-centered, and manually classified[4]. The data set has about 60,000 training images and about 10,000 test images with mini-batch size of 125 .

As in the original study [2], we used a network with three hidden layers (500-500-2000 units, respectively, for a total of about 1.6 million connections) and trained it with a biphasic procedure. RBM training was used in the first phase for unsupervised learning of the generative model. In the second phase, an output layer encoding the 10 digit classes was added on top of the deepest hidden layer and the entire network was fine-tuned with error back-propagation for discriminative learning. We examined the effect of parallelization on the unsupervised learning only, because the fine-tuning phase is optional (see, e.g., [3]) and it is not tied to a specific supervised learning algorithms. In particular, we trained the three hidden layers using 1, 2, 4, 8, 16, 20 and 40 computational cores. To obtain a reliable measure of the effect of the number of cores, we trained five networks for each number of cores. To obtain an objective measure of the goodness of the representations obtained, fine-tuning was then run on a single processor. We applied one of the two following stopping criteria: the first one simply used a fixed number of epochs (n=50, as in [2]). The second method was based on training convergence, in which learning was terminated when the mean reconstruction error on all training patterns RE did not decrease by less than a certain threshold value τ_{RE} for three consecutive iterations[5, 10], where τ_{RE} is a small empirically derived constant. In the fine tuning phase, back-propagation terminated when the RMS error (Root Mean Square Error) for all patterns did not decrease by a threshold value τ_{ERMS} for three consecutive iterations. The RMS error is typically used in problems with a large number of patterns[11].

To evaluate the performance, we collected the following measures: reconstruction error (measure of RBM learning), classification error (after fine-tuning), number of epochs to convergence (in case of convergence-based stopping criteria), and training time (for RBM only).

[4]MNIST: `http://yann.lecun.com/exdb/mnist/index.html`

3.1 Result and discussion

The results are shown in Figure 1 and Figure 2. Figure 1 shows the average (across five replica of the network) classification error (bars) and total RBM training time (line) as a function of the number of cores (ranging from 1 to 40), when the number of learning epochs was fixed (n=50). Figure 2(a) shows the classification error and total RBM training time when learning was stopped using a convergence criterion.

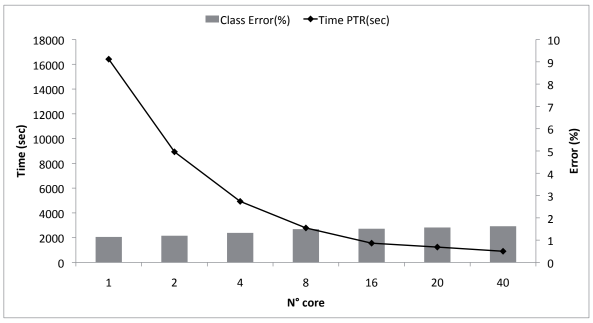
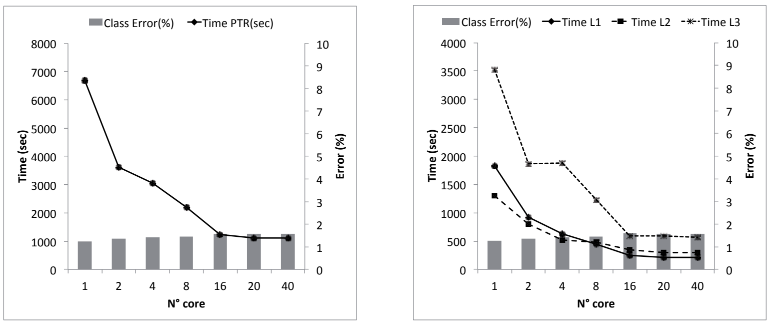


Fig. 1: Classification errors and RBM learning time as a function of number of cores. Training executed with a fixed number of epochs



(a) (b)

Fig. 2: Classification errors and RBM learning time as a function of number of cores using a convergence criterion for stopping. (a) total training time, (b) training time by hidden layer

As expected, RBM learning time quickly decreased as the number of computational cores increased (from 16500 sec for one core to 900 sec for 40 cores and fixed number of pretraining epochs). The decrease was well fit by a power function of core number (with a exponent of -0.86). At the same time, the classification error just slightly increased (from 1.14% for 1 core to 1.62% for 40 cores). RBM training based on a convergence stopping criterion yielded generally faster training times. Nonetheless, the number of cores still had a large influence on training speed, with a 6-fold decrease in learning time (from 6700 sec for 1 core to 1100 sec for 40 cores). Moreover, effects on classification and generalization were negligible (misclassification error of 1.25% for 1 core and 1.57% for 40 cores). This shows that the use of a convergence criterion based on reconstruction error can optimize RBM training. Figure 2(b) shows the effect of core number on the RBM training time for each hidden layer (the figure reports again classification errors, for reference). Layer 3, connecting 500 input units with 2000 feature units, required the greatest processing time due to its large number of connections.

Learning time appeared to asymptote at 16 cores when using the convergence criterion. This can be explained by the fact that the number of training epochs to convergence is strongly influenced by mini-batch size. In a follow-up experiment we observed that convergence speed decreased with increasing mini-batch size. Note that the number of patterns processed before the weights update is the product of core number and mini-batch size, thereby increasing when either factors increase. Therefore, the value of this product has a significant impact on scalability.

References

[1] G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, 2006.

[2] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.

[3] I. Stoianov and M. Zorzi. Emergence of a ”Visual Number Sense” in Hierarchical Generative Model. *Nature Neuroscience*, 15:194–196, 2012.

[4] P. Färber and K. Asanovic. Parallel neural network training on multi-spert. In *IEEE 3rd International Conference on Algorithms and Architectures for Parallel Processing*, Melbourne, Australia, December 1997.

[5] A. Margaris, S. Souravlas, E. Kotsialos, and M. Roumeliotis. Design and implementation of parallel counterpropagation networks using MPI. *Informatica*, 18(1):79–102, 2007.

[6] J. Fernández, M. Anguita, E. Ros, and J. L. Bernier. SCE Toolboxes for the development of high-level parallel applications. *Lecture Notes in Computer Science*, 3992:518–525, 2006.

[7] B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson, and N. C. Steele. Parallel implementations of feed-forward neural network using MPI and C# on .NET platform. In *Proceedings of the International Conference on Adaptive and Natural Computing Algorithms*, pages 534 – 537, Coibra, 2005.

[8] A. S. Ahmad, A. Zulianto, and E. Sanjaya. Design and implementation of parallel batch-mode neural network on parallel virtual machine. In *Industrial Electronic Seminar*, 1999.

[9] S. Mahapatra, R. N. Mahapatra, and B. N. Chatterji. A parallel formulation of back-propagation learning on distributed memory multiprocessors. *Parallel Computing*, 22(12):1661 – 1675, 1997.

[10] B. H. V. Topping, A. I. Khan, and A. Bahreininejad. Parallel training of neural networks for finite element mesh decomposition. *Computers & Structures*, 63(4):693 – 707, 1997. Computing in Civil and Structural Engineering.

[11] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., New York, NY, USA, 1995.