POLITECNICO DI TORINO
Repository ISTITUZIONALE

Parallel implementation of artificial neural network training

Original
Parallel implementation of artificial neural network training / Scanzio, S., Cumani, S., Gemello, R., Mana, F., Laface, P.. - STAMPA. - 1:(2010), pp. 4902-4905. (2010 IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP) Dallas (USA) 14-19 Marzo 2010) [10.1109/ICASSP.2010.5495108].

Availability:
This version is available at: 11583/2381224 since: 2017-11-21T14:20:33Z

Publisher:
IEEE

Published
DOI:10.1109/ICASSP.2010.5495108

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

05 September 2026

PARALLEL IMPLEMENTATION OF ARTIFICIAL NEURAL NETWORK TRAINING

Stefano Scanzio, Sandro Cumani*, Roberto Gemello^, Franco Mana^, P. Laface**

*Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino 10129, Italy
^Loquendo S.pA., Via Olivetti 6, Torino 10148, Italy

ABSTRACT

In this paper we describe the implementation of a complete ANN training procedure for speech recognition using the block mode back-propagation learning algorithm. We exploit the high performance SIMD architecture of GPU using CUDA and its C-like language interface. We also compare the speed-up obtained implementing the training procedure only taking advantage of the multi-thread capabilities of multi-core processors.
Our approach has been tested by training acoustic models for large vocabulary speech recognition tasks, showing a 6 times reduction of the time required to train real-world large size networks with respect to an already optimized implementation using the Intel MKL libraries.

Index Terms— Artificial Neural Network, Focused Attention Back-Propagation, GPU, CUDA, Fast Training

1. INTRODUCTION

State of the art speech recognition systems are based on acoustic-phonetic models of the words. Each acoustic unit is modeled by one or more states of a Hidden Markov Model (HMM). Gaussian Mixture Models (GMMs) are often used within each state to model the probability density of the acoustic patterns associated to that state. An attractive alternative to Gaussian mixture modeling is the use of an Artificial Neural Network (ANN) trained to estimate the posterior probability of each state given an acoustic pattern [1]. The main advantage of using a hybrid ANN-HMM approach in large vocabulary speech recognition is that the computation of the posterior probabilities of the HMM states takes a small fraction of the search time, moreover the ANN models are inherently discriminative.

The most widely used ANNs in speech recognition are feed-forward Multi Layer Perceptron (MLP) networks trained by the error back-propagation paradigm.

The core computation in MLPs, both in the feed-forward and in the back-propagation steps, is the inner product of a weight vector and of a feature vector (activation or error vector respectively). Several works have been published that exploit matrix multiplication to convert many inner-product operations into a single matrix multiply operation, and the capabilities of graphics-processors [2][3][4]. All these works focus, however, on the classification of static patterns.

In this paper we describe the implementation of the block mode back-propagation learning algorithm for speech recognition exploiting the high performance SIMD architecture of GPU. In our implementation we take into account all the peculiar aspects of training large scale sequential patterns, in particular, the re-segmentation of the training sentences, the block size for the feed-forward and for the back-propagation steps, and the transfer of huge amount of data.

The paper is organized as follows. Section 2 recalls the training steps of the ANN-HMM models. Sections 3 and 4 detail the techniques that can be used to accelerate the network training using single or multi-core CPUs, and GPU respectively. The experimental results are summarized in Section 5. Concluding remarks are reported in Section 6.

2. ANN-HMM MODEL TRAINING

The training databases usually include utterances of sentences or words collected from many different speakers, transcribed in terms of their corresponding phonetic units. Training hybrid ANN-HMM models, thus, requires these two alternate steps:

- Find the best alignment of the utterance frames to the states of the corresponding phonetic units.
- Find the weights of the network that better discriminate among the states, by producing for each state an estimate of the posterior probability P(state|**x**), where **x** is the input pattern.

In the following we will analyze the feed-forward and back-propagation training steps for a network having softmax output nodes and sigmoid activation functions for all the hidden nodes. The cross-entropy error function is used to compute the error between the output of the ANN and the target vector **t**.

The feed-forward step computes, for each layer, the outputs of the corresponding nodes given the layer input vector $\mathbf{X}_t = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)_t$ as:

$$net_i = b_i + \sum_{j=1}^{n} w_{ij} \cdot x_j \quad (1)$$

$$out_i = f(net_i) \quad (2)$$

where *n* is the number of nodes of the current layer, and $f(net_i)$ is the sigmoid function for all the hidden layers, and the softmax function for the output layer.

After the evaluation of the outputs produced by the feed-forward step, the weights of the network are updated on the basis of the error between the target vector **t** and the network outputs **out**. An error function that is appropriate for dealing with probabilities is the cross-entropy:

$$E_i = -t_i \cdot \ln(out_i) - (1 - t_i) \cdot \ln(1 - out_i) \quad (3)$$

The back-propagation algorithm proceeds by propagating the error from the output to the underlying hidden layers in order to correct the network weights. The procedure is summarized in Figure 1, where $e_i = \frac{\partial E_i}{\partial out_i}$, and

$$\partial_i^{(layer)} = \left[\frac{(1-t_i)}{(1-out_i)} - \frac{t_i}{out_i}\right] \cdot \left[out_i^{(layer)} \cdot \left(1 - out_i^{(layer)}\right)\right] \quad (4)$$

if *layer* is the output layer with softmax activation function and

$$\partial_i^{(layer)} = \left(\sum_{j=1}^{n} \partial_j^{(layer+1)} \cdot w_{ij}\right) \cdot \left[out_i^{(layer)} \cdot \left(1 - out_i^{(layer)}\right)\right] \quad (5)$$

if *layer* is a hidden layer with sigmoid activation function.

3. USING SINGLE AND MULTI-CORE CPUS

The first step toward an efficient straightforward implementation of the feed-forward and back-propagation procedures is to use efficient vector-by-matrix functions. More efficient implementations are, however, possible by updating the weights after processing a block of B input frames. This solution is a trade-off between batch and stochastic gradient back-propagation, which leads to accurate model estimation and allows parallel computation. In [5] high performance matrix multiply routines were introduced that, based on the work of [6], have been used to improve the training speed on a speech recognition data set.

The training algorithm can be further enhanced by using the so called focused-attention back-propagation (FABP) learning strategy [7]. FABP focuses attention on the patterns that are most difficult to learn. In FABP, the feed-forward step is performed, as usual, for all the patterns to compute the errors between the net outputs and the related targets. Back-propagation, however, is performed only for those patterns having a Mean Square Error (MSE) greater than a given threshold. This strategy not only speeds-up the training, but also improves the quality of the trained model reducing its dependence on the a priori probability of the classes in the training set. In FABP several training patterns fed to the network are not used for back-propagation (almost 80% in the last iterations). Thus, it is opportune that the feed-forward bunch (FFB) size is greater than the block size (BPB) used in the back-propagation step.

Taking care of these considerations the bunch feed-forward and the block back-propagation steps can be rewritten using matrix-by-matrix operations as shown in Figure 2 and Figure 3 respectively. In these figures, *T* indicates transposition, the superscript is the index of the network layer and the subscript gives the dimensions of the matrices, where B is the bunch size in the feed-forward step, and the block size in the back-propagation step. J and I are the number of output nodes of two adjacent layers. Introducing a dummy layer 0, the same index refers both to the layer weights and to its outputs. The variables η and β in Figure 3 are the dynamic learning rate adjustment and the momentum factor respectively.

The optimized implementation in the Math Kernel Library [8] of the Level 3 BLAS library functions has been used for the matrix-by-matrix multiplications in (2.2), (3.3) and (3.4). Moreover, since Intel provides the Integrated Performance Primitives for other vector operations, it is important to decompose operations such as (2.3), (2.4), (3.1), (3.2) and (3.5) in terms of a sequence of vector operations which can be mapped to fast library functions.

$$e_i^{(NLAYERS)} = \left[\frac{1 - t_i^{(NLAYERS)}}{1 - out_i^{(NLAYERS)}} - \frac{t_i^{(NLAYERS)}}{out_i^{(NLAYERS)}}\right] \quad \forall i \ (1.1)$$

for layer in range(NLAYERS, 1)

$$\partial_i^{(layer)} = (1 - out_i^{(layer)}) \cdot out_i^{(layer)} \cdot e_i^{(layer)} \quad \forall i \ (1.2)$$

$$e_j^{(layer-1)} = \sum_{i=1}^{N_{(layer-1)}} w_{ij} \cdot \partial_i^{(layer)} \quad \forall j \ (1.3)$$

$$\Delta w_{ij} = -\eta \cdot \partial_i^{(layer)} \cdot out_j^{(layer-1)} + \beta \cdot \Delta w_{ij} \quad \forall i \forall j \ (1.4)$$

$$w_{ij} = w_{ij} + \Delta w_{ij} \quad \forall i \forall j \ (1.5)$$

Figure 1. Back-propagation algorithm for a NLAYERS MLP.

$$\mathbf{out}_{(BxI)}^{(0)} = \mathbf{input}_{(BxI)} \quad (2.1)$$

for layer in range (1, NLAYERS)

$$\mathbf{net}_{(BxI)}^{(layer)} = \mathbf{out}_{(BxJ)}^{(layer-1)} \cdot (\mathbf{W}_{(IxJ)}^{(layer)})^T \quad (2.2)$$

if layer == NLAYERS

$$\mathbf{out}_{(BxI)}^{(layer)} = softmax(\mathbf{net}_{(BxI)}^{(layer)}) \quad (2.3)$$

else

$$\mathbf{out}_{(BxI)}^{(layer)} = sigmoid(\mathbf{net}_{(BxI)}^{(layer)}) \quad (2.4)$$

Figure 2. Implementation of the feed-forward step of the training algorithm for a NLAYERS MLP using matrix-by-matrix operations.

$$\mathbf{e}_{(BxI)}^{(NLAYERS)} = (1_{(BxI)} - t_{(BxI)}^{(NLAYERS)}) ./ (1_{(BxI)} - \mathrm{out}_{(BxI)}^{(NLAYERS)}) - (t_{(BxI)}^{(NLAYERS)} ./ \mathrm{out}_{(BxI)}^{(NLAYERS)}) \quad (3.1)$$

for layer in range(NLAYERS, 1)

$$\partial_{(BxI)}^{(layer)} = (1_{(BxI)} - \mathbf{out}_{(BxI)}^{(layer)}) .* \mathbf{out}_{(BxI)}^{(layer)} .* \mathbf{e}_{(BxI)}^{(layer)} \quad (3.2)$$

$$\mathbf{e}_{(BxJ)}^{(layer-1)} = \partial_{(BxI)}^{(layer)} \cdot \mathbf{W}_{(IxJ)}^{(layer)} \quad (3.3)$$

$$\Delta\mathbf{W}_{(JxI)}^{(layer)} = -\eta \cdot (\mathbf{out}_{(BxJ)}^{(layer-1)})^T \cdot \partial_{(BxI)}^{(layer)} + \beta \cdot \Delta\mathbf{W}_{(JxI)}^{(layer)} \quad (3.4)$$

$$\mathbf{W}_{(JxI)}^{(layer)} = \mathbf{W}_{(JxI)}^{(layer)} + \Delta\mathbf{W}_{(JxI)}^{(layer)} \quad (3.5)$$

Figure 3. Implementation of the back-propagation algorithm for a NLAYERS MLP using matrix-by-matrix operations. ./ and .* are the element-by-element division and multiplication respectively.

Further speed-up is obtained using the threaded MKL implementation of the matrix multiplication `cblas_sgemm` function that splits the execution over the available CPU cores.

Since the relative speed-up that can be achieved using these libraries is proportional to the size of the data, large feed-forward bunch size and back-propagation block sizes lead to better performance. However, the value of FFB must be bounded because all the patterns in the feed-forward bunch that are accepted by the FABP strategy, but exceed the BPB size, must be submitted again in the next feed-forward bunch introducing memory and computation overhead. In the experiments described in Section 5 the value of the bunch size FFB, and of the block size BPB have been set to 32 and 10 respectively.

4. SPEED-UP USING GPU AND CUDA

GPUs are graphics-oriented dedicated processors suited to computationally expensive but highly parallelizable tasks such as 3D graphic rendering. The main GPU architecture is characterized by the presence of a high number of floating point core processors which are able to perform parallel computations. The implementation of parallel computation tasks has been remarkably facilitated since the introduction by NVIDIA of the high-level programming language CUDA (Compute Unified Device Architecture), which provides a C-like interface to the programmable processors of the GPU and an efficient implementation of the BLAS library (CuBLAS).

In the CUDA environment a programmer sees the GPU as a multi-core processor allowing the concurrent execution of multiple threads which perform the same computations on different data. The computation is organized as a grid of thread blocks where each thread executes a single instruction set called kernel [9].

Since most of the computation in ANN training has a high degree of fine grain parallelism, the use of GPUs is particularly suited for this task. Examples of the use of the CUDA framework in the field of artificial neural networks are already present in literature. In particular, [4] presents a complete implementation of the training process for an MLP with sigmoid output units, assuming that all the training patterns and the corresponding targets are fixed and stored in the GPU memory.

Although our approach is similar to the latter, however, several problems have been taken into account for an effective implementation of the MLP training task for sequential patterns on a GPU. First of all, the epoch back-propagation approach proposed in [4] is not suited for training a network that has to model phonetic units for speaker independent speech recognition tasks. For these tasks the number of training patterns can easily exceed some millions, thus, the patterns cannot be stored in the GPU memory. Furthermore, since slow convergence problems easily arise when epoch back-propagation is used for such large data sets, it becomes mandatory to use the bunch training approach combined with FABP as described in Section 3. Care, however, is required to estimate the bunch and block sizes to achieve a good trade-off between model accuracy and training time.

To exploit the GPU computational power, we map the matrix implementation on either CuBLAS functions or specific kernels. Matrix multiplications are performed by means of the `cublasSgemm`, a fast and hardware-optimized implementation of matrix products function [10], both in the feed-forward (2.2) and in the back-propagation (3.3, 3.4) steps. Carefully tailored kernels have been implemented for the softmax and sigmoid functions, the MSE evaluation, and the update of the network weights.

Moreover, due to the architecture of the GPU, since CuBLAS functions give optimal performance with matrices having sizes that are multiples of power of 2, we enforce zero padding to our matrices which do not meet this condition. In particular, we pad to multiples of 32 and 16 the structures for the feed-forward bunch and back-propagation block respectively, and to multiples of 32 the weight matrices, obtaining a 10% improvement as shown in the last two rows of Table 2 in Section 5.

Although the CuBLAS library includes efficient functions to copy data from the host to the card and copy results back from the card to the host, these transfers of data can easily become the performance bottleneck. Thus, the network weights are kept in the GPU memory, but the input patterns are loaded on the GPU in bunches due to the small size of the GPU memory compared to the dimensions of the training data.

The core steps of the training algorithm are as follows:

1. Load a bunch FFB of input patterns on the GPU memory. Takes 2.5% of the total time using function `cudaMemcpy`.
2. Execute the feed-forward step and evaluate the MSE for each pattern. Takes 32.5% of the total time.
3. Transfer both the bunch MSEs and output patterns from the GPU to the main memory. The output patterns are required by the forced alignment performed by the Viterbi algorithm, running in the host machine, to obtain a new, more precise, association of the input patterns to the targets. The MSE of each pattern is simply tested in the host processor to select according to FABP the patterns on which back-propagation has to be performed. Takes 6.2% of the total time using function `cudaMemcpy`.
4. The pattern vectors in the GPU memory selected for back-propagation in step 3 are appended to the back-propagation matrix structures (of size BPB). Takes 10% of the total time.
5. Repeat all the previous steps until a sufficient number of patterns (the block size BPB) are selected for back-propagation.
6. Execute the back-propagation procedure for the block of selected patterns and update the weights. Takes 48.8% of the total time.
7. Since the weights have been updated, all the remaining patterns in the feed-forward bunch that were accepted by the FABP strategy, but exceed the BPB size, must be re-submitted in the next feed-forward bunch.
8. Repeat from 1. until all input patterns have been processed.

These steps are iterated until convergence is reached as decided by a stopping criterion based on maximum number of iterations, rate of decrease of the MSE, or recognition performance on a held out development set.

It is worth noting that most of the time is spent performing actual computations, while the transfer of inputs and outputs between CPU and GPU contributes to the total time for just 8.7%. The overhead of the memory transfers inside the GPU of the output patterns selected by the FABP strategy from the feed-forward to the back-propagation structures is significant but unavoidable for speeding-up the back-propagation computations. Excluding memory transfer times, and looking at the actual computations, most of the time (about 74%) is spent evaluating matrix products by means of the CuBLAS library function `cublasSgemm`, whereas the remaining time is used by the kernels.

5. EXPERIMENTS

Two set of experiments were performed to test the speed-up achieved by using the proposed approaches, without any loss of the recognition accuracy. The hybrid ANN-HMM models, detailed in [11], that are used by the Loquendo Automatic Speech Recognizer [12] were trained and tested. The models are 4-layer perceptrons with about 1M weights, trained with 150 to 500 hours of speech. The hardware setup was a HP xw8600 workstation equipped with a quad-core 3.0 GHz CPU, 1600 MHz FSB, 8 GB RAM, NVIDIA GTX280 GPU, and running Linux RedHat RHEL 5.2 EM64T.

The first set of experiments was devoted to training English models using the Wall Street Journal 0 corpus. Table 1 shows the elapsed training time, and the relative speed-up of four implementations.

Table 1. Training time, and relative speed-up for the WSJ0 corpus.

Training Implementation	Time hh:mm	Speed-up vs Standard C	Speed-up vs MKL
Standard C	42:35	-	-
Single thread MKL	11:36	3.7 x	-
Multi-Thread MKL	9:41	4.4 x	1.2 x
CUDA no-padding	2:29	17.1x	4.7x
CUDA with padding	2:14	19.1 x	5.2 x

The baseline is a standard C language implementation without the optimized matrix functions offered by the BLAS library, it is compared with single and multi thread programs using the INTEL MKL libraries, and finally with the support of a GPU board and its optimized routines. It is worth noting that to obtain a fair comparison of the GPU and MKL implementations, the bunch size FFB of the latter has been set to 15 rather than to 32. This has been done to avoid in the MKL implementation the significant overhead of re-processing in the next feed-forward step the patterns accepted by the focused attention mechanism but exceeding the back-propagation block size BPB. This problem is much less relevant in the GPU because bunch of patterns are processed in parallel.

In this case study a speed-up of 3.7 is obtained by enhancing an already optimized standard C language implementation to process bunch of patterns using the MKL libraries. The improvement obtained using the GPU is much larger: a factor of 19.1, which is 5.2 times faster than the single thread MKL implementation. Although the multi-thread MKL version does give a 20% improvement compared to the single thread, it fully occupies the 4 cores, which can be used instead for training different models in parallel.

In the second set of experiments, without changing the setup, the models of 6 different languages have been re-trained using the large corpora collected for creating the models used by the Loquendo decoder. In these experiments we computed the elapsed training times to evaluate the obtained speed-up and we checked the recognition accuracy on 8 different recognition application grammars to assess that fast computation does not produce statistically significant recognition performance variations. The grammars include common recognition tasks such as yes-no, connected digits, numbers, spelling, dates, etc. The reference models are the ones released with the Loquendo ASR recognizer.

The models were re-trained using the single thread MKL implementation on the same workstation hosting the NVIDIA GPU to obtain a fair comparison of the training times. On average, training of the models with the CUDA-GPU implementations is 5.9 faster than using the single thread MKL optimized implementation.

Finally, an additional NVIDIA GTX295 GPU board has ben added to the hardware configuration to test the possibility of training more networks in parallel. This board has two GPUs, but a lower clock both for its memory and core processors. In this configuration the workstation has 3 GPUs of comparable computational power.

A set of training sessions have been done using the Wall Street Journal corpus to verify whether the simultaneous run of more than one training session slows down the GPU-CUDA implementation, possibly generating congestion problems on the PCI-Express bus.

Table 2 shows the elapsed times, in hours and minutes, for training the 7236 sentences of the corpus. Each row in the Table represents a single (row 1 and 2) or multi-model (row 3, 4 and 5) train experiments.

Table 2. Training time, in hours and minutes, for single and parallel training sessions running on different GPUs

Test	GTX280		GTX295(1)		GTX295(2)	
	Time	% Incr	Time	% Incr	Time	% Incr
1	2:14	-	idle		idle	
2	idle		2:20	+4.5	idle	
3	2:14	0	2:20	+4.5	idle	
4	idle		2:20	+4.5	2:20	+4.5
5	2:15	+0.7	2:22	+6.0	2:21	+5.2

As shown in the last row, this configuration of the workstation is able to carry out three parallel sessions with a very small increase of the average training time. We can conclude that the GPUs and the PCI-Express 16x Gen2 bus are not a bottleneck for the fast GPU-CUDA training implementation.

6. CONCLUSIONS

We have studied and implemented a complete ANN training procedure for sequential patterns exploiting the computational power of inexpensive GPU boards. We have tested this approach for training acoustic models for large vocabulary speech recognition tasks, showing a 6 times reduction of the time required to train real-world large size networks with respect to an already optimized implementation using the INTEL MKL libraries.

The obtained speed-up not only improves the efficiency of the generation of acoustic models, but also makes easier the research activity related to acoustic modeling such as testing different definitions of the acoustic units, experimenting with different neural networks structures and topologies. Reducing the training time also allows training larger models with huge training corpora.

6. REFERENCES

[1] H. Bourlard, N. Morgan, “Connectionist Speech Recognition: A Hybrid Approach”. Kluwer Academic Publishers, 1993.

[2] K.S. Oh, K. Jung, “GPU implementation of neural networks”. Pattern Recognition, Vol. 37, pp.1311-1314, 2004.

[3] H. Jang, A. Park, K. Jung, “Neural Network Implementation Using CUDA and OpenMP”. In: Proc. Digital Image Computing: Techniques and Applications, DICTA '08, pp.155-161, 2008.

[4] S. Lahabar, P. Agarwal, P.J. Narayanan, ”High Performance Pattern Recognition on GPU”. In: Proc. National Conference on Computer Vision Pattern Recognition, Image Processing and Graphics, NCVPRIPG'08, pp. 154-159, 2008.

[5] Bilmes, J., Asanovic, K., Chin C., Demmel, J., 1997. Using PHiPAC to speed error back-propagation learning. In: Proc. Internat. Conference on Acoustics, Speech and Signal Processing, ICASSP-97, pp .4153-4156.

[6] D. Anguita, G. Parodi, R. Zunino, “An Efficient Implementation of BP on RISC-based Workstations”. Neurocomputing, Vol. 6, pp.57-65, 1994.

[7] J.C. Hoskins, “Speeding Up Artificial Neural Networks in the “Real” World”. In: Proc. 1989 IJCNN, p. 626, 1989.

[8] Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl

[9] NVIDIA: NVIDIA CUDA Programming Guide, http://developer.download.nvidia.com/ compute/cuda/2_0/docs/.

[10] V. Volkov, J.W. Demmel, “Benchmarking GPUs to tune dense linear algebra”. In: Proc. 2008 ACM/IEEE Conference on Supercomputing, pp. 1–11, 2008.

[11] D. Albesano, R. Gemello, F. Mana, “Hybrid HMM-NN Modeling of Stationary-Transitional Units for Continuous Speech Recognition”. Proc. Neural Information Processing, pp. 1112–1115, 1997.

[12] Loquendo ASR: www.loquendo.com/en/technology/asr.htm