Progress Report
24-31 October 2011

Amir Hesam Salavati
E-mail: hesam.salavati@epfl.ch

Supervisor: Prof. Amin Shokrollahi
E-mail: amin.shokrollahi@epfl.ch

Algorithmics Laboratory (ALGO)
Ecole Polytechnique Federale de Lausanne (EPFL)

November 4, 2011

1 Summary

In the last week, I was mainly busy in making the code for implementing the null space learning problem parallel. The motif was the necessity of having larger-scale simulations which requires a lot of computational time. Therefore, to speed things up, I modified the original algorithm so that it can be run in parallel using our cluster. Aside from that, I managed to read a new paper on a similar idea that I am implementing now. In what follows you can find more details on the code implementation as well as the paper.

2 Implementing the null space learning algorithm

The main idea of identifying the null basis is adopted from [1] with some modifications in order to make it realizable by our double layer network in a distributed fashion. The overall learning algorithm with our modifications is given by equation (1)

$$y_j = \sum_{i=1}^{m} x_i w_{ij} \tag{1a}$$

$$w_{ij} = w_{ij} - \alpha y_j \left(x_i - \frac{y_j w_{ij}}{\|w_{:j}\|^2} \right) - \lambda 2 \tanh(\sigma * w_{ij})(1 - (\tanh(\sigma w_{ij}))^2) \tag{1b}$$

$$x_i = x_i - 2\beta y_j w_{ij} \tag{1c}$$

In the above equations n is the size of patterns, m is the number of constraint nodes, x is the data vector, α, β and γ are small constants while *sigma* has a relatively large value.

In words, y_j is the projection of $x^{(j-1)}$ on the j^{th} basis vector. If for a given data vector x, y_j is equal to zero, namely, the data is orthogonal to the weight vector w_j, then according to equation (1b) the weight vector j will not be updated. However, if the data vector x has some projection over w_j then the weight vector is updated towards the direction to reduce this projection. Furthermore, since we are interested in finding m orthogonal vectors, for each such vectors, say w_j, we subtract the effect of the previous layer and feed the new layer with the new vector, according to equation (1c).

2.1 Faster algorithm: learn and recall at the same time!

So far, the algorithm yields a sparse matrix which satisfies the set of linear equations $Wx = 0$, for all vectors x in the training data set. However, since we have to go *several times* over 2^k such vectors, and for each vector update m vectors of length n, the whole learning process is extremely slow! Off course main part of this slowness is due to the large number of patterns that we want to memorize. Therefore, I have tried a different scheme in which the algorithm do the learning phase for a number of patterns, say 10000 of them. At the end of this learning phase, the algorithm has a rough estimation of the subspace that the training patterns belong to.

Then, the network starts doing the recall phase. During the recall phase, if the amount of constraint violation for each constraint node is less than a threshold, the algorithm assumes this is just a pattern that it had memorized before and do the normal error correction procedure explained in our paper [2]. However, if the amount of constraint violation is too much, it assumes the input

corresponds to a new pattern (possible noisy as well) and do a learning step to memorize this pattern. Now if this pattern is encountered later, it can be corrected.

Obviously, there is a trade off between accuracy and speed here, but this scenario makes sense biologically as well. Because we do not learn everything at once but rather do the process in a gradual manner.

2.2 Making the Algorithm Parallel

Obviously, in order for equations (1) to work, we must remove the projection of current data vector over the set of weights connected to constraint node y_j before moving to the node y_{j+1} to find the basis vector $j+1$. However, one might think of doing all these in parallel, i.e. all constraint nodes update the set of synaptic weights connected to them at the same time and then all these updates are added to the actual weights. By repeating this process several times, one might hope to get a valid weight matrix which is orthogonal to the set of data.

I implemented this process in two versions. In the first version, all weights are initialized centrally and stored on a file somewhere. Then, the code for learning the connectivity for one constraint node is executed many times in parallel. Each instance of the code picks one of the rows of the initial weight matrix as the initial vector and update it gradually until convergence is reached. Then the new weight is written on a file. In the end and when all instances are finished, this file contains the weight matrix which can be used in the recall phase.

The second version, which is more realistic, is essentially the same as the previous one but instead of initializing the weight vectors centrally, each instance of the code initialize the weights independently at random. The rest of the learning process is the same as before.

The first version was tested successfully with $\beta = 0$ in equation (1). Having $\beta = 0$ has the risk of ending up with redundant and exactly the same constraints. However, for large network sizes most of the constraints will be unique.

The second version has not been successful unfortunately. The main problem is that when different instances of the code do the initialization independently, the final weight matrix will contain all-zero columns with high probability. In that case, one can not correct any errors on the pattern nodes corresponding to those columns. And to avoid the all-zero solution, one will need a penalty function which results in the solution not to be relizable in parallel (at least in its current format).

3 Paper Summaries

In [3] the authors address the problem of finding the first m principle components of a data set using a neural network. They propose a learning method based on Hebbian learning for identifying the principle components. The suggested rule has also an anti-Hebbian term for making the derived components orthogonal. The proposed rule has two main advantages:

1. It makes it possible to learn components one by one. So if we are not aware of the exact number of principle components, we can learn them in a step-by-step fashion and add constraints whenever necessary.

2. Because of a variable step-size in the learning process, the algorithm is faster than its rivals.

More specifically, the model for APEX is given in figure 1. The model is a *feed-forward network* and composed of n input neurons, $\{x_1, \ldots, x_n\}$, connected to m output nodes, $\{y_1, \ldots, y_m\}$. There are two sets links in the network: the feed-forward weights $\{w_{ij}\}$ (or *Hebbian connections*), which converge to the principal components, and the lateral weights $\{c_j\}$ (or *anti-Hebbian connections*), which serve the orthogonalization purposes. In the end, the set of weights $\{w_{ij}\}$ will *converge to an*

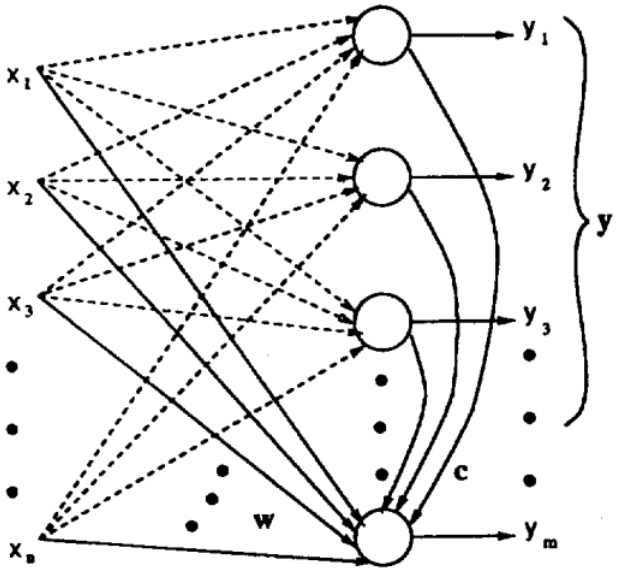

Figure 1: The APEX model: The solid lines denote the weights $\{w_{ij}\}$ and $\{c_j\}$, which are being trained to learn the m^{th} principle component.The dashed lines correspond to the weights of the already trained neurons. Since the lateral weights $\{c_j\}$ asymptotically converge to zero after being trained, they dont appear between the already trained units.

orthonormal basis set that span for the m-dimensional principal components subspace. The lateral weights $\{c_j\}$ will converge to zero.

Let's denote the input vector by $x = [x_1, \ldots, x_n]^T$ and the $m-1$ trained output elements by $\mathrm{y} = [y_1, \ldots, y_{m-1}]^T$. Since APEX finds the components one by one, at the m^{th} step the first $m-1$ rows of the weight matrix are equal to $e_1, \ldots, e_{m-1}$ and we would like to train the m^{th} row of the matrix such that it converges to e_m. To that end, let $\mathcal{W}^{m-1} = [e_1, \ldots, e_{m-1}]^T$, be the set of already converged rows and w be the weight vector corresponding to the output node m. Then from figure 1, we have the following equations for neural update rule:

$$y = \mathcal{W}^{m-1}x \tag{2a}$$

$$y_m = w^T x - c^T y \tag{2b}$$

Only w and c will be trained in the m^{th} step and $\mathcal{W}^{m-1}$ remains fixed. For iteration k (where we introduce pattern k to the network) in this step we have:

$$w_{k+1} = w_k + \beta_k(y_{mk}x_k - y_{mk}^2 w_k) \tag{3a}$$

$$c_{k+1} = c_k + \beta_k(y_{mk}\mathrm{y}_k - y_{mk}^2 c_k) \tag{3b}$$

where β_k is a *positive sequence* of step sizes. The c weights play the role of orthogonalization by subtracting the first $m-1$ components from the m^{th} neuron, making the output neuron m tend to become orthogonal to all the previous components. The initial w and c vectors were set to some *random values*.

The authors have proved the following theorem:

Theorem 1: Consider the algorithm defined in (3). Then, subject to the following conditions and at step m we have $w_k \to e_m$ (or $-e_m$) and $c_k \to 0$ when $k \to \infty$

1. The input sequence $\{x\}$ is at least wide-sense stationary with autocorrelation matrix $\mathcal{R}_x$ whose eigenvalues are **distinct** and **positive**. Further, we assume that the eigenvalues are arranged in the decreasing order, i.e. $\lambda_1 \geq \ldots \geq \lambda_n$.

2. For the step size parameter β_k we must have: $\sum_{k=0}^{\infty} \beta_k = \infty$ and $\beta_k \to 0$ as $k \to \infty$.

Aside from the above theorem, the authors have also used simulations to show the validity of their algorithm as well as its fast convergence speeds.

4 Future Works

Regarding implementing the code, we need to have some results for the error performance of the proposed algorithm in [2] combined with this new learning method over large networks. In addition, we have to overcome the all-zero solution issue in the second version of the code above.

Another step in future is to implement the ideas mentioned in [3]. More specifically, the idea of using variable learning step-size may help us make our algorithm faster. Implementing APEX is also another subject of future research specially because of its robust nature, i.e. one can add principle components one by one and does not need to know the exact number of principle components in advance.

References

[1] L. Xu, A. Krzyzak, E. Oja, Neural nets for dual subspace pattern recognition method, Int. J. Neur. Syst., Vol. 2, No. 3, 1991, pp. 169-184.

[2] K.R. Kumar, A.H. Salavati and A. Shokrollahi, *Exponential pattern retrieval capacity with non-binary associative memory*, Proc. IEEE Information Theory Workshop, 2011.

[3] S. Y., Kung, K. I. Diamantaras, J. S. Taur, *Adaptive Principal component EXtraction (APEX) and applications*, IEEE Trans. Signal Process., Vol. 42, No. 5, 1994, pp. 1202-1217.