Optimized Extreme Learning Machine for Big Data Applications using Python

Radu Dogaru*, Ioana Dogaru
University "Politehnica" of Bucharest, Natural Computing Laboratory,
Applied Electronics and Information Engineering Dept., Bucharest, Romania
*Corresponding author (E-mail : radu_d@ieee.org)

Abstract— **This paper reports an optimized implementation for the extreme learning machine (ELM). Among several modeling languages, Python with optimized linear algebra library support was found to be the best choice. A novel nonlinear function, the absolute value, is proposed and shown to provide the best performance and speed among other traditional nonlinearities. A fixed point implementation with finite resolution is proposed and evaluated, concluding that best accuracy is maintained for 2 bits quantization in the input layer and 8 bits in the output layer. The optimized ELM with hardware-convenient nonlinearities and finite precision is widely suitable for solving big data problems on various computing platforms such as FPGA, low cost microcontrollers, etc. In terms of speed, our optimized ELM solution outperforms state of the art Python implementations, providing at least 3 times faster training and retrieval on similar computing platforms.**

Keywords—neural networks; extreme learning machine; Python; high performance computing

I. INTRODUCTION

The field of "artificial intelligence" is now flourishing with many applications demanding high speed recognition of various types of data (big data, in most cases images and videos). Such applications are often migrating from standalone PCs to portable and mobile terminals or various internet of things (IoT) platforms. Although today many of the AI applications widely rely on deep-learning [1], one of the major disadvantages for this approach is the difficulty to optimize the model (the proper parameters for each of the many layers) because training of the model is computationally intensive and demanding. An alternative path was recently proposed, where training is concentrated into fast "engines" such as extreme learning machines (ELM) [2][3] or other similar approaches (e.g. the FSVC or SFSVC in [4]) while improvements of the performance are obtained by including several pre-processor layers based on the local receptive field (LRF) concepts [5][6][7]. This approach allows much faster training at comparable accuracies with deep-learning solutions. Fast training is particularly necessary when dealing with multi-parameter models, in order to speed-up the choice of the best model. Here we focus on the ELM model and investigated how we can effectively get the best performance in a practical implementation from this paradigm. Moreover we provide a solution that can be effectively implemented in a variety of platforms including FPGA or microcontrollers. Section II is a reminder of the ELM model while most of our results are exposed in Section III. Concluding remarks are summarized in Section IV showing that ELM with a very simple nonlinearity (absolute value) implemented in Python based on highly optimized Math libraries (such as Intel MKL) with reduced size (float32) or even low bit resolution fixed-point implementation provides a very efficient and easy to implement solution for a wide variety of big-data applications.

II. EXTREME LEARNING MACHINE MODEL

Briefly speaking, the extreme learning machine (ELM) [3] is a single layer perceptron with a hidden layer formed of weighted sum neurons. Although its authors emphasize the ELM concept as a very original one, it can be identified, as indicated in some of our previous works [4], just like a particular case of a kernel network fulfilling the Cover's theorem [8]. Such networks (which also include the FSVC or SFSVC [4], with an optimized implementation described in a companion paper) rely on simple methods to design the nonlinear hidden layer while training is only done in the output, linear layer. At least three training methods can be used in the linear layer, namely: i) Adaline or LMS training [9]; ii) pseudo-inverse or ridge regressor techniques as in the ELM literature; iii) linear SVM training. In such networks the first (hidden) layer using *m* neurons with a certain nonlinearity output $y = f(x)$ represents the non-linear preprocessor used to expand (since *m*>*n*) the input samples (grouped in the *n x N* matrix **Samples**) into a *m x N* hidden layer matrix **H=*f* (inW*Samples)**. The *m x n* matrix **inW** contains the parameters of the hidden layer. In the ELM framework, **inW** is simply (and fast) produced by a random number generator. Consequently, in ELM training we have to discuss about several *trials* each corresponding to a random computation of inW. Only the result giving the best accuracy is kept. Herein 5 trials were used in each training experiment. The output linear layer (Adaline) is trained with the pseudo-inverse method resulting in a *m x M* matrix **outW**. The output scores are simply computed as a matrix multiplication: **scores**=H^T**outW** and for each line of the scores matrix, the column indicating the largest score will also indicate the recognized class associated with the corresponding input sample. As explained above, other training methods can be used as well but given a computational platform with a highly optimized linear algebra

Reprint: To be published in the Proceedings of COMM2018 (The 12th International Communication Conference), Bucharest, Romania, 14-16 June 2018. https://www.comms.ro/program_comm2018.pdf

library, this is the best choice. The training time is O(m^2), quite large in the case of many big data applications requiring many hidden layer neurons. Particularly, training is done for the entire block of data using the following formulae: $(\mathbf{I}/C+\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{T}$ [6] where C is a regularization parameter, **I** is the unity matrix of size mxm and **T** is the "target" matrix containing the desired outputs for all samples in the training set. Consequently the implementation of ELM training (focusing only on the linear layer) is rather simple in any programming environment where a linear algebra math library is available. Herein we used the following Matlab ™ (works in Octave as well) line:

```
outW = (eye(m)/C + H * H') \ H * T';
```

For the Python implementations we derived the following line based on functions available in both NUMPY (np) and SCIPY packages:

```
outW = scipy.linalg.solve(np.eye(m)/C+np.dot(H,np.transpose(H)), np.dot(H,np.transpose(T)))
```

III. Results and Various Aspects of Optimizing the ELM

The main platform used to test our optimized implementation is a medium cost PC with Intel® Core™ I3-7100 CPU @ 3.9Ghz (2-cores), 8Gbytes of RAM, Windows10 operating system. The dataset used if not otherwise specified is MNIST[1] with N=60000 training and 10000 test samples, each sample being a digit image with n=784 pixels. This dataset is often used in other works and is a good reference for comparisons.

A. Math libraries and data types

As shown previously [10], in implementing algorithms dealing with linear algebra it is quite important to make use of highly optimized libraries such as Intel's MKL. Therefore one should check the availability of such libraries on their computational platform. Herein we consider two "open" license programming environments (Octave[2] and Python[3]) as well as a commercial license language (Matlab[4]), all coming with support from Intel MKL. In terms of data-types used for representing the variables, based on results in [10] the 32 bit floating point was chosen since it provides a significant (about 4 times) speed-up compared to the default "float64" data type. Since weight parameters are the result of a training process, this choice has no influence on the overall accuracy performance. As shown in [4] and confirmed herein, Intel MKL which is highly optimized to CPU architectures and array instructions, successfully compete with low to medium cost GPUs in terms of speed.

B. Influence of various modelling and simulation languages

To avoid clutter the 3 languages considered next are denoted as follows: Matlab R2008b= (M), Octave 4.2.1 = (O) and Python = (Py). Anaconda 5.0 distribution for Python 3.6 with NUMPY and SCIPY was used herein. Results indicated in Table I are the best in 5 trials (random choices for input layer) for each experiment. Training and testing times are averaged for all trials. The implementations (O) and (M) use the same ELM code. The Python code resulted from the Matlab one applying standard rules for Matlab to Python conversions. While Octave and Matlab exhibit similar performance (with crashes for Octave when using large numbers of neurons) it is clear that the "pseudo-inverse" implementation performs much better in Python (accepting float32 representation). Using "single" type (equivalent to float32) in (M) and (O) gives warning messages and increased computing times. For the retrieval phase the Python implementation allows 2-3 times speedups when compared to (O,M) implementations.

TABLE I. Influence of the software platform on the ELM performance (O = Octave; M= Matlab; Py = Python)

M (hidden neurons)		1000	4000	6000
Train time (s)	O	10.8	106	Fails
	M	10.07	103.5	403
	Py	**1.44**	**10 .5**	**21.7**
Test time (s)	O	0.36	1.46	Fails
	M	0.32	1.4	3
	Py	**0.19**	**0.69**	**1.05**

C. Nonlinearity of the hidden neurons

Various nonlinearities were proposed in the literature [2][3][11] since by its nature the kernel-network denoted as ELM allows any desired nonlinear function in the hidden layer with no dramatic influence (as in the case of back-propagation). Herein we choose the following 4 types of nonlinearities (denoted as numbers in parentheses): (0)– **hyperbolic tangent**: i.e. $y = \tanh(x)$; (1)–**"linsat"** (a piecewise linear approximation of the hyperbolic tangent: $y = 0.5(|x+1| - |x-1|)$ with the advantage of a simplified implementation in low-cost platforms; (2)- **"ReLU"** function – widely considered in deep-learning models as giving a very good performance over other kind of nonlinearities: $y = x + |x|$; (3)- proposed by us here – **the absolute value** – in fact a reduced version of the ReLU function having the advantage of compactness: $y = |x|$. Note that among the 4 types used herein, the last 3 have also the advantage of simple implementations, particularly useful for low-cost embedded platforms. Moreover, we shall observe that the absolute value type (3) is closer to the "multiquadric" nonlinearity defined as $y = \sqrt{1+x^2}$. This nonlinearity provides very good accuracies among other nonlinearities in ELM implementations such [11], but is computationally is more demanding than the simple absolute value type (3).

Table II presents results obtained with our Python ELM implementation for different nonlinearities (5 trials were used and results averaged – accuracy was chosen the best among the trials). The problem considered was MNIST.

[1] http://yann.lecun.com/exdb/mnist/
[2] https://www.gnu.org/software/octave/
[3] https://doingmathwithpython.github.io/anaconda-5.0.html
[4] https://www.mathworks.com/products/matlab.html

TABLE II. INFLUENCE OF NONLINEARITY FOR OPTIMIZED ELM (C=100)

N (neurons)		1000	2000	5000	10000
0 Tanh	Train	2.25	4.9	20	80.16
	Test	0.3	0.56	1.5	4.04
	Acc (%)	92.2	94.5	96	96.8
1 Linsat	Train	1.77	4.4	19	710 !
	Test	0.25	0.49	1.2	4.12
	Acc (%)	92.2	94.1	96	96.87
2 ReLU	Train	1.57	3.8	17	109
	Test	0.18	0.38	1	3.51
	Acc (%)	93.7	95.5	97	97.46
3 Abs	Train	**1.46**	**3.66**	**16.4**	**83**
	Test	**0.17**	**0.34**	**0.85**	**3.12**
	Acc (%)	**94.2**	**95.58**	**97.06**	**97.66**

To compare various implementations, in [10] we introduced the synapse time (defined as the ratio between the time to compute all synapses – i.e. the test time above – to the total number of synapses involved for a given number of neurons and test samples). Here the type (3) ELM obtained a very good time: t_syn = 0.85/(794*5000*10000)= 0.021ns, outperforming the 0.13ns obtained in [10] when using a low cost GPU (GeForce GTX 750) .

D. Quantization of weigth parameters

For implementations of ELM into computational platforms such as microcontrollers, FPGA or even dedicated hardware, it is important to know what are the effects of quantizing the weight parameters, for both the input and output layer. Herein we consider a fixed-point implementation of weights and consequently define *ni* as the number of bits for the input layer and *no* as the number of bits to represent the output layer. The quantization of weights is done in two modes:

i) **For both layers after ELM training using "float32" representation**. Each resulting parameter *w* (in both input and output layer) is then changed to $wq = round\left(w \cdot \left(-1 + 2^{nb-1}\right)\right)$. Table III presents the dependency of accuracy (expressed in %) on the number of bits for various *m* neurons in the hidden layer. The MNIST dataset was considered with type (3) nonlinearity (absolute value) and C=100. Accuracies are selected the best among 5 trials for the same setup.

TABLE III. INFLUENCE ON ELM PERFORMANCE FOR FIXED-POINT REPRESENTATIONS WITH FINITE NUMBER OF BITS *NI* ON THE INPUT LAYER

ni	2	3	4	5	6	7	float32
3000 on hidden	78.3	95.2	96.1	96.3	96.27	96.3	96.3
5000 on hidden	72.6	95.3	96.62	96.86	96.94	96.88	97.01

It is seen from the table that 5 or 6 bits suffice to get the same accuracy as in the case of using the floating point representation. A dramatic decrease in performance is observed only for 2 bit quantization. The influence of quantization for the output layer is depicted in the next table:

TABLE IV. INFLUENCE ON ELM ACCURACY FOR FIXED-POINT REPRESENTATIONS WITH FINITE NUMBER OF BITS *NO* ON THE OUTPUT LAYER.

no	3	4	5	6	7	8	float32
5000 on hidden	10	62.7	92.95	96.66	96.8	97.02	97.01

In this case, a larger *no* is needed (one explanation stands in the non-uniform distribution of weight parameters for the output layer) but still it is clear that *no=8* bits for the output layer suffices in order to have no compromise on accuracy. These results are quite encouraging since they prove that fixed-point implementations with 8 bits fixed-point representations provide maximum accuracy.

ii) **The quantization of the input layer weights is done now directly**, when the weights are generated. The output weights are quantified after learning, as in the first case. As shown in Table V, now only 2 bits suffice to represent weight parameters in the input layer, and the same accuracy as in the reference case (using float32 bit representation) is achieved with *no=8* bit. It is an important result, showing that implementation complexity can be dramatically reduced for the input layer, which is the most demanding in terms of computational resources. The problem considered herein is from the UCI Machine Learning Repository[5] with samples representing handwritten digits with 8x8=64 pixel resolution. But the above conclusion stands for any other problem we have considered, including MNIST.

TABLE V. EFFECTS OF INPUT LAYER QUANTIZATION ON THE ACCURACY OF THE ELM WITH M=800 NEURONS TYPE 3 NONLINEARITY AND C=100 FOR THE OPTD64 PROBLEM.

ni (bits)	**2**	**3**	**4**	**5**	**float32**
Accuracy for no=float	98.16	98.16	98.1	98.1	98.1
no (bits)	**6**	**7**	**8**	**9**	**float32**
Accuracy for ni =2	97.77	98.34	98.66	98.44	98.44

E. Comparison with other ELM implementations

We browsed the existent, publicly available software packages for ELM implementations, and found that no one uses the 32 bit representation, nor absolute value (type 3 nonlinearity), issues providing a high efficiency and good performance as indicated above. Finally we choose the ELM implementation from [11] for comparisons. Among the nonlinear functions, the multiquadric function in [11] was chosen since it is the closest to resemble the "type 3" shown above as giving the best performances. Under the same conditions (computing platform, problem, ELM parameters) it is clear that our approach is not only faster (at least 3 times faster for both training and retrieval phases) but also leads to a slightly better accuracy. Moreover, for large number of hidden layer neurons, the learning time becomes excessive in [11] as shown in Table VI for 5000 neurons where the training time becomes almost 20 times larger than in our case.

[5] http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

TABLE VI. COMPARISON BETWEEN OUR ELM AND THE ONE IN [LAMBERT] FOR SIMILAR CONDITIONS

m **(hidden neurons)**	**ELM implement**	**Train time (s)**	**Test time (s)**	**Accuracy (%)**
2000	Ours	3.66	0.34	95.58
	[11]	10.65	1.21	95.4
5000	Ours	16.4	0.85	97.06
	[11]	316	3	96.93

Another work [12] reports on HPELM, a Python toolkit. For the same MNIST dataset, they provide in Figure 9 of [12] training and testing time performances for their toolkit. We consider here the comparison for CPU (they used a better, 4-core CPU running at 4GHz, while ours has only 2 cores). For MNIST with m=4096 neurons their training time was of around 17 seconds and the retrieval time 3 seconds. As seen from the Table VI, for more neurons (m=5000) our implementation gives better times (16.4 seconds for training and only 0.85 seconds for testing). From Table I, with m=4000 neurons only 10 seconds are required for training and 0.69 seconds for retrieval.

IV. CONCLUDING REMARKS

The above results can be summarized as follows: i) Python with optimized Math library (e.g. Intel MKL) from NUMPY provides the best speed for both training and testing. Particularly, the solving of the "pseudo-inverse" using 32 bits seems to work better than in Octave/Matlab where numerous numerical errors and increasing computation times are observed; ii) Absolute value nonlinearity provides the best accuracy for a given number of neurons on the hidden layer and consequently the best speed in both training and retrieval; iii) The type (3) nonlinearity also provides a convenient way to quantify the weight parameters as fixed-point integers, favoring compact ELM implementations in portable and low-cost platforms. It is shown that 2 bits suffices to quantize input layer weights without sacrificing the overall performance. Similarly at most 8 bits fixed-point for the output layer are necessary to get the best possible accuracy for a given problem.

It is interesting to see what are the possibilities to use ELM for various demanding tasks such as recognizing various kind of images. The usual approach as shown in [5][6] is to employ additional local-receptive fields (LRF) layers preceding the ELM and constructed rather than learned, as it is the case in the deep-learning approaches. At the time of writing this paper we investigated both solutions reported above and concluded that the one in [5] works better, at least for the MNIST problem. Still some speed optimizations of these methods are necessary, or other alternative LRF methods should be investigated since their speed is generally much lower than the effective speed for recognition in the ELM trained layer. Using an Octave adapted code for the LRF filters (patches selected randomly from the training set) in [5] with the following parameters (filter size W=7; pooling size Q=8; pooling stride=6, number of filters per class=3) took 29 seconds to run on the entire set of 10000 test samples in MNIST. The ELM accuracy, when applied to the resulting n=480-sized feature vectors obtained from MNIST dataset (with C=0.01, type (3), m=3000) was 99.04% after 6.67 seconds of training and 0.46 seconds for retrieval (less than 1.5% of the time needed to filter the original images). Better accuracy of 99.15% was achieved using m=4000 neurons with training time =10 seconds and test time =0.58 seconds. Consequently, the proposed ELM implementation is definitely fast and accurate to solve any real world problem while similar approaches to optimize the additional LRF layers would provide important speed gains for deep structures using ELM engines as the main training stage. The reported ELM implementation is made available at [13]

REFERENCES

[1] G. E. Hinton, S. Osindero and Y. W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," in Neural Computation, vol. 18, no. 7, pp. 1527-1554, July 2006.

[2] G.-B. Huang, Q.-Y. Zhu and C.-K. Siew, "Extreme Learning Machine: Theory and Applications", Neurocomputing, vol. 70, pp. 489-501, 2006.

[3] G.-B. Huang, What are extreme learning machines? filling the gap between frank rosenblatts dream and john von neumanns puzzle, Cognitive Computation 7 (3) (2015) 263–278.

[4] R. Dogaru and I. Dogaru, "A super fast vector support classifier using novelty detection and no synaptic tuning," 2016 International Conference on Communications (COMM), Bucharest, 2016, pp. 373-376.

[5] M. D. McDonnell and T. Vladusich, "Enhanced image classification with a fast-learning shallow convolutional neural network," 2015 International Joint Conference on Neural Networks (IJCNN), Killarney, 2015, pp. 1-7.

[6] G. B. Huang, Z. Bai, L. L. C. Kasun and C. M. Vong, "Local Receptive Fields Based Extreme Learning Machine," in IEEE Computational Intelligence Magazine, vol. 10, no. 2, pp. 18-29, May 2015.

[7] W. Cao, X. Wang, Z. Ming, J. Gao, "A Review on Neural Networks with Random Weights", Neurocomputing, Volume 275, January 2018, pp. 278-287.

[8] T. M. Cover, "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition," in IEEE Transactions on Electronic Computers, vol. EC-14, no. 3, pp. 326-334, June 1965.

[9] B. Widrow, A. Greenblatt, Y. Kim, D. Park, "The No-Prop algorithm: A new learning algorithm for multilayer neural networks", Neural Networks, Volume 37, January 2013, Pages 182-188.

[10] R. Dogaru and I. Dogaru, "Optimization of GPU and CPU acceleration for neural networks layers implemented in python," 2017 5th International Symposium on Electrical and Electronics Engineering (ISEEE), Galati, 2017, pp. 1-6.

[11] Python implementation of ELM https://github.com/dclambert/Python-ELM

[12] A. Akusok, K. M. Björk, Y. Miche and A. Lendasse, "High-Performance Extreme Learning Machines: A Complete Toolbox for Big Data Applications," in IEEE Access, vol. 3, pp. 1011-1025, 2015.

[13] R. Dogaru, Ioana Dogaru – Fast and efficient ELM implementation in Python – available at: https://github.com/radu-dogaru/ELM-super-fast