High Performance Extreme Learning Machines: A Complete Toolbox for Big Data Applications

Anton Akusok*, Kaj-Mikael Björk†, Yoan Miche‡, Amaury Lendasse*

*Department of Mechanical and Industrial Engineering and the Iowa Informatics Initiative, The University of Iowa, Iowa City, IA 52242-1527, USA.
anton-akusok@uiowa.edu amaury-lendasse@uiowa.edu

†Department of Business Management and Analytics, Arcada University of Applied Sciences, 00550 Helsinki, Finland.
kaj-mikael.bjork@arcada.fi

‡Nokia Solutions and Networks Group, FI-02022 Espoo, Finland.
yoan.miche@nokia.com

Abstract—**This work presents a complete approach to a successful utilization of a high performance Extreme Learning Machines (ELMs) Toolbox[1] for Big Data. It summarizes recent advantages in algorithmic performance; gives a fresh view on the ELM solution in relation to the traditional linear algebraic performance; and reaps the latest software and hardware performance achievements. The results are applicable to a wide range of machine learning problems and thus provide a solid ground for tackling numerous Big Data challenges. The included toolbox is targeted at enabling the full potential of Extreme Learning Machines to the widest range of users.**

1. Introduction

Extreme Learning Machines [1], [2], [3], [4] (ELM) as important emergent machine learning techniques, are proposed for both ”generalized” Single-Layer Feed-forward Networks (SLFNs) [1], [3], [5], [6], [7] and multi layered feedforward networks [6]. Unlike traditional learning theories and learning algorithms, ELM theories show that hidden neurons need not be tuned in learning and their parameters can be independent of the training data, but nevertheless ELMs have universal approximation and classification properties [5], [6], [7]. In most cases, the ELM hidden neurons can be randomly generated, which means that all the parameters of the hidden neurons (e.g., the input weights and biases of additive neurons, the centres and the impact factors of RBF nodes, frequencies and the shift of Fourier series, etc) can be randomly generated and therefore also independent of the training data. Some related efforts had been attempted before [8], [9], [10] with parts of SLFN generated randomly or taken from a subset of data samples [11], however, they either lack proof of the universal approximation capability for fully randomized hidden neurons, or can be considered as specific cases of ELM [12].

ELM, consisting of a wide type of feed forward neural networks, is the first method [6], [7], which can universally approximate any continuous function with almost any nonlinear and piecewise continuous hidden neurons.

A distinct property of ELM is the non-iterative linear solution for the output weights, which is possible because there is no dependence between the input and output weights like in the Back-propagation [13] training procedure. A non-iterative solution of ELMs provides a speedup of 5 orders of magnitude compared to Multilayer Perceptron [14] (MLP) or 6 orders of magnitude compared to Support Vector Machines [15] (SVM), as shown in the experimental section.

ELM originally belongs to the set of regression methods [1], [16]. The universal approximation property implies that an ELM can solve any regression problem with a desired accuracy, if it has enough hidden neurons and training data to learn parameters for all the hidden neurons. ELMs are also easily adapted for classification problems [3]. For multiclass classification, the index of the output node with the highest output indicates the predicted label of input. Then the predicted class is assigned by the maximum output of an ELM. Multi-label classification [17] is handled similarly, but the predicted classes are assigned by all outputs, which are greater than some threshold value.

Extreme Learning Machines are well suited for solving Big Data [18] problems because their solution is so rapidly obtained. Indeed, they are used for analyzing Big Data [19], [20], [21], [22]. But only two ELM toolboxes [23], [24] of all[2] available can process a dataset larger than a given computer memory, and they both implement a particular method rather than focus on overall ELM performance. A GPU acceleration [25], [26] speeds up the computation significantly, but there is no ready to use implementation

1. Download from https://pypi.python.org/pypi/hpelm or install from terminal: `pip install hpelm`

2. http://www.ntu.edu.sg/home/egbhuang/elm_codes.html

before the current work in this article.

Extreme Learning Machines also benefit greatly from model structure selection and regularization, which reduces the negative effects of random initialization and over-fitting. The methods include L^1 [27], [28] and L^2 [29] regularization, as well as other methods [30] like handling imbalance classification [31]. The problem is again the absence of ready to use toolboxes, which are focused on particular existing methods [28]. One reason for this is found in the fact that these methods are challenging to implement since they are typically computationally intensive and are not well suitable for Big Data.

The goal of this work is to approach the vast field of Extreme Learning Machines from a practical performance point of view, and to provide an efficient and easy toolbox, which saves time of researchers and data analysts desiring to apply ELM to their existing problems. An analysis of training methods is done in this piece of software, to select the fastest, least bounded by memory, scalable and simplest way of training ELMs. An efficient implementation is created which suits even old machines of low performance, and the software also handles Big Data on modern workstations with accelerators. The proposed toolbox includes all major model structure selection options and regularization methods, tools for Big Data pre-processing and parallel computing. In the next two sections we explain theoretical and practical aspects of the ELMs methodology. Section 4 explains the actual ELM toolbox, and section 5 compares and discusses on the toolbox performance on various datasets, including test sets of Big Data.

2. Extreme Learning Machines Methodology

2.1. ELM model

An ELM is a fast training method for SLFN networks (Figure 1). A SLFN has three layers of neurons, but the name *Single* comes from the only layer of non-linear neurons in the model: the hidden layer. Input layer provides data features and performs no computations, while an output layer is linear without a transformation function and without bias.

In the ELM method, input layer weights $\mathbf{W}$ and biases $\mathbf{b}$ are set randomly and never adjusted (random distribution of the weights is discussed in section 3.1). Because the input weights are fixed, the output weights $\boldsymbol{\beta}$ are independent of them (unlike in Back-propagation [13] training method) and have a direct solution without iteration. For a linear output layer, such solution is also linear and very fast to compute.

Random input layer weights improve the generalization properties of the solution of a linear output layer, because they produce almost orthogonal (weakly correlated) hidden layer features. The solution of a linear system is always in a span of inputs. If the range of solution weights is limited, orthogonal inputs provide a larger solution space volume with these constrained weights. Small norms of the weights tend to make the system more stable and noise resistant as errors in input will not be amplified in the output of the linear system with smaller coefficients. Thus random hidden layer generates weakly correlated hidden layer features, which allow for a solution with a small norm and a good generalization performance.

A formal description of an ELM is following. Consider a set of N distinct training samples $(\mathbf{x}_i, \mathbf{t}_i),\ i \in [\![1, N]\!]$ with $\mathbf{x}_i \in \mathbb{R}^d$ and $\mathbf{t}_i \in \mathbb{R}^c$. Then a SLFN with L hidden neurons has the following output equation:

$$\sum_{j=1}^{L} \boldsymbol{\beta}_j \phi(\mathbf{w}_j \mathbf{x}_i + b_j),\ i \in [\![1, N]\!], \qquad (1)$$

with ϕ being the activation function (a sigmoid function is a common choice, but other activation functions are possible including linear) [3], [6], [7], $\mathbf{w}_i$ the input weights, b_i the biases and $\boldsymbol{\beta}_i$ the output weights.

The relation between inputs $\mathbf{x}_i$ of the network, target outputs $\mathbf{t}_i$ and estimated outputs $\mathbf{y}_i$ is:

$$\mathbf{y}_i = \sum_{j=1}^{L} \boldsymbol{\beta}_j \phi(\mathbf{w}_j \mathbf{x}_i + b_j) = \mathbf{t}_i + \epsilon_i,\ i \in [\![1, N]\!], \qquad (2)$$

where ϵ is noise. Here the *noise* includes both random noise and dependency on variables not presented in the inputs $\mathbf{X}$.

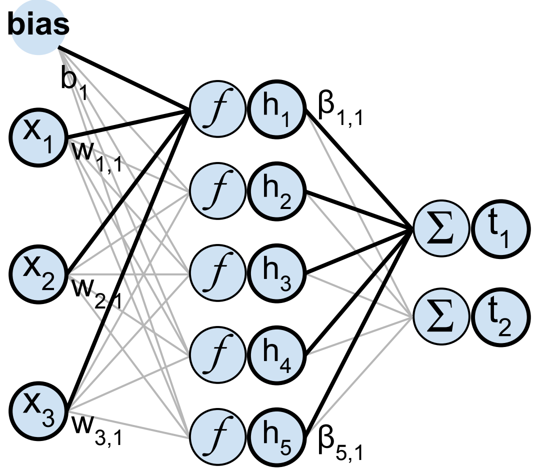


Figure 1: Computing the output of an SLFN (ELM) model.

2.2. Hidden Neurons

Hidden neurons transform the input data into a different representation. The transformation is done in two steps. First, the data is projected into the hidden layer using the input layer weights and biases. Second, the projected data is transformed. A non-linear transformation function greatly increases the learning capabilities of an ELM, because it is the only place where a non-linear part can be added in ELM method. After transformation, the data in the hidden layer representation $\mathbf{h}$ (see Figure 1) is used for finding output layer weights.

The hidden layer is not constrained to have only one type of transformation function in neurons. Different functions can be used (sigmoid, hyperbolic tangent, threshold, etc.) [3], [6], [7]. Some neurons may have no transformation function at all. They are linear neurons, and learn linear dependencies between data features and targets directly, without approximating them by a non-linear function. Usually the number of linear neurons equals the number of data features, and each of these neurons copy the corresponding feature (by using an identity $\mathbf{W}$ and zero $\mathbf{b}$).

Another type of neurons commonly present in ELMs is the Radial Basis Function (RBF) neurons [32]. They use distances to centroids as inputs to the hidden layer, instead of a linear projections. The non-linear projection function is applied as usual. ELMs with RBF neurons compute predictions based on similar training data samples, which helps solving tasks with a complex dependency between data features and targets. Any function (norm) of distances between samples and centroids can be used, for instance L^2, L^1 or L^∞ norms.

2.3. Matrix Form of ELMs

Practically, ELMs are often solved in a matrix form by a closed form solution. An implementation with matrices is easy to write and fast to run on computers. An ELM is written in a matrix form by gathering outputs of all hidden neurons into a matrix $\mathbf{H}$ as on equation 3. A graphical representation is shown in Figure 2. The matrix form of ELMs is used in the paper hereafter.

$$\mathbf{H} = \begin{bmatrix} \phi(\mathbf{w}_1\mathbf{x}_1 + b_1) & \cdots & \phi(\mathbf{w}_L\mathbf{x}_1 + b_L) \\ \vdots & \ddots & \vdots \\ \phi(\mathbf{w}_1\mathbf{x}_N + b_1) & \cdots & \phi(\mathbf{w}_L\mathbf{x}_N + b_L) \end{bmatrix}, \quad (3)$$

$$\boldsymbol{\beta} = \left(\boldsymbol{\beta}_1^T \cdots \boldsymbol{\beta}_L^T\right)^T, \qquad \mathbf{T} = \left(\mathbf{y}_1^T \cdots \mathbf{y}_N^T\right)^T. \quad (4)$$

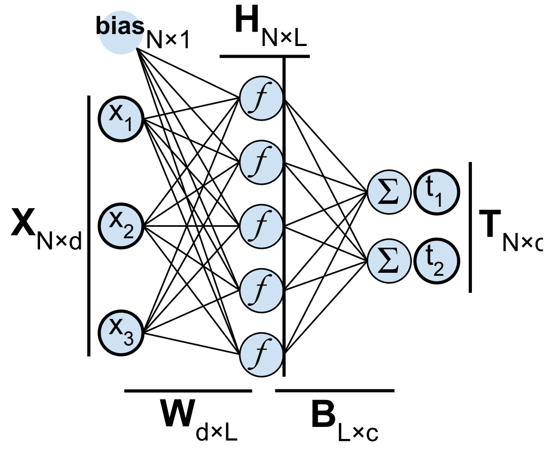


Figure 2: A matrix form of an ELM.

Although the ELM procedure include a training aspect, like other neural networks, the network structure itself is not noticeable in practice. Mathematically, there is only a matrix describing the projection between the two linear spaces. Thus an ELM is viewed as two projections: input $\mathbf{XW}$ and output $\mathbf{H}\boldsymbol{\beta}$, with a (non-linear) transformation between them $\mathbf{H} = \phi(\mathbf{XW} + \mathbf{b})$. The number of hidden neurons regulates the size of matrices $\mathbf{W}$, $\mathbf{H}$ and $\boldsymbol{\beta}$; but the network neurons are never treated separately.

With different types of hidden neurons, the first projection and transformation are performed independently for each type of neurons. Then the resulted sub-matrices $\mathbf{H}_1$ are concatenated along the second dimension. For two types of hidden neurons:

$$\mathbf{H} = [\mathbf{H}_1 \mid \mathbf{H}_2] = [\phi_1(\mathbf{XW}_1 + \mathbf{b}_1) \mid \phi_2(\mathbf{XW}_2 + \mathbf{b}_2)]. \quad (5)$$

Linear neurons are added into ELM by simply copying inputs into the hidden layer outputs:

$$\mathbf{H} = [\mathbf{H}_1 \mid \mathbf{H}_2 \mid \mathbf{X}] = [\phi_1(\mathbf{XW}_1 + \mathbf{b}_1) \mid \phi_2(\mathbf{XW}_2 + \mathbf{b}_2) \mid \mathbf{X}]. \quad (6)$$

2.4. ELM Solution with Pseudo-inverse

Most often, an ELM problem is over-determined ($N > L$), with the number of training data samples larger than the number of hidden neurons. For determined ($N = L$) and under-determined ($N < L$) instances, ELM should use regularization [3]. Otherwise it has a poor generalization performance.

A unique solution for an over-determined system is given by a minimum L_2 norm of the training error. It may be found using the Moore-Penrose generalized inverse [33] (pseudoinverse) of the matrix $\mathbf{H}$, denoted as $\mathbf{H}^\dagger$. As the matrix $\mathbf{H}$ has a full column rank, the pseudoinverse is computed as in equation (9).

$$\begin{aligned} \mathbf{H}\boldsymbol{\beta} &= \mathbf{T} & (7) \\ \boldsymbol{\beta} &= \mathbf{H}^\dagger \mathbf{T} & (8) \\ \mathbf{H}^\dagger &= (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T, & (9) \end{aligned}$$

The pseudoinverse is prone to numerical instabilities if the matrix $\mathbf{H}^T\mathbf{H}$ is close to singular. Practically (in Matlab® and Python), the implementations of the pseudoinverse include a small regularization term $\mathbf{H}^\dagger = (\mathbf{H}^T\mathbf{H} + \alpha\mathbf{I})\mathbf{H}^T$ where $\alpha = 50\epsilon$ and ϵ is the machine precision for a used type of floating point numbers. Adding a regularization term makes matrix $\mathbf{H}^T\mathbf{H}$ non-singular, and the same solution applicable also for determined and under-determined systems.

2.5. Classification with ELMs

An ELM is a regression model, but it is easily adapted for classification. To classify a dataset with ELM, data targets need to be set in a special encoding manner.

If the classes are categorical and independent, then one target is created for each class. Targets for the correct classes are set to one, and targets for irrelevant classes are set to zero. This encoding creates a unit length vector for each class, which is orthogonal to vectors of all other classes. Distances between target vectors of different classes are the same, so the class independence is kept. The predicted class is assigned according to the target with the largest ELM output.

If the classes are ordinal and have a ranking, then they are translated into real numbers. Only one target is created for all the classes, and a predicted class is the one with the closest number to an ELM output.

In a multi-label problem, a sample can have multiple correct classes. The targets are created similarly as in the independent classes problem formulation. The predicted classes are assigned for all ELM outputs greater than a threshold value.

Using ELM for classification with independent classes changes the way how the prediction error is calculated. The classification error does not penalize (or encourage) small changes in the ELM output values, which do not lead to a different classification. This makes a difference in the model structure selection (described in section 2.6), where an optimization w.r.t. the MSE regression error finds an incorrect optimal number of hidden neurons, and creates a model with a sub-optimal classification prediction performance.

2.6. Model Structure Selection in ELMs

Model structure selection prevents ELM from learning noise from data and over-fitting. It does so by artificially limiting the learning ability of an ELM. A training dataset has multiple instances of inputs, and the corresponding targets, which are generated by the projected data and an added noise. The *noise* term includes both random noise and projection from features not present in the inputs. Learning particular data samples with the associated noise is called over-fitting. An over-fitted ELM model has worse generalization performance (prediction performance on new data), which can be measured using a validation set of data. A model structure selection process finds an optimal generalization performance by changing the amount of model parameters or applying regularization to the model.

A hyper-parameter of ELMs, which governs the amount of effective parameters, is the number of hidden neurons. The optimum number of neurons is found with a validation set, a cross-validation procedure or a Leave-One-Out validation procedure (which has an efficient solution in ELMs). Hidden neurons can be added and removed randomly, or they can be ranked by their relevance to the problem. This ranking is called "Optimal Pruning" [28] and it achieves better performance with a trade-off of a longer runtime. Neuron pruning methods correspond to L^1-regularization.

Another model structure selection technique available in ELMs is the Tikhonov regularization [34]. It reduces an effective number of model parameters by reducing the influence of neuron outputs without removing neurons by themselves. Tikhonov regularization is efficient for achieving numerical stability in near-singular ELMs (and linear problems in general). This regularization corresponds to L^2-regularization, and can be combined with L^1 to achieve the best results [29].

Model structure selection is less important in Big Data tasks, because with a large number of samples a model learns to ignore noise. Large tasks are often complex enough not to overfit even at the limits of the hardware. Also, most model structure selection methods significantly increase runtime, which is a limiting factor for training large ELM models. For the provided reasons, only one fast neuron pruning method with a validation set is included in the toolbox part for large data.

3. ELMs in Practice

3.1. Data Normalization

Input data normalization is a critical preprocessing step for many Machine Learning methods, including ELMs. Raw data often has features of different scales, for example an age of a man is at a scale 1-100, and his annual salary in dollars is 3 orders of magnitude larger. Without normalization, small relative variations in the salary make large relative variations in the age negligible. Normalization sets all features at the same scale. Then all features have the same influence, and the training method learns which ones to use for the prediction.

In the ELM toolbox, weights can be given explicitly or generated automatically. Automatic weights generation assumes that the data has zero mean and unit variance. The generated weights keep the performance of neural network with sigmoid neurons near the optimum, and compensate for large number of inputs. The explanation and experimental evaluation of the automatic random weights parameters are given in the experimental Section 5.2.

3.2. ELM Solution with Best Linear Unbiased Estimator

The best linear unbiased estimator gives the optimal least squares solution to the matrix equation $\mathbf{X}\boldsymbol{\beta} = \mathbf{T}$ for stochastic vectors $\mathbf{x}$ and $\mathbf{t}$ combined into the corresponding matrices. It uses two theoretical correlation matrices

$$\mathbb{E}[\mathbf{x}^T\mathbf{x}] = \mathbf{C}_{xx},\ \mathbb{E}[\mathbf{x}^T\mathbf{t}] = \mathbf{C}_{xt} \tag{10}$$

which are assumed to be known. The best linear unbiased estimator of $\mathbf{T}$, denoted by $\mathbf{Y}$, is then

$$\mathbf{Y} = \mathbf{C}_{xx}^{-1}\mathbf{C}_{xt}\mathbf{X} = \boldsymbol{\beta}\mathbf{X}. \tag{11}$$

The inverse of $\mathbf{C}_{xx}$ exists because x is a stochastic variable for which $\mathbf{C}_{xx} = \mathbb{E}[x^T x]$ has a full rank.

The ELM problem has a finite amount of projected data samples $\mathbf{H}$ and corresponding targets $\mathbf{T}$, so the correlation matrices are replaced by their estimations

$$\mathbf{C}_{xx} \approx \mathbf{H}^T\mathbf{H} = \boldsymbol{\Omega}_h,\ \mathbf{C}_{xt} \approx \mathbf{H}^T\mathbf{T} = \boldsymbol{\Omega}_t, \tag{12}$$

and the ELM output weights are computed from those estimates

$$\boldsymbol{\beta} = (\mathbf{H}^T\mathbf{H})^{-1}(\mathbf{H}^T\mathbf{T}) = \boldsymbol{\Omega}_h^{-1}\boldsymbol{\Omega}_t. \tag{13}$$

The inverse of $\boldsymbol{\Omega}_h = \mathbf{H}^T\mathbf{H}$ matrix exists if it has full rank. In ELM model, the nonlinear random projection produces almost orthogonal features which are linearly independent. The number of hidden neurons (columns of $\mathbf{H}$) is smaller than the number of training samples (rows of $\mathbf{H}$), otherwise the liner model will learn training samples perfectly and overfit. Under such constraints, the rank of matrix $\mathbf{H}$ equals its number of columns, thus matrix $\mathbf{H}^T\mathbf{H} = \boldsymbol{\Omega}_h$ is full rank and its inverse exists.

3.3. Numerical Stability of an ELM solution with correlation matrices

If numerical instabilities are faced in the inverse, a regularization term is applied to the correlation matrix $\boldsymbol{\Omega}_h = \mathbf{H}^T\mathbf{H} + \alpha\mathbf{I}$, where α is a small positive constant. This approach is called Ridge Regression [35] aka. Tikhonov regularization [34]. A greater than zero parameter α reduces the effective number of variable in the model, increasing the inverse stability but decreasing predictive power. The default Ridge regression is used in all matrix inverse functions of Python (Numpy) and Matlab® with $\alpha = 50\epsilon$ where ϵ is a machine precision constant.

3.4. Out-of-memory Incremental ELMs

ELMs easily run out of memory for storing the matrix $\mathbf{H}$ with large number of data samples and hidden neurons. Previously this problem was tackled by iteratively updating the output weights. However, these methods are computationally slower because they perform updates of large matrices for each data sample [24], or need to calculate a solution repeatedly [3].

An easier way of finding solution of a large ELM is possible using the notations of estimated correlation matrices. It addresses memory limitation by being invariant to the number of training samples. The runtime is virtually the same as for a pseudoinverse solution on a machine with infinite memory. The comparison of computational complexity and memory requirements for the pseudo-inverse versus correlation matrices ELM solutions are presented in Table 1.

A memory requirement of a correlation ELM solution is constant for any number of training samples, because the correlation matrices $\boldsymbol{\Omega}_h$ and $\boldsymbol{\Omega}_t$ can be computed for batches of training data. The batch computation replaces the number of samples N in the memory requirements by a batch size. A good trade-off in terms of memory requirement and computational overhead is achieved with a batch size equal to L. The final $\boldsymbol{\Omega}_h$ and $\boldsymbol{\Omega}_t$ are computed from batches by a simple summation. The summation operation adds to runtime overhead, because in software and hardware implementations, matrix multiplication and summation are performed in a single operation[3]: $\texttt{gemm}(\mathbf{A},\mathbf{B},\mathbf{C}) = \mathbf{AB} + \mathbf{C}$.

TABLE 1: ELM computation and memory requirements; computations along the dimension $\tilde{N}$ can be performed incrementally in L-size batches.

Operation	Comp. complexity	Memory
Projection to hidden layer		
$\mathbf{X}_{N\times d}$	$\mathcal{O}(\tilde{N}d)$	$\mathcal{O}(\tilde{N}d)$
$\mathbf{H}_{N\times L} = f(\mathbf{XW}+\mathbf{b})$	$\mathcal{O}(\tilde{N}Ld + \tilde{N}L)$	$\mathcal{O}(\tilde{N}L)$
Pseudo-inverse solution		
$\mathbf{A}_{L\times N} = \mathbf{H}^\dagger$	$\mathcal{O}(NL^2 + L^3)$	$\mathcal{O}(NL)$
$\boldsymbol{\beta}_{L\times c} = \mathbf{AT}$	$\mathcal{O}(\tilde{N}Lc)$	$\mathcal{O}(\tilde{N}L + \tilde{N}c)$
Correlation matrices solution		
$\mathbf{A}_{L\times L} = \mathbf{H}^T\mathbf{H}$	$\mathcal{O}(\tilde{N}L^2)$	$\mathcal{O}(L^2)$
$\mathbf{B}_{L\times c} = \mathbf{H}^T\mathbf{T}$	$\mathcal{O}(\tilde{N}Lc)$	$\mathcal{O}(Lc)$
$\boldsymbol{\beta}_{L\times c} = \mathbf{A}^{-1}\mathbf{B}$	$\mathcal{O}(L^3 + L^2c)$	$\mathcal{O}(L^2 + 2Lc)$
Comparison of solutions		
$\boldsymbol{\beta}_\dagger = \mathbf{H}^\dagger\mathbf{T}$	$\mathcal{O}(NL^2 + L^3)$	$\mathcal{O}(NL)$
$\boldsymbol{\beta}_{\texttt{corr}} = (\mathbf{H}^T\mathbf{H})^{-1}(\mathbf{H}^T\mathbf{T})$	$\mathcal{O}(\tilde{N}L^2 + L^3)$	$\mathcal{O}(L^2 + 2Lc)$

3.5. Weighted Classification with ELMs

In a classification task with highly uneven number of data samples for different classes, ELM predictions are biased towards the class with the most data. This behaviour is improved by using a weighted linear system solution in the output layer of an ELM [31]. A weighted linear system has a Least Squares solution similar to the BLUE solution:

$$\boldsymbol{\Omega}_h = \mathbf{H}^T\mathbf{A}\mathbf{H},\ \boldsymbol{\Omega}_t = \mathbf{H}^T\mathbf{A}\mathbf{T}, \tag{14}$$

where $\mathbf{A} \in \mathbb{R}^{N\times N}$ is an arbitrary weight matrix. If only sample weights are used, the $\mathbf{A}$ matrix is diagonal; but these weights are complicated to obtain if they are not given explicitly. In a classification task, diagonal elements of $\mathbf{A}$ for all samples of class $j \in [\![1, c]\!]$ are given the same weight a_j

$$a_j = \frac{N}{\sum_{i=1}^{N}\mathbf{T}_{ij}},\ j \in [\![1, c]\!]. \tag{15}$$

The solution of ELM obtained this way is unbiased for any class. An additional multiplication by $\mathbf{A}$ is avoided by applying weights $\sqrt{a_j}$ directly to the rows of matrices $\mathbf{H}, \mathbf{T}$ which correspond to the data samples of a class j.

Alternatively, the correlation matrices can be computed for each class separately $\boldsymbol{\Omega}_h^1, \boldsymbol{\Omega}_t^1,\ \ldots,\ \boldsymbol{\Omega}_h^c, \boldsymbol{\Omega}_t^c$. Then the weights are applied during the summation of the correlation matrices $\boldsymbol{\Omega}_h$ and $\boldsymbol{\Omega}_t$:

$$\boldsymbol{\Omega}_h = \alpha_1\boldsymbol{\Omega}_h^1 + \ldots + \alpha_c\boldsymbol{\Omega}_h^c, \tag{16}$$
$$\boldsymbol{\Omega}_t = \alpha_1\boldsymbol{\Omega}_t^1 + \ldots + \alpha_c\boldsymbol{\Omega}_t^c. \tag{17}$$

3. http://www.netlib.org/blas/#_level_3

4. Toolbox Overview

The HP-ELM toolbox implements the state-of-the-art knowledge in ELMs and high-performance programming. It is written to save the time of end users on creating yet another implementation of ELM, which is better spent on their own research or application area instead.

An ELM is a simple method which can be written in three lines in Matlab®. But performance of such ELMs is sub-optimal. ELMs achieve best accuracy with parameter selection, regularization and pruning for small datasets, and best scalability with out-of-memory accelerated processing on Big Data. The toolbox is written to provide the best performing ELM implementation to all interested researches and end users.

4.1. How to get the toolbox

The toolbox is a Python library, also available from Matlab®. It is written in Python programming language using efficient numerical libraries *Numpy*[4] and *Scipy*[5].

The toolbox requires Python and the following libraries: *Numpy*, *Scipy*, *Numexpr* and *pyTables*[6]. The easiest way to get Python with all required libraries is to use the Anaconda[7] Python distribution. It is a one-click install on Windows/Linux/OSX, free and provides free MKL acceleration to all university affiliates. Any other Python installation will work as well.

To install the toolbox for CPU, open the console and type `pip install hpelm`. This will download and install the toolbox with all required libraries. Anaconda provides a python console on Windows; Linux and OSX have built-in ones.

To obtain an accelerated toolbox, first download and install MAGMA[8] math library for your accelerator: Nvidia GPU with CUDA, AMD GPU with OpenCL or Xeon Phi accelerator card (called MIC architecture). All versions of MAGMA are available from the website; it also has a forum for installation support. To build MAGMA, rename one of the `make.inc.xxxx` files as `make.inc` and edit that file according to your system installation. Then install MAGMA by running `make`, `make shared` and `make install` in console from MAGMA directory.

Second, download the toolbox archive from its repository https://pypi.python.org/pypi/hpelm or the latest version from Github[9], extract it and go to a sub-folder `./hpelm/acc`. There is an accelerated code which must be compiled. To get compilation flags, add your MAGMA library to `pkg-config` path, or use the same flags as MAGMA used to compile its tutorial files during an installation. To compile an accelerated ELM library, run `python setup_gpu.py build_ext --inplace` replacing `_gpu` by `_ocl` for OpenCL MAGMA or `_mic` for Xeon Phi MAGMA. You can test an acceleration by running `pyhton try_gpu.py` from the same folder. After that, go to the root directory of the toolbox and install the now-accelerated toolbox with `python setup.py install`.

4. http://www.numpy.org
5. http://www.scipy.org
6. http://www.pytables.org
7. http://continuum.io/downloads
8. http://icl.cs.utk.edu/magma/
9. https://github.com/akusok/hpelm

4.2. Big Data Versus Small Data

Based on the number of training samples and underlying model complexity, all machine learning tasks can be separated into two categories: *big data* and *small data*. In the *big data*, the number of samples is enough to learn the model accurately without over-fitting, but the training time is a limiting factor. For the *small data*, there is not enough samples for learning an underlying model exactly, thus a model structure selection is necessary to find an optimal model complexity.

Training a *small data* model is computationally intensive, but the whole data is kept in the working memory for quick access. The *big data* training algorithm relies on iterative processing of small chunks of data (which normally does not fit into memory), but with a huge amount of training samples there is no need for a model structure selection process (larger model provides better performance). Processing a *small data* which does not fit into memory is not implemented, because a typical server node has up to 256-512GB RAM, and anything large would certainly be limited by the computational speed. A *big data* for an easy problem which fits into memory is solved by either of the two first methods.

4.3. Out-of-memory Accelerated Big Data ELM

The HP-ELM toolbox for *big data* is provided by the `hpelm.HPELM` class. All data is stored in HDF5[10] format. The toolbox takes names of HDF5 files as inputs and outputs. Thus a size of processed data is limited only by a disk capacity.

The HDF5 file format provides a fast and convenient access to huge data matrices on a hard drive as if they are in memory: data can be read from or written to any place of a matrix. It also supports transparent data compression, and is native to Matlab®. A convention is used to *store only one matrix in one HDF5 file*. Additional utility functions `make_hdf5` and `normalize_hdf5` create HDF5 files from text/csv or matrix data, and normalize these files.

The `HPELM` class supports a GPU or Xeon Phi acceleration. The accelerated functions are provided by MAGMA library, an accelerated linear algebra library similar to LAPACK. It must be compiled by a user to get the acceleration, but it supports any brands of GPUs and Xeon Phi accelerators. Accelerated parts are correlation matrices computation from BLUE ELM solution, and the calculation of $\boldsymbol{\beta}$. These

10. http://www.hdfgroup.org/HDF5/

two operations take more than 95% of runtime for ELMs with very large numbers of hidden neurons.

The ELM solution is computed iteratively by reading chunks of data from HDF5 files. Only the $\mathbf{\Omega}_h$, $\mathbf{\Omega}_t$ and $\boldsymbol{\beta}$ matrices are stored in memory. The large $\mathbf{H}$ matrix is never obtained explicitly. Targets for new inputs are also predicted iteratively and saved into an HDF5 file; and the error is computed iteratively. This makes *Big Data* ELM independent of the number of samples in the dataset, so even the largest problems can be solved on a workstation with GPU.

`HPELM` has one model structure selection function that tests different numbers of hidden neurons on a validation set. It takes pre-computed $\mathbf{\Omega}_h$, $\mathbf{\Omega}_t$ as an input, and creates solutions $\boldsymbol{\beta}_k$ for different $k \in [\![3, L]\!]$ spaced equally on a logarithmic scale. Then the validation data is projected iteratively, and errors for all values of k are computed from the same projected data. This function does the most time consuming process of projecting the data (see section 5.5) only once. The optimal number of hidden neurons is chosen by a minimum validation error.

4.4. Model Structure Selection for Small Data ELM

The *small data* support in the HP-ELM toolbox is provided by the `hpelm.ELM` class. It has three types of model structure selection alternatives: with a validation set, with cross-validation and with a LOO validation error computed by PRESS statistics. All model structure selection methods find an optimal number of hidden neurons less of equal than current L. Neurons are ordered randomly, except when the L^1 regularization is used. These methods remove the extra neurons from the model and re-calculate the solution.

Both L^1 and L^2 regularization are available in ELM. The L^1 regularization is done by MRSR [36], a multi-output version of LARS [37]. It ranks the neurons starting from the most relevant to the problem. All model structure selection methods work better with such ranked neurons, with a trade-off of extra runtime.

The toolbox includes another method of performing L^1-regularization, based on an updated MRSR algorithm [38]. The original MRSR includes a part with $\mathcal{O}(2^c)$ complexity w.r.t. number of outputs c. It takes noticeable runtime with 10 outputs, and makes the method impractically slow with more than 15 outputs. The complexity of an updated version scales linearly with the number of outputs. It allows L^1 regularization for a larger set of problems, including an auto-encoder for ELMs in image processing [22].

L^2 regularization is a class parameter of ELM called `alpha` which can be changed freely. A notable benefit from L^2 regularization is making an ill-conditioned ELM solvable. One can use any single-variable optimization method to find an optimum value of L^2 parameter. The resulted method will be similar to TROP-ELM [29].

4.5. What kind of data does HP-ELM support?

The `ELM` supports matrices (second order tensors) as inputs, and `HPELM` uses names of HDF5 files as inputs. The utility function `make_hdf5` creates an HDF5 file from a matrix, or a text/csv file.

4.6. What about Classification?

The HP-ELM toolbox supports three kinds of classification: multi-class (one correct class for each sample), multi-label (arbitrary number of correct classes for each sample) and weighted multi-class (each class has a weight, it is independent of the number of samples in a class). ELM targets must have one feature per class (binary classification is a two-class multi-class classification), where the true class(es) for a sample are set to one and irrelevant classes are set to zero. This convention is required for correct work of the classification error and model structure selection. Classification is set with an argument while training, see section 4.8 below.

4.7. How to create an ELM?

An ELM is an object of `ELM` or `HPELM` class. Two mandatory parameters are numbers of input and output features. The `HPELM` also accepts a batch size for iterative processing, and a type of accelerator.

An ELM is created without any neurons. Neurons are added with `elm.add_neurons` function. It has two mandatory parameters: a number of neurons and their type, and two optional ones: projection matrix $\mathbf{W}$ and bias vector $\mathbf{b}$. Types of neurons are the following: `lin`, `sigm`, `tanh`, `rbf_l1`, `rbf_l2`, `rbf_linf`. For RBF neurons, $\mathbf{W}$ are coordinates of RBF centers and $\mathbf{b}$ are corresponding kernel widths. Multiple different types of neurons can be added to a single ELM.

4.8. How to train an ELM?

The `train` function provides a universal wrapper for training an ELM. Two mandatory parameters are data samples $\mathbf{X}$ and targets $\mathbf{T}$, and optional arguments and keyword arguments specify the selected way of training:

- `"V"` — perform a model structure selection using validation error; requires keyword arguments `Xv` and `Tv` for validation dataset
- `"CV"` — perform a model structure selection using cross-validation error; optional keyword argument `k` for number of data splits ($k \geq 3$)
- `"LOO"` — perform a model structure selection using PRESS LOO error
- `"OP"` — perform L^1 regularization that ranks neurons starting from the most useful one; works with any model structure selection
- `"c"`, `"mc"`, `"wc"` — use classification multi-class/multi-label/weighted multi-class error instead of MSE, see explanations above; `"wc"` requires keyword argument `w` for class weights vector

4.9. How to train a large ELM in parallel?

For an ELM with a large number of neurons trained on a huge dataset, almost all the running time is spent on computing $\mathbf{\Omega}_h$. Hopefully, an operation of computing $\mathbf{\Omega}_h$ is conveniently parallel: a large dataset can be split in n parts, matrices $\mathbf{\Omega}_h^i, \mathbf{\Omega}_t^i,\ i \in [\![1, n]\!]$ computed simultaneously for all the parts of a dataset using the same ELM parameters (loading the same ELM model). The results are combined together by a simple summation $\mathbf{\Omega}_h = \sum_{i=1}^n \mathbf{\Omega}_h^i,\ \mathbf{\Omega}_t = \sum_{i=1}^n \mathbf{\Omega}_t^i$. The output weights $\boldsymbol{\beta}$ will take seconds to compute.

To perform ELM training in parallel, first split the data into multiple parts and store them in HDF5 format required by `HPELM`. Then compute partial matrices $\mathbf{\Omega}_h^i, \mathbf{\Omega}_t^i$ using function `HPELM._project` on each data part separately. This operation takes the most runtime, and is easy to run in parallel. Save the outputs on a disk as they are computed. When all partial matrices are ready, obtain the final correlation matrices by a summation $\mathbf{\Omega}_h = \sum_i \mathbf{\Omega}_h^i,\ \mathbf{\Omega}_t = \sum_i \mathbf{\Omega}_t^i$. The output weights $\boldsymbol{\beta}$ are computed from $\mathbf{\Omega}_h, \mathbf{\Omega}_t$ by function `HPELM._solve_corr`.

To validate multiple different numbers of hidden neurons efficiently, use function `HPELM.train_hpv` with precomputed $\mathbf{\Omega}_h, \mathbf{\Omega}_t$ and a validation data set. It outputs errors for each of the given numbers of neurons, and solves $\boldsymbol{\beta}$ for an optimal number of neurons.

4.10. How to use a trained ELM?

The `predict` function takes inputs $\mathbf{X}$ and returns corresponding calculated outputs $\mathbf{Y}$. Works only on a trained ELM. For `HPELM`, the second input gives an HDF5 file name for $\mathbf{Y}$ where the predicted outputs are written, and the function returns nothing. ELM predictions $\mathbf{Y}$ are always real numbers, predicted classes are found by taking the maximum number (multi-class) or a threshold $\mathbf{Y} > 0.5$ (multi-label).

4.11. How to get an error of an ELM?

Error of model predictions is given by `error` function of ELM, which takes true targets $\mathbf{T}$ and predicted targets $\mathbf{Y}$ as inputs. It uses the same classification settings as the ones used for training, if any. For `HPELM`, the `error` function takes file names of HDF5 files containing $\mathbf{T}$ and $\mathbf{Y}$ matrices.

4.12. Three examples of HP-ELM toolbox

Below there are three examples of running the `ELM` and `HPELM` toolboxes in Python, with the data obtained from Matlab®. The input data has 9 features and the output has one. Example ELMs using 100 sigmoid and 9 linear neurons are given. If the data is already in Python, one can skip the import from Matlab® section.

Matlab® section for Examples 1 and 2. Here four variables: `x`, `y`, `xtest` and `ytest` are saved as a comma separated values (`.cvs` files).

```
csvwrite('x.csv',x)
csvwrite('t.csv',y)
csvwrite('xtest.csv',xtest)
csvwrite('ttest.csv',ytest)
```

Example 1, Python part. Here the `.csv` files are converted to HDF5 ones in Python, and an `HPELM` is trained with those files. Training and test errors are printed.

```
import hpelm

hpelm.make_hdf5('x.csv', 'x.h5', delimiter=',')
hpelm.make_hdf5('t.csv', 't.h5', delimiter=',')
hpelm.make_hdf5('xtest.csv', 'xtest.h5', delimiter=',')
hpelm.make_hdf5('ttest.csv', 'ttest.h5', delimiter=',')

model=hpelm.HPELM(9,1)
model.add_neurons(100,'sigm')
model.add_neurons(9,'lin)

model.train('x.h5','t.h5')
model.predict('x.h5','y.h5)
print model.error('y.h5','t.h5')

model.predict('xtest.h5','ytest.h5')
print model.error('ytest.h5','ttest.h5)
```

Example 2, Python part. Here the `ELM` model is trained with different model structure selection. Previously created `.csv` files are loaded into Python and normalized to zero mean and unit variance. Then a basic ELM is trained printing the training and test error. After that a 10-fold cross-validation is used to reduce the number of neurons, showing an updated test error and selected neurons in the model. Finally the model is re-trained using an L^1 regularization (`OP` parameter), showing again re-calculated test error and model neurons.

```
import hpelm
import numpy

x=numpy.loadtxt('x.csv',delimiter=',')
t=numpy.loadtxt('t.csv',delimiter=',')
xtest=numpy.loadtxt('xtest.csv',delimiter=',')
ttest=numpy.loadtxt('ttest.csv',delimiter=',')

xx=(x-x.mean(0))/x.std(0)
tt=(t-t.mean(0))/t.std(0)
xxtest=(xtest-x.mean(0))/x.std(0)
tttest=(ttest-t.mean(0))/t.std(0)

model=hpelm.ELM(9,1)
model.add_neurons(100,'sigm')
model.add_neurons(9,'lin')

model.train(xx,tt)
tth=model.predict(xx)
print model.error(tt,tth)
yytest=model.predict(xxtest)
print model.error(yytest,tttest)

model.train(xx,tt,'CV',k=10)
yytest=model.predict(xxtest)
print model.error(yytest,tttest)
print str(model)

model.train(xx,tt,'LOO','OP')
yytest=model.predict(xxtest)
print model.error(yytest,tttest)
print str(model)
```

Matlab® section for Example 3. The training data: `x` and `y` is saved as HDF5 files using build-in Matlab® functions. Note the *transpose operation*, as Matlab® uses Fortran matrix ordering by default for HDF5 files.

```
h5create('x.h5','/data',size(x'));
h5create('t.h5','/data',size(y'));
h5write('x.h5','/data',x');
h5write('t.h5','/data',y');
```

Python part for Example 3. An `HPELM` is built and trained using the HDF5 files created by Matlab®.

```
import hpelm
model=hpelm.HPELM(9,1)
model.add_neurons(100,'sigm')
model.add_neurons(9,'lin')
model.train('x.h5','t.h5')
model.predict('x.h5','y.h5')
print model.error('y.h5','t.h5)
```

4.13. How to use Gaussian (RBF) Neurons?

The ELM toolbox has Gaussian neurons. Centroids are given instead of a projection matrix $\mathbf{W}$ and kernel widths in a bias vector $\mathbf{b}$. There are three kinds of distance functions: L^2 (Euclidean), L^1 and L^∞. They are chosen by a type of neurons: *rbf_l2*, *rbf_l1* or *rbf_linf* correspondingly. The RBF neurons are about 10 times slower to compute than sigmoid ones, even though the computation is parallelized.

4.14. My ELM solution does not exist!

An ELM may not converge if there are a few input features (2-3) with a large number of hidden neurons, if the data features are strongly correlated and not independent, or if the number of data samples is close to the number of hidden neurons. In these cases, matrix $\mathbf{\Omega}_h$ will be almost singular, and it's inverse is numerically unstable.

The numerical stability problem is solved by increasing the value of the L^2 regularization parameter α (an ELM parameter called `alpha`). The default value of $\alpha = 10^{-9}$ can be increased up to 10^{-2} or higher. This reduces the effective number of parameters in the model. The regularization parameter α should be increased if the output matrix $\boldsymbol{\beta}$ has elements with a large magnitude (larger than $10^2 \dots 10^3$). However, it is not worthwhile increasing the parameter excessively, as this may reduce the accuracy of ELM predictions.

5. Experimental Results

5.1. Datasets

The HP-ELM toolbox is tested in three scenarios: regular datasets with regularization, large datasets and a Big Data problem.

Small datasets are 11 regression and 4 classification problems from the University of California at Irvine (UCI) Machine Learning Repository [39]. Ten different permutations of the datasets are taken without replacements, and for each of them 2/3 of the data is used for training and 1/3 for testing. The data permutations are obtained from the an author of the OP-ELM [28] paper exactly as they are used there. Comparison results for Support Vector Machines [40] (SVM), Multilayer Perceptron [41] (MLP), Gaussian Processes [42] (GP) are taken from the same article [28]. Small datasets are tested on ELMs with model structure selection and without.

Large datasets are 6 relatively large datasets, available from UCI Machine Learning repository with clear prediction targets. They are *Banana* dataset of two banana species, *Adult* dataset of people with annual income below/above \$50,000, *MNIST* handwritten digits dataset for classification of 10 digits based on their image representation, Record *Linkage* dataset for detecting duplicate person records with 5.5 millions samples, and *HIGGS* dataset for detecting processes which produce a Higgs boson or not, with 11 million samples (one of the largest UCI datasets available). Each dataset is split into training and test parts (respecting the guidelines where applicable), stored in HDF5 file format and normalized to zero mean and unit variance for all features. Categorical features from *Adult* dataset are encoded as binary inputs (one per each category); these are not normalized. Large datasets are tested without model structure selection, but multiple ELMs are built with different numbers of hidden neurons.

The **Big Data** is obtained from a Face/Skin Detection dataset [43]. It consists of 4000 photos of people with handmade masks for skin and faces, under various lightning conditions, surrounding and human skin colors. Skin occupies roughly 20% of the pixels in all images. The dataset is separated into 2000 training and 2000 test images. The problem is to classify each image pixel to be a skin or a non-skin. Dataset inputs are RGB color values of 7×7 pixel mask centered on a classified pixel. The 3-pixel wide boundaries of images are omitted. There are $7 \times 7 \times 3(\text{RGB}) = 147$ features and 10^9 (one billion) data samples in total. It gives a 1.1 TB dataset in HDF5 format when stored in double precision. Two separate datasets for all training and all test samples are created from training/test images. Data features (color values of pixels) are normalized to zero mean and unit variance. A single ELM is trained with 19,000 neurons, limited by the available GPU memory. Performance is tested on differently sized subsets of these 19,000 neurons, as explained in section 4.9.

5.2. Parameters of automatically generated random weights

Sigmoid function is a common choice of a non-linear transformation function for hidden nodes of ELM. However, it is sensitive to the range of input weights, which are $\mathbf{XW} + \mathbf{b}$. If inputs to the sigmoid function have small magnitude, it performs similarly to linear function. If these inputs have very large magnitude, it performs as a cutoff value. The effect can be seen by checking the difference between predictions of SLFNs with the same weights and sigmoid/linear/threshold transformation functions. If the input data has zero mean and unit variance, the range of inputs to SLFN is governed by the range of weights $\mathbf{W}$ generated from $\mathbf{W} = \mathcal{N}(0, s)$ with different values of standard deviation s. The effect is shown on Figure 3. The range of weights $\mathbf{W}$ also affects the performance, as on Figure 4.

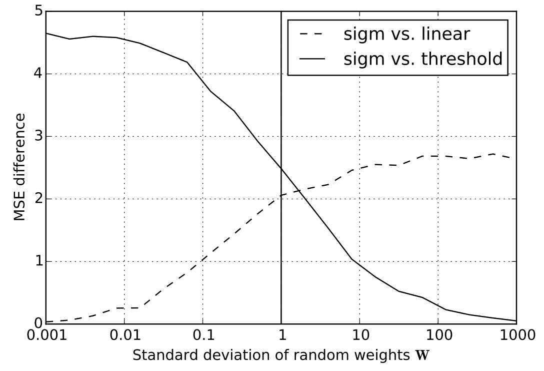


Figure 3: Mean squared error difference of predictions of SLFNs with 5 hidden neurons for Iris dataset (average over 100 runs), for different values of s in $\mathbf{W} = \mathcal{N}(0, s)$. For small s, outputs of sigmoid SLFN are similar to linear SLFN, and for large s they are similar to threshold SLFN.

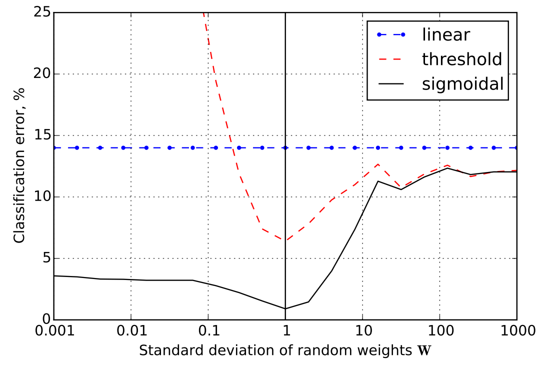


Figure 4: Test error of SLFNs with 25 hidden neurons on Iris dataset (averaged over 100 runs), for different values of s in $\mathbf{W} = \mathcal{N}(0, s)$. The data has 100 training and 50 test samples, balanced over the 3 classes.

Another issue is an increase in standard deviation of inputs to the transformation function, if the dataset has high dimensionality. For a single additive hidden layer neuron, an input to the transformation function $a_k = \sum_{i=1}^{d} x_i w_{i,k}$ is a sum of d components. If a single component has standard deviation s, then that sum has a larger standard deviation $\sqrt{d}s$. This leads to larger magnitudes of inputs to a transformation function and sub-optimal performance for large d, for example in MNIST dataset (see Figure 5). The effect of large input dimensionality is fixed by dividing the standard deviation s by $\sqrt{d}$, and generating weights as $\mathbf{W} = \mathcal{N}(0, s/\sqrt{d})$ (see Figure 6).

In the following experiments, ELM is used with automatically generated weights from $\mathbf{W} = \mathcal{N}(0, s/\sqrt{d})$. The input data is normalized to zero mean and unit variance. Biases are initialized from $\mathcal{N}(0, 1)$.

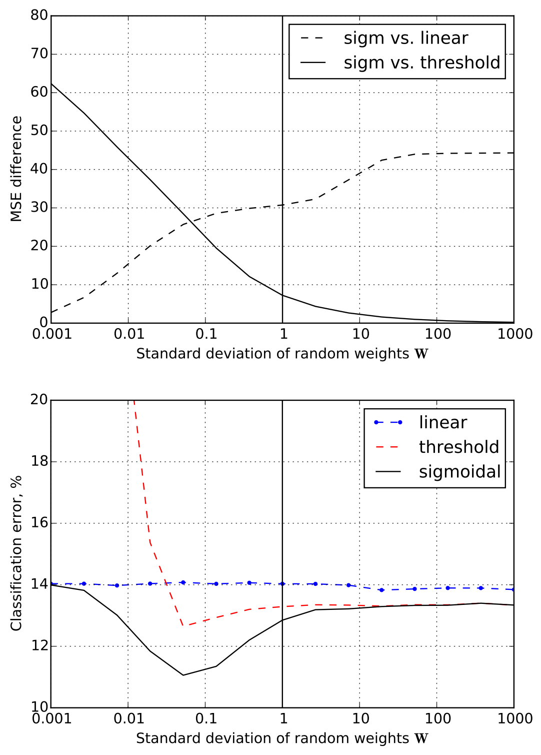


Figure 5: MSE difference (*top*) of predictions, and test error (*bottom*) of SLFNs with 500 hidden neurons on MNIST dataset (averaged over 10 runs), for different values of s in $\mathbf{W} = \mathcal{N}(0, s)$. The data has 60000 training and 10000 test samples. Due to high dimensionality of inputs, the optimal value of s differs from 1.

5.3. Performance on Small Datasets

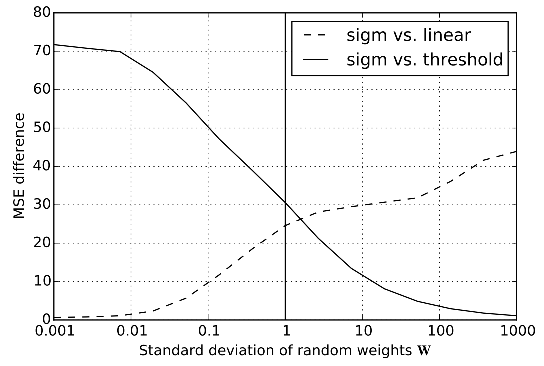
The performance results and runtime for regular size datasets are presented on Tables 2 and 3. Three ELM setups are tested using the toolbox: a basic ELM (*ELM*), an ELM with pruning of hidden neurons (*P-ELM*) using a Leave-One-Out error, and an OP-ELM (*OP-ELM*) which is an L^1 regularized *P-ELM*. The three ELMs are initialized with 100 hidden neurons and sigmoid activation function. The actual number of neurons in P-ELM and OP-ELM is smaller after pruning. In three regression problems the pruning algorithm has selected $> 95\%$ of neurons, pointing to an insufficient model complexity. For these tasks (denoted by an asterisk), the number of neurons is increased to 500 where the pruning algorithm selects $< 90\%$ of neurons on average; the accuracy and runtime for 500 L are reported. Experiments are run on a single 2.6GHz core on a cluster for comparable runtimes.

The MSE and classification performance of the proposed HP-ELM toolbox is consistent with the results of other methods. The basic ELM performs worse in some cases (Auto Price), but *P-ELM* and *OP-ELM* results are comparable to the best result between the other three methods.

Considering runtime, ELM is much faster than other

TABLE 2: Meas Squared Error (bold) and runtime in seconds for the regression datasets. Results denoted by $*$ are computed with 500 hidden neurons, as suggested by pruning.

	Abalone	Ailerons	Elevators	Computer	Auto P.	CPU	Servo	Breast C.	Bank	Stocks	Boston
ELM	**4.6**	**2.9e-8**	**2.1e-6**	**1.4e+1***	**8.4e+9**	**6.8e+4**	**5.6**	**6.3e+3**	**1.1e-3***	**1.1***	**2.2e+1**
	0.07	0.08	0.11	0.18	0.04	0.01	0.02	0.03	0.10	0.02	0.06
P-ELM	**4.6**	**2.9e-8**	**2.1e-6**	**1.3e+1***	**1.5e+7**	**9.5e+3**	**7.3e-1**	**1.2e+3**	**1.1e-3***	**8.1e-1***	**2.1e+1**
	0.12	0.24	0.26	2.5	0.06	0.06	0.08	0.05	1.1	0.10	0.11
OP-ELM	**4.6**	**2.9e-8**	**2.1e-6**	**1.4e+1***	**1.5e+7**	**6.3e+3**	**7.9e-1**	**1.2e+3**	**1.1e-3***	**7.8e-1***	**2.3e+1**
	1.2	1.2	1.2	8.9	0.80	0.78	0.78	0.83	6.4	1.9	0.79
SVM	**4.5**	**1.3e-7**	**6.2e-6**	**1.2e+2**	**2.8e+7**	**6.5e+3**	**6.9e-1**	**1.2e+3**	**2.7e-2**	**5.1e-1**	**3.4e+1**
	6.6e+4	4.2e+2	5.8e+2	3.2e+5	2.6e+2	3.2e+2	1.3e+2	3.2e+2	1.6e+3	2.3e+3	8.5e+2
GP	**4.5**	**2.7e-8**	**2.0e-6**	**7.7**	**2.0e+7**	**6.7e+3**	**4.8e-1**	**1.3e+3**	**8.7e-4**	**4.4e-1**	**1.1e+1**
	9.5e+2	2.9e+3	6.5e+3	6.3e+3	2.9	3.2	2.2	8.8	1.7e+3	4.1e+1	8.5
MLP	**4.6**	**2.7e-7**	**2.6e-6**	**9.8**	**2.2e+7**	**1.4e+4**	**2.2e-1**	**1.5e+3**	**9.1e-4**	**8.8e-1**	**2.2e+1**
	2.1e+3	3.5e+3	3.5e+3	8.2e+3	7.3e+2	5.8e+2	5.2e+2	8.0e+2	2.7e+3	1.2e+3	8.2e+2



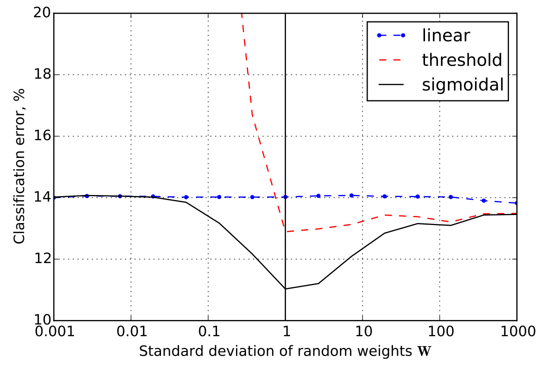


Figure 6: MSE difference (*top*) of predictions, and test error (*bottom*) of SLFNs with 500 hidden neurons on MNIST dataset (averaged over 10 runs), for different values of s. With input dimensionality fix $\mathbf{W} = \mathcal{N}(0, s/\sqrt{d})$, the optimal value of s is around 1 even for high dimensional data.

TABLE 3: Accuracy in % (bold) and runtime in seconds for the classification datasets.

	Iris	Wisconsin B.C.	Pima I.D.	Wine
ELM	**92.6**	**96.7**	**71.8**	**90.0**
	3.4e-3	0.02	0.02	0.01
P-ELM	**95.0**	**96.6**	**73.6**	**95.8**
	6.4e-3	0.12	0.09	0.07
OP-ELM	**95.6**	**95.7**	**75.0**	**95.3**
	0.08	0.83	0.82	0.81
SVM	**95.4**	**91.6**	**72.7**	**95.8**
	2.3e+2	2.9e+3	3.3e+3	3.8e2
GP	**95.6**	**97.3**	**76.3**	**96.1**
	0.76	6.1	5.8	1.9
MLP	**94.8**	**95.6**	**75.2**	**96.0**
	7.6e+2	1.7e+3	4.1e+2	1.2e+3

methods, and this speedup does not decrease the performance. A basic ELM is 6 orders of magnitude faster than SVM in *Computer* regression problem and 5 orders of magnitude faster in *Wisconsin Breast Cancer* classification problem, and it has better performance in both cases.

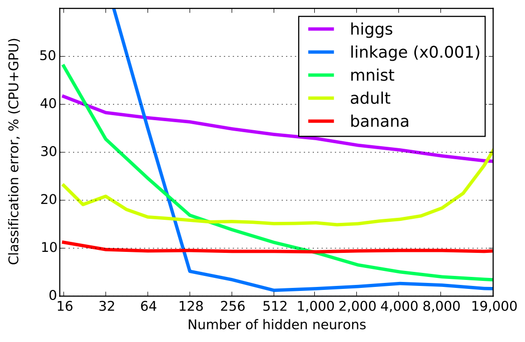
5.4. Performance on Large Datasets

Large datasets are classified with the toolbox on a workstation with 4-core 4GHz CPU and GTX Titan Black GPU. Additional experiments show runtime comparison with a cluster node having two 8-core 2.6GHz CPUs, and with a Macbook Air laptop having a 2-core 1.4GHz CPU.

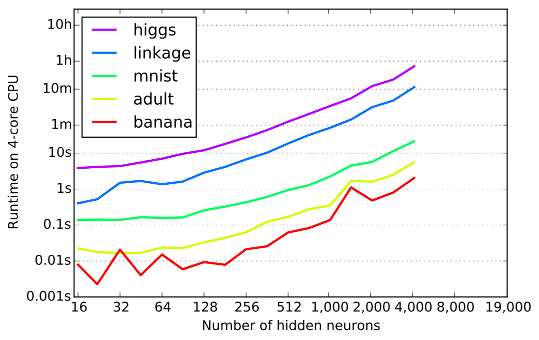
Datasets is split into training and test sets, stored in HDF5 format. They are processed by HPELM toolbox class on both CPU (up to 4096 hidden neurons) and GPU (up to 19,000 hidden neurons, limited by the GPU memory). The classification is done by a basic ELM model with sigmoid hidden neurons. Multiple models are trained for different numbers of hidden neurons. Prediction performance on a test set and training time are shown on Figure 7.

The results show fast training times even for the largest datasets with moderate numbers of neurons. Only the largest ELM models surpass the 1 hour training time. With low number of neurons, even HIGGS datasets is processed in a few seconds on any hardware including the laptop.

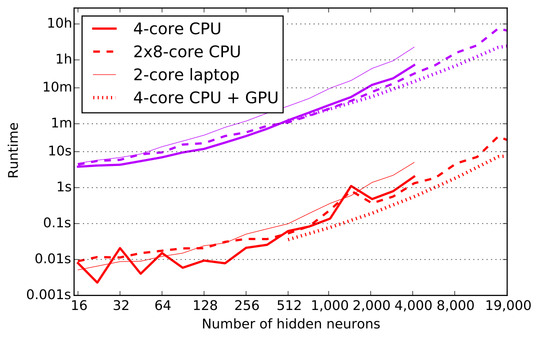
High computational power devices like GPU on multiprocessor nodes speedup ELM training with more than 1000 hidden neurons. This happens because operations with small matrices cannot fully utilize those devices, thus the sequential performance and disk access become limiting factors. With very high L, a speedup provided by the GPU is roughly 5 times, which is consistent with the relative theoretical CPU:GPU = 1:5 performance in double precision.



(a) Test errors



(b) Runtime of all datasets



(c) Runtime on different hardware

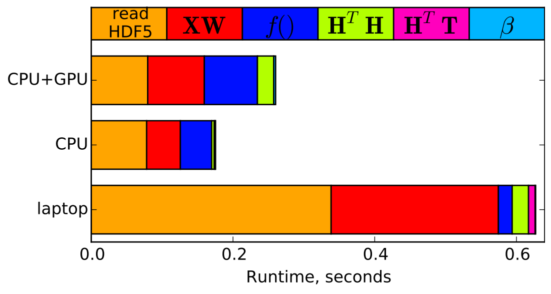
Figure 7: Test errors (*a*) and runtimes (*b*) on different hardware (*c*) for large datasets, on *logarithmic scale*. Runtime on different hardware is shown for two datasets only for image clarity. The 4-core CPU runs at 4 GHz, 2x8-core CPU run at 2.6 GHz, a 2-core laptop CPU runs an 1.4 GHz and the GPU is GTX Titan Black (similar to Tesla K40).

A low-power laptop performs surprisingly well in comparison with other hardware. The maximum difference in runtime (vs. a GPU at $L = 4096$) is only 10 times. For smaller numbers of neurons the runtime difference is even less. Thus a medium size ELM model can be trained fast even on a common laptop with a low-power CPU.

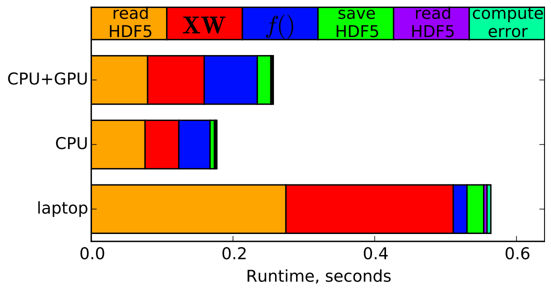
5.5. Runtime Analysis of HPELM with MNIST Dataset

The runtime analysis of the `HPELM` implementation from the toolbox is done on MNIST classification dataset. It has 60,000 training samples with 784 features, and 10 target classes. The training and test data is stored in HDF5 file format. Experiments are performed using a basic ELM with small (64) and large (4096) numbers of sigmoid hidden neurons. First, an ELM model is trained for each number of neurons. Second, classes are predicted for the training data and a mis-classification error is computed. The training data is used for prediction to obtain a comparable runtime.

The runtime for 3 different hardware setups is shown on Figure 8 (64 neurons) and Figure 9 (4096 neurons). The runtime is obtained with a Python line profiler[11] tool. Processing steps with insignificant runtime are omitted; altogether they take less than 1% of runtime.



(a) Training with 64 hidden neurons

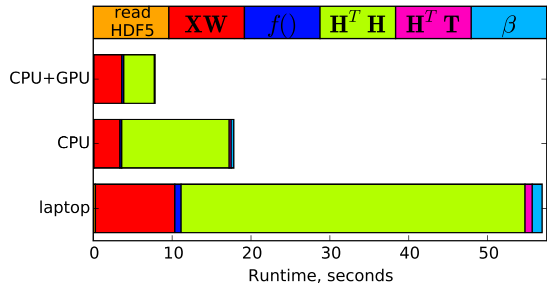


(b) Prediction with 64 hidden neurons

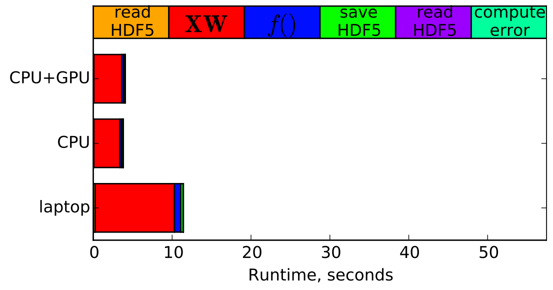
Figure 8: Training (*a*) and prediction (*a*) runtimes of a basic ELM for MNIST dataset with 64 neurons. ELM predictions are obtained on the same training dataset for comparable runtimes.

For a small number of hidden neurons, the training takes only 0.2 seconds on 4-core CPU. The runtime is spent on loading, projecting the data and applying a non-linear function. The largest overhead is reading data from an HDF5 file, where a laptop with a slow CPU spends half of the runtime. Applying a function has a larger overhead on 4-core CPU because it starts in parallel on all cores. Also, there is a small overhead for using a GPU to compute $\mathbf{H}^T\mathbf{H}$. The useful work ($\mathbf{XW}$, $\mathbf{H}^T\mathbf{H}$ and $f()$) takes about a half of

11. https://pypi.python.org/pypi/line_profiler



(a) Training with 4096 hidden neurons



(b) Prediction with 4096 hidden neurons

Figure 9: Training (*a*) and prediction (*b*) runtimes of a basic ELM for MNIST dataset with 4096 neurons. ELM predictions are obtained on the same training dataset for comparable runtimes.

the runtime, which is normal for such short runtimes with a universal toolbox.

With 4096 hidden neurons, $> 98\%$ of runtime is spent on actual computations. File access time and other overheads are negligible. Computing the covariance matrix $\mathbf{H}^T\mathbf{H} = \mathbf{\Omega}_h$ takes the most of runtime during training. The time to compute $\mathbf{\Omega}_h$ is reduced significantly by GPU acceleration. The prediction runtime on all devices is completely dominated by the cost of projecting the input data into hidden layer, which is not accelerated by the GPU.

Interestingly, computing weights $\boldsymbol{\beta}$ has an insignificant runtime when done from correlation matrices $\mathbf{\Omega}_h$ and $\mathbf{\Omega}_t$. Data read and write with HDF5 files is fast, thus the HDF5 file format is used in HP-ELM. Also, an application of a non-linear function takes little time, which is noticeable only on slow hardware and small models.

The GPU in the current HP-ELM accelerates the computation of $\mathbf{\Omega}_h$, $\mathbf{\Omega}_t$ and $\boldsymbol{\beta}$. It speeds up an ELM with large number of neurons (see Figure 9) because ELM computational complexity is cubic w.r.t. the number of neurons.

Overall, the runtime analysis shows high efficiency of the proposed toolbox. The effective runtime starts at 50% with a small ELM and goes over 98% for larger models. These computations are done by calling extremely well optimized BLAS matrix subroutines, which guarantee the smallest possible runtime. BLAS subroutimes are called by Numpy Python library, which can use various implementations of BLAS including open source ones.

Also the analysis clearly shows the part which requires acceleration in large ELM mode: the computation of $\mathbf{\Omega}_h = \mathbf{H}^T\mathbf{H}$. It is combined with $\mathbf{H}^T\mathbf{T}$ and $\boldsymbol{\beta}$ in a simple GPU-accelerated part, which however greatly benefits the training time of larger ELMs (reducing it twice on Figure 9).

5.6. Big Data Processing and Performance

An example Big Data problem has 0.5 billion training samples in 147-dimensional space. It is solved by training an ELM with 19,000 sigmoid hidden neurons. A weighted classification is used to counter imbalance between the two target classes. The computations are done by splitting the data into small parts with about 1 hour of processing time each. This prevents the loss of data in case of computer failure, and allows for parallelization.

The skin detection Big Data dataset sets two additional challenges to the ELM model compared to large datasets. It requires a balanced classification as the amount of data in the two classes is uneven (17% of skin and 83% of non-skin). It also requires testing different numbers of neurons to find out whether 19,000 hidden neurons is enough and if there is any over-fitting. The two aforementioned requirements become challenges, because the training time with 0.5 billion training data samples is so long that an ELM can be trained only once. More specifically, the matrix $\mathbf{\Omega}_h$ can be computed once as it takes more than 99% of the runtime. The model structure selection and class balancing must rely on one particular computed $\mathbf{\Omega}_h$. Same holds true for the matrix $\mathbf{\Omega}_t$, but with only two outputs it's much faster to compute.

An ELM is trained using one workstation with 4-core 4GHz CPU with GTX Titian Black GPU (similar to Tesla K40). Due to GPU acceleration, it took 135 hours in total which is less than a week. Runtimes for other hardware are estimated in Table 4. Without GPU acceleration, the processing time of Big Data problem becomes prohibitively large — almost 2 months using a laptop.

TABLE 4: Training time of an ELM with 19,000 hidden neurons on 0,5 billion samples with 147 features.

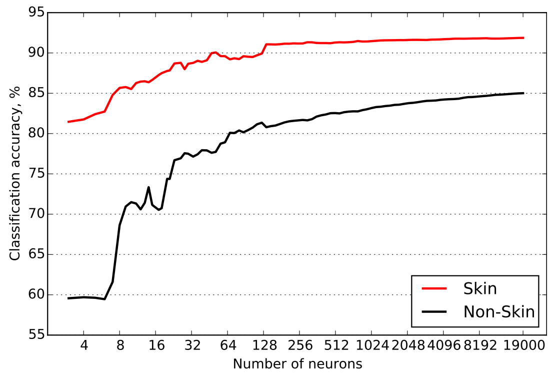
4-core CPU + GPU	**4-core CPU**	**laptop**	**2x8-core CPU**
5d 15h	$\approx$ 16d 4h	$\approx$ 51d 2h	$\approx$ 15d 5h

The final test accuracy with 19,000 hidden neurons is 86,46%, and the confusion matrix is presented on Table 5.

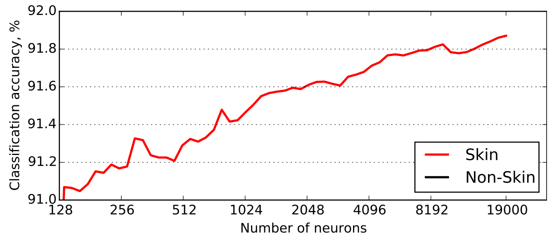
TABLE 5: Test Confusion matrix for ELM with 19,000 neurons.

True class	Predicted class	
	Non-skin	**Skin**
Non-skin	374,840,727	66,107,142
Skin	9,626,072	108,786,059

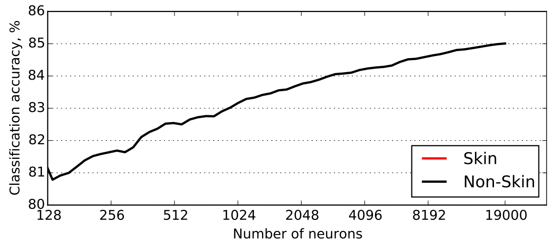
Test accuracy for different numbers of neurons is computed from a single matrix $\mathbf{\Omega}_h$, as explained in section 4.9. 100 different numbers of neurons are tested, spaced equally on a logarithmic scale from 3 to 19,000. For each number, ELM output weights $\boldsymbol{\beta}$ are solved and a separate confusion matrix is computed. The classification results for skin and non-skin from these confusion matrices are shown on Figure 10. Getting this test accuracy plot took 60 hours: 13 hours to obtain hidden layer output $\mathbf{H}$, and 47 hours to compute confusion matrices for all the 100 different numbers on hidden neurons.



(a) Test classification accuracy.



(b) Zoom on skin accuracy.



(c) Zoom on non-skin accuracy.

Figure 10: Test classification accuracy for skin and non-skin pixels. Model does not overfit with 19,000 neurons. Note the logarithmic x axis.

The test accuracy plot shows very good results for skin pixels classification, owing to a balanced classification method used. An ELM without class balancing would be heavily biased towards predicting non-skin pixels, which are 83% of samples in the dataset. The improvement of skin classification accuracy slows down past 128 hidden neurons, so a smaller ELM can be used for detecting skin. However, the non-skin classification accuracy grows steadily up to the maximum number of neurons. This can be explained by a higher variety of non-skin pixel masks than skin ones. ELM does not overfit even with 19,000 neurons, although the performance gain decreases at large numbers of hidden neurons.

6. Conclusion

The paper presents a methodology and a toolbox for highly scalable Extreme Learning Machines. This toolkit creates generalized SLFNs and trains them using ELM methods, but can be a building block for future works on multi-layered ELMs. It is fast, easy to install and easy to use. It solves classification and regression problems on all kinds of datasets — small ones with model structure selection and regularization, and large ones with accelerated computations.

The toolbox is optimized to reduce overheads, including a fast file storage and efficient matrix algebra libraries. It includes an accelerated part for which any accelerator can be used: CUDA-based GPU, OpenCL-based GPU, or a Xeon Phi card.

Big Data problems are the ultimate target of this toolbox. Efficient file storage and an easily parallelized solution method are the necessary parts of the toolbox dealing with Big Data. The GPU acceleration is a key feature which allows solving the largest problems on modest hardware, like a personal workstation with a powerful video card.

References

[1] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 2. IEEE, 2004, pp. 985–990.

[2] ——, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 13, pp. 489 – 501, 2006, neural Networks Selected Papers from the 7th Brazilian Symposium on Neural Networks (SBRN '04) 7th Brazilian Symposium on Neural Networks. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231206000385

[3] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification." *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 42, no. 2, pp. 513–529, Apr. 2012. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/21984515

[4] E. Cambria *et al.*, "Extreme Learning Machines." *IEEE Intelligent Systems*, vol. 28, no. 6, pp. 30–59, 2013. [Online]. Available: http://sentic.net/extreme-learning-machines.pdf

[5] G.-B. Huang, L. Chen, and C.-K. Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 879–892, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1650244

[6] G.-B. Huang, "What are Extreme Learning Machines? Filling the Gap between Frank Rosenblatts Dream and John von Neumanns Puzzle," *(In Press), To Appear in Cognitive Computation*, 2015.

[7] ——, "An insight into extreme learning machines: random neurons, random features and kernels," *Cognitive Computation*, vol. 6, no. 3, pp. 376–390, 2014.

[8] H. White, "An additional hidden unit test for neglected nonlinearity in multilayer feedforward networks," in *Neural Networks, 1989. IJCNN., International Joint Conference on*, 1989, pp. 451–455 vol.2.

[9] ——, "Chapter 9 Approximate Nonlinear Forecasting Methods," ser. Handbook of Economic Forecasting, C. W. J. G. G. Elliott and A. Timmermann, Eds. Elsevier, 2006, vol. 1, pp. 459–512. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574070605010098

[10] Y.-H. Pao, G.-H. Park, and D. J. Sobajic, "Learning and generalization characteristics of the random vector functional-link net," *Neurocomputing*, vol. 6, no. 2, pp. 163–180, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0925231294900531

[11] W. Schmidt, M. Kraaijveld, and R. Duin, "Feedforward neural networks with random weights," in *Pattern Recognition, 1992. Vol.II. Conference B: Pattern Recognition Methodology and Systems, Proceedings., 11th IAPR International Conference on*, Aug 1992, pp. 1–4.

[12] B. Igelnik and Y.-H. Pao, "Stochastic choice of basis functions in adaptive function approximation and the functional-link net," *Neural Networks, IEEE Transactions on*, vol. 6, no. 6, pp. 1320–1329, Nov. 1995.

[13] S. Haykin and N. Network, *Neural Networks: A comprehensive foundation*, 2004.

[14] F. Rosenblatt and C. Nonr, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, pp. 386–408, Nov. 1958. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/13602029

[15] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://www.springerlink.com/index/K238JX04HM87J80G.pdf

[16] G.-b. Huang, S. Member, Q.-y. Zhu, K. Z. Mao, C.-k. Siew, P. Saratchandran, and N. Sundararajan, "Can threshold networks be trained directly?" *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 3, pp. 187–191, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1605431

[17] G. Tsoumakas and I. Katakis, "Multi-Label Classification: An Overview," *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, pp. 1–17, 2007.

[18] Y. Zhai, Y. Ong, and I. Tsang, "The Emerging "Big Dimensionality"," *IEEE Computational Intelligence Magazine*, vol. 9, no. 3, pp. 14–26, 2014.

[19] Y. Miche, A. Akusok, J. Hegedus, and R. Nian, "A Two-Stage Methodology using K-NN and False Positive Minimizing ELM for Nominal Data Classification," *Cognitive Computation*, pp. 1–26, 2014.

[20] A. Akusok, A. Grigorievskiy, A. Lendasse, and Y. Miche, "Image-based Classification of Websites," in *Machine Learning Reports 02/2013*, ser. Machine Learning Reports, T. Villmann and F.-M. Schleif, Eds., vol. ISSN: 18. Saarbrücken, Germany: Workshop of the GI-Fachgruppe Neuronale Netze and the German Neural Networks Society in connection to GCPR 2013, Sep. 2013, pp. 25–34. [Online]. Available: http://www.techfak.uni-bielefeld.de/~fschleif/mlr/mlr_02_2013.pdf

[21] A. Akusok, Y. Miche, J. Karhunen, K.-M. Bjork, R. Nian, and A. Lendasse, "Arbitrary category classification of websites based on image content," *IEEE Computational Intelligence Magazine*, vol. 10, no. 2, pp. 30–41, May 2015.

[22] G.-B. Huang, Z. Bai, L. Kasun, and C. M. Vong, "Local receptive fields based extreme learning machine," *IEEE Computational Intelligence Magazine*, vol. 10, no. 2, pp. 18–29, May 2015.

[23] N.-Y. Liang, G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A fast and accurate online sequential learning algorithm for feedforward networks," *Neural Networks, IEEE Transactions on*, vol. 17, no. 6, pp. 1411–1423, 2006.

[24] A. van Schaik and J. Tapson, "Online and adaptive pseudoinverse solutions for ELM weights," *Neurocomputing*, vol. 149, Part, no. 0, pp. 233–238, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231214011485http://arxiv.org/abs/1405.7777

[25] M. van Heeswijk, Y. Miche, T. Lindh-Knuutila, P. Hilbers, T. Honkela, E. Oja, and A. Lendasse, "Adaptive Ensemble Models of Extreme Learning Machines for Time Series Prediction," in *Artificial Neural Networks ICANN 2009*, ser. Lecture Notes in Computer Science, C. Alippi, M. Polycarpou, C. Panayiotou, and G. Ellinas, Eds. Springer Berlin Heidelberg, 2009, vol. 5769, pp. 305–314. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04277-5_31

[26] M. van Heeswijk, Y. Miche, E. Oja, and A. Lendasse, "GPU-accelerated and parallelized ELM ensembles for large-scale regression," *Neurocomputing*, vol. 74, no. 16, pp. 2430–2437, Sep. 2011.

[27] Y. Miche, P. Bas, C. Jutten, O. Simula, and A. Lendasse, "A Methodology for Building Regression Models using Extreme Learning Machine: OP-ELM." in *ESANN*, 2008, pp. 247–252. [Online]. Available: http://dblp.uni-trier.de/db/conf/esann/esann2008.html\#MicheBJSL08

[28] Y. Miche, A. Sorjamaa, P. Bas, O. Simula, C. Jutten, and A. Lendasse, "OP-ELM: optimally pruned extreme learning machine," *Neural Networks, IEEE Transactions*, vol. 21, no. 1, pp. 158–162, Jan. 2010. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/20007026http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5350449

[29] Y. Miche, M. van Heeswijk, P. Bas, O. Simula, and A. Lendasse, "TROP-ELM: A double-regularized ELM using LARS and Tikhonov regularization," *Neurocomputing*, vol. 74, no. 16, pp. 2413–2421, Sep. 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S092523121100258X

[30] Q. Yu, Y. Miche, E. Eirola, M. van Heeswijk, E. Séverin, and A. Lendasse, "Regularized extreme learning machine for regression with missing data," *Neurocomputing*, vol. 102, pp. 45–51, 2013. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84870244730\&partnerID=40\&md5=4bc9805ef198a3de44fd2c2a976834b5

[31] W. Zong, G.-B. Huang, and Y. Chen, "Weighted extreme learning machine for imbalance learning," *Neurocomputing*, vol. 101, no. 0, pp. 229–242, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231212006479

[32] S. Chen, C. F. N. Cowan, and P. M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *Neural Networks, IEEE Transactions on*, vol. 2, no. 2, pp. 302–309, Mar. 1991.

[33] C. R. Rao and S. S. K. Mitra, "Generalized inverse of a matrix and its applications," *Wiley Series in Probability and Mathematical Statistics (New York)*, 1971. [Online]. Available: http://web.mse.uiuc.edu/group/downloads/Aftab/VufoilsfromHangXiao2-2-10/euclid.bsmsp.1200514113.pdf

[34] A. N. Tikhonov, "Solution of incorrectly formulated problems and the regularization method," *Soviet Math. Dokl.*, vol. 4, pp. 1035–1038, 1963.

[35] A. E. Hoerl and R. W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[36] T. Simil, J. Kacprzyk, E. Oja, T. Similä, and J. Tikka, "Multiresponse sparse regression with application to multidimensional scaling," *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005*, vol. 3697, pp. 97–102, 2005. [Online]. Available: http://www.springerlink.com/index/10RLNNRAP4GYXJJG.pdf

[37] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least Angle Regression," *The Annals of Statistics*, vol. 32, no. 2, pp. 407–451, 2004.

[38] T. Simila and J. Tikka, "Common subset selection of inputs in multiresponse regression," in *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, 2006, pp. 1908–1915.

[39] M. Lichman, "UCI Machine Learning Repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[40] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 27:1—-27:27, 2011.

[41] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information science and statistics, M. Jordan, J. Kleinberg, and B. Schölkopf, Eds. Springer, 2006, vol. 4, no. 4. [Online]. Available: http://www.library.wisc.edu/selectedtocs/bg0137.pdf

[42] C. Rasmussen, "Gaussian Processes in Machine Learning," in *Advanced Lectures on Machine Learning*, ser. Lecture Notes in Computer Science, O. Bousquet, U. von Luxburg, and G. Rätsch, Eds. Springer Berlin Heidelberg, 2004, vol. 3176, pp. 63–71. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28650-9_4

[43] S. L. Phung, A. Bouzerdoum, and S. Chai D., "Skin segmentation using color pixel classification: analysis and comparison," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 1, pp. 148–154, Jan. 2005.