World Wide Web
DOI 10.1007/s11280-013-0236-2

# ELM*: distributed extreme learning machine with MapReduce

**Junchang Xin · Zhiqiong Wang · Chen Chen · Linlin Ding · Guoren Wang · Yuhai Zhao**

Received: 19 September 2012 / Revised: 9 June 2013 / Accepted: 14 June 2013


**Abstract** Extreme Learning Machine (ELM) has been widely used in many fields such as text classification, image recognition and bioinformatics, as it provides good generalization performance at a extremely fast learning speed. However, as the data volume in real-world applications becomes larger and larger, the traditional centralized ELM cannot learn such massive data efficiently. Therefore, in this paper, we propose a novel Distributed Extreme Learning Machine based on MapReduce framework, named ELM*, which can cover the shortage of traditional ELM whose learning ability is weak to huge dataset. Firstly, after adequately analyzing the property of traditional ELM, it can be found out that the most expensive computation part of the matrix Moore-Penrose generalized inverse operator in the output weight vector calculation is the matrix multiplication operator. Then, as the matrix multiplication operator is decomposable, a Distributed Extreme Learning Machine (ELM*) based on MapReduce framework can be developed, which can first calculate the matrix multiplication effectively with MapReduce in parallel, and then calculate the corresponding output weight vector with centralized computing.

J. Xin (✉) · C. Chen · L. Ding · G. Wang · Y. Zhao
Key Laboratory of Medical Image Computing (NEU), Ministry of Education, Shenyang, China
e-mail: xinjunchang@ise.neu.edu.cn

G. Wang
e-mail: wanggr@mail.neu.edu.cn

Y. Zhao
e-mail: zhaoyuhai@ise.neu.edu.cn

J. Xin · C. Chen · L. Ding · G. Wang · Y. Zhao
College of Information Science & Engineering, Northeastern University,
Shenyang, Liaoning, China

Z. Wang
Sino-Dutch Biomedical & Information Engineering School, Northeastern University,
Shenyang, Liaoning, China
e-mail: wangzq@bmie.neu.edu.cn

Published online: 29 June 2013

Therefore, the learning of massive data can be made effectively. Finally, we conduct extensive experiments on synthetic data to verify the effectiveness and efficiency of our proposed ELM* in learning massive data with various experimental settings.

**Keywords** Extreme learning machine · Massive data processing · Cloud computing · MapReduce

## 1 Introduction

With the advent of information technology, a large amount of training data has been accumulated in many machine learning applications. Generally, the larger the training sample, the better the classifier, although the returns begin to diminish once a certain volume of training data is exceeded [27]. However, as the data volume to be stored and analyzed largely exceeds the storage and computing capacity of a single machine, traditional centralized training algorithms cannot cope with large-scale training datasets efficiently. Therefore, it is needed to scale up traditional machine learning techniques by using massively parallel architectures, which can learn efficiently from the massive training data.

Due to the characteristics of excellent generalization performance, rapid training speed and little human intervene, Extreme Learning Machine (ELM) [10–14, 17] has recently attracted increasing attention from more and more researchers [15]. ELM and its variants have been extensively used in many fields, such as text classification, image recognition, handwritten character recognition, mobile object management and bioinformatics [1, 9, 16, 18, 21, 23–26, 28, 30–33]. Moreover, as a prominent parallel data processing technique, MapReduce [2–4] can process the decomposable problems with huge amounts of data effectively and has been widely used in a broad range of applications and organizations for its scalability, ease-of-use and fault-tolerance [5, 7, 19, 20, 29]. Though both ELM and MapReduce have been studied extensively in the past few years respectively, the problem that how to scale up ELM to make it learn large-scale training dataset by using MapReduce framework efficiently is not studied deeply all along, which becomes a challenge as well.

During the computation course of ELM, the most expensive computation part of the matrix Moore-Penrose generalized inverse operator in output weight vector calculation is the matrix multiplication operator. The matrix multiplication operator is decomposable and the computation cost of these processing courses is large in the whole ELM. If this part of computation can be computed in parallel, the ELM can process the huge amounts of data. In this paper, we propose Distributed Extreme Learning Machine based on MapReduce (ELM*) to improve the scalability of traditional ELM and make it learn the large scale training dataset rapidly. The contributions of this paper can be summarized as follows.

- We prove theoretically that the most expensive computation part of traditional ELM is the decomposable matrix multiplication operator, which indicate that the performance of ELM can be improved by MapReduce.
- A novel Distributed Extreme Learning Machine (ELM*) based on distributed MapReduce framework is proposed, which can learn massive data efficiently in parallel.

– Last but not least, our extensive experimental studies using synthetic data show that our proposed ELM* can learn massive data effectively, which can fulfill the requirements of many real-world applications.

The remainder of the paper is organized as follows. Section 2 briefly introduces ELM, PELM [8] and MapReduce. The theoretical foundation and the computational details of the proposed ELM* are introduced in Section 3. The experimental results are reported in Section 4 to show the effectiveness and efficiency of our proposed ELM*. Finally, we conclude this paper in Section 5.

## 2 Background

In this section, we describe the background for our work, which includes a brief overview of traditional ELM and PELM, and then a detailed description of distributed MapReduce framework is proposed.

### 2.1 ELM

ELM [10, 11] has been originally developed for single hidden-layer feedforward neural networks (SLFNs) and then extended to the "generalized" SLFNs where the hidden layer need not be neuron alike [12, 13]. ELM first randomly assigns the input weights and the hidden layer biases, and then analytically determines the output weights of SLFNs. It can achieve better generalization performance than other conventional learning algorithms at a extremely fast learning speed. Besides, ELM is less sensitive to user-specified parameters and can be deployed faster and more conveniently [14, 17].

For $N$ arbitrary distinct samples $(\mathbf{x}_j, \mathbf{t}_j)$, where $\mathbf{x}_j = [x_{j1}, x_{j2}, \ldots, x_{jn}]^T \in \mathbb{R}^n$ and $\mathbf{t}_j = [t_{j1}, t_{j2}, \ldots, t_{jm}]^T \in \mathbb{R}^m$, standard SLFNs with $L$ hidden nodes and activation function $g(x)$ are mathematically modeled as

$$\sum_{i=1}^{L} \beta_i g_i(\mathbf{x}_j) = \sum_{i=1}^{L} \beta_i g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) = \mathbf{o}_j \qquad (j = 1, 2, \ldots, N) \tag{1}$$

where $\mathbf{w}_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^T$ is the weight vector connecting the $i$th hidden node and the input nodes, $\beta_i = [\beta_{i1}, \beta_{i2}, \ldots, \beta_{im}]^T$ is the weight vector connecting the $i$th hidden node and the output nodes, $b_i$ is the threshold of the $i$th hidden node, and $\mathbf{o}_j = [o_{j1}, o_{j2}, \ldots, o_{jm}]^T$ is the $j$th output vector of the SLFNs [10].

The standard SLFNs with $L$ hidden nodes and activation function $g(x)$ can approximate these $N$ samples with zero error. It means $\sum_{j=1}^{L} ||\mathbf{o}_j - \mathbf{t}_j|| = 0$ and there exist $\beta_i$, $\mathbf{w}_i$ and $b_i$ such that

$$\sum_{i=1}^{L} \beta_i g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) = \mathbf{t}_j \qquad (j = 1, 2, \ldots, N) \tag{2}$$

The equation above can be expressed compactly as follows:

$$\mathbf{H}\beta = \mathbf{T} \tag{3}$$

where $\mathbf{H}(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_L, b_1, b_2, \ldots, b_L, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L)$

$$= [h_{ij}] = \begin{bmatrix} g(\mathbf{w}_1 \cdot \mathbf{x}_1 + b_1) & g(\mathbf{w}_2 \cdot \mathbf{x}_1 + b_2) & \ldots & g(\mathbf{w}_L \cdot \mathbf{x}_1 + b_L) \\ g(\mathbf{w}_1 \cdot \mathbf{x}_2 + b_1) & g(\mathbf{w}_2 \cdot \mathbf{x}_2 + b_2) & \ldots & g(\mathbf{w}_L \cdot \mathbf{x}_2 + b_L) \\ \vdots & \vdots & \vdots & \vdots \\ g(\mathbf{w}_1 \cdot \mathbf{x}_N + b_1) & g(\mathbf{w}_2 \cdot \mathbf{x}_N + b_2) & \ldots & g(\mathbf{w}_L \cdot \mathbf{x}_N + b_L) \end{bmatrix}_{N \times L} \tag{4}$$

$$\beta = \begin{bmatrix} \beta_{11} & \beta_{12} & \ldots & \beta_{1m} \\ \beta_{21} & \beta_{22} & \ldots & \beta_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ \beta_{L1} & \beta_{L2} & \ldots & \beta_{Lm} \end{bmatrix}_{L \times m} \quad \text{and} \quad \mathbf{T} = \begin{bmatrix} t_{11} & t_{12} & \ldots & t_{1m} \\ t_{21} & t_{22} & \ldots & t_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ t_{N1} & t_{N2} & \ldots & t_{Nm} \end{bmatrix}_{N \times m} \tag{5}$$

$\mathbf{H}$ is called the hidden layer output matrix of the neural network and the $i$th column of $\mathbf{H}$ is the $i$th hidden node output with respect to inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$. The smallest norm least-squares solution of the above linear system is:

$$\hat{\beta} = \mathbf{H}^{\dagger}\mathbf{T} \tag{6}$$

where $\mathbf{H}^{\dagger}$ is the Moore-Penrose generalized the inverse of matrix $\mathbf{H}$. Then the output function of ELM can be modeled as follows.

$$f(\mathbf{x}) = \mathbf{h}(\mathbf{x})\beta = \mathbf{h}(\mathbf{x})\mathbf{H}^{\dagger}\mathbf{T} \tag{7}$$

The computational process for ELM training is given in Algorithm 1. There are three important input parameters in the ELM training, the training set $\mathcal{N} = \{(\mathbf{x}_j, \mathbf{t}_j) | \mathbf{x}_j \in \mathbb{R}^n, \mathbf{t}_j \in \mathbf{R}^m, j = 1, 2, \ldots, N\}$, the hidden node output function $g(\mathbf{w}_i, b_i, \mathbf{x}_j)$ and the hidden node number $L$. Only properly setting the related parameters, ELM can start its training process. Firstly, ELM randomly generates $L$ pairs of hidden node parameters $(\mathbf{w}_i, b_i)$ (Lines 1–2). Then, according to the input and randomly generated parameters, it calculates the hidden layer output matrix $\mathbf{H}$ by using (4) (Line 3). Finally, utilizing (6), it calculates the corresponding output weight vector $\beta$ (Line 4). After completing the above training process, the output of the new dataset can be predicted by ELM according to (7).

**Algorithm 1** ELM training

**1** **for** $i = 1$ to $L$ **do**
**2** Randomly generate hidden node parameters $(\mathbf{w}_i, b_i)$
**3** Calculate the hidden layer output matrix $\mathbf{H}$
**4** Calculate the output weight vector $\beta = \mathbf{H}^{\dagger}\mathbf{T}$

## 2.2 PELM

He Q. et al. [8] designed and implemented an efficient parallel ELM for regression based on MapReduce framework named PELM. Through analyzing the mechanism of ELM algorithm, $\mathbf{H}$ is a $N \times L$ dimension matrix, so $\mathbf{H} \times \mathbf{H}^T$ is a $N \times N$ dimension matrix. However, in massive data mining, $N$ is always a very large number, so $\mathbf{H} \times \mathbf{H}^T$ must be a too large matrix for memory, which makes it impossible to execute ELM algorithm in the way of memory-residence. In most cases, the number of hidden nodes is much less than the number of training samples, $L \ll N$. According to the

matrix theory, in SVD method, a small matrix $\mathbf{H}^T \times \mathbf{H}$ could be calculated instead of the large matrix $\mathbf{H} \times \mathbf{H}^T$. PELM uses two MapReduce jobs to obtain the final ELM results. The concrete course is as follows.

The first MapReduce course computes the hidden node output matrix $\mathbf{H}$, which is the independent variables matrix of $N$ samples. The MapReduce computation of hidden layer mapping is just mapping samples to space represented by hidden nodes. The second MapReduce job of matrix generalized inverse computes the matrix multiplication $\mathbf{H}^T \times \mathbf{H}$ and $\mathbf{H}^T \times \mathbf{T}$ in parallel. In map function, the elements of hidden node output vector are firstly parsed, and then they multiply each other together to form the intermediate results. In reduce function, the intermediate results are merged, sorted and summed to achieve the matrix multiplication $\mathbf{H}^T \times \mathbf{H}$ and $\mathbf{H}^T \times \mathbf{T}$.

In PELM, after the first MapReduce job obtains the output matrix $\mathbf{H}$, $\mathbf{H}$ is written into HDFS. The second MapReduce job should read the output matrix $\mathbf{H}$ from HDFS, and then computes the final ELM results. Therefore, there are numerous intermediate results (i.e. $N \times L$ dimension matrix $\mathbf{H}$) transformed during the two MapReduce jobs, which increase the processing time of ELM based on MapReduce framework. Comparing with PELM, our ELM* combines the two MapReduce jobs and only use one MapReduce job to obtain the final ELM results. It does not only reduce the transmitting cost of numerous intermediate results, but also enhance the processing efficiency.

### 2.3 MapReduce

The MapReduce framework was originally proposed by Google for performing distributed computations on a large number of computers [2–4], with one of its open source implementations Hadoop.[1] The main idea of MapReduce is to hide the details of parallel execution and allow users to only focus on the data processing strategies. It provides two primitive functions, map and reduce, to describe the distributed computation task. The map function takes a key-value pair $(k_1, v_1)$ as input and generates intermediate key-value pairs $[(k_2, v_2)]$. The reduce function combines all intermediate key-value pairs $(k_2, [v_2])$ with the same key into output key-value pairs $[(k_3, v_3)]$. Developers can implement their applications using these two functions. And then, MapReduce runtime distributes and executes the task automatically. Thus, the complexity of parallel programming is dramatically reduced. Non-professional programmers can easily deploy their parallel applications by merely specifying the map function and the reduce function, whereas task scheduling, data distribution, load balancing, and fault tolerance are handled automatically.

The typical procedure of a MapReduce job is illustrated in Figure 1. First, the input to a MapReduce job starts as the dataset stored on the underlying distributed file system (e.g. GFS [6] and HDFS [22]), which is split in a number of files across machines (Phase I: Input). Next, the MapReduce job is partitioned into independent map tasks. Each map task processes a logical split of the input dataset. The map task reads the data and applies the user-defined map function on each record, and then buffers the resulting output. This intermediate data is sorted and

[1] http://hadoop.apache.org/

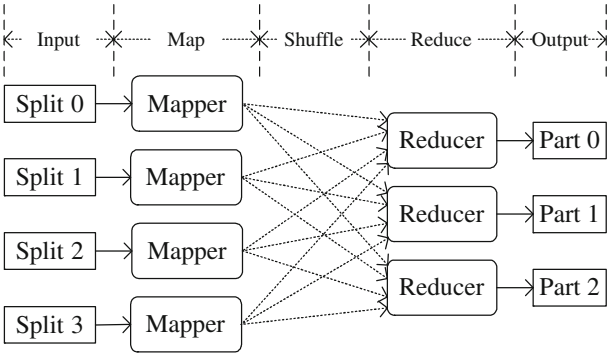



**Figure 1** Illustration of MapReduce

partitioned for reduce phase, and written to the local disk of the machine executing the corresponding map task (Phase II: Map). After that, the intermediate data files from the already completed map tasks are fetched by the corresponding reduce task following the "pull" model (Similarly, the MapReduce job is also partitioned into independent reduce tasks). The intermediate data files from all the map tasks are sorted accordingly (Phase III: Shuffle). Then, the sorted intermediate data is passed to the reduce task. The reduce task applies the user-defined reduce function to the intermediate data and generates the output data (Phase IV: Reduce). Finally, the output data from the reduce task is generally written back to the distributed file system (Phase V: Output).

## 3 Distributed extreme learning machine with MapReduce

In this section, the preliminaries which are the theoretical foundation of our work are first introduced. And then, the computational details of our proposed ELM* are presented.

### 3.1 Preliminaries

The orthogonal projection method can be efficiently used in ELM [17]: $\mathbf{H}^{\dagger} = \left(\mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T$ if $\mathbf{H}^T\mathbf{H}$ is nonsingular or $\mathbf{H}^T\left(\mathbf{H}\mathbf{H}^T\right)^{-1}$ if $\mathbf{H}\mathbf{H}^T$ is nonsingular. According to the ridge regression theory, it suggests that a positive value $1/\lambda$ is added to the diagonal of $\mathbf{H}^T\mathbf{H}$ or $\mathbf{H}\mathbf{H}^T$ in the calculation of the output weights $\beta$. The resultant solution is stable and tends to have better generalization performance [17]. That is, in order to improve the stability of ELM we can have

$$\beta = \left(\frac{\mathbf{I}}{\lambda} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T} \tag{8}$$

and the corresponding output function of ELM is:

$$f(\mathbf{x}) = \mathbf{h}(\mathbf{x})\beta = \mathbf{h}(\mathbf{x})\left(\frac{\mathbf{I}}{\lambda} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T} \tag{9}$$

Or we can have

$$\beta = \mathbf{H}^T \left( \frac{\mathbf{I}}{\lambda} + \mathbf{H}\mathbf{H}^T \right)^{-1} \mathbf{T} \tag{10}$$

and the corresponding output function of ELM is:

$$f(\mathbf{x}) = \mathbf{h}(\mathbf{x})\beta = \mathbf{h}(\mathbf{x})\mathbf{H}^T \left( \frac{\mathbf{I}}{\lambda} + \mathbf{H}\mathbf{H}^T \right)^{-1} \mathbf{T} \tag{11}$$

During the training process of large-scale dataset, the number of training data is much larger than the dimensionality of the feature space, that is to say, $N \gg L$. According to $N \gg L$, the size of $\mathbf{H}^T\mathbf{H}$ is much smaller than that of $\mathbf{H}\mathbf{H}^T$. Thus, it is a better choice of using (8) to calculate the output weight vector $\beta$ of ELM training. Moreover, (9) is the corresponding output function. The process of ELM on huge data training can be concluded into Algorithm 2. Algorithm 2 is similar to Algorithm 1. However, the only difference between them is that Algorithm 2 uses (8) to calculate the output weight vector $\beta$, while Algorithm 1 uses (6) to do so.

**Algorithm 2** ELM huge data training

**1** **for** $i = 1$ to $L$ **do**
**2** Randomly generate hidden node parameters $(\mathbf{w}_i, b_i)$
**3** Calculate the hidden layer output matrix $\mathbf{H}$
**4** Calculate the output weight vector $\beta = \left(\mathbf{I}/\lambda + \mathbf{H}^T\mathbf{H}\right)^{-1} \mathbf{H}^T\mathbf{T}$

However, when the amount of training data goes beyond some limitations, Algorithm 2 still has two inevitable problems that cannot be solved efficiently with a single computer. One is to compute the hidden layer output matrix $\mathbf{H}$, and another is to compute $\mathbf{H}^T\mathbf{H}$ and $\mathbf{H}^T\mathbf{T}$ which involved in calculating the output weight vector $\beta$. In the following, we continue to analyze the characters of ELM, and find the part that can be processed in parallel, and transplant it into MapReduce framework. In this way, we can make ELM extend to the scale of any data volume efficiently.

Let $\mathbf{U} = \mathbf{H}^T\mathbf{H}$, $\mathbf{V} = \mathbf{H}^T\mathbf{T}$, and we can get,

$$\beta = \left( \frac{\mathbf{I}}{\lambda} + \mathbf{U} \right)^{-1} \mathbf{V} \tag{12}$$

Moreover, according to (4), we have $h_{ij} = g(\mathbf{w}_j \cdot \mathbf{x}_i + b_j)$, and $h_{ij}^T = h_{ji} = g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i)$. On the basis of the formulas of matrix multiplication, we can further get,

$$u_{ij} = \sum_{k=1}^{N} h_{ik}^T h_{kj} = \sum_{k=1}^{N} h_{ki} h_{kj} = \sum_{k=1}^{N} g(\mathbf{w}_i \cdot \mathbf{x}_k + b_i) g(\mathbf{w}_j \cdot \mathbf{x}_k + b_j) \tag{13}$$

$$v_{ij} = \sum_{k=1}^{N} h_{ik}^T t_{kj} = \sum_{k=1}^{N} h_{ki} t_{kj} = \sum_{k=1}^{N} g(\mathbf{w}_i \cdot \mathbf{x}_k + b_i) t_{kj} \tag{14}$$

According to (13), we know that the item $u_{ij}$ in $\mathbf{U}$ can be expressed by the summation of $h_{ki}$ multiplied by $h_{kj}$. Here, $h_{ki}$ is the $i$th element in the $k$th row of matrix $\mathbf{H}$, and $h_{kj}$ is the $j$th element in the same row. Apparently, both $h_{ki}$ and $h_{kj}$

are from row $h_k$ of the same training data record $\mathbf{x}_k$, which has nothing to do with the other data in the training set.

Similarly, according to (14), we know that item $v_{ij}$ in $\mathbf{V}$ can be expressed by the summation of $h_{ki}$ multiplied by $t_{kj}$. Here, $t_{kj}$ is the $j$th element in the $k$th row of matrix $\mathbf{T}$. Apparently, both $h_{ki}$ and $t_{kj}$ are respectively from the corresponding rows $h_k$ and $t_k$ of the same training data record $\mathbf{x}_k$, which has nothing to do with the other training data as well.

Since $\mathbf{U}$ is a $L \times L$ matrix, the computation cost of calculating $\left(\frac{\mathbf{I}}{\lambda} + \mathbf{U}\right)^{-1}$ is very small. At the same time, as $\mathbf{V}$ is a $L \times m$ matrix, the computation cost of multiplying $\mathbf{V}$ is also very small. Thus, we can make a conclusion that the most expensive computational part in output weight vector $\beta$ calculation process is to solve matrixes $\mathbf{U}$ and $\mathbf{V}$. However, the process of calculating $\mathbf{U}$ and $\mathbf{V}$ are both decomposable. So we can use MapReduce framework to realize parallel computing. Moreover, making full use of the good character of MapReduce framework, we can realize the process of ELM training without the constrain of the calculation and storage ability of a single machine, and then realize the efficient learning of massive training data.

## 3.2 ELM* description

According to the above analysis, we know that the process of calculating matrices $\mathbf{U}$ and $\mathbf{V}$ can be realized by MapReduce framework, which is an efficient realization to cope with massive training data. In this section, we first propose our naive ELM* algorithm (ELM*-Naive) in Section 3.2.1, and then propose our improved ELM* algorithm (ELM*-Improved) in Section 3.2.2.

### *3.2.1 Naive ELM**

From the analysis, we find that matrices $\mathbf{U}$ and $\mathbf{V}$ can share the calculation of $h_k$ in matrix $\mathbf{H}$, and the calculation of partial summation of $u_{ij}$ and $v_{ij}$ are independent. Thus, we can calculate matrices $\mathbf{U}$ and $\mathbf{V}$ during one MapReduce process as shown in Algorithm 3.

Algorithm 3 introduces the naive algorithm of ELM*. This algorithm has two classes, Class Mapper (Lines 1–11) and Class Reducer (Lines 12–17). Class Mapper contains one method, Map (Lines 2–11), while Class Reducer also contains one method, Reduce (Lines 13–17).

In the Map method of Mapper, first, we initial the local variable $h$ (Line 3), and then resolve training record $s$, which means dividing $s$ into training data $\mathbf{x}_k$ and its corresponding training result $t$ (Line 4). What is more, we calculate the partial result of hidden layer output matrix corresponding to $\mathbf{x}_k$ (Lines 5–6). Next, the class Mapper computes the partial summation of elements in matrixes $\mathbf{U}$ and $\mathbf{V}$ then generates the outputs (Lines 7–11). In the Reduce method of Reducer, first, we initialize a temporary variable $uv$ (Line 14). And then, we combine the intermediate summation of different mappers which have the same Key, and furthermore, get the final summation of the corresponding element of the Key (Lines 15–16). Finally, we store the results into the distributed file system (Line 17).

Taking the advantages of MapReduce framework, we realize the calculation of matrices $\mathbf{U}$ and $\mathbf{V}$ for massive training data. As neither of the size of $\mathbf{U}$ and $\mathbf{V}$ overstep the processing ability of a single machine, the process of ELM* is similar to Algorithm 2, which is shown in Algorithm 4. First, we generate $L$ pairs of hidden

**Algorithm 3** ELM*-naive training

```
1  class MAPPER
2    method MAP(sid id, sample s)
3      h = new ASSOCIATIVEARRAY
4      (x, t) = Parse(s)
5      for i = 1 to L do
6        h[i] = g(w_i · x + b_i)
7      for i = 1 to L do
8        for j = 1 to L do
9          context.write(triple ('U', i, j), h[i]h[j])
10       for j = 1 to m do
11         context.write(triple ('V', i, j),h[i]t[j])
12 class REDUCER
13   method REDUCE(triple p, sum [s_1, s_2, ...])
14     uv = 0
15     for all sum s ∈ [s_1, s_2, ...]do
16       uv = uv + s
17       context.write(triple p, sum uv)
```

node parameters $(\mathbf{w}_i, b_i)$ randomly (Lines 1–2). And then, we calculate matrices $\mathbf{U}$ and $\mathbf{V}$ according to the input parameters and randomly generate parameters using Algorithm 4 (Line 3). Finally, using (12), we solve out the output weight vector $\beta$ (Line 4). The ELM* learning process for massive training data finishes after the complement of the training process above, and then we can predict the output results of new dataset according to (9).

**Algorithm 4** ELM*

```
1  for i = 1 to L do
2      Randomly generate hidden node parameters (w_i, b_i)
3  Calculate U = H^T H, V = H^T T with MapReduce
4  Calculate the output weight vector β = (I/λ + U)^{-1} V
```

*3.2.2 Improved ELM**

Because the process of calculating $\mathbf{U}$ and $\mathbf{V}$ are both decomposable, this computation can be resolved by MapReduce framework. Furthermore, to decrease the computation and communication cost during the whole computing course, we can first compute the local summation of matrices $\mathbf{U}$ and $\mathbf{V}$ in the Map phase. Therefore, we propose an improved MapReduce algorithm and the process of calculating matrices $\mathbf{U}$ and $\mathbf{V}$ based on MapReduce framework is shown in Algorithm 5.

This algorithm has two classes, Class Mapper (Lines 1–20) and Class Reducer (Lines 21–26). Class Mapper contains three methods, Initialize (Lines 2–4), Map (Lines 5–14) and Close (Lines 15–20), while Class Reducer only contains one method, Reduce (Lines 22–26). The Class Reducer of Algorithm 5 is the same as Algorithm 3, so we only introduce the Class Mapper in the following.

In the Initialize method of Mapper, we initialize two arrays, $u$ and $v$, which are used to store the intermediate summation of the elements in matrices $\mathbf{U}$ and $\mathbf{V}$

respectively. In the Map method of Mapper, first, we initial the local variable $h$ (Line 6), and then resolve training record $s$, which means dividing $s$ into training data $\mathbf{x}_k$ and its corresponding training result $t$ (Line 7). What is more, we calculate the partial result of hidden layer output matrix corresponding to $\mathbf{x}_k$ (Lines 8–9). Next, we calculate the partial summation of elements in matrix $\mathbf{U}$ and $\mathbf{V}$ respectively, and store the results into their corresponding local variables $u$ and $v$ (Lines 10–14). In the Close method of Mapper, the intermediate summation stored in $u$ and $v$ is emitted by the mapper (Lines 16–20).

**Algorithm 5** ELM*-improved training

```
1  class Mapper
2    method Initialize()
3      u = new AssociativeArray
4      v = new AssociativeArray
5    method Map(sid id, sample s)
6      h = new AssociativeArray
7      (x, t) = Parse(s)
8      for i = 1 to L do
9        h[i] = g(w_i · x + b_i)
10     for i = 1 to L do
11       for j = 1 to L do
12         u[i, j] = u[i, j] + h[i]h[j]
13       for j = 1 to m do
14         v[i, j] = v[i, j] + h[i]t[j]
15   method Close()
16     for i = 1 to L do
17       for j = 1 to L do
18         context.write(triple ('U', i, j),sum u[i, j])
19       for j = 1 to m do
20         context.write(triple ('V', i, j),sum v[i, j])
21 class Reducer
22   method Reduce(triple p, sum [s_1, s_2, ...])
23     uv = 0
24     for all sum s ∈ [s_1, s_2, ...]do
25       uv = uv + s
26       context.write(triple p, sum uv)
```

## 4 Performance evaluation

In this section, the performance of our ELM* is evaluated in details with various experimental settings comparing with PELM [8], containing our ELM*-Naive algorithm and ELM*-Improved algorithms. We first describe the platform used in our experiments in Section 4.1. Then we present and discuss the experimental results of the evaluation in Section 4.2.

### 4.1 Experimental platform

All our experiments are run on a cluster with 9 computers which are connected in a high speed Gigabit network. Each computer has an Intel Quad Core 2.66 GHZ CPU, 4 GB memory and CentOS Linux 5.6. One computer is set as the Master node and the others are set as the Slave nodes. We use Hadoop version 0.20.2 and configure it to run up to 4 map tasks or 4 reduce tasks concurrently per node. Therefore, at any point in time, at most 32 map tasks or 32 reduce tasks can run concurrently in our cluster. In our paper, we test the performance of ELM processing huge amounts of data. In actual applications, there is no real dataset that can reach the scale of massive data. Therefore, we use our synthetic datasets to evaluate the performance of our algorithms with MapReduce framework.

Our ELM* uses the same equation to compute the output weight vector $\beta$ as the traditional ELM substantially. The only difference is that ELM* computes two intermediate results $\mathbf{H}^T\mathbf{H}$(i.e. $\mathbf{U}$) and $\mathbf{H}^T\mathbf{T}$ (i.e. $\mathbf{V}$) in MapReduce framework. The traditional ELM computes $\mathbf{H}^T\mathbf{H}$ and $\mathbf{H}^T\mathbf{T}$ in the centralized environment. Consequently, when all the relative parameters $(L, \mathbf{w}_i, b_i, \mathbf{I})$, are the same as the traditional ELM, $\beta$ of the traditional ELM using (8) is the same as $\beta$ of ELM* using (12). The classifiers of the traditional ELM and ELM* are completely identical, so the classification result is certainly just the same. In other words, our ELM* has the same accuracy as the traditional ELM. In short, ELM* only uses MapReduce framework to complete a part of the computation of the output weight vector $\beta$ of the traditional ELM without changing the training results, and then cannot influence the classifier accuracy. Therefore, we do not evaluate the classifier accuracy in comparison of the traditional ELM.

In our experiments, we only evaluate the learning speed and the speedup of ELM without affecting the efficiency of ELM. Table 1 summarizes the parameters used in our experimental evaluation, along with their ranges and default values shown in bold. In each experiment, we vary a single parameter, while setting the remainders to their default values. The speedup of a larger system with $m$ computers is measured as (15).

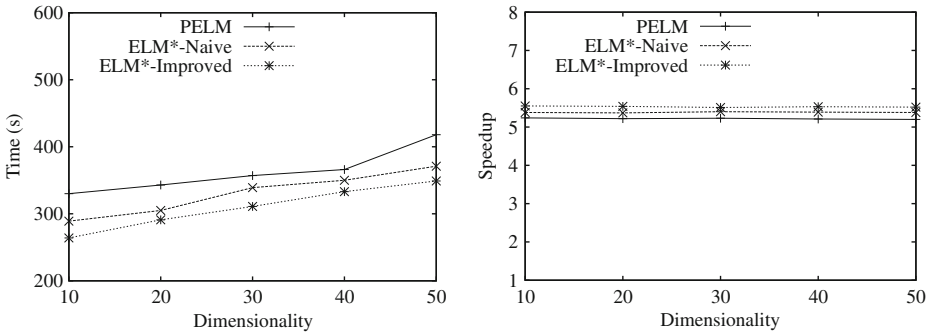
$$speedup(m) = \frac{computing\ time\ on\ 1\ computer}{computing\ time\ on\ m\ computer} \tag{15}$$

### 4.2 Experimental results

First, we investigate the influence of training data dimensionality shown in Figure 2. As shown in Figure 2a, with the change of dimensionality, the running time of our two ELM* algorithms and PELM all increase slightly. The performance of our two ELM*

**Table 1** Experimental parameters

| Parameter | Range and default |
|---|---|
| Dimensionality | 10, 20, 30, 40, **50** |
| Number of hidden nodes | 100, 150, **200**, 250, 300 |
| Number of records | 3 M(1.4 G), 4 M(1.86 G), **5 M**(2.3 G), 6 M(2.8 G), 7 M(3.27 G) |
| Number of nodes | 1, 2, 3, 4, 5, 6, 7, **8** |

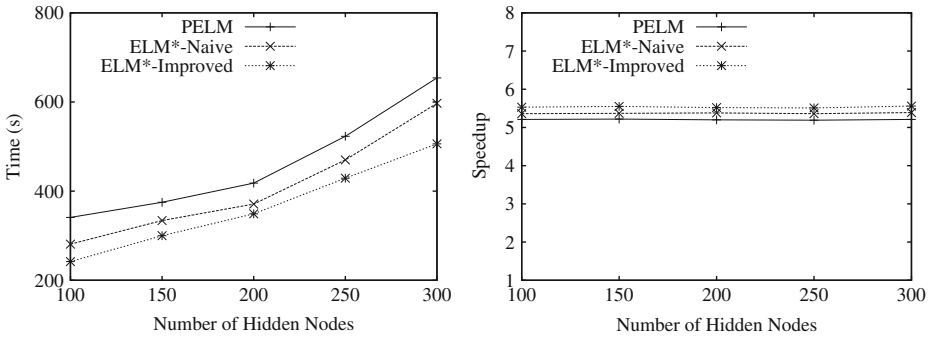



(a) Running Time (b) Speedup

**Figure 2** The influence of dimensionality

algorithms are all optimal to PELM. The performance of our improved ELM* algorithm (ELM*-Improved) is better than the naive ELM* algorithm (ELM*-Naive). With the increasing of dimensionality, the running time for calculating corresponding row $h_k$ in hidden layer output matrix **H** for each data record increases slightly so as to make the running time for MapReduce process increases slightly. With the increasing of dimensionality, the number of hidden nodes do not change, so the size of matrices **U** and **V** dose not change. That is, the amount of data transmitting in the cluster has no apparent increase, and the time of the intermediate results transmitting in cluster does not also increase. Therefore, the training time increases slightly with the increase of dimensionality. PELM uses two phases MapReduce computation to realize ELM, so the computation cost is larger than our ELM* algorithms. Our ELM*-Improved algorithm can decrease the transmitting time and enhance the performance by the reason of local summation of elements in the matrix, so its performance is better than the ELM*-Naive algorithm. Figure 2b shows the speedup of changing the dimensionality. A system with m-times the number of computers yields a speedup $m$ in theory. However,in practice, the linear speedup is difficult to achieve because the communication cost increases when the number of computers becomes large. The speedup of our two ELM* algorithms is optimal to PELM.

Then, we investigate the influence of the number of hidden nodes in ELM in Figure 3. As illustrated in Figure 3a, with the number of hidden nodes increases, the running time increases. The running time of PELM is longer than our two ELM** algorithms. The running time of our ELM*-Improved algorithm is shorter than the ELM*-Naive algorithm. That is because when the number of hidden nodes increases, the size of matrices **U** and **V** also are enlarged. That is to say, the amount of intermediate results of MapReduce is enlarged. This does not only increase the computing time of MapReduce, but also increases the transmitting time of the intermediate results in MapReduce cluster. Therefore, the training time increases with the increasing of the number of hidden nodes. PELM contains two phases MapReduce computation, so there are more intermediate results transmitting during the processing course than our ELM* algorithms. Our ELM*-Improved algorithm can save transmitting time, so its running time is shorter than ELM*-Naive algorithm. Figure 3b shows the speedup of changing the number of hidden nodes. We can see

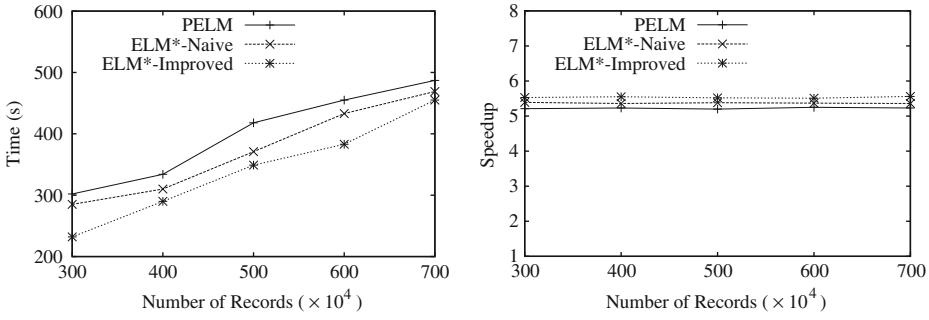



(a) Running Time (b) Speedup

**Figure 3** The influence of number of hidden nodes

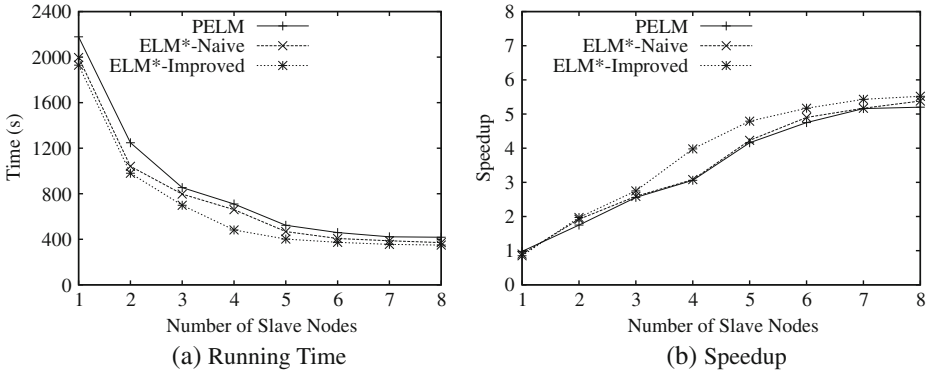
that the speedup keeps a stable status. Our ELM* can reach a higher speedup than PELM.

Next, we investigate the influence of the number of training records in Figure 4. Figure 4a shows that with the increasing of the number of training records, the running time increases. The running time of PELM is longer than our two ELM* algorithms. The performance of our ELM*-Improved algorithm is optimal to the ELM*-Naive algorithm the same as the above experiments. As the number of training records increases, the total amount of data that MapReduce needed to process is enlarged, which increases both the computing time of MapReduce and the transmitting time of the intermediate results in MapReduce cluster. Therefore, the training time increases with the increasing of the number of training records. PELM contains two passes MapReduce computation. The first MapReduce computes $\mathbf{H}$ and the second MapReduce computes $\mathbf{H}^{\dagger}$. There are huge amounts of data transmitting during these MapReduce computations, so the running time of PELM is longer than our ELM*. Local computation can save transmitting time, so the performance of our ELM*-Improved algorithm is superior to the ELM*-Naive algorithm. Figure 4b shows the speedup of changing the number of records. Although the liner speedup cannot reach, our ELM* is better than PELM.




(a) Running Time (b) Speedup

**Figure 4** The influence of number of records




(a) Running Time (b) Speedup

**Figure 5** The influence of number of slave nodes

Finally, we discuss the influence of the number of working slave nodes in the Cluster in Figure 5. Figure 5a shows that with the increasing of working slave nodes quantity, the running time decreases rapidly. However, the performance of ELM* is better than PELM. Meanwhile, our ELM*-Improved algorithm has the better scalability than the ELM*-Naive algorithm by the reason of its local computation saving the transmitting time. And it is reasonable that the increasing amount of working slave nodes makes the tasks that can be run by Map and Reduce at the same time increase. This, as a result, improves the parallelism of computing and process efficiency. Therefore, the training time decreases when the amount of working slave nodes increases. Although increasing the parallelism of the system, the transmitting cost of PELM is larger than ELM*, so the running time of ELM* is optimal to PELM. Figure 5b shows the speedup of changing the number of slave nodes. We can see that the overall trend of the speedup is close to liner growth. With the increasing of the number of slave nodes, the speedup values is far from the theoretical value because the communication cost increases when the number of computers becomes large.

In conclusion, no matter how the experimental parameters changes, our two ELM* algorithms can all rapidly complete the learning process of massive training data. Moreover, the two ELM* algorithms all have a good character of scalability, which is an efficient tool to cope with large-scale data learning and have a wild prospect of real-world applications.

## 5 Conclusions

ELM and MapReduce has an unparalleled advantage of other similar technologies, which attract widely attention in machine learning and distributed data processing communities respectively. In this paper, we combine the advantage of ELM and MapReduce, and propose a Distributed Extreme Learning Machine based on MapReduce framework (ELM*). ELM* makes full use of the parallel computing ability of MapReduce framework and realizes efficient learning of large-scale training data. Specifically, through analyzing the character of traditional ELM, we found that the most expensive computational in it is the multiplying of two matrixes. Then, we transform the multiplying of matrices into a summation form, which

suits MapReduce framework well. Furthermore, we propose a Distributed Extreme Learning Machine (ELM*) based on MapReduce framework. Finally, in the Cluster environment, we use simulation data to do a detailed validation of the character of ELM*. The experiment results show that ELM* can learn massive training data efficiently and make great sense of improving large-scale data learning applications.

**Acknowledgements** This research was partially supported by the National Natural Science Foundation of China under Grant Nos. 60933001, 61025007, 61100022, and 61272182; the National Basic Research Program of China under Grant No. 2011CB302200-G; the 863 Program under Grant No. 2012AA011004, the Public Science and Technology Research Funds Projects of Ocean Grant No. 201105033, and the Fundamental Research Funds for the Central Universities under Grant No. N110404009.

## References

1. Chacko, B.P., Krishnan, V.R.V, Raju, G., Anto, P.B.: Handwritten character recognition using wavelet energy and extreme learning machine. Int. J. Mach. Learn. Cybern. **3**(2):149–161 (2012)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of Symposium on Operating System Design and Implementation (OSDI), pp. 137–150 (2004)
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
4. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
5. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC), pp. 810–818 (2010)
6. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 29–43 (2003)
7. Ghoting, A., Krishnamurthy, R., Pednault, E.P.D., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S.: SystemML: declarative machine learning on MapReduce. In: Proceedings of the 27th International Conference on Data Engineering (ICDE), pp. 231–242 (2011)
8. He, Q., Shang, T., Zhuang, F., Shi, Z.: Parallel extreme learning manchine for regression based on MapReduce. Neurocomputing **102**(2), 52–58 (2013)
9. Huang, G.-B., Liang, N.-Y., Rong, H.-J., Saratchandran, P., Sundararajan, N.: On-line sequential extreme learning machine. In: Proceedings of the IASTED International Conference on Computational Intelligence (CI), pp. 232–237 (2005)
10. Huang, G.-B., Zhu, Q.-Y., Siew, C.-K.: Extreme learning machine: theory and applications. Neurocomputing **70**(1–3), 489–501 (2006)
11. Huang, G.-B., Chen, L., Siew, C.-K.: Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Trans. Neural Netw. **17**(4), 879–892 (2006)
12. Huang, G.-B., Chen, L.: Convex incremental extreme learning machine. Neurocomputing **70**(16–18), 3056–3062 (2007)
13. Huang, G.-B., Chen, L.: Enhanced random search based incremental extreme learning machine. Neurocomputing **71**(16–18), 3460–3468 (2008)
14. Huang, G.-B., Ding, X., Zhou, H.: Optimization method based extreme learning machine for classification. Neurocomputing **74**(1–3), 155–163 (2010)
15. Huang, G.-B., Wang, D. H., Lan, Y.: Extreme learning machines: a survey. Int. J. Mach. Learn. Cybern. **2**(2), 107–122 (2011)
16. Huang, G.-B., Wang, D. H., Lan, Y.: Extreme learning machines: a survey. Int. J. Mach. Learn. Cybern. **2**(2):107–122 (2011)
17. Huang, G.-B., Zhou, H., Ding, X., Zhang, R.: Extreme learning machine for regression and multiclass classification. IEEE Trans. Syst. Man Cybern. Part B Cybern. **42**(2), 513–529 (2012)
18. Liang, N.-Y., Huang, G.-B., Saratchandran, P., Sundararajan, N.: A fast and accurate on-line sequential learning algorithm for feedforward networks. IEEE Trans. Neural Netw. **17**(6), 1411–1423 (2006)

19. Lin, Y., Lv, F., Zhu S., Yang, M., Cour, T., Yu, K., Cao, L., Huang, T.S.: Large-scale image classification: fast feature extraction and SVM training. In: Proceedings of the 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1689–1696 (2011)
20. Panda, B., Herbach, J. S., Basu, S. , Bayardo, R. J.: PLANET: massively parallel learning of tree ensembles with MapReduce. In: Proceedings of the 35th International Conference on Very Large Data Bases (VLDB ), pp. 1426–1437 (2009)
21. Rong, H.-J., Huang, G.-B., Sundararajan, N., Saratchandran, P.: On-line sequential fuzzy extreme learning machine for function approximation and classification problems. IEEE Trans. Syst. Man Cybern. Part B **39**(4), 1067–1072 (2009)
22. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2010)
23. Sun, Y., Yuan, Y., Wang, G.: An OS-ELM based distributed ensemble classification framework in P2P networks. Neurocomputing **74**(16), 2438–2443 (2011)
24. Wang, G., Zhao, Y., Wang, D.: A protein secondary structure prediction framework based on the extreme learning machine. Neurocomputing **72**(1–3), 262–268 (2008)
25. Wang, B., Wang, G., Li, J., Wang, B.: Update strategy based on region classification using ELM for mobile object index. Soft Comput. **16**(9), 1607–1615 (2012)
26. Wang, X., Shao, Q., Miao, Q., Zhai, J.: Architecture selection for networks trained with extreme learning machine using localized generalization error model. Neurocomputing **102**(1), 3–9 (2013)
27. Witten, I.H., Frank, E., Hell, M.A.: Data Mining: Practical Machine Learning Tools and Technique, 3rd edn. Morgan Kaufmann (2011)
28. Wu, J., Wang, S., Chung, F.: Positive and negative fuzzy rule system, extreme learning machine and image classification. Int. J. Mach. Learn. Cybern. **2**(4):261–271 (2011)
29. Yang, H.C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-Reduce-Merge: simplified relational data processing on large clusters. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1029–1040 (2007)
30. Zhai, J., Xu, H., Wang, X.: Dynamic ensemble extreme learning machine based on sample entropy. Soft Comput. **16**(9), 1493–1502 (2012)
31. Zhang, R., Huang, G.-B., Sundararajan, N., Saratchandran, P.: Multi-category classification using an extreme learning machine for microarray gene expression cancer diagnosis. IEEE/ACM Trans. Comput. Biol. Bioinform. **4**(3), 485–495 (2007)
32. Zhao, X., Wang, G., Bi, X., Gong, P., Zhao, Y.: XML document classification based on ELM. Neurocomputing **74**(16), 2444–2451 (2011)
33. Zhu, Q.-Y., Qin, A. K., Suganthan, P. N., Huang, G.-B.: Evolutionary extreme learning machine. Pattern Recogn. **38**(10), 1759–1763 (2005)