Distributed algorithm to train neural networks using the Map Reduce paradigm

Cristian Mihai BARCA [1], Claudiu Dan BARCA [2],
[1] Electronics, Communications and Computers, University of Pitesti, Romania
[2]The Romanian-American University, Bucharest, Romania

With rapid development of powerful computer systems during past decade, parallel and distributed processing becomes a significant resource for fast neural network training, even for real-time processing. Different parallel computing based methods have been proposed in recent years for the development of system performance. The two main methods are to distribute the patterns that are used for training – training set level parallelism, or to distribute the computation performed by the neural network – neural network level parallelism. In the present research work we have focused on the first method.

Keywords: *Artificial Neural Networks, Machine Learning ,Map-Reduce Hadoop, Distributed System*

1 Introduction

An Artificial Neural Network (ANN), usually called neural network (NN), is a mathematical or computational model that is inspired by the structure and/or functional aspects of biological neural networks. From a biological point of view, a neural network consists of an interconnected group of artificial neurons, and it processes information using a connection approach to computation. In most cases, an ANN is an adaptive system that changes its structure based on external or internal information flowing through the network during the learning phase. Specifically and technically saying, in a neural network model simple nodes (also called neurons or units), are connected together to form a directed grap hence the term neural network. While a neural network does not have to be structurally adaptive by itself, its practical use comes with algorithms designed to alter the weights of the connections in the network to produce the desired signal flow. Nowadays, neural networks are perceived as non-linear statistical data modeling tools; in the past the linearity of the neural networks was considered an inconvenience. They are usually used to model complex relationships between inputs and outputs or to find patterns in data. Neural networks are applicable in virtually every situation in which a relationship between the predictor variables (independents, inputs) and predicted variables (dependents, outputs) exists, even when that relationship is very complex [1, 2]. In the artificial intelligence field, neural networks have been successfully applied to identification and control (vehicle control, process control), game-playing and decision making (backgammon, chess, racing), pattern recognition (radar systems, face identification, object recognition), sequence recognition (gesture, speech, handwritten text recognition), medical diagnosis, financial applications, data mining (or knowledge discovery in databases), visualization and e-mail spam filtering [3]. For such neural networks practical applications, fast response to external events within short time are highly demanded and expected. However, the extensively used gradient descent based learning algorithms obviously cannot satisfy real-time learning needs in many applications, especially large scale application and when higher learning accuracy and generalization performance are required [4]. This drawback made artificial neural networks to lose their scientific

enthusiasm, but not for long time. As the hardware performances (multithreading, multicore, big clusters and grids) began to increase so the neural networks regained interest[5].

2. Challenges, Motivation and Aim

When applied on a large scale, large scale learning system, in cases such as Netflix [6] or Google spam filters [7], machine learning mechanisms require massive amounts of computational resources. These resources are also needed when we talk about neural networks and only because of the training phase that passes through an iterative process with a very slow convergence rate. For daily problems where the training data is large, training times of the order of days and weeks are not unusual on serial computers. This has been the main problem for artificial neural network use in real-world applications and has also made its wider acceptability decrease. At first sight the graph structure of a neural network may reveal several degrees of parallelism within it, such as weight parallelism, node parallelism, and layer parallelism, but because of its strong dependencies between layers (a complete feedforward network connects weights from any node of one layer with any node of a next layer) it makes the whole problem very difficult to apply on a large scale distributed system (a network-based parallelism requires fine grained synchronization and communication – high overhead). Also, the size of the training set raises a challenge for the computational complexity of the neural network learning algorithm. This is more obvious in cases where training data is being continuously generated, and it must be absorbed quickly by the model, or else the model will become stale. One good example of such a situation is monitoring a person living at home [8, 9], with a set of sensors, in the context of AAL (Ambient Assisted Living [10]). Assuming that we are building a classifier that announces the caregivers when the monitored person is in a dangerous situation, what is normal or not can change on a daily basis, especially when the person suffers a chronic illness with complications.

Recent research has shown the possibility of using distributed computing for machine learning, known as distributed learning. While in the past the whole attention has been drawn over the neural network training with multiprocessors or multicore computers using network level parallelism or pipeline [4, 11, 12, 13], now is pointed on using large scale distributed machines with training set parallelism [14, 15]. Basically, a training set parallelism means that the training set is split across multiple processing units. Each unit has a local copy of data-subset and collects change-values (deltas) for the given training patterns. After that, the data is updated using different aggregation methods. By far, for processing large distributed data Map-Reduce seems to be the right choice – it is highly scalable and also has a great potential in distributed learning [16]. Map-Reduce is a framework introduced by Google in 2004 for processing highly distributable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster or a grid. Computational processing can occur on data stored either in a filesystem (unstructured) or in a database (structured) [17]. It allows programmers to write functional- style (using two functions map and reduce) code that is automatically parallelized and scheduled in a distributed system. Google's Map Reduce implementation runs on a large commodity cluster (simple machines wired up in a local network) and is highly scalable: a typical Map-Reduce job processes many terabytes of data on thousands of machines. Software engineers find the system easy to use: hundreds of Map- Reduce programs have been implemented and upwards of one thousand Map-Reduce jobs are executed on Google's clusters every day [18]. However, there are also cases where researchers use in

their experiments Phoenix [19] or Hadoop [16] for the implementation of Map-Reduce and in others they developed custom frameworks using PThreads or other multicore methods [20] (multicore Map-Reduce frameworks).

3. **Distributed Learning Algorithms**

On top of the parallel architecture comes the machine learning algorithm. In order to use Map-Reduce, a machine learning algorithm must be mapped on both the map-stage and reduce-stage. One parallel programming method [20] that is easily applied to many different learning algorithms is to adapt them in a certain "summation form", which allows easy parallelization on multicore and distributed computers. Algorithms that calculate sufficient statistics or gradients fit this model, and since these calculations may be batched, they are expressible as a sum over data points. Divide the dataset into as many pieces as there are cores, give each core its share of the data to sum the equations over, and aggregate the results at the end. This form of the algorithm is the "summation form" and allows adapting Map-Reduce paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms. A brief description of backpropagation's (NN learning algorithm) Map-Reduce adaptation is presented by Cheng-Tao Chu & Co. [20]. The authors define a simple network structure (with one hidden layer) and use each mapper to propagate subsets of data through the networks (each mapper has its own network) and to compute the partial gradient for each of the weights in the network when the error is back propagated. After that, reducers sums the partial gradients from each mapper and do a batch gradient descent to update the weights of the network. In this manner backpropagation performs a batch learning phase, otherwise updating a set of values after each training example creates a bottleneck for parallelization. The authors have implemented their adapted algorithm on top of a custom multicore Map- Reduce framework – their aim was to prove that the "summation form" of a learning algorithm allows an easy parallelization on multicore computers.

3.1 Research Approach

During our research we covered many scientific papers with topics about using neural networks on large scale, read thoroughly and understood different learning algorithms and also dug for several comercial neural training engines' insights. However, it is clear by now that we are going to focus on the idea of shaping a batch neural network training algorithm into a Map-Reduce form – applying the same map phase (split the data between mappers and process the partial gradient of its local network's weights) and for sure calling a complete reduce stage, instead of combiners, with the appropiate weight update function, which depends on the algorithm (we acheive parallel batch-training and parallel update).
The approach followed by this research is to:
- have a custom extensible/modular engine for neural network training (for feedforward networks) that will be easy to adapt with Hadoop's framework (we would have used Encog for this phase, but it is well encapsulated and a reverse engineering would have taken a long time)
-add backpropagation, batch backpropagation and resilient backpropagation support in the engine, error plotting mechanism and object based run configuration (set up some specific members of an object, eg. input path, maximum number of epochs to train, etc, and send it to the base class)
-adapt resilient backpropagation to a Map-Reduce form and do analysis of large data by applying multiple identical neural networks to learn several sub dataset
-use Hadoop's Map-Reduce implementation (but keeping the same interfaces, we want

both serial and distributed methods to be integrated under the same base application)
-find a different mechanism than HDFS (hadoop distributed filesystem) to pass the internal data of a neural network to a mapper/reducer (weights, errors, network structure),
e.g. Cassandra Database
-exploit at maximum capacity the reduce stage of the Map-Reduce paradigm (not just a single reducer) by performing a parallel update (almost 30% computational size) of the neural network
-find datasets for accuracy, consistency, convergence, speed and scalability tests; the datasets used in our experiments will be from publicly available projects, will cover many of neural network's practice usage (classification, detection, prediction, etc) and some of them will be large enough to demonstrate our framework's performances (obtain speedup with Map-Reduce).

3.2. Algorithms Research

As one would believe there is not a single algorithm for designing a neural network model, each of the available with its own advantages and disadvantages. The main difference between them lies in formulation of how to alter the weights of the neurons and in the relations of the neurons to their environment [4]. Though, we restrained our view just on two of them that we encountered during our research: backpropagation and resilient back-propagation.

4. Resilient Backpropagation in Map-Reduce Form

*Resilient backpropagation(Rprop)*is a learning scheme that solves the convergence's issue of the batch backpropagation algorithm. The difference between these two methods resides in the update function of the weight, which has the following features [21]:

- eliminates the negative influence of the size of the gradient on the weight step
- the direction of the weight update is indicated by the sign of the gradient

We know from [21] he previous sections that the resilient backpropagation algorithm respects the summation form (batch learning), so now let us examine each stage of our rprop Map-Reduce form like we did in the word count example (Figure 1):

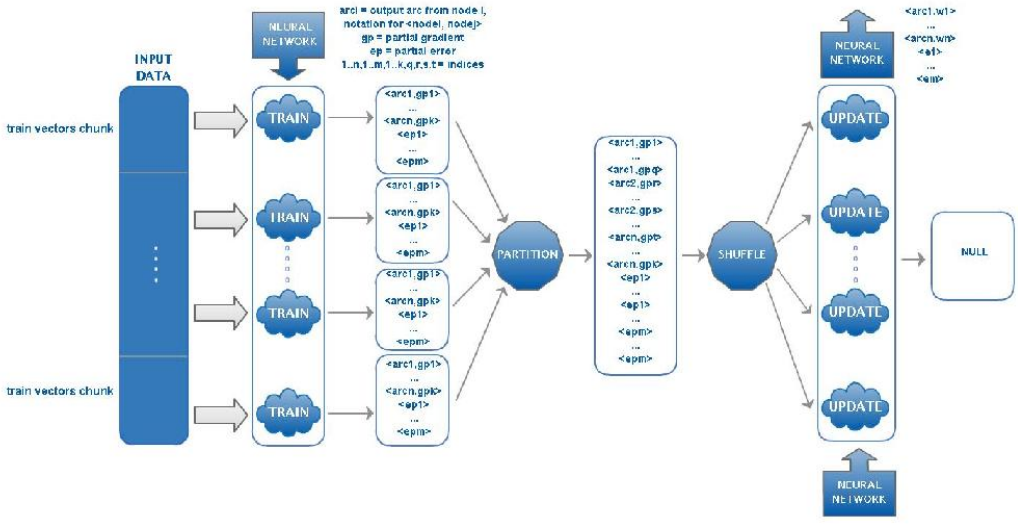

Fig. 1. Map-Reduce applied on resilient backpropagation algorithm

Data split
-the input data is split in equal size chunks of training vectors ('\n' separator)
Map
- each chunk is taken by a mapper and independently trained with the updated neural network pulled from the system – partial gradients are computed for every arc of the neural network and also the partial sum of the output neurons' squared error; each mapper outputs pairs of values <arc i, gp> - where arci is an output arc from nodei to the nodej (arci = <ni,nj>), gp the partial gradient associated with arc i - and pairs of values <epx> meaning the partial sum of the output neuron nx squared errors
Partition
- when the map stage is finished the collection of output pairs <arci, gp> is partitioned by the neuron ni defined in arci ; we will have as many list of pairs <arci, <gp, … gp>> as different neurons ni are in the collection; analogue for <epx>
Shuffle
- each list of <arci, <gp, … gp>> is sent to only one available reducer
Reduce
- sums (aggregates) the partial gradients from the given list to compute the total gradient of its correspondent arc; the final gradient is used to update the arc's weight to a new value that will be (along with the rest of the results from the other reducers) saved into the system in order to be used in the next training epoch.

5. The classes Driver – Mapper – Reducer

Basically, our three main classes (Driver – Mapper – Reducer) respect the same structure like those presented above, but it is obvious that they are more complex (figure2). We are now going to briefly explain them from a technical point of view in the following paragraphs:

Driver: Our driver reads the run parameters XML file uploaded by the user on the cluster (manually or using the GUI client) and then creates a network object structure (using the same RunParams class that fnn uses) based on the specified values from the file (input neurons, number of middle layers, etc). In the same time a shorter XML file with just the names of the experiment and network is copied to Hadoop's Distributed Cache [22]. Mappers need to know from where they are going to read the network structure and the updated weights' values during their phase (the network and experiment names are names of Cassandra column families). On start, the neural network is initialized with random values and is pushed into the NetStruct Cassandra's column family for further use in the next phases. One step before pushing the network we try to establish a connection to the database and also we check the presence of the key space we use. If there is no such keyspace a new one with the default name 'mrtsdb' is created automatically. After that, our driver starts the map-reduce cycle by launching computational jobs until it achieves the desired network's error or the maximum number of epochs (both values are taken from the XML file). At the end of the cycle we save the resulted neural network in Cassandra's NetSave column family.
Mapper
In the current framework's version our mapper takes the input from HDFS (in a future version it may also take the input directly from Cassandra Database). We use the default *TextInputFormat* to get chunks delimited by the carriage return symbol ('\n'). Each chunk represents an equal set of training vectors delimited by "#" symbol – the dataset is manually parsed prior to its upload on HDFS. Before reading the input data we use the file copied to the Distributed Cache to know the location of the network structure with the updated values from Cassandra (or the initial ones if is the first map-stage run). From an input chunk we extract the training vectors and we train them with the updated local network (using

fnn component in a lightweight version – without plotting features). Once an epoch has finished we collect the partial gradients and the sum of squared errors, values that we pass in a pair <key,value> formation to the following stage.

Reducer

Our reducer receives the list <key, <values...>> partitioned and sorted by the shuffle and sort stage. A key means a node and the values represent serialized internal objects which contain information (weight value, gradient, and so on) about the arcs that start from the corespondent key- node. The only things that a reducer does are: aggregates data to obtain a global value (upon all the mappers) and updates the weights or the output node squared error, using the previous arc values and the global computed value (we query Cassandra to retrieve those data).

6. Workflow

The initial workflow was based on a simplified diagram modelled with a natural language notation (figure 3). After we went gone deeper into the implementation, we reshaped the workflow using proper technical notations (pair notations for the get and put operations implemented in cassdb component – figure 4). It helped us to define exactly the way we want our software framework to work, without knowing the hardware infrastructure we are going to experiment on. We also used this diagram to separate concerns between custom components, for example: client (mrts-client), database support (cassdb), import-export (mrts-io), driver – mapper – reducer (mrts), and so on.

In a real scenario our project works as follows:

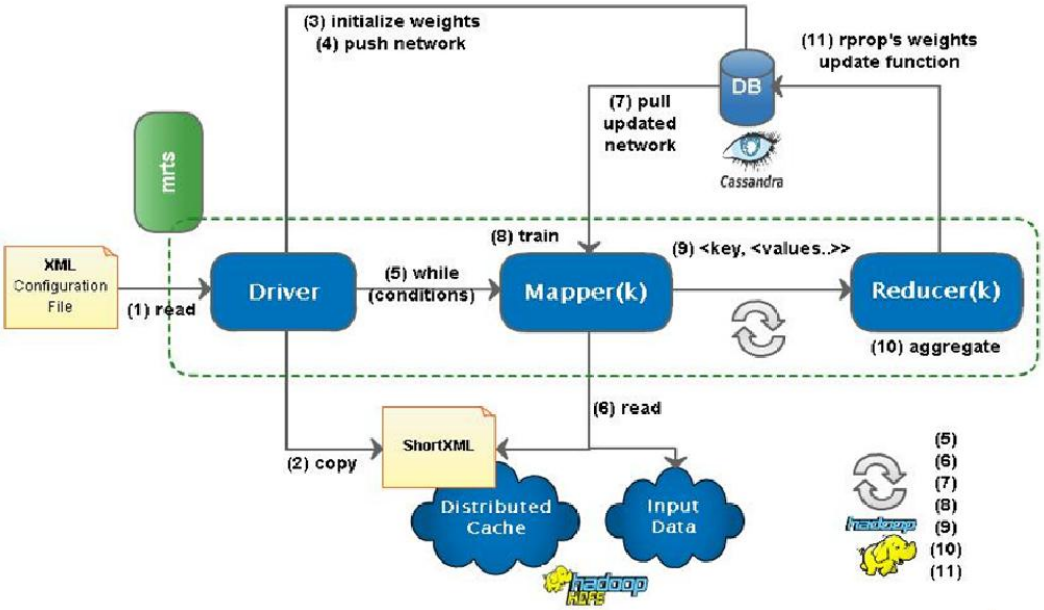

Fig. 2. Mrts component (internal workflow, driver-mapper-reduce cycle)

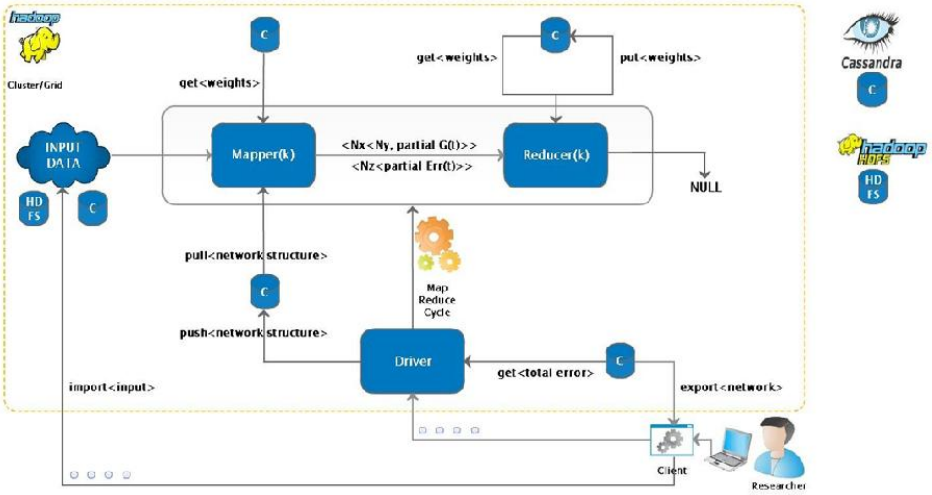

Fig. 3. Basic Workflow – without technical notations

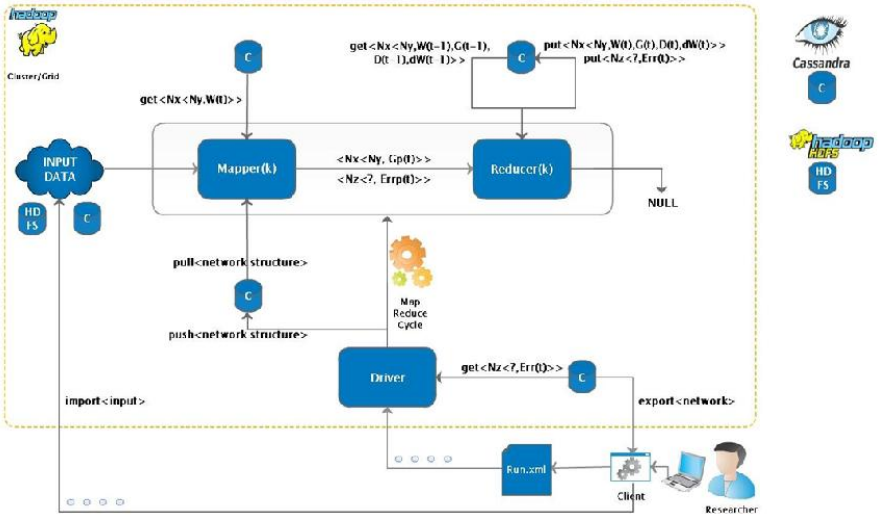

Fig. 4. Extended Workflow – with specific get/put notations (Symbol "?" represents a wildcard integer value bellow or equal to 0)

• A researcher sets an experiment and starts the training process. The experiment is defined using a GUI – we set the neural network structure (number of layers and neurons), training parameters (maximum number of epochs, neural network target error, etc.), the data input path and the data output path (for extracting and saving the network on local computer – in CSV [23] or a serialized format). All of the experiment's configuration, including the network's structure, will be written to a .xml formatted file, which will be sent to the Hadoop cluster along with the start command.

• The request to start the training process is received by the Driver which will have the job to:
- push the network structure in Cassandra
- initialize the network's weights and save them in Cassandra

- start the Map-Reduce cycle and run it while we have get the expected network error or the maximum number of epochs
• A Map-Reduce cycle, as we already know, has two components, each with its own set of tasks to perform:
Mapper:
- creates a local neural network using the structure and the weights' values from Cassandra:
- receives the training vectors chunk
- trains the current neural network over the given data
- outputs the partial gradient values and errors
Reducer:
- receives the partial gradient values or the partial output error and sums them up to compute the total values (total gradient or sum of squared errors)
- now that we have the final values, the weights and the output errors (all determined by the given list of pairs), the reducer updates and saves them in Cassandra
- the HDFS output of a reducer is set to NULL (we used Cassandra for output)

Conclusions

We developed an extensible framework (client-server application) for training neuralnetworks using the Map-Reduce paradigm.
Our serial implementation of resilient back propagation achieved similar accurate results compared with Encog.
Our convergence tests applied on the researched algorithms showed that the resilient backpropagation algorithm was the right choice for our purposes: batch training and good convergence rate.

References

[1] Statsoft – Neural Network. Url: www.statsoft.com/textbook/neural-networks/ button=2
[2] Wikipedia – Artificial Neural Network. Url:en.wikipedia.org/wiki/Artificial_neural_network
[3] Neural Networks Applications. Url: www.peltarion.com/doc
[4] Kiran Kumar Kaki. Parallelized Backpropagation Neural Network Algorithm using Distributed System. Master thesis, Thapar University, Patiala, January 2009.
[5] Lungu Ion, Bâra Adela, Căruţaşu George, Pîrjan Alexandru, Oprea Simona-Vasilica, Prediction intelligent system in the field of renewable energies through neural networks, Journal of Economic Computation and Economic Cybernetics Studies and Research, Vol. 50, No. 1/2016, pp. 85-102, ISSN online 1842– 3264, ISSN print 0424 – 267X.
[6] Netflix. Url: www.netflixprize.com
[7] Google spam. Url: www.google.com/mail/help/fightspam/spamexplained.html
[8] Vikramaditya Jakkula and Diane J. Cook. Detecting Anomalous Sensor Events in Smart Home Data for Enhancing the Living Experience. Artificial Intelligence and Smarter Living — The Conquest of Complexity: Papers, AAAI Workshop, 2011.
[9] Vikramaditya R. Jakkula, Diane J. Cook, and Aaron S. Crandall. Temporal pattern discovery for anomaly detection in smart homes. Proceedings of the 3rd IET International Conference on Intelligent Environments (IE 07), Germany, 2007.
[10] Wikipedia - Ambient Assisted Living. Url: en.wikipedia.org/wiki/Assisted_living
[11] Jigisha Gandhi, Shitanshu Perekh. Deployment of Neural Network on Multi-CoreArchitecture. International Journal of Engineering Research & Technology (IJERT)Vol. 1 Issue 3, May 2012.

[12] Markus Scholz, Prof.Dr.rer. nat. Peter F. Stadler. Development of an artificial neural network on a heterogeneous multicore architecture to predict a successful weight loss in obese individuals. Bachelor thesis, Leipzig, March 2008.
[13] Honghoon Jang, Anjin Park, Keechul Jung. Neural Network Implementation using CUDA and OpenMP. Dicta, 2008.
[14] Kritsanatt Boonkiatpong and Sukree Sinthupinyo. Applying Multiple Neural Networks on Large Scale Data. International Conference on Information and Electronics Engineering IPCSIT vol.6, IACSIT Press, Singapore, 2011.
[15] Quoc V. Le and Co. Building High-level Features Using Large Scale Unsupervised Learning. Appearing in Proceedings of the 29th International Conference on Machine Learning, Edinburgh, Scotland, UK, 2012.
[16] Dan Gillick, Arlo Faria, John DeNero. MapReduce: Distributed Computing for Machine Learning. 2006.
[17] Wikipedia – Map-Reduce. Url: en.wikipedia.org/wiki/MapReduce
[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing onLarge Clusters. Google Inc. OSDI, 2004.
[19] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. Computer Systems Laboratory, Stanford University, 2007.
[20] Cheng-Tao Chu et al. Map-Reduce for machine learning on multicore. NeuralInformation Processing Systems (NIPS) Conference, 2007.
[21] Martin Riedmiller. Rprop – Description and Implementation Details. Technical Report, January,1994.
[22] Hadoop Map-Reduce Tutorial. Url:hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html
[23] OpenCSV. Url: opencsv.sourceforge.net

Cristian Mihai BÂRCĂ graduated from Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest in 2012, and holds a master degree in Parallel and Distributed Computer Systems, Vrije Universiteit Amsterdam since 2014.His scientific fields of interest and expertise include database systems and Web Technologies

Claudiu Dan BÂRCĂ graduated from Faculty of Computer Science for Business Management, Romanian American University in 2007, and holds a master degree in Economic Informatics since 2008 and a PhD in the field of Engineering Sciences since 2013. He is an assistant lecturer within the Faculty of Computer Science for Business Management having nine years of teaching experience. He also has good research and publishing activity: he was a member of the research teams of international and national projects. His core competences are in software programming and connected areas.