City University of London

School of Mathematics, Computer Science & Engineering

M.Sc. Data Science

Project Report

Parallelized, multi-CPU, multi-node training for Restricted Boltzmann Machines

Supervisor:
Dr. Tillman Weyde

Student:
Nikolay Manchev

January 2017

Contents

List of Figures

List of Tables

Declaration

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed: **Nikolay Manchev**

Abstract

The aim of this project is to look into parallelization of Restricted Boltzmann Machines, their scalability and suitability for processing large data sets in the context of "big data".

The research question this project looks into is: **Can a CPU-based, parallelized version of the Restricted Boltzmann Machine be developed and how does it compare to the standard, non-parallelized implementation?**

We develop a custom Restricted Boltzmann Machine that can offload distributed scalar and matrix operations to Spark and MapReduce, and we test the implementation functionally (as feature extractor and as participant in a classification machine learning pipeline) and in terms of scalability and performance. We compare the parallelized multi-node runtime versus single-node execution, and we also compare the results with a GPU-accelerated RBM implementation.

We try to determine the relationship between data volumes and cluster nodes, and the impact they have on the training times of the Restricted Boltzmann Machine. We also look at a price-performance ratio, comparing parallelized CPU-based RBMs versus single-node GPU-based training.

We unambiguously show that parallelized RBMs can handle larger datasets and that introducing more cluster nodes (to a certain threshold) has a positive impact on the training times. We also demonstrate that the Spark platform is better suited for RBM training compared to MapReduce, and that simply offloading operations to graphic processors does not lead to substantial performance increase.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor Dr. Tillman Weyde for always being responsive, ready to provide assistance, constantly engaging, and sharing deep insights over the course of the project. His encouragement and excellent suggestions on the direction of the work kept me motivated and greatly influenced the quality of my work.

Furthermore I would like to thank all the outstanding lectures in the M.Sc. Data Science Programme. Modules like Introduction to Data Science, Big Data, Machine Learning, Visual Analytics, and RMPI to name a few, built the foundations for this work and sparkled my interest in many other aspects of the Data Science field.

Finally, I'd like to thank the teaching assistants, technical staff, and the administration of City University of London for their patience, support, and readiness to help in any possible way.

1 Introduction and Objectives

In this section we introduce the problems associated with processing large amounts of unstructured data, and we present the Restricted Boltzmann Machine – a generative stochastic neural network that can impose structure and reduce the complexity of unstructured and semi-structured data sets. We highlight certain obstacles to using Restricted Boltzmann Machine in a "big data" setting, and we present existing approaches for paralellizing the Restricted Boltzmann Machine's training process.

We also establish the main research question that the project aims to investigate, and we list a set of of precisely defined objectives that can provide validation and empirical data to support the investigation process. This chapter also outlines our work plan and provides a high-level description of the report structure.

1.1 Restricted Boltzmann Machines for Big Data: Problem background

In this age of "big data" we have been observing an unprecedented growth in the amount of data that are available for analysis. Kitchin (2014) estimates that the data growth will follow an exponential increase for the foreseeable future.

At the same time, "data variety" is a key characteristic of big data, postulating that the type of data in a big data context is not strictly limited to relational or structured, but it varies between different formats (structured, unstructured, and semi-structured). According to McKinsey Global Institute (2011), the difficulties associated with handling unstructured and semi-structured data prompt organisations to discard substantial amounts of data, which they deem too difficult to process.

One way to mitigate the difficulties around unstructured data processing is to impose a structure on the data, thus enabling easier processing by use of familiar tools. Artificial Neural Networks (ANN) have been successfully used in this manner (Isakki & Rajagopalan 2001), as they can extract features in unsupervised fashion and detect non-linear patterns in semi-structured datasets.

Restricted Boltzmann Machines (RBM), a type of stochastic ANN, have been shown to have a wide range of applications in this context. They have been successfully used as generative models for dimensionality reduction (Hinton & Salakhutdinov 2006), collaborative filtering (Salakhutdinov et al. 2007), classification (Larochelle & Bengio 2008), and extraction of semantic document representation (Xing et al. 2005).

Training a Restricted Boltzmann Machine in the context of "big data", however, turns out to be problematic. When operating with millions and billions of parameters, the parameter estimation process for a conventional, non-parallelized Restricted Boltzmann Machine can take weeks (Raina et al. 2009). The constraint of using a single machine for fitting the model

introduces an additional limitation, which negatively impacts the scalability of the model.

There have been numerous attempts to develop a model for training Restricted Boltzmann Machines in parallelized fashion, the majority of which are based on computing with graphical processing units (GPUs).

Raina et al. (2009) have demonstrated that using parallelized GPU-based implementation of Restricted Boltzmann Machines can reduce the training time for a deep belief network model with 100 million parameters from several weeks to one day. Other studies have have shown that parallelized GPU implementation can outperform a CPU based RBM training by a factor of 2 (Scanzio et al. 2010, Vesely et al. 2010).

1.1.1 Limitations of GPU-based parallelization

Although GPU-based parallelization appears to provide significant advantage in reducing RBM training times, it also introduces certain limitations.

Raina et al. (2009) point out that copying data from RAM to the GPU's shared memory introduces a bottleneck – it appears that only 0.5% of the time required for a parallelized matrix multiplication goes to the actual computation. The remaining 99.5% is spent on moving data between RAM and shared GPU memory.

As GPUs are also limited in terms of on-board memory, this limitation propagates and negatively impacts the size of the Restricted Boltzmann Machine. A 4GB of shared memory, which is a considerable amount for a GPU architecture, can only fit up to 1 billion parameters – a modest amount in the context of deep networks (Zhu et al. 2013).

Stacking multiple GPUs together also seems to be inefficient due to the communication induced overhead. Zhu et al. (2013) suggest a new algorithm that uses memory slicing, sequentially feeding batches of data to the shared memory, thus increasing the size of the model that can be trained. This algorithm, however, is still susceptible to the latency issues arising from memory transfer operations. Zhu et al. acknowledge that memory transfer is so time consuming that it can completely negate the performance gain introduced by GPU-based computations.

These limitations have also been identified by Ly et al. (2008) who observed a "significant overhead" gone into memory transfers and threads synchronization. This experience leads them to admit that the "implementation may not be capable of being efficiently scaled" – a problem that will essentially prevent the application of GPU training for large scale problems.

There is also the concern of increased economic costs associated with using GPU-based architectures. An AWS EC2 instance with a single GPU is over 30% more expensive than an equivalent CPU-only configuration[1].

[1]Comparing a *c3.2xlarge* and *g2.2xlarge* per hour usage cost as listed at `https://aws.amazon.com/ec2/pricing/`. As of the writing of this report, the current per hour price for an *c3.2xlarge* instance is $0.478 versus $0.702 for a *g2.2xlarge* instance.

It appears that because of the aforementioned limitations most of the research around parallel RBM training is focused on a single GPU to CPU comparison (Raina et al. 2009, Zhu et al. 2013, Ly et al. 2008, Wang et al. 2014). However, a number of workarounds have been introduced in an attempt to circumvent said limitations.

Chen et al. (2012) present a pipelined approximation to back-propagation, which parallelizes the computation with respect to layers. This approach avoids the repeated copying of data latencies by distributing the layers across GPUs. The authors demonstrate a 3.3 times end-to-end speed-up, however the number of GPUs is directly tied to the number of layers in the neural network, thus introducing a degree of inflexibility in the architecture.

Another approach presented by Coates et al. (2013) implements parallelized computation by initially partitioning the training data, and then assigning a dedicated GPU to each individual partition. This method, however, is more suitable for convolutional neural networks and not beneficial in fully connected networks.

1.2 Objectives and beneficiaries

The research question this project aims to answer is: **Can a CPU-based, parallelized version of the Restricted Boltzmann Machine be developed and how does it compare to the standard, non-parallelized implementation?**

To answer this question we have established a set of objectives:

- Develop a parallelized Restricted Boltzmann Machine implementation, that can be run in a standalone (one node) or distributed (Hadoop) environment
- Prove that the implementation is functionally sound by testing its ability to learn features in an unsupervised fashion
- Optionally: Prove that the implementation can integrate with existing machine learning algorithms, and that such combined pipeline exhibits the benefits expected by including an RBM in the process (e.g. reduced training times, increased accuracy if using classifiers etc.)
- Test the implementation in terms of performance and scalability and evaluate the impact of using distributed versus standalone environment
- Optionally: Test the implementation on a standalone, but GPU-enhanced infrastructure and see the impact of GPU versus multi-node CPU-based training
- Optionally: Look at the price-performance ratio when processing data with a GPU-based RBM versus multi-node Hadoop-based RBM.

The product of the work will be a parallelized RBM implementation that can run on a clustered computing framework (Hadoop), combined with a thorough analysis of its performance. We will look at the impact of parallelization by gathering performance statistics while running the RBM and contrasting this to the same RBM implementation running on multiple clustered machines.

The project will contribute to the body of knowledge of artificial neural networks, especially in the context of "big data". It will provide insight on how suitable for CPU-based, multi-node parallelization the RBM model is, and potentially demonstrate improvements that can help with one of the biggest limitations of Restricted Boltzmann Machines – the considerable amount of time required in their training phase.

This project would not limit the beneficiaries to researchers in the area of ANN only. Ideally, a parallelized RBM implementation could help any individual or organisation who deal with unstructured data and big datasets. As of the writing of the report there is still no RBM implementation available for Apache Spark, although requests have been logged (*Add Restricted Boltzmann machine(RBM) algorithm to MLlib* 2016) for developing one for the Spark's machine learning library. The Apache Mahout project (a MapReduce-based Machine Learning platform) have been working on an RBM implementation since 2010 and it appears that this is still work in progress (*Apache Mahout: Boltzmann Machines* 2016). The large-scale distributed machine learning platform SystemML, which can offload execution to both Spark and MapReduce, also currently lacks an RBM implementation (*SystemML Algorithms Reference* 2016).

1.3 Work plan

During this research we strictly follow the project plan outlined in the project proposal and we strictly adhere to the four key milestones:

- Milestone 1 – completion of literature review, dataset selection, and the clustered cloud instances provisioned
- Milestone 2 – completion of a working implementation of a parallelized RBM with optimal network topology
- Milestone 3 – completion of all experiments (e.g. running the RBM on a single node, multiple-nodes, testing impact of parameters change etc.)
- Milestone 4 – completion of the report

A detailed breakdown of the work plan is shown in Figure 1.

The only deviation from the project proposal is in the optional "GPU versus multi-node CPU comparison". As library incompatibility prevented SystemML from working on the Amazon provisioned cloud instance, we had to resort to one of our backup options. Instead of

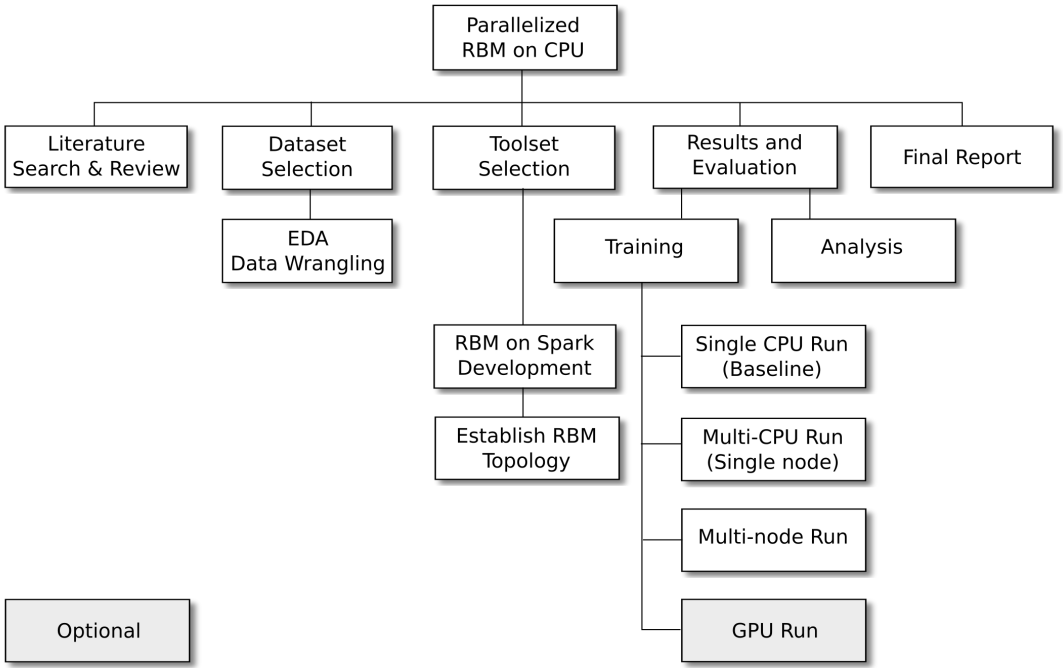


Figure 1: Work breakdown

testing the custom developed RBM training model on a GPU-enabled hardware, we tested a third-party Theano-based RBM implementation from DeepLearning[2] (see *DeepLearning 0.1 Documentation – Restricted Boltzmann Machines* 2016). The technical reasons behind this decision are discussed in detail in Section 3.3.

1.4 Report structure

In this chapter of the report we outlined the objectives and introduced some of the limitations of the existing Restricted Boltzmann Machine implementations. We introduced GPU-assisted RBM learning, and we covered the objectives, beneficiaries, and work plan for the research.

In **Chapter 2 – Context** we give context for the work that was carried out. We introduce the Boltzmann Machine model and the concept of Boltzmann Learning. We talk about certain optimisations (e.g. Gibbs sampling for avoiding the computation of the partition function) and we highlight certain disadvantages of the Boltzmann learning rule. We present the Restricted Boltzmann Machine and the improvements delivered by Contrastive Divergence Learning. We also discuss the two widely popular distributed processing platforms - MapReduce and Apache Spark. We discuss the problem of optimal execution plan generation in distributed environments and present SystemML as our choice of platform for compiling and executing machine learning algorithms in a distributed fashion.

In **Chapter 3 – Methods** we present and discuss our custom Restricted Boltzmann Machine implementation. We discuss selected elements of the implementation and we introduce

[2]DeepLearning is a set of freely available Theano tutorials published and maintained by LISA lab.

the dataset selected for its functional testing (MNIST). This section contains details on our functional testing approach, the optimal parameters selection, and execution monitoring. It also covers the RBM testing in a machine learning pipeline with two classification algorithms – Support Vector Machines and Naïve Bayes. We report on the results of the functional testing by showing a subset of learnt features and we show changes in accuracy and convergence times when our RBM is introduced in a machine learning pipeline. This section also introduces the Theano GPU-based RBM implementation from DeepLearning, which we also subject to functional testing.

As part of our methods we also cover the selection of the dataset that we use for the scalability testing. We go over the exploratory data analysis and preparation of the data, and we present a series of custom scripts that had to be developed to extract and reconcile different aspects of the data, and also to bring the dataset into a format that is suitable for processing with a Restricted Boltzmann Machine.

We finish the section by discussing the execution environment for the multi-node and GPU-based testing (i.e. overall architecture and specific hardware details), and we also explain our approach to running experiments and statistics collection.

In **Chapter 4 – Results** we present the raw data from running a series of experiments with the selected dataset, the custom RBM, and the GPU-based RBM implementation. We show the impact of changing different model parameters (e.g. the size of the mini-batches) and we look at the performance of the custom RBM when executed using two different back-ends – MapReduce and Spark.

We investigate the scalability of the implementation by varying the number of observations in the dataset, and we also look into how the different back-ends utilise clustered resources (e.g. CPU and memory).

We measure the impact of parallelization by starting with a single executor (a non-parallelized single-node environment) and gradually increase the number of nodes the cluster is allowed to use. We perform a statistical analysis in an attempt to understand the relationship between number of participating nodes and execution times. We also repeat the experiments with the GPU-based RBM and present the scalability and performance results from these experiments.

In **Chapter 5 – Discussion** we evaluate the results in light of our objectives. We highlight our key findings and discuss other aspects of the problem, considering the additional knowledge gained as part of the work so far. We present the answer to our research question and we show that we managed to experimentally confirm certain assumptions on the effectiveness of the RBM training. We present certain surprising shortcomings of the RBM training algorithm used in the experiments and we draw conclusions on the effectiveness of parallelization of the Restricted Boltzmann Machine training process in general.

Chapter 6 – Evaluation, Reflections, and Conclusions contains an evaluation of the

project and discusses the work that has been carried out as part of the research.

We summarize the findings and we present the contributions of the project such as increasing the understanding of the impact of parallelization in the context of training Restricted Boltzmann Machines, and we cover the acceptance of our custom RBM implementation as part of the Apache SystemML project.

We report on how closely the work followed the project plan and where we had to deviate and use mitigation strategies. We also list a potential follow up areas that appear to be good candidates for future work.

1.5 Conclusion

In this chapter we established the research question the project aims to address – **Can a CPU-based, parallelized version of the Restricted Boltzmann Machine be developed and how does it compare to the standard, non-parallelized implementation?**

We justified the relevance of the topic with the recent increase of interest in processing of high volumes of unstructured and semi-structured data, where Restricted Boltzmann Machine can bring value because of their applications in classification, dimensionality reduction, collaborative filtering etc.

We looked at various examples of parallelized RBM learning using graphic processors, and we explained the limitations imposed by the GPU architecture that constrains the scalability of this approach.

2 Context

In this section we present the existing state of knowledge based on the literature review, and we look at theoretical and practical implications of developing and using parallelized Restricted Boltzmann Machines. We first look at the Boltzmann Machine and Boltzmann learning, highlighting certain disadvantages of the model (e.g. sensitivity to statistical errors, computational complexity), and we discuss specific benefits that arise from imposing topological restrictions and training using Contrastive Divergence.

We also report on possible execution platforms (MapReduce, Apache Spark), and we discuss the problem of optimal plan generation induced by changes in the cluster topology. This is an important issue, as looking at the scalability of the implementation requires a set of tests with various cluster configurations. We overcome this obstacle with a layer of abstraction – SystemML, a large scale machine learning platform that runs on top of MapReduce and Spark and transparently handles optimal plan generation.

2.1 The Boltzmann Machine

A Boltzmann Machine (BM) is a stochastic recurrent neural network initially introduced by Ackley et al. (1985).

The Boltzmann Machine is similar in structure and learning algorithm to the Hopfield network – its building blocks are neuron-like units, and the machine is represented as a complete undirected graph.

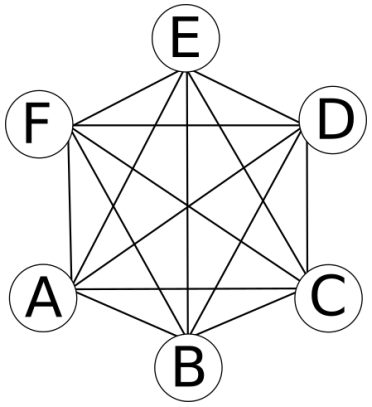


Figure 2: A graphical representation of a Boltzmann Machine with six symmetrically connected units.

Unlike the Hopfield network, the Boltzmann Nachine uses a probabilistic update rule – each node switches states based on a probability function. The consequence is that the network is not always forced to switch to a lower energy state. Instead, it is allowed to transition to higher energy states, thus escaping local minima on the energy surface.

The probability function is chosen in such a way that it encourages state changes that lead to great reduction of the overall energy. On the other hand, if the transition does not lead to a significant gain in reduction of energy, the probability of switching states is much lower.

This concept is borrowed from metallurgy where initially the system is brought into a high energy state by melting the metal. The temperature is then slowly reduced until a stable, low energy configuration is reached.

Building up on this idea, the probability function of the Boltzmann Machine also features a "temperature" parameter, which is initially set to a high value and gradually decreased until the system reaches a thermal equilibrium. At higher temperatures the system is much more likely to jump to a higher energy state. As the temperature is lowered, the probability of transitioning to a lower energy state increases, eventually approaching 1. This probabilistic technique for approximating the global minimum is often referred to as "simulated annealing".

The global energy, E, of a Boltzmann Machine is expressed as:

$$E = -\Big(\sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i s_i\Big) \tag{1}$$

Where:

- s_i is the state of unit $i \in \{0, 1\}$
- w_{ij} is the weight between units s_i and s_j
- θ_i is the bias of unit i

The "energy gap" of unit i is the difference in E that results from unit i being "off" versus "on", and it is given as:

$$\Delta E_i = \sum_j w_{ij} s_j + \theta_i \tag{2}$$

ΔE_i can also be expressed as the difference of the energies in the two possible states:

$$\Delta E_i = E_{i=off} - E_{i=on} \tag{3}$$

Each unit's probability of transitioning to state of lower energy is given by the following probabilistic rule

$$p_{i=on} = \frac{1}{1 + e^{-\frac{\Delta E_i}{T}}} \tag{4}$$

where T is an artificial notion of the *pseudotemperature* of the system.

The Boltzmann Machine is run by sequentially updating the units according to Equation 4. Repeating the update process long enough allows the network to reach a stationary distribution (thermal equilibrium).

The network can settle into one of a large number of global energy states (E), however the probability of a specific state is solely determined by the temperature and energy of the state vector and not by the machine's initial state.

The probability of the network settling in an energy state E_i can be expressed as

$$p_i = \frac{1}{Z} e^{-\frac{E_i}{k_B T}} \tag{5}$$

where k_B is the Boltzmann constant and Z is the partition function (sum of the energies of all possible state vectors), which is defined as

$$Z = \sum_i e^{-\frac{E_i}{k_B T}} \tag{6}$$

Equation 5 represents a canonical (Gibbs) distribution, which is often seen in statistical mechanics.

Following the Gibbs distribution we can conclude that as the network approaches thermal equilibrium (i.e. the probabilities of energy states no longer change), states with lower energy become more probable.

2.2 Boltzmann learning

The Boltzmann Machine comprises neurons that operate in binary fashion – they are either "on" (+1) or "off" (-1). The neurons are split into two functional groups: hidden and visible, as shown in Figure 3.

The choice of hidden/visible units is often arbitrary as the Boltzmann Machine does not have a clear distinction between layers in contrast to other neural network topologies (e.g. feedforward neural network).

The visible units serve as an interface to the machine and can be *clamped* – i.e. their state is kept fixed. The hidden neurons always operate freely.

"Learning" a Boltzmann Machine means adjusting the parameters of the network in such way that the probability distribution the machine represents fits the training data. The learning process occurs in two phases[3]:

- *Positive phase* – In this phase training data from $\boldsymbol{X} = \{x_1, x_2, ..., x_N\}$ is clamped to the network visible nodes and the network operates without the ability to change this subset of the state vector

[3] As described by Haykin (1998)

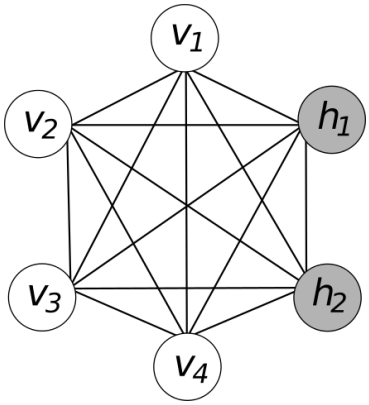


Figure 3: A Boltzmann Machine with four visible $V = \{v_1, v_2, v_3, v_4\}$ and two hidden $H = \{h_1, h_2\}$ units.

- *Negative phase* – In this phase the network is allowed to operate freely and no nodes are kept in a clamped condition

Using equations 5 and 6 we can establish a log-likelihood function, given the model parameters $\boldsymbol{w}$ and a training example $\boldsymbol{v}$

$$\begin{aligned} L(\boldsymbol{w}|\boldsymbol{v}) = \ln p(\boldsymbol{v}|\boldsymbol{w}) &= \ln \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \\ &= \ln \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} - \ln \sum_{\boldsymbol{v},\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \end{aligned} \tag{7}$$

Differentiating $L(\boldsymbol{w}|\boldsymbol{v})$ with respect to w we obtain:

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}} \left(\ln \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \right) - \frac{\partial}{\partial \boldsymbol{w}} \left(\ln \sum_{\boldsymbol{v},\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \right) \tag{8}$$

Ackley et al. (1985) show that (8) simplifies to

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{T} \left(p_{ij}^{+} - p_{ij}^{-} \right) \tag{9}$$

Where p_{ij}^{+} is the probability of units i and j both being "on" with the network operating in positive phase, and p_{ij}^{-} is the probability of units i and j both being "on" with the network operating in negative phase.

We can use (9) to construct an update rule for gradient ascent-based learning.

$$\Delta w_{ij} = \eta \left(p_{ij}^{+} - p_{ij}^{-} \right) \tag{10}$$

where η is a learning rate parameter.

The update rule is applied in both the positive and negative phases.

An interpretation of the two phases as given by Beale & Jackson (1990) is that in the positive phase, weight that connect units that are both "on" are increased (or "strengthened"). In the negative phase, after the network has reached thermal equilibrium, the weights of *any* two units that are "on" is decreased, allowing the network to "unlearn" poor associations.

Although the update rule for the BM is straightforward it presents a challenge in terms of computation – the probability distributions required in the positive and negative phases can only be obtained after computing the partition function Z. As evident in equation (6), the computation of Z requires a summation over an exponential number of possible configurations.

2.2.1 MCMC-based gradient computation

An approach that avoids the computation of the partition function is to use a simple Gibbs sampling – a simple Markov Chain Monte Carlo-based algorithm for producing samples from the joint distribution of multiple random variables.

Since we know how to compute the probability of each neuron based on the states of all other neurons in the network (see equation 4), we can use Algorithm 1 to sample the state space distribution instead of computing it using the partition function.

Algorithm 1 Gibbs sampling algorithm

1: Draw $x^{(0)}$ from the state space
2: **repeat**
3: **for** iteration $i = 1, 2, ...$ **do** ▷ x^i is sampled using eq. (4)
4: $x_1^{(i)} \sim p(X_1 = x_1 | X_2 = x_2^{(i-1)}, X_3 = x_3^{(i-1)}, ..., X_D = x_D^{(i-1)})$
5: $x_2^{(i)} \sim p(X_2 = x_2 | X_1 = x_1^{(i)}, X_3 = x_3^{(i-1)}, ..., X_D = x_D^{(i-1)})$
6: $\vdots$
7: $x_D^{(i)} \sim p(X_D = x_D | X_1 = x_1^{(i)}, X_2 = x_2^{(i)}, ..., X_D = x_{D-1}^{(i)})$
8: **end for**
9: **until** Gibbs sampling has reached equilibrium

If the assumption that we can use a limited number of samples from the state space to explain the entire state space probability distribution holds, we can use this technique to substantially reduce the computational aspect of the learning process.

2.3 Disadvantages of applying the Boltzmann learning rule

As pointed out by Haykin (1998), the combination of negative and positive phase "stabilizes the distribution of synaptic weights", but it also presents the following limitations:

- *Computationally heavy model* – The need of running the network until thermal equilibrium is reached in both the positive and negative phases makes anything bigger than a trivial Boltzmann Machine impractical. The time needed for the network to settle grows exponentially with the machine size, preventing the model from scaling up.

  Even with the Gibbs sampling trick the computation is still quite expensive, as each iteration of Algorithm 1 requires as many computations as the number of neurons in the network. This is due to the fact that the BM is a fully connected network and each neuron is influenced by all other neurons in the system.

- *Sensitivity to statistical errors* – Because the update rule in equation 10 exploits the difference between two average correlations (computed in the positive and negative phase), similarity in these two measures becomes a problem especially in the presence of sampling noise. This has the effect of causing the weights to follow a random path.

Based on these limitations, it has been shown (Fisher & Igel 2010) that without careful tailoring of the learning parameters to the specific training set, the machine can fail to model the probability distribution of the training data correctly.

2.4 Restricted Boltzmann Machines

The shortcomings of the Boltzmann Machine outlined in Section 2.3 can be address by modifying the network topology and imposing certain restrictions in order to overcome the time-consuming aspects of the Boltzmann learning rule.

This new topology, suggested by Smolensky (1986), is known as Restricted Boltzmann Machine (RBM).

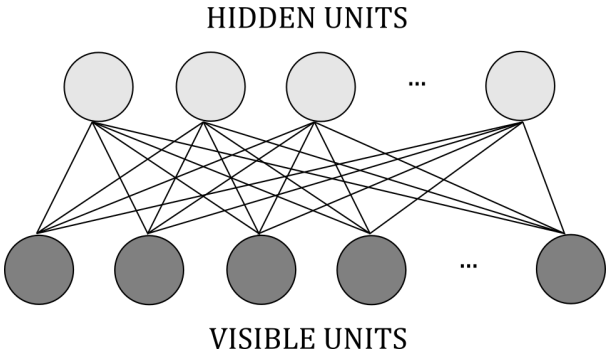


Figure 4: Restricted Boltzmann Machine

The RBM has two layers of processing elements – a visible layer and a hidden layer. The visible layer contains all visible neurons and corresponds to the components of a training record. The hidden units model dependencies between the visible units and can be viewed as explanatory factors.

There are connections between each node in opposite layers, but there are no lateral connections between elements within the same layer. In other words, the structure of the RBM forms a bipartite graph. The imposed restriction allows more efficient training, as demonstrated by Hinton (2002).

It has been shown that similarly to a Multilayer Perception, RBMs are *universal approximators* (Le Roux & Bengio 2008). However their universal approximation property is in the context of learning the parameters of an input distribution.

As a consequence of the lack of lateral connections, the energy function of a Restricted Boltzmann Machine is bilinear:

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i w_{i,j} h_j \tag{11}$$

where a_i is the bias of the visible units, and b_j is the bias for the hidden units.

The state probability is identical to the general Boltzmann machine (as defined by equations (5) and (6))and is given by

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)} \tag{12}$$

The restrictions applied to the RBM topology imply that the visible variables are independent given the state of the hidden units and vice versa. The conditional probabilities for the visible and hidden layers are then given by

$$\begin{aligned} p(v|h) &= \prod_i p(v_i|h) \\ p(h|v) &= \prod_j p(h_j|v) \end{aligned} \tag{13}$$

We can interpret the RBM as a stochastic neural network, where each individual node corresponds to a stochastic neuron with sigmoid activation $\sigma(x) = 1/(1+e^{-x})$.

In the simpler case of binary units the individual activation probabilities are then given by

$$\begin{aligned} p(v_i = 1|h) &= \sigma\left(a_i + \sum_j w_{i,j} h_j\right) \\ p(h_i = 1|v) &= \sigma\left(b_j + \sum_i w_{i,j} v_i\right) \end{aligned} \tag{14}$$

2.4.1 Restricted Boltzmann Machine Training

RBMs are trained to maximise the product of probabilities assigned to the training data set V:

$$\underset{\boldsymbol{w}}{\operatorname{argmax}} \prod_{\boldsymbol{v} \in V} p(\boldsymbol{v}) \tag{15}$$

The parameters $\boldsymbol{w}$ can be learned from the data using maximum likelihood approach similar to the technique described in section 2.2.

By differentiating equation 12, it can be shown that the derivative of the log probability of a training vector $\boldsymbol{v}$ with respect to the model parameters is given by

$$\frac{\partial \log p(\boldsymbol{v})}{\partial \boldsymbol{w}_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{16}$$

where $\langle v_i h_j \rangle_{data}$ is an expected value in the data distribution and $\langle v_i h_j \rangle_{model}$ is an expected value when the machine is sampling from its equilibrium distribution.

This leads to the following update rules for a Restricted Boltzmann Machine:

$$\begin{aligned} w_{ij} &\leftarrow w_{ij} + \eta[\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}], \\ b_i &\leftarrow b_i + \eta[\langle v_i \rangle_{data} - \langle v_i \rangle_{model}], \\ c_j &\leftarrow c_j + \eta[\langle h_j \rangle_{data} - \langle h_j \rangle_{model}] \end{aligned} \tag{17}$$

where η is a learning rate constant, and b_i and c_j are the bias units for the i-th visible and j-th hidden neurons.

Gibbs sampling is typically used to obtain unbiased samples from $\langle v_i h_j \rangle_{data}$ and $\langle v_i h_j \rangle_{model}$. As we previously pointed out, Gibbs sampling in a fully connected Boltzmann machine is slow. In the case of Restricted Boltzmann Machines, however, the restrictions imposed on the network make the hidden units conditionally independent from the visible units, so getting a sample from $\langle v_i h_j \rangle_{data}$ can be performed in one parallel step.

Sampling from $\langle v_i h_j \rangle_{model}$ is more involved as it requires alternating sampling between the visible and hidden layers. This technique is illustrated in Figure 5.

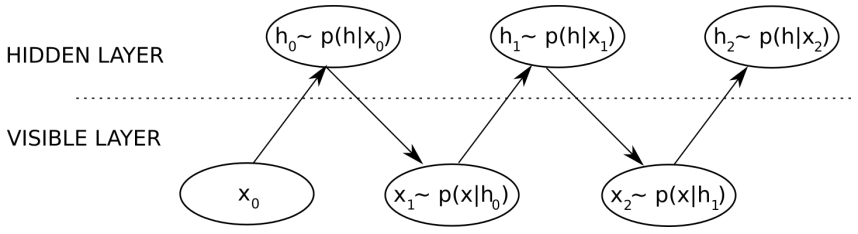


Figure 5: Alternating Gibbs sampling between visible and hidden layers.

The layer-wise Gibbs sampling can start with the visible layer being at any random state. One iteration of the sampling consist of updating all the hidden neurons in parallel, followed by a parallel update of all visible units. The updates are performed using the activation probability rules defined in equation 14.

Performing alternating Gibbs sampling for a very long will produce an unbiased sample of $\langle v_i h_j \rangle_{model}$, however this procedure is still computationally intensive – as the number of units in the network increases, a greater number of samples must be gathered to accurately explain the distribution generated by the network.

The essence of this maximum-likelihood learning is the minimisation of the Kullback-Leibler divergence $\mathrm{KL}(p_0||p_\infty)$. In order to guarantee an unbiased sample, the number of sampling steps needed in the update rule goes to infinity (see Hinton 2002).

$$w_{ij} \leftarrow w_{ij} + \eta[\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_\infty] \tag{18}$$

2.4.2 Contrastive Divergence Learning

In an attempt to resolve the infinite sampling steps problem, Hinton suggested a new learning algorithm for Restricted Boltzmann Machines called "Contrastive Divergence" (CD) (see Hinton 2002, Carreira-Perpinan & Hinton 2005).

Contrastive Divergence significantly reduces the number of Gibbs sampling steps by exploiting the fact that learning still works well if $\langle v_i h_j \rangle_{model}$ is replaced by $\langle v_i h_j \rangle_{reconstruction}$.

This "reconstruction" is obtained as follows:

1. Set a training example on the visible units and update all hidden units in parallel
2. Update all visible units in parallel to get a "reconstruction"
3. Update all hidden units using the reconstruction

Instead of minimising the Kullback-Leibler divergence, CD learning approximately follows the difference of two divergences:

$$\mathrm{CD}_n = \mathrm{KL}(p_0||p_\infty) - \mathrm{KL}(p_n||p_\infty) \tag{19}$$

Although this a "crude approximation" (Hinton 2012) of of the gradient of the log probability, it works well enough and provides estimates with very small bias (Carreira-Perpinan & Hinton 2005).

In CD learning, the sampling starts at the data distribution and the chain is run for a small number of steps (for $n = 1$ the algorithm is commonly referred to as CD1, for $n = 2$ – CD2

etc.). This greatly reduces the computation time and leads to the following modified update rule

$$w_{ij} \leftarrow w_{ij} + \eta[\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{reconstruction}] \tag{20}$$

where the $\langle v_i h_j \rangle_{reconstruction}$ is obtained by running just one Gibbs sampling step (in the case of CD1).

Algorithm 2 RBM training using CD1

v_1 is a sample from the training distribution
$\boldsymbol{a}$ is the visible units bias vector
$\boldsymbol{b}$ is the hidden units bias vector
η is a learning rate parameter
$\boldsymbol{w}$ is the weight matrix of the RBM
for all hidden units $i = 1, 2, ...$ **do** ▷ POSITIVE PHASE
    Propagate visible activation upwards : $p(h_{1,i} = 1|v_1) = \sigma\left(b_i + \sum_j w_{i,j} v_{1,j}\right)$
    Sample $h_{i1} \in \{0, 1\}$ from $p(h_{1,i}|v_1)$
end for
for all visible units $j = 1, 2, ...$ **do** ▷ NEGATIVE PHASE
    Propagate hidden activation downwards : $p(v_{2,j} = 1|h_1) = \sigma\left(a_j + \sum_i w_{i,j} h_{1,i}\right)$
    Sample $v_{2,j} \in \{0, 1\}$ from $p(v_{2,j}|h_1)$
end for
for all hidden units $i = 1, 2, ...$ **do**
    Compute $p(h_{2,i} = 1|v_2) = \sigma\left(b_i + \sum_j w_{i,j} v_{2,j}\right)$
end for
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(\boldsymbol{h_1}\boldsymbol{v_1}^T - p(\boldsymbol{h}_2 = 1|\boldsymbol{v}_2)\boldsymbol{v}_2{}^T)$ ▷ Update weights
$\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(\boldsymbol{v}_1 - \boldsymbol{v}_2)$ ▷ Bias update for visible units
$\boldsymbol{b} \leftarrow \boldsymbol{b} + \eta(\boldsymbol{h}_1 - p(\boldsymbol{h}_2 = 1|\boldsymbol{v}_2))$ ▷ Bias update for hidden units

A detailed description of CD1 in pseudo-code is listed in Algorithm 2. This implementation is a slightly modified version of the algorithm, originally presented by Bengio (2009).

Use of mini-batches

Although it is possible to apply Algorithm 2 to all observations in the training dataset in a sequential fashion, Hinton (2012) suggests that it is more effective to divide the input into groups of 10 to 100 observations (mini-batches), and update the weights and biases after estimating the gradient over the entire mini-batch.

Another important suggestion is that it is also helpful to divide the gradient computed on the mini-batch by the number of cases in the mini-batch, thus avoiding changes in the learning rate when the size of the mini-batch changes.

2.5 MapReduce and Apache Spark

In order to develop and evaluate the performance of Restricted Boltzmann Machines against relatively large data sets we need to utilise a cluster computing framework. Such approach will ideally allow us to offload technical tasks like data distribution and job orchestration to the underlying framework, thus allowing us to concentrate on the data, the algorithm, and the evaluation aspects.

MapReduce has emerged as the standard programming model for clustered big data processing. Inspired by the *map* and *reduce* functions commonly used in functional programming, MapReduce was initially proposed by Dean & Ghemawat (2004) as concept for distributed processing of large data sets. At the present time there are many frameworks that implement the MapReduce model (e.g. Hadoop, Couchdb, Riak, Infinispan and others).

Under the MapReduce model, programmers specify a map and a reduce function, and the framework automatically handles the parallelization and execution of the job on a large cluster of commodity machines. This model is at the core of *Apache Hadoop* (2016) – the open source platform for storing and processing of very large datasets.

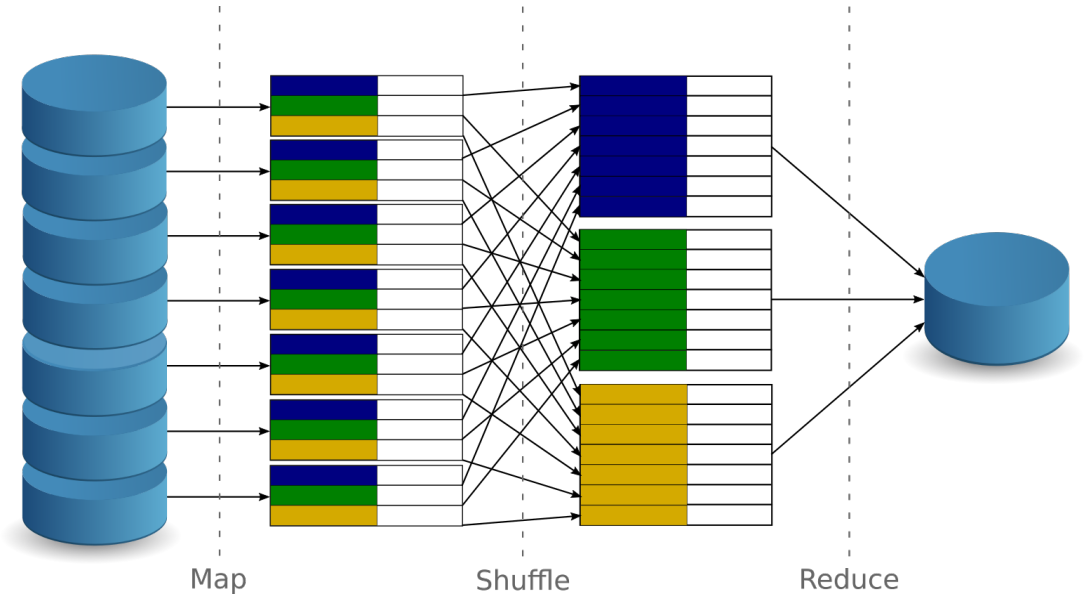


Figure 6: Key steps in a MapReduce processing job.

A MapReduce job comprises three processing steps, as illustrated in Figure 6:

- *map* step – this step processes a key/value pair and generates a set of intermediate key/value pairs. Each worker node applies the *map* function to its locally stored data

and writes the results to a temporary storage space

- *shuffle* step – worker nodes redistribute the intermediate key/value pairs over the network, such that all pairs with the same key end up on the same worker node
- *reduce* step – the *reduce* function is used to merge all key/value pairs having the same intermediate key. The values are typically merged using an aggregate function

A key advantage of the MapReduce framework is that it can automatically perform the job execution in parallel, given that the *map* functions are independent and that the *reduce* function is associative.

Combining the framework with a distributed file system that keeps multiple copies of the data on separate machines (e.g. HDFS) enables a MapReduce powered cluster to automatically handle node and task failures – if a node or a task it is running fails, the cluster can automatically restart this task on another surviving node that keeps a local copy of the same input data.

This combination of automatic parallelization and transparent reliability makes MapReduce well suited for handling large datasets and long-running processing jobs. It has been shown that MapReduce can be successfully used to sort a 10PB input set on a cluster comprising 8'000 machines (Dvorsky 2016).

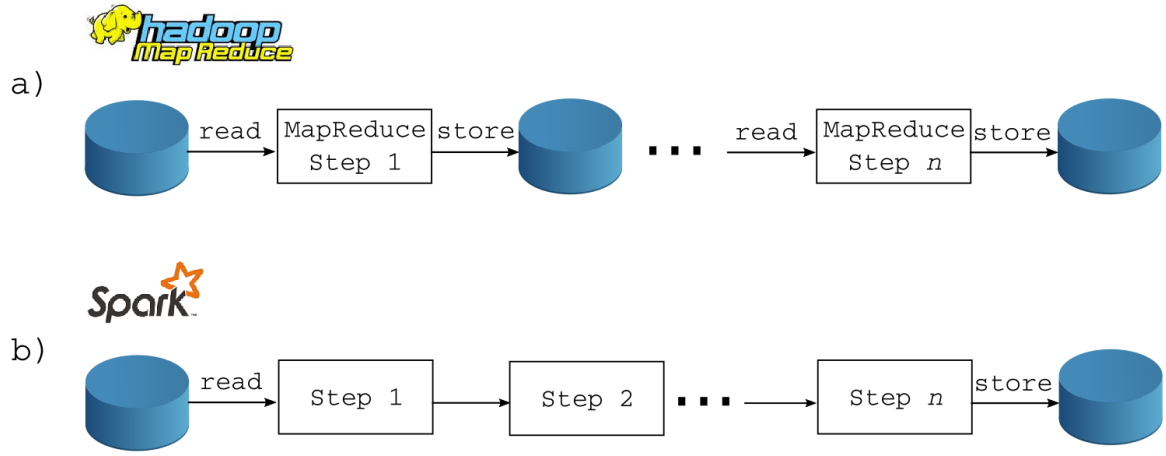


Figure 7: Workflow orchestration – MapReduce vs. Spark.

The parallelization and scalability advantages provided by MapReduce make it well suited for the purposes of the project and enables us to research the feasibility of RBMs in an x86-based clustered environment, as Apache Hadoop is specifically designed to take advantage of inexpensive, x86-based commodity hardware (see *Best practices for selecting Apache Hadoop hardware* 2016).

There is, however, a certain performance disadvantage that arises from the way MapReduce handles complex workflows. As every processing in MapReduce must be expressed as

a step executing the *map* and *reduce* functions, more complex workflows are expressed by chaining a series of map-reduce steps. Having an empty *map* or *reduce* function in a series of map-reduce steps also allows the developers to construct sequences like *map*→*map*→*reduce* or *map*→*reduce*→*reduce*.

Each map-reduce step, however, reads its input from the local worker node disks and stores its output back to disk when finished. This means that in a more complex workflow or in a workflow, which implements iterative processing, the entire data set must be read and stored back to disk at each individual map-reduce step (as illustrated in Figure 7-a). According to Zaharia et al. (2012) "This incurs substantial overheads due to data replication, disk I/O, and serialization...".

Knowing that disk input-output operations are orders of magnitude slower than their in-memory equivalent (reading 1MB sequentially from memory takes 250'000 ns vs. 20'000'000 ns when reading from disk[4]), it is easy to see how the MapReduce architecture can lead to inefficiency in complex, iterative tasks.

To overcome the inefficiency issues of MapReduce, Zaharia et al. (2012) present a distributed memory abstraction called Resilient Distributed Datasets (RDDs). RDDs are fault-tolerant by design as they log the transformations applied at each processing step, thus allowing automatic reconstruction should a task or a node fails. They represent parallel data structures that can persist intermediate results in-memory, making the abstraction very efficient for iterative distributed processing. They also feature a rich set of operators like *map*, *filter*, and *join* and are expressive enough to represent any processing that MapReduce is capable of. Programmers are also given manual control on the persistence behaviour of individual RDDs, so they can choose what to keep in memory and what to spill to disks. RDDs also allow manual repartitioning for placement optimisation.

Zaharia et al. have implemented the RDD abstraction in a system called Spark – an open source computing framework, which they later donated to and is being managed by the Apache Software Foundation (*Apache Spark: Lightning-fast cluster computing* 2016).

Using the RDD abstraction, Spark is capable of running iterative algorithms by looping over the dataset in-memory, instead of accessing the local drives at each iteration (Figure 7-b).

Moreover, a Spark program is defined as a directed acyclic graph - it does not have the restriction to represent each job as a map-reduce sequence, thus allowing the construction of more complex workflows. The workflows can also be fully inspected and optimised before execution commences - another advantage that gives Spark performance leverage compared to MapReduce.

Indeed, Zaharia et al. show that when testing iterative processing, Spark is moderately faster at the first iteration compared to MapReduce (as both have to initially read the data

[4]See Norvig (2014)

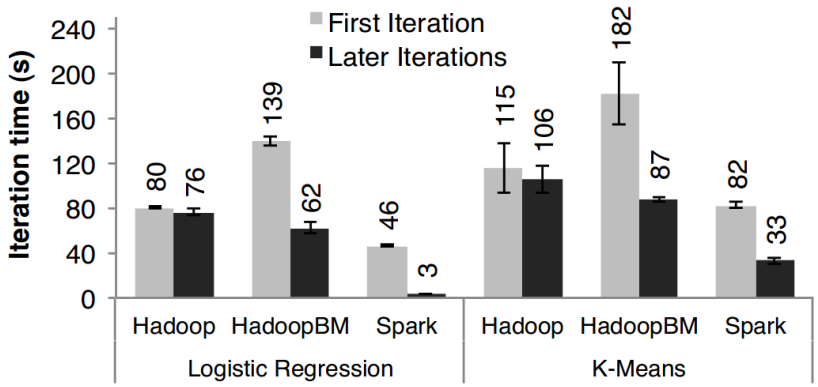


Figure 8: Duration of the first and later iterations in Hadoop, HadoopBinMem, and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster, by Zaharia et al. (2012)

from disk), in subsequent iterations Spark is much faster than MapReduce – 25.3 times faster when running Logistic Regression training against a 100 GB dataset (Figure 8).

As the training algorithm for Restricted Boltzmann Machines is iterative in nature (see Algorithm 2) we will look into what impact does the platform selection (MapReduce vs. Spark) have on execution runtime.

2.6 Optimal plan generation and SystemML

MapReduce and Spark have emerged as the de facto standard for parallel processing of big datasets. When dealing with complex machine algorithms, however, the performance is heavily influenced by the algorithm's execution plan. This issue has been identified by Ghoting et al. (2011), who also point out that the actual execution plan has to be hand-tuned for different input sizes and cluster configurations.

We can illustrate the issue with a rudimentary example based on linear regression. Say we want to model a linear relationship between an set of input features represented as a matrix X, and a set of target variables in vector $\boldsymbol{y}$.

We express the model hypothesis as

$$h(\boldsymbol{x}) = \sum_{i=0}^{n} \theta_i x_i = \theta^T \boldsymbol{x} \tag{21}$$

and establish the following cost function $J(\theta)$

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h(\boldsymbol{x}_i) - y_i)^2 \tag{22}$$

Dataset characteristics	
Dimensionality of X	300 million x 500
Dimensionality of y	300 million
Size of X	4 TB
Size of y	9 GB
Cluster configuration	
Map Task JVM Memory	3.5 GB
Master JVM Memory	7 GB
HDFS block size	128 MB

Table 1: Execution plan example - dataset characteristics and cluster configuration

The normal equations approach produces the following solution for the model coefficients that minimise $J(\theta)$:

$$\theta = (X^T X)^{-1} X^T \boldsymbol{y} \tag{23}$$

Let assume that the dataset used for our example has the characteristics outlined in Table 1. The input matrix and the vector of target values are stored as plain text files in HDFS, and the number of features in the input matrix is 500.

Table 1 also reveals that the amount of memory available to each worker in our Hadoop cluster is capped to 3.5 GB and the master node can utilise up to 7 GB of memory.

One way of solving Equation 23 is to establish a system of equations and call a simple *solve()*[5] function that solves the system using QR decomposition. This can formally be written as Algorithm 3.

Algorithm 3 Linear regression unsing a direct solver

1: $a = X^T X$

2: $b = X^T \boldsymbol{y}$

3: $\theta = solve(a, b)$ ▷ QR-based decomposition solver

It appears that for the given dataset and cluster configuration (as showin in Table 1) the most optimal implementation of Algorithm 3 is the one shown in Figure 9.

This plan breaks the input matrix X into blocks of 1'000 by 1'000 elements. It then takes the transpose of $\boldsymbol{y}$ and puts it in the distributed cache. The execution consists of a single map step, which encapsulates two operations – $a = X^T X$ and $b^T = \boldsymbol{y}^T X$.

[5]Most data processing languages provide such function out of the box (e.g. *solve()* in R, *numpy.linalg.solve()* in Python etc.))

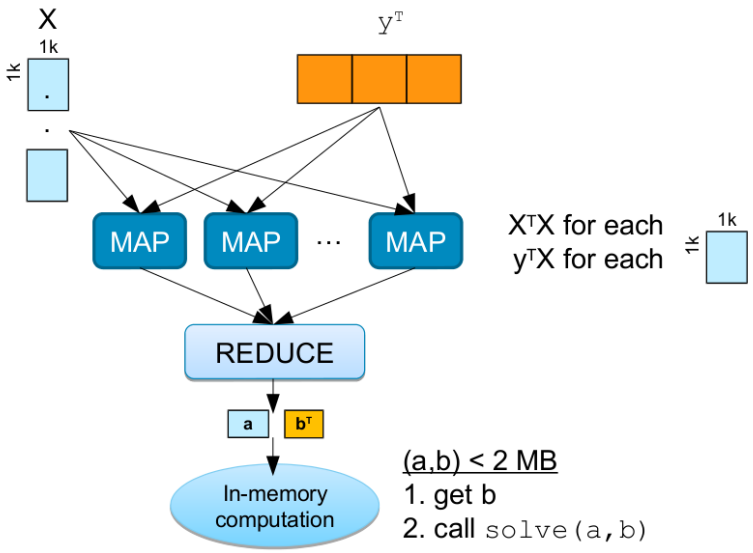


Figure 9: Optimal execution plan for Algorithm 3 in MapReduce

After the reduce phase we have the values for a and b^T, however their combined size is expected to be less than 2 MB, so the QR decomposition can be completed on the driver alone, without further map-reduce operations.

The problem identified by Ghoting et al. arises from changes in the dataset or modifications to the cluster, which are introduced after the optimal execution plan has been established.

Table 2 reveals how minor changes in the input dataset or the cluster configuration lead to four dramatically different execution plans, turning the manually tuned plan into a non-working or non-optimal solution.

To mitigate impact of data and configuration changes on the execution plan, Ghoting et al. (2011) present a scalable, declarative machine learning platform called SystemML. The system was initially designed by the authors as part of their work at IBM Research – Almaden, but it was open sourced in November 2015 and is currently managed by the Apache Software Foundation project – *Apache SystemML* (2016).

In SystemML algorithms are expressed in a high-level language – Declarative Machine learning Language (DML). DML is compiled on-demand, and the execution is coordinated by SystemML. The SystemML platform can use either MapReduce or Spark as a back-end for running the algorithm against datasets in Hadoop (HDFS).

The architecture of SystemML is given in Figure 10 and shows the four key components of SystemML:

- DML – DML is a fully fledged programming language, the syntax of which closely resembles the R Programming Language. It supports two main data types – scalars and matrices, and exposes various input/output, control, and assignment operators. The language features are discussed in details in *DML Reference* (2016)

Changes	Effect	Impact
More attributes added to X. New matrix dimensions are 300 million x 1'500.	There is not enough memory to increase the size of the 1k x 1k blocks.	Plan must be modified and the $X^T X$ computation performed in two chained map-reduce steps.
More observations added to X. New matrix dimensions are 600 million x 500.	The increase of number of observations in $\boldsymbol{y}$ prevents it from fitting in the distributed cache.	Plan must be modified and the $X^T \boldsymbol{y}$ computation performed in two chained map-reduce steps.
The number of observations and features of X is reduced. New matrix dimensions are 1 million x 100.	Plan is not optimal as triggering a map-reduce job is no longer needed. All computations can be performed in the driver alone.	The plan must be modified to perform all computation on the driver system only.
Cluster config. changed – new JVM memory for Map Tasks is 1.5 GB.	The reduced worker memory prevents $\boldsymbol{y}$ from fitting in the distributed cache.	Plan must be modified and the $X^T \boldsymbol{y}$ computation performed in two chained map-reduce steps.

Table 2: Impact of data and cluster changes on the MapReduce execution plan.

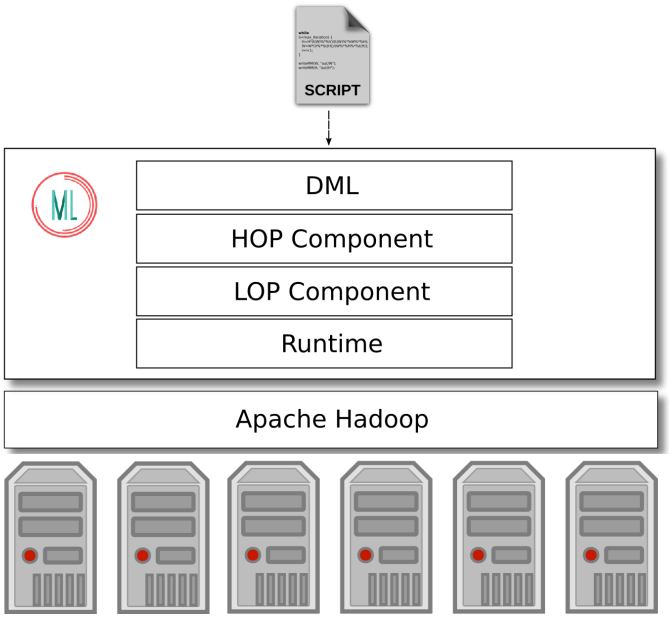


Figure 10: SystemML architecture

- High-Level Operator (HOP) Component – the HOP component generates a set of possible execution plans, representing the operations on matrices and scalars as a Directed Acyclic Graph (DAG). It then uses a cost based optimizer to select the most effective execution plan from the generated set of possible plans

- Low-Level Operator (LOP) Component – the LOP component generates low-level physical execution plans (LOPDags) over key-value pairs. It is responsible for identifying operations that can be packaged together in a single map step (in other words operations "piggybacking"). The LOP component is also responsible for simplification rewrites – it removes unnecessary operations (e.g. $(X^T)^T \rightarrow X$, $X - 0 \rightarrow X$, $X \times X \rightarrow X^2$, $tr(X \times Y) \rightarrow \sum X \times Y^T$ etc.)

- Runtime – the runtime component can execute jobs written in DML. It supports three[6] execution modes:

  - Standalone – All operations are executed internally in the SystemML Control Program (CP) and no distributed infrastructure is used at all. The entire execution happens on a single machine
  - MapReduce Hybrid – Operations that fit the memory of the node when the Control Program runs are executed internally, all other operations are distributed using a MapReduce back-end
  - Spark Hybrid – Similarly to MapReduce Hybrid, operations that fit the memory available to the Control Program are run locally, and all other operations are offload to a Spark back-end

The capability of SystemML to offload the distributed matrix operations to Spark and MapReduce, without requiring changes in the DML script is key to our research.

Not only does SystemML allow us to run the RBM algorithm without concerns about the optimal execution when we change the number of nodes in the cluster, but it also allows us to run the same algorithm on a single node, on a MapReduce cluster, or on a Spark cluster, without any changes to the algorithm source code.

This functionality allows us to vary the number of physical nodes and change the back-end distributed framework, without having to worry about introducing code changes or having to worry about any code change induced impact on the outcome of the experiments.

[6]Technically speaking, there are five execution modes – Standalone, MapReduce, MapReduce Hybrid, Spark, and Spark Hybrid. The non-hybrid modes offload all operations to the back-end and are less efficient compared to the hybrid modes, so we see no reason to look or run tests with them at all. SystemML also supports a High Performance Computing back-end via Open MPI. This, however, is completely out of scope for this research.

2.7 Conclusion

In this section we established the theoretical foundations for the parallelized RBM, and we presented a formal definition of the CD1 RBM learning algorithm that we plan to implement (see Algorithm 2). We also aim to use mini-batches, as there is clear evidence that this approach increases the effectiveness of the training process (Hinton 2012).

We discussed two models for parallel high-volume data processing - MapReduce and RDD (Spark), which we intend to test our custom RBM with. We also presented the problem of selecting an optimal execution plan when the data and cluster configuration change, and we outlined how we intend to resolve this issue using SystemML. A key advantage provided by SystemML is that it will enable us to modify the parameters of the experiments (e.g. cluster configuration) while keeping the RBM implementation intact – SystemML will automatically recompile the DML code to capture the changes in the infrastructure, guaranteeing an optimal execution for the RBM training.

3 Methods

In this section we present our custom DML RBM implementation based on Algorithm 2 with some enhancements (e.g. automatic normalization, use of mini-batches), and we report on how we tested the implementation to ensure it is functionally correct. We look into selecting optimal parameters and test the impact on accuracy when the RBM participates in a machine learning pipeline. We also repeat the functional tests with a Theano-based (GPU) RBM implementation.

In Section 3.4 we introduce the dataset selected for the scalability tests. We perform exploratory data analysis, identify certain quality issues, and we report on the preparation and data wrangling scripts we had to develop in order to transform the data into an RBM friendly format.

We also report on the architecture of our Hadoop cluster, which will run the MapReduce and Spark scalability tests. We discuss the hardware, the operating system, and specific Hadoop configuration parameter. We discuss the separate GPU-enabled environment and the way we plan to conduct the experiments.

3.1 Parallelised, CPU-based Restricted Boltzmann Machine implementation

As SystemML did not feature an out of the box implementation of a Restricted Boltzmann Machine, we had to create one using SystemML's custom programming language – DML.

The implementation is based on the CD1 algorithm outlined in Algortihm 2, and it is listed in Appendix B on page 104 (see Listings 1, 2, and 3).

As outlined in Section 2.4.2, the use of mini-batches is expected to speed up the convergence. The algorithm shown in Listing 1 uses mini-batches as suggested. We also developed an alternate version (Listing 2), which does not use mini-batches and instead performs computations with the entire data set at each iteration. Our expectations are that the non-minibatch version will perform worse, but we wanted to investigate the impact of the mini-batch approach in the context of Hadoop, so having this simplified version could help us by serving as a base-line.

The RBM implementation in Listings 1 and 3 was suggested to the SystemML development team, underwent code review, and was accepted into the project via pull request #188 as evident in the SystemML pull request history on GitHub: `https://github.com/apache/incubator-systemml/pull/188`.

The implementation consists of two scripts – *rbm_minibatch.dml*, which trains an RBM, and *rbm_run.dml*, which runs a dataset through a trained RBM.

rbm_minibatch.dml takes a matrix of features $\boldsymbol{X}$, trains an RBM, and outputs its weights($\boldsymbol{w}$) and biases($\boldsymbol{a}$, $\boldsymbol{b}$). A detailed list of the parameters of the script is shown in Table 3. It is evident from the list of parameters that the script improves on Algortihm 2 by processing $\boldsymbol{X}$ in mini-batches.

Parameter	Type	Default	Description
X	String	-	Location to read the matrix X of feature vectors. This location can be a resource on a local or a distributed file system (HDFS).
W	String	-	Location to store the learned RBM weights ($\boldsymbol{w}$).
A	String	-	Location to store the bias vector for the visible units ($\boldsymbol{a}$).
B	String	-	Location to store the bias vector for the hidden units ($\boldsymbol{b}$).
batchsize	Int	100	Number of observations in one batch.
vis	Int	Infer	Number of units in the visible layer. If not set, this value is inferred from the number of features in $\boldsymbol{X}$.
hid	Int	2	Number of units in the hidden layer
epochs	Int	10	Number of training epochs
alpha	Double	0.1	Learning rate ($\boldsymbol{\eta}$)
fmt	String	text	Matrix output format for W,A, and B: "text" (a text-based sparse format with the matrices serialized in triplets of rowID, columnID, and value), "csv", "mm" (Matrix Market - similar to text, but also encapsulates metadata), and "binary" (SystemML proprietary).

Table 3: Input parameters for the *rbm_minibatch.dml* script

The standard batch algorithm runs the positive/negative phase over the entire matrix of features, but it has been shown that we can use a subset $X' \in X$ instead, decreasing computational effort between parameter updates at each iteration (Fisher & Igel 2014). The script also allows the user to fine-tune the size of the mini-batches (X') via the *batchsize* parameter.

The *rbm_run.dml* script uses the weights and biases produced by *rbm_minibatch.dml* and processes a matrix of observations using the trained RBM. The processing is carried out in two steps.

In step one the script computes the transitioning probability $p(h = 1|v)$, by using the energy gap (Equation 2) in Equation 4.

$$p(h = 1|v) = \frac{1}{1 + e^{-(\boldsymbol{X} * \boldsymbol{w} + \boldsymbol{b})}} \tag{24}$$

In step two we sample $p(h = 1|v)$, and we store the sample as the output of passing an

Parameter	Type	Default	Description
X	String	-	Location of the matrix of observations X.
W	String	-	Location of the RBM's weight vector ($\boldsymbol{w}$).
A	String	-	Location of the RBM's visible units bias vector ($\boldsymbol{a}$).
B	String	-	Location of the RBM's hidden units bias vector ($\boldsymbol{b}$).
O	String	-	Location to store the processed observations.
fmt	String	csv	Matrix output format for O (see Table 3).

Table 4: Input parameters for the *rbm_run.dml* script

observation through the machine.

Both scripts check the range of the input data and use the following transformation to rescale it to $[0, 1]$ if need (see lines 87–93 in *rbm_minibatch.dml* and 71–77 in *rbm_run.dml*).

$$\boldsymbol{X}_{rescaled} = \frac{\boldsymbol{X} - \min(\boldsymbol{X})}{\max(\boldsymbol{X}) - \min(\boldsymbol{X})} \tag{25}$$

The parallel execution capabilities of the model arise from the fact that the scripts operate on top of SystemML, and that the data is stored in a distributed fashion in HDFS. SystemML handles the DML compilation, high-level operator optimisation, and low-level operator optimisation (see section 2.6), while the data locality computations and lazy evaluation is handled by Apache Spark, as discussed in section 2.5.

3.2 Functional testing of the DML RBM scripts

To make sure that the DML RBM implementation is functionally sound we carried out a series of functional tests. The scripts were tested with a relatively small dataset (MNIST), using the Standalone mode of SystemML.

Dataset

The dataset of choice for the functional testing is the Mixed National Institute of Standards and Technology database (MNIST). The dataset contains 60'000 training and 10'000 testing images of handwritten digits (see LeCun et al. 2016).

The data in MNIST comes from the original NIST Special Database 19 (Grother 1995), with the original images normalised to fit a 20x20 pixels frame and anti-aliased to introduce gray scale levels. The images were then centred in an 28x28 pixel square. Figure 11 depicts a sample from the MNIST dataset.

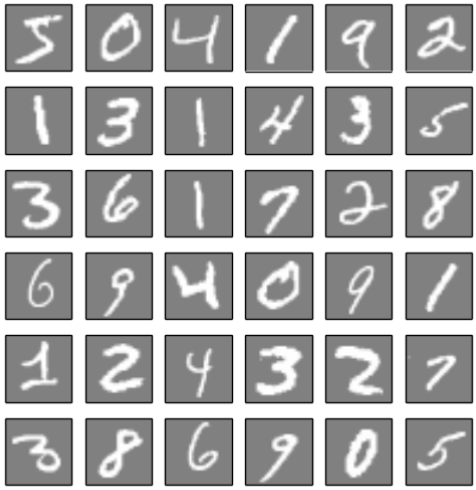


Figure 11: A sample of 36 images from the MNIST dataset.

MNIST is originally distributed in four compressed files:

- *train-images-idx3-ubyte.gz* – 60'000 training set images
- *train-labels-idx1-ubyte.gz* – labels for the training set
- *t10k-images-idx3-ubyte.gz* – 10'000 test set images
- *t10k-labels-idx1-ubyte.gz* – labels for the test set

The data is in a proprietary binary format and is encoded in the MSB first format (high endian). The training and test files have the following format as given in LeCun et al. (2016):

```
TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label

The labels values are 0 to 9.
```

```
TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel

Pixels are organized row-wise. Pixel values are 0 to 255.
0 means background (white), 255 means foreground (black).
```

To make the dataset easier to manipulate and more suitable for processing in DML we developed a custom Python script to re-code the data, and transform it into a matrix of input features ($\boldsymbol{X}$) and a vector of target labels ($\boldsymbol{y}$).

$$\boldsymbol{X}_{m,n} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{26}$$

For the training set the dimensions of matrix $\boldsymbol{X}$ are 60'000 by 784 ($28 \times 28 = 784$ pixel values) and the length of the $\boldsymbol{y}$ vector is also 60'000 (one label per image). The transformed data was stored in four comma-separated values (CSV) files as follows:

- *mnist_train_X.csv* – the $\boldsymbol{X}$ matrix for 60'000 training images. Total file size is 127.4MB.
- *mnist_train_Y.csv* – the $\boldsymbol{y}$ vector for the training images. File size is 228.2KB.
- *mnist_test_X.csv* –the $\boldsymbol{X}$ matrix for 10'000 test images. Total file size is 21.3MB.
- *mnist_test_Y.csv* – the $\boldsymbol{y}$ vector for the training images. File size is 38.0KB.

We also created the corresponding matrix dimension (MTD) files. These are JavaScript Object Notation (JSON) files, which SystemML needs for the generation of the optimal execution plan. They contain the matrix dimension and optional additional data on the matrix (e.g number of non-zero elements, data format etc.). For example, the MTD file for *mnist_train_X.csv* has the following contents:

```
{
    "data_type": "matrix",
    "value_type": "double",
    "rows": 60000,
    "cols": 784,
    "nnz": 8994156,
    "format": "csv",
    "header": false,
    "sep": ",",
    "description": {
        "author": "SystemML"
}
```


We will refer to the aforementioned combination of CSV and MTD files as "the pre-processed MNIST dataset" for brevity. This pre-processed data set was used to carry out the functional testing of the RBM scripts, as outlined in the next section.

Functional testing approach

We started the functional testing phase by training an RBM with 200 features for 50 epochs, varying the network learning rate in the $[0.01, 0.90]$ range. The number of neurons in the hidden layer (200) was selected based on a grid search over the parameters for a Bernoulli RBM against the MNIST data set as shown in Rosebrock (2014), however, as the aim of this work is not to improve prediction accuracy, we did not rigorously looked into the optimal number of features and took this number at face value.

The resulting changes of the normalised sum of squared errors (SSE) on the training data, which is computed as $\frac{1}{N}\sum(\boldsymbol{y} - p(\boldsymbol{v_2} = 1|\boldsymbol{h_1}))^2$ is shown in Figure 12.

Figure 12 reveals that the error drops down with each consecutive epoch of training, which is what we would expect from a functioning neural network.

Changes in the learning rate η impact the starting position of the error curve, as the initial SSE measurement is taken at the end of the first epoch. Higher learning rate allows for steeper drop of the metric at the end of epoch 1 – that is an expected behaviour.

Looking at the changes in SSE, it appears that there are no benefits in increasing η beyond 0.1. The $\eta = 0.3$ run actually has higher error at the end of epoch 1, and it also has higher training error at the end of the training cycle (epoch 50). At the same time $\eta = 0.6$ and $\eta = 0.9$ show clear indications of overshooting.

After fixing the learning rate to 0.1 we let the network train for 200 epochs and tried to identify the optimal stopping time for the learning process.

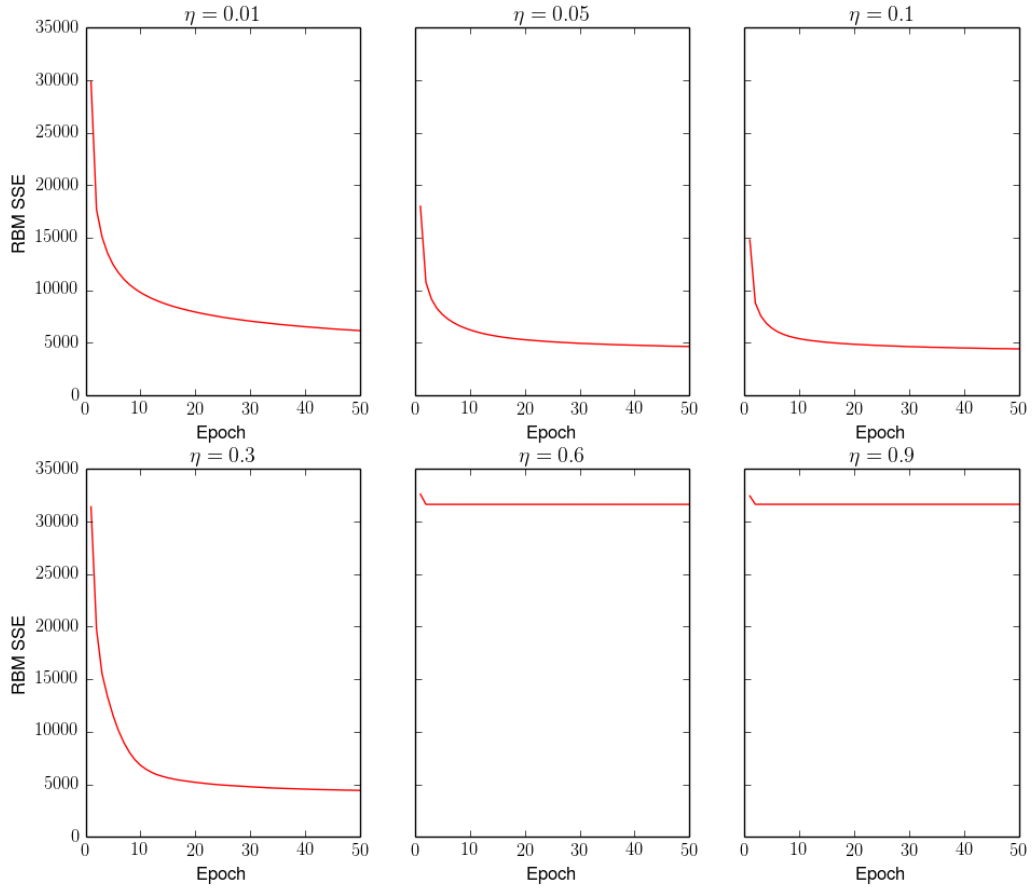


Figure 12: Changes in the SSE for an RBM with 200 features trained for 50 epochs against the pre-processed MNIST dataset.

The data in Figure 13 suggests that it does not pay off to keep the training beyond 200 epochs as the reduction of the training error almost flatlines. Inspecting the data behind the plot also reveals that after epoch 100 the error rate starts to oscillate slightly, with the rate of oscillation increasing as it approaches 200.

Training the network for 200 epochs takes around 48 minutes and 27 seconds (averaged across 10 runs) on a 4-core Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz machine with 12GB of RAM, however it should be noted that this operation was performed using the SystemML standalone runtime and no advanced parallelisation was taking place.

We also looked into the total runtime for the training process, while gradually increasing the number of training epochs. The results from this test are shown in Figure 14, which suggests that the RBM scales linearly (at least in a single machine setting).

It is also worth noting that the runtime is not impacted by η as there is no error-based convergence criteria – the network always iterates over the data until the maximum number of

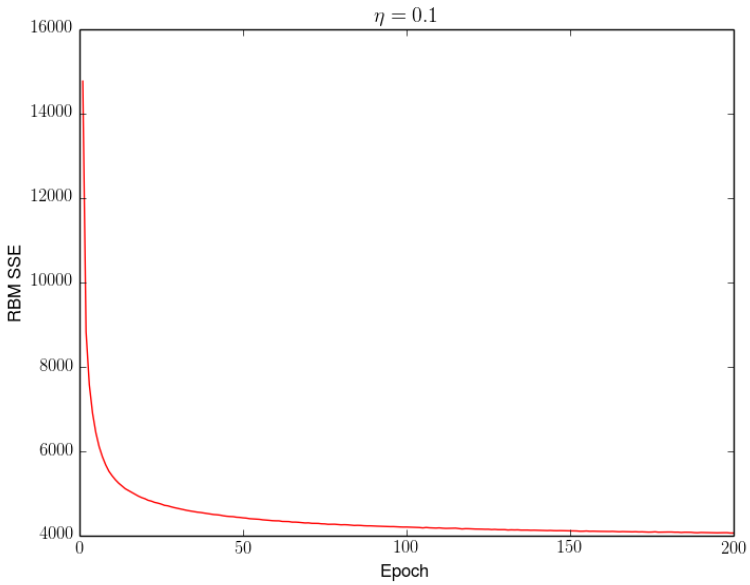


Figure 13: Changes in the SSE for an RBM with 200 features trained for 200 epochs against the pre-processed MNIST dataset with a learning rate $\eta = 0.1$.

epochs constraint is reached.

We then looked at the features learnt by the trained RBM. The dimensions of the matrix of learnt weights is $m \times n$, where m is the number of inputs and n is the number of neurons in the hidden layer. For the pre-processed MNIST dataset this is 784 (28×28 pixels) $\times$ 200. Figure 15 shows a subset of 36 components, and the full set of learnt features is available in Appendix C (page 115).

To further evaluate the performance and feasibility of the RBM-based feature extraction we built a set of machine learning pipelines, feeding the learnt features to two different classifiers and measuring the accuracy based on predictions on the test data set. The constructed machine learning pipeline is shown in Figure 16.

We start by training a Restricted Boltzmann Machine on the pre-processed MNIST training subset. The machine runs for 100 epochs with learning rate $\eta = 0.1$ and 200 neurons in the hidden layer. The resulting synaptic weights $\boldsymbol{w}$ are then used with the *rbm_run.dml* script to process both the training and test subsets. The resulting subsets at the end of this step are a training subset – a $60'000 \times 200$ matrix and a test subset – a $10'000 \times 200$ matrix.

These matrices are then fed to two separate classifiers - a Support Vector Machine (SVM) classifier and a Naïve Bayes classifier, both available as existing algorithms in SystemML. Using the two classifier we measure the prediction accuracy by training them on the 200 features train subset and testing on the 200 features test subset, effectively employing a hold-out method.

We perform the same train/test routine with the full pre-processed MNIST set (no feature extraction) and we compare the changes in prediction accuracy. The results from these runs

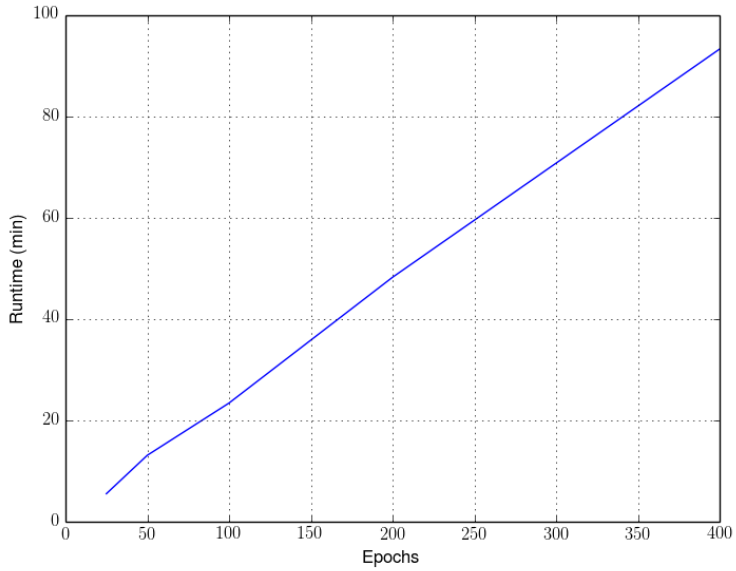


Figure 14: Runtime (in minutes) while varying the number of training epochs against the pre-processed MNIST dataset.

	Pre-processed MNIST		**200 RBM extracted MNIST features**	
Classifier	Naïve Bayes	SVM	Naïve Bayes	SVM
Accuracy (%)	83.65	91.55	84.35	91.29
Training Runtime (s)	5.73	27.49	2.61	10.31

Table 5: Accuracy changes for training on the complete pre-processed MNIST dataset (768 x 60'000) and a 200 features extract (200 x 60'000) generated by an RBM. The accuracy score is calculated using hold-out validation with a separate set of 10'000 observations.

are shown in Table 5.

Functional testing results

Figure 15 confirms that the machine is able to learn – it successfully stores the acquired knowledge in its synaptic weights as expected. In addition, Table 5 reveals that running Naïve Bayes and SVM classifiers against 200 features does not harm the accuracy when compared to running the algorithms against the complete pre-processed MNIST dataset.

It is worth noting, that we did not explicitly aim to improve the accuracy based on the feature extraction. It has been show that this is indeed achievable, and RBMs can outperform other approaches when their parameters (e.g. number of neurons in the hidden layer) are purposely fine-tuned (Abdollahi & Nasraoui 2016).

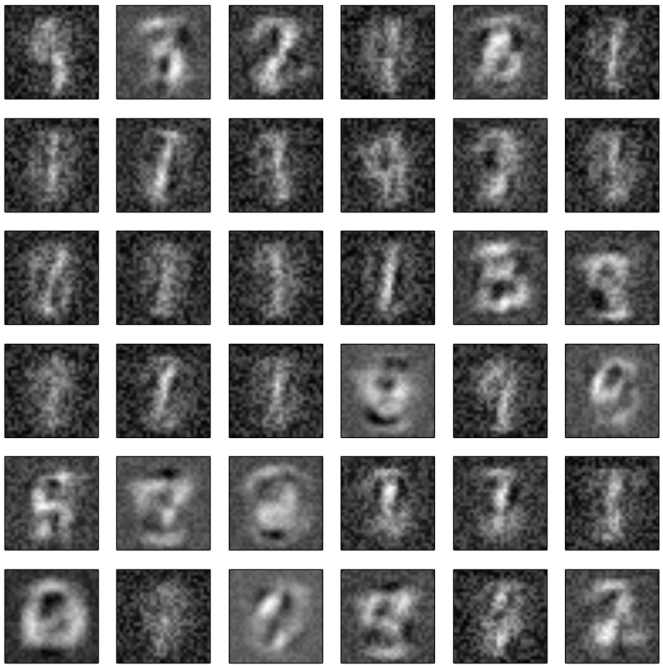


Figure 15: The first 36 features (out of 200) learnt by the RBM while training against the pre-processed MNIST dataset.

On a side note, Table 5 also makes it evident that replacing the full dataset with a set of extract features substantially speeds up the convergence times for both the Naïve Bayes and the SVM classifiers. In fact, the convergence times for the extracted features subset are more than two times shorter than the convergence times on the complete dataset.

Based on the results of the functional testing we can conclude that we have a working Restricted Boltzmann Machine implementation that should be cable of running in a parallelized, multi-CPU context (as it is built on top of the SystemML platform). We can now look at its GPU-based counterpart and the dataset used for comparing and contrasting the two implementations.

3.3 GPU-based RBM

The latest version of SystemML (0.11.0) comes with experimental GPU support (*SystemML: Initial prototype for GPU backend* 2016), which would have made it an ideal platform for comparing the pure CPU-based versus GPU assisted RBM training. The GPU back-end uses the JCuda wrapper to access Nvidia's Compute Unified Device Architecture (CUDA), and Amazon[7]

[7]This is relevant, as Amazon is the selected cloud provider for running all experiments (pure CPU-based and GPU assisted). See Section 3.5 for additional details on the selected infrastructure.

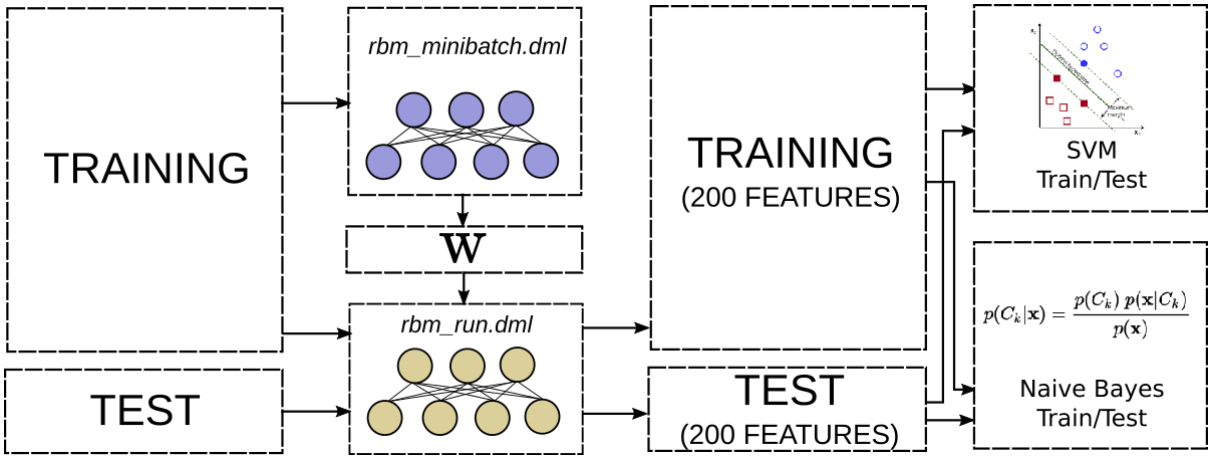


Figure 16: Machine Learning pipeline for the functional testing of the RBM against the pre-processed MNIST dataset

provide instances with High-performance NVIDIA GPUs and pre-installed CUDA 0.7.5.

Unfortunately, the JCuda wrapper currently supports only CUDA version 0.8.0 as stated in its documentation – "It looks like right now only 0.8.0 or higher versions are usable due to adding of new library in 0.8.0 JCuda" (*Mavenized JCuda* 2016).

Upgrading CUDA to 0.8.0 is not an option, as 0.8.0 has dependencies[8] that are not available in the Amazon's Linux AMI distribution – the operating system that comes with the GPU-enabled Amazon instances.

As a workaround, we decided to use Theano – a numerical computation library for Python, that can work with CUDA 0.7.5 and supports compilation of computation for GPU-based execution (see *Theano 0.8.2 documentation* 2016).

Deep Learning (2016) provides an implementation of a Restricted Boltzmann Machine for Theano, which is very similar to our DML-based implementation. Similarly to our RBM implementation, the Theano RBM implementation uses a sigmoid activation function, Contrastive Divergence, and mini-batches in its main loop. We can control the number of training epochs, learning rate (η), number of neurons in the hidden layer, and the size of mini-batches. Details on the implementation and the corresponding source code is given in *DeepLearning 0.1 Documentation – Restricted Boltzmann Machines* (2016).

By default the Theano-based RBM uses Persistent Contrastive Divergence (PCD) – it relies on a single Markov chain that has a persistent state as suggested by Tieleman (2008). We, however, felt that this may introduce an unfair advantage in the algorithm alone, so we modified the code by setting the persistent chain to "None", thus forcing the algorithm to restart the chain as it learns, instead of preserving its state.

As with the DML script, we tested the Theano-based RBM training script (*rbm.py*) against

[8]For example the CUDA 0.8.0 depends on *systemd*, which is not supported in Amazon Linux AMI

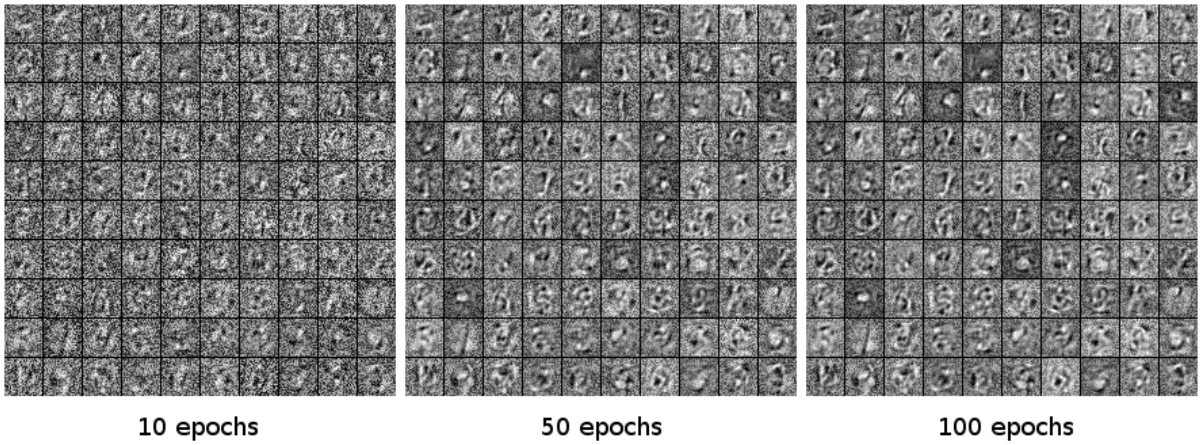


Figure 17: GPU-based mini-batch RBM training on the MNIST dataset. 100 epochs, 200 neurons in the hidden layer, learning rate $\eta = 0.1$.

Figure 18: GPU-based mini-batch RBM training on the raw MNIST dataset (not rescaled to $[0; 1]$. Even after 100 epochs the RBM has not learnt any useful features.

the MNIST dataset. Features learnt by the RBM under training conditions similar to what we used with DML script are shown in Figure 17.

Interestingly, it appears that the features learned by the Theano GPU-based training script contain more noise compared to our RBM implementation, although they get better with continuous training. They are also less distinct and more difficult to interpret compared to the features learnt by the DML script (see Figure 15).

Another interesting observation is that the GPU-based script does not automatically rescale the inputs, and if we do not rescale the MNIST values to the $[0; 1]$ interval, the RBM is unable to learn. Instead, the data stored in its weight is of no use and does not change no matter how long we let the RBM learn (see Figure 18).

This introduces the necessity of an additional pre-processing step that will rescale all training

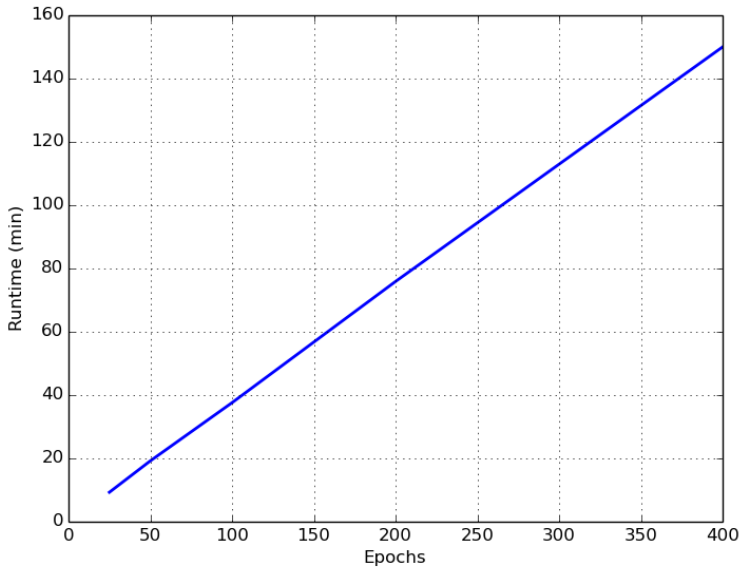


Figure 19: Runtime (in minutes) for the GPU-based training, while varying the number of training epochs against the MNIST dataset.

examples as per Equation 25, however, modifying the Theano RBM script to check the input range and normalize the data automatically is a very simple fix that can easily be added to its code.

Looking at the time needed to complete the training while varying the number of training epochs does not present any surprises – the runtime increases linearly with the increase of the number of epochs (Figure 19). This behaviour is identical to the DML RBM (see Figure 14).

A key difference, though, is that the GPU-based RBM optimizes a pseudo-likelihood as a proxy of the likelihood function (see *DeepLearning 0.1 Documentation – Restricted Boltzmann Machines* 2016):

$$\mathrm{PL}(x) = \prod_i \mathrm{P}(x_i|x_{-i}) \tag{27}$$

$$\log \mathrm{PL}(x) = \sum_i \log \mathrm{P}(x_i|x_{-i}) \tag{28}$$

Figure 20 shows how the absolute value of this cost function changes with continuous training. A visual comparison of the change against the change of SSE in the DML RBM reveals similarities (Figure 13).

Based on the reduction of the cost and the capability shown by the GPU-based RBM to learn features from the MNIST dataset, we can conclude that it is indeed a correct RBM implementation. We will therefore omit the pipeline testing (i.e. chaining a classifier after the RBM and looking at the accuracy of the output) and instead focus on the scalability of the

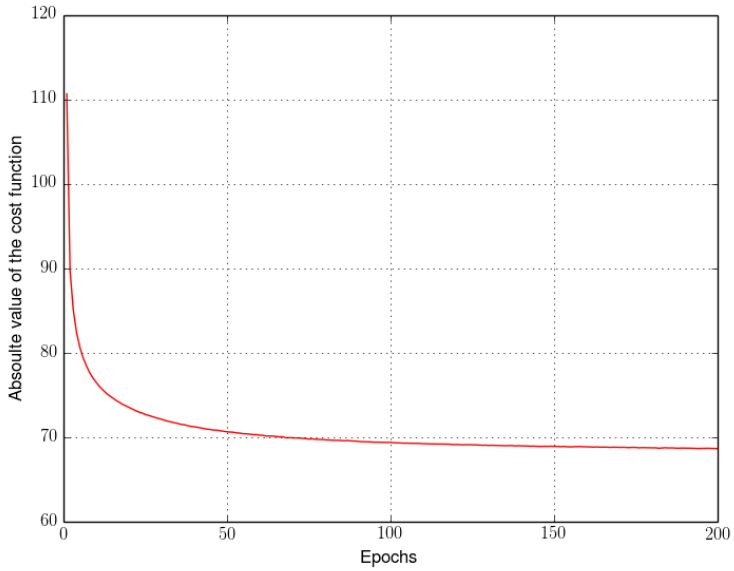


Figure 20: Changes in the cost function for the GPU-based training against the MNIST dataset.

implementation.

3.4 Project Gutenberg Dataset

A cornerstone requirement in this research is to test and validate the RBM implementation in the context of big data. We wanted to be able to confirm that the RBM implementation is able to scale and work in a multi-node, multi-CPU context, and that it is cable of processing datasets bigger than MNIST (ideally in the gigabytes range).

We looked at various freely available datasets and their suitability for feature extraction via RBMs, and we finally came to decision to use an extract from the Project Gutenberg catalogue (*Project Gutenberg* 2016).

This decision was motivated by several factors – first, as the oldest digital library (Jeffrey 2008) the dataset is relatively large (currently Project Gutenberg offer over 53'000 e-books); second, the releases are made in plain text, which facilitates processing; third, we already pointed out that ANNs have been successfully used against textual data (e.g. for classification as demonstrated by Larochelle & Bengio 2008, and extraction of semantic document representation as shown by Xing et al. 2005), which in a way reduces the risk of testing the algorithms against unsuitable data; last but not least, proofreaders and other affiliated projects provide additional meta-data for the documents, making the data suitable for supervised learning (e.g. texts are labelled with author, genre, year of publication etc.)

It is worth noting, however, that most of the documents are labelled with multiple subjects, some of the subject categories are very broad (e.g. "English literature"), and there is a

substantial overlap between categories (e.g. "English literature" and "Great Britain"). It also appears that the encoding of the subjects is a mixture of Library of Congress classification codes (*Classification and Shelflisting, The Library of Congress* 2016) and free text descriptions, so the subject labels are somewhat inconsistent. This leads to the expectation that a subject based classification attempt would not yield very high accuracy, but again we are mostly interested in the size of the data set and this issue would not impact the RBM training in any way.

For the purpose of the coursework we downloaded the latest dual layer DVD released by the project (*Project Gutenberg: The CD and DVD Project* 2016), which contains 29'500 books. The DVD is released as a single ISO with a size of 7.8GB.

3.4.1 Data exploration and preparation

Although the documents in Project Gutenberg are released as plain text files, the data set itself is not immediately ready for processing. The following issues were identified after inspecting a set of randomly selected documents and meta-data files:

- The ISO image features a complex and to a certain extent obscure directory structure, with non-descriptive paths like *1/0/0/0/10005/* being the norm.
- Most of the text files are included as ZIP archives.
- The meta-data is provided as a separate package and is not reconciled with the contents of the ISO
- There are certain quality issues with both the meta-data and the documents in the data set (e.g. missing meta-data, deviations from the established format etc.)
- Most, but not all, of the documents feature additional headers and footers added by the project, which are sometimes inconsistent and not straightforward to detect.
- The meta-data information is released in XML format, which requires additional parsing and processing for extracting the labelled data.
- Some of the documents are images, which we are not interested in as we can't process them in the context of text classification.
- There are duplicate entries in the documents collection (i.e. some documents appear multiple times in the dataset, often under different path and file name).

To remedy the aforementioned issues with the dataset each document had to be processed, any quality concerns addressed, and all present discrepancies resolved. Given the size of the dataset, we decided to implement the processing job as a PySpark script, running on top of the

Spark cluster that was provisioned for the RBM experiments. The benefits of this approach are that the raw Gutenberg data can be downloaded directly on the cluster and stored in HDFS, thus enabling us to do all the pre-processing, transformation to RBM-friendly format, and the actual RBM training in-place, without the need to move data to and out of the cluster.

Another key advantage of conducting the data preparation via Spark is that we can again leverage the automatic recovery and parallelization features provided by Spark and HDFS and make the processing time-efficient and robust.

The entire processing logic is contained within a single PySpark script named *pre-process.py* [9]

The only exception from this rule is the unpacking of the ZIP files in the dataset. We do this manually before running the *pre-process.py* script by first creating a dedicated directory for each ZIP file in the dataset, and then unpacking the individual files in their target directory. This is required to protect us from accidentally overwriting documents that might have the same name in the archive. We handle the creation/unpacking task with a simple BASH script:

```
find $DATASET_PATH -name "*.zip" | while read filename; do mkdir -p
"$TARGET_PATH‘dirname "$filename" | sed ’s|ˆ/[ˆ/]*||’‘"; done;
```

```
find $DATASET_PATH -name "*.zip" | while read filename; do unzip -o -d
"$TARGET_PATH‘dirname "$filename" | sed ’s|ˆ/[ˆ/]*||’‘" "$filename"; done;
```

After all archives are unpacked in `$TARGET_PATH` we start the execution of *pre-process.py* against the `$TARGET_PATH` directory structure. A flowchart of the operations performed by *pre-process.py* is shown in Figure 21.

Text file processing

The script starts by traversing a file system path (passed as a command line parameter) and extracting the fully qualified names of all files under this path. It also reads a list of stop words from a plain text file, which contents is shown in Appendix D on page 116.

The script then iterates over all fully qualified file names and performs the following operations:

1. If the file size is greater than 4 MB the file is skipped – we do this to filter out the images contained within the dataset. In the exploratory data analysis we confirmed that all text files in the collection are of size below 4 MB and this gives us an easy way to do the filtering based on the document size instead of having to do more complex filtering based on the file extension.

[9] We do not include the source code of the script in this report for brevity, but it is available in the source code CDs, and we also discuss all the transformation that the scripts performs on the raw dataset.

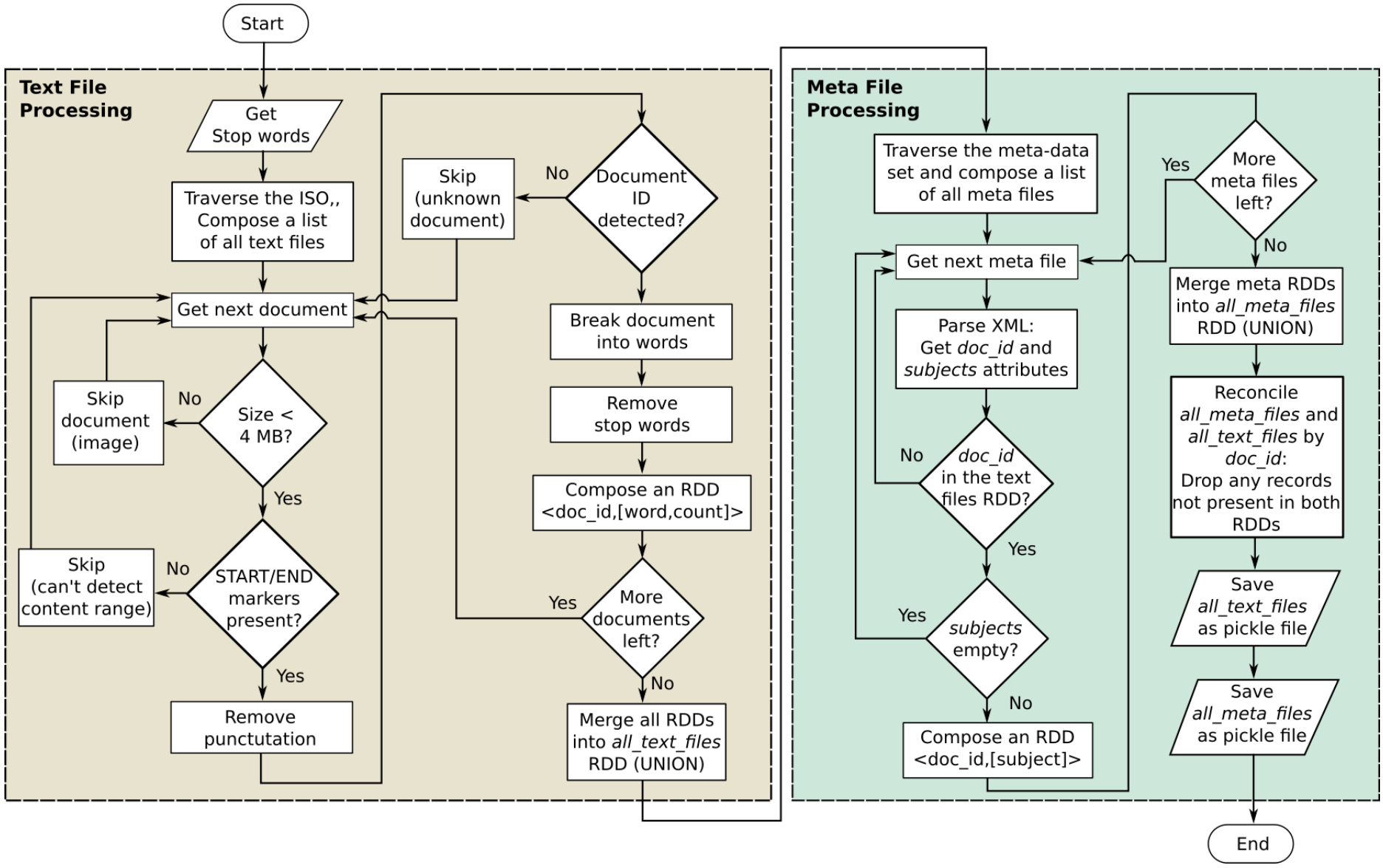


Figure 21: Project Gutenberg data set processing – flowchart diagram.

2. We load the contents of the current document in an RDD using the *textFile()* method from the Spark context.

3. The content of all documents in project Gutenberg is supposed to be surrounded by start/end markers, which separate it from the headers and footers contained in the individual text files. An example of a header followed by a start marker looks like this:

```
The Project Gutenberg EBook of Apocolocyntosis, by Lucius Seneca

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.net


Title: Apocolocyntosis

Author: Lucius Seneca

Release Date: November 10, 2003 [EBook #10001]
[Date last updated: April 9, 2005]

Language: English

Character set encoding: ASCII

*** START OF THIS PROJECT GUTENBERG EBOOK APOCOLOCYNTOSIS ***
```

An excerpt from the footer of this document, preceded by a document end marker look like this:

```
*** END OF THIS PROJECT GUTENBERG EBOOK APOCOLOCYNTOSIS ***

***** This file should be named 10001.txt or 10001.zip *****
This and all associated files of various formats will be found in:
        http://www.gutenberg.net/1/0/0/0/10001/

Produced by Ted Garvin, Ben Courtney and PG Distributed Proofreaders
```

```
Updated editions will replace the previous one--the old editions
will be renamed.
```

In order to strip the headers and footers, we compile two regular expressions

```
1 re.compile('^\*\*\*(.*)START OF(.*)PROJECT GUTENBERG(.*)')
2 re.compile('^\*\*\*(.*)END OF(.*)PROJECT GUTENBERG(.*)')
```

matching them to the document and computing the indices of the matched lines. We then *filter()* and *map()* operations to retain only the lines within the start/end marker indices.

4. We remove any punctuation from the RDD file, using a distributed *map()* operation, combined with the Python *translate()* function over each line in the RDD. The list of characters we remove is based on the *string.punctuation* constant in Python, which provides a list of ASCII characters that the language considers punctuation in the C locale.

5. Next, we look for the document ID that should be provided in the document header. We use the following regular expression to match this entry

```
1 re.compile('^.*(\[[Ee][Bb]ook|\[[Ee][Tt]ext).*$')
```

Upon successful matching we extract the digits from this entry. If the expression is unmatched or the digit extraction unsuccessful we skip the entire document as we have no means of matching it to the meta-data information.

6. We then break the text lines into words via a *flatMap()* operation, we convert them to lower case, and we remove the stop words.

7. Finally, we count the number of occurrence of each word in the document by mapping the RDD to a *((document_id, word),1)* and running a *reduceByKey(add)* action. We then remap the RDD to a *(document_id, (word, count)* tuple and run another *reduceByKey(add)*, which effectively reduces the document RDD to its final form – *(document_id, [word, count])*.

This process is repeated for each document in the data set. When the list of fully qualified documents is depleted, we merge all document RDDs into a global RDD (*all_text_files*) via a *union()* operation. This completes the text file processing part of the script.

Meta-data processing

The next step of data preparation is the parsing and reconciliation of the meta-data files. Project Gutenberg distribute the description of their entire catalogue as a collection of files in Resource Description Format (RDF).

The RDF catalogue is distributed as a single archive (see *Project Gutenberg: The Complete Gutenberg Catalog* 2016, and it contains not only RDF entries but also other files, which are of no interest (e.g. README files, index files etc.) To address this the processing script only retains files with the **.rdf* extension – all other files in the meta files directory are automatically filtered out.

We start the meta file processing phase by traversing the directory that contains all the **.rdf* files and then processing the files iteratively. As the RDF format is essentially an XML document we use the *Python ElementTree XML API* (2016) to parse the individual file.

The script then tries to find the

```
<pgterms:ebook rdf:about="ebooks/.....">
```

element and reads the trailing value of the `about` attribute, which should contain the document id. This will allow the script to match the RDF file to its corresponding text document by document ID.

The script then looks for all `<dcterms:subject>` groups and reads the RDF description value. This data is used to construct a list of subjects for the specific document.

At this stage the script aims to construct an RDD of the form *(document_id, [subject_1, subject_2,..., subject_n])*. If the document ID can not be detected, the subject list is empty, or the XML fails to parse, the document is simply discarded.

We also verify that the parsed meta-file is relevant to the collection of previously processed text files. If the document ID picked up from the RDF file is not in the set of document IDs from the unified texts RDD, the meta-data is also discarded, as this is meta-data for a document which contents is not available for processing and analysis.

After the processing of all RDF files in the list of fully qualified file names is completed, the individual RDD files are merged into a master RDD via a *union()* operation, and a *reduceByKey()* action is then triggered for the master RDD. This is required to merge any records who refer to the same document ID (it looks like multiple RDF files appear in the meta-data collection for the same document ID).

The documents and meta-data RDDs are then reconciled by document ID. This is needed in case there are entries in the documents RDD that do not have a corresponding meta-data entry in the meta-data RDD.

After the two RDDs are reconciled, they are saved as in HDFS as two SequenceFiles of pickled objects – *all_text_files* (containing the documents) and *all_meta_files* (containing the meta-data entries). This concludes the processing of the raw data and resolves all issues identified in Section 3.4.1.

3.4.2 Transformation to an RBM-ready dataset

After the data quality issues have been resolved, the next step is to transform the dataset into a format suitable for processing with an RBM.

The text documents and their respective classification labels must be transformed into a matrix as shown below.

$$\boldsymbol{D} = \begin{pmatrix} y_1 & x_{1,1} & \cdots & x_{1,n} \\ y_2 & x_{2,1} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_m & x_{m,1} & \cdots & x_{m,n} \end{pmatrix} \tag{29}$$

The matrix D contains the data for all m documents in the data set. Each row of the matrix is a vector of type $(y, x_1, \cdots, x_n)$, representing a single document, where y is the class label followed by an n dimensional feature vector.

We use a bag of words model, as suggested by Harris (1954), to construct the feature vector. This bag of words algorithm is typically implemented as a two step process, and it is outlined in Algorithm 4.

Algorithm 4 Bag of words algorithm

1: Assign a fixed id to each word occurring in all documents ▷ i.e. build an indexed global dictionary
2: **for** document $d = 1, 2, \ldots, n$ **do** ▷ Repeat for each document
3: **for each** word w in d **do**
4: Get the index for word w from the dictionary and store it in j
5: Count the number of occurrences of w
6: Store the number of occurrences in $X[d, j]$
7: **end for**
8: **end for**

The output of the algorithm is a feature vector of a chosen size n, where the position of the elements refers to their index in a global dictionary, and the values reflect the number of occurrences of each word (term frequency).

A feature vector based on term-frequency contains the number of times each word appears in the document. There is also a binary-weighing approach, where the feature vector is binary (1/0) and it only indicates if a word from the dictionary is present in the document at all. This second approach is used in the Weka 3 machine learning system (see *Weka 3: Data Mining Software in Java* 2016).

We prepared a dedicated script for transforming the data set to the RBM-ready matrix D. The name of the script is *rbm-transform.py* and its source code is shown in Appendix E on page

118.

The script supports both feature generation methods. A document processed using binary-weighing results in the following entry in the D matrix[10]

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
1,0,0,1,1,1,1,1,1,1,1,0,1,1,0,0,1,1,1,1,1,
1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,
...
```

The same document processed using the term-frequency metric in the feature vector looks like this:

```
1,66,130,47,26,142,50,30,14,53,81,59,25,32,1,53,37,31,30,31,
41,13,6,30,5,9,10,5,11,3,35,42,5,9,35,59,61,25,53,18,
0,47,15,1,1,4,29,19,7,26,29,6,5,39,14,46,5,5,7,0,
36,0,0,7,18,26,2,33,11,41,25,0,38,17,0,0,5,14,11,1,
8,10,47,0,11,1,13,38,15,5,1,12,7,10,1,9,25,0,20,16,
10,43,10,13,6,8,6,12,6,33,10,40,14,2,5,9,6,7,27,13,
26,22,0,4,3,3,17,9,10,0,5,7,22,27,5,2,3,15,5,2,
...
```

Note, that the locations of the zeroes in the two outputs are identical. If a term is not present in the document at all (for the binary-weighing) its number of appearance is also zero (for the term-frequency weighing). This validates that the script implements both approaches correctly. The choice of binary vs. term-frequency feature vector is selectable via the *boolean_bag* parameter, and the complete workflow of the *rbm_transform* script is shown in Figure 22.

The script starts by reading the combined document and meta-data RDDs from HDFS. It then applies a *flatMap()* transformation to the *(document_id, subject)* tuple from the meta-data RDD, creating a *subjects* RDD of *(subject, 1)* tuples. It then triggers a *reduceByKey()* action on *subjects*, swaps the place of the values in the tuples, and uses the *top()* method to get the top N most frequent subjects, where N is a user set parameter.

[10]Please note that for this example we used a small number of documents and a fairly large dictionary, so most of the words are indeed present in the dictionary. The sparsity of zeros in this feature vector is normal and actually expected.

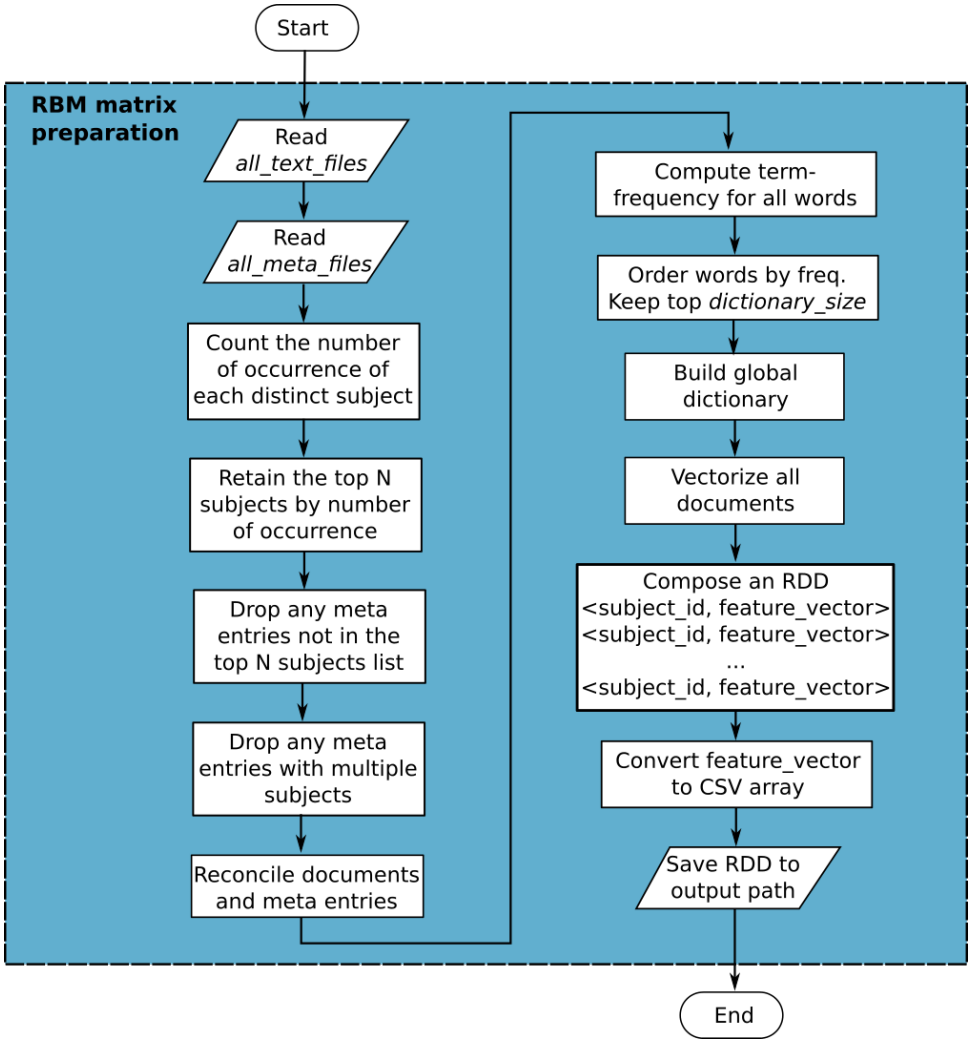


Figure 22: *rbm_transform* script – flowchart diagram.

The script proceeds by applying a set of *flatMap()* and *filter()* on the documents RDD, that retain only the documents with subjects in the "top N" list. It also encodes the subjects using IDs, so that the most common subject is encoded as the number 0, the second on the list as the number 1, and so on.

The script also counts the number of subject entries per document, and removes all documents with multiple subjects, as such entries can damage the accuracy of the classification algorithm (an issue we identified in the exploratory data analysis phase in Section 3.4.1).

The next step is to reconcile again the meta-data and document RDDs, as at this stage some documents have been removed, but their meta-data entries are still present. This is done by extracting all *document_ID*s from the documents RDD and then using them to filter out any entries in the meta-data RDD that are not in this list of IDs.

The execution proceeds by building a dictionary of size *dict_size* (a user specified parameter), based on the words from all text documents. The dictionary is built by computing the term frequency for each word in the documents RDD (mapping all words to a set of *(word, 1)* tuples and then reducing the tuples by key). Having the frequencies of all words in the set, we then take an ordered (descending) subset of size *dict_size* – this constitutes a dictionary of *dict_size* most frequently used words.

The scripts then uses the documents and the dictionary to vectorize all document representations. There are two implementations of the vectorization method and the *boolean_bag* parameter controls which one the script executes. If set to TRUE, the script uses a binary-weighing vectorization algorithm. If set to FALSE, the dictionary builds a bag of words with the frequencies of each word. The result of the vectorization is an RDD containing tuples with a subject ID and a feature vector:

$$
\begin{gathered}
(subject_i d, [x_1, x_2, \ldots, x_{dict\_size}]) \\
(subject_i d, [x_1, x_2, \ldots, x_{dict\_size}]) \\
\ldots \\
(subject_i d, [x_1, x_2, \ldots, x_{dict\_size}])
\end{gathered}
$$

This is a representation of the D matrix as defined in Equation 29.

The final step of the transformation process is to join all values in the RDD using the "," character, which converts the RDD into CSV format. The script then persists the RDD to disk, using a *saveAsTextFile()* operation and terminates.

This concludes the transformation of the data into a labelled feature vector in CSV format.

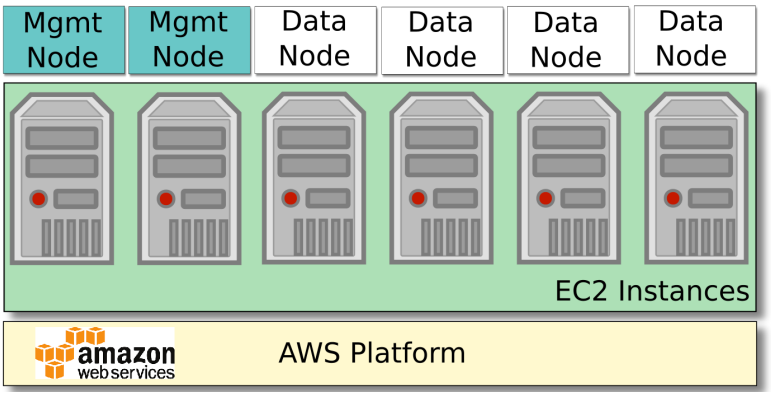


Figure 23: Architecture of the HDP cluster.

3.5 Architecture of the Hadoop cluster

For running the experiments we staged up a Hadoop cluster using the Hortonworks distribution of Hadoop (Hortonworks Data Platform – HDP). The choice of HDP as a platform was based on the following facts:

- The HDP software is provided free of charge
- The platform is 100% open-source
- HDP is ODPi compliant, which suggests that our set up can easily be replicated with any other ODPi[11] compliant Hadoop distribution

We used the Amazon Web Services (AWS), and more specifically – the Elastic Cloud 2 (EC2) infrastructure for provisioning and managing the Hadoop nodes. EC2 provides capacity elasticity, which makes it ideal for carrying out scalability tests.

For the Hadoop nodes we provisioned a group of six "General Purpose m4.large" virtual instances as shown in Figure 23. The hardware specifications of an individual instance are listed in Table 6.

Two of the six nodes were assigned the management role – they ran the Hadoop Name Node, Job Tracker, and other management services. The remaining four nodes had the DataNode role assigned to them and they ran the Task Trackers and Spark Executor services.

For the installation the HDP installation we followed the EC2-based installation process outlined by Hortonworks – *Deploying a Hadoop Cluster on Amazon EC2 with HDP2* (2016). All but two of the HDP parameters were left at the default value:

- *dfs.replication* – We set the default block replication to 4, to make sure that each data block is available on every data node, so that MapReduce and Spark can leverage data locality

[11] ODPi is a non-profit organisation, which provides common Apache Hadoop runtime, reference implementation, and test suites.

Component	Description
Instance Type	m4.large
vCPU(s)	2 x 2.4 GHz, Intel Xeon E5-2676v3
Memory	8 GB
Storage	EBS Optimised – 40 GB magnetic
Network performance	Moderate

Table 6: EC2 Hadoop node hardware specification

- *dfs.blocksize* – We set the default HDFS block size for new files to 64 MB

Apache SystemML version 0.10.0 was deployed on the first management node by simply downloading and extracting the latest binary package available at `https://systemml.apache.org`. As part of the configuration process, we set the following defaults in the SystemML's *sparkDML.sh* script, which drives the Spark back-end execution:

```
DEFAULT_SPARK_HOME=/usr/hdp/current/spark-client
DEFAULT_SYSTEMML_HOME=/home/ec2-user/systemml-0.10.0
driver_memory="--driver-memory 4G"
executor_memory="--executor-memory 2G"
```

The functionality of the cluster was tested by first uploading a subset of the Gutenberg data set – a 10'000 RBM-ready entries file named *rbm-full-10k.csv*. The file was placed on the local file system of the first management node. We then used a SystemML script called *splitXY.dml*[12] to separate the input features (X) from the class labels (Y).

We ran the script using the MapReduce back-end:

```
hadoop jar SystemML.jar -f scripts/utils/splitXY.dml -nvargs
X=/user/ec2-user/rbm-full-10k.csv y=1 OX=/user/ec2-user/rbm-10k-x.csv
OY=/user/ec2-user/rbm-10k-y.csv ofmt=csv
```

And we repeated the process using the Spark back-end:

```
./sparkDML.sh -f utils/splitXY.dml --nvargs X=/user/ec2-user/rbm-full-10k.csv
y=1 OX=/user/ec2-user/rbm-10k-x.csv OY=/user/ec2-user/rbm-10k-y.csv ofmt=csv
```

After making sure that both commands execute successfully, we put the resulting files into HDFS (via a *hadoop fs -put* command) and we verified the block size, the replication factor, and that no under-replicated blocks exist on the cluster:

[12]This script is part of the standard SystemML distribution

Component	Description
Instance Type	g2.2xlarge
vCPU(s)	8x 2.6GHz Intel Xeon E5-2670 (Sandy Bridge)
Memory	15 GB
Storage	EBS Optimised – 20 GB magnetic
GPU	1x NVIDIA GPU with 1'536 CUDA cores and 4GB of video memory
Network performance	High

Table 7: EC2 G2 instance hardware specification

```
$ hdfs dfs -stat 'name=%n blocksize=%o' rbm-10k-x.csv/0-m-00000
name=0-m-00000 blocksize=67108864
$ hdfs fsck / | grep 'Under replicated' | awk -F':' '{print $1}'
$
```

3.6 Environment for the GPU-based training

We selected a g2.2xlarge EC2 instance to server as a platform for the GPU-based training. The G2 instances are optimized for graphics-intensive applications and feature NVIDIA GRID K520 GPUs . The hardware specifications of the selected instance are listed in Table 7.

In order to access the GPU, the instance requires the installation of an appropriate NVIDIA driver. Amazon provides an Amazon Machine Image (AMI) witn 64-bit Linux and compatible builds of the NVIDIA kernel drivers for NVIDIA GRID K520.

The Linux AMI is based on Red Hat Enterprise Linux. This, and the fact that it comes with the NVIDIA drivers pre-installed made it a natural choice for an operating system for the G2 instance.

The only changes we had to make from the default build was the installation of the following additional Python, GCC, and BLAS[13] packages:

```
python-devel python-nose python-setuptools gcc gcc-gfortran gcc-c++
blas-devel lapack-devel atlas-devel
```

We then installed the Python package manager (`pip`), NumPy, SciPy, and Theano. We also ran a thorough test of the deployed frameworks using:

```
python -c "import numpy; numpy.test()"
```

[13]Basic Linear Algebra Subprograms

Label	LC Description
PS	American literature
PR	English literature
PZ	Fiction and juvenile belles letters
PQ	French literature, Italian literature, Spanish literature, Portuguese literature
AP	Periodicals
Short stories	–
Fiction	–
PT	German literature, Dutch literature, Flemish literature since 1830, Afrikaans literature, Scandinavian literature, Old Norse literature: Old Icelandic and Old Norwegian, Modern Icelandic literature, Faroese literature, Danish literature, Norwegian literature, Swedish literature
Science fiction	–
DA	Great Britain

Table 8: Top 10 subjects

```
python -c "import scipy; scipy.test()"
python -c "import theano; theano.test()"
```

All tests completed successfully with a total runtime of about 2 hours. We copied the MNIST dataset, the RBM-ready CSV data from Project Gutenberg, and the Theano RBM scripts onto the G2 instance. At this point the G2 instance was ready for running the functional and scalability tests for a GPU-based Restricted Boltzmann Machine training.

3.7 Running the experiments

After cleaning up the raw data with the text and meta-data processing scripts (discussed in Section 3.4.1), we transformed the documents into a matrix of labels and feature vectors using the RBM data transformation script (Listing 4).

We set the number of subjects to ten ($N = 10$) and the extracted labels for the top 10 subjects and their description is shown in Table 8. We sampled the RBM-ready dataset (with replacement) and created a set of document collections of varying size as shown in Table 9. Please note that we used the SystemML *splitXY.dml* script to separate the feature vectors and class labels into separate files.

The resulting files were kept in HDFS, and the execution of the RBM training was handled

Document count	Features file	Size of features file	Labels file
10'000	rbm-10k-x.csv	0.26 GB	rbm-10k-y.csv
20'000	rbm-20k-x.csv	0.53 GB	rbm-20k-y.csv
40'000	rbm-40k-x.csv	1.00 GB	rbm-40k-y.csv
60'000	rbm-60k-x.csv	1.60 GB	rbm-60k-y.csv
80'000	rbm-80k-x.csv	2.10 GB	rbm-80k-y.csv
100'000	rbm-100k-x.csv	2.61 GB	rbm-100k-y.csv

Table 9: Document count and respective file names for the sampled RBM data

using the *yarn jar* command for the MapReduce back-end and the *spark-submit* command for the Spark back-end.

We ran a series of experiments to determine the optimal batch-size and then trained RBMs using the prepared collections of varying document counts.

We collected a group of metrics like average training time per epoch in seconds, which was typically based on an average over 10 sequential training epochs. The time was measured using the wall clock time produced by the *time* command and includes the latency for standing and compiling the programs by SystemML.

We also looked at cluster specific metrics like the CPU, network, and memory utilisation. This information was extracted from Ambari – the Hadoop management framework.

In addition, SystemML provides an option to print the execution plan generated for each run, so we also looked at how the plan changes with the increase of input data and changes in the resource availability (e.g. number of Spark executors). An excerpt of a plan description is shown below.

```
...
------CP createvar pREADX /user/ec2-user/rbm-10k-x.csv false MATRIX csv 10905
 10000 -1 -1 29704283 false false , true 0.0
------CP print BEGIN RBM MINIBATCH TRAINING SCRIPT.SCALAR.STRING.true
_Var1.SCALAR.STRING
------CP print Reading X....SCALAR.STRING.true _Var2.SCALAR.STRING
------CP createvar _mVar3 scratch_space//_p21881_172.31.1.50//_t0/temp1
 true MATRIX binaryblock 10905 10000 1000 1000 29704283 false
------MR-Job[
----------  jobtype        = CSV_REBLOCK
----------  input labels   = [pREADX]
----------  recReader inst =
----------  rand inst      =
```

```
----------  mapper inst    =
----------  shuffle inst   = MR csvrblk 0.MATRIX.DOUBLE 1.MATRIX.DOUBLE
1000 1000 false , true 0.0
----------  agg inst       =
----------  other inst     =
----------  output labels  = [_mVar3]
----------  result indices = ,1
----------  num reducers   = 10
----------  replication    = 1 ]
------CP uamin _mVar3.MATRIX.DOUBLE _Var4.SCALAR.DOUBLE 2
------CP uamax _mVar3.MATRIX.DOUBLE _Var5.SCALAR.DOUBLE 2
...
```

The CP prefix about denotes a Control Program operation – a pure single-node, in-memory operation. The MR-Job prefix stands for a distributed MapReduce job. The snippet above starts with the creation of a temporary variable to hold the input data (persistent read for X), then it prints some information on the command line, sets up a scratch space, and triggers a MapReduce operation to read the data from HDFS into memory. The program finishes with the computation of the maximum and minimum values in the X matrix (*uamax* and *uamin*), which are performed in the driver's memory only. More details on reading the output of the SystemML programs execution plan is given in Boehm et al. (2014) and Boehm (2015).

3.8 Conclusion

So far we showed that we have a custom parallelized DML implementation based on Algorithm 2, which can correctly act as a feature extractor, and it can speed up the convergence of other algorithms without hurting their accuracy. We also confirmed that a GPU-based RBM implementation can process the data set used in the functional tests with similar outcomes.

We have also prepared the necessary distributed infrastructure (Hadoop), following best practice recommendations, for running the scalability tests. A fairly large dataset was selected, explored, and transformed in line with the requirements of the RBM model and the pending scalability tests.

We also established our plan for running a series of tests with different back-ends (Standalone, MapReduce, and Spark), while varying the cluster configuration, the model parameters, and the size of the dataset. Our expectation is that the data gathered from the experiments should provide sufficient evidence and will enable us to address the research question.

4 Results

In this chapter we present the outputs from the experimental work outlined in Section 3. We start with showing the impact of batch-size selection on our custom RBM implementation, and we proceed with reporting the results from the scalability tests using the MapReduce and Spark back-ends. We show the impact of increased data volumes on the learning process runtime, and we perform statistical analysis in an attempt to establish the type of relationship between runtime and the number of participating cluster nodes.

We also report on the maximum amount of data that a non-parallelized (single node) RBM can handle, and we use this as a baseline for putting the results from the parallelized execution in context. We also repeat the scalability tests with the GPU-based RBM implementation and we compare the effectiveness of the multi-node versus single GPU node parallelization.

4.1 CPU-based RBM

4.1.1 Selection of optimal batch size

Hinton (2012) provides a recipe for dividing the training set into mini-batches. In the case of a small number of classes with equal probability, the suggestion is to set the batch size to the number of classes.

Batch size	Network Error	Runtime (s)
10	52'348'826	4005
25	5'936'424	2258
50	5'498'178	1524
100	5'283'563	1187
200	5'654'799	1160
400	5'329'277	1087
800	545'471	1050
1'600	523'448	1057
3'200	–	2202 (failed)

Table 10: Impact of the batch size on the runtime and network error at the end of the RBM training.

In order to select the optimal batch-size we started with a size of 10 as the base-line (following Hinton's recommendation and given that we had 10 distinct subjects/classes). We used a sample of 20'000 documents and we ran the RBM training script (*rbm_minibatch.dml*) for 5 epochs. The number of neurons in the hidden layer was set to 500, and we used a value of 0.1 for η (as determined in Section 3.2). The script execution was performed using the Spark back-end.

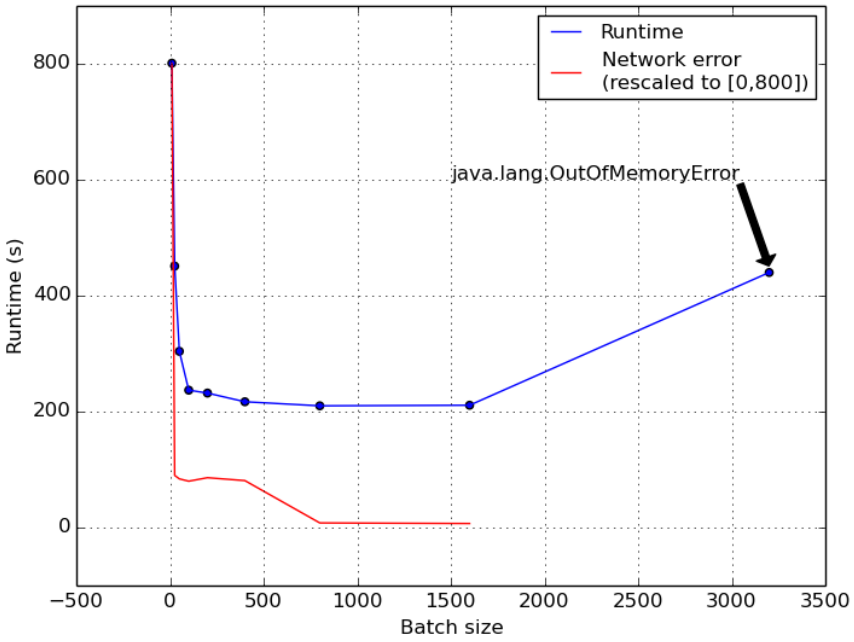


Figure 24: Average time per epoch for different batch sizes. Hybrid Spark back-end with 20'000 documents

The results from this experiment are listed in Table 10. The runtime and the rescaled network error at the end of the final epoch is also plotted in Figure 24. It appears that a batch size of 10 doesn't work well in a Spark-based environment. In our implementation each batch loop triggers multiple Spark operations and the latency associated with performing said operations substantially impacts the runtime.

It is also worth noting that the reduction of network error[14] at the end of the training is viable at least with 100 examples in each mini batch. We can attribute this effect to the fact that we could not guarantee that all classes will be represented in each mini-batch, so most of the batches were contributing to a sampling error in the process of estimating the gradient. This error reduces naturally with the increase of the batch size, leading to better models at the end of the training phase.

Increasing the batch size beyond 800 does not show any significant effect on the runtime or the network error, and in increasing it further leads to the execution failing with an *OutOfMemoryErorr*. This is also expected as increasing the batch size also increases the dimensions of the $\boldsymbol{X}$ matrix (see Listing 1) beyond what the local Spark executor can fit in their reserved memory space.

Given these results we settled on using two different batch sizes. A batch size of 500 is

[14]Please note that as this is unsupervised learning, the term "network error" is used to denote the normalised squared difference between the data and the states of the visible units predicted by the model – $\frac{1}{N}\sum(\boldsymbol{y} - p(\boldsymbol{v_2} = 1|\boldsymbol{h_1}))^2$, as explained in Section 3.2

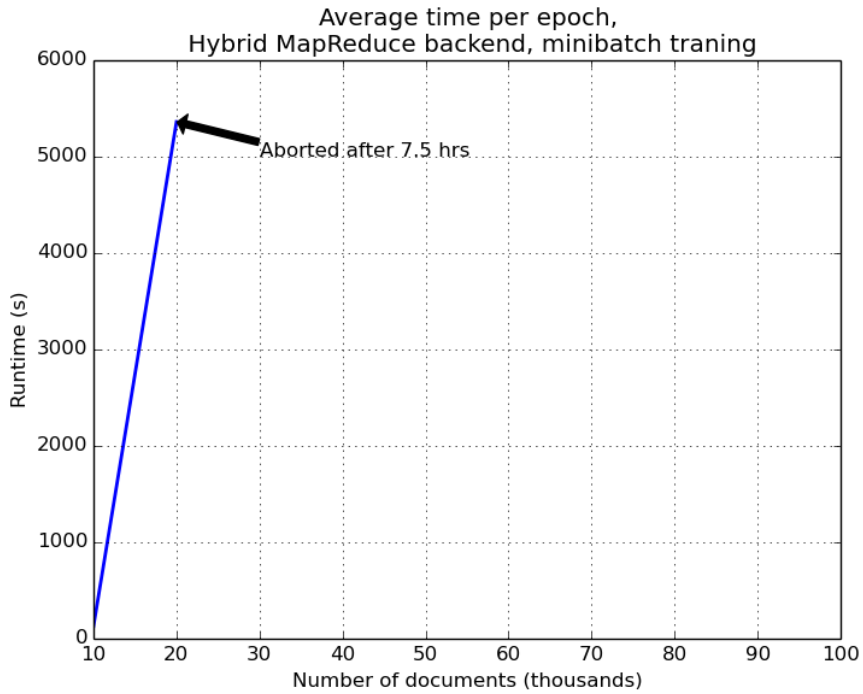


Figure 25: Average time per epoch for a MapReduce-based RBM training using mini-batches and varied number of documents in the data set.

a good compromise of memory requirements, runtime, and reduction of error. On the other hand a batch size of 100 would provide the lowest memory footprint combined with the lowest execution time per epoch (although it will require more epochs to achieve similar levels of error reduction as with the 500 batch size).

4.1.2 Results with the MapReduce back-end

We started the distributed execution using the MapReduce back-end and 10'000 documents in the data set. We tried to train an RBM with 500 features, computing the average training times from 10 epochs and using a mini-batch size of 100 observations. The results from this trial are shown in Figure 25.

The processing of the 10'000 data set took about 10 minutes and 40 seconds (640 seconds). The execution of the next set of documents – the 20'000 documents did not complete and we aborted the execution after letting it run for 7.5 hours (26'808 seconds). During this time the system successfully processed 744 MapReduce jobs and there was no indication that the job will fail. The processing time that the execution was taking, however, was making it infeasible.

Looking at the difference in the execution plans generated by SystemML makes it evident that in fact the 10'000 entries data set was so small, that SystemML was capable of performing most of the processing in the master node alone, not needing to trigger distributed MapReduce execution. For example, the code below computes the minimum and maximum values in the X

matrix for the purposes of normalization (see Listing 1, lines 88 and 89):

```
# Rescale to interval [0,1]
minX = min(X)
maxX = max(X)
```

When using the 10'000 entries set SystemML translates this code to:

```
...
------CP uamin _mVar3.MATRIX.DOUBLE _Var4.SCALAR.DOUBLE 2
------CP uamax _mVar3.MATRIX.DOUBLE _Var5.SCALAR.DOUBLE 2
------CP assignvar 0.1.SCALAR.DOUBLE.true learning_rate.SCALAR.DOUBLE
...
```

With the 20'000 entries data set the two control program operations (*uamin* and *uamax*) get replaced with a distributed MapReduce job that computes the minimum and maximum using *uamin* and *uamax* as mapper and aggregation functions.

```
...
------CP createvar _mVar4 scratch_space//_p31194_172.31.1.50//_t0/temp2 true
  MATRIX binaryblock 1 1 1000 1000 -1 false
------CP createvar _mVar5 scratch_space//_p31194_172.31.1.50//_t0/temp3 true
  MATRIX binaryblock 1 1 1000 1000 -1 false
------MR-Job[
----------  jobtype        = GMR
----------  input labels   = [_mVar3]
----------  recReader inst =
----------  rand inst      =
----------  mapper inst    = MR uamin 0.MATRIX.DOUBLE 1.MATRIX.DOUBLE false,
MR uamax 0.MATRIX.DOUBLE 2.MATRIX.DOUBLE false
----------  shuffle inst   =
----------  agg inst       = MR amin 1.MATRIX.DOUBLE 3.MATRIX.DOUBLE,
MR amax 2.MATRIX.DOUBLE 4.MATRIX.DOUBLE
----------  other inst     =
----------  output labels  = [_mVar4, _mVar5]
----------  result indices = ,3,4
----------  num reducers   = 10
----------  replication    = 1 ]
------CP castdts _mVar4.MATRIX.DOUBLE.false _Var6.SCALAR.DOUBLE
------CP rmvar _mVar4
```

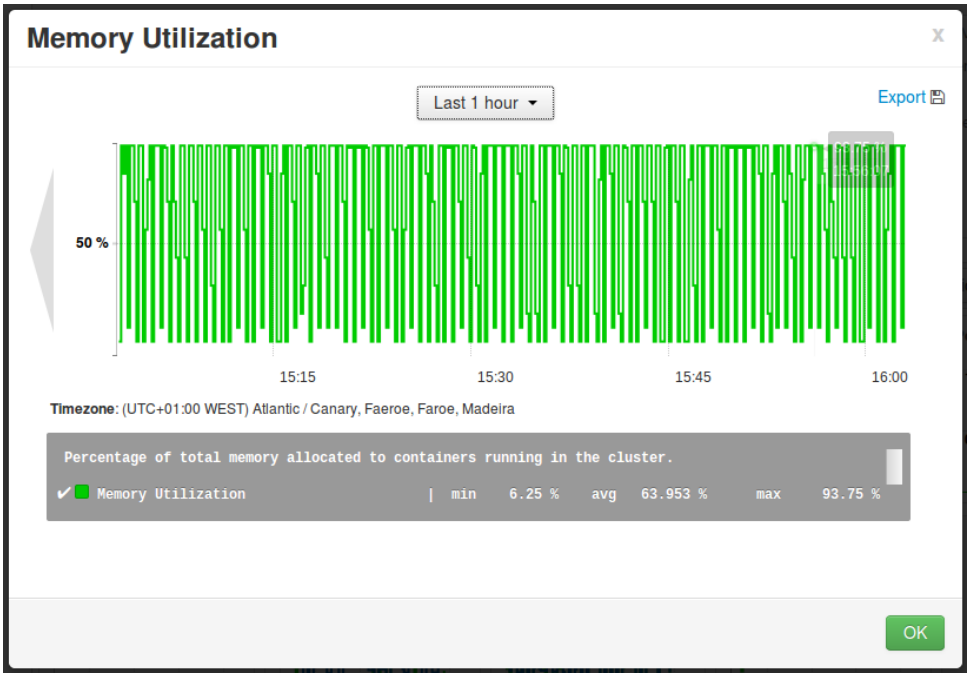


Figure 26: Executor memory utilisation during mini-batch RBM training with MapReduce back-end.

```
------CP castdts _mVar5.MATRIX.DOUBLE.false _Var7.SCALAR.DOUBLE
------CP rmvar _mVar5
------CP assignvar 0.1.SCALAR.DOUBLE.true learning_rate.SCALAR.DOUBLE
...
```

As expected, the distributed processing results in a series of MapReduce jobs. The number of jobs and the latency for staging and running them, however, takes a substantial toll on the runtime. It also appears that this series of hundreds of MapReduce jobs also hurts the overall utilisation of the cluster.

This is clearly visible in Figure 26 and Figure 27. The jigsaw profiles of memory and CPU utilisation reflect the nature of the workload when using MapReduce jobs – when a job is running it allocates memory and uses the available CPU capacity. When the job completes all resources are deallocated and there is a sharp drop in the utilisation until the next job in the series is staged for execution. We can conclude that this constant state shift of resources combined with the latency associated with scheduling new jobs leads to the unacceptably long run times.

Looking at the execution plan for the 20'000 examples job reveals that there are 8 MapReduce jobs enclosed in the epoch *for* loop (see Listing 1, lines 105 to 142). Given that we run the training process using 100 entries in each batch, this would result in $\frac{20000}{100} \times 8 = 1600$ MapReduce jobs per epoch. Based on this figure we could speculate that training a single epoch would take around 16 hours (given that it takes roughly $\frac{26808}{744} = 36.03$ seconds to process a single job

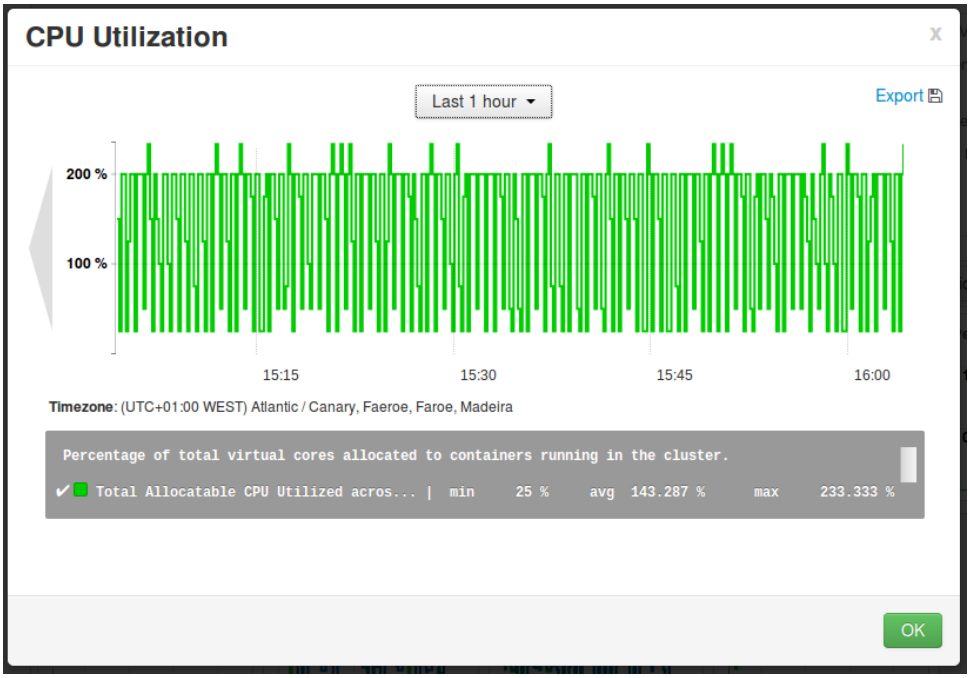


Figure 27: Executor CPU utilisation during mini-batch RBM training with MapReduce back-end.

of the workload).

Out of curiosity we also used SystemML to generate an execution plan for processing the 100'000 data set to see if the increased dimensionality of the input will have impact on the plan. The resulting execution plan was 100% identical to the plan generated by SystemML for the 20'000 data set, so trying bigger sets was proven pointless given that fairly small sets can not be successfully processed using this execution plan.

In an attempt to remedy the situation we looked at running the RBM training without using mini-batches, trying to process the entire data set in one iteration per epoch as shown in Listing 2. The logic behind this decision is that removing the inner *for* loop that loops over each batch, should in theory reduce the number of MapReduce jobs that needs to be staged and executed.

The results from the no mini-batch executions are shown in Figure 28. The modified script was capable of processing both the 10'000 and the 20'000 data sets, producing similar runtimes[15] for the non-distributed (10'000) run – 139 seconds for the non-minibatch version versus 128 seconds for the mini-batch run. Parallelization aside, this result alone confirms the findings of Hinton (2012) that mini-batch training is the more efficient approach for training RBMs.

The non-mini-batch version of the script allowed us to process the 20'000 data set in a distributed fashion for a total time of 1 hour and 54 minutes (5550 seconds).

Looking at the execution plan generated by SystemML confirms our expectations in terms

[15] Averaged over 10 epochs

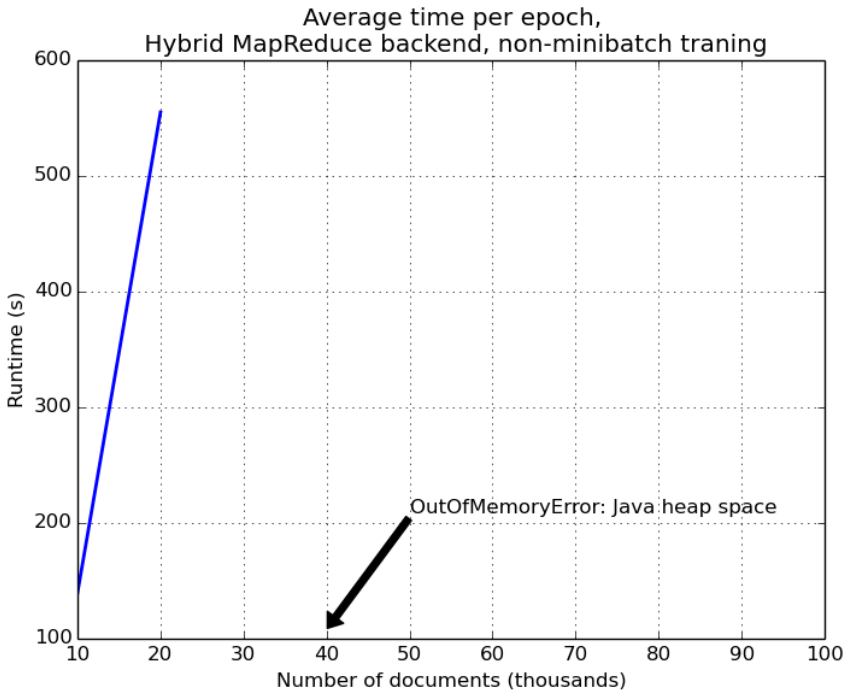


Figure 28: Average time per epoch for a MapReduce-based RBM training without mini-batches and using varied number of documents in the data set.

of number of MapReduce jobs. The mini-batch training program generates an execution plan featuring 10 unique MapReduce jobs (8 in the epoch loop):

```
# Memory Budget local/remote = 637MB/860MB/1147MB
# Degree of Parallelism (vcores) local/remote = 2/40/8
PROGRAM ( size CP/MR = 0/10 )
--MAIN PROGRAM
----GENERIC (lines 63-83) [recompile=false]
...
```

The non-mini-batch version has only 5 unique MapReduce jobs and just 3 of them within the epoch *for* loop:

```
# Memory Budget local/remote = 637MB/860MB/1147MB
# Degree of Parallelism (vcores) local/remote = 2/40/8
PROGRAM ( size CP/MR = 0/5 )
--MAIN PROGRAM
----GENERIC (lines 63-82) [recompile=false]
...
```

It is also worth noting that while the epoch and batch loops in the mini-batch version of the script have static plans

```
...
----FOR (lines 99-150)
------GENERIC (lines 102-102) [recompile=false]
--------CP assignvar 0.SCALAR.INT.true epoch_err.SCALAR.INT
------FOR (lines 105-145)
--------GENERIC (lines 108-108) [recompile=false]
----------CP + batch_start.SCALAR.INT.false 100.SCALAR.INT.true _Var15.SCALAR.INT
...
```

the non-mini-batch version has a recompilation hook in its execution plan:

```
...
------CP rmvar visible_units_count
----FOR (lines 98-133)
------GENERIC (lines 101-130) [recompile=true]
...
```

This usually happens when SystemML is unsure of the dimensions of an intermediate result and recompiles parts of the program during its execution, after the dimensions in question become available. This typically leads to a more efficient execution plan.

Although the non-mini-batch version of the script was capable of performing a distributed processing with the 20'000 data set, it failed at the next stepping stone – the 40'000 data set. This execution failed with an "Out of memory Java heap space" error just 11 minutes into the execution (697 seconds). It appears that without the use of mini-batches the 40'000 has passed the threshold that will let the partitioned data fit in the memory of the worker node executors.

In summary, the MapReduce back-end could not efficiently distribute the mini-batch training process due to the explosion of MapReduce jobs caused by the two inner loops and the latency associated with staging the individual jobs. Modifying the training script to reduce the number of jobs helps to a certain extent, but increasing the size of the data set quickly reaches the resource limits of the worker node executors.

4.1.3 Results with the Spark back-end

We repeated the scalability experiments using the *rbm-minibatch.dml* script on SystemML with execution push down to Spark. This back-end performed much better compared to MapReduce and SystemML was able to process all datasets successfully. The processing times (averaged over 10 epochs) for the individual feature files are shown in Table 11 and Figure 29.

Similar to the results with the MapReduce back-end, SystemML does not use distributed computations for the 10'000 examples dataset. It is evident that when processing the 10'000

Document count	Features file	Runtime (s)
10'000	rbm-10k-x.csv	121
20'000	rbm-20k-x.csv	241
40'000	rbm-40k-x.csv	703
60'000	rbm-60k-x.csv	978
80'000	rbm-80k-x.csv	1'759
100'000	rbm-100k-x.csv	2'672

Table 11: Average time per training epoch (in seconds), using hybrid Spark back-end and mini-batch training

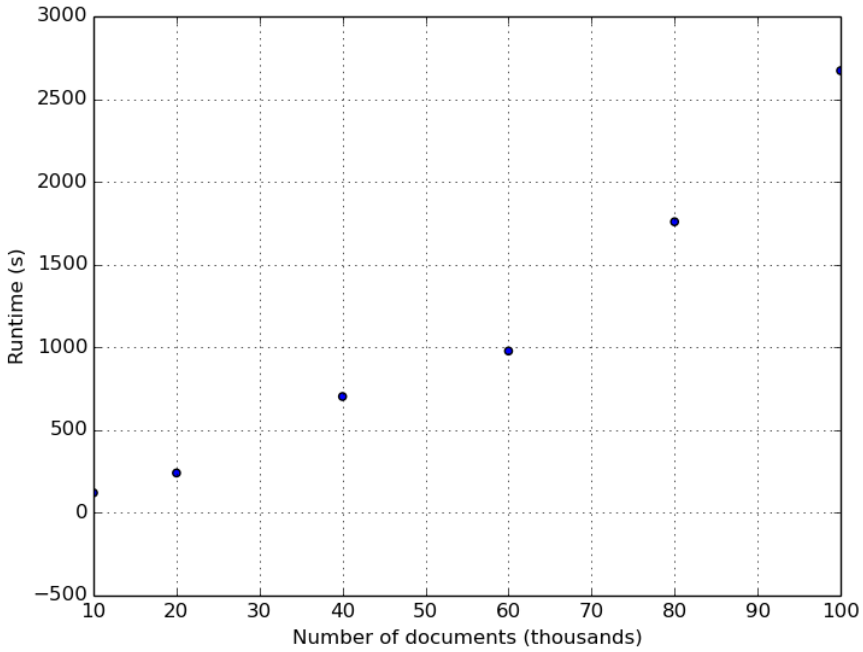


Figure 29: Average time per epoch for a mini-batch, Spark-based RBM training using varied number of documents in the data set.

examples, the driver does not allocate and use resources in the executors(Figure 30a). The 20'000 examples dataset, however, requires a distributed computation as it does not fit in the driver's memory, so SystemML runs a series of distributed matrix operations that utilize Spark executor resources (Figure 30b).

Comparing the non-distributed execution (MapReduce vs. Spark on the 10'000 examples) shows that even in local execution mode Spark is slightly faster than MapReduce – 608 seconds vs. 640 seconds for the MapReduce back-end.

This is interesting, especially given the fact that the MapReduce execution plan features just 9 local MapReduce operations in contrast with the Spark-based plan that has 19 Spark operations.

It appears indeed that the MapReduce execution plan tries to pack as many operations as possible in a single map step. This behaviour is expected and is in line with the low level operations DAG (LOP DAG) construction phase. For example, the following assignment from the *rbm-minibatch.dml* scripts

```
X = (X-minX)/(maxX-minX)
```

translates into a single operation in the MapReduce-based plan:

```
...
--------MR-Job[
------------  jobtype        = GMR
------------  input labels   = [X, minX, _Var10]
------------  recReader inst =
------------  rand inst      =
------------  mapper inst    = MR - 0.MATRIX.DOUBLE minX.SCALAR.DOUBLE.false
  1.MATRIX.DOUBLE, MR / 1.MATRIX.DOUBLE _Var10.SCALAR.DOUBLE.false
  2.MATRIX.DOUBLE
------------  shuffle inst   =
------------  agg inst       =
------------  other inst     =
------------  output labels  = [Xtemp]
------------  result indices = ,2
------------  num reducers   = 0
------------  replication    = 1 ]
...
```

There are, however, two Spark operations handling the same assignment in the Spark-based plan:

Executors (5)

Memory: 0.0 B Used (2.4 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
1	ip-172-31-41-3.eu-west-1.compute.internal:44053	0	0.0 B / 511.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr
2	ip-172-31-41-5.eu-west-1.compute.internal:60440	0	0.0 B / 511.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr
3	ip-172-31-41-2.eu-west-1.compute.internal:35764	0	0.0 B / 511.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr
4	ip-172-31-41-6.eu-west-1.compute.internal:51717	0	0.0 B / 511.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr
driver	172.31.41.3:47182	0	0.0 B / 457.9 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stderr stdout

(a)

Executors (5)

Memory: 0.0 B Used (2.4 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
1	ip-172-31-41-5.eu-west-1.compute.internal:36799	0	0.0 B / 511.1 MB	0.0 B	0	0	4487	4487	1.2 m	174.7 GB	22.1 MB	192.4 MB	stdout stderr
2	ip-172-31-41-6.eu-west-1.compute.internal:37291	0	0.0 B / 511.1 MB	0.0 B	0	0	8862	8862	1.7 m	322.3 GB	43.4 MB	336.8 MB	stdout stderr
3	ip-172-31-41-2.eu-west-1.compute.internal:37363	0	0.0 B / 511.1 MB	0.0 B	0	0	6491	6491	1.4 m	244.2 GB	31.1 MB	257.1 MB	stdout stderr
4	ip-172-31-41-3.eu-west-1.compute.internal:56131	0	0.0 B / 511.1 MB	0.0 B	0	0	2099	2099	43.6 s	72.2 GB	9.3 MB	103.9 MB	stdout stderr
driver	172.31.41.5:55984	0	0.0 B / 457.9 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stderr stdout

(b)

Figure 30: (a) Resource utilisation in the Spark executors when processing the 10'000 examples dataset. (b) Resource utilisation in the Spark executors when processing the 20'000 examples dataset.

Document count	**Distributed operations in the execution plan**
10'000	19
20'000	22
40'000	24
60'000	24
80'000	27
100'000	27

Table 12: Change in number of distributed Spark operations with increase of the size of the training dataset. Note, that the first entry in the table is actually CP driven execution, so the 19 Spark operations are executed locally on the driver machine.

```
...
--------CP createvar _mVar10 scratch_space//_p7783_172.31.5.22//_t0/temp2 true
MATRIX binaryblock 10905 10000 1000 1000 -1 false
--------SPARK - X.MATRIX.DOUBLE minX.SCALAR.DOUBLE.false _mVar10.MATRIX.DOUBLE
--------CP - maxX.SCALAR.DOUBLE.false minX.SCALAR.DOUBLE.false _Var11.SCALAR.DOUBLE
--------CP createvar _mVar12 scratch_space//_p7783_172.31.5.22//_t0/temp3 true
MATRIX binaryblock 10905 10000 1000 1000 -1 false
--------SPARK / _mVar10.MATRIX.DOUBLE _Var11.SCALAR.DOUBLE.false
_mVar12.MATRIX.DOUBLE
--------CP rmvar _mVar10
...
```

This does not appear to be a problem, as the Spark operations run within the same Spark application and this approach does not suffer from the penalty introduce by the latency of scheduling separate MapReduce jobs.

Increasing the number of examples in the dataset consistently leads to a higher number of Spark operations, as SystemML schedules more and more tasks for distributed processing as the size of the matrices grows. The change in number of distributed operations with increase of the input size is given in Table 12.

Another experiment we ran was to try and asses the impact of number of processing nodes (executors) on the processing time. We selected a relatively large set (60'000 examples) and trained the RBM for 10 epochs using mini-batches. In line with our expectations, adding more nodes to the cluster decreases the processing – see Figure 31.

For completeness, we also looked at the performance of the Spark back-end when training the

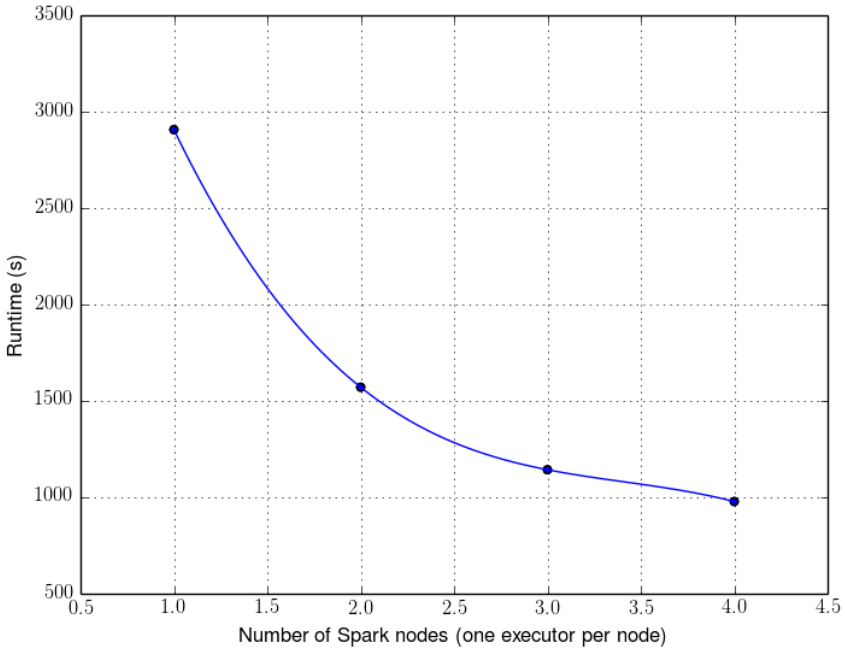


Figure 31: Impact on runtime when adding more processing nodes to the cluster. 60'000 examples, average runtime over 10 training epochs.

RBM without using mini-batches. Our findings from this experiment were consistent with the no-mini-batch training using the MapReduce back-end – the script fails with an out-of-memory error and interestingly, it was not even able to process the 20'000 dataset.

Looking at resource utilisation, the Spark back-end performed much better. The jigsaw profile observed during the MapReduce runs (see Figure 26 and Figure 27) is no longer present. The memory and CPU resource utilisation is steady, as the Spark operations run within the same application and are not staged as separate jobs.

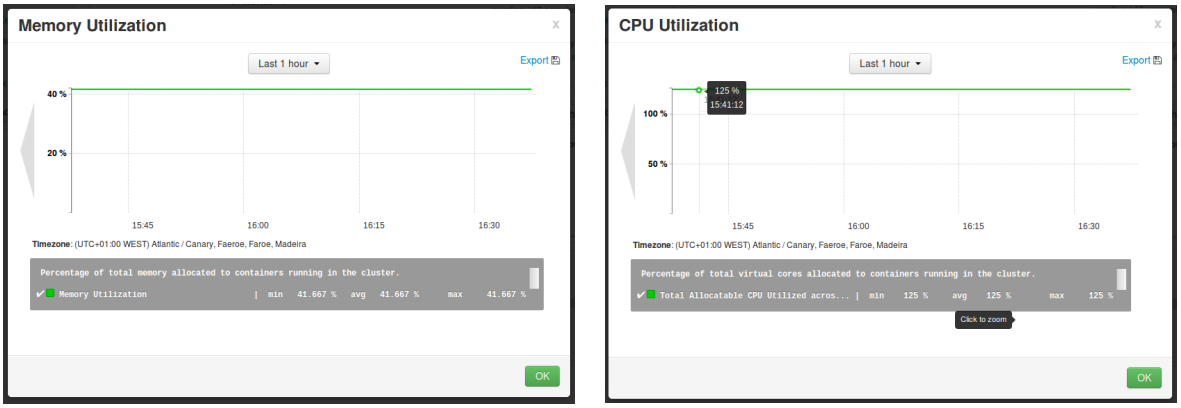
The peak memory utilisation for the executors never went beyond 42% and the CPU utilisation peaked around 125%[16]. The average memory utilisation for the entire Spark cluster stayed around 35%, and the average CPU utilisation about 77%. Looking at Disk I/O and Network utilisation also confirmed that the cluster was not starved for resources at any given point, and the training was running with maximum efficiency.

Statistical Analysis

We made an attempt to determine the type of relationship between the number of documents and the average training time per epoch. We were interested to see if the RBM training scales linearly with increase of its training data.

To answer this question we regressed the data in Figure 29, applying a Linear Regression

[16]The CPU utilisation is above 100% because of the Hyper-threading used in the CPUs



(a) (b)

Figure 32: Memory and CPU utilisation of the containers running the Spark executors during the RBM training.

Regression type	**Intercept**	x	x^2	x^3
Simple Regression	0.1389	0.0009	-	-
2^{nd} degree polynomial	0.2768	0.7896	0.0062	-
3^{rd} degree polynomial	0.9180	0.3650	0.9020	0.3780

Table 13: Coefficient significance test for different regression model. This table shows the probability of observing a value greater or equal than the test statistic ($\Pr > |t|$).

model. The resulting model produced a coefficient of determination $R^2 = 0.9379$, indicating that the model explains most of the variability of the response data around its mean.

While R^2 is an indicator of how well the model fits the data, it is a biased estimate, and it cannot be used to formally test the hypothesis of the relationship between the model and the dependent variable. The p-value from the regression model (p-value $= 0.000941$), however, suggests that the training volumes are a significant predictor on the expected runtime.

We also looked at regressing the data using higher order polynomials and checking for coefficient significance. The resulting p-values are shown in Table 13. Using a cut-off point of $\alpha = 0.05$ it does not appear that we get more than one significant coefficient in each model.

This would normally suggest a linear relationship, but we cannot confirm this conclusion mainly because we do not have sufficient data to validate this assumption (i.e. using only 6 measurements violates the "one in ten" rule suggested by Harrell et al. (1984)).

The plot of the residuals from the simple linear regression model (Figure 33) is also inconclusive and may hint towards a curvilinear relationship. It is therefore suggested that more data would be needed to make a confident estimation about the type of the relationship and the scalability of the training process and the underlying infrastructure.

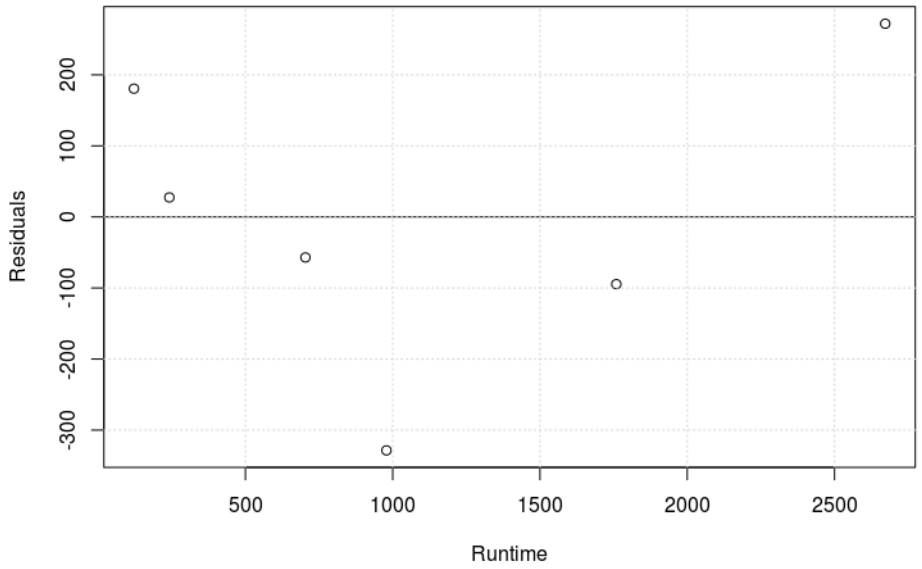


Figure 33: Runtime (s) vs. Residuals – Hybrid Spark back-end, mini-batch training.

4.2 Single-node training

In order to answer the research question and compare the distributed training process to a "non-parallelized implementation", we also ran the DML training script using the Standalone SystemML back-end. In this mode, all the operations are carried out on a single machine, inside the control program, and no Spark or MapReduce back-end is utilised.

The average training time per epoch using Standalone execution mode is shown in Table 14. Compared to the training with the Spark back-end, the results show that on a relatively small datasets the single node execution outperforms the distributed training.

The single node, however, was unable to scale beyond 20'000 examples, failing with an

Document count	**Features file**	**Runtime Spark back-end(s)**	**Runtime Standalone(s)**
10'000	rbm-10k-x.csv	121	96
20'000	rbm-20k-x.csv	241	189
40'000	rbm-40k-x.csv	703	Failed
60'000	rbm-60k-x.csv	978	Failed
80'000	rbm-80k-x.csv	1759	Failed
100'000	rbm-100k-x.csv	2672	Failed

Table 14: Average runtime per training epoch (in seconds). Comparison between parallelized Spark back-end and single node Standalone back-end

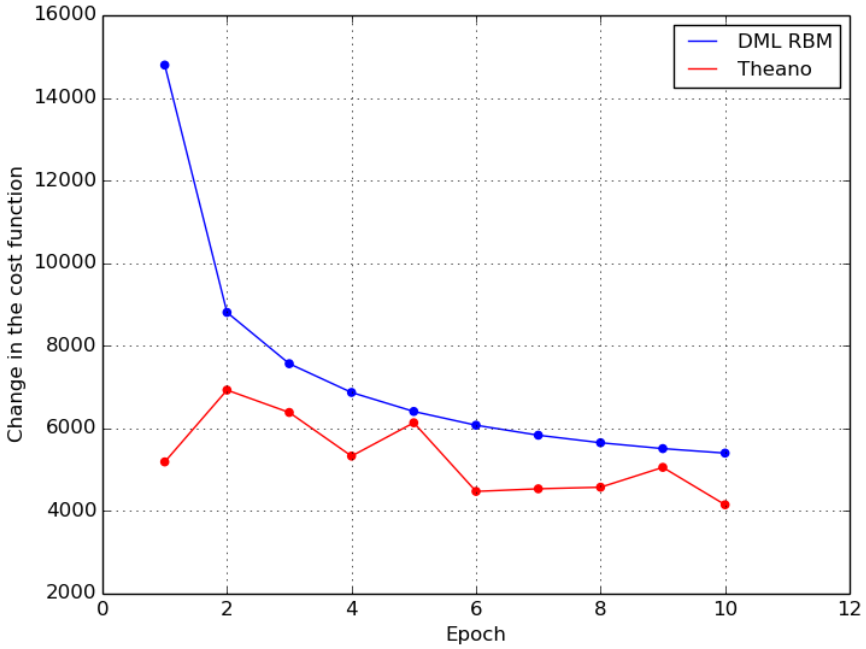


Figure 34: Changes in the cost function with continuous training – DML RBM versus DeepLearning Theano

"OutOfMemoryError" error, while trying to allocate matrices for storing the data.

Looking at the detailed execution plan, it appears that with the Standalone back-end, SystemML tries to load the entire dataset in memory, despite the fact that the training is carried out in mini-batches. This is in line with the hybrid execution concept of SystemML, where the control program is used only when the data set fits fully in memory and distributed computation is used otherwise. This behaviour is very beneficial when working with datasets that require distributed processing, and it is understandable that forcing the Standalone mode harms the scalability of the algorithm.

4.3 GPU-based RBM

In terms of performance, the GPU-based RBM was better compared to the SystemML RBM, exhibiting an average runtime per epoch of 89 seconds for the 10'000 examples subset. The CPU-based training for the same dataset produced an average epoch training time of 121 seconds (see Table 15 vs. Table 11).

In terms of scalability, however, the GPU-based implementation was no match for the distributed CPU-based training – it could not process any data set bigger than 10'000 examples, and it constantly failed with "OSError: [Errno 12] Cannot allocate memory" error. This error is a clear indication that the system was unable to fit the complete dataset in the GPUs memory.

Another interesting finding is that the decrease of the cost function with continuous training

Document count	Features file	Runtime (s)
10'000	rbm-10k-x.csv	89
20'000	rbm-20k-x.csv	Failed

Table 15: Average runtime per training epoch (in seconds), using the Theano RBM implementation

	Epochs	Training time per epoch (s)	Total time (hr)	Price per hour ($)	Total cost ($)
CPU Instance	100	121	4	0.119	0.476
GPU Instance	100	89	3	0.702	2.106

Table 16: Price comparison between CPU and GPU-based RBM training. Measured for 10'000 documents using the EC2 pricing for Amazon's EU Ireland region.

was somehow problematic for the Theano RBM. Figure 34 reveals that the cost of function of the CPU-based RBM is strictly decreasing with each consecutive training epoch. The pseudo-likelihood function (Equation 27) that the GPU-based RBM optimises, although decreasing, was also oscillating and was less stable compared to the SSE metric for the SystemML RBM.

This, combined with the lack of scalability (due to the fact the GPU implementation tries to load the entire dataset in-memory) suggests that the implementation of the Theano RBM is somewhat problematic.

Unfortunately the complete inability of the GPU-based training script to scale prevented us from conducting further tests and applying statistical analysis to determine the relationship between data and runtimes.

We can, however, look at the financial cost of training an RBM using GPU instances and compare this to the CPU-based training. We could only look at the 10'000 records dataset though, as this is the biggest number of records that we can successfully process using the GPU instance.

A breakdown of the costs is shown in Table 16. If we use a training duration of 100 as a ballpark figure (based on the tests in the functional testing phase), we see that although the training time using a GPU instance is shorter, the higher price per hour for using the instance offsets this advantage and the total processing cost for the GPU-based training is four times higher.

To be fair, this comparison includes just a single CPU instance as the processing of 10'000 records does not require parallelization. If we need to do a computation with the entire Hadoop cluster this price increases significantly – the total cost per hour for all six instances is $6 \times 0.476 =$ \$2.856, which is above the "per hour" cost for a GPU instance. Currently, however, this cluster

can process ten times more data than a GPU instance can handle, so comparing the pure instance cost is not entirely fair.

4.4 Conclusion

The results in this section indicate that our methods are appropriate and valid. We successfully demonstrated that the custom RBM implementation is capable of processing relatively large datasets, and that it undoubtedly benefits from parallelization, as the statistical analysis confirms a strong relationship between the number of participating cluster nodes and the RBM training time.

We also validated the software stack, confirming that the EC2, HDFS, MapReduce/Spark, SystemML, and a custom machine learning algorithm combination integrates and scales up as intended.

We established that the RBM algorithm is limited in the amount of data that it can process when operating on a single node. This finding is also valid for the parallelized GPU-based implementation. In the next section we discuss the results in details and we introduce potential improvements for both the parallelized and single node (CPU/GPU) versions.

5 Discussion

The work carried out as part of the research project resulted in several interesting findings. First and foremost, we were able to answer the main research question stipulated in the research proposal: "Can a CPU-based, parallelized version of the Restricted Boltzmann Machine be developed and how does it compare to the standard, non-parallelized implementation?"

Without any doubt, the answer is positive. We successfully developed a parallelized version of Restricted Boltzmann Machine that can operate in "big data" context, accessing data stored in Hadoop, and using MapReduce or Spark as a back-end execution engine.

As part of the functional testing of the implementation, we showed that the machine is capable of learning complex features, and it can participate in machine learning pipelines, substantially reducing the convergence time for the classification algorithms in the pipeline, without harming their accuracy (Table 5). We also studied the effect of changes in parameters like learning rate and batch size on the training progress of the RBM. In terms of scalability, we demonstrated that the RBM can be successfully deployed on a distributed Hadoop cluster (6 Hadoop nodes) and that it can process a relatively large dataset, containing up to 100'000 examples (2.61 GB). It appears that with the Spark back-end *the algorithm parallelizes and scales up without any observable issues*, utilising the available resource in a balanced fashion (Figure 32).

We handled the comparison of a non-parallelized versus parallelized version by restricting the number of nodes participating in the cluster, and running another series of tests using a Standalone (single node, no Spark or MapReduce) execution mode. We determined that *a single node is limited in the amount of data it can process* (Table 14). It does not scale beyond certain dataset size (20'000 documents), however, for smaller sets it outperforms the parallelized version. We can attribute the latter to the latency associated with staging and tracking distributed jobs, the node-to-node communication overhead, and the second file system layer involved in the distributed processing (HDFS is a user space file system, deployed on top of EXT4 in this particular test).

We also confirmed that adding just one extra node (i.e. using parallelization over just two machines) nearly halves the training time (Figure 31), so the performance advantage of a non-Spark training vanishes as soon as we introduce a second node to the execution environment.

Based on the above we can conclude that *a parallelized version of a Restricted Boltzmann Machine is better in both scalability and performance* – it can process more data than a non-parallelized implementation, and it performs better with the addition of more executors/processing nodes.

The inability of the single node RBM to scale beyond a certain document count, however, arises from *limitations of the execution platform SystemML* and not from the RBM algorithm.

Looking at the detailed execution plans hints that SystemML tries to load the entire dataset in-memory before commencing processing. This behaviour is expected as the control program should carry out only operations that fit the driver's memory, although in the context of mini-batch RBM training this is completely unnecessary. As we process the training set using mini-batches (each containing 100 examples to be specific), we do not have to load the entire dataset in-memory. Instead we can read and train the RBM on partitions of 100 examples and update the model weights accordingly. A better plan generation can account for this and potentially negate the scalability advantage of a distributed back-end.

If we do not force single node execution on SystemML, the platform handles the optimal plan generation without issues and inspecting the execution plans shows that as the size of the input data increases, the system increases the number of parallel operations accordingly (Table 12).

Using Spark for the back-end execution we measured the impact of increasing the data volume and the introduction of additional nodes. Our findings reveal that adding more training data is consistent with the increase of the time needed to complete a training epoch (Figure 29). Although we process the examples in batches of identical size, more input data means more mini-batches per epoch, so this is in line with our understanding of the training process.

As already mentioned, *adding more processing nodes decreases the total training time* in a consistent manner (Figure 31). We were, however, unable to reliably determine the type of relationship between input data size and total runtime. The statistical analysis (page 76) shows a very strong positive correlation (p-value = 0.000941) between the volume of training data and the time required to process it. Trying higher order polynomial regression does not lead to more significant coefficients, but the plot of residuals suggests a curvilinear relationship. As we do not have sufficient data (10 measurements for every degree of freedom, as suggested by the "one in ten" rule) to reliably determine the relationship we can not make a confident statement at this time.

Although we proved that using a parallelized RBM on top of SystemML with the Spark back-end can increase the volume of training data and decrease the training time, this is not the case when using the MapReduce back-end. Based on the results from our experiments we can conclude that *MapReduce is not suitable for training parallelized Restricted Boltzmann Machines* in the context of "big data".

There are two aspects that make MapReduce unfit for training RBMs. First of all, the limitation of expressing everything as a series of chained Map-Reduce jobs, combined with the nested loops of the algorithm (an outer loop for the epochs and an inner loop to go over each mini-batch) results in the breaking of the training into hundreds, potentially thousands, of independent MapReduce jobs. This, combined with the fact that the only mechanism of communication between the chained tasks is via the file system (i.e. the output of task n is

persisted, and task $n + 1$ loads it back into memory to use it as input) leads to unacceptably long runtimes (Figure 25).

Training an RBM on 20'000 documents takes about 20 minutes with the Spark back-end. Switching to MapReduce led to a series of jobs, which we aborted after 7.5 hours and 744 successfully completed MapReduce jobs, as there was no indication that the processing was going to complete soon (not even one epoch was completed in that time).

Modifying the RBM training algorithm to omit the use of mini-batches mitigated the number of independent MapReduce jobs and the 20'000 documents set was successfully processed, taking about 46 minutes to complete. Although capable of successfully training the RBM, this approach still led to a processing time that was over two times slower compared the Spark runtime with the same input. Another problem we uncovered is that *a non-mini-batch implementation is less capable of scaling up*. The non-mini-batch script failed with an "out of memory" error at the next stepping stone of 40'000 documents and was incapable of processing any datasets containing more than 20'000 documents.

The scalability problem arises from the fact that not using mini-batches requires that albeit partitioned, the whole feature matrix X must be loaded in memory for the training to commence. This gives an advantage to the MapReduce processing at it reduces the number of jobs required but the trade-off is that the system won't be able to process sets that do not completely fit in the cluster memory.

We verified that this non-mini-batch limitation manifests itself with the Spark back-end as well. Interestingly, the Spark-based processing failed with an "out of memory" error sooner than the MapReduce training, and it was not capable of processing even the 20'000 documents set (although it handled the 10'000 set with no issues and again topped MapReduce in terms of performance).

We observed similar behaviour while investigating the impact of the batch size on the average time per training epoch using Spark (Figure 24) – increasing the size of the mini-batches beyond a certain threshold (3'200 for a 20'000 documents collection) again leads to an "out of memory" error, as the partitioned matrices no longer fit in the local executor memory. Testing the SSE reduction with various batch sizes, however, allowed us to select an optimal size and prove that the RBM is capable of scaling up and processing much bigger sets (100'000 examples).

In terms of resource utilisation MapReduce exhibited a less stable behaviour compared to the Spark back-end. The scheduling and execution of hundreds of independent jobs takes a toll on the resource allocation, so the cluster was constantly trapped in a resource allocation/deallocation routine, resulting in a jigsaw utilization profiles (Figure 27 and Figure 26).

Based on the above findings, we can confidently state that *Spark is a better suited platform for parallelized training of Restricted Boltzmann Machines*, as it provides unmatched scalability, better performance, and more stable resource utilization compared to MapReduce.

It is also worth noting, that as part of the MapReduce mini-batch/non-mini-batch testing we *independently verified the suggestion of Hinton (2012)* that mini-batch training is a more efficient approach for training RBMs (see Section 4.1.2). Although Hinton talks about the advantages of using mini-batches "on GPU boards or in Matlab", we confirm that this is also advantageous in a Hadoop environment and especially when using Spark as a distributed processing framework.

As an optional objective of the research, we also wanted to look into how an x86/Hadoop based RBM compares to a parallelized GPU-based RBM training. It has been successfully demonstrated that a parallelized GPU-based RBM implementation can effectively parallelize the training process and substantially reduce the training time Raina et al. (2009).

Ideally, we would have tested the same DML RBM script using SystemML but with a GPU back-end. Unfortunately, the GPU support in SystemML is still in experimental phase and we encountered an incompatibility between the JCuda wrapper and the operating system of the GPU-enabled machine (Amazon Linux AMI). As highlighted in Section 3.3, this incompatibility prevented us from using SystemML and we had to rely on the fall back plan, namely to use a third party RBM implementation that is GPU capable – the Theano Restricted Boltzmann Machine from DeepLearning 0.1 (see *DeepLearning 0.1 Documentation – Restricted Boltzmann Machines* 2016).

During our experiments with the GPU-based RBM we identified certain *areas for potential improvement of the Theano GPU-based RBM*. Firstly, the Theano RBM implementation performed worse in the functional testing phase. The features it learns contain more noise compared to the features learnt by the DML RBM (Figure 15 and Figure 17). While training the RBMs on the Gutenberg data, we also noticed that the decrease in the cost function is less clearly expressed and less stable in the Theano RBM in comparison to our custom RBM implementation (Figure 34).

The Theano RBM implementation also assumes that the input features matrix is normalized. The script does not check this assumption and if the data is not normalized it silently keeps working but no features are actually being learnt (Figure 18). To remedy this we had to pre-process and normalize the data fed to the RBM. This additional processing time is not accounted for in the average epoch runtimes for the RBM, but it should not shift the results significantly as the normalization of the input is a fairly simple and quick operation, especially compared to the RBM learning times. Modifying the Theano RBM to check the input ranges and normalize the data automatically should be a fairly simple task and it will remove the need for additional pre-processing.

To compare the GPU and the fully CPU-based RBM implementations we ran the two models on the same datasets, using identical parameters (e.g. mini-batch size, learning rate etc.) We then looked into the average training times per epoch that the two models produce. As expected, the *GPU-based training produced shorter runtimes than the DML RBM*. What

was entirely unexpected was that *the performance gain from the GPU was approximately 36%* – nowhere near the up to 72.6 times speed-up reported by Raina et al. (2009). This finding is perplexing, especially given the fact the the hardware used by Raina et al. was far less capable than the GPU board used in our experiment[17].

Investigating this finding uncovered a fundamental difference between the Raina et. al's RBM implementation and the Theano RBM implementation. The former have developed an implementation with the GPU architecture in mind – their training algorithm does not copy the entire feature matrix in the GPU's memory, instead it iteratively transfer sections of the feature matrix to the GPU, then uses mini-batches from this transfer to get the direction of the gradient, and then uses GPU-specific calculation based on blocks and threads to compute the weight updates.

The Theano RBM, on the other hand, starts by copying the entire feature matrix into the shared GPU memory. The RBM then implements a standard RBM training algorithm without any specific optimisation with the GPU architecture in mind. It appears that the expectation is that once all data and operations are offloaded to the GPU, the operations speed-up alone should be sufficient to produce good runtimes. As it turns out, that is not what actually happens. The initial attempt to copy the entire dataset to the shared memory has negative impact on scalability, restricting the Theano RBM to processing only datasets that completely fit in the GPU's memory. It also appears that relying on fast operations alone does not produce the expected reduction of training time and the RBM exhibits only a marginal performance improvement over a non-GPU-based training.

This leads us to a key finding in our research – the simple act of *offloading data and operations to a GPU board does not bring substantial advantage, unless the algorithm itself is modified to leverage specific features of the GPU architecture.* At this point we can speculate that this finding probably generalizes to all GPU-assisted machine learning algorithms. If the algorithm is not specifically designed to exploit the benefits of the GPU architecture, it will be unlikely that an order of magnitude change in the runtime will be observed.

We can draft a similar conclusion about the parallelization provided by SystemML. We did not introduce any specific optimization in the RBM training algorithm to specifically exploit the parallelized Hadoop infrastructure. Instead, we did a fairly standard implementation and relying on mini-batches and the optimization of distributed matrix operations provided by SystemML. This turned out to be relatively successful, given that our custom RBM was capable of processing data sets that were an order of magnitude larger than what the GPU-based RBM managed to handle. This approach also showed consistent speed-ups when adding more processing nodes. We can, however, assume with a relatively high confidence that *if we were to tailor the algorithm explicitly for the underlying Hadoop environment, we should be able to get even better results.*

[17]Raina et al. used an Nvidia GeForce GTX 280 graphics card with 1 GB of memory and just 240 CUDA cores

We can also speculate that the inability of the Theano-based RBM to scale up is mostly due to the way the code is implemented, and it is not a constraint of the GPU platform alone. It should be fairly easy to modify the RBM code to copy subsets of the data to the shared memory (e.g. one mini-batch a time) instead of trying to load the entire data set in the on-board memory. Such modified version can give us a better understanding of where the performance advantage is, as it will be capable of processing the entire Gutenberg set, thus giving us concrete runtime metrics for comparison with the clustered RBM.

Looking at the price-performance ratio when using multi-node versus GPU-based training turned out to be somewhat problematic. If we limit the investigation only to the 10'000 examples dataset (that is as much as the GPU-based implementation was capable of processing) we will have to compare a single GPU instance to a single Hadoop node. This is because such a small dataset does not trigger a distributed computation as SystemML is smart enough to run everything in the driver alone.

In that case, although the GPU-based training takes less time, *the higher price per hour for a GPU instance negates the performance advantage* and it is still about four times cheaper to use a CPU instance for the RBM training (Table 16).

When a distributed processing is required, the number of instances in our cluster increases to six, and this also impacts the "per instance hour" cost. If we were to process a 100'000 examples set, the average epoch time using the six nodes would be about 2'672 seconds (Table 14). If we assume that a properly implemented GPU-based training (i.e. one that scales up by using the input data partitioning approach introduced by Raina et al. (2009)) scales up linearly, then we would expect an average runtime for the GPU-based training in the region of $10 \times 89 = 890$ seconds.

Using the EC2 pricing we can compute that the "per hour" cost for the Hadoop cluster would be $\$0.119 \times 6 = \$0.714$ versus \$0.702 for a single GPU instance. The faster processing of the GPU-based training however will result in a total runtime of about 25 hours versus 75 hours needed for the Hadoop cluster, thus making the multi-node CPU-based training three times more expensive compared to the GPU-assisted processing. We cannot, however, consider this a confident finding as the assumption that an optimal GPU-based implementation will scale up linearly is a very bold one.

It is also interesting to consider the price/performance implications of a GPU-assisted Spark cluster. As previously mentioned, SystemML is introducing an experimental GPU support (*SystemML: Initial prototype for GPU backend* 2016), so if the Hadoop worker nodes are equipped with GPU cards, the Spark back-end should be able to utilize them for back-end computations. If we assume that introducing GPU cards to the Hadoop nodes will bring their EC2 pricing to a level similar to the g2.2xlarge instance, then the total "per hour" cost for the cluster will be in the region of \$4.212. Even if we assume an ideal scenario where a perfect parallelization

is achieved and the processing time using 6 GPU-enabled nodes is 6 times shorter than the processing time of single instance, this would still be insufficient to offset this extra cost. The total cost of training the RBM against 100'000 examples for 100 epochs would approximately amount to 5 hours $\times$ \$4.212 = \$21.06, where a single GPU node could theoretically process the same amount of data for 25 hours $\times$ \$0.702 = \$17.55. Of course, this calculation is highly speculative. It is therefore suggested, that *a more in-depth research in the price-performance ratio is required* to make a confident estimation on the break even point.

6 Evaluation, Reflections, and Conclusions

In this section we evaluate the project work, highlight the project achievements, suggest areas for future work, and discuss personal reflections.

The experimental results and the follow-up discussion enabled us to successfully address the research question and we consider this as a key achievement of the project. We also show that the work has potential for development into a research publication as the research informed interesting findings, and the work has already been accepted by the two biggest technical conferences on big data in Europe (Manchev 2016, 2017).

Based on our findings it appears that we have developed the most scalable RBM implementation to date. A further evidence for the interest in this topic and the quality of the work is the decision of the Apache SystemML project to include our RBM implementation as one of the algorithms in the SystemML platform (see Section 6.2).

6.1 Choice and fulfilment of objectives

The choice of objectives for this project was ambitious. We had to develop a software product for a distributed platform, where no reference implementation exists, and we had to prove that this implementation is functionally correct. We also had to select, prepare, and test the product with a dataset of substantial complexity, as Restricted Boltzmann Machines are often used for dimensionality reduction on large datasets with high number of features. We had to set up and execute a series of experiments to determine the scalability and performance implications that arise from parallelizing the execution on multiple physical nodes, which on its own also required the provisioning and preparation of a distributed execution environment. Optionally, we decided to look into how GPU-assisted training compares to multi-node parallelization and repeat the scalability/performance experiments using GPU-based RBM training.

The literature review was thorough and greatly facilitated the development of the parallelized RBM. We reviewed 31 articles to inform the project proposal, and this number nearly doubled in the course of various project activities and the report creation.

The project plan was adequate and was followed without deviations. The only exception was the unforeseen incompatibility between the operating system for the GPU-instances in the Amazon cloud and specific libraries needed by SystemML (see Section 3.3). Nevertheless, we successfully processed the in-scope dataset with a GPU-based RBM and managed to draw interesting conclusions on the scalability and performance introduced by training RBMs using graphic processors.

6.2 Achievements of the project

We can report, with great satisfaction, that all core and optional objectives established in the proposal (Section 1.2) were successfully completed. This allowed us to answer the research question with a high level of confidence and also present additional findings based on the experiments that were conducted as part of the project work.

This project contributed to the body of knowledge of Artificial Neural Networks – a dynamic and fast growing area that attracts substantial attention not only in academia but also from the industry, as it has wide application in practical data processing and knowledge abstraction.

It appears that the outcomes of this project draw a fair share of attention from the big data industry. A presentation based on the project work was submitted and selected for publication by the Big Data Spain conference – the second largest technical big data conference in Europe. The session titled "Multi-node Restricted Boltzmann Machines for Big Data" (see Manchev 2016) was presented on 18th November 2016 and was well accepted and a fair share of positive feedback was received from participants.

This session was also submitted and accepted for presentation at the biggest European big data technical conference – "Strata + Hadoop World London 2016" (see Manchev 2017), and it will be delivered in May 2017.

The project also delivered the first scalable, distributed RBM training implementation that can work on top of Hadoop (using the MapReduce framework or Spark for executing distributed matrix operations). It appears, that combined with the "big data" capabilities of Hadoop and the optimal execution plan selection provided by SystemML, we have developed the most scalable Restricted Boltzmann Machine implementation currently available. This custom DML implementation was also suggested to and accepted, after a thorough code review, by the Apache SystemML team. The code is now part of the official Apache SystemML project and can be found in the Apache SystemML GitHub repository – `https://github.com/apache/incubator-systemml`.

6.3 Ethical and Legal Implications

As highlighted in the project proposal, this project did not engage any participants other than the author so no social, emotional, or physical concerns were encountered in line of the project work.

There is a potential consideration surrounding the fact that the training of an RBM model might require access to sensitive data. For example, healthcare providers tend to discard 90 percent of the data they produce (McKinsey Global Institute 2011), as it is unstructured and difficult to process. RBMs could help in this situation by imposing structure on the data (e.g. used as feature extractors), but sensitive data should be provided for the training phase of the

model, which can also be used to identify individuals. We make a suggestion (see Section 6.4) on further investigation in that area.

All the software used in the project proposal is released under copy-left licences (Spark and SystemML are licensed under Apache 2.0, LaTeXis licensed under LPPL, Linux is licensed under GPLv2, Theano has a proprietary license but it is distributed for free and permits modifications of the source code etc.)

After donating the purposely developed parallelized RBM training code to the Apache SystemML project, we also granted rights for the source code to be distributed and available under Apache Licence 2.0.

6.4 Future work

Although all project objectives were successfully met and the research question answered in full, we identified several areas that we believe deserve further investigation.

- First and foremost, we realize that the CPU-based multi-node versus GPU-based training comparison was not entirely fair due to the inability of the Theano RBM implementation to scale up to the volumes that we could handle in Hadoop. We can suggest two potential ways of rectifying this situation:
  - Re-run the GPU-based experiments once CUDA 0.8.0 becomes available in the Amazon Linux AMI – this would be the fairest approach as it guarantees that we are using the same algorithm on the same underlying hardware
  - As the RBM training algorithm is fully deterministic, we could re-implement it like for like using the Theano framework – this is less desirable, as there are numerous optimisations that SystemML does before running the code. Not all of them are transparent and although we can identify what happens under the covers by looking at the generated execution plans, there is still risk that we may miss something and end up with a not fully identical Theano implementation. However, this will be still better than using the Theano RBM from DeepLearning, whose implementation is much more different compared to the DML RBM
- In the experiments involving the Spark back-end we settled on using one worker process (executor) per physical node. The Spark framework, however, allows multiple executors per physical node. We did not look into this aspect and would be interesting to see how within the node parallelization could impact scalability and performance. Would we be able to process faster and more data if we use two executors on the same worker nodes instead of one? This is a question that can easily be investigated and build upon the work conducted as part of this project

- We did not plan and did not have extra time to actually test the impact of using the RBMs in a pipeline for the Gutenberg project. We know that RBMs have successfully been used for classification (Larochelle & Bengio 2008) and semantic document representation (Xing et al. 2005). It would be interesting to see if our RBM implementation can bring any benefits compared to running classification algorithms on the raw Gutenberg data directly

- One of our key findings was related to the fact that simply offloading the operations to the GPU does not lead to the speedup that can be achieved by carefully engineering the training algorithm to exploit the GPU architecture. This finding is somehow applicable to the DML RBM, as the parallelization is transparently handled by SytemML. SystemML, however, provides other mechanisms that give finer control over the execution. For example, there is *parfor* (parallel *for*) operator, which explicitly performs a concurrent loop on multiple physical nodes. We can easily envisage a re-engineering of the DML RBM script to better exploit the benefits of the distributed architecture, and we can assume with a good confidence that an RBM modified in this way will provide better scalability and performance. This are of future work seems quite attractive, especially given the fact the theoretical foundations are already available – Su & Chen (2015) introduce a distributed Stochastic Gradient Descent based on model averaging – they partition the data and train individual models, averaging the parameters at the end of each epoch. This limits the network communication only to exchange of parameters at the end of each epoch.

- The privacy issues we mentioned in regard to the legal implications of RBM training (Section 6.3) are also worth looking into. Li (2013) suggests a privacy-preserving method for training Restricted Boltzmann Machines and it might be interesting to see if the large-scale parallelized training has any negative impact on the method outlined. Another way of looking into this is to investigate what modifications can be added to the DML RBM, so that it uses the privacy-preserving method for learning.

6.5 Personal reflections

From a personal perspective, I think that this project was challenging. There were numerous aspects that pushed me outside of my comfort zone, as I had no previous experience with SystemML, Theano, and the EC2 platform. For the custom RBM implementation I had to learn a new programming language – DML, for which the online resources were very scarce as SystemML was open sourced and introduced as an Apache project just one year ago (November 2015).

The project, however, was also very rewarding. It increased my theoretical and practical knowledge in Neural Networks and also enabled me to do something that also has a practical application. The acceptance of my project-based talks at the two largest technical conferences

on Big Data validates the relevance of the topic, and it also motivates me to continue my learning and research work in the area of Neural Networks.

The careful planing and the literature reviewed for the project proposal definitely paid off, and despite the tight schedule I never had concerns that the project might be at risk.

It is difficult to say what could have been done differently if we were to start the project again. Given the newly acquired knowledge and knowing the constraints of the Theano RBM implementation, we would have definitely planned for more time on the GPU-assisted RBM training topic. We would have tried to circumvent the Amazon Linux AMI limitations and find a way to run the DML RBM with GPU support in order to get a fairer comparison on the scalability and performance.

There were also some scripts and features, which I developed but never used. For example, the pre-processing scripts for the Gutenberg project can generate two type of RBM inputs – feature vectors based solely on the presence of a word and feature vectors based on the term frequency. As I did not include the Gutenberg-trained RBMs in a pipeline, this functionality never came to use. I do not, however, consider this an effort in vain, as I gained additional knowledge and experience in feature vector representation.

On overall, working on the project was a very pleasant experience as the topic and the interesting technologies involved kept me highly motivated. I feel personal satisfaction, as I believe the main research question was answered, other interesting findings were uncovered, the RBM implementation was accepted by a large machine learning project (SystemML), and a good list of future work topics were identified.

7 Appendix A – Project Proposal

Parallelized, multi-CPU, multi-node training for Restricted Boltzmann Machines

Nikolay Manchev,
School of Mathematics, Computer Science & Engineering,
City University London

April 5, 2016

1 Introduction

The last decade has witnessed an unprecedented growth in the amount of data that are being generated on a daily basis. Studies have been conducted to estimate the amounts of data involved and they agree on the prognosis that data generation is set to grow exponentially for the foreseeable future[1].

At the same time, most of this data is in unstructured or semi-structured format. Storing and operating with unstructured data is a difficult tasks, prompting owners to discard a substantial amount of the data they generate. For instance, it appears that healthcare providers discard 90 percent of the data they produce[2].

Finding effective ways for imposing structure over unstructured data is key to optimising the storage requirements and for enabling owners to gain insights from their data. Artificial Neural Networks (ANN) are well suited for processing and finding non-linear patterns in unstructured data, and they have indeed been used to successfully transform unstructured content into structured data[3].

A Restricted Boltzmann Machine (RBM) is a stochastic artificial neural network with wide range of applications and can be used as generative model for dimensionality reduction[4], collaborative filtering[5], classification[6], and extraction of semantic document representation[7].

Restricted Boltzmann Machines are also used as building blocks for the multi-layer learning architecture known as Deep Belief Network (DBN)[8].

Unfortunately, parameter learning for RBMs is computationally intensive and for large models it often means operating with millions and billions of parameters. In the context of big data training an RBM can take a substantial amount of time. Raina et al.[9] point out that parameter learning using conventional, non-parallelized implementation on a single CPU can take weeks. RBMs can be parallelized using graphical processing units (GPUs), but GPUs introduce certain drawbacks, which prevent them for achieving the scalability required for true "big data" processing.

The question this project aims to answer is: **Can a CPU-based, parallelized version of the Restricted Boltzmann Machine be developed and how does it compare to the standard, non-parallelized implementation?**

The product of the work will be a parallelized RBM implementation that can run on a clustered computing framework (Apache Spark), combined with a thorough analysis of its performance using a single CPU, using multiple CPUs on a single machine, and using multiple CPUs on multiple clustered machines. Optionally, we would also look at how it compares to existing GPU-based implementations.

The project will contribute to the body of knowledge of artificial neural networks, especially in the context of "big data". It will provide insight on how suitable for parallelization the RBM

model is, and potentially demonstrate improvements that can help with one of the biggest limitations of Restricted Boltzmann Machines – the considerable amount of time required in their training phase.

2 Critical Context

2.1 Restricted Boltzmann Machines for Big Data

A Boltzmann Machine is a stochastic artificial neural network introduced by Hinton and Sejnowski[10]. It is a fully connected, stochastic and generative network that is capable of learning and reproducing a probability distribution over a given data set.

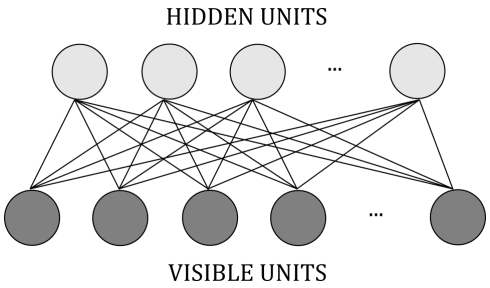


Figure 1: Restricted Boltzmann Machine

Restricted Boltzmann Machines (RBM) have two layers of processing elements – a visible layer and a hidden layer. There are connections between each node in opposite layers, but there are no connections between elements within the same layer. In other words, the structure of the RBM forms a bipartite graph. The imposed restriction allows more efficient training, using contrastive divergence, as demonstrated by Hinton in [11].

2.2 Parallelized RBMs on GPU

Various attempts have been made to train Restricted Boltzmann Machines in parallelized fashion. It appears that the majority of research in this area is focused on computing with graphics processors.

Raina et al.[9] developed the main principles of training deep belief networks on a Graphical Process Unit (GPU). They have successfully demonstrated that using parallelized implementation on GPU can reduce the training time for a four layer deep belief network with 100 million parameters from several weeks to a day.

Raina et al. demonstrate that a DBN, where each layer consists of an RBN, can be trained using contrastive divergence or convex optimisation. Data is transferred from RAM to the GPU's shared memory in small batches and updates are computed using the GPU's two-level parallelism. This process is repeated until a certain convergence criterion is satisfied.

Studies have successfully demonstrated that parallelized GPU implementation can outperform a CPU based RBM training by a factor of 2[12, 13].

2.3 Drawbacks of parallelizing using GPU

Zhu et al.[14] admit that the GPU architecture also introduces difficulties:

- GPUs impose a limit on the amount of memory available for the computation, thus limiting the size of the RBM that can be trained. A 4GB of shared memory, which is a considerable amount for a GPU architecture, can only fit up to 1 billion parameters – a modest amount in the context of deep networks;

- Stacking multiple GPUs together seems to be inefficient due to the communication induced overhead and the increased economic costs

To work around the first limitation Zhu et al. introduce an algorithm that uses memory slicing, sequentially feeding batches of data to the shared memory, thus increasing the size of the model that can be trained. This approach however is susceptible to another issue identified by Raina et al.[9] – a bottleneck introduced by the latency of copying data from RAM to the GPU's shared memory.

For example, it appears that only 0.5% of the time required for a parallelized matrix multiplication goes to the actual computation. The remaining 99.5% is spent on moving data between RAM and shared GPU memory. This is also recognized by Zhu et al.[14] who acknowledge that memory transfer is so time consuming that it can completely negate the performance gain of GPU computing.

These limitations have also been identified by Ly et al.[15] who observed a "significant overhead" gone into memory transfers and threads synchronization. This experience leads them to admit that the "implementation may not be capable of being efficiently scaled" – a problem that will essentially prevent the application of GPU training for large scale problems.

It appears that in order to avoid the aforementioned limitations most of the research around parallel RBM training is focused on using a single GPU and measuring its performance against a single CPU[9, 14, 15, 16].

The literature review revealed only a handful of multi-GPU based training attempts. Each of those approaches the problem differently and not all of them are applicable to RBMs.

Su et al.[17] introduce a distributed Stochastic Gradient Descent based on model averaging – they partition the data and train individual models, averaging the parameters at the end of each epoch. This limits the network communication only to exchange of parameters at the end of each epoch.

Coates et al.[18] present a model where each GPU operates on an individual partition of the neural network, however this approach is suitable for convolutional neural networks and not beneficial in fully connected networks.

The pipeline training approach suggested by Chen et al.[19] has been show to achieve a 3.3x speedup, however in this architecture the number of GPUs is directly tied to the number of layers in the neural network.

The work carried out and the issues identified in the area of multi-GPU RBM training prompt the question if RBMs can be effectively trained in a multi-CPU environment and how does this process compare to existing parallelized RBM training approaches.

2.4 Spark as a general in-memory cluster computing engine

The development of a messaging and coordination system for clustered computing from scratch is an enormous task and it is way beyond the scope of the proposed research. In order to focus on the development and performance analysis of the multi-CPU, multi-node, parallelized RBM we will employ an existing clustered computing framework that can handle the convergence of the x86 worker nodes for us.

In 2012 AMPLabs at UC Berkeley introduced the Resilient Distributed Dataset (RDD) – a distributed memory abstraction and execution engine for clustered computing called Spark[20].

Spark and the RDD abstraction are optimized for in-memory iterative algorithm processing. This platform can combine the computing capacity of multiple x86 nodes and transparently partition and process datasets across a group of clustered worker nodes (Figure 2). Studies have shown that by pinning data partitions in-memory Spark is about 2.5x to 5x faster for processing big datasets compared to the traditional MapReduce framework[21].

Apache Spark's machine learning library (MLlib) utilizes the Spark engine and provides a wide range of machine learning algorithms for regression, classification, dimensionality reduction

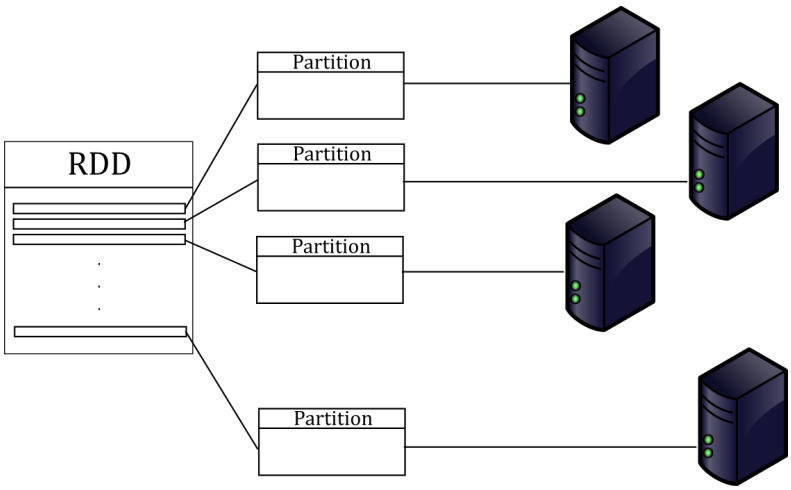


Figure 2: RDD memory partitioning

etc[22].

There is also the Apache SystemML project, which provides declarative machine learning capabilities using Spark or MapReduce as a back-end engine[23]. SystemML includes a set of out of the box algorithms for descriptive statistics, classification, clustering, matrix factorization and other.

The wide adoption and clustering functionality of Spark makes it a logical platform choice for running multi-CPU, multi-node, parallelized Restricted Boltzmann Machine against big datasets. Unfortunately, neither MLlib nor SystemML have an out of the box implementation of a Restricted Boltzmann Machine. In fact, there is an issue (SPARK-4251) registered in the Spark issues tracking system that asks for an implementation of Restricted Boltzmann Machines for MLlib[24].

3 Approaches

3.1 Literature Search & Review

We will initiate the project by conducting a comprehensive literature search and review in order to establish the learning algorithm (e.g. k-step contrastive divergence, parallel tempering etc.) and evaluate any impact or modifications required for successful parallelization. To conduct the search and review we will employ various sources (e.g. CITY Library, Google Scholar) as suggested in [25].

We will also look at a possible RBM on GPU implementation that we could use in the training phase. As a mitigation strategy, we will also try to find GPU-based performance results on a freely available dataset, which we can use to carry out the CPU vs. GPU comparison, without having to set up and do the GPU training as part of this project.

3.2 Data selection and wrangling

Based on the literature review we will identify a suitable dataset to use in the training and analysis phases. The dataset should fulfil two key criteria – a) it should be suitable for an RBM learning task and b) it should be large enough to put the experiment in a "big data" context.

To compose a list of possible datasets we will first look at what has been used in similar research work – for example [6, 26] use the MNIST dataset (70'000 images of handwritten digits), while Raina et al.[9] obtain input samples by randomly extracting image blocks from a

large dataset of natural images. We will also investigate other possible open data repositories like data.gov.uk, Kaggle, AWS Public Data Sets, UCI Machine Learning Repository and others.

Once the dataset has been selected we will perform exploratory data analysis using the R programming language. We will look for and filter out any invalid data, perform feature selection, apply transformation and normalisation (if needed), split the dataset into training and testing sets.

Tests will be performed with the modified dataset to establish a performance baseline expected from an RBM model on this dataset. We will train an established RBM implementation (e.g. MATLAB's RBM Toolbox, DL4J) using the dataset and we will record a series of metrics that will provide insight into the model performance – accuracy, confusion matrices (for classification), changes in the training/test error over time and others. We will experiment with different stopping criteria like rate of change in the error function, number of epochs etc.

3.3 Toolset selection and RBM development

The clustered computing engine of choice for implementing the parallelized Restricted Boltzmann Machine will be Apache Spark.

Spark provides APIs for Python, Java, and Scala. The SystemML runtime environment also provides R-like syntax for developing algorithms for the Spark back-end. We will investigate if there are any performance or parallelization related implications of selecting one API against another, and we will select the programming language accordingly.

The development of the RBM code will be performed and tested on a standalone Spark installation. A set of functional tests will be established to check the network performance against a well known, non-Spark RBM implementations (the baseline). The purpose of the functional check is to confirm that the Spark implementation is performing to a satisfactory level in terms of accuracy – runtimes will not be compared with the baseline as they will be meaningless in this context due to the latency associated with scheduling and running clustered jobs.

After the parallelized RBM is confirmed to pass the functional requirements it will be deployed on a multi-node Spark cluster.

The deployment infrastructure for the Spark cluster will be cloud based. There are numerous providers that offer on-demand Spark clusters with configurable number of worker nodes and number of cores. Cloud-based Spark clusters are available from Amazon, Databricks, Microsoft (Azure), IBM (Bluemix) and others.

A series of test runs will be carried out to establish the most optimal network topology. We will follow the methodology suggest by Haykin[27] to determine the optimal network parameters by successively varying the number of nodes per layer and other network parameters like learning rate and momentum.

3.4 Results and evaluation

To measure the effect of parallelization we will perform a series of tests, using the parallelized RBM and the dataset. We will test the network performance on a single-core and single CPU (there are mechanisms in Spark to limit the engine to a certain number of cores).

We will then test the same dataset and RBM configuration on a single node, configured to use multiple CPUs.

The final step will involve testing the RBM on a multi-node Spark cluster, where each host has multiple CPUs. To allow the cluster to take advantage of data locality we will place the dataset in a distributed file system (HDFS), spanning all Spark cluster nodes.

As an optional element of the research we would also like to compare the RBM running on a multi-node cluster to a parallelized GPU based implementation. Ideally we would arrange access to a cloud based system with GPU capabilities (Amazon provides EC2 instances with

1'535 CUDA cores NVIDIA GPUs). We will upload the dataset to the GPU cloud instance and use an existing RBM on CUDA implementation, like the one provided by the Deep Learning Group at University of Bonn[28].

We will measure the accuracy from the test runs using k-fold cross validation. We will also record metrics like runtime, memory use and others, that we will compare in order to evaluate the impact caused by the cluster topology changes (from single core to multi-core, multi-node).

After the data from the RBM training is collected we will try to quantify the impact of changing the cores/CPUs – the collected metrics will be analysed to establish if a statistically significant effect exists.

We will also compare the best performing parallelized RBM to the one bound to a single core and investigate if the parallelization provides any benefits in terms of processing speed. Optionally, we will compare the best performing parallelized version to the RBM running on GPU to examine how they compare to each other.

The metrics processing and analysis will be performed using R.

3.5 Final report

The compilation of the final report will commence in parallel with the literature search and review and it will be an on-going activity for the duration of the project.

The report will be written using LaTeX. All plots based on statistical data will be created using R (Base Plotting System or ggplot2). For drawing schematics and block diagrams we will utilize the Inkscape graphics editor.

4 Work Plan

This section outlines the work plan of the proposal as shown in Figure 3 and a Gantt chart is displayed in Figure 4.

We have identified four milestones for the project. The first milestone is after the literature view has been completed, the dataset selected, and the development platform (cloud instances) established. The second milestone is reached after we have a working implementation of a parallelized RBM with optimal network topology (number of nodes etc.) The next milestone marks the end of the model training, after we have enough data to analyse the performance and the impact of parallelization. The final milestone is reached on completion of the report.

The Gantt chart also includes 4 weeks of extra time for contingency, to reflect the mitigation strategy outlined in the Risk Register (Table 1).

5 Risks

The risk assessment and management is based on the strategy outlined by Dawson[29].

The key risks associated with the research are outlined in Table 1. We have listed a combination of external and internal risks and the corresponding alleviation strategies.

We intend to keep the risk register up to date by adding, removing, and re-evaluating individual risks as the project progresses.

6 Ethical, Legal and Professional Issues

The majority of this project involves technical work and statistical evaluation of computer models carried out by the author. No other participants will be engaged and we foresee no social, emotional, or physical discomfort being caused as a result of this work.

Certain concerns may be raised due to the requirements of available data for the training process of Restricted Boltzmann Machines. For example, an RBM-based classifier used by a

Description	Likelihood	Consequence	Impact	Mitigation
Failing to identify suitable dataset for testing the RBM network	1	5	5	Use the MNIST dataset as outlined by [6, 26], which is freely available in the MINST database of handwritten digits.
Unable to set up and execute GPU tests	2	2	4	Use results from RBN on GPU testing with the MNIST dataset reported by others(e.g. [30]). Test the multi-node, CPU-based RBM against the MNIST dataset and compare the results.
Unable to implement multi-node, CPU-based RBM	2	4	8	Accept and analyse what prevents the distributed implementation. Document the findings in the final report. Proceed with investigating the parallelization using single-node, multi-CPU approach.
Unable to provision cloud based Spark instances on Amazon cloud	1	3	3	Use another Spark instances cloud provider like IBM (Bluemix) or Microsoft (Azure).
Unable to provision GPU instance on Amazon	1	3	3	Use another GPU instances cloud provider like IBM (Bluemix), Rescale, or Microsoft (Azure).
Insufficient understanding of the RBM model	2	5	10	Plan for extra reading time (1 week). Have a shortlist of reading resources at hand [11, 12, 26, 27].
Insufficient understanding of the Spark platform	1	5	5	Plan for extra reading time (1 week). Books on Spark programming already procured (Wampler & Payne's "Programming Scala", Ryza et al. – "Advanced Analytics with Spark").
Loss of data (results, report draft)	1	5	5	Establish a cloud backup strategy from day one (GitHub for code, Box.com for final report and other documents).
Milestone delayed	2	4	8	Plan for 1 week of extra time.
Programming difficulties with the RBM implementation	2	5	10	1) Plan for 1 week of extra time. 2) Fall back to an existing CPU based implementation and focus on single-node, multi-CPU.
Selected data set not in appropriate format.	2	4	8	Accept. Restructure the data set in the "EDA and Data Wrangling" phase.
Researcher bias.	2	5	10	Base any conclusions only on collected metrics (accuracy, memory use) and statistical analysis.
Inconclusive results.	1	5	5	Accept, as long as the analysis is solid and backed up by sufficient evidence.

Table 1: Proposal Risk Register

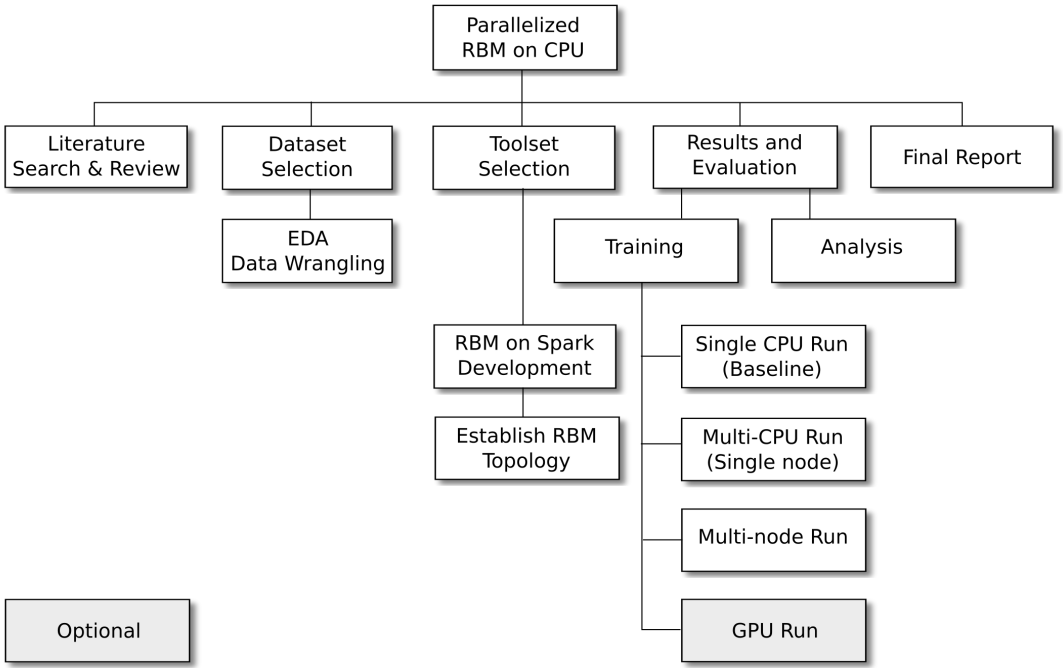


Figure 3: Work breakdown

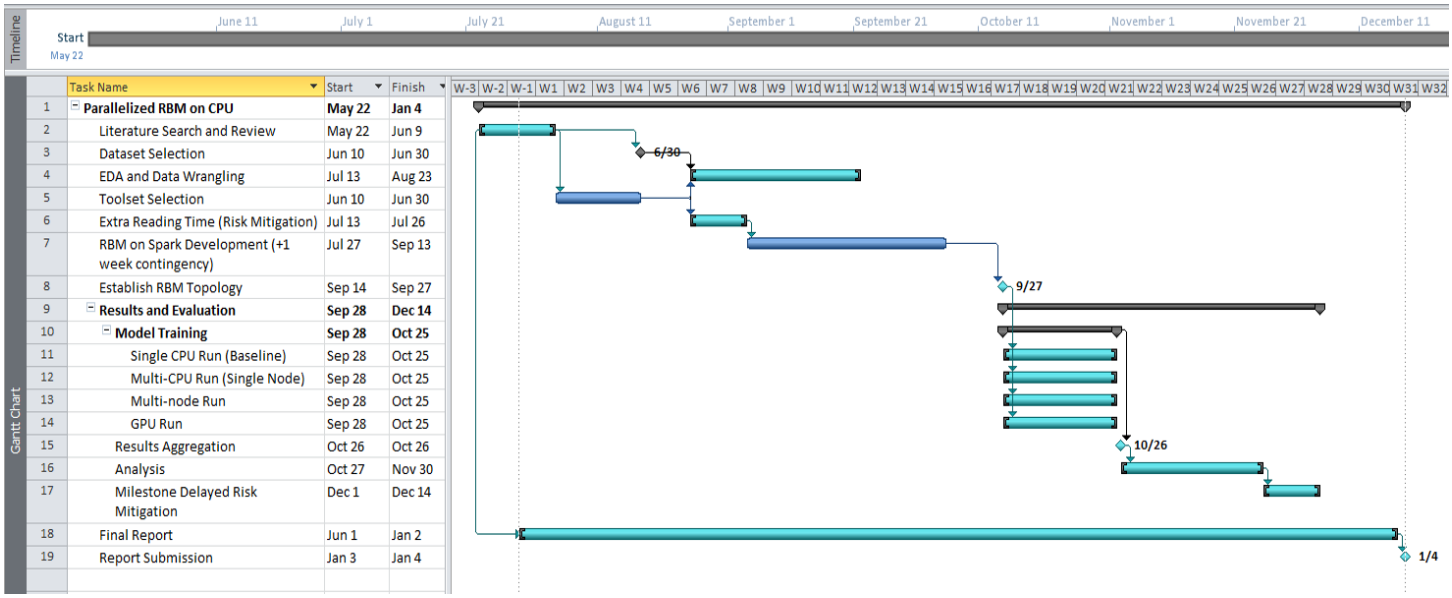


Figure 4: Gantt Chart

medical institution will require access to potentially sensitive and private patient information. This issue has been studied in details by Li[31] and resolved by a suggested privacy-preserving method for training Restricted Boltzmann Machines. However, all datasets considered in this research will come from public repositories and working and storing private or sensitive data is not anticipated.

All software that is required for conducting the research is released under copy-left licences (Spark is licensed under Apache 2.0, LaTeXis licensed under LPPL, R is licensed under GPLv3 etc.) Based on that we believe that there is no risk of breaching any copyright laws.

A Research Ethics check-list can be found at the end of the proposal.

References

[1] Kitchin, Rob, *The Data Revolution: Big Data, Open Data, Data Infrastructures & Their Consequences* 2014, SAGE Publications Ltd, ISBN 9781446287484, pp. 67–80

[2] McKinsey Global Institute, *Big Data: The Next Frontier for Innovation, Competition & Productivity*, 2011, McKinsey & Company, Inc., 485 Madison Avenue, New York NY 10022, USA

[3] Isakki, P.; Rajagopalan, S.P., *Mining Unstructured Data using Artificial Neural Network and Fuzzy Inference Systems Model for Customer Relationship Management*, 2001, IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 4

[4] Hinton, G.E.; Salakhutdinov R. R., *Reducing the dimensionality of data with neural networks*, Science, vol. 313, issue 5786, pp. 504–507

[5] Salakhutdinov, R. R.; Mnih, A.; Hinton, G. E., *Restricted Boltzmann machines for collaborative filtering*, Proceedings of the International Conf. on Machine Learning, 2007, vol. 24, pp. 791–798

[6] Larochelle, H.; Bengio, Y, *Classification using discriminative restricted Boltzmann machines*, Proceedings of the 25th international conference on Machine learning - ICML, 2008, pp. 536

[7] Xing, E. P.; Yan, R.; Hauptmann, A. G., *Mining associated text and images with dual-wing harmoniums*, Proceedings of the 21st Conf. on Uncertainty in Artificial Intelligence, 2005, AUAI Press

[8] Hinton, G.E., *Learning multiple layers of representation*, Trends in Cognitive Sciences, vol. 11, number 10, 2007, pp. 428–434

[9] Raina, R.; Madhavan, A.; Ng, A. Y., *Large-scale Deep Unsupervised Learning Using Graphics Processors*, Proceedings of the 26th Annual International Conference on Machine Learning, 2009, pp. 873–880, Montreal, Quebec, Canada

[10] Hinton, G. E.; Sejnowski, T.J., *Learning and relearning in Boltzmann machines*, 1986, Parallel distributed processing: explorations in the microstructure of cognition, vol. 1, pp. 282–317, MIT Press Cambridge, MA, USA

[11] Hinton, G., *Training products of experts by minimizing contrastive divergence*, Neural Computation, 2002, vol. 14, no. 8, pp.1771-1800

[12] Scanzio, S.; Cumani, S.; Gemello, R.; Mana, F.; Laface, P., *Parallel implementation of artificial neural network training*, Acoustics Speech and Signal Processing, 2010, IEEE, pp. 4902–4905

[13] Vesely, K.; Burget, L.; Grezl, F., *Prallel training of neural networks for speech recognition*, Text, Speech and Dialog, 2010, Springer, pp.439–446

[14] Zhu, Y.; Zhang, Y.; Pan, Y., *Large-scale Restricted Boltzmann Machines on Single CPU*, IEEE International Conference on Big Data, 2013, pp. 169–174 Silicon Valley, CA

[15] Ly, D. L.; Paprotski, V.; Yen, D., *Neural Networks on GPUs: Restricted Boltzmann Machine*, Department of Electrical and Computer Engineering, University of Toronto, ON, Canada, M5S 3G4

[16] Wang, Y.; Li, B.; Luo, R.; Chen, Y.; Xu, N.; Yang, H., *Energy efficient neural networks for big data analytics*, 2014, Design, Automation and Test in Europe Conf. and Exhibition, pp. 1–2, Dresden

[17] Su, H; Chen, H, *Experiments on parallel training of deep neural network using model averaging*, 2015, Neural and Evolutionary Computing (cs.NE), Cornell University Library

[18] Coates A.; Huval, B.; Wang, T.; Wu D; Catanzaro, B.; Ng, A., *Deep learning with COTS HPC systems*, Proceedings of The 30th International Conference on Machine Learning, 2013, p.1337-1345

[19] Chen, X.; Eversole, A.; Li, G.; Yu, D.; Seide, F., *Pipelined back-propagation for context-dependent deep neural networks*, INTERSPEECH,2012.

[20] Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.; Shenker, S.; Stoica, I., *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing*, Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, 2012, p.2–2, San Jose, CA

[21] Shi, J.; Qiu, Y.; Minhas, U. F.; Jiao, L.; Wang, Ch.; Reinwald, B.; Özcan, F., *Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics*, Proc. VLDB Endow., 2015, vol. 8, number 13, p. 2110–2121

[22] *Machine Learning Library (MLlib) Guide*, `http://spark.apache.org/docs/latest/mllib-guide.html`, visited on 19/03/2016

[23] Ghoting, A.; Krishnamurthy, R.; Pednault, E.; Reinwald, B.; Sindhwani, V.; Tatikonda, S.; Tian, Y.; Vaithyanathan, Sh., *SystemML: Declarative machine learning on MapReduce*, 2011, Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, p.231–242

[24] *Add Restricted Boltzmann machine(RBM) algorithm to MLlib*, `https://issues.apache.org/jira/browse/SPARK-4251`, visited on 19/03/2016

[25] Dykes, J., *Effective use of literature*, INM373 RMPI, 2015, City University

[26] Fisher, A.; Igel, Ch., *Training restricted Boltzmann machines: An introduction*, Pattern Recognition, 2014, vol. 47, issue 1, pp. 24–39

[27] Haykin, S., *Neural Networks: A Comprehensive Foundation*, 1998, Prentice Hall, ISBN:0132733501

[28] Shulz, H.; Mueller, A., *CUV*, `https://github.com/deeplearningais/CUV/tree/master/examples/rbm`

[29] Dawson, Ch., *Projects in computing and information systems: a student's guide*, 2009, Addison-Wesley, ISBN 9786612317651, pp.80–85

[30] O'Shea, K.T., *Massively Deep Artificial Neural Networks for Handwritten Digit Recognition*, CoRR, 2015, vol. abs/1507.05053

[31] Li, Y., *Privacy-preserving and reputation system in distributed computing with untrusted parties*, Ph.D. Thesis, 2013, University of New York, UMI 3598687

8 Appendix B – RBM implementation in DML

The RBM implementation below was accepted and is now part of the Apache SystemML project. Latest version of the scripts is also available at `https://github.com/apache/incubator-systemml`[18].

Listing 1: CD1 RBM training algorithm in DML

```
# -------------------------------------------------------------
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
# -------------------------------------------------------------

# THIS SCRIPT IMPLEMENTS A TRAINING ALGORITHM FOR RESTRICTED BOLTZMANN MACHINES
#
# The training algorithm uses one-step Contrastive Divergence as suggested
# in [Bengio, Y., Learning Deep Architectures for AI, Foundations and
# Trends in Machine Learning, Volume 2 Issue 1, January 2009, p. 1  127].
# This implemention is slightly modified and uses mini-batches to speed
# up the learning process.
#
# INPUT PARAMETERS:
# -------------------------------------------------------------
# NAME        TYPE    DEFAULT   MEANING
# -------------------------------------------------------------
# X           String  ----      Location (on HDFS) to read the matrix X of feature
#                               vectors
# W           String  ----      Location to store the estimated RBM weights
```

[18] Please note that in the SystemML codebase the *rbm_run.dml* had to be renamed to *rbm_predict*. This was requested by the package maintainer on the grounds of consistency with the naming of other algorithms in SystemML.

```
# A           String  ---- Location to store the visible units bias vector
# B           String  ---- Location to store the hidden units bias vector
# batchsize   Int     100  Number of observations in a single batch
# vis         Int     ---- Number of units in the visible layer. If not specified
#                          this will be set
#                          to the number of features in X
# hid         Int     2    Number of units in the hidden layer
# epochs      Int     10   Number of training epochs
# alpha       Double  0.1  Learning rate
# fmt         String  text Matrix output format for W,A, and B, usually "text"
#                          or "csv"
# -------------------------------------------------------------------------
# OUTPUT: Matrices containing weights and biases for the trained network
#
#        OUTPUT SIZE:    OUTPUT CONTENTS:
# W      vis x hid       RBM weights
# A      1   x vis       Visible units biases
# B      1   x hid       Hidden units biases
#
# To run an observation through the trained network use the rbm_run.dml script.
# Alternatively, you can simply compute P(h=1|v)
# using 1.0 / (1 + exp(-(X %*% W + B)) and get the values for the hidden layer
# by sampling from P(h=1|v).
#

# HOW TO INVOKE THIS SCRIPT - EXAMPLE:
# hadoop jar SystemML.jar -f rbm.dml -nvargs X=$INPUT_DIR/X W=$OUTPUT_DIR/w
#   A=$OUTPUT_DIR/a B=$OUTPUT_DIR/b batchsize=100 vis=10 hid=2 epochs=10 alpha
    =0.1

# ------------------------- START OF RBM_MINIBATCH.DML -------------------------

# Default values
learning_rate       = ifdef($alpha, 0.1)
hidden_units_count  = ifdef($hid, 2)
epoch_count         = ifdef($epochs, 10)
batch_size          = ifdef($batchsize, 100)
output_format       = ifdef($fmt, "text")

print("BEGIN RBM MINIBATCH TRAINING SCRIPT")

print("Reading X...")
X = read($X)

# If number of visible units is not set,
```

```
# set it to the number of features in X
visible_units_count = ifdef ($vis, ncol(X))

# Get the total number of observations
N = nrow(X)

# Rescale to interval [0,1]
minX = min(X)
maxX = max(X)
if (minX < 0.0 | maxX > 1.0) {
  print("X not in [0,1]. Rescaling...")
  X = (X-minX)/(maxX-minX)
}

# Initialise a weight matrix (visible_units_count x hidden_units_count)
w = 0.1 * rand(rows = visible_units_count, cols = hidden_units_count,
    pdf = "uniform")

# Setup biases and initialise with 0
a = matrix(0, rows = 1, cols = visible_units_count)    # Visible units bias
b = matrix(0, rows = 1, cols = hidden_units_count)     # Hidden units bias

# Train the RBM
print("Training starts...")

for (epoch in 1:epoch_count) {

  # Cumulative error for this epoch
  epoch_err = 0

  # Loop over each batch
  for (batch_start in seq(1, N, batch_size)) {

    # Compute the end of the batch for indexing
    batch_end = batch_start + batch_size - 1

    # Is this the last batch?
    if (batch_end > N) {
      batch_end = N
    }

    # Present the current minibatch to the visible layer
    v_1 = X[batch_start:batch_end,]

    # Get the number of observations in v_1
```

```
126     batch_N = nrow(v_1)
127
128     # POSITIVE PHASE
129
130     # Compute the probabilities P(h=1|v)
131     p_h1_given_v = 1.0 / (1 + exp(-(v_1 %*% w + b)))
132
133     # Sample from P(h=1|v)
134     h1 = p_h1_given_v > rand(rows = batch_N, cols = hidden_units_count)
135
136     # NEGATIVE PHASE
137
138     # Compute the probabilities P(v2=1|h1)
139     p_v2_given_h1  = 1.0 / (1 + exp(-(h1 %*% t(w) + a)))
140
141     # Compute the probabilities P(h2=1|v2)
142     p_h2_given_v2 = 1.0 / (1 + exp(-(p_v2_given_h1 %*% w + b)))
143
144     # UPDATE WEIGHTS AND BIASES
145     w = w + learning_rate * ((t(v_1) %*% p_h1_given_v - t(p_v2_given_h1) %*%
146         p_h2_given_v2) / batch_N)
147     a = a + (learning_rate / batch_N) * (colSums(v_1) - colSums(p_v2_given_h1))
148     b = b + (learning_rate / batch_N) * (colSums(p_h1_given_v) -
149         colSums(p_h2_given_v2))
150
151     # COMPUTE ERROR
152     minibatch_err = sum((v_1 - p_v2_given_h1) ^ 2) / batch_N
153     epoch_err = epoch_err + minibatch_err
154   }
155
156   print("Epoch " + epoch + " completed. Cummulative error is " + epoch_err)
157
158 }
159
160 # Save the RBM model
161 print("Saving RBM weights...")
162 write(w, $W, format=output_format)
163
164 print("Saving RBM biases...")
165 write(a, $A, format=output_format)
166 write(b, $B, format=output_format)
167
168 print("END OF RBM MINIBATCH TRAINING SCRIPT")
169
170 # ---------- END OF RBM_MINIBATCH.DML ----------
```

Listing 2: CD1 RBM training algorithm in DML – no mini-batches

```
# -------------------------------------------------------------
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
# -------------------------------------------------------------

# THIS SCRIPT IMPLEMENTS A TRAINING ALGORITHM FOR RESTRICTED BOLTZMANN MACHINES
#
# The training algorithm uses one-step Contrastive Divergence as suggested
# in [Bengio, Y., Learning Deep Architectures for AI, Foundations and
# Trends in Machine Learning, Volume 2 Issue 1, January 2009, p. 1  127].
# This implemention does not use mini-batches and instead tries to iterate
# over the entire dataset
#
# INPUT PARAMETERS:
# -------------------------------------------------------------
# NAME        TYPE    DEFAULT  MEANING
# -------------------------------------------------------------
# X           String  ----     Location (on HDFS) to read the matrix X of feature
#                              vectors
# W           String  ----     Location to store the estimated RBM weights
# A           String  ----     Location to store the visible units bias vector
# B           String  ----     Location to store the hidden units bias vector
# vis         Int     ----     Number of units in the visible layer. If not specified
#                              this will be set
#                              to the number of features in X
# hid         Int     2        Number of units in the hidden layer
# epochs      Int     10       Number of training epochs
# alpha       Double  0.1      Learning rate
```

```
# fmt         String  text Matrix output format for W,A, and B, usually "text"
#                           or "csv"
# ------------------------------------------------------------------------------
# OUTPUT: Matrices containing weights and biases for the trained network
#
#         OUTPUT SIZE:    OUTPUT CONTENTS:
# W       vis x hid       RBM weights
# A       1   x vis       Visible units biases
# B       1   x hid       Hidden units biases
#
# To run an observation through the trained network use the rbm_run.dml script.
# Alternatively, you can simply compute P(h=1|v) using
# 1.0 / (1 + exp(-(X %*% W + B)) and get the values for the hidden layer by
# sampling from P(h=1|v).
#

# HOW TO INVOKE THIS SCRIPT - EXAMPLE:
# hadoop jar SystemML.jar -f rbm.dml -nvargs X=$INPUT_DIR/X W=$OUTPUT_DIR/w
#         A=$OUTPUT_DIR/a B=$OUTPUT_DIR/b vis=10 hid=2 epochs=10 alpha=0.1


# ------------------------- START OF RBM_NOMINIBATCH.DML -----------------------

# Default values
learning_rate        = ifdef($alpha, 0.1)
hidden_units_count   = ifdef($hid, 2)
epoch_count          = ifdef($epochs, 10)
output_format        = ifdef($fmt, "text")

print("BEGIN RBM MINIBATCH TRAINING SCRIPT")

print("Reading X...")
X = read($X)

# If number of visible units is not set,
# set it to the number of features in X
visible_units_count = ifdef ($vis, ncol(X))

# Get the total number of observations
N = nrow(X)

# Rescale to interval [0,1]
minX = min(X)
maxX = max(X)
if (minX < 0.0 | maxX > 1.0) {
```

```
  print("X not in [0,1]. Rescaling...")
  X = (X-minX)/(maxX-minX)
}

# Initialise a weight matrix (visible_units_count x hidden_units_count)
w = 0.1 * rand(rows = visible_units_count, cols = hidden_units_count,
               pdf = "uniform")

# Setup biases and initialise with 0
a = matrix(0, rows = 1, cols = visible_units_count) # Visible units bias
b = matrix(0, rows = 1, cols = hidden_units_count)  # Hidden units bias

# Train the RBM
print("Training starts...")

for (epoch in 1:epoch_count) {

  # Cumulative error for this epoch
  epoch_err = 0

  # Present the current minibatch to the visible layer
  v_1 = X

  # POSITIVE PHASE

  # Compute the probabilities P(h=1|v)
  p_h1_given_v = 1.0 / (1 + exp(-(v_1 %*% w + b)))

  # Sample from P(h=1|v)
  h1 = p_h1_given_v > rand(rows = N, cols = hidden_units_count)

  # NEGATIVE PHASE

  # Compute the probabilities P(v2=1|h1)
  p_v2_given_h1  = 1.0 / (1 + exp(-(h1 %*% t(w) + a)))

  # Compute the probabilities P(h2=1|v2)
  p_h2_given_v2 = 1.0 / (1 + exp(-(p_v2_given_h1 %*% w + b)))

  # UPDATE WEIGHTS AND BIASES
  w = w + learning_rate * ((t(v_1) %*% p_h1_given_v -
      t(p_v2_given_h1) %*% p_h2_given_v2) / N)
  a = a + (learning_rate / N) * (colSums(v_1) - colSums(p_v2_given_h1))
  b = b + (learning_rate / N) * (colSums(p_h1_given_v)
      - colSums(p_h2_given_v2))
```

```
135
136   # COMPUTE ERROR
137   epoch_err = sum((v_1 - p_v2_given_h1) ^ 2) / N
138
139   print("Epoch " + epoch + " completed. Error is " + epoch_err)
140   #print(epoch + "," + epoch_err)
141
142 }
143
144 # Save the RBM model
145 print("Saving RBM weights...")
146 write(w, $W, format=output_format)
147
148 print("Saving RBM biases...")
149 write(a, $A, format=output_format)
150 write(b, $B, format=output_format)
151
152 print("END OF RBM MINIBATCH TRAINING SCRIPT")
153
154 # ---------- END OF RBM_NOMINIBATCH.DML ----------
```

Listing 3: Script for processing data using the trained RBM

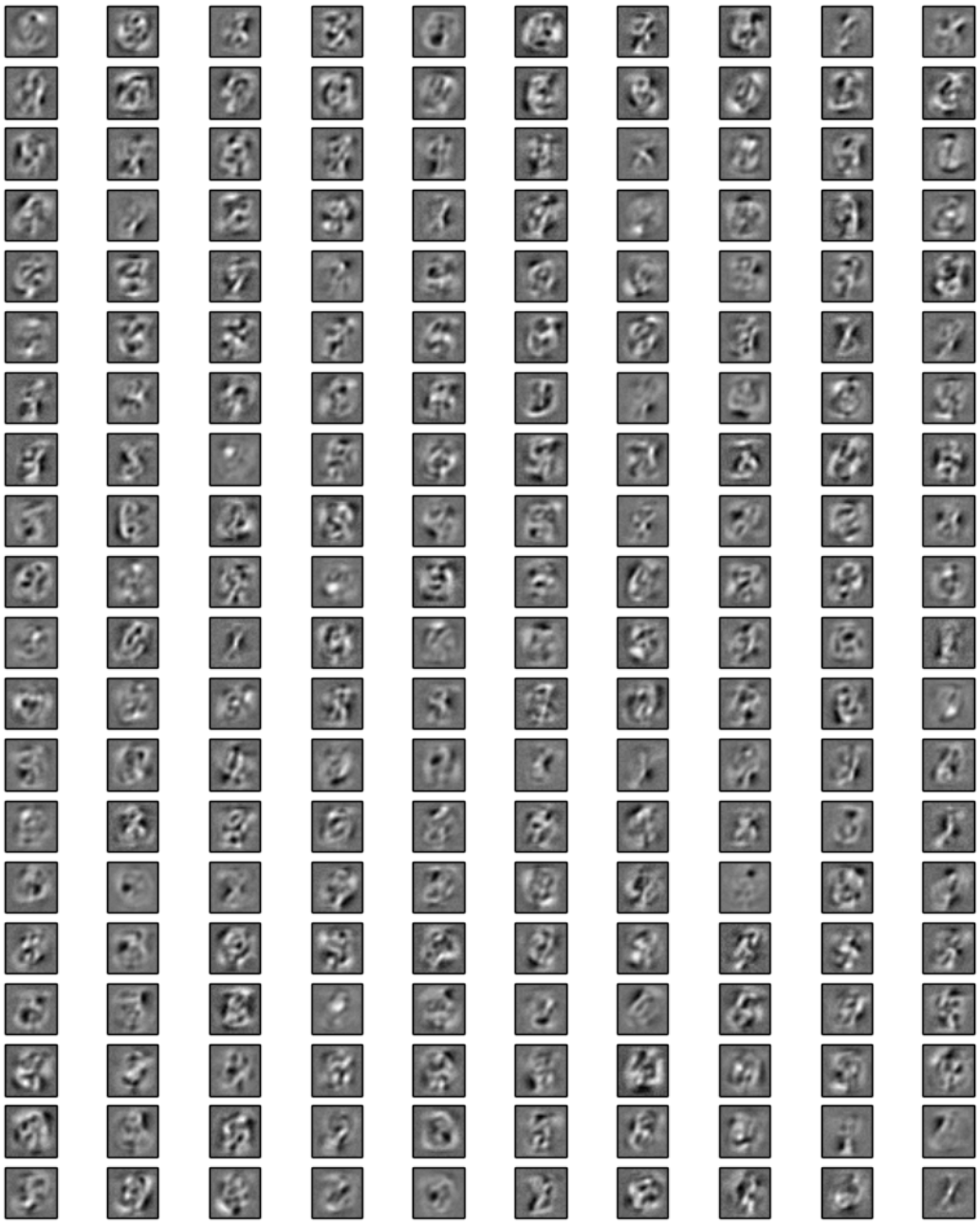
```
# -------------------------------------------------------------
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
# -------------------------------------------------------------

# THIS SCRIPT USES A TRAINED RESTRICTED BOLTZMANN MACHINES TO PROCESS A SET
# OF OBSERVATIONS
#
# The RBM training is handled by the rbm_minibatch.dml script.
#
# INPUT PARAMETERS:
# -------------------------------------------------------------------------
# NAME        TYPE    DEFAULT  MEANING
# -------------------------------------------------------------------------
# X           String  ---      Location (on HDFS) to read the matrix of observations X
# W           String  ---      Location (on HDFS) to read the RBM's weight vector
# A           String  ---      Location (on HDFS) to read the RBM's visible units
#                              bias vector
# B           String  ---      Location (on HDFS) to read the RBM's hidden units
#                              bias vector
# O           String  ---      Location to store the processed observations
# fmt         String  csv      Matrix output format for O, usually "text" or "csv"
# -------------------------------------------------------------------------
# OUTPUT: A matrix containing a sample of P(h=1|v) for each observation in X
#
#         OUTPUT SIZE:    OUTPUT CONTENTS:
# O       N x ncol(W)     RBM outputs for each observation in X
#
```

```
45
46 # HOW TO INVOKE THIS SCRIPT - EXAMPLE:
47 # hadoop jar SystemML.jar -f rbm_run.dml -nvargs X=$INPUT_DIR/X W=$INPUT_DIR/w
48 #    A=$INPUT_DIR/a B=$INPUT_DIR/b O=$OUTPUT_DIR/O fmt="csv"
49
50 # ------------------------------ START OF RBM_RUN.DML ------------------------------
51
52 # Default values
53 output_format          = ifdef($fmt, "csv")
54
55 print("BEGIN RBM SCRIPT")
56
57 print("Reading model data...")
58 #Read data set
59 X = read($X)
60
61 # Get the total number of observations
62 N = nrow(X)
63
64 # Read model coefficients
65 w = read($W)
66 a = read($A)
67 b = read($B)
68
69 hidden_units_count = ncol(w)
70
71 # Rescale to interval [0,1]
72 minX = min(X)
73 maxX = max(X)
74 if (minX < 0.0 | maxX > 1.0) {
75   print("X not in [0,1]. Rescaling...")
76   X = (X-minX)/(maxX-minX)
77 }
78
79 print("Running RBM...")
80
81 # Compute the probabilities P(h=1|v)
82 p_h1_given_v = 1.0 / (1 + exp(-(X %*% w + b)))
83
84 # Sample from P(h=1|v)
85 h1 = p_h1_given_v > rand(rows = N, cols = hidden_units_count)
86
87 print("Saving hidden layer sample...")
88 write(h1, $O, format=output_format)
89
```

```
90 print("END RBM SCRIPT")
91
92 # ---------------------------------- END OF RBM_RUN.DML ----------------------------------
```

9 Appendix C – The full set of features learnt in the functional testing phase



200 features learnt by the RBM while training against the pre-processed MNIST dataset.

10 Appendix D – Contents of stopwords_en.txt

i, a, as, able, about, above, according, accordingly, across, actually, after, afterwards, again, against, aint, all, allow, allows, almost, alone, along, already, also, although, always, am, among, amongst, an, and, another, any, anybody, anyhow, anyone, anything, anyway, anyways, anywhere, apart, appear, appreciate, appropriate, are, arent, around, as, aside, ask, asking, associated, at, available, away, awfully, be, became, because, become, becomes, becoming, been, before, beforehand, behind, being, believe, below, beside, besides, best, better, between, beyond, both, brief, but, by, cmon, cs, came, can, cant, cannot, cant, cause, causes, certain, certainly, changes, clearly, co, com, come, comes, concerning, consequently, consider, considering, contain, containing, contains, corresponding, could, couldnt, course, currently, definitely, described, despite, did, didnt, different, do, does, doesnt, doing, dont, done, down, downwards, during, each, edu, eg, eight, either, else, elsewhere, enough, entirely, especially, et, etc, even, ever, every, everybody, everyone, everything, everywhere, ex, exactly, example, except, far, few, fifth, first, five, followed, following, follows, for, former, formerly, forth, four, from, further, furthermore, get, gets, getting, given, gives, go, goes, going, gone, got, gotten, greetings, had, hadnt, happens, hardly, has, hasnt, have, havent, having, he, hes, hello, help, hence, her, here, heres, hereafter, hereby, herein, hereupon, hers, herself, hi, him, himself, his, hither, hopefully, how, howbeit, however, id, ill, im, ive, ie, if, ignored, immediate, in, inasmuch, inc, indeed, indicate, indicated, indicates, inner, insofar, instead, into, inward, is, isnt, it, itd, itll, its, its, itself, just, keep, keeps, kept, know, knows, known, last, lately, later, latter, latterly, least, less, lest, let, lets, like, liked, likely, little, look, looking, looks, ltd, mainly, many, may, maybe, me, mean, meanwhile, merely, might, more, moreover, most, mostly, much, must, my, myself, name, namely, nd, near, nearly, necessary, need, needs, neither, never, nevertheless, new, next, nine, no, nobody, non, none, noone, nor, normally, not, nothing, novel, now, nowhere, obviously, of, off, often, oh, ok, okay, old, on, once, one, ones, only, onto, or, other, others, otherwise, ought, our, ours, ourselves, out, outside, over, overall, own, particular, particularly, per, perhaps, placed, please, plus, possible, presumably, probably, provides, que, quite, qv, rather, rd, re, really, reasonably, regarding, regardless, regards, relatively, respectively, right, said, same, saw, say, saying, says, second, secondly, see, seeing, seem, seemed, seeming, seems, seen, self, selves, sensible, sent, serious, seriously, seven, several, shall, she, should, shouldnt, since, six, so, some, somebody, somehow, someone, something, sometime, sometimes, somewhat, somewhere, soon, sorry,

specified, specify, specifying, still, sub, such, sup, sure, ts, take, taken, tell, tends, th, than, thank, thanks, thanx, that, thats, thats, the, their, theirs, them, themselves, then, thence, there, theres, thereafter, thereby, therefore, therein, theres, thereupon, these, they, theyd, theyll, theyre, theyve, think, third, this, thorough, thoroughly, those, though, three, through, throughout, thru, thus, to, together, too, took, toward, towards, tried, tries, truly, try, trying, twice, two, un, under, unfortunately, unless, unlikely, until, unto, up, upon, us, use, used, useful, uses, using, usually, value, various, very, via, viz, vs, want, wants, was, wasnt, way, we, wed, well, were, weve, welcome, well, went, were, werent, what, whats, whatever, when, whence, whenever, where, wheres, whereafter, whereas, whereby, wherein, whereupon, wherever, whether, which, while, whither, who, whos, whoever, whole, whom, whose, why, will, willing, wish, with, within, without, wont, wonder, would, would, wouldnt, yes, yet, you, youd, youll, youre, youve, your, yours, yourself, yourselves, zero

11 Appendix E – The RBM data transformation script

This script is used to transform the pre-processed Gutenberg dataset into a *label-feature* vector matrix suitable for RBM learning. The functionality of this script it discussed in details in Section 3.4.2.

Listing 4: RBM data transformation script in Python/PySpark

```
# Individual Project, INM363, City University
# (C) 2016 Nikolay Manchev
# This work is licensed under the Creative Commons Attribution 4.0 International License.
# To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.
#
# Usage: spark-submit rbm-transform.py
#
# To set specific processing parameters, please refer to the PATHS, LIMITS, AND PARAMETERS section

import csv
import re
import sys
import os
import os.path
import fnmatch
import datetime
import string

from pyspark import SparkContext
from pyspark import SparkConf

from shutil import rmtree
from operator import add
from time import time

# Prints elapsed time between start_time and now.
# Used to time execution of different functions
def time_exec(start_time, message):
  exec_time = time() - start_time
  print message + ' Time elapsed: %s' % str(datetime.timedelta(seconds = int(exec_time)))

# Finds the top N most frequent subjects
# Expect input is in the format of:
#    (DocumentID, [subject1, subject2, ...., subjectm])
```

```
35 # Returns a list of subjects
36 def get_top_subjects(rdd, n=10):
37
38   subjects = rdd.flatMap(lambda (bookid, s): (s)).map(lambda (s): (s,1))
39
40   subjects = subjects.reduceByKey(add)
41   subjects = subjects.map(lambda (sub, count): (count, sub))
42   subjects = subjects.top(n)
43
44   # Drop the count
45   subjects =  [x[1] for x in subjects]
46   return subjects
47
48 # Generates a bag of words dictionary containing the top N most frequently
49 # encountered terms.
50 #
51 # Returns an RDD in the format of
52 # (word, occurrences)
53 #
54 # Parameters:
55 #   * data          - an RDD in the format of (textid, [(word1,count1),...,(
      wordn,countn)])
56 #   * dict_size     - size of the dictionary (top N)
57 def get_dict(data, dict_size):
58   # Keep only the words and add an 1 for counting
59   word_freq = data.flatMap(lambda (textid, wclist): [(wc[0], wc[1]) for wc in
        wclist])
60   # Sum
61   word_freq = word_freq.reduceByKey(add)
62   dictionary = word_freq.takeOrdered(dict_size, key = lambda x: -x[1])
63   # Discard the counts, retain the words only
64   dictionary = [x[0] for x in dictionary]
65   return dictionary
66
67 # Vectorizes all documens in the data RDD using the dictionary,
68 # according to the feature creation strategy set by the
69 # boolean_bag parameter
70 #
71 # Returns an RDD in the format of
72 # (docid, [0,0,0,1.....,0])  - for binary vector
73 # (docid, [0,11,0,25,.....,0])  - for term frequency vector
74 #
75 # Parameters:
76 #   * data          - an RDD in the format of (textid, [(word1,count1),...,(
      wordn,countn)])
```

```
#   * dictionary     - a list containing the global dictionary (word1, word2,...)
def vectorize(data, dictionary):
  if (boolean_bag):
    # Drop the count, keep the words only
    data = data.map(lambda (textid, x): (textid, [i[0] for i in x]))
    data = data.map(lambda (textid, x): (textid, boolean_bag_of_words(x,
        dictionary)))
  else:
    # Build a bag with frequencies
    data = data.map(lambda (textid, x): (textid, [(i[0],i[1]) for i in x]))
    data = data.map(lambda (textid, x): (textid, bag_of_words(x, dictionary)))
  return data

# Constructs a frequency-based feature vector
#
# Returns an array of the form
#   [0,11,0,25,.....,0]
# where each element represents the number of times each dictionary word appears
    in the document
#
# Parameters:
#   * word_counts    - an RDD in the format of (textid, [(word1,count1),...,(
    wordn,countn)])
#   * dictionary     - a list containing the global dictionary (word1, word2,...)
def bag_of_words(word_counts, dictionary):

   words  = [x[0] for x in word_counts]
   counts = [x[1] for x in word_counts]

   dict = []
   for term in dictionary:
     try:
      index = words.index(term)
      dict.append(counts[index])
     except:
      dict.append(0)

   return dict

# Constructs a binry feature vector
#
# Returns an array of the form
#   [0,1,0,1,.....,0]
# where each element is a boolean (1/0) indicating if a dictionary term appears
    in the document
```

```
118 #
119 # Parameters:
120 #   * words          - an array containing all words in the document [word1,
      word2, ...]
121 #   * dictionary     - a list containing the global dictionary (word1, word2,...)
122 def boolean_bag_of_words(words, dictionary):
123     return [(term in words)+0 for term in dictionary]
124
125 if __name__ == "__main__":
126
127   #************************************************************************
128   # PATHS, LIMITS, AND PARAMETERS
129   #************************************************************************
130
131   # Use binary-weighing for the feature vector
132   boolean_bag = False
133
134   # Default dictionary size
135   dictionary_size = 10000
136
137   # Top N subjects to retain
138   top_subjects = 10
139
140   # Directories to load files from.
141   input_path    = '/data/GUT/pre-processed'
142   csv_save_path = '/data/GUT/rbm'
143
144   #************************************************************************
145   # CREATE SPARK CONTEXT
146   #************************************************************************
147
148   config = SparkConf()
149
150   sc = SparkContext(conf=config, appName="rbm-transform")
151
152   # ************************************************************************
153   # LOADING TEXT & XML FILES
154   # ************************************************************************
155
156   # Start the execution timer
157   start_time = time()
158
159   print '*****************************************************************'
160   print 'Reading pre-processed data from %s' % input_path
161   print '*****************************************************************'
```

```
162
163 doc_read_path = input_path + '/docs.rdd'
164 meta_read_path = input_path + '/meta.rdd'
165
166 all_text_files = sc.pickleFile(doc_read_path)
167 all_meta_files = sc.pickleFile(meta_read_path)
168
169 time_exec(start_time, 'TOTAL READING TIME:')
170
171 start_time = time()
172
173 print '*****************************************************************'
174 print 'Number of input files : %s' % all_text_files.count()
175 print 'Retaining top %s subjects...' % top_subjects
176 print '*****************************************************************'
177
178 subjects = get_top_subjects(all_meta_files, top_subjects)
179
180 # Print the top N subjects
181 for x in subjects:
182   print '%s - %s'% (subjects.index(x) + 1, x)
183
184 # Retain only the meta files with subject in the top N subjects list
185 # Remove any entries with multiple subjects in the top N list -- they can hurt
      the classification
186 all_meta_files = all_meta_files.flatMap(lambda (docid, data): [(docid, x) for x
       in data])
187 all_meta_files = all_meta_files.filter(lambda (docid, sub): sub in subjects)
188 all_meta_files = all_meta_files.map(lambda (docid, sub): (docid, [sub])).
      reduceByKey(add)
189 all_meta_files = all_meta_files.filter(lambda (docid, sub): len(sub) == 1).map(
      lambda (docid, sub): (docid, sub[0]))
190
191 all_meta_keys = all_meta_files.keys().collect()
192 all_text_files = all_text_files.filter(lambda (docid, values): docid in
      all_meta_keys)
193
194 print '*****************************************************************'
195 print 'Number of retained text files : %s' % all_text_files.count()
196 print '*****************************************************************'
197
198 time_exec(start_time, 'SUBJECT FILTERING TIME:')
199
200 # Generate the global dictionary
201
```

```
202 start_time = time()
203
204 print '**********************************************************************'
205 print 'Generating dictionary...'
206 print '**********************************************************************'
207
208 dictionary = get_dict(all_text_files, dictionary_size)
209
210 time_exec(start_time, 'DICTIONARY GENERATION TIME:')
211
212 start_time = time()
213
214 # Create the feature vectors
215
216 print '**********************************************************************'
217 print 'Vectorizing texts...'
218 print '**********************************************************************'
219
220 all_text_files = vectorize(all_text_files, dictionary)
221
222 time_exec(start_time, 'VECTORIZATION TIME:')
223
224 # Save the matrix of documents D
225 #
226 #      | y_1 x_11 x_12 ... x_1n |
227 # D = | ... ... ... ... .... |
228 #      | y_m x_m1 x_m2 ... x_mn |
229 #
230
231 start_time = time()
232
233 print '**********************************************************************'
234 print 'Saving results...'
235 print '**********************************************************************'
236
237 rbm_data = all_text_files.join(all_meta_files)
238 # Remove the docid
239 rbm_data = rbm_data.map(lambda (docid, values): values)
240
241 # Transform to csv values
242 rbm_data = rbm_data.map(lambda (features, subject): ((subjects.index(subject) +
        1), features))
243
244 rbm_data = rbm_data.map(lambda (sub, val): (sub, ','.join(str(d) for d in val))
    )
```

```
245
246 if os.path.exists(csv_save_path):
247   rmtree(csv_save_path)
248
249 csv_lines = rbm_data.map(lambda (data): ','.join(str(d) for d in data))
250 csv_lines.coalesce(1,True).saveAsTextFile(csv_save_path)
251
252 time_exec(start_time, 'SAVING RESULTS TIME:')
```

List of abbreviations

AMI ..... Amazon Machine Image

ANN ..... Artificial Neural Network

AWS ..... Amazon Web Services

BM ..... Boltzmann Machine

CD ..... Contrastive Divergence

CSV ..... Comma-separated values

CUDA ..... Compute Unified Device Architecture

DAG ..... Directed Acyclic Graph

DML ..... Declarative Machine learning Language

EC2 ..... Elastic Cloud 2

GPU ..... Graphical Processing Unit

HDFS ..... Hadoop Distributed File System

HDP ..... Hortonworks Data Platform

HOP ..... High-Level Operator

JSON ..... JavaScript Object Notation

JVM ..... Java Virtual Machine

LOP ..... Low-Level Operator

MCMC ..... Markov Chain Monte Carlo

MLP ..... Multilayer Perceptron

MTD ..... Matrix Dimensions file

PCD ..... Persistent Contrastive Divergence

RBM ..... Restricted Boltzmann Machine

RDD ..... Resilient Distributed Dataset

RDF ..... Resource Description Framework

SSE ..... Sum of Squared Errors

SVM ..... Support Vector Machine

References

Abdollahi, B. & Nasraoui, O. (2016), 'Explainable restricted boltzmann machines for collaborative filtering', ICML Workshop on Human Interpretability in Machine Learning.

Ackley, D., Hinton, G. & Sejnowski, T. (1985), 'A learning algorithm for boltzmann machines', *Cognitive Science* pp. 147–169.

Add Restricted Boltzmann machine(RBM) algorithm to MLlib (2016), `https://issues.apache.org/jira/browse/SPARK-4251`. Visited on 18/11/2016.

Apache Hadoop (2016), `http://hadoop.apache.org`. Visited on 01/11/2016.

Apache Mahout: Boltzmann Machines (2016), `https://mahout.apache.org/users/classification/restricted-boltzmann-machines.html`. Visited on 17/11/2016.

Apache Spark: Lightning-fast cluster computing (2016), `http://spark.apache.org`. Visited on 02/11/2016.

Apache SystemML (2016), `https://systemml.apache.org`. Visited on 02/11/2016.

Beale, R. & Jackson, T. (1990), *Neural Computing: An Introduction*, IOP Publishing Ltd.

Bengio, Y. (2009), 'Learning deep architectures for ai', *Foundations and Trends in Machine Learning* **2**(1), 1–127.

Best practices for selecting Apache Hadoop hardware (2016), `http://hortonworks.com/blog/best-practices-for-selecting-apache-hadoop-hardware/`. Visited on 06/11/2016.

Boehm, M. (2015), 'Costing generated runtime execution plans for large-scale machine learning programs', *CoRR* .

Boehm, M., Burdick, D., Evfimievski, A., Reinwald, B., Reiss, F., Sen, P., Tatikonda, S. & Tian, Y. (2014), 'Systemml's optimizer: Plan generation for large-scale machine learning programs', *IEEE Data Eng. Bull* **3**(37).

Carreira-Perpinan, M. & Hinton, G. (2005), On contrastive divergence learning, *in* 'Intelligence, Artificial and Statistics, 2005, Barbados'.

Chen, X., Eversole, A., Li, G., Yu, D. & Seide, F. (2012), Pipelined back-propagation for context-dependent deep neural networks, *in* 'INTERSPEECH'.

Classification and Shelflisting, The Library of Congress (2016), `http://www.loc.gov/aba/cataloging/classification/`. Visited on 04/11/2016.

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B. & Ng, A. (2013), 'Deep learning with cots hpc systems', *Proceedings of The 30th International Conference on Machine Learning* pp. 1337–1345.

Dean, J. & Ghemawat, S. (2004), 'Mapreduce: simplified data processing on large clusters', *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation* **6**, 10–10.

Deep Learning (2016), `http://deeplearning.net/tutorial/index.html`. Visited on 11/11/2016.

DeepLearning 0.1 Documentation – Restricted Boltzmann Machines (2016), `http://deeplearning.net/tutorial/rbm.html`. Visited on 10/11/2016.

Deploying a Hadoop Cluster on Amazon EC2 with HDP2 (2016), `http://hortonworks.com/blog/deploying-hadoop-cluster-amazon-ec2-hortonworks`. Visited on 11/10/2016.

DML Reference (2016), `https://apache.github.io/incubator-systemml/dml-language-reference.html`. Visited on 04/11/2016.

Dvorsky, M. (2016), 'History of massive-scale sorting experiments at google', Google Cloud Big Data and Machine Learning Blog – `https://cloud.google.com/blog/big-data/2016/02/history-of-massive-scale-sorting-experiments-at-google`.

Fisher, A. & Igel, C. (2010), 'Empirical analysis of the divergence of gibbs sampling based learning algorithms for restricted boltzmann machines', *Proceedings of the 20th international conference on Artificial neural networks : Part III* pp. 208–217.

Fisher, A. & Igel, C. (2014), 'Training restricted boltzmann machines: An introduction', *Pattern Recognition* **47**(1), 24–39.

Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y. & Vaithyanathan, S. (2011), 'Systemml: Declarative machine learning on mapreduce', *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering* pp. 231–242.

Grother, P. (1995), 'Nist special database 19 – handprinted forms and characters database'. Visual Image Processing Group, Advanced Systems Division, National Institute of Standards and Technology.

Harrell, F. E. J., Lee, K. L., Califf, R. M., Pryor, D. B. & Rosati, R. A. (1984), 'Regression modelling strategies for improved prognostic prediction', *Stat Med.* **3**(2), 143–152.

Harris, Z. (1954), 'Distributional structure', *Word* **10**(23), 146–162.

Haykin, S. (1998), *Neural Networks: A Comprehensive Foundation, Second Edition*, Prentice Hall.

Hinton, G. (2002), 'Training products of experts by minimizing contrastive divergence', *Neural Computation* **14**(8), 1771–1800.

Hinton, G. (2012), A practical guide to training restricted boltzmann machines., *in* 'Neural Networks: Tricks of the Trade (2nd ed.)', Vol. 7700 of *Lecture Notes in Computer Science*, Springer, pp. 599–619.

Hinton, G. & Salakhutdinov, R. R. (2006), 'Reducing the dimensionality of data with neural networks', *Science* **313**(5786), 504–507.

Isakki, P. & Rajagopalan, S. (2001), 'Mining unstructured data using artificial neural network and fuzzy inference systems model for customer relationship management', *IJCSI International Journal of Computer Science Issues* **8**(4).

Jeffrey, T. (2008), 'Project gutenberg digital library seeks to spur literacy', Archived from the original on 14 March 2008. Retrieved 20 August 2007.

Kitchin, R. (2014), *The Data Revolution: Big Data, Open Data, Data Infrastructures & Their Consequences*, SAGE Publications Ltd.

Larochelle, H. & Bengio, Y. (2008), 'Classification using discriminative restricted boltzmann machines', *Proceedings of the 25th international conference on Machine learning - ICML* p. 536.

Le Roux, N. & Bengio, Y. (2008), 'Representational power of restricted boltzmann machines and deep belief networks', *Neural Computation* **20**(6), 1631–49.

LeCun, Y., Cortes, C. & Burges, C. (2016), 'The mnist database of handwritten digits', `http://yann.lecun.com/exdb/mnist/`. Visited on 04/11/2016.

Li, Y. (2013), Privacy-preserving and reputation system in distributed computing with untrusted parties, PhD thesis, University of New York. UMI 3598687.

Ly, D. L., Paprotski, V. & Yen, D. (2008), 'Neural networks on gpus: Restricted boltzmann machine'.

Manchev, N. (2016), 'Multi-node restricted boltzmann machines for big data', `https://www.bigdataspain.org/program/`.

Manchev, N. (2017), 'Multi-node restricted boltzmann machines for big data', `http://conferences.oreilly.com/strata/strata-eu`.

Mavenized JCuda (2016), `https://github.com/MysterionRise/mavenized-jcuda`. Visited on 17/11/2016.

McKinsey Global Institute (2011), 'Big data: The next frontier for innovation, competition & productivity', 485 Madison Avenue, New York NY 10022, USA.

Norvig, P. (2014), 'Approximate timing for various operations on a typical pc', `http://norvig.com/21-days.html#answers`. Visited on 06/11/2016.

Project Gutenberg (2016), `https://www.gutenberg.org/wiki/Main_Page`. Visited on 03/11/2016.

Project Gutenberg: The CD and DVD Project (2016), `http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project`. Visited on 04/11/2016.

Project Gutenberg: The Complete Gutenberg Catalog (2016), `http://www.gutenberg.org/cache/epub/feeds/rdf-files.tar.zip`. Visited on 04/11/2016.

Python ElementTree XML API (2016), `https://docs.python.org/2/library/xml.etree.elementtree.html`. Visited on 07/11/2016.

Raina, R., Madhavan, A. & Ng, A. Y. (2009), 'Large-scale deep unsupervised learning using graphics processors', *Proceedings of the 26th Annual International Conference on Machine Learning* pp. 873–880.

Rosebrock, A. (2014), 'Applying deep learning and a rbm to mnist using python', `http://www.pyimagesearch.com/2014/06/23/applying-deep-learning-rbm-mnist-using-python`.

Salakhutdinov, R., Mnih, A. & Hinton, G. (2007), 'Restricted boltzmann machines for collaborative filtering', *Proceedings of the International Conference on Machine Learning* **24**, 791–798.

Scanzio, S., Cumani, S., Gemello, R., Mana, F. & Laface, P. (2010), 'Parallel implementation of artificial neural network training', *Acoustics Speech and Signal Processing* pp. 4902–4905.

Smolensky, P. (1986), 'Information processing in dynamical systems: foundations of harmony theory', *Parallel distributed processing: explorations in the microstructure of cognition* **1**, 194–281.

Su, H. & Chen, H. (2015), 'Experiments on parallel training of deep neural network using model averaging', *Neural and Evolutionary Computing (cs.NE)* .

SystemML Algorithms Reference (2016), `https://sparktc.github.io/systemml/algorithms-reference.html`. Visited on 18/11/2016.

SystemML: Initial prototype for GPU backend (2016), `https://github.com/apache/incubator-systemml/blob/5decbe64b362e29901f15231c672c7b7816ed55a/docs/devdocs/gpu-backend.md`. SystemML GitHub Repository.

Theano 0.8.2 documentation (2016), `http://deeplearning.net/software/theano/`. Visited on 17/11/2016.

Tieleman, T. (2008), Training restricted boltzmann machines using approximations to the likelihood gradient, *in* 'ICML'.

Vesely, K., Burget, L. & Grezl, F. (2010), 'Prallel training of neural networks for speech recognition', *Text, Speech and Dialog* pp. 439–446.

Wang, Y., Li, B., Luo, R., Chen, Y., Xu, N. & Yang, H. (2014), 'Energy efficient neural networks for big data analytics', *Design, Automation and Test in Europe Conference and Exhibition* pp. 1–2.

Weka 3: Data Mining Software in Java (2016), `http://www.cs.waikato.ac.nz/ml/weka/index.html`. Visited on 04/11/2016.

Xing, E. P., Yan, R. & Hauptmann, A. G. (2005), 'Mining associated text and images with dual-wing harmoniums', *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence* .

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S. & Stoica, I. (2012), 'Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing', *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* pp. 2–2.

Zhu, Y., Zhang, Y. & Pan, Y. (2013), 'Large-scale restricted boltzmann machines on single gpu', *IEEE International Conference on Big Data* pp. 169–174.