LCD: A Fast Contrastive Divergence Based Algorithm for Restricted Boltzmann Machine

Lin Ning
Department of Computer Science
North Carolina State University
lning@ncsu.edu

Randall Pittman
Department of Computer Science
North Carolina State University
rbpittma@ncsu.edu

Xipeng Shen
Department of Computer Science
North Carolina State University
xshen5@ncsu.edu

***Abstract*—Restricted Boltzmann Machine (RBM) is the building block of Deep Belief Nets and other deep learning tools. Fast learning and prediction are both essential for practical usage of RBM-based machine learning techniques. This paper proposes *Lean Contrastive Divergence (LCD)*, a modified Contrastive Divergence (CD) algorithm, to accelerate RBM learning and prediction without changing the results. LCD avoids most of the required computations with two optimization techniques. The first is called *bounds-based filtering*, which, through triangle inequality, replaces expensive calculations of many vector dot products with fast bounds calculations. The second is *delta product*, which effectively detects and avoids many repeated calculations in the core operation of RBM, Gibbs Sampling. The optimizations are applicable to both the standard contrastive divergence learning algorithm and its variations. Results show that the optimizations can produce several-fold (up to 3X for training and 5.3X for prediction) speedups.**

I. Introduction

Recent years have witnessed a rapidly growing interest in neural network-based deep learning, which has brought some significant advancements to a number of domains, ranging from image processing to speech recognition, automatic translation, business analytics, and so forth.

The artificial neural networks used in deep learning are of various kinds and structures. Most of them fall into one of three categories: Convolutional Neural Networks (CNN), Restricted Boltzmann Machine (RBM) networks, and Recursive Neural Networks (RNN). For their different natures, they are each good at some kinds of tasks. For instance, CNN is the popular for image processing while RBM is the primary networks used in modern speech recognition.

What is common is that these networks all take a long time to train, and meanwhile, in their usage for prediction, they are all expected to work efficiently, in both time and energy. Training time determines how soon the network can get ready for use. Despite the use of GPU clusters, the process still often takes days or weeks, especially when the width or depth of the networks have to get tuned through trial-and-error explorations. The efficiency at their usage time (for predictions) determines the quality of service they offer and the energy they consume, which are essential as these networks are often used for interactions with users on edge or mobile devices (e.g., smartphones).

This material is based upon work supported by DOE Early Career Award (DE-SC0013700), and National Science Foundation (NSF) CCF-1455404, CCF-1525609, CNS-1717425, CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

This work focuses on the efficiency of RBM networks. Unlike CNN that has received many studies, RBM networks have remained much less explored. It is however still important. As an effective way to extract meaningful high-level representations (e.g., hidden features in images) from various kinds of data (binary, integer and real values), RBM is the primary building block of Deep Belief Network (DBN), Deep Boltzmann Machine (DBM), and other influential types of neural networks. As Hinton and others [1] have shown, RBM is equivalent to infinite directed nets with tied weights, which suggests an efficient learning algorithm for multilayer RBM-based deep nets in which the weights are not tied. Since then, RBM-based networks, especially DBN, have been serving as essential deep learning tools for various domains, from image processing to automatic speech recognition, call routing system, spoken language understanding, sentiment classification, movie ratings and so forth (e.g., [2], [3], [4]). Additionally, they have been used as possible solutions to bioinformatics problems such as MRI images classification, health monitoring and gene or protein studies (e.g.,[5], [6], [7]).

Like all other deep learning tools, RBM-based learning is time consuming, facing efficiency concerns in both its training and its usage. The de facto learning algorithm for RBM is Contrastive Divergence (CD) [8], an algorithm based on Gibbs Sampling. Later studies have proposed some variations of the CD algorithm (e.g., [9], [10]). For instance, Persistent Contrastive Divergence (PCD) [10] modifies the basic CD algorithm by initializing a Markov Chain at the state in which it ended for the previous model. Although these algorithms improve the convergence rate as well as the learning result sometimes, they still leave a large room for efficiency improvement; moreover, they do not help with the efficiency of the usage of the networks.

In this work, we present Lean Contrastive Divergence (LCD), an enhanced algorithm that accelerates the training and predicting processes from a different perspective. It could speed up the class of RBM-based deep learning, improving the efficiency of both the training and the usage of the networks. Instead of introducing new approximations as previous studies did, LCD detects and avoids some unnec-

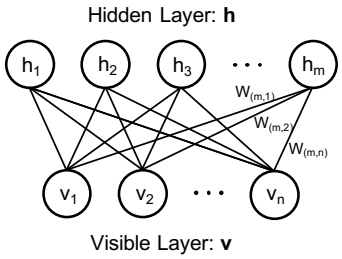


Fig. 1. A Binary RBM with n visible units and m hidden units.

essary computations in RBM without affecting the quality of the resulting network. It achieves this by introducing two algorithmic optimizations. In both the training and usage of RBM, dot products between vectors consume most of the time. These two optimizations avoid computing unnecessary dot products or part of them from two different aspects. The first optimization (Section III-A) is called *bounds-based filtering*, which uses lower and upper bounds to find unnecessary dot product calculations and avoid them completely. The second optimization (Section III-B) is called *delta product*, which does a more fine-grained redundancy elimination for some operations in the (necessary) dot products.

To evaluate the efficacy of LCD, we experiment with the basic binary RBM and its typical variations, and measure the performance of RBM training on GPU, as well as the performance of RBM usage on mobile devices. The results on seven public datasets show that using LCD brings significant speedups (up to 3.0X for training and 5.3X for usage). These performance benefits come without sacrifice of accuracy. The optimizations only avoid unnecessary computations and do not change the semantics of the algorithm, so they do not change the learning or prediction results.

II. Background

Figure 1 illustrates an RBM. It is composed of two layers of units: a visible layer ($\mathbf{v}$) with n visible units and a hidden layer ($\mathbf{h}$) with m hidden units. They are denoted by a visible unit vector $\mathbf{v}$ and a hidden unit vector $\mathbf{h}$ respectively. An RBM is characterized by a set of parameters: $\theta = (\mathbf{a}, \mathbf{b}, \mathbf{W})$, where, $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$ are the bias vectors for the visible and hidden layers respectively, and $\mathbf{W} \in \mathbb{R}^{m \times n}$ is the weight matrix that contains the weights on the edges between each pair of visible-hidden units.

Depending on whether the inputs to an RBM (i.e., the elements in $\mathbf{v}$) have binary or other types of values, it is called Binary RBM or other kinds of RBM (e.g., Gaussian-Bernoulli RBM for real values [11]). We take Binary RBM as the example to briefly explain how an RBM learns from inputs to decide the values for its parameters $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{W}$.

The standard learning algorithm for an RBM is called Contrastive Divergence (CD) as shown in Alg. 1. It is based on Gibbs Sampling, which involves iterative two-way value propagations between the visible and hidden layers (computing $\mathbf{h}$ from $\mathbf{v}$ and computing $\mathbf{v}$ from $\mathbf{h}$ at lines 8 and 9 in Alg. 1) of the RBM. The propagations are based on the following conditional probabilities calculations:

$$P(h_j = 1|\mathbf{v}) = \sigma(b_j + \mathbf{v}^T W_{(:,j)}) \quad (1)$$

$$P(v_i = 1|\mathbf{h}) = \sigma(a_i + W_{(i,:)}\mathbf{h}), \quad (2)$$

where, $\sigma(\cdot)$ is the sigmoid activation function. The activation function of the hidden units is

$$h_j = \begin{cases} 1 & if \;\; r < P(h_j = 1|\mathbf{v}) \\ 0 & otherwise \end{cases} \quad (3)$$

where, r is a random number between 0 and 1. The function for visible units is in the same form. Alg. 2 shows the steps for sampling the hidden units from the visible units.

For every input, CD starts a Markov Chain by assigning an input vector to the visible units and performs k steps of Gibbs Sampling (i.e., k times of two-way value propagations between the two layers). The corresponding algorithm is denoted as CD-k. As illustrated in Alg 1, it consists of a number of epochs. In each epoch, the algorithm goes through the inputs in batches. It is worth noting that it conducts k steps of Gibbs Sampling on each input, but updates the RBM parameters only at the end of a batch. So, in the processing of a batch, the weights remain unchanged.

There are some variations of CD (e.g., PCD) for RBM learning, but they all share a Gibbs Sampling-based learning process similar to CD. In all these algorithms, the repeated computations of the vector dot products (lines 8 and 9 in Alg 1) take most of the learning time. In the standard CD, for instance, in an E-epoch CD-k training of an RBM (m hidden units and n visible units) over I inputs, the vector dot products amount to $O(E \cdot I \cdot k \cdot m \cdot n)$ computations, which is the main reason for the long learning time of an RBM and RBM-based nets.

In the usage of an RBM for prediction, the computation is simpler. For a given input, it does one-time forward propagation, computing $\mathbf{h}$ from $\mathbf{v}$. The main part is again the dot products (between $\mathbf{v}$ and the weight matrix $\mathbf{W}$.)

III. Optimizing the CD Algorithm

As the previous section mentions, all the variants of RBM-based networks are centered around the two-level fully connected network shown in Figure 1. And most of the computations on the network are vector dot products between either the visible vectors and the weight vectors or between the hidden vectors and the weight vectors. Therefore, if we can speed up the calculations of these vector dot products, we can speed up all the RBM-based networks.

The way we tackle the problem is to find and avoid unnecessary computations in the calculations of the dot products. The first method, *bounds-based filtering*, is based on an observation that the dot products are used only at the condition check at line 4 in Algorithm 2. So the method tries to use some conservative bounds of the dot products for the check, and computes the dot products only when the check fails. The challenge there is how to attain the conservative bounds

Algorithm 1 original CD algorithm

1: **Input:** input dataset, number of inputs N, batch size N_b, number of training epochs N_e, number of gibbs sampling steps k, input vector dimension n, size of hidden layer m
2: **for** $e = 1$ **to** N_e **do**
3: **for** $batch = 1$ **to** N/N_b **do**
4: **for** $q = 1$ **to** N_b **do**
5: Let the q^{th} input be vector $\mathbf{v}_d$
6: sample $\mathbf{h}$ from $\mathbf{v}$
7: **for** $step = 1$ **to** k **do**
8: sample $\mathbf{v}$ from $\mathbf{h}$
9: sample $\mathbf{h}$ from $\mathbf{v}$
10: **end for**
11: update Δ_W, Δ_a and Δ_b
12: **end for**
13: update parameters $\theta = (\mathbf{a}, \mathbf{b}, W)$
14: **end for**
15: **end for**

Algorithm 2 sample $\mathbf{h}$ from $\mathbf{v}$

1: **Input:** visible unit vector $\mathbf{v}$, hidden unit vector $\mathbf{h}$, dimension n, size of hidden layer m
2: **for** $j = 1$ **to** m **do**
3: $P(h_j = 1|\mathbf{v}) = \sigma(b_j + \sum_{i=1}^{n} v_i W_{(i,j)})$
4: $h_j = rand() < P(h_j = 1|\mathbf{v})$
5: **end for**

efficiently. Section III-A will explain our solution built on Triangle Inequality.

The second method, *delta product*, complements the first method at a lower level. Rather than avoiding a dot product completely, it tries to avoid some unnecessary operations within the calculations of a dot product. It is based on an insight that frequently different vectors in RBM calculations have common values in some of their elements. So if two such vectors both need to multiply a common weight vector, we may reuse the calculation results on one to save some calculations on the other. Section III-B explains how to effectively materialize this idea in RBM algorithms.

A. Bounds-Based Filtering

The first optimization, *bounds-based filtering*, tries to replace the expensive dot product calculations with much cheaper bounds calculations whenever possible. It is based on the following insight.

Insight on Necessity: The key insight for this optimization is about the necessity of the dot products. From Algorithm 1 and 2, we can see that the purpose of calculating the dot products $\mathbf{v}^T W_{(:,j)}$ is for computing the activation probability $P(h_j = 1|\mathbf{v})$ (line 3 in Alg. 2), while the only use of that probability is to compare with a random number (line 4 in Alg. 2) to determine the value of h_j. So the actual information we need is the comparison result, rather than the exact value of the dot product. If somehow we can figure out the lower bounds and upper bounds of $P(h_j = 1|\mathbf{v})$, denoted as lb and ub, we may be able to set h_j's values without the dot products (r is the random number):

$$h_j = \begin{cases} 1 & if \;\; r < lb(P(h_j = 1|\mathbf{v})) \\ 0 & if \;\; r > ub(P(h_j = 1|\mathbf{v})). \end{cases} \quad (4)$$

The dot products would be needed only if r is between the two bounds.

Bounds-based Filtering through Triangle Inequality: Our *bounds-based filtering* is an optimization that translates that insight into speedups. The key challenge for leveraging that insight is how to attain reasonably tight bounds of the probabilities with much less time than the dot product calculation takes. After investigating a number of methods, we find the following Triangle Inequality based method most effective. It helps estimate the bounds of $\mathbf{v}^T W_{(:,j)}$. Because the sigmoid function is monotonically increasing and b_j is a constant, knowing the bounds of $\mathbf{v}^T W_{(:,j)}$ immediately leads to the bounds of the probability $P(h_j = 1|\mathbf{v})$:

$$lb(P(h_j = 1|\mathbf{v})) = \sigma(b_j + lb(\mathbf{v}^T W_{(:,j)}) \quad (5)$$

$$ub(P(h_j = 1|\mathbf{v})) = \sigma(b_j + ub(\mathbf{v}^T W_{(:,j)}). \quad (6)$$

Considering two vector $\mathbf{v}$ and $\mathbf{w}$, the Triangle Inequality based method (or TI-based method) is based on the following classical form of the dot product $\mathbf{v} \cdot \mathbf{w}$ calculation:

$$\mathbf{v} \cdot \mathbf{w} = \frac{1}{2}(|\mathbf{v}|^2 + |\mathbf{w}|^2 - d^2(\mathbf{v}, \mathbf{w})), \quad (7)$$

where, $|\mathbf{v}|$ and $|\mathbf{w}|$ are the lengths of the two vectors, and $d(\mathbf{v}, \mathbf{w})$ is the distance between $\mathbf{v}$ and $\mathbf{w}$.

Our method uses Triangle Inequality to help estimate the bounds of $d(\mathbf{v}, \mathbf{w})$. As illustrated in Figure 2, Triangle Inequality tells us that $|d(A, B) - d(A, C)| < d(B, C) < d(A, B) + d(A, C)$ (d for distance). So if we have a visible unit vector $\mathbf{v}$, a weight vector $\mathbf{w_j} = W_{(:,j)}$ corresponding to sampling hidden unit h_j, we may introduce another point $\mathbf{w_L}$—which we call a *landmark point*; with that, we can form a triangle as illustrated in Figure 3 and calculate the bounds of $d(\mathbf{v}, \mathbf{w_j})$ by using the distances $d(\mathbf{v}, \mathbf{w_L})$ and $d(\mathbf{w_L}, \mathbf{w_j})$. The bounds can then lead to the bounds of $\mathbf{v} \cdot \mathbf{w_j}$ and then the bounds of $P(h_j = 1|\mathbf{v})$.

For the TI-based method to work effectively, landmarks shall be close to the corresponding $\mathbf{w_j}$ points. Therefore, we randomly pick l weight vectors from the weight matrix $\mathbf{W}$ as the landmarks. Then, for each weight vector in $\mathbf{W}$, we identify the landmark that is closest to that weight vector and uses that landmark for the bounds estimations related with that weight vector. These operations are done at the beginning of each batch in the training process as the weight matrix gets updated only then. Empirically we find that $l = \sqrt{m}/2$ (m is the number of hidden units) works well.

Overhead analysis: At a first glance, the bound calculation requires two distances and may seem even more expensive than the direct calculation of $d(\mathbf{v}, \mathbf{w_j})$. The key is in reuses: (1) the distance $d(\mathbf{v}, \mathbf{w_L})$ can be reused across the calculations of

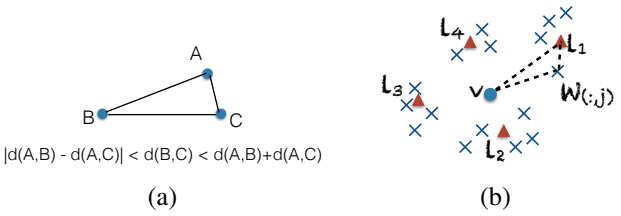


Fig. 2. (a) Illustrations of Triangle Inequality. (b) Four landmarks (L_1 to L_4) are introduced, which help construct triangles with visible and weight vectors.

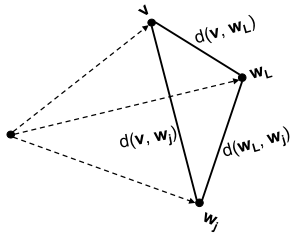


Fig. 3. Illustration of using TI to calculate the bound of dot product when sampling $\mathbf{h}$ from $\mathbf{v}$

many different $\mathbf{w_j}$ for a given $\mathbf{v}$; (2) because W gets updated only at the end of a batch processing as Algorithm 1 has shown, the distance $d(\mathbf{w_L}, \mathbf{w_j})$ can be reused across all the inputs $\mathbf{v}$ in a batch throughout all the Gibbs Sampling steps. Overall, the main overhead brought by this optimization comes from two parts: the calculation of the length of $\mathbf{v}$ and finding the closest landmark for each weight vector; the latter gives the length of $\mathbf{w_j}$ and the distance $d(\mathbf{w_L}, \mathbf{w_j})$ as side products. Therefore, when CD-k sampling is used, for l landmarks, the bounds computations for a batch containing B inputs require $m + m \cdot l + B \cdot k$ distances. Comparing to the original CD-K algorithm which requires $m \cdot B \cdot k$ distances, this overhead is small (as $B \cdot k$ is typically much larger than l.)

B. Delta Product

The second optimization complements the first one. Rather than avoiding an entire dot product calculation, it finds and avoids repeated computations in a dot product calculation.

Insight on Similarity: In either the visible layer or the hidden layer, we observe in the experiments that between any two consecutive Gibbs Sampling steps, differences exist only in a few units, which means only a few units flip from 0 to 1 or from 1 to 0 across Gibbs Sampling steps. For instance, on a dataset MNIST [12], the fractions of units that flip across epochs is around 40% at the beginning of a RBM training, and then quickly drops to 10-20% at later epochs. (More details in Section IV.)

We further observe that the large similarities even exist across different inputs. As the statistics in five public datasets which are shown in the second column of Table II (input similarity), on average, 62-93% visible units are the same between two connective input vectors (the default order of the inputs in the datasets was used, which appears to be random). Since the weight matrix W stays unchanged throughout a batch of processing, the computations on the non-flipped units are actually repeating their computations in the previous Gibbs Sampling step.

Optimization: The insight suggests that if we can effectively reuse the computations on the unchanged units, we may achieve some good speedups. We design *delta product* optimization to enable the reuse.

We again use the sampling of hidden units for explanation. Let $\mathbf{v}^q$ be the visible unit vector used to sample the hidden unit vector $\mathbf{h}^{q+1}$ during the $(q+1)^{th}$ Gibbs Sampling step. We use c_j^q to represent the input of the sigmoid function such that

$$c_j^q = b_j + (\mathbf{v}^q)^T W_{(:,j)} \tag{8}$$

Hence the formula for calculating the conditional probability of turning on the j^{th} hidden unit becomes $P(h_j^{q+1} = 1 | \mathbf{v^q}) = \sigma(c_j^q)$. Let $S_{0\to1}$ and $S_{1\to0}$ be the sets of visible units that change their states ($0 \to 1$ and $1 \to 0$) during the sampling of $\mathbf{v}^{q+1}$. For example, if $\mathbf{v}^q = \{0,1,1,0\}$ and $\mathbf{v}^{q+1} = \{0,0,1,1\}$, we construct the sets as $S_{0\to1} = \{v_4\}$ and $S_{1\to0} = \{v_2\}$. In the $(q+2)^{th}$ Gibbs Sampling step, when calculating the probability using $P(h_j^{q+2} = 1 | \mathbf{v^{q+1}}) = \sigma(c_j^{q+1})$, instead of computing $c_j^{q+1} = b_j + (\mathbf{v}^{q+1})^T W_{(:,j)}$, we calculate the following:

$$c_j^{q+1} = c_j^q + \sum_{v_t \in S_{0\to1}} w_{tj} - \sum_{v_t \in S_{1\to0}} w_{tj} \tag{9}$$

In this way, we reuse the result computed from the previous iteration and only calculate the changed parts instead of the full dot product $\sum_i v_i^{q+1} w_{ij}$. The overhead is to get the two sets $S_{0\to1}$ and $S_{1\to0}$, which is $O(n)$, while the original dot product takes $O(mn)$ operations.

For non-binary RBMs, the visible units are real values while the hidden units are binaries. Therefore, *bounds-based filtering* can be used for sampling the hidden units while *delta product* can be used for sampling the visible units.

IV. Evaluations

This section reports the efficacy of LCD for eliminating redundant computations and improving the speed of both the training and prediction of RBM.

A. Methodology

Learning Algorithms: Our experiments include both the standard learning algorithm CD, and the recently proposed variant PCD. For both algorithms, there are two main parameters: the learning rate and the number of Gibbs Sampling steps. Following the common practice in previous RBM studies [11], [15], [16], we set the learning rate to 0.01 for Binary RBMs, and 0.005 for Gaussian-Bernoulli RBMs in all experiments. The number of Gibbs Sampling steps (k) determines the numbers of samplings conducted consecutively on one input during the training process. Some earlier work used $k = 1$.

TABLE I
DATASETS AND NETWORK CONFIGURATIONS FOR BINARY RBM (BRBM) AND GAUSSIAN-BERNOULLI RBM (GRBM). (N: DATASET SIZE; N: # VISIBLE UNITS; T: # EPOCHS)

Type	Dataset	N	n	t
BRBM	MNIST [12] (binary)	50000	784	20
	f-MNIST [13] (binary)	50000	784	20
	CAL101 [14]	4100	784	300
	20Newsgroup [14]	8500	100	100
	micro-norb [15]	15000	1024	100
GRBM	MNIST [12]	50000	784	20
	f-MNIST [13]	50000	784	20
	CBCL [16]	2000	784	500
	Olivetti [11]	384	4096	300

Later studies [10], [15] have proved that using a larger k value (e.g., 5, or 10) helps improve the resulting RBM networks in both prediction accuracy and the convergence rates. To get to a prediction accuracy of 95.5% on MNIST dataset, for instance, using k= 10 takes 50% less time than using k=1 [10]. We hence include these two settings (k=5 and k=10) in our experiments.

Datasets: Our experiments use seven public datasets; column 2,3 of Table I list the datasets and their sizes. MNIST and f-MNIST are used for both Binary RBM and Gaussian-Bernoulli RBM. The pixel values of the raw images in these two datasets range from 0 to 255. They are converted to either binary numbers or real value numbers according to which type of RBM being used. Many previous studies [14], [15], [13], [16], [11] have shown successes in applying RBM networks on these datasets.

RBM Network Configurations: The number of visible units of RBM is determined by the size (i.e. numbers of pixels) of an input image, while the number of hidden units in the RBMs are adopted from some previous studies that have shown successful applications of RBM networks on the datasets. As our optimizations do not change the learning results of the original algorithms, we use a fixed number of epochs as the termination criterion when comparing the original with the optimized algorithms. The rightmost two columns in Table I list the number of visible units of the networks and the number of epochs we use in the experiments. We use 500 hidden units for all experiments.

Performance Measurement: For the performance measurements, we use a Geforce GTX 980 GPU (2048 CUDA cores, 4GB global memory) for training, and a tablet (Nexus 7 NVIDIA Tegra 3 T30L, Quad-Core Cortex-A9, 1GB memory)for prediction. All programs are written in C (and CUDA for GPU) and compiled with GCC and CUDA compilers. For time measurements, we repeat the measurements 20 times and report the average.

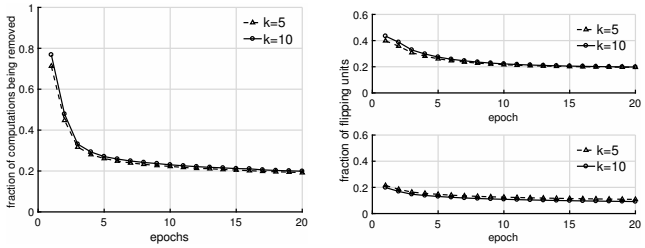
B. Improvements on Training

In this part, we first report the amount of computations of RBM training that are saved by *bounds-based filtering* and *delta product*, and then report the speedups on GPU.

Computation Savings: Table II reports the fractions of computations saved by the optimizations for the training of

TABLE II
INPUT SIMILARITY & FRACTIONS OF COMPUTATIONS SAVED BY THE OPTIMIZATIONS

Dataset	input similarity	bounds filter	delta product	LCD
MNIST	0.82	0.27	0.82	0.87
f-MNIST	0.82	0.84	0.90	0.98
CAL101	0.68	0.09	0.86	0.88
20Newsgroup	0.93	0.72	0.75	0.90
micro-norb	0.62	0.08	0.81	0.82



(a) (b)

Fig. 4. (a) Computations avoided by bounds-based filtering across epochs on MNIST dataset. (b) The fraction of flipped units (upper: visible; lower: hidden) across epochs on MNIST dataset.

Binary RBM (k=10). (Similar amounts of savings are observed when k=5.) The *bounds-based filtering* saves 8-84% computations while *delta product* saves 75-90%. Combining them (LCD) removes more (82-98%).

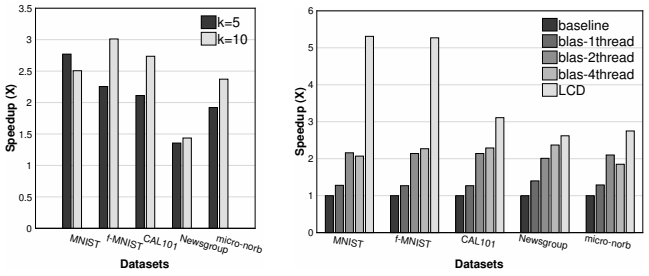
Bounds-based filtering is more effective in the earlier epochs. As illustrated in Figure 4(a), on MNIST dataset, it removes 77% computations at the first epoch. The removing rate decreases to around 20% in the last epoch. It is because the weights are initialized with small random values at the beginning of the training. So the calculated bounds are tight and can remove many computations. As the weights get larger and more discrete, the bounds become looser. Due to the nature of the sigmoid function (the "S" shape), the benefits of the optimization become smaller.

Delta product, on the other end, works better in the later epochs. If a unit changes its state between two consecutive sampling steps, we call it a flipping unit. We can tell from Figure 4(b) that the numbers of flipping visible and hidden units are about 20% to 40% at the beginning and then drop to 10% to 20% through the learning process on MNIST. Therefore, *delta product* saves a large portion of computations by leveraging the incremental changes of the units across Gibbs Sampling steps.

Our optimizations also reduce the computations of PCD by 67-90%, similar to the reductions on CD. The reduction of computations on Gaussian RBM are as much as 49-58%.

In the following, we will concentrate on the discussions on the Binary RBM trained by the standard CD algorithm.

Speedups: Figures 5 report the speedups of the GPU implementation of LCD. The baseline is a version we adapted from a public Caffe implementation [17] of the standard RBM on GPU, which uses cuBLAS matrix multiplication to efficiently process a batch of inputs.



(a) Training (b) Prediction

Fig. 5. (a) Speedups of RBM training on GTX980. The two bars of each dataset correspond to the speedups with k=5 and k=10. (b) Speedups of RBM usage for predictions on a mobile device. Both the `baseline` and the LCD versions use a single core.

As shown in Figures 5 (a), our optimization generally results in 2-3X speedups when $k = 5$ and $k = 10$. The speedup is also greater when $k = 10$ as opposed to $k = 5$. A higher value of k results in more Gibbs Sampling steps, which thus takes advantage of the *delta product* optimization more frequently. In general, the speedup for the GPU *delta product* implementation can be expected to increase as k increases, because the *delta product* operation is faster than a cuBLAS matrix multiplication kernel.

The 20Newsgroup dataset has less speedups than other datasets. This is due to the low number of visible nodes (100) for the 20Newsgroup dataset, which decreased the amount of computation that could be saved with the *delta product* implementation.

We note that the speedups are on double-precision implementations. Speedups on single-precision implementations are about half as large due to their less computations.

C. Improvements on Prediction

As Section II mentions, LCD can be applied to the usage of RMB for predictions. As the usage of RBM often happens on edge or mobile devices, we measure the performance of the prediction on a Nexus tablet. Single precision implementations are used.

We compare our LCD version with both the baseline version (`baseline`) and the ones using BLAS (a highly optimized linear algebra library for parallel and vector operations [18]). The tablet we use has 4 cores, so the BLAS versions could use up to 4 threads. Both the `baseline` and LCD versions use a single core. For the BLAS versions, we measure the performance using 1,2 and 4 threads.

As shown in Figures 5 (b), LCD outperforms both the baseline version and the BLAS versions on all the datasets. It brings up to 5.3X speedup over the `baseline` version. The BLAS version, for leveraging the parallelism in the processor, runs faster than the baseline version. But as it goes from two cores to four cores, no much improvement (even some slight slowdowns) shows up due to the contentions in the memory bandwidth and cache. Using only a single core, the LCD version runs much faster than the BLAS version regardless of how many cores it uses. It shows that LCD not only accelerates the RBM prediction time, but needs to consume much less energy.

V. Conclusion

This paper presents LCD, an optimized CD algorithm for speeding up RBM training and usage by detecting and avoiding unnecessary computations. It consists of two novel techniques, *bounds-based filtering* and *delta product*. The former avoids computing unnecessary dot products, and the latter avoids some repeated computations in a dot-product calculation. The results show that LCD speeds up the training of RBM on GPU by 2-3X, and the usage of RBM on a tablet by 2.6-5.3X. Given the fundamental role of RBM in deep belief networks and other deep learning tasks, these results indicate the promise of LCD for significantly enhancing the efficiency of a class of deep learning applications.

References

[1] G. Hinton., S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.

[2] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE TASLP*, vol. 20, no. 1, pp. 30–42, 2012.

[3] P. Bell, P. Swietojanski, and S. Renals, "Multitask learning of context-dependent targets in deep neural network acoustic models," *IEEE/ACM TASLP*, vol. 25, no. 2, pp. 238–247, 2017.

[4] R. Sarikaya, G. E. Hinton, and A. Deoras, "Application of deep belief networks for natural language understanding," *IEEE/ACM TASLP*, vol. 22, no. 4, pp. 778–784, 2014.

[5] W. H. Pinaya, A. Gadelha, O. M. Doyle, C. Noto, A. Zugman, Q. Cordeiro, A. P. Jackowski, R. A. Bressan, and J. R. Sato, "Using deep belief network modelling to characterize differences in brain morphometry in schizophrenia," *Scientific Reports*, vol. 6, 2016.

[6] R. Cao, D. Bhattacharya, J. Hou, and J. Cheng, "Deepqa: improving the estimation of single protein model quality with deep belief networks," *BMC bioinformatics*, vol. 17, no. 1, p. 495, 2016.

[7] F. Liu, C. Ren, H. Li, P. Zhou, X. Bo, and W. Shu, "De novo identification of replication-timing domains in the human genome by deep learning," *Bioinformatics*, p. btv643, 2015.

[8] G. Hinton, "A practical guide to training restricted boltzmann machines," Tech. Rep., 2010.

[9] S. Wang, R. Frostig, P. Liang, and C. D. Manning, "Relaxations for inference in restricted boltzmann machines," in *ICLR'14*, 2014.

[10] T. Tieleman, "Training restricted boltzmann machines using approximations to the likelihood gradient," in *Proceedings of ICML'08*. ACM, 2008, pp. 1064–1071.

[11] K. Cho, T. Raiko, and A. Ilin., "Gaussian-bernoulli deep boltzmann machine," in *Proceedings of IJCNN'13*, 2013, pp. 1–7.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[13] K. Cho, T. Raiko, and A. Ilin, "Enhanced gradient and adaptive learning rate for training restricted boltzmann machines," in *Proceedings of ICML'11*, 2011.

[14] B. M. Marlin, K. Swersky, B. Chen, and N. Freitas, "Inductive principles for restricted boltzmann machine learning," in *Proceedings of AISTATS'10*, 2010, pp. 509–516.

[15] T. Tieleman and G. Hinton, "Using fast weights to improve persistent contrastive divergence," in *Proceedings of ICML'09*. ACM, 2009, pp. 1033–1040.

[16] T. Yamashita, M. Tanaka, E. Yoshida, Y. Yamauchi, and H. Fujiyoshii, "To be bernoulli or to be gaussian, for a restricted boltzmann machine," in *ICPR'14*. IEEE, 2014, pp. 1520 – 1525.

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[18] L. S. B. et al., "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002.