Combining approximate inference methods for efficient learning on large computer clusters

Zhenwen Dai[1,2], **Jacquelyn A. Shelton**[1], **Jörg Bornschein**[1], **Abdul Saboor Sheikh**[1], **Jörg Lücke**[1,2]
[1] Frankfurt Institute for Advanced Studies,Germany
[2] Physics Dept., Goethe-University Frankfurt, Germany
{dai, shelton, bornschein, sheikh, luecke}@fias.uni-frankfurt.de

1 Introduction

An important challenge in machine learning is to develop learning algorithms that can handle large amounts of data at a realistically large scale. This entails not only the development of algorithms that can be efficiently trained to infer parameters of the model in a given dataset, but also demands careful thought about the tools (both software and hardware) used in their implementation.

Based on the previously developed framework of parallel Expectation Maximization (EM) learning [**?**], we extend it to different models with corresponding parallelization techniques. To further tackle problems of computational complexity and to utilize the capability of the parallel computing hardware (CPU/GPU clusters), we developed a set of techniques which can be catered to specific large-scale learning problems. For instance, we design a dynamic data repartition technique for "Gaussian sparse coding" (Sec. 3.2), use specialized GPU kernels for translation invariant learning (Sec. 3.3), and show how sampling can be used to further scale the learning on very high dimensional data (Sec. 3.4). We propose these as examples of a parallelization toolbox which can be creatively combined and exploited in model-task driven ways.

The framework is a lightweight and easy to use implementation of Python which facilitates the development of massive parallel machine learning algorithms using Message Passing Interface (MPI) for communication between the compute nodes. Once algorithms are integrated into the framework, they can be executed on large numbers of processor cores and can be applied to large sets of data. Some of the numerical experiments we performed ran on InfiniBand interconnected clusters and used up to 5000 parallel processor cores with more than 10^{17} floating point operations. For reasonably balanced meta-parameters (number of data points vs. number of latent variables vs. number of model parameters to be inferred), we observe close to linear runtime scaling behavior with respect to the number of cores in use.

2 Parallel EM Learning in Sparse Coding Models

We discuss parallelization of our algorithms for training Sparse Coding (SC) models on large data sets. Sparse Coding (SC) models assume that each observation $\vec{y} = (y_1, \dots, y_D)$ is associated to a (continuous or discrete) sparse latent variable $\vec{s} = (s_1, \dots, s_H)$, where sparsity means that most of the components s_h in $\vec{s}$ are zero. Each data point is generated according to the data model $p(\vec{y} \,|\, \Theta) = \sum_{\vec{s}} p(\vec{y} \,|\, \vec{s}, \Theta)\, p(\vec{s} \,|\, \Theta)$ where Θ denotes the model parameters. Typically, $p(\vec{y} \,|\, \vec{s}, \Theta)$ is modelled as a Gaussian with a mean $\vec{\mu}$ given by some interactions of basis vectors $W_h \in \mathbb{R}^D$ and the latent variable $\vec{s}$ (e.g. linear superposition $\vec{\mu} = \sum_h s_h W_h$ for SC models). Typical choices for $p(\vec{s} \,|\, \Theta)$ are Bernoulli, Laplace, or spike-and-slab distributions.

Such multi-causal data generation processes can be well formulated in terms of probabilistic generative models, which can be optimized in the framework of maximum likelihood learning (see, e.g., [**?**, **?**] for an overview). A practical approach to optimize the parameters Θ of a generative model given the data set $\mathcal{Y}$ is to maximize the data log-likelihood $\mathcal{L}(\Theta)$:

$$\Theta^* \quad = \quad \arg\max_{\Theta}\{\mathcal{L}(\Theta)\} \;\text{ with }\; \mathcal{L}(\Theta) = \log\big(p(y^{(1)},\ldots,y^{(N)}|\Theta)\big) = \sum_{n=1}^{N} \log p(y^{(n)}\,|\,\Theta), \quad (1)$$

$$\text{where} \quad p(y\,|\,\Theta) = \int_s p(y\,|\,s,\Theta)\,p(s\,|\,\Theta)ds. \quad (2)$$

Note that in order to maximize (**??**), we have to integrate over the entire space of s, or sum over it in the case of discrete variables s. The optimal parameters that maximize the data log-likelihood under the generative model can be sought by Expectation Maximization (EM) algorithm (see eg., [**?**]), which iteratively optimizes a lower bound $\mathcal{F}(\Theta,q)$ of the likelihood w.r.t. the parameters Θ and a distribution q:

$$\mathcal{L}(\Theta) \geq \mathcal{F}(\Theta, q_{\Theta'}) = \sum_{n=1}^{N}\sum_{s} q_n(s|\Theta')\log\frac{p(y^{(n)},s|\Theta)}{q_n(s|\Theta')}. \quad (3)$$

Each iteration consists of an E-step and an M-step. The E-step optimizes the lower bound w.r.t. to the distributions $q_n(s)$ by setting them equal to the posterior distributions $q_n(s) \leftarrow p(s^{(n)}|y^{(n)},\Theta)$ while keeping the parameters Θ fixed, denoted by Θ'. The M-step then optimizes $\mathcal{F}(\Theta,q_{\Theta'})$ w.r.t. the parameters Θ keeping the distributions $q_n(s)$ fixed. In general, the derived update equations in the M-step take the form:

$$\theta^{\text{new}} = \left(\sum_{n=1}^{N}\langle f(y^{(n)},s)\rangle_{q_n(s)}\right)\left(\sum_{n=1}^{N}\langle g(s)\rangle_{q_n(s)}\right)^{-1}, \quad (4)$$

where θ is some parameter to update, f and g are model dependent update functions, and $\langle\cdot\rangle_{q_n}$ are their expectation values w.r.t. the distribution q_n. In the above form, the computationally expensive part is calculating the expectation values, and this is done independently for each data point. With this observation, we can use shared-nothing parallelization. Specifically, we partition data points and evenly distribute them among the nodes/cores of the cluster. (This generic strategy has been further adapted in a model-driven way in Sec. 3.2.) At each iteration, each node/core computes the expected sufficient statistics locally, gathers the local results from all the other nodes with the "AllReduce" function of MPI and computes the new parameters. Note that the computations in the E-step for the expectation values in Eqn. (**??**) consume most of the computing time (parallelized and summed locally on each node), whereas and the computations of the parameter updates in the M-step are relatively inexpensive (synchronously updating the parameters over all the nodes).

3 Illustrative Models: Combining the tools

3.1 Expectation Truncation (ET): As a first step in scaling our models to high dimensions, we apply a truncated variational EM approach based on a preselection hidden spaces carrying most probability mass. In the context of probabilistic approaches, it has recently been shown that preselection of the most relevant latent variables can be formulated as a variational approximation to exact inference [**?**], referred to as expectation truncation, or ET. In ET, the posterior distribution $p(\vec{s}\,|\,\vec{y}^{(n)},\Theta)$ is approximated by distribution $q_n(\vec{s};\Theta)$ that only has support on a subset $\mathcal{K}_n \subset \{0,1\}^H$ of the state space:

$$p(\vec{s}\,|\,\vec{y}^{(n)},\Theta) \approx q_n(\vec{s};\Theta) = \frac{p(\vec{s}\,|\,\vec{y}^{(n)},\Theta)}{\sum\limits_{\vec{s}\,'\in\mathcal{K}_n} p(\vec{s}\,'\,|\,\vec{y}^{(n)},\Theta)}\,\delta(\vec{s}\in\mathcal{K}_n) = \frac{p(\vec{s},\vec{y}^{(n)}\,|\,\Theta)}{\sum\limits_{\vec{s}\,'\in\mathcal{K}_n} p(\vec{s}\,',\vec{y}^{(n)}\,|\,\Theta)}\,\delta(\vec{s}\in\mathcal{K}_n) \quad (5)$$

where $\delta(\vec{s}\in\mathcal{K}_n) = 1$ if $\vec{s}\in\mathcal{K}_n$ and zero otherwise. The subsets $\mathcal{K}_n$ are chosen in a data-driven way using a deterministic *selection function* and represent the preselected latent states $H' \leq H$. These sets vary per data point $\vec{y}^{(n)}$ and should contain most of the probability mass $p(\vec{s}\,|\,\vec{y})$ while at the same time being significantly smaller than the whole latent space. With such a $\mathcal{K}_n$, Eqn. **??** results in a good approximations to the posterior. Since for many applications the posterior mass is concentrated in small volumes of the state space, the approximation quality can stay high even for relatively small sets $\mathcal{K}_n$. This approximation can be used to compute efficiently the expectation values needed in the M-step (**??**):

$$\langle g(\vec{s})\rangle_{p(\vec{s}\,|\,\vec{y}^{(n)},\Theta)} \approx \langle g(\vec{s})\rangle_{q_n(\vec{s};\Theta)} = \frac{\sum_{\vec{s}\in\mathcal{K}_n} p(\vec{s},\vec{y}^{(n)}\,|\,\Theta)\,g(\vec{s})}{\sum_{\vec{s}\,'\in\mathcal{K}_n} p(\vec{s}\,',\vec{y}^{(n)}\,|\,\Theta)}\,. \quad (6)$$

Eqn. **??** represents a reduction in required computational resources as it involves only summations (or integrations) over the smaller state space $\mathcal{K}_n$. For sparse coding models, for instance, we can exploit that the posterior mass lie close to low-dimensional subspaces to define the sets $\mathcal{K}_n$ [**?**, **?**]. Such a subspace can be determined using a computationally inexpensive selection function at the beginning of each E-step. The selection functions compute those candidate hidden units s_h that are most likely to have contributed to a data point $\vec{y}^{\,(n)}$. Appropriate selection functions $\mathcal{S}_h(\vec{y},\Theta)$, e.g. for sparse coding models, can be realized as any efficiently computable function $f(\vec{y},\Theta)$ with a norm that correlates with the probabilities $p(s_h = 1 \,|\, \vec{y}^{\,(n)},\Theta)$ [**?**, **?**]; here a $\mathcal{S}_h(\vec{y}^{\,(n)})$ yielding a reasonable definition of $\mathcal{K}_n$ is:

$$\mathcal{S}_h(\vec{y}^{\,(n)}) = (\vec{W}_h^{\mathrm{T}} \,/\, ||\vec{W}_h||)\, \vec{y}^{\,(n)}, \qquad \text{with} \;\; ||\vec{W}_h|| = \sqrt{\textstyle\sum_{d=1}^{D} (W_{dh})^2}\,. \tag{7}$$

Other $\mathcal{S}_h(\vec{y},\Theta)$ are found via deterministic relations $\vec{s} = f(\vec{y},\Theta)$ in the limit of no data noise [**?**, **?**].

In the following examples, we avoid the exact evaluation of expectations values by using the approximation in Eqn. **??** and selection function in Eqn. **??**. Using this approximation scheme and the massive parallel implementation, we have been able to train and compare models with non-trivial priors and non-linear interactions with more than 1300 observed and up to 1600 latent variables [**?**]. These experiments were performed on up to 5000 CPU cores in parallel.

3.2 Dynamic data repartitioning: We have seen (in Sec. 3.1) that ET is an approximate learning approach that discriminatively selects the most relevant causes of an observation $\vec{y}$ in order to variationally compute the posterior. In a parallel setup, we can further benefit from the preselection step of ET to dynamically repartition and redistribute the data. At each E-step, we can cluster $\mathcal{Y}$ based on the most relevant causes chosen by the preselection step for each $\vec{y}^{\,(n)}$. The resulting clusters can then be distributed among nodes/cores to perform the next E-step. This approach not only pursues a natural partitioning of data, but in a parallel setting, it can prove to be more efficient than a uniform distribution of data. By maximizing the similarity among data points assigned to an individual processing unit, we can minimize across all the units redundant computations that are tied to specific states of the causes. For instance in a sparse coding model with a two-part (combining a continuous and discrete distribution) prior [**?**], the ET based posterior (**??**) takes the following form:

$$p(\vec{s},\vec{z}\,|\,\vec{y}^{\,(n)},\Theta) \;\;\approx\;\; \frac{\mathcal{N}(\vec{y}^{\,(n)};\vec{\mu}_{\vec{s}},C_{\vec{s}})\,\mathrm{Bernoulli}(\vec{s};\vec{\pi})\,\mathcal{N}(\vec{z};\,\vec{\kappa}_{\vec{s}}^{(n)},\Lambda_{\vec{s}})}{\sum_{\vec{s}\,'\in\mathcal{K}_n}\mathcal{N}(\vec{y}^{\,(n)};\vec{\mu}_{\vec{s}'},C_{\vec{s}'})\,\mathrm{Bernoulli}(\vec{s}';\vec{\pi})}\,\delta(\vec{s}\in\mathcal{K}_n). \tag{8}$$

Here the parameters $\vec{\mu}_{\vec{s}}$, $C_{\vec{s}}$ and $\Lambda_{\vec{s}}$ entirely depend on a state $\vec{s}$ of causes. Also, $\vec{\kappa}_{\vec{s}}^{(n)}$ takes prefactors that can be precomputed given $\vec{s}$. To compute (**??**), it turns out that our dynamic data redistribution strategy is more efficient than a static (and uniform) data distribution approach. This is illustrated in Fig. **??**, which shows empirical E-step speedup over the static data distribution strategy taken as a baseline. The error bars were generated by performing 15 trials per given data size N. For all the trials, model scale (i.e., data dimensionality) and ET parameters were kept constant[1]. Each trial was run in parallel on 24 computing nodes. The red plot in the figure also shows the speedup as a result of an intermediate approach. There we initially uniformly distributed the data samples which were then only locally clustered by each processing unit at every E-step. The blue plot on the other hand shows the speedup as a result of globally clustering and redistributing the data prior to an E-step. It's important to note that all reported results also take into account the cost of data clustering and repartitioning.

We optimize the data clustering process by having each processing unit cluster its own data locally and then merging the resulting clusters globally. To avoid unfair workload distribution, we also bound the maximum cluster size. Currently we pick (per iteration) top α percentile of cluster sizes as the threshold. Any cluster larger than α is evenly broken into smaller clusters of maximum α size[2]. Moreover, to minimize communication overhead among computational units, we only recluster and distribute indices of the datapoints. This entails that the actual data must reside in a shared memory structure which is efficiently and dynamically accessable by all the computational units. Otherwise, all the units require their own copy of the whole dataset.

[1]The observed and the latent dimensions of the GSC model [**?**] were 25 and 20 respectively. The ET parameters H' and γ (maximum number of active causes in a given latent state) were 8 and 5 respectively.

[2]The α for the reported experiments was 5.

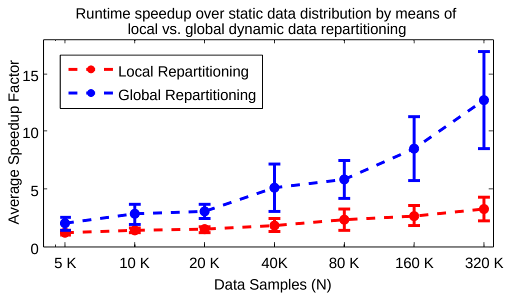

Figure 1: E-step runtime speedup (of the ET version of the GSC model [?]) over the static data distribution strategy taken as a baseline. The red plot shows the speedup when initially uniformly distributed data samples were only clustered locally by each processing unit, while the blue plot shows the speedup as a result of globally clustering and redistributing the data. The runtimes include the time taken by clustering and repartitioning modules

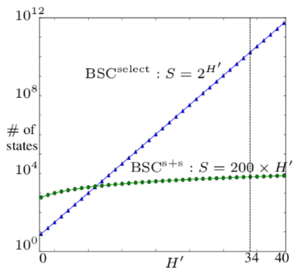

Figure 2: Complexity in terms of states the algorithm needs to consider (log-scale) for various values of H', corresponding to experiments of sparse coding models using ET or ET+sampling.

It is important to note that although we have illustrated the gains of dynamic data repartitioning using a specific sparse coding model which typically involves state-dependent computationally expensive operations, the technique itself is inherently generic as it is based on a variational EM approach which can be applied to a larger range of models and problems (see [?] Sec. 4).

3.3 Hybrid Parallelization with GPUs: As shown, with data point partitioning and ET, learning can be done efficiently at a large scale. Beyond this, with suitable specialized hardware the speed can be further increased. To demonstrate the performance gain, let us look at a computationally more demanding task. In the previously mentioned models, no transformations of causes are considered. In realistic data however, especially visual data, visual objects (causes) vary in positions, scales and rotations within an image set. To learn causes under various translations without supervision, we add another hidden variable X which defines the positions of causes selected by $\vec{s}$ (only positions are considered here). It dramatically increases the size of the joint space of hidden variables $(\vec{s}, X)$.

CPU vs. GPU: The computation of the expected sufficient statistics is an ideal situation for data parallelism: each configuration of $(\vec{s}, X)$ within $\mathcal{K}_n$ is evaluated independently and summed together. This parallelism scheme demands high memory bandwidth, while the memory structure of this CPU is inefficient in this situation as it is designed for low-latency data access instead of high bandwidth. We use GPU computation because the architecture of its memory system is structured to deliver high bandwidth [?]. As a comparison, the memory bandwidth of Nvidia GTX480 is 177.4 GB/s, but DDR3-1333 in dual-channel mode only provides a peak bandwidth of 21.3 GB/s. Beyond the simple comparison of peak bandwidth, the large number of hardware threads on GPU can better tolerate memory latency, making peak performance easier to reach. Furthermore, the GPU's hardware implemented $\exp/\log$ instructions are helpful because probability calculations require a large amount of these. The Nivida GTX480 can handle 60 $\exp/\log$ operations per clock cycle with 32-bit floating-point precision [?], whereas a CPU usually takes hundreds of clock cycles per such an operation.

BLAS vs. specialized kernels: Both Nvidia and AMD offer their Basic Linear Algebra Subprograms (BLAS) library to help programming with their GPUs. If an algorithm can be perfectly formulated as matrix operations, it would be very straightforward to program with BLAS. However, in our algorithm in order to formulate an algorithm into matrix operations, parameters of each cause need to be aligned with data points for every configuration of $(\vec{s}, X)$ within $\mathcal{K}_n$. These alignments are implemented by matrix shifting, which is a very inefficient memory operation. Instead, we wrote specialized GPU kernels in which matrix shifting is avoided by manipulating the indices of matrices.

Overall, we divide the data points according to the number of GPU cards (18 GTX480 on 5 different nodes) and assign every graphics card a dedicated CPU process. Sufficient statistic expectations are replaced with computation by a specialized GPU kernel, and controlled by CPU-GPU synchronization via PyOpenCL. With different data sets, we observe about 10-20 times speed up comparing with only using CPUs in terms of total computing time. Such two-layer parallel computational structure (MPI-GPU) is convenient and generally applicable to many learning algorithms where the computational intensive parts can be replaced by GPU kernels.

3.4 Sampling: Although ET (Sec. 3.1) is a deterministic approach very different than the stochastic nature of sampling, their formulations as approximations to expectation values (**??**) allow for a straight-forward combination of both approaches to speed up the expensive calculations necessary for inference in high dimensions [**?**] – an observation from which all of the above models discussed could benefit. Specifically, given a data point, $\vec{y}^{(n)}$, we first approximate the expectation value (**??**) using the variational distribution $q_n(\vec{s};\Theta)$ as defined by preselection (**??**). Second, we approximate the expectations w.r.t. $q_n(\vec{s};\Theta)$ using sampling. The combined approach is thus given by:

$$\langle g(\vec{s})\rangle_{p(\vec{s}\,|\,\vec{y}^{(n)},\Theta)} \approx \langle g(\vec{s})\rangle_{q_n(\vec{s};\Theta)} \approx \tfrac{1}{M}\sum_{m=1}^{M} g(\vec{s}^{(m)}) \quad \text{with} \quad \vec{s}^{(m)} \sim q_n(\vec{s};\Theta), \tag{9}$$

where $\vec{s}^{(1)}$ to $\vec{s}^{(M)}$ denote samples from the truncated distribution q_n. Instead of drawing from a distribution over the entire state space, approximation (**??**) requires only samples from a potentially very small subspace $\mathcal{K}_n$. In the subspace $\mathcal{K}_n$, most of the original probability mass is concentrated in a smaller volume, thus MCMC algorithms perform more efficiently; results in a smaller space to explore, shorter burn-in times, and reduces the number of samples necessary. Compared to use of ET in inference alone, combining it with sampling allows for an increase in efficiency as soon as the number of samples required for a good approximation is less than the number of states in $\mathcal{K}_n$. For example, in the case of sparse coding models, the number of computations necessary to compute the expectation values in (**??**) is

$$\mathcal{O}\big(NS(\underbrace{D}_{p(\vec{s},\vec{y})} + \underbrace{1}_{\langle\vec{s}\rangle} + \underbrace{H'}_{\langle\vec{s}\vec{s}^T\rangle})\big) \tag{10}$$

where S either, (for models with ET alone, Sec. 3.1) denotes the number of hidden states or (for models with both ET and sampling) denotes the number of samples, that contribute to the calculation of the expectation values. Specifically, in experiments with high-dimensional data ($N = 500,000$ on natural image patches, observed data $D = 40 \times 40 = 1,600$ pixels, and hidden dimensions $H = 1,600$, with $H' = 34$ and $S = 200$) we see major improvement in the scalability of H' over use of ET alone, as shown in Fig. **??** (see [**?**]). This combination of approaches suggests strong potential for scaling of other models to learn on large dimensional data, e.g. in the model discussed in Sec. 3.2.

4 Discussion

As we illustrated with the above examples, there are already many tools available which we can use in various combinations in order to accomplish learning at a large scale. We just need to carefully evaluate the task and goals of a given algorithm, and as an additional model selection step, creatively select the combinations of tools at hand to address i.e. problems of memory usage, large dimensional data, and large data sets.