DISTRIBUTED OPTIMIZATION OF BINARY AUTOENCODERS USING AUXILIARY COORDINATES

Miguel Á. Carreira-Perpiñán and **Mehdi Alizadeh**, EECS, UC Merced

BayLearn
Bay Area Machine Learning Symposium

1 Abstract

Many powerful machine learning systems are based on the composition of multiple processing layers, such as deep nets, which gives rise to nonconvex objective functions. A general, recent approach to optimize such "nested" functions is the method of auxiliary coordinates (MAC) (Carreira-Perpiñán & Wang, 2014), This decomposes the optimization into steps that alternate between training single layers and updating the coordinates. It has the advantage that it reuses existing single-layer algorithms, introduces parallelism, and does not need to use chain-rule gradients, so it works with nondifferentiable layers. With large-scale problems, or when distributing the computation is necessary for faster training, the dataset may not fit in a single machine. It is then essential to limit the amount of communication between machines so it does not obliterate the benefit of parallelism. We describe a general way to achieve this, ParMAC, and illustrate it in learning binary autoencoders for fast information retrieval.

Funded by NSF award IIS–1423515 and a Google Faculty Research Award.

2 MAC for binary autoencoders

Although MAC (and ParMAC) apply more generally to multiple layers, we focus on the case of the composition of two functions, in particular the binary autoencoder (BA) (Carreira-Perpiñán & Raziperchikolaei, 2015):

$$\mathbf{X}_{D\times N}=(\mathbf{x}_1,\dots,\mathbf{x}_N):\ E_{\mathrm{BA}}(\mathbf{h},\mathbf{f})=\sum_{n=1}^{N}\|\mathbf{x}_n-\mathbf{f}(\mathbf{h}(\mathbf{x}_n))\|^2 \begin{cases}\text{encoder } \mathbf{h}\text{: } \mathbb{R}^D\to\{0,1\}^L\\ \text{decoder } \mathbf{f}\text{: } \{0,1\}^L\to\mathbb{R}^D.\end{cases}$$

- Like a usual autoencoder but with a binary code layer.
- Nonsmooth, nonconvex problem; chain-rule gradients inapplicable.

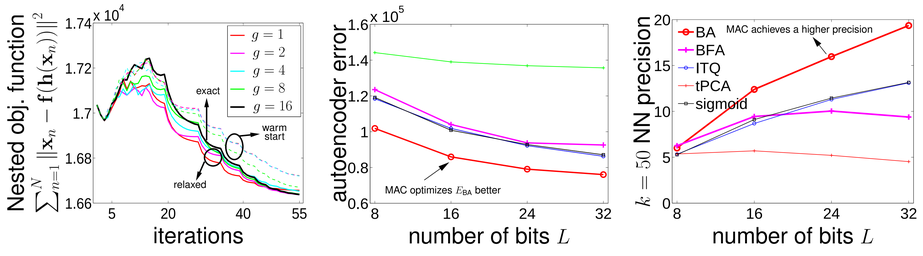
MAC introduces an auxiliary coordinate $\mathbf{z}_n=\mathbf{h}(\mathbf{x}_n)\in\{0,1\}^L$ for each data point, corresponding to its L-bit code vector (in the bottleneck layer), and uses a penalty method to optimize the constrained problem. This results in optimizing

$$E_Q(\mathbf{h},\mathbf{f},\mathbf{Z};\mu)=\sum_{n=1}^{N}\left(\|\mathbf{x}_n-\mathbf{f}(\mathbf{z}_n)\|^2+\mu\,\|\mathbf{z}_n-\mathbf{h}(\mathbf{x}_n)\|^2\right)\quad\text{s.t.}\quad\mathbf{z}_1,\dots,\mathbf{z}_N\in\{0,1\}^L$$

and driving the penalty parameter $\mu\to\infty$. We alternate two steps over $\mathbf{Z}$ and $(\mathbf{h},\mathbf{f})$:

- M step ("model" or "minimization") over $(\mathbf{h},\mathbf{f})$ for fixed $\mathbf{Z}$: $L+D$ independent problems (L single-bit hash functions, D decoders). With linear $\mathbf{h}$, $\mathbf{f}$ this simply involves fitting L SVMs to $(\mathbf{X},\mathbf{Z})$ and D linear regressors to $(\mathbf{Z},\mathbf{X})$.
- C step ("coordination") over $\mathbf{Z}$ for fixed $(\mathbf{h},\mathbf{f})$: this separates over the N codes $\mathbf{z}_1,\dots,\mathbf{z}_N$. Solved by enumeration if L is small or by alternating optimization.

Training BAs with MAC beats approximate approaches such as relaxing the codes or the step function in the encoder. It yields state-of-the-art binary hash functions $\mathbf{h}$, which can be used for fast approximate nearest-neighbor search using Hamming distances (e.g. for image retrieval).



3 ParMAC for binary autoencoders

MAC introduces embarrasing parallelism within each step: $L+D$ independent models in the M step and N independent codes in the C step. We now show how to turn this into a distributed, low-communication **ParMAC** algorithm.
With large datasets in distributed systems, it is imperative to minimize data movement over the network because the communication time generally far exceeds the computation time in modern architectures. In MAC we have 3 types of data: the original training data $\mathbf{X}$, the auxiliary coordinates $\mathbf{Z}$, and the model parameters. Usually, the latter type is far smaller. In ParMAC, we never communicate training or coordinate data; each machine keeps a disjoint portion of $(\mathbf{X},\mathbf{Z})$ corresponding to a subset of the points. Only model parameters are communicated (for BAs, the hash functions $\mathbf{h}$ and decoders $\mathbf{f}$).

Using P machines, ParMAC iterates as follows:

- M step: the hash functions $\mathbf{w}_l$ and decoders $\mathbf{f}_d$ visit each machine. This implies we train them with stochastic gradient descent, where one "epoch" for model $\mathbf{w}_l$ corresponds to $\mathbf{w}_l$ having visited all P machines (within each machine, data are also split into minibatches).
- C step: identical to MAC, each point's coordinates $\mathbf{z}_n$ are optimized independently, in parallel over machines (since each machine contains $\mathbf{x}_n$, $\mathbf{z}_n$, and all the model parameters).

How does the communication occur?

- C step: no communication at all.
- M step: all models $\mathbf{w}_l$ are communicated asynchronously in parallel.
  - With e epochs, the entire model parameters are communicated $e+1$ times. The last round of communication is needed to ensure each machine has the most updated version of the model for the C step.
  - This communication can be implemented with a circular topology.
  - This introduces no bottlenecks, unlike the use of parameter servers that gather parameters from the workers, update them, and broadcast them back to the workers.

Convergence guarantees (to a local optimum): since the only approximation to the original MAC algorithm is incurred by using SGD in the M step, and we can guarantee convergence of SGD under certain conditions, we recover the original convergence guarantees for MAC.

ParMAC has additional advantages:

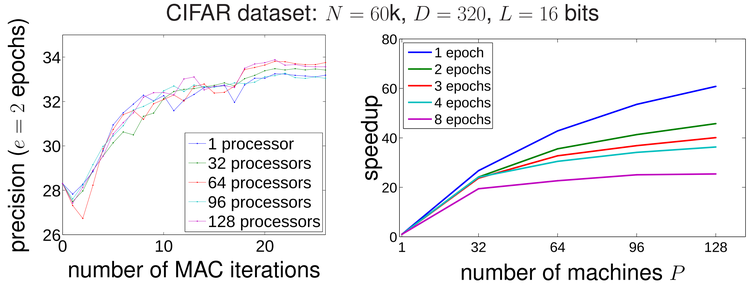
- Data shuffling in each epoch is easy by randomizing the sequence in which machines are visited for each model (the circular topology).
- Load balancing can be controlled in advance, by loading more data in faster machines, or on the fly, by having faster machines run more iterations.
- Fault tolerance is easy by replacing a faulty machine with a new one where the corresponding $\mathbf{X}$ are loaded.
- Streaming can be done by adding, removing or replacing machines and their data on the fly.

In general with multilayer models, e.g. deep nets, ParMAC locks a portion of the input, output and coordinate data $(\mathbf{X},\mathbf{Y},\mathbf{Z})$ inside each machine, and each hidden unit weight vector is an independent model that is communicated in the M step. Each weight vector may only depend on a subset of the dimensions of $(\mathbf{X},\mathbf{Y},\mathbf{Z})$, and each dimension of $\mathbf{z}_n$ may only depend on a subset of the weights.

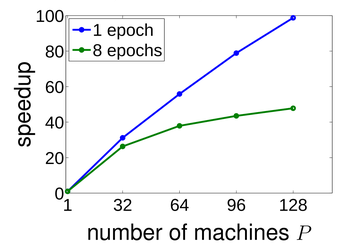
4 Experiments on a distributed cluster

We have trained BAs with ParMAC (implemented in C with MPI) in both distributed machines (UCSD Triton Shared Computer Cluster, with 128 nodes having 4 GB RAM per core) and shared-memory machines (with 32 cores and a total of 256 GB of RAM), achieving near-linear speedups even with small datasets. We are able to train BAs with linear and kernel hash functions on the SIFT1B dataset (`http://corpus-texmex.irisa.fr`), containing $N=1$ billion images represented as $D=128$ SIFT features, using $L=64$ bits and $P=128$ machines, in around a day. This would take months in a single machine with enough RAM to hold the data and parameters (which would require several TB).

CIFAR dataset: $N=60$k, $D=320$, $L=16$ bits



SIFT1M dataset:
$N=1$M, $D=128$, $L=16$ bits



SIFT1B dataset:
$N=100$M, $D=128$, $L=64$ bits

Hash function	Recall @R=100	Time (h) distrib.	Time (h) shared
linear SVM	61.5%	29.30	11.04
kernel SVM	66.1%	83.44	32.19