The 30000' Summary and 3 Cool Results

John Langford (Yahoo!) with Ron Bekkerman(Linkedin) and
Misha Bilenko(Microsoft)

`http://hunch.net/~large_scale_survey`

This hour's outline

1. A summary of results
2. Cool uses of GPUs
3. Terascale linear

A comparison is virtually impossible

Prediction performance varies wildly depending on the problem–method pair.

A comparison is virtually impossible

Prediction performance varies wildly depending on the problem–method pair.

But we can try: use Input complexity/time.

A comparison is virtually impossible

Prediction performance varies wildly depending on the problem–method pair.

But we can try: use Input complexity/time.

⇒ No credit for creating complexity then reducing it. (Ouch!)

A comparison is virtually impossible

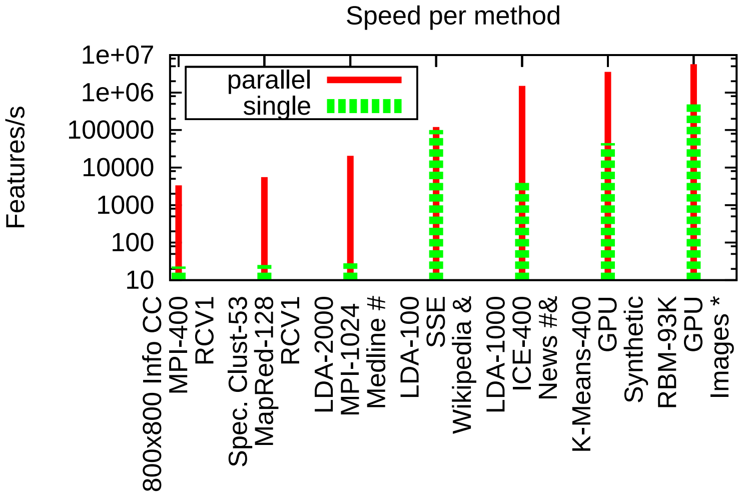
Prediction performance varies wildly depending on the problem–method pair.

But we can try: use Input complexity/time.

⇒ No credit for creating complexity then reducing it. (Ouch!)

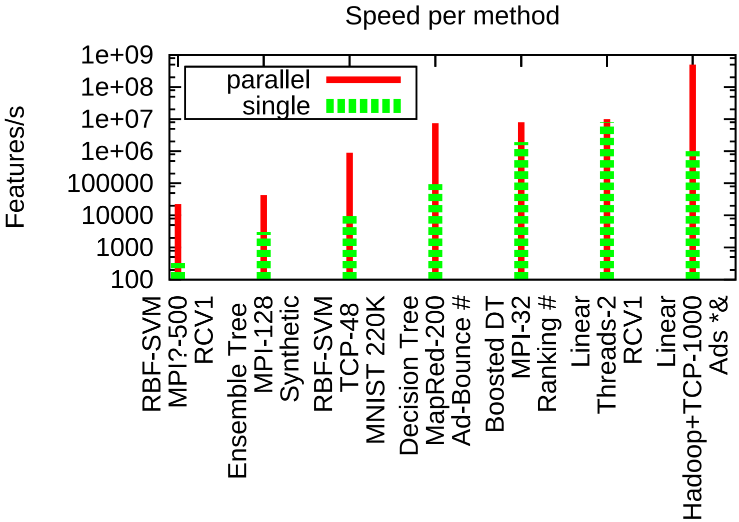
Most interesting results reported. Some cases require creative best-effort summary.

Unsupervised Learning



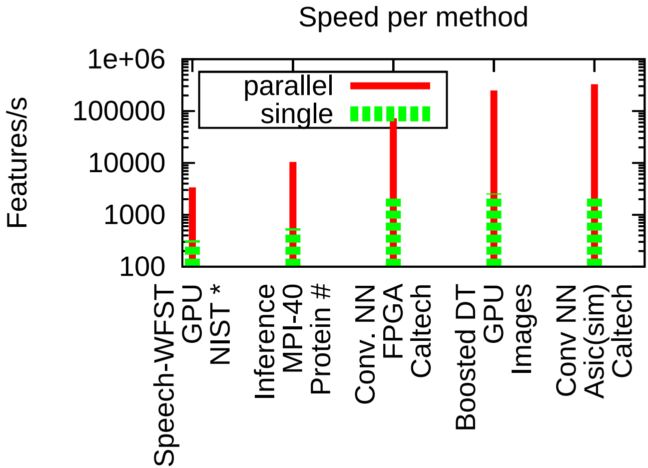
= Prev. * = Next. & = New

Supervised Training



= Prev. * = Next. & = New

Supervised Testing (but not training)



= Prev. * = Next.

My Flow Chart for Learning Optimization

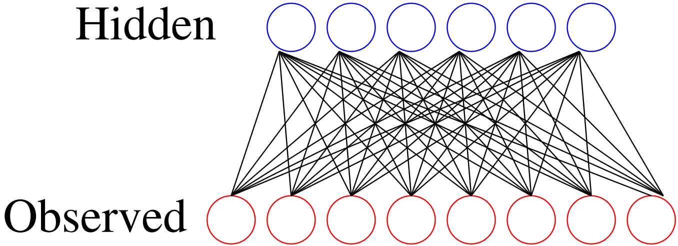
1. Choose an efficient effective algorithm
2. Use compact binary representations.
3. If (Computationally Constrained)
4. then GPU
5. else
    1. If few learning steps
    2. then ~~Map-Reduce~~ AllReduce
    3. else Research Problem.

This hour's outline

1. A summary of results
2. Cool uses of GPUs
   1. RBM learning
   2. Speech Recognition
3. Terascale linear

Restricted Boltzmann Machine learning CRN11 Chapter 18

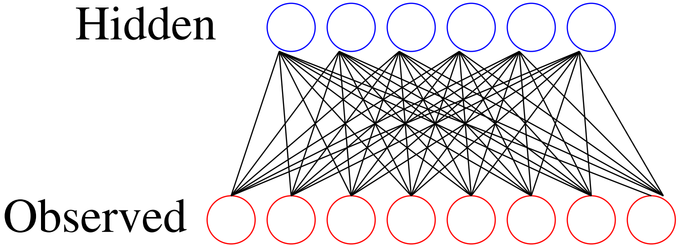
Goal: Learn weights which predict hidden state given features that can predict features given hidden state *



(*) Lots of extra details here.

Restricted Boltzmann Machine learning CRN11 Chapter 18

Goal: Learn weights which predict hidden state given features that can predict features given hidden state *



1. Number of parameters = hidden*observed = quadratic pain
2. An observed useful method for creating relevant features for supervised learning.

(*) Lots of extra details here.

RBM parallelization

GPU = hundreds of weak processors doing vector operations on shared memory.

1. Activation levels of hidden node i is $\text{sig}(\sum_j w_{ij} x_j)$.

RBM parallelization

GPU = hundreds of weak processors doing vector operations on shared memory.

1. Activation levels of hidden node i is $\mathrm{sig}(\sum_j w_{ij} x_j)$.
2. Given activation levels, hidden nodes are independently randomly rounded to $\{0, 1\}$.

RBM parallelization

GPU $=$ hundreds of weak processors doing vector operations on shared memory.

1. Activation levels of hidden node i is $\mathrm{sig}(\sum_j w_{ij} x_j)$.
2. Given activation levels, hidden nodes are independently randomly rounded to $\{0, 1\}$.
3. Predict features given hidden units just as step 1.

RBM parallelization

GPU $=$ hundreds of weak processors doing vector operations on shared memory.

1. Activation levels of hidden node i is $\text{sig}(\sum_j w_{ij} x_j)$.
2. Given activation levels, hidden nodes are independently randomly rounded to $\{0, 1\}$.
3. Predict features given hidden units just as step 1.
4. Shift weights to make reconstruction more accurate.

RBM parallelization

GPU = hundreds of weak processors doing vector operations on shared memory.

1. Activation levels of hidden node i is $\text{sig}(\sum_j w_{ij}x_j)$. A GPU is perfectly designed for a dense matrix/vector dot product.
2. Given activation levels, hidden nodes are independently randomly rounded to $\{0, 1\}$. Good for GPUs
3. Predict features given hidden units just as step 1. Perfect for GPUs
4. Shift weights to make reconstruction more accurate. Perfect for GPUs

Parallelization Techniques

1. Store model in GPU memory and stream data.
2. Use existing GPU-optimized matrix operation code.
3. Use multicore GPU parallelism for the rest.

This is a best-case situation for GPUs. x10 to x55 speedups observed.

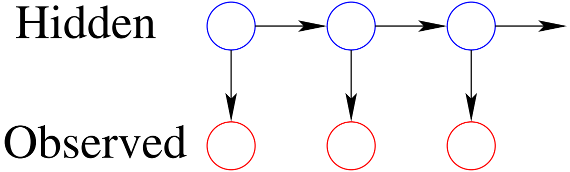
Parallelization Techniques

1. Store model in GPU memory and stream data.
2. Use existing GPU-optimized matrix operation code.
3. Use multicore GPU parallelism for the rest.

This is a best-case situation for GPUs. x10 to x55 speedups observed.
But, maybe we just sped up a slow algorithm?

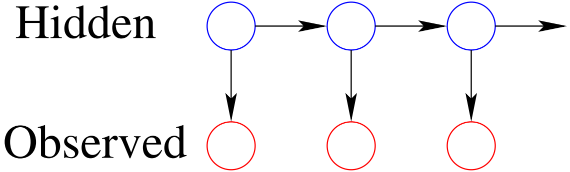
GPUs for Speech Recognition CGYK11 Chapter 21

Given observed utterances, we want to reconstruct the original (hidden) sequence of words via an HMM structure.



GPUs for Speech Recognition CGYK11 Chapter 21

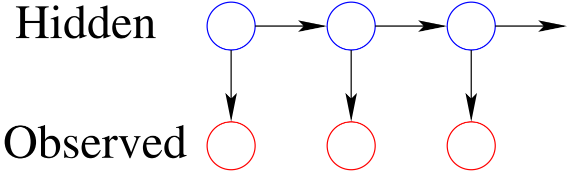
Given observed utterances, we want to reconstruct the original (hidden) sequence of words via an HMM structure.



Standard method of decoding: forward-backward algorithm using Bayes law to find the most probable utterance.

GPUs for Speech Recognition CGYK11 Chapter 21

Given observed utterances, we want to reconstruct the original (hidden) sequence of words via an HMM structure.

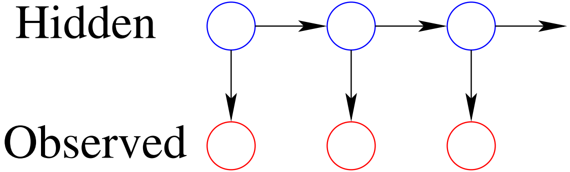


Standard method of decoding: forward-backward algorithm using Bayes law to find the most probable utterance.

Naively, this is trivially parallelized just as before.

GPUs for Speech Recognition CGYK11 Chapter 21

Given observed utterances, we want to reconstruct the original (hidden) sequence of words via an HMM structure.



Standard method of decoding: forward-backward algorithm using Bayes law to find the most probable utterance.

Naively, this is trivially parallelized just as before. But it's not.

1. The observation is non-binary. The standard approach matches the observed sound with one of very many different recorded sounds via nearest neighbor search.
2. The state transitions are commonly beam searched rather than using Bayesian integration.
3. The entire structure is compiled into a weighted finite state transducer, which is what's really optimized.

The approach used

Start with a careful systematic analysis of where parallelization might help.

The approach used

Start with a careful systematic analysis of where parallelization might help.

1. SIMD instructions: Use carefully arranged datastructures so single-instruction-multiple-data works.
2. Multicore over 30 cores of GPU.
3. Use Atomic instructions (Atomic max, Atomic swap) = thread safe primitives.
4. Stick model in GPU memory, using GPU memory as (essentially) a monstrous cache.

The approach used

Start with a careful systematic analysis of where parallelization might help.

1. SIMD instructions: Use carefully arranged datastructures so single-instruction-multiple-data works.
2. Multicore over 30 cores of GPU.
3. Use Atomic instructions (Atomic max, Atomic swap) = thread safe primitives.
4. Stick model in GPU memory, using GPU memory as (essentially) a monstrous cache.

Result: x10.5 speedup. Crucially, this makes the algorithm faster than real time.
GPUs help, even for highly optimized algorithms.

This hour's outline

1. A summary of results
2. Cool uses of GPUs
    1. RBM learning
    2. Speech Recognition
3. Terascale linear

Terascale Linear Learning ACDL11

Given 2.1 Terafeatures of data, how can you learn a linear predictor $f_w(x) = \sum_i w_i x_i$?

Terascale Linear Learning ACDL11

Given 2.1 Terafeatures of data, how can you learn a linear predictor $f_w(x) = \sum_i w_i x_i$?

1. No single machine algorithm.

Terascale Linear Learning ACDL11

Given 2.1 Terafeatures of data, how can you learn a linear predictor $f_w(x) = \sum_i w_i x_i$?

1. No single machine algorithm.
2. No multimachine algorithm requiring bandwidth $\propto$ Tbytes for any single machine.

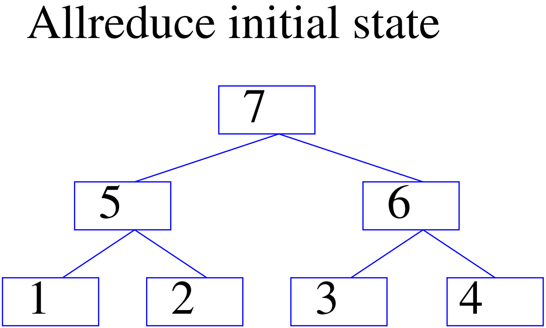
Terascale Linear Learning ACDL11

Given 2.1 Terafeatures of data, how can you learn a linear predictor $f_w(x) = \sum_i w_i x_i$?

1. No single machine algorithm.
2. No multimachine algorithm requiring bandwidth $\propto$ Tbytes for any single machine.

It is necessary but not sufficient to have an efficient communication mechanism.

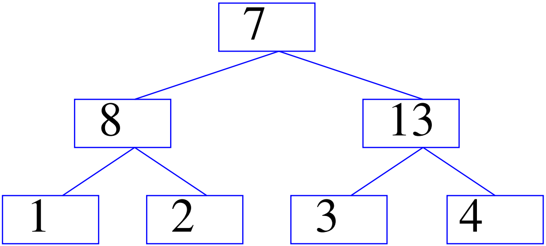
MPI-style AllReduce



AllReduce = Reduce+Broadcast

MPI-style AllReduce

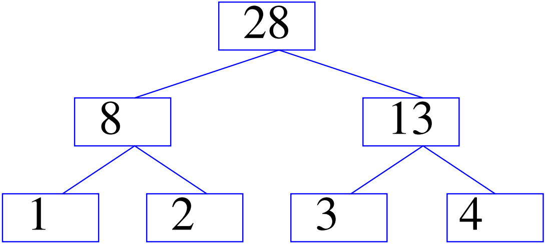
Reducing, step 1



AllReduce = Reduce+Broadcast

MPI-style AllReduce

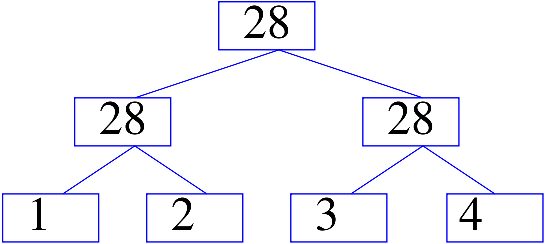
Reducing, step 2



AllReduce = Reduce+Broadcast

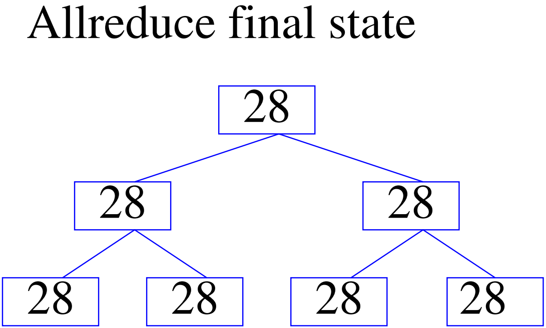
MPI-style AllReduce

Broadcast, step 1



AllReduce = Reduce+Broadcast

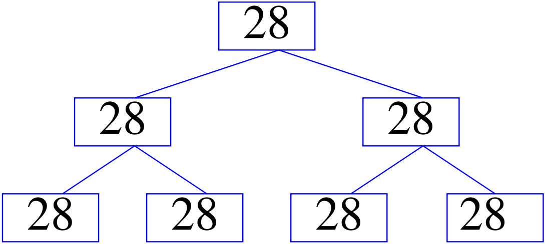
MPI-style AllReduce



AllReduce = Reduce+Broadcast

MPI-style AllReduce

Allreduce final state



AllReduce = Reduce+Broadcast

Properties:

1. Easily pipelined so no latency concerns.
2. Bandwidth $\leq 6n$.
3. No need to rewrite code!

An Example Algorithm: Weight averaging

```
$n =$ AllReduce(1)
While (pass number $<$ max)
  1. While (examples left)
       1. Do online update.
  2. AllReduce(weights)
  3. For each weight $w \leftarrow w/n$
```

An Example Algorithm: Weight averaging

```
$n =$ AllReduce(1)
While (pass number $<$ max)
  1. While (examples left)
       1. Do online update.
  2. AllReduce(weights)
  3. For each weight $w \leftarrow w/n$
```

Other algorithms implemented:

1. Nonuniform averaging for online learning
2. Conjugate Gradient
3. LBFGS

Approach Used: Preliminaries

Optimize so few data passes required.
Basic problem with gradient descent = confused units.
$f_w(x) = \sum_i w_i x_i$
$\Rightarrow \frac{\partial (f_w(x)-y)^2}{\partial w_i} = 2(f_w(x) - y)x_i$ which has units of i.
But w_i naturally has units of $1/i$ since doubling x_i implies halving w_i to get the same prediction.
Crude fixes:

1. Newton: Multiply inverse Hessian: $\frac{\partial^2}{\partial w_i \partial w_j}^{-1}$ by gradient to get update direction.
2. Normalize update so total step size is controlled.

Approach Used: Preliminaries

Optimize so few data passes required.
Basic problem with gradient descent = confused units.
$f_w(x) = \sum_i w_i x_i$
$\Rightarrow \frac{\partial (f_w(x)-y)^2}{\partial w_i} = 2(f_w(x) - y)x_i$ which has units of i.
But w_i naturally has units of $1/i$ since doubling x_i implies halving w_i to get the same prediction.
Crude fixes:

1. Newton: Multiply inverse Hessian: $\frac{\partial^2}{\partial w_i \partial w_j}^{-1}$ by gradient to get update direction...but computational complexity kills you.
2. Normalize update so total step size is controlled...but this just works globally rather than per dimension.

Approach Used

1. Optimize hard so few data passes required.
    1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.

Approach Used

1. Optimize hard so few data passes required.
    1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.
    2. Dimensionally correct, adaptive, online, gradient descent for small-multiple passes.
        1. Online = update weights after seeing each example.
        2. Adaptive = learning rate of feature i according to $\frac{1}{\sqrt{\sum g_i^2}}$ where g_i = previous gradients.
        3. Dimensionally correct = still works if you double all feature values.
    3. Use (2) to warmstart (1).

Approach Used

1. Optimize hard so few data passes required.
    1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.
    2. Dimensionally correct, adaptive, online, gradient descent for small-multiple passes.
        1. Online = update weights after seeing each example.
        2. Adaptive = learning rate of feature i according to $\frac{1}{\sqrt{\sum g_i^2}}$ where g_i = previous gradients.
        3. Dimensionally correct = still works if you double all feature values.
    3. Use (2) to warmstart (1).
2. Use map-only Hadoop for process control and error recovery.

Approach Used

1. Optimize hard so few data passes required.
    1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.
    2. Dimensionally correct, adaptive, online, gradient descent for small-multiple passes.
        1. Online = update weights after seeing each example.
        2. Adaptive = learning rate of feature i according to $\frac{1}{\sqrt{\sum g_i^2}}$ where g_i = previous gradients.
        3. Dimensionally correct = still works if you double all feature values.
    3. Use (2) to warmstart (1).
2. Use map-only Hadoop for process control and error recovery.
3. Use custom AllReduce code to sync state.

Approach Used

1. Optimize hard so few data passes required.
   1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.
   2. Dimensionally correct, adaptive, online, gradient descent for small-multiple passes.
      1. Online = update weights after seeing each example.
      2. Adaptive = learning rate of feature i according to $\frac{1}{\sqrt{\sum g_i^2}}$ where g_i = previous gradients.
      3. Dimensionally correct = still works if you double all feature values.
   3. Use (2) to warmstart (1).
2. Use map-only Hadoop for process control and error recovery.
3. Use custom AllReduce code to sync state.
4. Always save input examples in a cachefile to speed later passes.

Approach Used

1. Optimize hard so few data passes required.
    1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.
    2. Dimensionally correct, adaptive, online, gradient descent for small-multiple passes.
        1. Online = update weights after seeing each example.
        2. Adaptive = learning rate of feature i according to $\frac{1}{\sqrt{\sum g_i^2}}$ where g_i = previous gradients.
        3. Dimensionally correct = still works if you double all feature values.
    3. Use (2) to warmstart (1).
2. Use map-only Hadoop for process control and error recovery.
3. Use custom AllReduce code to sync state.
4. Always save input examples in a cachefile to speed later passes.
5. Use hashing trick to reduce input complexity.

Approach Used

1. Optimize hard so few data passes required.
   1. L-BFGS = batch algorithm that builds up approximate inverse hessian according to: $\frac{\Delta_w \Delta_w^T}{\Delta_w^T \Delta_g}$ where Δ_w is a change in weights w and Δ_g is a change in the loss gradient g.
   2. Dimensionally correct, adaptive, online, gradient descent for small-multiple passes.
      1. Online = update weights after seeing each example.
      2. Adaptive = learning rate of feature i according to $\frac{1}{\sqrt{\sum g_i^2}}$ where g_i = previous gradients.
      3. Dimensionally correct = still works if you double all feature values.
   3. Use (2) to warmstart (1).
2. Use map-only Hadoop for process control and error recovery.
3. Use custom AllReduce code to sync state.
4. Always save input examples in a cachefile to speed later passes.
5. Use hashing trick to reduce input complexity.

Open source in Vowpal Wabbit 6.0. Search for it.

Empirical Results

2.1T sparse features
17B Examples
16M parameters
1K nodes
70 minutes

The end

Right now there is extreme diversity:

1. Many different notions of large scale.
2. Many different approaches.

What works generally?
What are the natural “kinds” of large scale learning problems?
And what are good solutions for each kind?
The great diversity implies this is really the beginning.

Numbers References

RBMs	Adam Coates, Rajat Raina, and Andrew Y. Ng, Large Scale Learning for Vision with GPUs, in the book.
speech	Jike Chong, Ekaterina Gonina Kisun You, and Kurt Keutzer, in the book.
Teralinear	Alekh Agarwal, Olivier Chapelle, Miroslav Dudik, and John Langford, Vowpal Wabbit 6.0.
LDA II	M. Hoffman, D. Blei, F. Bach, "Online Learning for Latent Dirichlet Allocation," in Neural Information Processing Systems (NIPS) 2010, Vancouver, 2010.
LDA III	Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, Alexander Smola, Scalable Inference in Latent Variable Models, In Submission.

Online References

Adapt I H. Brendan McMahan and Matthew Streeter Adaptive Bound Optimization for Online Convex Optimization, COLT 2010.

Adapt II John Duchi, Elad Hazan, and Yoram Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, COLT 2010.

Dim I http://www.machinedlearnings.com/2011/06/dimensional-analysis-and-gradient.html

Import Nikos Karampatziakis, John Langford, Online Importance Weight Aware Updates, UAI 2011

Batch References

BFGS I Broyden, C. G. (1970), "The convergence of a class of double-rank minimization algorithms", Journal of the Institute of Mathematics and Its Applications 6: 7690.

BFGS II Fletcher, R. (1970), "A New Approach to Variable Metric Algorithms", Computer Journal 13 (3): 317322.

BFGS III Goldfarb, D. (1970), "A Family of Variable Metric Updates Derived by Variational Means", Mathematics of Computation 24 (109): 2326.

BFGS IV Shanno, David F. (July 1970), "Conditioning of quasi-Newton methods for function minimization", Math. Comput. 24: 647656.

L-BFGS Nocedal, J. (1980). "Updating Quasi-Newton Matrices with Limited Storage". Mathematics of Computation 35: 773782.