Speeding up stochastic gradient descent

Yoshua Bengio

December 6th, 2007

NIPS WORKSHOP ON EFFICIENT MACHINE LEARNING

Thanks to: Olivier Delalleau, Frederic Bastien, Nicolas Le Roux, Pierre-Antoine Manzagol
Motivations:

- **large-scale AI learning tasks**
  - Insufficient Depth ⇒ poor generalization
- Stochastic gradient or online second order methods asymptotically preferable

Challenge: efficient training of deep architectures

- Greedy layerwise learning of *multiple levels of abstractions*
- Parallelizing stochastic gradient in deep networks
- Online natural gradient approximations
- Continuation methods for optimizing deep architectures
Machine Vision Example

very high level representation:

\[
\begin{array}{c}
\text{MAN} \\
\text{SITTING}
\end{array}
\]

... etc ...

slightly higher level representation

raw input vector representation:

\[
\mathbf{x} = \begin{bmatrix}
23 & 19 & 20 \\
x_1 & x_2 & x_3 \\
& & x_n
\end{bmatrix}
\]

- **MAN** abstraction corresponds to convoluted set of images (some very far in pixel distance)
- multiple levels of representation
- multiple levels of computation
- not clear which low & intermediate-level abstractions are good
- want to learn representations at all levels
- low & intermediate representations can be shared across many tasks
Each node $\in$ computations element set.
Left: compute $x \times \sin(a \times x + b)$, depth 4.
Right: elements = artificial neurons $f(x) = \tanh(b + w'x)$.
Computes a multi-layer neural network of depth 3.
Current Learning Algorithms: Depth

Depth = number of **levels** of composition of adaptable elements:

- kernel machines: **shallow**
- multi-layer neural networks: usually shallow, can be deep?
- decision trees: shallow
- boosting: adds one level, but generally **shallow**
When a function can be compactly represented by a deep architecture, it may need a very large architecture to be represented by an insufficiently deep one.
Insufficient Depth

- FFT time $O(n^2)$ with depth 1, $O(n \log n)$ with depth $\log n$
- Two-layer logic gates circuits:
  - can represent any function.
  - most functions require exponential nb gates (e.g. parity).
  - $\exists$ functions computable with a polynomial-size depth $k$ circuit, requiring exponential size with depth $k - 1$.
- Similar result holds for threshold neuron.

(Hastad, 1986; Yao, 1985; Wegener, 1987; Hastad and Goldmann, 1991; Bengio, 2003)
Theory suggests that DEPTH is necessary for statistical efficiency.

Until 2006, we knew no way to train a deep neural net to obtain better results than a shallow one (1 or 2 hidden layer) except for convolutional neural nets (Bengio and Le Cun, 2007).

- Not convex

Traditional approach to train neural nets: random initial weights + stochastic gradient descent.

Training seemed to get stuck in sub-optimal solutions, local minima or plateaus or just a too convoluted error surface.

Still not clear why it is so difficult to optimize deep architectures by gradient-based techniques, and why it is easier with convolutional nets.
What happened in 2006?

Geoff Hinton, Simon Osindero and Yee-Wye Teh published a Neural Computation paper on “A fast learning algorithm for Deep Belief Nets” (2006), that introduces these ideas:

- A deep unsupervised network could be trained greedily, layer by layer.
- Each layer an RBM modeling its inputs.
- Each layer outputs a representation of its input.
- This unsupervised net is a good initialization for a supervised net.

Presumably easier to learn “locally” (within each layer) than having to coordinate all the layers in a deep network.
Bottou & Le Cun (NIPS’2003) clearly demonstrate that with abundant training data and limited computing resources, online learning asymptotically outperforms any batch algorithm.

Bottou & Bousquet (NIPS’2007) study the trade-off between approximation error (too small class), estimation error (variance), and optimization error (not enough CPU time): to reach the best generalization error, large-scale learning makes it necessary to allow some optimization error.

Latter paper compares convergence rates of 1st and 2nd order batch methods with 1st and 2nd order online methods. Both are $O(.)$ similar, up to condition number vs input dimension.
The last 15 years of NIPS have been dominated by the **underfitting** problem.

Semi-supervised learning for AI: **TONS OF DATA** (mostly unlabeled), e.g. R. Collobert with 17 BILLION examples, and Google?

Local non-parametric models (a la SVM, nearest-neighbor, etc.) won’t scale because of statistical AND computational reasons.

Main challenge: training large models with non-convex optimization = **underfitting**
Natural Gradient

Let $\mathcal{L}$ be a cost function defined as

$$\tilde{\mathcal{L}}(\theta) = \int_x L(\theta, x) \tilde{p}(x) \, dx$$

Its gradient with respect to the parameters $\theta$ is

$$\tilde{g} = \int_x \frac{\partial L(\theta, x)}{\partial \theta} \tilde{p}(x) \, dx = \int_x g(\cdot, x) \tilde{p}(x) \, dx$$

The direction of the natural gradient (Amari, 1998) is

$$\tilde{\mathcal{C}}^{-1} \tilde{g}$$

with $\tilde{\mathcal{C}}$ the covariance of the gradients

$$\tilde{\mathcal{C}} = \int_x [g(\cdot, x) - \tilde{g}(\theta)] [g(\cdot, x) - \tilde{g}(\theta)]^T \tilde{p}(x) \, dx$$
Natural gradient minimizes overfitting

No $\tilde{p}$, only iid sample $\{x_1, \ldots, x_n\}$: empirical cost
$L(\theta) = \sum_{i=1}^{n} L(\theta, x_i)$ whose gradient wrt $\theta$ is

$$
g = \sum_{i=1}^{n} g(\cdot, x_i) = \sum_{i=1}^{n} g_i
$$

Central Limit Theorem $\Rightarrow$ $g \sim \mathcal{N}\left(\tilde{g}, \frac{\tilde{C}}{n}\right)$

Assuming Gaussian$(0; \sigma^2)$ prior on $\tilde{g}$, direction $u$ minimizing $E[u^T \tilde{g}]$ is

$$
u \propto \tilde{C}^{-1} g
$$

$\Rightarrow$ go in directions where gradients not only are strong, but also agree.
Blue: samples gradient
Red: mean gradient
Green: natural gradient
Axis: gradient covariance (e-vector $\times$ e-value)
Exact Natural Gradient is Impractical

1. Computing \( C \) is \( \mathcal{O}(np^2) \) with \( p \) parameters
2. Computing \( C^{-1} g \) is \( \mathcal{O}(p^3) \)
3. Memory for \( C \) is \( \mathcal{O}(p^2) \)
4. For 784 – 300 – 10 neural network (small network MNIST):
   212 Gb

Other problems arise in the online setting. Indeed, this algorithm needs a good estimate of \( C \). Although \( C \) varies smoothly, it must be evaluated over a large number of samples.
Low rank + moving covariance $C$ with forgetting factor $\gamma$.

1. Gram matrix of gradients, computing $k$ eigenvectors is $O(pk^2)$ (instead of $O(p^2k)$)

2. Do it every $b$ steps: $O\left(\frac{pk^2}{b}\right)$.

\[
C_i = \gamma^i C + \sum_{k=1}^{i} \gamma^{i-k} g_k g_k^T + \lambda I \quad i = 1, \ldots, b
\]

\[
v_i = C_i^{-1} g_i
\]

\[
X_i = \begin{bmatrix}
\gamma^{\frac{i}{2}} U & \gamma^{\frac{i-1}{2}} g_1 & \cdots & \gamma^{\frac{1}{2}} g_{i-1} & g_i
\end{bmatrix}
\]

\[
C_i = X_i X_i^T + \lambda I \quad v_i = X_i \alpha_i \quad g_i = X_i y_i
\]

\[
\alpha_i = (X_i^T X_i + \lambda I)^{-1} y_i
\]

\[
v_i = X_i (X_i^T X_i + \lambda I)^{-1} y_i
\]  

(1)
Neural nets & mixture models: almost independent groups of parameters

⇒ block diagonal approximation (one block per group)
Experimental Results: MNIST

50,000 training examples, 784-800-10 neural net

CPU time (in seconds)

Classification error on the training set

- Block diagonal TONGA
- Stochastic batchsize=1
- Stochastic batchsize=400
- Stochastic batchsize=1000
- Stochastic batchsize=2000

CPU time (in seconds)

Classification error on the test set

- Block diagonal TONGA
- Stochastic batchsize=1
- Stochastic batchsize=400
- Stochastic batchsize=1000
- Stochastic batchsize=2000

CPU time (in seconds)

Negative log-likelihood on the training set

- Block diagonal TONGA
- Stochastic batchsize=1
- Stochastic batchsize=400
- Stochastic batchsize=1000
- Stochastic batchsize=2000

CPU time (in seconds)

Negative log-likelihood on the test set

- Block diagonal TONGA
- Stochastic batchsize=1
- Stochastic batchsize=400
- Stochastic batchsize=1000
- Stochastic batchsize=2000

Yoshua Bengio
Experimental Results: Rectangles Data

900,000 training examples, 784-200-200-100-2 neural net

Yoshua Bengio
"Trade-off" between batch and stochastic:

\[
\theta_{k+1} \leftarrow \theta_k - \epsilon_k \sum_{t=s_k}^{s_k+b} \frac{\partial C(\theta_t, z_t)}{\partial \theta}
\]

Typical size of mini-batches: on the order of 10's or 100's.
Computing Faster Products

Bottleneck computation in neural nets: matrix-vector product.

Use minibatches: matrix-matrix product $\Rightarrow$ use BLAS

BLAS implementations:

- ACML (AMD)
- ATLAS: Automatically Tuned Linear Algebra Software
- NVIDIA: GPU, float only currently
- Goto (Kazushige Goto from University of Texas at Austin)
- MKL (Intel)
- BLAS: reference version from netlib.org

Yoshua Bengio
Comparing Different BLAS

1000 times $1024 \times 1024 \times 1024 \times 1024$ float on Intel Core2 Quad

### Uniprocessor

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>BLAS</th>
<th>ACML</th>
<th>ATLAS</th>
<th>Goto</th>
<th>MKL</th>
<th>NVIDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td>84.7</td>
<td>47.6</td>
<td>8.3</td>
<td>8.1</td>
<td>6.0</td>
<td>6.7</td>
<td>2</td>
</tr>
<tr>
<td>Speedup vs C</td>
<td>1</td>
<td>1.8</td>
<td>10.2</td>
<td>10.5</td>
<td>14.1</td>
<td>12.6</td>
<td>42.35</td>
</tr>
</tbody>
</table>

### Multiprocessor

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Goto1</th>
<th>Goto2</th>
<th>Goto4</th>
<th>MKL1</th>
<th>MKL2</th>
<th>MKL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td>84.7</td>
<td>6.0</td>
<td>5.6</td>
<td>3.2</td>
<td>6.7</td>
<td>3.7</td>
<td>1.5</td>
</tr>
<tr>
<td>Speedup vs C</td>
<td>1</td>
<td>14.1</td>
<td>15.1</td>
<td>26.5</td>
<td>12.6</td>
<td>22.9</td>
<td>56.5</td>
</tr>
</tbody>
</table>
512 hidden, 512 inputs, 32 examples mini-batch: 512 $\times$ 512 $\times$ 512 $\times$ 32

<table>
<thead>
<tr>
<th></th>
<th>BLAS</th>
<th>ACML</th>
<th>ATLAS</th>
<th>Goto</th>
<th>MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (s)</strong></td>
<td>403</td>
<td>318</td>
<td>158</td>
<td>131</td>
<td>135</td>
</tr>
<tr>
<td><strong>Speedup vs BLAS</strong></td>
<td>1</td>
<td>1.3</td>
<td>2.6</td>
<td>3.1</td>
<td>3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>BLAS</th>
<th>Goto1</th>
<th>Goto2</th>
<th>Goto4</th>
<th>MKL1</th>
<th>MKL2</th>
<th>MKL4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (s)</strong></td>
<td>403.1</td>
<td>131.1</td>
<td>120.6</td>
<td>104.8</td>
<td>134.9</td>
<td>96.6</td>
<td>117.0</td>
</tr>
<tr>
<td><strong>Speedup vs BLAS</strong></td>
<td>1</td>
<td>3.1</td>
<td>3.4</td>
<td>3.8</td>
<td>3.0</td>
<td>4.2</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Goto is GOOD and FREE.
Using 8-core and 16-core machines, NOT 100’s on a cluster.

- Even on SMPs, communication is a bottleneck!
- Use existing parallel implementations of BLAS to speed-up matrix-matrix multiplications: low speed-up unless using very large mini-batches (⇒ equivalent to batch)
- Multiple simultaneous **read access** in same memory area: OK
- Multiple simultaneous **write access** in same memory area: NOT OK!
Split data into $c$ chunks
Each of the $c$ cores sees one chunk of the data
Perform mini-batch gradient descent with parameters in shared memory:
- if all cores are updating parameters simultaneously: poor performance (time wasted waiting for memory write locks)
- proposed solution: at any time, only one core is allowed to update parameters. The index of the next core to update is stored in shared memory, and incremented after each update.
The Big Picture

Index of CPU allowed to update: 1

Data part 1 → CPU 1

Internal update 1 → Parameters → CPU 2

CPU 2 → Data part 2

Parameters

Internal update 2
Experiments

- UCI Letter dataset
- 16000 training + 4000 test, 16-dimensional input, 750 hidden, 26 classes
- Target test classification error: 3%
- Minibatch sizes optimized in \{4,8,16,32,64,128\}
- Learning rate optimized in [.001,.002,.005,. . . ,.05,.1]
- RESULTS ARE HIGHLY VARIABLE (5 initial conditions)
Our algorithm is compared with a similar form of parallelization in which we do not try to avoid write collisions:

- Each core works on a different mini-batch.
- After the mini-batch they share (sum) their gradients.
- All update their parameters with total gradient from all cores.
Proposed method gives good speed-up in terms of raw "samples/sec" speed.

Preliminary experiment measuring TIME PER EXAMPLE, not taking convergence time into account.

- \( N = 16 \) cores
- Speedup = 13 fold = 80% of linear speed-up
Data-parallelization = bigger mini-batches

BUT decrease frequency of updates ⇒ lower convergence speed

Suggested experimental setting:

- Identify target training error $e$ from simple experiments
- For each number of cores $c$, measure time needed to reach error $e$

Hyper-parameters (learning rate and mini-batch size in particular) need to be re-optimized for each value of $c$. 
Results with Training Convergence

* Our algorithm
* Naive
Continuation Methods: optimize a sequence of gradually less smooth cost functions leading to target cost function.
Can prove that the greedy layer-wise approach to training RBMs is a discrete continuation method. Adding each layer removes a constraint.

Stochastic gradient descent from small parameters is nearly a continuation method (with parameter norm as the smoothness control).

A curriculum can be framed as a continuation method (changing a sampling probability on examples according to how difficult they currently are to learn).
Conclusions

- Large-scale AI learning tasks **Insufficient Depth ⇒ poor generalization**
- Need stochastic/minibatch gradient or online second order methods
- Optimizing deep architectures?
  - Greedy layerwise learning of *multiple levels of abstractions*
  - Parallelizing stochastic gradient in deep networks: BLAS + architecture-aware solutions
  - Natural gradient can be efficiently approximated **online**: BEATS STOCHASTIC GRADIENT
  - Continuation methods for global optimization of deep architectures
Available on my web page:

- these slides
- review paper on deep architectures and Deep Belief Nets
  \textit{Learning Deep Architectures for AI}
- TONGA (NIPS 2007 paper) \textit{Top-Momoute Online Natural Gradient Algorithm}
Natural gradient works efficiently in learning.

Learning deep architectures for AI.
Technical Report 1312, Dept. IRO, Université de Montréal.

Scaling learning algorithms towards AI.

Hastad, J. (1986).
Almost optimal lower bounds for small depth circuits.

On the power of small-depth threshold circuits.

Training products of experts by minimizing contrastive divergence.
*Neural Computation, 14*:1771–1800.

Generative models for discovering sparse distributed representations.

A fast learning algorithm for deep belief nets.
*Neural Computation, 18*:1527–1554.

Unsupervised discovery of non-linear structure using contrastive backpropagation.
Extracting distributed representations of concepts and relations from positive and negative propositions.

The Complexity of Boolean Functions.
John Wiley & Sons.

Separating the polynomial-time hierarchy by oracles.