Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis

TAL BEN-NUN* and TORSTEN HOEFLER, ETH Zürich

Deep Neural Networks (DNNs) are becoming an important tool in modern computing applications. Accelerating their training is a major challenge and techniques range from distributed algorithms to low-level circuit design. In this survey, we describe the problem from a theoretical perspective, followed by approaches for its parallelization. Specifically, we present trends in DNN architectures and the resulting implications on parallelization strategies. We discuss the different types of concurrency in DNNs: synchronous and asynchronous stochastic gradient descent; distributed system architectures; communication schemes; and performance modeling. Based on these approaches, we extrapolate potential directions for parallelism in deep learning.

CCS Concepts: • General and reference — Surveys and overviews; • Computing methodologies — Neural networks; Distributed computing methodologies; Parallel computing methodologies; Machine learning;

Additional Key Words and Phrases: Deep Learning, Distributed Computing, Parallel Algorithms

ACM Reference format:

1 INTRODUCTION

Machine Learning, and in particular Deep Learning [LeCun et al. 2015], is a field that is rapidly taking over a variety of aspects in our daily lives. In the core of deep learning lies the Deep Neural Network (DNN), a construct inspired by the interconnected nature of the human brain. Trained properly, the expressiveness of DNNs provides accurate solutions for problems previously thought to be unsolvable, simply by observing large amounts of data. Deep learning has been successfully implemented for a plethora of subjects, ranging from image classiﬁcation [Huang et al. 2017], through speech recognition [Amodei et al. 2016] and medical diagnosis [Curean et al. 2013], to autonomous driving [Bojarski et al. 2016] and defeating human players in complex games [Silver et al. 2017] (see Fig. 1 for more examples).
What is Deep Learning good for?

- Digit Recognition
- Object Classification
- Segmentation
- Image Captioning
- Gameplay AI
- Translation
- Neural Computers
- Gigantic Language Models
- Towards Real Physics

A very active area of research!

<table>
<thead>
<tr>
<th>Year</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
<th>2016</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs.AI</td>
<td>1,081</td>
<td>1,765</td>
<td>1,022</td>
<td>1,105</td>
<td>1,929</td>
<td>2,790</td>
</tr>
<tr>
<td>cs.CV</td>
<td>577</td>
<td>852</td>
<td>1,349</td>
<td>2,261</td>
<td>3,627</td>
<td>5,693</td>
</tr>
</tbody>
</table>

23 papers per day!
How does Deep Learning work?

Layer-wise weight update

- ImageNet (1k): 180 GB
- ImageNet (22k): A few TB
- Industry: Much larger

- 100-200 layers deep
- ~100M-2B parameters
- 0.1-8 GiB parameter storage

- 10-22k labels
- growing (e.g., face recognition)
- weeks to train

Deep Learning is Supercomputing!

What is Deep Learning used for?
- Digit Recognition
- Object Classification
- Segmentation
- Image Captioning
- Gameplay AI
- Translation
- Neural computers
- Routing
A brief theory of supervised deep learning

labeled samples $x \in X \subset \mathcal{D}$

network structure (fixed)
weights $w$ (learned)

$f(x): X \rightarrow Y$

$w^* = \arg\min_{w \in \mathbb{R}^d} \mathbb{E}_{x \sim \mathcal{D}}[\ell(w, x)]$

layer-wise weight update

true label $l(x)$

label domain $Y$

$\ell_{sq}(w, x) = (f(x) - l(x))^2$

$\ell_{0-1}(w, x) = \begin{cases} 0 & f(x) = l(x) \\ 1 & f(x) \neq l(x) \end{cases}$

$\ell_{ce}(w, x) = -\sum_i l(x)_i \cdot \log \frac{e^{f(x)_i}}{\sum_k e^{f(x)_k}}$

$\begin{array}{l}
\text{convolution 1} \\
\text{convolution 2} \\
\text{pooling} \\
\text{convolution 3} \\
\text{fully connected}
\end{array}$
**Stochastic Gradient Descent**

Layer storage = \(|w_l| + |f_l(o_{l-1})| + |\nabla w_l| + |\nabla o_l|

\[
w^* = \arg\min_{w \in \mathbb{R}^d} \mathbb{E}_{x \sim \mathcal{D}}[\ell(w, x)]
\]

\[
f_1(x) \quad \text{convolution 1}
\]

\[
f_2(f_1(x)) \quad \text{convolution 2}
\]

\[
\text{pooling}
\]

\[
\ldots
\]

\[
f(x) \quad \text{fully connected}
\]
Trends in deep learning: hardware and multi-node

The field is moving fast – trying everything imaginable – survey results from 227 papers in the area of parallel deep learning

Deep Learning is largely on distributed memory today!

T. Ben-Nun, T. Hoefler: Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis, arXiv Feb 2018
Trends in **distributed** deep learning: node count and communication

The field is moving fast – trying everything imaginable – survey results from 227 papers in the area of parallel deep learning

Deep Learning research is converging to MPI!

T. Ben-Nun, T. Hoefler: Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis, arXiv Feb 2018
Minibatch Stochastic Gradient Descent (SGD)

Performance

Validation Error

A B C

Minibatch Size

Cat 0.54
Dog 0.28
Airplane 0.07
Horse 0.04
Bicycle 0.03
Truck 0.02

T. Ben-Nun, T. Hoefler: Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis, arXiv Feb 2018
Microbatching (μ-cuDNN) – how to implement layers best in practice?

- In cuDNN there are ~16 convolution implementations
- Performance depends on temporary memory (workspace) size
- Key idea: segment minibatch into microbatches, reuse workspace, use different algorithms
- How to choose microbatch sizes and algorithms?

Microbatching Strategy

- none (undivided)
- powers-of-two only
- any (unrestricted)

Dynamic Programming (Space Reuse)

Integer Linear Programming (Space Sharing)

Fast (up to 4.54x faster on DeepBench)

Yosuke Oyama, Tal Ben-Nun, TH and Satoshi Matsuoka: μ-cuDNN: Accelerating Deep Learning Frameworks with Micro-Batching, Cluster 2018
Model parallelism – limited by network size

- Parameters can be distributed across processors
- Mini-batch has to be copied to all processors
- Backpropagation requires all-to-all communication every layer

U.A. Muller and A. Gunzinger: Neural Net Simulation on Parallel Computers, IEEE Int’l Conf. on Neural Networks 1994
Pipeline parallelism – limited by network size

- Layers/parameters can be distributed across processors
- Sparse communication pattern (only pipeline stages)
- Mini-batch has to be copied through all processors

G. Blelloch and C.R. Rosenberg: Network Learning on the Connection Machine, IJCAI’87
Data parallelism – limited by batch-size

- Simple and efficient solution, easy to implement
- Duplicate parameters at all processors

Hybrid parallelism

- Layers/parameters can be distributed across processors
- Can distribute minibatch
- Often specific to layer-types (e.g., distribute fc layers but handle conv layers data-parallel)
  - Enables arbitrary combinations of data, model, and pipeline parallelism – very powerful!
Updating parameters in **distributed** data parallelism

Parameter server (sharded) \( w' = u(w, \nabla w) \)

\[ T = 2L + 2P \gamma m/s \ G \]

- Collective operations
- Topologies
- Neighborhood collectives
- RMA?

Hierarchical Parameter Server
S. Gupta et al.: Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study. ICDM’16

Adaptive Minibatch Size
S. L. Smith et al.: Don't Decay the Learning Rate, Increase the Batch Size, arXiv 2017
Parameter (and Model) consistency - centralized

- Parameter exchange frequency can be controlled, while still attaining convergence:

- DistBelief [Dean et al. 2012] moved the idea to distributed
- Trades off “statistical performance” for “hardware performance”

---

J. Dean et al.: Large scale distributed deep networks, NIPS’12.
F. Niu et al.: Hogwild: A lock-free approach to parallelizing stochastic gradient descent, NIPS’11.
Parameter (and Model) consistency - decentralized

- Parameter exchange frequency can be controlled, while still attaining convergence:

- May also consider limited/slower distribution – gossip [Jin et al. 2016]
Parameter consistency in deep learning

Using physical forces between different versions of $w$:

$$w^{(t+1,i)} = w^{(t,i)} - \eta \nabla w^{(t,i)} - \alpha (w^{(t,i)} - \bar{w}_t)$$

$$\bar{w}_{t+1} = (1 - \beta)\bar{w}_t + \frac{\beta}{m} \sum_{i=1}^{m} w^{(t,i)}$$

S. Zhang et al.: Deep learning with Elastic Averaging SGD, NIPS’15
Parameter consistency in deep learning

Synchronous SGD

Stale-Synchronous SGD

Asynchronous SGD (HOGWILD!)

Model Averaging (e.g., elastic)

Consistent

Inconsistent

T. G. Dietterich: Ensemble Methods in Machine Learning, MCS 2000
Communication optimizations

- Different options how to optimize updates
  - Send $\nabla w$, receive $w$
  - Send FC factors $(o_{l-1}, o_l)$, compute $\nabla w$ on parameter server
    
    **Broadcast factors to not receive full $w$**
  - Use lossy compression when sending, accumulate error locally!

- Quantization
  - Quantize weight updates and potentially weights
  - Main trick is stochastic rounding [1] – expectation is more accurate
    
    **Enables low precision (half, quarter) to become standard**
  - TernGrad - ternary weights [2], 1-bit SGD [3], …

- Sparsification
  - Do not send small weight updates or only send top-k [4]
    
    **Accumulate omitted gradients locally**

---

[3] F. Seide et al. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs, In Interspeech 2014
Sparsification – top-\(k\) Stochastic Gradient Descent

- Pick the \(k\)-largest elements of the vector at each node!
  - Accumulate the remainder locally (convergence proof, similar to async. SGD with implicit staleness bounds [1])

**Assumption 1.** There exists a (small) constant \(\xi\) such that, for every iteration \(t \geq 0\), we have:

\[
\left\| \text{TopK} \left( \frac{1}{P} \sum_{p=1}^{P} (\alpha \tilde{G}^p_t(v_t) + \epsilon^p_t) \right) - \sum_{p=1}^{P} \frac{1}{P} \text{TopK} \left( \alpha \tilde{G}^p_t(v_t) + \epsilon^p_t \right) \right\| \leq \xi \| \alpha \tilde{G}_t(v_t) \|.
\]

**Discussion.** We validate Assumption 1 experimentally on a number of different learning tasks in Section 6 (see also Figure 1). In addition, we emphasize the following points:

(a) RCV1 convergence.  
(b) Linear regression.  
(c) ResNet110 on CIFAR10.

SparCML – Quantified sparse allreduce for decentral updates

\[ \nabla w_1 + \nabla w_2 + \nabla w_3 + \nabla w_4 \]

Microsoft Speech Production Workload Results – 2 weeks \(\rightarrow\) 2 days!

<table>
<thead>
<tr>
<th>System</th>
<th>Dataset</th>
<th>Model</th>
<th># of nodes</th>
<th>Algorithm</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piz Daint</td>
<td>ImageNet</td>
<td>VGG19</td>
<td>8</td>
<td>Q4</td>
<td>1.55 (3.31)</td>
</tr>
<tr>
<td>Piz Daint</td>
<td>ImageNet</td>
<td>AlexNet</td>
<td>16</td>
<td>Q4</td>
<td>1.30 (1.36)</td>
</tr>
<tr>
<td>Piz Daint</td>
<td>MNIST</td>
<td>MLP</td>
<td>8</td>
<td>Top16.Q4</td>
<td>3.65 (4.53)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Top16.Q4</td>
<td>19.12 (22.97)</td>
</tr>
</tbody>
</table>

C. Renggli, TH et al. SparCML: High-Performance Sparse Communication for Machine Learning, arXiv 2018
Optimizing parallel deep learning systems is a bit like navigating Tokyo by public transit---at first glance impossibly complex but eventually doable with the right guidelines---
Deep500: An HPC Deep Learning Benchmark and Competition

- Integrates tensorflow, pytorch, caffee2 into a single benchmarking framework
  - Separate definition of benchmark metrics, shared across all levels
- Lean reference implementations – simple to understand and change
  - Operators (layer computations)
  - Optimizers (SGD etc.)
  - Distribution schemes (cf. Horovod)
  
  Similar to reference LINPACK benchmark
- Supports optimization of components
  - E.g., no need to reimplement an optimizer to replace gradient compression!

Easily compare to all frameworks!
HPC for Deep Learning – Summary

- Deep learning is HPC – very similar computational structure, in fact very friendly
  - Amenable to specialization, static scheduling, all established tricks - microbatching
- Main bottleneck is communication – reduction by trading off

<table>
<thead>
<tr>
<th>Parameter Consistency</th>
<th>Parameter Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Bounded synchronous SGD</td>
<td>• Lossless compression of gradient updates</td>
</tr>
<tr>
<td>• Central vs. distributed parameter server</td>
<td>• Quantization of gradient updates</td>
</tr>
<tr>
<td>• EASGD to ensemble learning</td>
<td>• Sparsification of gradient updates</td>
</tr>
</tbody>
</table>

- Very different environment from traditional HPC
  - Trade-off accuracy for performance!
- Performance-centric view in HPC can be harmful for accuracy!

T. Hoefler: “Twelve ways to fool the masses when reporting performance of deep learning workloads”
  (my humorous guide to floptimization in deep learning will be published this week during IPAM)
How to **not** do this

“Twelve ways to fool the masses when reporting performance of deep learning workloads”
(my humorous guide to floptimize deep learning, blog post Nov. 2018)

Twelve ways to fool the masses when reporting performance of deep learning workloads! (not to be taken too seriously)

Deep learning and HPC

- Deep learning is HPC
  - In fact, it’s probably (soon?) bigger than traditional HPC
    *Definitely more money...*

- Interest in the HPC community is tremendous
  - Number of learning papers at HPC conferences seems to be growing exponentially
    *Besides at SC18, whut?!*

- Risk of unreality
  - HPC people know how to do HPC
  - And deep learning is HPC, right?
    *Not quite ... while it’s really similar (tensor contractions)*
    *But it’s also quite different!*
“Statistical performance” vs. “hardware performance”

- **Tradeoffs between those two**
  - Very weird for HPC people – we always operated in double precision
    - *Mostly out of fear of rounding issues*

- **Deep learning shows how little accuracy one can get away with**
  - Well, examples are drawn randomly from some distribution we don’t know …
  - Usually, noise is quite high …
  - So the computation doesn’t need to be higher precision than that noise
    - *Pretty obvious! In fact, it’s similar in scientific computing but in tighter bounds and not as well known*

- **But we HPC folks like flop/s! Or maybe now just ops or even aiops? Whatever, fast compute!**
  - A humorous guide to **floptimization**
  - Twelve rules to help present your (not so great?) results in a much better light
1) Ignore accuracy when scaling up!

- Too obvious for this audience
  - Was very popular in 2015!

- Surprisingly many (still) do this

Learning community’s self-correction (Y. LeCun)

Scalability without a good baseline? (D. Bailey)

HPC picking up!
2) Do not report test accuracy!

- Training accuracy is sufficient isn’t it?
3) Do not report all training runs needed to tune hyperparameters!

- Report the best run – SGD is a bit fragile, so don’t worry
  
  At the end, the minutes for the final run matter most!
4) Compare outdated hardware with special-purpose hardware!

- Tesla K20 in 2018!? 
  
  *Even though the older machines would win the beauty contest!*
5) Show only kernels/subsets when scaling!

- Run layers or communication kernels in isolation
  - Avoids issues with accuracy completely 😊

Doesn’t that look a bit like GoogLeNet?

VS.
6) Do not consider I/O!

- Reading the data? Nah, make sure it’s staged in memory when the benchmark starts!
7) Report highest ops numbers (whatever that means)!

- Yes, we’re talking ops today, 64-bit flops was so yesterday!
  - If we don’t achieve a target fast enough, let’s redefine it!
    
    *And never talk about how many more of those ops one needs to find a solution, it’s all about the rate, op/s!*

- Actually, my laptop achieves an “exaop”:
  - each of the $3 \times 10^9$ transistors switching a binary digit each at $2.4 \times 10^9$ Hz
8) Show performance when enabling option set A and show accuracy when enabling option set B!

- Pretty cool idea isn’t it? Hyperparameters sometimes conflict
  
  So always tune the to show the best result, whatever the result shall be!
9) Train on (unreasonably) large inputs!

- The pinnacle of floptimization! Very hard to catch!
  
  *But Dr. Catlock Holmes below can catch it.*

Low-resolution cat (244x244 – 1 Gflop/example)

VS.

High-resolution cat (8kx8x – 1 Tflop/example)
10) Run training just for the right time!

- Train for fixed wall-time when scaling processors
  - so when you use twice as many processors you get twice as many flop/s!
  
  *But who cares about application speedup?*
11) Minibatch sizing for fun and profit – weak vs. strong scaling.

- All DL is strong scaling – limited model and limited data
- So just redefine the terms relative to minibatches:
  - Weak scaling keeps MB size per process constant – overall grows (less iterations per epoch, duh!)
  - Strong scaling keeps overall MB size constant (better but harder)

- Microbatching is not a problem!
12) Select carefully how to compare to the state of the art!

- Compare either time to solution or accuracy if both together don’t look strong!

  There used to be conventions but let’s redefine them.
Turning 180-degree – Deep Learning for HPC – Neural Code Comprehension

- In 2017, GitHub reports 1 billion git commits in 337 languages!
- Can DNNs understand code?
- Previous approaches read the code directly $\rightarrow$ suboptimal (loops, functions)

$$\begin{align*}
\text{double thres} &= 5.0; \\
\text{if} \ (x < \text{thres}) & \ x = y \times y; \\
\text{else} & \ x = 2.0 \times y;
\end{align*}$$

\[
\begin{align*}
\text{if} (x < \text{thres}) & \ x = y \times y; \\
\text{else} & \ x = 2.0 \times y;
\end{align*}
\]

Dataflow (basic blocks)

contextual Flow Graph

Ben-Nun, Jakobovits, TH: Neural Code Comprehension: A Learnable Representation of Code Semantics, NIPS 2018
Deep Learning for HPC – Neural Code Comprehension

- Embedding space (using the Skip-gram model)

Ben-Nun, Jakobovits, TH: Neural Code Comprehension: A Learnable Representation of Code Semantics, NIPS 2018
Deep Learning for HPC – Neural Code Comprehension

Table 3: Algorithm classification test accuracy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Accuracy [%]</td>
<td>88.2</td>
<td>84.8</td>
<td>94.0</td>
<td>94.83</td>
</tr>
</tbody>
</table>

Table 4: Heterogeneous device mapping results

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Prediction Accuracy [%]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Grewe et al. [27]</td>
<td>DeepTune [17]</td>
</tr>
<tr>
<td>AMD Tahiti 7970</td>
<td>73.38</td>
<td>83.68</td>
</tr>
<tr>
<td>NVIDIA GTX 970</td>
<td>72.94</td>
<td>80.29</td>
</tr>
</tbody>
</table>

Table 5: Speedups achieved by coarsening threads

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon HD 5900</td>
<td>1.21</td>
<td>1.10</td>
<td>1.17</td>
<td>1.25</td>
</tr>
<tr>
<td>AMD Tahiti 7970</td>
<td>1.01</td>
<td>1.05</td>
<td>1.23</td>
<td>1.07</td>
</tr>
<tr>
<td>NVIDIA GTX 480</td>
<td>0.86</td>
<td>1.10</td>
<td>1.14</td>
<td>1.02</td>
</tr>
<tr>
<td>NVIDIA Tesla K20c</td>
<td>0.94</td>
<td>0.99</td>
<td>0.93</td>
<td>1.03</td>
</tr>
</tbody>
</table>

Table 2: Analogy and test scores for inst2vec

<table>
<thead>
<tr>
<th>Context Size</th>
<th>Syntactic Analogies</th>
<th>Semantic Analogies</th>
<th>Semantic Distance Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Types</td>
<td>Options</td>
<td>Conversions</td>
</tr>
<tr>
<td>1</td>
<td>101 (18.04%)</td>
<td>13 (24.53%)</td>
<td>100 (6.63%)</td>
</tr>
<tr>
<td>2</td>
<td>226 (40.36%)</td>
<td>45 (84.91%)</td>
<td>134 (8.89%)</td>
</tr>
<tr>
<td>3</td>
<td>125 (22.32%)</td>
<td>24 (45.28%)</td>
<td>48 (3.18%)</td>
</tr>
</tbody>
</table>

Optimal tiling

Predicts which device is faster (CPU or GPU)
Hyperparameter and Architecture search

- Meta-optimization of hyper-parameters (momentum) and DNN architecture
  - Using Reinforcement Learning [1] (explore/exploit different configurations)
  - Genetic Algorithms with modified (specialized) mutations [2]
  - Particle Swarm Optimization [3] and other meta-heuristics

---

**Reinforcement Learning [1]**

**Evolutionary Algorithms [4]**

---

[3] P. R. Lorenzo et al.: Hyper-parameter Selection in Deep Neural Networks Using Parallel Particle Swarm Optimization, GECCO’17
GoogLeNet in more detail

- ~6.8M parameters
- 22 layers deep

C. Szegedy et al. Going Deeper with Convolutions, CVPR’15
Computing fully connected layers

$$f_i(x) = \sigma(\sum w_{i,j} x_j + b_i)$$

$$\mathbf{w} = \mathbf{w}_1, \mathbf{x} = \mathbf{x}_1, \mathbf{b} = \mathbf{b}_1$$

$$O(C_{out} \cdot C_{in} \cdot N) \quad O(\log C_{in})$$
$$O(C_{in} \cdot N \cdot C_{out}) \quad O(\log N)$$
$$O(C_{in} \cdot C_{out} \cdot N) \quad O(\log C_{out})$$
## Computing convolutional layers

<table>
<thead>
<tr>
<th>Method</th>
<th>Work (W)</th>
<th>Depth (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>( N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x )</td>
<td>( [\log_2 C_{in}] + [\log_2 K_y] + [\log_2 K_x] )</td>
</tr>
<tr>
<td>im2col</td>
<td>( N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x )</td>
<td>( [\log_2 C_{in}] + [\log_2 K_y] + [\log_2 K_x] )</td>
</tr>
<tr>
<td>FFT</td>
<td>( c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{in} + N \cdot C_{out}) + HWN \cdot C_{in} \cdot C_{out} )</td>
<td>( 2[\log_2 HW] + [\log_2 C_{in}] )</td>
</tr>
<tr>
<td>Winograd</td>
<td>( \alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P )</td>
<td>( 2[\log_2 r] + 4[\log_2 \alpha] + [\log_2 C_{in}] )</td>
</tr>
</tbody>
</table>

\( \alpha \equiv m - r + 1, \quad P \equiv N \cdot [H/m] \cdot [W/m] \)

---

M. Mathieu et al.: Fast Training of Convolutional Networks through FFTs, ICLR'14  
A. Lavin and S. Gray: Fast Algorithms for Convolutional Neural Networks, CVPR'16