Scaling Machine Learning
Three easy pieces

Alex Smola
CMU & Marianas Labs
github.com/dmlc
Outline

Main Contributors

• Dave Andersen (ParameterServer)
• Mu Li (MXNet, ParameterServer, FM)
• Manzil Zaheer (LDA)
• Tianqi Chen (MXNet)
• Ziqi Liu (Recommender, FM)
• Jean-Baptiste Tristan (LDA)
• Yu-Xiang Wang (Recommender, FM)

Topics

• Parameter Server Basics Logistic Regression (Classification)
• Memory Subsystem Matrix Factorization (Recommender)
• Large Distributed State
  • Factorization Machines
  • Latent Variable Models
• GPUs
  MXNET and applications
Parameter Server

Server

Server

Server

Server

Data (local or cloud)

read

process

process

process

process

process

process

write

update

Data (local or cloud)

local state
(or copy)

Data (local or cloud)

read

process

process

process

process

process

process

write

update

Data (local or cloud)

local state
(or copy)

Data (local or cloud)

read

process

process

process

process

process

process

write

update

Data (local or cloud)

local state
(or copy)
Multicore

Data (local or cloud)

read

process

write

update

Data (local or cloud)

Parameter Server

local state (or copy)
GPUs (for Deep Learning)

Data (local or cloud)

read

write

update

Parameter Server

local state (or copy)
Details

• **Parameter Server Basics**
  Logistic Regression (Classification)

• **Memory Subsystem**
  Matrix Factorization (Recommender)

• **Large Distributed State**
  • Factorization Machines (CTR)
  • Latent Dirichlet Allocation (LDA)

• **GPUs**
  MXNET and applications
Marianas Labs

**Estimate Click Through Rate**

$p(\text{click} | \text{ad}, \text{query}, \text{w})$

$\Rightarrow y = \text{x}$

**Qualcomm Machine Learning - qualcomm.com**

Qualcomm is Teaching Robots to Solve Problems. Welcome to Today.

**Enhanced Machine**

Get better results by combining different machine learning techniques.

**What is Machine Learning**

A Machine Learning Introduction.

**Scholarly articles for machine learning**

- Genetic algorithms and machine learning - Goldberg - Cited by 1971
- An introduction to MCMC for machine learning - Andrieu - Cited by 1261
- Machine learning for the detection of oil spills in ... - Kubat - Cited by 750

**Machine learning** is a subfield of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence. **Machine learning** explores the construction and study of algorithms that can learn from and make predictions on data.
Click Through Rate (CTR)

- Linear function class
  \[ f(x) = \langle w, x \rangle \]

- Logistic regression
  \[ p(y|x, w) = \frac{1}{1 + \exp(-y \langle w, x \rangle)} \]

- Optimization Problem
  \[
  \text{minimize} \sum_{i=1}^{m} \log(1 + \exp(-y_i \langle w, x_i \rangle)) + \lambda \|w\|_1
  \]

- Solve distributed over many machines (typically 1TB to 1PB of data)
Optimization Algorithm

- **Compute gradient on data**
- \( l_1 \) norm is nonsmooth, hence proximal operator
  \[
  \arg\min_w \|w\|_1 + \frac{\gamma}{2} \|w - (w_t - \eta g_t)\|_2
  \]
- **Updates for \( l_1 \) are very simple**
  \[
  w_i \leftarrow \text{sgn}(w_i) \max(0, |w_i| - \epsilon)
  \]
- **All steps decompose by coordinates**
- **Solve in parallel (and asynchronously)**
Parameter Server Template

- Compute gradient on (subset of data) on each client
- Send gradient from client to server asynchronously
  \[
push(key\_list,value\_list,timestamp)\]
- Proximal gradient update on server per coordinate
- Server returns parameters
  \[
pull(key\_list,value\_list,timestamp)\]
- Lots of tricks for bandwidth saving

Smola & Narayananmurthy, 2010, VLDB
Gonzalez et al., 2012, WSDM
Dean et al, 2012, NIPS
Shervashidze et al., 2013, WWW
Google, Baidu, Facebook,
Amazon, Yahoo, Microsoft
Solving it at scale

- **2014 - Li et al., OSDI’14**
  - 500 TB data, $10^{11}$ variables
  - Local file system stores files
  - 1000 servers (corp cloud),
  - 1h time, 140 MB/s learning

- **2015 - Online solver**
  - 1.1 TB (Criteo), $8 \cdot 10^8$ variables, $4 \cdot 10^9$ samples
  - S3 stores files (no preprocessing) - better IO library
  - 5 machines (c4.8xlarge),
  - 1000s time, 220 MB/s learning per machine
Details

• Parameter Server Basics
  Logistic Regression (Classification)

• Memory Subsystem
  Matrix Factorization (Recommender)

• Large Distributed State
  • Factorization Machines (CTR)
  • Latent Dirichlet Allocation (LDA)

• GPUs
  MXNET and applications
Recommender Systems

- Users $u$, movies $m$ (or projects)
- Function class
  \[ r_{um} = \langle v_u, w_m \rangle + b_u + b_m \]
- Loss function for recommendation (Yelp, Netflix)
  \[ \sum_{u \sim m} (\langle v_u, w_m \rangle + b_u + b_m - y_{um})^2 \]
Recommender Systems

- Regularized Objective
  \[
  \sum_{u \sim m} (\langle v_u, w_m \rangle + b_u + b_m + b_0 - r_{um})^2 + \frac{\lambda}{2} \left[ \|U\|_{\text{Frob}}^2 + \|V\|_{\text{Frob}}^2 \right]
  \]

- Update operations
  \[
  v_u \leftarrow (1 - \eta_t \lambda) v_u - \eta_t w_m (\langle v_u, w_m \rangle + b_u + b_m + b_0 - r_{um})
  \]
  \[
  w_m \leftarrow (1 - \eta_t \lambda) w_m - \eta_t v_u (\langle v_u, w_m \rangle + b_u + b_m + b_0 - r_{um})
  \]

- Very simple SGD algorithm (random pairs)
- This should be cheap ...

memory subsystem
This should be cheap ...

- $O(md)$ burst reads and $O(m)$ random reads
- Netflix dataset
  $m = 100$ million, $d = 2048$ dimensions, 30 steps
- Runtime should be > 4500s
  - 60 GB/s memory bandwidth = 3300s
  - 100 ns random reads = 1200s

We get 560s. Why?

Liu, Wang, Smola, RecSys 2015
Power law in Collaborative Filtering

Netflix dataset

# of ratings

# movies

# ratings
Key Ideas

• **Stratify ratings by users**
  (only 1 cache miss / read per user / out of core)

• **Keep frequent movies in cache**
  (stratify by blocks of movie popularity)

• **Avoid false sharing between sockets**
  (key cached in the wrong CPU causes miss)

<table>
<thead>
<tr>
<th>K</th>
<th>SC-SGD L1 Cache</th>
<th>SC-SGD L3 Cache</th>
<th>GraphChi L1 Cache</th>
<th>GraphChi L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.84%</td>
<td>0.43%</td>
<td>12.77%</td>
<td>2.21%</td>
</tr>
<tr>
<td>256</td>
<td>2.85%</td>
<td>0.50%</td>
<td>12.89%</td>
<td>2.34%</td>
</tr>
<tr>
<td>2048</td>
<td>3.3%</td>
<td>1.7%</td>
<td>15%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
### Key Ideas

#### GraphChi Partitioning

<table>
<thead>
<tr>
<th>K</th>
<th>SC-SGD L1 Cache</th>
<th>SC-SGD L3 Cache</th>
<th>GraphChi L1 Cache</th>
<th>GraphChi L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.84%</td>
<td>0.43%</td>
<td>12.77%</td>
<td>2.21%</td>
</tr>
<tr>
<td>256</td>
<td>2.85%</td>
<td>0.50%</td>
<td>12.89%</td>
<td>2.34%</td>
</tr>
<tr>
<td>2048</td>
<td>3.3%</td>
<td>1.7%</td>
<td>15%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
## Key Ideas

**SC-SGD partitioning**

<table>
<thead>
<tr>
<th>K</th>
<th>SC-SGD L1 Cache</th>
<th>SC-SGD L3 Cache</th>
<th>GraphChi L1 Cache</th>
<th>GraphChi L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.84%</td>
<td>0.43%</td>
<td>12.77%</td>
<td>2.21%</td>
</tr>
<tr>
<td>256</td>
<td>2.85%</td>
<td>0.50%</td>
<td>12.89%</td>
<td>2.34%</td>
</tr>
<tr>
<td>2048</td>
<td>3.3%</td>
<td>1.7%</td>
<td>15%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
Speed (c4.8xlarge)

Netflix - 100M, 15 iterations
Yahoo - 250M, 30 iterations
Convergence

- GraphChi blocks (users, movies) into random groups
- Poor mixing
- Slow convergence
Details

- Parameter Server Basics
  Logistic Regression (Classification)
- Memory Subsystem
  Matrix Factorization (Recommender)
- Large Distributed State
  - Factorization Machines (CTR)
  - Latent Dirichlet Allocation (LDA)
- GPUs
  MXNET and applications
A Linear Model is not enough

$p(y|x, w)$

Marianas Labs
Factorization Machines

- Linear Model
  \[ f(x) = \langle w, x \rangle \]

- Polynomial Expansion (Rendle, 2012)
  \[ f(x) = \langle w, x \rangle + \sum_{i<j} x_i x_j \text{tr} \left( V_i^{(2)} \otimes V_j^{(2)} \right) + \]
  \[ \sum_{i<j<k} x_i x_j x_k \text{tr} \left( V_i^{(3)} \otimes V_j^{(3)} \otimes V_k^{(3)} \right) + \ldots \]

memory hog

too large for individual machine
Prefetching to the rescue

- Most keys are infrequent (power law distribution)
- Prefetch the embedding vectors for a minibatch from parameter server
- Compute gradients and push to server
  - Variable dimensionality embedding
  - Enforcing sparsity (ANOVA style)
  - Adaptive gradient normalization
  - Frequency adaptive regularization (CF style)
A more interesting observation is that these memory adaptive constraints do not affect the test accuracy. To the contrary, we even see a slight improvement when the dimension $k$ is greater than 8 for CTR2. The reason could be that the model capacity control is of great importance when the dimension $k$ is large. And these memory adaptive constraints can provide additional capacity control besides the $\ell_2$ and $\ell_1$ regularizers.

5.3 Fixed-point Compression

We evaluate lossy fixed-point compression for data communication. By default, both the model and gradient entries are represented as 32 bit floats. In this experiment, we compress these values to lower precision integers. More specifically, given a bin size $b$ and number of bits $n$, we represent $x$ by the following $n$-bit integer $z$:

$$z = j x b \times 2^n k + z,$$  

where $j$ is a Bernoulli random variable chosen such as to ensure that $E[z] = 2^n x b$. 

(Criteo 1TB)
Faster Solver (small Criteo)

(a) Total data sent by workers in one iteration. The compression rates from 4-byte to 1-byte are 4.2x and 2.9x for Criteo2 and CTR2, respectively.

(b) The relative test logloss comparing to no fixed-point compression.

Figure 3: Compressing model and gradient using the fixed-point compression, where 4-byte means using the default 32-bit floating-point format.

We implemented the fixed-point compression as a user-defined filter in the parameter server framework. Since multiple numbers are communicated in each round, we choose $b$ to be the absolute maximum value of these numbers. In addition, we used the key caching and lossless data compression (via LZ4) filters.

The results for $n = 8, 16, 24$ are shown in Figure 3. As expected, fixed-point compression linearly reduces the network traffic volume, since the traffic is dominated by communicating the model and gradient. A more interesting observation is that we obtained a 4.2x compression rate from 32-bit floating-point to 8-bit fixed-point on Criteo2. The reason is the latter improves the compression rate for the following lossless LZ4 compression.

We observed different effects of accuracy on these two datasets: CTR2 is robust to the number precision, while Criteo2 has a 6% increase of logloss if only using 1-byte presentation. However, a medium compression rate even improves the model accuracy. This might be because the lossy compression acts as a regularization to the objective function.

5.4 Comparison with LibFM

Figure 4: Comparison with LibFM on a single machine. The data preprocessing time for LibFM is omitted.

To our best knowledge, there is no publicly released distributed FM solver. Hence we only compare DiFacto to the popular single machine package LibFM developed by Rendle [14]. We only report results on Criteo1 and CTR1 on a single machine, since LibFM fails on the other two larger datasets. We perform a similar grid search of the hyperparameters as we did for DiFacto. As LibFM only uses single thread, we run DiFacto with 1 worker and 1 server in sequential execution order. We also report the performance using 10 workers and 10 servers on a single machine for reference.

The results are shown in Figure 4. As can been seen, DiFacto converges significantly faster than LibFM, it uses 2 times fewer iterations to reach the best model. This is because the adaptive learning rate used in DiFacto better models the data sparsity and the adaptive regularization and constraints can further accelerate the convergence. In particular, the latter results in a lower test logloss on the CTR1 dataset, where the number of features exceeds the number of examples, requiring improved capacity control.

Also note that DiFacto with a single worker is twice slower than LibFM per iteration. This is because the data communication.


Details

• Parameter Server Basics
  Logistic Regression (Classification)

• Memory Subsystem
  Matrix Factorization (Recommender)

• Large Distributed State
  • Factorization Machines (CTR)
  • Latent Dirichlet Allocation (LDA)

• GPUs
  MXNET and applications
Clustering & Topic Models

Clustering:
- Group objects by prototypes

Topics:
- Decompose objects into prototypes
Clustering & Topic Models

Clustering

Latent Dirichlet Allocation

\[ \alpha \]

\[ \theta \]

\[ y \]

\[ x \]

prior

cluster probability

cluster label

instance

\[ \alpha \]

\[ \theta \]

\[ y \]

\[ x \]

topic probability

topic label

instance

Carnegie Mellon University
\[ p(z, x | \alpha, \beta) = \prod_{i=1}^{m} p(z_i | \alpha) \prod_{k=1}^{k} p(\{x_{ij} | z_{ij} = k\} | \beta) \]
Collapsed Sampler

Griffiths & Steyvers, 2005

\[ p(z, x | \alpha, \beta) = \prod_{i=1}^{m} p(z_i | \alpha) \prod_{k=1}^{k} p(\{x_{ij} | z_{ij} = k\} | \beta) \]

\[ \frac{n^{-ij}(t, d) + \alpha_t}{n^{-i}(d) + \sum_t \alpha_t} \]

\[ \frac{n^{-ij}(t, w) + \beta_t}{n^{-i}(t) + \sum_t \beta_t} \]

language prior

\[
\begin{align*}
\text{fast mixing} \\
\text{topic} \\
\text{probability} \\
\text{topic label} \\
\text{instance} \\
\end{align*}
\]
Gibbs Sampler

- For 1000 iterations do
  - For each document do
    - For each word in the document do
      - Resample topic for the word
      - Lock (word, topic) table
      - Update local (document, topic) table
      - Update (word, topic) table
      - Unlock (word, topic) table

this kills parallelism
Stochastic Cellular Automata

- Locks are evil
- Collapsing everything is not necessary
  - Collapse documents
  - Mix and alternate language model
  - Exchange between machines

Iteration 2t+1

Zaheer, Tristan, Smola, 2016 (submitted)
Speed (4 machines, 1000 topics)

740M tokens

Graph showing the per word log-likelihood over time for different methods: SCA, CGS, and CVB0.
Speed (4 machines, 1000 topics)

- 1.1B tokens
- >540M samples/s
- SCA
- CGS
- CVB0

scales to 170B tokens
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Dataset</th>
<th>Number of topics</th>
<th>Size of vocabulary</th>
<th>Number of documents</th>
<th>Number of tokens</th>
<th>Hardware</th>
<th>Year</th>
<th>Speed (tokens/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo! LDA</td>
<td>PubMed data</td>
<td>2K</td>
<td>140K</td>
<td>8.2M</td>
<td>797M</td>
<td>10 machines Hadoop cluster</td>
<td>2010</td>
<td>&lt; 13M</td>
</tr>
<tr>
<td>lightLDA</td>
<td>Bing “web chunk” data</td>
<td>1,000K</td>
<td>50K</td>
<td>1,200M</td>
<td>200,000M</td>
<td>24 machines (480 cores)</td>
<td>2014</td>
<td>60M</td>
</tr>
<tr>
<td>F+Nomad LDA</td>
<td>Amazon product reviews (from SNAP)</td>
<td>1K</td>
<td>1,680K</td>
<td>30M</td>
<td>1,500M</td>
<td>32 Xeon E5 CPUs (640 cores)</td>
<td>2014</td>
<td>110M</td>
</tr>
<tr>
<td>ESCA</td>
<td>100 copies of Wikipedia (kept on solid state disks)</td>
<td>1K</td>
<td>290K</td>
<td>677M</td>
<td>128,000M</td>
<td>20 nodes Amazon Cloud c4.8xlarge</td>
<td>2015</td>
<td>1,000M</td>
</tr>
</tbody>
</table>
Details

- Parameter Server Basics
  Logistic Regression (Classification)
- Memory Subsystem
  Matrix Factorization (Recommender)
- Large Distributed State
  - Factorization Machines (CTR)
  - Latent Dirichlet Allocation (LDA)
- GPUs
  MXNET and applications
Several deep learning frameworks

- **Torch7**: matlab-like tensor computation in Lua
- **Theano**: symbolic programming in Python
- **Caffe**: configuration in Protobuf

**Problem - need to learn another system for a different programming flavor**

**MXNET**

- provides programming interfaces in a consistent way
- multiple languages: C++/Python/R/Julia/…
- fast, memory efficient, and distributed training
Programming Interfaces

- Tensor Algebra Interface
  - compatible (e.g. NumPy)
  - GPU/CPU/ARM (mobile)
- Symbolic Expression
  - Easy Deep Network design
  - Automatic differentiation
- Mixed programming
  - CPU/GPU
  - symbolic + local code

**Python**

```python
>>> import mxnet as mx
>>> a = mx.nd.ones((2, 3), ... mx.gpu(2))
>>> print (a * 2).asnumpy()
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
```

**Julia**

```julia
using MXNet
mlp = @mx.chain mx.Variable() =>
  mx.FullyConnected(num_hidden=128) =>
  mx.Activation(act_type=:relu) =>
  mx.FullyConnected(num_hidden=64) =>
  mx.Activation(act_type=:relu) =>
  mx.FullyConnected(num_hidden=10) =>
  mx.Softmax()
```
from data import ilsvrc12_iterator
import mxnet as mx

## define alexnet
input_data = mx.symbol.Variable(name="data")

# stage 1
conv1 = mx.symbol.Convolution(data=input_data, kernel=(11, 11), stride=(4, 4), num_filter=96)
relu1 = mx.symbol.Activation(data=conv1, act_type="relu")
pool1 = mx.symbol.Pooling(data=relu1, pool_type="max", kernel=(3, 3), stride=(2,2))
ln1 = mx.symbol.LRN(data=pool1, alpha=0.0001, beta=0.75, knorm=1, nsize=5)

# stage 2
conv2 = mx.symbol.Convolution(data=ln1, kernel=(5, 5), pad=(2, 2), num_filter=256)
relu2 = mx.symbol.Activation(data=conv2, act_type="relu")
pool2 = mx.symbol.Pooling(data=relu2, kernel=(3, 3), stride=(2, 2), pool_type="max")
ln2 = mx.symbol.LRN(data=pool2, alpha=0.0001, beta=0.75, knorm=1, nsize=5)

# stage 3
conv3 = mx.symbol.Convolution(data=ln2, kernel=(3, 3), pad=(1, 1), num_filter=384)
relu3 = mx.symbol.Activation(data=conv3, act_type="relu")
conv4 = mx.symbol.Convolution(data=relu3, kernel=(3, 3), pad=(1, 1), num_filter=384)
relu4 = mx.symbol.Activation(data=conv4, act_type="relu")
conv5 = mx.symbol.Convolution(data=relu4, kernel=(3, 3), pad=(1, 1), num_filter=256)
relu5 = mx.symbol.Activation(data=conv5, act_type="relu")
pool3 = mx.symbol.Pooling(data=relu5, kernel=(3, 3), stride=(2, 2), pool_type="max")

# stage 4
flatten = mx.symbol.Flatten(data=pool3)
fc1 = mx.symbol.FullyConnected(data=flatten, num_hidden=4096)
relu6 = mx.symbol.Activation(data(fc1), act_type="relu")
dropout1 = mx.symbol.Dropout(data=relu6, p=0.5)

# stage 5
fc2 = mx.symbol.FullyConnected(data=dropout1, num_hidden=4096)
relu7 = mx.symbol.Activation(data=fc2, act_type="relu")
dropout2 = mx.symbol.Dropout(data=relu7, p=0.5)

# stage 6
fc3 = mx.symbol.FullyConnected(data=dropout2, num_hidden=1000)
softmax = mx.symbol.SoftmaxOutput(data=fc3, name='softmax')

## data
batch_size = 256
train, val = ilsvrc12_iterator(batch_size=batch_size, input_shape=(3,224,224))

## train
num_gpus = 4
gpus = [mx.gpu(i) for i in range(num_gpus)]
model = mx.model.FeedForward(
  ctx=gpus,
symbol=softmax,
num_round=20,
learning_rate=0.01,
momentum=0.9,
wd=0.00001)
model.fit(X=train, eval_data=val, batch_end_callback=mx.callback.Speedometer(batch_size=batch_size))
Engine

- All operations issued into backend C++ engine
  Binding languages’ performance does not matter
- Engine builds the read/write dependency graph
  - lazy evaluation
  - parallel execution
  - sophisticated memory allocation
Distributed Deep Learning
Distributed Deep Learning

Plays well with S3 Spot instances & containers
Results

- Imagenet datasets (1m images, 1k classes)
- Google Inception network (88 lines of code)

40% faster due to parallelization  50% GPU memory reduction
Distributed Results

g2.2xlarge network limit

Convergence on 4 x dual GTX980

Marianas Labs

Carnegie Mellon University
Amazon g2.8xlarge

- 12 instances (48 GPUs) @ $0.50/h spot
- Minibatch size 512
- BSP with 1 delay between machines
- 2 GB/s bandwidth between machines (awful)

```
10.113.170.187, 10.157.109.227, 10.169.170.55, 10.136.52.151, 10.45.64.250, 10.166.137.100,
10.97.167.10, 10.97.187.157, 10.61.128.107, 10.171.105.160, 10.203.143.220, 10.45.71.20 (all over the place in availability zone)
```

- Compressing to 1 byte per coordinate helps a bit but adds latency due to extra pass (need to fix)
- **37x speedup on 48 GPUS**
- Imagenet’12 dataset in trained in 4h, i.e. $24 (with alexnet; googlenet even better for network)
Mobile, too

- 10 hours on 10 GTX980
- Deployed on Android
Side-effects of using MXNET

Alex Smola <alex@smola.org>
1:29 PM (1 hour ago)

to David, Dan, Hyeontaek, FAWN, maas-users

Haha. Zichao's code is getting more efficient over time.

Zichao Yang
1:33 PM (57 minutes ago)

to Alex, David, FAWN, maas-users

sorry... I was indeed running experiment using 8 gpus last night and suddenly found the cluster was down :(

Mu Li

to Alex, David, FAWN, maas-users

sometime zichao and I are using gpus at the same time. though it looks like mxnet can get more powers due to the multiple streams for each gpu. (didn't see an apparent slow down for mxnet even zichao is also using the gpus)
## Tensorflow vs. MXNET

<table>
<thead>
<tr>
<th></th>
<th>Languages</th>
<th>MultiGPU</th>
<th>Distributed</th>
<th>Mobile</th>
<th>Runtime Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensorflow</td>
<td>Python</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>?</td>
</tr>
<tr>
<td>MXNET</td>
<td>Python, R, Julia, Go, Javascript</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

dmlc.ml
### Some numbers

<table>
<thead>
<tr>
<th></th>
<th>Googlenet batch 32 (24)</th>
<th>Alexnet batch 128</th>
<th>VGG batch 32 (24)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Torch7</strong></td>
<td>172ms</td>
<td>135ms</td>
<td>459ms</td>
</tr>
<tr>
<td><strong>Caffe</strong></td>
<td>170ms</td>
<td>165ms</td>
<td>448ms</td>
</tr>
<tr>
<td><strong>MXNET</strong></td>
<td>172ms (new and improved)</td>
<td>147ms</td>
<td>456ms</td>
</tr>
<tr>
<td><strong>Tensor flow</strong></td>
<td>940ms</td>
<td>433ms</td>
<td>980ms</td>
</tr>
</tbody>
</table>
Some numbers
Summary

Main Contributors

- Dave Andersen (ParameterServer)
- Mu Li (MXNet, ParameterServer, FM)
- Manzil Zaheer (LDA)
- Tianqi Chen (MXNet)
- Ziqi Liu (Recommender, FM)
- Jean-Baptiste Tristan (LDA)
- Yu-Xiang Wang (Recommender, FM)

Topics

- Parameter Server Basics
  Logistic Regression (Classification)
- Memory Subsystem
  Matrix Factorization (Recommender)
- Large Distributed State
  Factorization Machines
- Latent Variable Models
- GPUs
  MXNET and applications