MxNet Gluon

Basics, Computer Vision, NLP (and even more NLP)
Part IV (Distributed Training)

Leonard Lausen
Haibin Lin
Alex Smola

aws

Outline

8:30-9:15	Installation and Basics (NDArray, AutoGrad, Libraries)
9:15-9:30	Neural Networks 101 (MLP, ConvNet, LSTM, Loss, SGD) - Part I
9:30-10:00	Break
10:00-10:30	Neural Networks 101 (MLP, ConvNet, LSTM, Loss, SGD) - Part II
10:30-11:00	Computer Vision 101 (Gluon CV)
11:00-11:30	Parallel and distributed training
11:30-12:00	Data I/O in NLP (and iterators)
12:00-13:30	Break
13:30-14:15	Embeddings
14:15-15:00	Language models (LM)
15:00-15:30	Sequence Generation from LM
15:30-16:00	Break
16:00-16:15	Sentiment analysis
16:15-17:00	Transformer Models & machine translation
17:00-17:30	Questions

Training

Stochastic Gradient Descent

- **Optimization Problem**

Training loss

Weight decay

$$\underset{w}{\text{minimize}} \frac{1}{m} \sum_{i=1}^{m} l(y_i, f(x_i, w)) + \lambda \|w\|_2^2$$

- **Stochastic Gradient Approximation**
  - **Gradient on Minibatch**

$$g_B := \frac{1}{b} \sum_{i \in B} \partial_w l(y_i, f(x_i, w)) \text{ and } w \leftarrow \text{Optimizer}(w, g_B)$$

  - **Update w with advanced first order solver**
    (Adam, AdaGrad, Momentum, Eve, SGD, SGLD …)

Stochastic Gradient Descent

- SGD
- DCASGD
- NAG
- Adam
- SGLD
- AdaGrad
- RMSProp
- AdaDelta
- Ftrl
- Adamax
- Nadam

- **Stochastic Gradient Descent (SGD)**

$$w \leftarrow (1 - 2\eta\lambda)w - \eta g_b$$

- **SGD with Momentum and Clipping**

$$s \leftarrow \mu s + \eta \operatorname{clip}(g_b) + \eta\lambda w$$

$$w \leftarrow w - s$$

Quite often learning rate is piecewise constant (this yields better accuracy in practice, albeit convergence is slower)

- **Improved conditioning, momentum, precision …**

Momentum

- Average over recent gradients
  - Helps with local minima
  - Flat (noisy) gradients

$$m_t = (1 - \lambda)m_{t-1} + \lambda g_t$$

$$w_t \leftarrow w_t - \eta_t g_t - \tilde{\eta}_t m_t$$

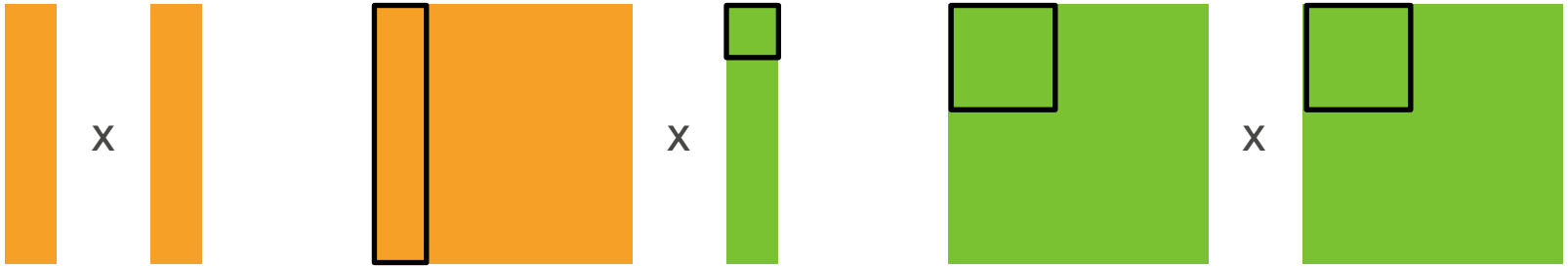
- Can lead to oscillations for large momentum
- Nesterov's accelerated gradient

$$m_{t+1} = \mu m_t + \epsilon g(w_t - \mu m_t)$$

$$w_{t+1} = w_t - m_{t+1}$$

https://blogs.princeton.edu/imabandit/2013/04/01/acceleratedgradientdescent/
https://www.cs.cmu.edu/~ggordon/10725-F12/slides/09-acceleration.pdf

Minibatch



- Aggregate gradients over a few instances before updating
  - Reduces variance in gradients
  - Better for vectorization (GPUs)
    (vector, vector) < (vector, matrix) < (matrix, matrix)

$$\langle x, x' \rangle \; < \; Mx \; < \; MX$$

  - Large minibatch may need lots of memory
    (and may slow updates).
- **For multiple GPUs you need to scale up mini batch size (256GPUs and 20 samples/GPU = mini batch of 5120!)**

Learning rate decay

- **Constant**
  (requires schedule for piecewise constant, tricky)
- **Useful hack**
  Keep learning rate constant until no improvement on validation data, then reduce rate.
- **Polynomial decay**
  Canonical choice for convex solvers $\eta = \frac{\alpha}{\sqrt{\beta + t}}$
- **Exponential decay**
  not recommended

AdaGrad

- **Adaptive learning rate** (preconditioner)

$$\eta_{ij}(t) = \frac{\eta_0}{\sqrt{K + \sum_t g_{ij}^2(t)}}$$

- For directions with large gradient, decrease learning rate aggressively to avoid instability
- If gradients start vanishing, learning rate decrease reduces, too
- **Local variant**

$$\eta_{ij}(t) = \frac{\eta_t}{\sqrt{K + \sum_{t'=t-\tau}^{t} g_{ij}^2(t')}}$$

time window

Duchi, Hazan, Singer, 2010
http://www.magicbroom.info/Papers/DuchiHaSi10.pdf

Capacity control

- Minimizing loss can lead to overfitting
- **Weight decay**

$$w_{t+1} \leftarrow w_t - \eta_t g_t$$

$$w_{t+1} \leftarrow (1 - \lambda) w_t - \eta_t g_t$$

prevents parameters from diverging

- **Gradient clipping**
  - Overheated GPU
  - Numerical instabilities
  - Hacky but necessary (e.g. deep architectures)

Dropout

Goals

- **Avoid parameter sensitivity**
- **Distributed representation** (info in > 1 dim)

Solution

- Zero out half of the nodes in each layer (Dropout)
- Or zero-out weights (DropConnect), slightly better

Srivastava, Hinton, Krizhevski, Sutskever, Salakhutdinov http://jmlr.org/papers/v15/srivastava14a.html
http://cs.nyu.edu/~wanli/dropc/

Dropout

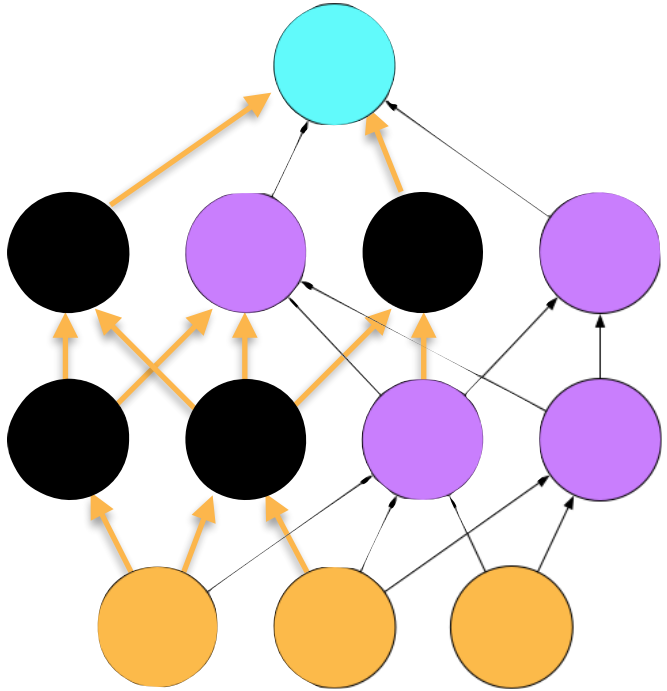
- Without dropout

$$y_i = W x_i$$
$$x_{i+1} = \sigma(y_i)$$

- With dropout

$$z_{ij} = \xi_{ij} x_{ij}$$
$$y_i = W z_i$$
$$x_{i+1} = \sigma(y_i)$$

  Draw a new mask for every step
- Update via backprop



Dropout

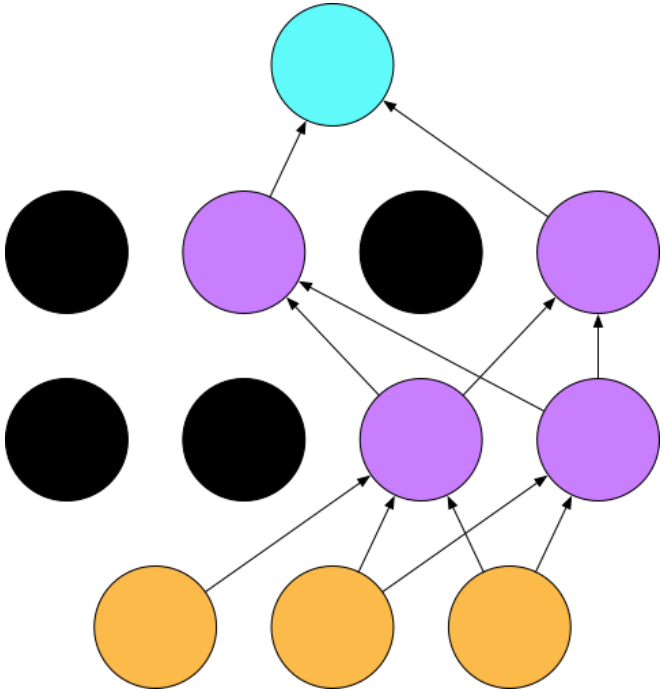
- Without dropout

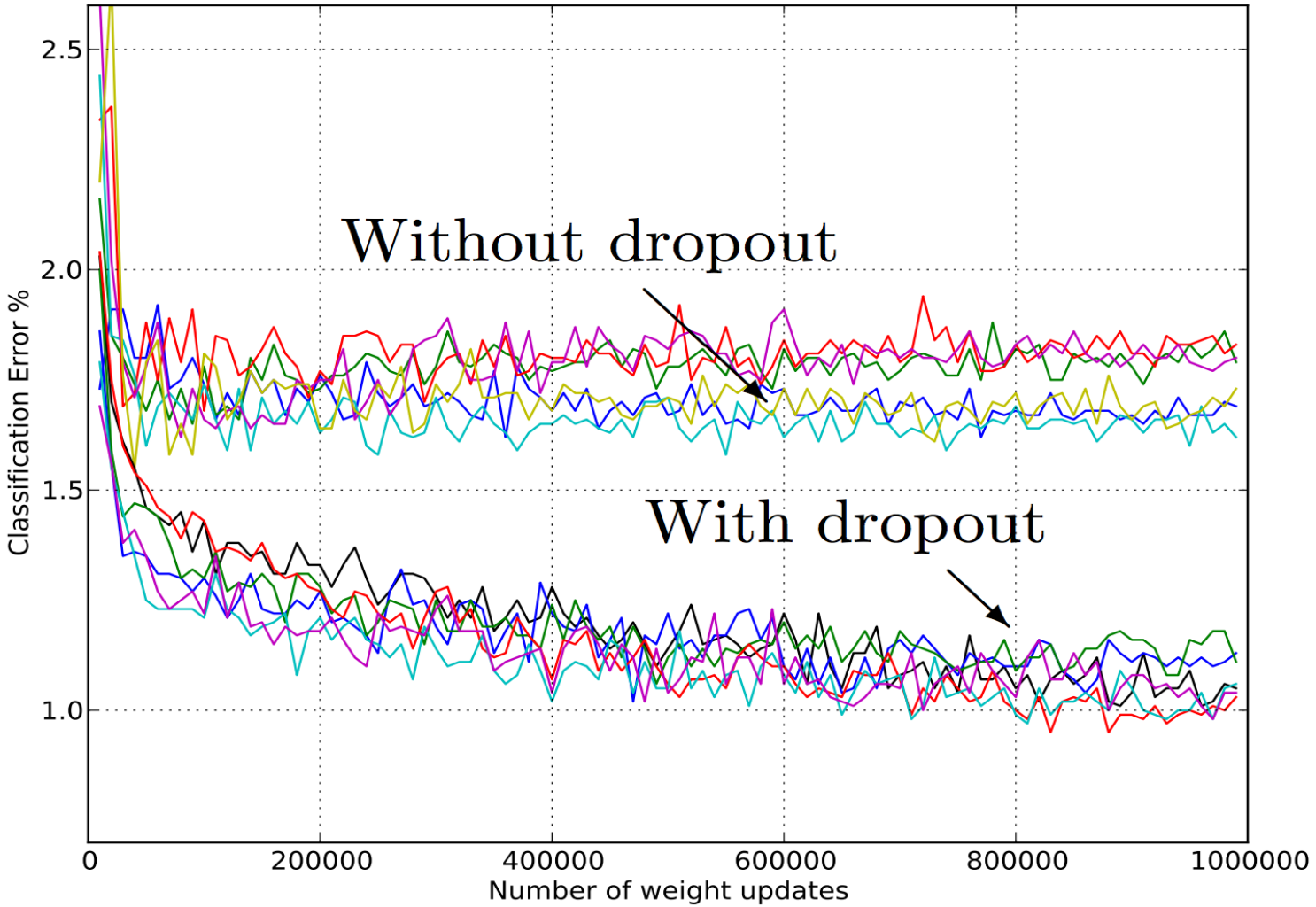
$$y_i = W x_i$$
$$x_{i+1} = \sigma(y_i)$$

- With dropout

$$z_{ij} = \xi_{ij} x_{ij}$$
$$y_i = W z_i$$
$$x_{i+1} = \sigma(y_i)$$

  Draw a new mask for every step
- Update via backprop

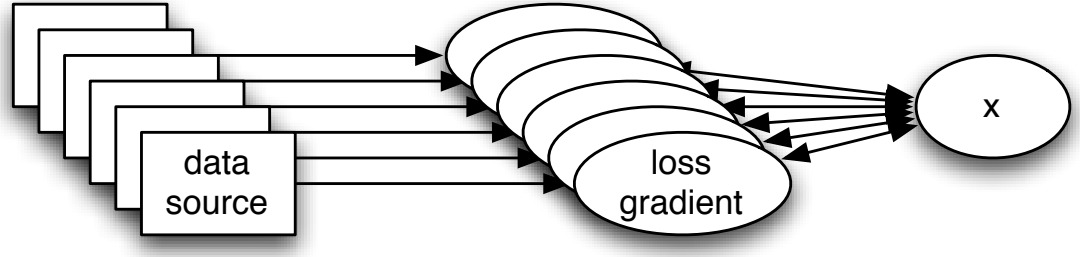




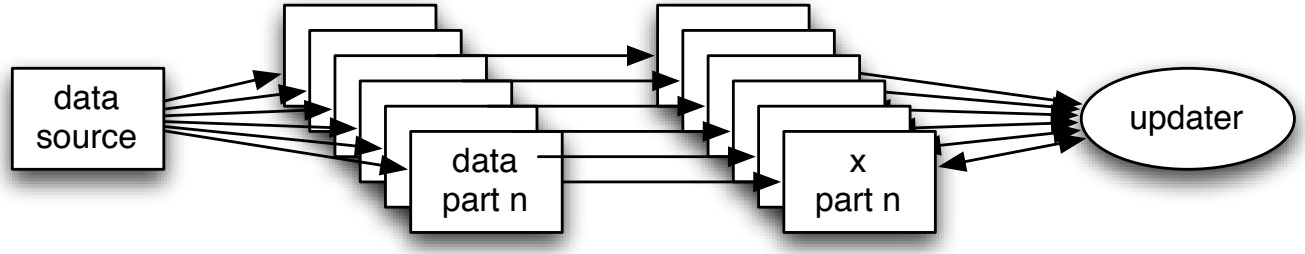
Parallel and Distributed Training

Parallelization

- **Data Parallel (easier & more efficient)**
  Distribute data over multiple GPUs / CPUs / machines

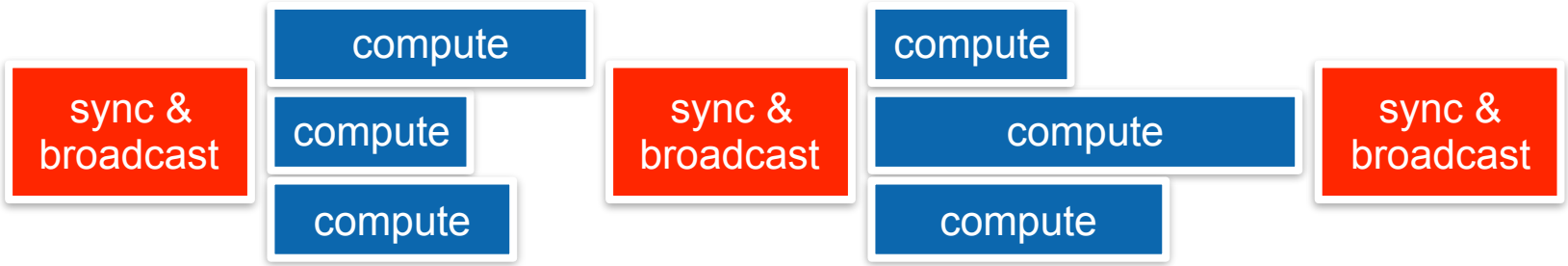


- **Model Parallel (only for huge models)**
  Distribute parts of the computation over multiple GPUs / CPUs

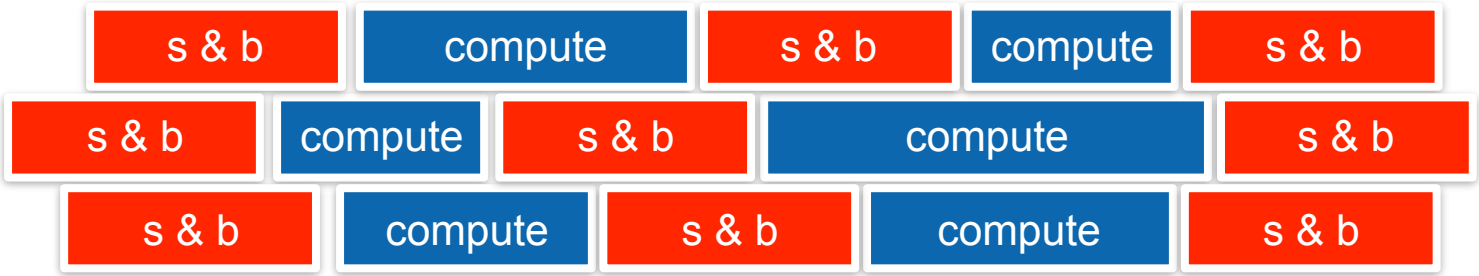


Parallelization

- **Synchronous - easy for homogeneous jobs**



- **Asynchronous - efficient for high time dispersion**

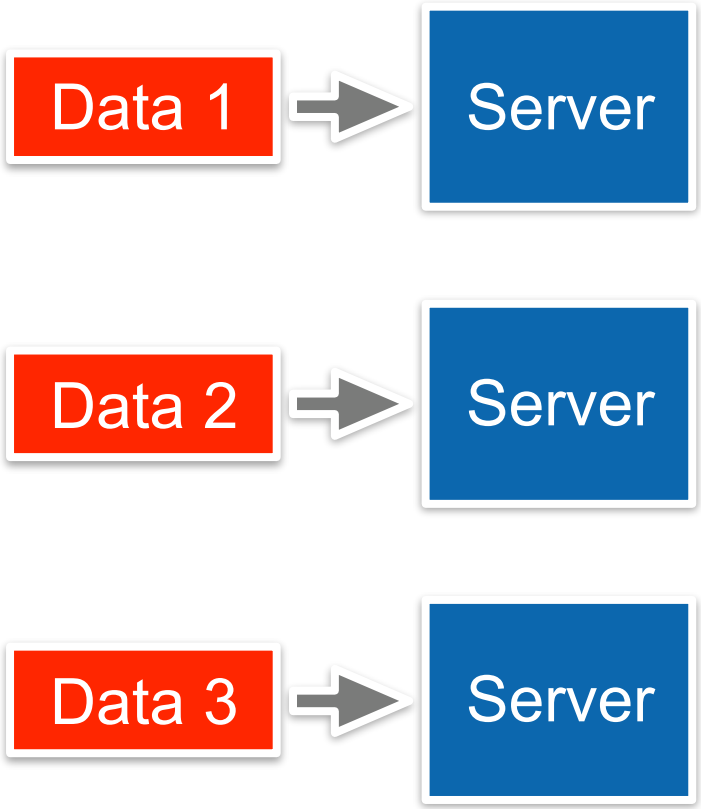


Parallelization - Pragmatic Strategy

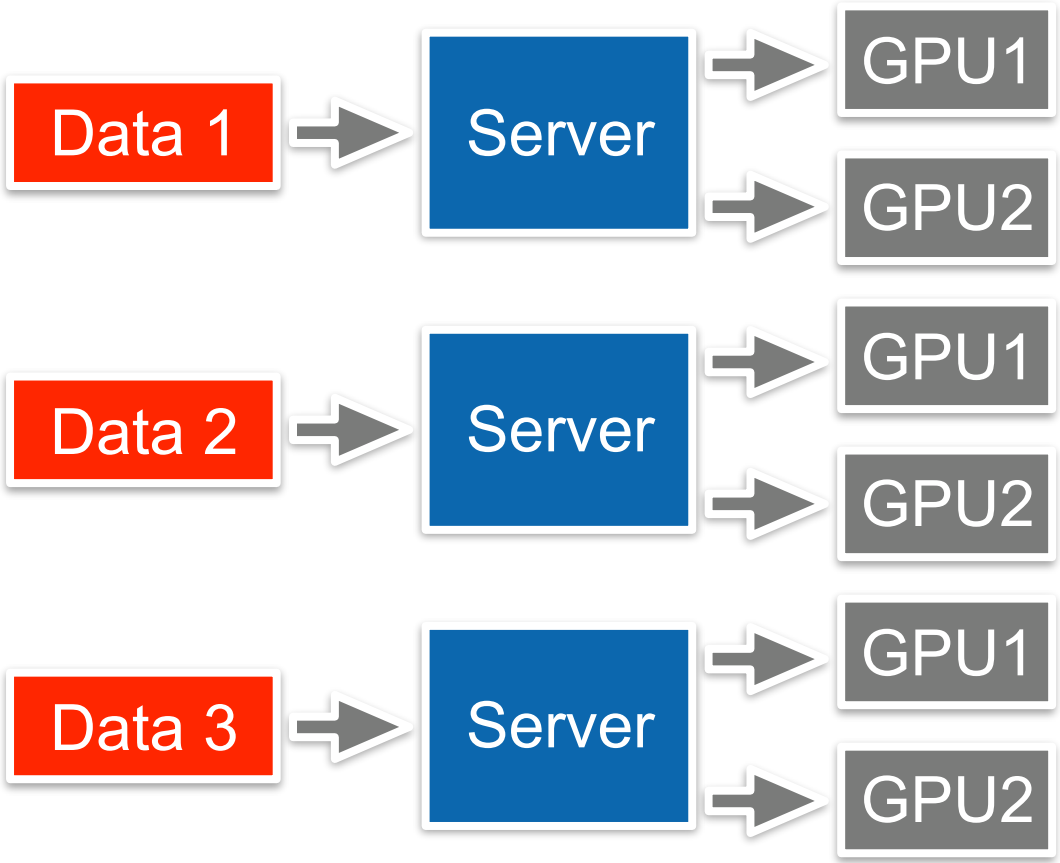
- **Data Parallel & Synchronous**
  - Even for 512 GPUs you only need 32 P2.16xlarge
    (so the speed dispersion is negligible).
  - GPUs have plenty of RAM
    (only an inefficient framework needs model parallelization)
- **Algorithm Template** (shard data over machines)
  - Compute gradient on mini batch on each GPU
  - Aggregate gradients into (key,value) store
  - Broadcast them back to GPUs

Parallelization - Pragmatic Strategy

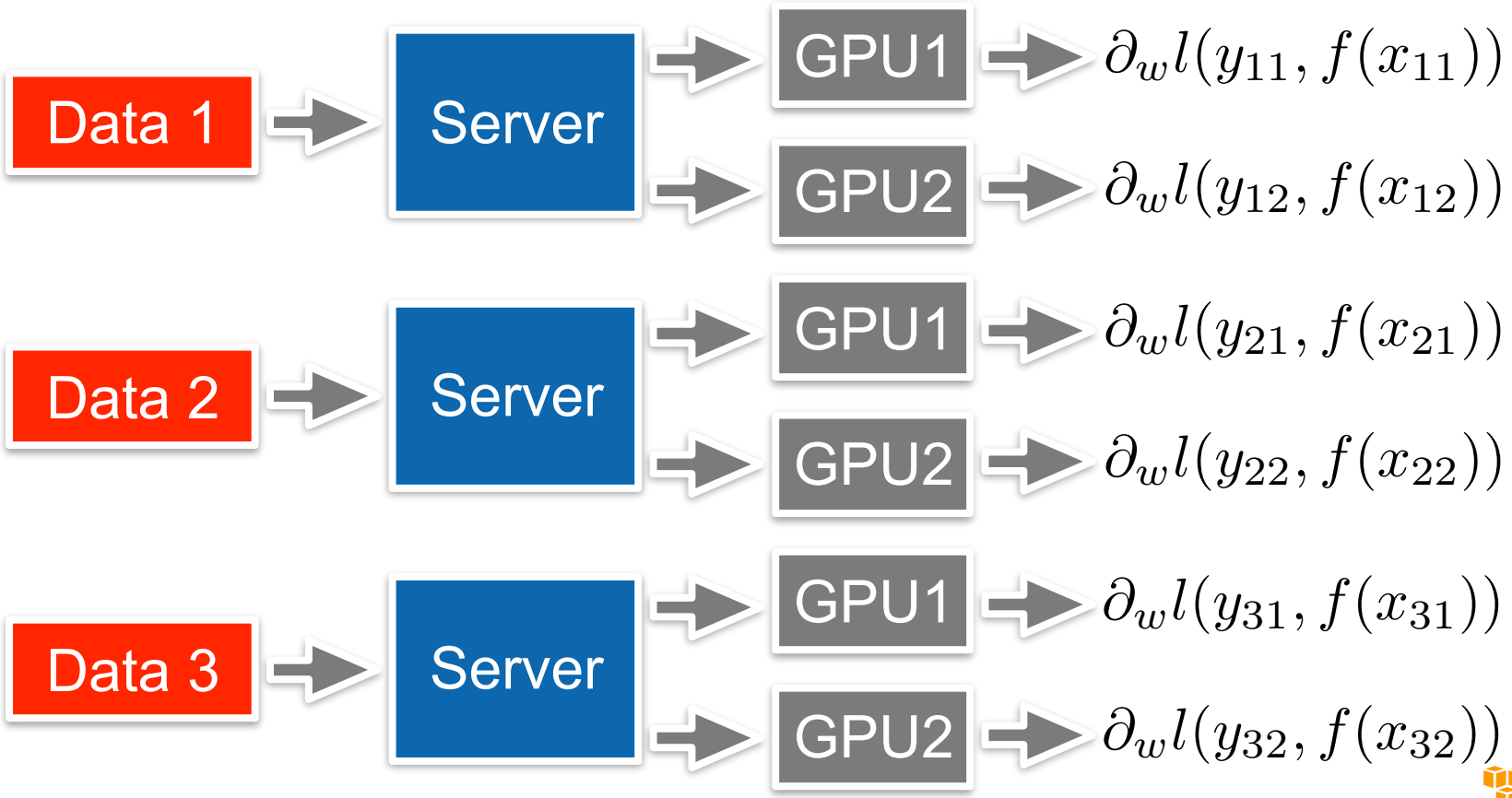
Parallelization - Pragmatic Strategy



Parallelization - Pragmatic Strategy



Parallelization - Pragmatic Strategy



Parallelization - Pragmatic Strategy

$$\partial_w l(y_{11}, f(x_{11}))$$

$$\partial_w l(y_{12}, f(x_{12}))$$

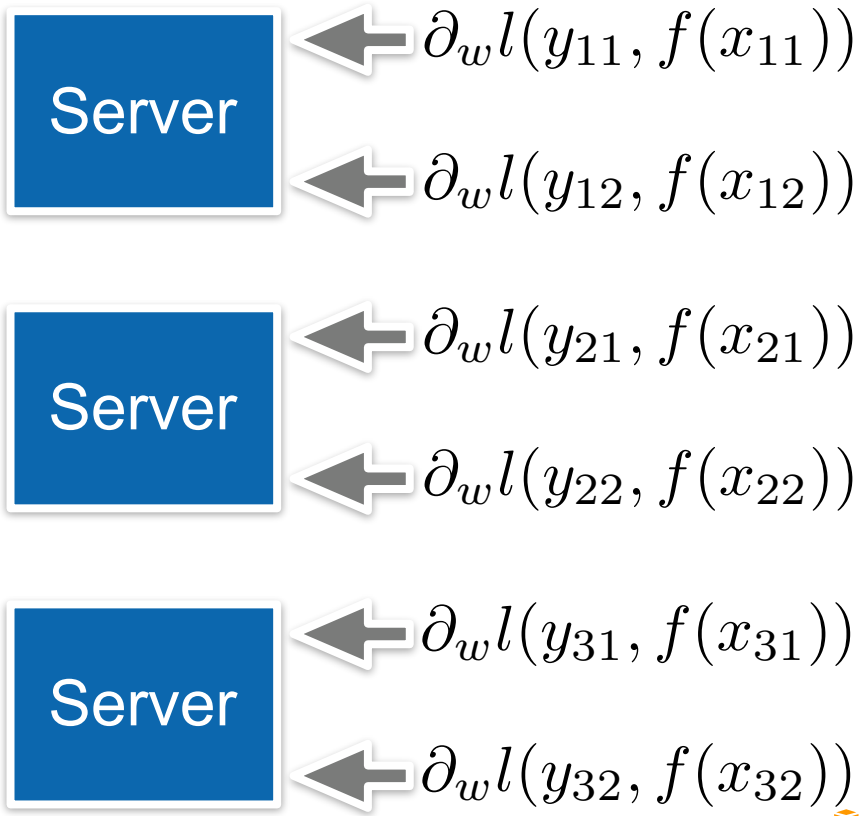
$$\partial_w l(y_{21}, f(x_{21}))$$

$$\partial_w l(y_{22}, f(x_{22}))$$

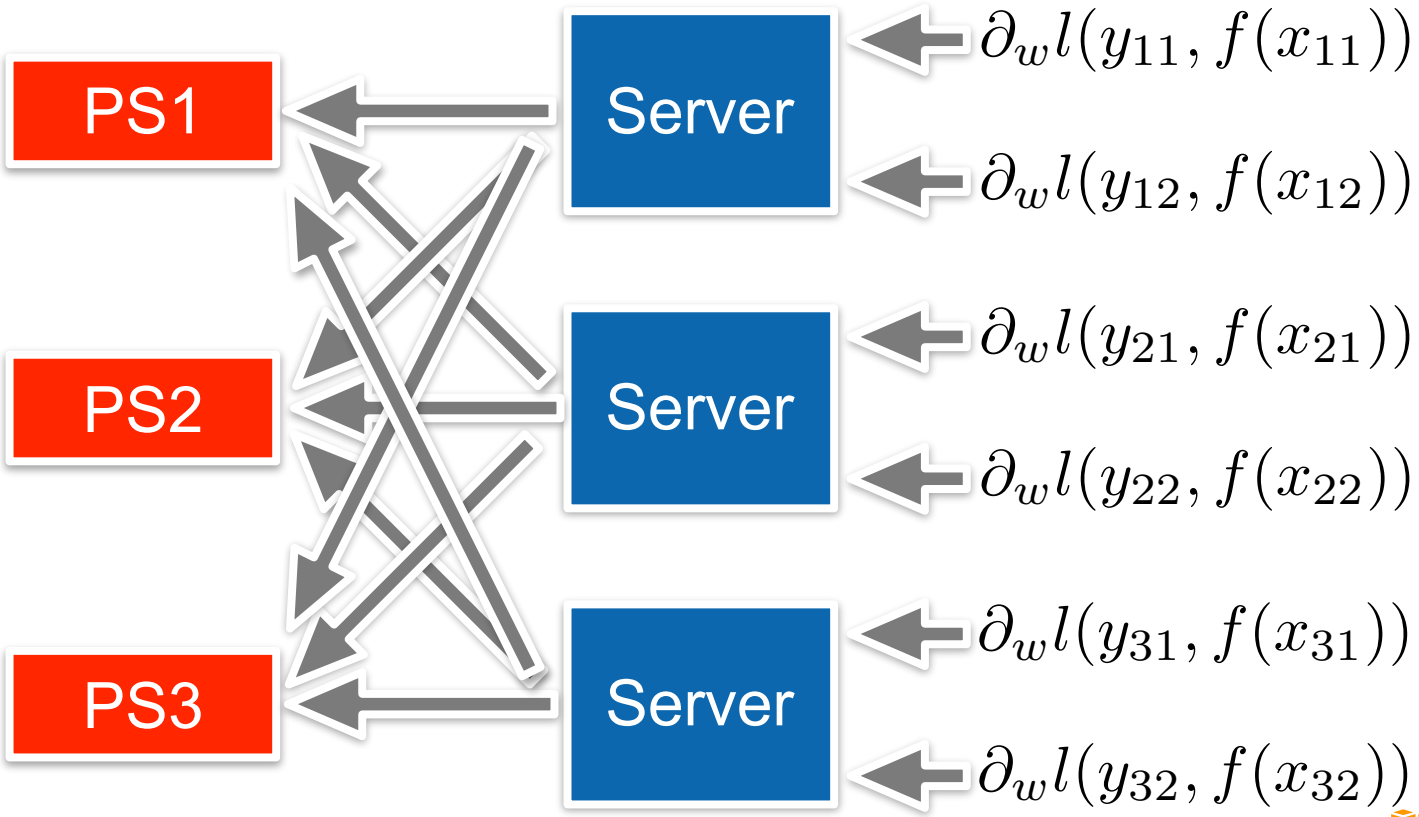
$$\partial_w l(y_{31}, f(x_{31}))$$

$$\partial_w l(y_{32}, f(x_{32}))$$

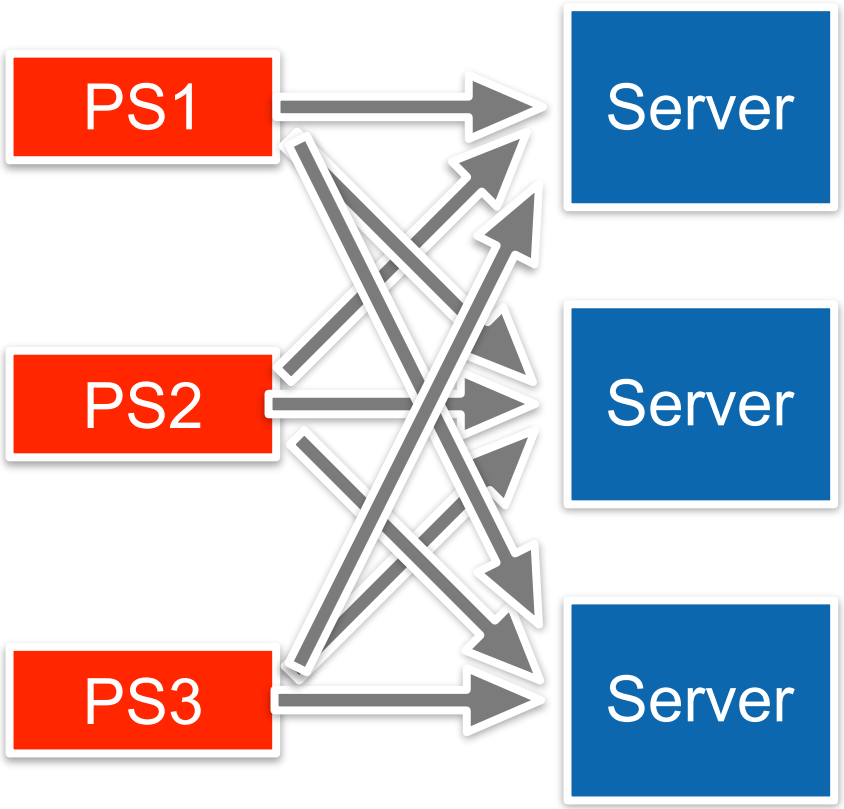
Parallelization - Pragmatic Strategy



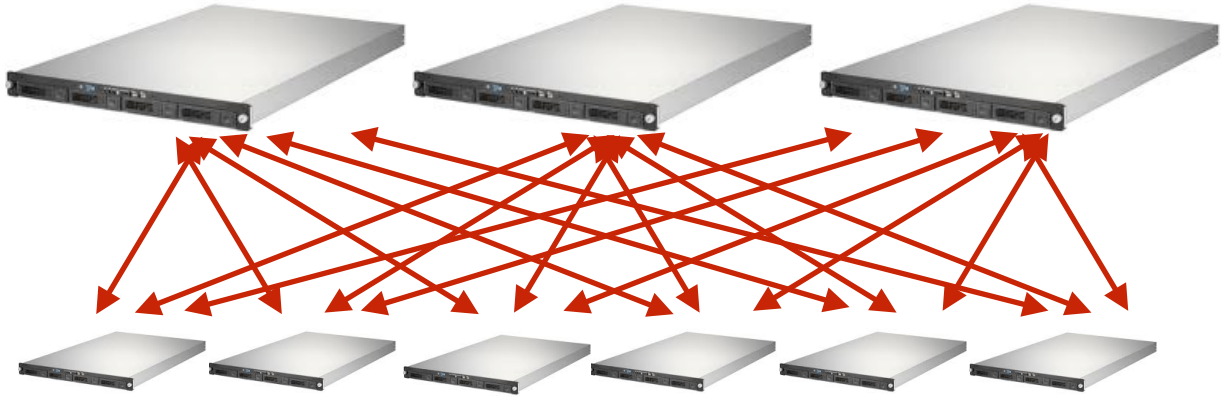
Parallelization - Pragmatic Strategy



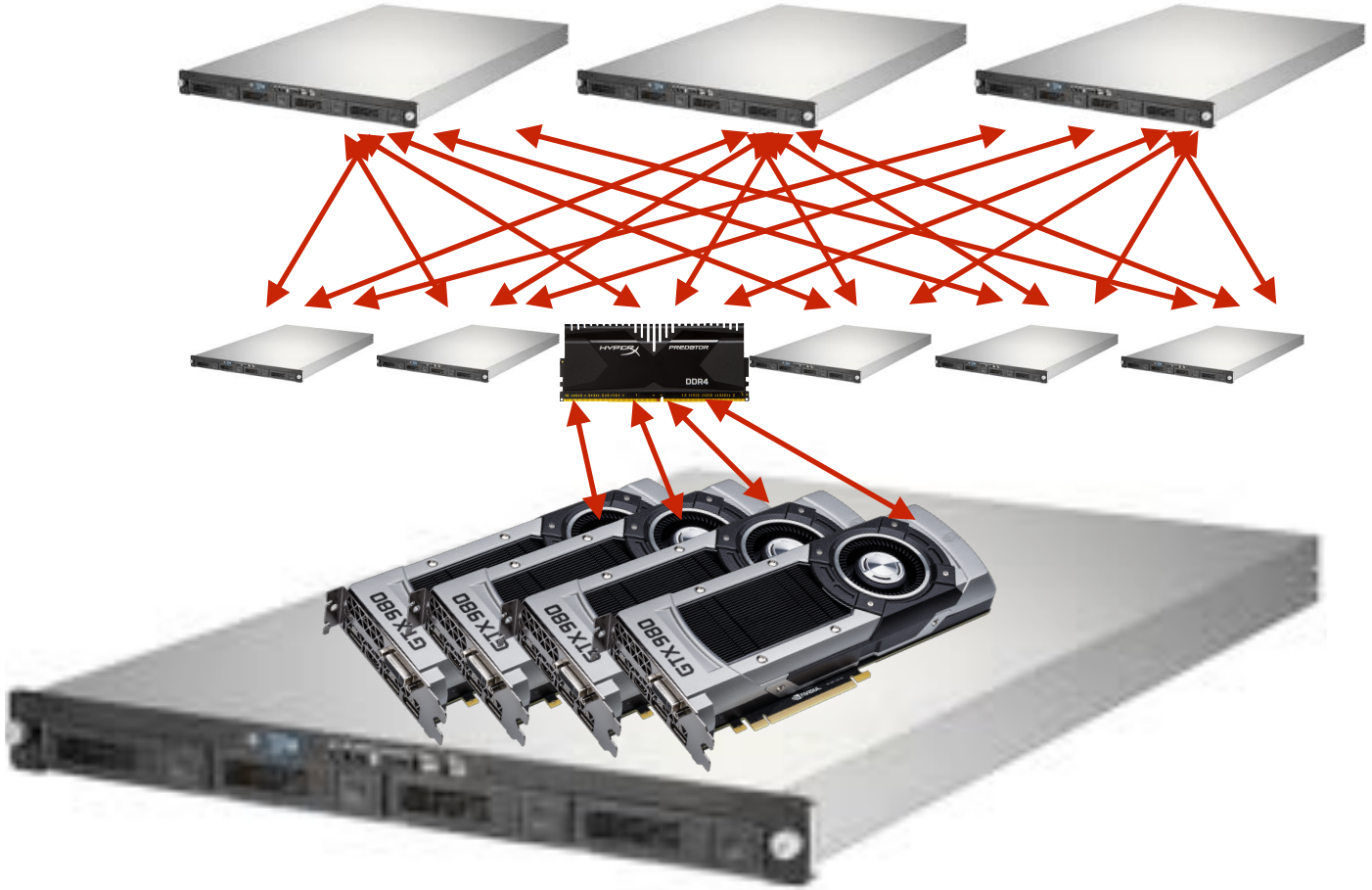
Parallelization - Pragmatic Strategy



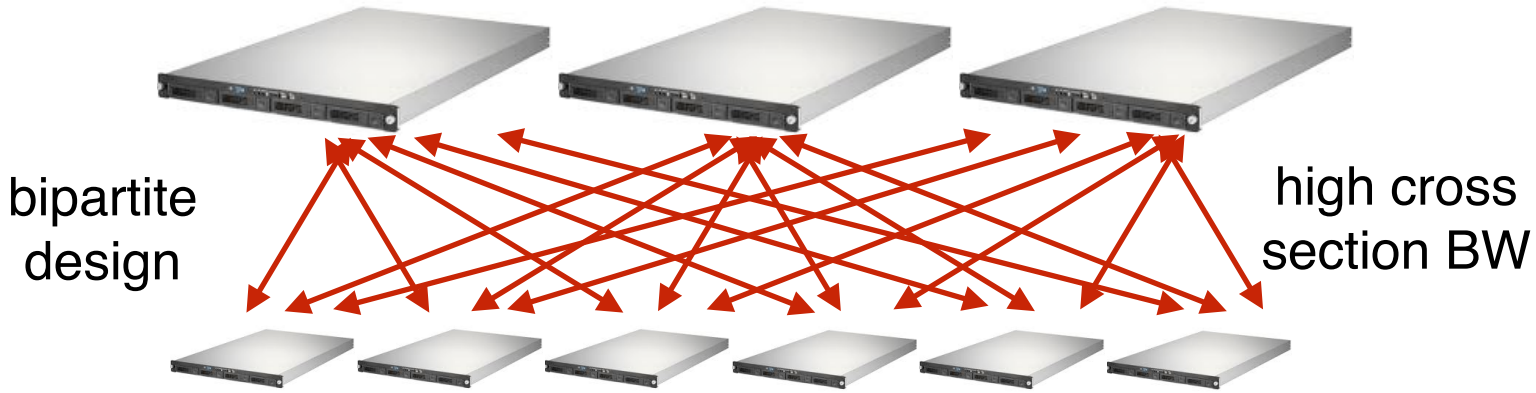
Parameter Server aka (key,value) store



Parameter Server aka (key, value) store



Parameter Server (hierarchical for DL)



Clients have local parameter view

Servers have shard of parameter space

Client-server synchronization

- Reconciliation protocol
- Synchronization schedule
- Load distribution algorithm

Smola & Narayanamurthy, 2010, VLDB
Gonzalez et al., 2012, WSDM
Dean et al, 2012, NIPS
Shervashidze et al., 2013, WWW
Google, Baidu, Facebook,
Amazon, Yahoo, Microsoft

put(keys,values,clock)
get(keys,values,clock)

amazon web services

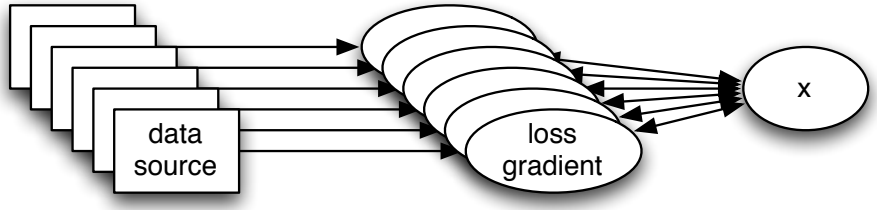
Distributed Training Learning Rates

Doing stupid things really fast

- **Single GPU**
  Small minibatch for training (16 to 64 observations).

$$g_B := \frac{1}{b} \sum_{i \in B} \partial_w l(y_i, f(x_i, w)) \text{ and } w \leftarrow \text{Optimizer}(w, g_B)$$

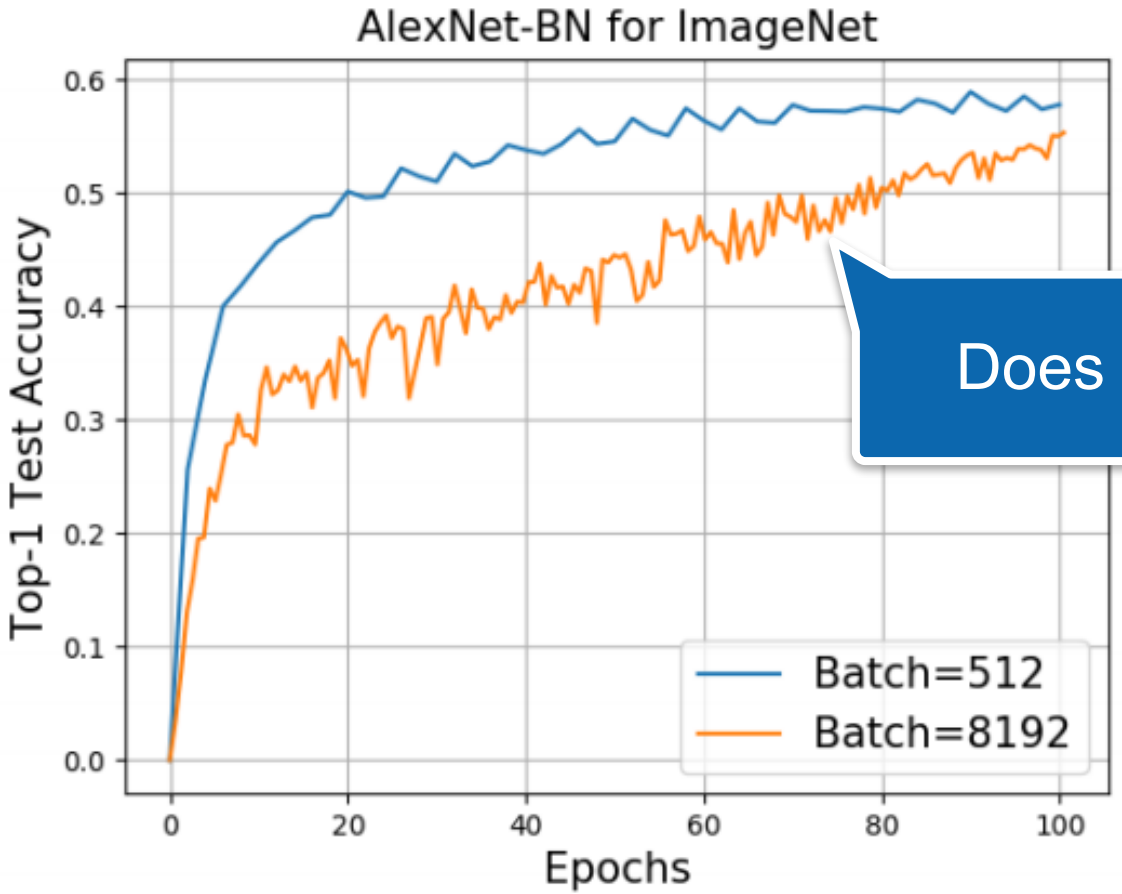
- **Multi GPU / Multi Servers**



  - What if all gradients look similar?
  - Averaging over them reduces variance $O\left(1/\sqrt{b}\right)$
- **Upscale gradients with mini batch size**

aws

Doing stupid things really fast



Does not work!

Fixing it (well motivated heuristics, mostly)

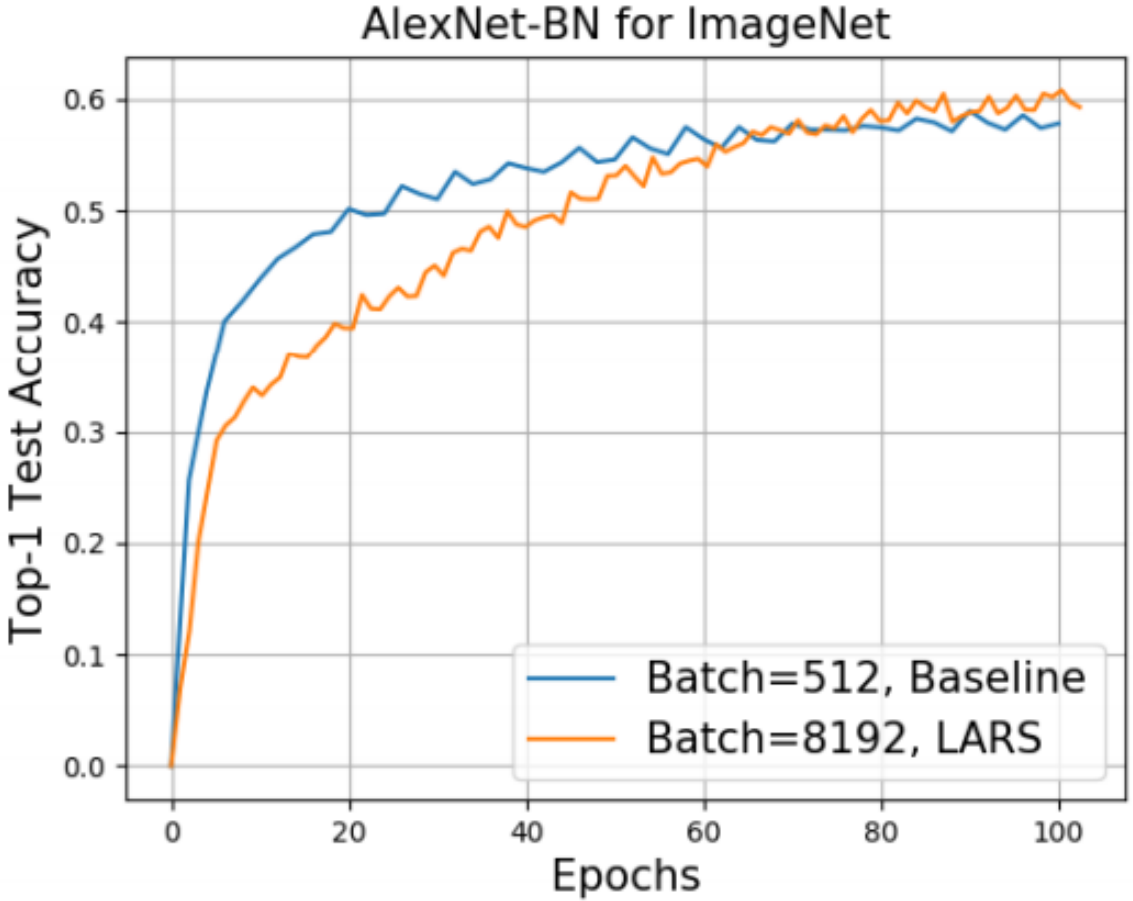
- **Scale up updates** by mini batch size (e.g. Li, 2016)

$$\frac{1}{|B|}\sum_{I\in B}\partial_w l(y_i, f(x_i, w)) \text{ replaced by } \sum_{I\in B}\partial_w l(y_i, f(x_i, w))$$

- **Warm up** to larger initial learning rate (Goyal et al., 2017)
- Batch Normalization doesn't work well when mini batches are too large. Rescale it (and no need to regularize)
- Layer-wise learning rate (LARS) (You et al., 2017)

$$\lambda_l = \frac{\|w_l\|}{\|g_l\| + \beta\|w_l\|}$$

Being smart at scale (You et al., 2017)



aws