
SHOCKWAVE

Hardware Accelerators for Training Deep Neural Networks

Ardavan Pedram and Kunle Olukotun

Funding for this Tutorial was provided by the National Science Foundation Division of Computing and Communication Foundations under award number 1563113.

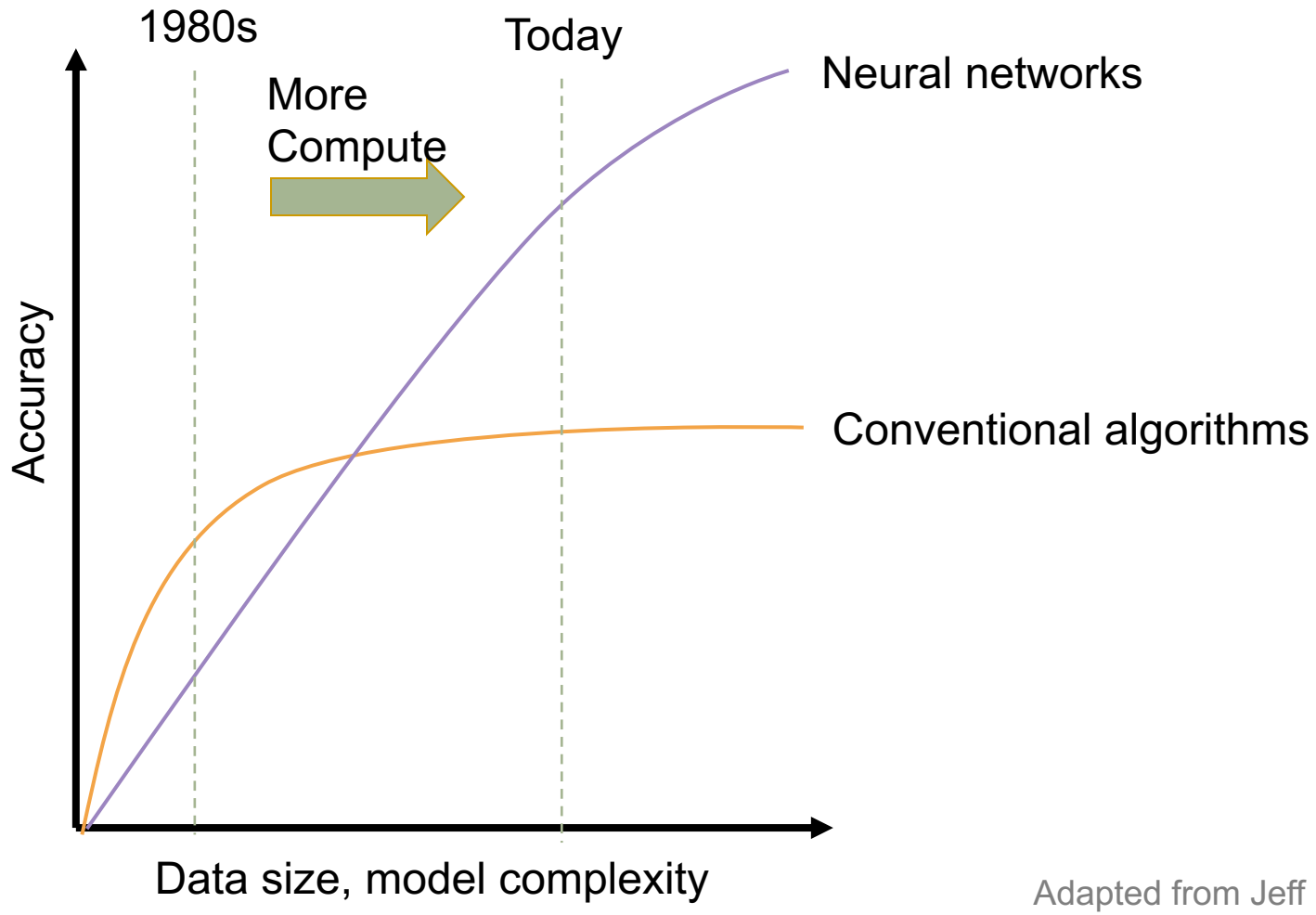
Two Big Trends in Computing

- Success of Machine Learning
  - Incredible advances in image recognition, natural language processing, and knowledge base creation
  - Society-scale impact: autonomous vehicles, scientific discovery, and personalized medicine
  - Insatiable computing demands for training and inference

- Moore's Law is slowing down
  - Dennard scaling is dead
  - Computation is now limited by power
  - Conventional computer systems (CPU) stagnate

Demands a new approach to designing computer systems for ML

The Rise of Machine Learning



Adapted from Jeff Dean
HotChips 2017

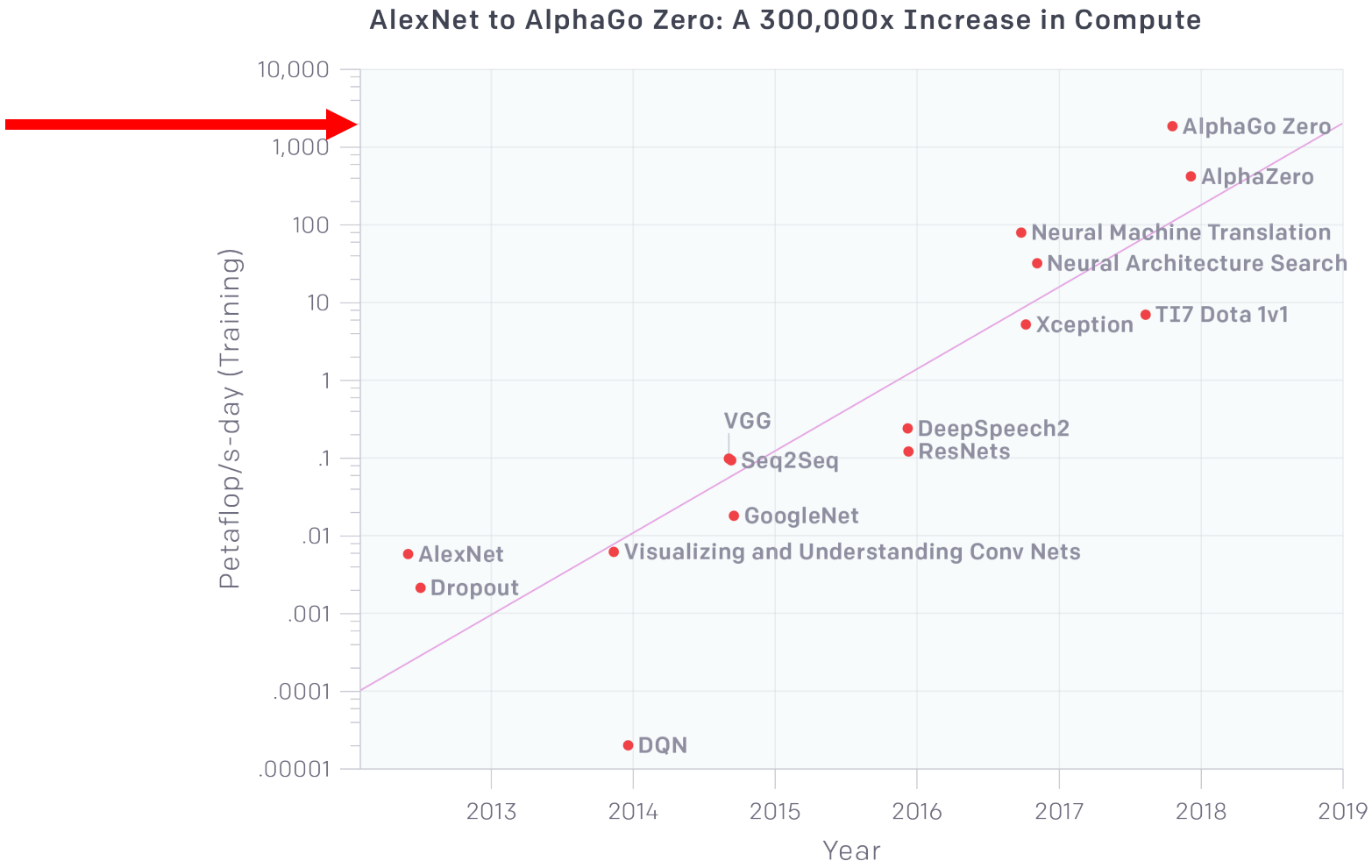
ML Training is Limited by Computation

From EE Times – September 27, 2016

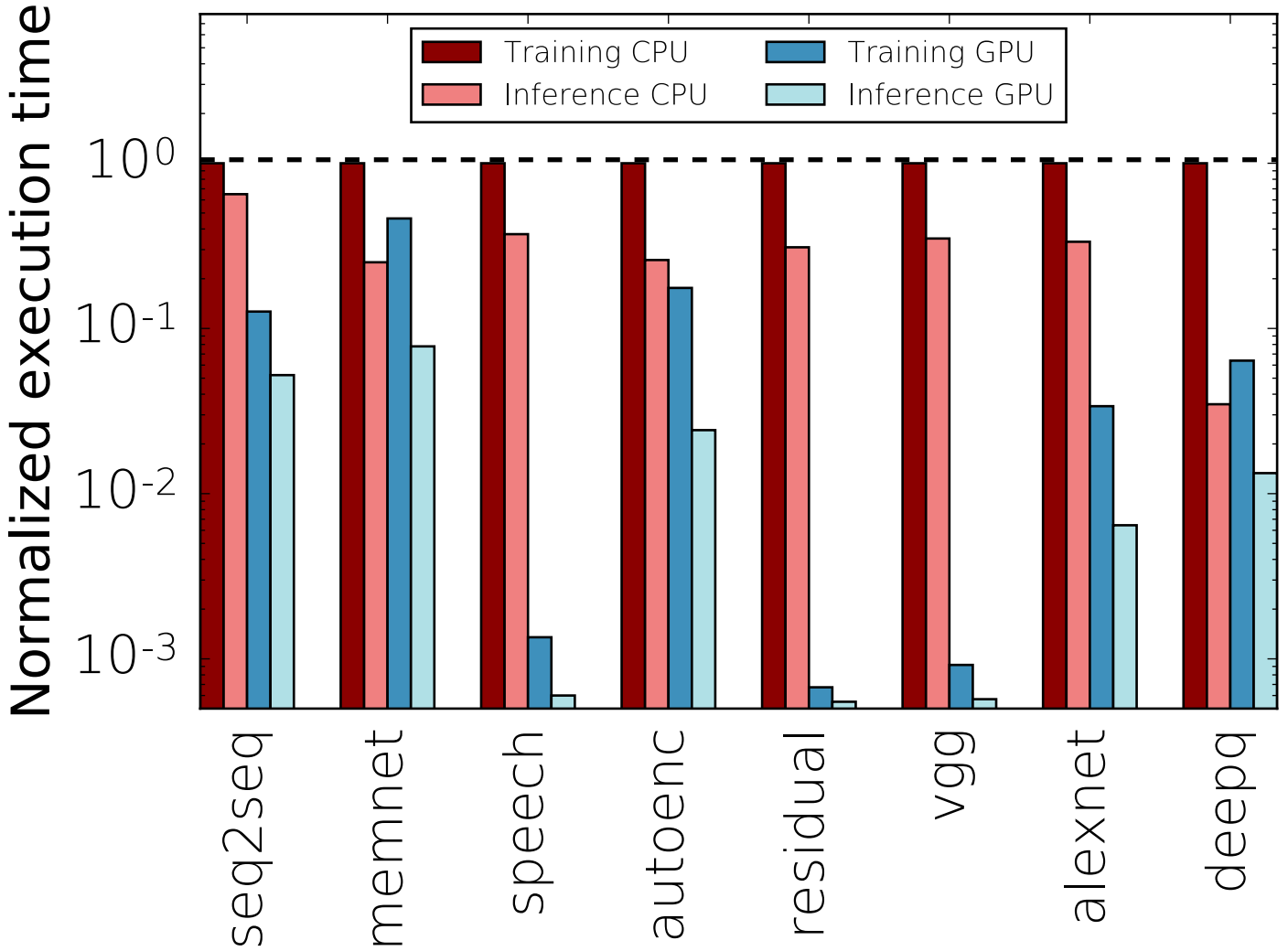
"Today the job of training machine learning models is limited by compute, if we had faster processors we'd run bigger models...in practice we train on a reasonable subset of data that can finish in a matter of months. We could use improvements of several orders of magnitude – 100x or greater."

Greg Diamos, Senior Researcher, SVAIL, Baidu

AI and Compute

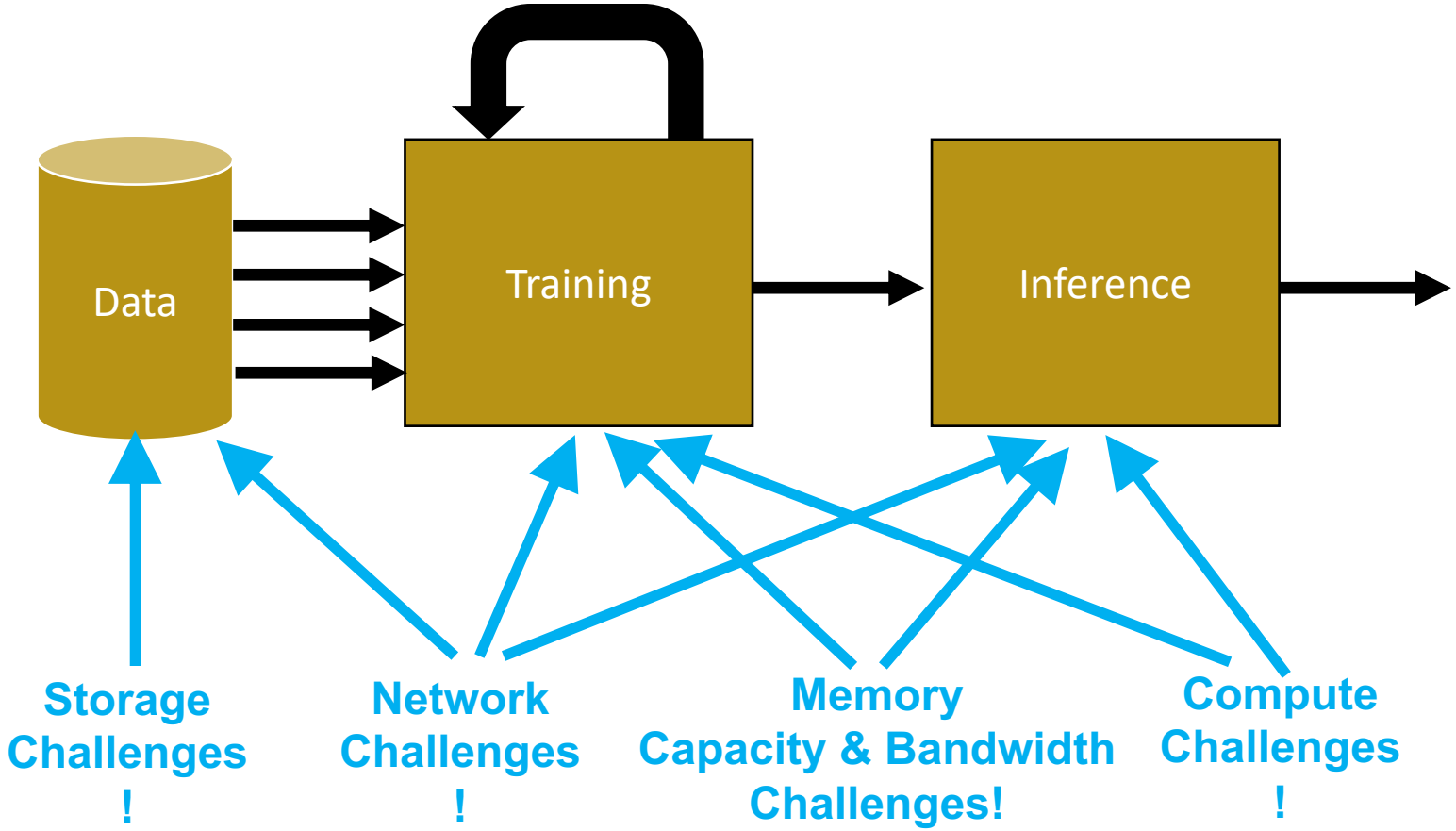


Why Training is Important?

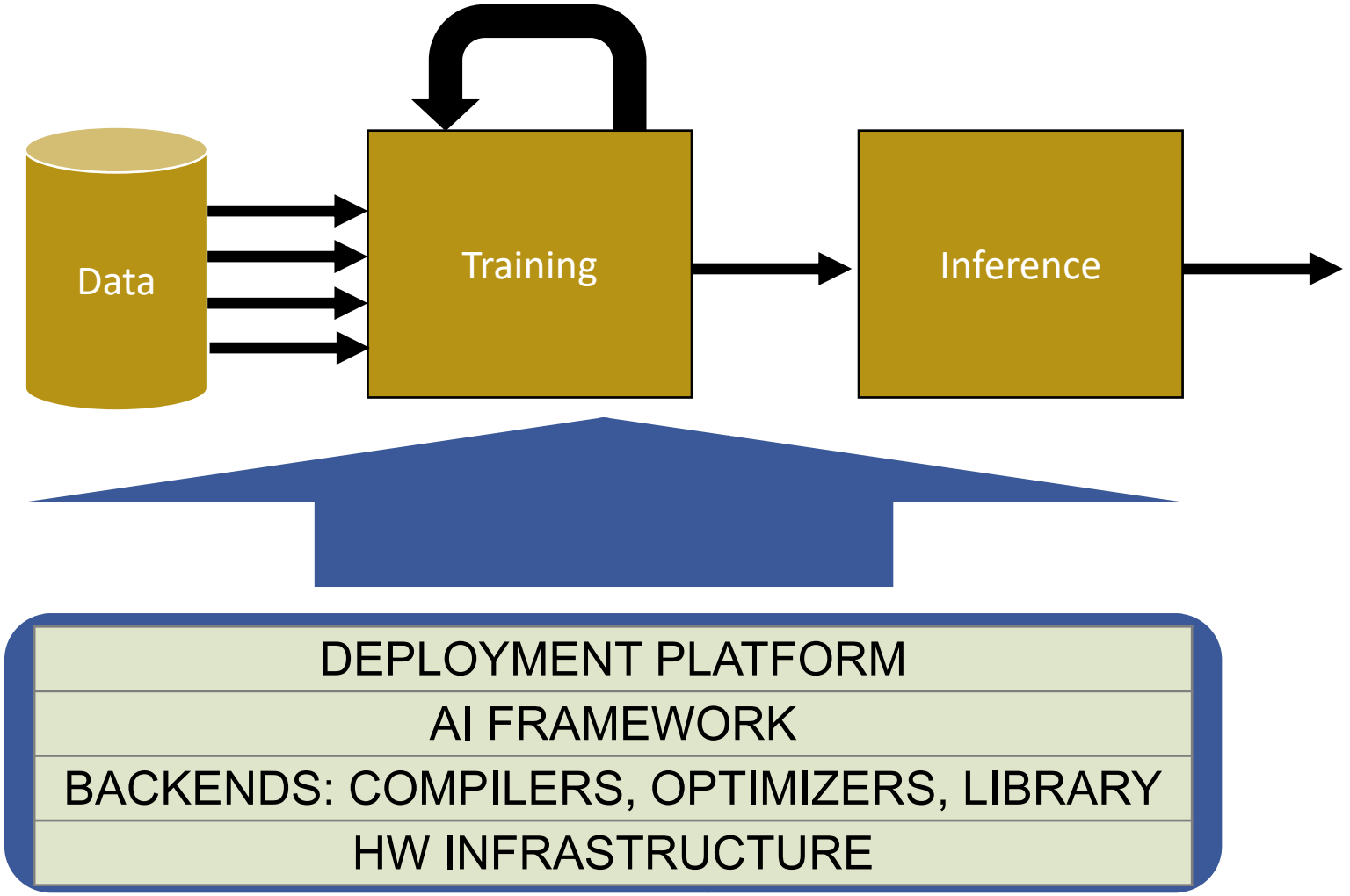


https://arxiv.org/abs/1608.06581

Infrastructure Challenge



What Lies Beneath?



Source: Mikhail Smelyanskiy

Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Training

1. Fundamentals of Training
   - Gradient Descent
   - Backpropagation
   - Stochastic Gradient Descent (SGD)
   - Mini Batch SGD
   - Optimization
   - Generalization
   - Momentum
   - Learning Rate
   - Misc.
2. Computation of training
   - Data Dependency
   - Parallelism
   - Memory requirements
3. Normalization
   - Batch Norm
   - Layer Norm
   - Group Norm
   - Stream Norm
   - Online Norm

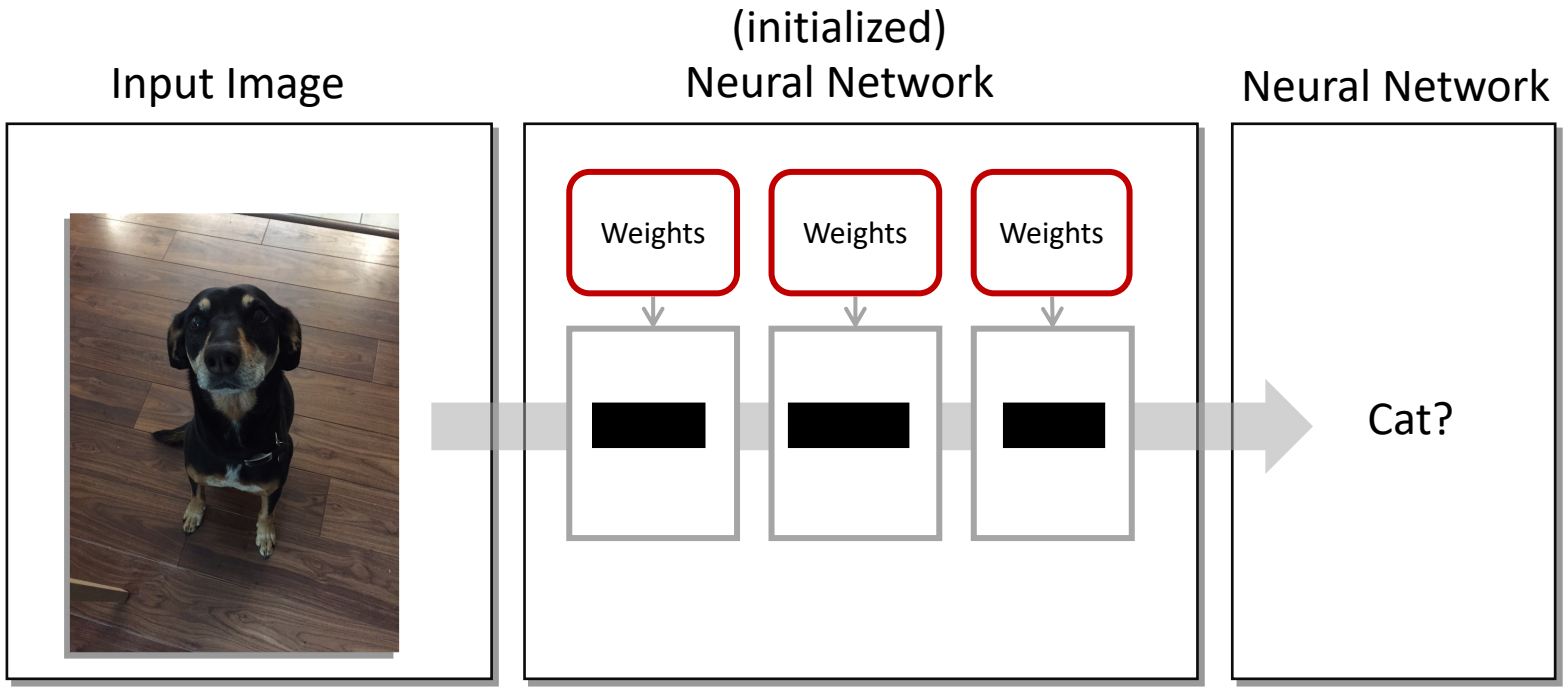
Training

4. Low/Mixed Precision training
   - Floating-point representations
   - Loss Scaling
   - 8-bit Floating-Point Training
5. Sparsity and training
   - Pruning
   - Sparse Training
   - Structural Sparse Training
6. Scaling training
   - Scaling Computation power
   - Scaling Network Size
   - Large Batch Training
   - Asynchronous Training
7. Accelerators for Training
8. Benchmarks
9. Conclusion and Q&A

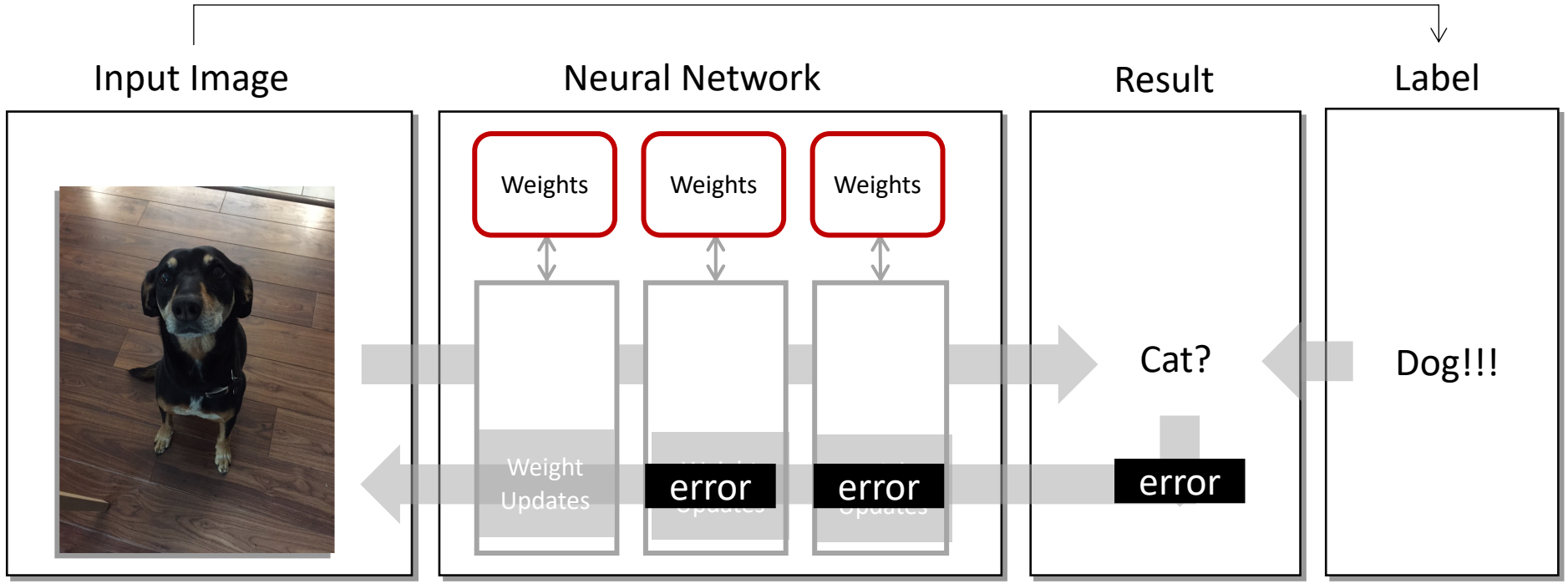
Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Recap


Input Image
(initialized)
Neural Network
Neural Network
Weights
Weights
Weights
Cat?

Recap

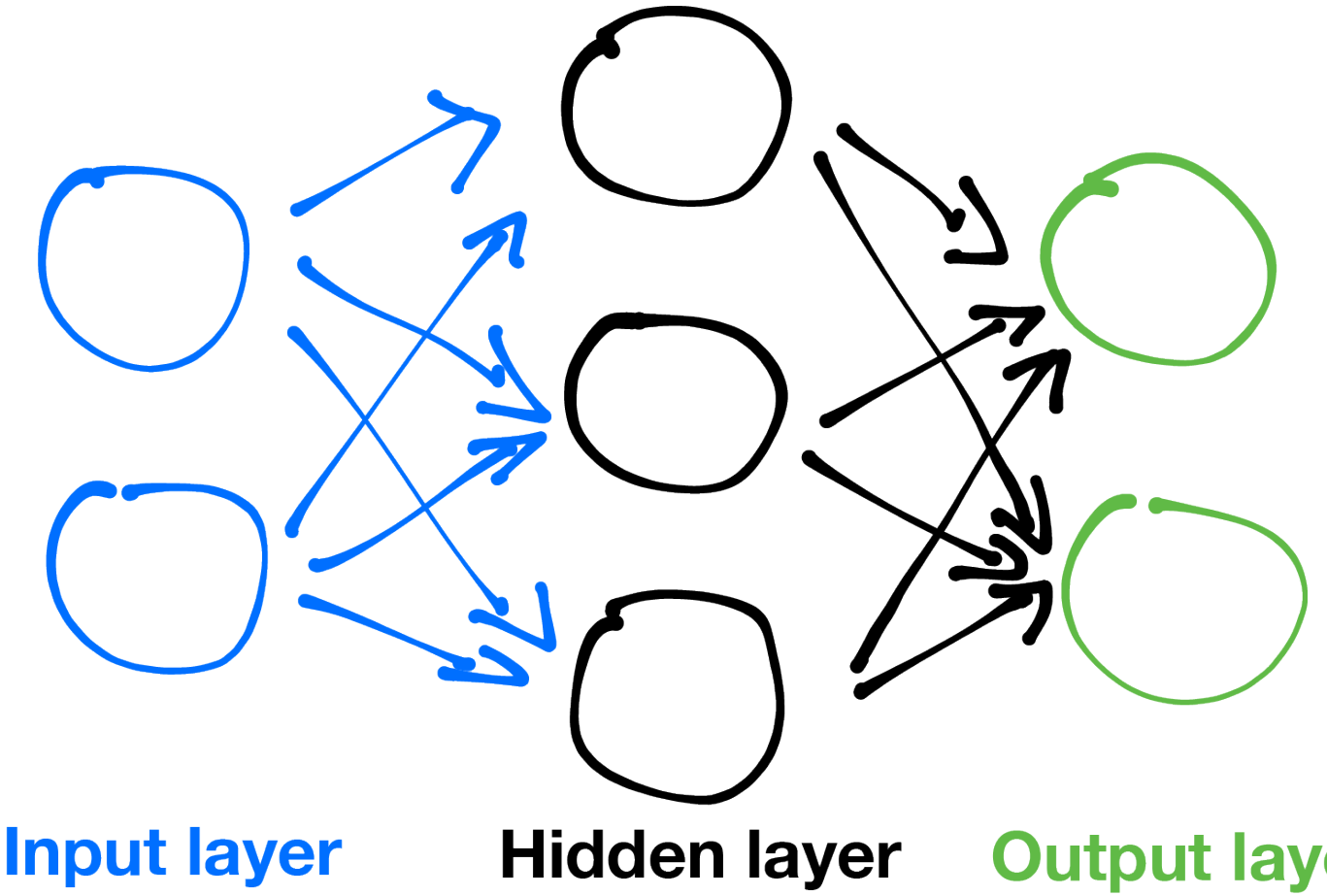

Input Image
Neural Network
Result
Label
Weights
Weights
Weights
Cat?
Dog!!!
Weight Updates
error
error
error

How Familiar are You with Backpropagation?

- ❑ I can derive it by heart for MLP
- ❑ I can derive it by heart for CNNs/RNNs
- ❑ Somewhat familiar
- ❑ Just have heard about it

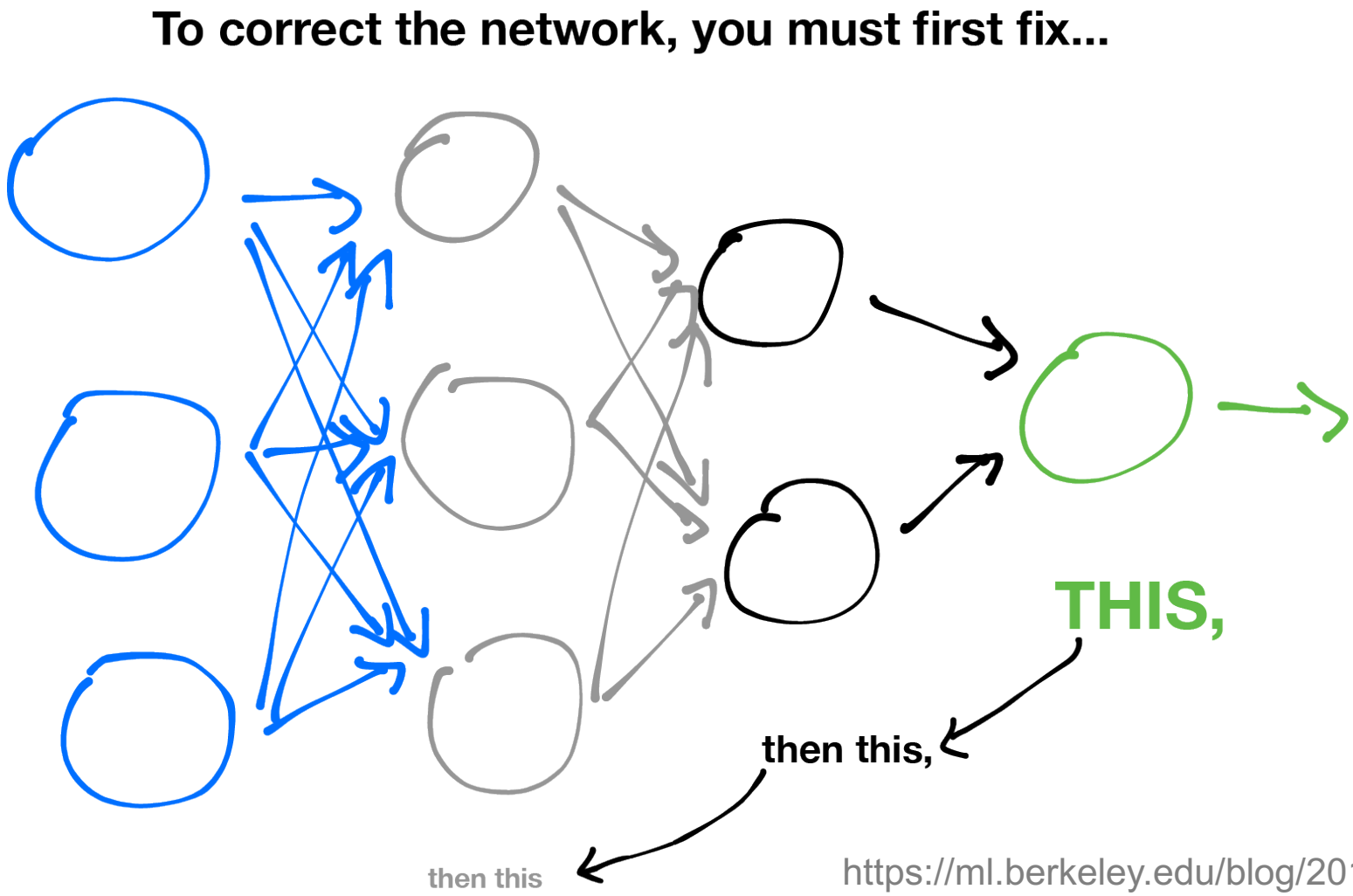
Recap: Neural Network

- What is the basic operation in each layer?



https://ml.berkeley.edu/blog/2017/02/04/tutorial-3/

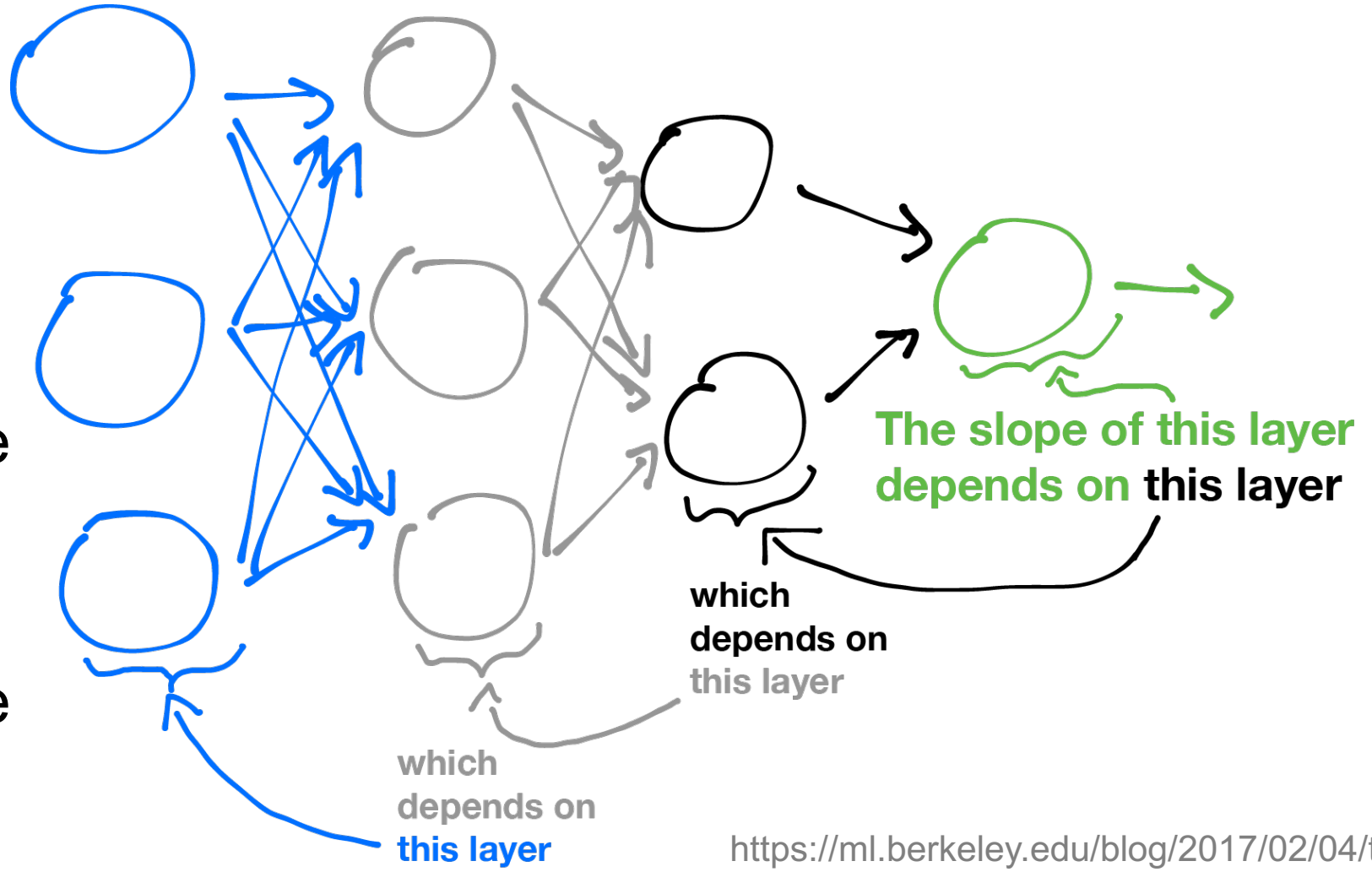
Propagate The Error Backwards



https://ml.berkeley.edu/blog/2017/02/04/tutorial-3/

Backpropagation

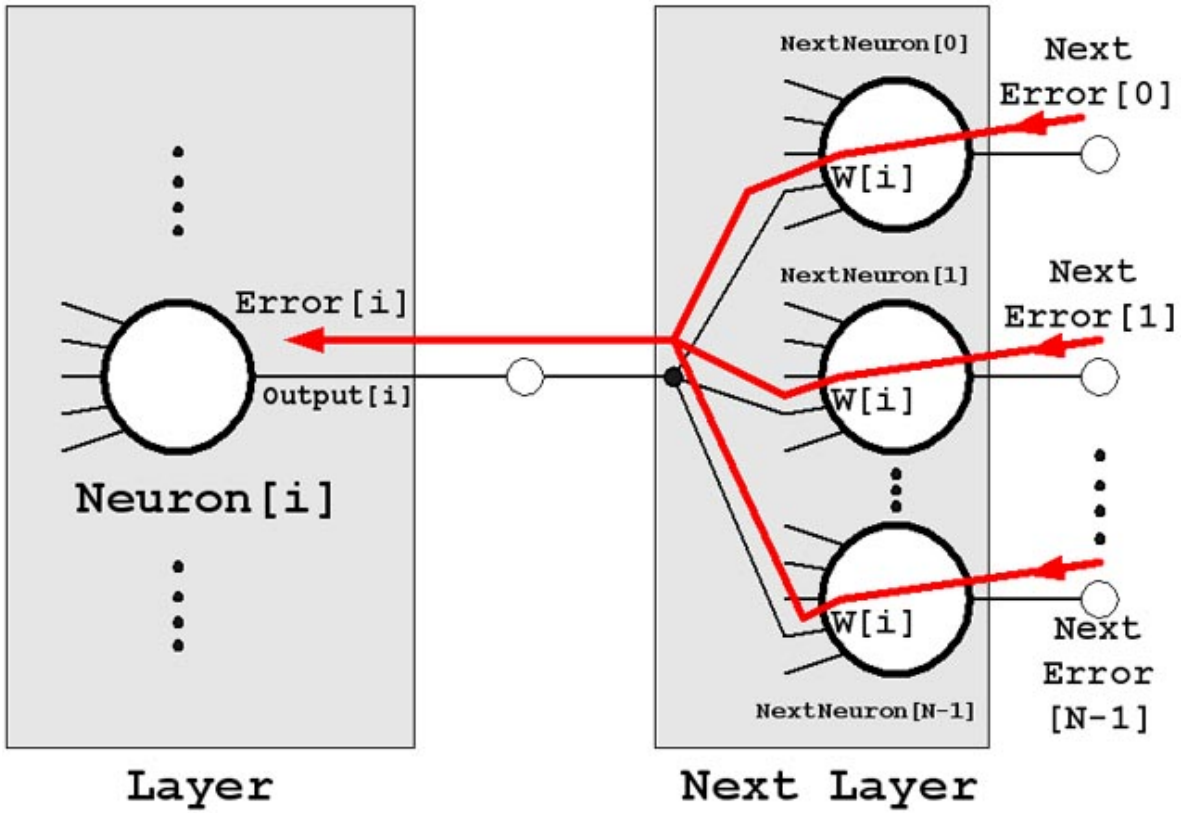
- What is the slope?
- What is the operation?



https://ml.berkeley.edu/blog/2017/02/04/tutorial-3/

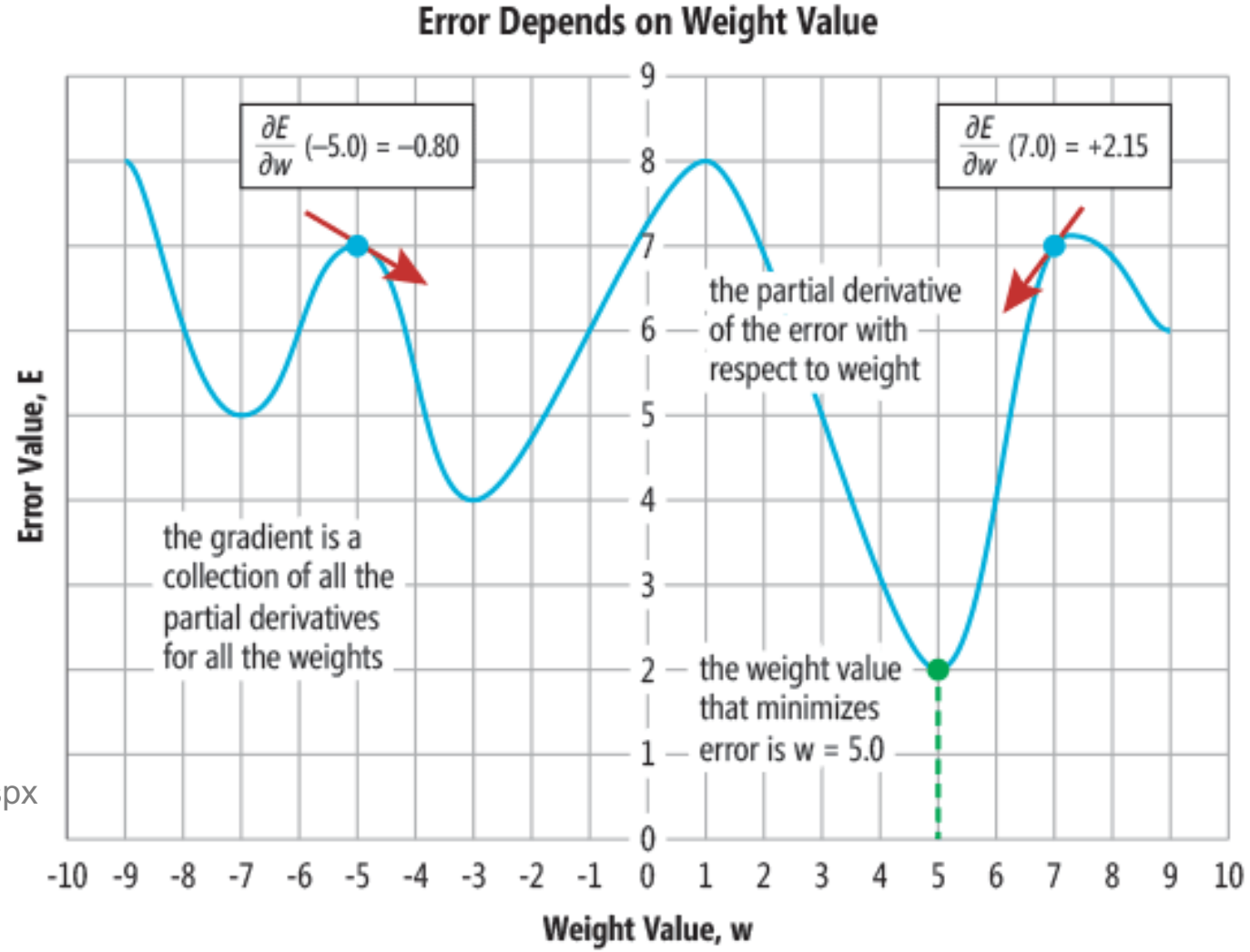
Backpropagation

- Any Similarities
  with Forward pass?



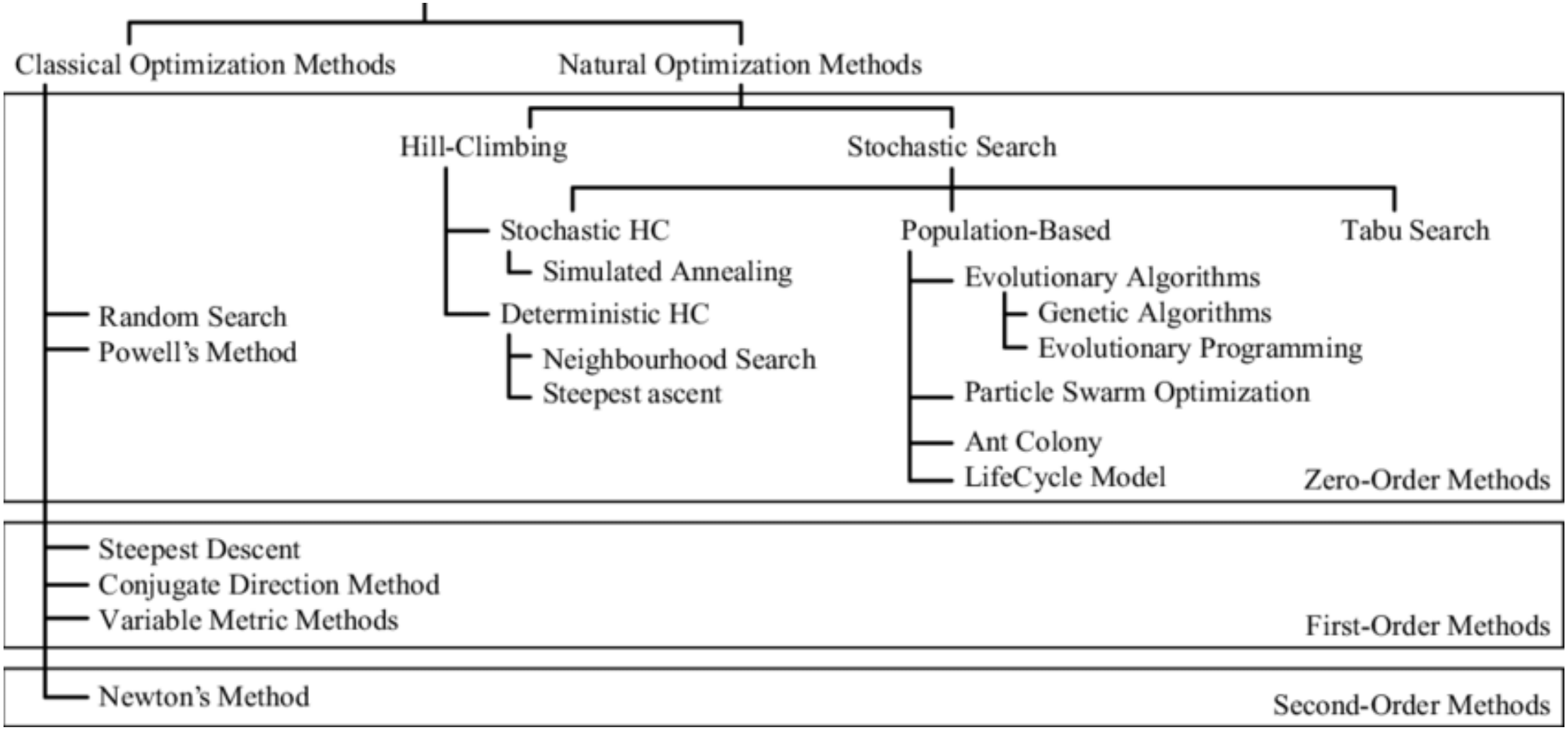
Gradient Descent

- What is the goal here?

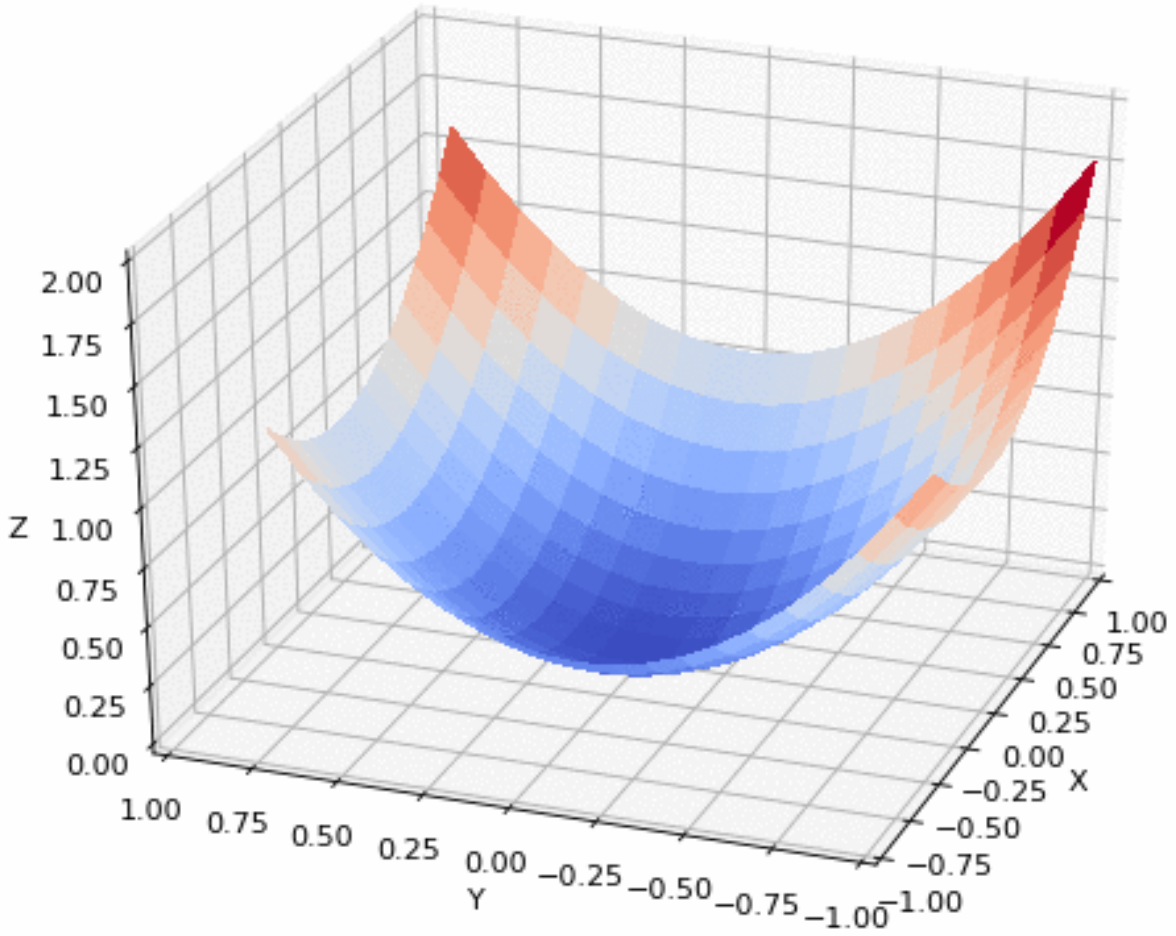


https://msdn.microsoft.com/en-us/magazine/dn913188.aspx

How Do We Solve This Problem?



Find The Minimum



Sebastian Ruder, “An overview of gradient descent optimization algorithms” arXiv:1609.04747 15 Jun 2017
https://arxiv.org/pdf/1609.04747.pdf

Optimization Methods

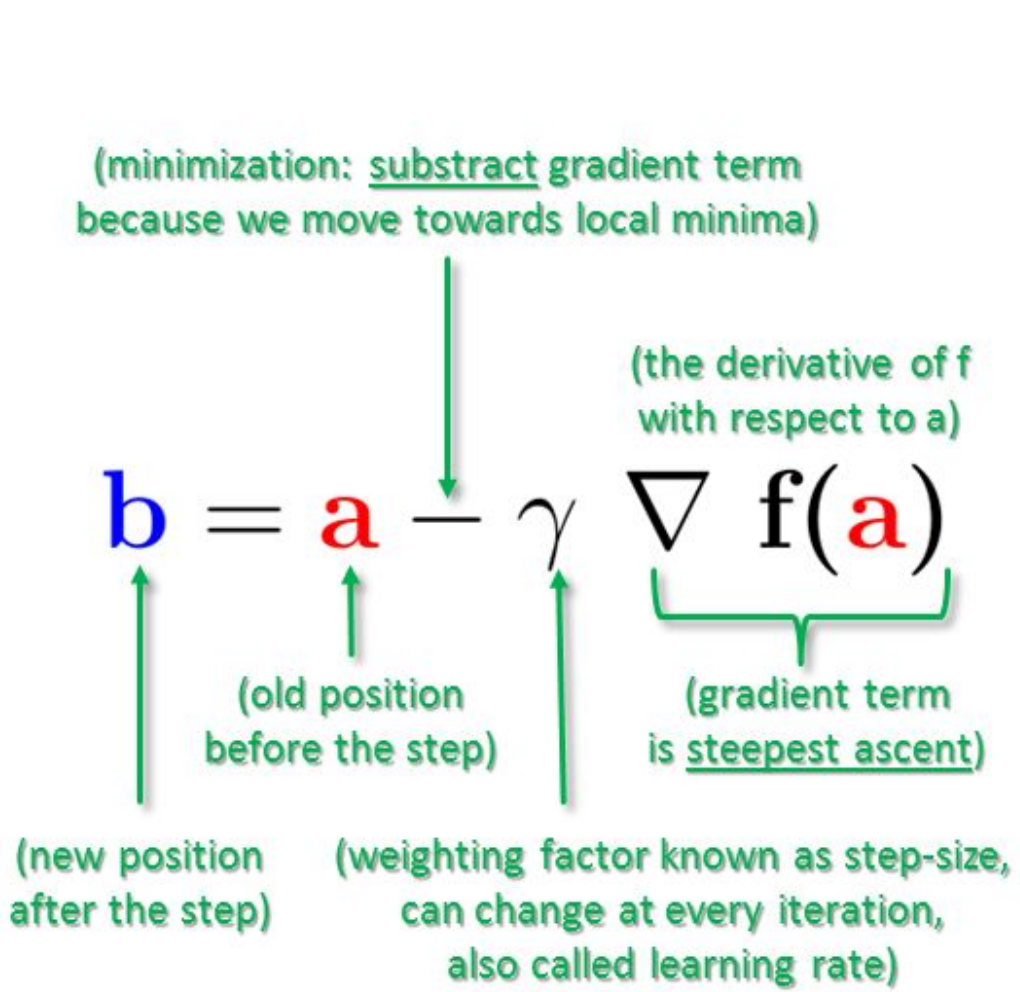
- First-order Methods
  - Approximate f(x) as a plane:
  - Gradient descent (+ various tweaks)

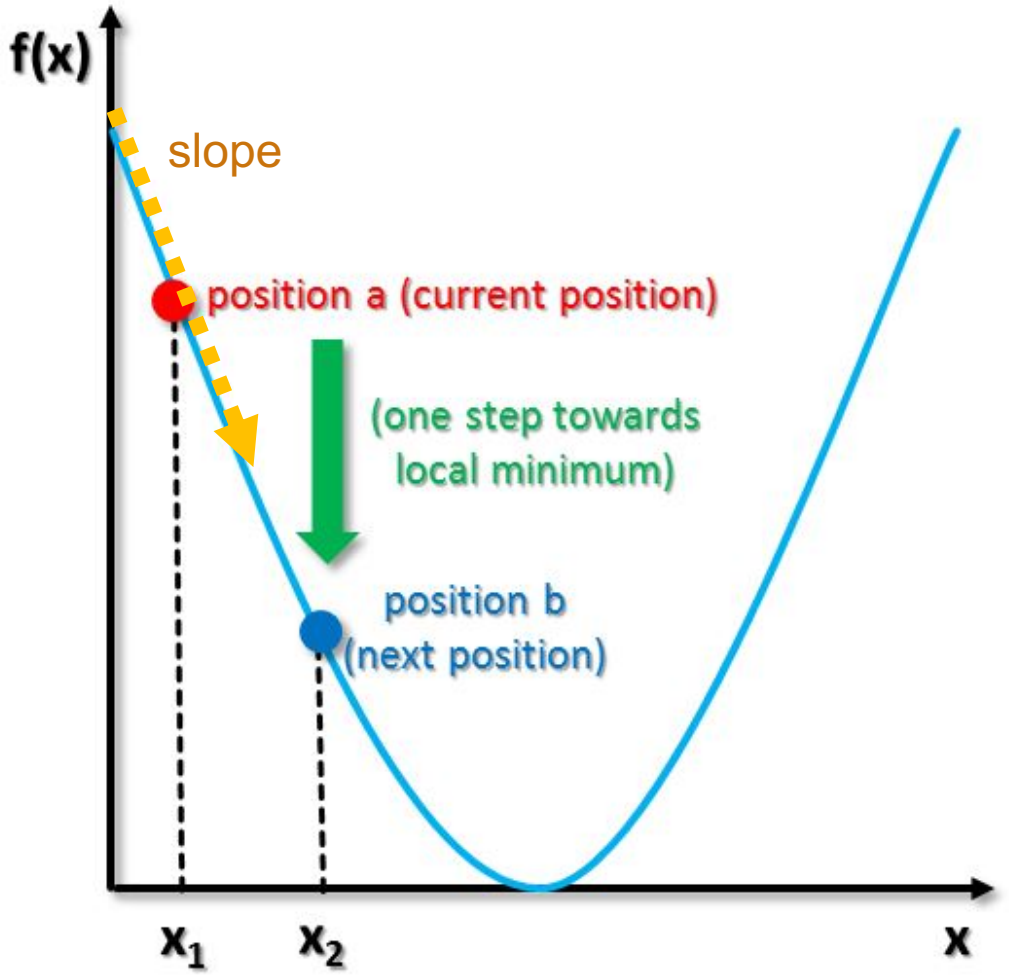
$$\hat{f}(w) = w^T b + c$$

- Second-order Methods
  - Approximate f(x) as quadratic form:
  - Conjugate gradient – "correct" the gradient to avoid undoing work
  - Newton's method – use second derivatives to move directly towards optimum
  - Quasi-Newton methods – approximate Newton's method using successive gradients to estimate curvature

$$\hat{f}(w) = \frac{1}{2} w^T A w - w^T b + c$$

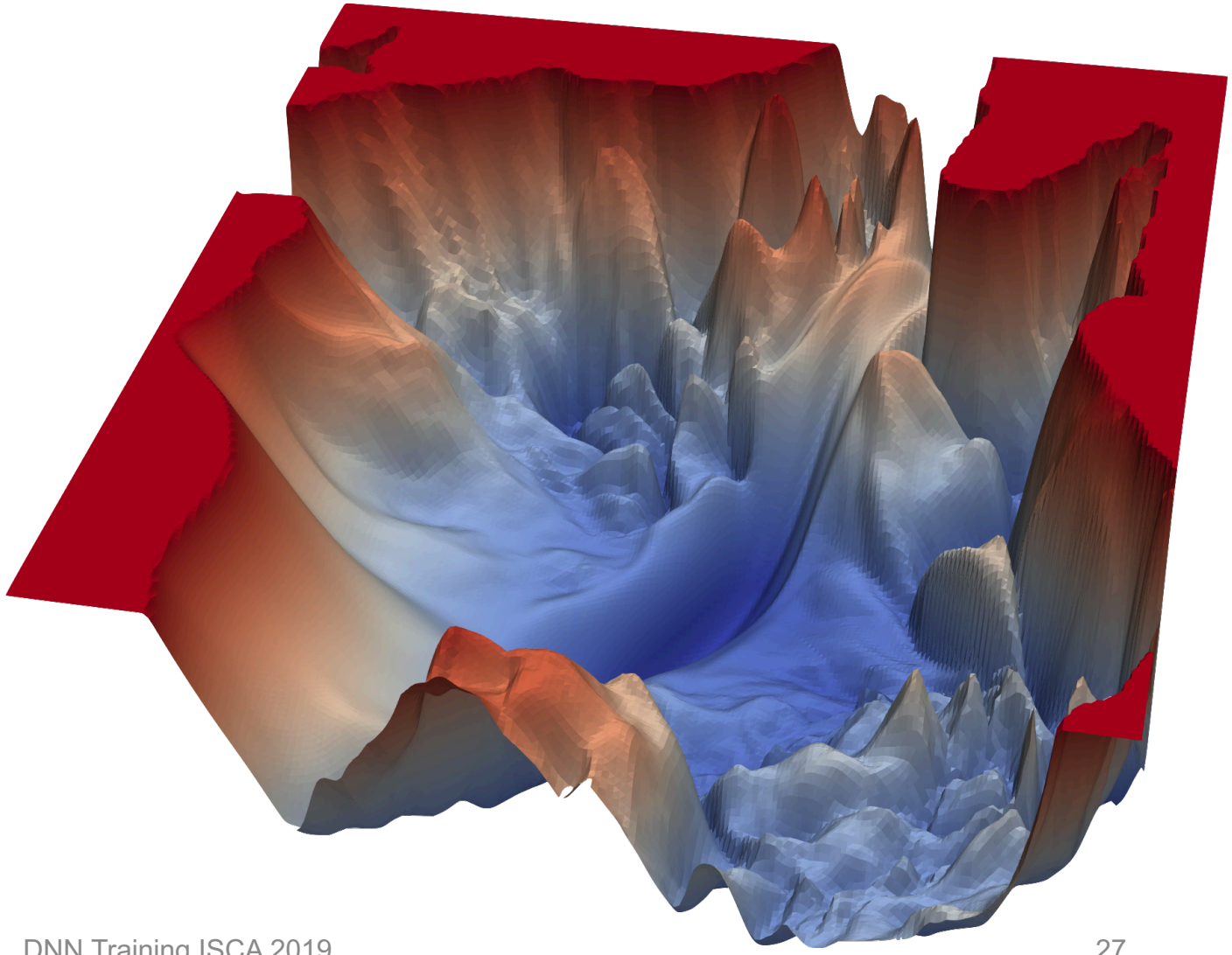
Gradient Descent





http://www.big-data.tips/wp-content/uploads/2016/06/formula-gradient-descent.jpg

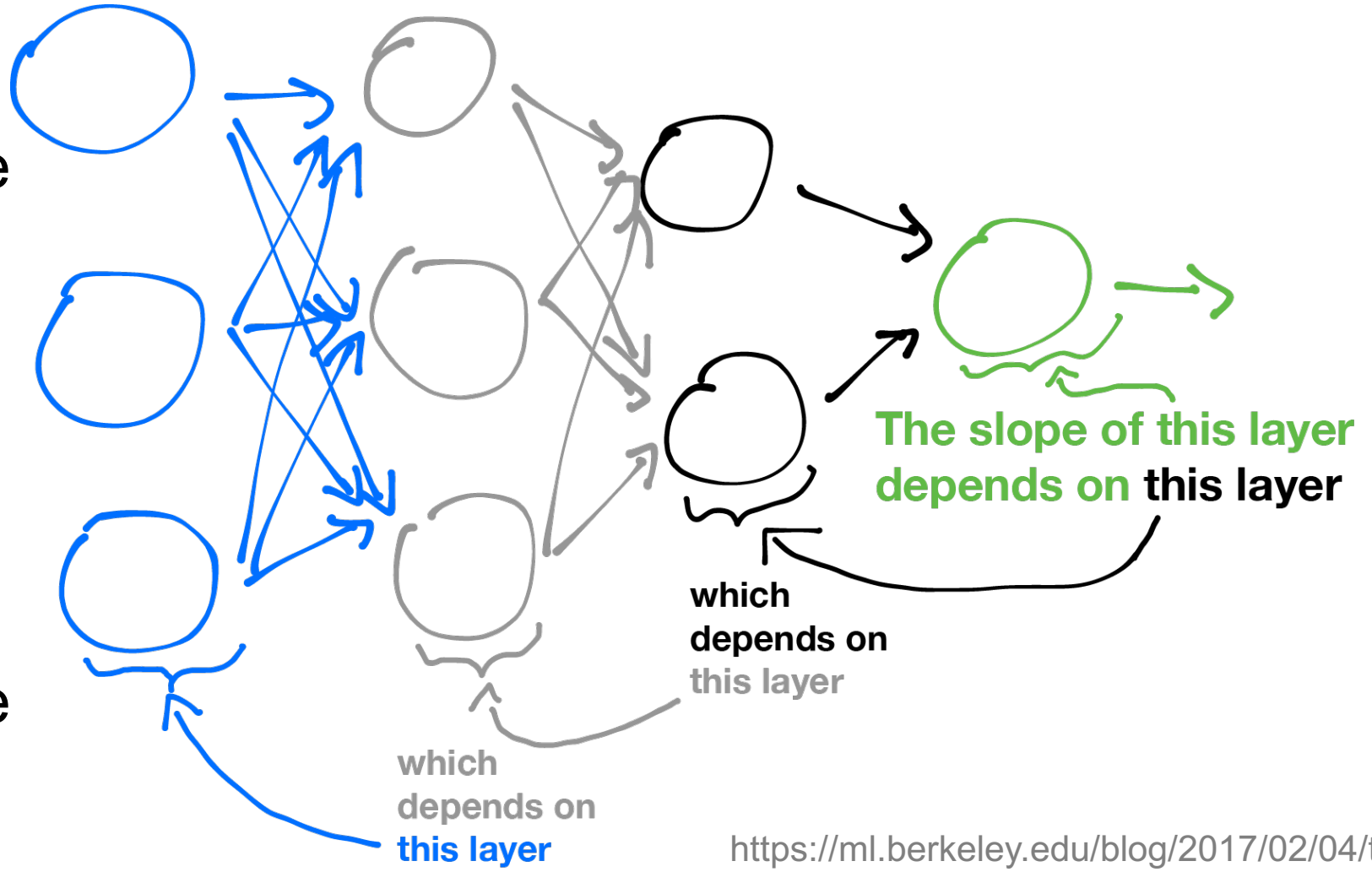
Gradient Surface



H. Li et. al,
https://www.cs.umd.edu/~tomg/projects/landscapes/

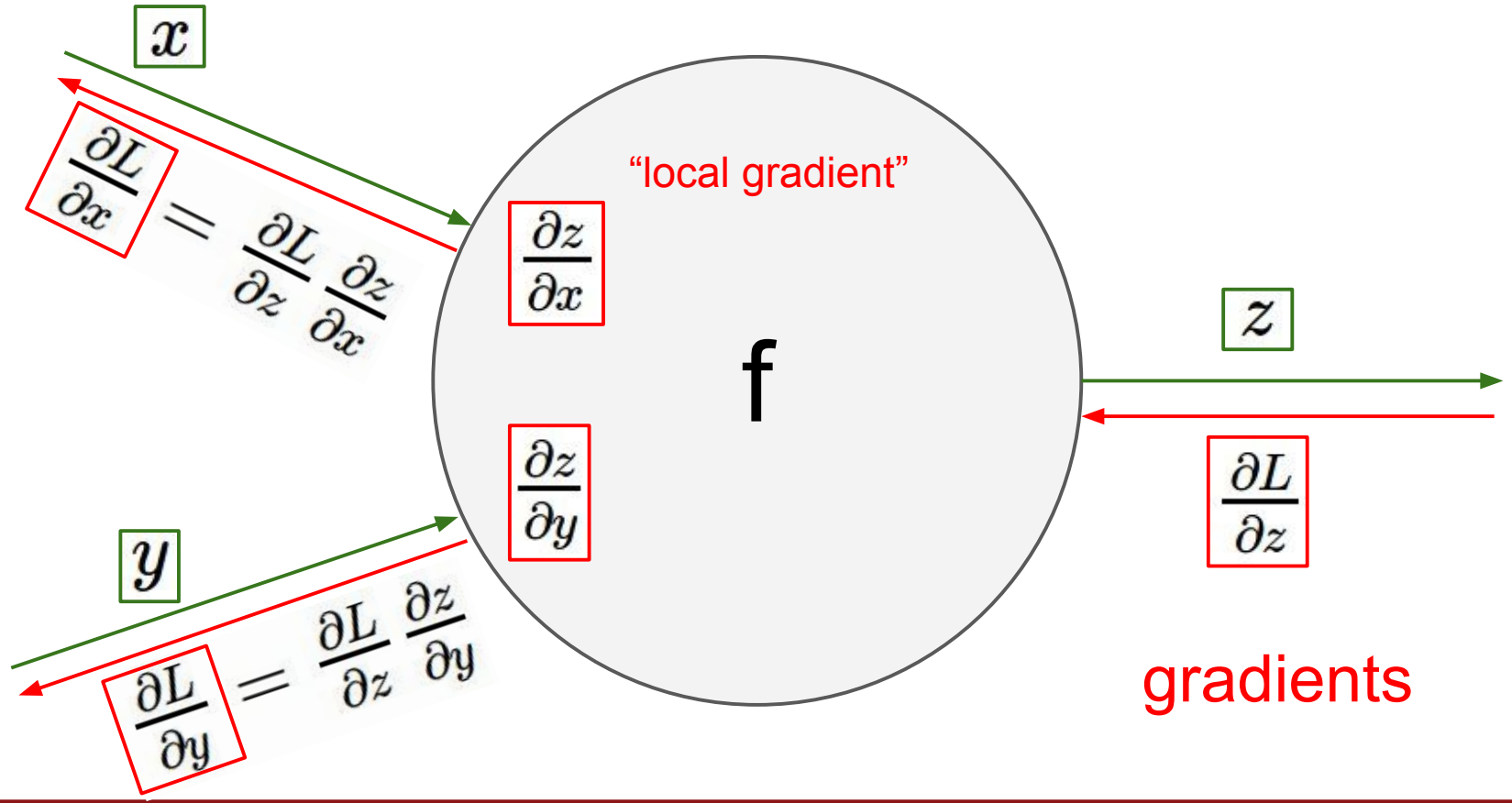
Back to… Backpropagation

- What is the slope?
- Chain rule?
- What is the operation?

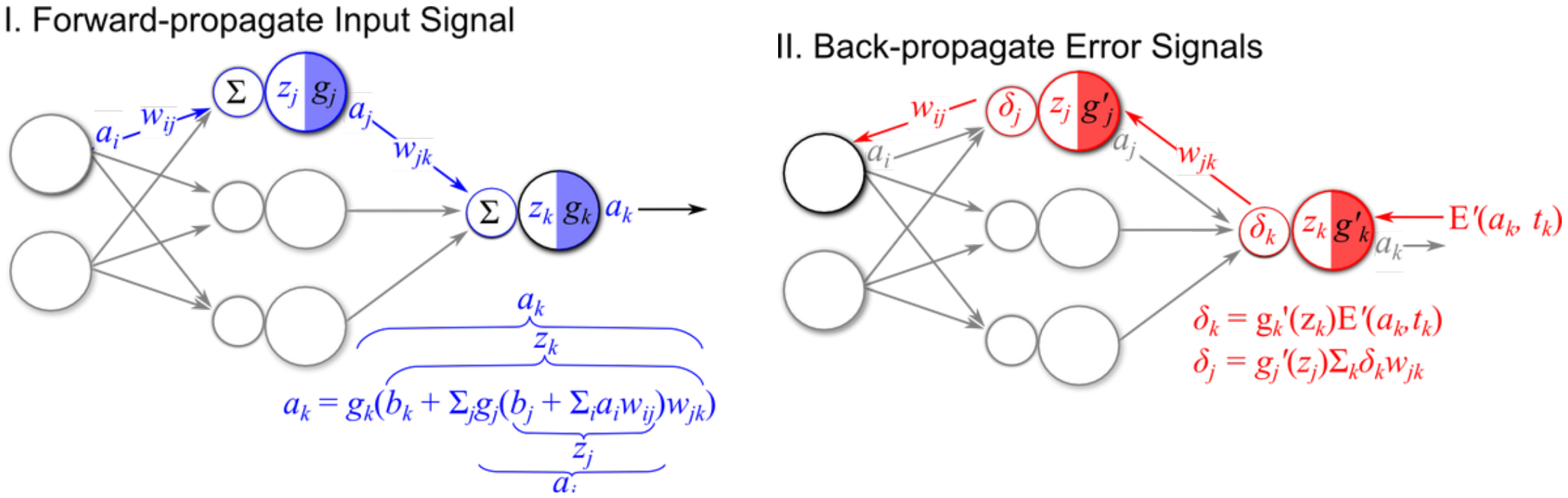


https://ml.berkeley.edu/blog/2017/02/04/tutorial-3/

Chain Rule

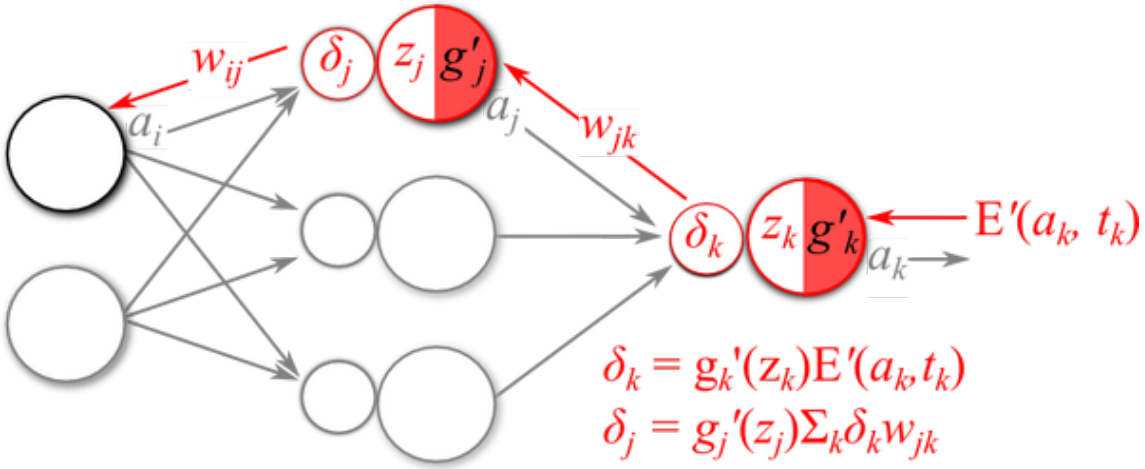

x
y
z
f
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
gradients

Recap


I. Forward-propagate Input Signal
II. Back-propagate Error Signals
a_i
w_{ij}
Σ
z_j g_j
a_j
w_{jk}
z_k g_k
a_k
$a_k = g_k(b_k + \Sigma_j g_j(b_j + \Sigma_i a_i w_{ij}) w_{jk})$
δ_j
z_j g'_j
δ_k
z_k g'_k
$E'(a_k, t_k)$
$\delta_k = g_k'(z_k)E'(a_k, t_k)$
$\delta_j = g_j'(z_j)\Sigma_k \delta_k w_{jk}$

Delta Update and Weight Update

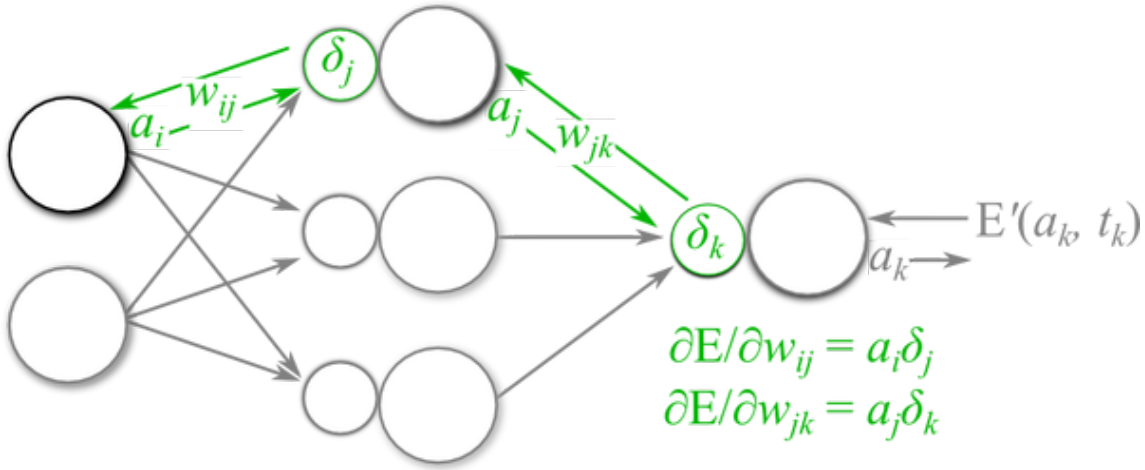
II. Back-propagate Error Signals



$\delta_k = g_k'(z_k)E'(a_k,t_k)$

$\delta_j = g_j'(z_j)\Sigma_k\delta_k w_{jk}$

III. Calculate Parameter Gradients



$\partial E/\partial w_{ij} = a_i\delta_j$

$\partial E/\partial w_{jk} = a_j\delta_k$

IV. Update Parameters

$w_{ij} = w_{ij} - \eta(\partial E/\partial w_{ij})$

$w_{jk} = w_{jk} - \eta(\partial E/\partial w_{jk})$

for learning rate η

https://theclevermachine.wordpress.com/category/algorithms/gradient-descent/

Gradient Descent Variants

1. (Batched) Gradient Descent
2. Stochastic Gradient Descent
3. Mini-batch Stochastic Gradient Descent

What is the Difference?

Gradient Descent Variants

1. (Batched) Gradient Descent
2. Stochastic Gradient Descent
3. Mini-batch Stochastic Gradient Descent

What is the Difference?

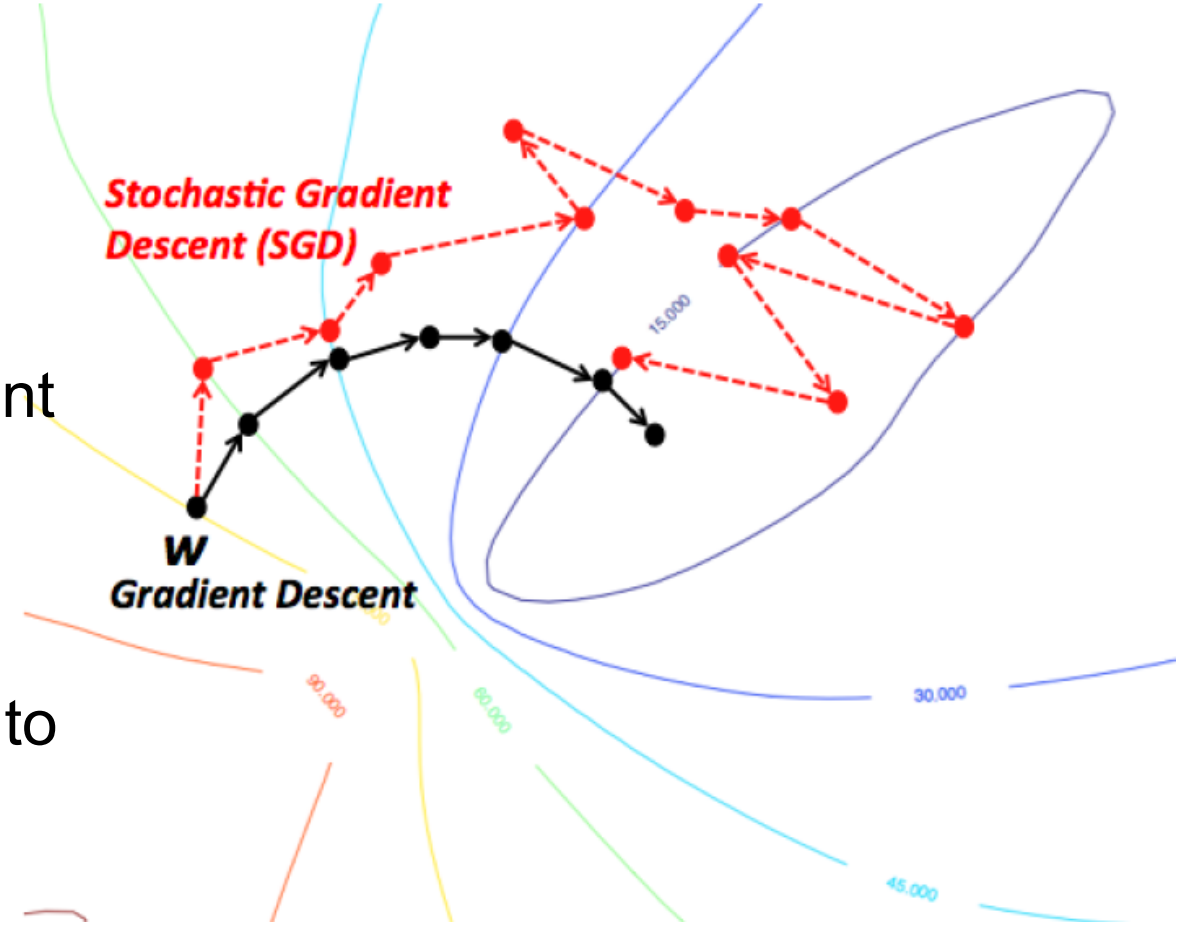
- The accuracy of the parameter update
- The time it takes to perform an update

Gradient Descent Variants

1. (Batched) Gradient Descent
   - One update with entire Data Set
2. Stochastic Gradient Descent
   - One update with a random sample
3. Mini-batch Stochastic Gradient Descent
   - A mini-batch smaller than Data set

What are the tradeoffs?

- SGD's fluctuation is large but it enables to jump to new and potentially better local minima.



Training as optimization problem

Neural Network is just a complex function

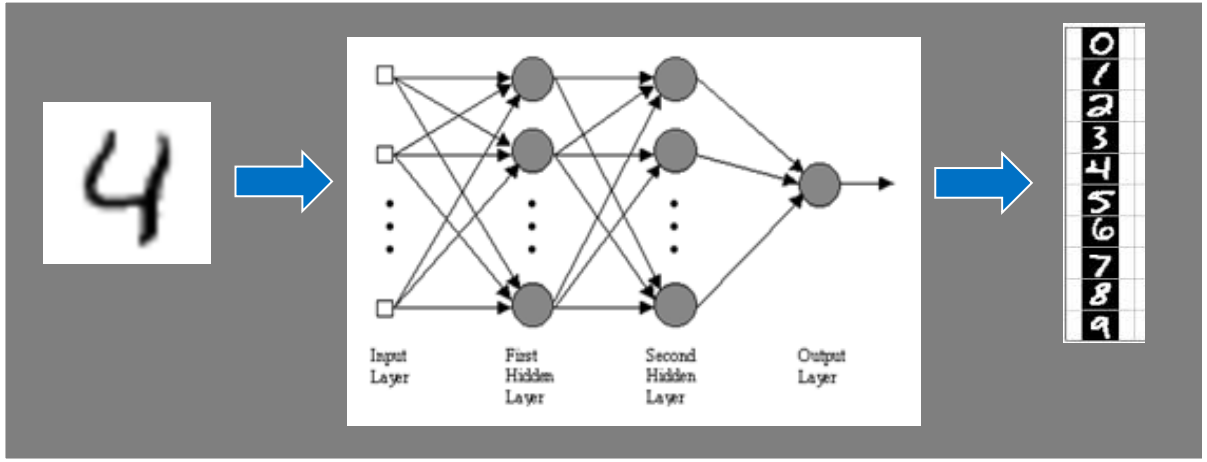
$$y=f(x, w),$$

where

x - input (e.g. image)

w - network parameters (weights)

y - output (e.g. probability that x belongs to each classes)



Batch Gradient Descent

We want to minimize loss over training set with N samples (x_n, y_n):

$$L(w) = \frac{1}{N} \sum_{n=1}^{N} E(f(x_n, w), y_n)$$

Batch GD:

1. accumulate gradients over all samples in training set

$$\frac{\partial E}{\partial w}(t) = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E}{\partial w}(w, (x_n, y_n))$$

2. update W:

$$w(t+1) = w(t) - \lambda * \frac{\partial E}{\partial w}(t)$$

Issue:
Imagenet has > 10^6 images→ gradient computation for the whole set is expensive

Stochastic Gradient Descent

Stochastic Gradient Descent (on-line learning):

1. Randomly choose sample (x_k, y_k):
2. $W(t+1) = W(t) - \lambda * \frac{\partial E}{\partial w}(w, (x_k, y_k))$

Stochastic Gradient Descent with mini-batches:

1. divide the dataset into small mini-batches
2. compute the gradient using a single mini-batch
3. make an update
4. move to the next mini-batch …

Don’t forget to shuffle dataset between epochs!

Batch Gradient Descent

Advantage

- It is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

Disadvantages

- It can be very slow
- It is intractable for datasets that do not fit in memory
- It does not allow us to update our model online

"Use stochastic gradient descent when training time is the bottleneck"

Leon Bottou, Stochastic Gradient Descent Tricks

Mini Batch Stochastic Gradient Descent

Advantages

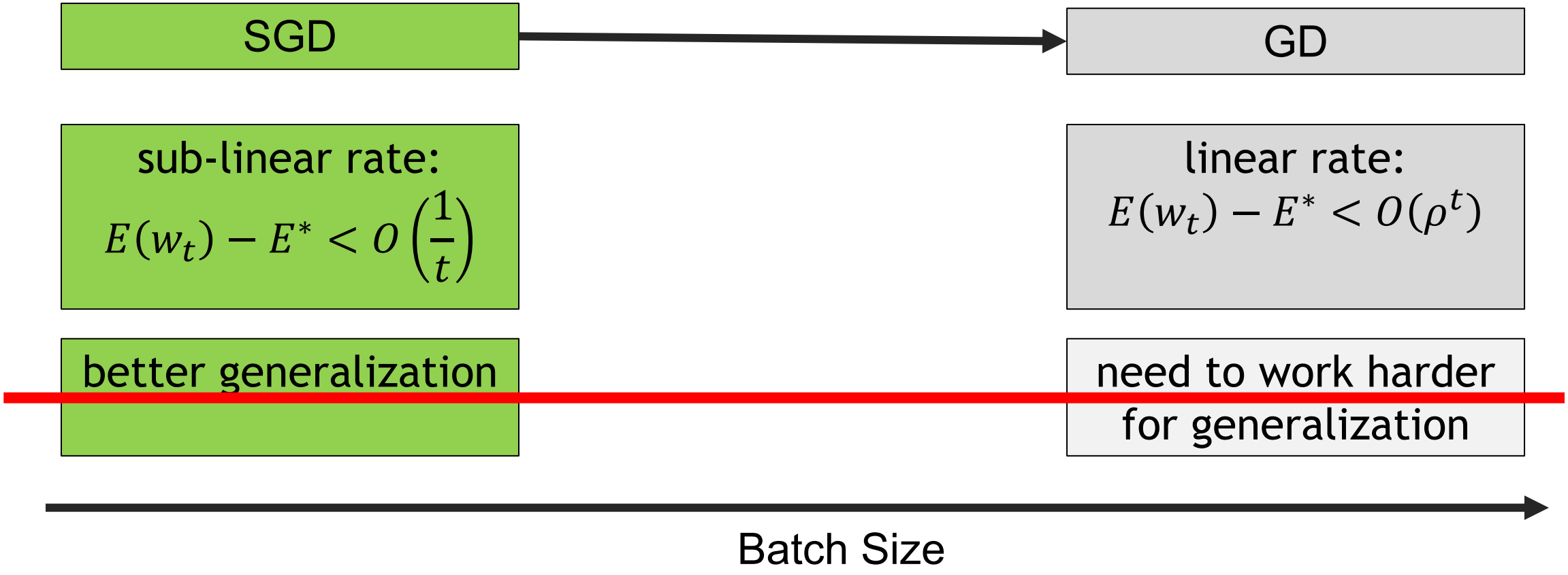
- It reduces the variance of the parameter updates
- Behaves better in local minima and saddle points
- Make use of highly optimized matrix optimizations common to deep learning libraries that make computing the gradient very efficiently

Disadvantages

- Need to set mini batch size as part of hyper parameter optimization for training
- Large batch sizes are not easy to deal with

Large Batch: SGD vs GD

Large batch: SG becomes less noisy and close to "true" gradient



In SGD We Trust

Stochastic Gradient Methods play a key role in Machine Learning

Optimization Methods for Large-Scale Machine Learning

Léon Bottou* Frank E. Curtis† Jorge Nocedal‡

June 16, 2016

Minimizing Finite Sums with the Stochastic Average Gradient

Mark Schmidt
schmidtm@cs.ubc.ca

Nicolas Le Roux
nicolas@le-roux.name

Francis Bach
francis.bach@ens.fr

Stochastic Gradients Methods for DL

Two most popular SG methods:

- SGD with momentum (ConvNets)
- Adam (RNNs, Attention)

Other useful SG methods:

- RMSProp
- SAG (Stochastic Average Gradient)
- NAG ("Nesterov's Accelerated Gradient")
- Averaged SGD

S. Ruder, "An overview of gradient descent optimization algorithms"
http://ruder.io/optimizing-gradient-descent/

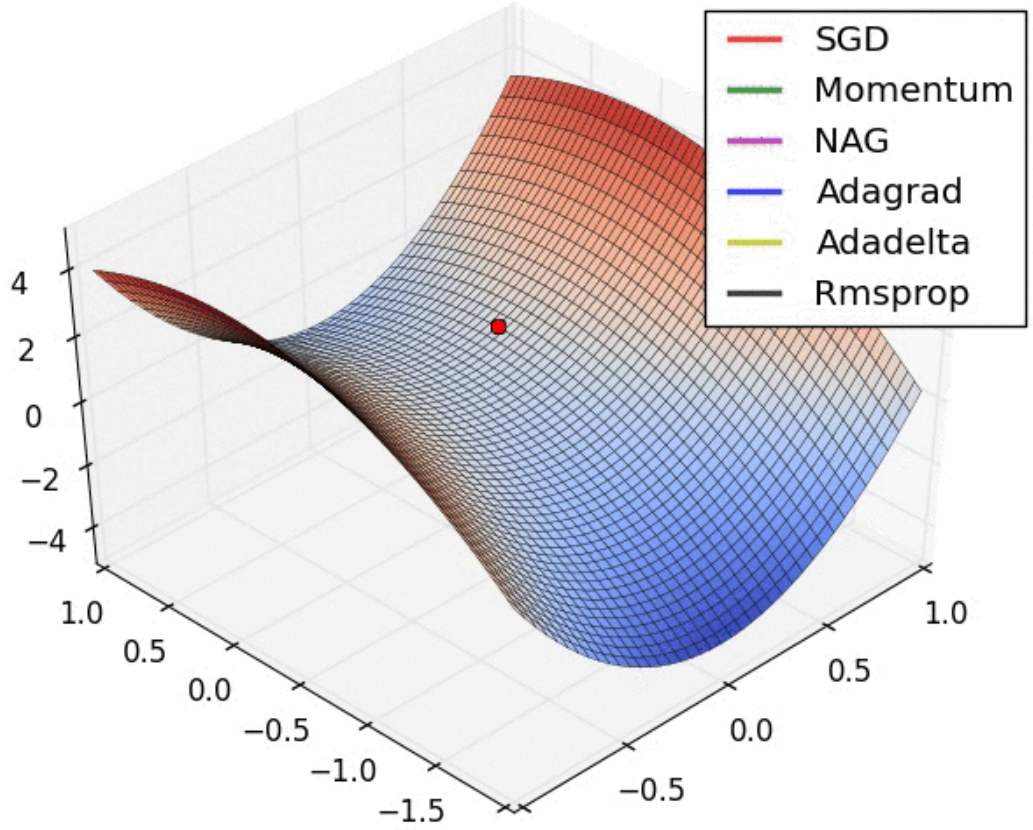
SGD: Two Kinds of Efficiency

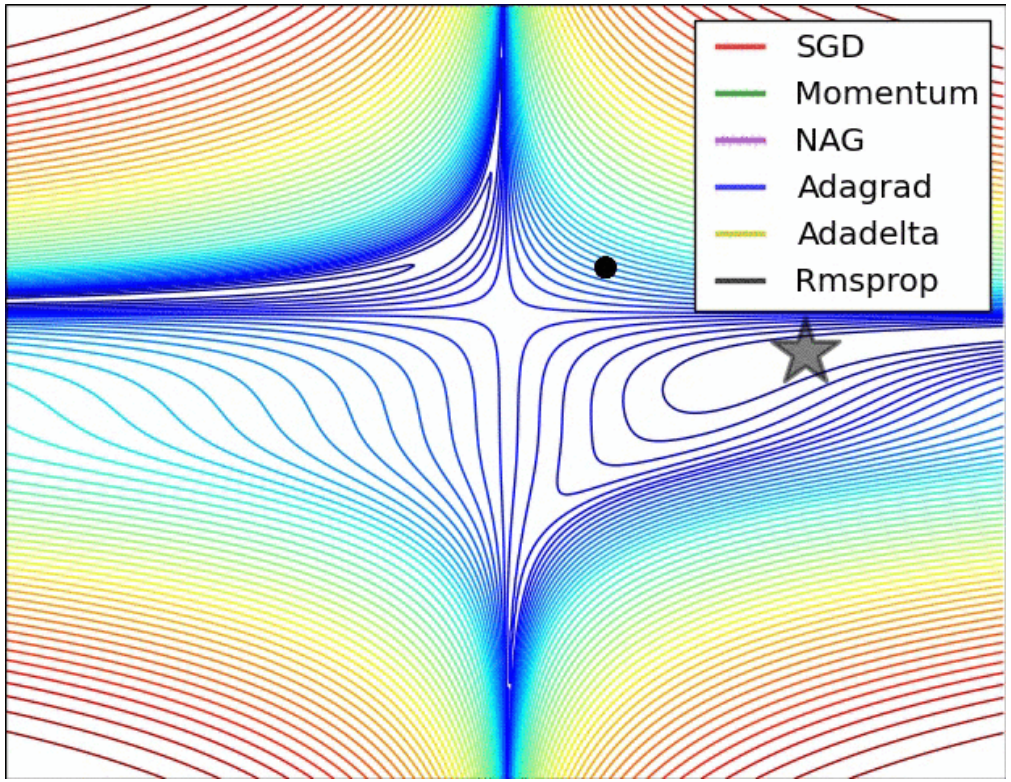
- **Statistical efficiency**: **how many iterations** do we need to get the desired accuracy level?
  - Depends on **the problem** and implementation
- **Hardware efficiency**: **how long** it takes to run each iteration?
  - Depends on **the hardware** and implementation

trade off hardware and statistical efficiency
to maximize performance

Ce Zhang and Christopher Ré.. DimmWitted: Proc. VLDB `14

SGD Optimizers





Sebastian Ruder, “An overview of gradient descent optimization algorithms” arXiv:1609.04747 15 Jun 2017
https://arxiv.org/pdf/1609.04747.pdf

Classical GD with momentum

The momentum method (Polyak, 1964) is a classical convex technique for accelerating GD

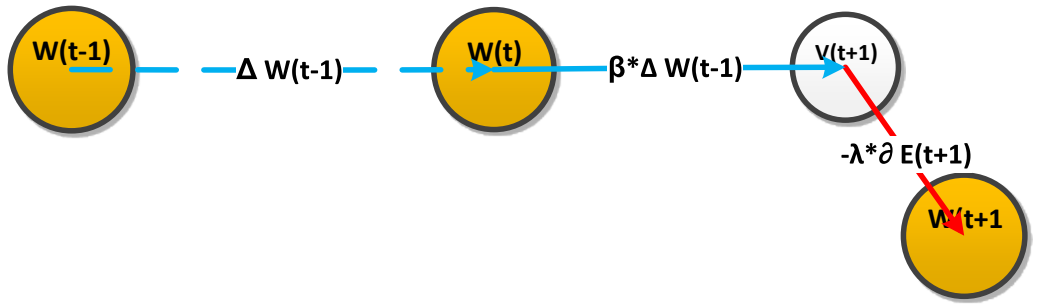
1. Compute momentum:

$$\boldsymbol{M(t+1) = \beta * M(t) - \lambda(t) * \frac{\partial E}{\partial w}(t)}$$

here $\boldsymbol{\beta}$ is momentum coeff $\boldsymbol{0 < \beta < 1}$ and $\boldsymbol{\lambda}$ is learning rate

2. Update weights:

$$W(t+1) = W(t) + M(t+1)$$



G. Goh, "Why Momentum Really Works" , https://distill.pub/2017/momentum/

SGD with momentum

1. Compute SGD over current batch B with samples $(\boldsymbol{x_n, y_n})$:

$$\boldsymbol{G(t) = \frac{1}{B}\sum_{n=1}^{B} \frac{\partial E(w,(x_n,y_n))}{\partial w}}$$

2. Compute momentum:

$$\boldsymbol{M(t+1) = \beta * M(t) - \lambda(t) * G(t)}$$

2. Update weights update:

$$W(t+1) = W(t) + M(t+1)$$

I. Sutskever , "On the importance of initialization and momentum in deep learning"

SGD with momentum as gradients averaging

Let's assume that $\boldsymbol{\lambda}$ is fixed for simplicity.

$$\boldsymbol{M(t+1) = \quad \beta * M(t) - \lambda * G(t) =}$$

$$\boldsymbol{= -\lambda * G(t) + \beta * \left(M(t-1) - \lambda * G(t-1)\right) =}$$

$$\boldsymbol{= -\lambda * G(t) - \lambda * \beta * G(t-1) + \beta * M(t-1) = \cdots}$$

$$\boldsymbol{= -\lambda * \left(\sum_{k=0}^{t} \beta^{t-k} * G(k)\right)}$$

Momentum works as weighted moving average of stochastic gradients:

- First average gradients over batch
- Then weighted "moving" average over previous steps.
- $\boldsymbol{\beta}$ controls how large is averaging window

Stochastic Average Gradient (SAG)

1. Compute SGD over current batch B with samples $(\boldsymbol{x_n, y_n})$:

$$\boldsymbol{G(t) = \frac{1}{B}\sum_{n=1}^{B} \frac{\partial E(w,(x_n,y_n))}{\partial w}}$$

2. Compute **moving average** SG :

$$\boldsymbol{M(t+1) = \beta * M(t) + (1-\beta) * G(t)}$$

2. Update weights update:

$$W(t+1) = W(t) - \boldsymbol{\lambda(t)} * M(t+1)$$

Adagrad and RMSProp

Adagrad (Duchi…, 2010) adapts learning rate for each weight using second momentum:

$$V_i(t) = V_i(t-1) + G_i^2(t)$$

$$\Delta W_i(t+1) = -\frac{\boldsymbol{\lambda}}{\sqrt{V_i(\boldsymbol{t})+\epsilon}} * G_i(t)$$

But Adagrad decreases step sizes too aggressively early in the optimization

RMSProp (Tieleman & Hinton..., 2012) computes running average of 2nd momentum for gradients

$$V_i(t) = \gamma * V_i(t-1) + (1-\gamma) * G_i^2(t)$$

$$\Delta W_i(t+1) = -\frac{\boldsymbol{\lambda(t)}}{\sqrt{V_i(\boldsymbol{t})+\epsilon}} * G_i(t)$$

RMSPROP requires to "manually" decrease global learning rate $\boldsymbol{\lambda(t)}$

Adam

1. Compute stochastic gradient :

$$G(t) = \frac{1}{B}\sum_{n=1}^{B}\frac{\partial E(w,(x_n,y_n))}{\partial w}$$

2. Compute 1st and 2nd momentum:

$$M(t+1) = \beta_1 * M(t) + (1-\beta_1) * G(t)$$

$$V(t+1) = \beta_2 * V(t) + (1-\beta_2) * G^2(t)$$

3. Update weights:

$$W(t+1) = W(t) - \lambda * \frac{M(t+1)}{\sqrt{V(t+1)} + \epsilon}$$

Adam

1. Compute stochastic gradient :

$$G(t) = \frac{1}{B}\sum_{n=1}^{B}\frac{\partial E(w,(x_n,y_n))}{\partial w}$$

2. Compute 1st and 2nd momentum and fix the bias

$$M(t+1) = \beta_1 * M(t) + (1-\beta_1) * G(t) \qquad \widehat{M}(t+1) = \frac{M(t+1)}{1-\beta_1^t}$$

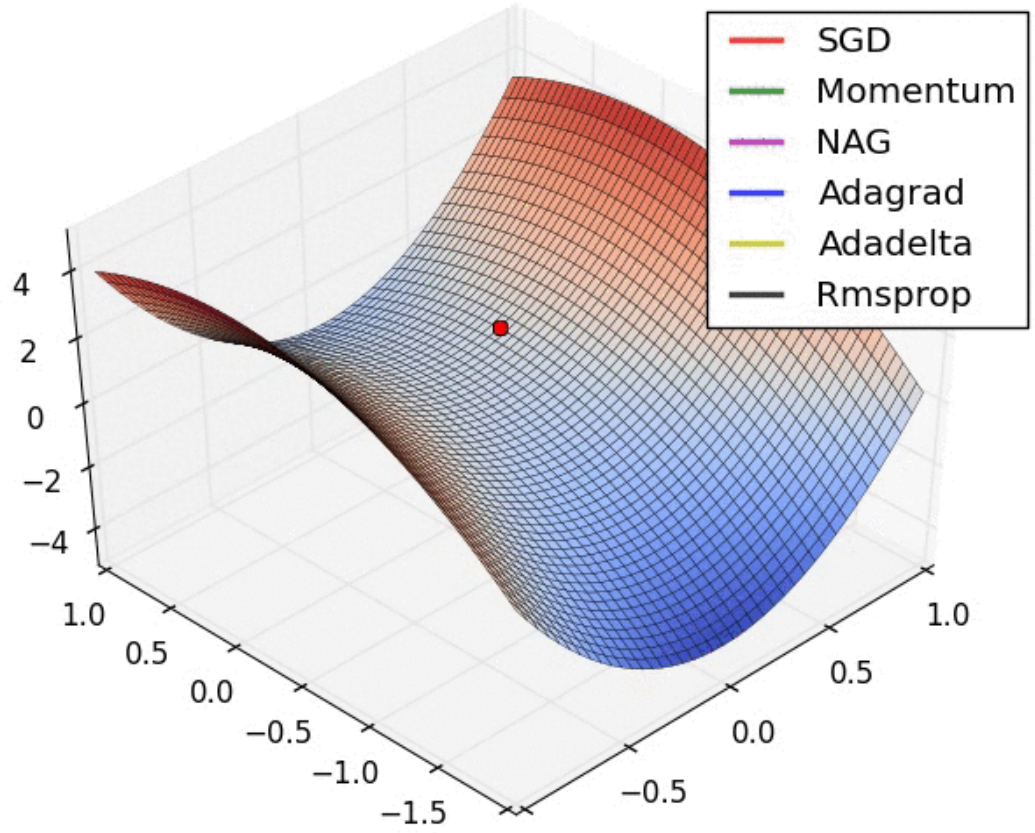
$$V(t+1) = \beta_2 * V(t) + (1-\beta_2) * G^2(t) \qquad \hat{V}(t+1) = \frac{V(t+1)}{1-\beta_2^t}$$

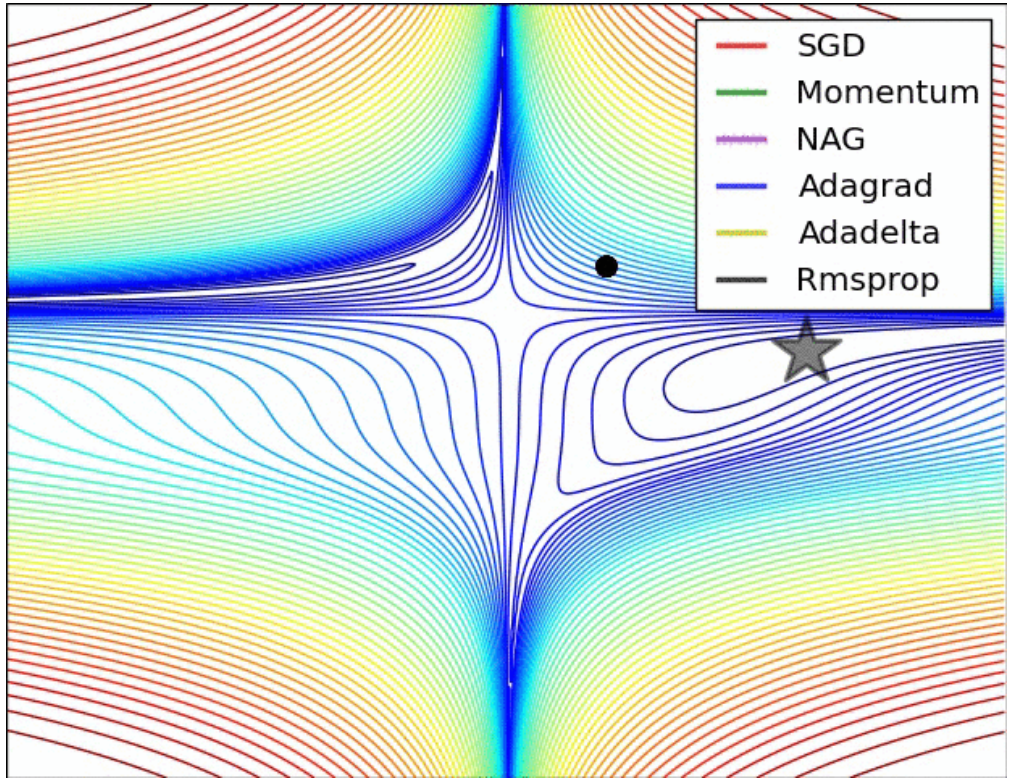
3. Update weights:

$$W(t+1) = W(t) - \lambda * \frac{\widehat{M}(t+1)}{\sqrt{\hat{V}(t+1)} + \epsilon}$$

Exercise: There is a bug in this algorithm. Can you build a counterexample?
Hint: https://arxiv.org/pdf/1705.08292.pdf and https://openreview.net/forum?id=ryQu7f-RZ

SGD Optimizers





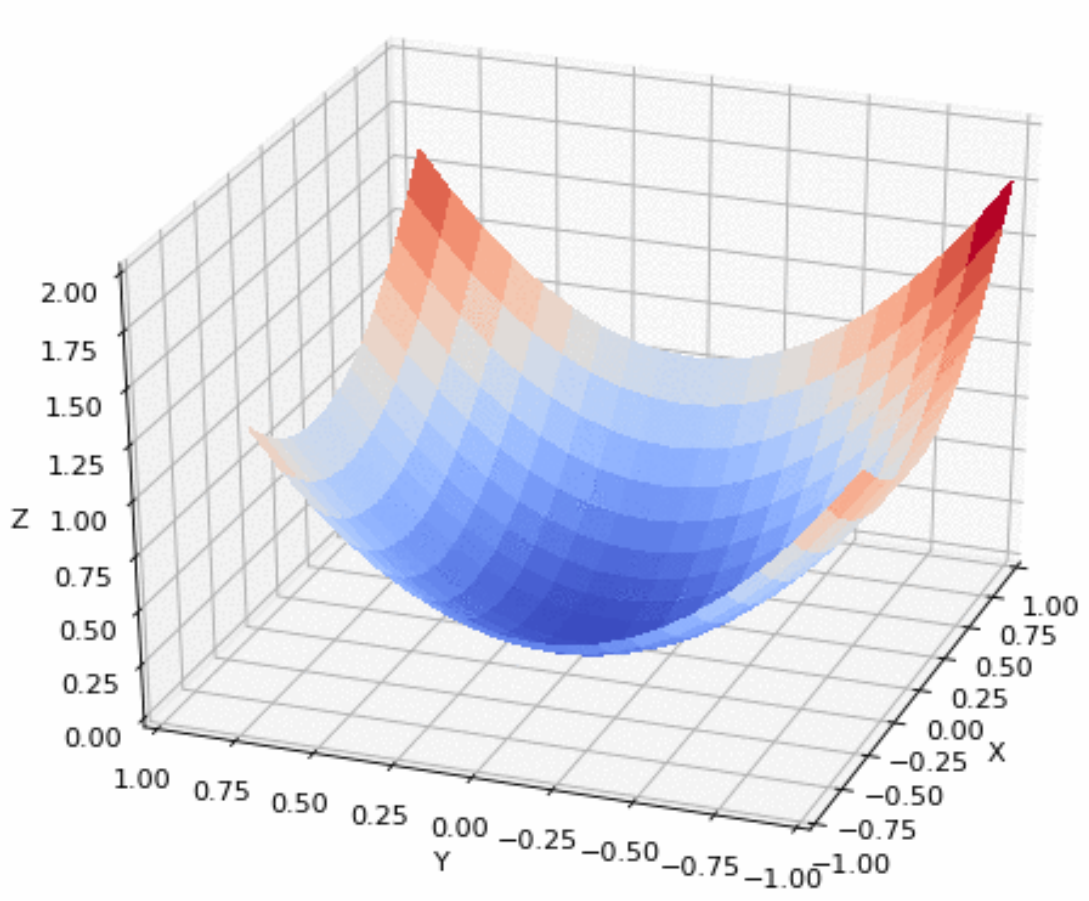
Sebastian Ruder, “An overview of gradient descent optimization algorithms” arXiv:1609.04747 15 Jun 2017
https://arxiv.org/pdf/1609.04747.pdf

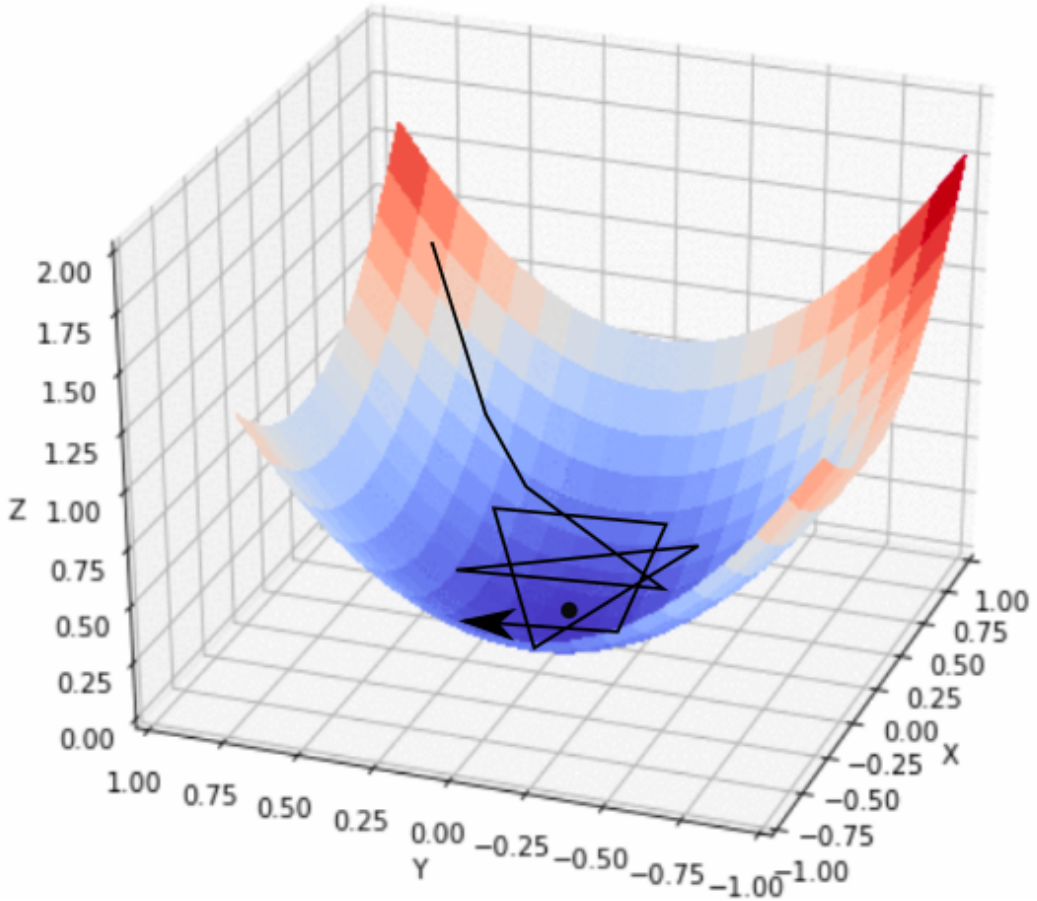
Hyperparameters

There are a lot of hyperparameters:

- Model + initialization
- Optimization algorithm
- **Learning Rate**
- Regularization (weight decay, data augmentation, dropout,..)
- Mini-Batch size
- Training duration

Learning Rate





Sebastian Ruder, “An overview of gradient descent optimization algorithms” arXiv:1609.04747 15 Jun 2017
https://arxiv.org/pdf/1609.04747.pdf

Learning Rate

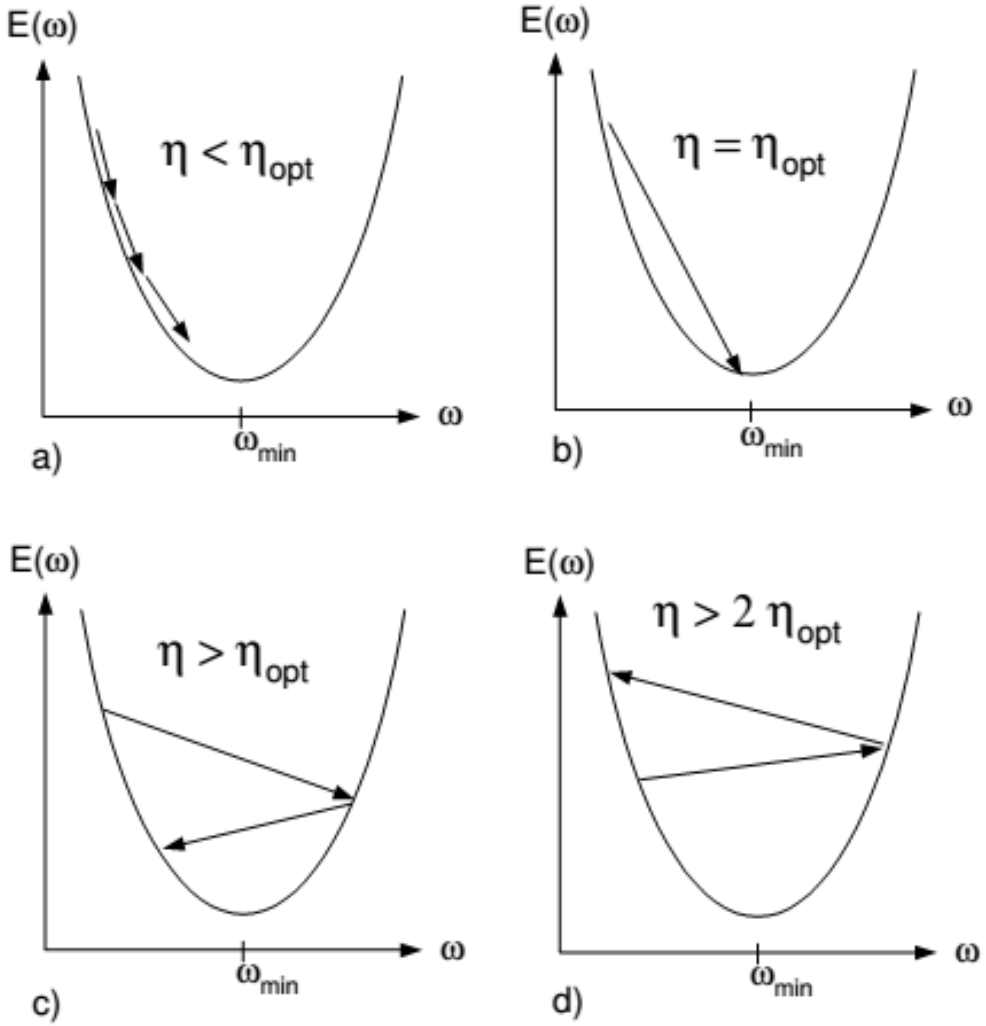
Classical convex optimization:

- GD – fixed small LR
- GD with momentum – decreasing LR

$$\sum_{t=1}^{\infty} \frac{1}{\lambda^2(t)} < \infty \text{ and } \sum_{t=1}^{\infty} \frac{1}{\lambda(t)} = \infty$$

Deep Learning:

- Function non-convex
- SGD – decreasing LR



Why SG-based methods work?

Deep Learning:

- loss function is non-convex:
  - large number of solutions
  - large number of saddles (and local minima?)
- "No bad local minima" hypothesis:
  - large NNs have no local minima, only saddle points and a global minima
- SG algorithms can easily escape from saddle points

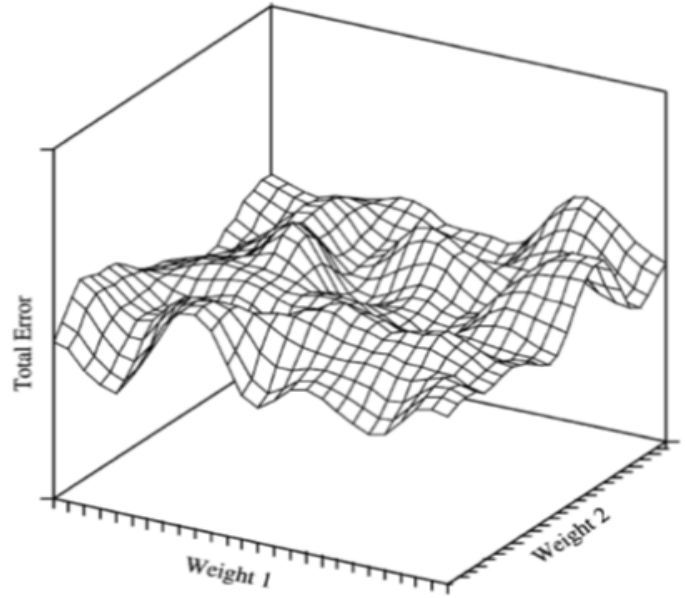


Fig. 1. Illustration of an error surface. Two weights define the x–y plane, and the height indicates the error at each point in weight space.

Wilson, *The general inefficiency of batch training for gradient descent learning*

Baldi, P. and Hornik, K. Neural networks and principal component analysis: Learning from examples without local minima. Neural Networks, 2(1):53–58, 1989.
R. Pascanu, "On the saddle point problem for non-convex optimization", http://arxiv.org/abs/1405.4604
Dauphin, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", http://arxiv.org/pdf/1406.2572v1.pdf

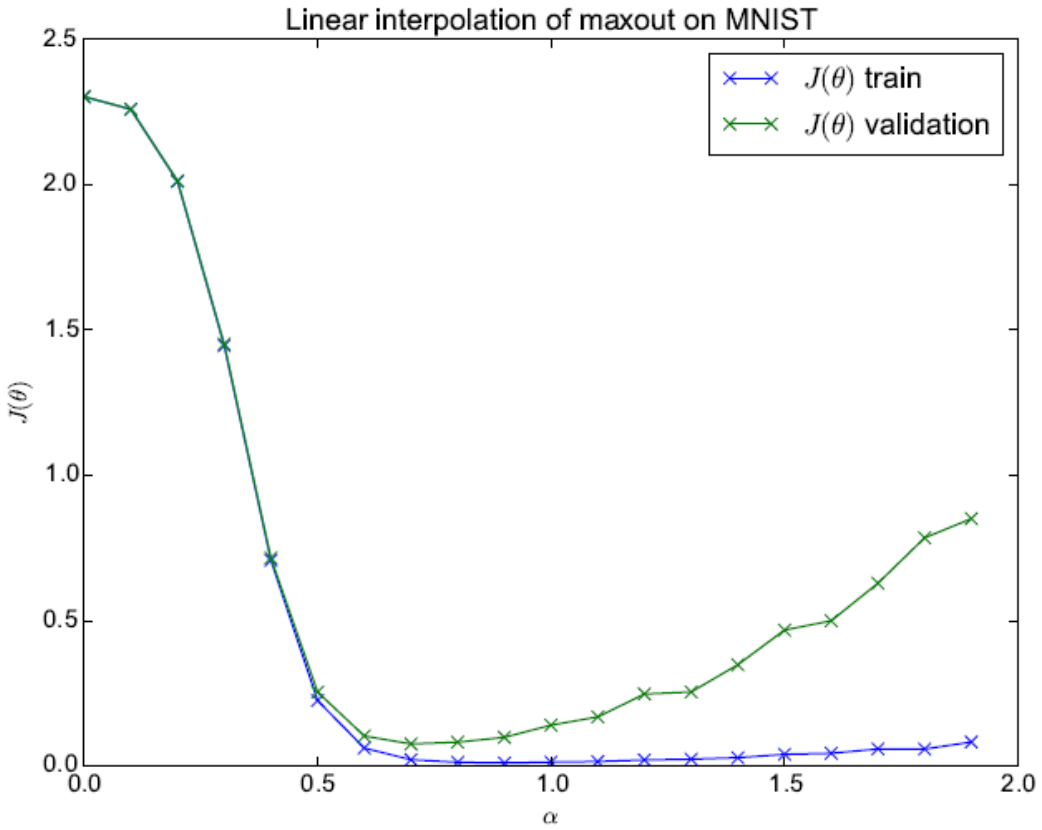
Loss Function Surface

Simple technique to visualize loss function:

compute loss function along the line which connects initial point X_0 with solution X_1:

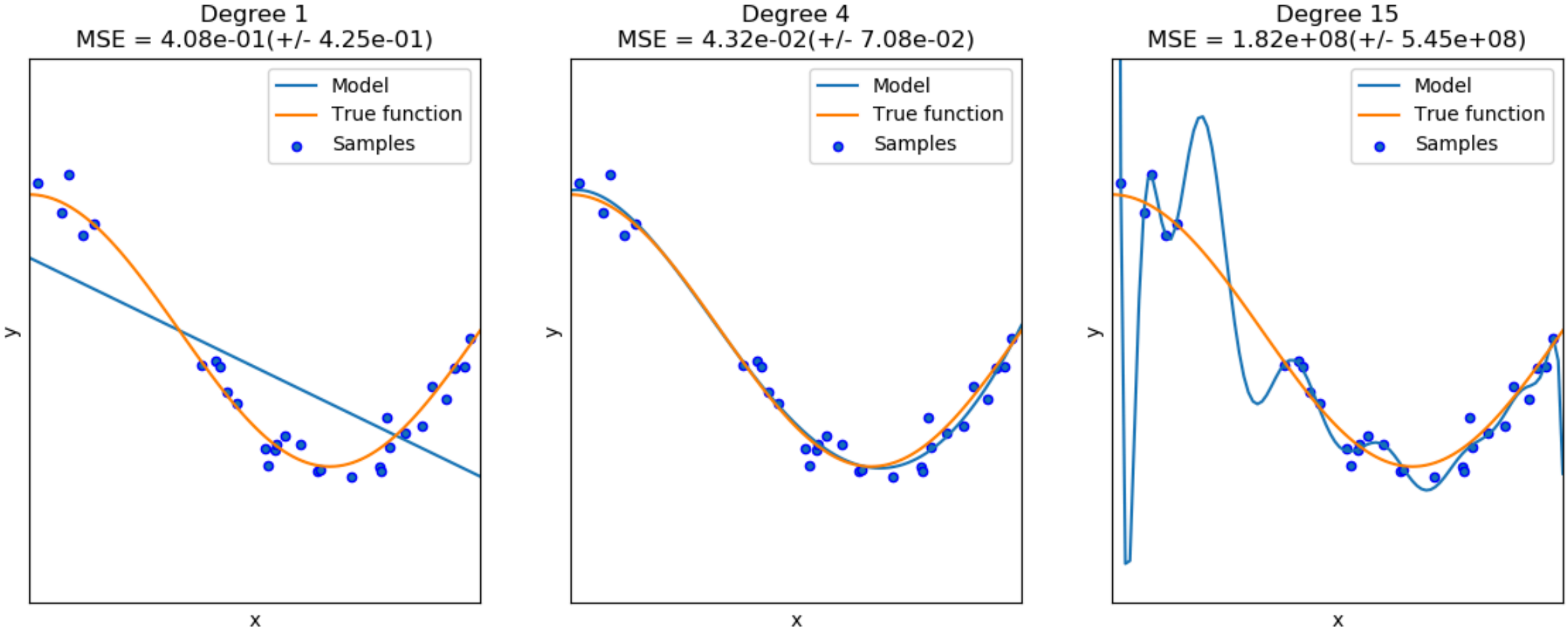
$$x(\theta) = x_0 * (1 - \theta) + x_{min} * \theta$$

"These experiments show that SGD does encounter obstacles, such as a ravine that shapes its path, but we never found evidence that local minima or saddle points slowed the SGD trajectory"



I.Goodfellow, O.Vinyals, A. Saxe, "Qualitatively Characterizing NN optimization problems"

Generalization and Overfitting



http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html

Machine Learning = Optimization + Generalization

We minimize the loss on training set:

$$L(w) = \frac{1}{N} \sum_{n=1}^{N} E(f(x_n, w), y_n)$$

But we test it on different set of samples.

Generalization gap = testing loss - training loss

How we can reduce it:

- Data augmentation,
- weight decay,
- dropout,
- batch-norm,…

Let's take a look at this problem from *optimization* point of view

Generalization: Convex Case

Let's take some nice machine learning problem with smooth and convex function.

Train loss: $y(w) = (w - x_0)^2$ for x_0=1

Test loss: $y(w) = (w - x_1)^2$ where x_1=0.98

Training:

- Set initial w_0=0
- Optimization algorithm:
  - Gradient Descent
  - learning rate = fixed , $\lambda = 0.1$
  - Num of steps 10

Found solution $w_1 = 0.995$.

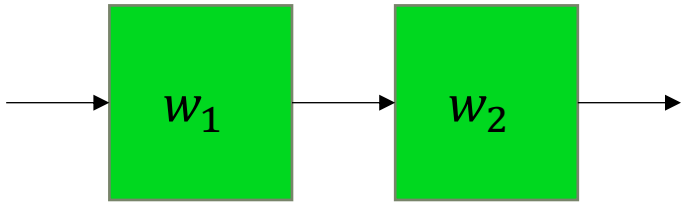
Training loss = 0.000025

Test loss = 0.000225

We found a good minima!

Generalization: Deep Learning

Change our simple quadratic problem to 2-layer MLP (for simplicity w/o non-linear part)



We call this model *deep linear network.*

Now we have complex non-convex optimization problem:

minimize $y(w_1, w_2) = (w_1 * w_2 - 1)^2$

This is non-convex functions:

- many saddle points,
- no local minima
- many global solutions: $w_1 * w_2 = 1$

Minima: Good, Bad and Ugly

The error function

$$y(w_1, w_2) = (w_1 * w_2 - 1)^2$$

Saddle point at (0,0)

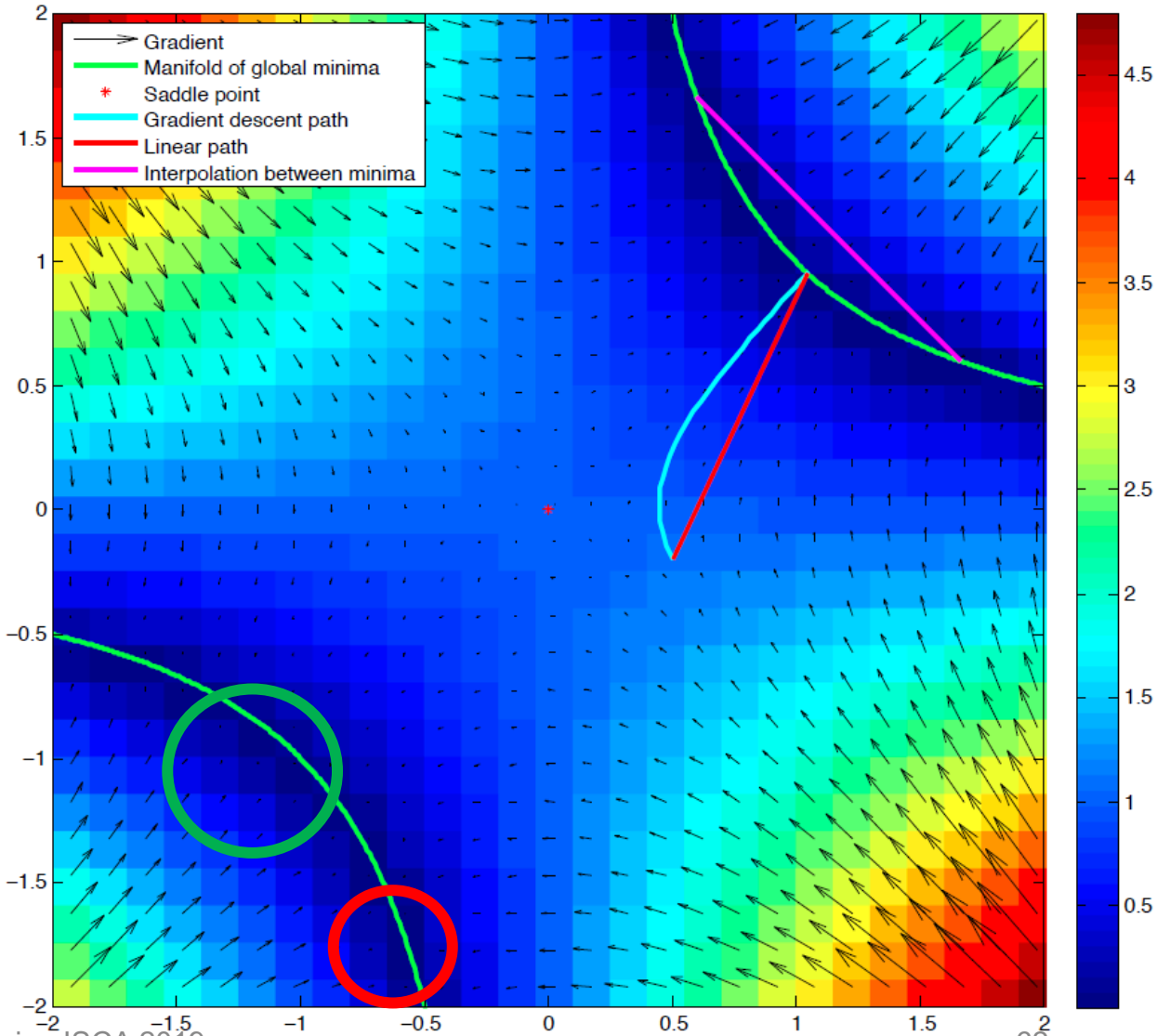
No local ("ugly") minima

All minima are global

Some of them are better than others:

- Sharp minima – bad regularization
- Flat minima – good regularization

I.Goodfellow, O.Vinyals, A. Saxe,
"Qualeitatively Characterizing NN optimization problems



Source: Boris Ginsburg

Optimization is Hard, Generalization is H arder

- Most popular. convolutional networks for image classification have millions of parameters; trained with SGD methods they easily fit a **random labeling** of the training data
- Why large neural networks generalize so well in practice?

UNDERSTANDING DEEP LEARNING REQUIRES RE-THINKING GENERALIZATION

Chiyuan Zhang*
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

Benjamin Recht†
University of California, Berkeley
brecht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

Summary

- Deep Learning = Data + Model + Optimization + Generalization
- Stochastic Optimization is different from classical convex optimization
- Optimization algorithms
  - SGD with momentum
  - Adam
  - Learning rate policy
- Generalization:
  - Training loss vs testing loss
  - There are no local (“ugly”) minima, only good (“flat”) and bad (“sharp”)

Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Backpropagation Revisited in Vector Matrix Form

Slides for Chapter 10, Deep learning
of *Data Mining* by
I. H. Witten, E. Frank,
M. A. Hall, and C. J. Pal

Deep neural network architectures

- Compose computations performed by many layers
- Denoting the output of hidden layers by $\mathbf{h}^{(l)}(\mathbf{x})$, the computation for a network with *L* hidden layers is:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}\left[\mathbf{a}^{(L+1)}\left(\mathbf{h}^{(L)}\left(\mathbf{a}^{(L)}\left(\ldots\left(\mathbf{h}^{(2)}\left(\mathbf{a}^{(2)}\left(\mathbf{h}^{(1)}\left(\mathbf{a}^{(1)}(\mathbf{x})\right)\right)\right)\right)\right)\right)\right]$$

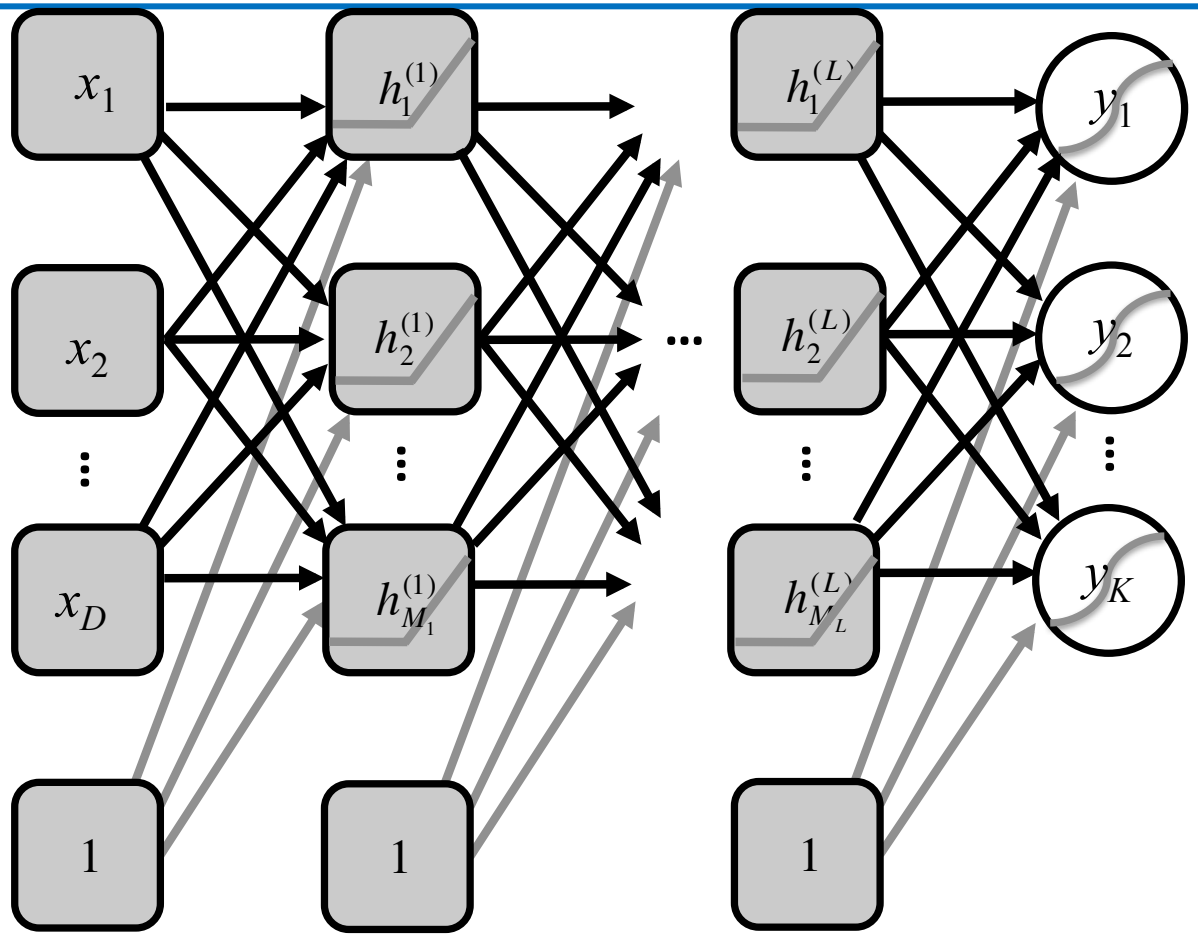
- Where *pre-activation functions* $\mathbf{a}^{(l)}(\mathbf{x})$ are typically linear, of the form $\mathbf{a}^{(l)}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}$ with matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$
- This formulation can be expressed using a single parameter matrix θ with the trick of defining $\hat{\mathbf{x}}$ as x with a 1 appended to the end of the vector; we then have

$$\mathbf{a}^{(l)}(\hat{\mathbf{x}}) = \boldsymbol{\theta}^{(l)}\hat{\mathbf{x}} \quad , l = 1$$

$$\mathbf{a}^{(l)}(\hat{\mathbf{h}}^{(l-1)}) = \boldsymbol{\theta}^{(l)}\hat{\mathbf{h}}^{(l-1)} \quad , l > 1$$

Source I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal

Deep feedforward networks



- Unlike Bayesian networks the hidden units here are *intermediate deterministic computations* not random variables, which is why they are not represented as circles
- However, the output variables y_k are drawn as circles because they can be formulated probabilistically

Backpropagation in Matrix Vector Form

- Backpropagation is based on the chain rule of calculus
- Consider the loss for a single-layer network with a softmax output (which corresponds exactly to the model for multinomial logistic regression)
- We use multinomial vectors **y**, with a single dimension $y_k = 1$ for the corresponding class label and whose other dimensions are 0
- Define $\mathbf{f} = [\ f_1(\mathbf{a}),\ \ldots,\ f_K(\mathbf{a})\ ]^T$, and $a_k(\mathbf{x};\boldsymbol{\theta}_k) = \boldsymbol{\theta}_k^T\mathbf{x}$, $\mathbf{a}(\mathbf{x};\boldsymbol{\theta}) = [\ a_1(\mathbf{x};\boldsymbol{\theta}_1),\ \ldots,\ a_K(\mathbf{x};\boldsymbol{\theta}_K)\ ]^T$ where θ_k is a column vector containing the k^{th} row of the parameter matrix
- Consider the softmax loss for **f**(**a**(**x**))

Logistic regression and the chain rule

- Given loss $L = -\sum_{k=1}^{K} y_k \log f_k(\mathbf{x}), \quad f_k(\mathbf{x}) = \dfrac{\exp(a_k(\mathbf{x}))}{\sum_{c=1}^{K} \exp(a_c(\mathbf{x}))}.$
- Use the chain rule to obtain

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} \frac{\partial \mathbf{f}}{\partial \mathbf{a}} \frac{\partial L}{\partial \mathbf{f}} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} \frac{\partial L}{\partial \mathbf{a}}.$$

- Note the order of terms - in vector matrix form terms build from right to left

$$\frac{\partial L}{\partial a_j} = \frac{\partial}{\partial a_j}\left[-\sum_{k=1}^{K} y_k \left[a_k - \log\left[\sum_{c=1}^{K} \exp(a_c)\right]\right]\right]$$

$$= -\left[y_{k=j} - \frac{\exp(a_{k=j})}{\sum_{c=1}^{K} \exp(a_c)}\right] = -\left[y_j - p(y_j \mid \mathbf{x})\right] = -\left[y_j - f_j(\mathbf{x})\right],$$

Source I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal

Matrix vector form of gradient

- We can write $$\frac{\partial L}{\partial \mathbf{a}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] \equiv -\Delta$$

  and since $$\frac{\partial a_j}{\partial \boldsymbol{\theta}_k} = \begin{cases} \frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{\theta}_k^{\mathrm{T}} \mathbf{x} = \mathbf{x} & , j = k \\ 0 & , j \neq k \end{cases}$$

  we have $$\frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} = \mathbf{H}_k = \begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}$$

- Notice that we avoid working with the partial derivative of the vector **a** with respect to the matrix θ, because it cannot be represented as a matrix — it is a multidimensional array of numbers (a tensor).

A compact expression for the gradient

- The gradient (as a column vector) for the vector in the *k*th row of the parameter matrix

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k}\frac{\partial L}{\partial \mathbf{a}} = -\begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right]$$

$$= -\mathbf{x}(y_k - f_k(x)).$$

- With a little rearrangement the gradient for the entire matrix of parameters can be written compactly:

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right]\mathbf{x}^{\mathrm{T}} = -\Delta\mathbf{x}^{\mathrm{T}}.$$

Consider now a multilayer network

- Using the same activation function for all L hidden layers, and a softmax output layer
- The gradient of the k^{th} parameter vector of the L+1$^{\text{th}}$ matrix of parameters is

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(L+1)}} = \frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}}, \qquad \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} = -\Delta^{(L+1)}$$

$$= -\frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \Delta^{(L+1)}$$

$$= -\mathbf{H}_k^L \Delta^{(L+1)} \quad \Rightarrow \quad \frac{\partial L}{\partial \boldsymbol{\theta}^{(L+1)}} = -\Delta^{(L+1)} \tilde{\mathbf{h}}_{(L)}^{\mathrm{T}}.$$

where $\mathbf{H}_k^L$ is a matrix containing the activations of the corresponding hidden layer, in column k

Backpropagating errors

- Consider the computation for the gradient of the k^{th} row of the L^{th} matrix of parameters
- Since the bias terms are constant, it is unnecessary to backprop through them, so

$$\frac{\partial L}{\partial \mathbf{\theta}_k^{(L)}} = \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{\theta}_k^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}}$$

$$= -\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{\theta}_k^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \Delta^{(L+1)}, \qquad \Delta^{(L)} \equiv \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \Delta^{(L+1)}$$

$$= -\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{\theta}_k^{(L)}} \Delta^{(L)}$$

- Similarly, we can define $\Delta^{(l)}$ recursively in terms of $\Delta^{(l+1)}$

Backpropagating errors

- The backpropagated error can be written as simply

$$\Delta^{(l)} = \frac{\partial\, \mathbf{h}^{(l)}}{\partial\, \mathbf{a}^{(l)}} \frac{\partial\, \mathbf{a}^{(l+1)}}{\partial\, \mathbf{h}^{(l)}} \Delta^{(l+1)}, \qquad \frac{\partial\, \mathbf{h}^{(l)}}{\partial\, \mathbf{a}^{(l)}} = \mathbf{D}^{(l)}, \qquad \frac{\partial\, \mathbf{a}^{(l+1)}}{\partial\, \mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)},$$

$$\Delta^{(l)} = \mathbf{D}^{(l)} \mathbf{W}^{\mathrm{T}(l+1)} \Delta^{(l+1)}$$

  where $\mathbf{D}^{(l)}$ contains the partial derivatives of the hidden-layer activation function with respect to the pre-activation input.

- $\mathbf{D}^{(l)}$ is generally diagonal, because activation functions usually operate on an elementwise basis
- $\mathbf{W}^{\mathrm{T}(l+1)}$ arises from the fact that $\mathbf{a}^{(l+1)}(\mathbf{h}^{(l)}) = \mathbf{W}^{(l+1)}\mathbf{h}^{(l)} + \mathbf{b}^{(l+1)}$

A general form for gradients

- The gradients for the k^{th} vector of parameters of the l^{th} network layer can therefore be computed using products of matrices of the following form

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(l)}} = -\mathbf{H}_k^{(l-1)}\mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\cdots\mathbf{D}^{(L)}\mathbf{W}^{\mathrm{T}(L+1)}\Delta^{(L+1)}, \qquad \frac{\partial L}{\partial \boldsymbol{\theta}^{(l)}} = -\Delta^{(l)}\hat{\mathbf{h}}_{(l-1)}^{\mathrm{T}}$$

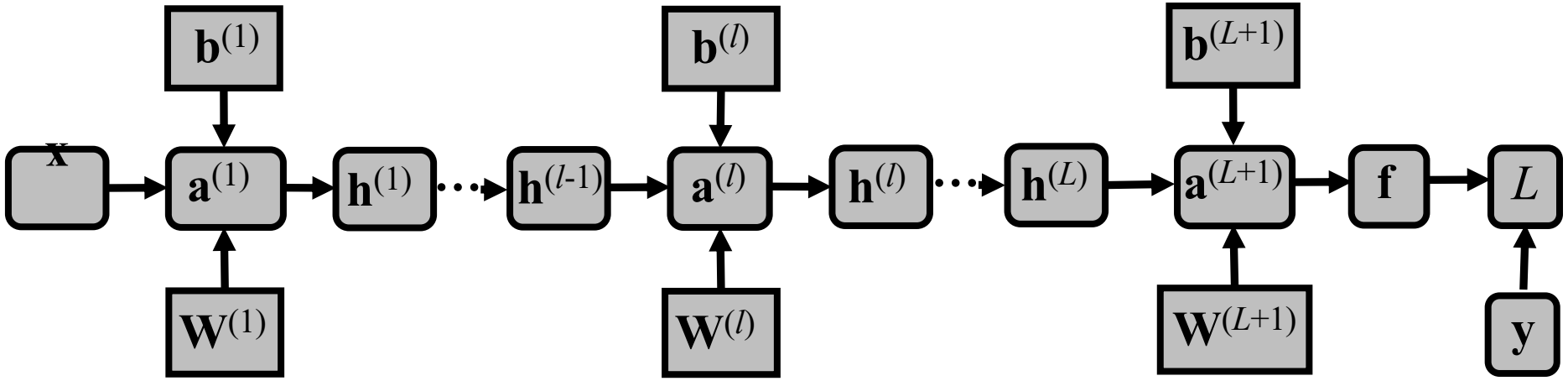
- When l=1, $\hat{\mathbf{h}}_{(0)} = \hat{\mathbf{x}}$, the input data with a 1 appended
- Note: since **D** is usually diagonal the corresponding matrix-vector multiply can be transformed into an element-wise product $\circ$ by extracting the diagonal for **d**

$$\Delta^{(l)} = \mathbf{D}^{(l)}(\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)}) = \mathbf{d}^{(l)} \circ (\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)})$$

Visualizing backpropagation

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \qquad \mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \qquad \mathbf{a}^{(L+1)} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}$$

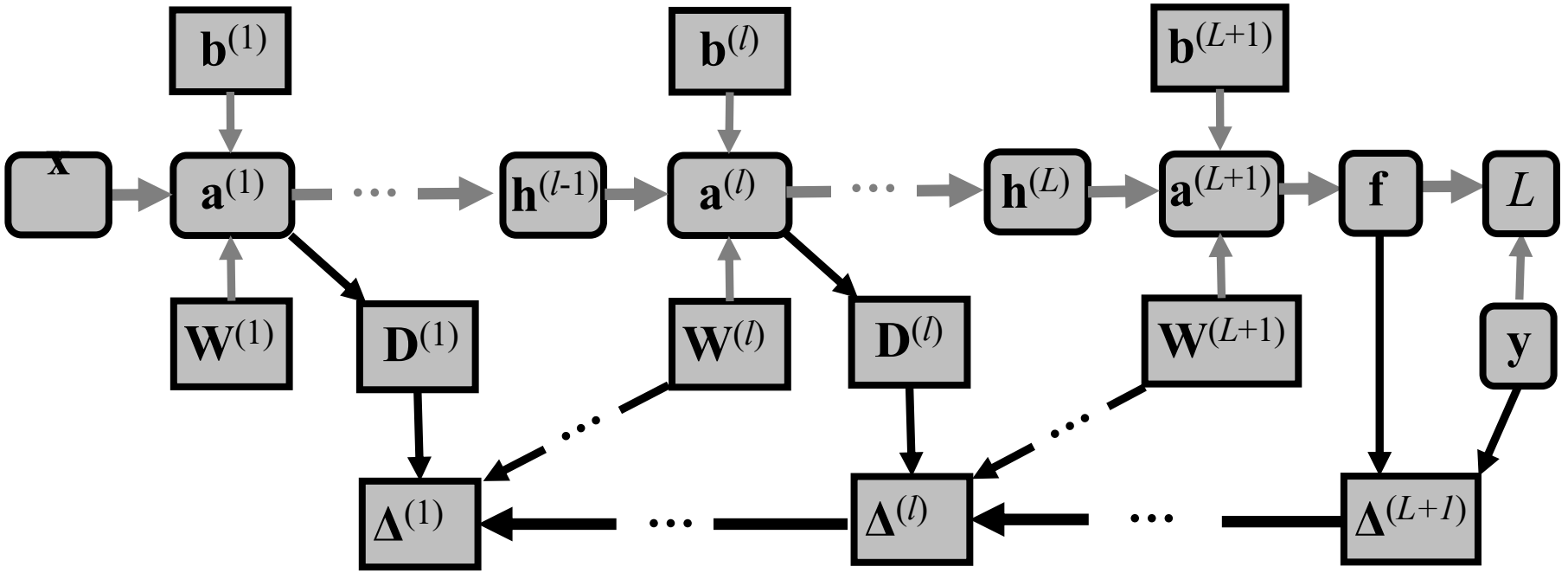
$$\mathbf{h}^{(1)} = \text{act}(\mathbf{a}^{(1)}) \qquad \mathbf{h}^{(l)} = \text{act}(\mathbf{a}^{(l)}) \qquad \mathbf{f} = \text{out}(\mathbf{a}^{(L+1)})$$



- In the forward propagation phase we compute terms of the form above
- The figure above is a type of computation graph, (which is different from the probability graphs we saw earlier)

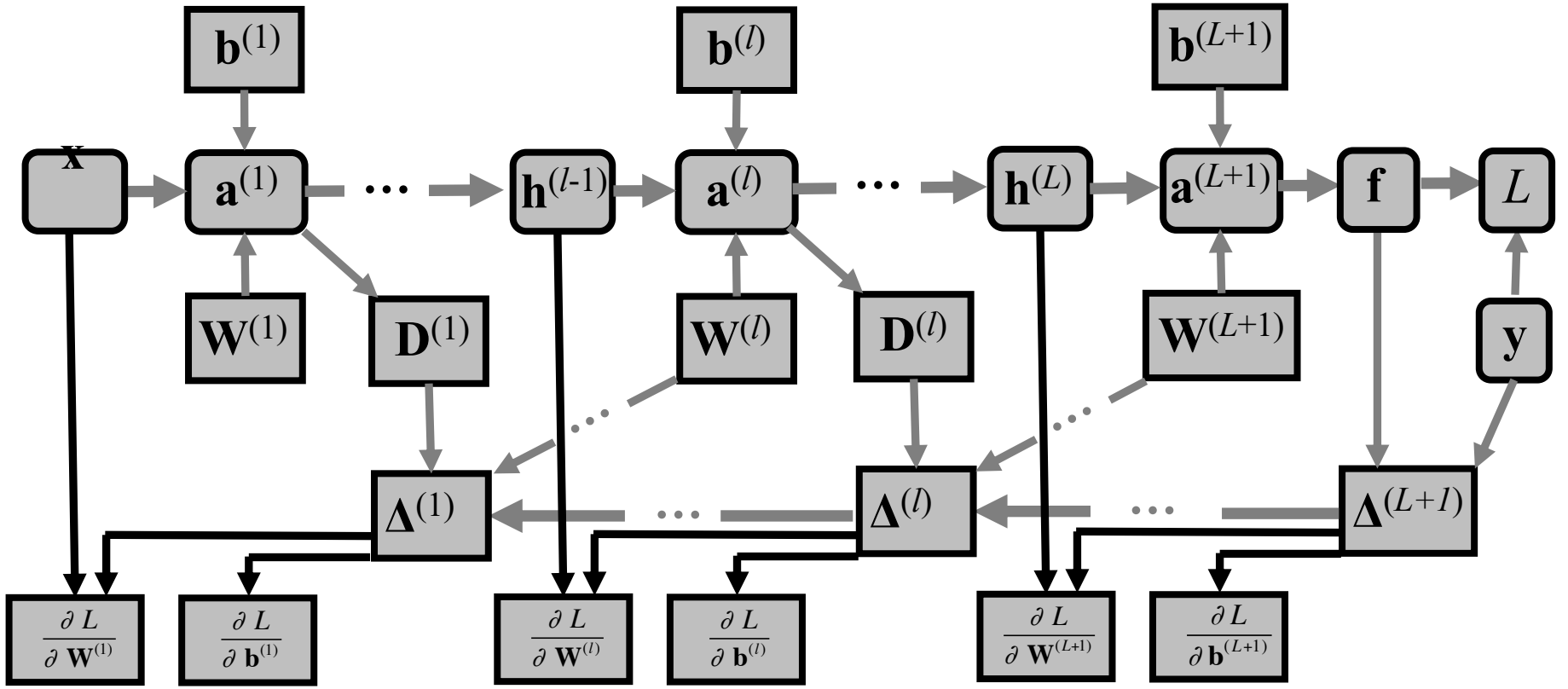
Visualizing backpropagation

- In the backward propagation phase we compute terms of the form below



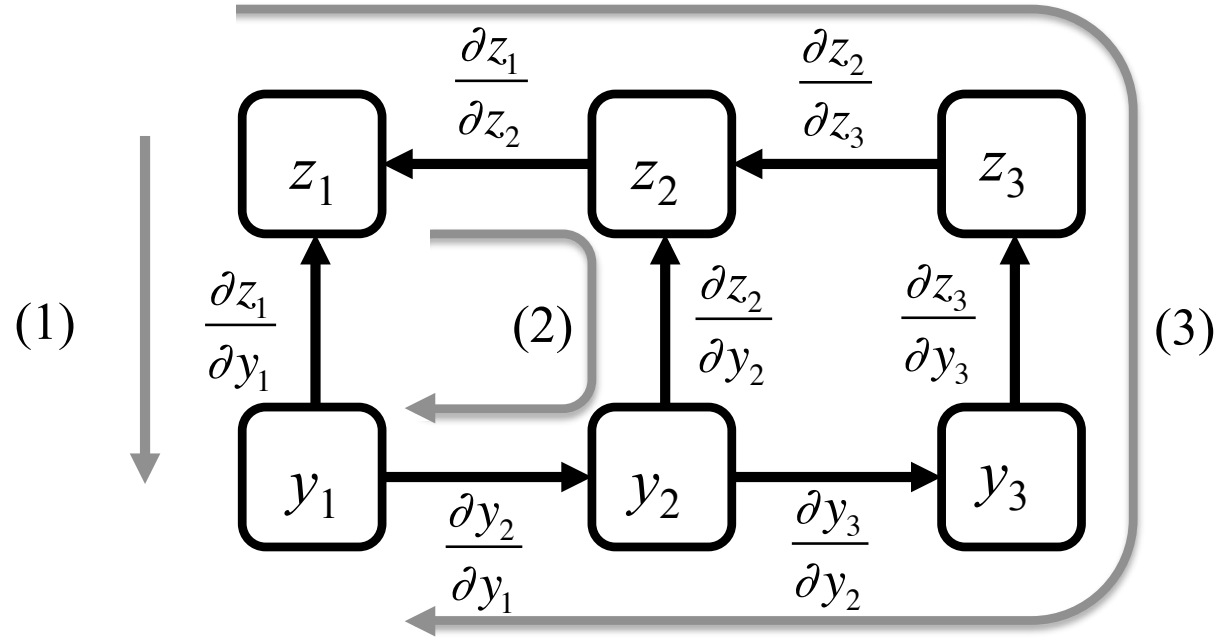
Visualizing backpropagation

- We update the parameters in our model using the simple computations below



$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = -\Delta^{(1)}\mathbf{x}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(1)}} = -\Delta^{(1)} \qquad \frac{\partial L}{\partial \mathbf{W}^{(l)}} = -\Delta^{(l)}\mathbf{h}_{(l-1)}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(l)}} = -\Delta^{(l)}$$

Computation graphs



- For more complicated computations, computation graphs can help us keep track of how computations decompose, ex. $z_1 = z_1(y_1, z_2(y_2(y_1), z_3(y_3(y_2(y_1)))))$

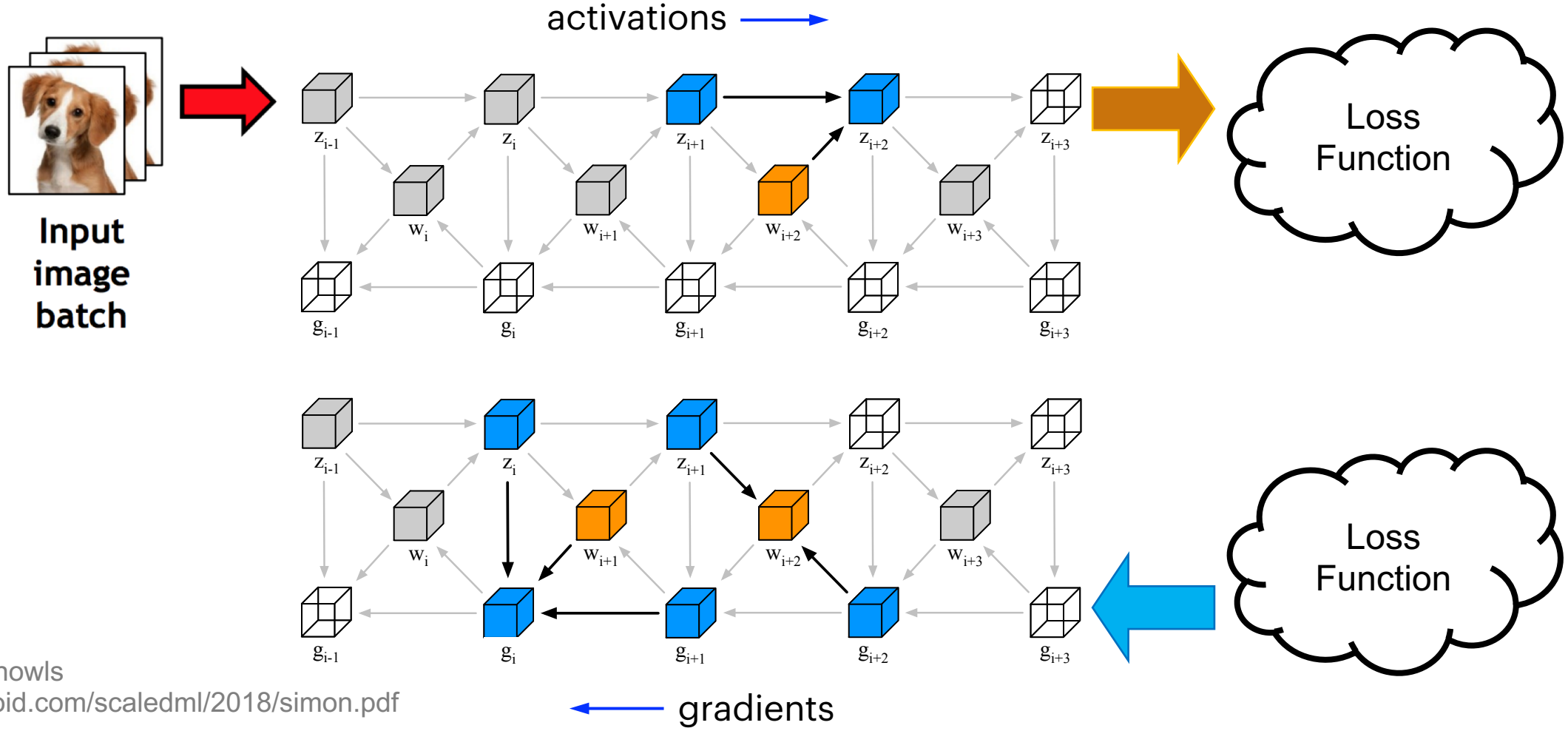
$$\frac{\partial z_1}{\partial y_1} = \underbrace{\frac{\partial z_1}{\partial y_1}}_{(1)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(2)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_3}\frac{\partial z_3}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(3)}$$

Source I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal

Bibliographic Notes & Further Reading

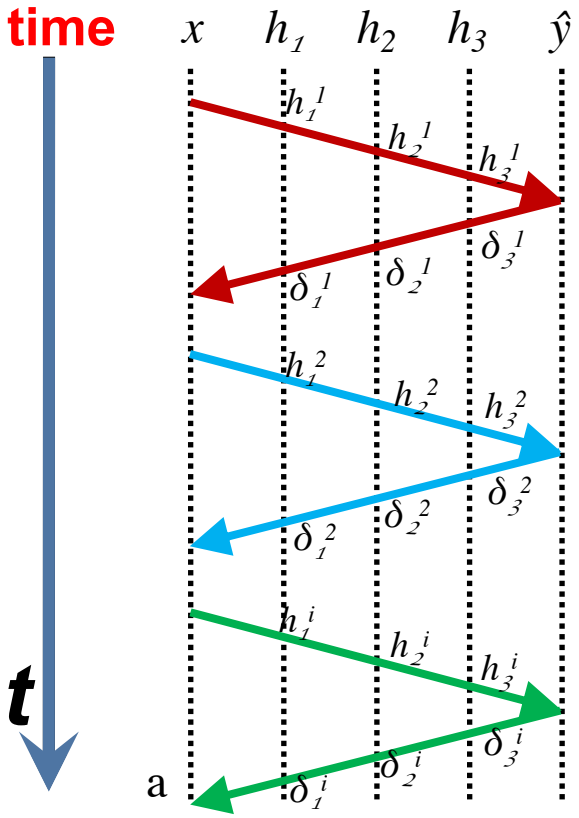
- Recent Course with Matrix-Vector Formulation of BP
  - online courses(e.g. by Hugo Larochelle)
  - Rojas (1996)'s text,

What are the data dependencies?

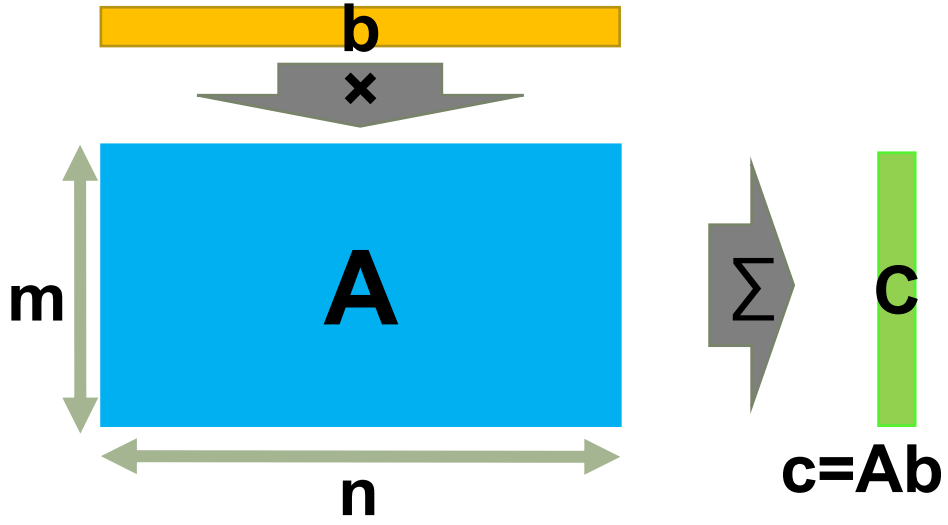


Source: Simon Knowls
https://www.matroid.com/scaledml/2018/simon.pdf

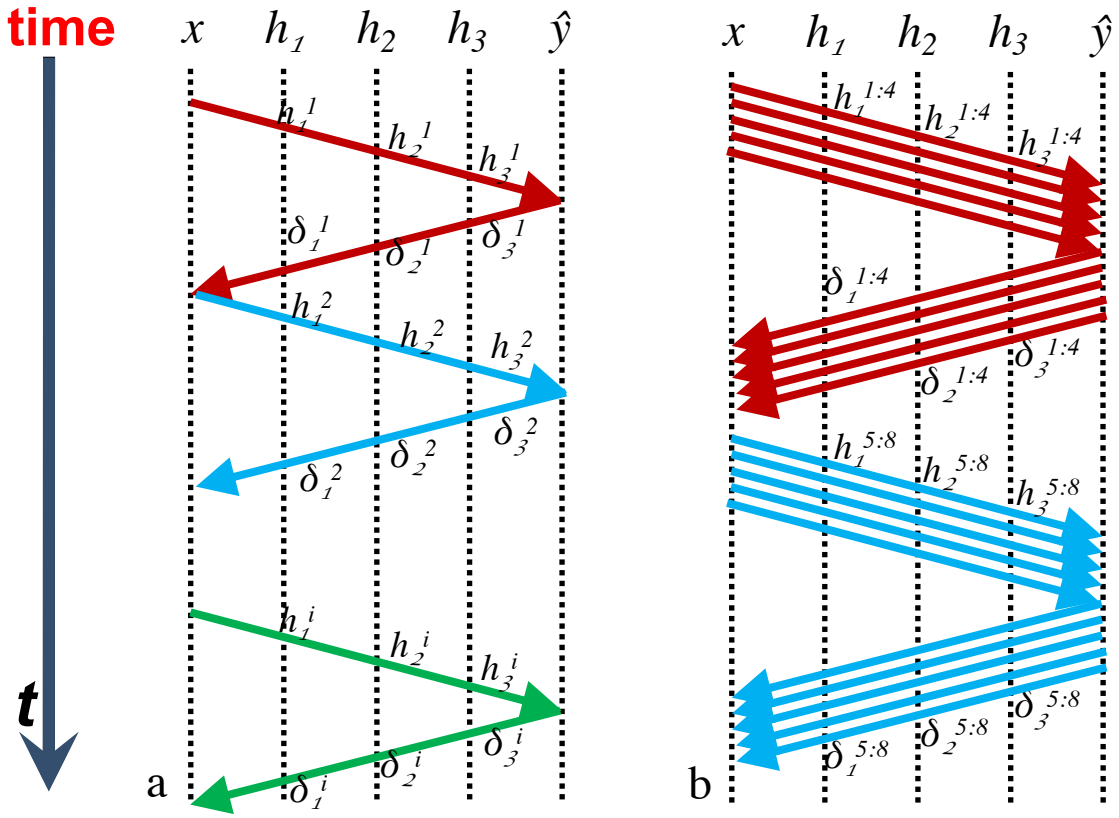
Stochastic Gradient Descent



- GEMV
  - Inherently inefficient
  - Requirements
    - Broadcast (systolic /non-systolic)
    - Reduction

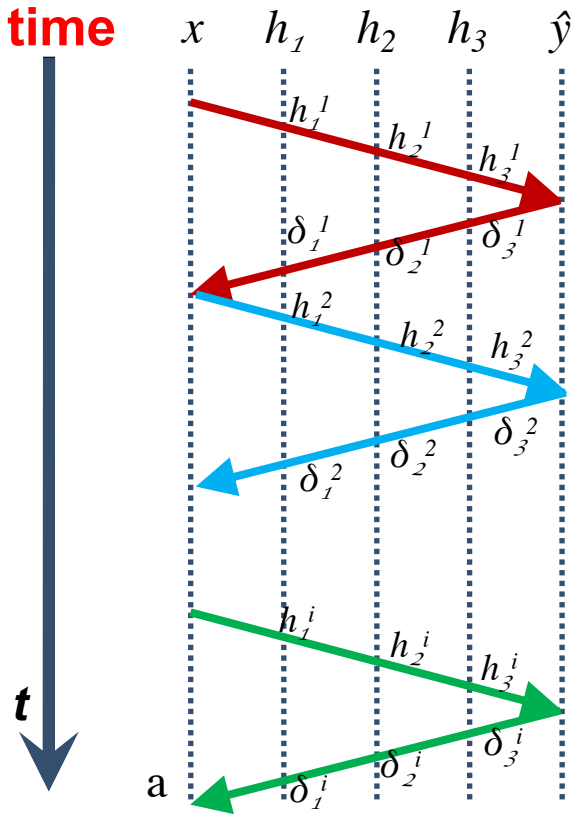


Mini-Batched Gradient Descent

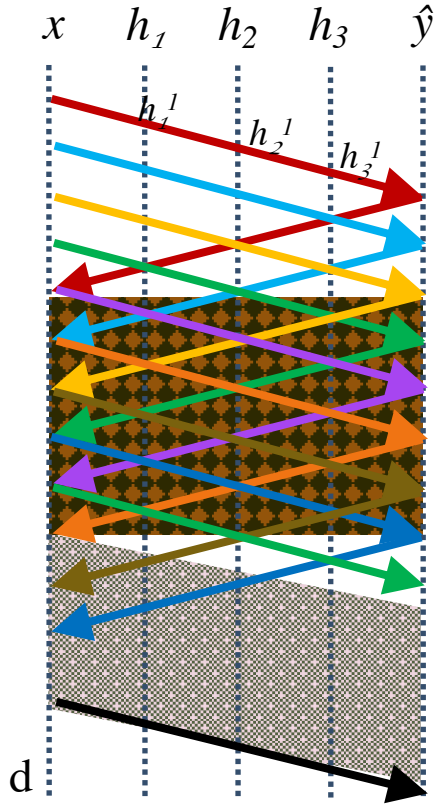


- Data parallelism
  - GEMV➔ GEMM
  - GEMM: Memory efficient kernel
  - # of weight updates / batch size

Pipelined Backpropagation

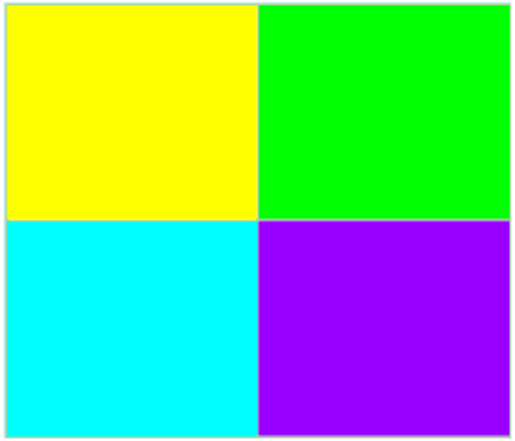


- Pipeline parallelization
  - Pipelining inputs
  - Layer locality
  - More efficient GEMVs
    - **Smaller reduction tree**
  - Weight temporal locality
    - Update and consume immediately



What about Convolution?

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}



∂h_{11}	∂h_{12}
∂h_{21}	∂h_{22}

∂hij: the derivative of the loss function w.r.t the output of the convolution layer

The backward pass is also a convolution

$$\partial W_{11} = X_{11}\partial h_{11} + X_{12}\partial h_{12} + X_{21}\partial h_{21} + X_{22}\partial h_{22}$$

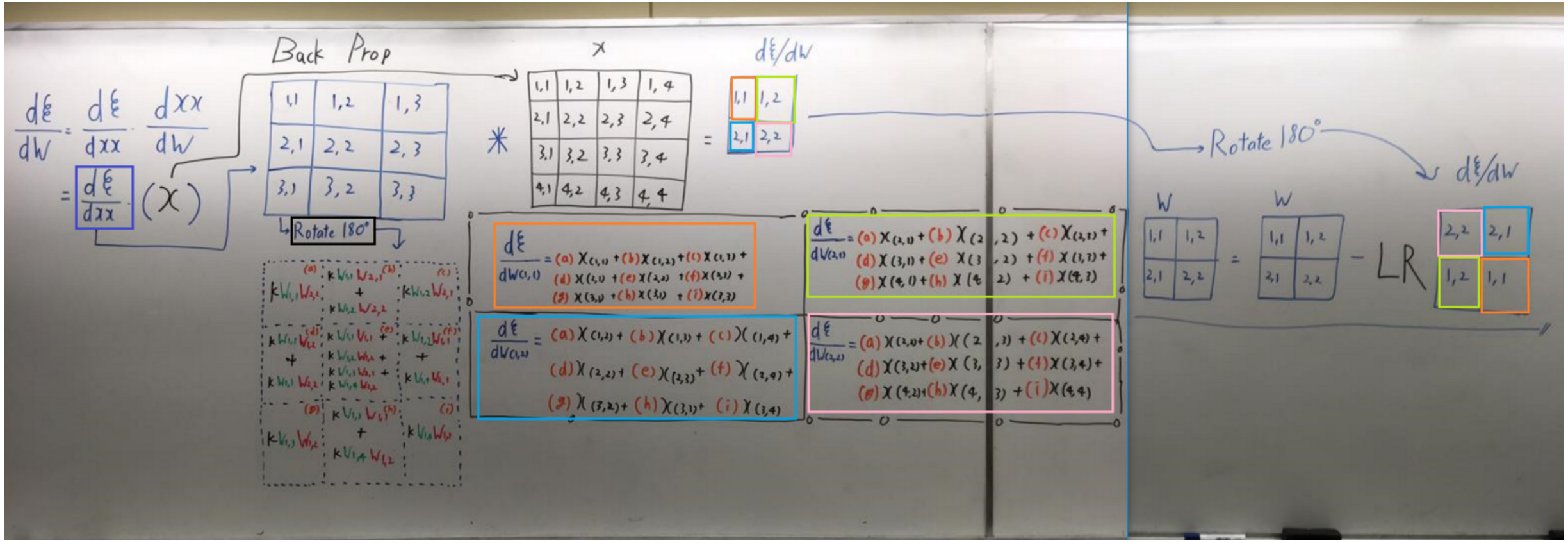
$$\partial W_{12} = X_{12}\partial h_{11} + X_{13}\partial h_{12} + X_{22}\partial h_{21} + X_{23}\partial h_{22}$$

$$\partial W_{21} = X_{21}\partial h_{11} + X_{22}\partial h_{12} + X_{31}\partial h_{21} + X_{32}\partial h_{22}$$

$$\partial W_{22} = X_{22}\partial h_{11} + X_{23}\partial h_{12} + X_{32}\partial h_{21} + X_{33}\partial h_{22}$$

https://mc.ai/demystifying-convolutional-neural-networks/

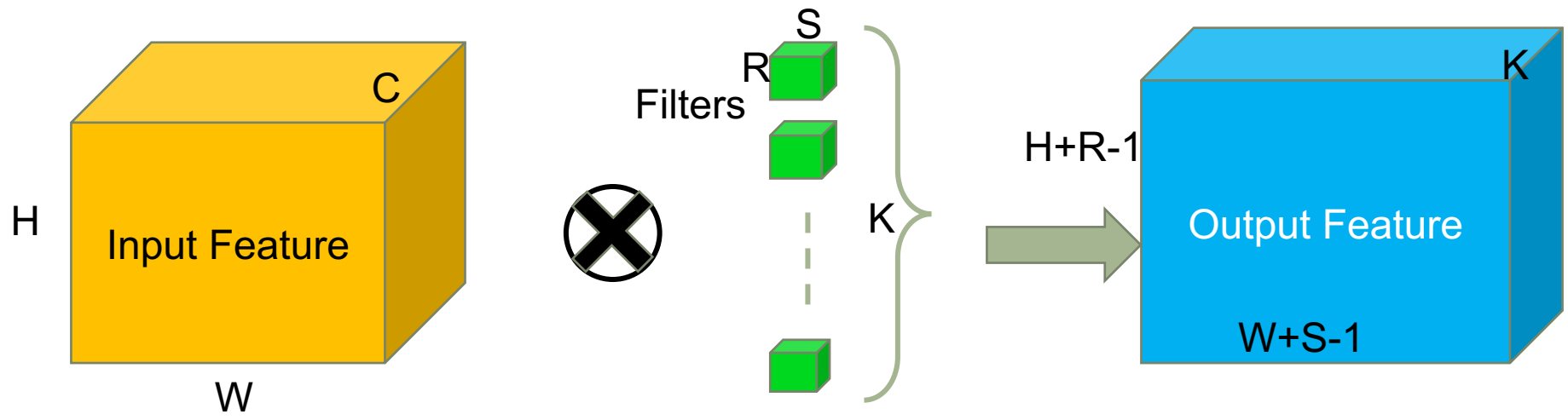
Jae Did it With Cool Details



https://towardsdatascience.com/only-numpy-understanding-back-propagation-for-transpose-convolution-in-multi-layer-cnn-with-c0a07d191981

And another video on it: https://youtu.be/XIKUuNDDJZg

Convolutional Layer Dimensions



- Activation_In: H (Height), W (Width), C (Input Channels)
- Weights : R (Kernel Height), S (Kernel Width), Output Channels (K)
- Activation_Out : H* (Height), W* (Width), K (Output Channels)

Dualities in Training Convolution Layers

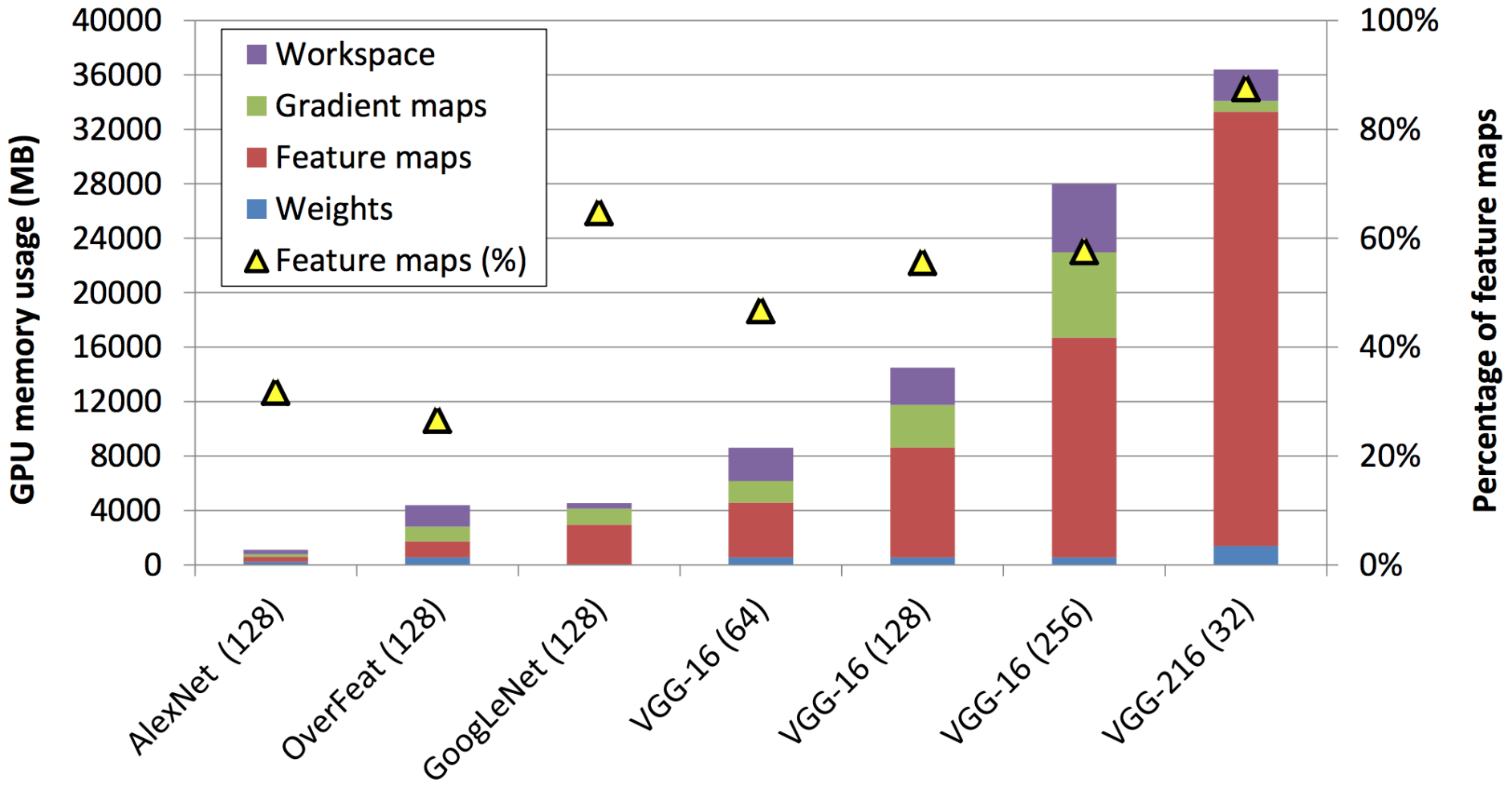
- Forward Pass Number of Input Channels is the Reduction Dimension
- Backward Pass Number of Output Channels is the Reduction Dimension
- Weight Update Performs a Convolution on large Images to create small windows of Kernels
- If you ignore the convolution with sliding window, the operation is identical to MLP

- Reduction Dimension is Highlighted
  - HWC ⊛ RSCK = H*W*K Forward
  - H*W*K ⊛ RSCK = HWC Backward
  - H*W*K ⊛ HWC = RSCK Weight Update

Memory Requirements

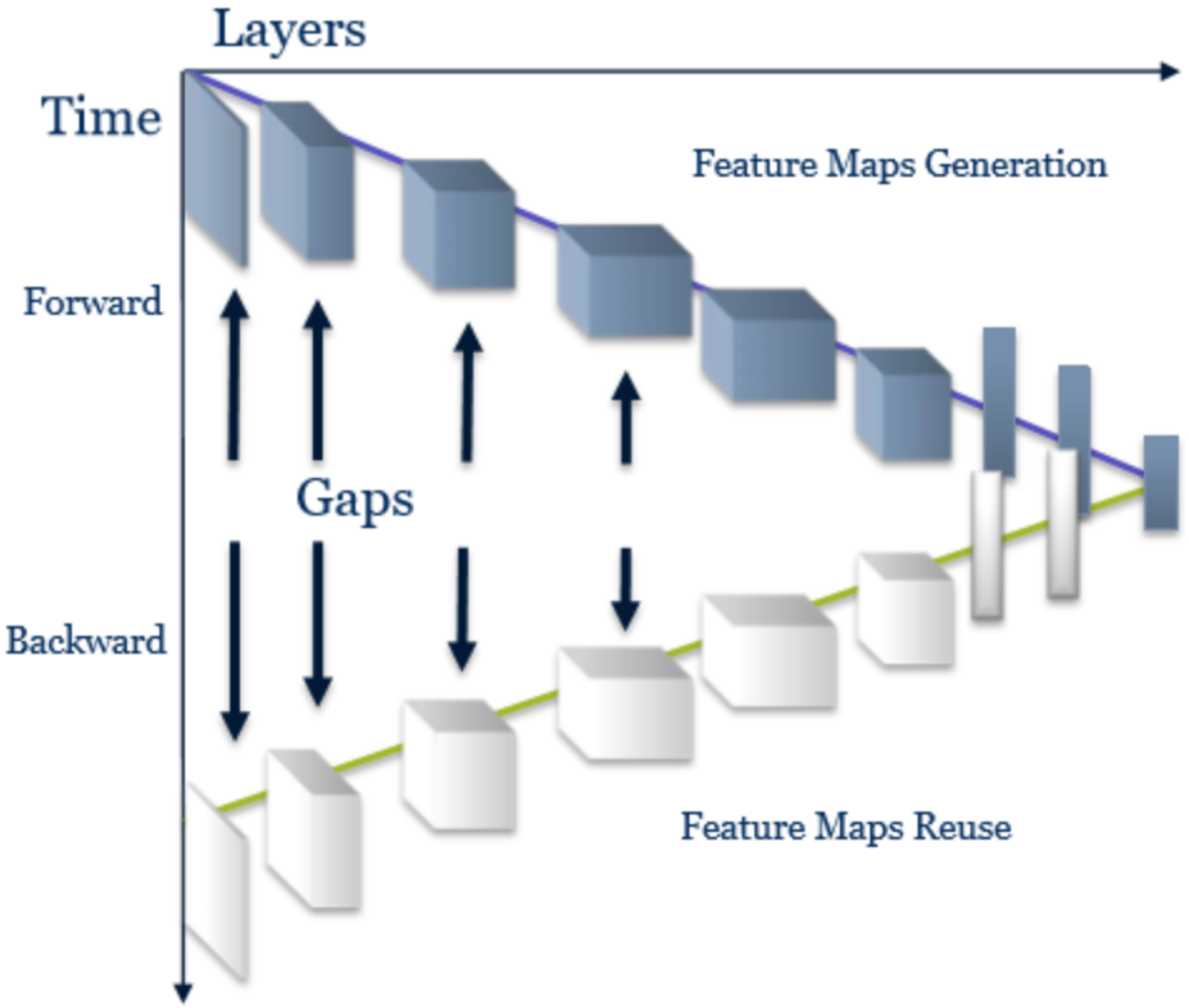
- Activations
- Weights

Activations Take Most of The Memory



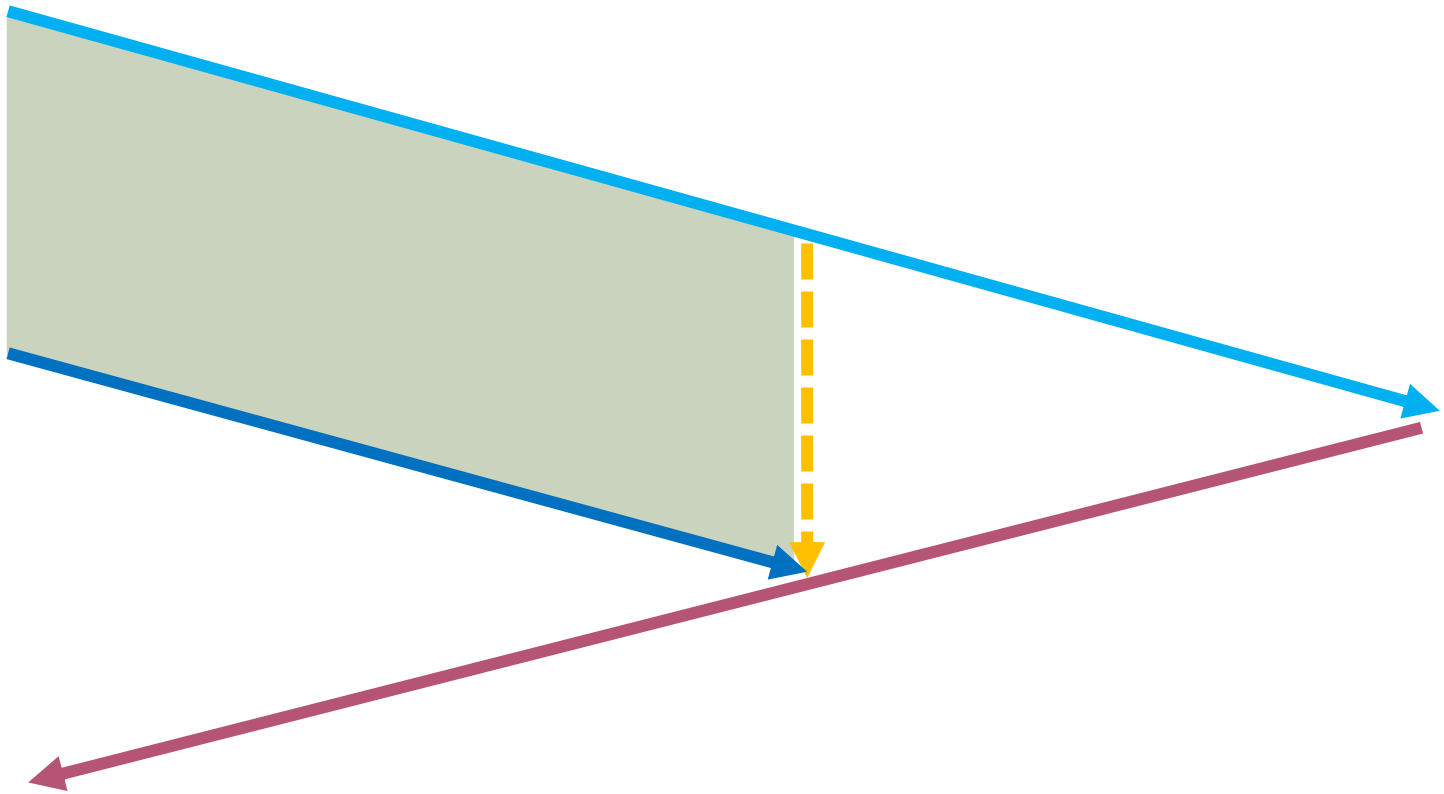
https://arxiv.org/pdf/1602.08124.pdf

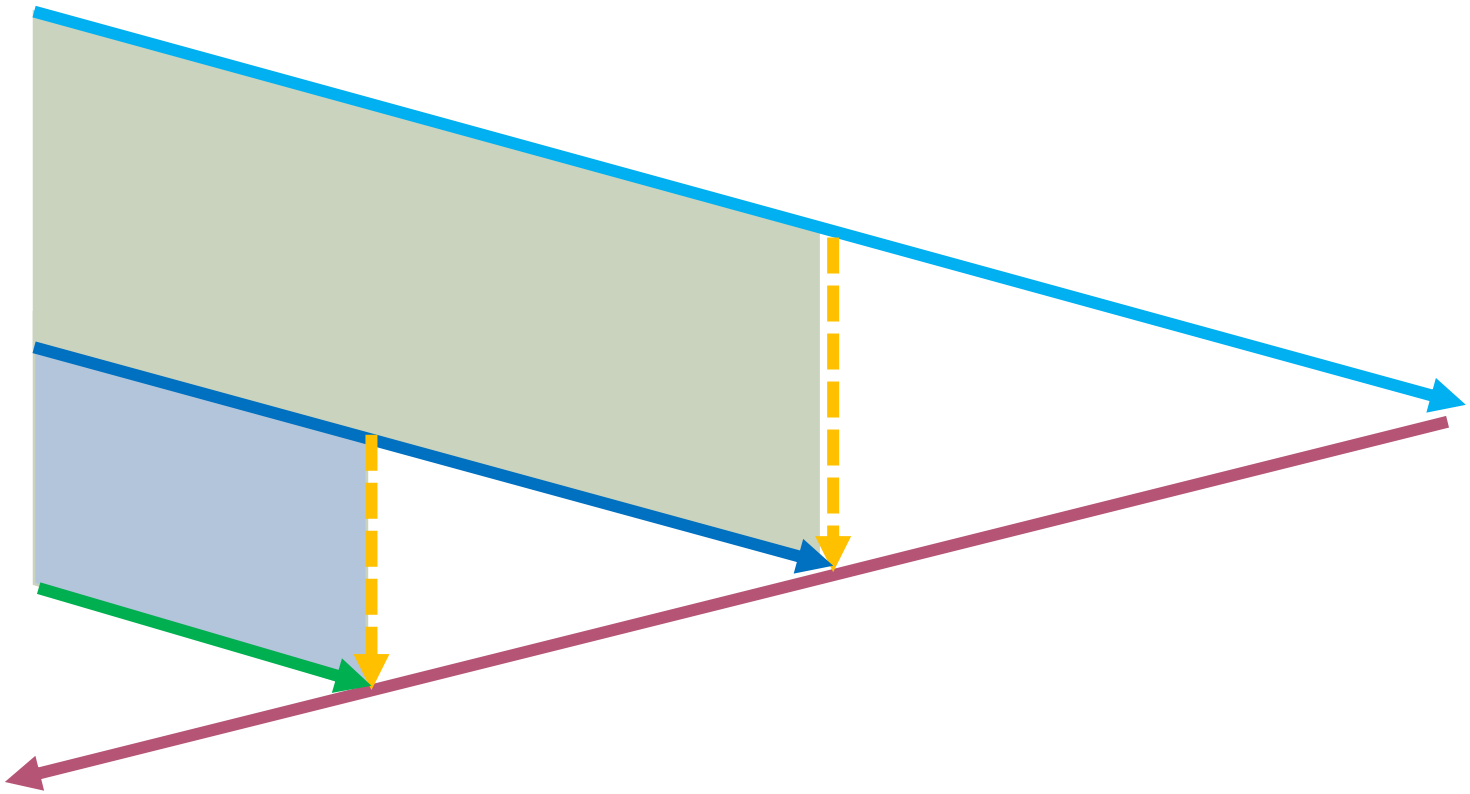
Storage is Quadratic Function of Depth



https://medium.com/syncedreview/how-to-train-a-very-large-and-deep-model-on-one-gpu-7b7edfe2d072

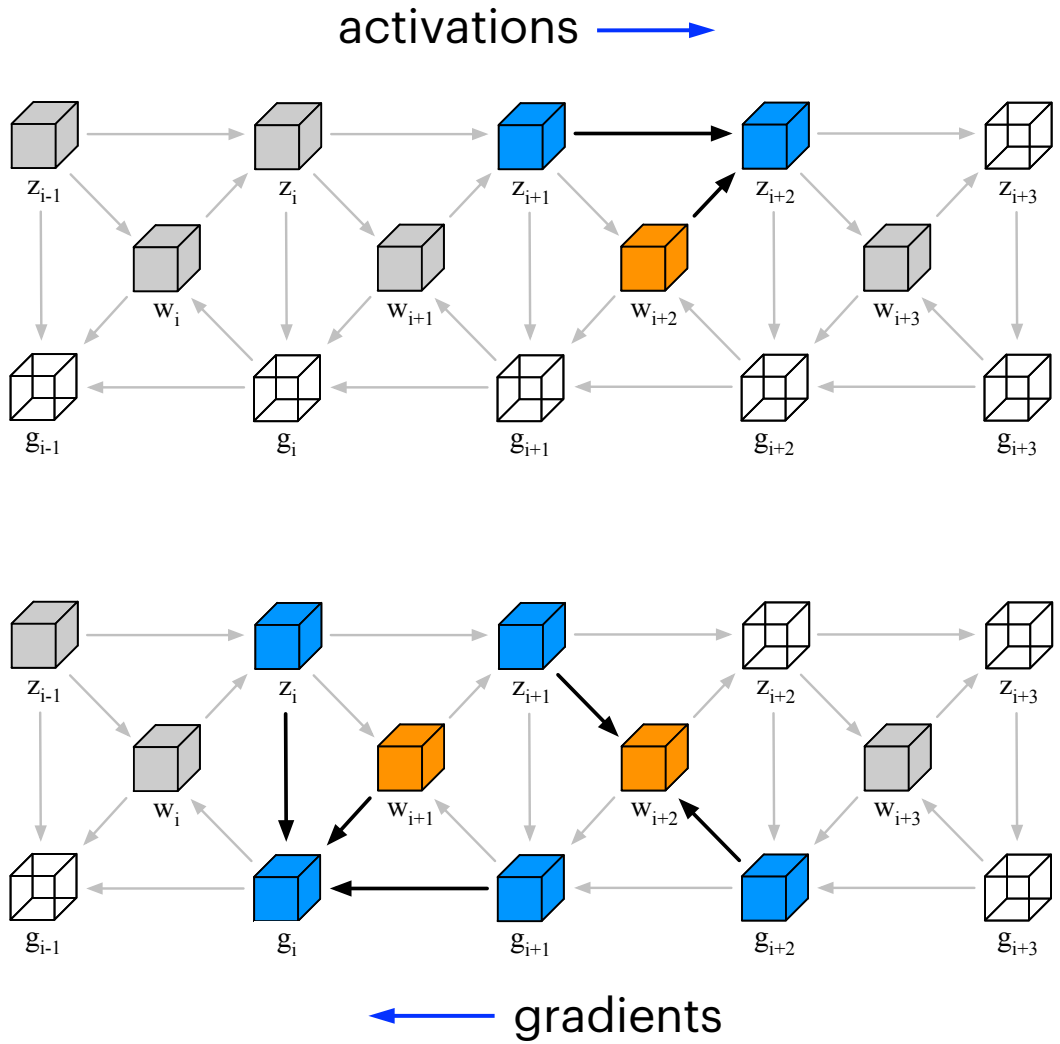
Reverse Checkpointing





Recompute WHAT YOU Can't Remember

- Reverse Checkpointing
- Recompute the Activations from sparse snapshots
- Trade most storage for one repeat of forwards pass compute



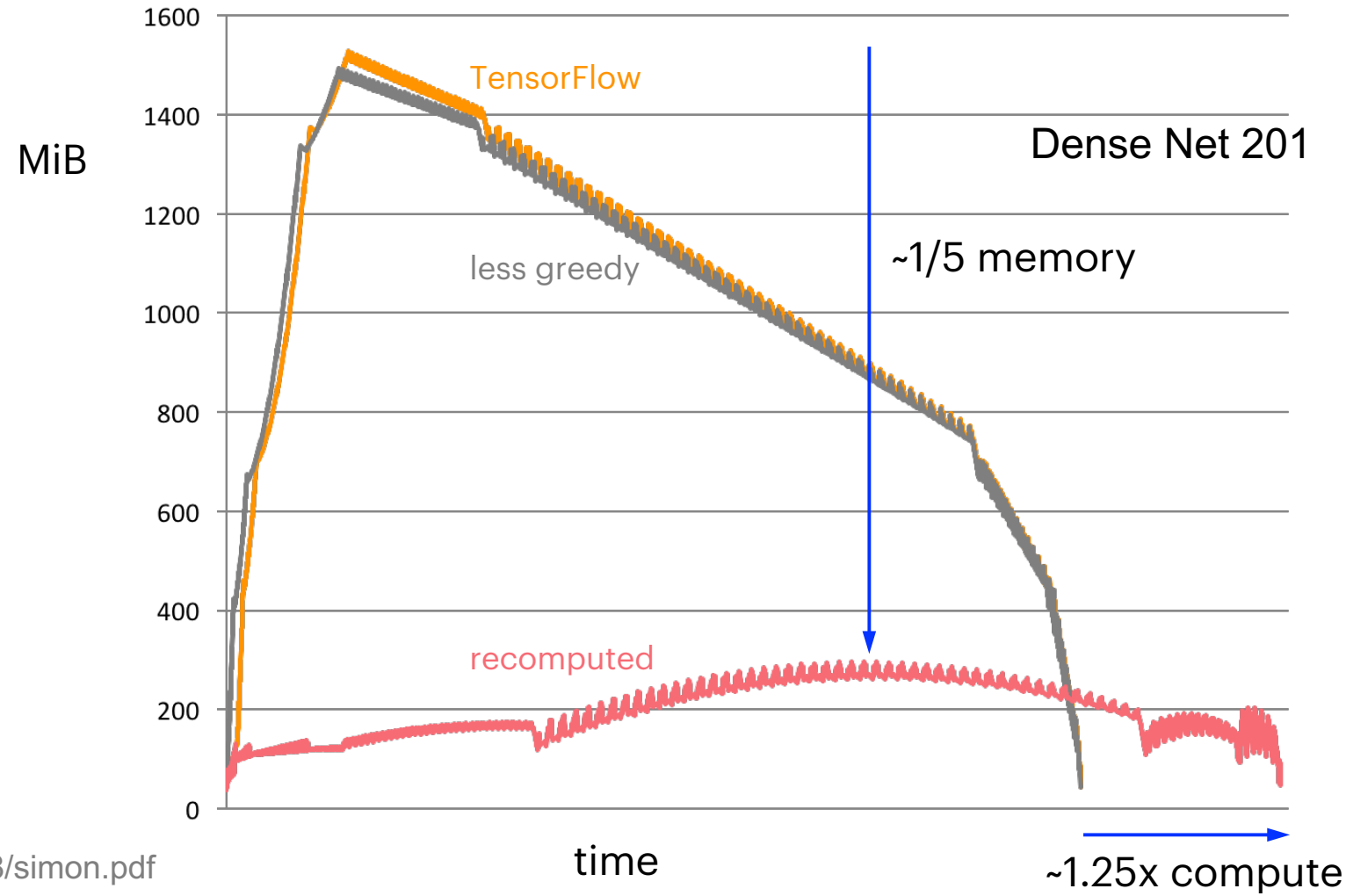
Source: Simon Knowls
https://www.matroid.com/scaledml/2018/simon.pdf

25% More Compute Saves 80% memory

Naive strategy: memorize activations only at input of each residue block

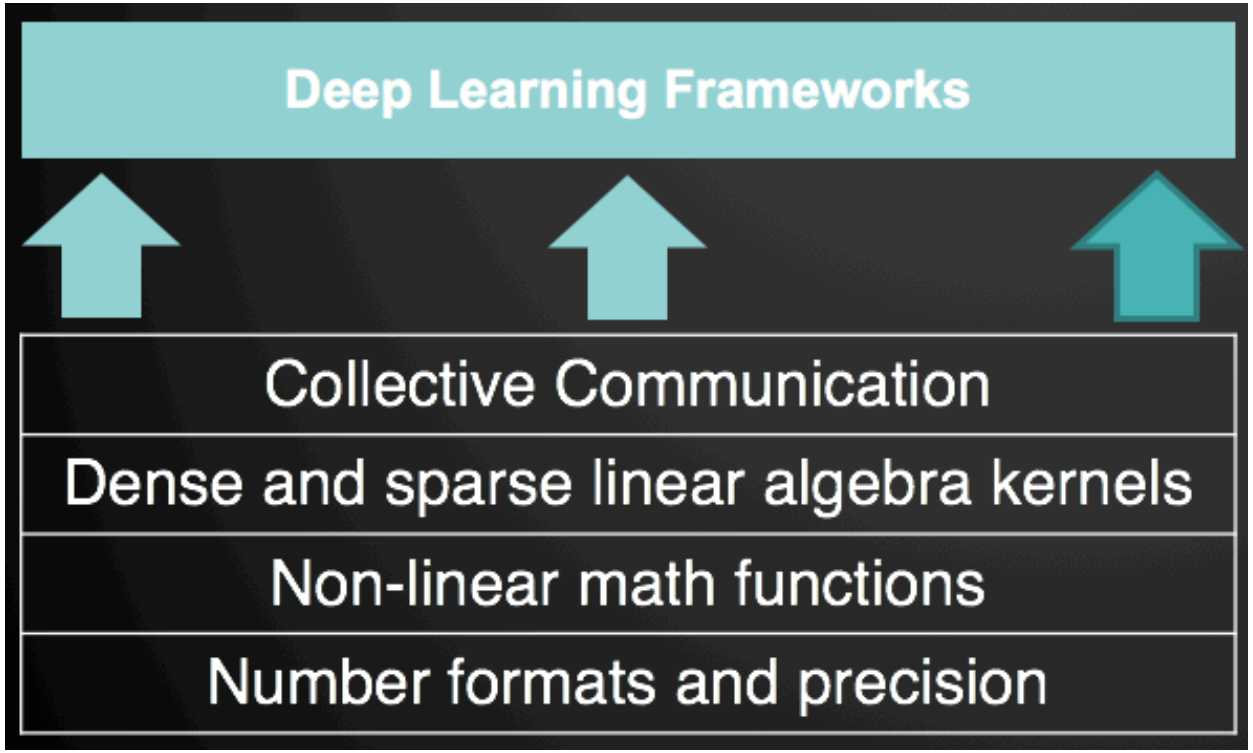
Batch=16 executing on CPU, recording total memory allocated for weights + activations.

f16 weights and activations, single weight copy.



Source: Simon Knowls
https://www.matroid.com/scaledml/2018/simon.pdf

Deep Learning – Fixed Function Workload



- DL is fixed fixed function workload
- Spends most time in matrix multiply
- Maps well to specialized ASIC
- Fast, energy-efficient (high reuse), co-designed

What Do We Need to Support?

- **GEMM:** General Matrix Matrix multiplication
- **GEMV:** General Matrix Vector multiplication
- **Parallel Patterns**
  - **Collective communications**
    - ❖Gather
    - ❖Reduce
    - ❖All gather
    - ❖All reduce
    - ❖Broadcast
    - ❖All-to-All

Collective Communications

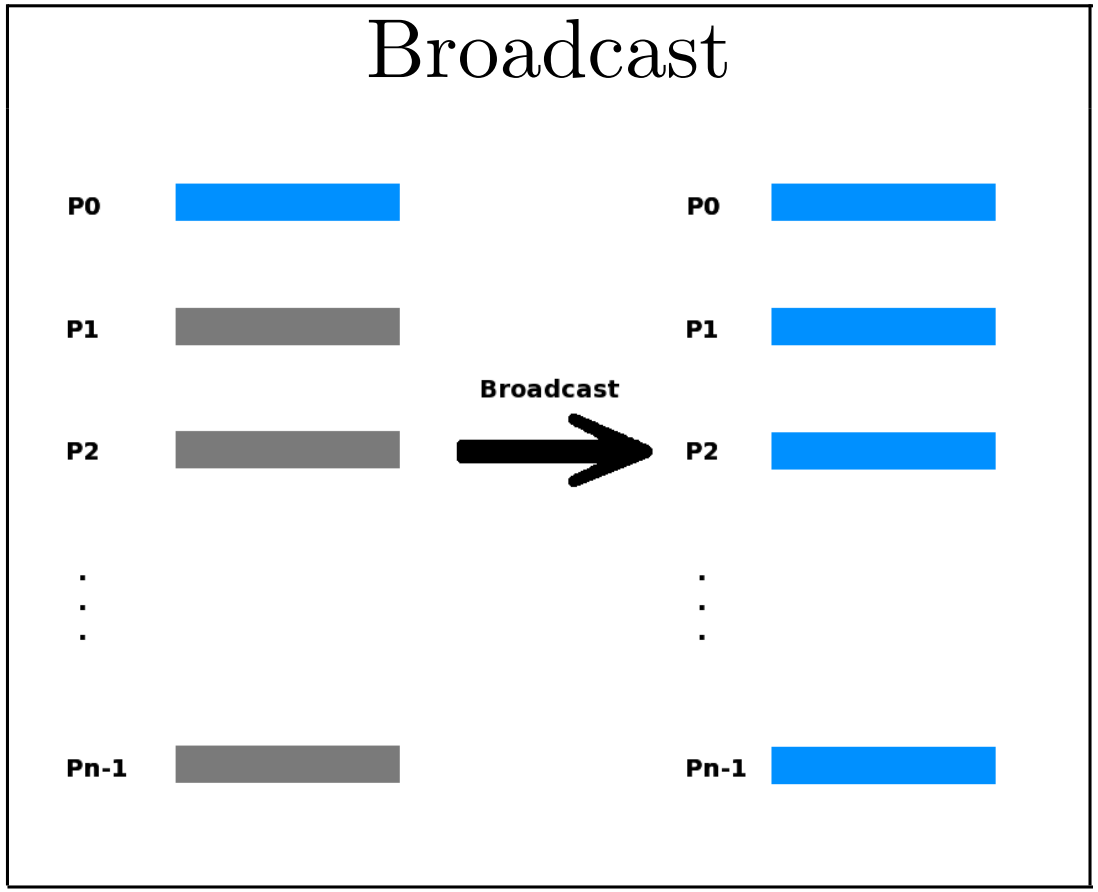
- Collective operations are called by all processes within a communicator
- Various algorithms to implement them based on
  - Latency
  - BW
- Can Combine two or more to create an algorithm for a third collective

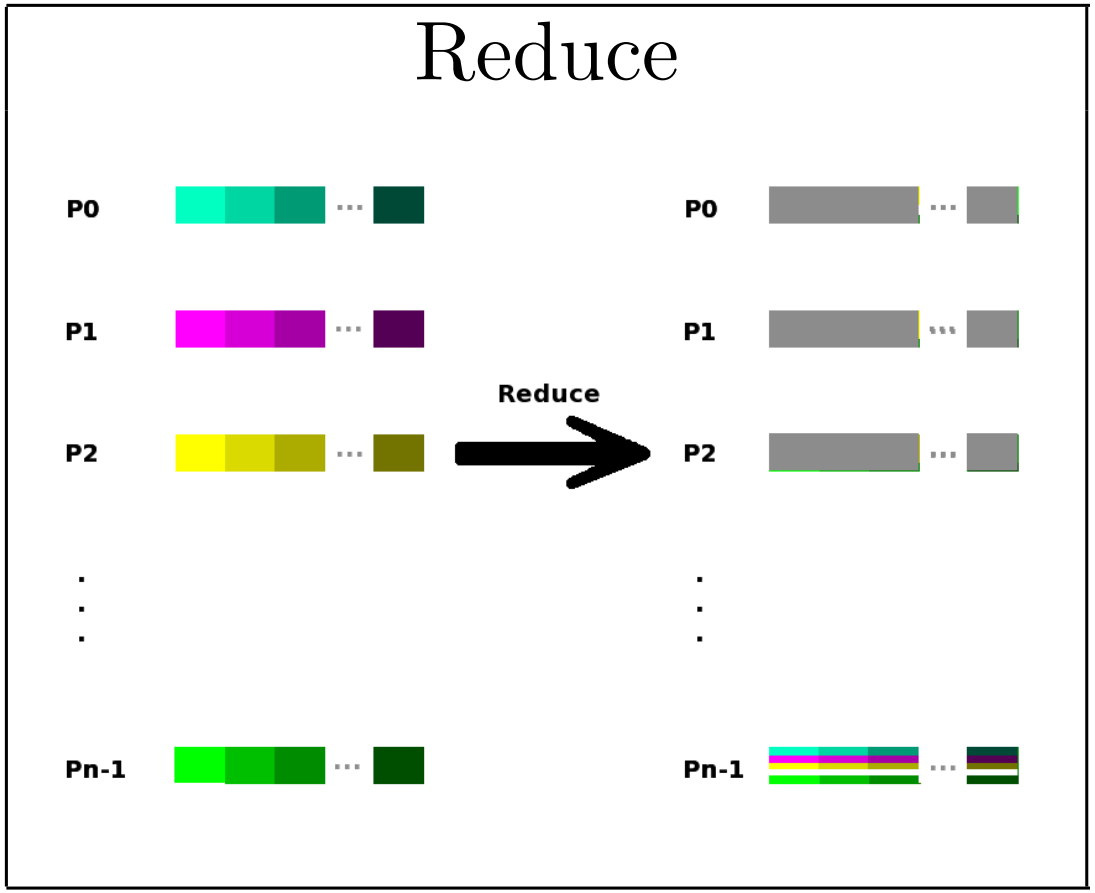
https://trace.tennessee.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1320&context=utk_graddiss

Collective Communications

Operation	Before				After			
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Permute	x_0	x_1	x_2	x_3	x_1	x_0	x_3	x_2
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Broadcast	x				x	x	x	x
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Reduce(-to-one)	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$			

Collective Communications



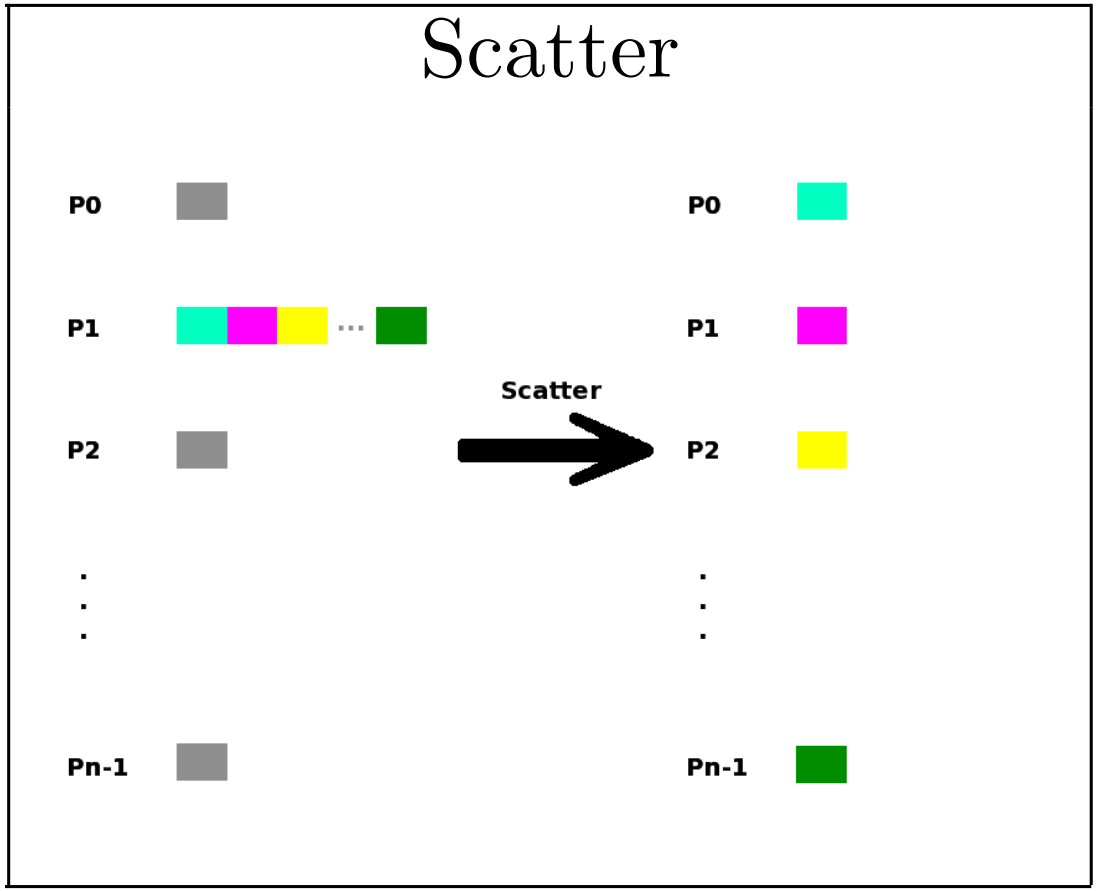


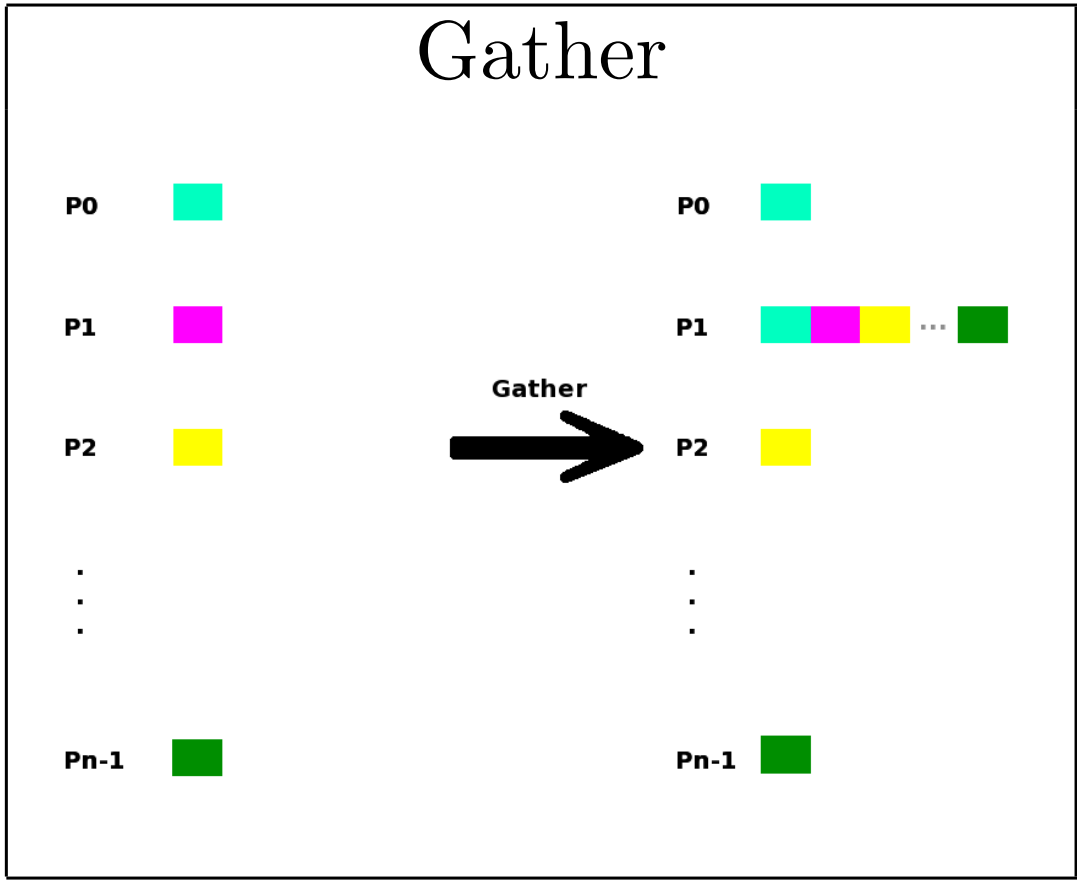
Source: Jelena Pjesivac-Grbovic

Collective Communications

Scatter	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0				x_0			
	x_1					x_1		
	x_2						x_2	
	x_3							x_3
Gather	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0				x_0			
		x_1			x_1			
			x_2		x_2			
				x_3	x_3			

Collective Communications



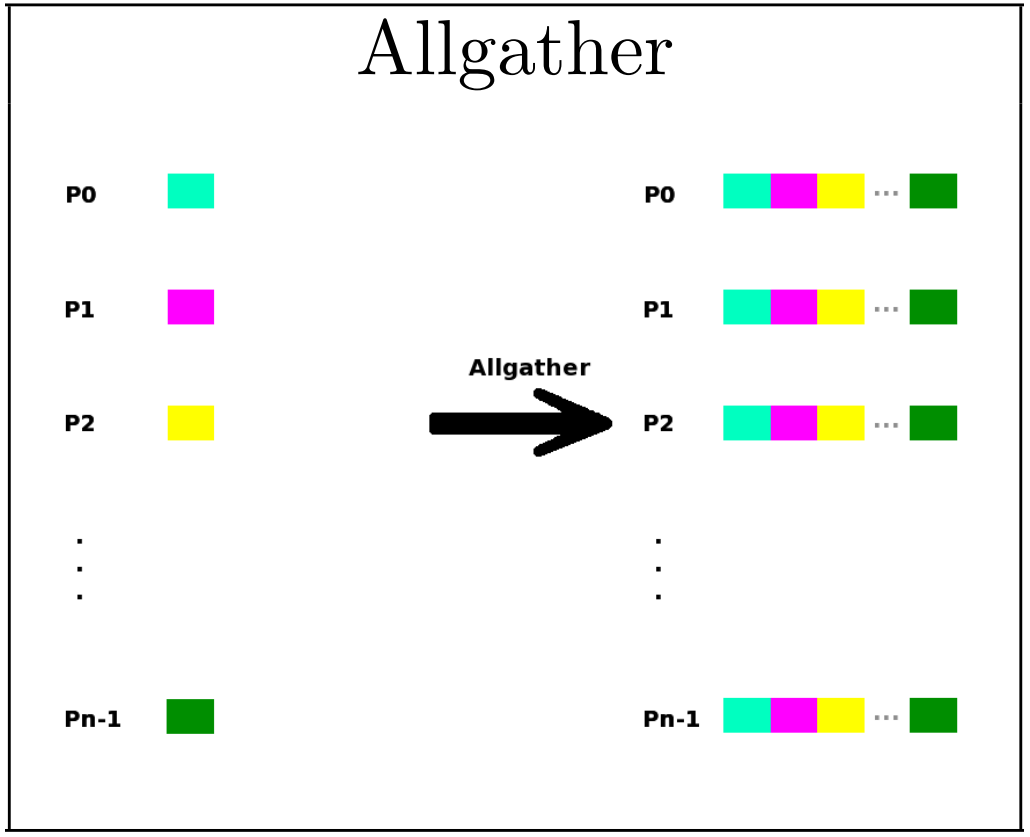


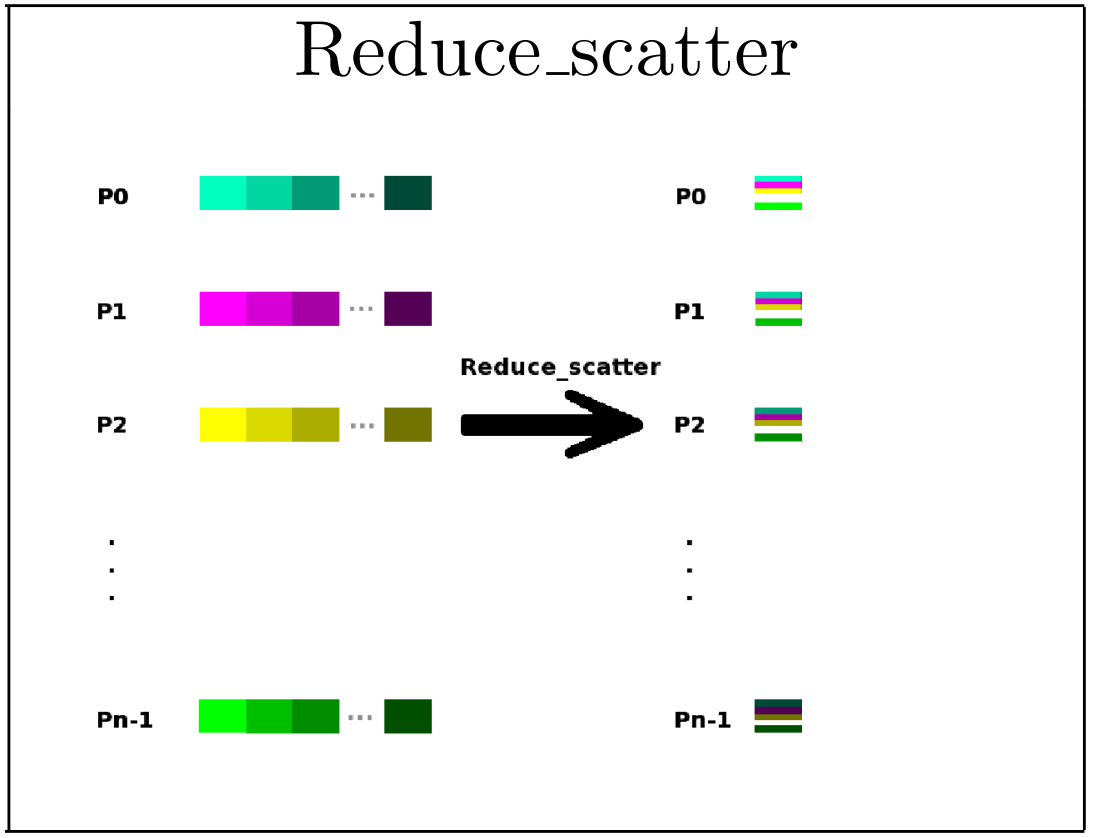
Source: Jelena Pjesivac-Grbovic

Collective Communications

Allgather	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0				x_0	x_0	x_0	x_0
		x_1			x_1	x_1	x_1	x_1
			x_2		x_2	x_2	x_2	x_2
				x_3	x_3	x_3	x_3	x_3
Reduce-scatter	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$	$\sum_j x_0^{(j)}$			
	$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$		$\sum_j x_1^{(j)}$		
	$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$			$\sum_j x_2^{(j)}$	
	$x_3^{(0)}$	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(3)}$				$\sum_j x_3^{(j)}$

Collective Communications



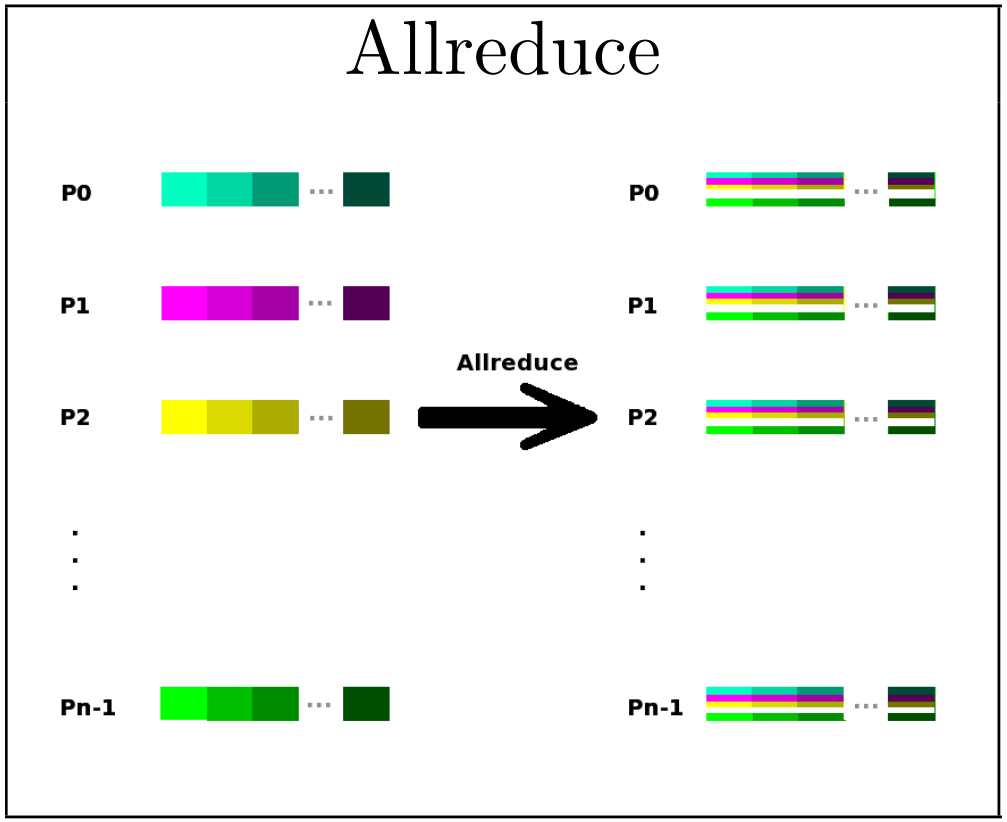


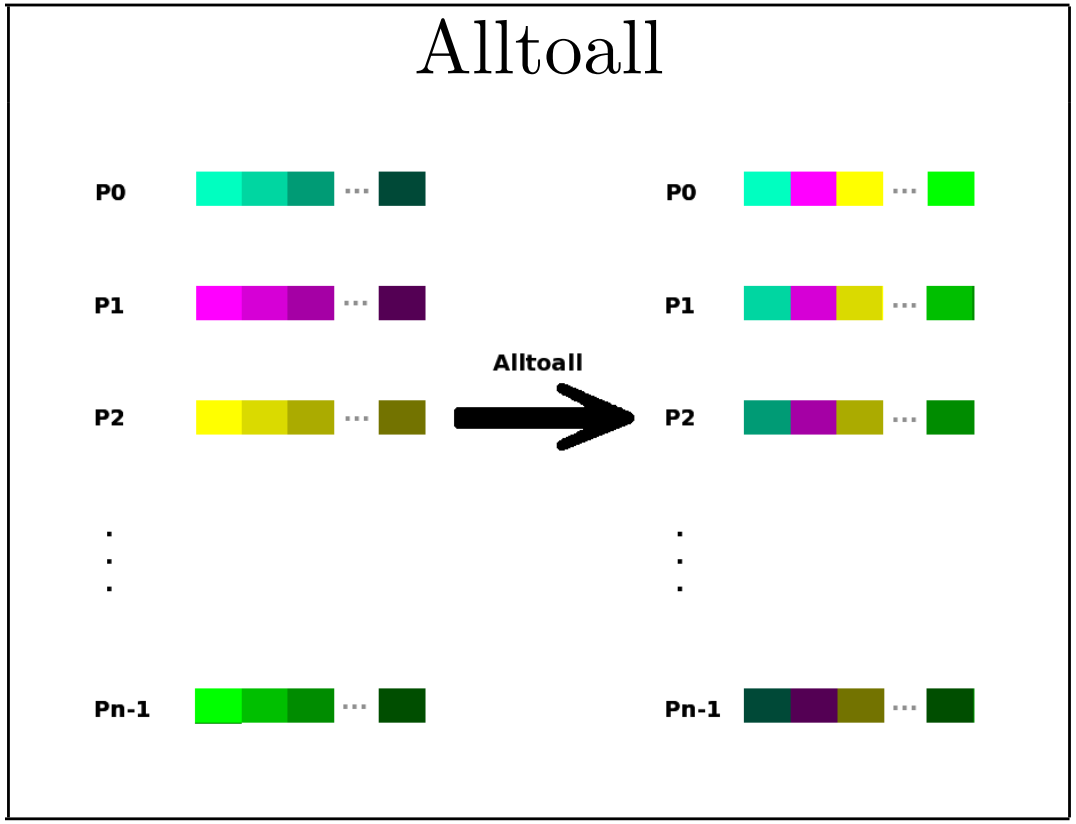
Source: Jelena Pjesivac-Grbovic

Collective Communications

Allreduce	Node 0 | Node 1 | Node 2 | Node 3 $x^{(0)}$ | $x^{(1)}$ | $x^{(2)}$ | $x^{(3)}$	Node 0 | Node 1 | Node 2 | Node 3 $\sum_j x^{(j)}$ | $\sum_j x^{(j)}$ | $\sum_j x^{(j)}$ | $\sum_j x^{(j)}$
All-to-all	Node 0 | Node 1 | Node 2 | Node 3 $x_0^{(0)}$ | $x_0^{(1)}$ | $x_0^{(2)}$ | $x_0^{(3)}$ $x_1^{(0)}$ | $x_1^{(1)}$ | $x_1^{(2)}$ | $x_1^{(3)}$ $x_2^{(0)}$ | $x_2^{(1)}$ | $x_2^{(2)}$ | $x_2^{(3)}$ $x_3^{(0)}$ | $x_3^{(1)}$ | $x_3^{(2)}$ | $x_3^{(3)}$	Node 0 | Node 1 | Node 2 | Node 3 $x_0^{(0)}$ | $x_1^{(0)}$ | $x_2^{(0)}$ | $x_3^{(0)}$ $x_0^{(1)}$ | $x_1^{(1)}$ | $x_2^{(1)}$ | $x_3^{(1)}$ $x_0^{(2)}$ | $x_1^{(2)}$ | $x_2^{(2)}$ | $x_3^{(2)}$ $x_0^{(3)}$ | $x_1^{(3)}$ | $x_2^{(3)}$ | $x_3^{(3)}$

Collective Communications





Source: Jelena Pjesivac-Grbovic

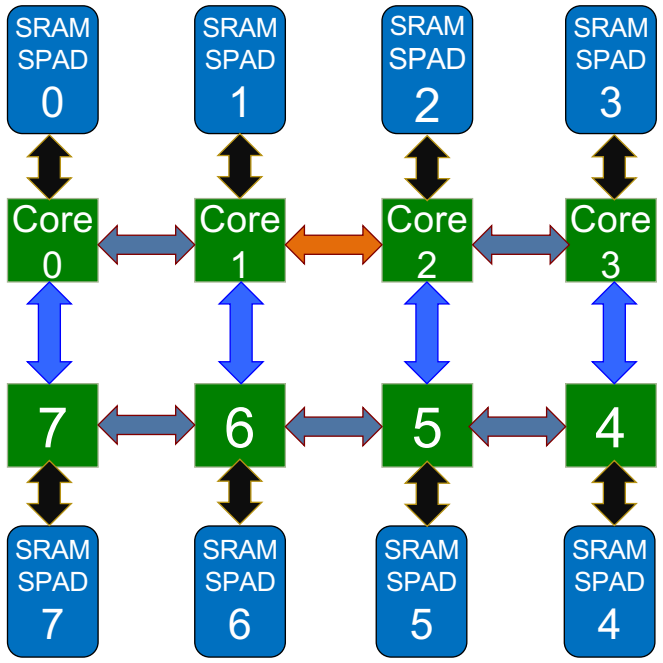
Collective Communications

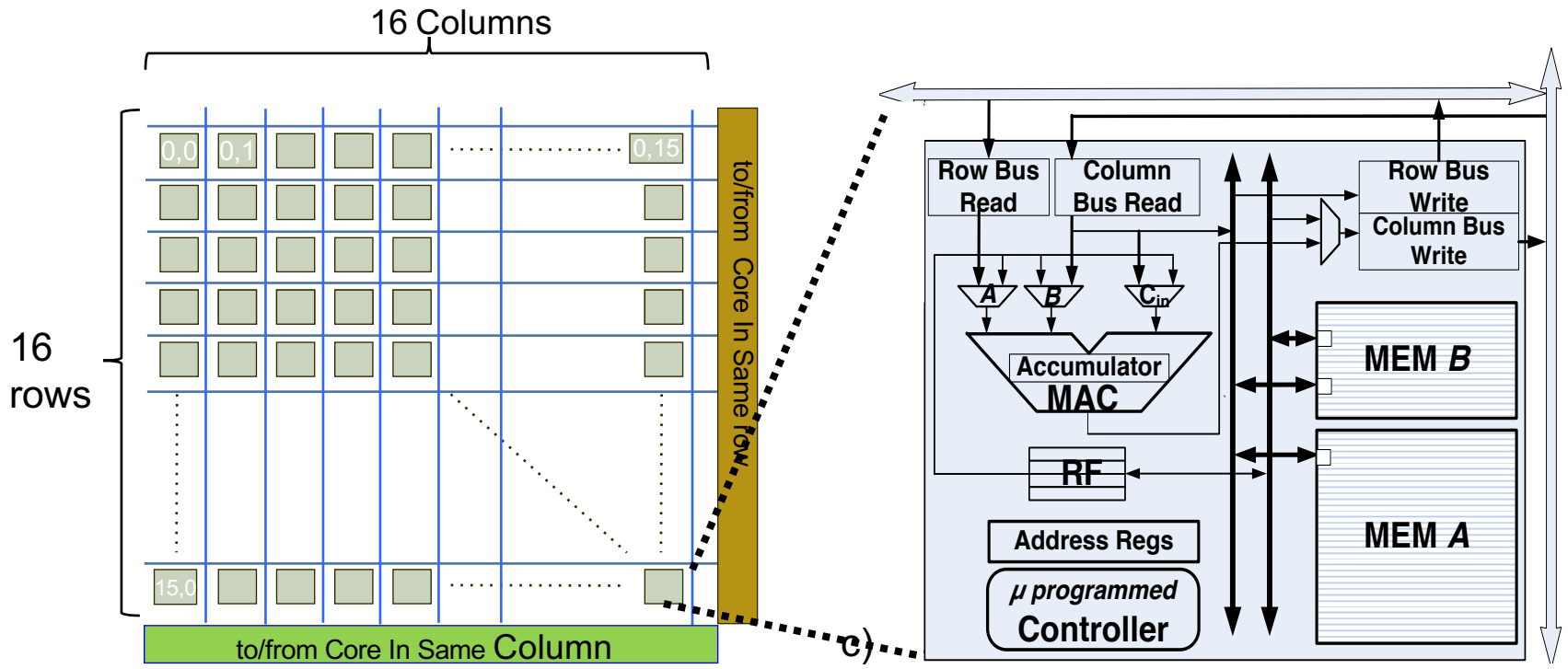
Operation	personalized?	small messages	large messages (segmented)
Broadcast	no	Flat/Binary/N-ary/Binomial Tree	Piplelined/Double/Split Binary Tree, Chain, Van de Geijn Algorithm, HW specific multicast
Barrier	no	Flat/Binary/Binomial Tree, Dissemination (butterfly), Tournament	
Reduce	no	Flat/Binary/N-ary/Binomial Tree, Gather + Reduce	Piplelined/Double/Split Binary Tree, Chain, Gather + Reduce, Vector halving + distance doubling + Binomial Tree
Scatter	yes	Flat/Binary/N-ary/Binomial Tree	Piplelined/Double Tree, Chain
Gather	no	Flat/Binary/N-ary/Binomial Tree	Piplelined/Double Tree, Chain
Allgather	no	Recursive Doubling, Gather + Broadcast, Bruck	Ring
Allreduce	no	Recursive Doubling, Bruck (with reduce) , Allgather followed by Reduce, Reduce followed by Broadcast	Ring, Rabenseifner Algorithm, Recursive Doubling, Vector Halving with Distance Doubling, Binary blocks
AlltoAll	yes		

Source: Jelena Pjesivac-Grbovic

A Case Study for MLPs

CATERPILLAR Architecture

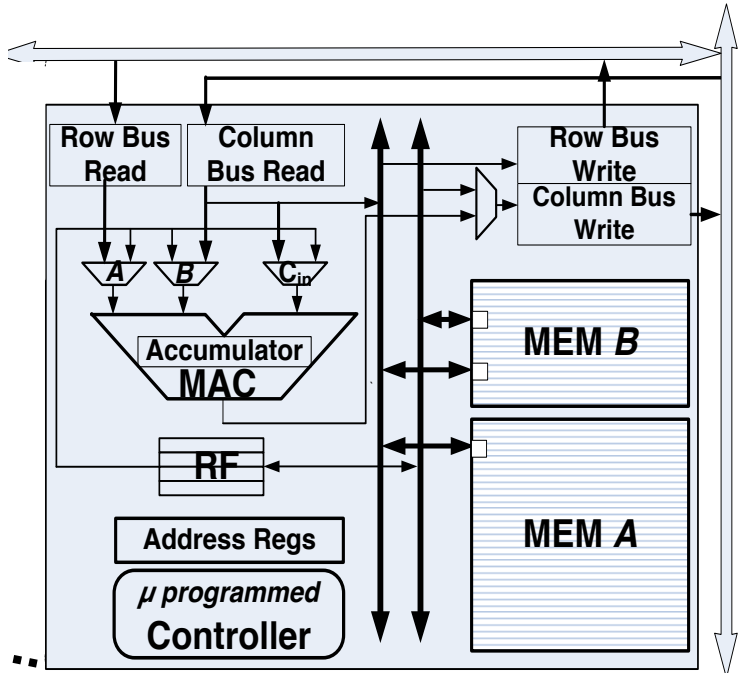




https://arxiv.org/abs/1706.00517

CATERPILLAR Architecture

- PE
  - Native Support for Inner Product
  - 3 Levels of memory hierarchy
    - Accumulator
    - Mem B (2 ports)
    - Mem A (1 port)
  - Distributed Memory Programming Model
  - State Machine Reprogrammable

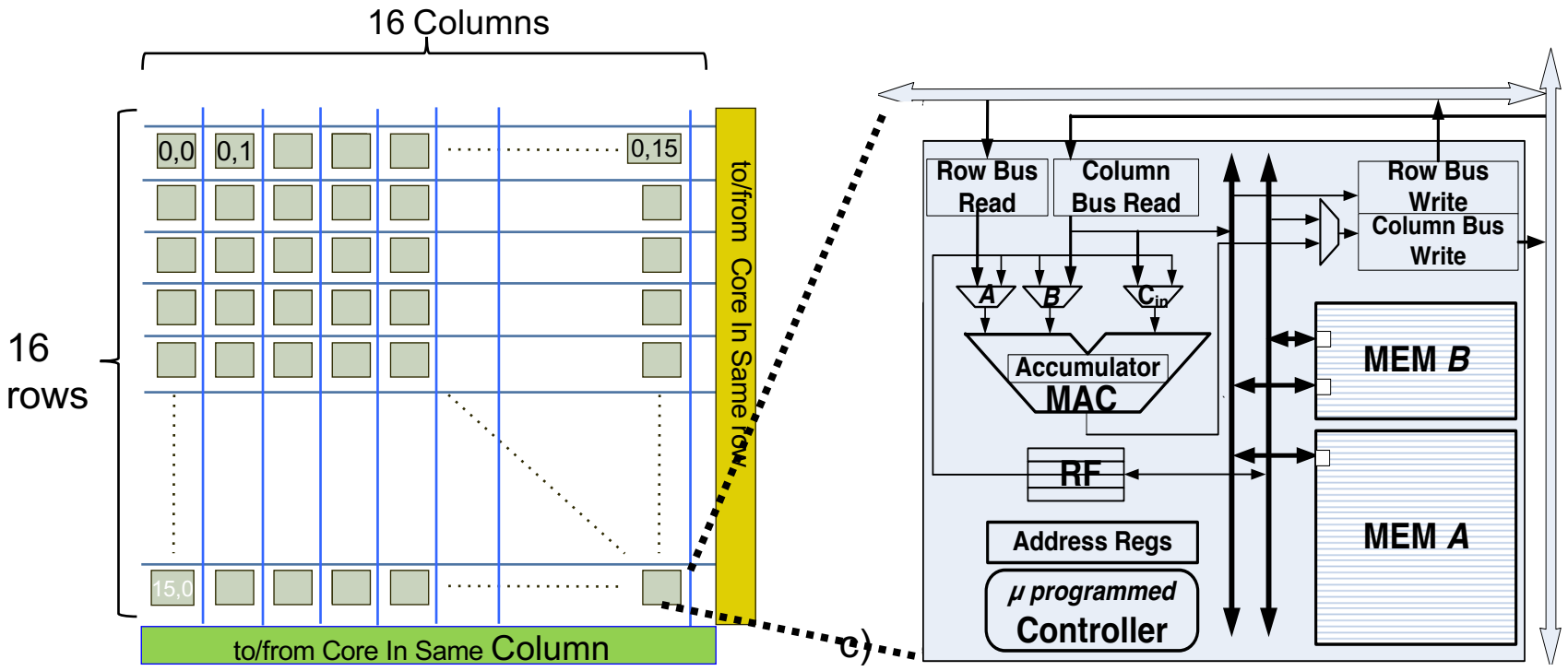


http://users.ece.utexas.edu/~gerstl/publications/UT-CERC-12-02.LAP.pdf

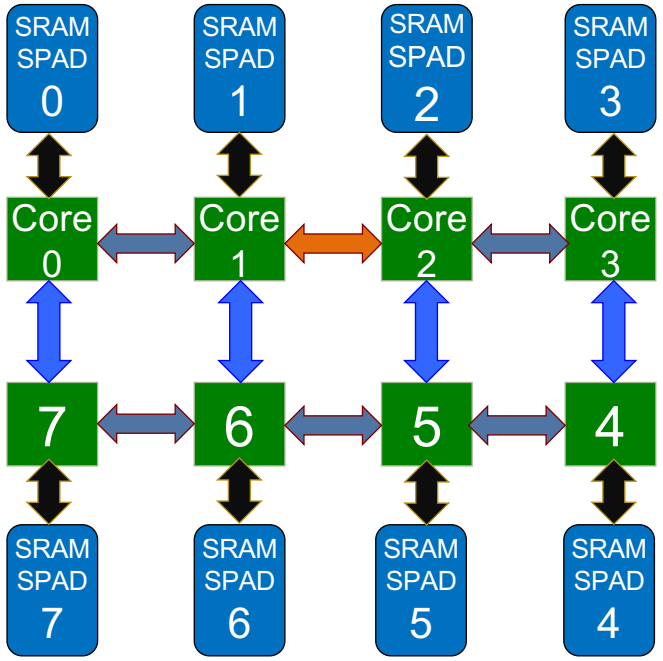
The Linear Algebra Core PE

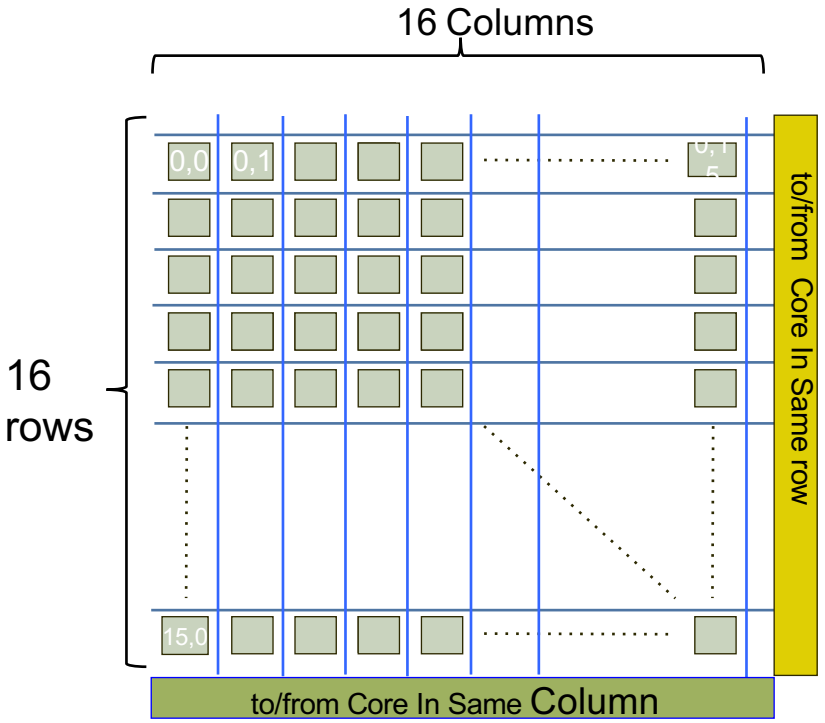
CATERPILLAR Architecture

- Core
  - GEMM
    - Optimized for Rank-1 Updates
    - Broadcast bus
  - GEMV
    - Systolic between neighboring PEs
    - Accelerate reduction



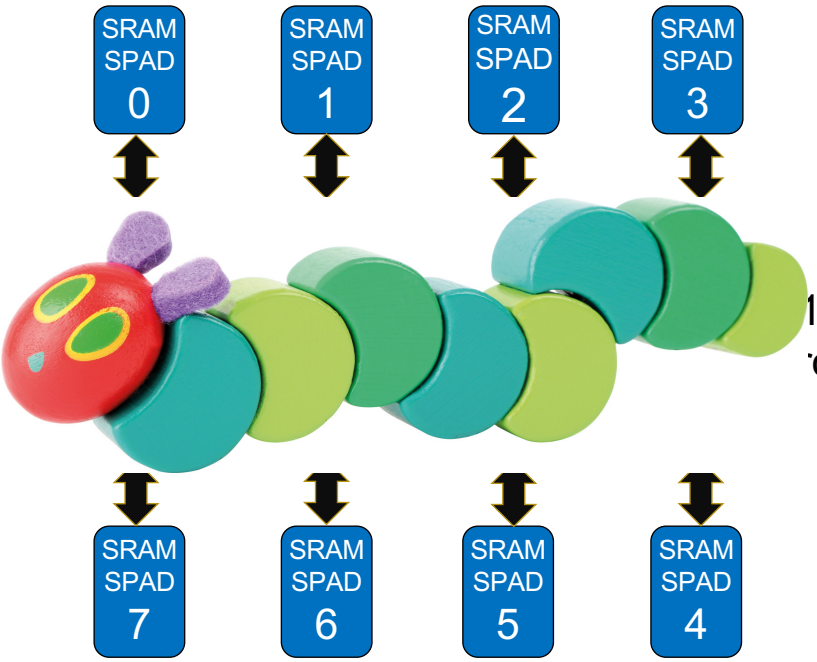
CATERPILLAR Architecture

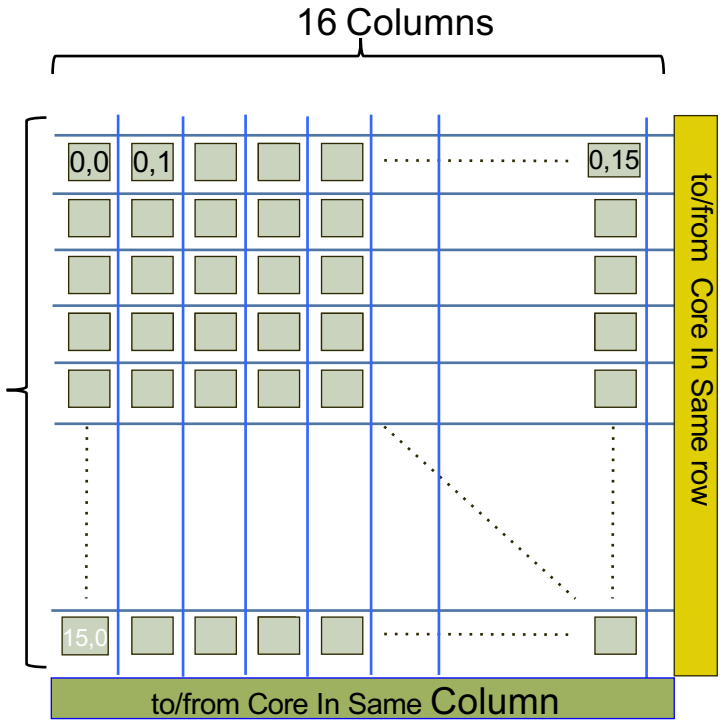




- Multicore
  - Ring of Cores
  - Reconfigurable
- Support for Collective Comms
  - All gather
  - Reduce
  - All Reduce
  - Systolic/Parallel

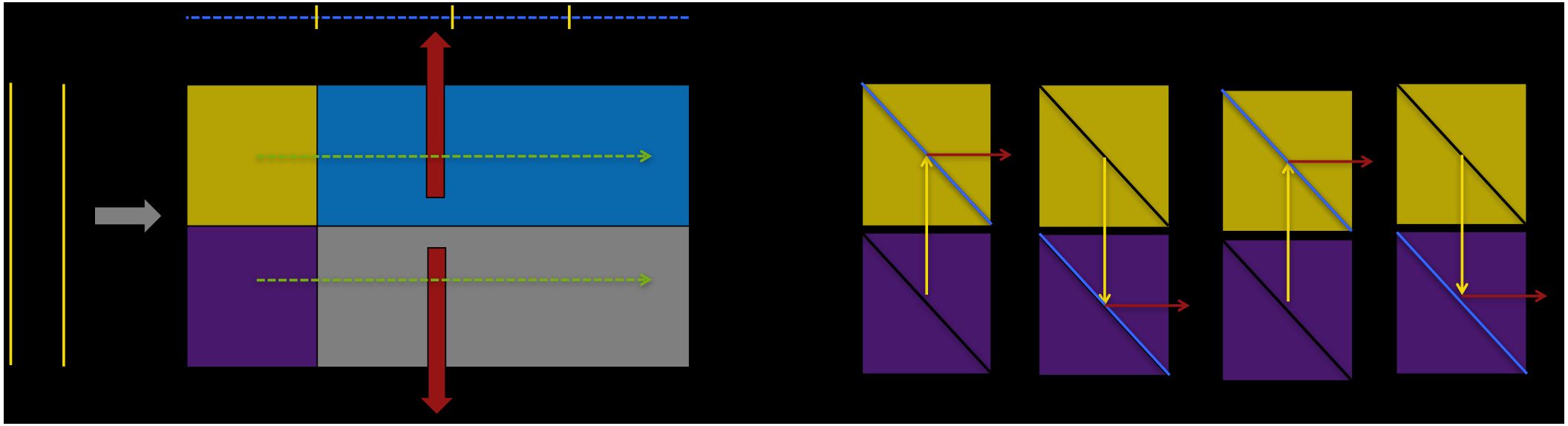
CATERPILLAR Architecture



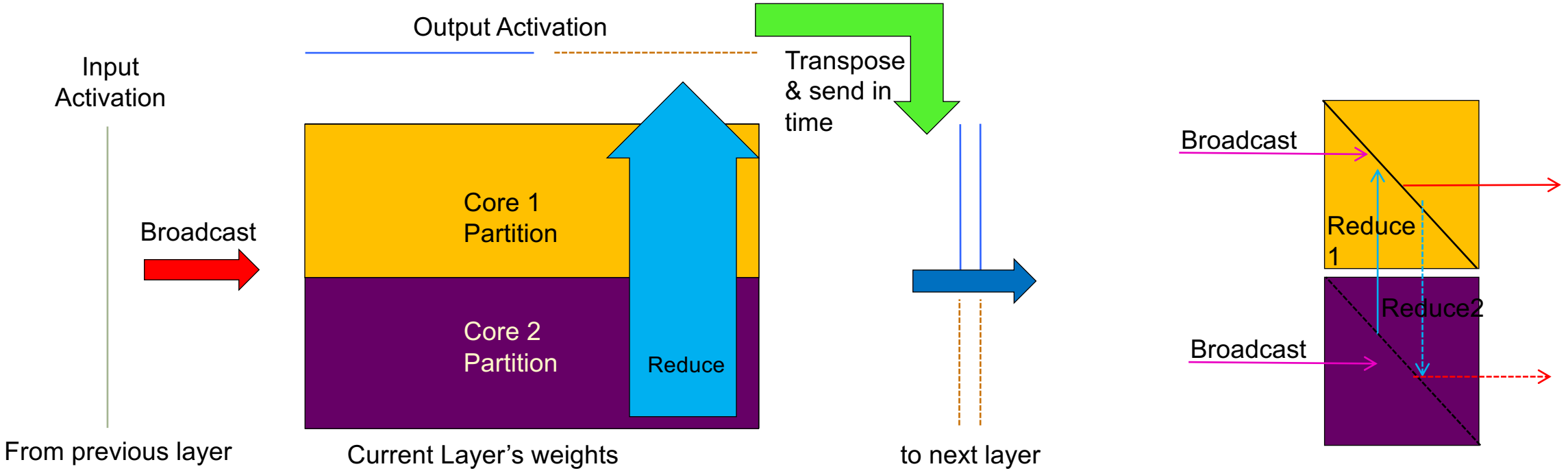


- Multicore
  - Ring of Cores
  - Reconfigurable
- Support for Collective Comms
  - All gather
  - Reduce
  - All Reduce
  - Systolic/Parallel

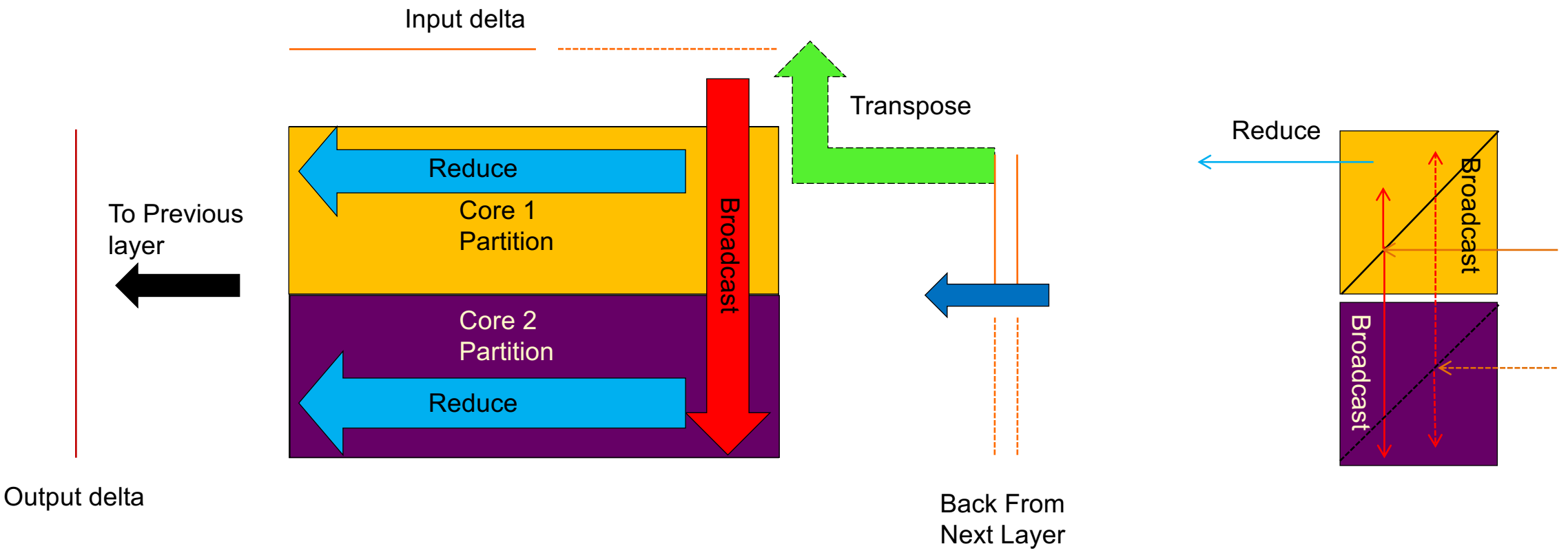
GEMV



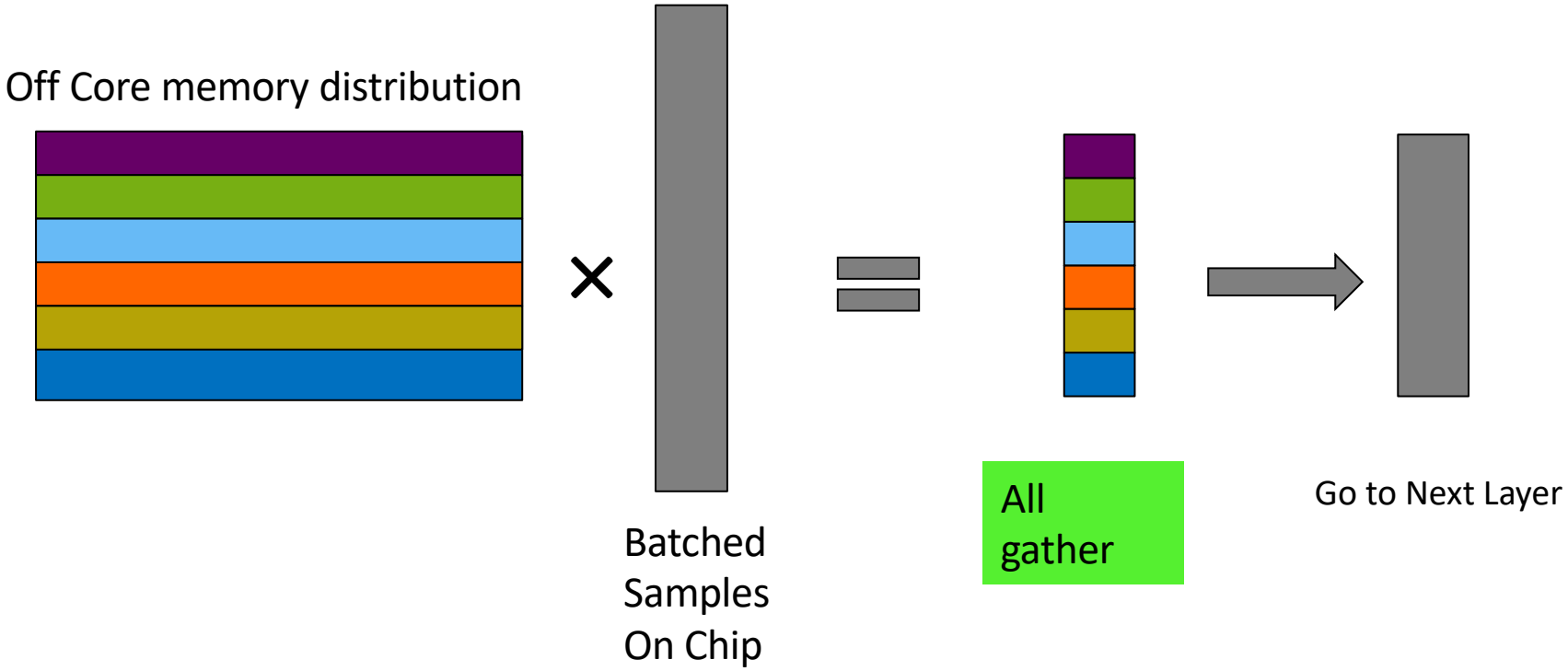
Forward Path



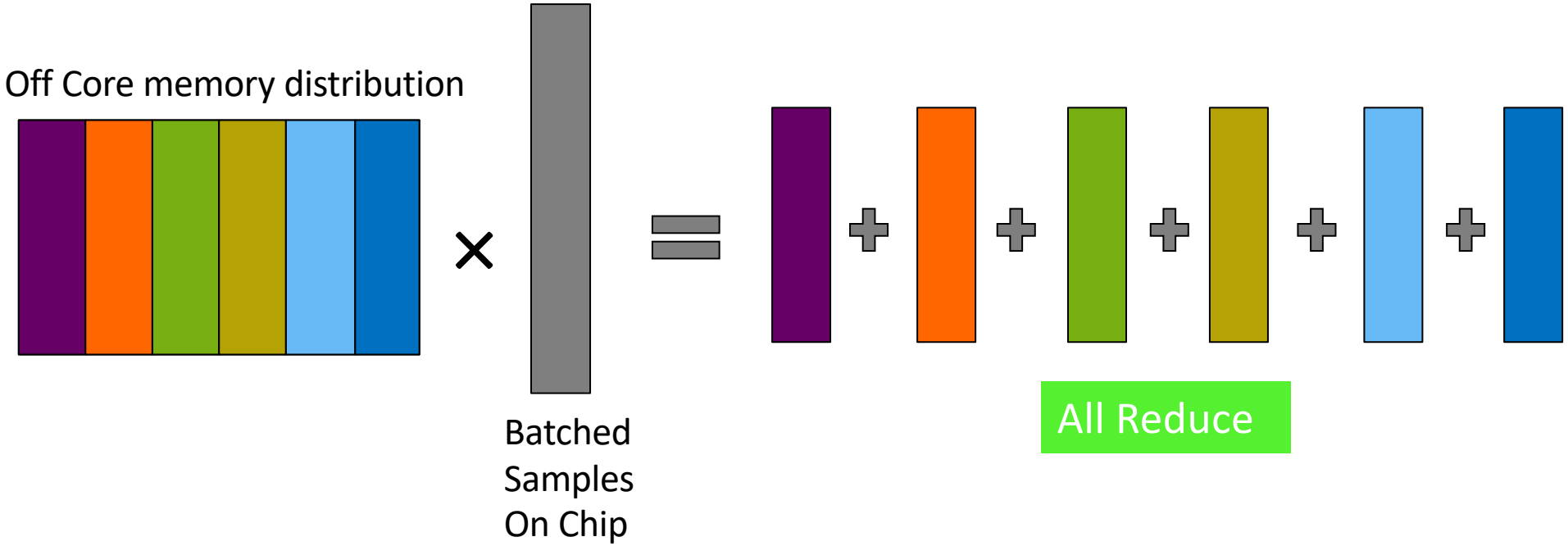
Delta Path


Input delta
Transpose
Reduce
To Previous layer
Core 1 Partition
Broadcast
Core 2 Partition
Reduce
Output delta
Back From Next Layer
Reduce
Broadcast
Broadcast

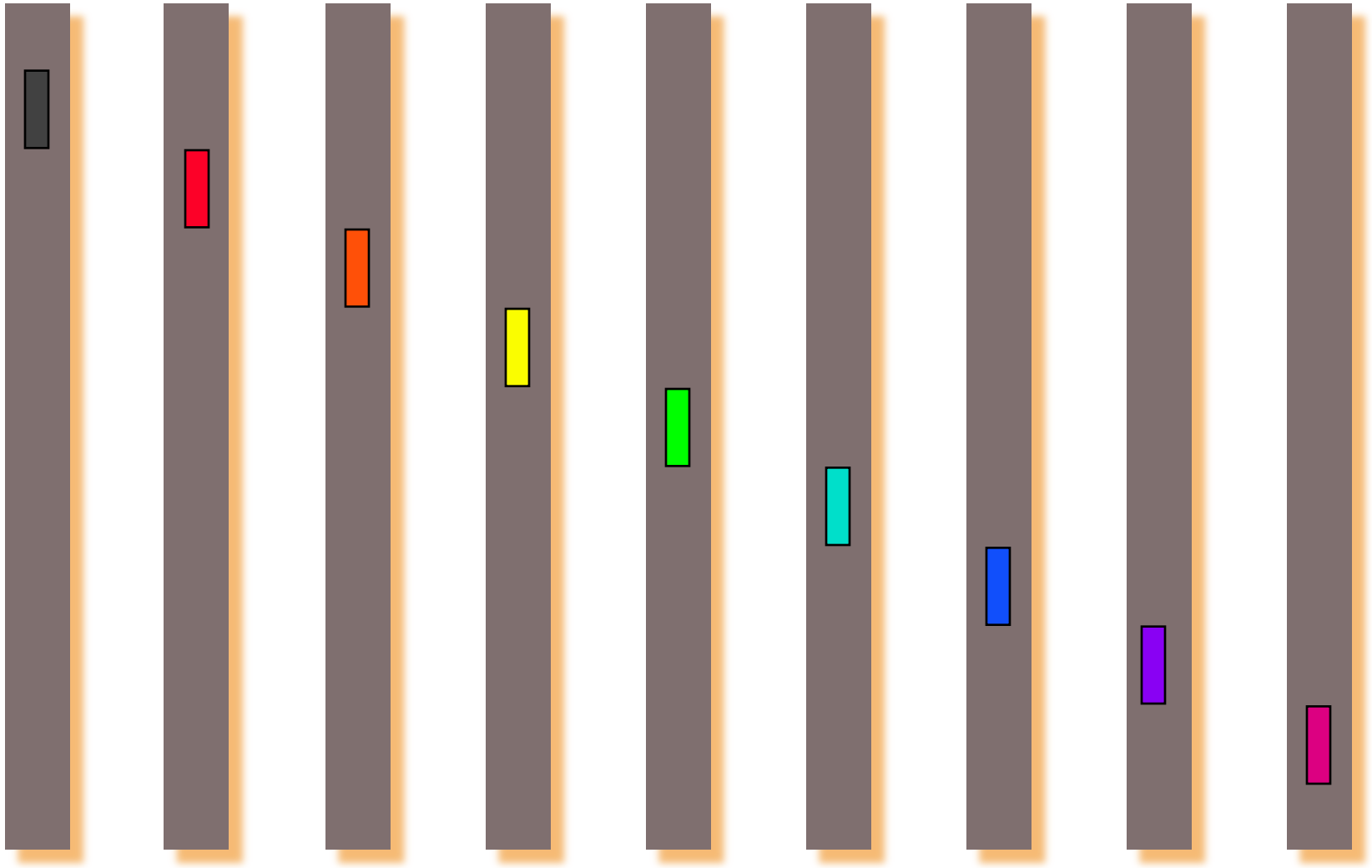
Multicore GEMM



Multicore GEMM

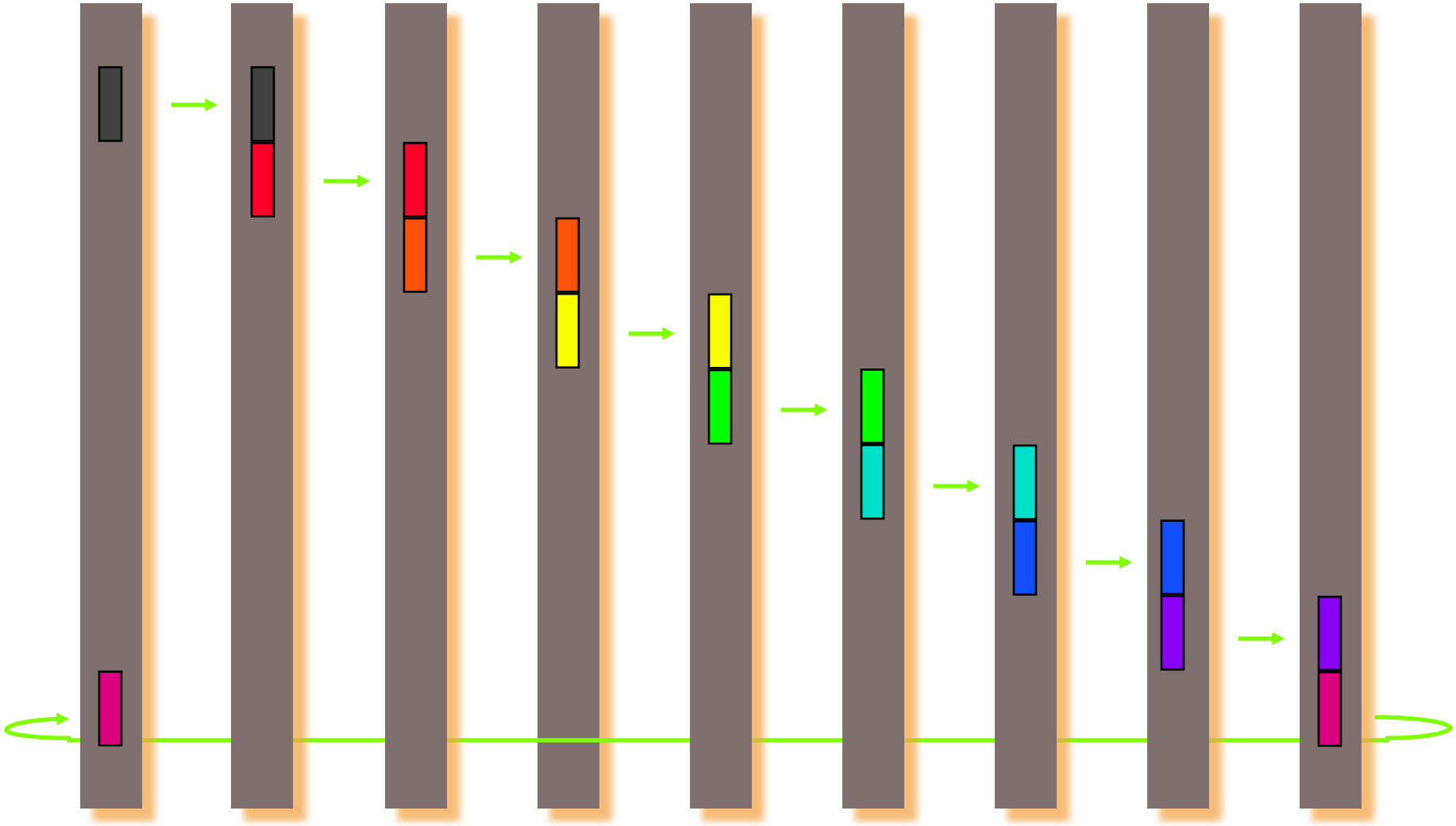


The Bucket Algorithm

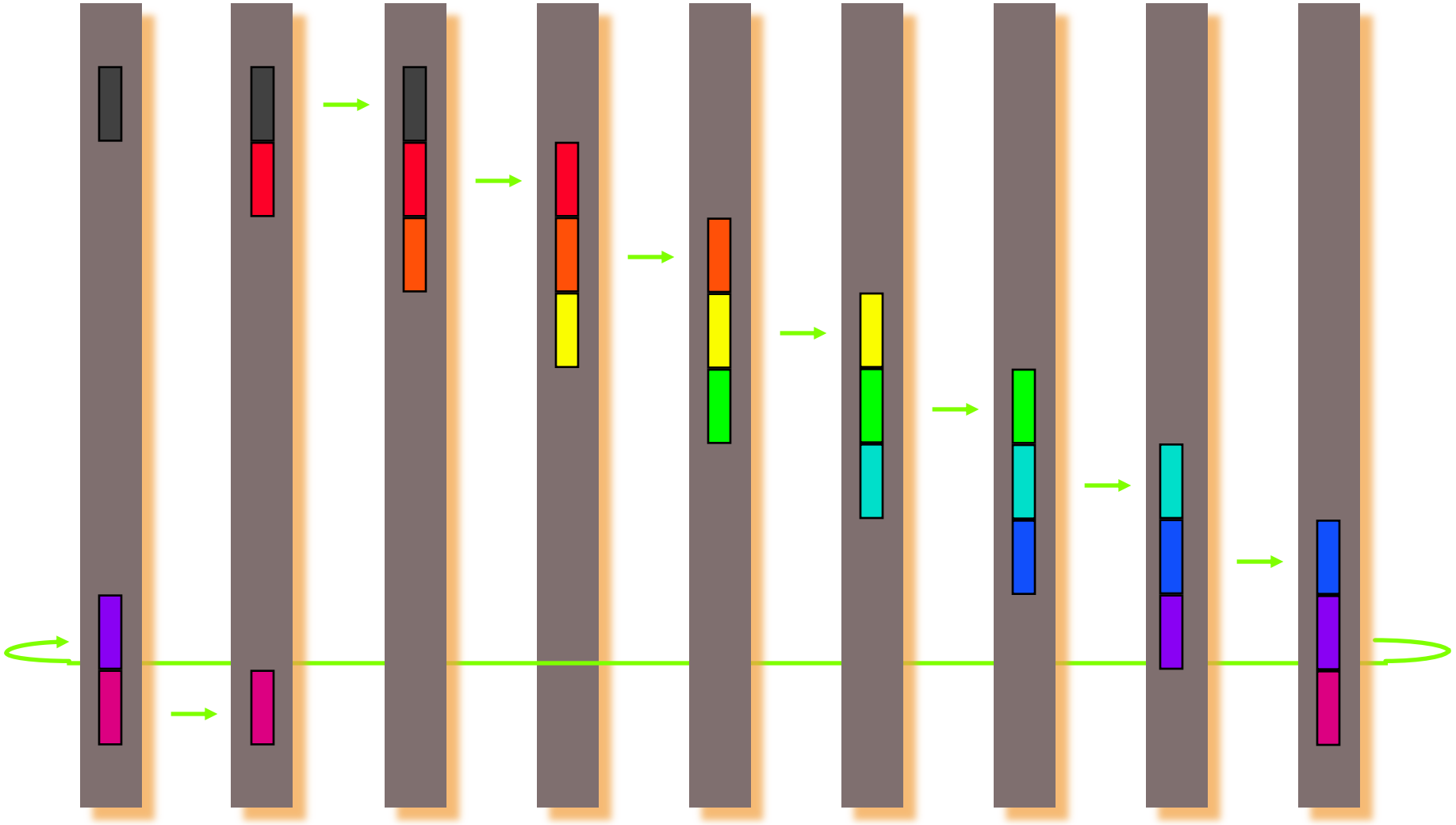


Source: Robert van de Geijn

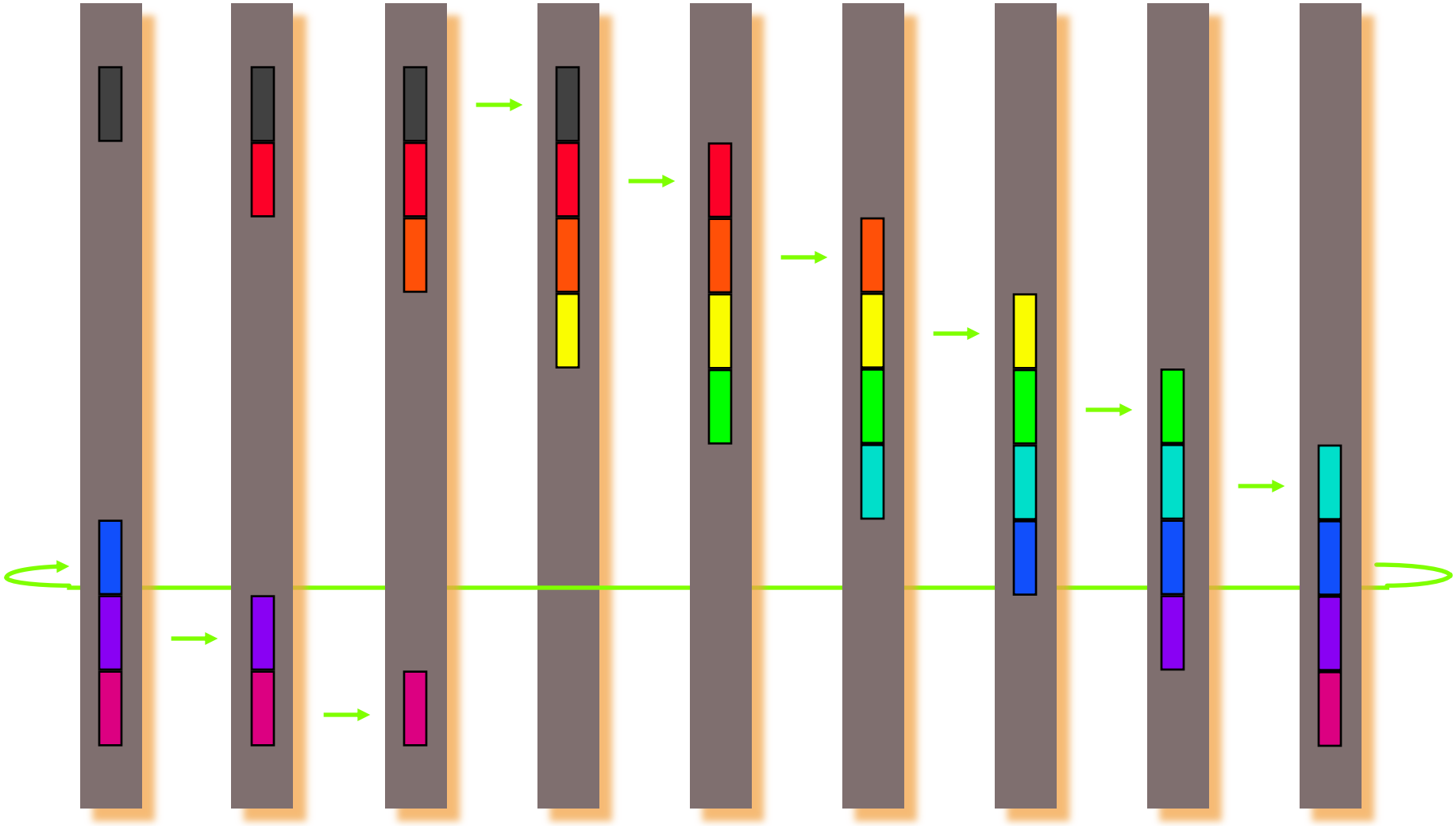
The Bucket Algorithm



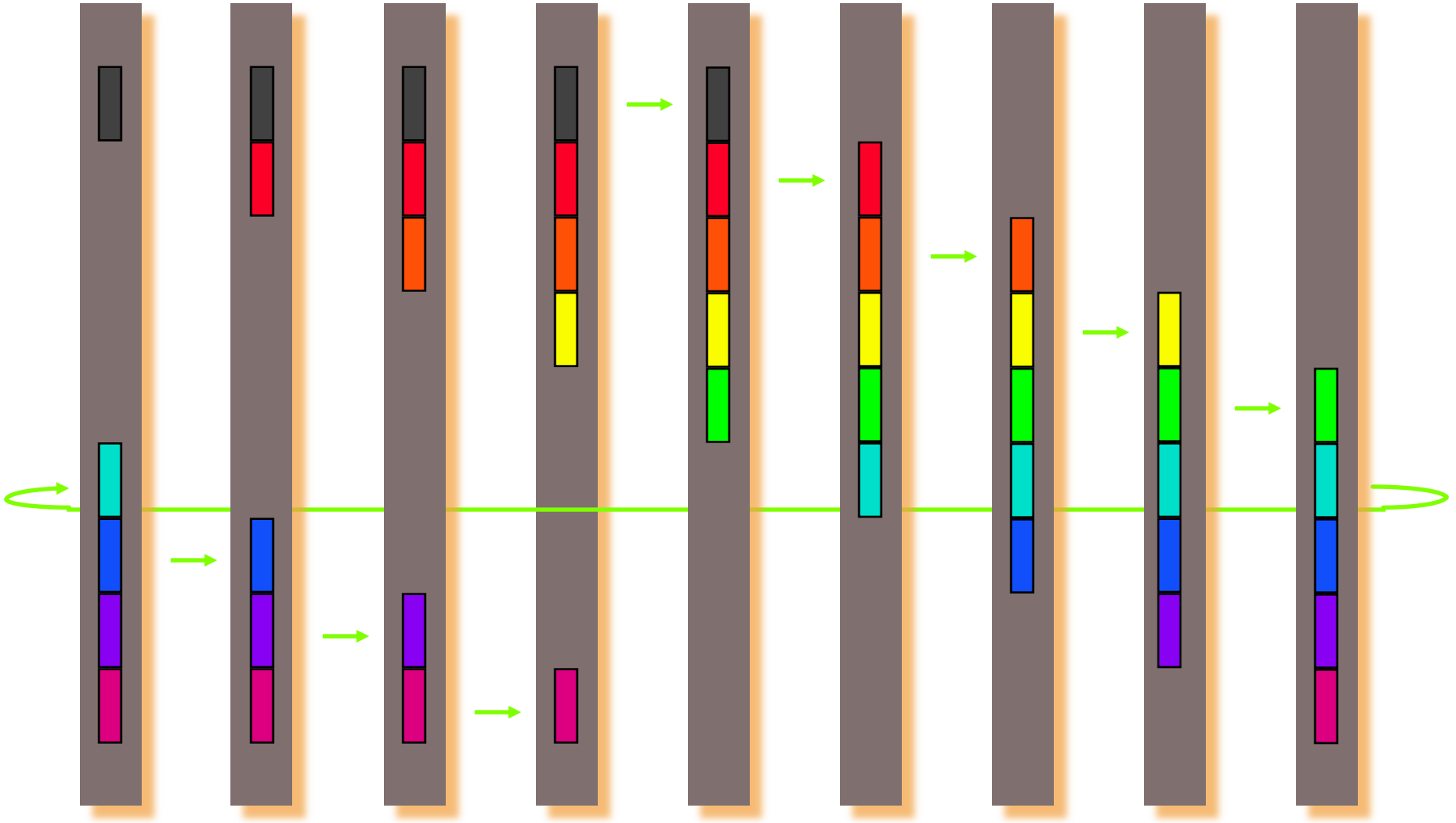
The Bucket Algorithm



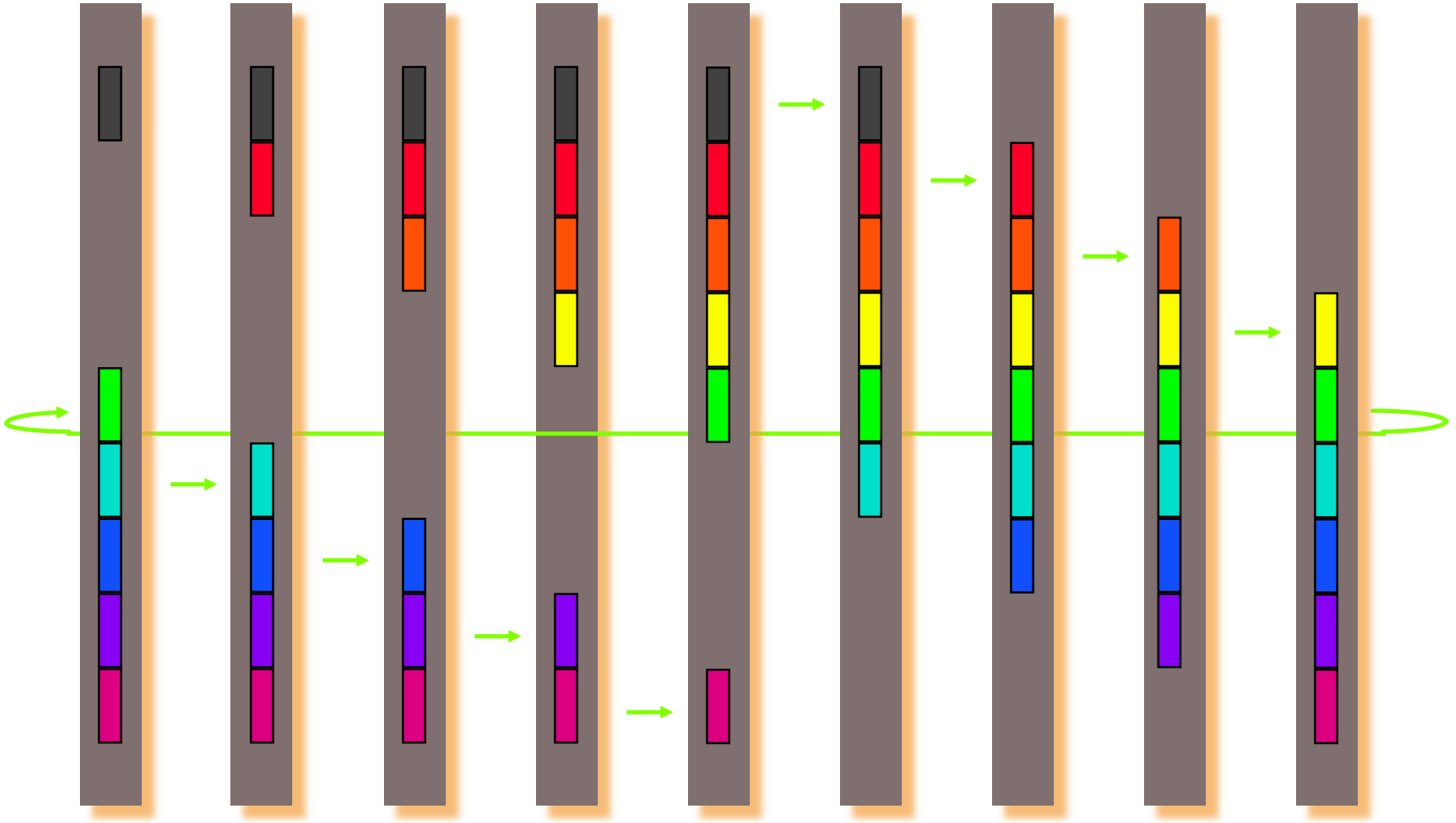
The Bucket Algorithm



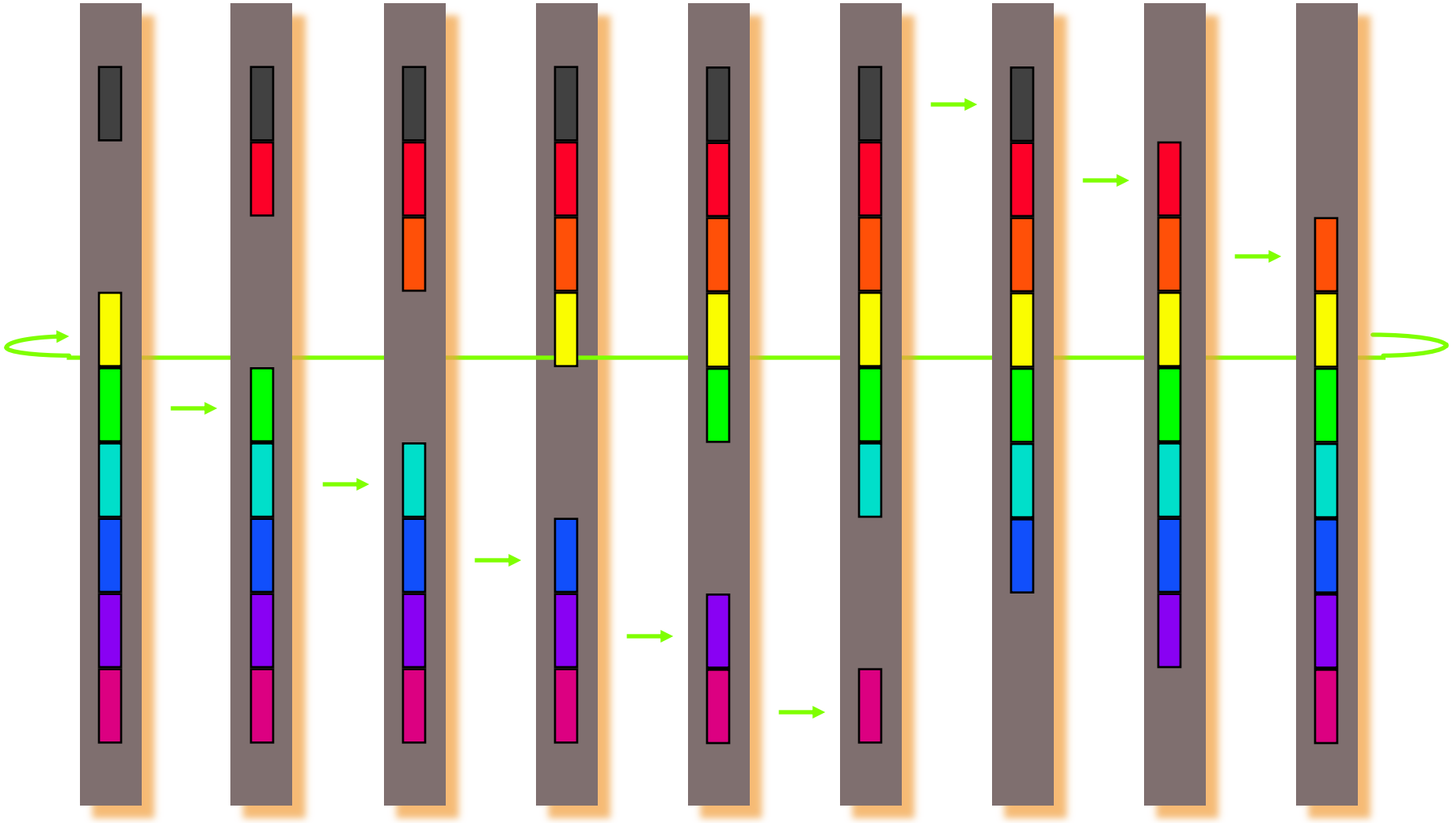
The Bucket Algorithm



The Bucket Algorithm

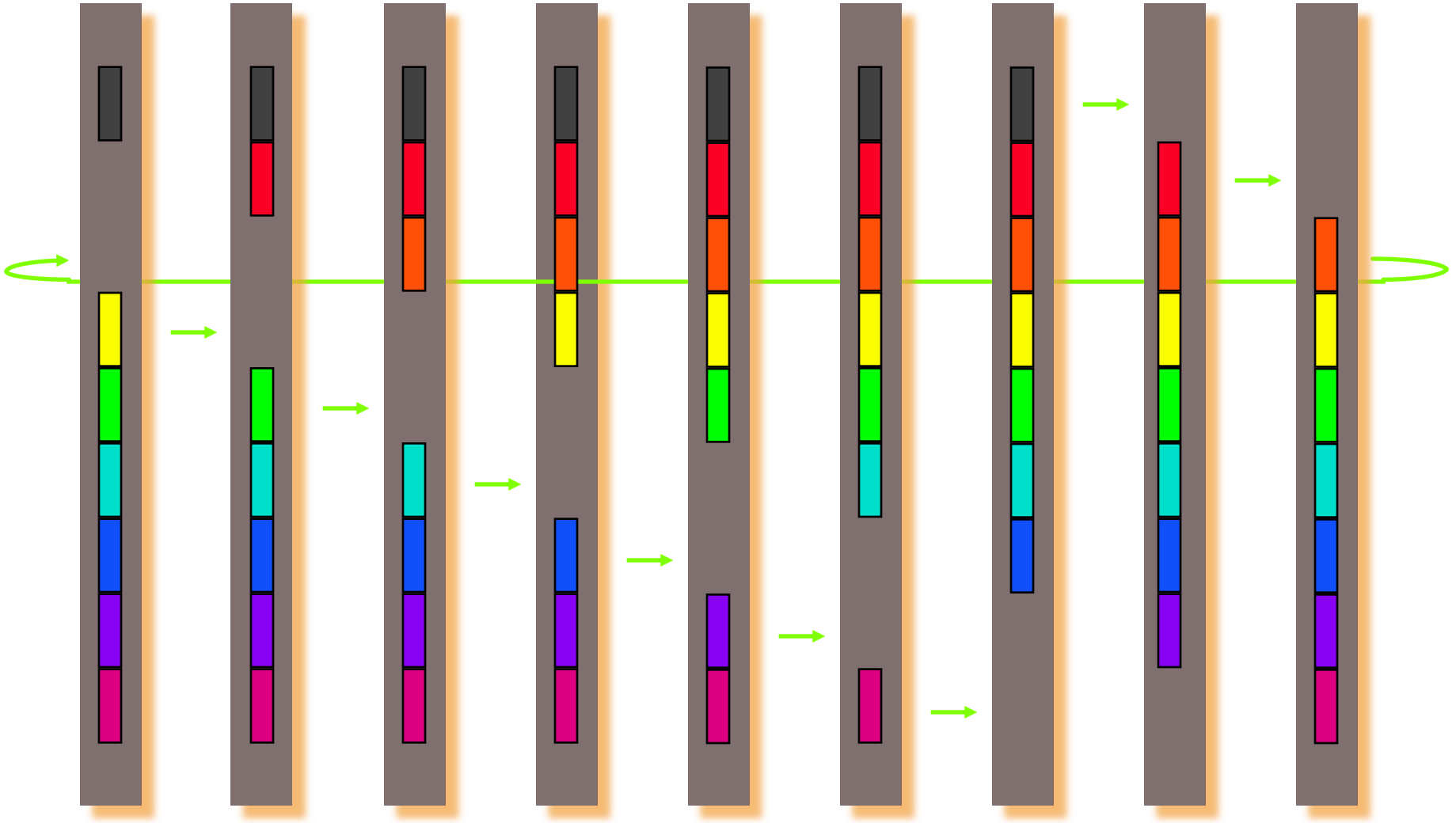


The Bucket Algorithm



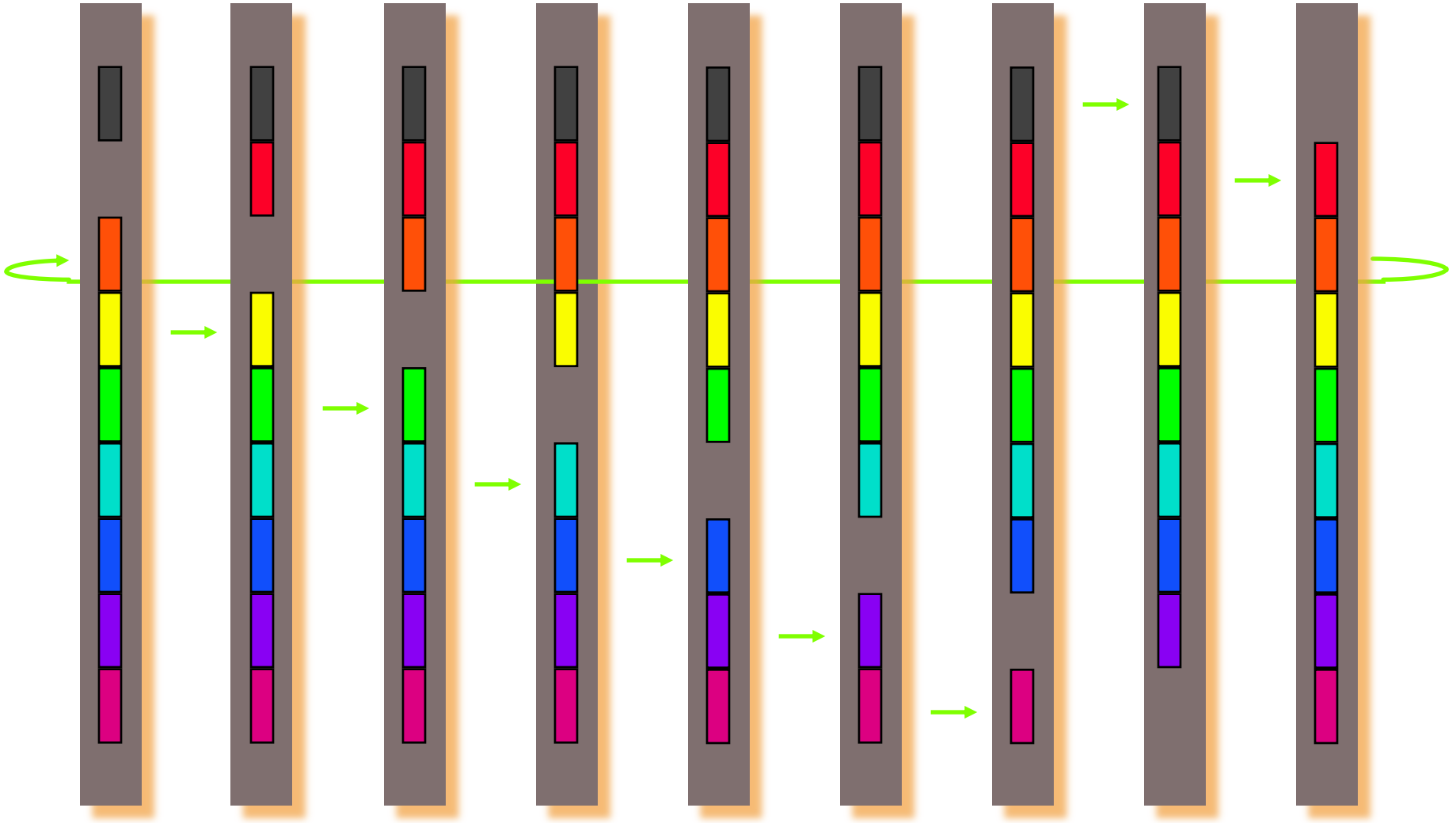
Source: Robert van de Geijn

The Bucket Algorithm

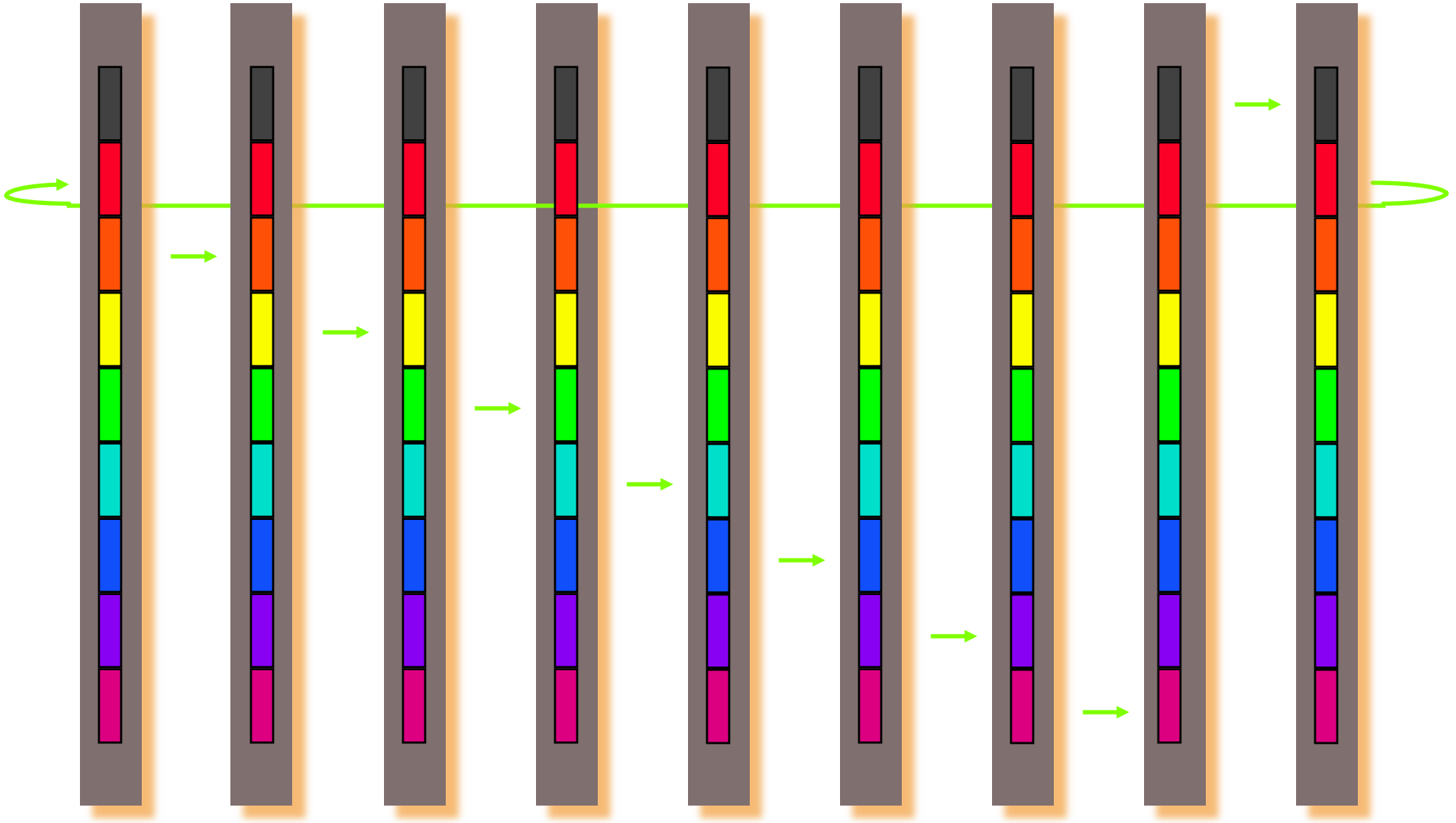


Source: Robert van de Geijn

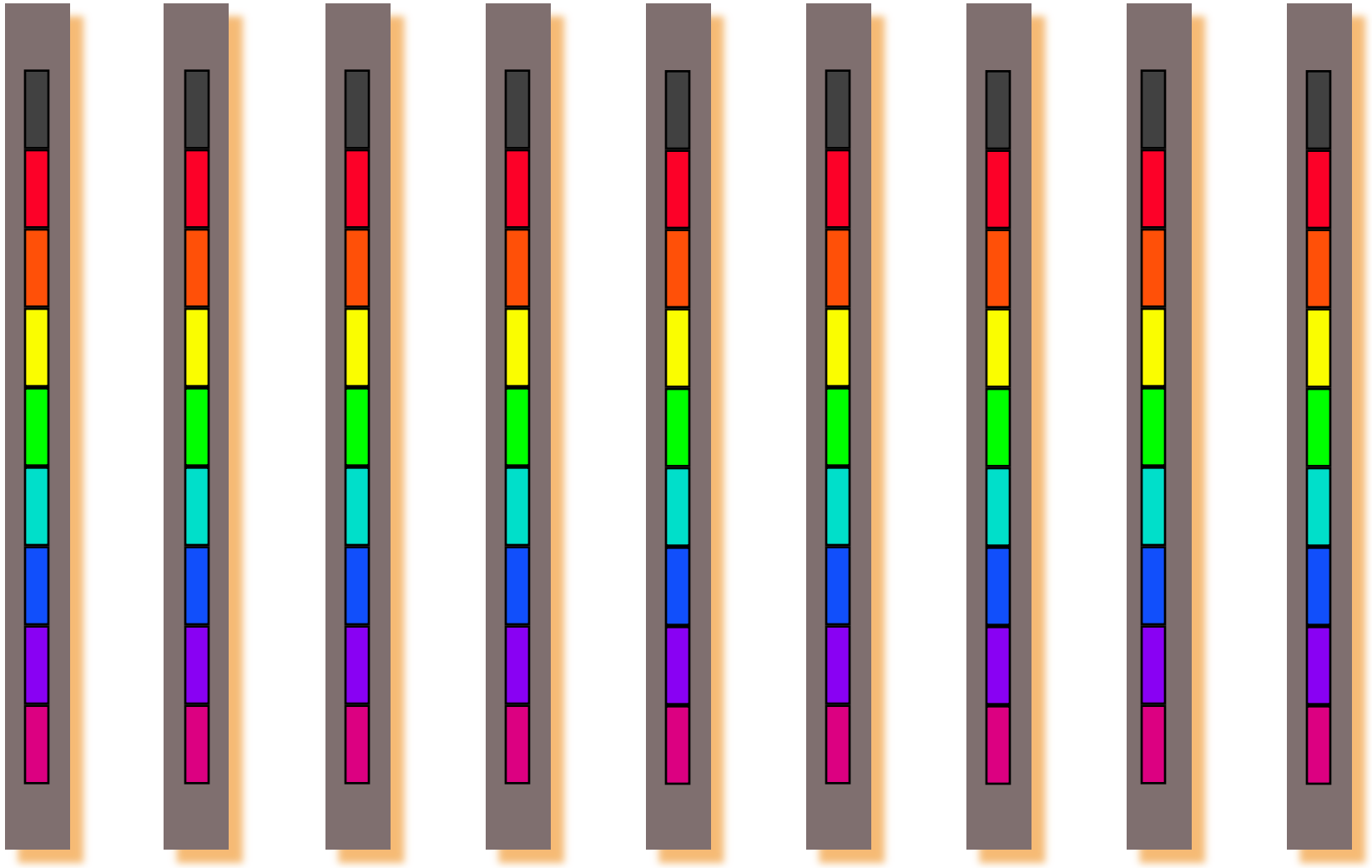
The Bucket Algorithm



The Bucket Algorithm



The Bucket Algorithm



Source: Robert van de Geijn

Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Normalization

- Lowers learning rates to prevent
  - Exploding/Vanishing gradients
- Batch Normalization reparameterizes the network
  - Control the magnitude & mean of the activations
  - **Independent** of all other layers

- Batch Norm
  - Smoother optimization landscape
  - (Bounds the magnitude of the gradients much more tightly)
  - More predictive and stable behavior of the gradients
  - Faster training

http://mlexplained.com/2018/11/30/an-overview-of-normalization-methods-in-deep-learning/

https://arxiv.org/abs/1805.11604

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

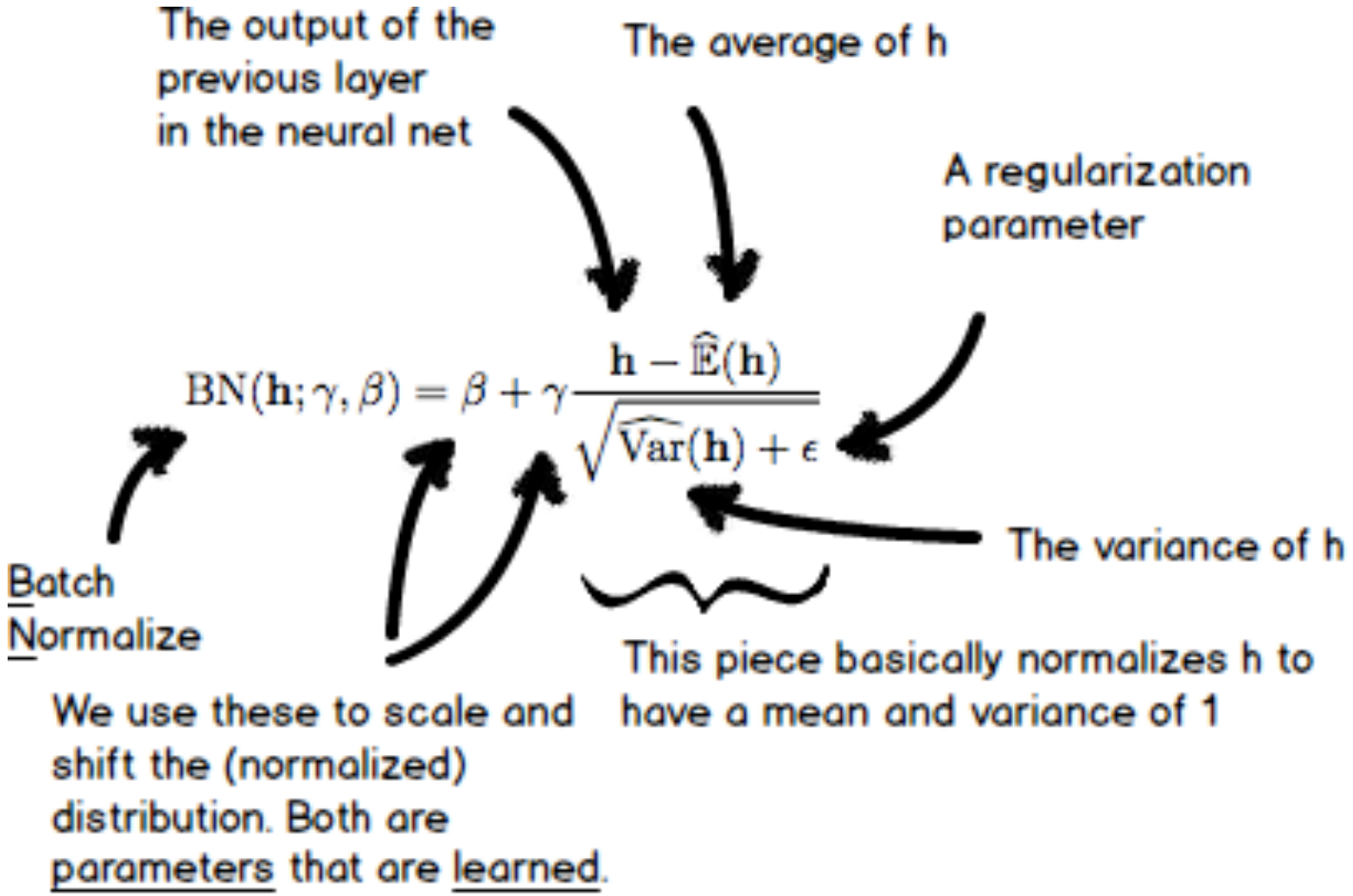
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization



http://sebastiaanboer.com/deep%20learning/2016/03/03/recurrent-batch-normalization.html

Batch Norm

- Bottleneck in Training
- What is the Operation?
- Why is it a bottleneck?
- How can we fix it?

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization is not Perfect

- Large activation memory
- Infrequent weight update
- Violates SGD: true gradient cannot be recovered
- Asymmetry between training & inference
- Not applicable for recurrent networks
  - fit a separate batch normalization layer for each time-step
  - store the statistics for each time-step during training
- Problem with small batch sizes

Source: Ilya Sharapov

Memory Requirements of BN is Huge

Table 1: Memory for training (GB).

Network	Online Norm	Batch 32	Batch 128
ResNet-50, ImageNet	1	2	4
ResNet-50, PyTorch[a]	2	5	15
U-Net, 150^3 voxels	1	29	115
U-Net, 250^3 voxels	6	195	785
U-Net, 1024^2 pixels	2	31	123
U-Net, 2048^2 pixels	5	137	546

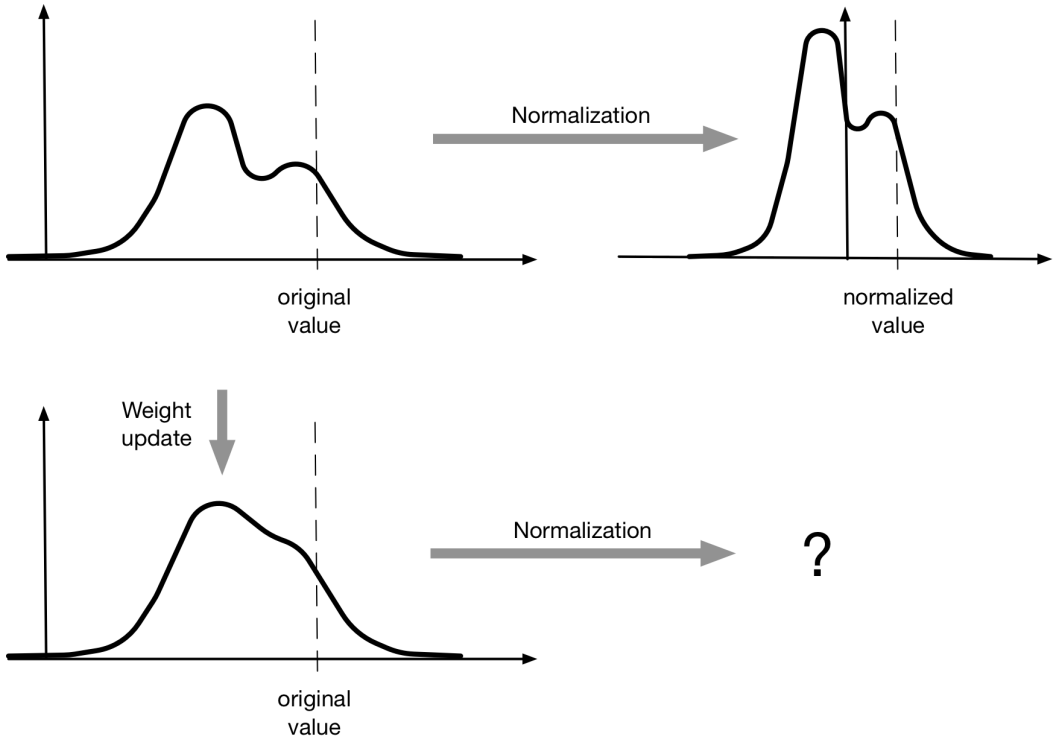
[a] PyTorch stores multiple copies of activations for improved performance.

https://arxiv.org/abs/1905.05894

Normalization: Definition

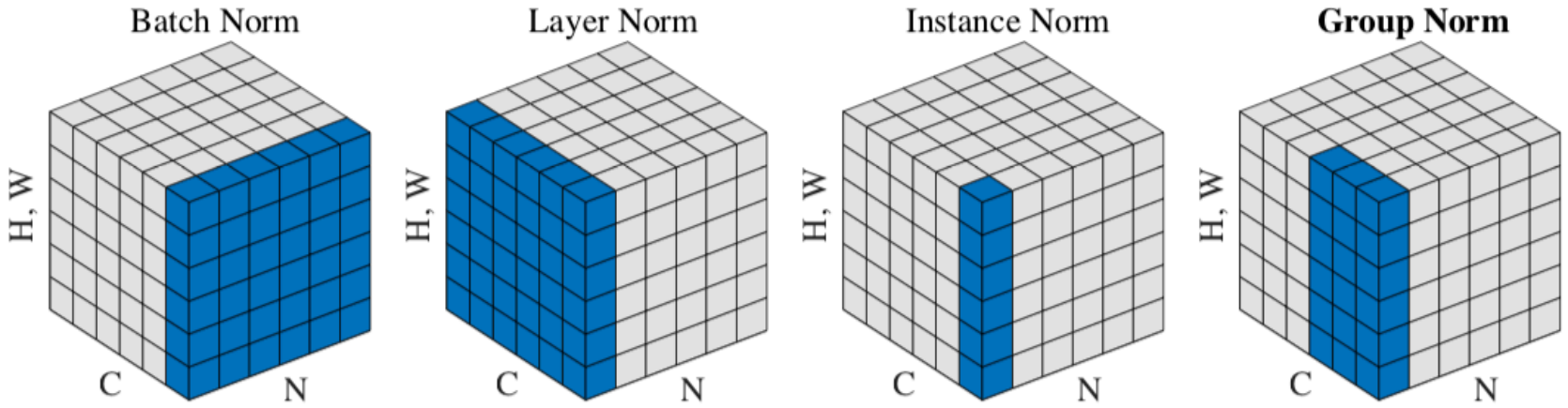
$$f_{\mathbb{X}}[x] \equiv \frac{x - \mu[x]}{\sigma[x]} \qquad x \sim \mathbb{X}$$

- Normalization is a *statistical operator*
- Each output depends on *all* samples
- The distribution changes with every weight update
- Exact computation of “true normalizer” or its gradient is impractical



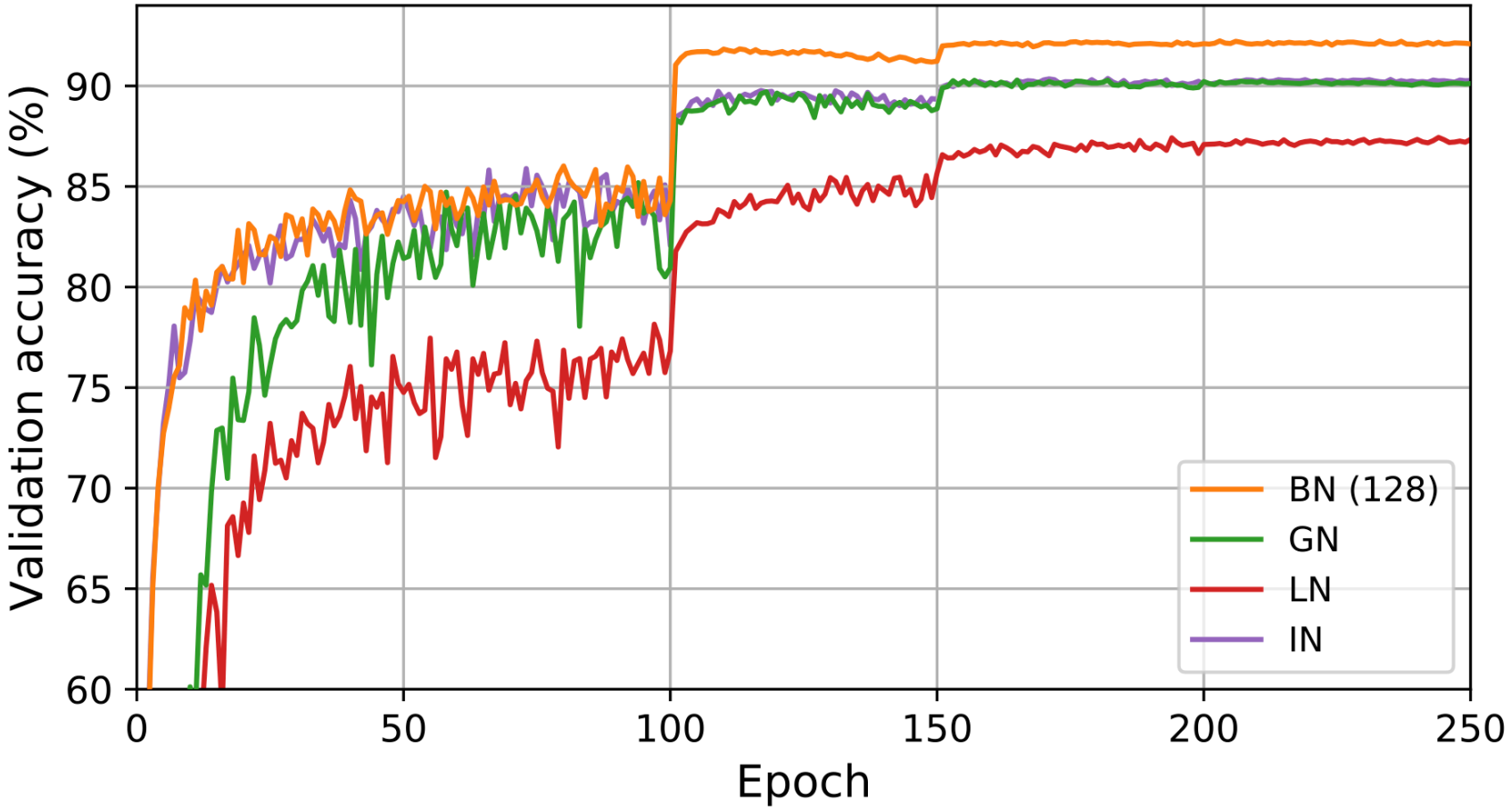
Batch-Free Normalization: Functional Methods

- Layer, Instance, Group Normalization
- Weight Normalization and Normalization Propagation
- Replace the normalization operator with a normalization function
- Fit SGD framework but perform worse than Batch Norm



https://arxiv.org/pdf/1803.08494.pdf

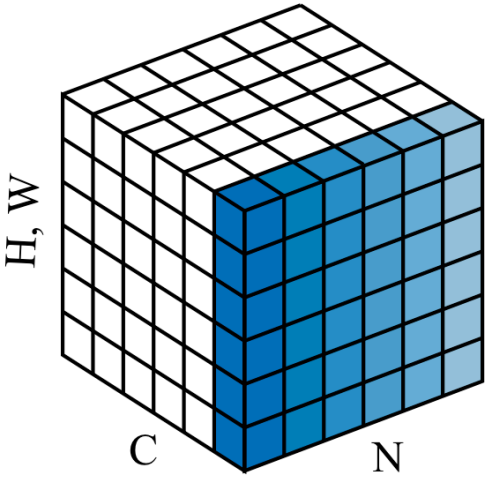
Normalization Matters



https://arxiv.org/abs/1905.05894

Batch-Free Normalization: Heuristic Methods

- Batch Renormalization and Streaming Normalization
- Use past iterations to augment the current forward and backward passes
- Do not differentiate through normalization.
- Use more data to generate better estimates of forward statistics
- However, they lack correctness and stability guarantees.

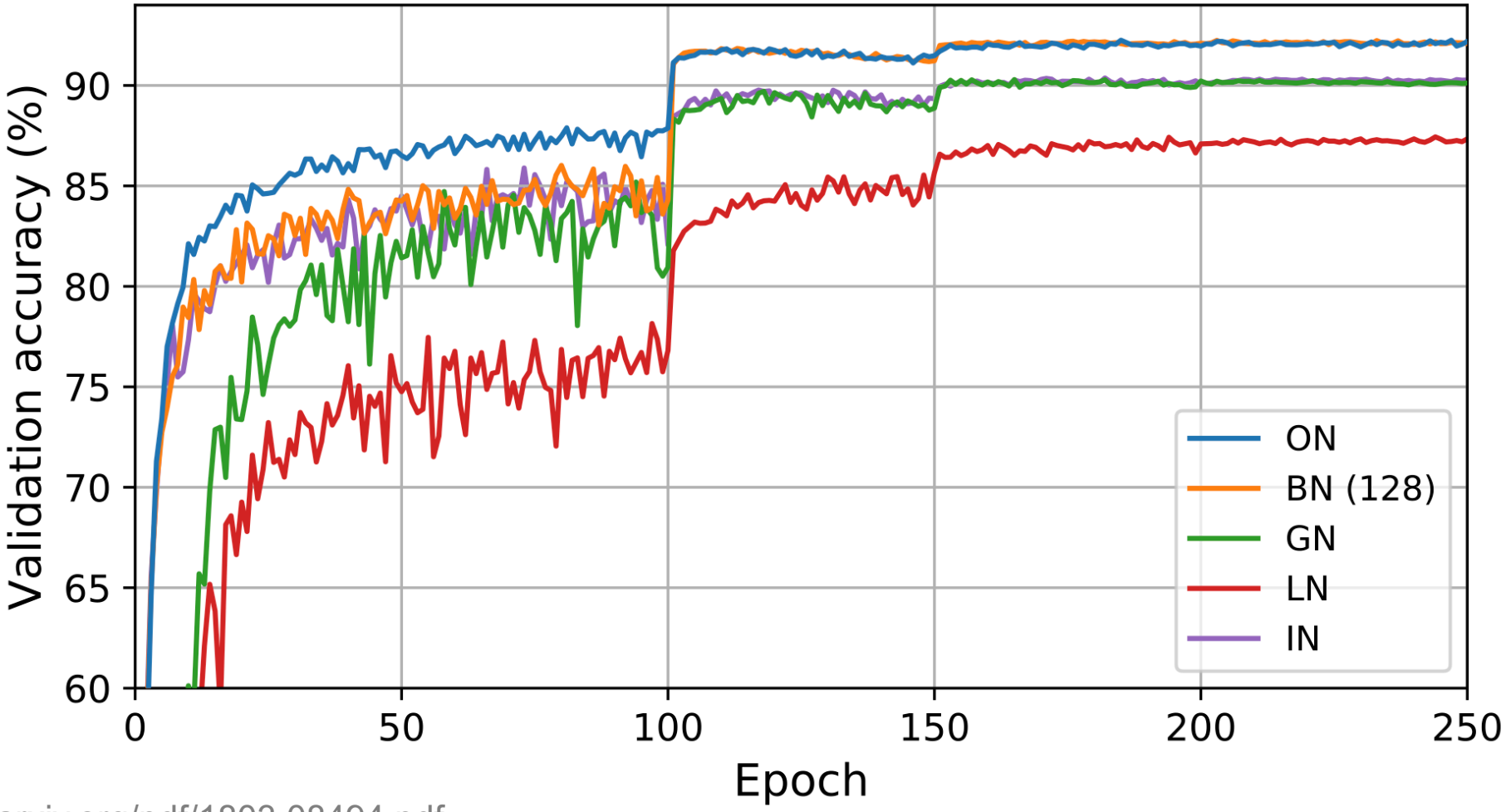


Online Normalization Properties

- **Property 1:** Output generated by the forward pass of Online Normalization is asymptotically mean zero and unit variance.
- **Property 2:** The deviation of the output of Online Normalization from normal distribution is a Lipschitz function with respect to errors in estimates of mean and variance of its input.
- **Property 3:** The backward pass of Online Normalization generates uniformly bounded gradients.
- **Property 4:** Accumulated errors $\varepsilon^{(y)}$ and $\varepsilon^{(1)}$ that track deviations from orthogonality conditions in Online Normalization are bounded.

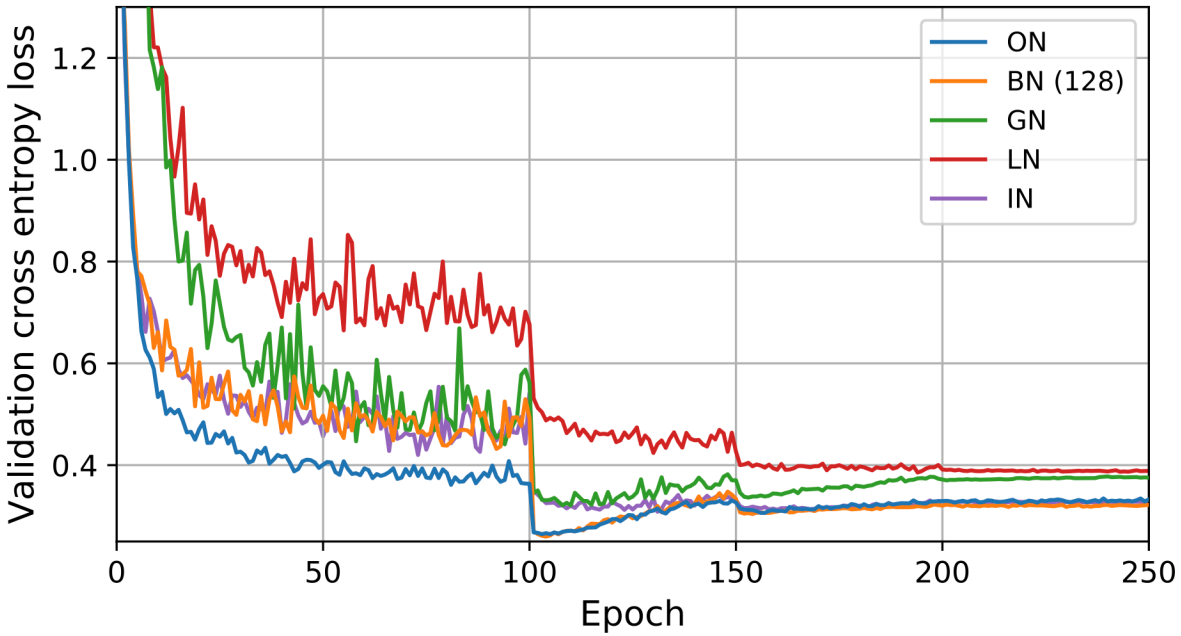
https://arxiv.org/pdf/1803.08494.pdf

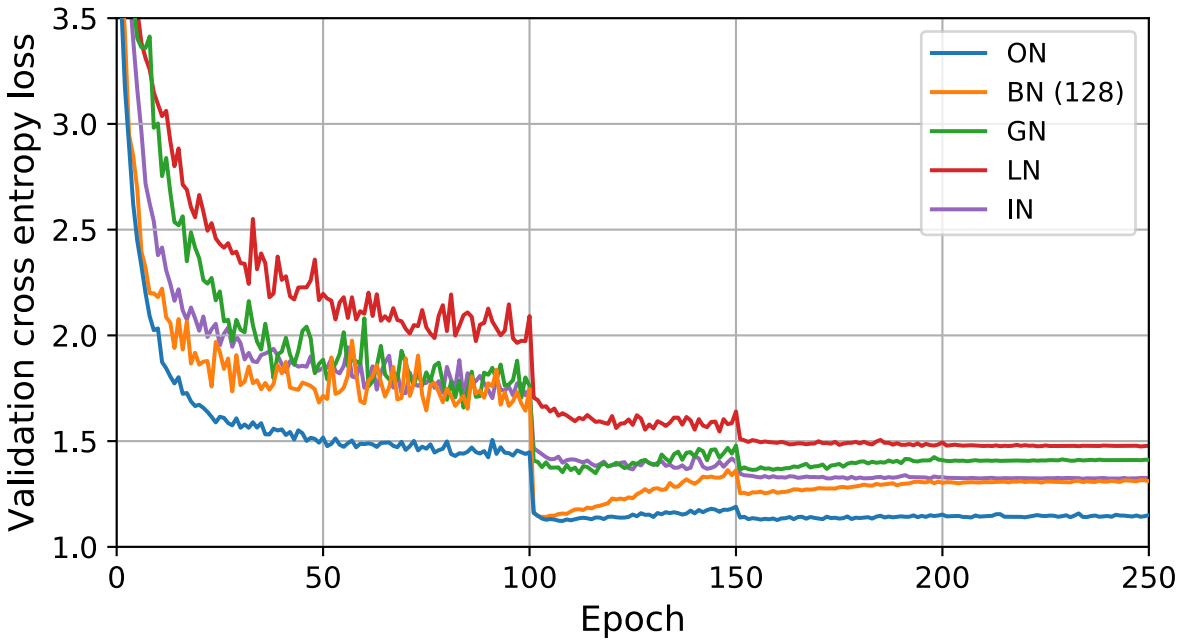
Results – CIFAR10/ResNet20



https://arxiv.org/pdf/1803.08494.pdf

CIFAR / ResNet20 Validation loss





https://arxiv.org/pdf/1803.08494.pdf

ResNet Summary

Table 2: Best validation loss and accuracy.

Normalizer	CIFAR-10 ResNet-20		CIFAR-100 ResNet-20		ImageNet ResNet-50	
Online	**0.26**	(**92.3%**)	**1.12**	(**68.6%**)	**0.94**	(76.3%)
Batch[a]	**0.26**	(92.2%)	1.14	(**68.6%**)	0.97	(**76.4%**)
Group	0.32	(90.3%)	1.35	(63.3%)		(75.9%)[b]
Instance	0.31	(90.4%)	1.32	(63.1%)		(71.6%)[b]
Layer	0.39	(87.4%)	1.47	(59.2%)		(74.7%)[b]
Weight		-		-		(67 %)[b]
Propagation		-		-		(71.9%)[b]

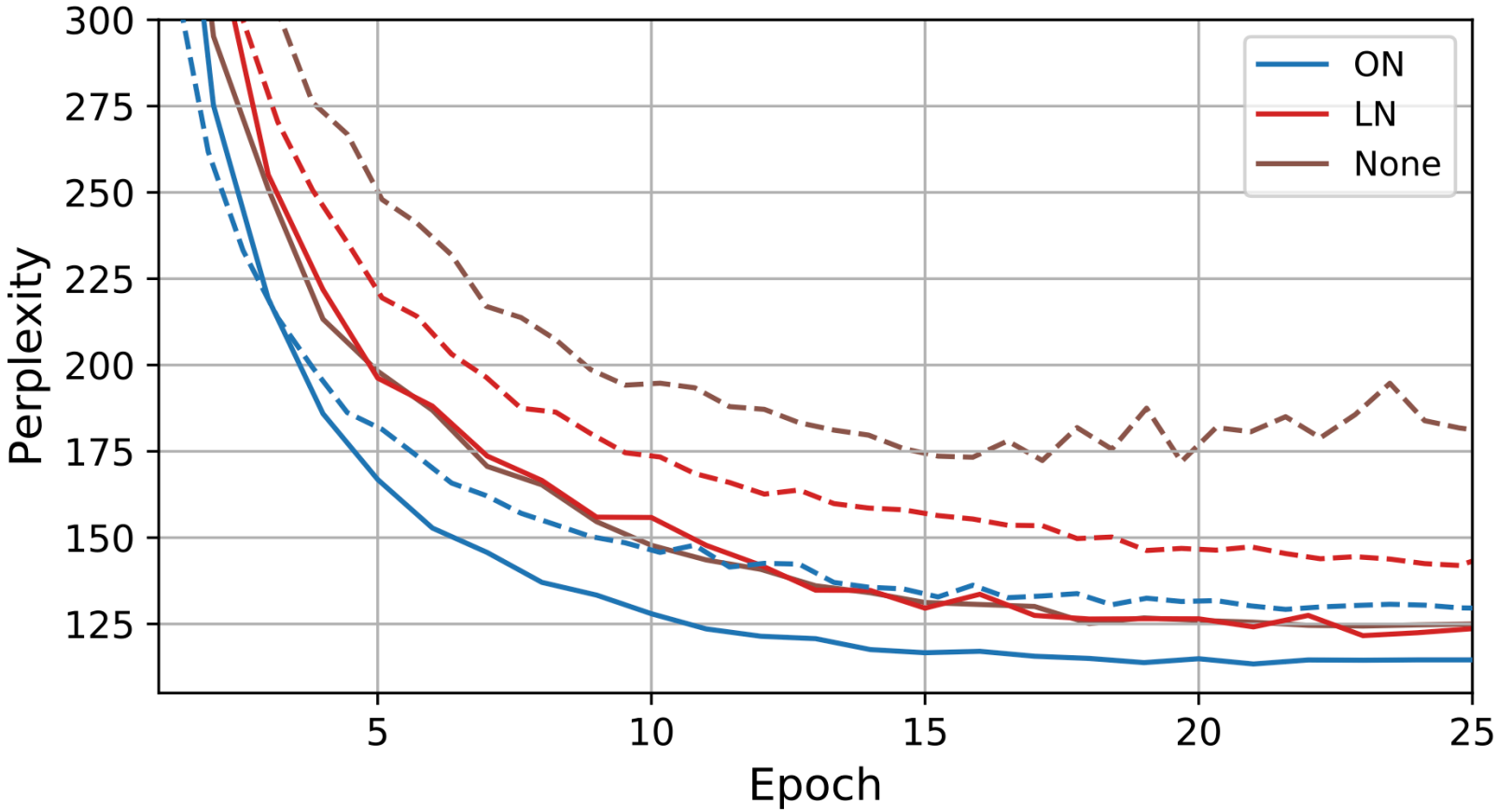
[a] Batch size 128 for CIFAR and 32 for ImageNet.

[b] Data from [7, 22, 23].

https://arxiv.org/pdf/1803.08494.pdf

Recurrent Networks

- RNN (dashed lines) and LSTM (solid lines)



https://arxiv.org/pdf/1803.08494.pdf

Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Low/Mixed Precision Training

- Accelerate Math
  - Multiplication cost is much cheaper
- Reduce Memory Bandwidth
  - FP16 half of FP32
- Reduce Memory Consumption
  - Halve the size of activation and gradient tensors
  - Enables larger minibatches or larger input sizes

Mixed Precision Training vs FP32

ILSVRC12 Classification Networks, Top-1 Accuracy

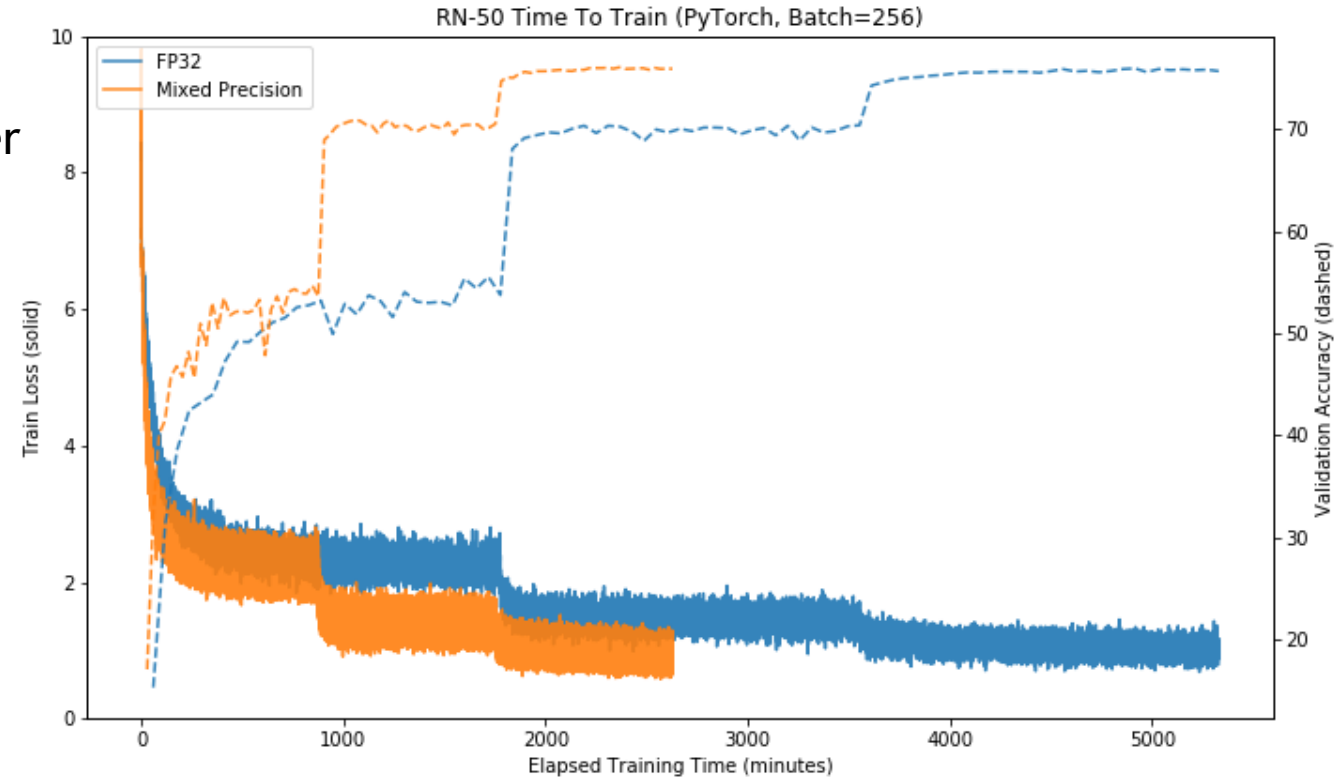
	FP32 Baseline	Mixed Precision
AlexNet	56.8%	56.9%
VGG-D	65.4%	65.4%
GoogLeNet	68.3%	68.4%
Inception v2	70.0%	70.0%
Inception v3	73.9%	74.1%
Resnet 50	75.9%	76.0%
ResNeXt 50	77.3%	77.5%

In all cases hyper-parameters were the same as for FP32 training

Resnet 50 Training

4 Lines of Code => 2.3x Training Speedup

- Single-GPU runs using TorchVision ImageNet example
  - NVIDIA PyTorch 18.08-py3 container
  - **AMP for mixed precision**
- Minibatch=256 per GPU
- Single GPU RN-50 speedup for FP32 -> M.P. (with 2x batch size):
  - MxNet: **2.9x**
  - TensorFlow: **2.2x**
  - TensorFlow + XLA: **~3x**
  - PyTorch: **2.3x**
- Work ongoing to bring to 3x everywhere



Mixed Precision Training Beyond Imagenet

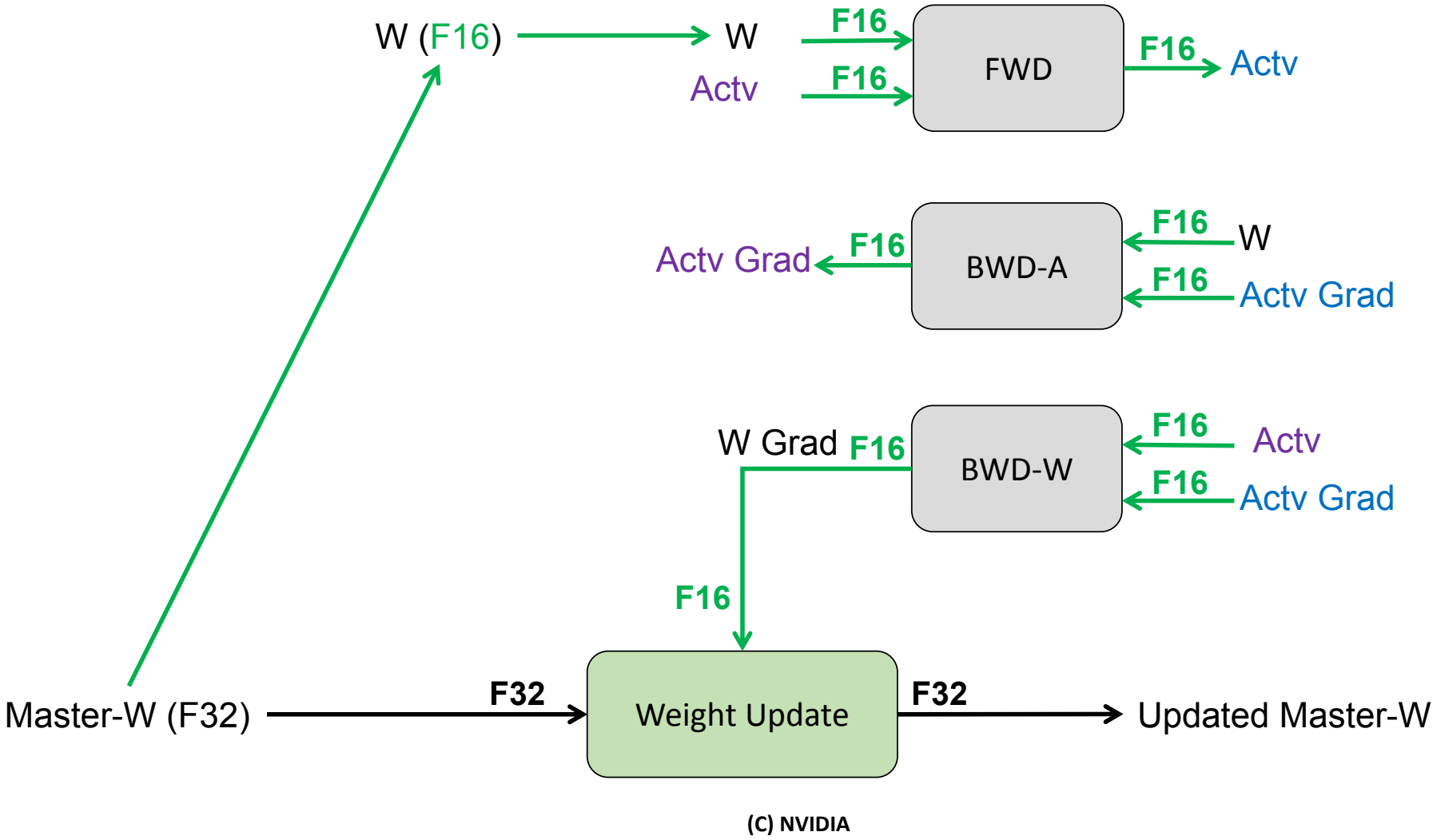
Model	FP32 -> M.P. Speedup	Comments
GNMT (Translation)	2.3x	Iso-batch size
FairSeq Transformer (Translation)	2.9x 4.9x	Iso-batch size 2x lr + larger batch
ConvSeq2Seq (Translation)	2.5x	2x batch size
Deep Speech 2 (Speech recognition)	4.5x	Larger batch
wav2letter (Speech recognition)	3.0x	2x batch size
Nvidia Sentiment (Language modeling)	4.0x	Larger batch

*In all cases trained to same accuracy as FP32 model

Weight Update

- **FP16 mantissa is sufficient for some networks, some require FP32**
- **Sum of FP16 values whose ratio is greater than 2^{11} is just the larger value**
  - FP16 has a 10-bit mantissa, binary points have to be aligned for addition
  - Weight update: if ***w >> lr * dw*** then update doesn't change *w*
    - Examples: multiplying a value by 0.01 leads to ~2^7 ratio, 0.001 leads to ~2^{10} ratio
- **Conservative recommendation:**
  - FP32 update:
    - Compute weight update in FP32
    - Keep a master copy of weights in FP32, make an FP16 copy for fwd/bwd passes
- **If FP32 storage is a burden, try FP16 – it does work for some nets**
  - ie convnets

Mixed Precision Training for Nvidia GPUs



Pointwise Operation

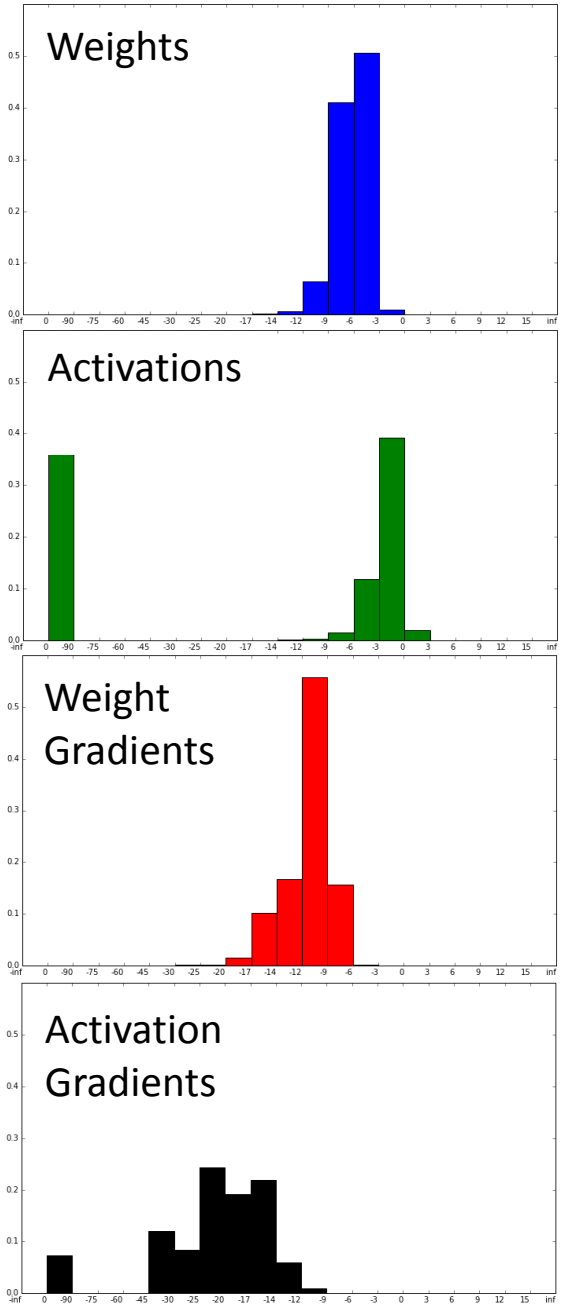
- **FP16 is safe for most of these: ReLU, Sigmoid, Tanh, Scale, Add, …**
  - Inputs and outputs to these are value in a narrow range around 0
  - FP16 storage saves bandwidth -> reduces time
- **FP32 math and storage is recommended for:**
  - operations f where $|f(x)| >> |x|$
    - Examples: Exp, Square, Log, Cross-entropy
  - These typically occur as part of a normalization or loss layer that is unfused
  - FP32 ensures high precision, no perf impact since bandwidth limited
- **Conservative recommendation :**
  - Leave pointwise ops in FP32 (math and storage) unless they are known types
  - Pointwise op fusion is a good next step for performance
    - Use libraries for efficient fused pointwise ops for common layers (eg BatcNorm)

Reductions

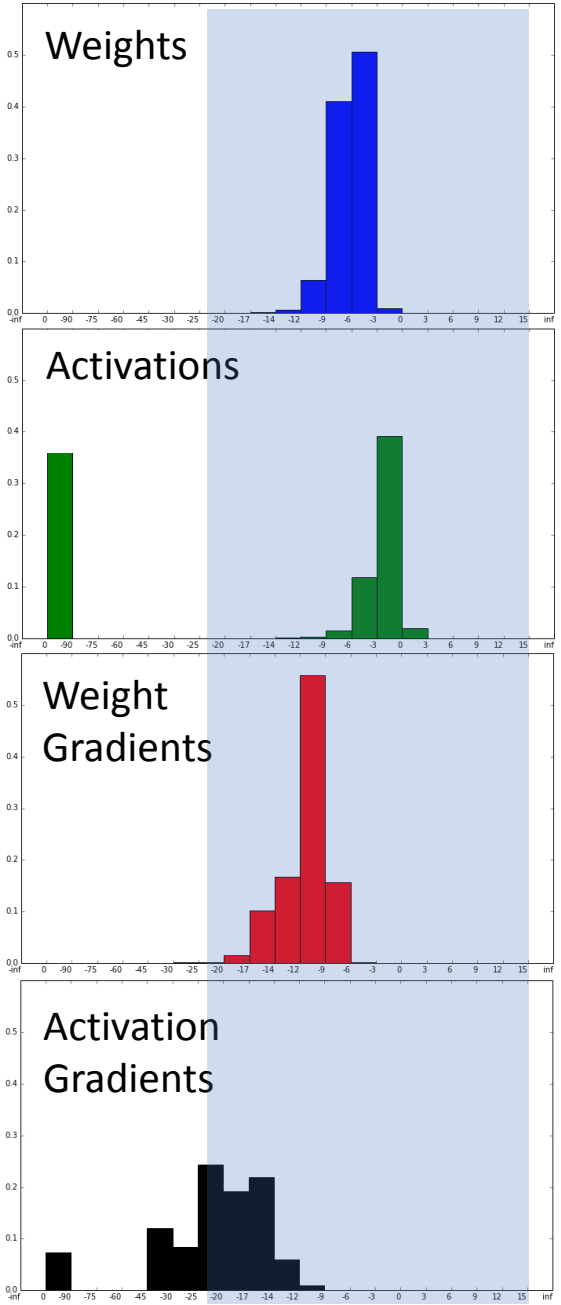
- **Examples:**
  - Large sums of values: L1 norm, L2 norm, Softmax
- **FP32 Math:**
  - Avoids overflows
  - Does not affect speed – these operations are memory limited
- **Storage:**
  - FP32 output
  - Input can be FP16 if the preceding operation outputs FP16
    - If your training frameworks supports different input and output types for an op
    - Saves bandwidth -> speedup

A Note on Normalization and Loss Layers

- **Normalizations:**
  - Usually constructed from primitive ops (reductions, squares, exp, scale)
  - Storage:
    - Input and normalized output can be in FP16
    - Intermediate results should be stored in FP32
  - Ideally should be fused into a single op:
    - Avoids round-trips to memory -> faster
    - Avoids intermediate storage
- **Loss, probability layers:**
  - Softmax, cross-entropy, attention modules
  - FP32 math, FP32 output

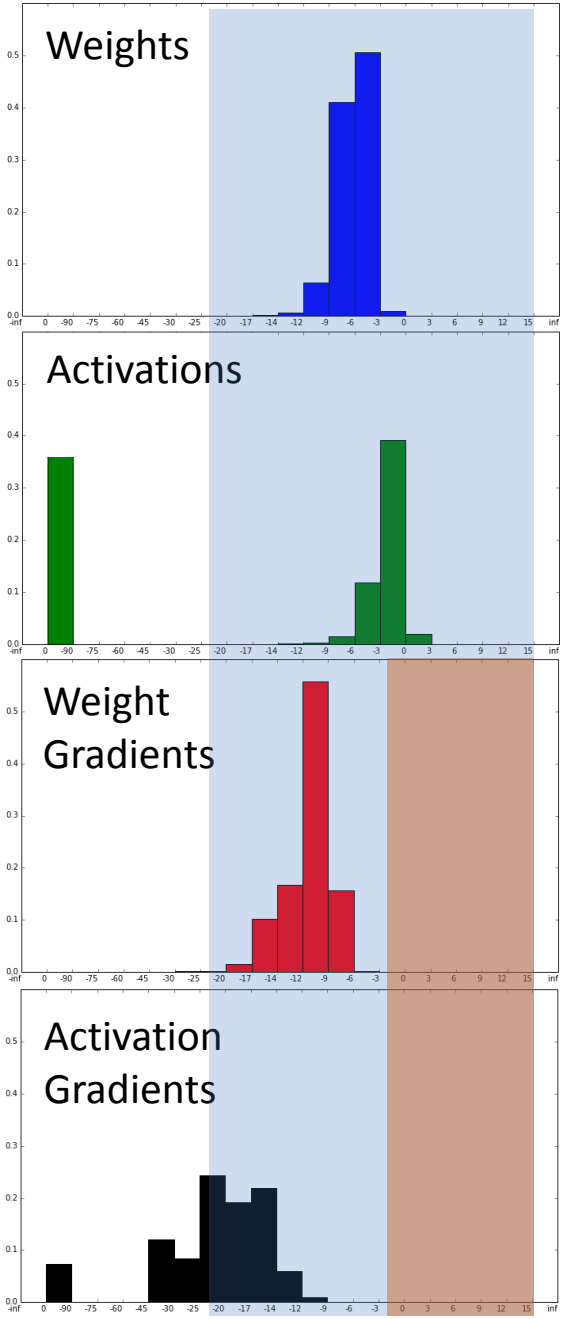

Weights
Activations
Weight Gradients
Activation Gradients

(C) NVIDIA



Range representable in FP16: ~40 powers of 2

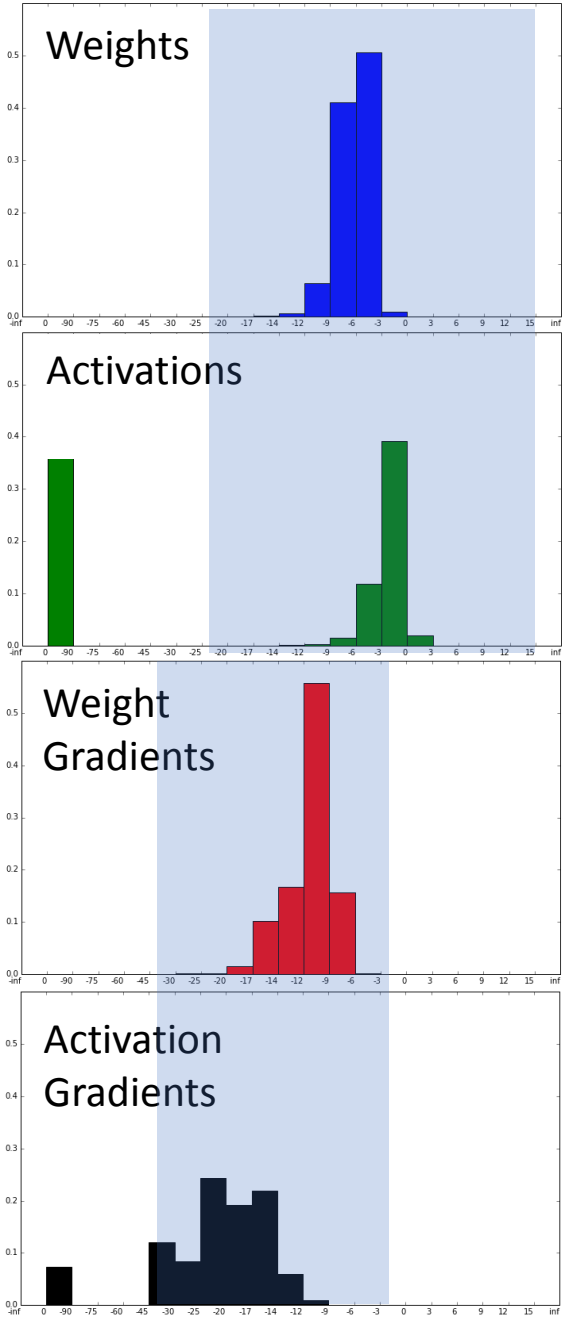
(C) NVIDIA



Range representable in FP16: ~40 powers of 2

Gradients are small, don't use much of FP16 range

FP16 range not used by gradients: ~15 powers of 2



Range representable in FP16: ~40 powers of 2

Gradients are small, don't use much of FP16 range

FP16 range not used by gradients: ~15 powers of 2

Loss Scaling:

multiply the loss by some constant ***s***

by chain rule backprop scales gradients by ***s***

preserves small gradient values

unscale the weight gradient before update

Loss Scaling

- **Algorithm**
  - Pick a scaling factor ***s***
  - for each training iteration
    - Make an fp16 copy of weights
    - Fwd prop (fp16 weights and activations)
    - Scale the loss by ***s***
    - Bwd prop (fp16 weights, activations, and gradients)
    - Scale *dW* by **1/*s***
    - Update *W*
- **For simplicity:**
  - Apply gradient clipping and similar operations on gradients after 1/s scaling
    - Avoids the need to change hyperparameters to account for scaling
- **For maximum performance: fuse unscaling and update**
  - Reduces memory accesses
  - Avoids storing weight gradients in fp32

Automatic Loss Scaling

- **Frees users from choosing a scaling factor**
  - Too small a factor doesn't retain enough small values
  - Too large a factor causes overflows
- **Algorithm**
  - Start with a large scaling factor ***s***
  - for each training iteration
    - Make an fp16 copy of weights
    - Fwd prop
    - Scale the loss by ***s***
    - Bwd prop
    - Update scaling factor *s*
      - If *dW* contains Inf/NaN then reduce ***s***, skip the update
      - If no Inf/NaN were detected for *N* updates then increase ***s***
    - Scale *dW* by **1/*s***
    - Update *W*

The automatic part

A Note on Gradient During Training

- **Popular belief:**
  - Gradient magnitudes become smaller during training
- **Observation:**
  - Not true for a number of networks
  - Thus FP16 range is not a concern
  - Normalizations especially tend to maintain distributions throughout training

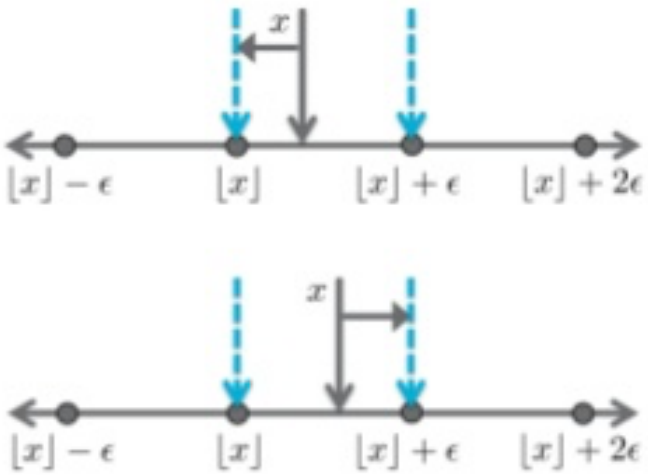
Summary of 16-32 Mixed Precision Training

- **Mixed precision training benefits:**
  - Math, memory speedups
  - Larger minibatches, larger inputs
- **Mixed precision matches FP32-trained accuracy for a variety of:**
  - Tasks: classification, regression, generation
  - Problem domains: images, language translation, language modeling, speech
  - Network architectures: feed forward, recurrent
  - Optimizers: SGD, Adagrad, Adam
  - With the same hyper-parameters as FP32 training
- **Automatic Loss Scaling simplifies training**
  - For example, Amp for PyTorch: https://github.com/NVIDIA/apex/tree/master/apex/amp
- **Inference:**
  - Can be purely FP16: storage and math (use library calls with FP16 accumulation)
  - Turing Tensor Cores also provide int8/int4/int1 matrix operations

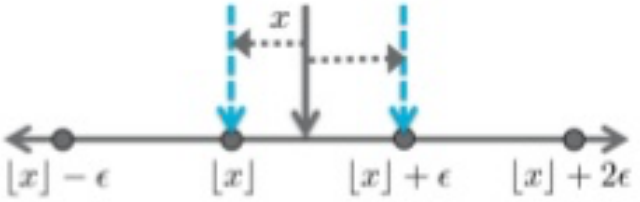
Stochastic Rounding

Fixed-Point Arithmetic: Rounding Modes

Round-to-nearest



Stochastic rounding



$$Round\,(x, \langle \mathrm{IL}, \mathrm{FL} \rangle) = \begin{cases} \lfloor x \rfloor & \text{w.p. } 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{w.p. } \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases}$$

- Non-zero probability of rounding to either $\lfloor x \rfloor$ or $\lfloor x \rfloor + \epsilon$
- Unbiased rounding scheme: expected rounding error is zero

https://www.slideshare.net/SuyogGupta1/hwdlnov2015-62497713

8-Bit Floating-Point Training

- FP8 (1,5,2) format
- Chunk-based computations
  - Hierarchically applied to GEMM
  - 8-bit multiplications
  - 16-bit additions
- Floating point stochastic rounding in the weight update process

Table 1: Dynamic range comparison between proposed FP8 and other existing floating point formats.

Data Type	Bit Format (s, e, m)	Max Normal	Min Normal	Min Subnormal
IEEE-754 float	1, 8, 23	3.40e38	1.17e−38	1.40e−45
IEEE-754 half-float	1, 5, 10	65 535	6.10e−5	5.96e−8
FP8 (proposed)	1, 5, 2	57 344	6.10e−5	1.52e−5

https://arxiv.org/abs/1812.08011

Training DNN with 8-Bit FP Numbers

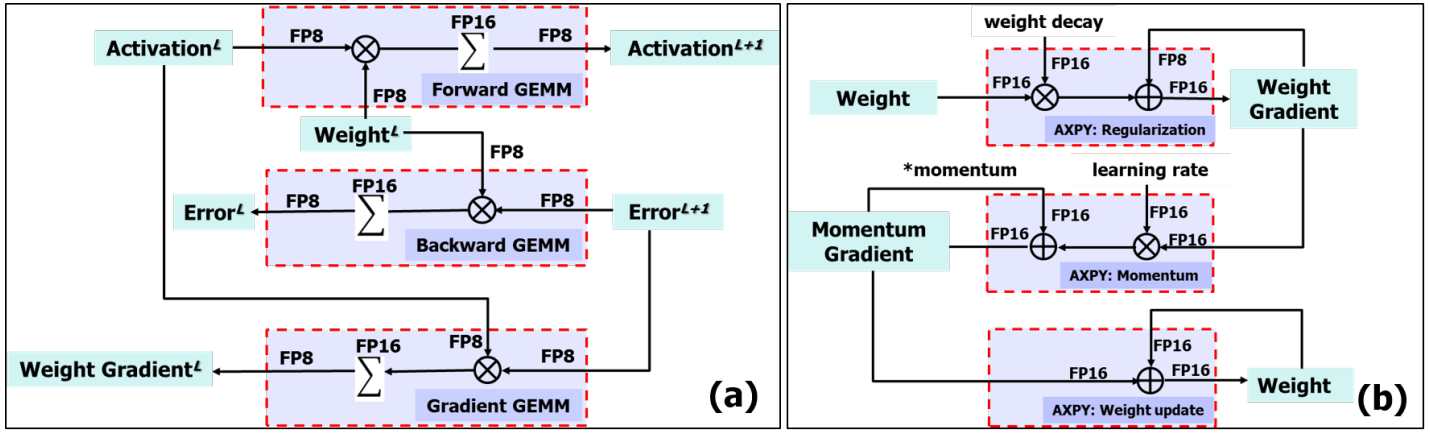


Figure 2: A diagram showing the precision settings for (a) three GEMM functions during forward and backward passes, and (b) three AXPY operations during a standard SGD weight update process.

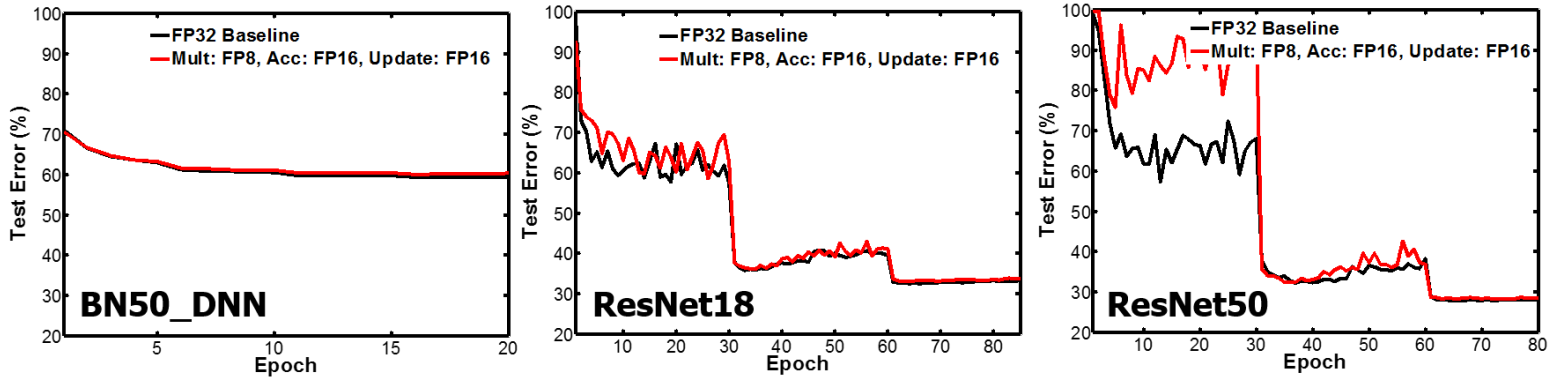


Figure 4: Reliable model convergence results across a spectrum of models and datasets using a chunk size of 64. $FP8$ is used for representations and $FP16$ is used for accumulation and updates.

Mixed Precision Training with 8-bit Floating Point

- FP8 (1,5,2) format with master copy of weights
  - 32-bit ⇒16-bit
- Enhanced loss scaling
- Stochastic rounding
  - Address gradient noise in early epochs
- Slightly better results than full precision

Table 3: Comparison of our method with the only other FP8 training method on Imagenet-1K[7] data set. W, A, E, G, MasterWts represent the precision setting for weights, activations, error, weight gradients and mater copy of weights respectively.

Method, Format	W,A,E,G	MasterWts	Resnet-18 (top-1 error %)	Resnet-50 (top-1 error %)
Wang et al.[24], FP8	8,8,8,8	16	33.05	28.28
Ours, FP8	8,8,8,8	16	30.29	24.30

Mixed Precision Training with 8-bit Floating Point

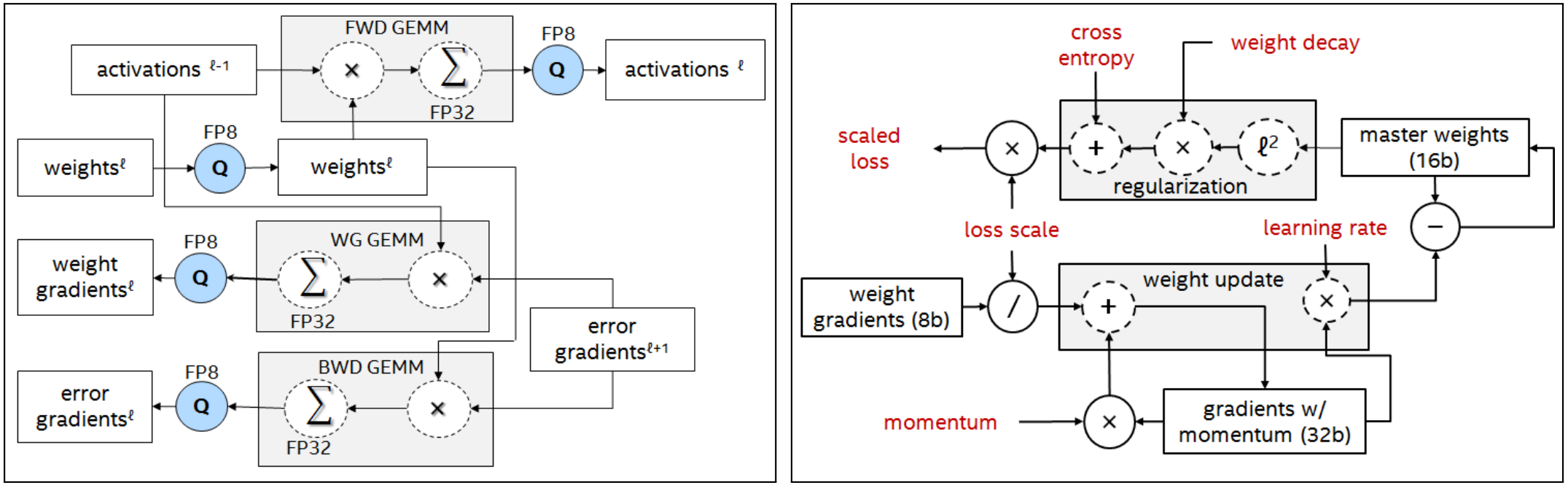
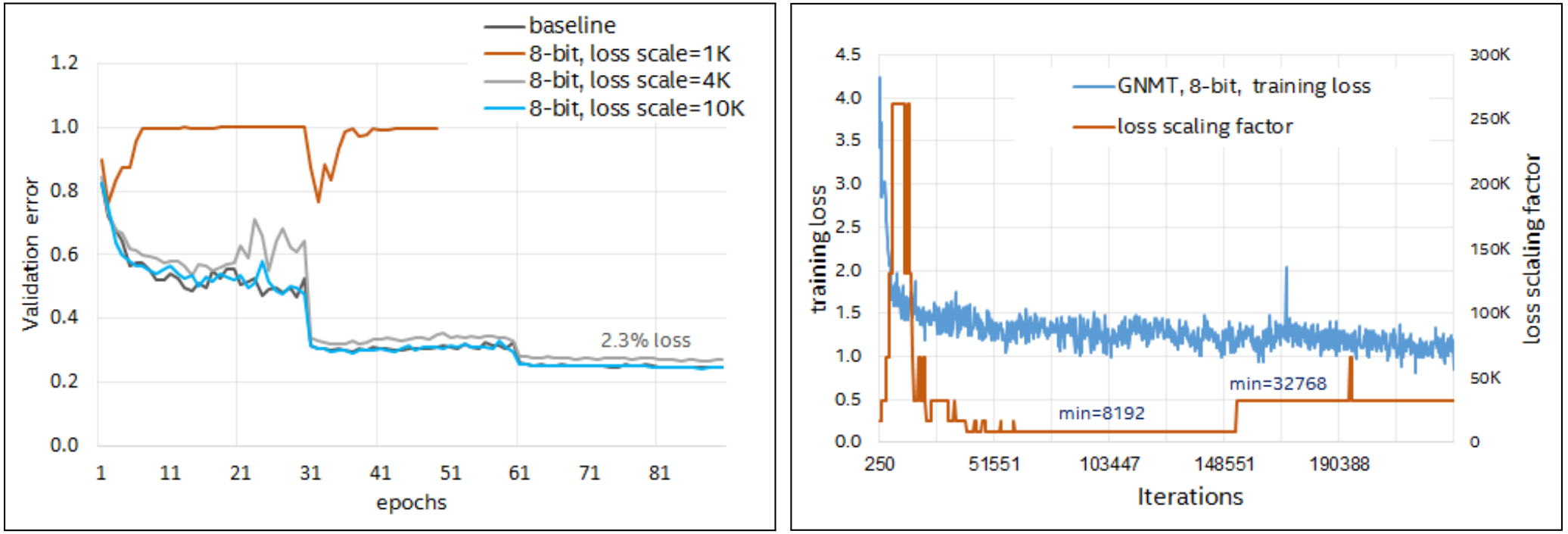


Figure 1: Mixed precision data flow for FP8 training. (left) precision settings for key compute kernels in Forward, Backward, and Weight Update passes, (right) flow diagram of the weight update rule.

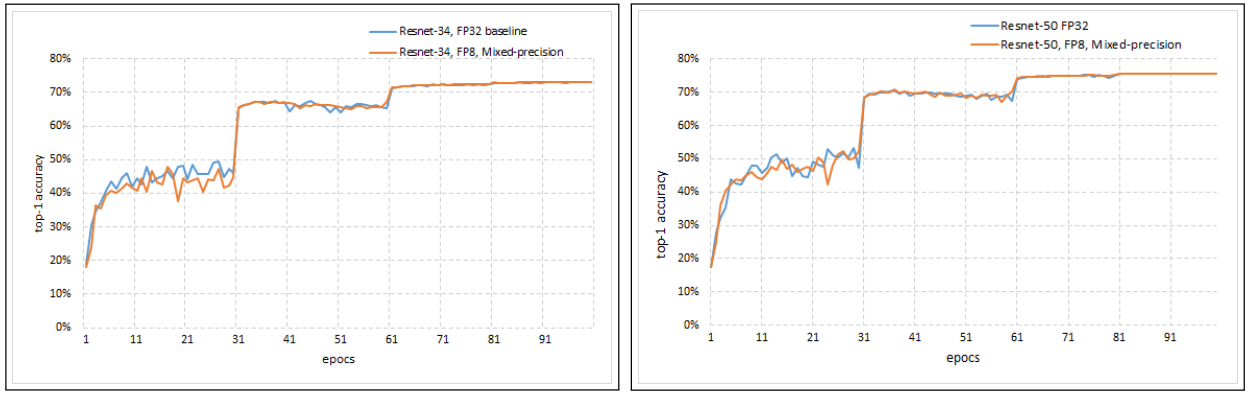
Enhanced and Dynamic Loss Scaling



(a) (b)

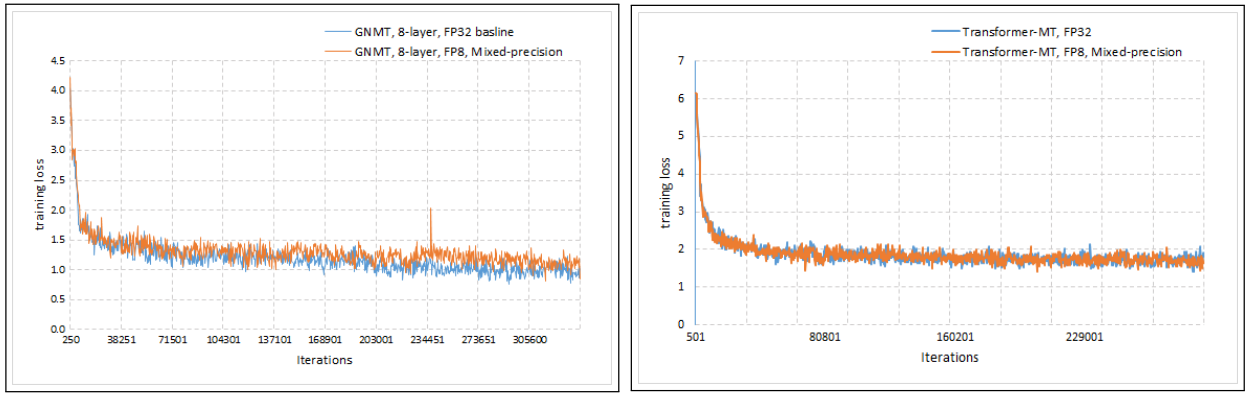
Figure 2: Convergence behaviour of FP8 training using enhanced loss scaling. (left) Resnet-50[12] failed to converge with loss scale=1000, performed better with 2.3% accuracy loss at loss scale=4000 and showed full convergence at loss scale=10 000, (right) Dynamic loss scaling with gradually increasing minimum threshold for the scaling factor.

Better Accuracy Than Baseline



(a) (b)

Figure 5: Convergence plots showing Top-1 validation accuracy for. (a) Resnet-34[12] (b) Resnet-50[12] on imagenet-1K[7] dataset.



(a) (b)

Figure 6: Convergence plots showing training loss for (a) 8-layer GNMT[26] and, (b) 6-layer Transformer[23] trained on WMT16 English->German dataset.

8-bit Floating Point Training

- Extended to GNMT, Resnet 18/34/50, and Transformers
- Master copy of weights
  - 32-bit ⇒16-bit
- Enhanced loss scaling
- Stochastic rounding
  - Gradient noise in the early epochs
- Better than Full Precision

- FP8 (1,5,2) format
- Chunk-based computations
  - Hierarchically applied to GEMM
  - 8-bit multiplications
  - 16-bit additions
- Floating point stochastic rounding in the weight update process

https://arxiv.org/abs/1905.12334

https://arxiv.org/abs/1812.08011

Training

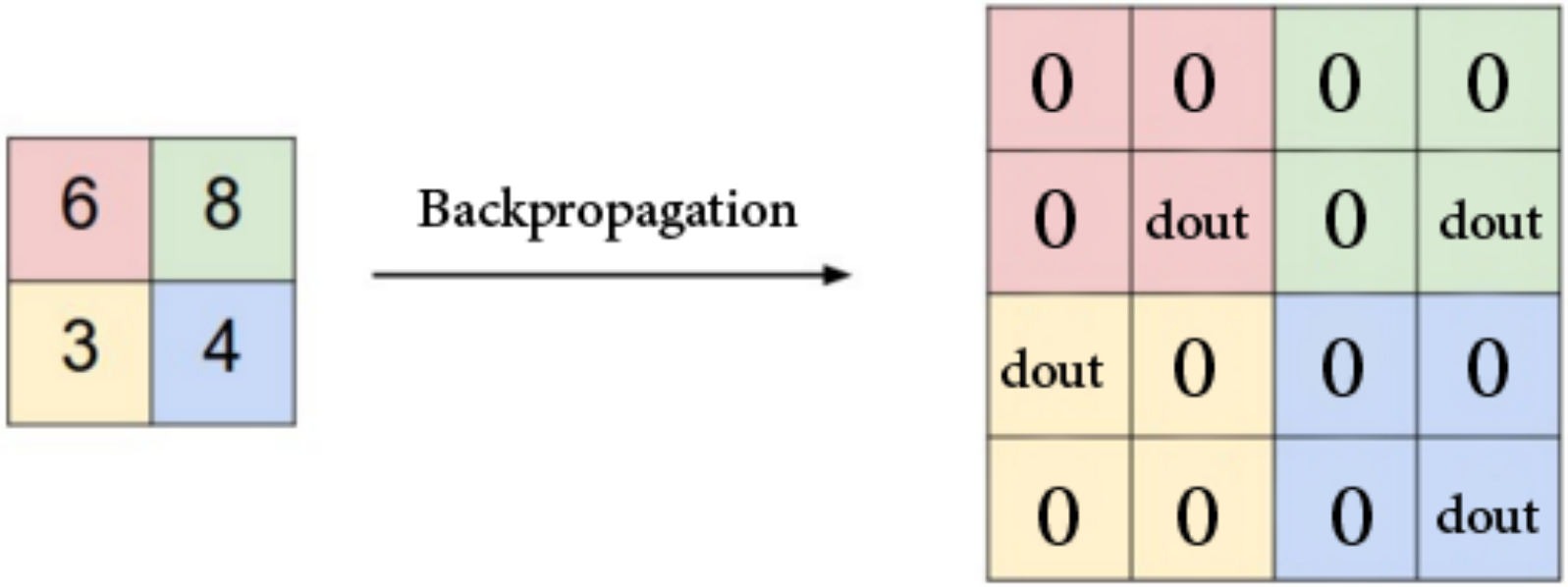
1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Sparsity

- Activation Sparsity
  - RELU Forward
  - RELU derivation Backward
  - MAXPOOL (Backward)

- Weight Sparsity
  - Compression/Pruning
  - Sparse Training
  - Structural Sparsity

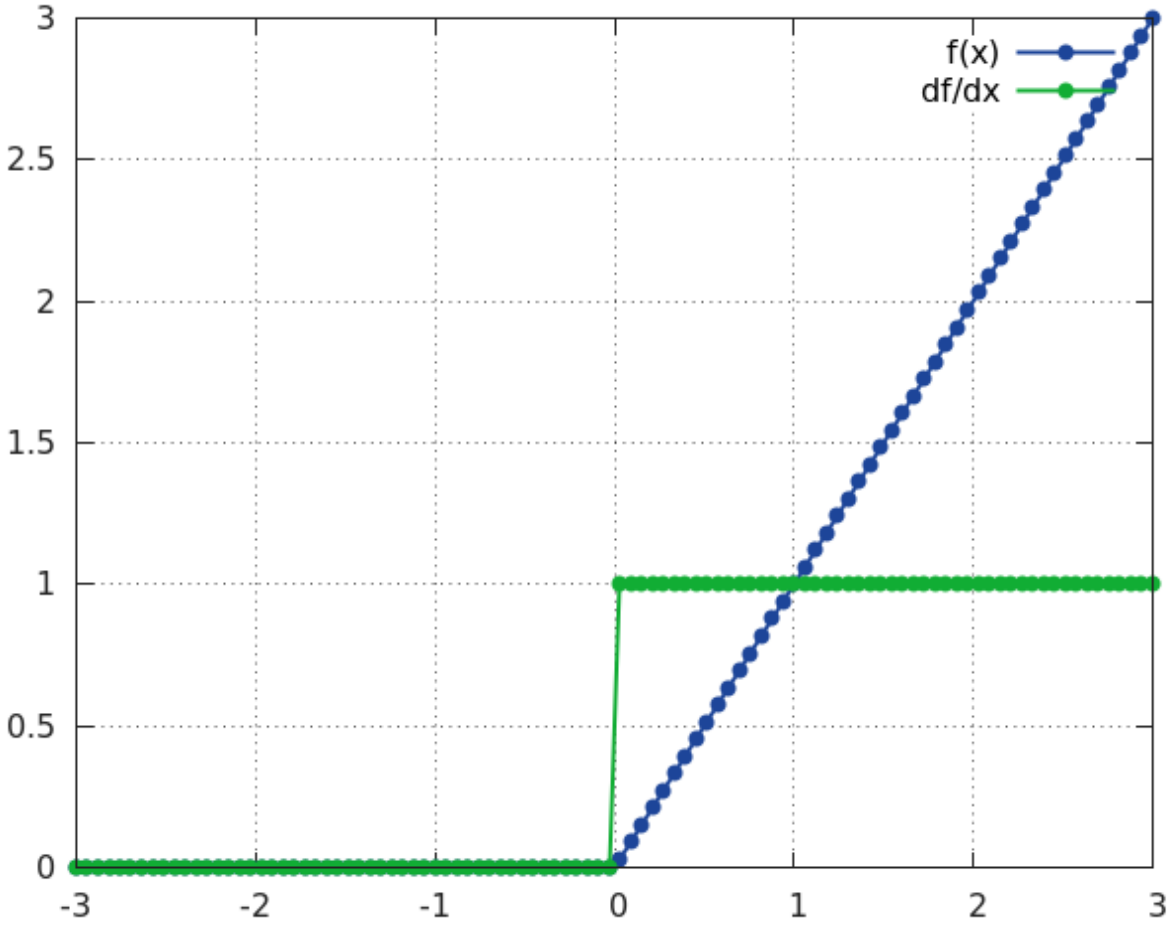
Pooling Layer



In the process we replace the maximum values before max pooling with 1 and set all the non maximum values to zero then use the **Chain rule** to multiply the gradient by them.

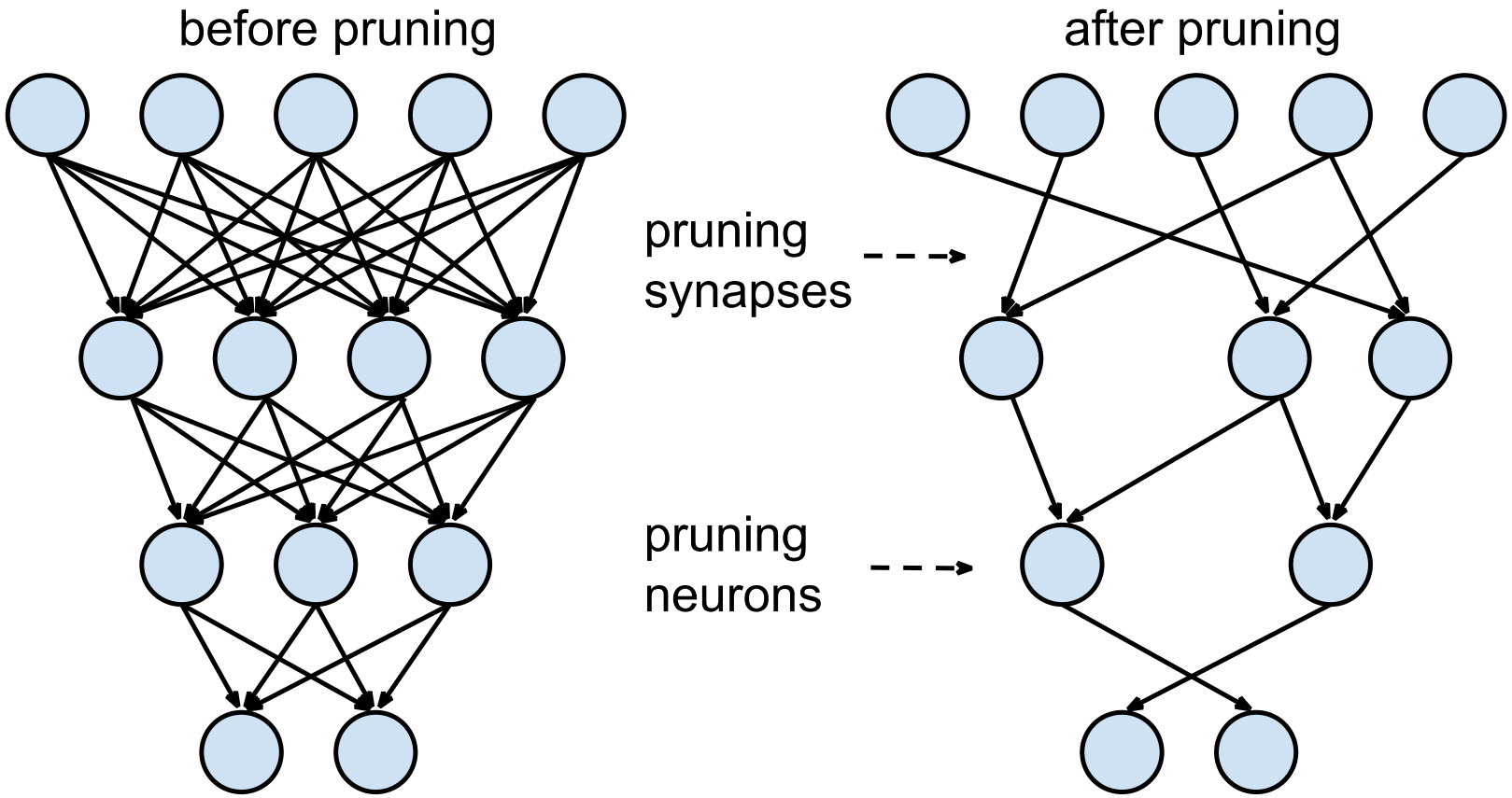
https://mc.ai/demystifying-convolutional-neural-networks/

ReLu Derivation

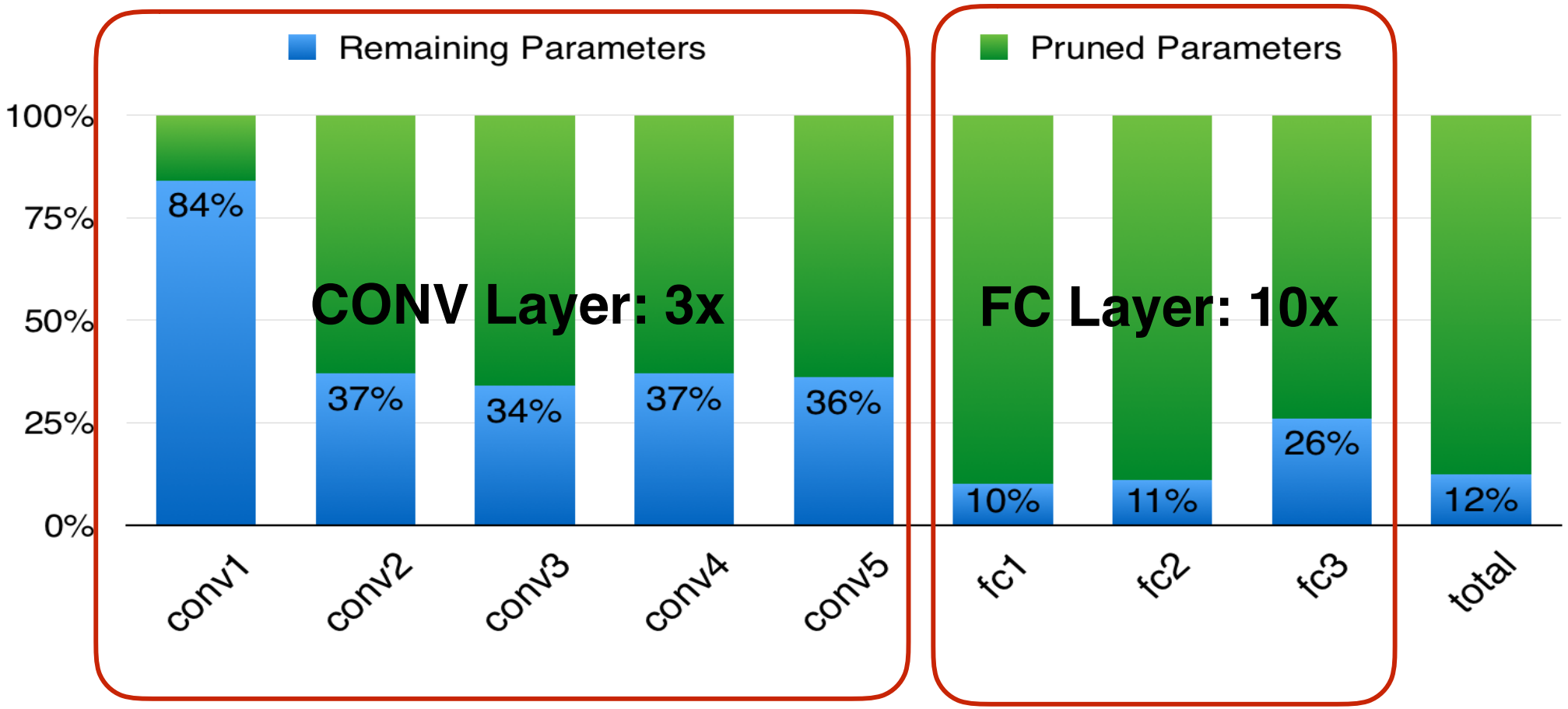


https://miro.medium.com/max/1280/1*akqBxW4dv35MKlU8a1-0Ag.png

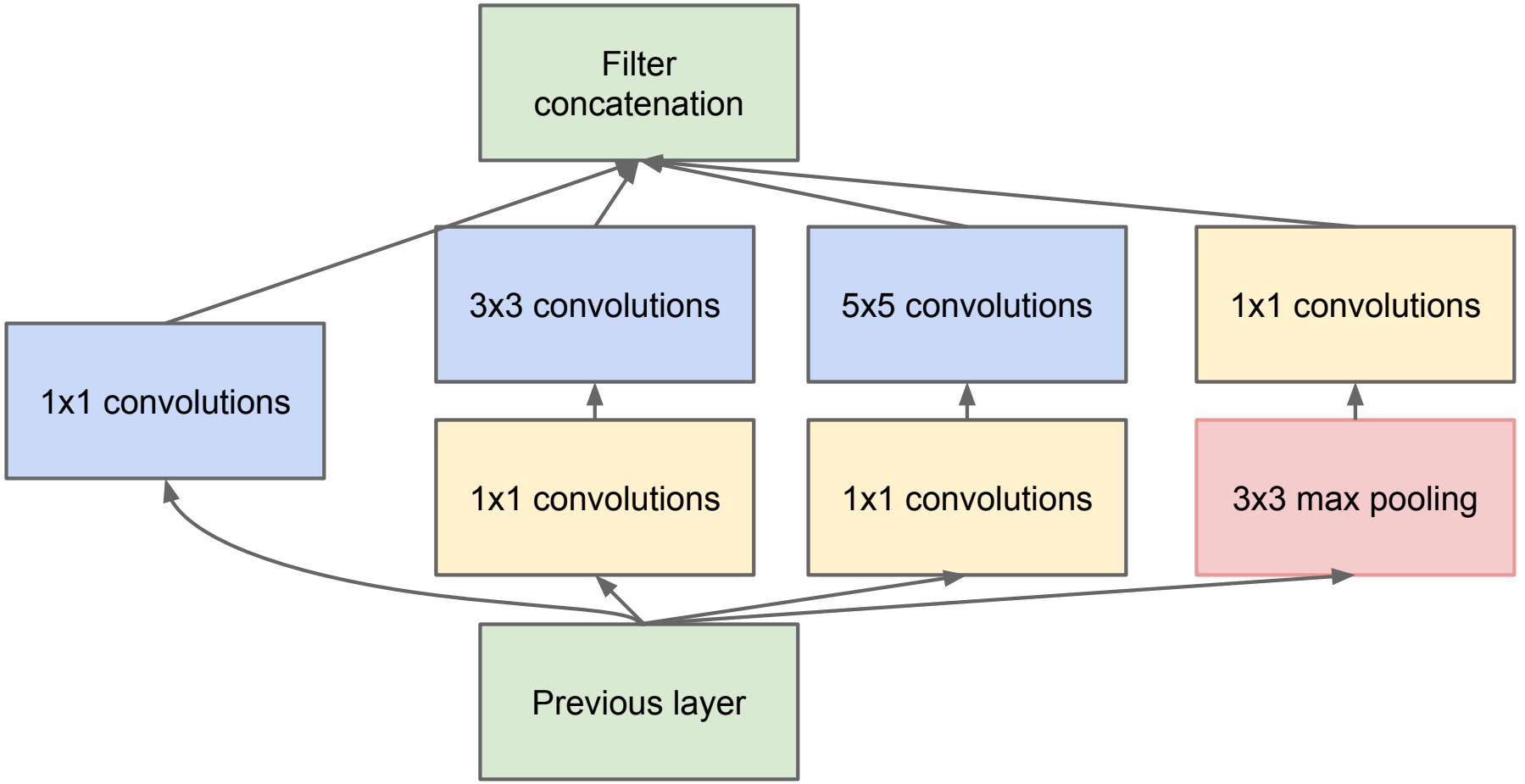
Pruning/Compressing The Network



Alexnet Pruning



Newer Deeper Models are Sparse Architectures



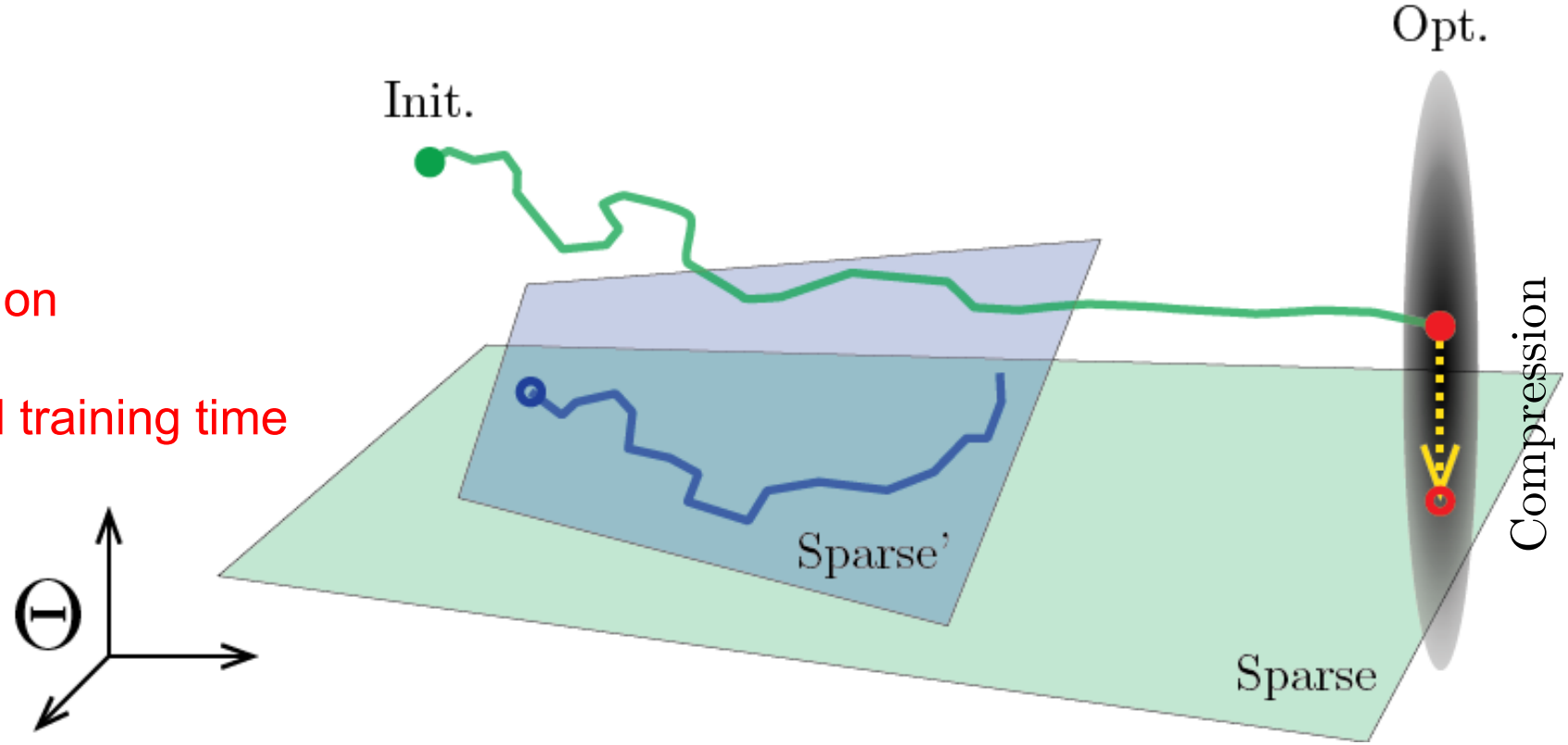
Pruning new models
is not as pruning easy as AlexNet

Sparse Training vs Compression

Easy: post-training (sparse) compression

Hard: direct training of sparse networks

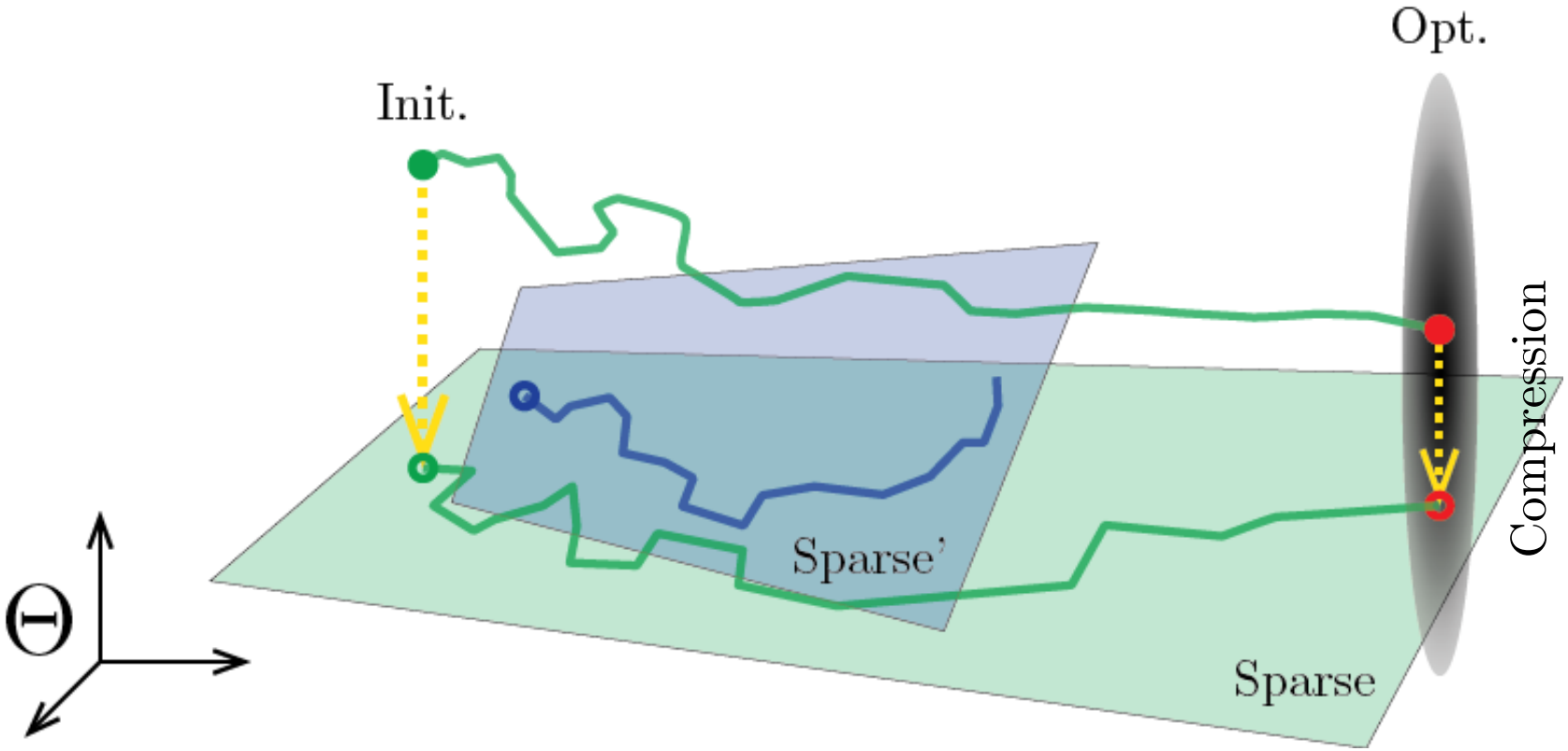
Compression increases end-to-end training time



https://icml.cc/media/Slides/icml/2019/201(12-11-00)-12-12-10-4913-parameter_effic.pdf

Lottery Ticket

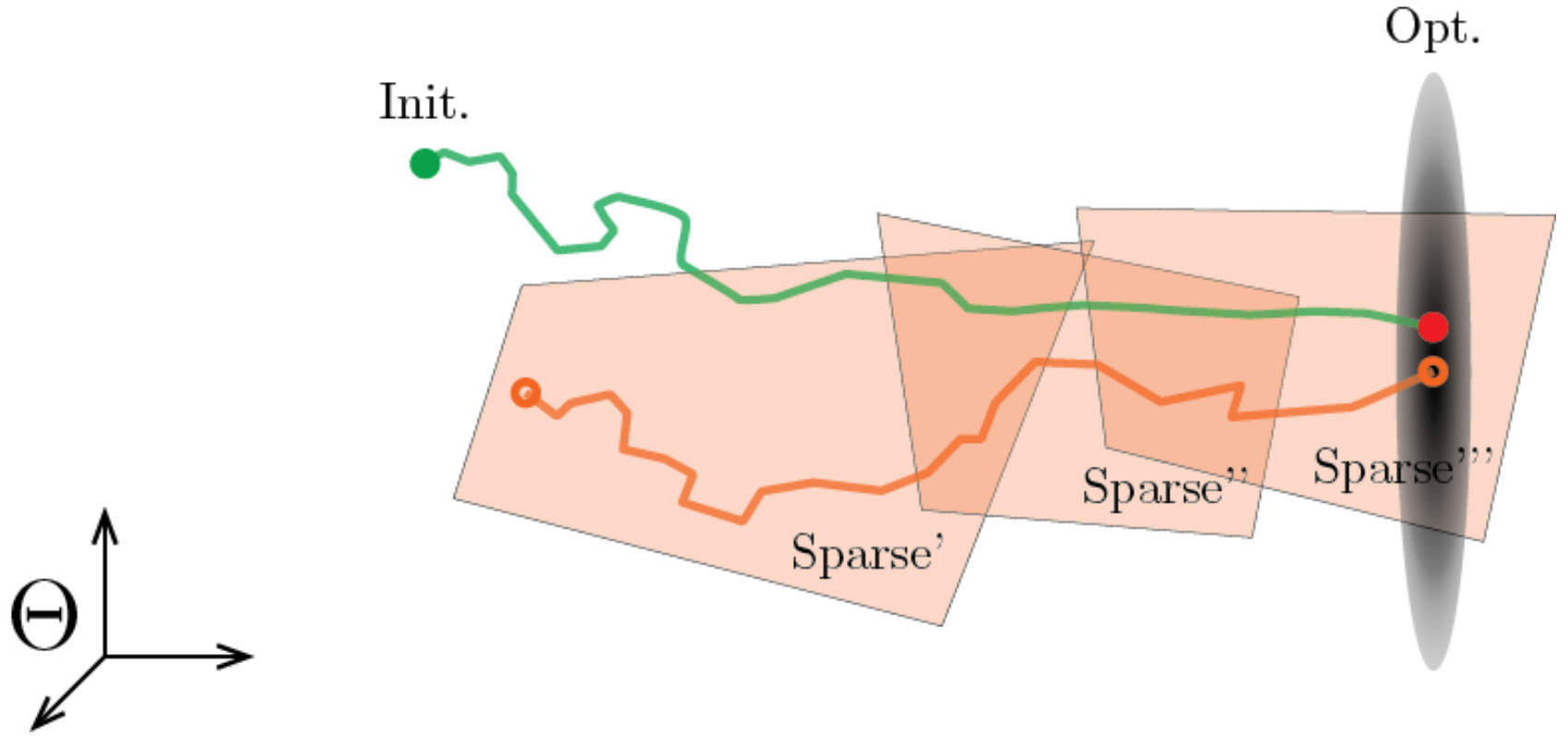
post hoc identification of trainable sparse nets



https://icml.cc/media/Slides/icml/2019/201(12-11-00)-12-12-10-4913-parameter_effic.pdf

Dynamic Sparse Reparametrization

training-time structural exploration



https://openreview.net/forum?id=S1xBioR5KX

Direct Training Sparse Networks

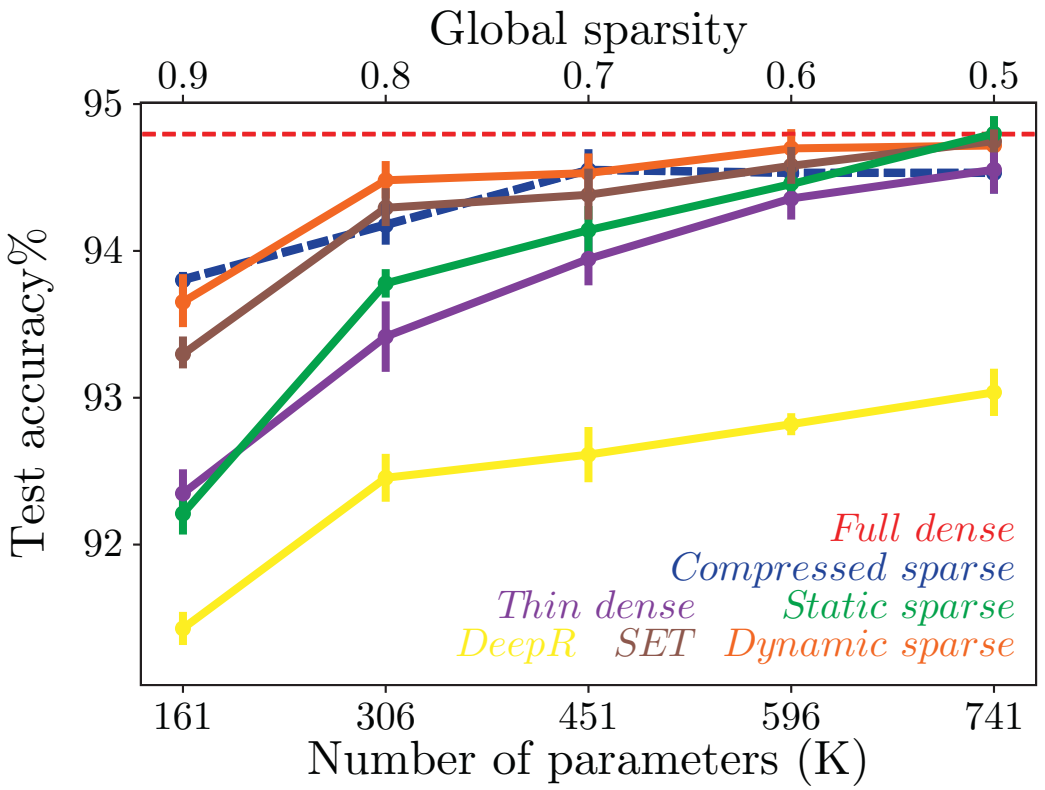
- **Generalize as well as post-training compression**

- **Directly trained sparse nets**:
  *are they "winning lottery tickets"? -**NO***

https://openreview.net/forum?id=S1xBioR5KX

Post Training vs Direct Sparse Training

WRN-28-2 on CIFAR10

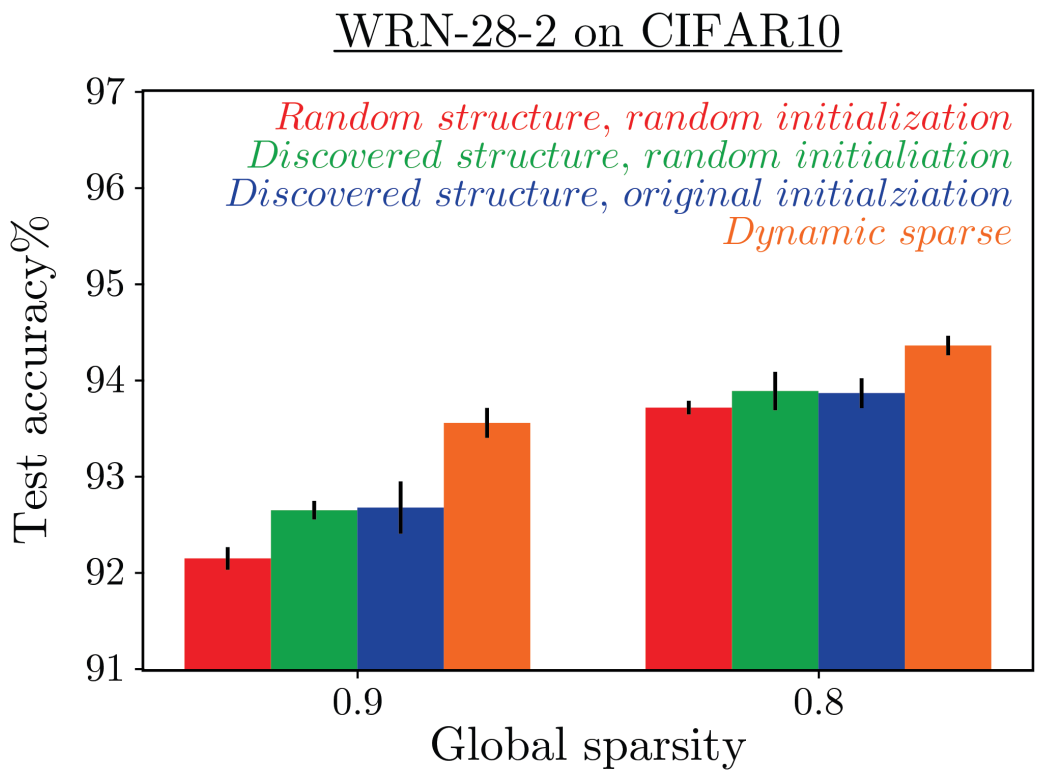


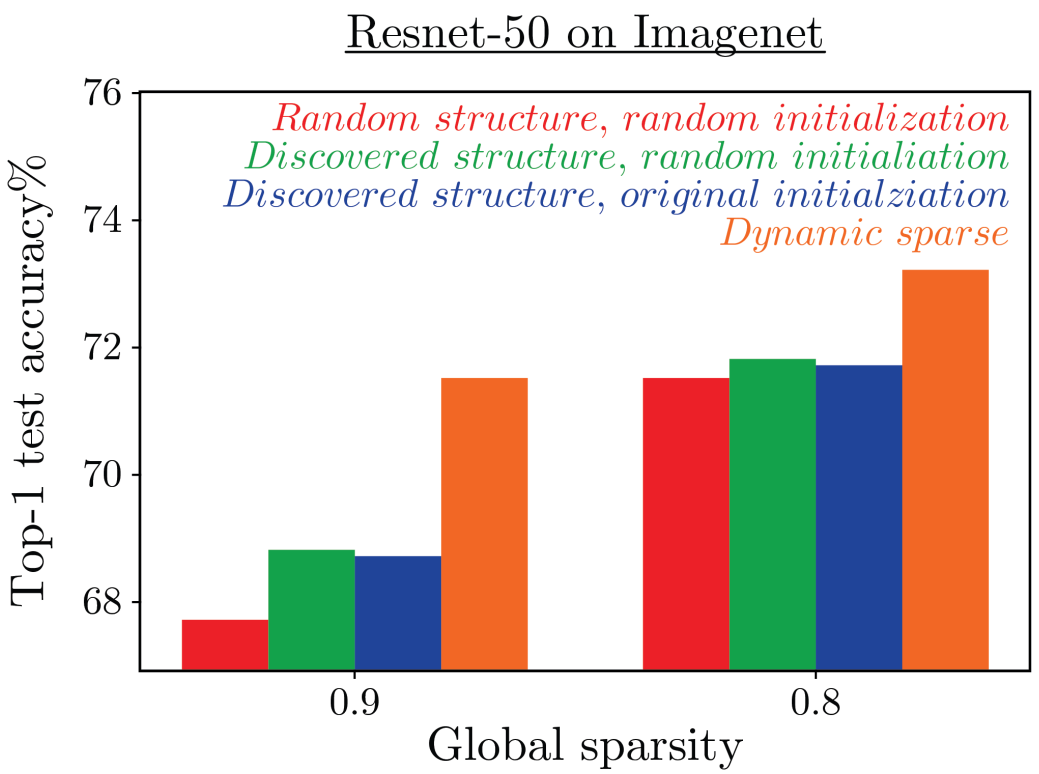
Resnet-50 on Imagenet

Sparsity (# Param)	0.8 (7.3M)		0.9 (5.1M)		0.0 (25.6M)	
Thin dense	72.4 [-2.5]	90.9 [-1.5]	70.7 [-4.2]	89.9 [-2.5]		
Static sparse	71.6 [-3.3]	90.4 [-2.0]	67.8 [-7.1]	88.4 [-4.0]		
DeepR (Bellec et al., 2017)	71.7 [-3.2]	90.6 [-1.8]	70.2 [-4.7]	90.0 [-2.4]	74.9 [0.0]	92.4 [0.0]
SET (Mocanu et al., 2018)	72.6 [-2.3]	91.2 [-1.2]	70.4 [-4.5]	90.1 [-2.3]		
Dynamic sparse (Ours)	**73.3** [**-1.6**]	**92.4** [**0.0**]	**71.6** [**-3.3**]	**90.5** [**-1.9**]		
Compressed sparse (Zhu & Gupta, 2017)	73.2 [-1.7]	91.5 [-0.9]	70.3 [-4.6]	90.0 [-2.4]		

https://openreview.net/forum?id=S1xBioR5KX

Exploration of Structural Degrees of Freedom

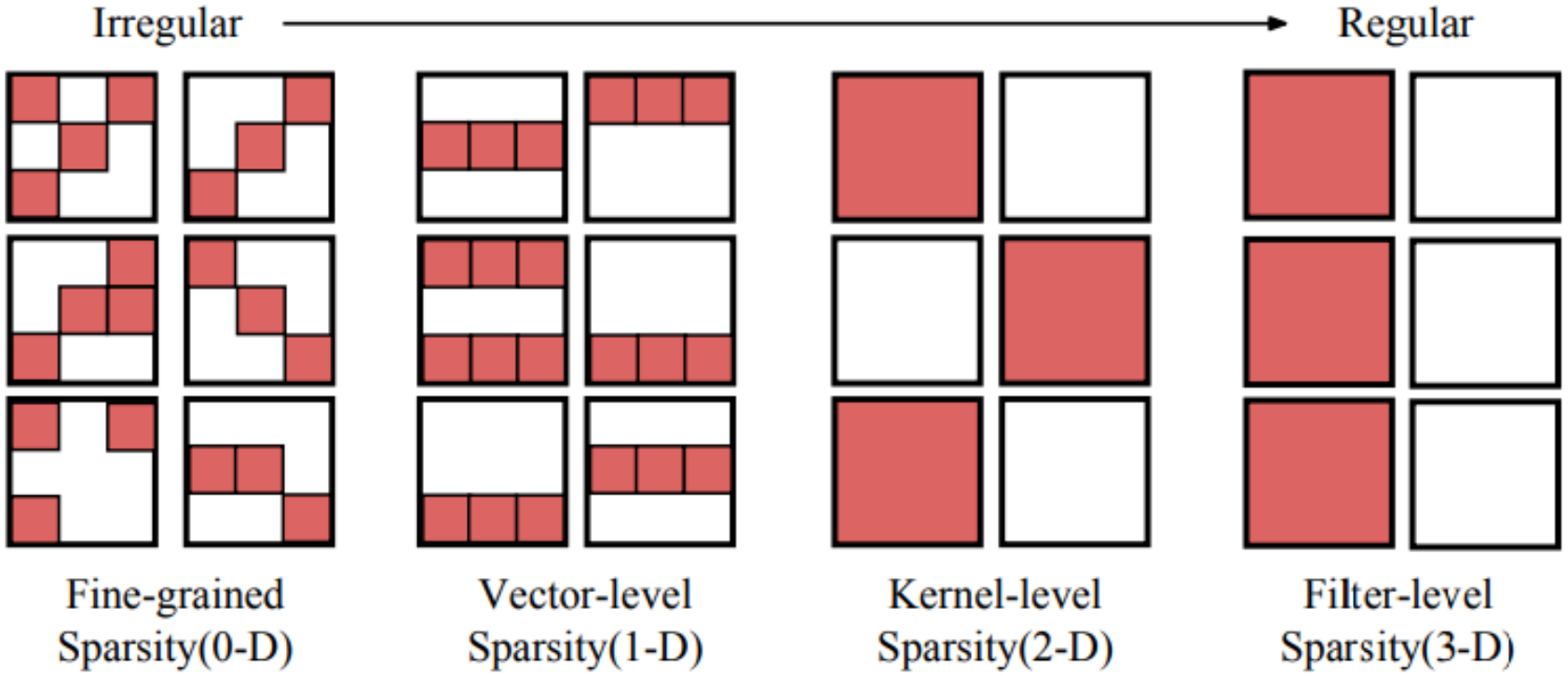




https://openreview.net/forum?id=S1xBioR5KX

Training Structured Sparse Models

- Fine-grained pruning – *requires special accelerators to perform inference on compressed models*
- Structural pruning – *tends to have reduced accuracy*



Mao et. al "Exploring the Granularity of Sparsity in Convolutional Neural Networks"

Structured Sparsity for RNNs & CNNS

- RNNs [1]
  - 90% sparsity reduces relative accuracy by 10% to 20%
  - Solution: Make the sparse model larger
  - Large sparse model still have less parameters compared to the small dense baseline and achieves a slight increase in accuracy

- CNNs [2]
  - **Pruning (Post Hoc)** with large granularity will greatly hurt accuracy
  - Due to index savings, coarse-grain pruning can still achieve space savings even at a lower overall sparsity

[1]- Sharan Narang, Erich Elsen, Gregory Diamos, Shubho Sengupta, "Exploring Sparsity In Recurrent Neural Network", ICLR 2017. https://arxiv.org/abs/1704.05119
[2]- Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, William J. Dally," Exploring the Granularity of Sparsity in Convolutional Neural Networks" CVPR'17 TMCV workshop. https://arxiv.org/abs/1705.08922

Dynamic Sparse with Kernel Granularity

Structural Sparse Training

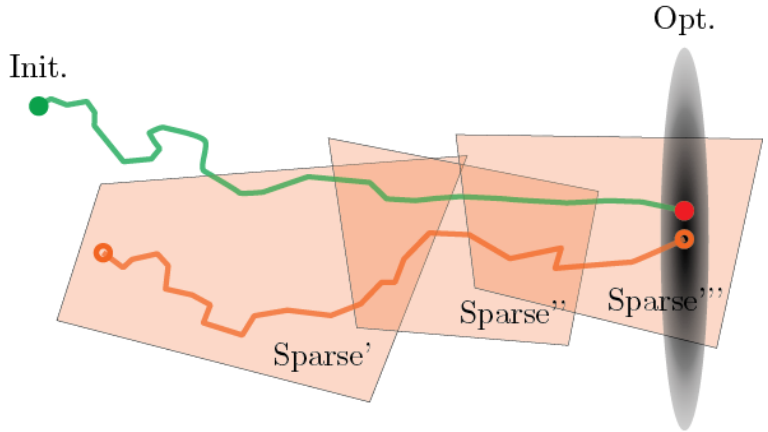


Table 7: Test accuracy% (top-1, top-5) of Resnet-50 on Imagenet for different levels of granularity of sparsity

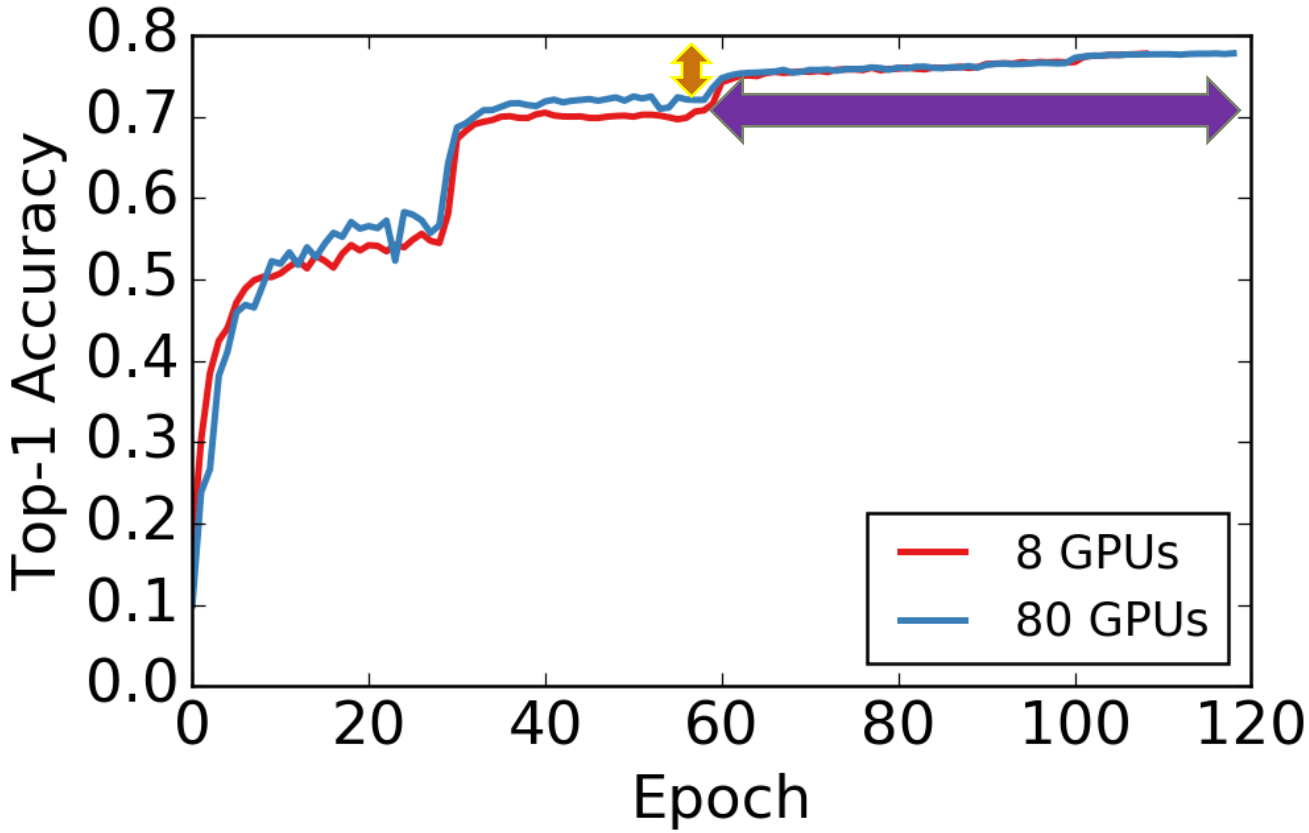
Final overall sparsity (# Parameters)	0.8 (7.3M)		0.9 (5.1M)	
Thin dense	71.6 [-3.3]	90.3 [-2.1]	69.4 [-5.5]	89.2 [-3.2]
Dynamic sparse (kernel granularity)	72.6 [-2.3]	91.0 [-1.4]	70.2 [-4.7]	89.8 [-2.6]
Dynamic sparse (non-structured)	**73.3 [-1.6]**	**92.4 [0.0]**	**71.6 [-3.3]**	**90.5 [-1.9]**

Numbers in square brackets are differences from the *full dense* baseline.

https://openreview.net/forum?id=S1xBioR5KX

Training Time for that Last 1% accuracy

- The Last 1-2% of Convergence Accuracy costs 30% of Time
- Loosing 1-2% accuracy over sparse training
- Dense Training is Better



Pruning Introduces Indirection/Indexing

- Alvarez et al. adds regularizer term in the training loss
  - Parameter matrix of each layer to have low rank
  - Explicitly account for post training compression
- Pruning techniques use indexing techniques
- Slow down training time
- Can one train sparse networks without increasing training time?

Zhou et al. “Less is more: Towards compact cnns.” In *European Conference on Computer Vision*, pp. 662–677. Springer, 2016.
Wen et al. “Learning structured sparsity in deep neural networks.” *Advances in Neural Information Processing Systems* 29, pp. 2074–2082. Curran Associates, Inc., 2016.
Alvarez, J. M. and Salzmann, M. “Compression-aware training of deep networks.” In *Advances in Neural Information Processing Systems*, pp. 856–867, 2017.

Group Lasso Regularization [1]

- Modifies the optimization loss function
  - Consideration for weight magnitude
- Induces sparsity in the certain groups
  - e.g. input and output channels ⇒*removal of entire channels*
- *Expensive to include at the start of training*
- Been used to compress models but not reduce computation [2]

[1] Meier, L., Van De Geer, S., and Buhlmann, P. "The group lasso for logistic regression." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(1): 53–71, 2008.
[2] Zhou et al. "Less is more: Towards compact CNNs." In *European Conference on Computer Vision*, pp. 662–677. Springer, 2016.

Learning Structured Sparsity with Group Lasso

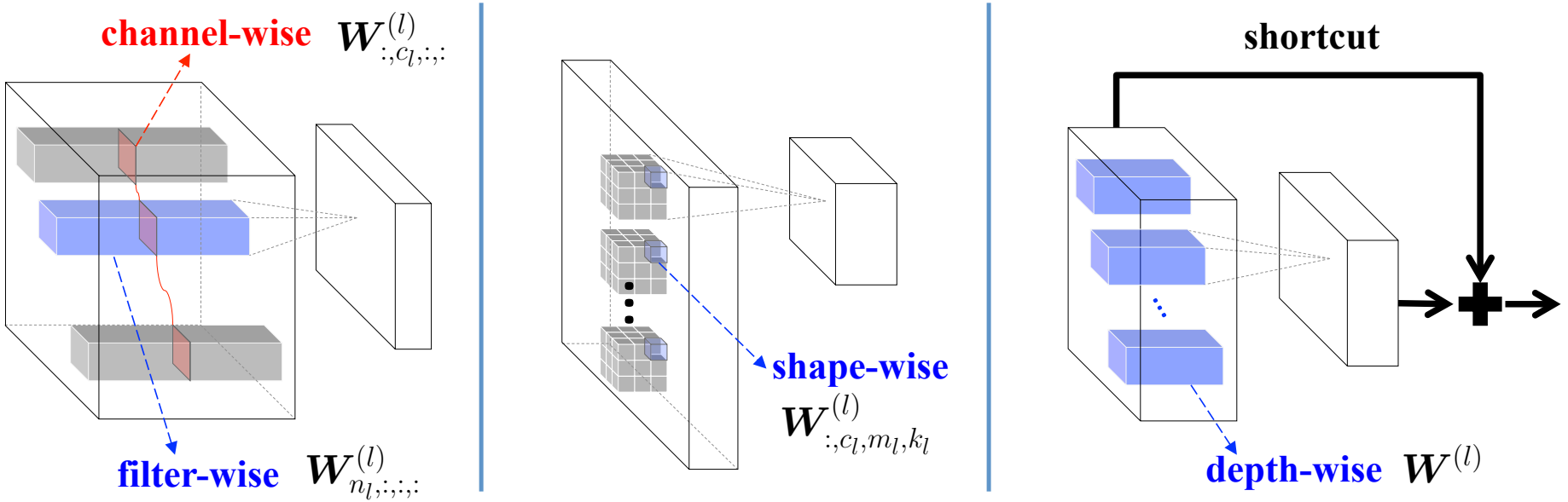


Figure 2: The proposed structured sparsity learning (SSL) for DNNs. Weights in filters are split into multiple groups. Through group Lasso regularization, a more compact DNN is obtained by removing some groups. The figure illustrates the filter-wise, channel-wise, shape-wise, and depth-wise structured sparsity that were explored in the work.

https://arxiv.org/pdf/1608.03665.pdf

Layer-wise Pruning Through Channel Pruning

- Overlap between input and output channel lasso groups (Fig. 3a),
- Unimportant layers are removed
  - Without additional layer-wise weight regularization
- An input channel becomes sparse (Fig. 3b)
- Gradually sparsify all the intersecting output channels (c),

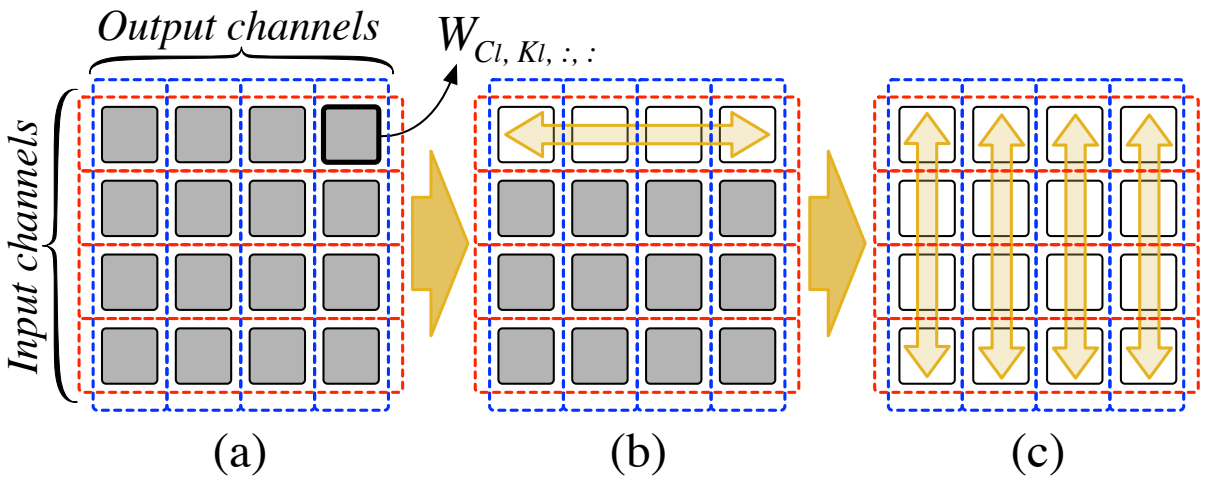
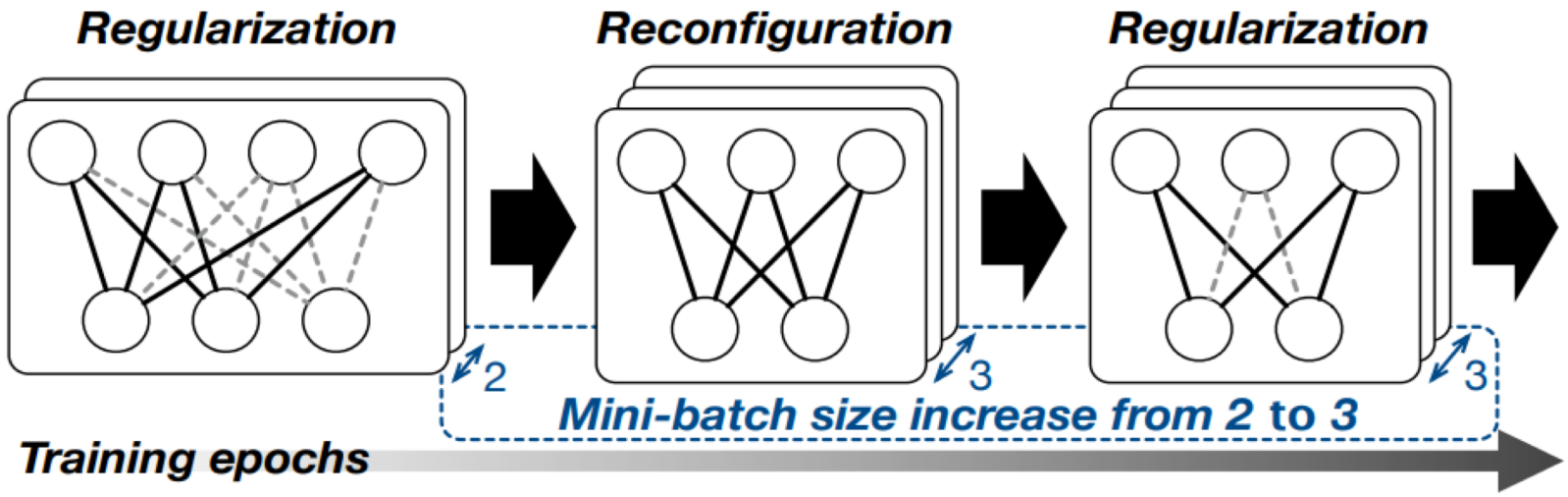


Fig. 3. Group lasso regularization structure of a convolution layer: Weights of a filter (each square box) affect the sparsification of weights in both input and output channels (red and blue dotted boxes). The white filters are zeroed-out after sparsification.

PruneTrain

- *Gradually pruning*
- *Dynamically reconfiguring*
- Dynamic mini-batch size adjustment

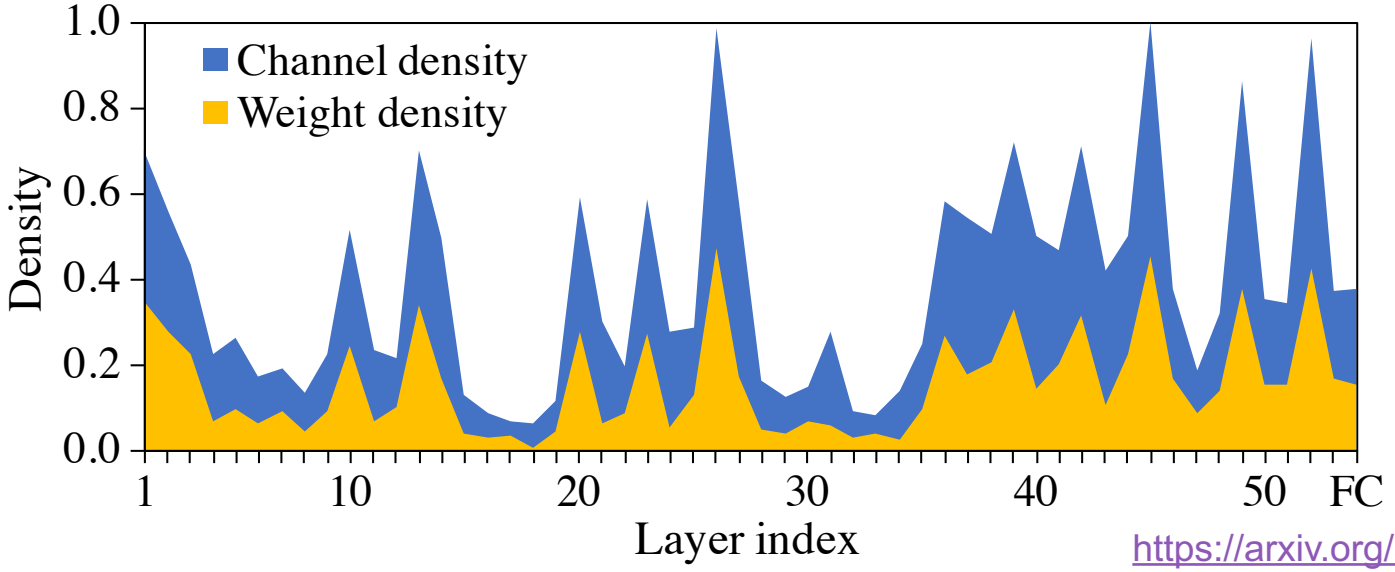


https://arxiv.org/abs/1901.09290

PruneTrain: Channel-Union Pruning

Short-cut connections

- Keep channel *i* if the weights for the preceding layer(s) at that channel or the weights for the successive layer(s) at that channel are dense: "channel union"
- No indexing since any pruned channel *i* is no longer in both preceding layer or successive layer



https://arxiv.org/abs/1901.09290

Sparse Training Time with Cost of Accuracy Drop

Dataset	Network	Val. Accuracy Δ (fine-tunning)	Train. FLOPs (E2E time)	Inference FLOPs
CIFAR10	ResNet32	-1.8%	47% (81%)	34%
	ResNet50	-1.1%	50% (81%)	30%
	VGG11	-0.7%	43% (57%)	35%
	VGG13	-0.6%	44% (57%)	37%
CIFAR100	ResNet32	-1.4%	68% (88%)	54%
	ResNet50	-0.7%	47% (66%)	31%
	VGG11	-1.3%	53% (74%)	43%
	VGG13	-1.1%	58% (67%)	48%
ImageNet	ResNet50	-1.87% (-1.58%)	63% (73%)	47%
		-1.58% (-0.98%)	71% (78%)	57%

Even When Dense Training The Last 1-2% of Convergence Accuracy costs 30% of Training Time

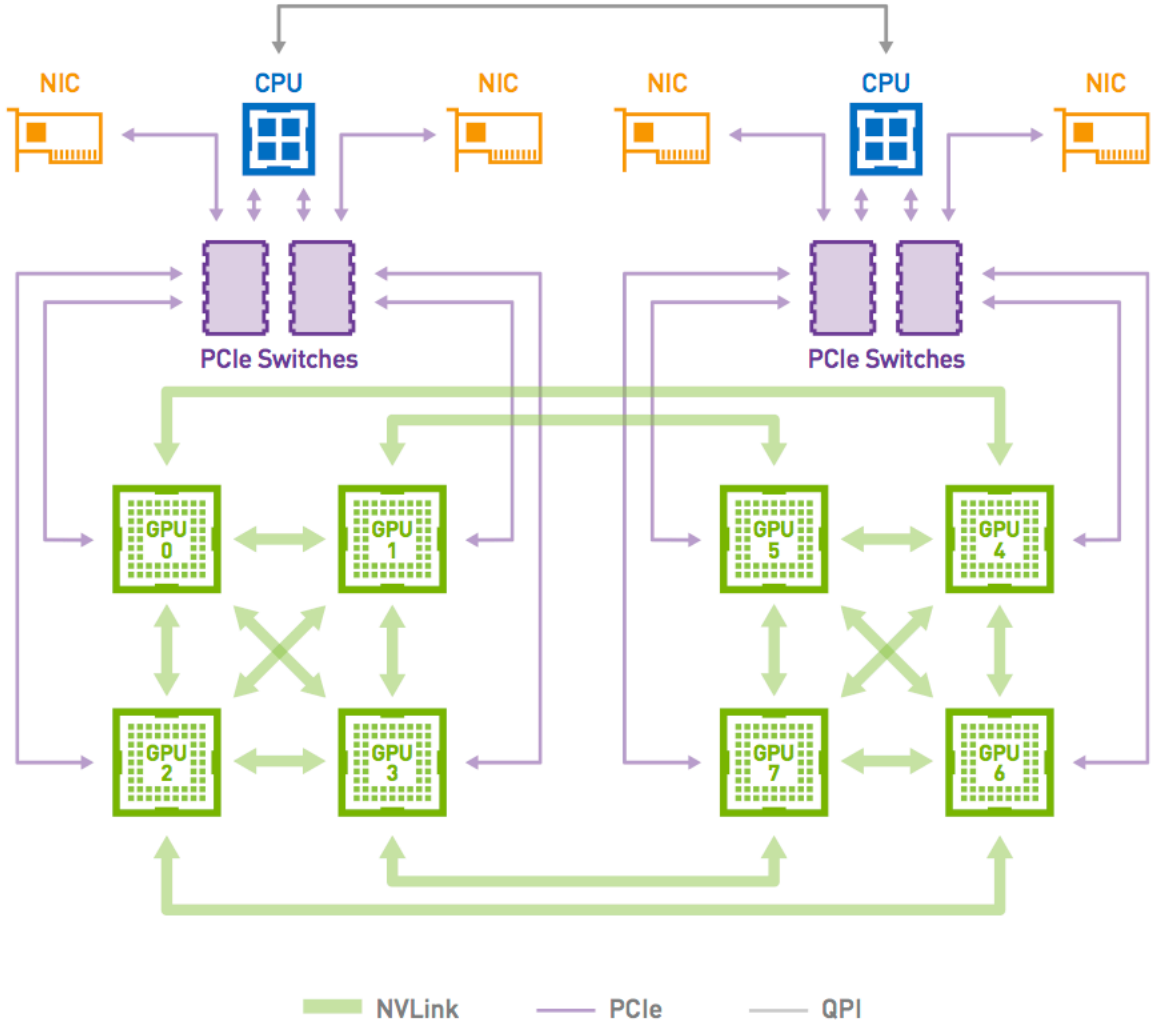
Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Scaling of Training

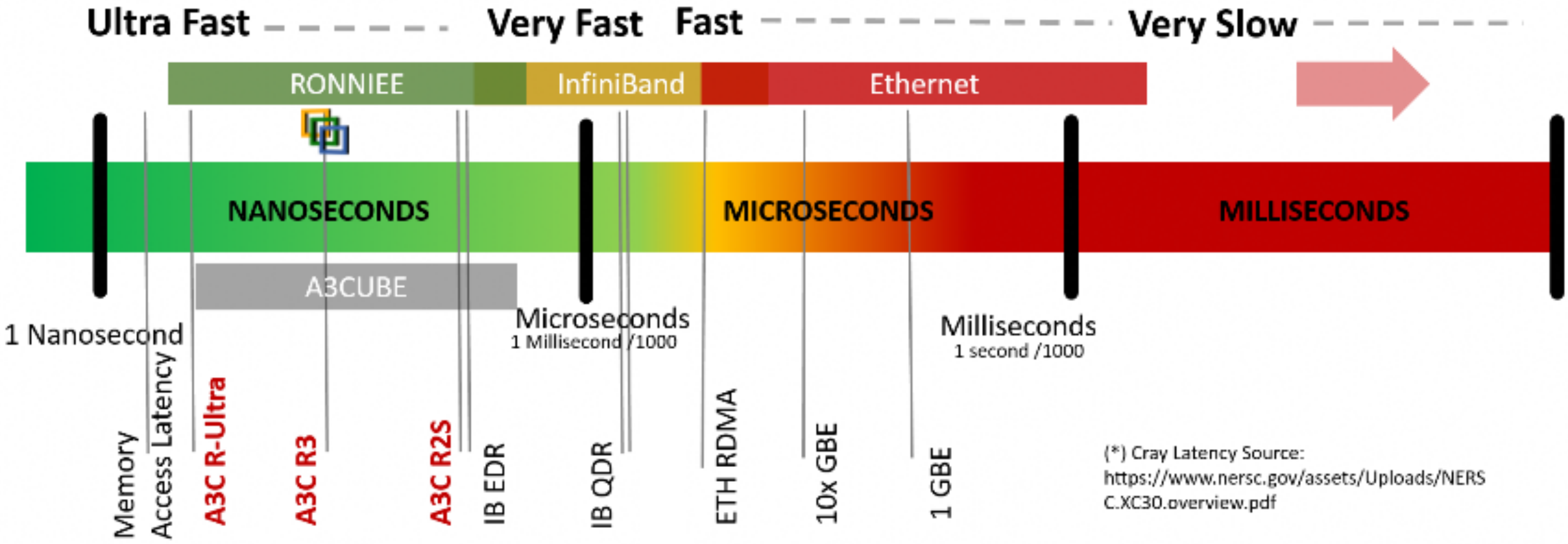
- Scaling the problem: Same system, bigger network
  - Memory bottleneck
  - Cost of computation vs. communication
- Scaling the system: Bigger system
  - Synchronization bottleneck
  - Data communication on the cloud
  - Cloud scale synchronized SGD
  - Asynchronous SGD

SCALING THE SYSTEM



Mark Harris, “NVIDIA DGX-1: The Fastest Deep Learning System”, April 2017.

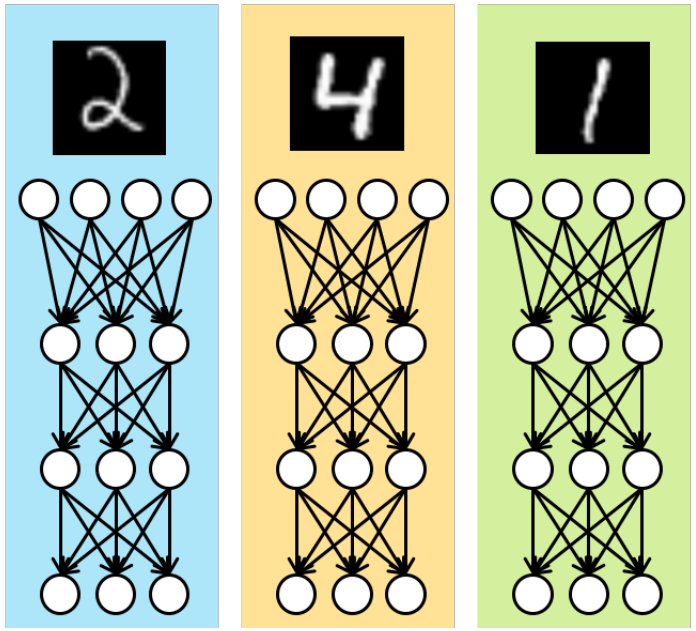
Latency IS A Bottleneck



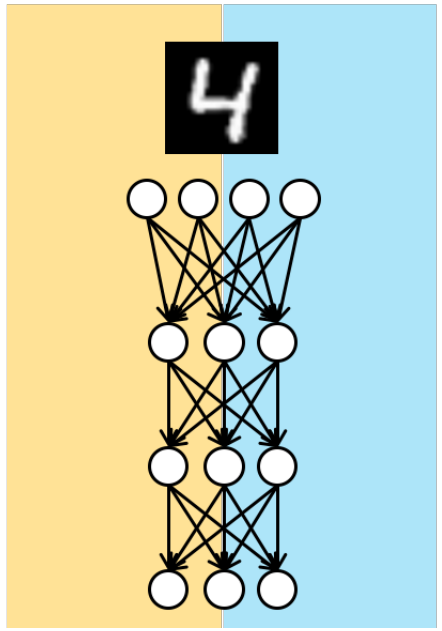
http://www.a3cube-inc.com/-latency-matters.html

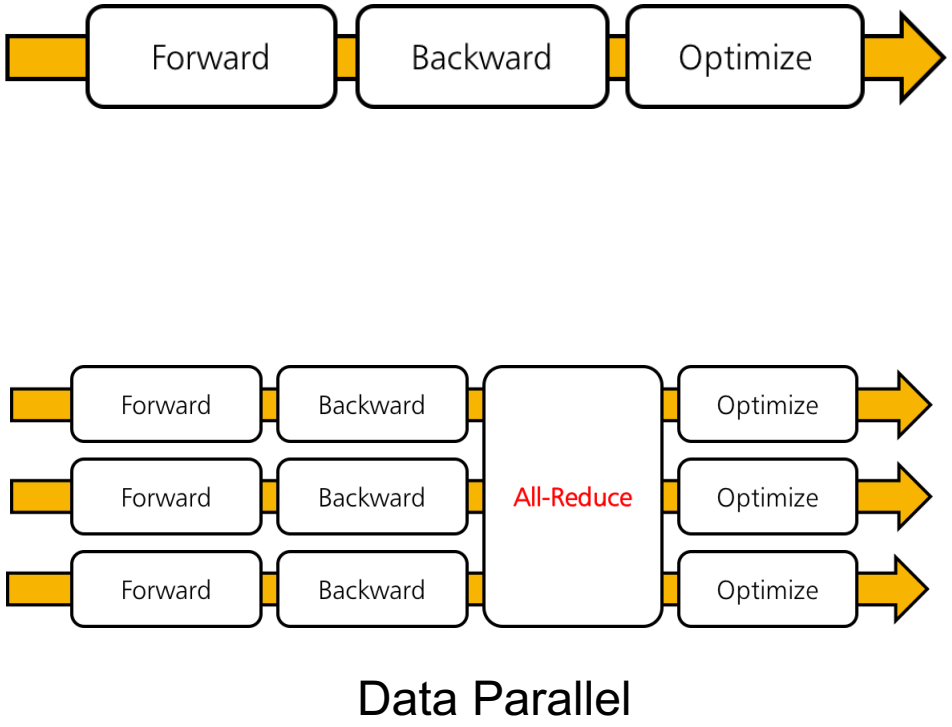
Data vs. Model Parallelism

Data Parallel



Model Parallel



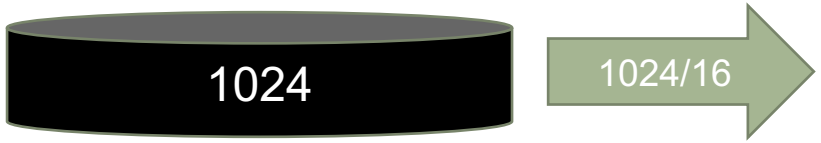


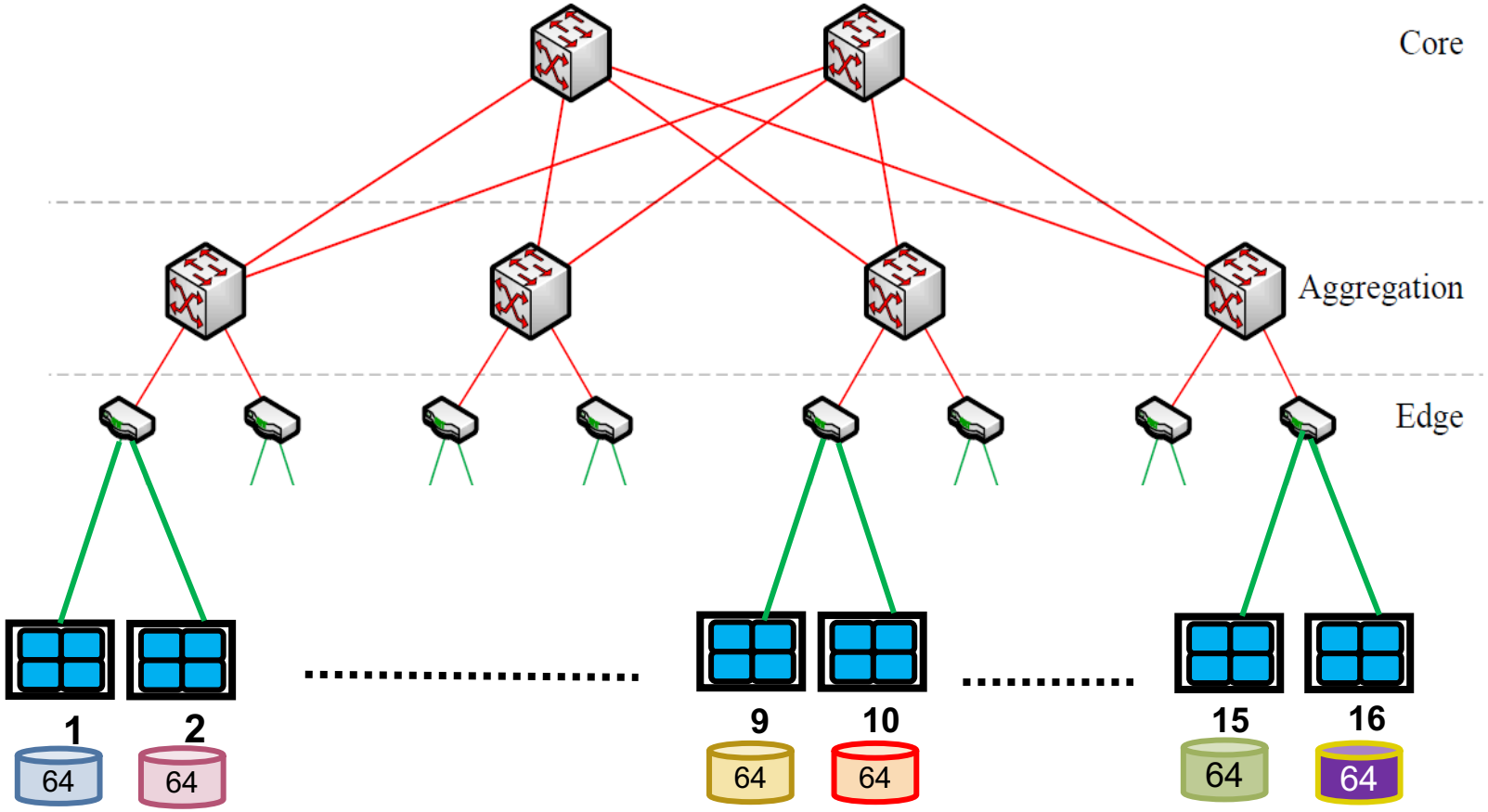
Data Parallel

Distributed SGD Exploits data Parallelism

- Entire model on each processor
- Distribute the SGD batch evenly across each processor

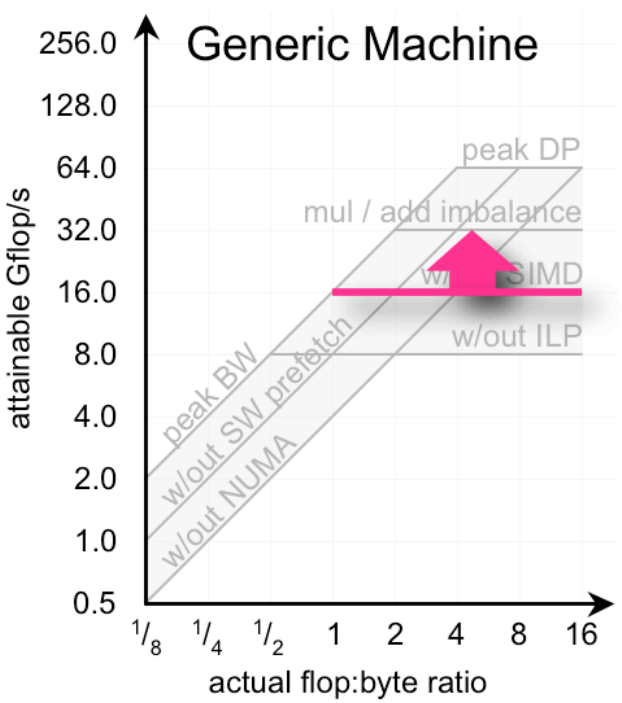
  (aka per-processor batch):
  - 1024 batch distributed over 16 PEs
  - Batch of 64/PE
- Communicate gradient updates all-to-all





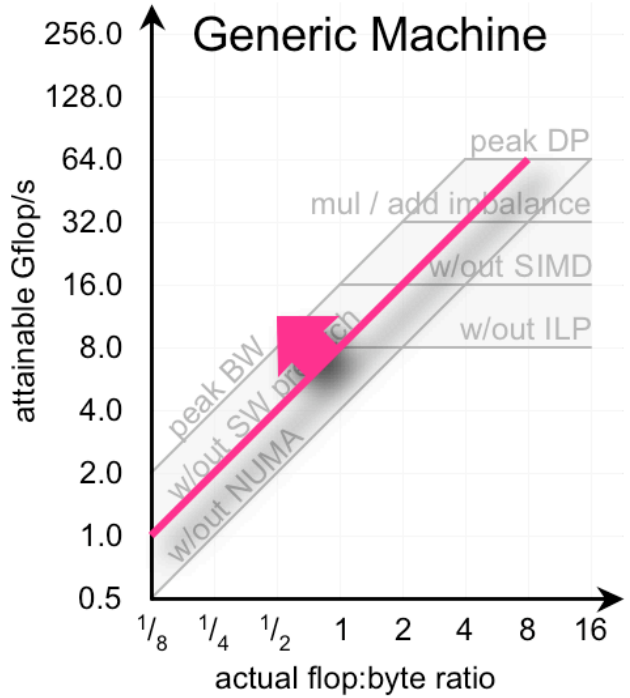
(Data center picture from)
Mohammad Al-Fares, Alexander Loukissas, Amin Vahdat,
"A scalable, commodity data center network architecture" ACM SIGCOMM 2008 conference on Data communication.

Roofline Model For Distributed Training



(a) maximizing in-core performance

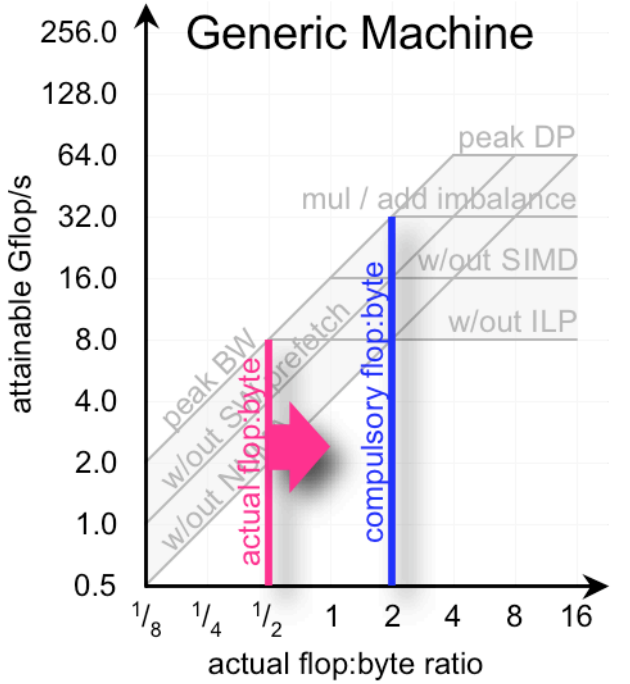
Use accelerators



(b) maximizing bandwidth

Interconnect network BW

DRAM BW



(c) minimizing traffic

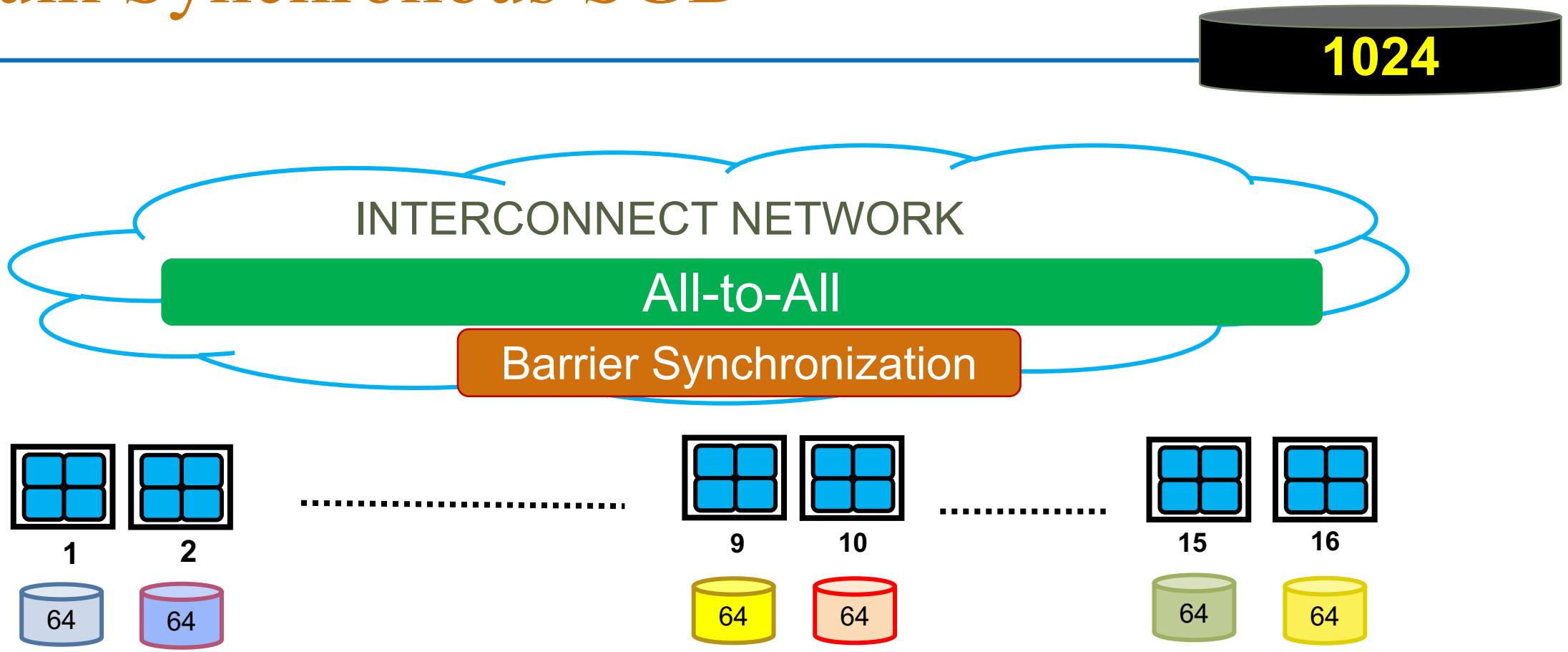
Node locality

Exploit sparsity

Less comm / synch

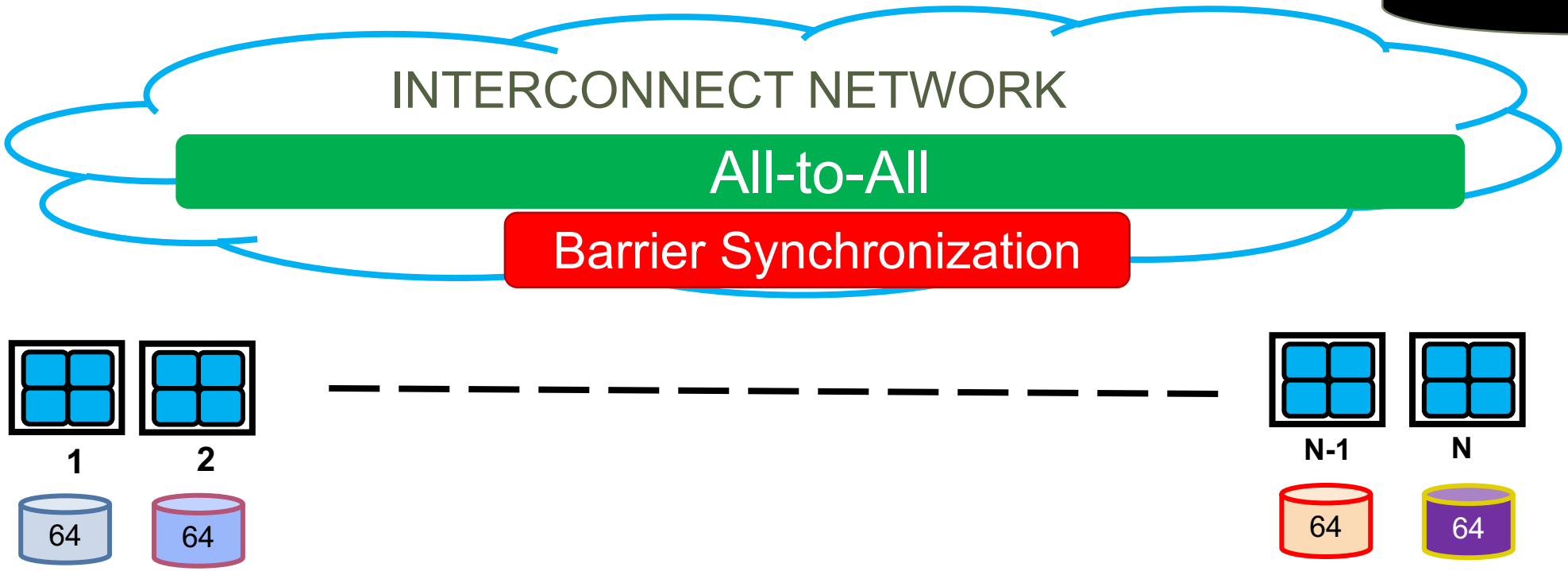
Samuel Williams, Andrew Waterman, and David Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures". Communications of the ACM April 2008.

Bulk Synchronous SGD



- Synchronization bottleneck
- Various approach to ameliorate this but the problem is inherent

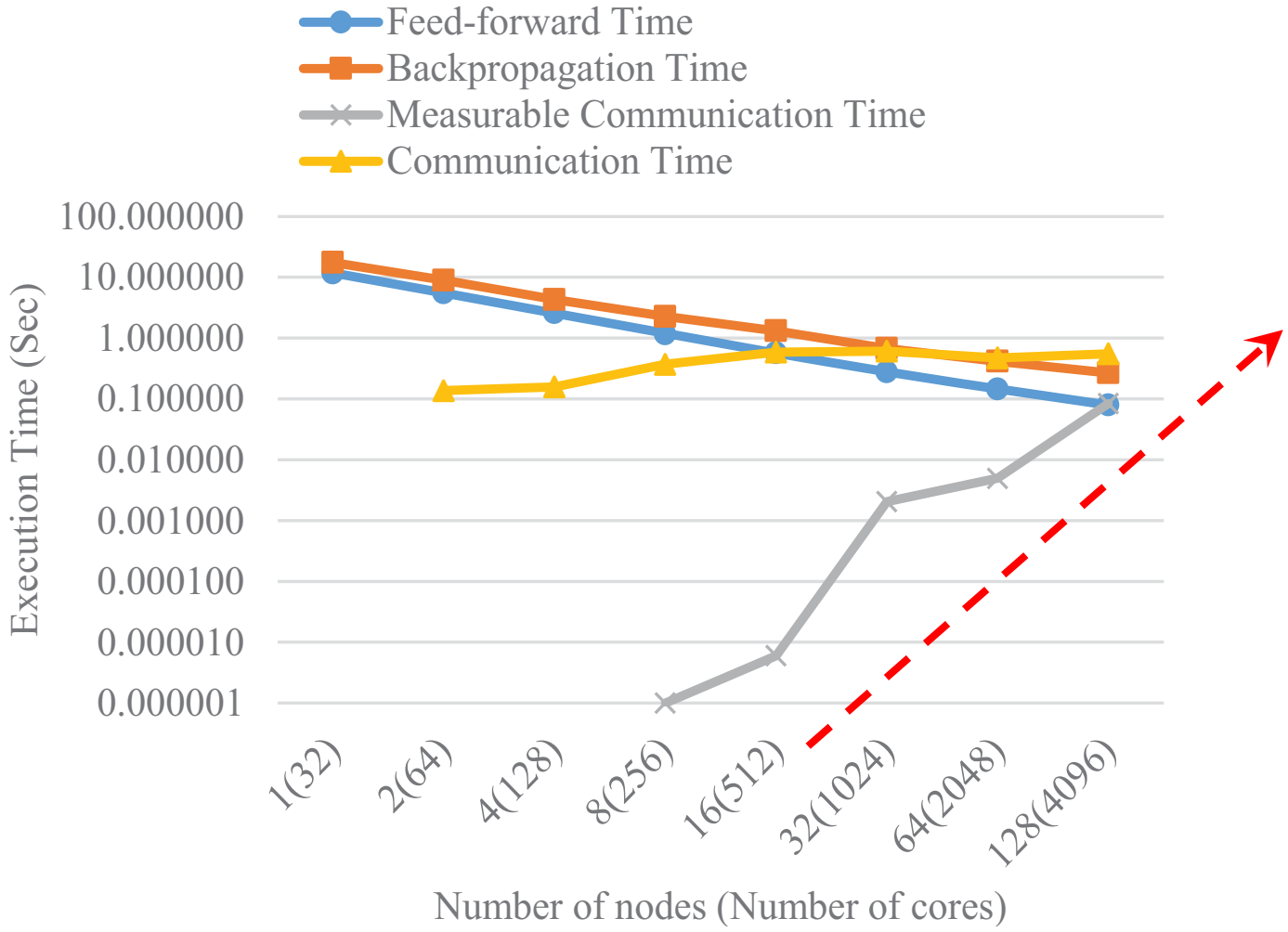
We **MUST** Increase The Batch size



- If we want to keep scaling synchronous SGD then we have to keep increasing the batch size
- N=256 -> Batch Size=**16K**

Overlap Communication and Compute To Hide Network Latency

- Breakdown for VGG
- Minibatch 256

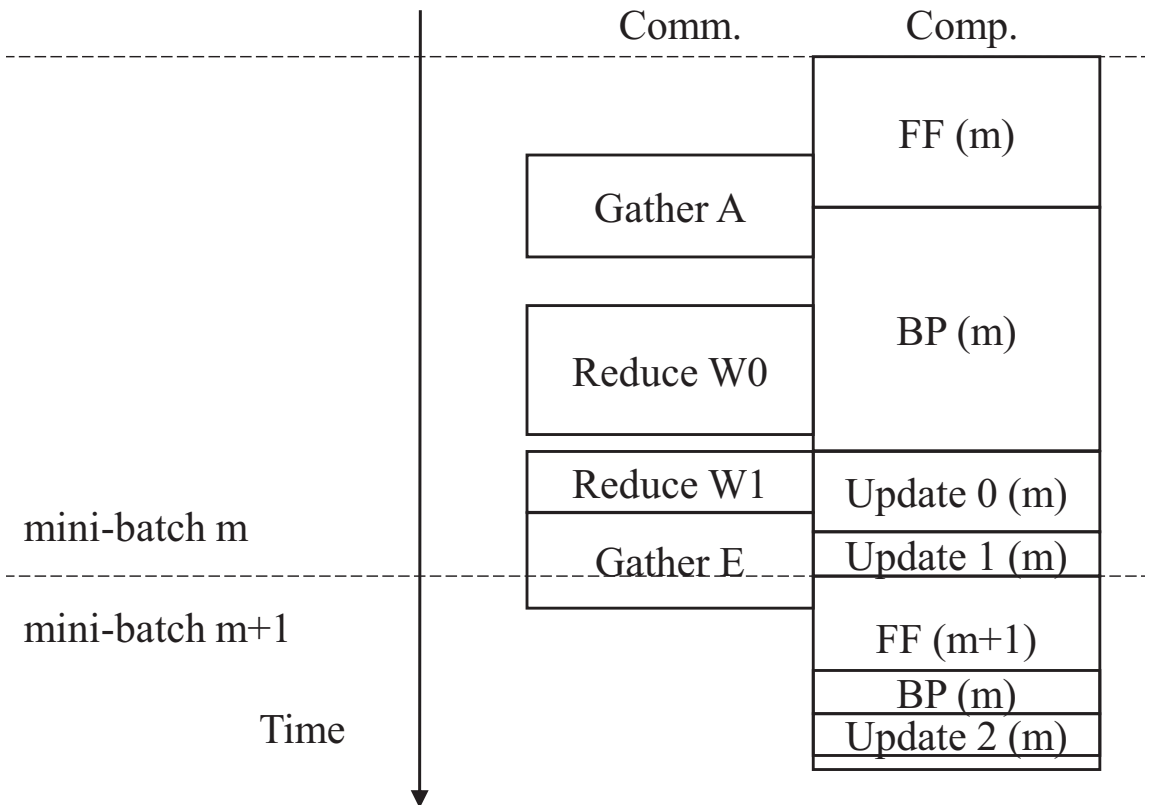


Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao, “Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication “, IEEE 24th International Conference on High Performance Computing, 2017.

Das, Dipankar, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. "Distributed deep learning using synchronous stochastic gradient descent." *arXiv preprint arXiv:1602.06709* (2016).

Hiding Communication Latencies

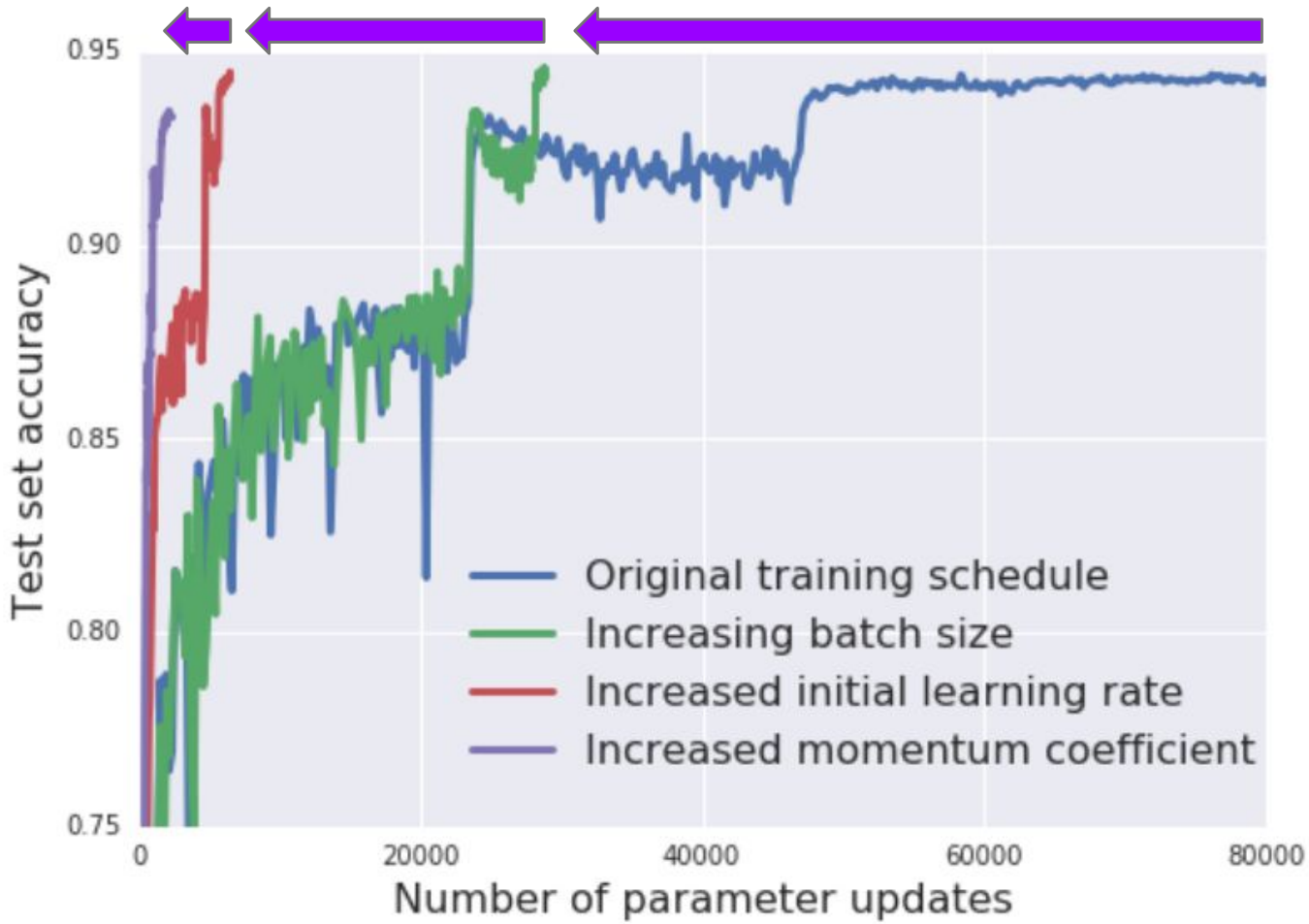
- Time-flow chart with maximized overlap
- Linear speedup
  - All communications are hidden behind the computation
- Gradient computation and parameter update at the first fully-connected layers are delayed to the next mini-batch training



Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao, "Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication ", IEEE 24th International Conference on High Performance Computing, 2017.

Don't Decay the Learning Rate, Increase the Batch Size

- The key difficulty
  - Numerical optimization
- Decrease # of parameter updates
- Batch Size vs. learning rate
- CIFAR-10 & Imagenet
- Not general enough

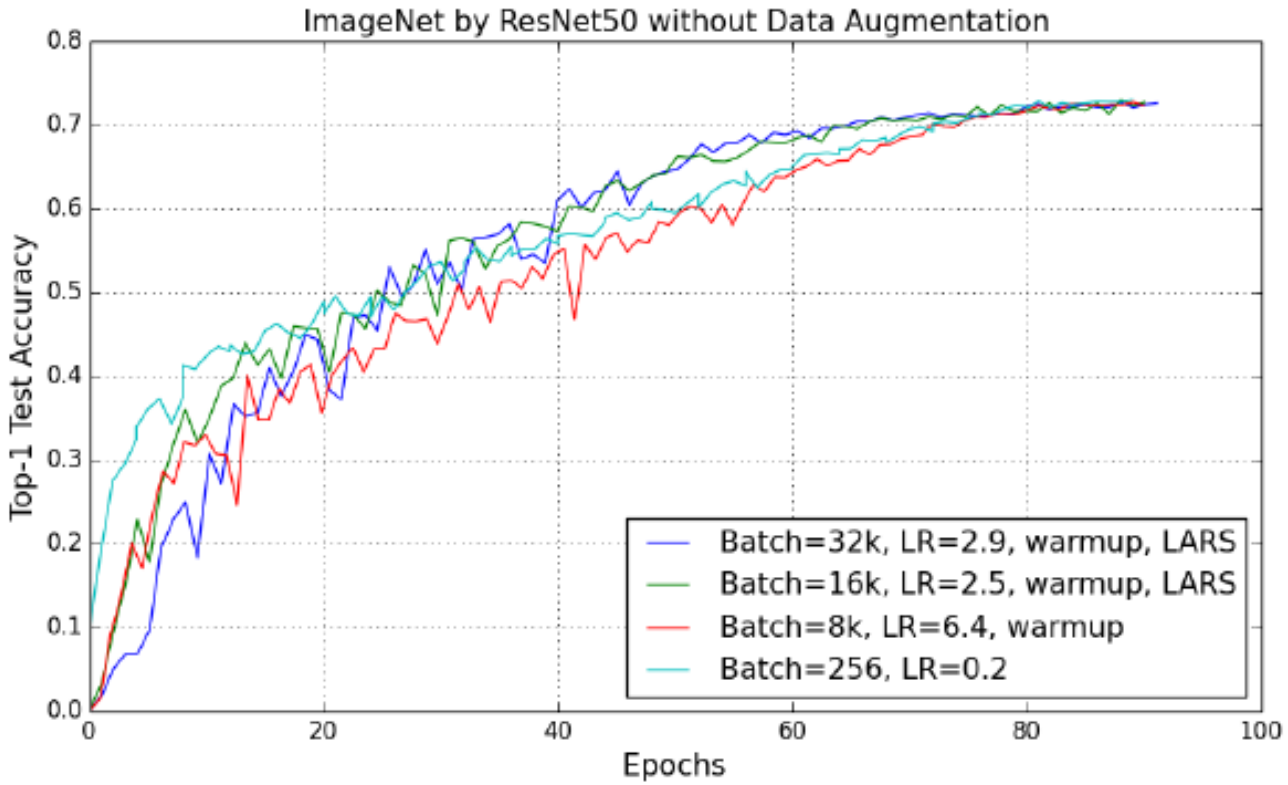


Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, Quoc V. Le," Don't Decay the Learning Rate, Increase the Batch Size," ICLR 2018. https://arxiv.org/abs/1711.00489

Layer-Wise Adaptive Learning Rate

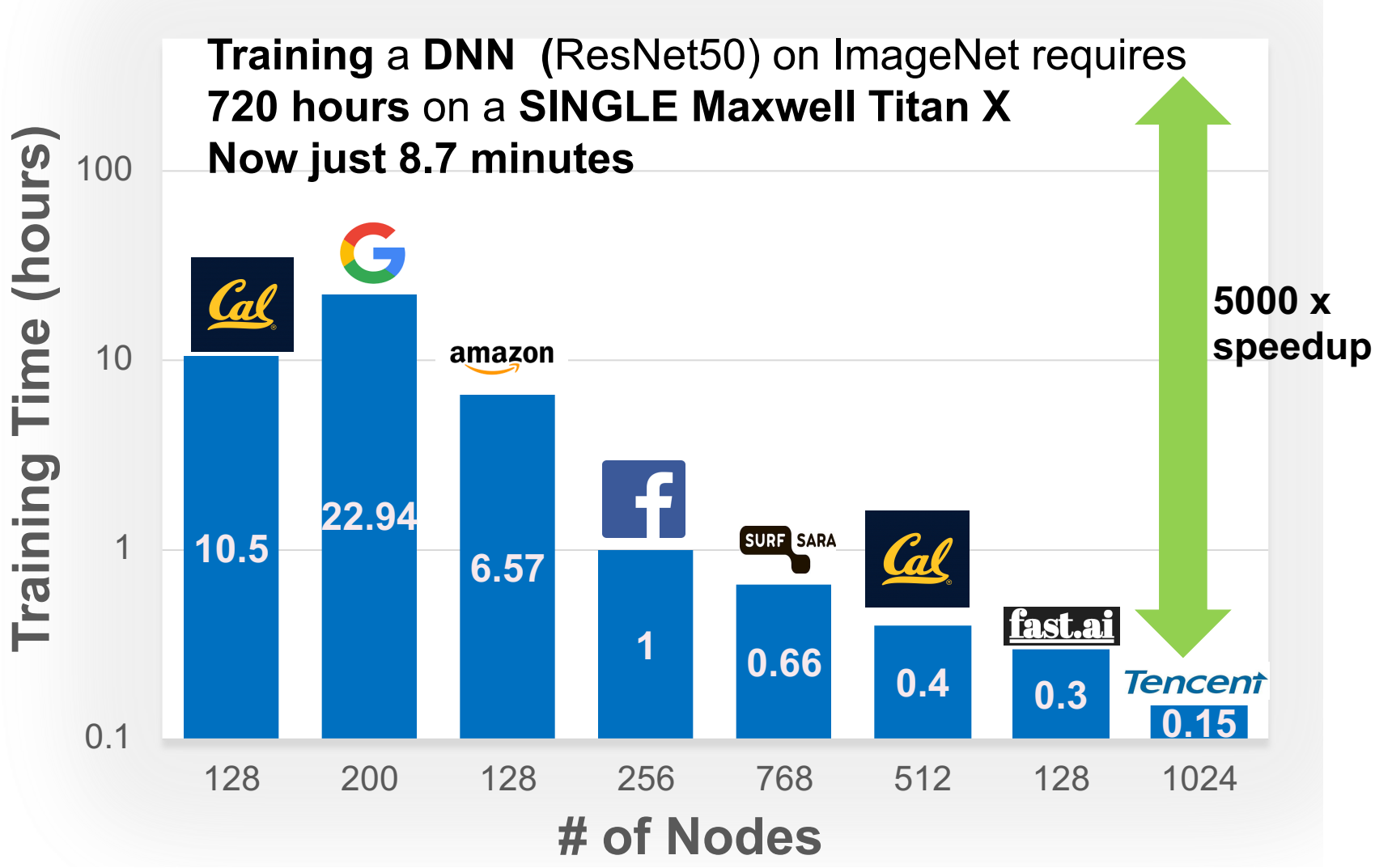
RESNET-50 WITH LARS: B→ 32K

- LARS:
  - Adapts the learning rate for each layer
  - Scaling to
    B=8K for Alexnet
    B=32K for Resnet-50.
- Above 32K without accuracy loss is still open problem

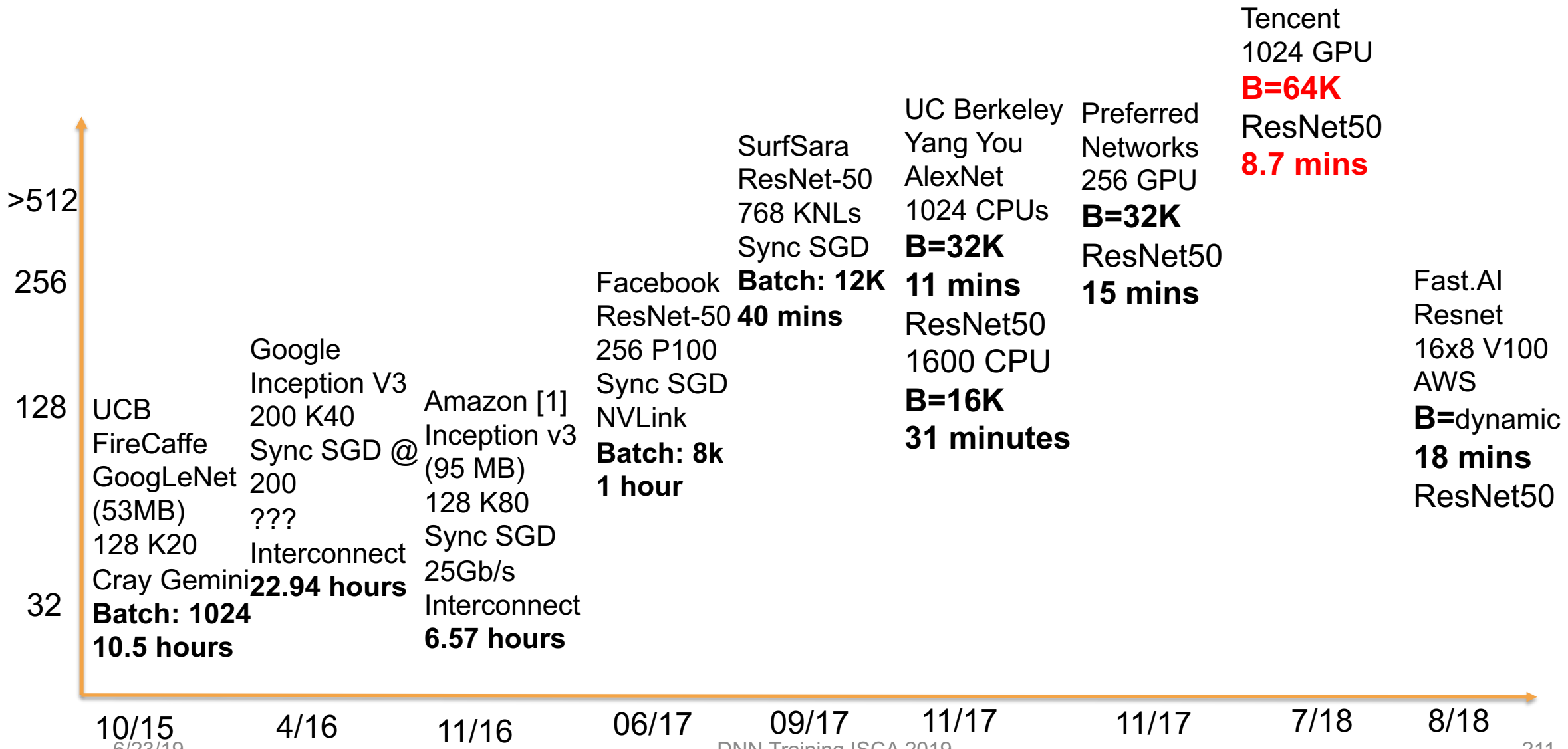


- You, Yang, Igor Gitman, and Boris Ginsburg. "Scaling SGD Batch Size to 32K for ImageNet Training." *arXiv preprint arXiv:1708.03888* (2017).
- You, Yang, Zhao Zhang, C. Hsieh, James Demmel, and Kurt Keutzer. "ImageNet training in minutes." ICPP 2018. https://arxiv.org/abs/1709.05011

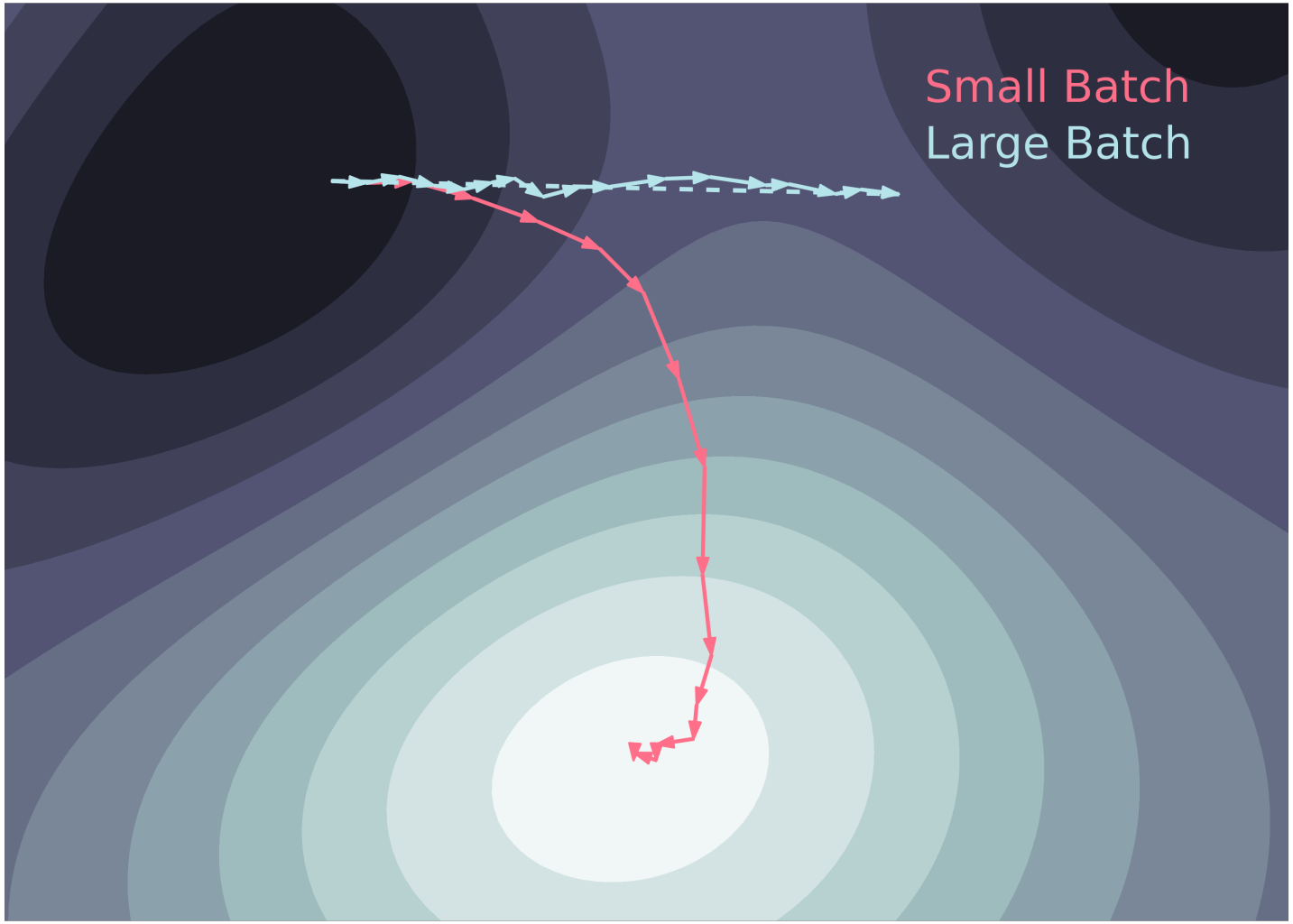
Recent Progress: August 2018



Pushing Synchronous SGD Farther

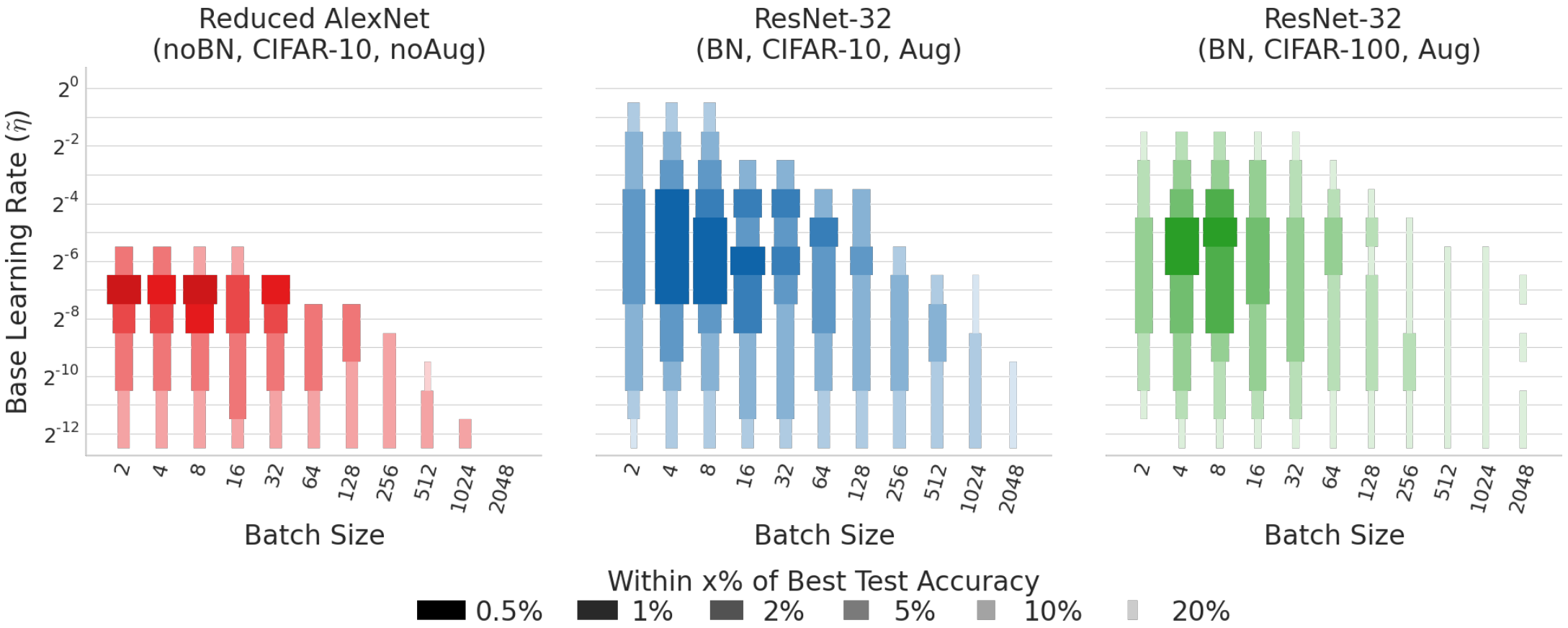


Large Batches Miss The Minimum



Dominic Masters, Carlo Luschi, “Revisiting Small Batch Training for Deep Neural Networks” arXiv:1804.07612 2017.
https://www.graphcore.ai/posts/revisiting-small-batch-training-for-deep-neural-networks

LARGE BATCHs Miss Test Accuracy



Dominic Masters, Carlo Luschi, “Revisiting Small Batch Training for Deep Neural Networks” arXiv:1804.07612 2017. https://arxiv.org/abs/1804.07612

Scaling Training With LARGE BATCHEs

Cons

- Constrained approach: Need to employ large batches to capture efficiency
- May not achieve target accuracy
- Only demonstrated on CNNs
- More prone to adversarial attacks[1]

Pros

- Robust: Relatively less hyperparameter tuning
- Sequential consistency
- Good infrastructural support from HPC frameworks (MPI)
- Fault tolerance is practically handled by snapshots and rollback otherwise

[1] https://arxiv.org/pdf/1802.08241.pdf

Asynchronous SGD

- Hogwild![1]
  - Asynchronous on shared memory
- Distributed asynchronous SGD
  - Googles training (Dist belief)[2]
  - Accuracy has degraded at large scaling >32 [3,4]
- Deep gradient compression[5]

[1]- Feng Niu, Benjamin Recht, Christopher R´e and Stephen J. Wright, “Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”, NIPS 2011. https://people.eecs.berkeley.edu/~brecht/papers/hogwildTR.pdf

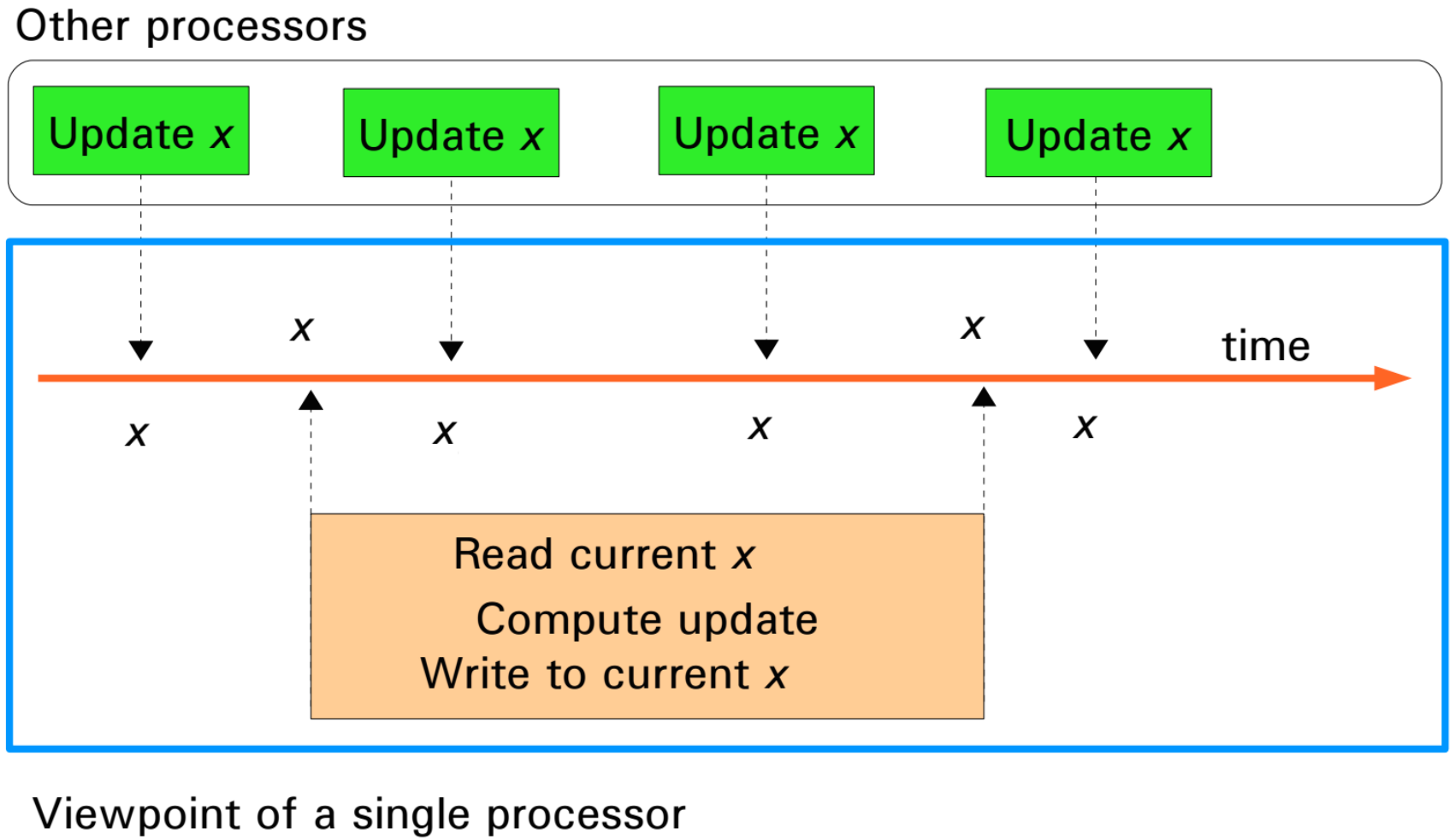
[2] - Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng , “Large Scale Distributed Deep Networks”. https://ai.google/research/pubs/pub40565

[3]- Zhang, Sixin, Anna E. Choromanska, and Yann LeCun. "Deep learning with elastic averaging SGD." In Advances in Neural Information Processing Systems, pp. 685-693. 2015.

[4]- Jin, Peter H., Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. "How to scale distributed deep learning?." arXiv preprint arXiv:1611.04581 (2016). NIPS MLSys 2017.

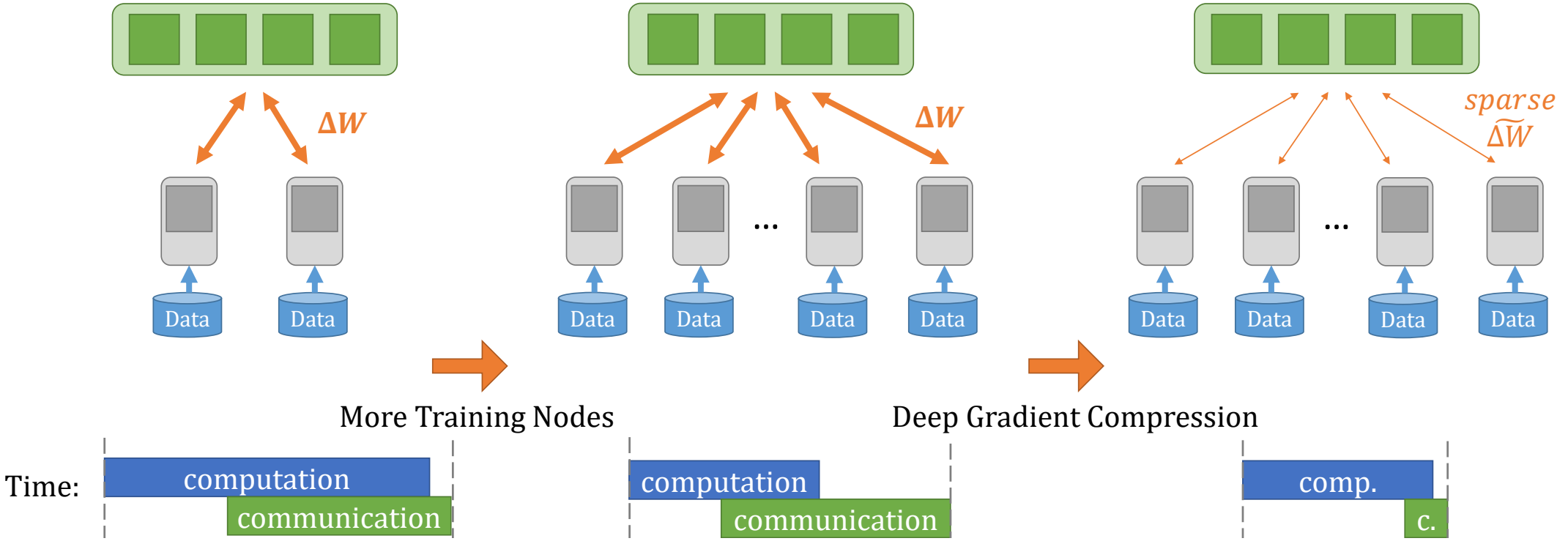
[5]- Yujun Lin, Song Han, Huizi Mao, Yu Wang, William J. Dally, “Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training” ICLR 2018. https://arxiv.org/abs/1712.01887

HOGWILD!



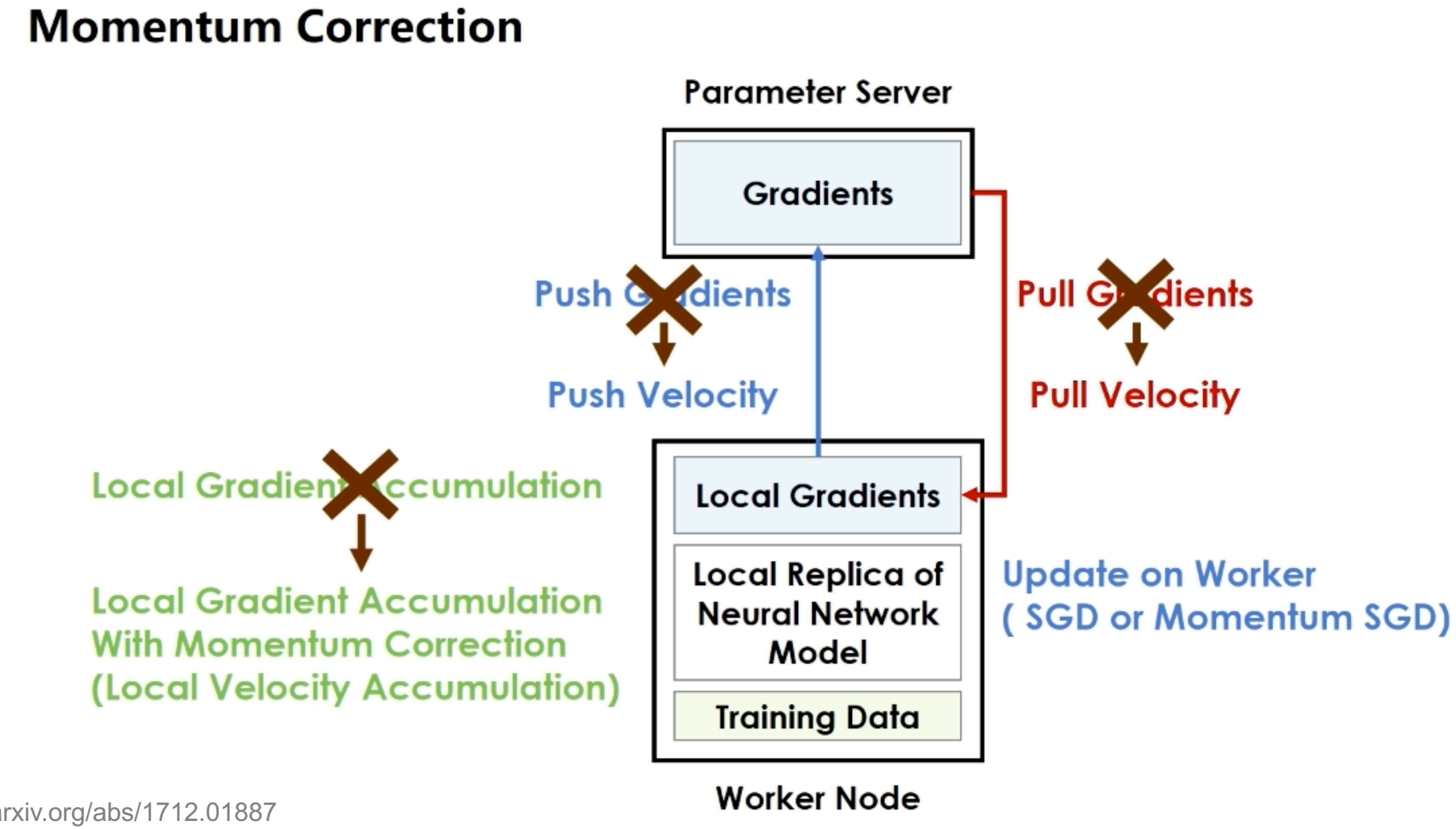
Feng Niu, Benjamin Recht, Christopher R´e and Stephen J. Wright, “Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”, NIPS 2011. https://people.eecs.berkeley.edu/~brecht/papers/hogwildTR.pdf

Deep Gradient Compression



https://arxiv.org/abs/1712.01887

Deep Gradient Compression



https://arxiv.org/abs/1712.01887

Asynchronous and Synchronous SGD

Asynchronous SGD

- Parameters are updated by parallel workers asynchronously
- Updates may be delayed or "lost'
- Number of parameter servers manage parameters

Pros

- Used for very large models that do not fit in one machine
- Can scale to very large clusters with slow network

Cons

- Staleness of updates sensitive to network characteristics

Synchronous SGD

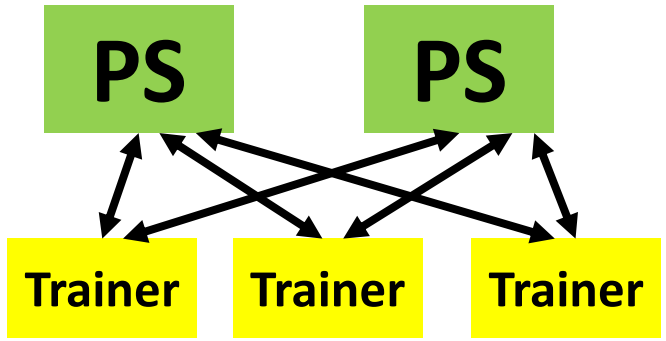
- Workers sync parameter gradients after each iteration
- Updates are always in sync
- Computation is function of the total batch size only

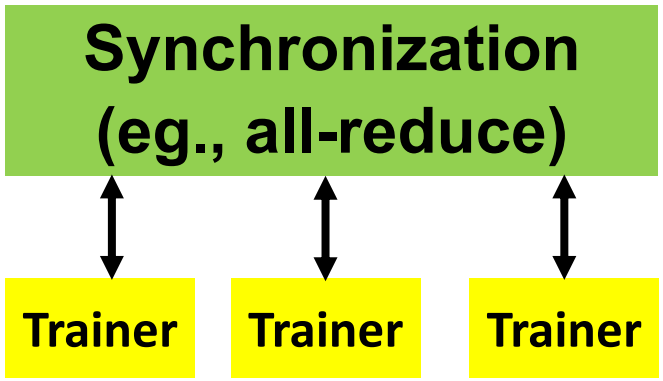
Pros

- Reproducible results up to machine precision
- Scales well if large batch size is used

Cons

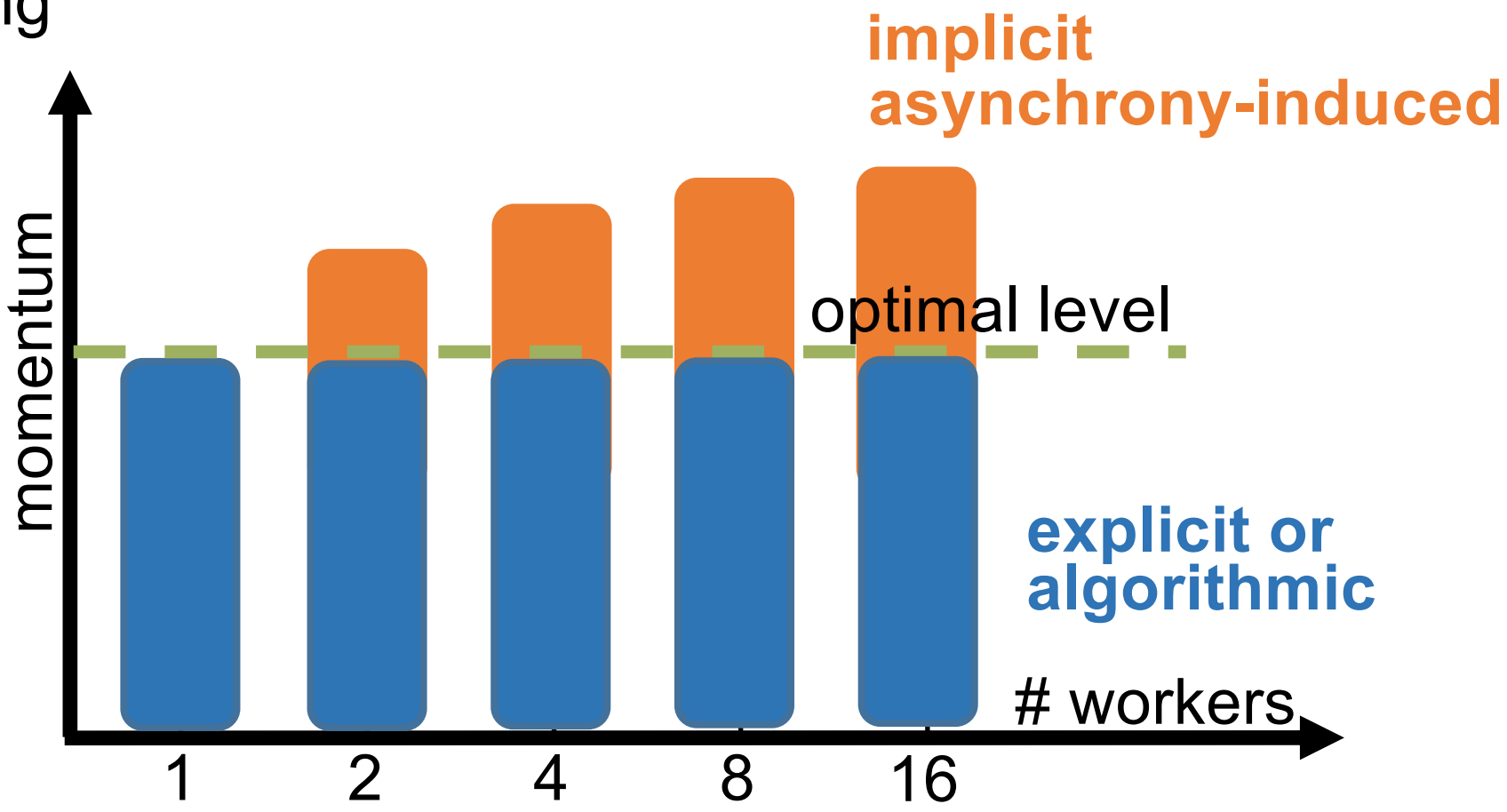
- Does not scale well to large clusters





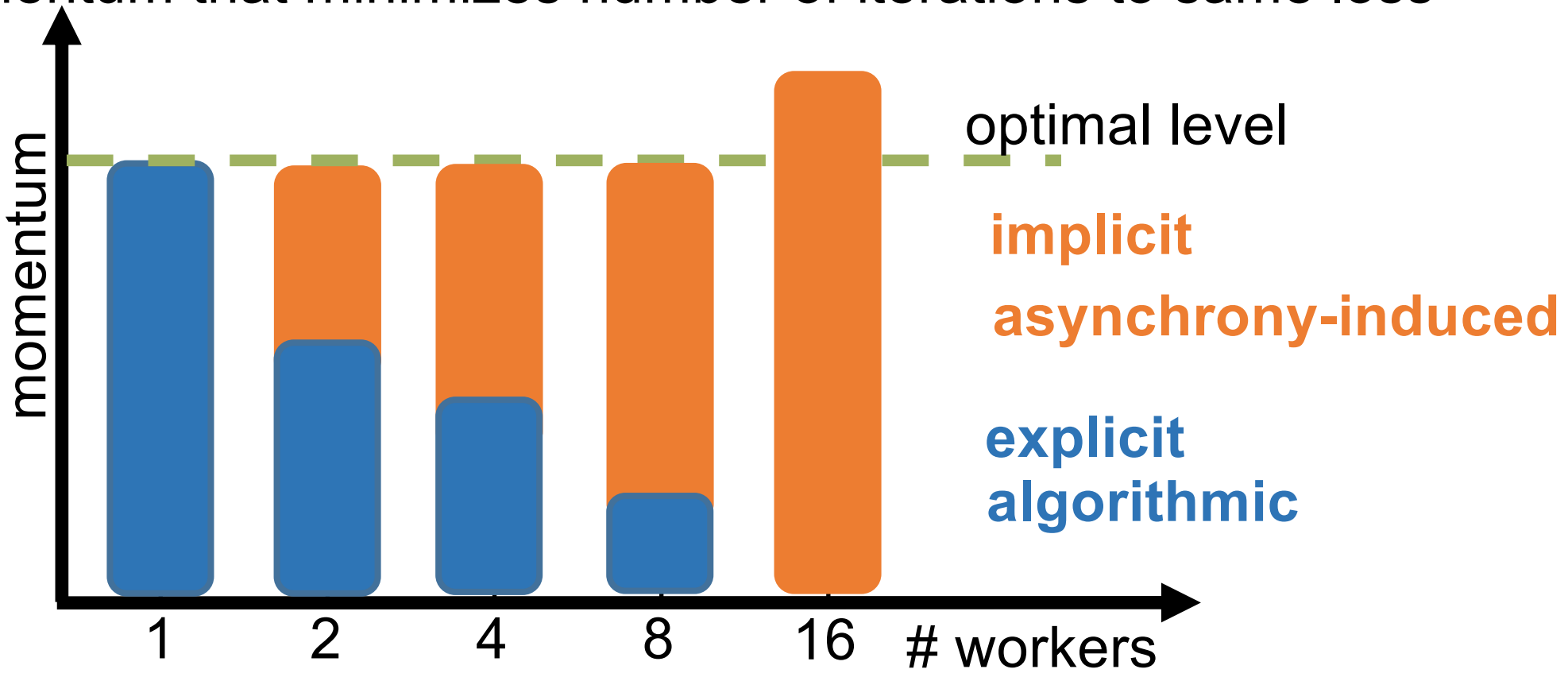
Asynchrony Begets Momentum

- No Tuning



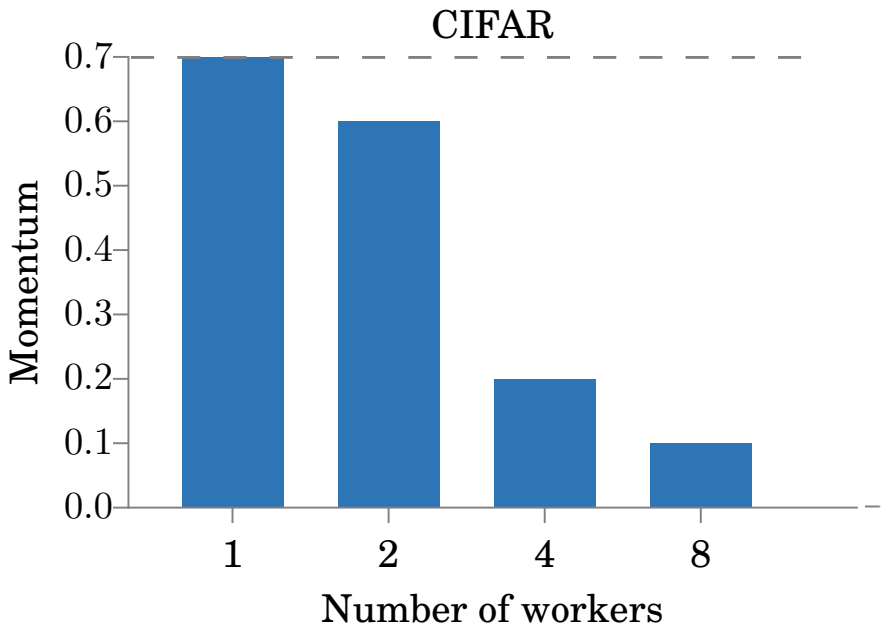
Asynchrony Begets Momentum

- Tuning
  Pick momentum that minimizes number of iterations to same loss



Validation

- Experiment



Momentum grid: 0.1, 0.2, ..., 0.8, 0.9

LR grid: 0.01, 0.005, 0.001

8 × g2.8xlarge

33 × c4.4xlarge

Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

ML Benchmarks

Types of ML Benchmarks: Quality

- Quality Benchmarks: measure the accuracy of networks
  - **ImageNet**: Fei-Fei Li 2012, 1.2M image standard dataset for measuring classification accuracy as well as a yearly competition
    - Revolutionized image classification
  - **Mnist**: hand written digit dataset
  - **CIFAR**: small image classification
  - Many more: https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research#Object_detection_and_recognition
    - Image data
    - Text data
    - Sound data
    - Signal data
    - Physical data
    - Biological data
    - Anomaly data
    - Question Answering data
    - Multivariate data

Types of ML Benchmarks: Performance

- Performance Benchmarks: measure the speed of network execution
  - Frameworks and hardware are both measured
    - Inference: throughput, latency
    - Training: throughput, time to accuracy

Benchmark	Breadth of Types	Accuracy requirement	Support	Submission rules and publication of results
Conv bench	1	No	Individual	No
DeepBench	Kernels	No	Corporate	No
DAWN Bench	2	Yes	University	Minimal
Fathom	8	No	University	No
MLPerf	7	Yes	Industry	Extensive

Lies, Damn Lies, ….

Common Benchmark cheats

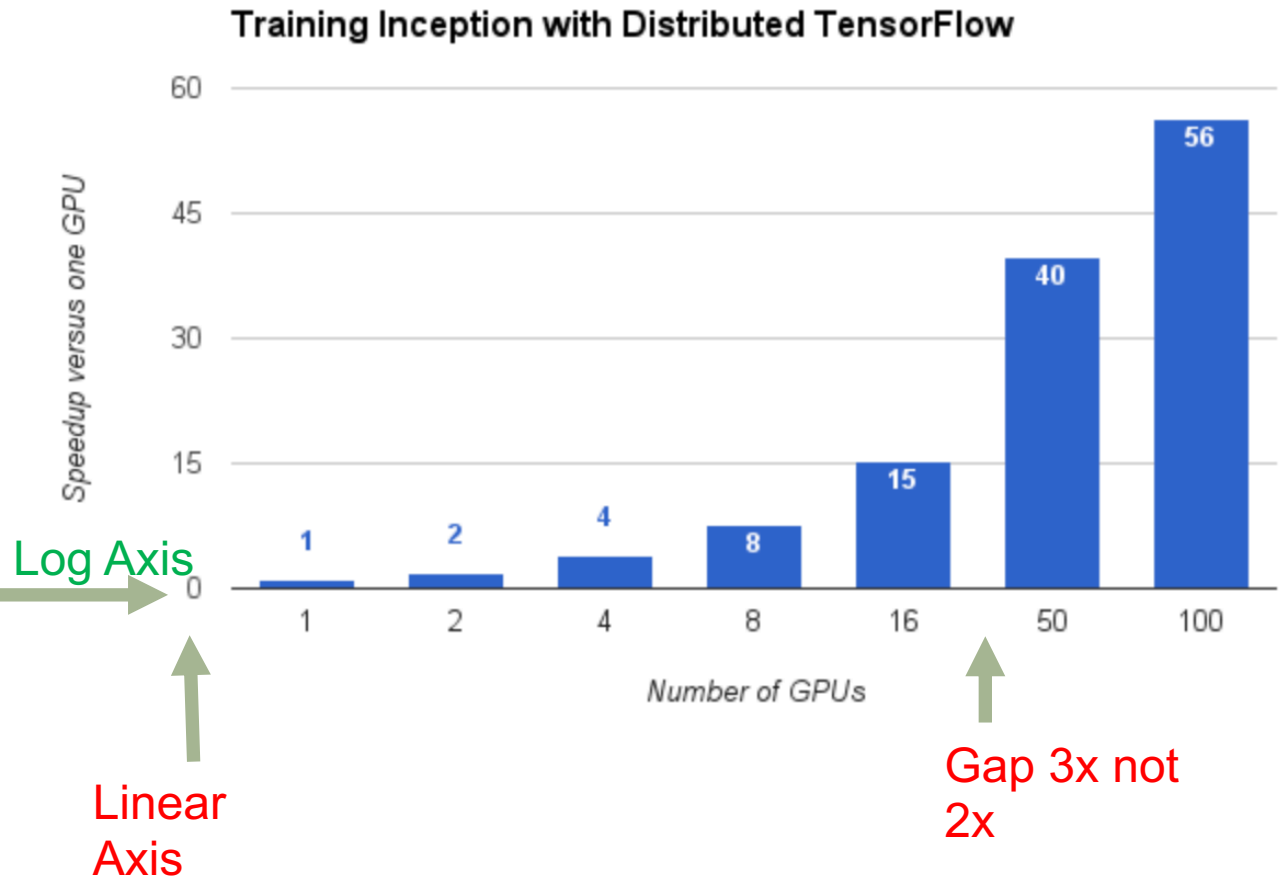
- Choosing thresholds that the comparison system cannot achieve
  - Latency cutoff of 7ms, comparison system has latency floor of 8 ms
- Cherry picking results
  - Run the benchmark 20 times and publish the best result
- Normalize to a metric of advantage even if scalability doesn't hold
  - If you don't win on performance compare performance/W, performance/$, performance/lb …

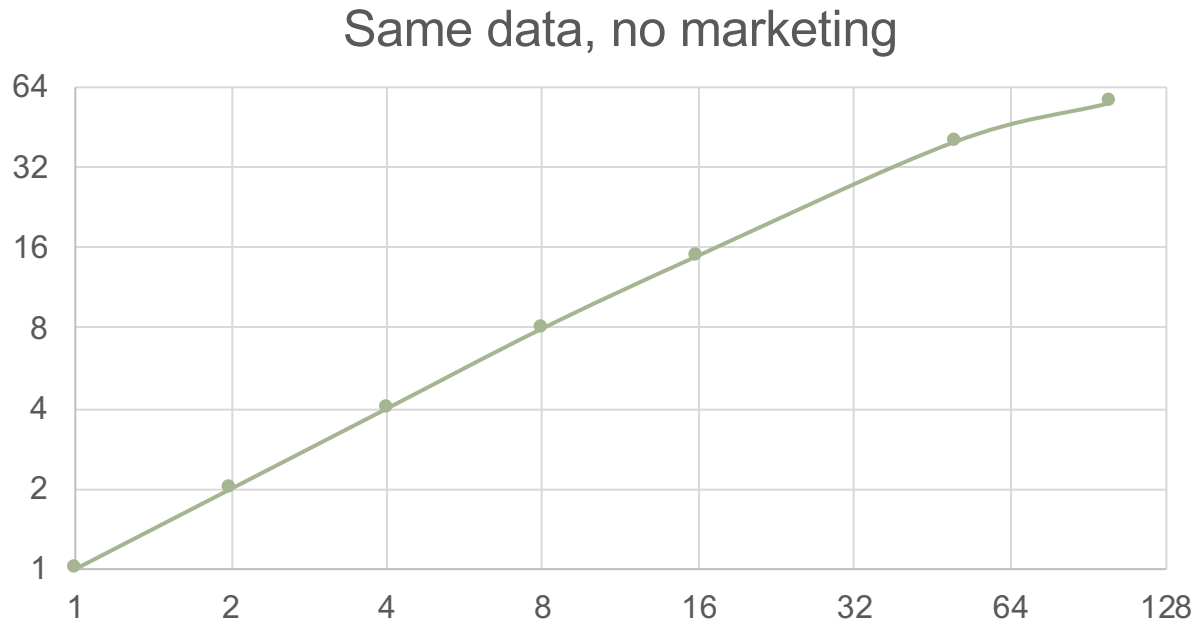
Source: Gary Lauterbach Cerebras

Cheats unique to ML

- ML is statistical compute, results are subject to variation
- Do massive search to find fastest training on benchmark data
  - Hyper-parameters: learning rate, batch size …
  - Fine grain verification to cherry pick first accuracy above threshold
  - Initialization seeds
- Techniques just ”game” for the benchmark data set and do not generalize

Misleading presentation





Derek Murray, "Announcing TensorFlow 0.8 – now with distributed computing support!", April 2016. https://ai.googleblog.com/2016/04/announcing-tensorflow-08-now-with.html

Source: Gary Lauterbach Cerebras

MLPERF

- Tutorial from Yesterday
- https://sites.google.com/g.harvard.edu/mlperf-bench/home

Training

1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Training Accelerator Systems

Volta V100 GPU

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

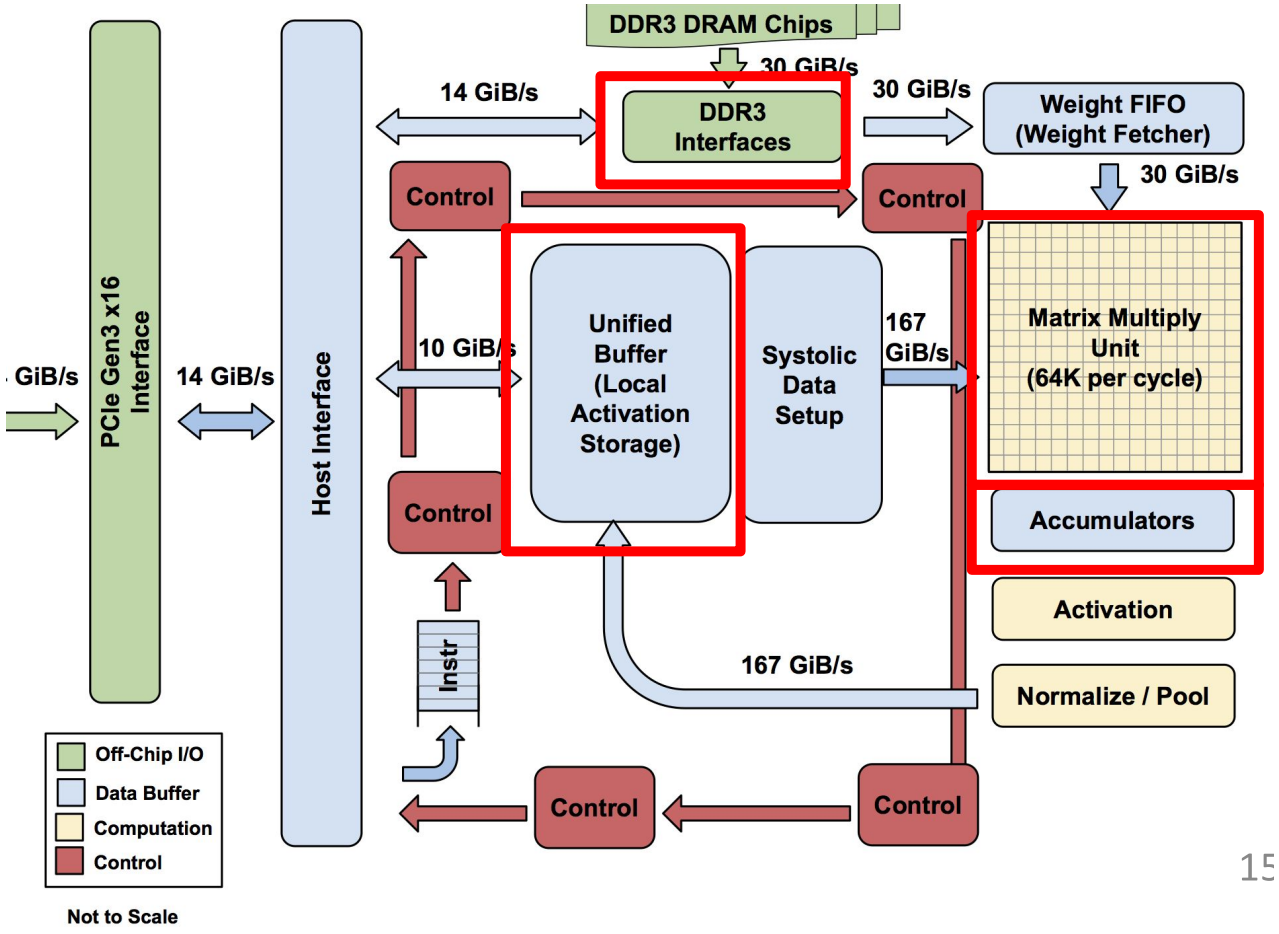
16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

*full GV100 chip contains 84 SMs

https://www.servethehome.com/nvidia-v100-volta-update-hot-chips-2017/

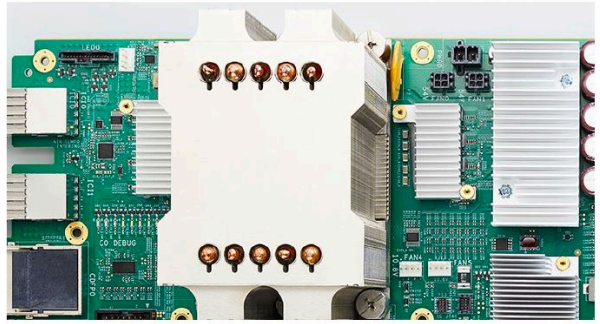
TPU V1: High-level Chip Architecture

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
  - Systolic array
- 700 MHz clock rate
- Peak: 92T operations/second
  - 65,536 * 2 * 700M
- 4 MB of on-chip Accumulator memory
- 24 MB of on-chip Unified Buffer (activation memory)
- Two 2133MHz DDR3 DRAM channels
- 8 GB of off-chip weight DRAM memory
- vs GPU and CPU
  - >25X as many MACs vs GPU
  - >100X as many MACs vs CPU

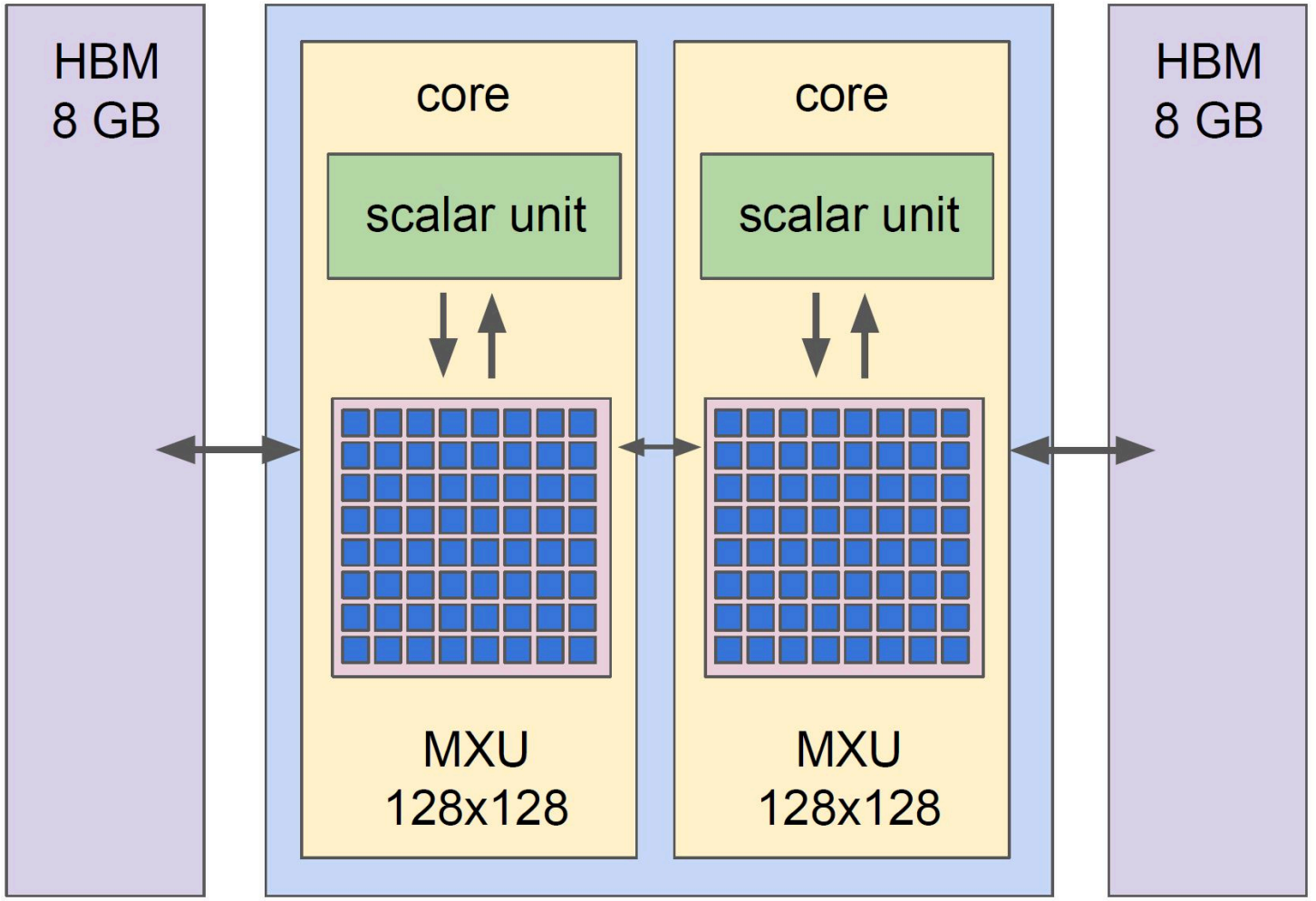


TPU V2 for Training

TPUv2 Chip

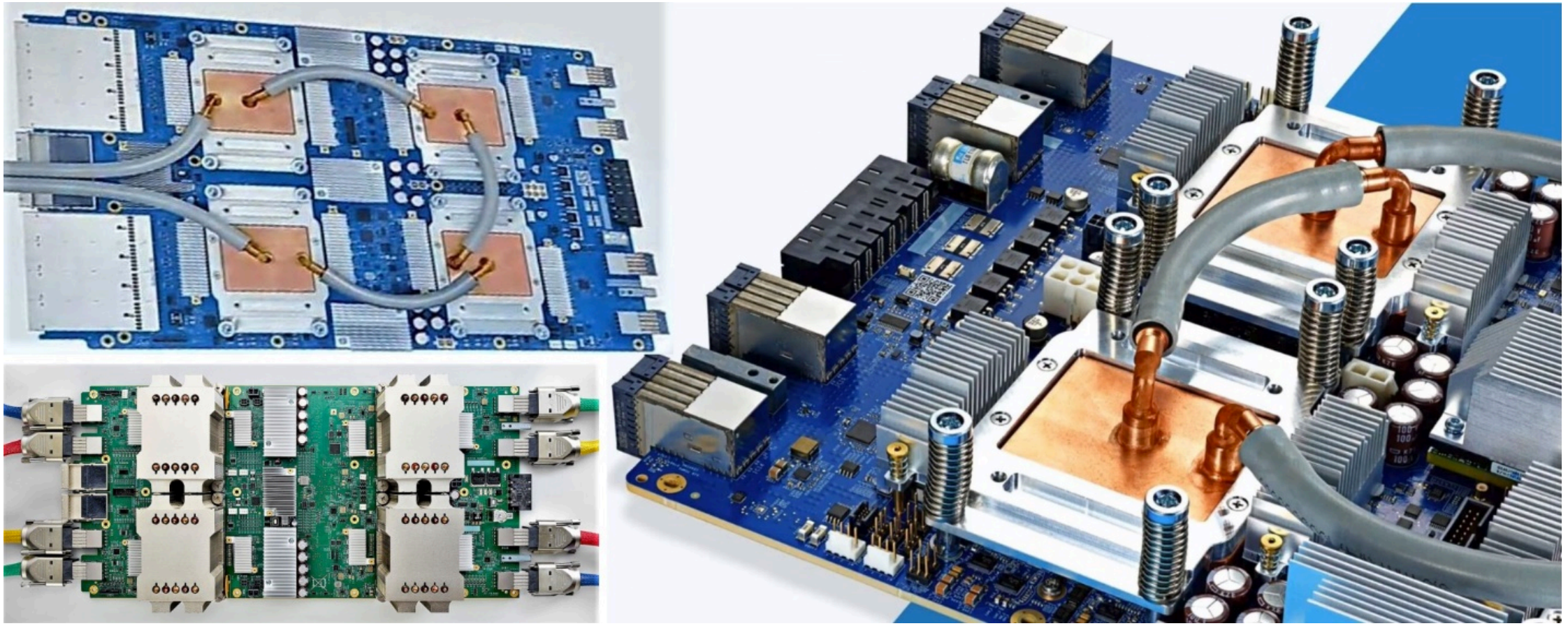


- 16 GB of HBM
- 600 GB/s mem BW
- Scalar unit: 32b float
- MXU: 32b float accumulation but reduced precision for multipliers
- 45 TFLOPS



https://www.matroid.com/scaledml/2018/jeff.pdf

TPU V3

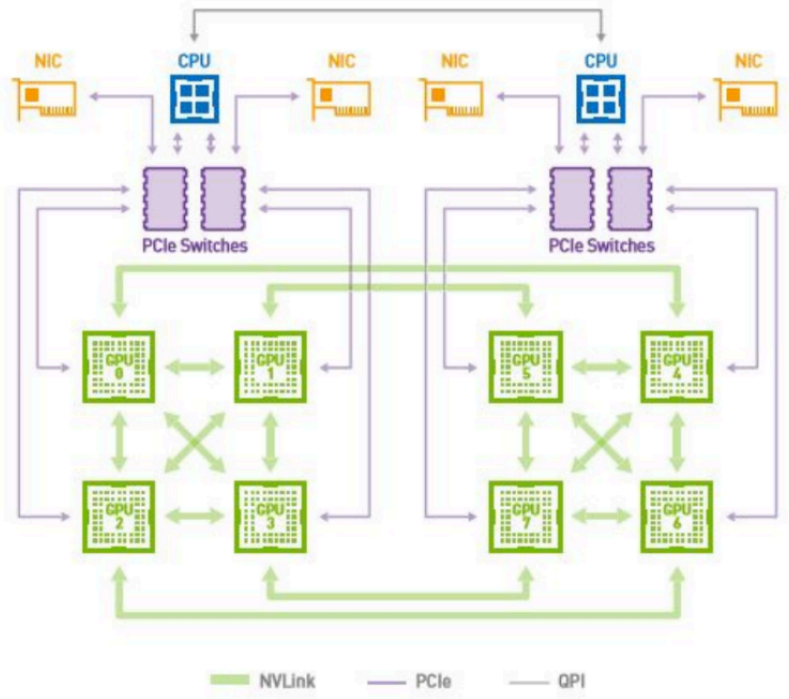


https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/

Nvidia DGX-1

NVIDIA DGX-1

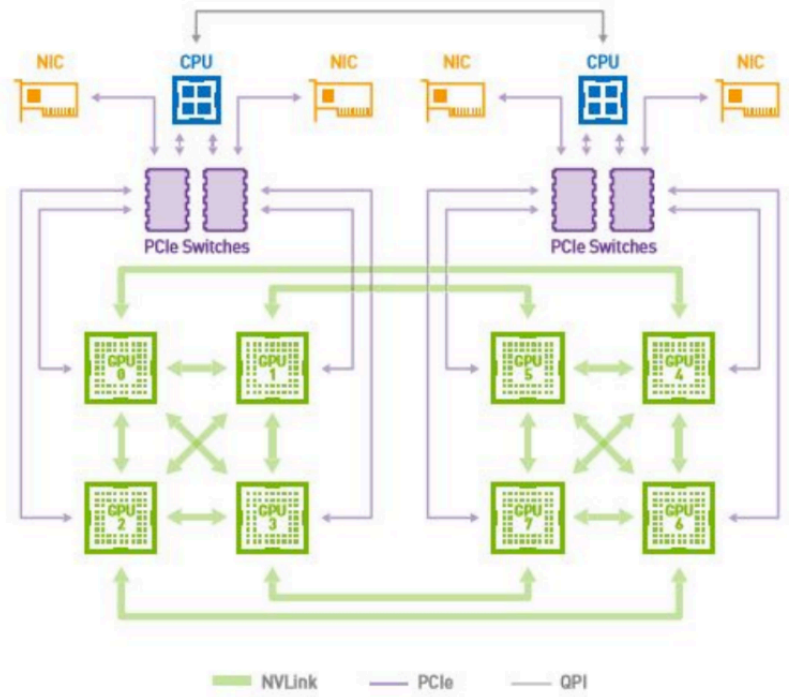
- V100: GPU with proprietary interfaces
- Blocking internal interconnect (Hyper cube mesh)
- **4** x 100Gb IB RDMA NICs
- Management & Scale-out Traffic bottleneck over PCIe



Habana AI

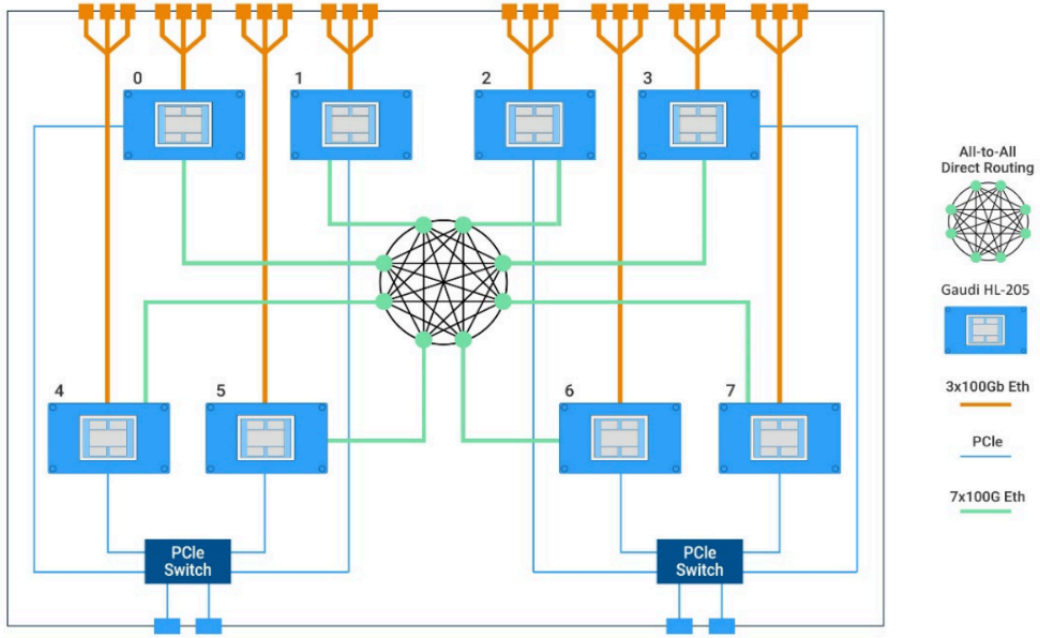
NVIDIA DGX-1

- V100: GPU with proprietary interfaces
- Blocking internal interconnect (Hyper cube mesh)
- **4** x 100Gb IB RDMA NICs
- Management & Scale-out Traffic bottleneck over PCIe



HLS-1

- Gaudi: on-chip compute + **Standard** RDMA RoCE for scale-up and scale-out
- Non-blocking, all-2-all internal interconnect
- **24** x 100GbE RDMA RoCE for scale-out
- Separate PCIe ports for external Host CPU traffic

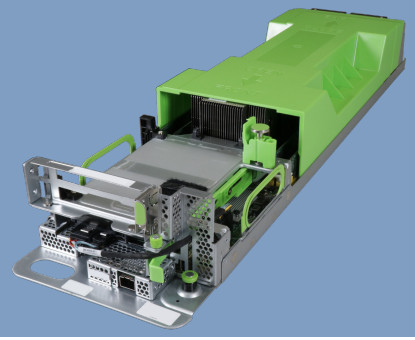


ML Hardware at Facebook

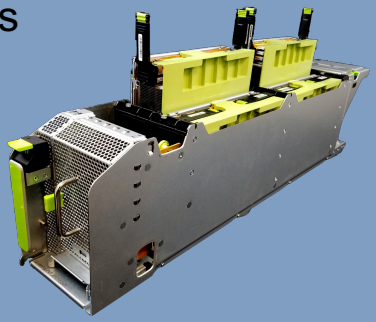
Bryce Canyon
- High density storage
- Storage-heavy tasks

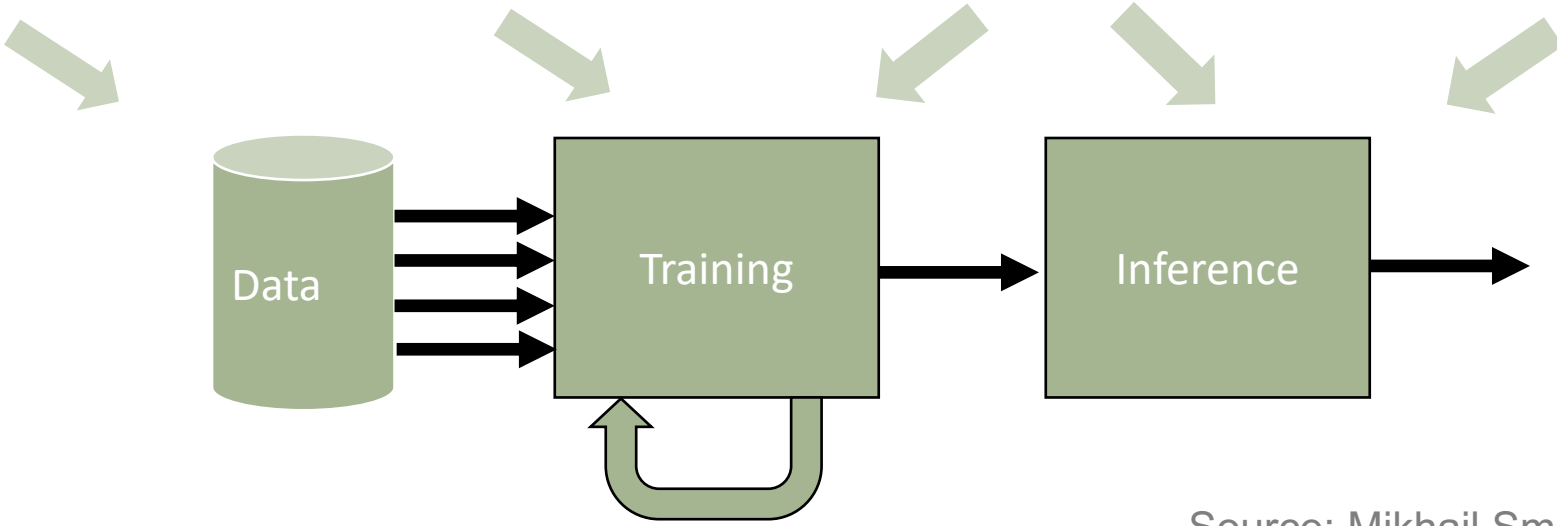
Big Basin
- 8 NVIDIA P100/V100's
- Very compute-heavy tasks

Tioga Pass
- Dual socket, many cores Skylake CPU
- Compute and mem-heavy tasks



Twin Lakes
- Single socket, fewer cores Skylake CPU
- Compute and low latency tasks





Limitation of Current Systems

- Dense Matrix Multiplication Centric
- Lack of Utilization
- Difficult to Program
- Imbalance between
  - Memory Hierarchy Capacity
  - Compute
  - Interconnect
- Lack of Support for Efficient Coarse Grain DataFlow
  - Streaming
  - Collective Communications
  - Model Parallelism

Training

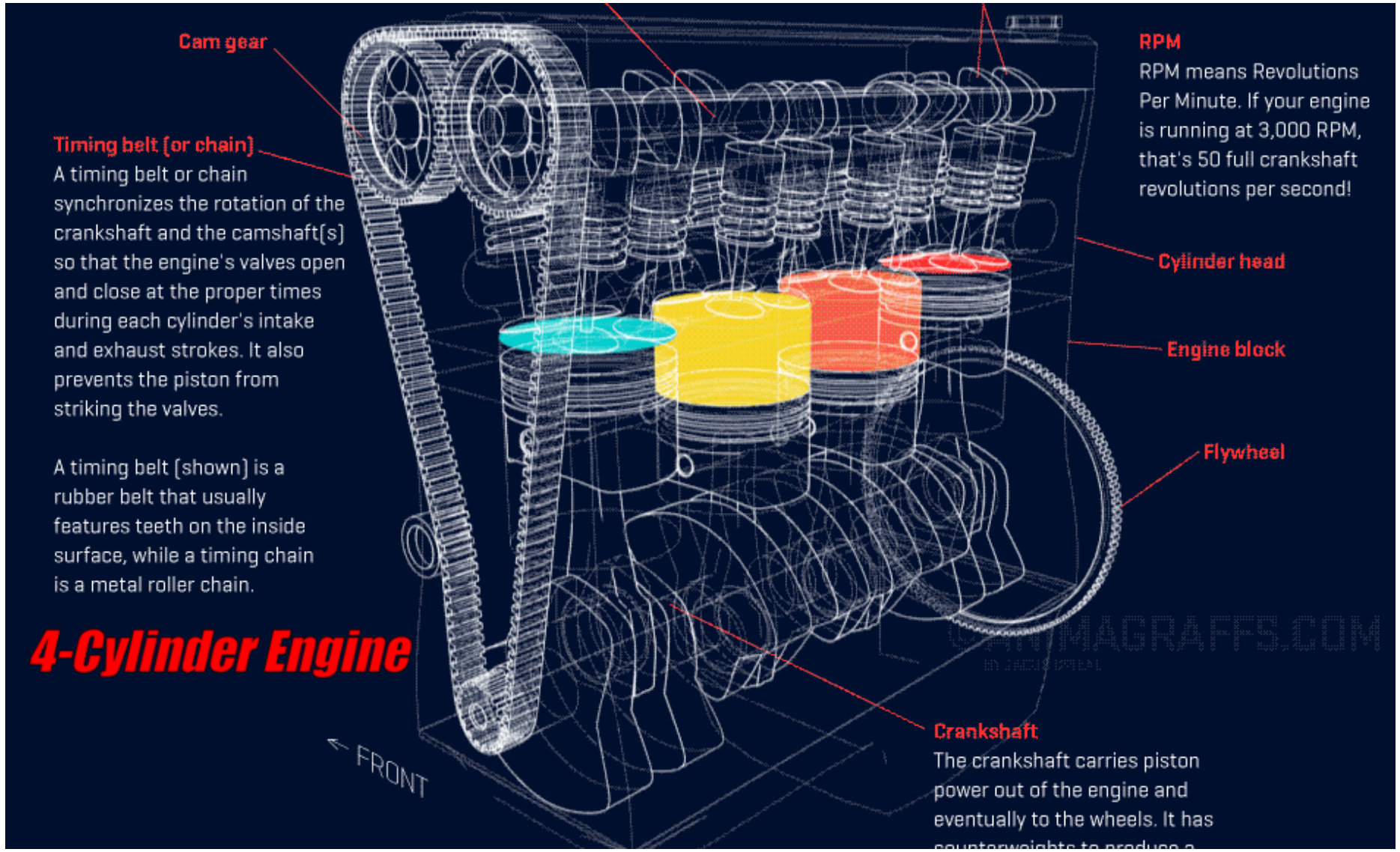
1. Fundamentals of training
2. Computation of training
3. Normalization
4. Low/Mixed Precision
5. Sparsity
6. Scaling training
7. Benchmarking
8. Training Accelerators
9. Conclusion

Stanford CS217:
Fall 2018 and Winter 2020

https://cs217.stanford.edu/

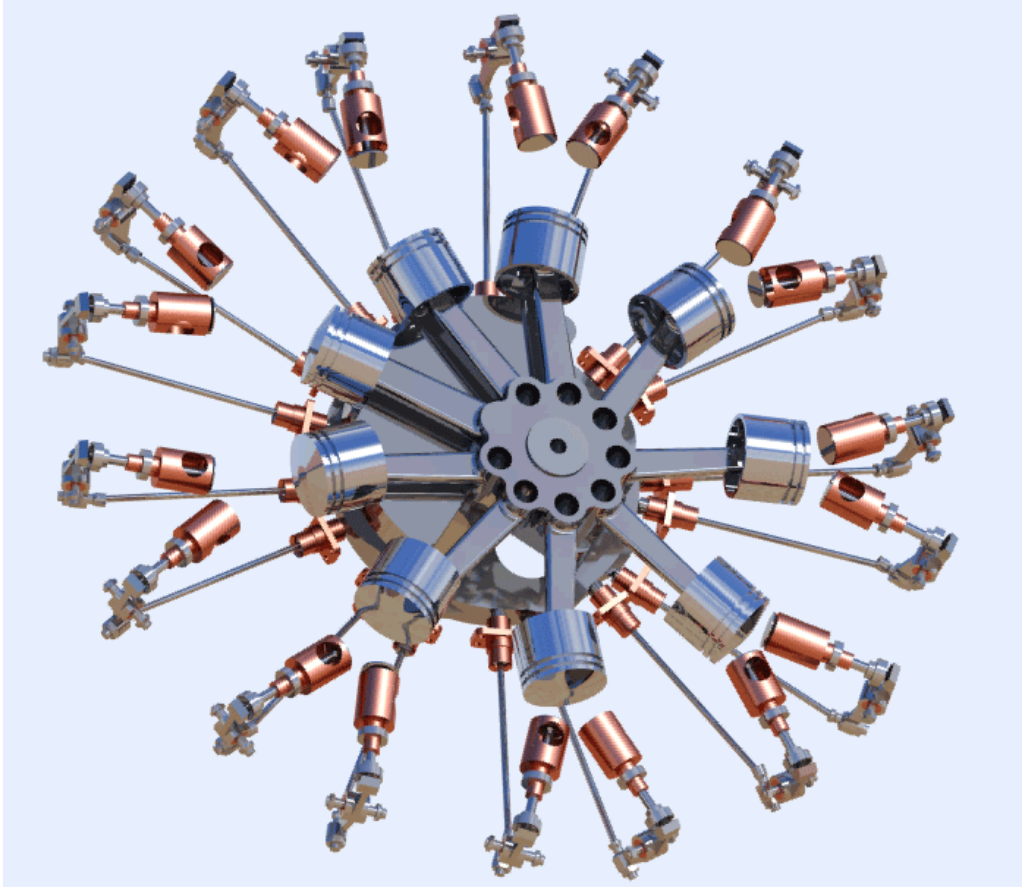
Will be available through SCPD

Stanford CS217: Engines On The Course Website?



Radial Engine

- Propeller engine
- Lower flying range
- Less aerodynamic
- Distributed performance
- Central shaft synchronization
- Too many moving parts

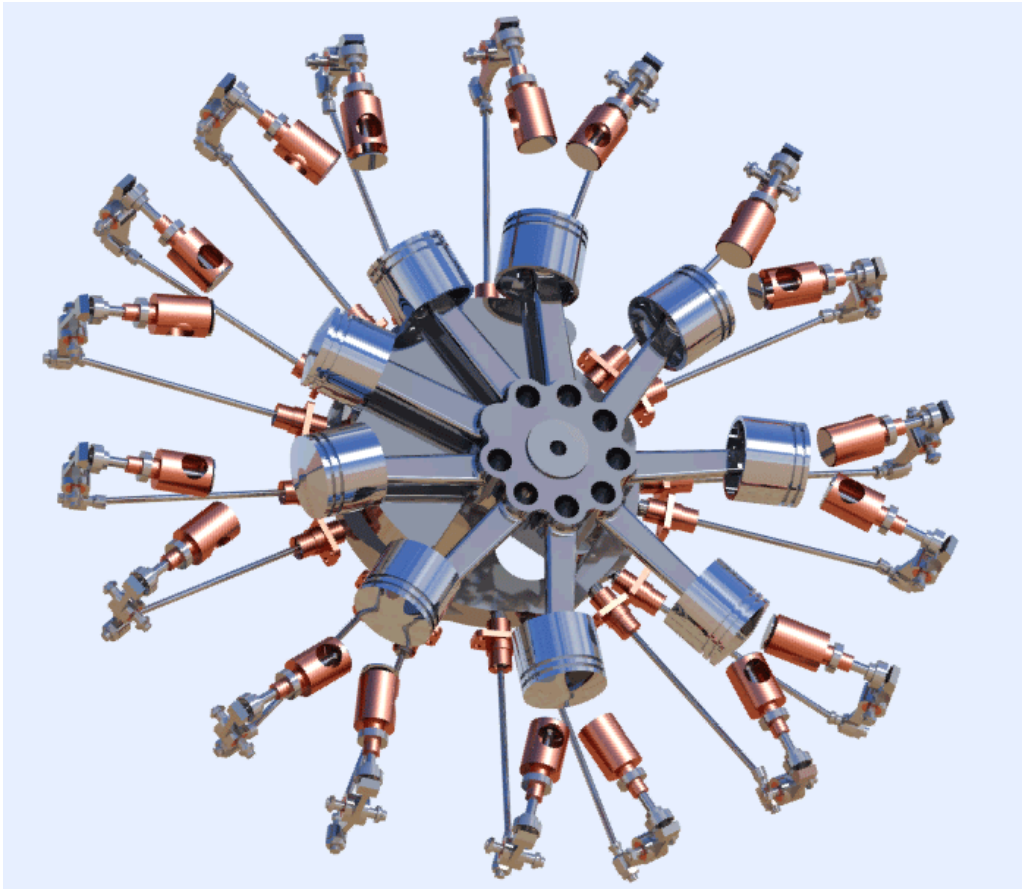


1. https://imgur.com/gallery/79Qo0/comment/12698133
2. By RichardWheeler from wikipedia

“If we all worked on the assumption that what is accepted as true is really true, there would be little hope of advance.” Orville Wright

Radial Engine

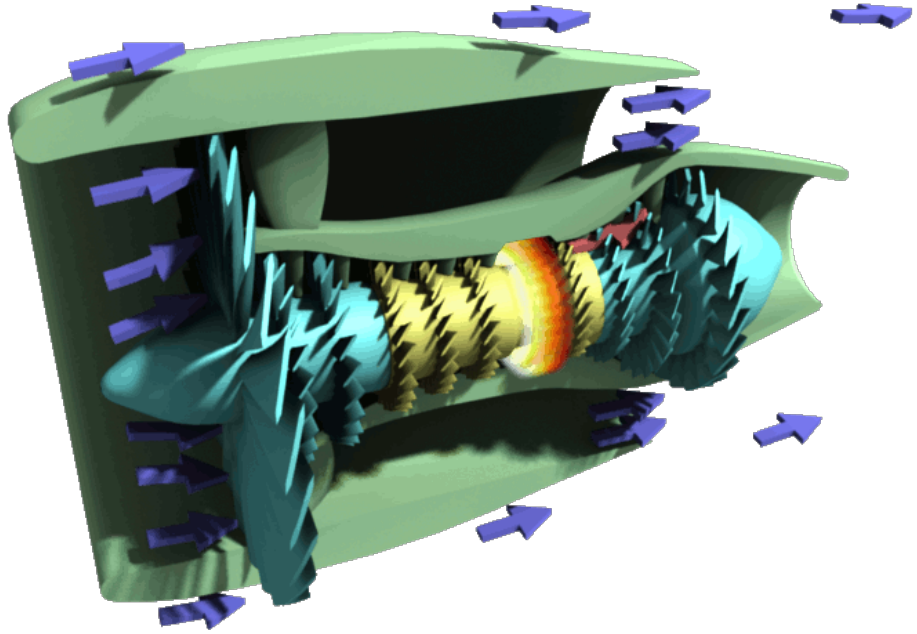
- Propeller engine
- Lower flying range
- Less aerodynamic
- Distributed performance
- Central shaft synchronization
- Too many moving parts

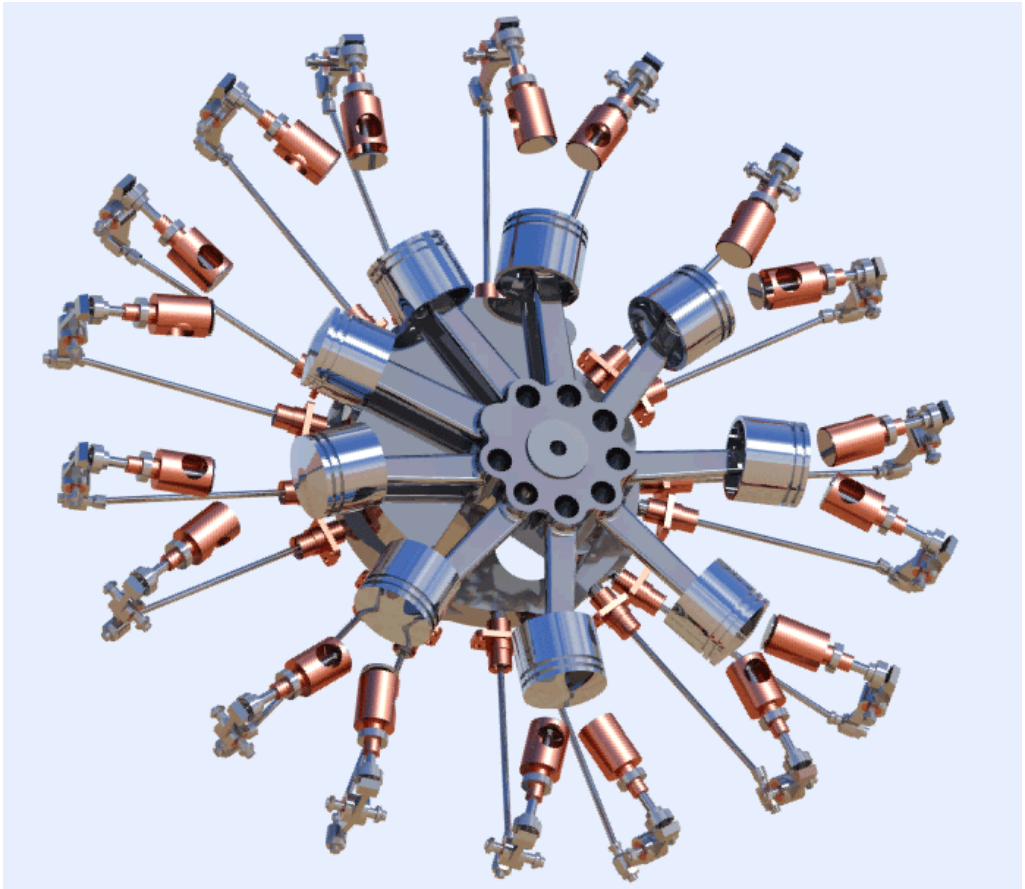


1. https://imgur.com/gallery/79Qo0/comment/12698133
2. By RichardWheeler from wikipedia

“If we all worked on the assumption that what is accepted as true is really true, there would be little hope of advance.” Orville Wright

Turbo Fan Jet Engine





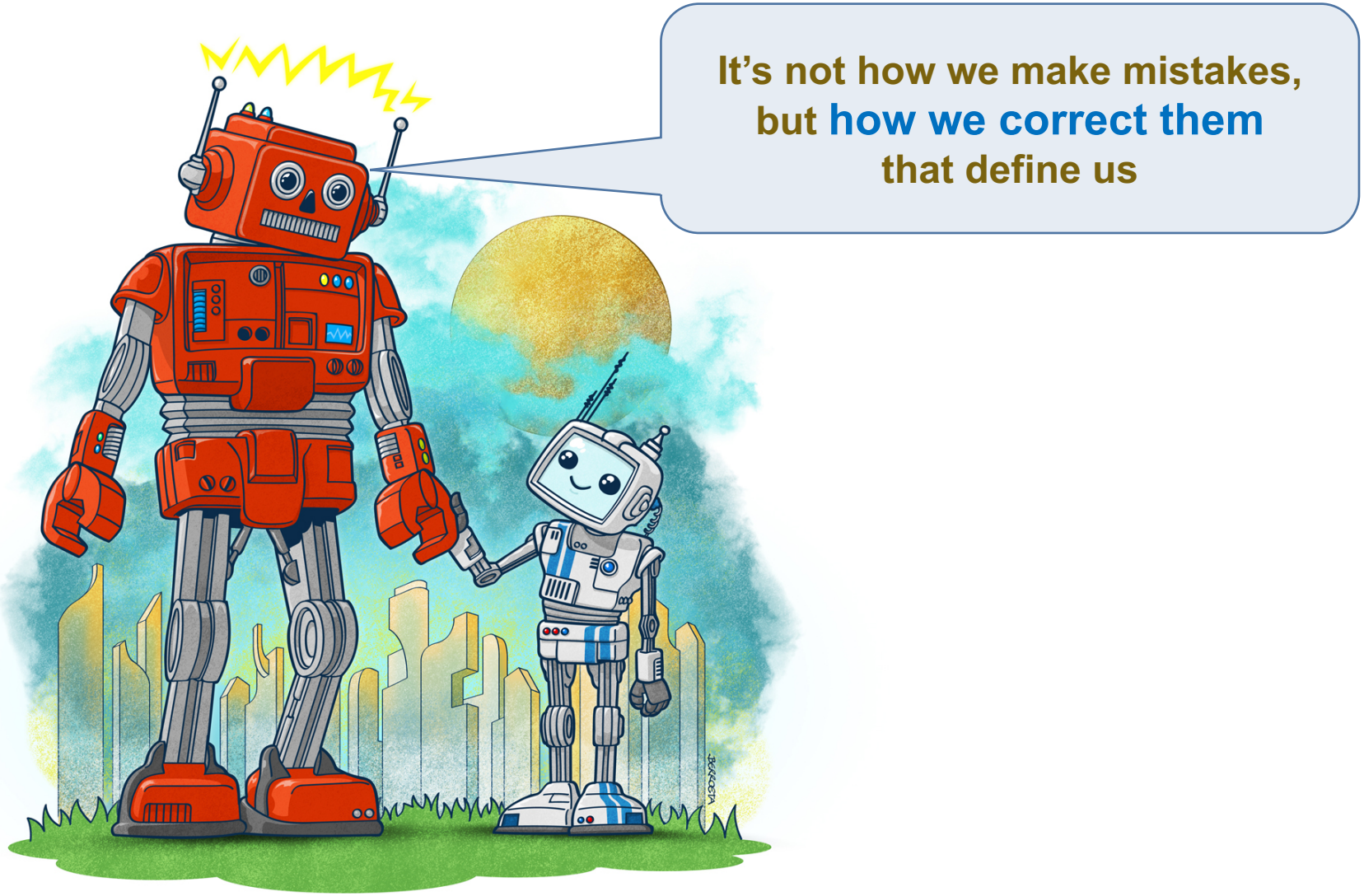
1. https://imgur.com/gallery/79Qo0/comment/12698133
2. By RichardWheeler from wikipedia

Conclusion- Today

- Current accelerators:
  - GEMM centric to overcome memory wall
  - Require large batch size to achieve high utilization/performance
- Training is a tradeoff between design simplicity, performance, and convergence
- Low Precision is feasible today
- Sparse training enforces accuracy drop and performance gains are inconclusive
- Scaling training is a matter of processor performance & communication latency
- Scaling synchronous SGD requires large batch methods
  - Accuracy may require extensive hyperparameter tuning
  - Very large batch sizes only demonstrated on CNNs
- Minimize the impact of communication latency bottleneck
  - Use asynchronous approaches, but those have all negatively impacted accuracy
  - Hide communication latency by pipelining gradients
- Benchmarks emerging with some difficulty

Our 10 Year Photo Challenge

ISCA 2009 in Austin, Texas


It’s not how we make mistakes,
but how we correct them
that define us