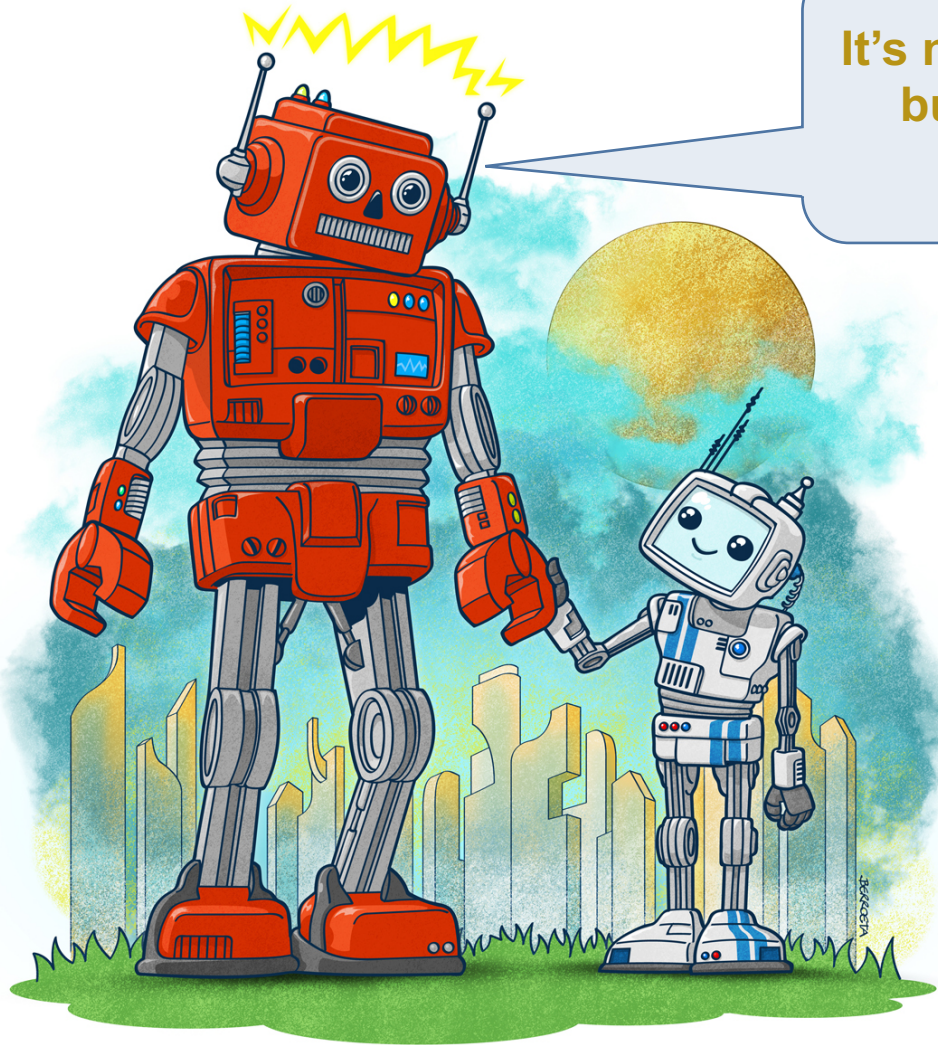
Training 1

CS 217 Stanford

Ardavan Pedram



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

Training

1. Fundamentals of training
2. Optimizing learning
3. Efficient training
4. Scaling training

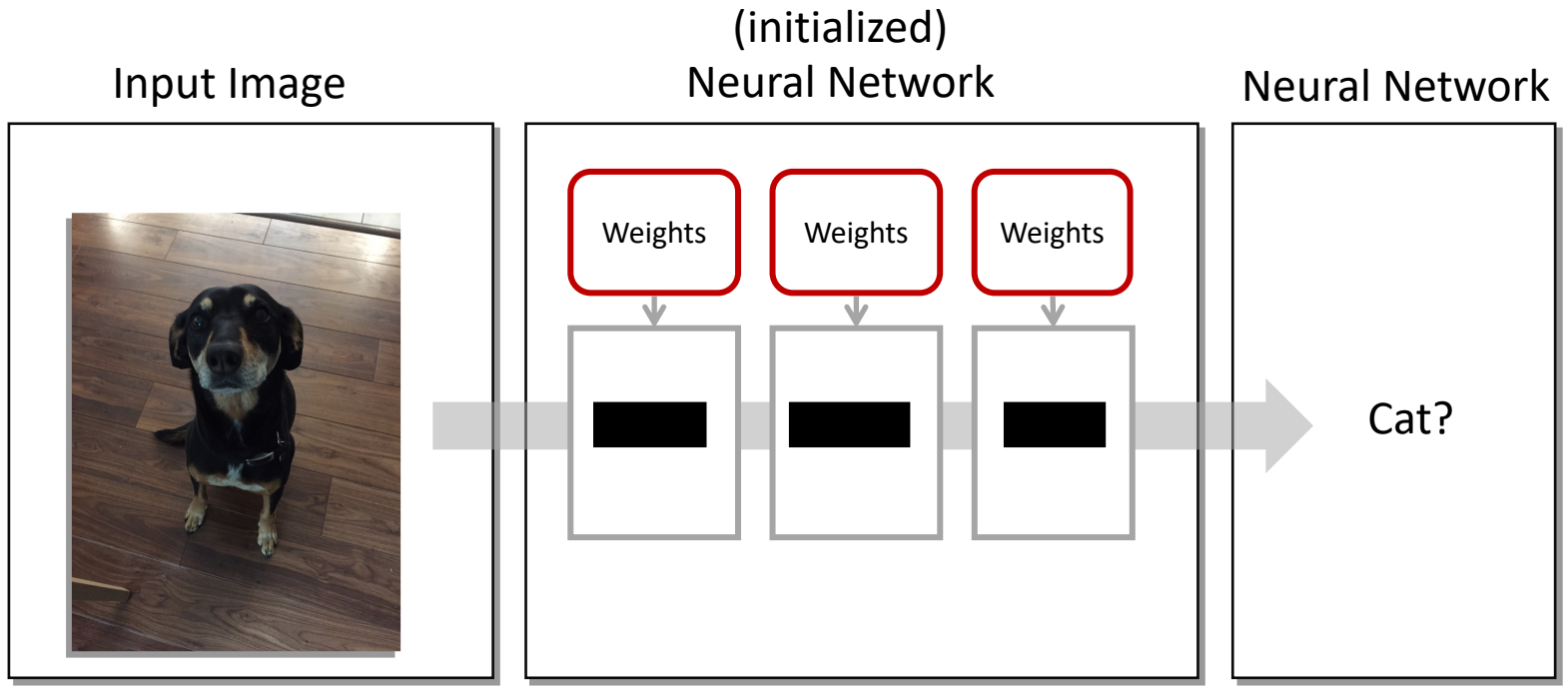
Training

1. Fundamentals of Training
   - Gradient Descent
   - Backpropagation
   - Stochastic Gradient Descent
   - Mini Batch Gradient Descent
2. Optimizing learning [Boris]
   - Momentum
   - Learning Rate
   - Misc.
3. Efficient training
   - Low Precision
   - Sparsity
4. Scaling training
   - Scaling Computation power
   - Scaling Network Size
   - Large Batch Training
   - Asynchronous Training

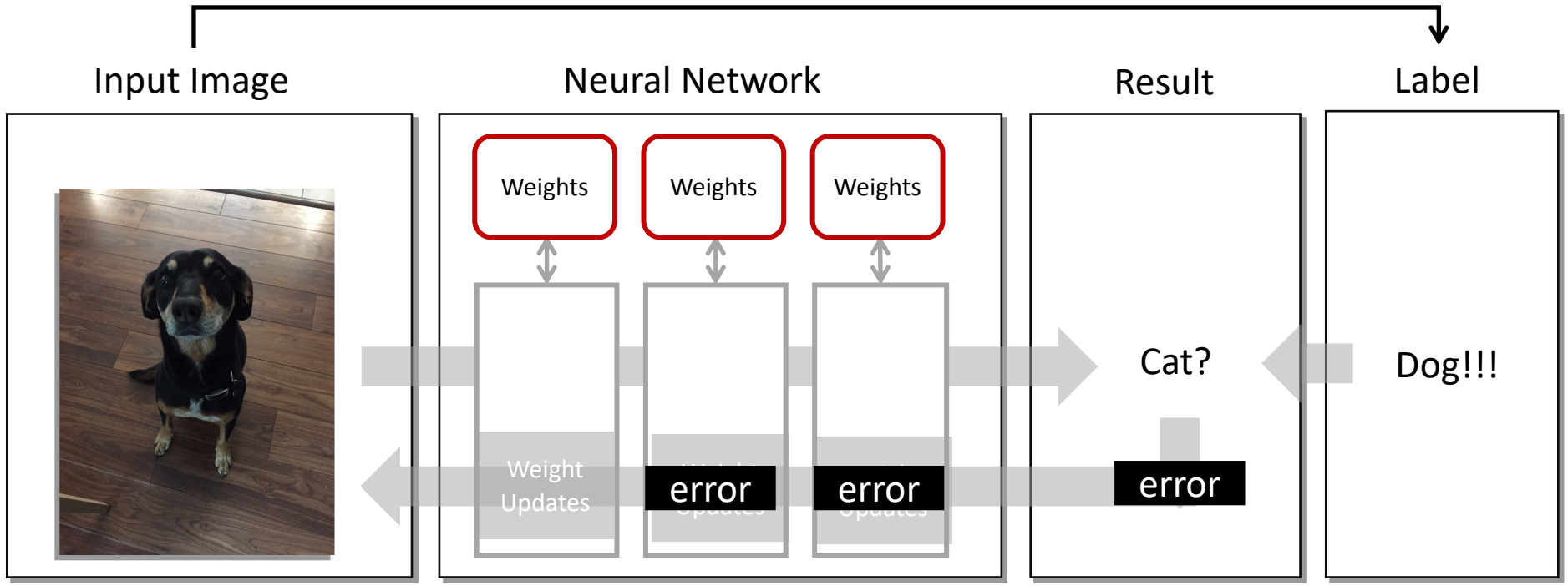
Fundamentals of Training

- Gradient Descent
- Backpropagation
- Batch Normalization
- Collective Communications

Recap

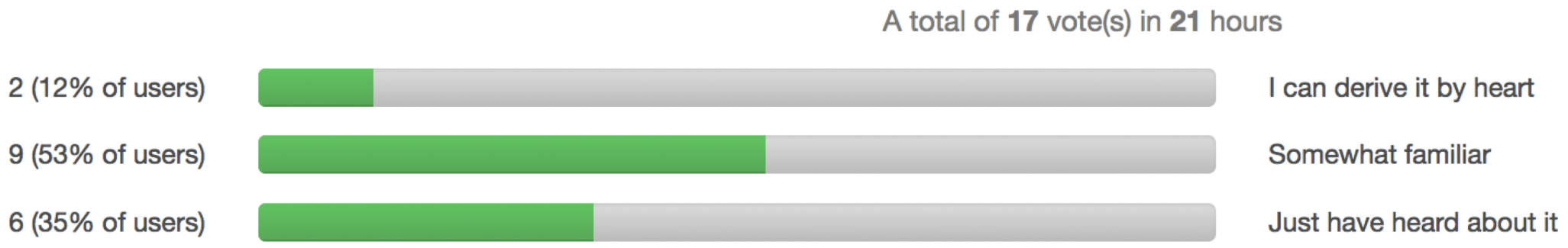

Input Image
(initialized)
Neural Network
Neural Network
Weights
Weights
Weights
Cat?

Recap



Source: Michaela Blott. Xilinx

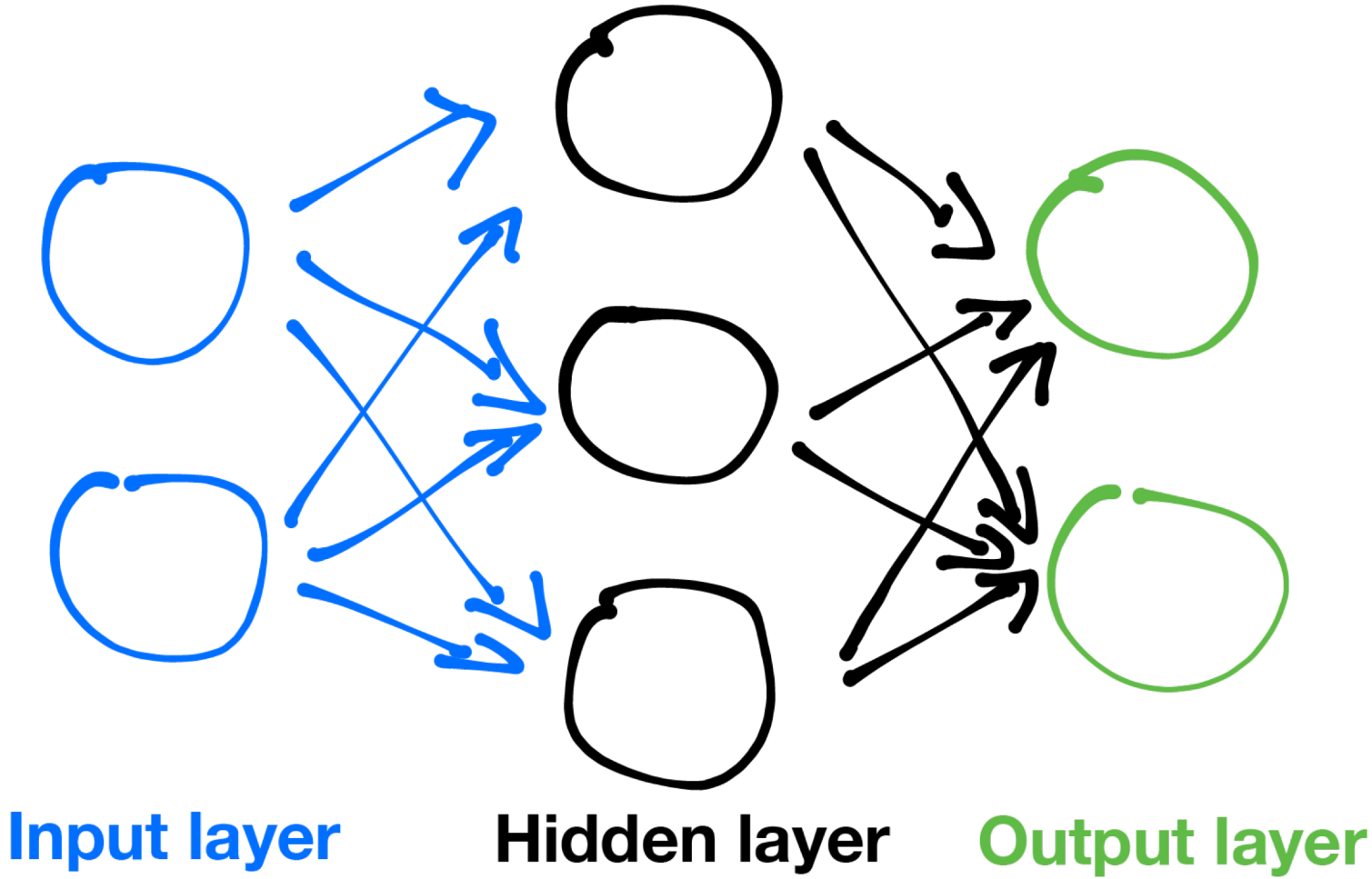
How Familiar are you with Backpropagation?



- My Mistake : Define “Familiar”

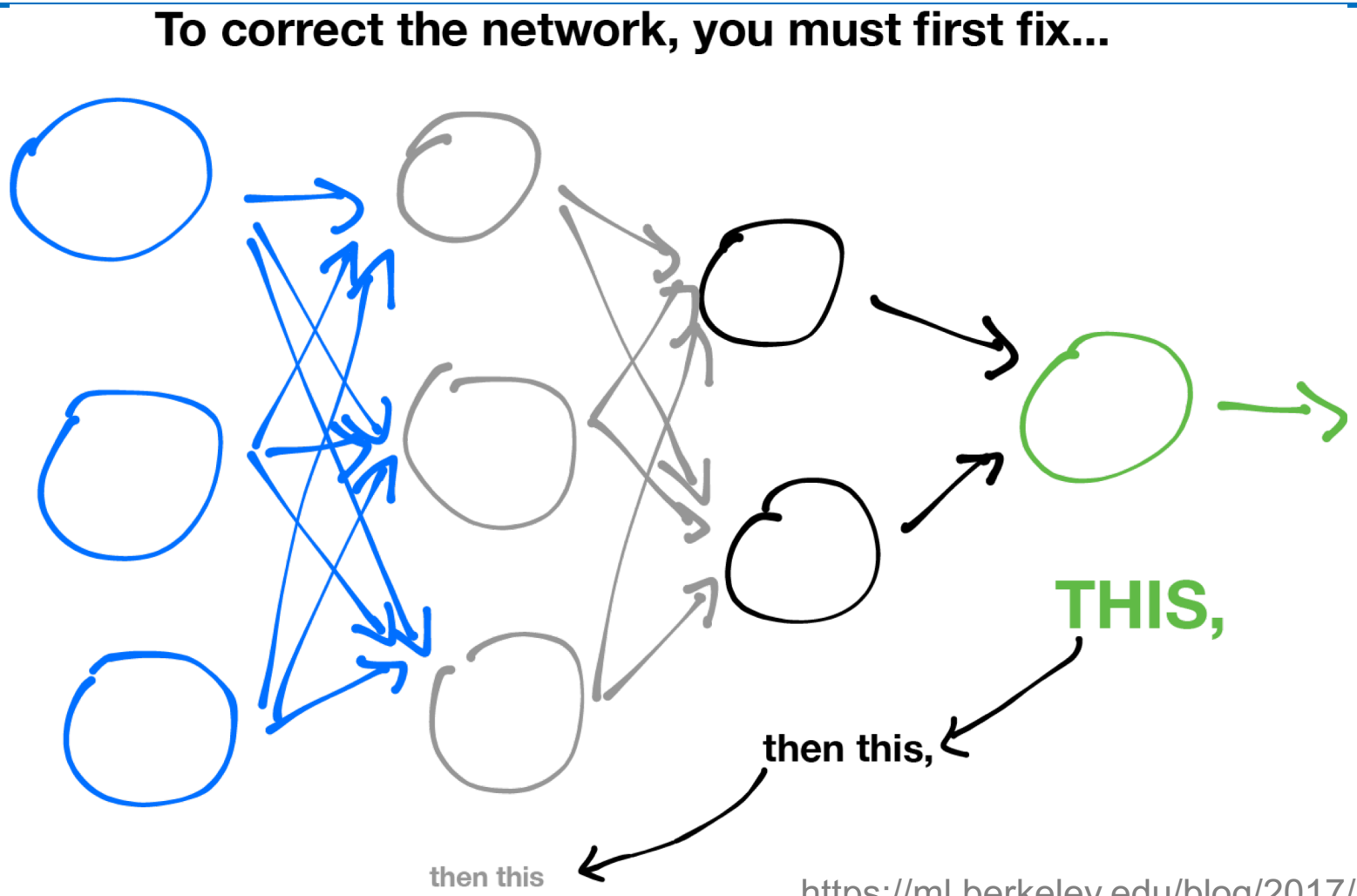
Recap: Neural Network

- What is the basic operation in each layer?



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

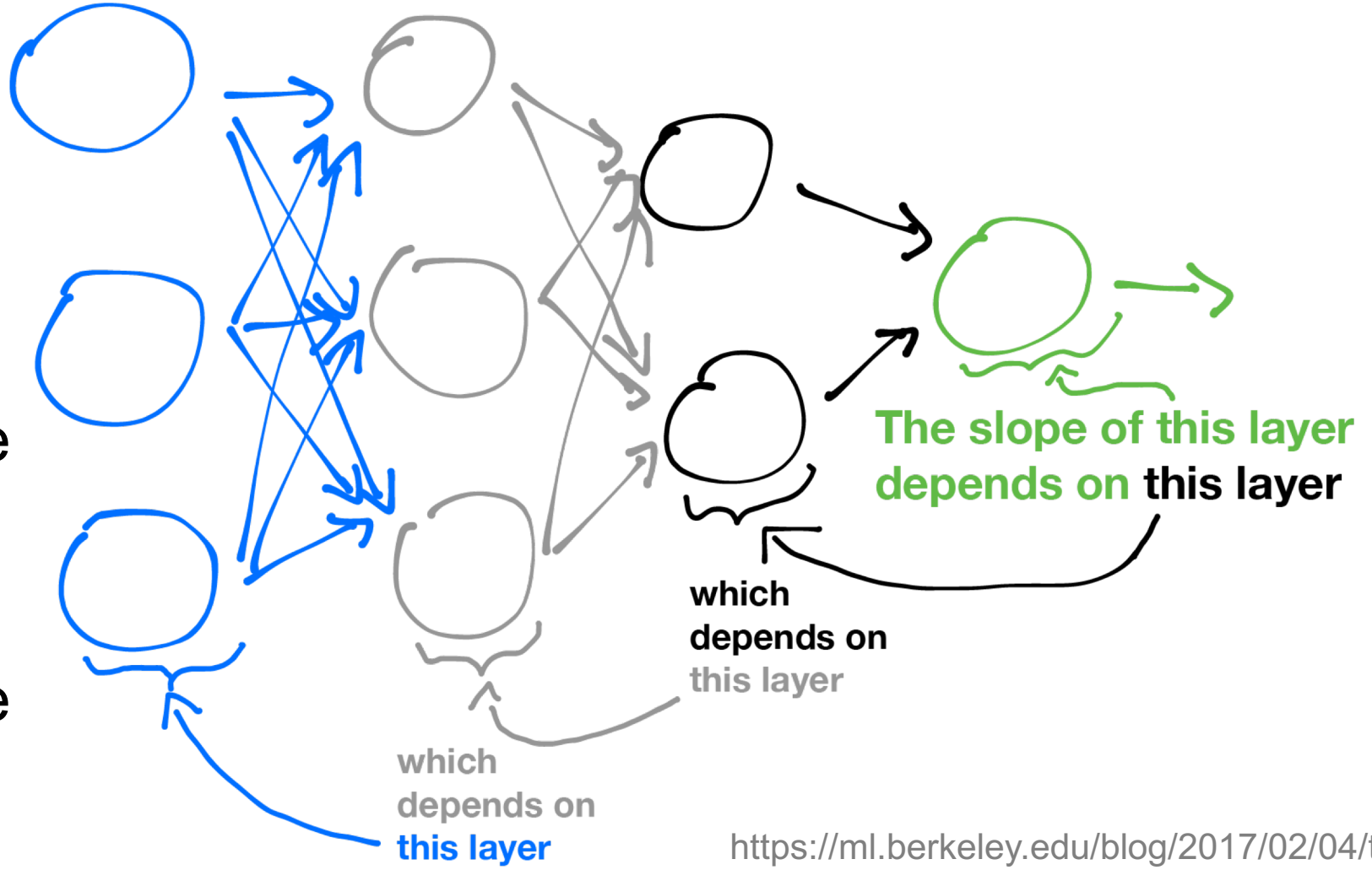
Propagate The Error Backwards



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

Backpropagation

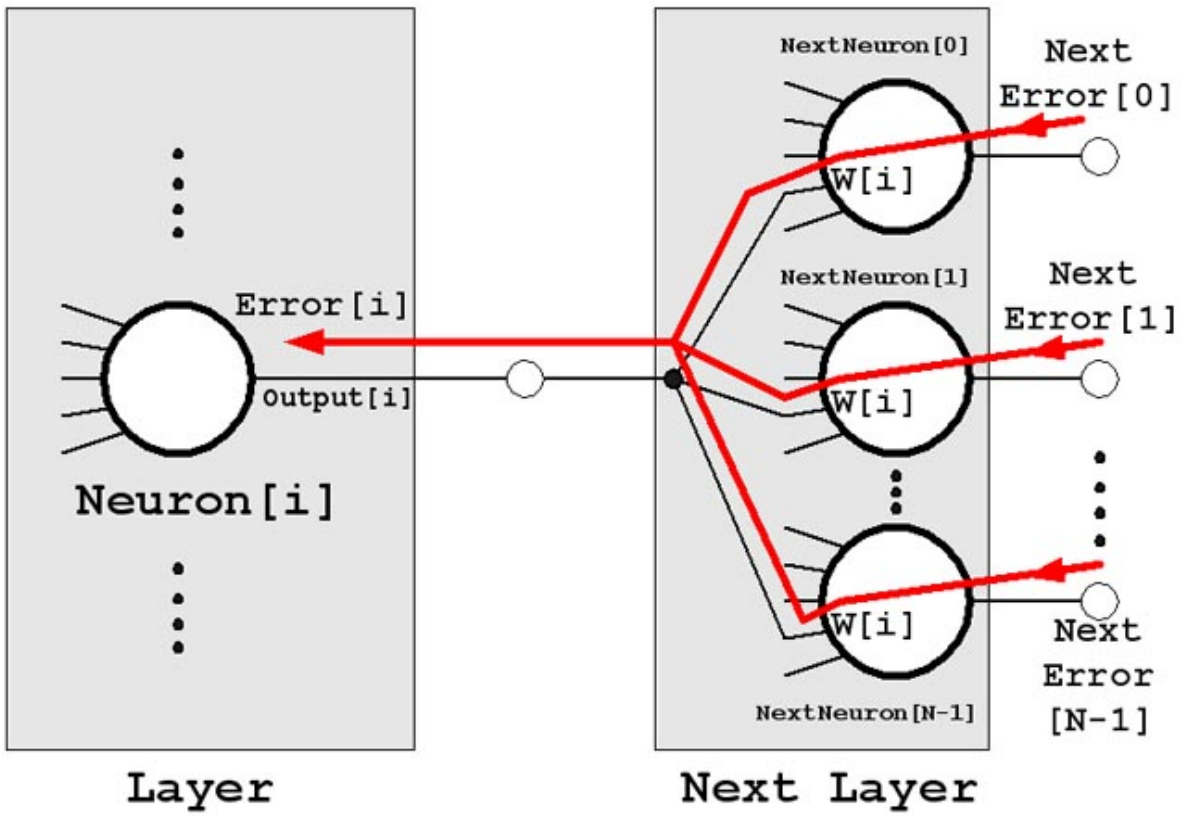
- What is the slope?
- What is the operation?



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

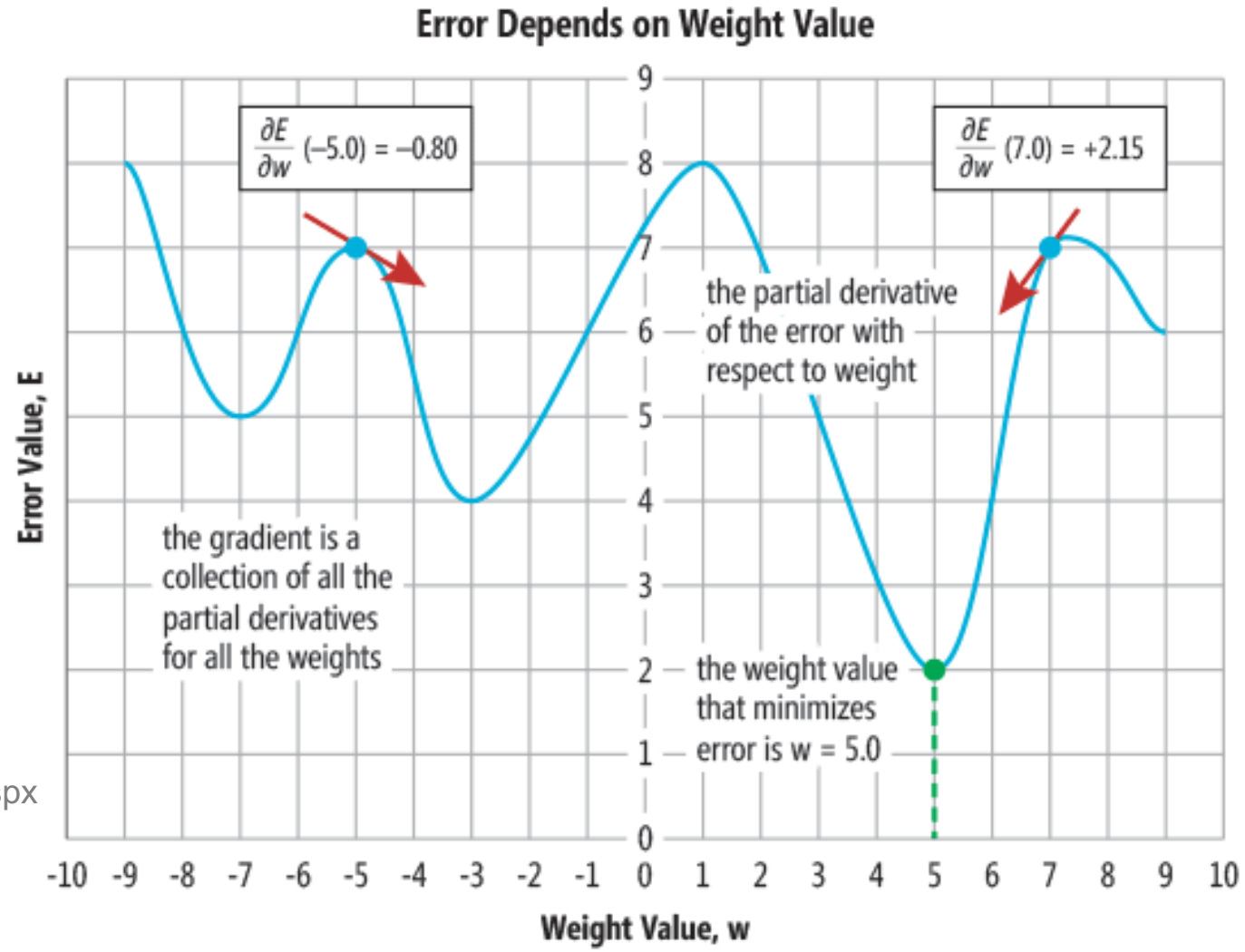
Backpropagation

- Any Similarities with Forward pass?



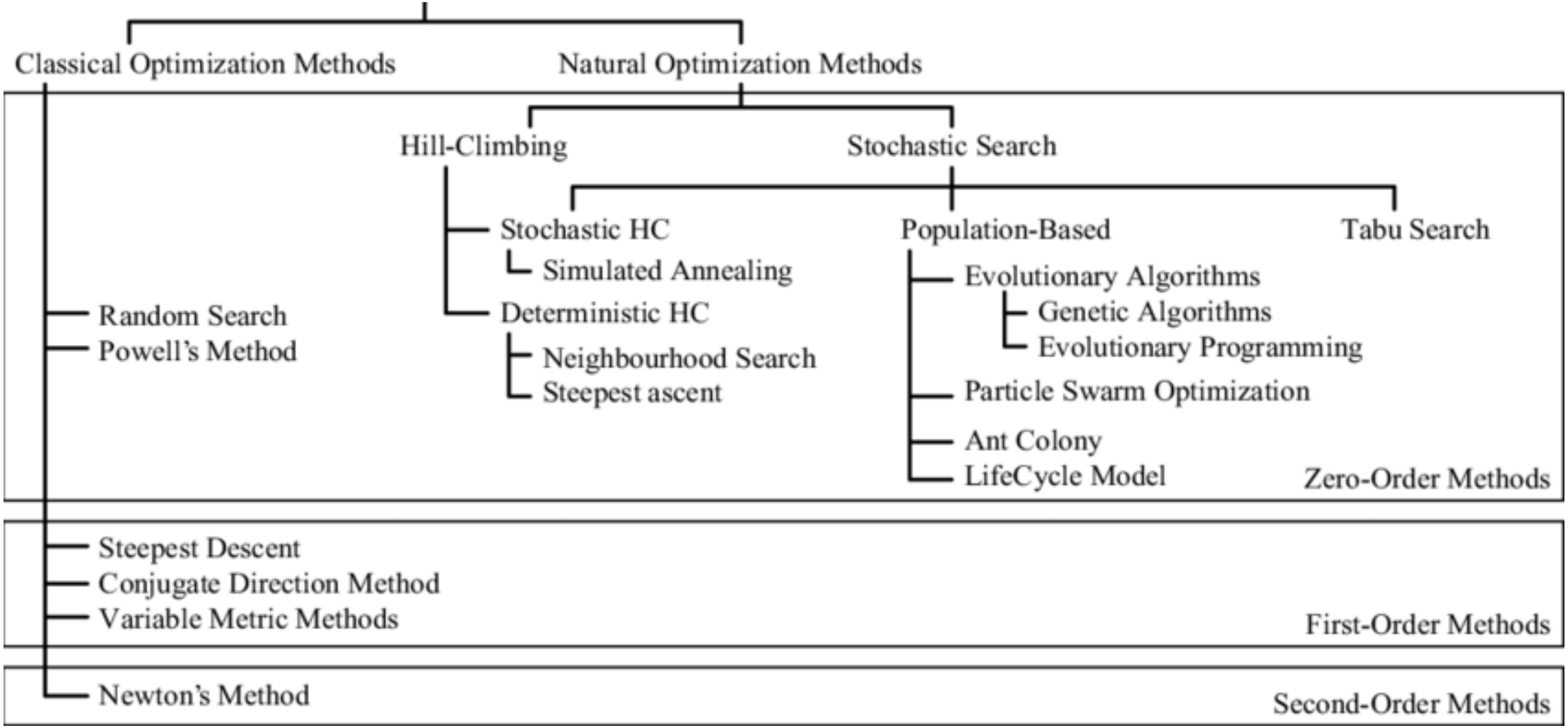
Gradient Descent

- What is the goal here?

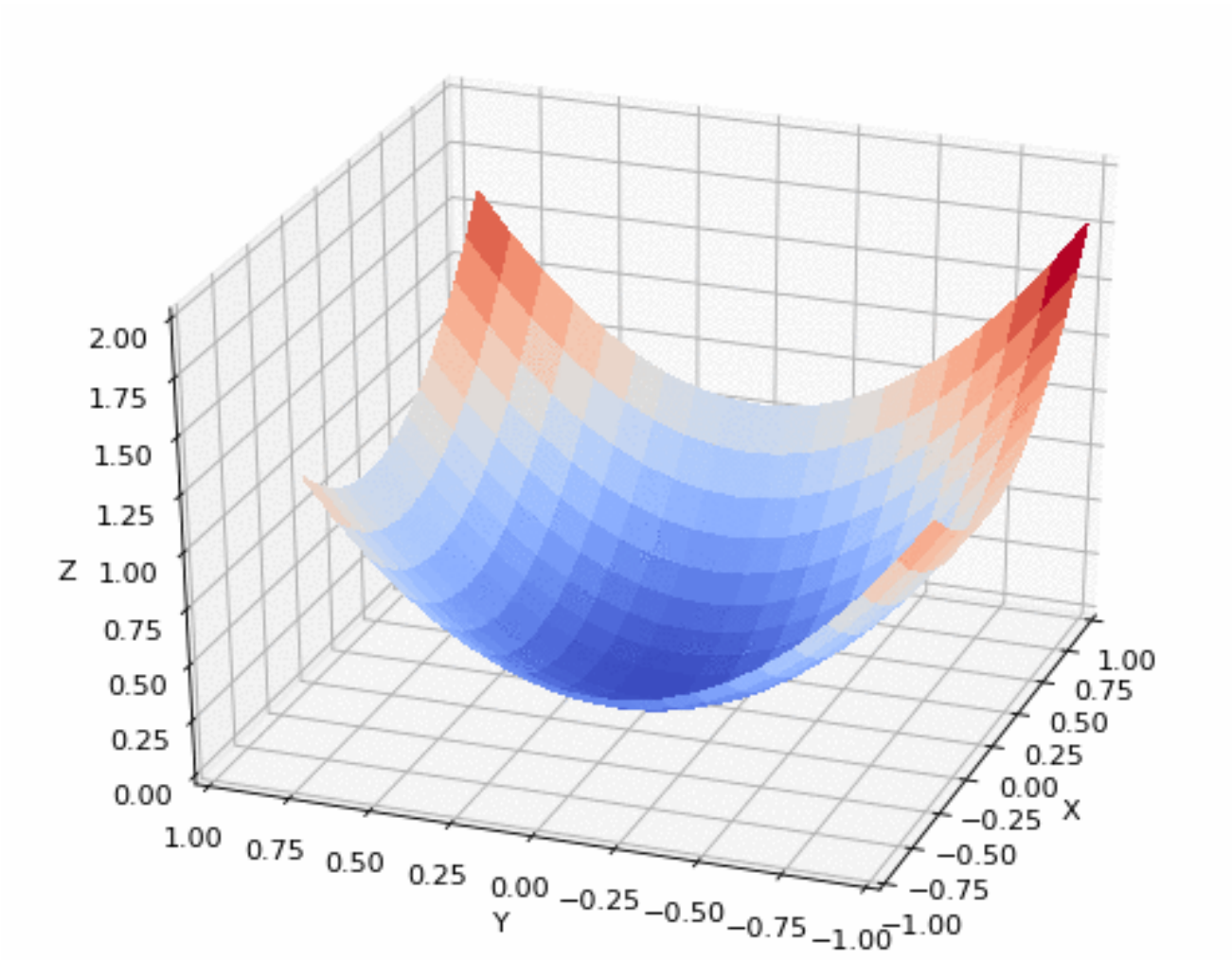


https://msdn.microsoft.com/en-us/magazine/dn913188.aspx

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

(minimization: substract gradient term because we move towards local minima)

(the derivative of f with respect to a)

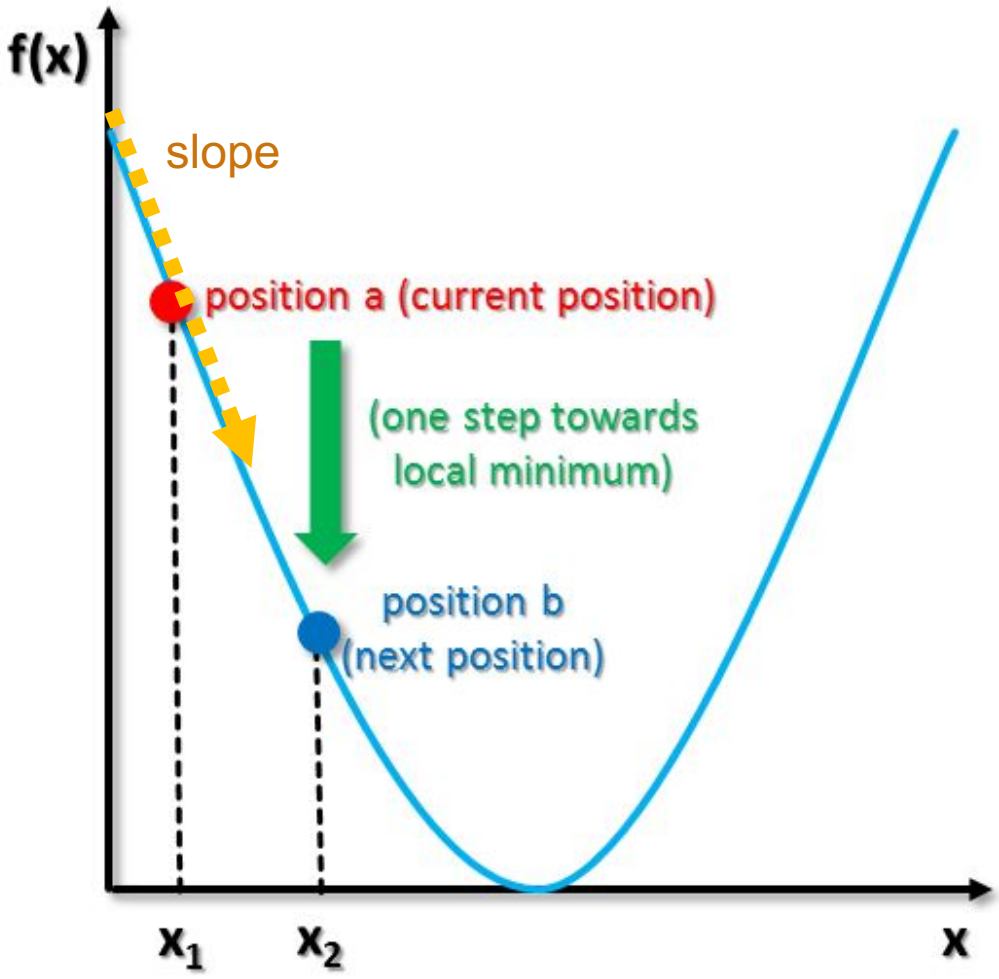
$$\mathbf{b} = \mathbf{a} - \gamma \nabla \mathbf{f}(\mathbf{a})$$

(old position before the step)

(gradient term is steepest ascent)

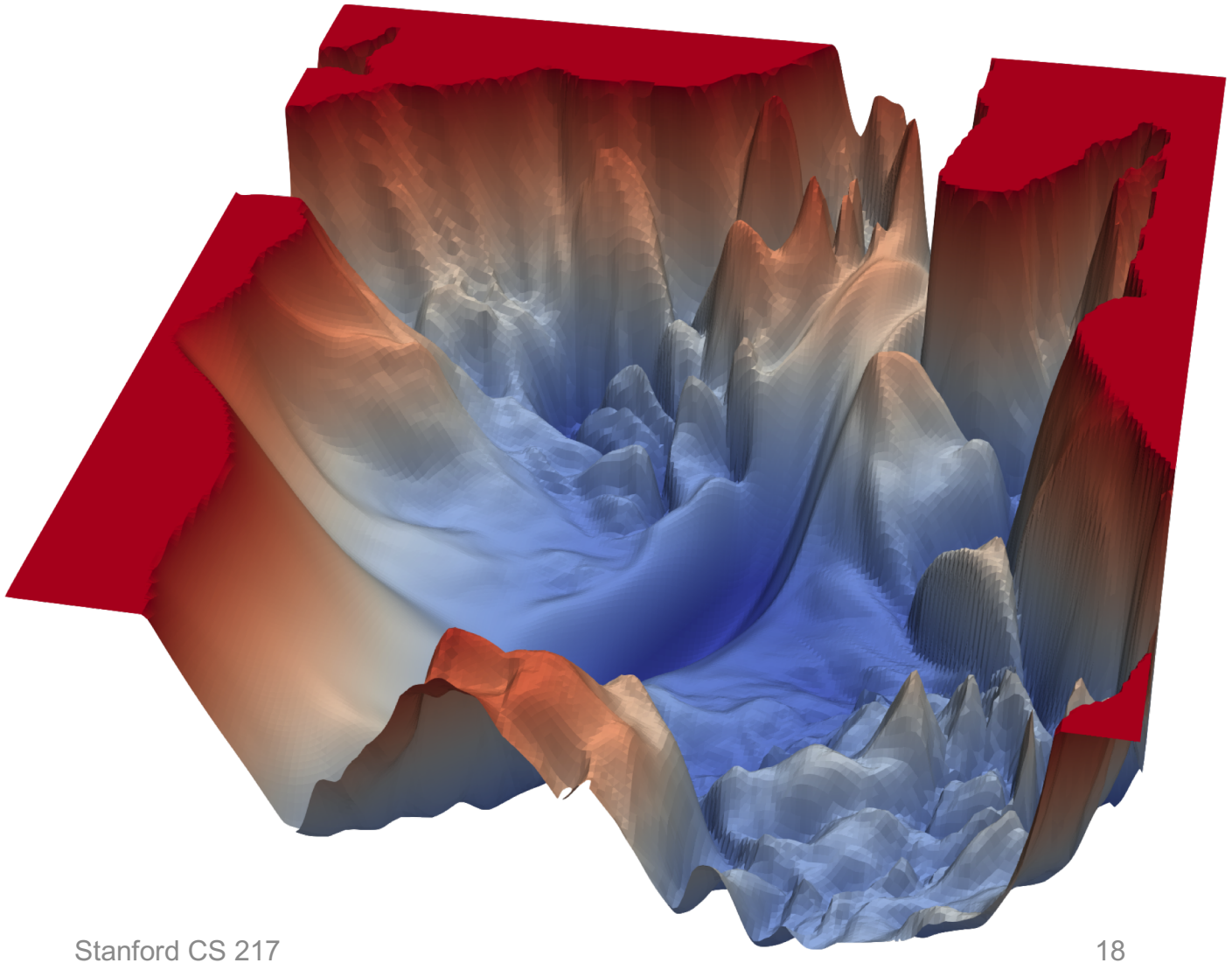
(new position after the step)

(weighting factor known as step-size, can change at every iteration, also called learning rate)



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

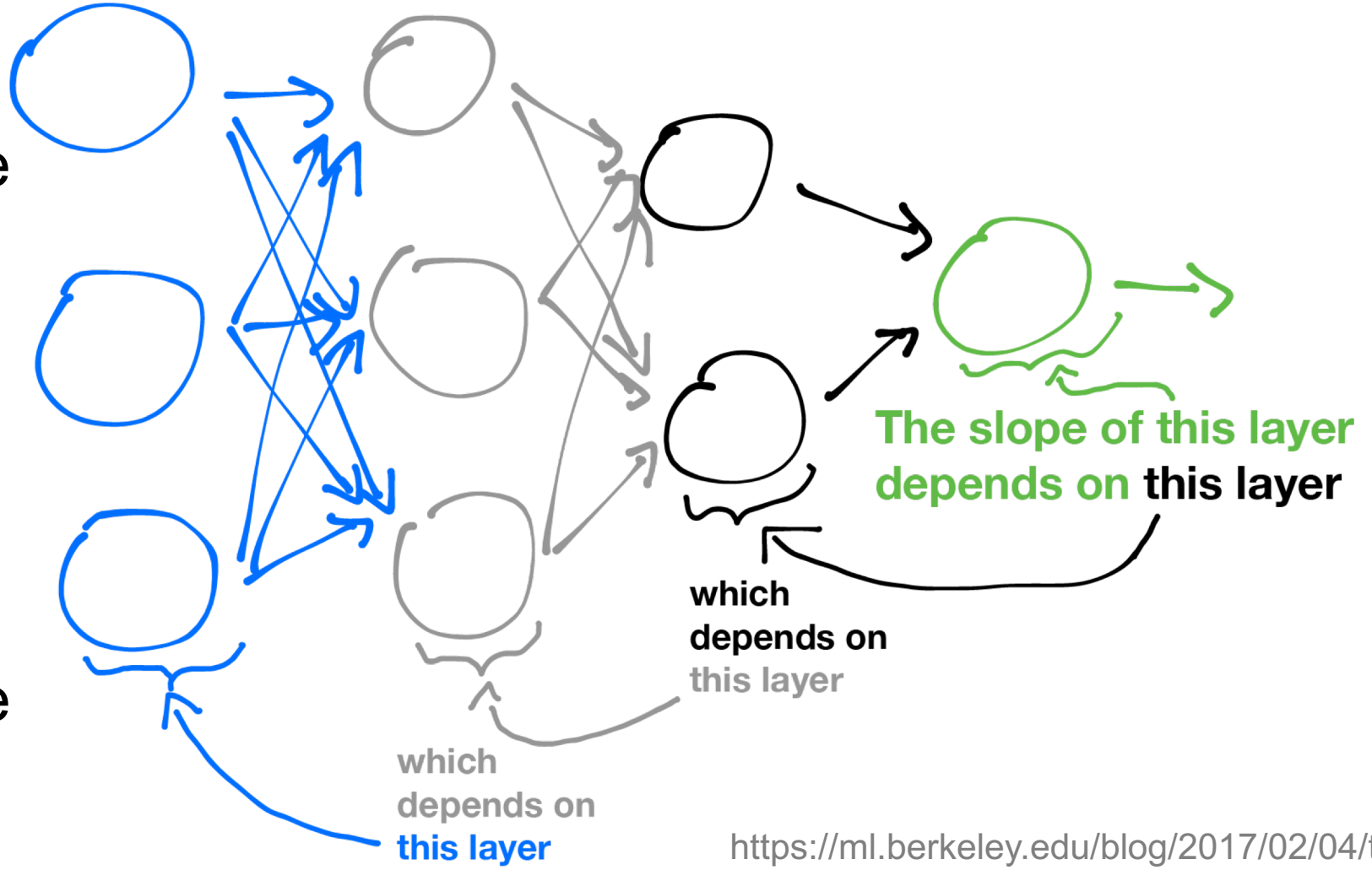
Gradient Surface



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

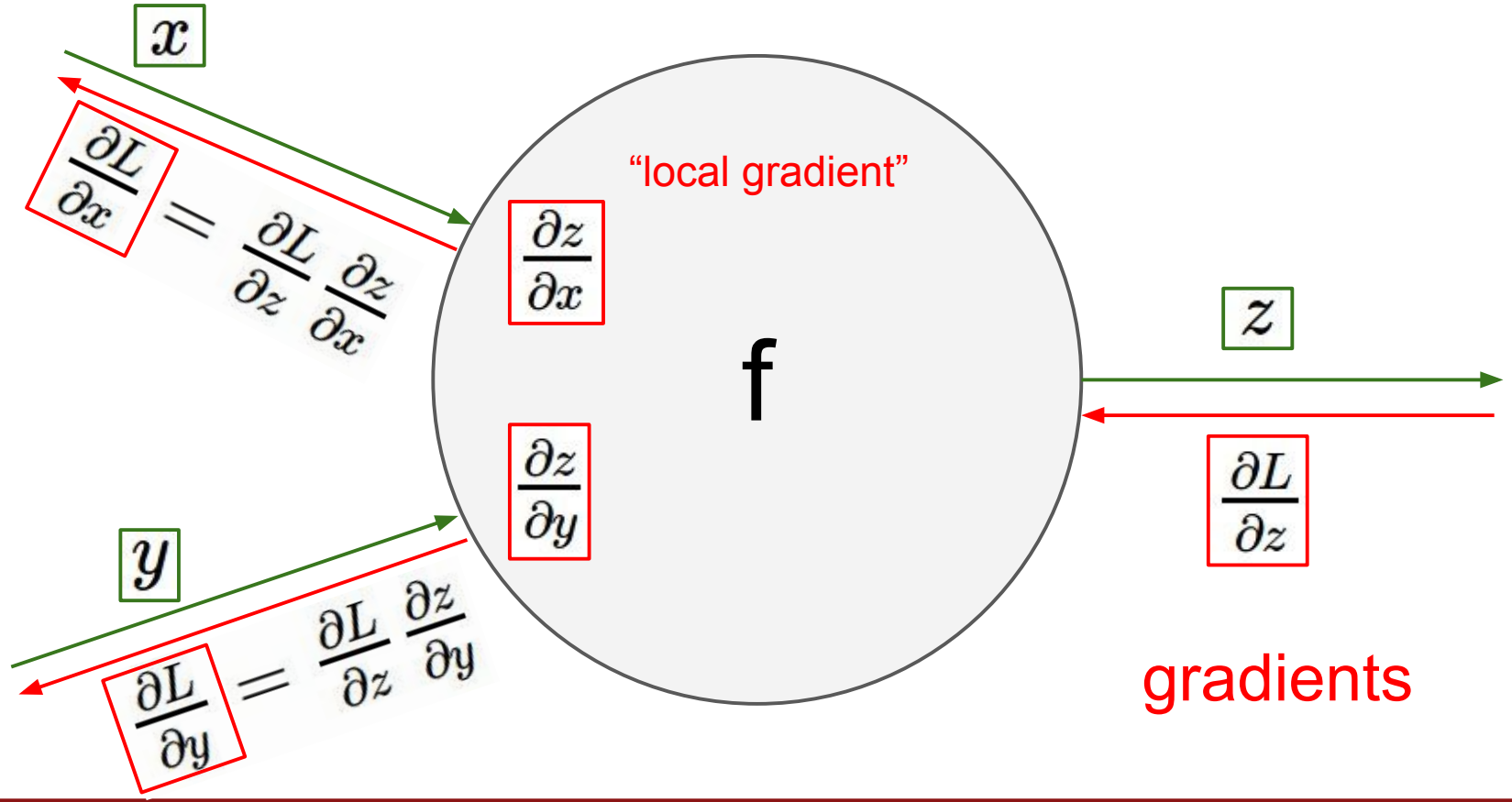
Back to… Backpropagation

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



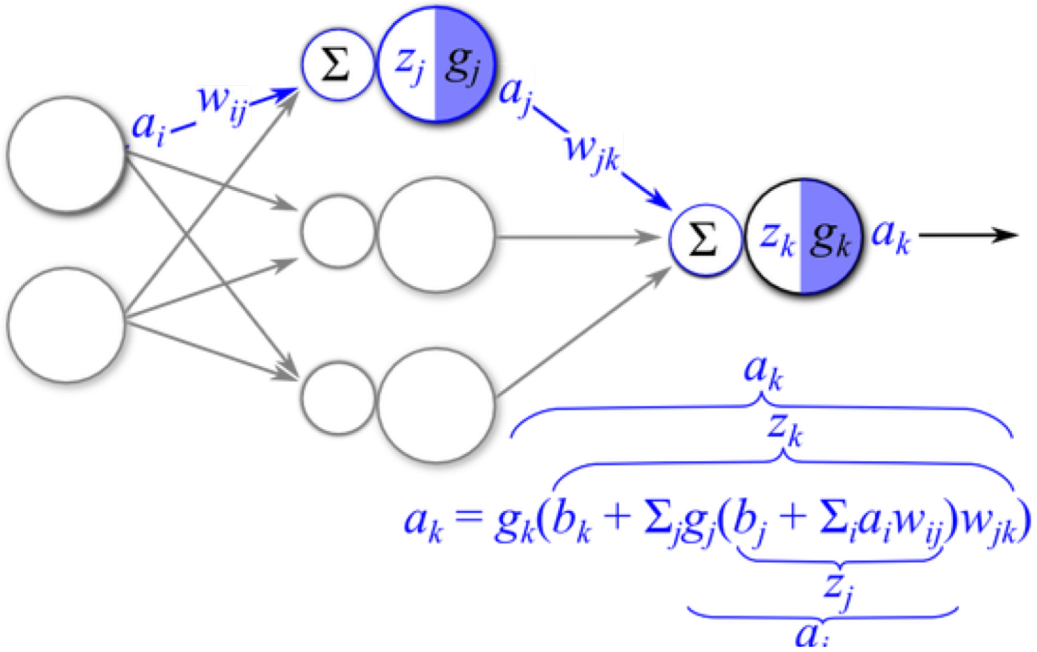
https://ml.berkeley.edu/blog/2017/02/04/tutorial-3/

Chain Rule

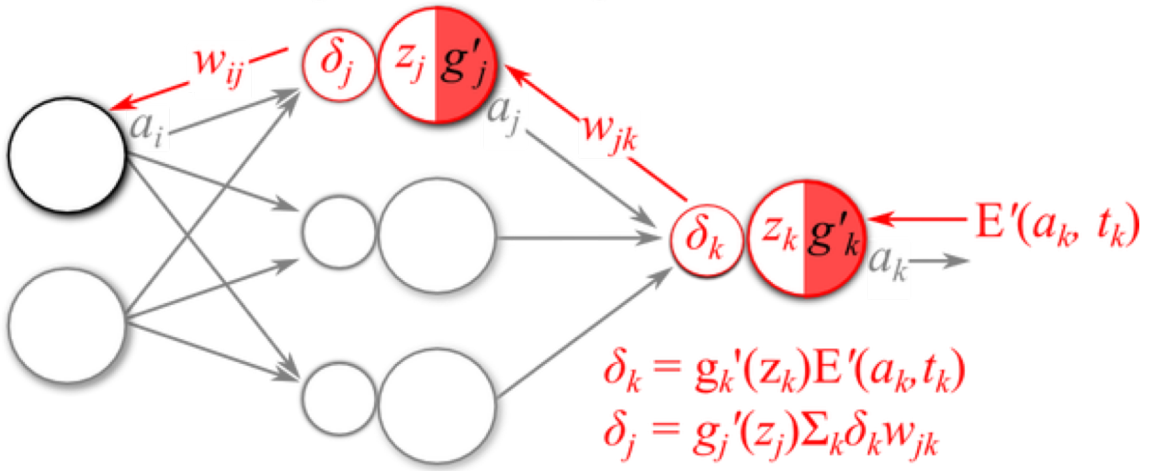

x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial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

Recap

I. Forward-propagate Input Signal

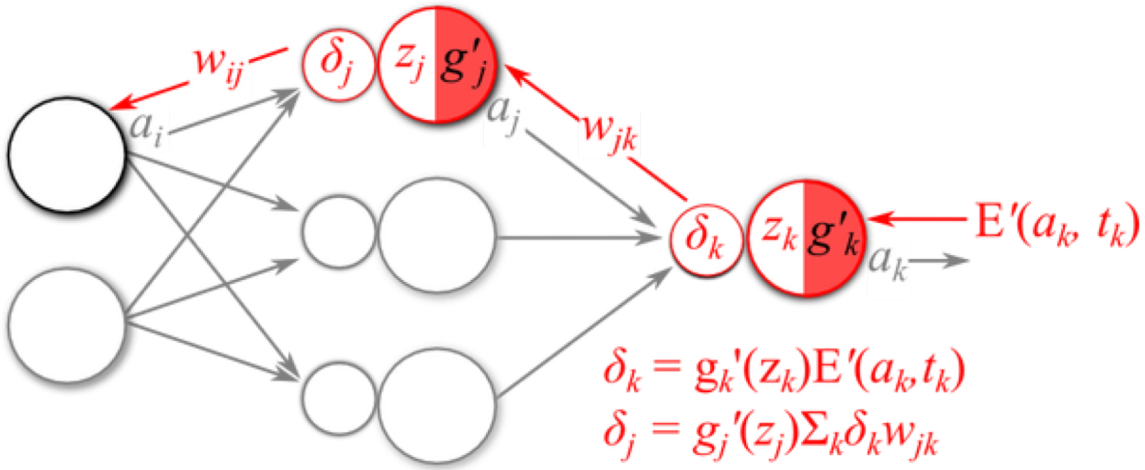


II. Back-propagate Error Signals



Delta Update and Weight Update

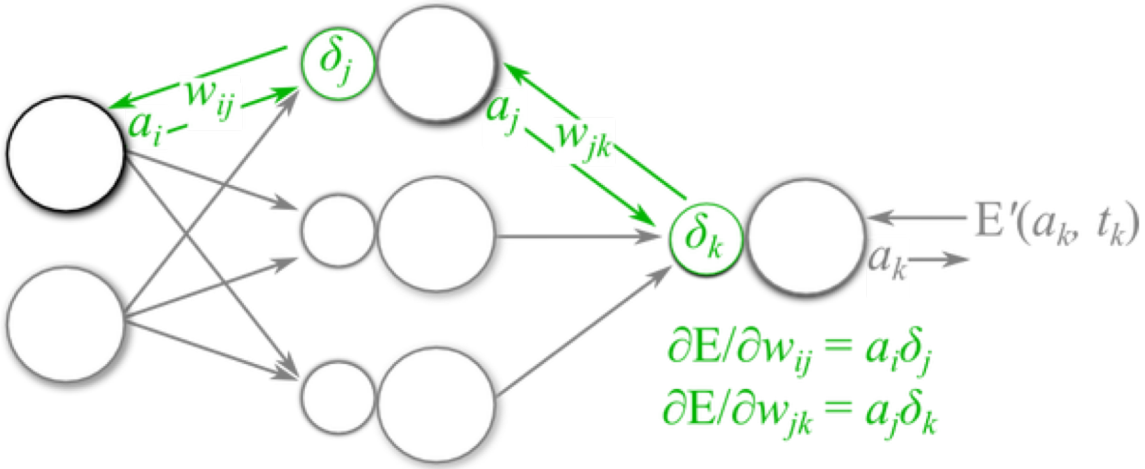
II. Back-propagate Error Signals



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

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

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/

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:

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

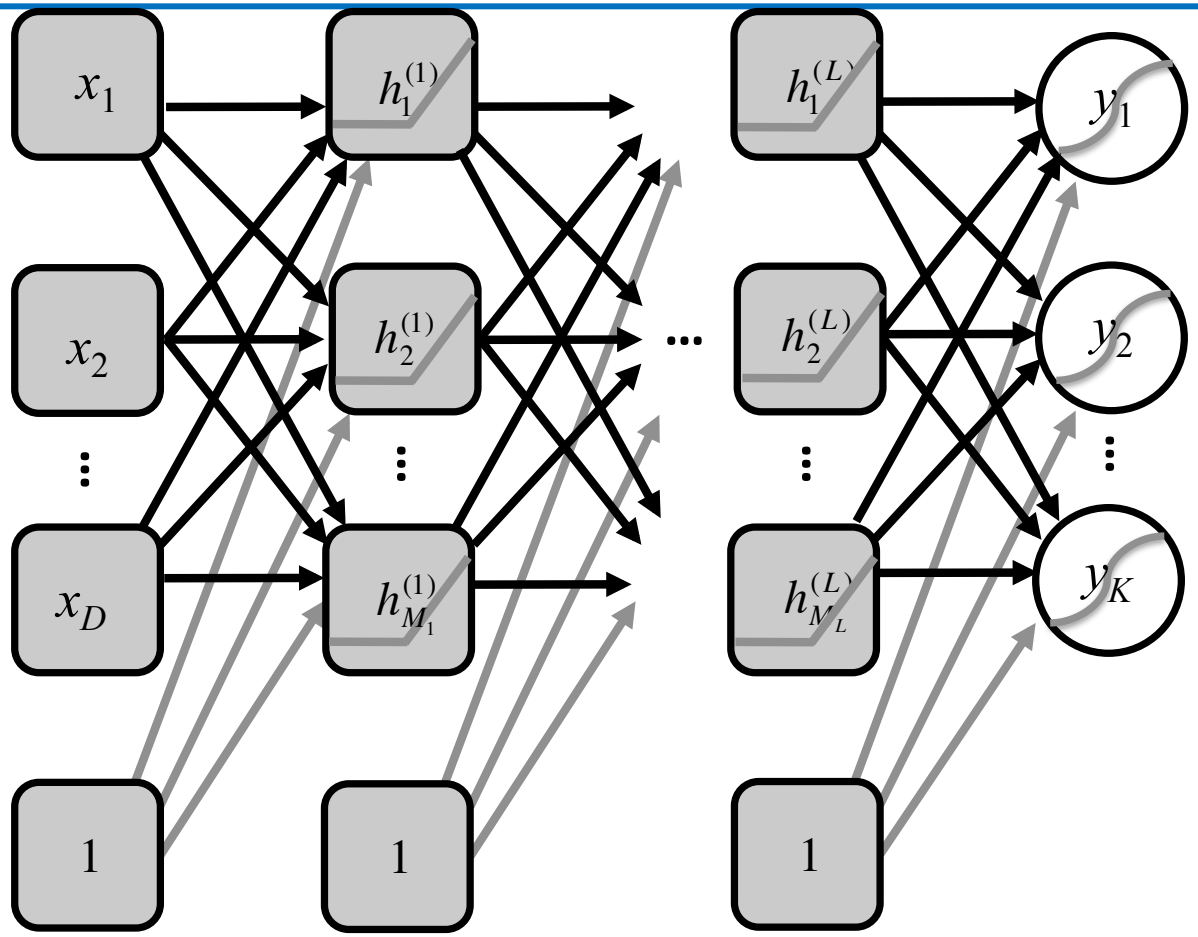
- Where *pre-activation functions* $\mathbf{a}^{(l)}(\mathbf{x})$ are typically linear, of the form $\mathbf{a}^{(l)}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}$ with matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$
- This formulation can be expressed using a single parameter matrix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

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

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

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 kth 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

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

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

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

  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 kth parameter vector of the L+1th matrix of parameters is

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

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

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

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 kth row of the Lth matrix of parameters
- Since the bias terms are constant, it is unnecessary to backprop through them, so

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

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

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

- 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 kth vector of parameters of the lth network layer can therefore be computed using products of matrices of the following form

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

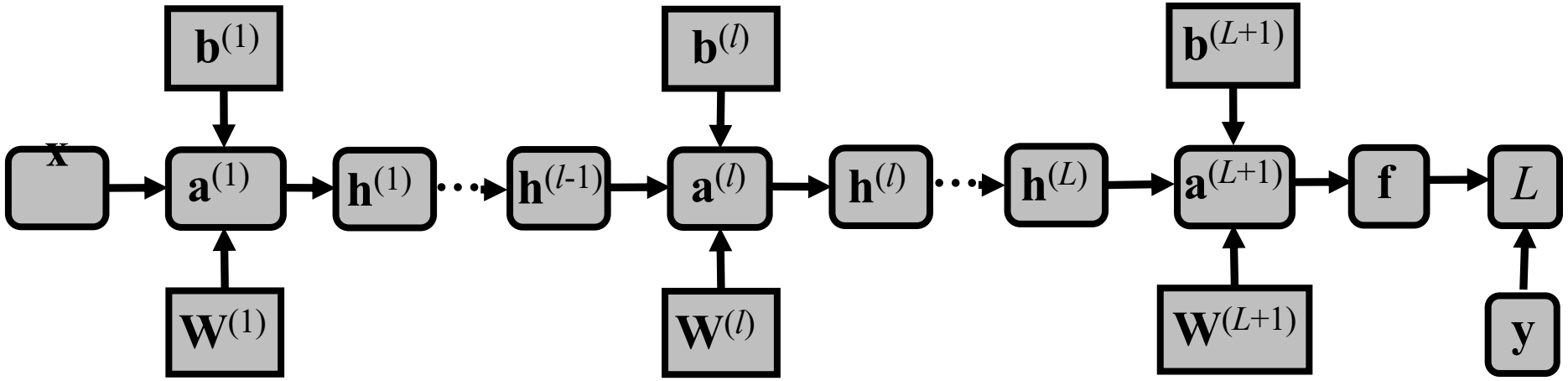
- When l=1, $\hat{\mathbf{h}}_{(0)} = \hat{\mathbf{x}}$, the input data with a 1 appended
- Note: since **D** is usually diagonal the corresponding matrix-vector multiply can be transformed into an element-wise product $\circ$ by extracting the diagonal for **d**

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

Visualizing backpropagation

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

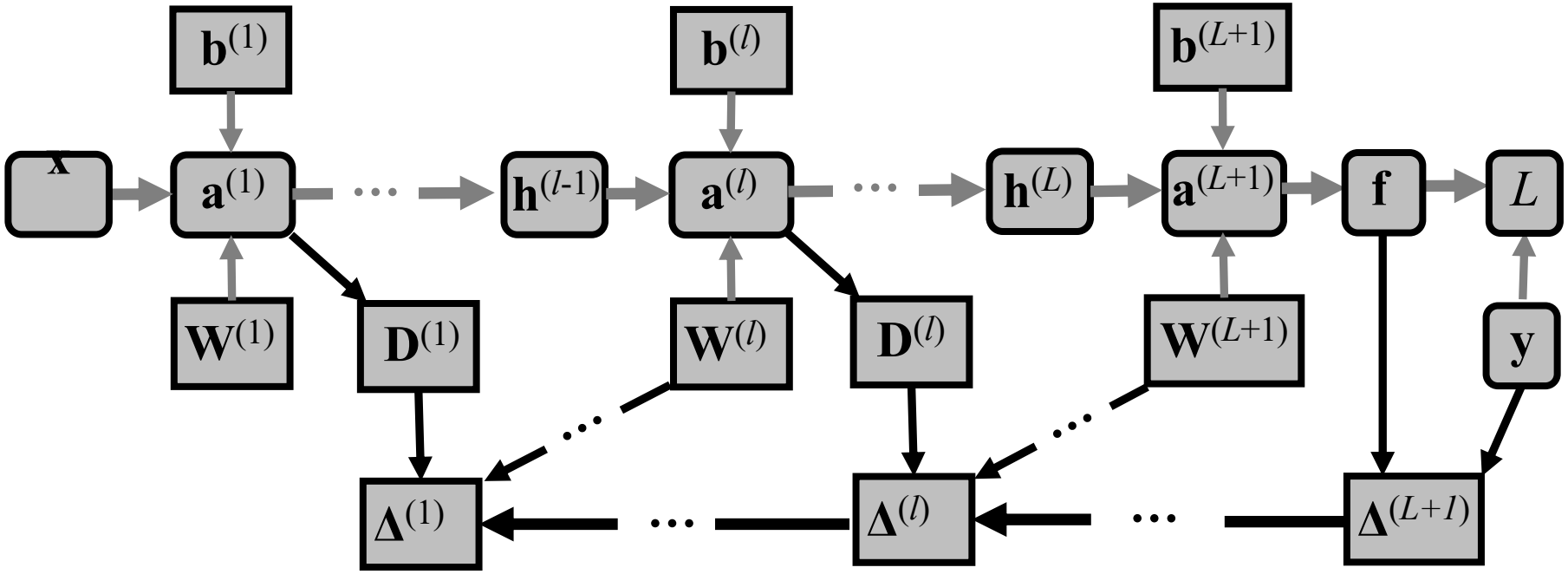
$$\mathbf{h}^{(1)} = \mathrm{act}(\mathbf{a}^{(1)}) \qquad \mathbf{h}^{(l)} = \mathrm{act}(\mathbf{a}^{(l)}) \qquad \mathbf{f} = \mathrm{out}(\mathbf{a}^{(L+1)})$$



- 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

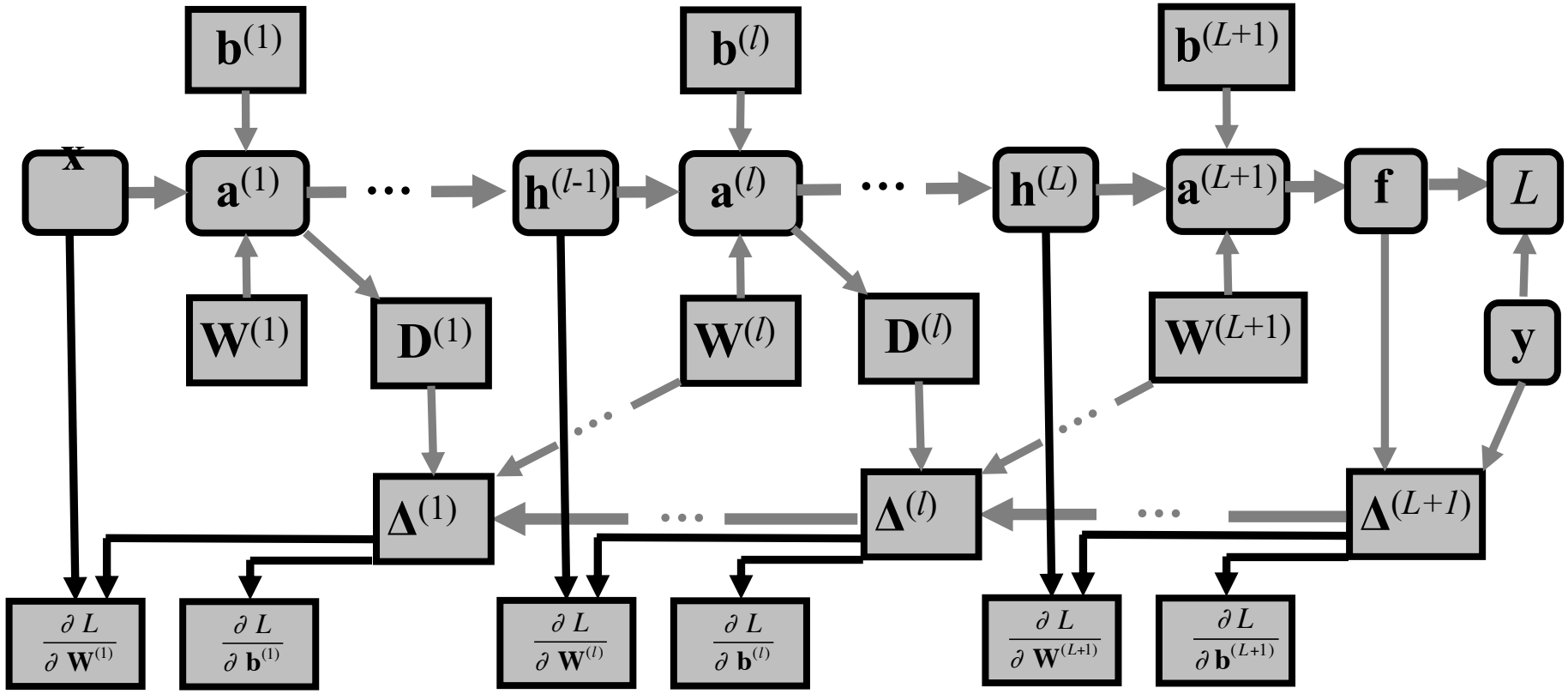
- In the backward propagation phase we compute terms of the form below



$$\Delta^{(l)} = \mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)} \qquad \frac{\partial\,\mathbf{h}^{(l)}}{\partial\,\mathbf{a}^{(l)}} = \mathbf{D}^{(l)} \qquad \frac{\partial\,\mathbf{a}^{(l+1)}}{\partial\,\mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)} \qquad \frac{\partial\,L}{\partial\,\mathbf{a}^{(L+1)}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] = -\Delta^{(L+1)}$$

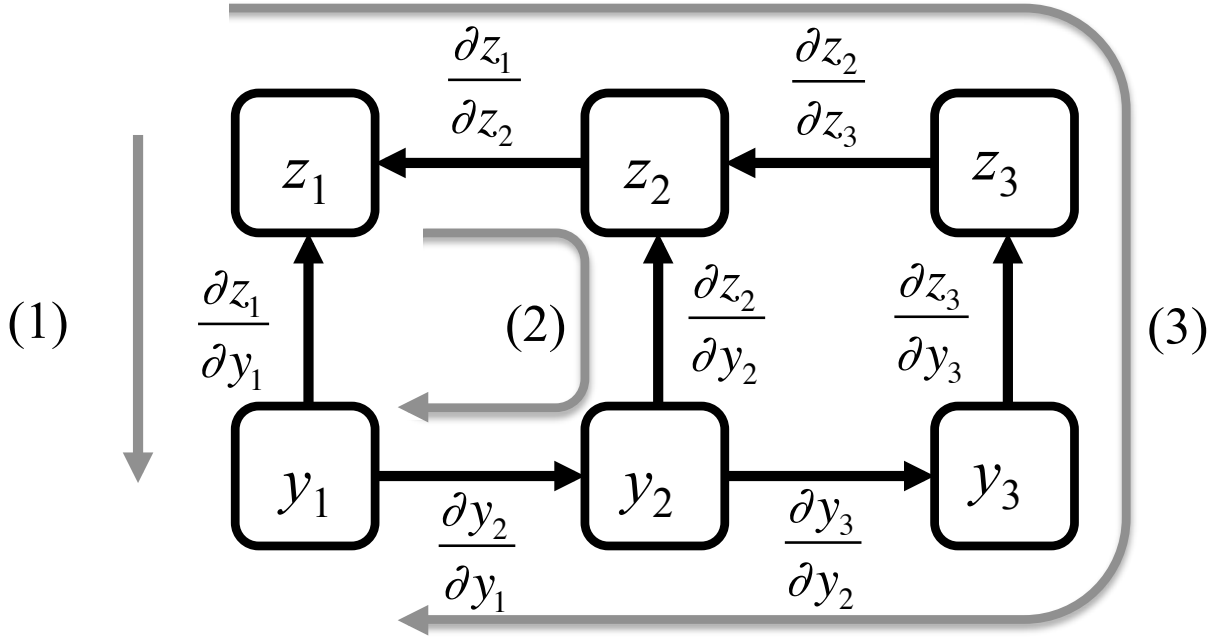
Visualizing backpropagation

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



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

Computation graphs



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

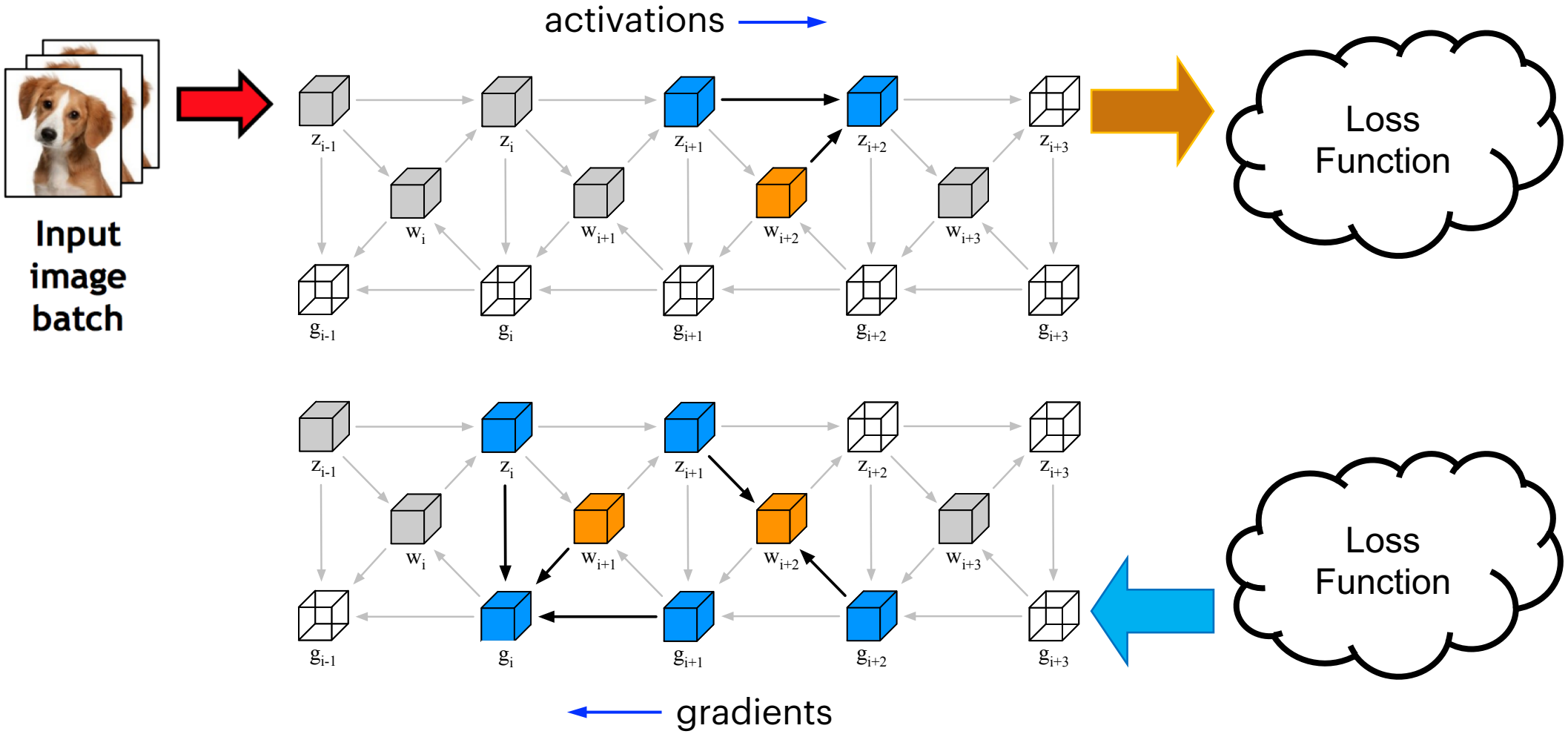
$$\frac{\partial z_1}{\partial y_1} = \underbrace{\frac{\partial z_1}{\partial y_1}}_{(1)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(2)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_3}\frac{\partial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

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

Bibliographic Notes & Further Reading

- Many neural network books (Haykin, 1994; Bishop, 1995; Ripley, 1996) do not formulate backpropagation in vector-matrix terms.
- However, recent online courses (e.g. by Hugo Larochelle), and Rojas (1996)'s text, do adopt this formulation, as we have done here

What are the data dependencies?



Simon Knowles, "Graphcore Intelligent Processing Unit (IPU)" Deep Learning at Supercomputer Scale Workshop, NIPS-17.
https://www.matroid.com/scaledml/2018/simon.pdf

Gradient Descent Variants

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

What is the Difference?

Gradient Descent Variants

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

What is the Difference?

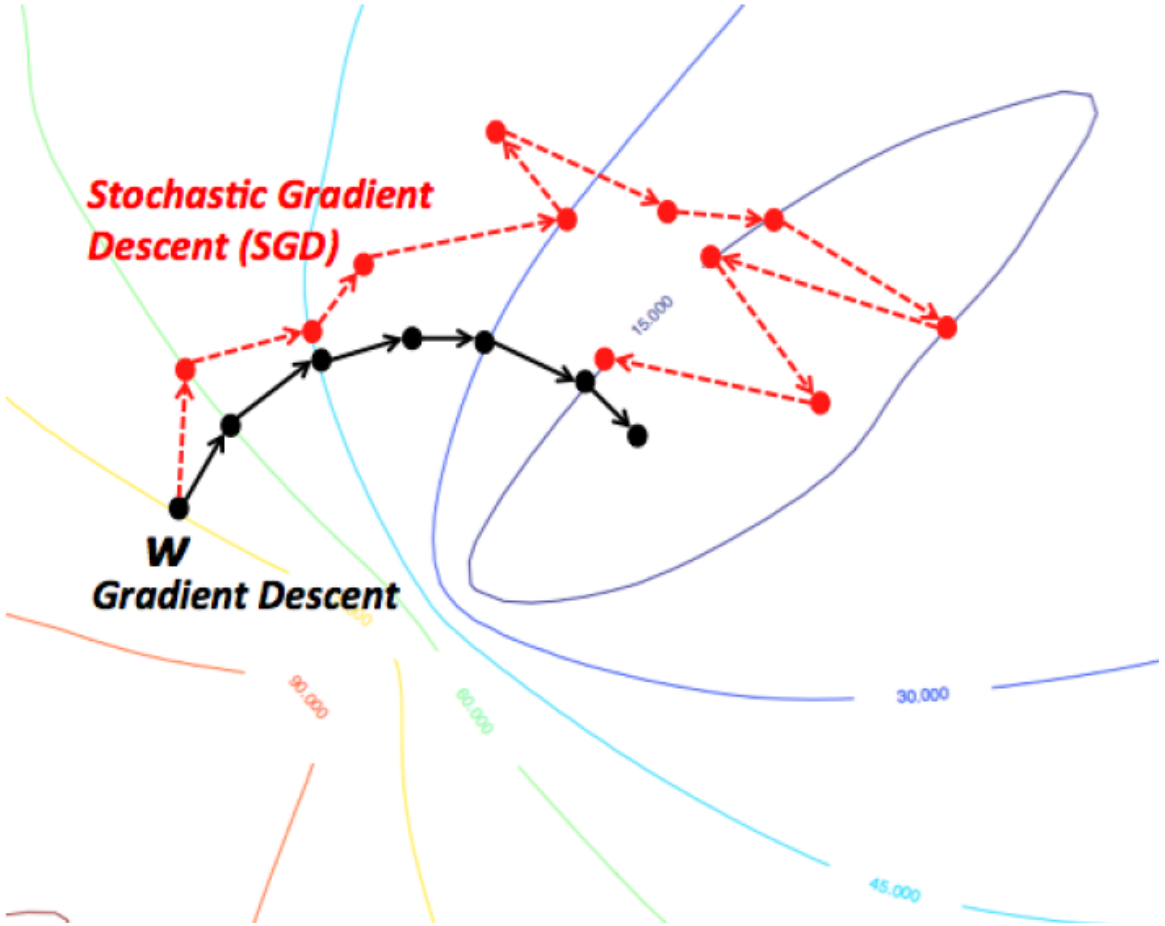
- The accuracy of the parameter update
- The time it takes to perform an update

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 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.



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

Mini Batch gradient descent

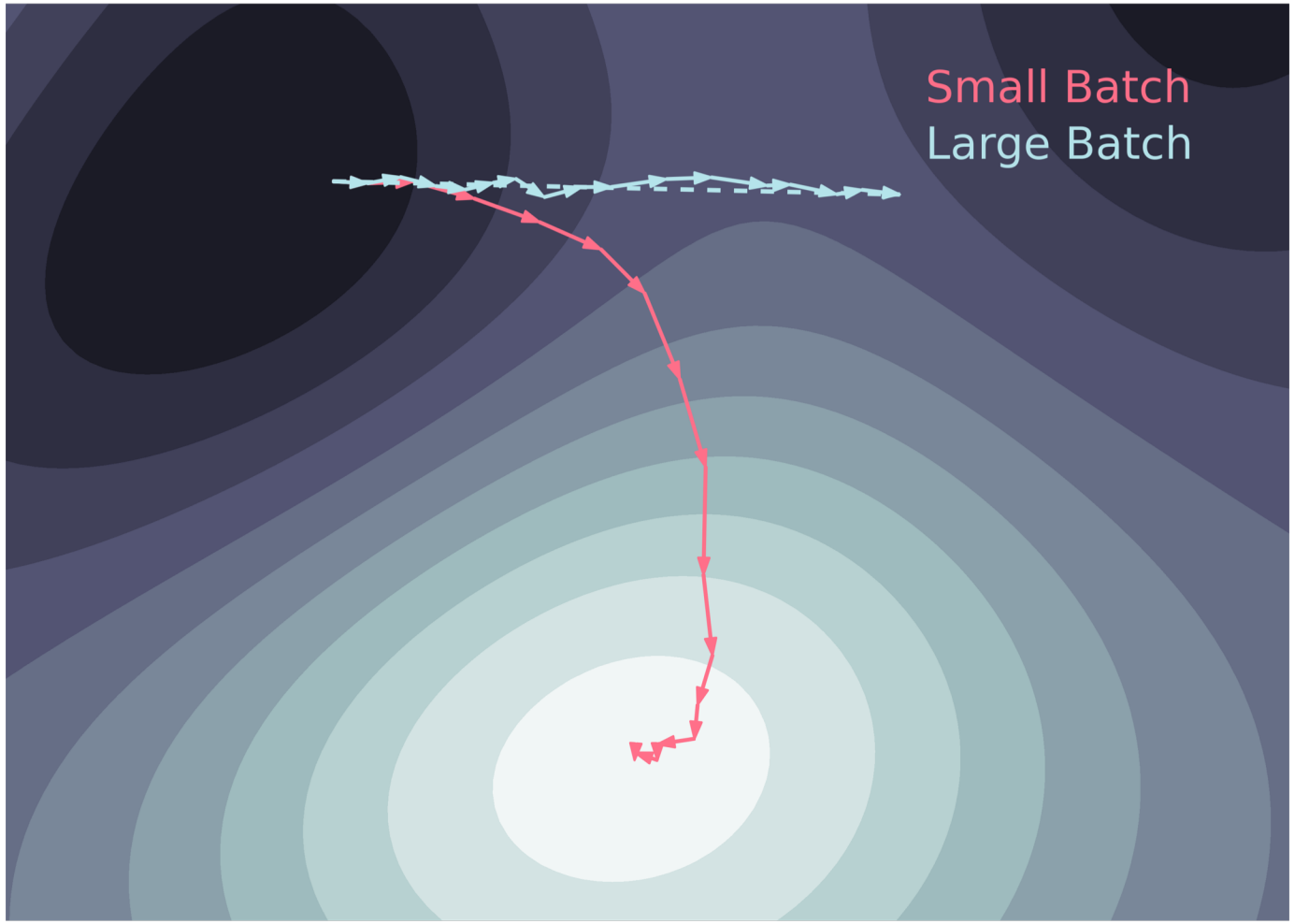
Advantages

- It reduces the variance of the parameter updates.
- More stable convergence.
- Make use of highly optimized matrix optimizations common to deep learning libraries that make computing the gradient very efficiently

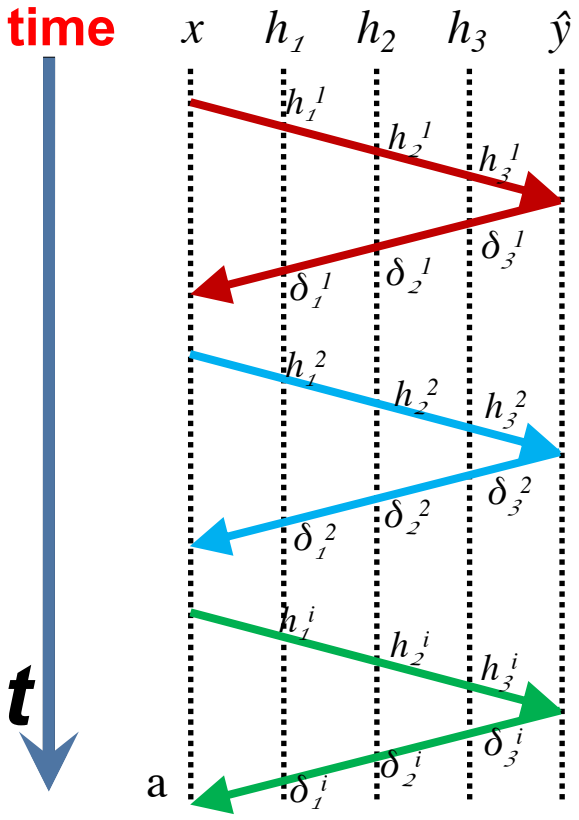
Disadvantages

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

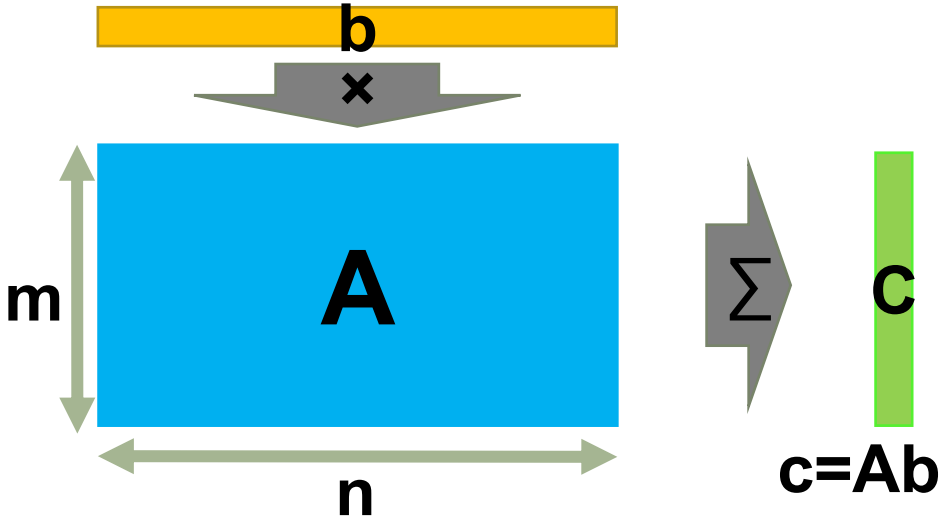
Large Batches Could Miss the Minimum



Stochastic Gradient Descent

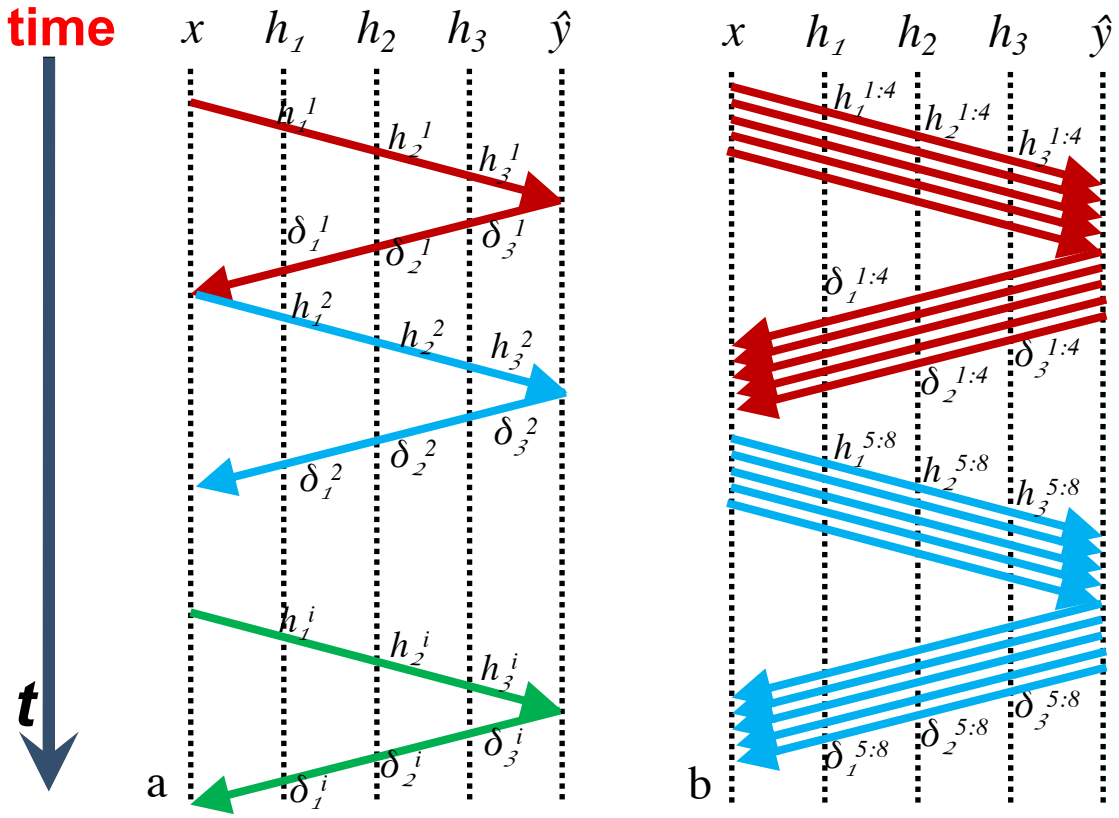


- GEMV
  - Inherently inefficient
  - Requirements
    - Broadcast (systolic /non-systolic)
    - Reduction



https://arxiv.org/abs/1706.00517

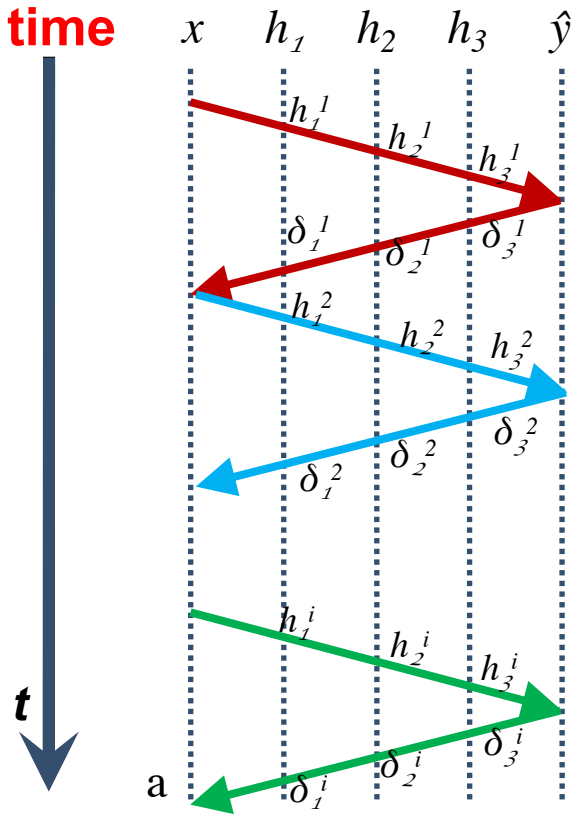
Mini-Batched Gradient Descent



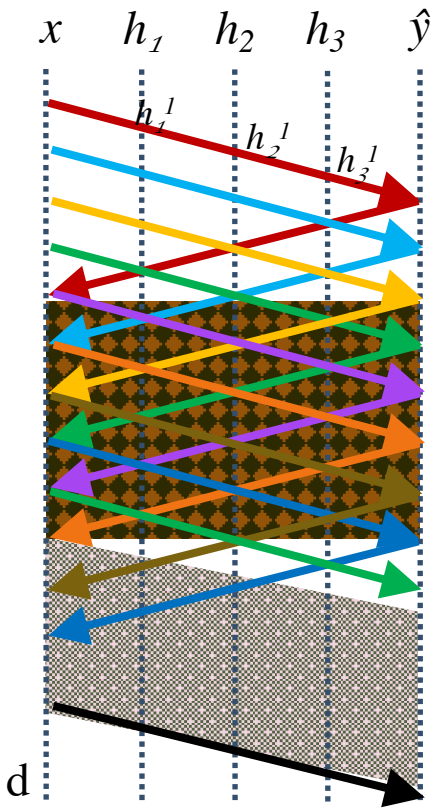
- Data parallelism
  - GEMV➔ GEMM
  - GEMM: Memory efficient kernel
  - # of weight updates / batch size

https://arxiv.org/abs/1706.00517

Pipelined Backpropagation

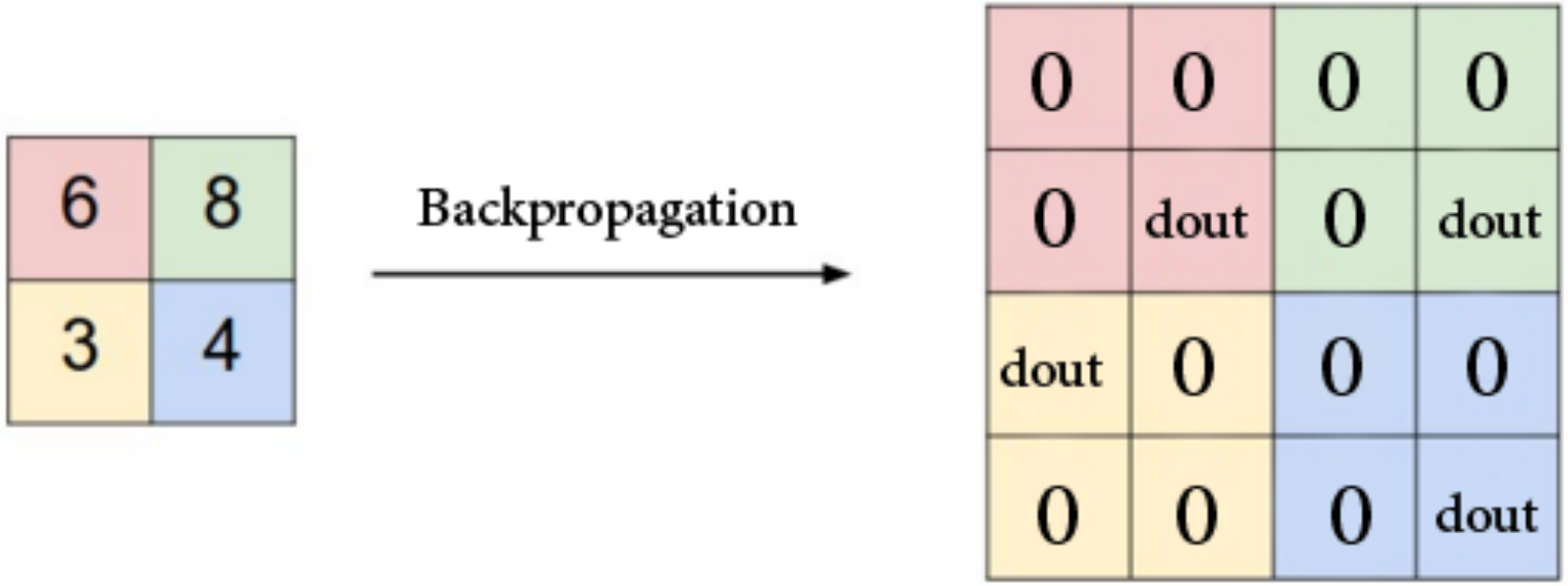


- Pipeline parallelization
  - Pipelining inputs
  - Layer locality
  - More efficient GEMVs
    - **Smaller reduction tree**
  - Weight temporal locality
    - Update and consume immediately



https://arxiv.org/abs/1706.00517

Pooling Layer

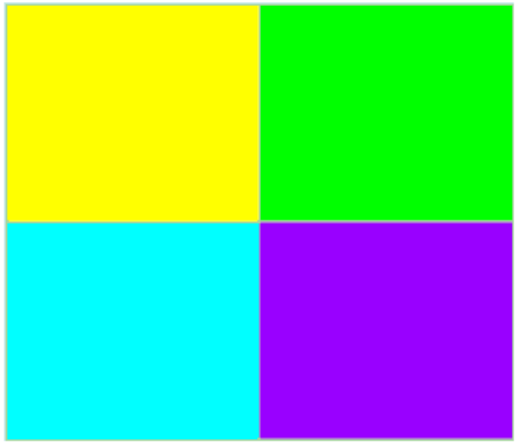


In the process we replace the maximum values before max pooling with 1 and set all the non maximum values to zero then use the **Chain rule** to multiply the gradient by them.

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

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

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

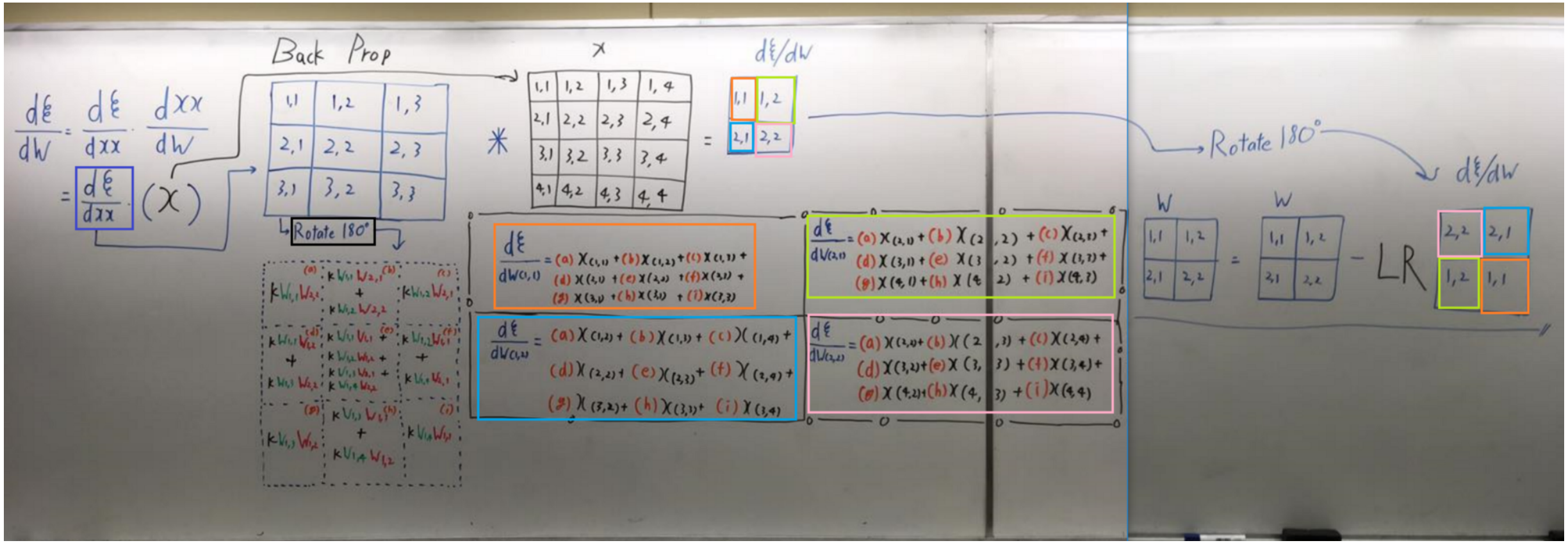
$$\partial W_{12} = X_{12}\partial h_{11} + X_{13}\partial h_{12} + X_{22}\partial h_{21} + X_{23}\partial h_{22}$$

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

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

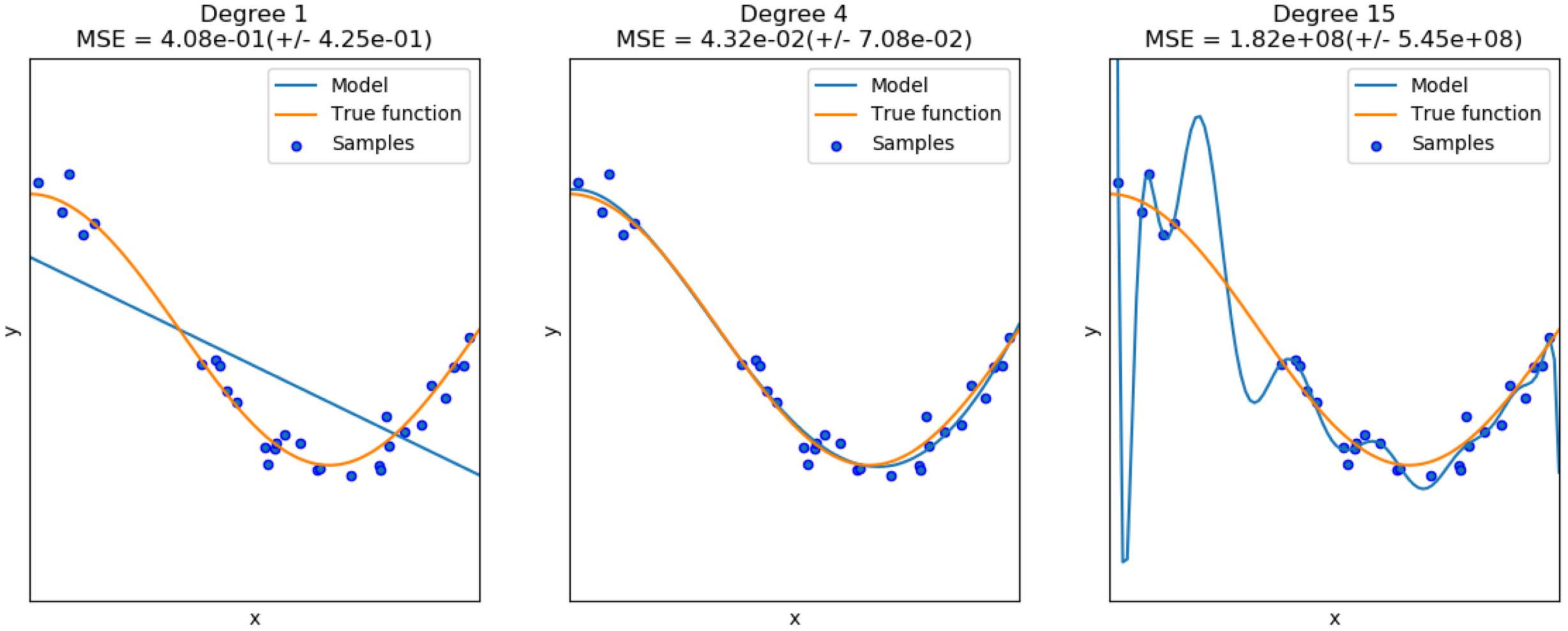
https://mc.ai/demystifying-convolutional-neural-networks/

Jae did it with cool details



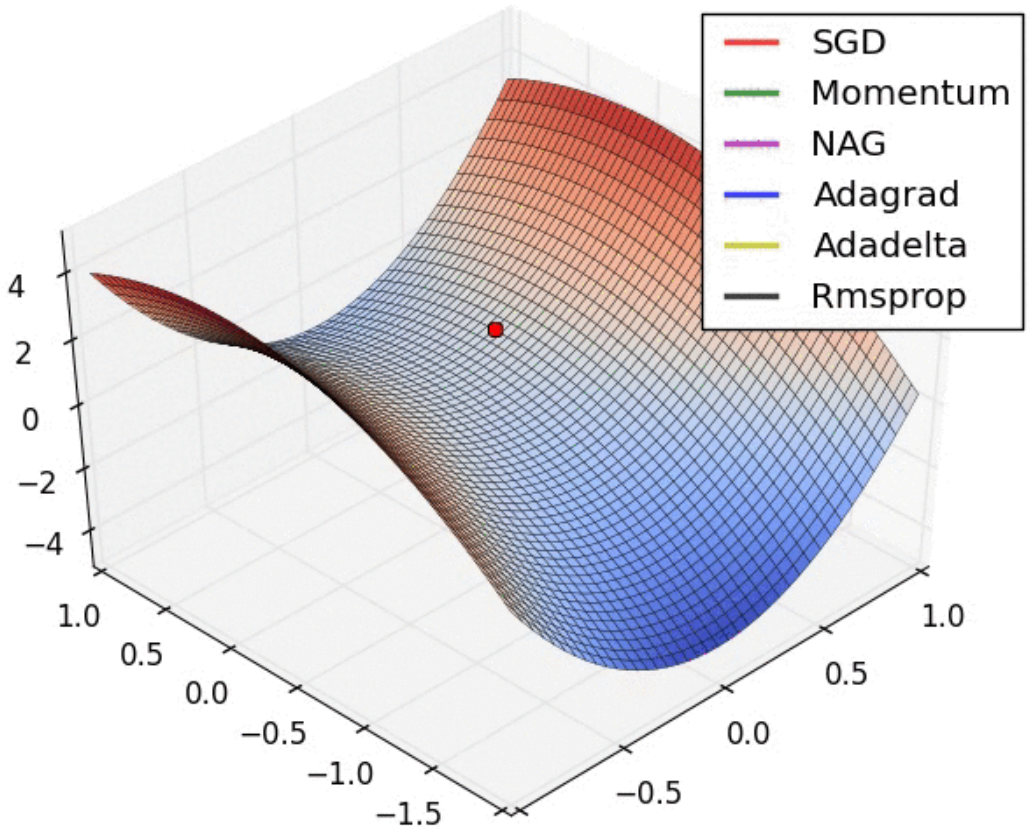
https://towardsdatascience.com/only-numpy-understanding-back-propagation-for-transpose-convolution-in-multi-layer-cnn-with-c0a07d191981

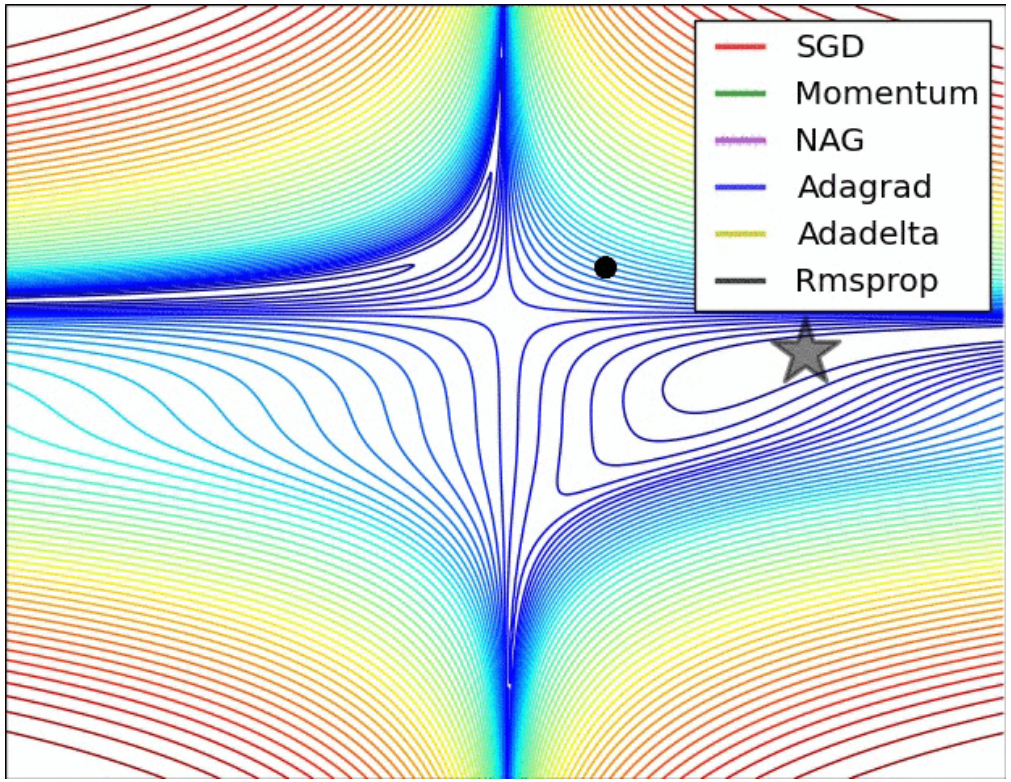
Generalization and overfitting



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

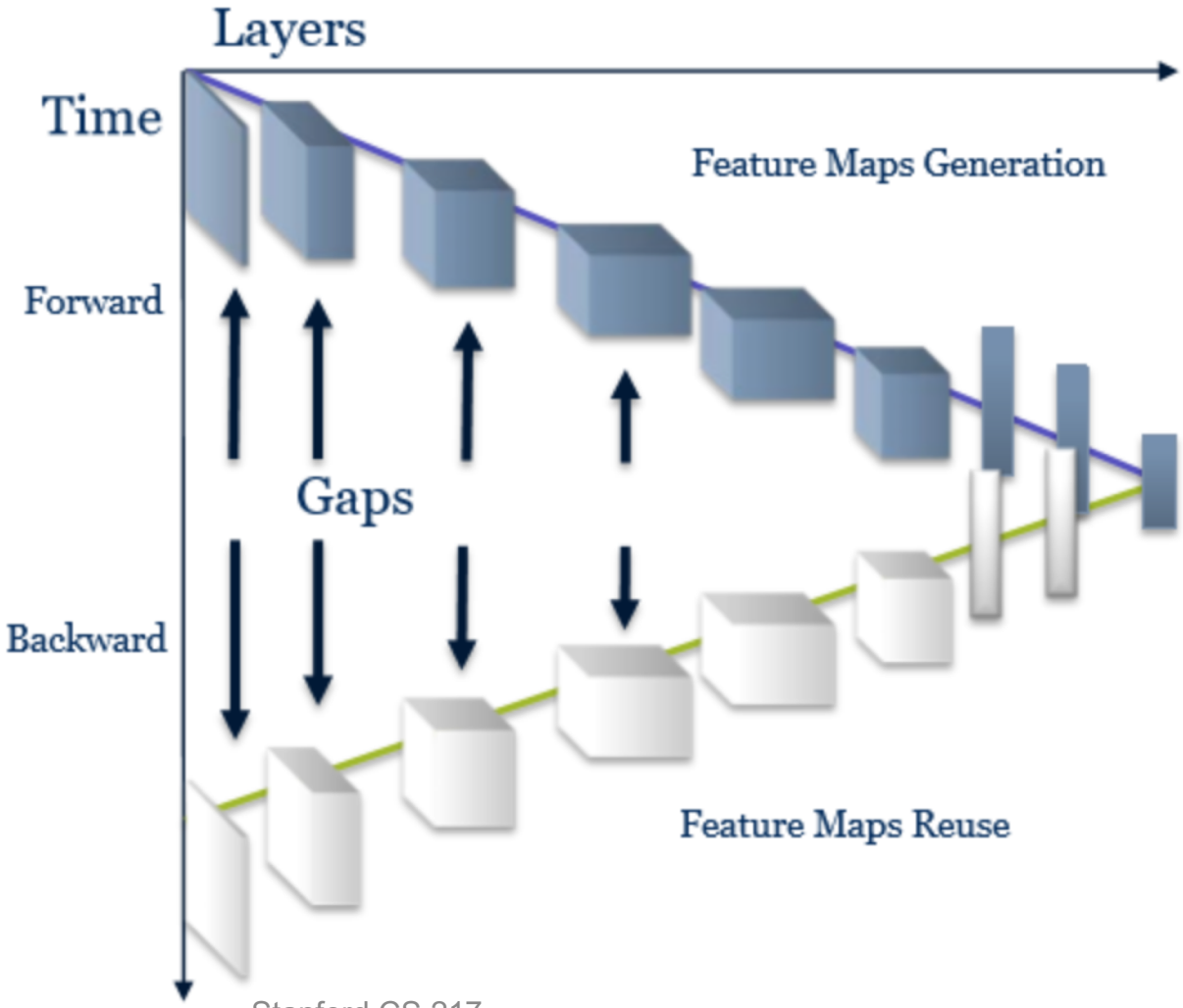
SGD Optimizers





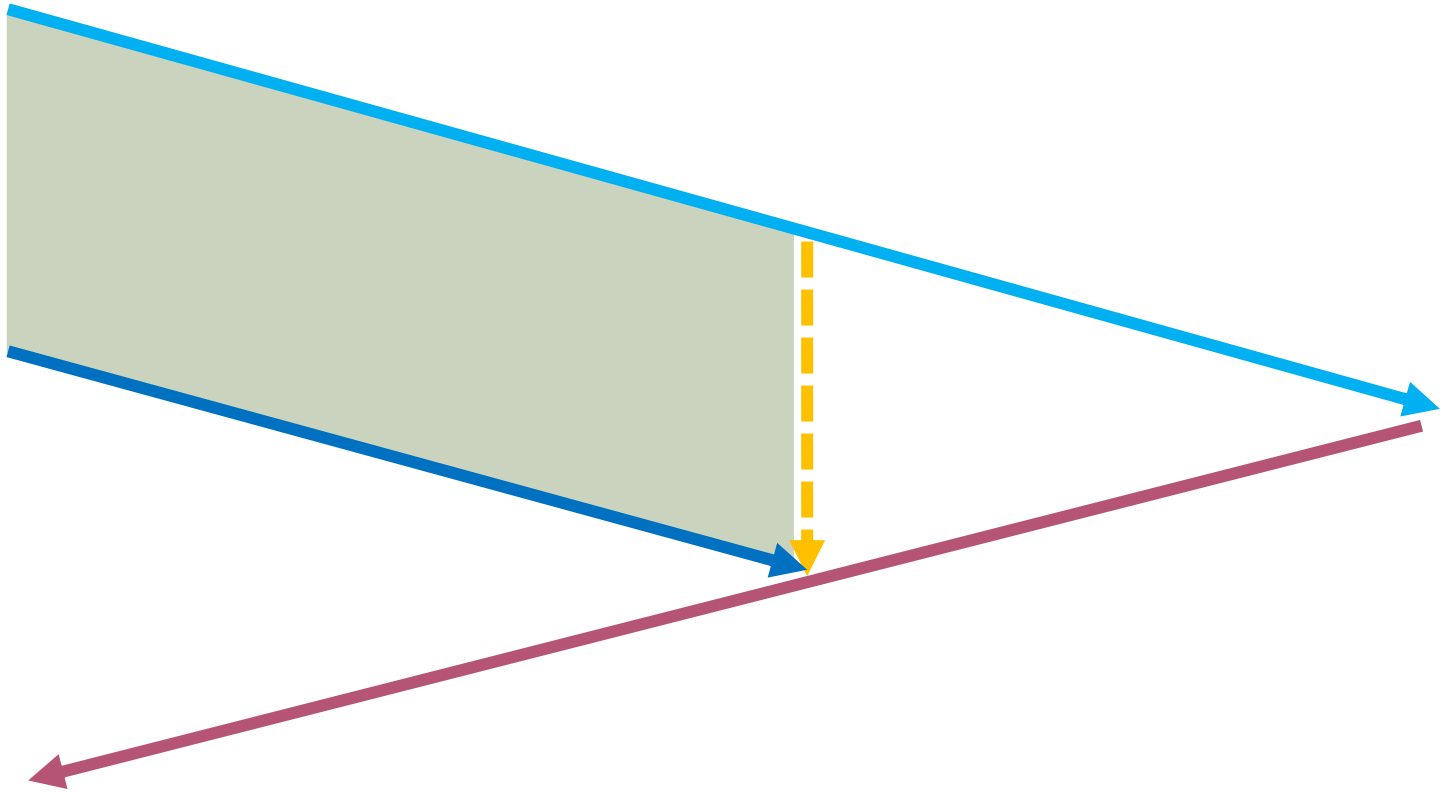
Sebastian Ruder, "An overview of gradient descent optimization algorithms" arXiv:1609.04747 15 Jun 2017
https://arxiv.org/pdf/1609.04747.pdf

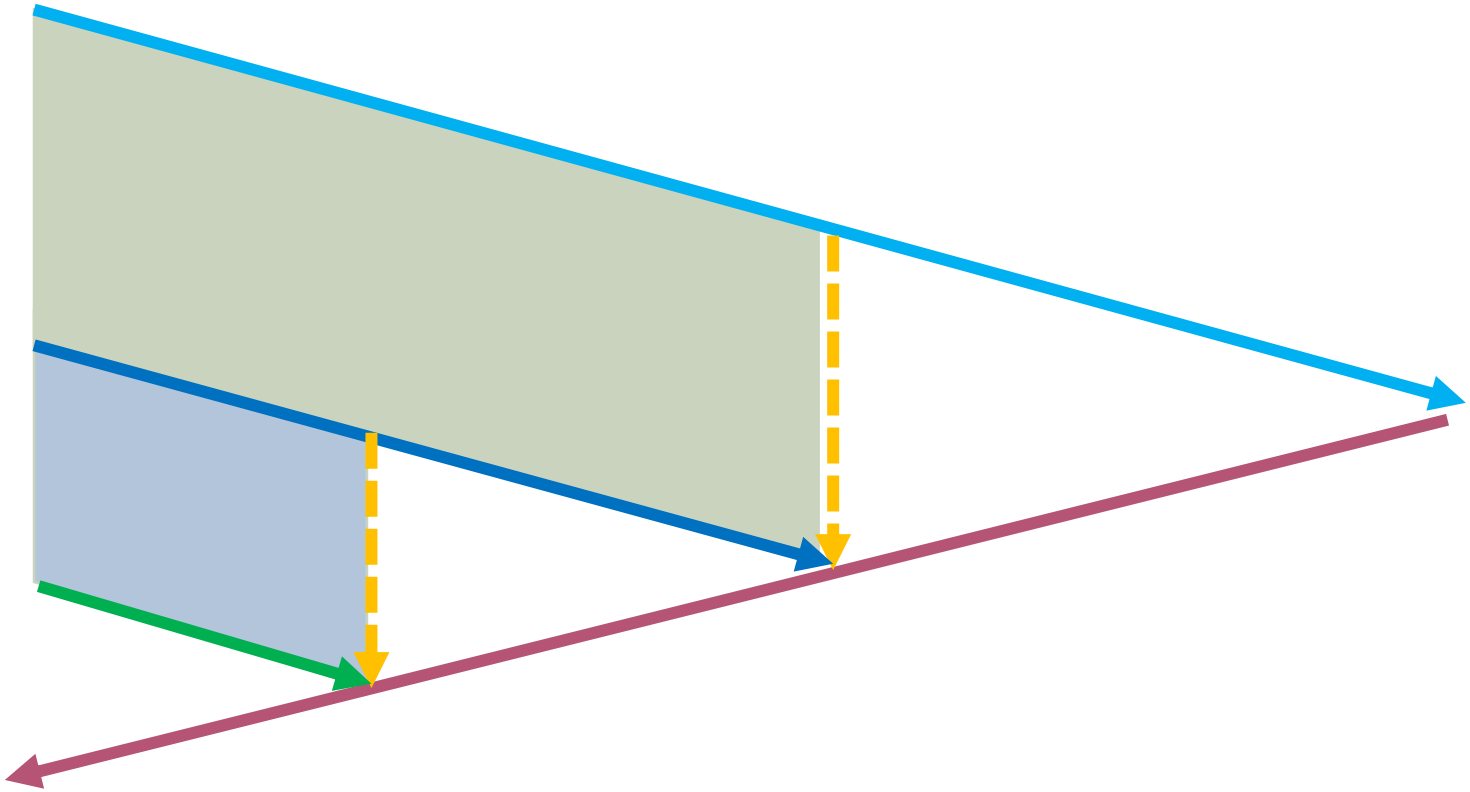
Storage is Quadratic Function of Depth



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

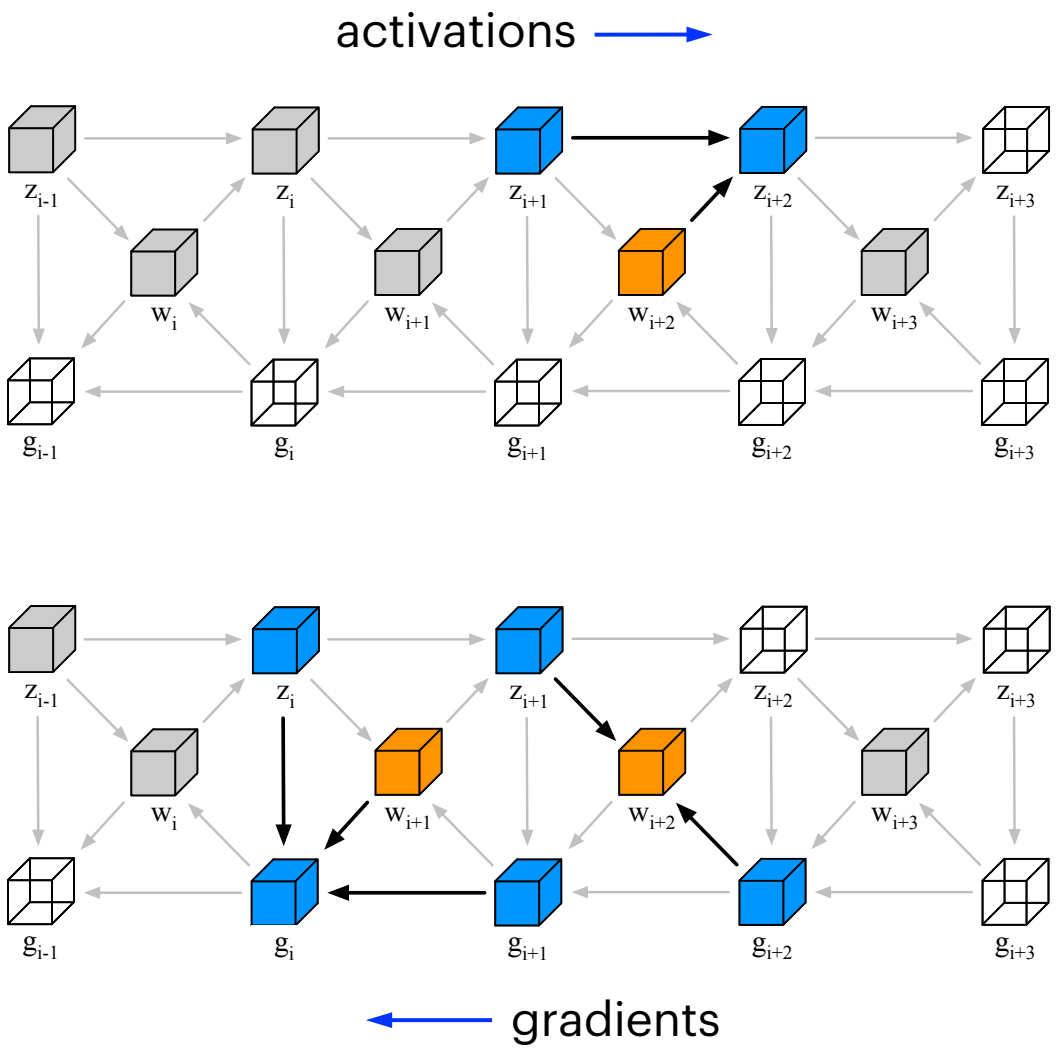
Reverse Checkpointing





Recompute WHAT YOU Can't Remember

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



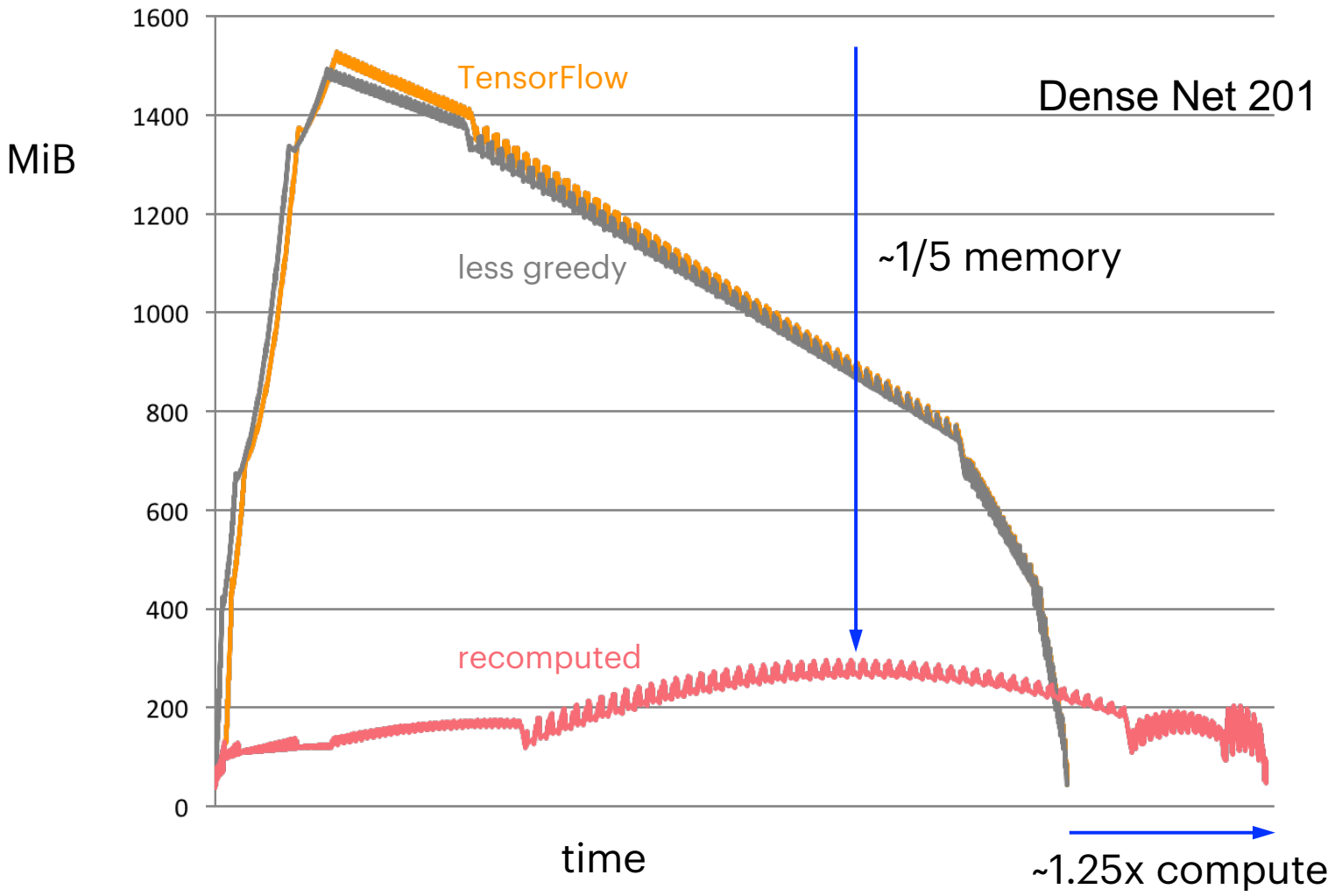
Simon Knowles, "Graphcore Intelligent Processing Unit (IPU)" Deep Learning at Supercomputer Scale Workshop, NIPS-17. https://www.matroid.com/scaledml/2018/simon.pdf

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.



Simon Knowles, "Graphcore Intelligent Processing Unit (IPU)" Deep Learning at Supercomputer Scale Workshop, NIPS-17. https://www.matroid.com/scaledml/2018/simon.pdf

Batch Normalization

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

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

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

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

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

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

Batch Normalization

The output of the previous layer in the neural net

The average of h

A regularization parameter

$$\mathrm{BN}(\mathbf{h};\gamma,\beta) = \beta + \gamma \frac{\mathbf{h} - \widehat{\mathbb{E}}(\mathbf{h})}{\sqrt{\widehat{\mathrm{Var}}(\mathbf{h}) + \epsilon}}$$

The variance of h

Batch Normalize

This piece basically normalizes h to have a mean and variance of 1

We use these to scale and shift the (normalized) distribution. Both are parameters that are learned.

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: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

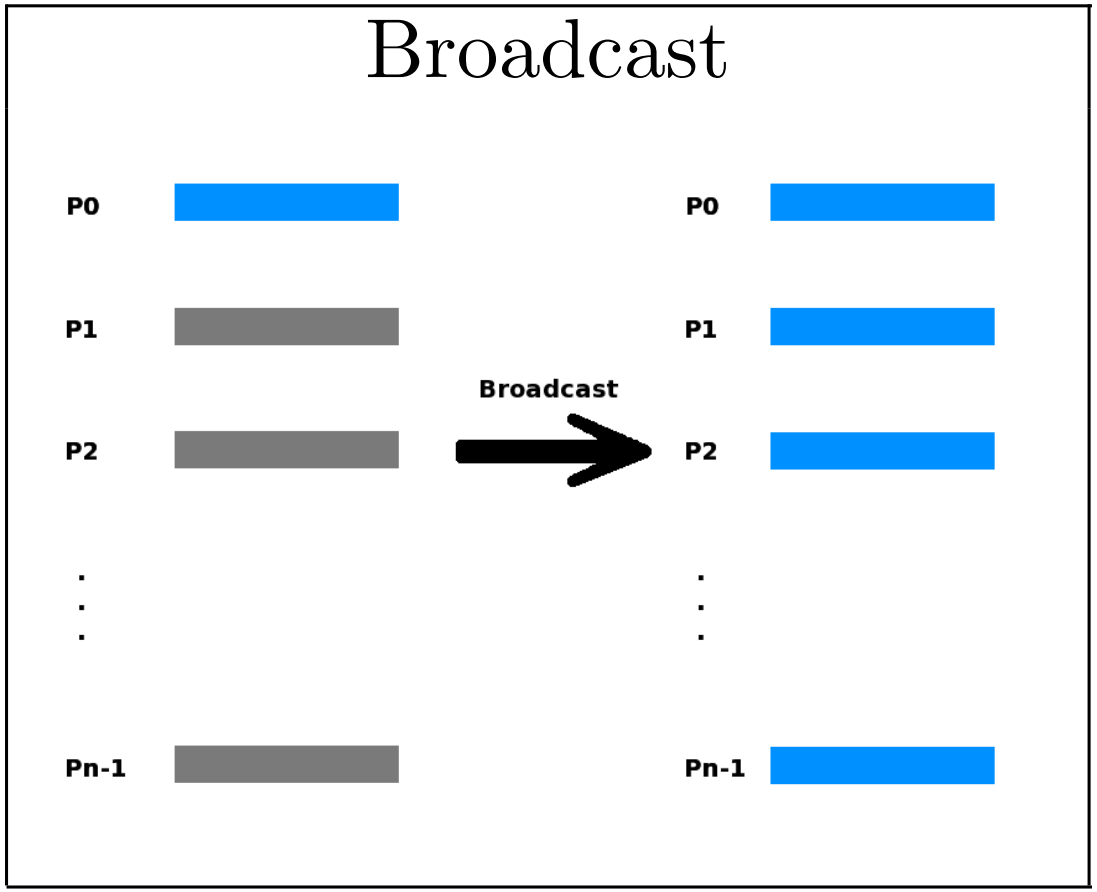
What Do We Need to Support?

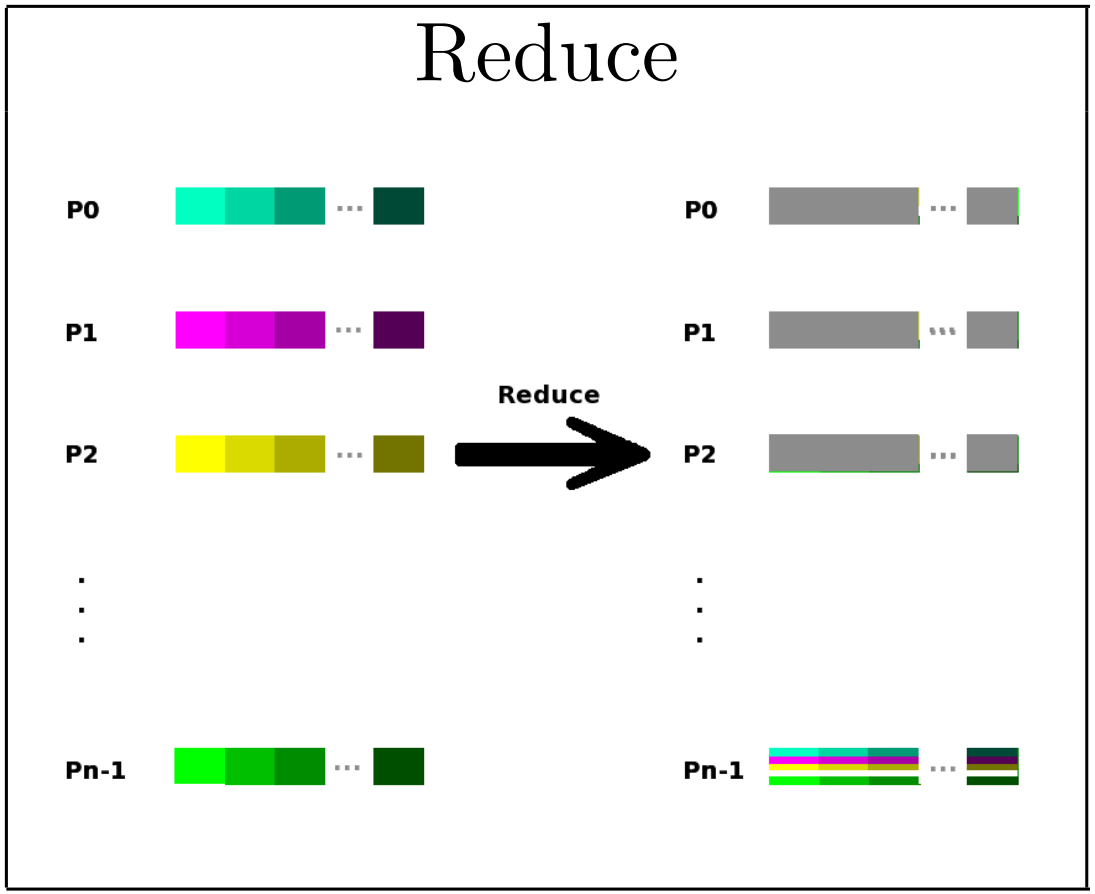
- **GEMM:** General Matrix Matrix multiplication
- **GEMV:** General Matrix Vector multiplication
- **Collective communications**
  - Gather
  - Reduce
  - All gather
  - All reduce
  - Broadcast
  - All-to-All

Collective Communications

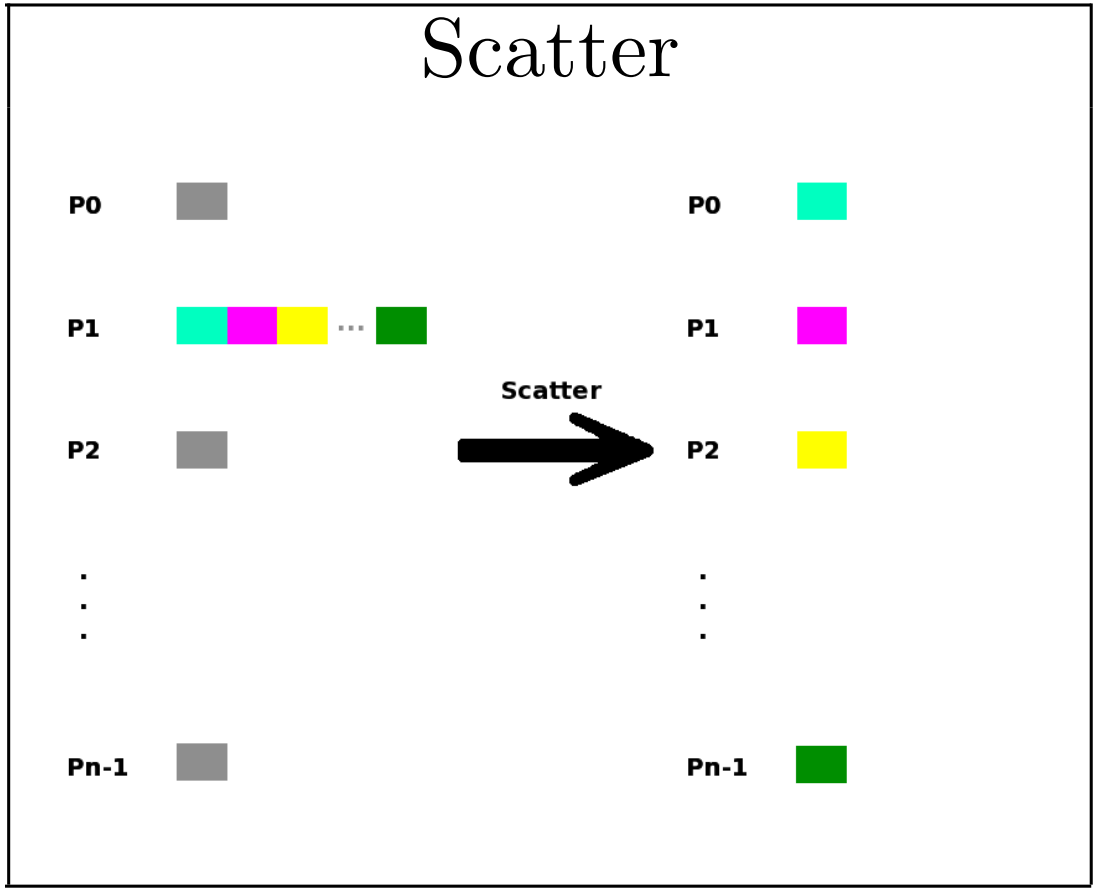
- Collective operations are called by all processes within a communicator
- Various algorithms to implement them based on
  - Latency
  - BW
- Can Combine two or more to create an algorithm for a third collective

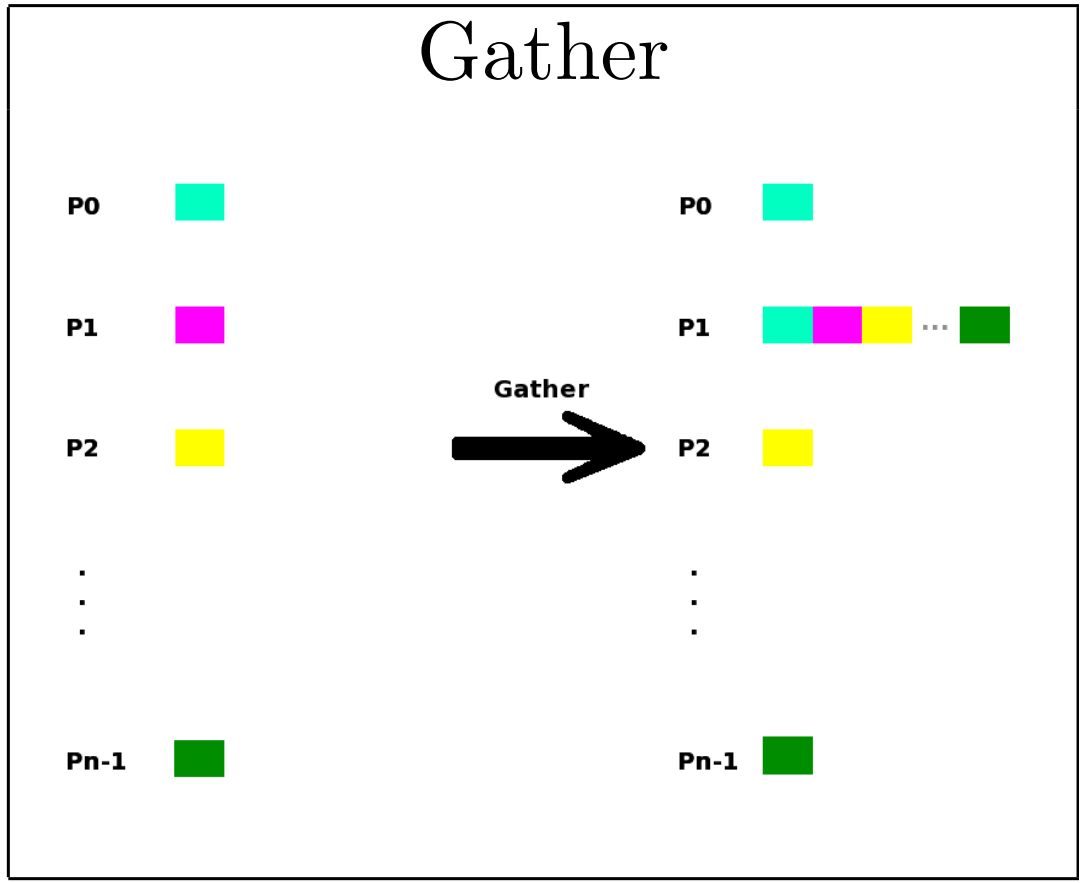
Operation	Before				After			
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Permute	x_0	x_1	x_2	x_3	x_1	x_0	x_3	x_2
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Broadcast	x				x	x	x	x
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Reduce(-to-one)	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$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


Broadcast
P0
P1
P2
Pn-1
Broadcast
P0
P1
P2
Pn-1


Reduce
P0
P1
P2
Pn-1
Reduce
P0
P1
P2
Pn-1

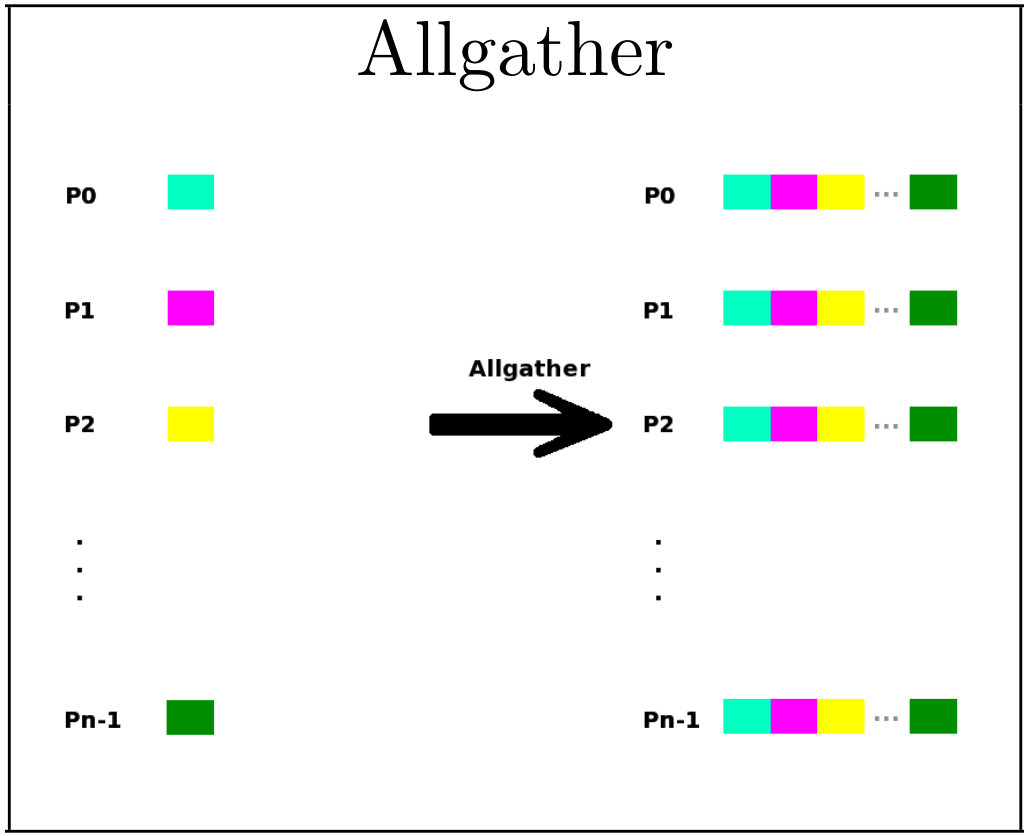
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Scatter	x_0 x_1 x_2 x_3				x_0	x_1	x_2	x_3
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Gather	x_0	x_1	x_2	x_3	x_0 x_1 x_2 x_3			

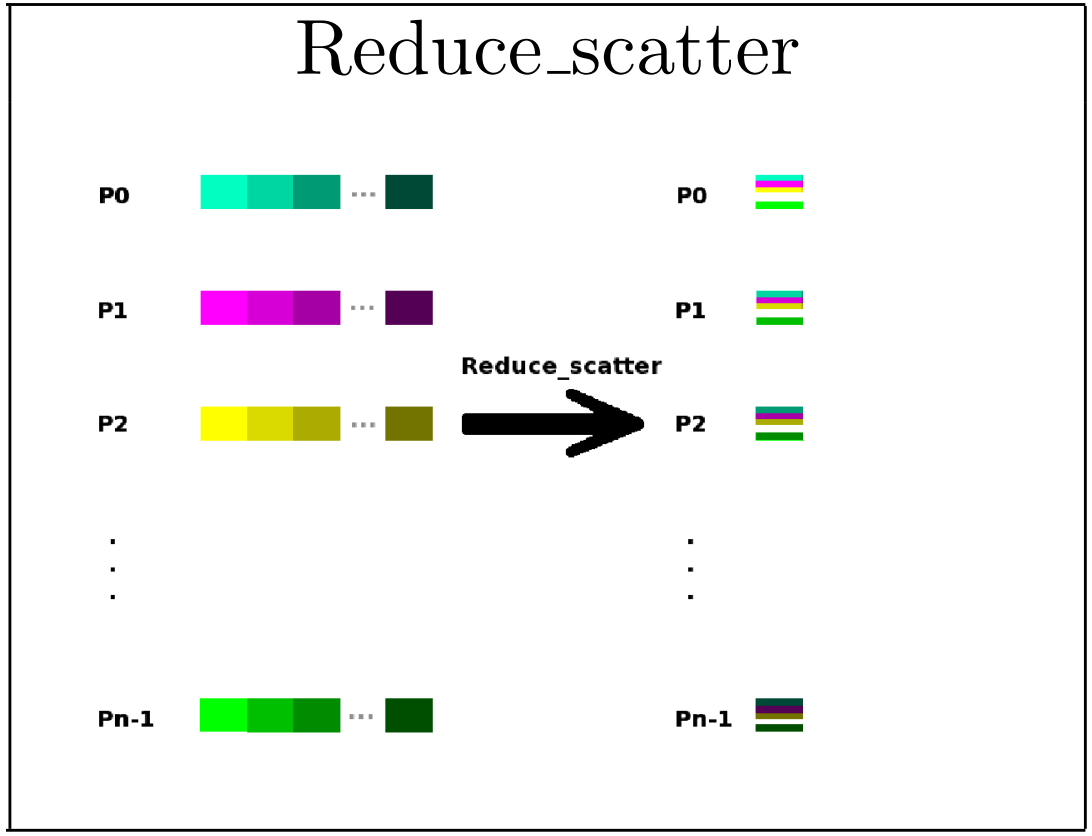

Scatter
P0
P1
P2
Pn-1
Scatter
P0
P1
P2
Pn-1


Gather
P0
P1
P2
Pn-1
Gather
P0
P1
P2
Pn-1

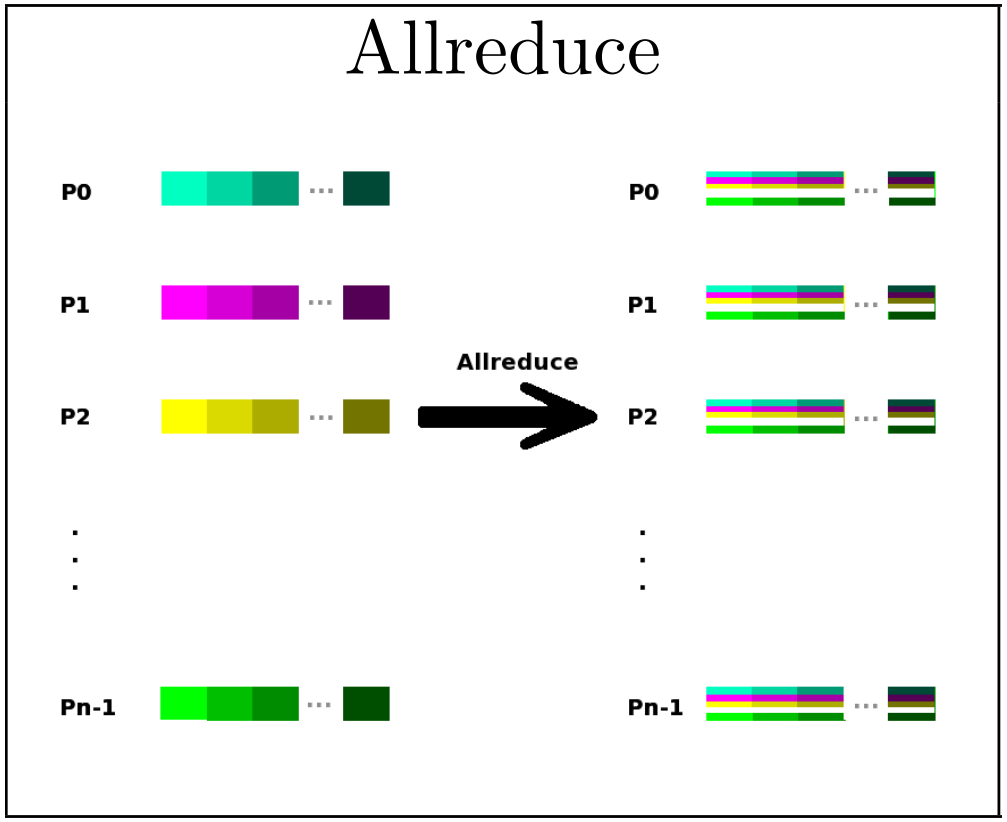
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Allgather	x_0				x_0	x_0	x_0	x_0
		x_1			x_1	x_1	x_1	x_1
			x_2		x_2	x_2	x_2	x_2
				x_3	x_3	x_3	x_3	x_3

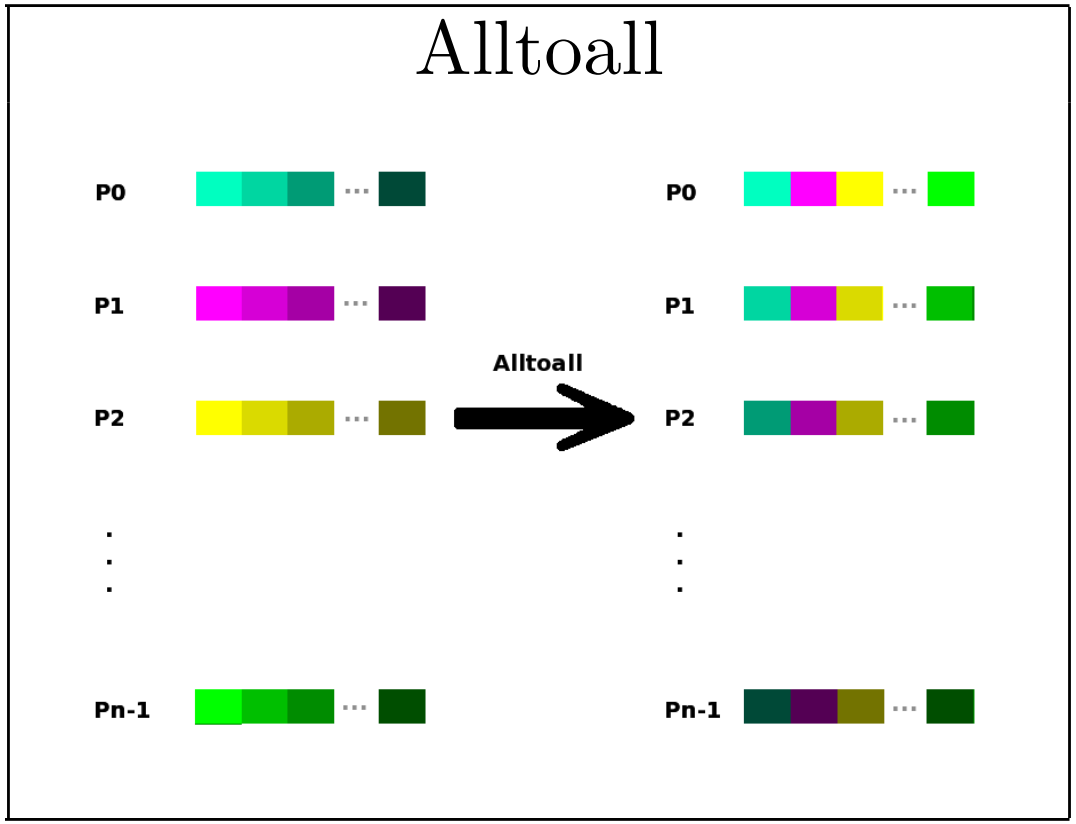
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Reduce-scatter	$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$	$\sum_j x_0^{(j)}$			
	$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$		$\sum_j x_1^{(j)}$		
	$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$			$\sum_j x_2^{(j)}$	
	$x_3^{(0)}$	$x_3^{(1)}$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


Allgather
P0
P1
P2
Pn-1
Allgather
P0
P1
P2
Pn-1


Reduce_scatter
P0
P1
P2
Pn-1
Reduce_scatter
P0
P1
P2
Pn-1

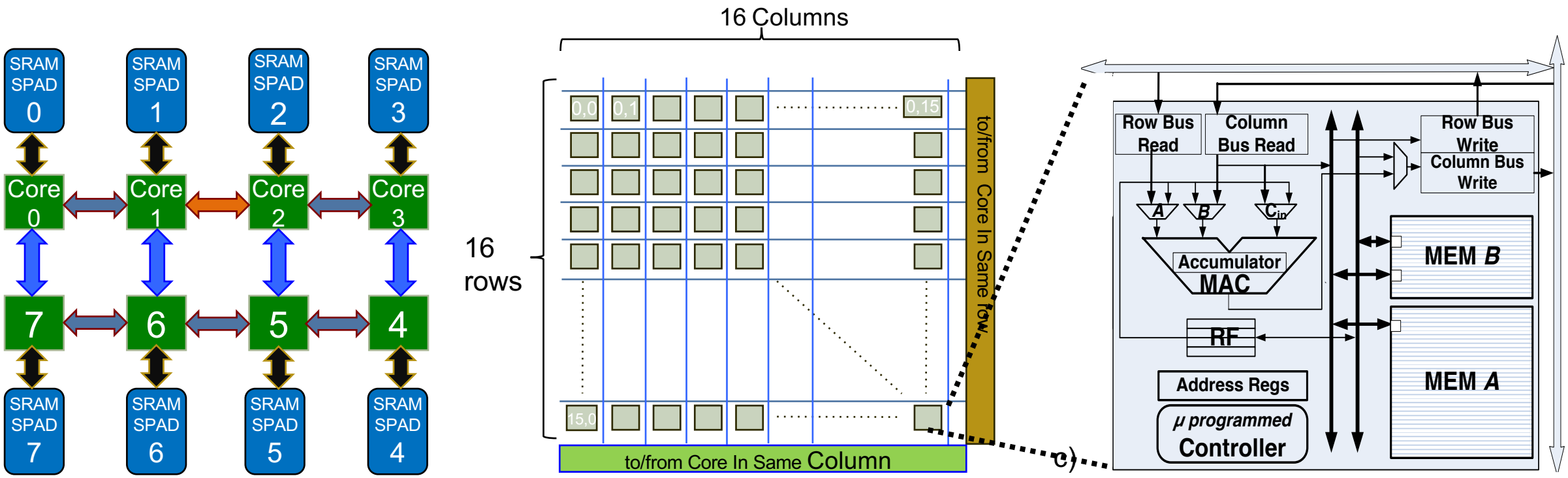
Allreduce	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$
All-to-all	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$	$x_0^{(0)}$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


Allreduce
P0
P1
P2
Pn-1
Allreduce
P0
P1
P2
Pn-1


Alltoall
P0
P1
P2
Pn-1
Alltoall
P0
P1
P2
Pn-1

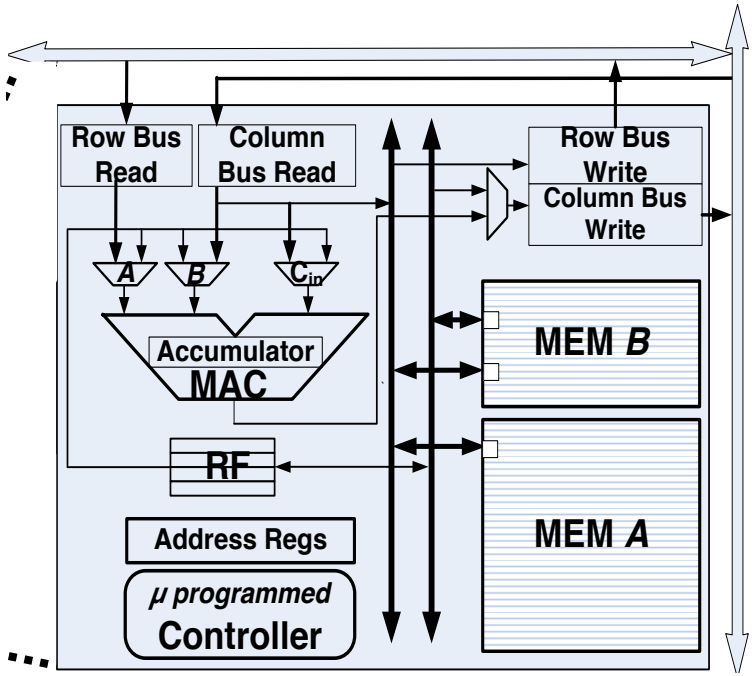
Operation	personalized?	small messages	large messages (segmented)
Broadcast	no	Flat/Binary/N-ary/Binomial Tree	Piplelined/Double/Split Binary Tree, Chain, Van de Geijn Algorithm, HW specific multicast
Barrier	no	Flat/Binary/Binomial Tree, Dissemination (butterfly), Tournament	
Reduce	no	Flat/Binary/N-ary/Binomial 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		

CATERPILLAR Architecture



CATERPILLAR Architecture

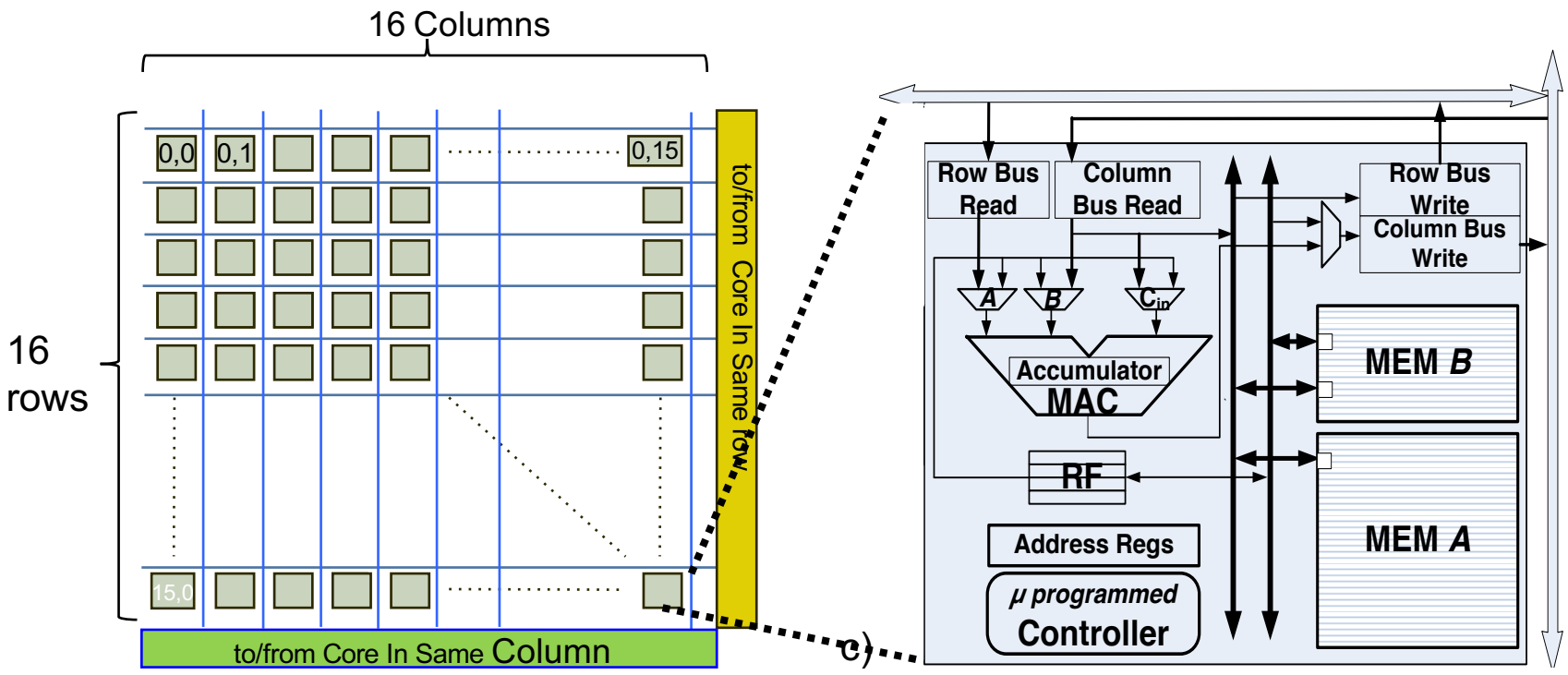
- PE
  - Native Support for Inner Product
  - 3 Levels of memory hierarchy
    - Accumulator
    - Mem B (2 ports)
    - Mem A (1 port)
  - Distributed Memory Programming Model
  - State Machine Reprogrammable



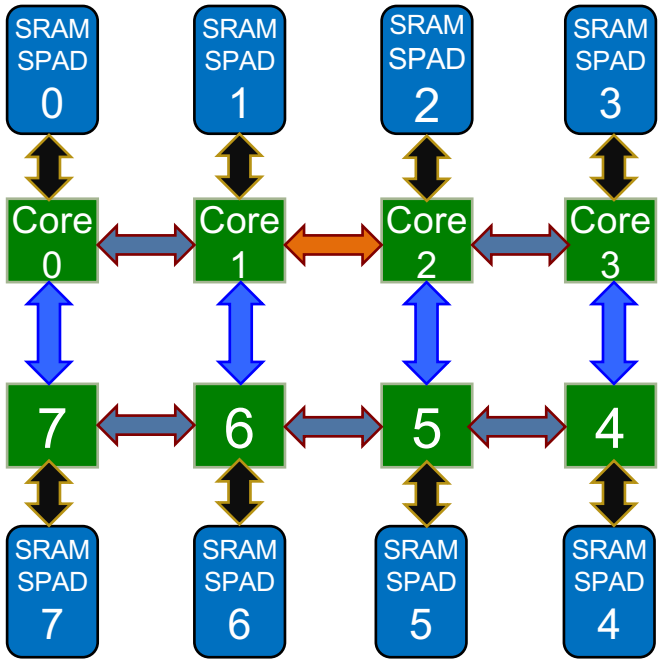
The Linear Algebra Core PE

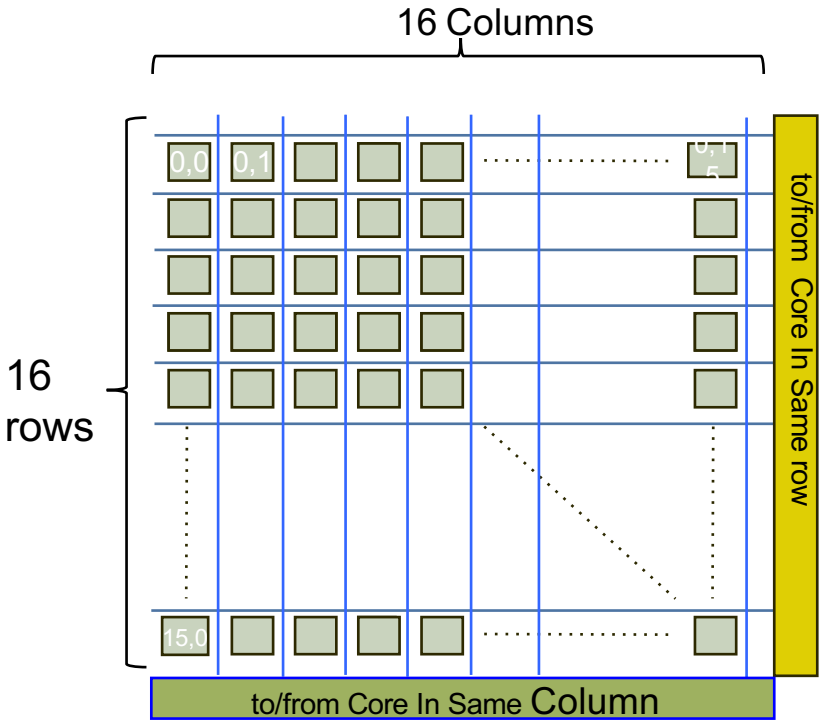
CATERPILLAR Architecture

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



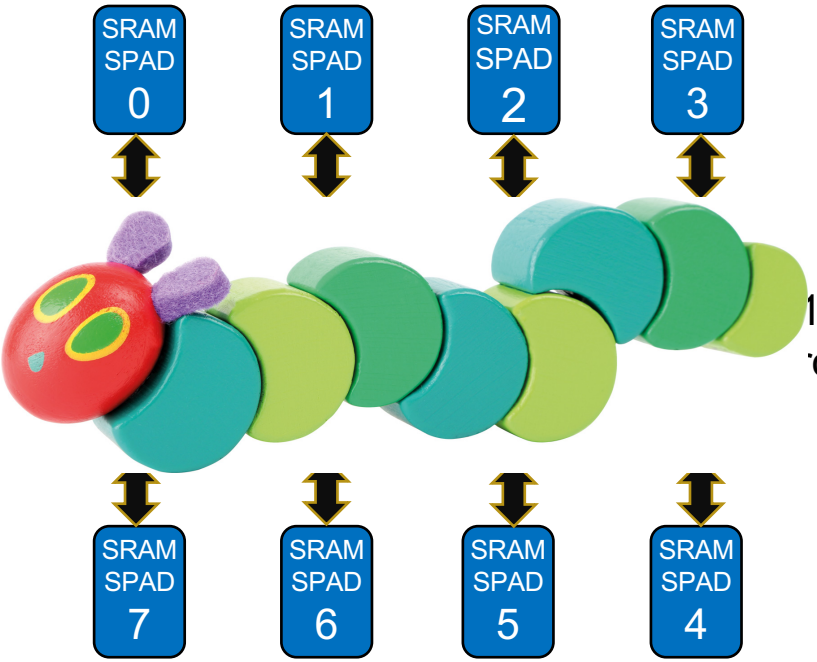
CATERPILLAR Architecture

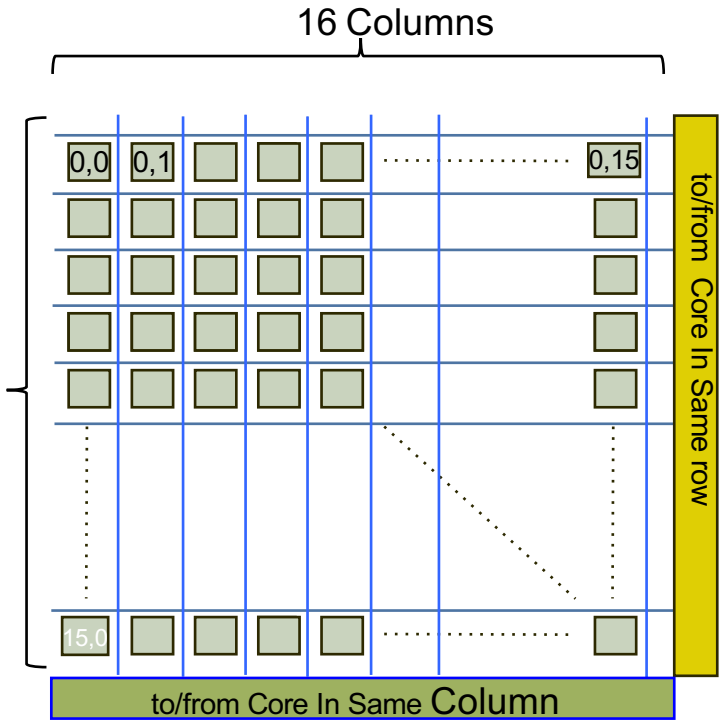




- Multicore
  - Ring of Cores
  - Reconfigurable
- Support for Collective Comms
  - All gather
  - Reduce
  - All Reduce
  - Systolic/Parallel

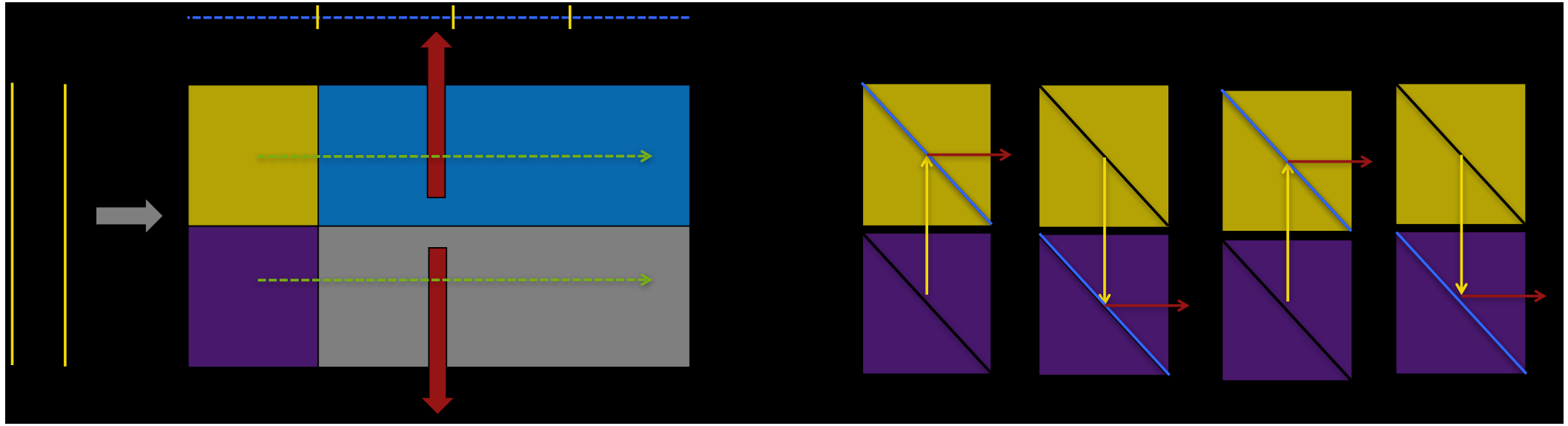
CATERPILLAR Architecture



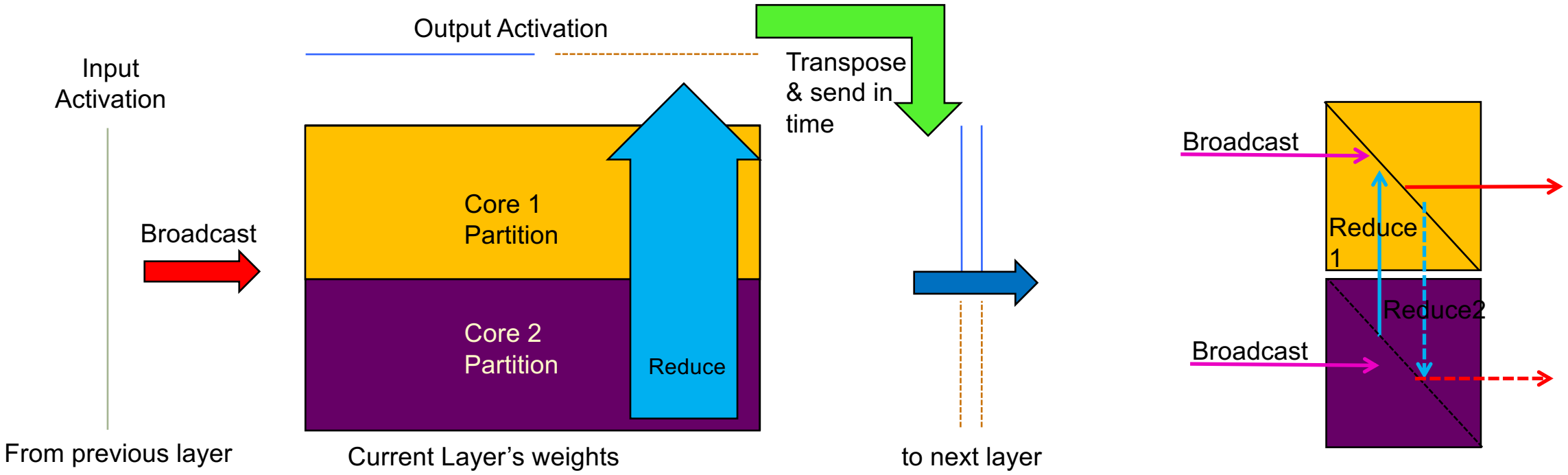


- Multicore
  - Ring of Cores
  - Reconfigurable
- Support for Collective Comms
  - All gather
  - Reduce
  - All Reduce
  - Systolic/Parallel

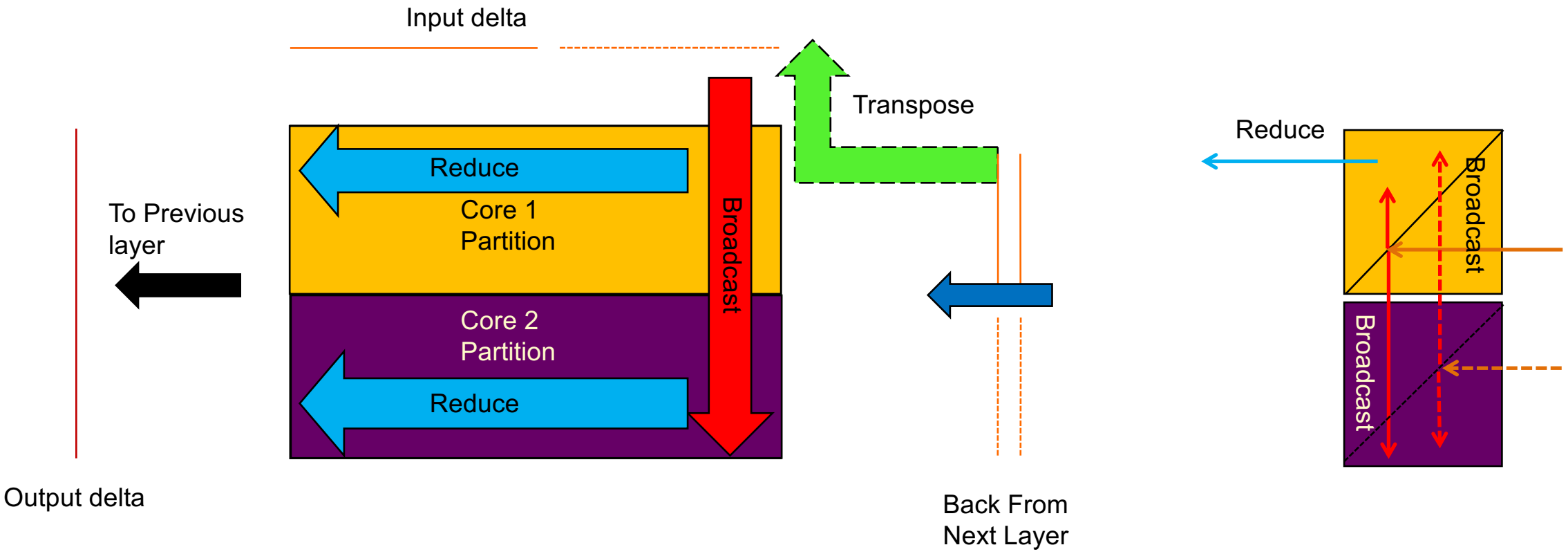
GEMV



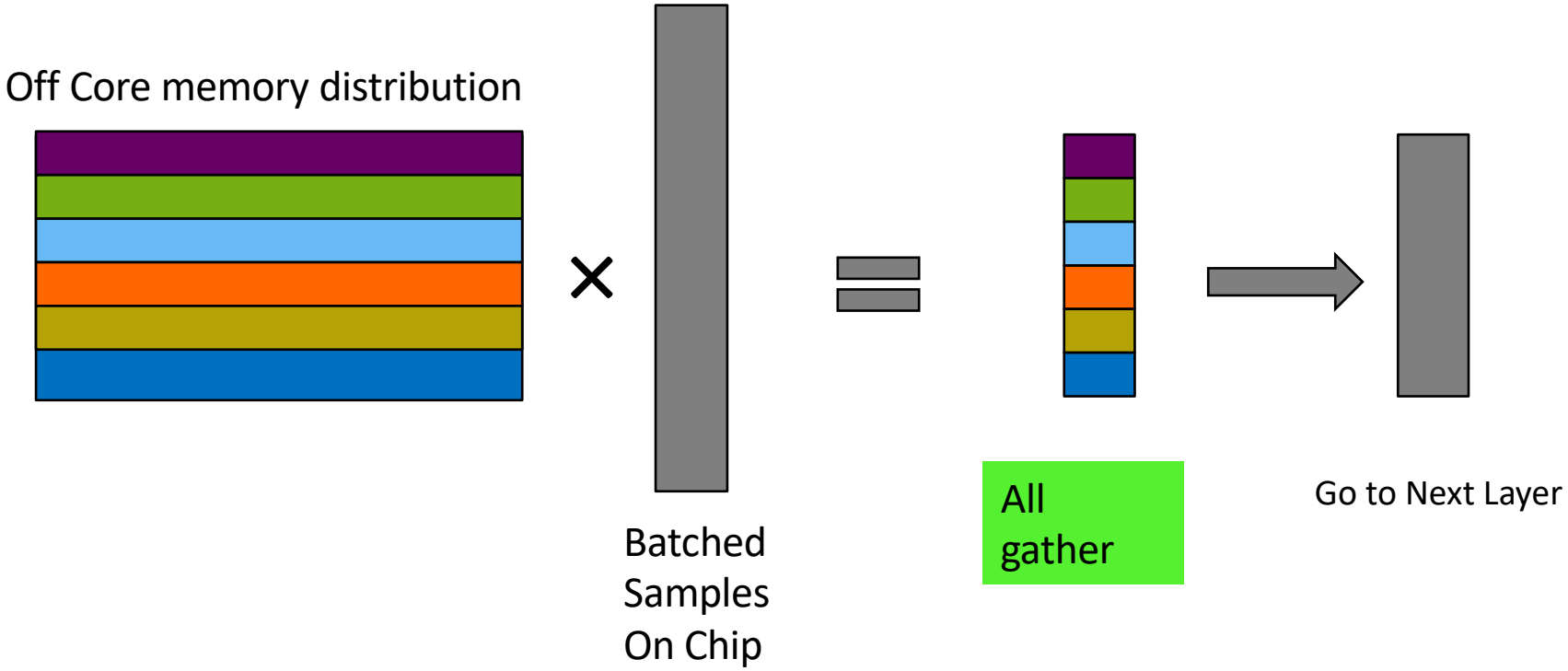
Forward Path



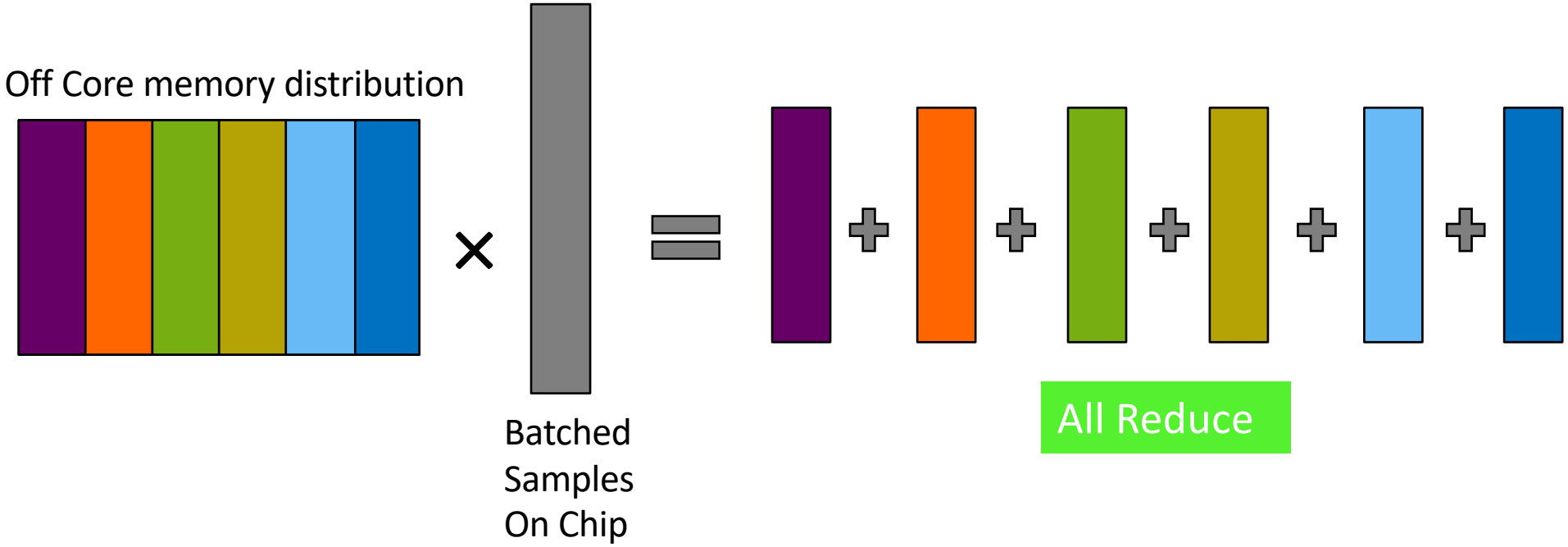
Delta Path


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

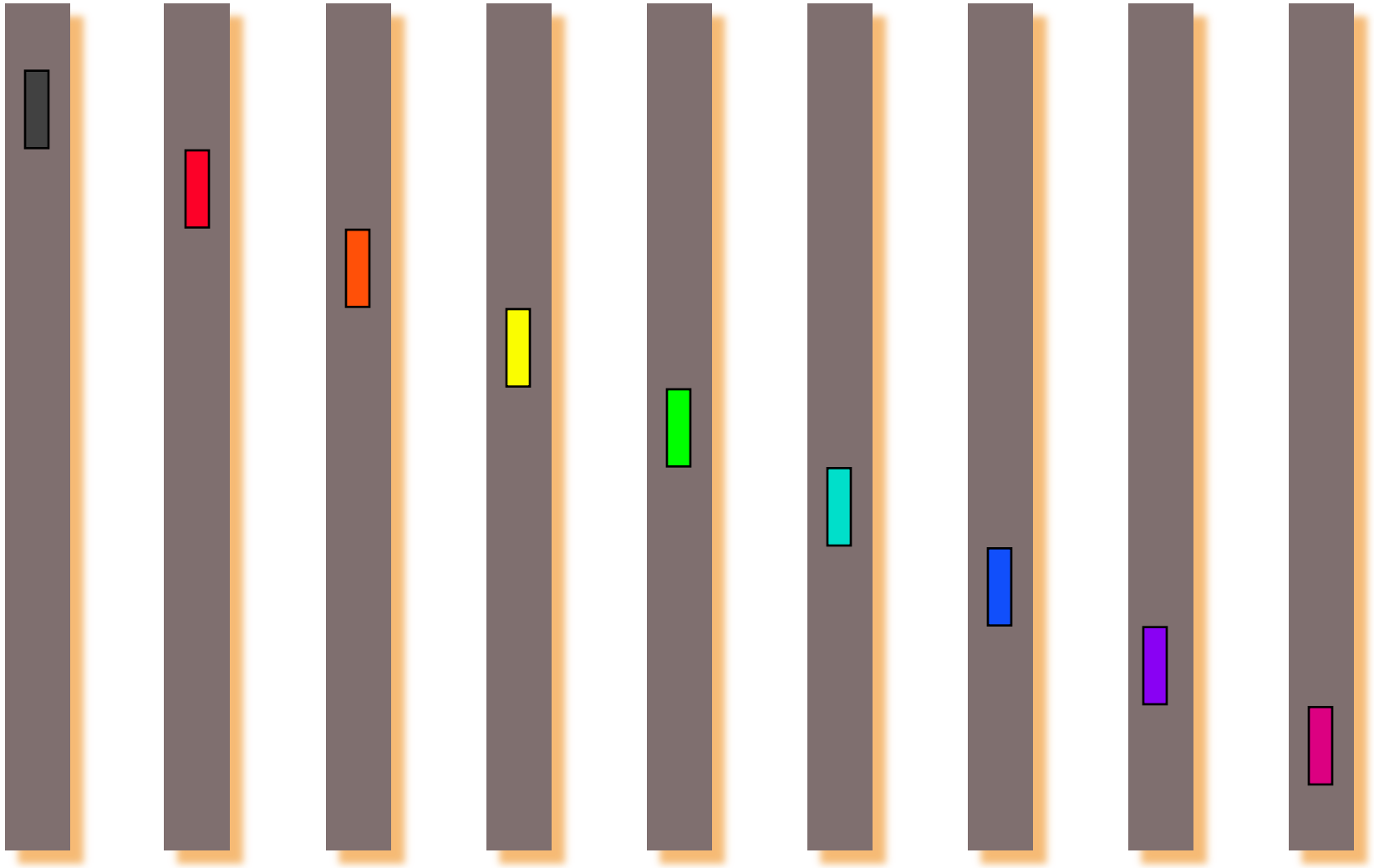
Multicore GEMM



Multicore GEMM

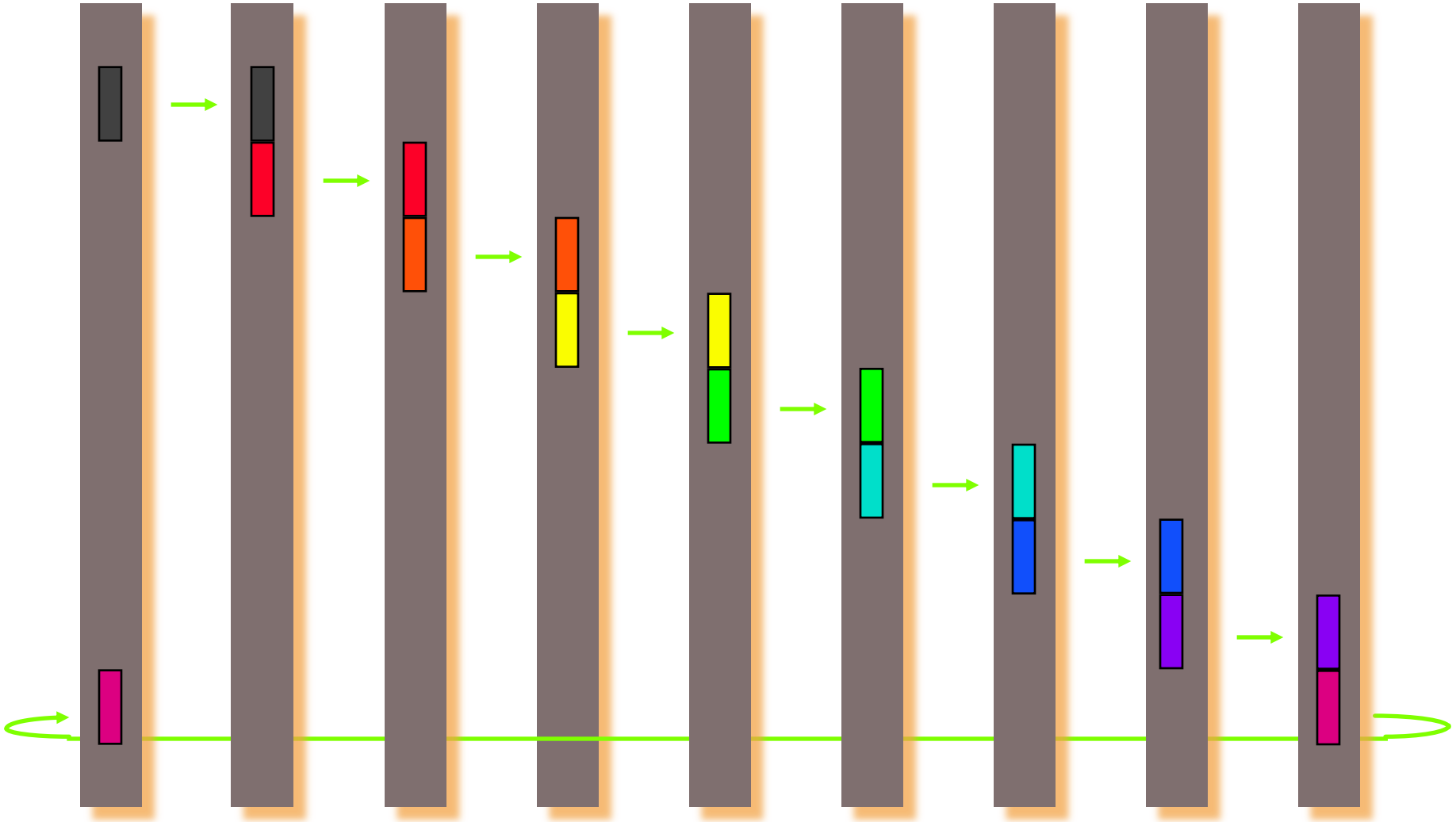


The Bucket Algorithm



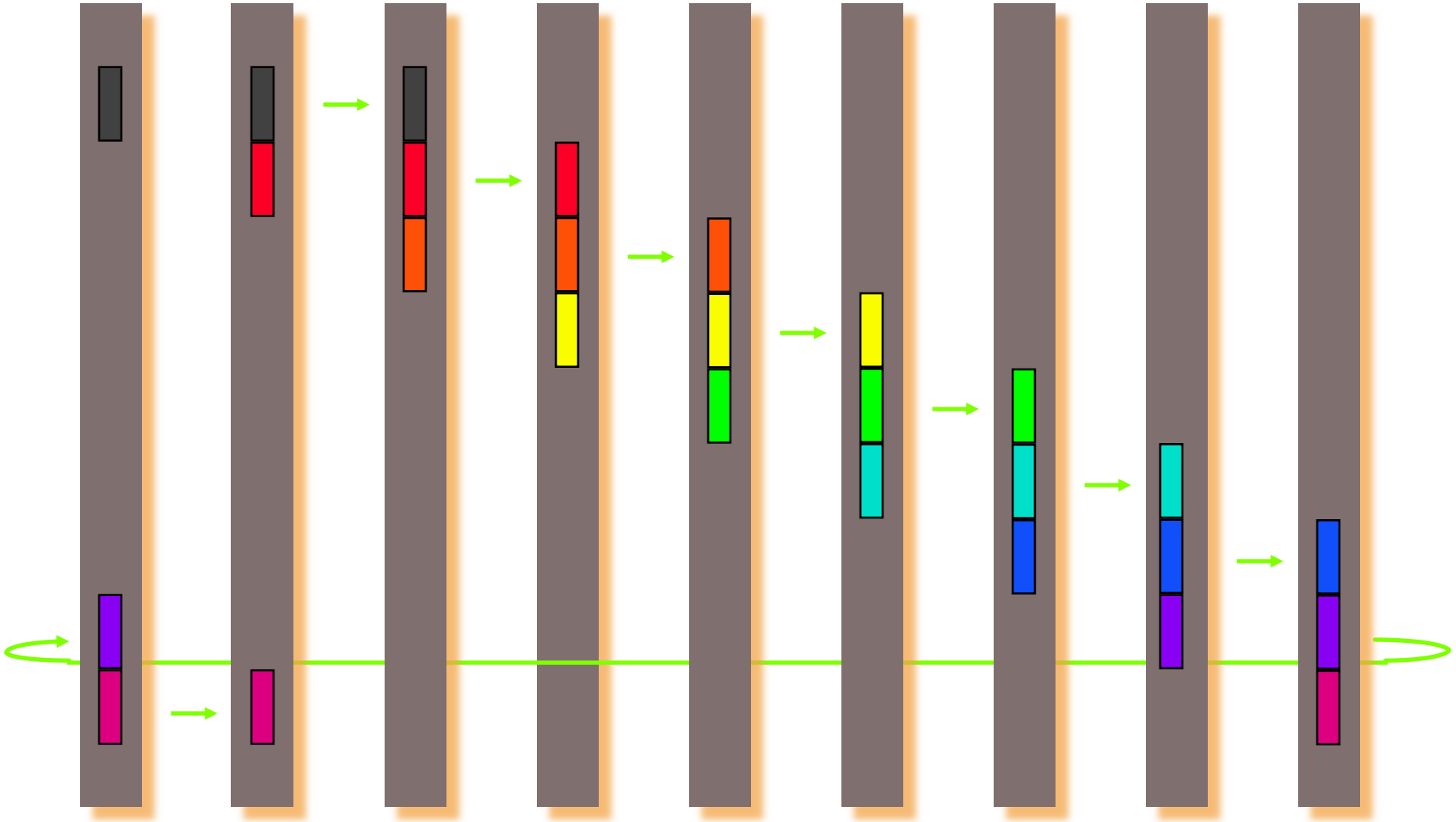
Source: Robert van de Geijn

The Bucket Algorithm



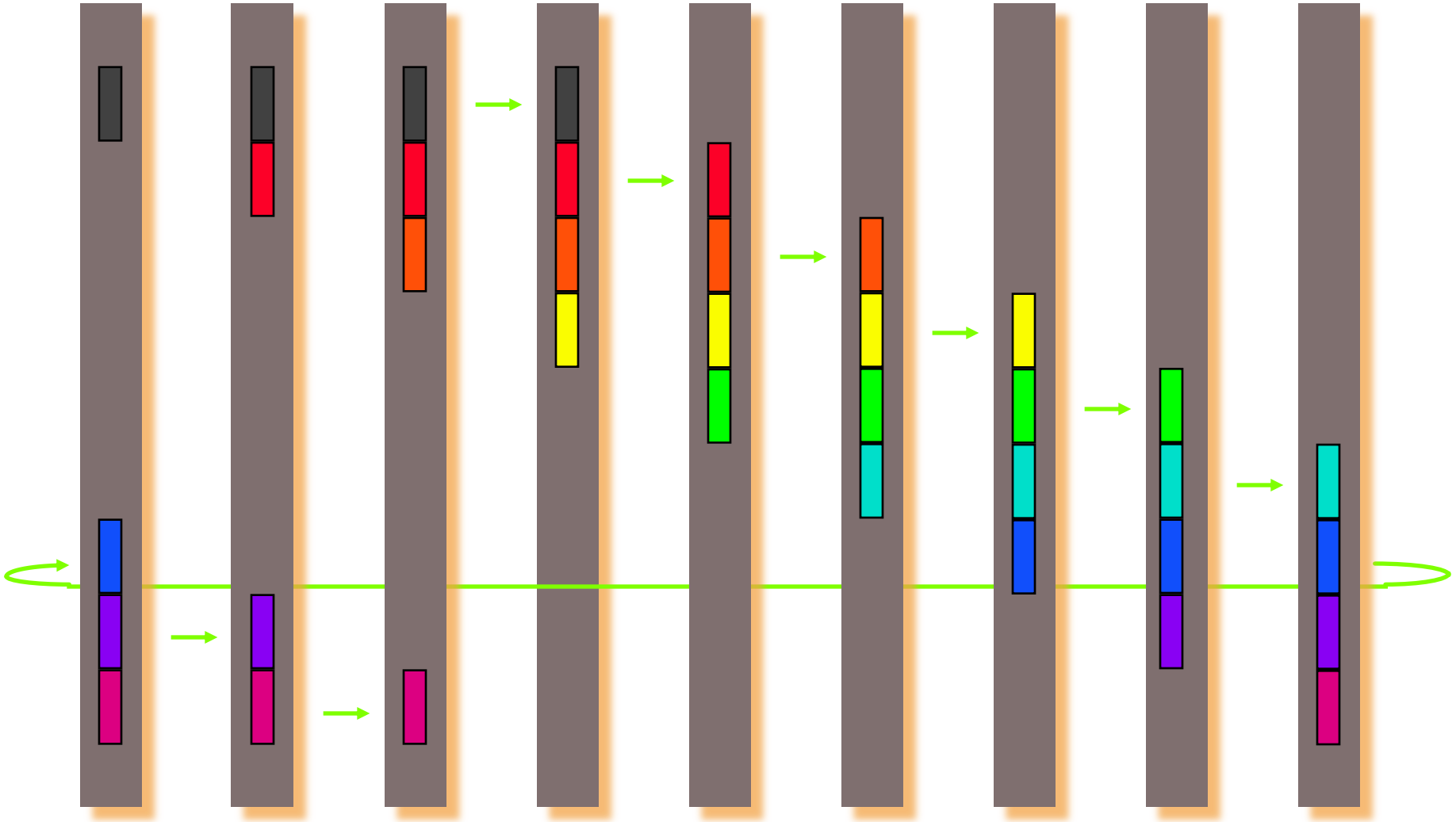
Source: Robert van de Geijn

The Bucket Algorithm



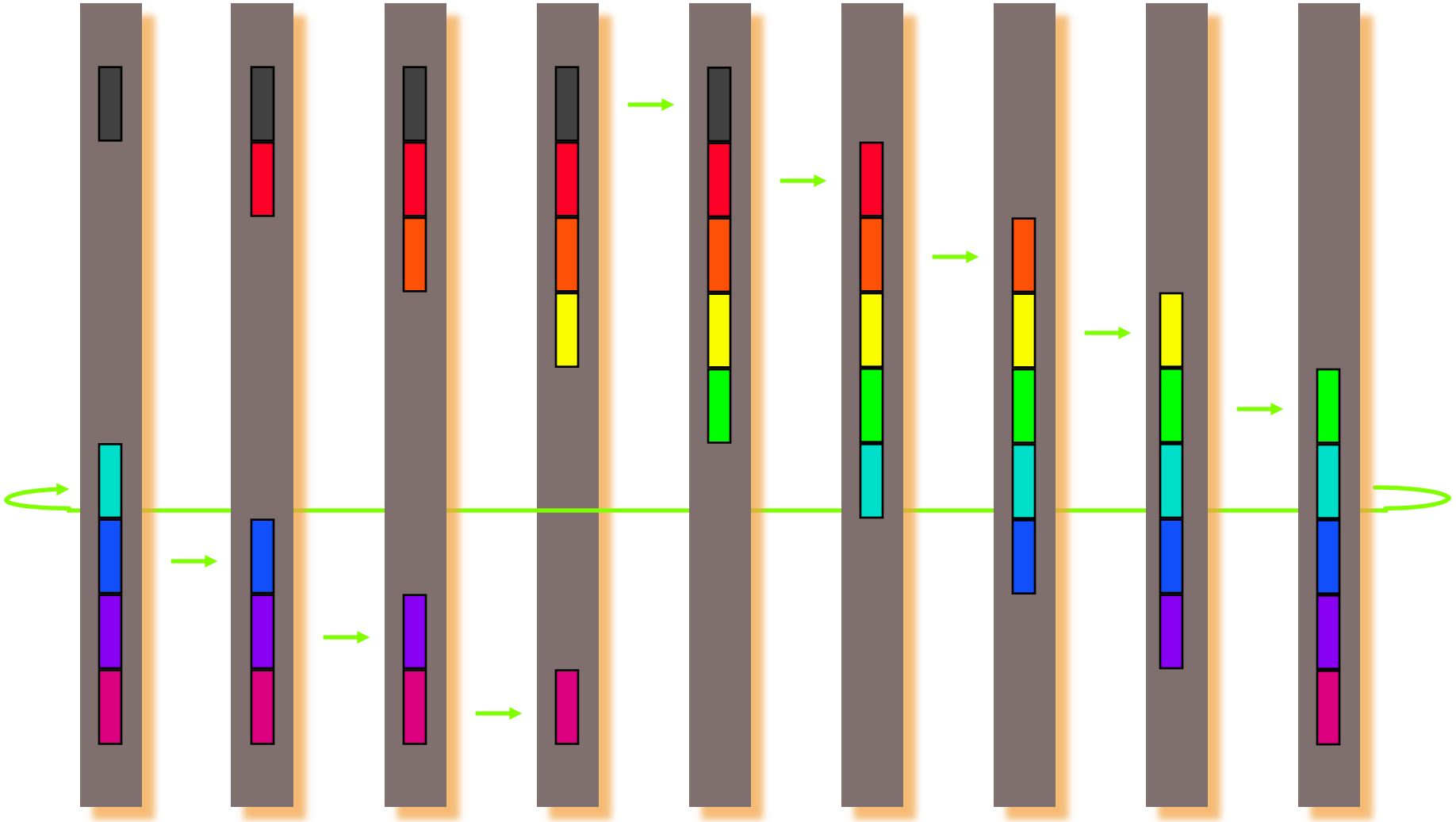
Source: Robert van de Geijn

The Bucket Algorithm



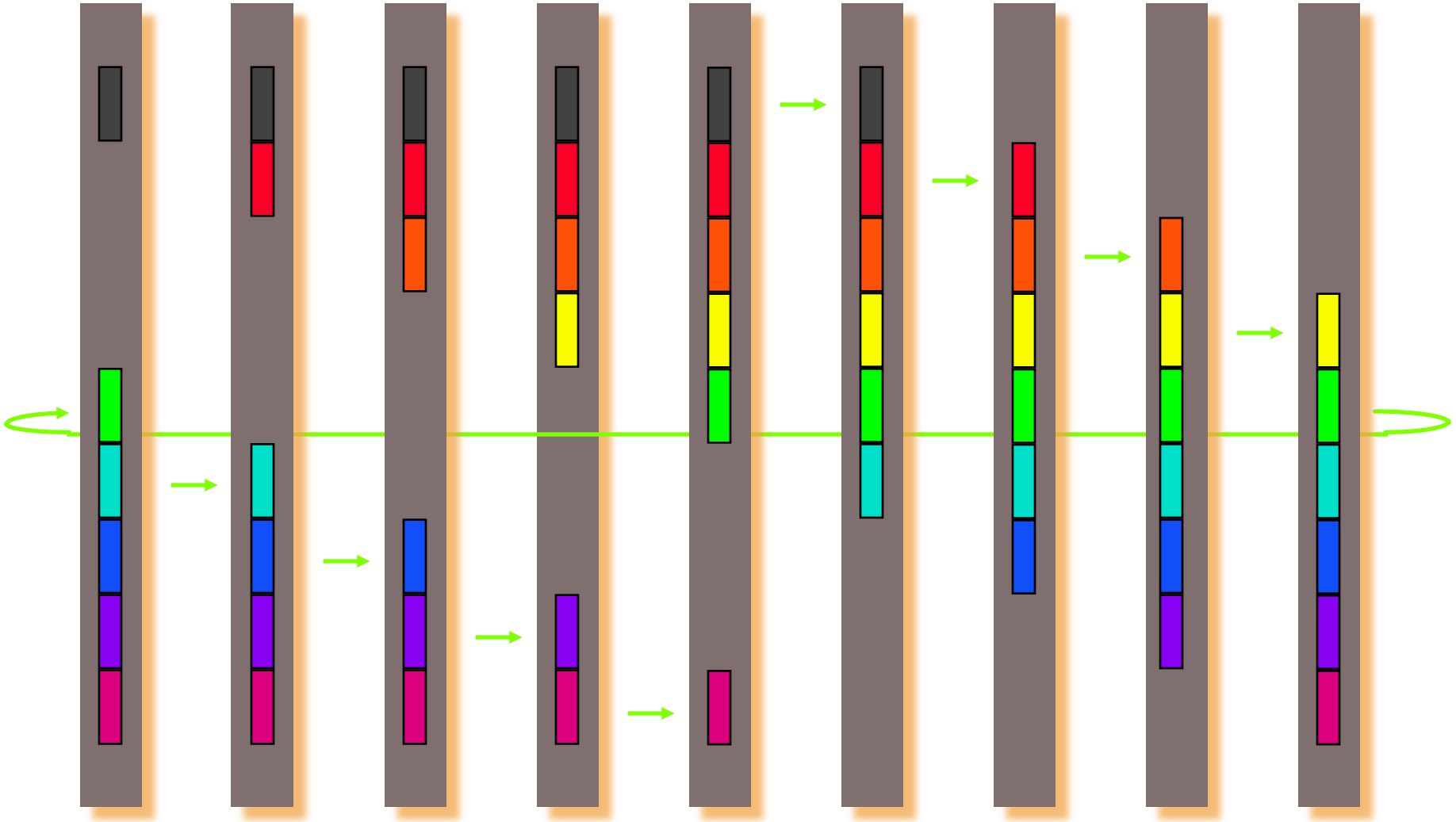
Source: Robert van de Geijn

The Bucket Algorithm



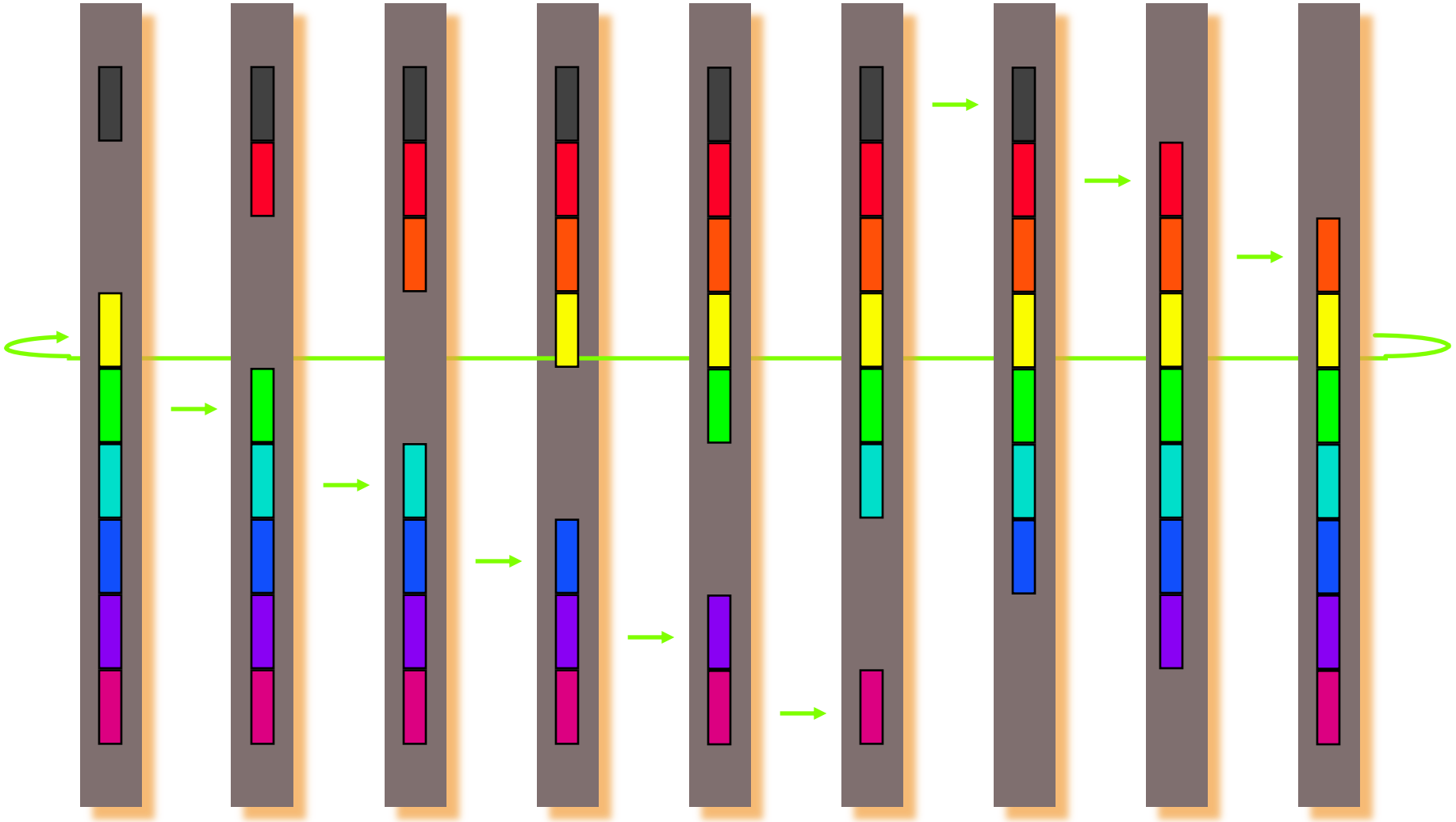
Source: Robert van de Geijn

The Bucket Algorithm



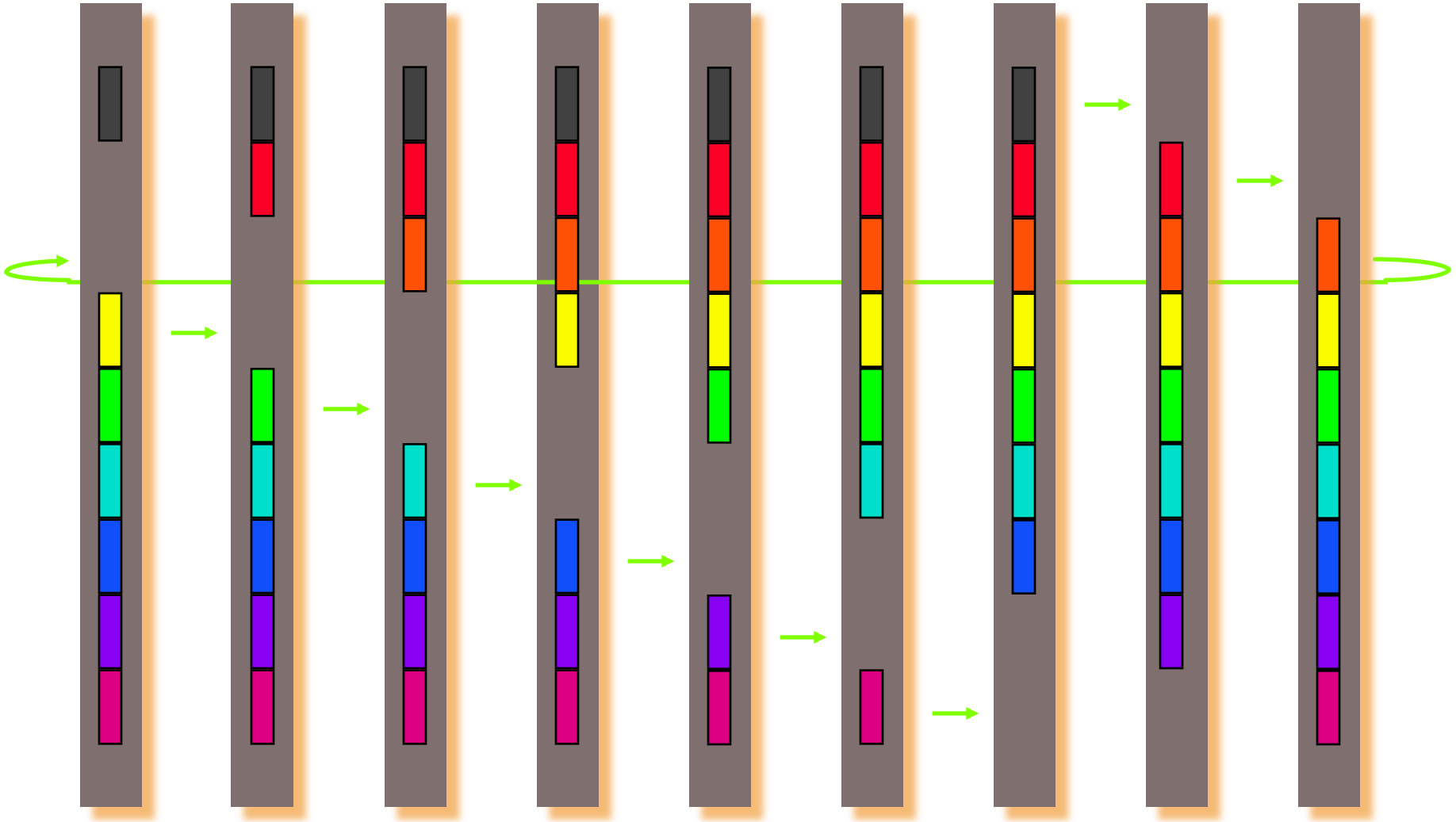
Source: Robert van de Geijn

The Bucket Algorithm



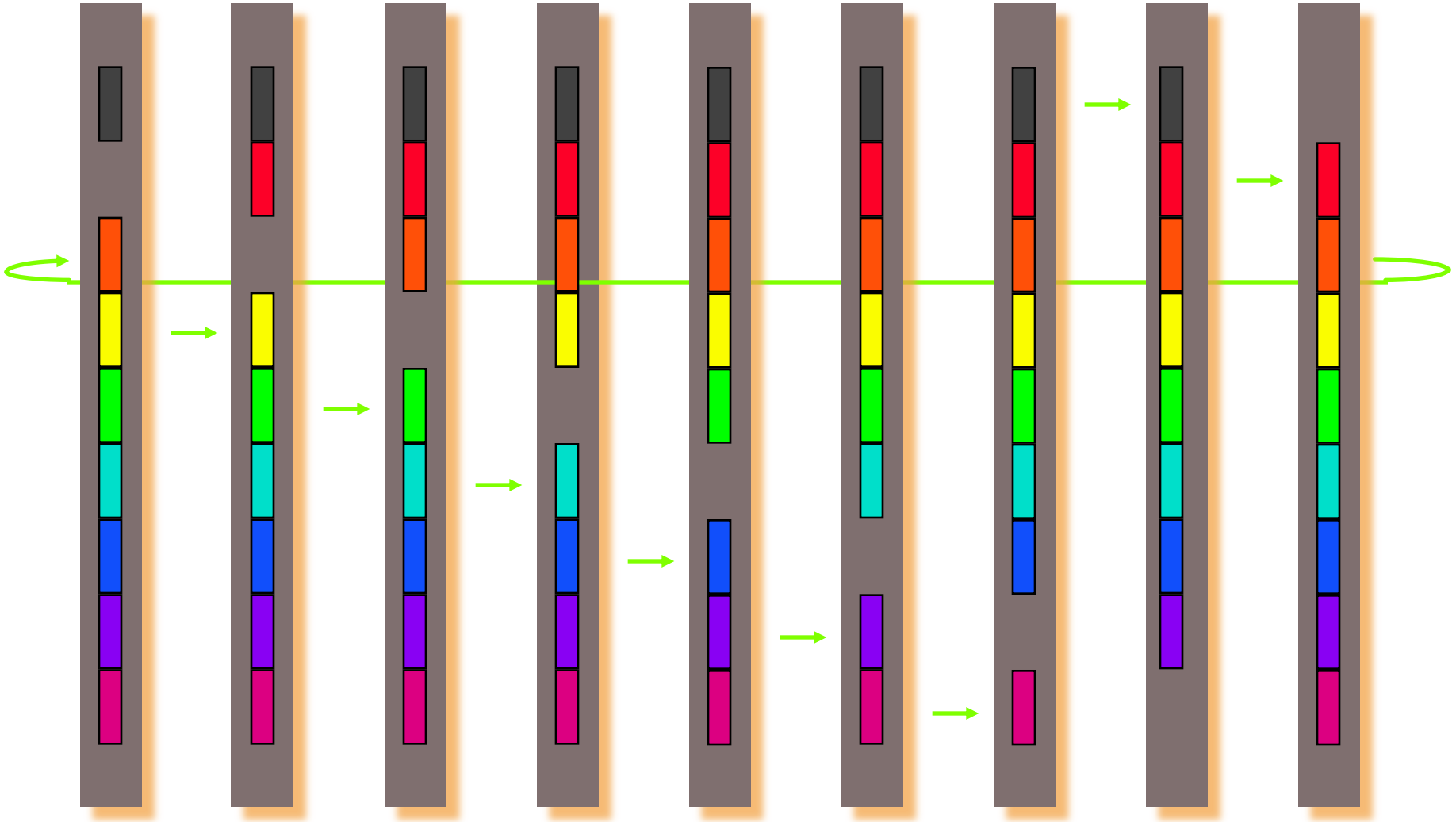
Source: Robert van de Geijn

The Bucket Algorithm



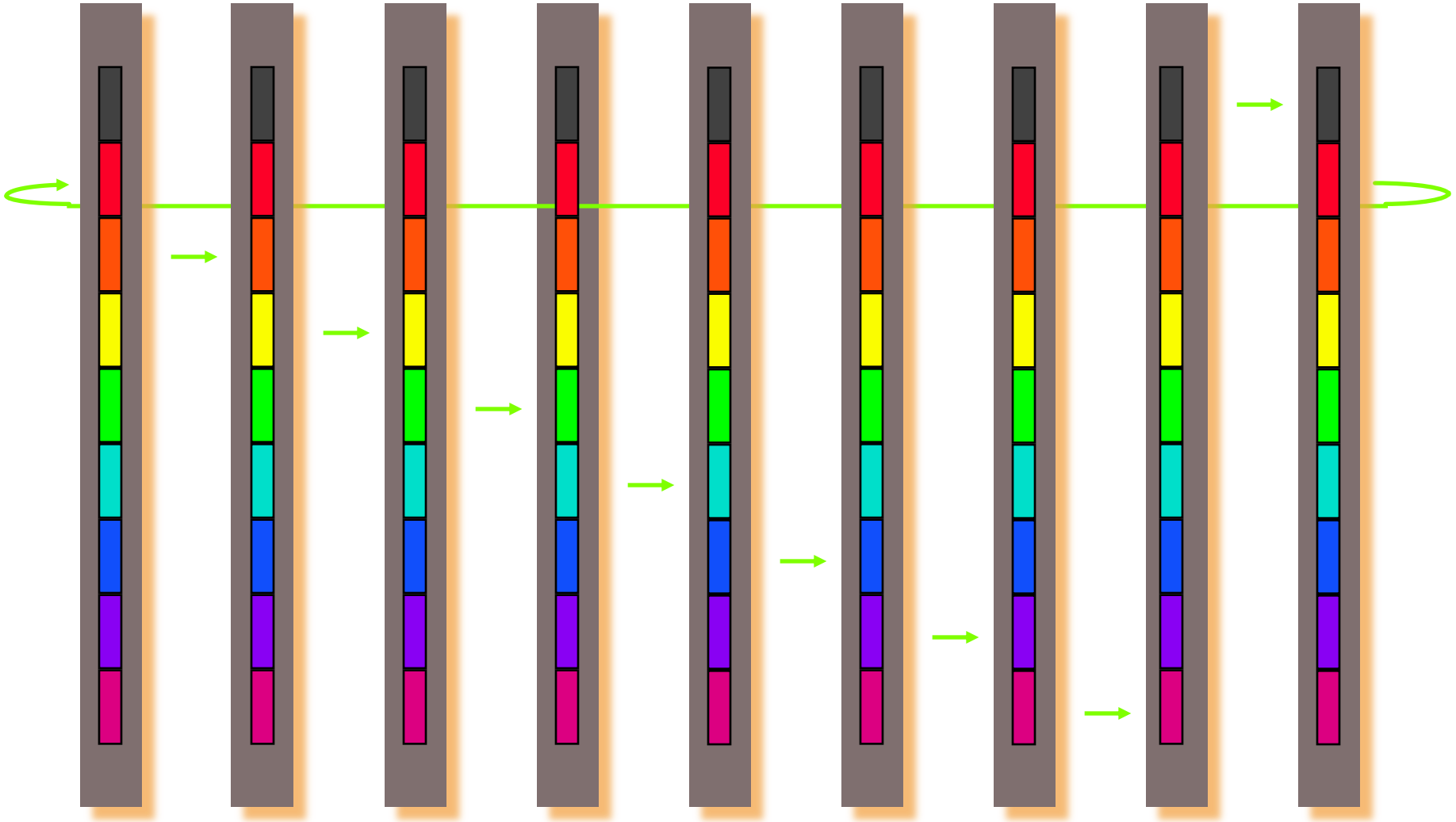
Source: Robert van de Geijn

The Bucket Algorithm



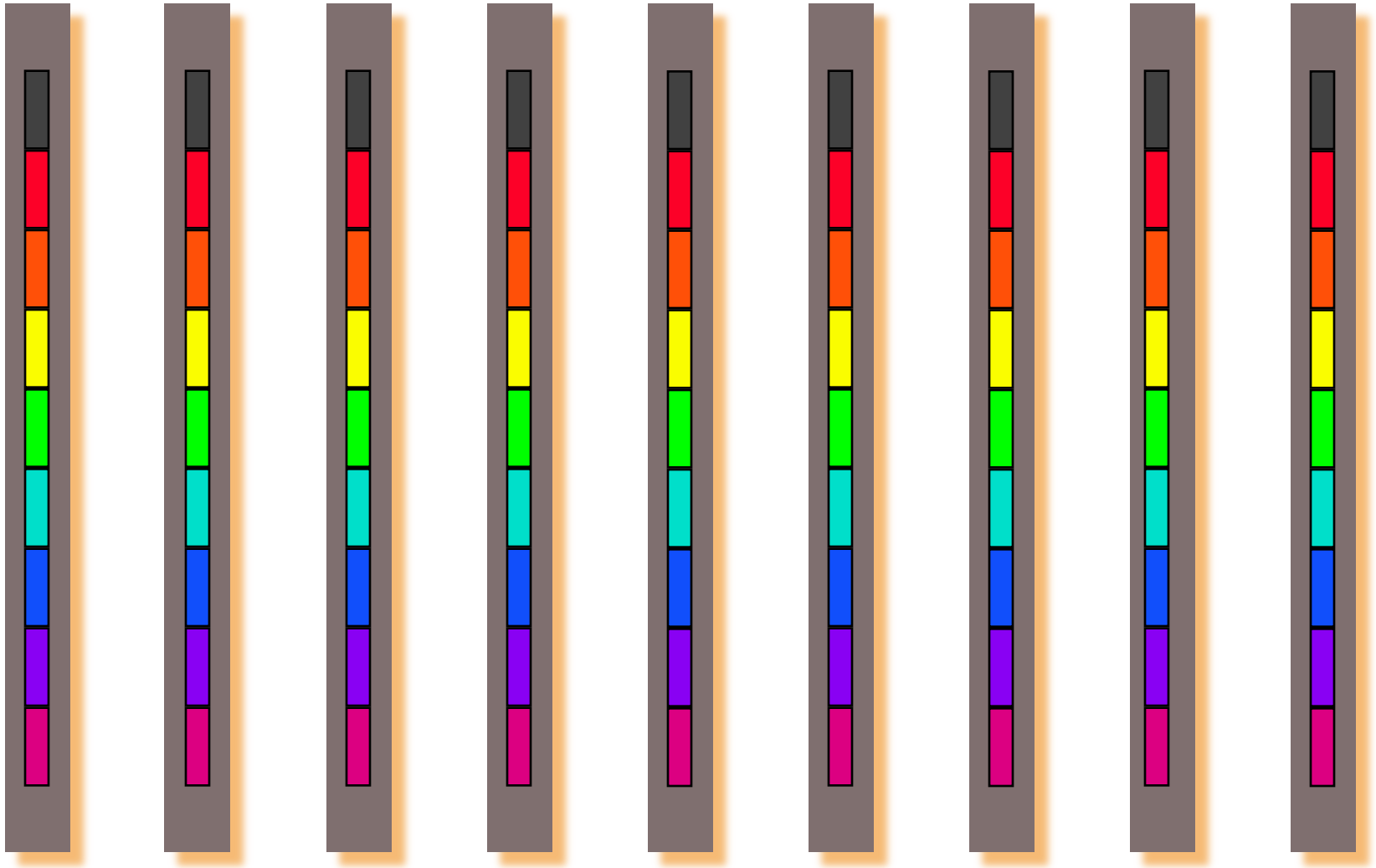
Source: Robert van de Geijn

The Bucket Algorithm



Source: Robert van de Geijn

The Bucket Algorithm



Source: Robert van de Geijn