Intro Machine Learning
And
Parallel Asynchronous Iterations

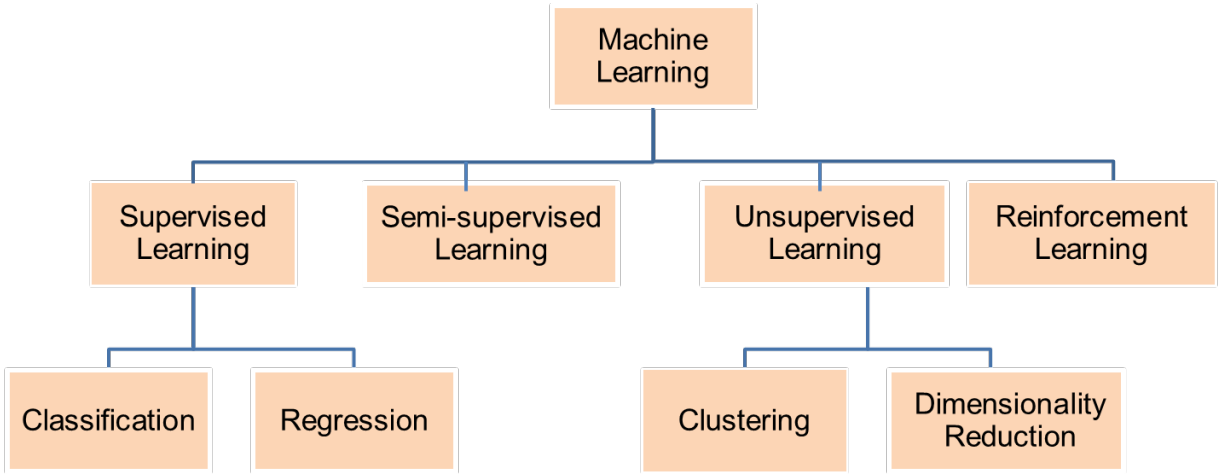
I. Learning from Data

Supervised learning: involves building a statistical model for predicting, or estimating, an *output* based on one or more *inputs*. (e.g. linear regression alg. and SVM)

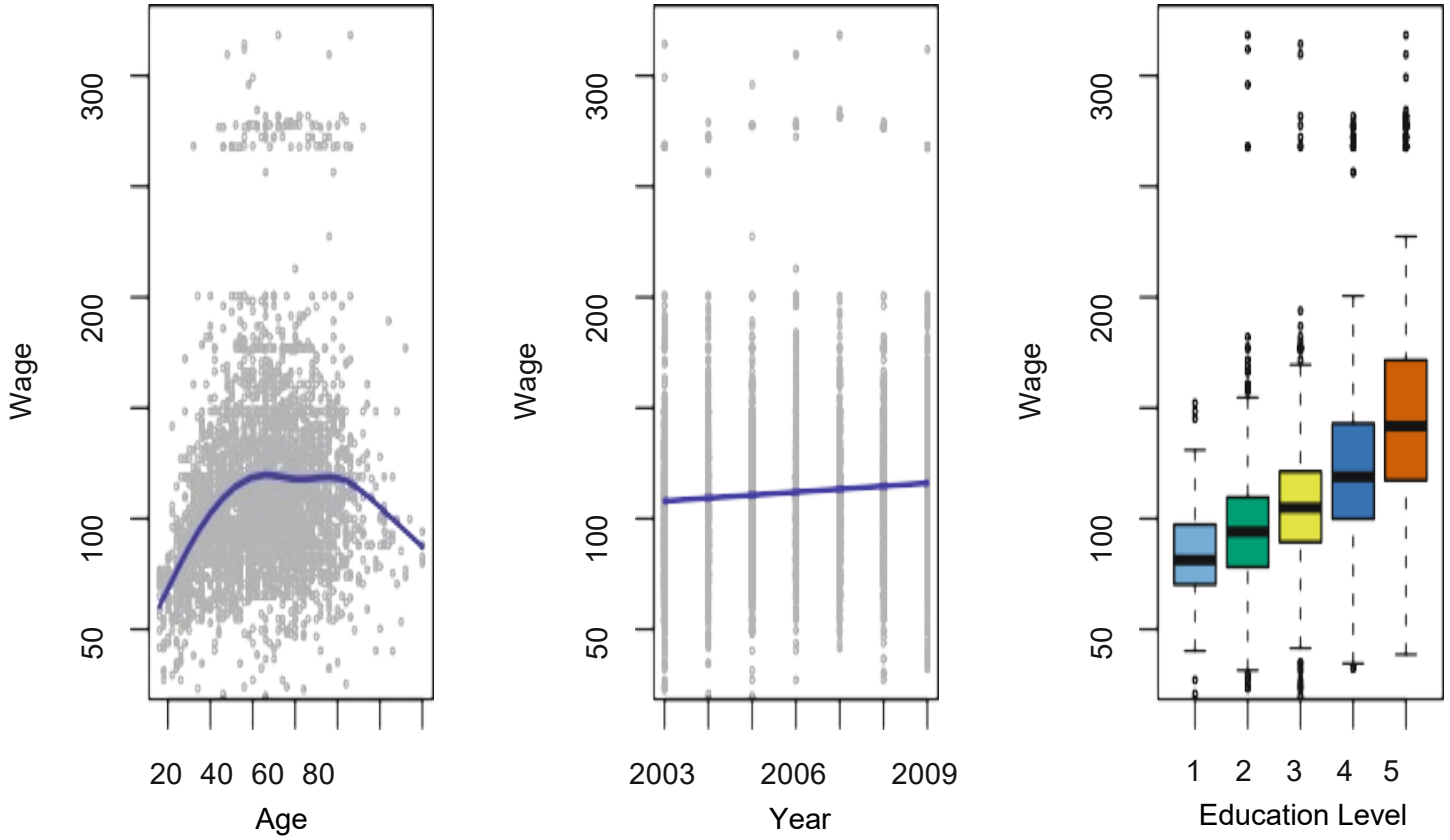
Unsupervised learning: there are inputs but no supervising output; nevertheless we can learn relationships and structure from such data. (e.g., *K*-means clustering alg.)

Regression: predicting a *continuous* or *quantitative* output value.

Classification: predict a non-numerical value—that is, a *categorical* or *qualitative* output.



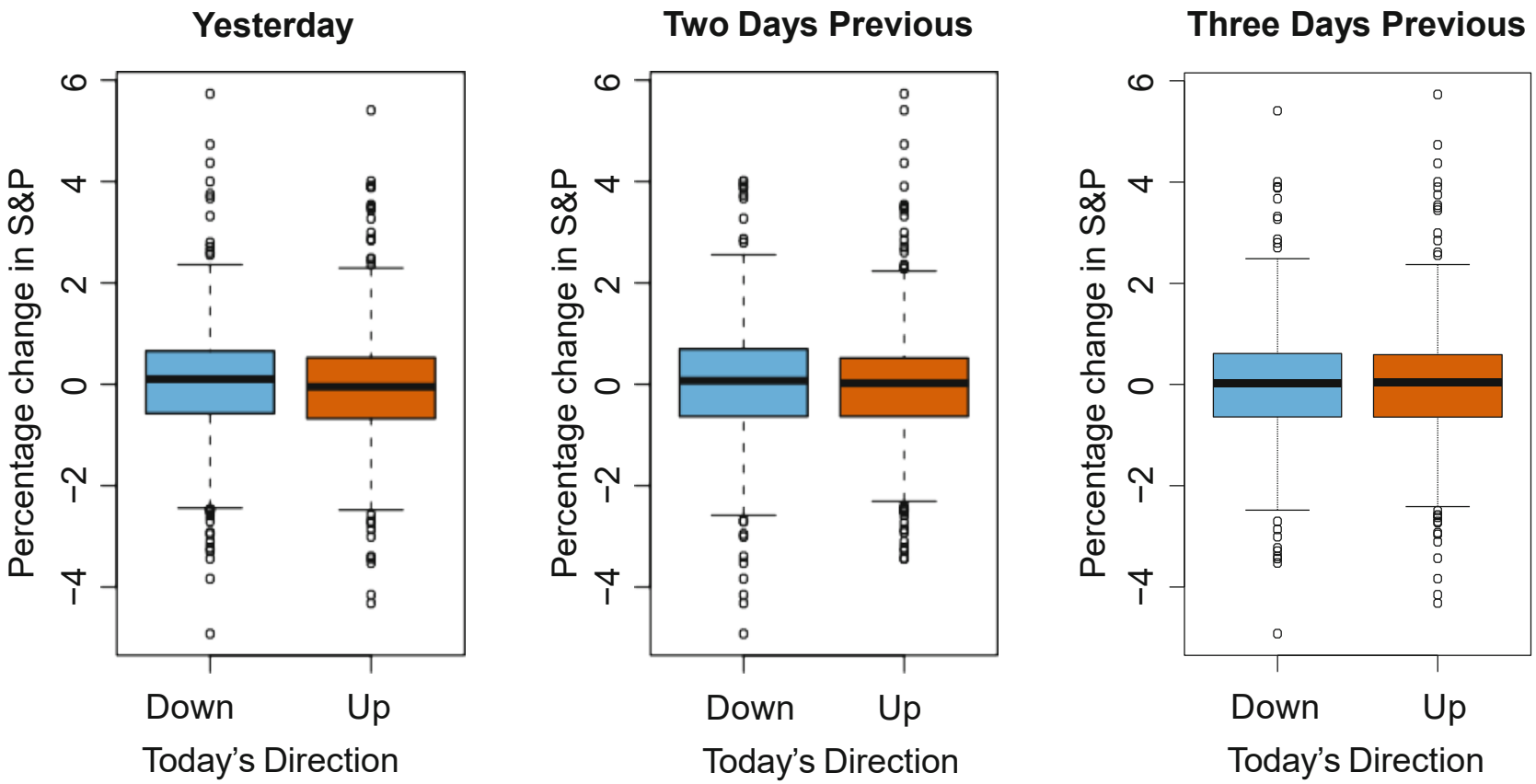
Example 1: Wage data



Wage *data, which contains income survey information for males from the central Atlantic region of the United States.* Left: wage *as a function of* age*.* Center: wage *as a function of* year*.* Right: *Boxplots displaying* wage *as a function of* education*, with* 1 *indicating the lowest level (no high school diploma) and* 5 *the highest level (an advanced graduate degree).*

Problem: predicting a person's wage by combining his age, his education, and the year

Example 2: Stock market data



Left: *Boxplots of the previous day's percentage change in the S&P index for the days for which the market increased (648 days) or decreased (602 days), obtained from the* Smarket *data 2001-2005.* Center and Right: *Same as left panel, but the percentage changes for 2 and 3 days previous are shown.*

Problem: predicting up/down of S&P based on previous *k* day's percentage changes.
→ A classification problem, here two classes {up, down}

II. A simple view on prediction

$$Y = f(X) + \varepsilon$$

where f is an unknown function to be determined through learning, $X = (X_1, X_2, \ldots, X_p)$ representing p predictors/features, ε is a Random *error term*, which is independent of X and has mean zero.

f represents the *systematic* information that X provides about Y.

Consider an estimate $\hat{f}$ and a set of predictors X, which yields the prediction $\hat{Y} = \hat{f}(X)$. Assume for a moment that both $\hat{f}$ and X are fixed. Then,

$$E(Y - \hat{Y})^2 = E[f(X) + \varepsilon - \hat{f}(X)]^2 = \underbrace{\mathrm{E}[f(X) - \hat{f}(X)]^2}_{\text{reducible}} + \underbrace{\mathrm{Var}(\varepsilon)}_{\text{irreducible}}$$

Goal: designing algorithms for estimating *f* with the aim of minimizing the reducible error.

How to estimate f ?

A. Parametric methods:

1. Assume a functional form, or shape, of f . For example, f is linear in X:

$$f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p.$$

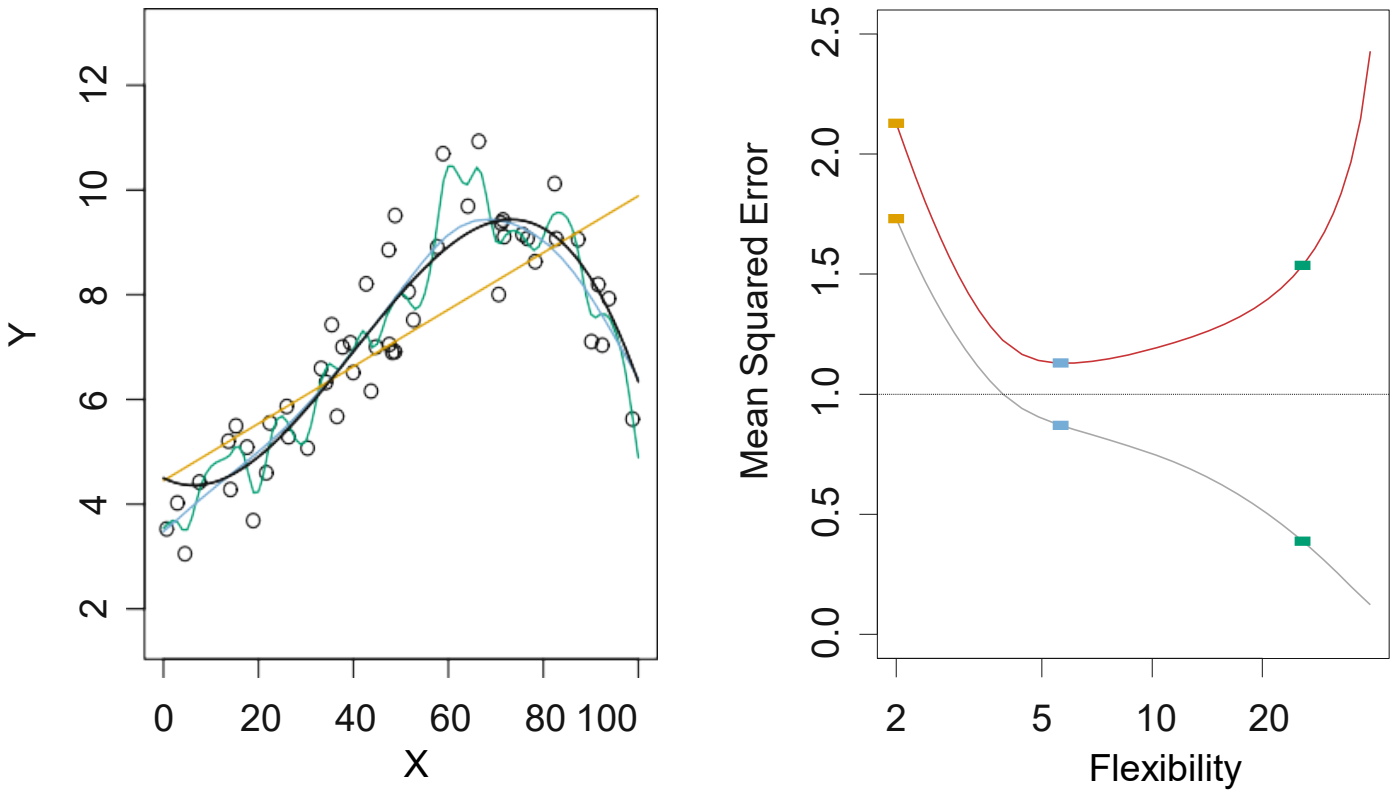
this is a *linear model*. Instead of having to estimate an entirely arbitrary p-dimensional function $f(X)$, one only needs to estimate the $p + 1$ coefficients $\beta_0, \beta_1, \ldots, \beta_p$.

2. After a model has been selected, apply a procedure that uses the training data to *fit* or *train* the model. In the above example we need to estimate the parameters $\beta_0, \beta_1, \ldots, \beta_p$.

B. Non-parametric methods:

No explicit assumptions about the functional form of f . Instead they seek an estimate of f that gets as close to the data points as possible.

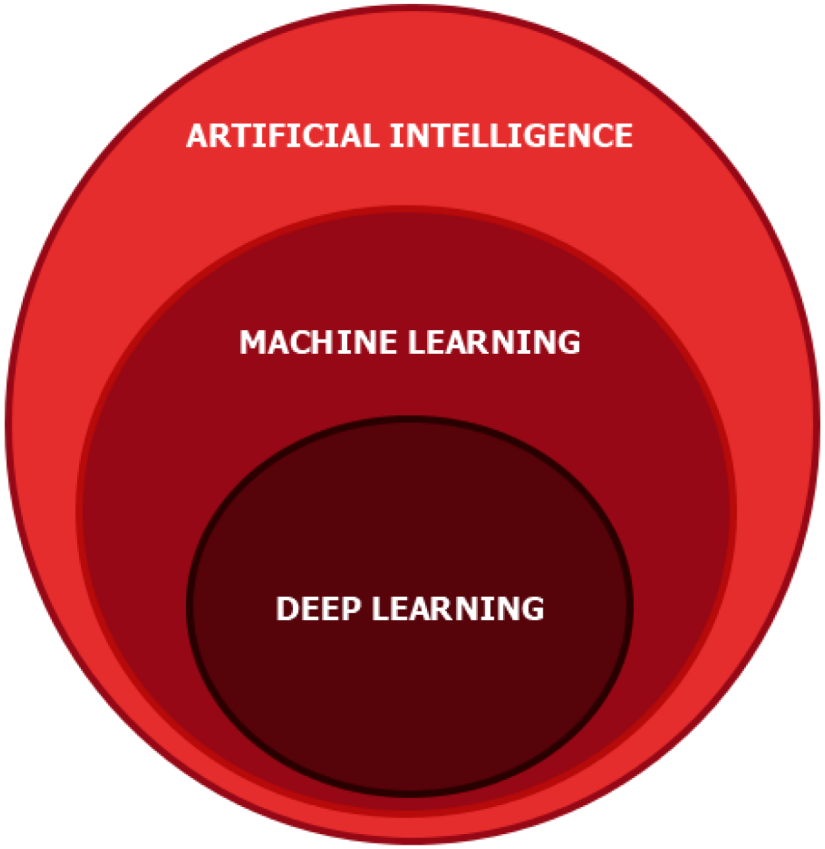
$MSE_{test} \neq MSE_{training}$



Left: *Data simulated from f, shown in black. Three estimates of f are shown: the linear regression line (orange curve), and two smoothing spline fits (blue and green curves).*

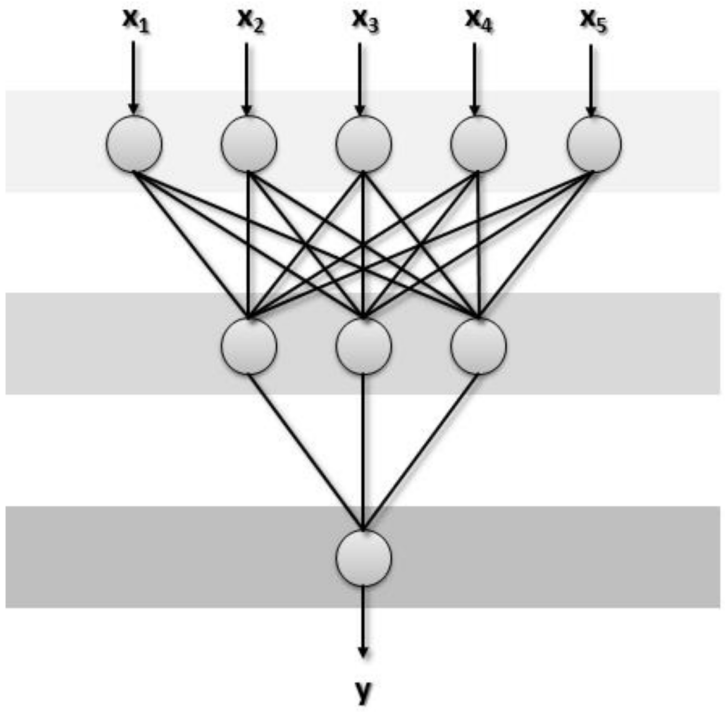
Right: *Training MSE (grey curve), test MSE (red curve), and minimum possible test MSE over all methods (dashed line). Squares represent the training and test MSEs for the three fits shown in the left-hand panel.*

Deep Learning – Definition



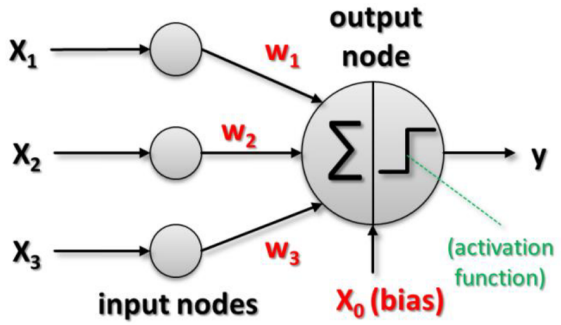
- **Artificial Intelligence**
  - The concept of machines being able to carry out tasks in a way we would consider intelligent
- **Machine Learning**
  - Computer systems that improve with experience and data
- **Deep Learning**
  - Is a subset of machine learning where the system is represented as a nested hierarchical features, where each feature is defined in relation to simpler features.

Artificial Neural Network (ANN)



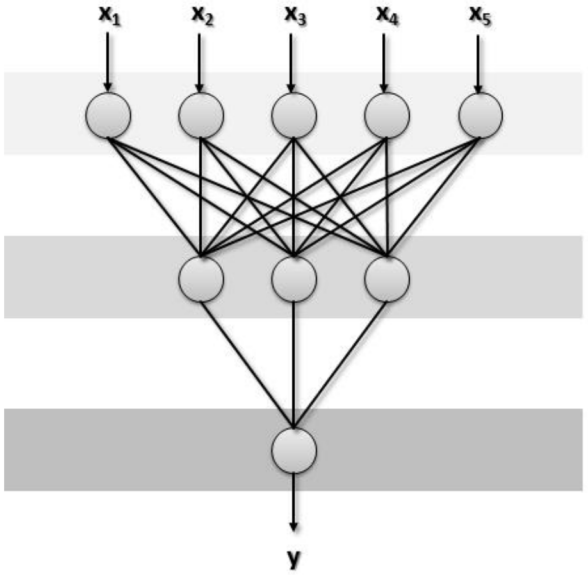
- A computational model of biological learning
- synomous with deep learning
- The nodes simulate neurons and the edges simulate synapses with weight values.

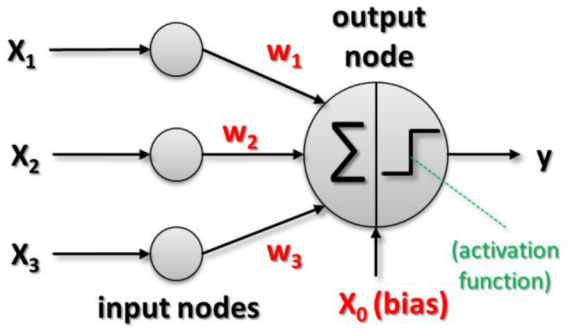
- Neurons modelled as perceptron's that “fire” their activation function when the sum of weights crosses a certain threshold.



Neural Networks - Timeline

- 1943: The first mathematical model of the human brain
- 1957: Perceptron
- 1965: The first multi-layered network
- 1987: Multi-layered Perceptron (backpropagation)
- 1995: Support Vector Machines (SVMs)
- 1998: Gradient based learning
- 2006: Deep Neural Network
- 2011: AlexNet (CNNs)
- 2014: Generative Adversarial Networks (GANs)

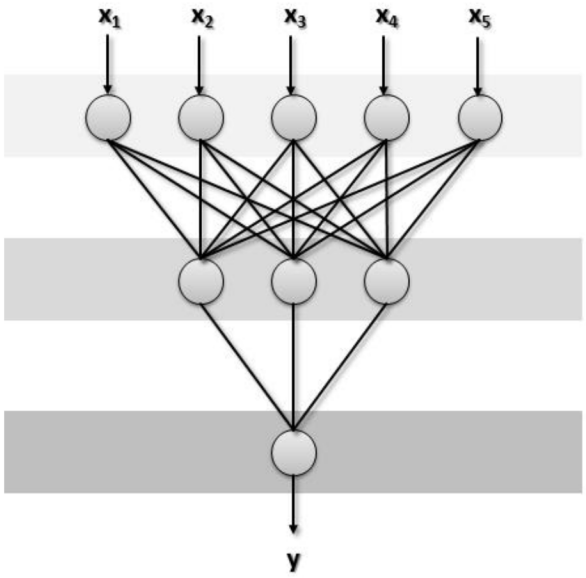


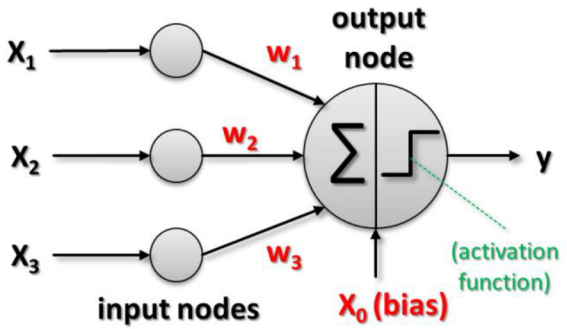


Neural Networks - Resurgence

The renaissance of neural networks via deep learning, accelerated by:

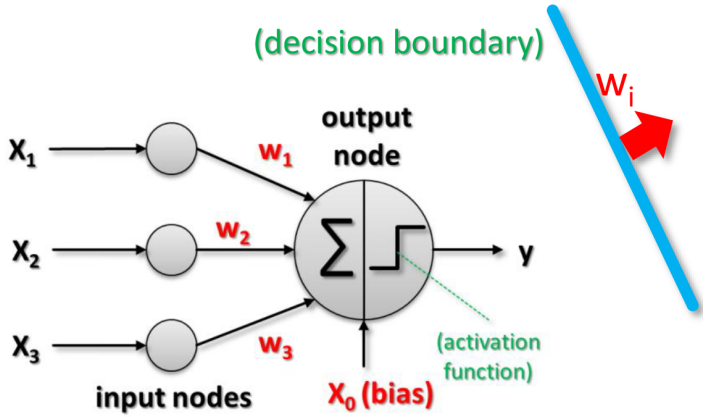
- Big Data
  - The first web page 1992
  - 163 zettabytes (1 million petabytes) by 2025
- Advances in Computation
  - Multi-core CPUs
  - Many-core GPUs
- Improved architecture and techniques
  - CNNs
  - GANs
  - Capsule Networks



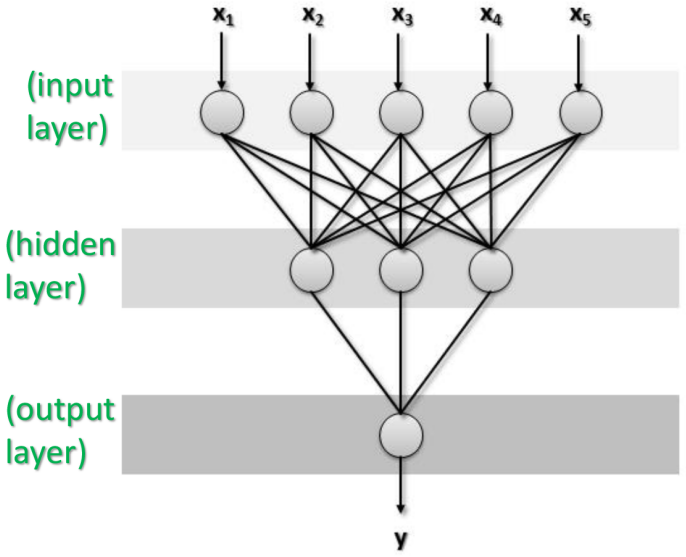


Multi Layer Perceptrons – Artificial Neural Networks

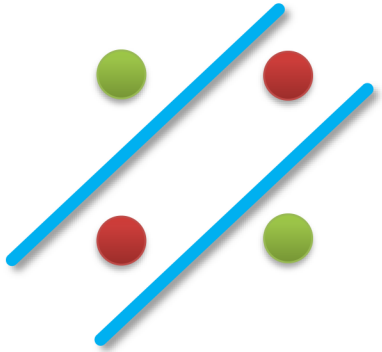
- Key Building Block
  - Perceptron learning model
  - Simplest linear learning model
  - Linearity in learned weights w_i
  - One decision boundary
- Artificial Neural Networks (ANNs)
  - Creating more complex structures
  - Enable the modelling of more complex relationships in the datasets
  - May contain several intermediary layers
  - E.g. 2-4 hidden layers with hidden nodes
  - Use of activation function that can produce output values that are nonlinear in their input parameters

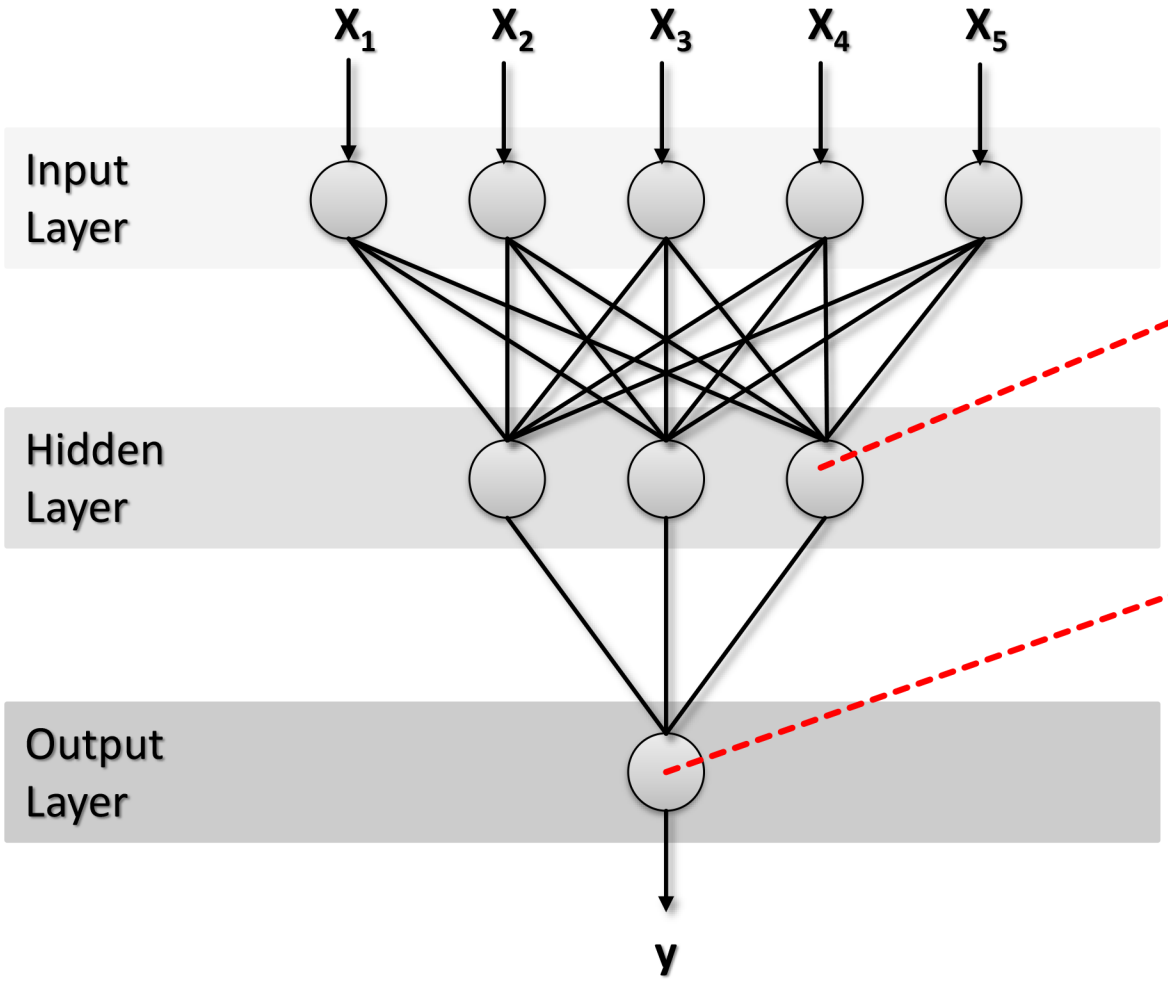


(perceptron model)



Artificial Neural Networks (ANN) – Layers & Nodes





- Think each hidden node as a 'simple perceptron' that each creates one hyperplane
- Think the output node simply combines the results of all the perceptrons to yield the 'decision boundary' above
- Feed-forward neural network: nodes in one layer are connected only to the nodes in the next layer ('a constraint of network construction')

ANN - Learning Algorithm & Optimization

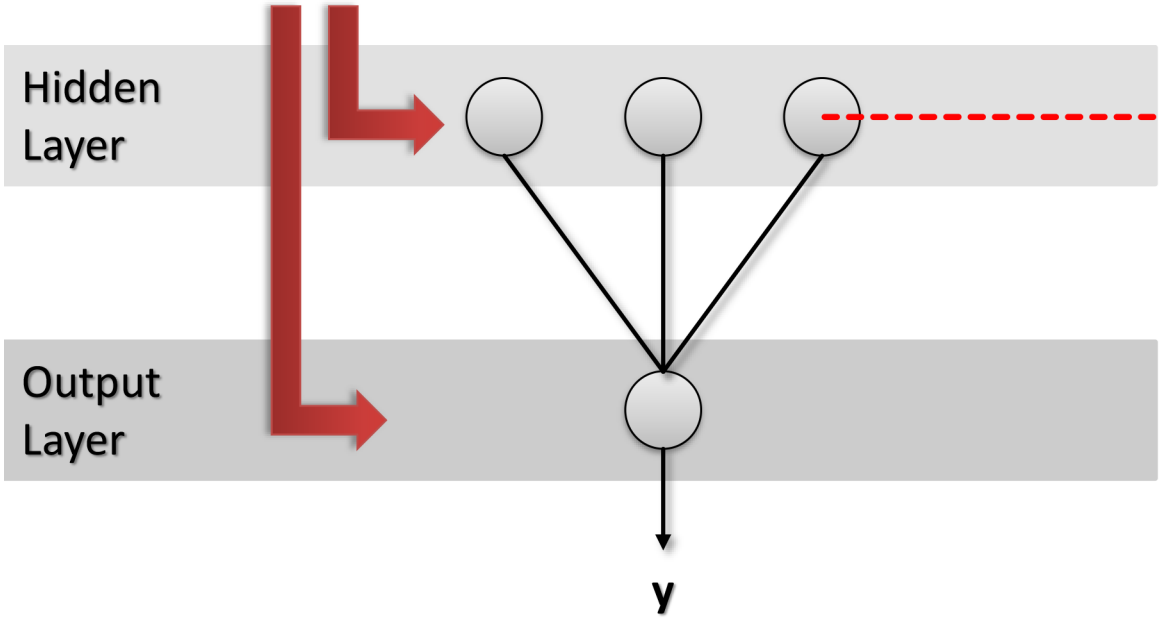
- Determine a set of weights **w** that 'minimize the total sum of squared errors':

$y = \text{sign}(w \cdot x)$

Linear perceptron

Sum of squared errors depend on w, because predicted class y is a 'function of the weights' assigned to the hidden and output nodes

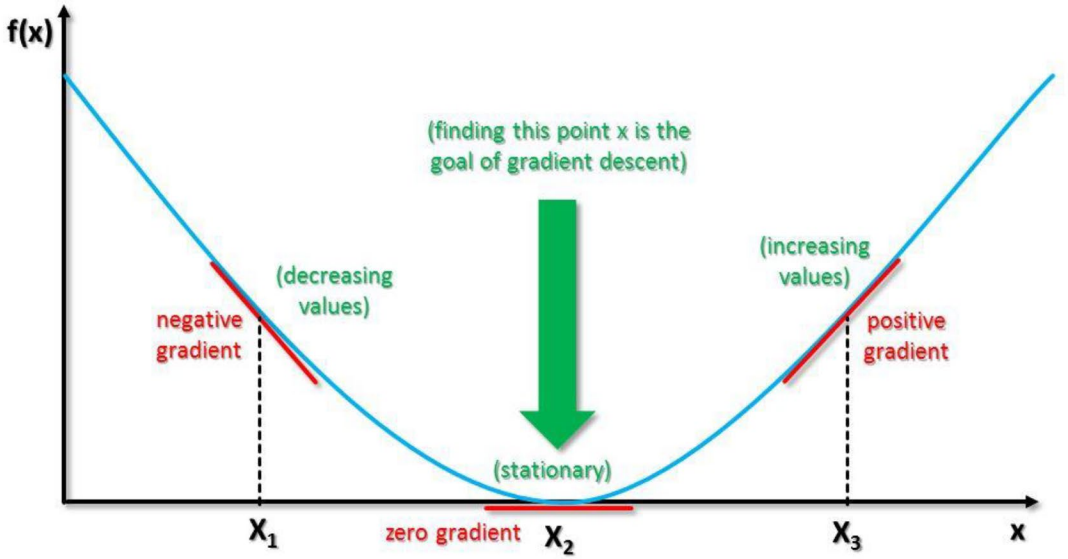
$$E(w) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \acute{y}_i)^2$$



- Error term, associated with each hidden node

- Error function is quadratic in its parameters and a global minimum can be easily found
- Other objective / loss functions possible, e.g. categorical cross-entropy

Gradient Descent Method (1)



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

(the derivative of f with respect to a)

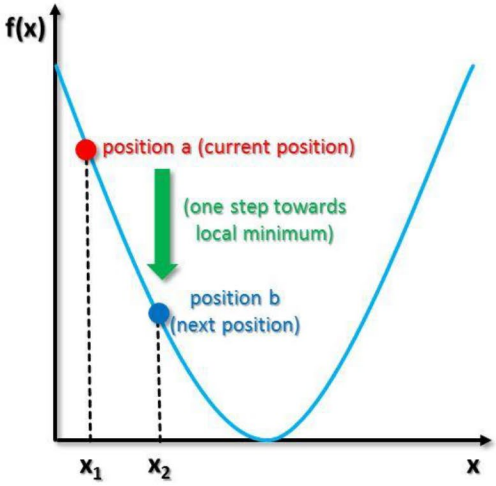
$$b = a - \gamma \nabla f(a)$$

(new position after the step)

(old position before the step)

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

(gradient term is steepest ascent)



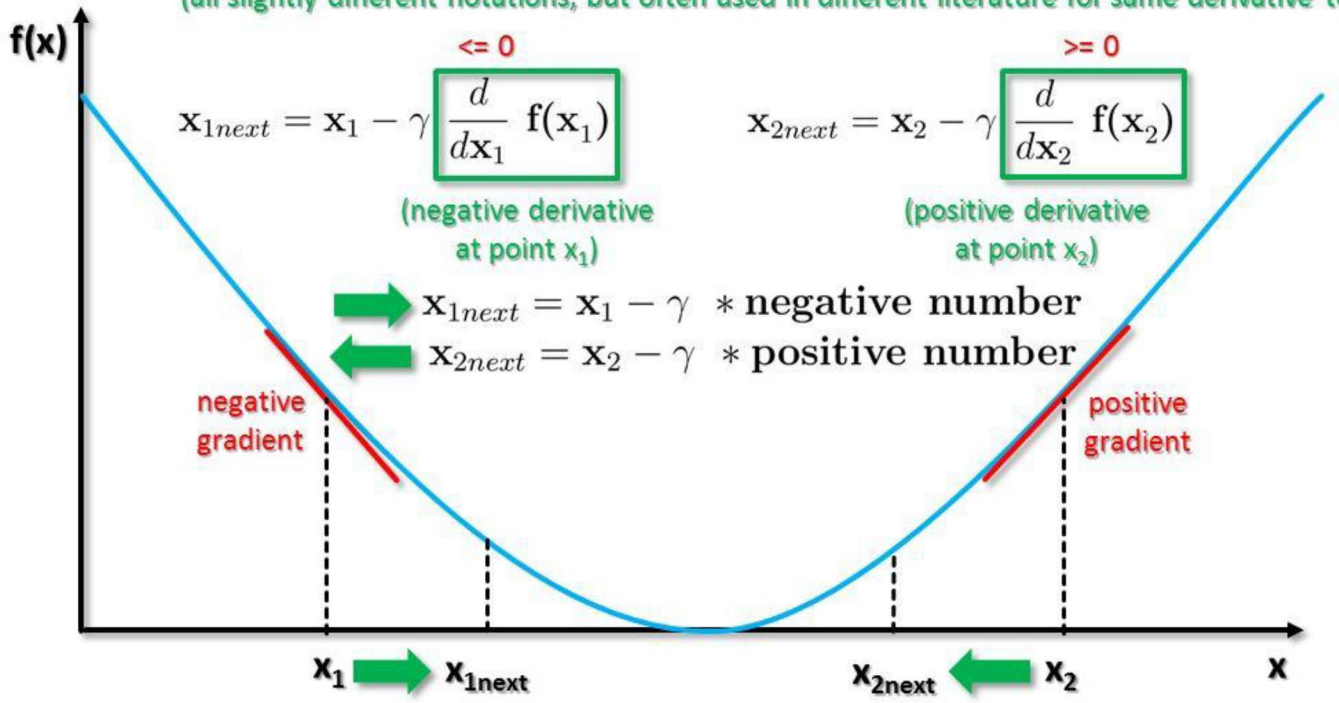
From www.big-data.tips/gradient-decent

Gradient Descent Method (2)

- Gradient Descent (GD) uses all the training samples available for a step within a iteration
- Stochastic Gradient Descent (SGD) converges faster: only one training sample used per iteration

$$\mathbf{b} = \mathbf{a} - \gamma \nabla \mathrm{f}(\mathbf{a}) \quad \mathbf{b} = \mathbf{a} - \gamma \frac{\partial}{\partial \mathbf{a}} \mathrm{f}(\mathbf{a}) \quad \mathbf{b} = \mathbf{a} - \gamma \frac{d}{d\mathbf{a}} \mathrm{f}(\mathbf{a})$$

(all slightly different notations, but often used in different literature for same derivative term)



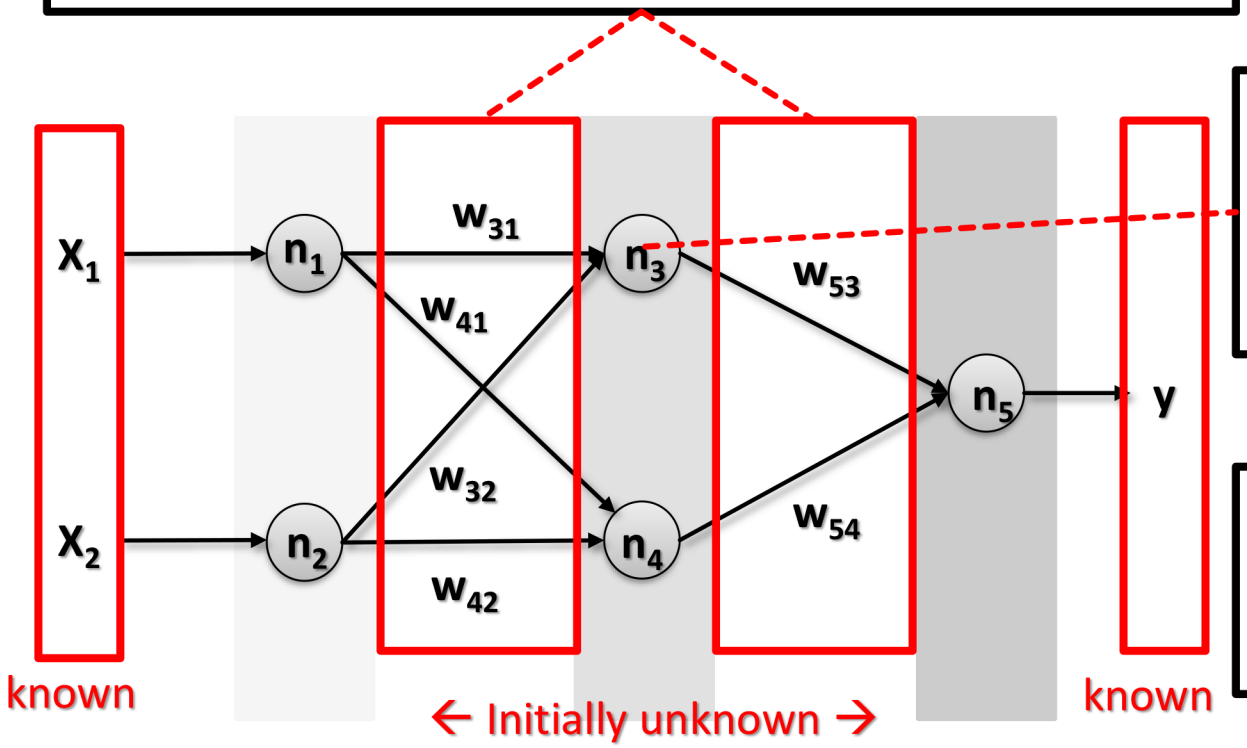
[10] Big Data Tips, Gradient Descent

ANN – Backpropagation Algorithm (BP) Basics

- One of the most widely used algorithms for supervised learning
  - Applicable in multi-layered feed-forward neural networks

- **'Gradient descent method' can be used to learn the weights of the output and hidden nodes of a artificial neural network**

[11] Introduction to Data Mining



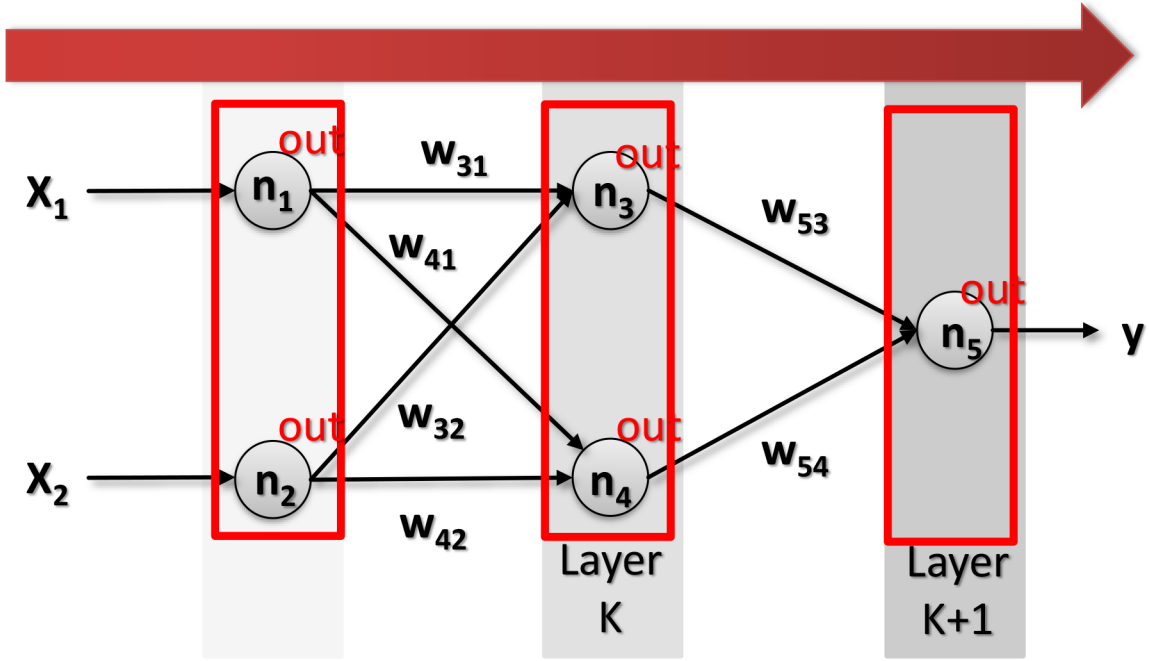
- **Hidden nodes problem: computing error term hard: $\partial E / \partial wj$**
- **Their Output values are unknown to us (here)…**

- **The backpropagation algorithm solves exactly this problem with two phases per iteration(!)**

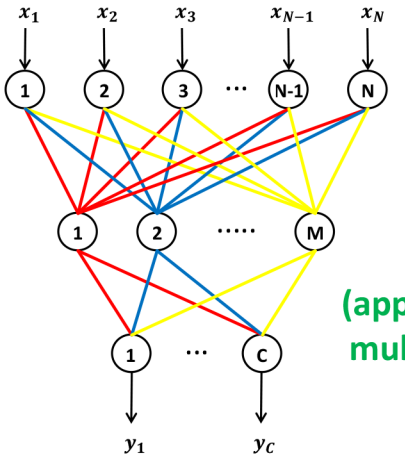
H. Lee et al, 2009 [11]

ANN – Backpropagation Algorithm Forward Phase

1. ‘Forward phase (does not change weights, re-use old weights)’:
   - Weights obtained from the previous iteration are used to compute the output value of each neuron in the network (‘initialize weights randomly’)
   - Computation progresses in the ‘forward direction’, i.e. outputs ‘out’ of the neurons at level k are computed prior to level k+1



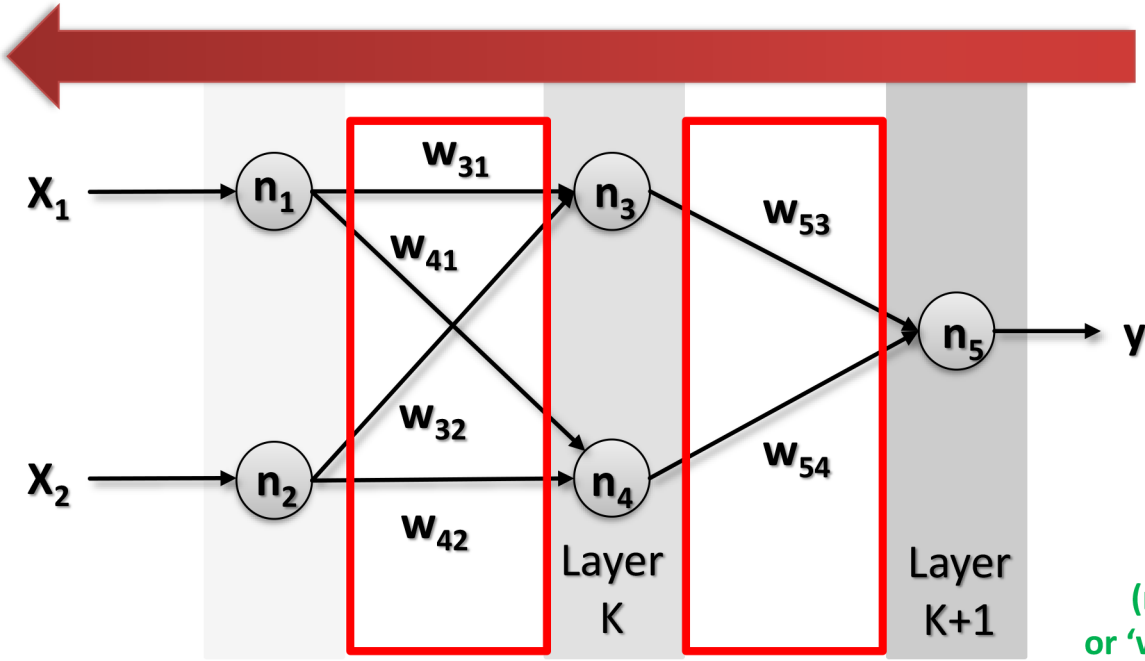
- **Use corresponding ‘activation function’ but with ‘old weights’**



(applicable for multiclass to)

ANN – Backpropagation Algorithm Backward Phase

2. ‘Backward phase (‘learning’ → change the weights in the ANN)’:
   - Weight update formula is applied in the ‘reverse direction’
   - Weights at level K + 1 are updated before the weights at level k
   - Idea: use the errors for neurons at layer k + 1 to estimate errors for neurons at layer k



$$w_j <- wj - \lambda \frac{\partial E(w)}{\partial w_j}$$

weight update formula
of the ‘gradient descent method’

Now that can compute
the error one-by-one

$$E_{in}(\mathbf{w}) + \frac{\lambda}{N}\mathbf{w}^T\mathbf{w}$$

(regularization method ‘weight decay‘
or ‘weight drop‘ is used in neural networks‘)

Training:
calculates the ‘weights’ through minimization.
(forward: ‘weights’-matrix multiplications;
backward: chain-rule layer-by-layer adjusting the weights w)

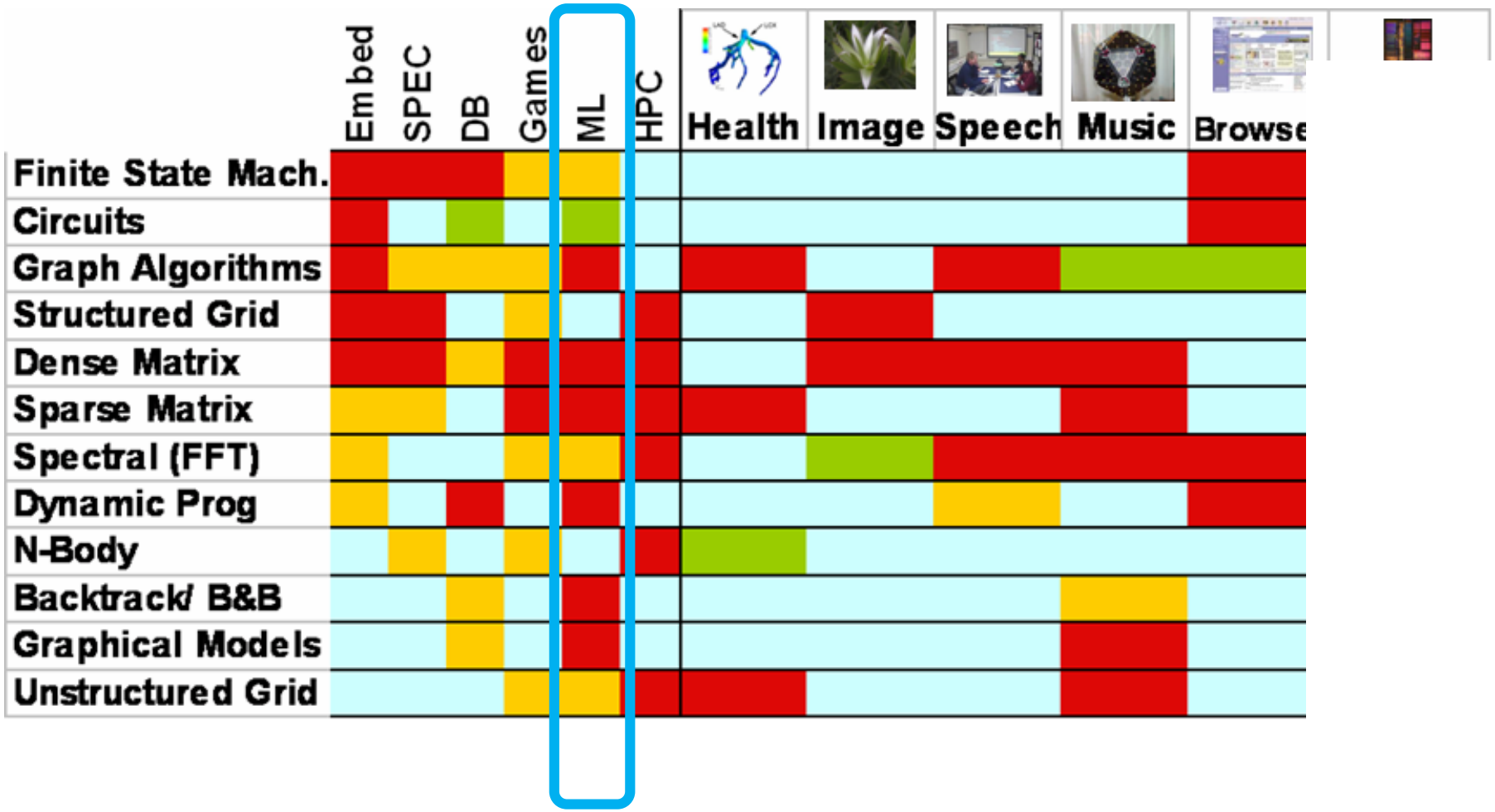
Gradient decent or stochastic gradient descent:
Sequential iteration.

Parallel optimization (training):
parallel asynchronous iteration algorithms

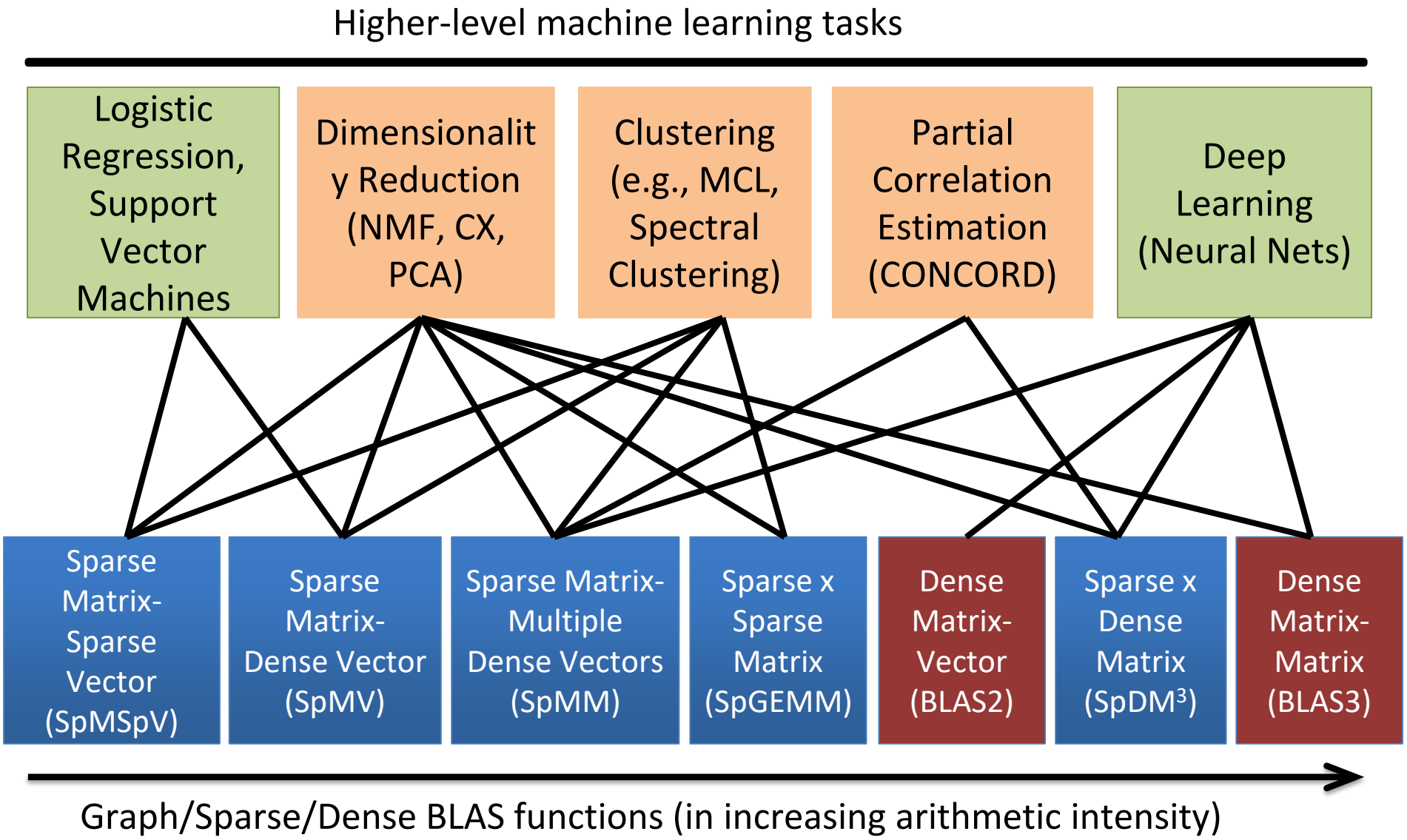
Article: (see Brightspace)
Asynchronous Parallel Stochastic Gradient Descent - A Numeric Core for Scalable Distributed Machine Learning Algorithms, by Janis Keuper and Franz-Josef Pfreundt,

Motifs

The Motifs (formerly “Dwarfs”) from
“The Berkeley View” (Asanovic et al.)
Motifs form key computational patterns



Machine Learning relies a lot on Linear Algebra



Parallelism in Machine Learning

Implicit Parallelization: Keep the overall algorithm structure (the sequence of operations) intact and parallelize the individual operations.

Example: parallelizing the BLAS operations in previous figure

+ Often achieves exactly the same accuracy (e.g., model parallelism in DNN training)

- Scalability can be limited if the critical path of the algorithm is long

Explicit Parallelization: Modify the algorithm to extract more parallelism, such as working on individual pieces whose results can later be combined

Examples: CA-SVM and data parallelism in DNNs

+ Significantly better scalability can be achieved

- No longer the same algorithmic properties (e.g. HogWild!).

Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, F. Niu et al, NIPS 2011

Slides source: Aidyn Buluc

Parallelization Opportunities

Different from the data parallelism with 'owner-compute' rule (distribute output)

1. **Data parallelism**

   Distribute the input (sets of images, text, audio, etc.)

   a) **Batch parallelism**

   - Distribute a group
     processor
   - *When people mer*
     this is what they n

   b) **Domain parallelism**

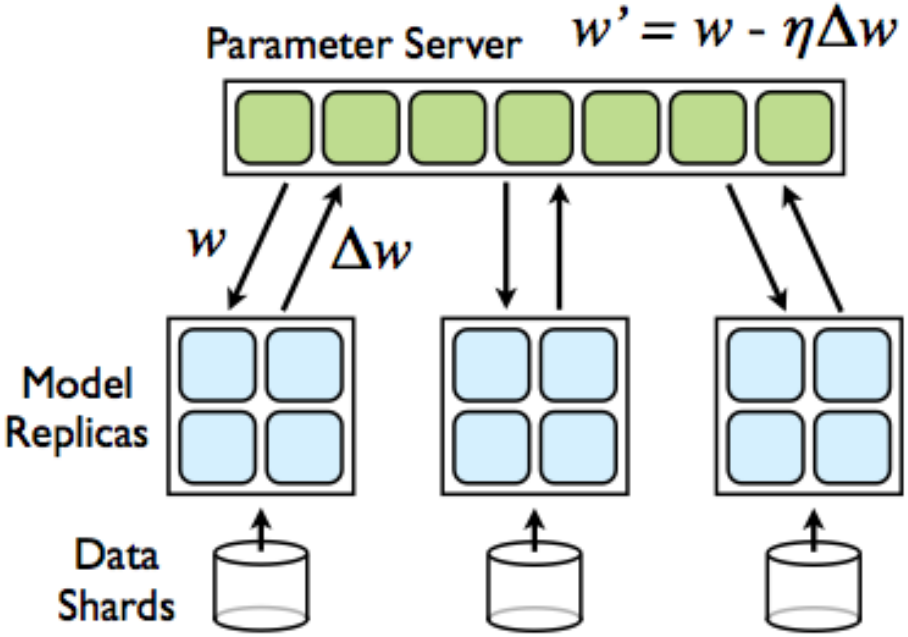
   - Data points/features of individual sample are subdivided and distribute parts to processors.

2. **Model parallelism:**

   Distribute the neural network (i.e. its weights)

Epoch: the entire training set is used once;
Mini-batch: a subset of training samples, weights are updated once the entire mini-batch is used.
SDG: update the weights for every training sample (mini-batch size=1)

Batch Parallelism #1



Parameter server is some sort of master process

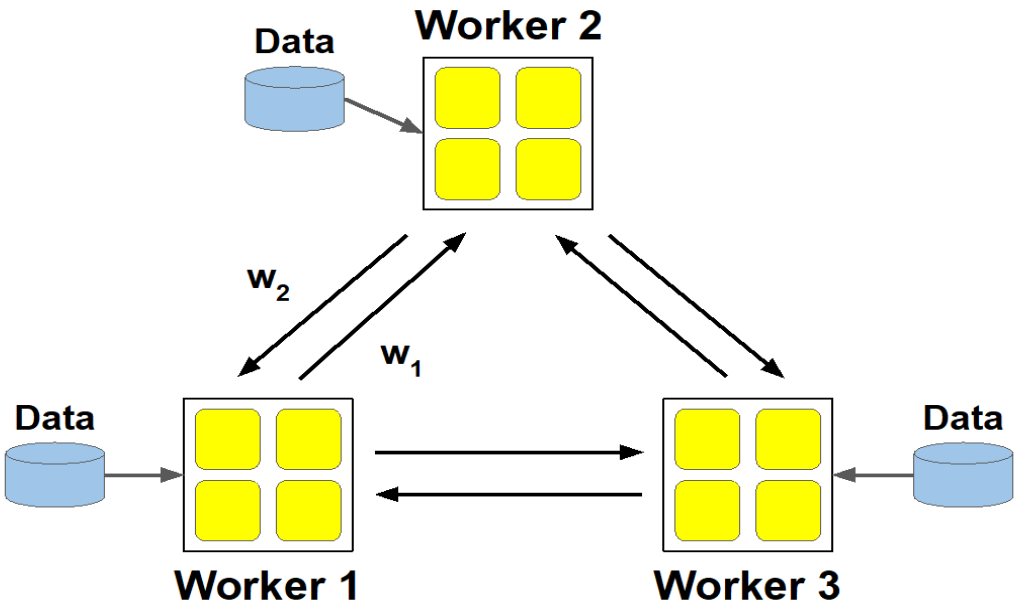
- The fetching and updating of gradients in the parameter server can be done either ***synchronously*** or ***asynchronously***.
- Both has pros and cons. Over-synchronization hurts performance where asynchrony is not-reproducible and might hurt convergence

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

Batch Parallelism #2

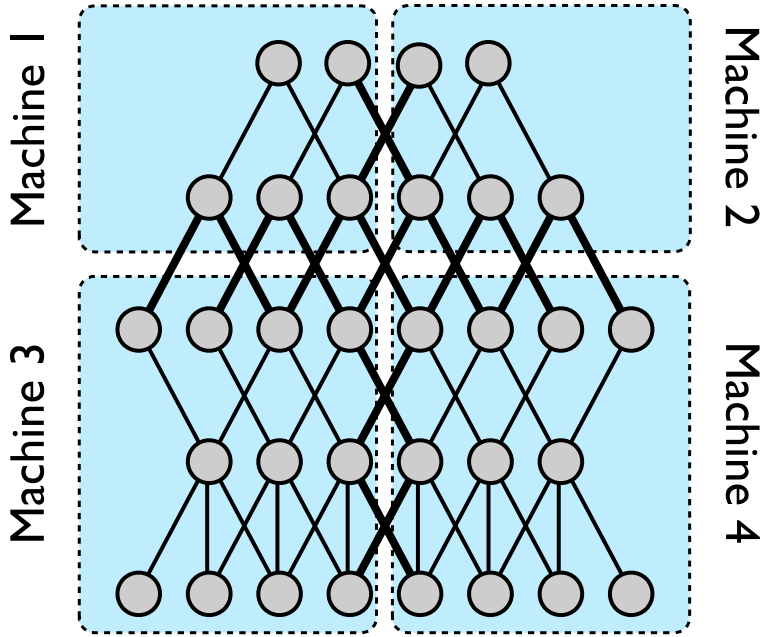
Options to avoid the parameter server bottleneck

1. **For synchronous SGD:** Perform all-reduce over the network to update gradients (good old MPI_Allreduce)
2. **For asynchronous SGD:** Peer-to-peer gossiping



Peter Jin, Forrest Iandola, Kurt Keutzer, “How to scale distributed deep learning?” NIPS ML Sys 2016

Model Parallelism



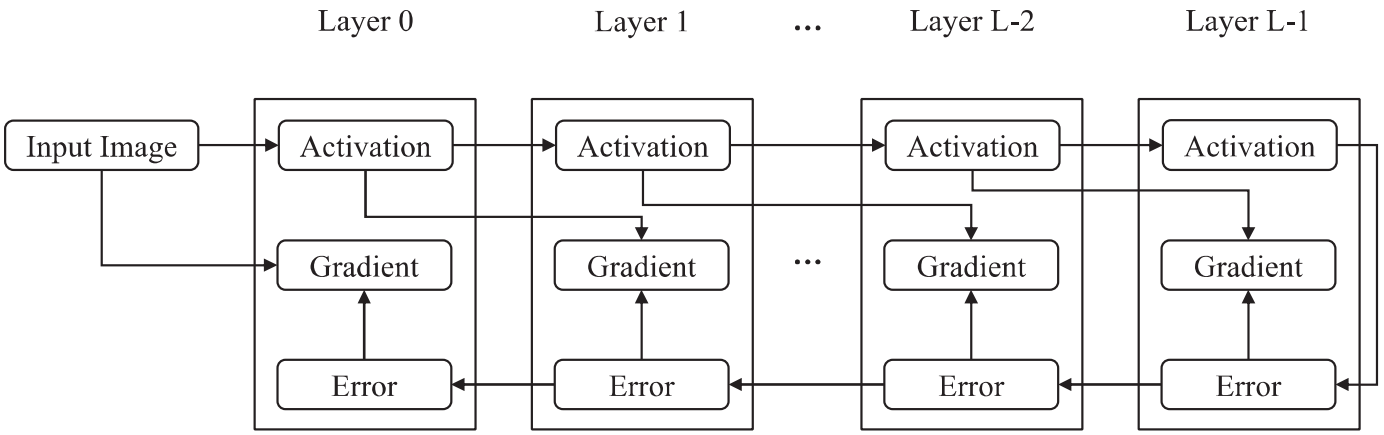
(inter layer parallelism is better called pipelining)

Interpretation #1: Partition your neural network into processors
Interpretation #2: Perform your matrix operations in parallel

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

The overlapping opportunities

- In backpropagation, the errors are propagated from the last layer to the first layer and data dependency exists between any two consecutive layers.
- In contrast, the gradients in different layers are independent of each other.



Activations are propagated from left to right in forward stage; errors are propagated from right to left in backpropagation stage. Gradients are computed using the activations and errors. Arrows indicated data dependencies.

S. Lee, et al. "Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication.", HiPC 2017

Algorithm 1 Mini-Batch SGD CNN Training Algorithm
(M: the number of mini-batches, L: the number of layers

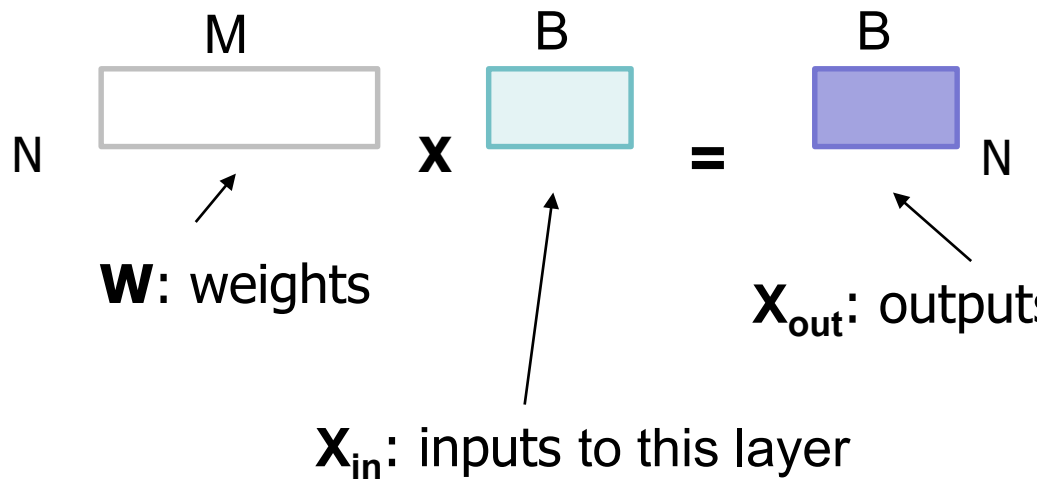
1: **for** each mini batch $m = 0, ...M-1$ **do**
2: Initialize $\Delta W = 0$
3: Get the m^{th} mini batch, D^m.
4: **for** each layer $l = 0, ...L-1$ **do**
5: Calculate activations A^l based on D^m.
6: **for** each layer $l = L-1, ...0$ **do**
7: Calculate errors E^l.
8: Calculate weight gradients ΔW^l.
9: **for** each layer $l = 0, ...L-1$ **do**
10: Update parameters, W^l and B^l.

Forward: $$a_n^l = \sigma(\sum_{i=0}^{|W|-1} w_i^l a_{n+i}^{l-1} + b_n^l)$$

Backward: $$e_n^l = \sum_{i=0}^{|W|-1} w_i^{l+1} e_{n-i}^{l+1},$$

where a_n^l, b_n^l, and e_n^l are the n^{th} activation, bias, and error in layer l respectively, w_i^l is an weight on the i^{th} connection between layer l and layer $l-1$, $|W|$ is the number of weights in a feature map, and σ is the activation function.

SGD training of NNs as matrix operations

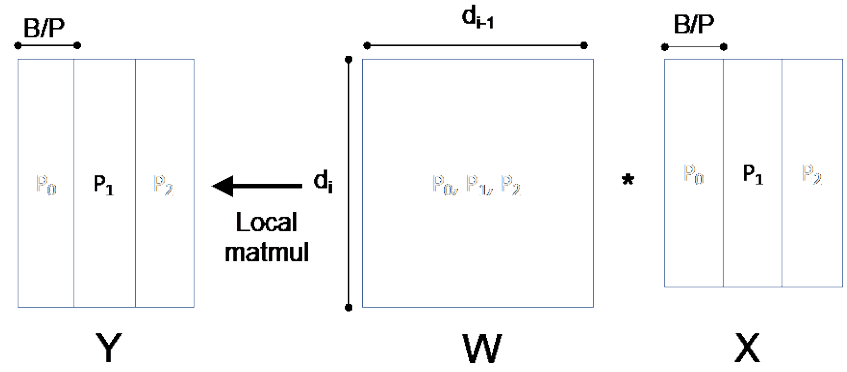


N = the number of outputs
M = the number of inputs
B = the size of the minibatch

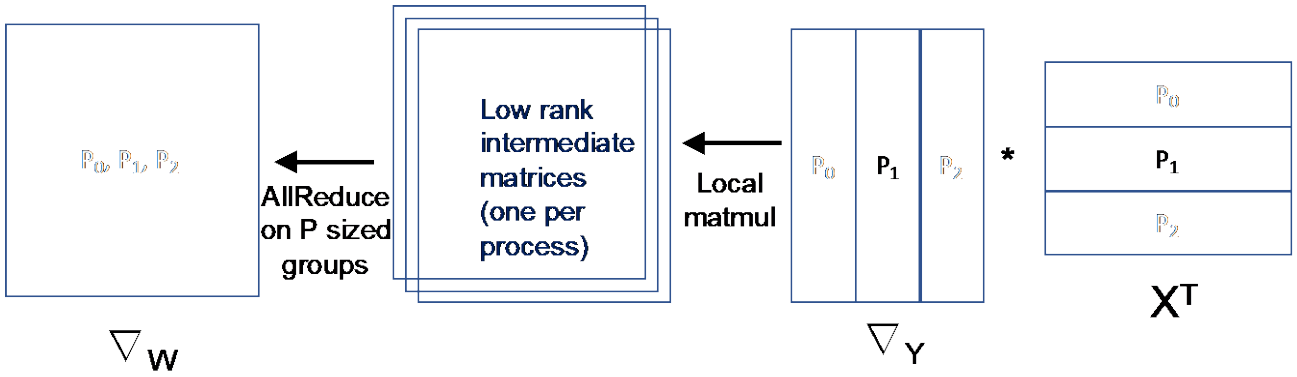
The impact to parallelism:

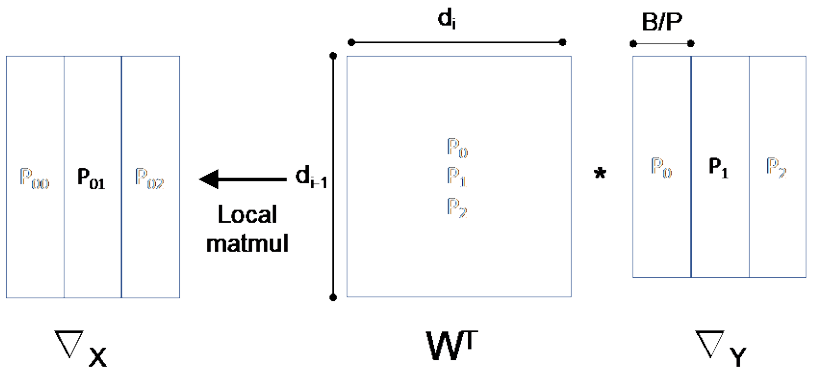
- **W** is replicated to processor, so it doesn't change
- $\mathbf{X_{in}}$ and $\mathbf{X_{out}}$ gets skinnier if we only use data parallelism, i.e. distributing **b=B/p** mini-batches per processor
- GEMM (i.e. matrix-matrix multiply) performance suffers as *matrix dimensions get smaller* and *more skewed*
- **Result:** Data parallelism can hurt single-node performance

Data Parallel SGD training of NNs as matrix operations



1. Which matrices are replicated?
2. Where is the communication?
3. Which steps can be overlapped?





$\nabla_Y = \partial L / \partial Y$ = how did the lost function change as output activations change?

$\nabla_X = \partial L / \partial X$

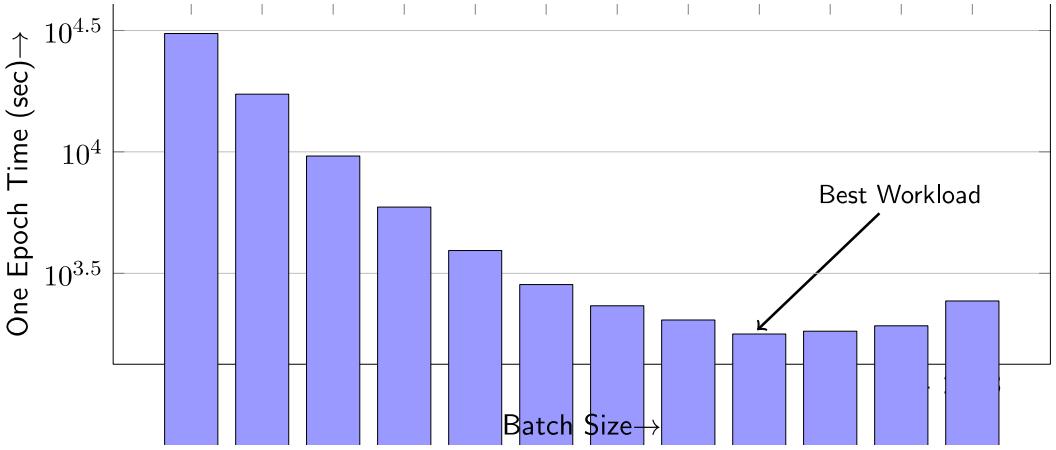
$\nabla_W = \partial L / \partial W$

Batch Parallel Strong Scaling

- **Per-iteration communication cost of batch parallelism is independent of the batch size:**
  - **larger batch → less communication per epoch (full pass over the data set)**

$$T_{comm}(batch) = 2\sum_{i=0}^{L}\left(\alpha\log(P) + \beta\frac{P-1}{P}|W_i|\right)$$

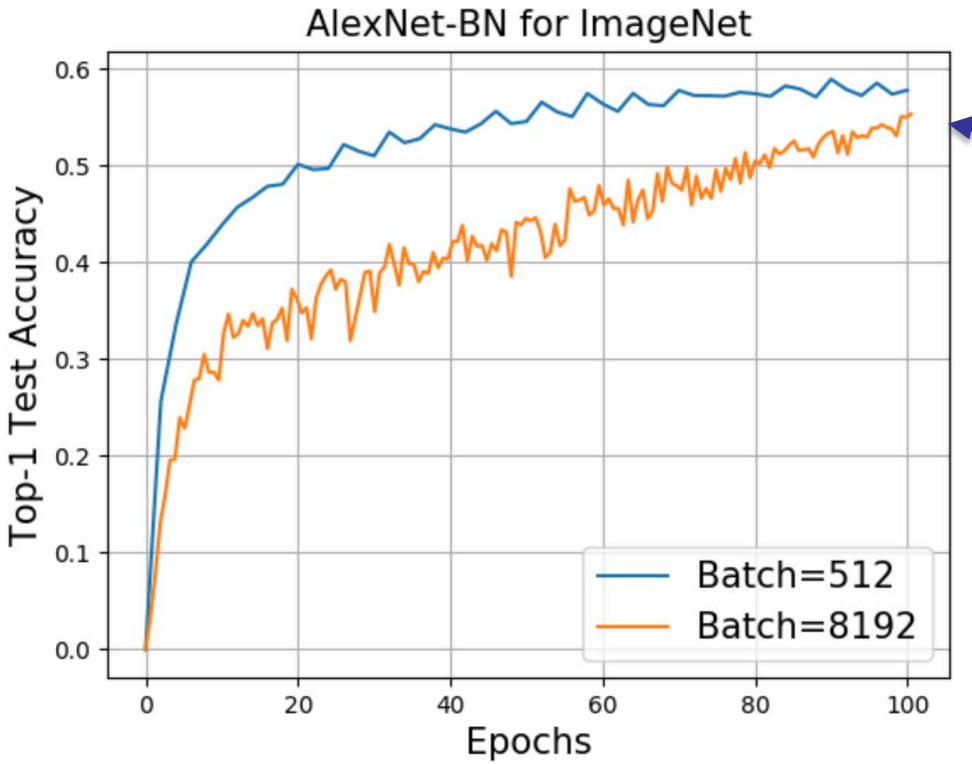
- But processor utilization goes down significantly for P>>1
  - Result: Batch parallel has poor strong scaling



One epoch training time of AlexNet computed on a single KNL

Problems with Batch Parallelism

- Batch parallel scaling is limited to B
  - Larger Batch -> higher strong scaling efficiency
- But SGD does not perform well for large batch

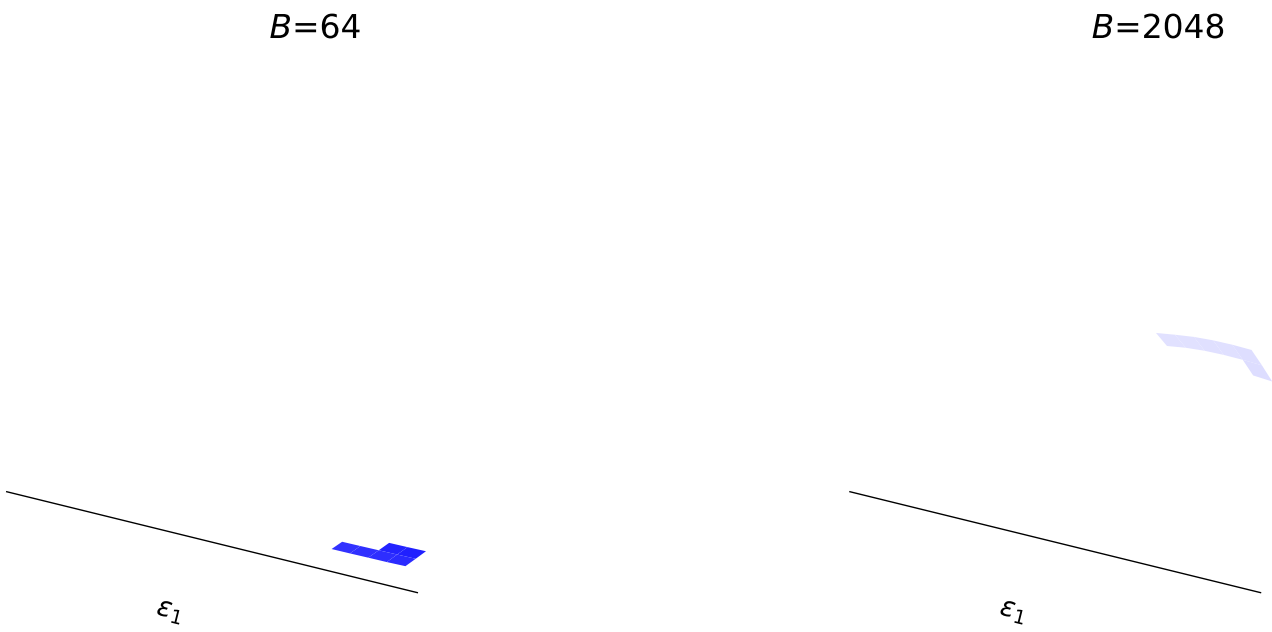


Performance Gap

Ginsburg, Boris, Igor Gitman, and Yang You. "Large Batch Training of Convolutional Networks with Layer-wise Adaptive Rate Scaling." (2018).

Problems with Batch Parallel

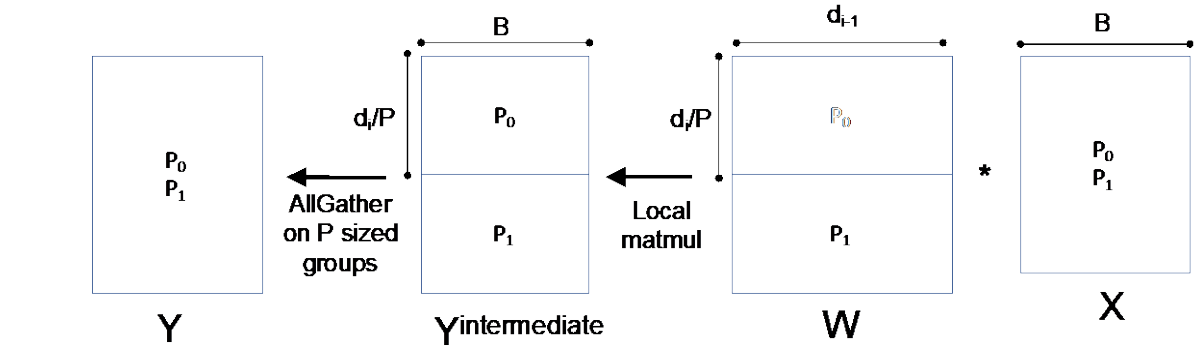
- The objective function implicitly changes in large batch SGD

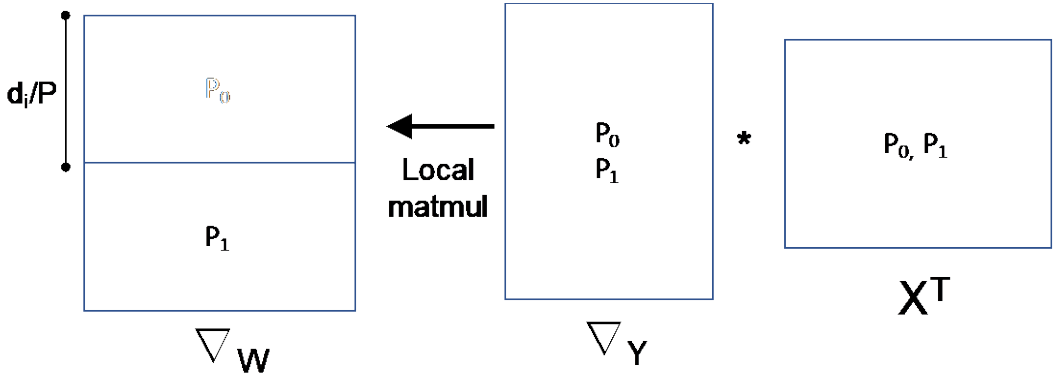


Landscape of loss function for small and large batch at the end of training. Large batch converges to a suboptimal point

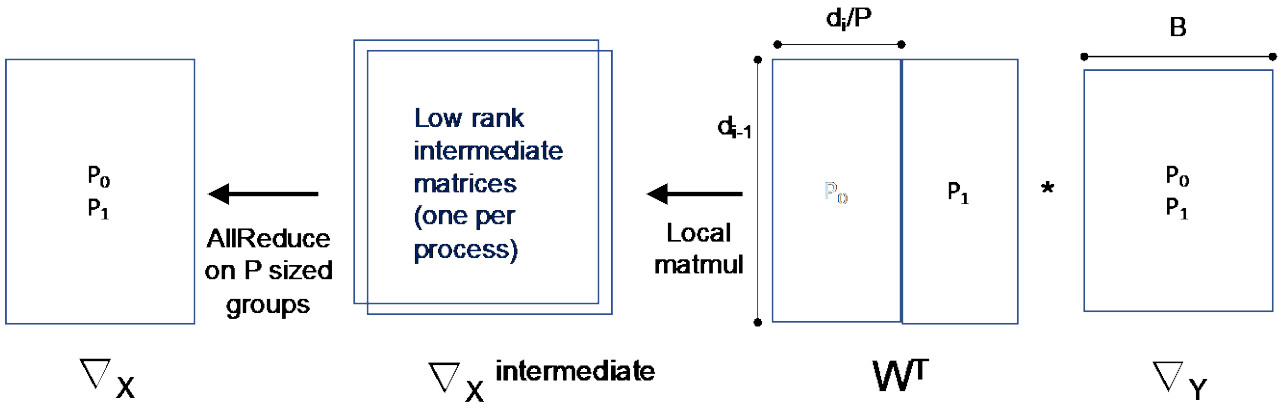
Z. Yao, A. Gholami, Q. Lei, K. Keutzer, M. Mahoney. Hessian-based Analysis of Large Batch Training and Robustness to Adversaries, arXiv:1802.08241

Model Parallel SGD training of NNs as matrix operations



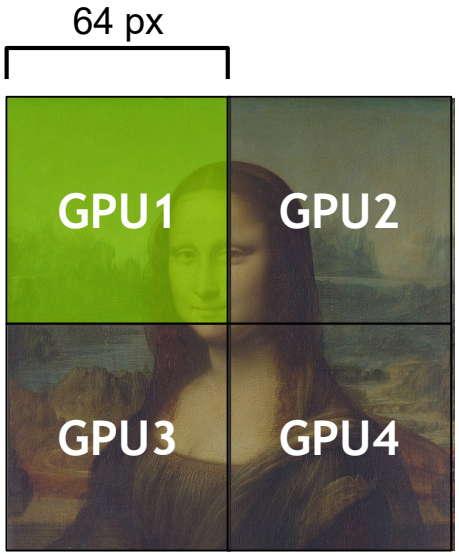


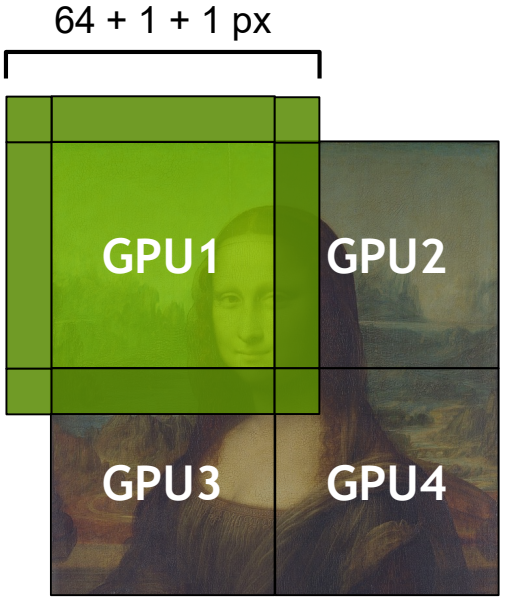
1. Which matrices are replicated?
2. Where is the communication?
3. How can matrix algebra capture both model and data parallelism?



Domain Parallel

- The general idea is the same as *halo regions* or *ghost zones* used to parallelize stencil codes in HPC
  - Before a convolution, exchange local receptive field boundary data



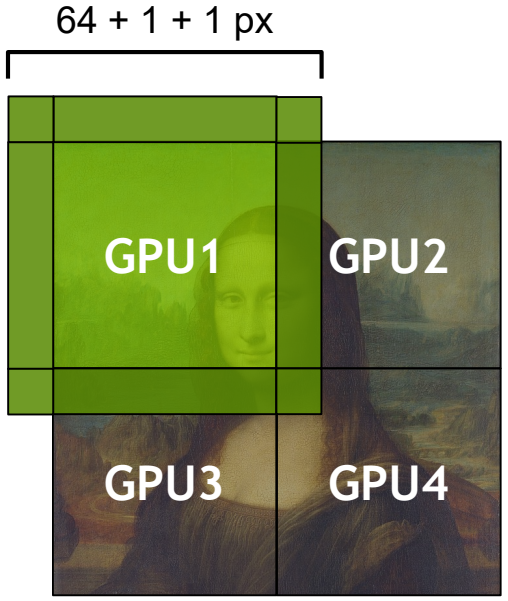


Peter Jin, Boris Ginsburg, and Kurt Keutzer. "Spatially Parallel Convolutions" ICLR Workshop Track, 2018

Communication Complexity of Domain Parallel

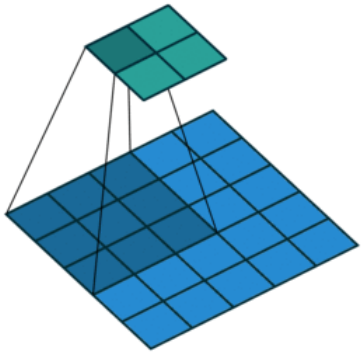
- Additional communication for halo exchange during forward and backwards pass
  - Negligible cost for early layers for which activation size is large (i.e. convolutional)

$$T_{comm}(domain) = \sum_{i=0}^{L} \left(\alpha + \beta B X_W^i X_C^i k_h^i / 2\right)$$
$$+ \sum_{i=0}^{L} \left(\alpha + \beta B Y_W^i Y_C^i k_w^i / 2\right)$$
$$+ 2 \sum_{i=0}^{L} \left(\alpha \log(P) + \beta \frac{P-1}{P} |W_i|\right)$$



Domain Parallel Scaling

- Domain parallel scaling on V100 GPUs
  - B=32, C=64, K=3, D=1, R=1



Resolution	GPUs	Fwd. wall-clock	Bwd. wall-clock
128 × 128	1	2.56 ms (1.0×)	6.63 ms (1.0×)
	2	1.52 ms (1.7×)	3.50 ms (1.9×)
	4	**1.23 ms (2.1×)**	**2.33 ms (2.8×)**
256 × 256	1	10.02 ms (1.0×)	26.81 ms (1.0×)
	2	5.34 ms (1.9×)	11.79 ms (2.3×)
	4	**3.11 ms (3.2×)**	**6.96 ms (3.9×)**
512 × 512	1	45.15 ms (1.0×)	126.11 ms (1.0×)
	2	20.18 ms (2.2×)	60.15 ms (2.1×)
	4	**10.65 ms (4.2×)**	**26.76 ms (4.7×)**

Peter Jin, Boris Ginsburg, and Kurt Keutzer. "Spatially Parallel Convolutions" ICLR Workshop Track, 2018
Figure: Dumoulin, V., Visin, F.. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016.