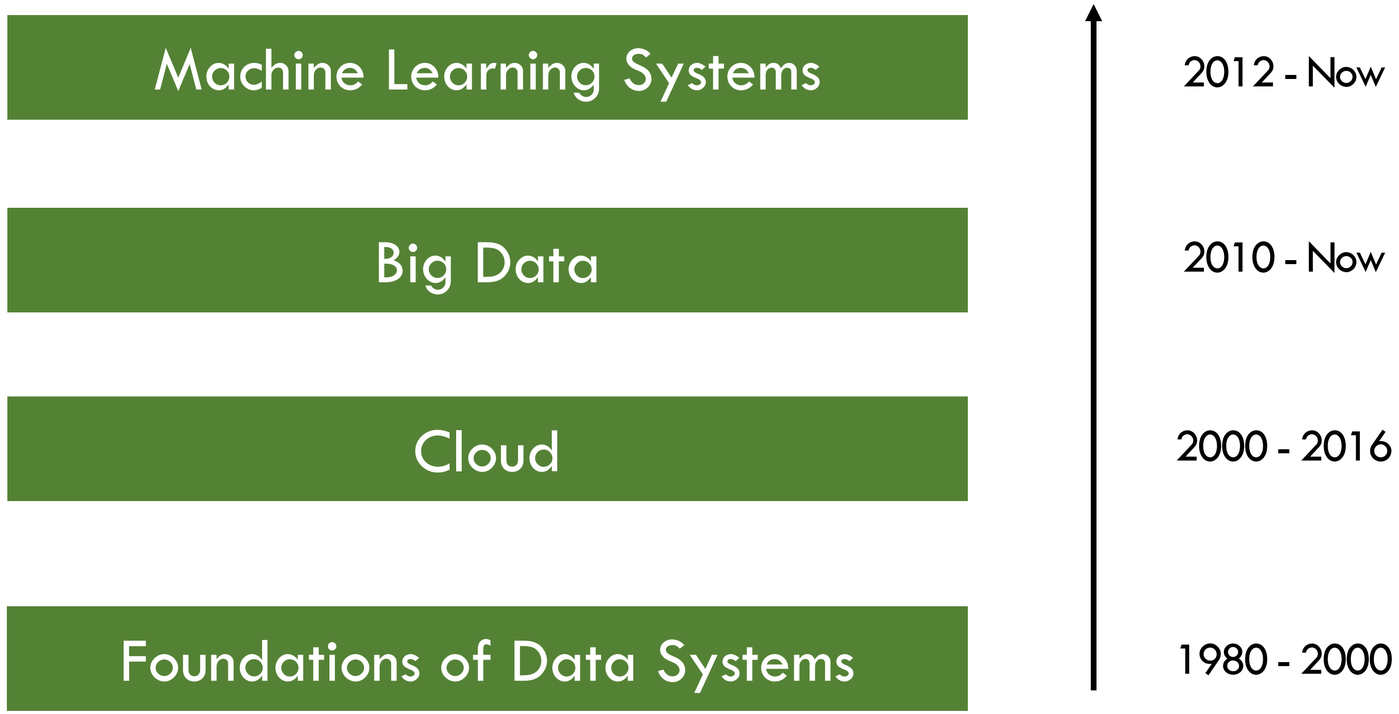
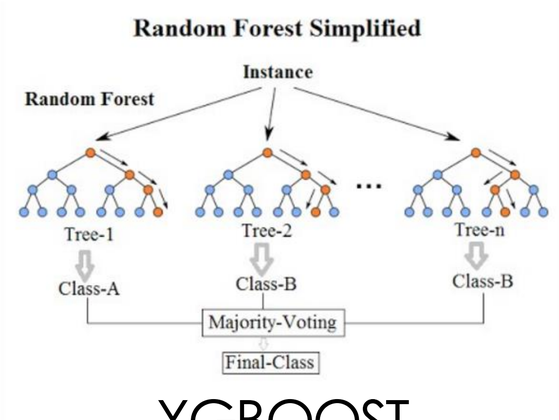
Where We Are


Machine Learning Systems
Big Data
Cloud
Foundations of Data Systems
2012 - Now
2010 - Now
2000 - 2016
1980 - 2000

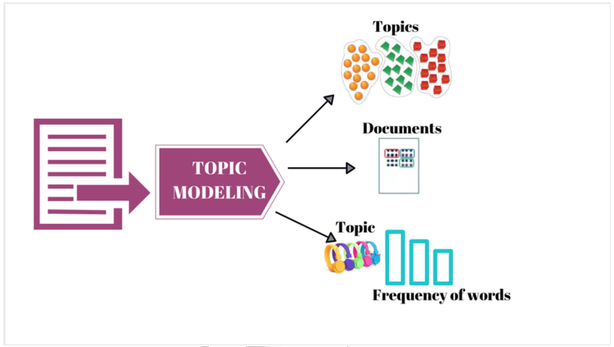
ML Era (roughly starts from 2008, even before Spark has taken off)

- ML wes still very diverse (a.k.a. in a mess) in 2012

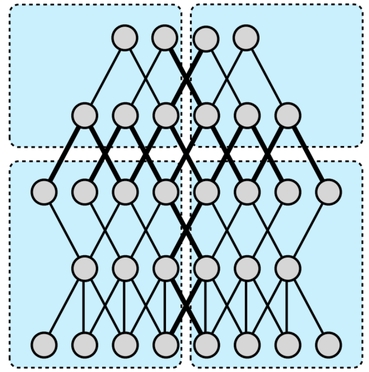


XGBOOST

Spark mllib



LDA



Torch (lua) / Theano / distbelief

Diversity -> Good News or Bad New?

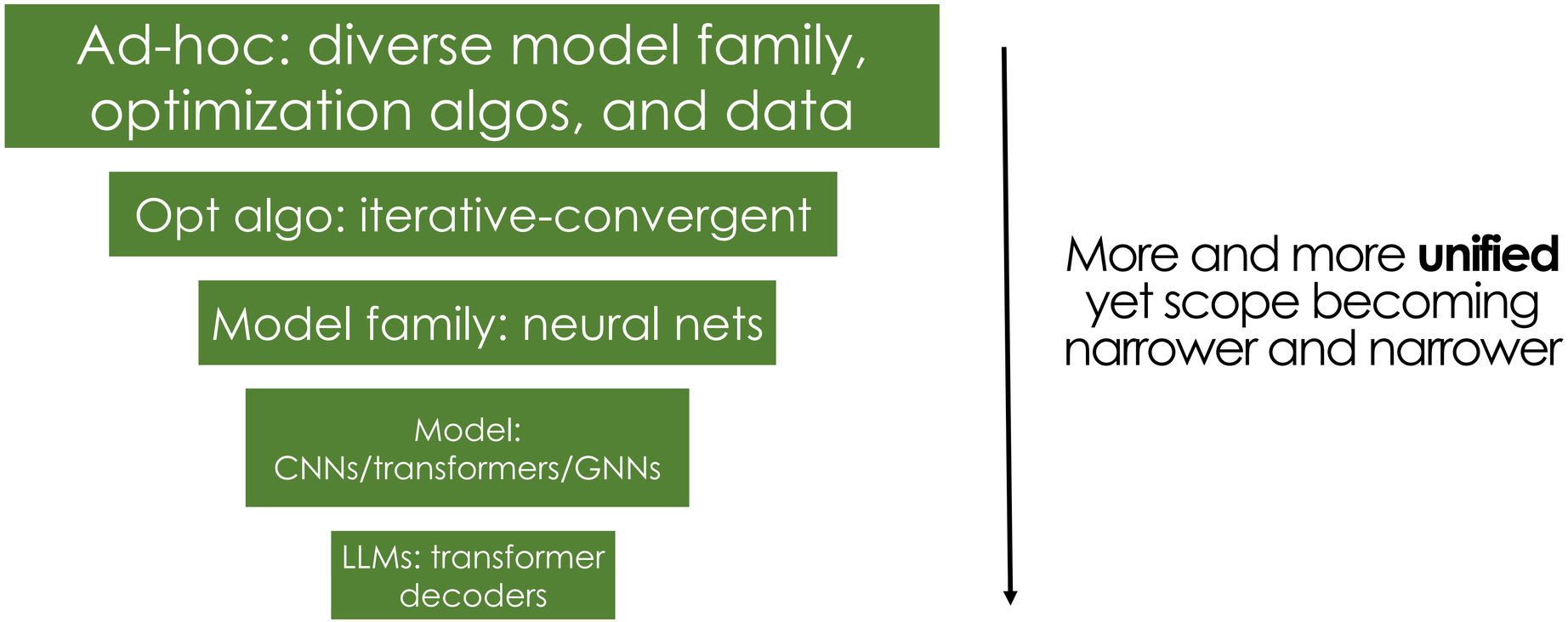
- ML is so diverse
  - Cons:
    - There is no unified model / computation
    - Hard to build a programming model / interface that cover a diverse range of applications
    - No idea where the system bottlenect is
  - Pros:
    - A lot of opportunities: Gold mining era

ML Systems Plan in DSC 204A

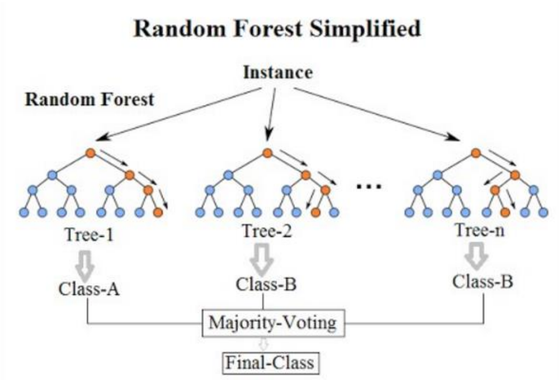
- ML System history: history of unification
- First Unification: iterative-convergence algorithm
  - Parameter server
- Second unification: Neural networks
  - Autodiff libraries: tensorflow, pytorch, etc.
- Third unification: scaling up transformers and LLMs
  - Flash attention, paged attention, and how to scale up
- Want to dive into each topic? Enroll DSC 299 offered next quarter

ML System history

- ML Systems evolve as more and more ML components (models/optimization algorithms) are unified

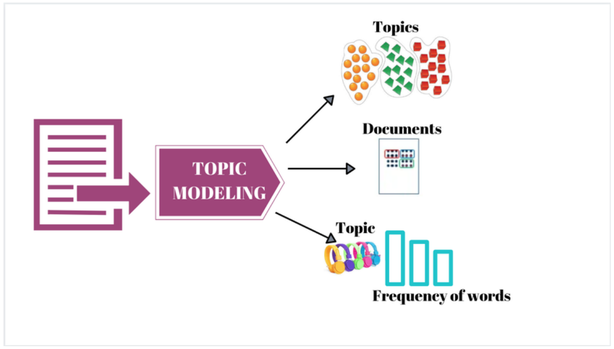


The first Unified component: Iterative-convergence Algo

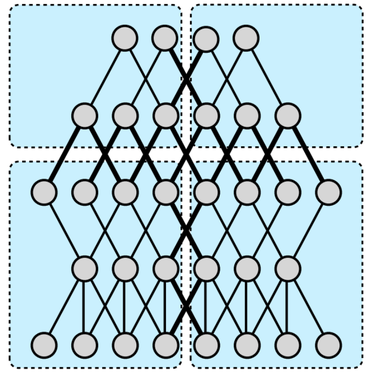


Gradient boosting tree

Coordinate descent



EM Algorithm



Gradient descent

Example: Gradient Descent

Gradient / backward computation

Recall collective communication

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$

objective data

- The first unification:
  - Most ML algorithms are **iterative-convergent**
  - **iterative-convergent** is the master equation behind

How to Distribute this Equation?

Gradient / backward computation

$$\theta^{(t)} = \theta^{(t-1)} + \boxed{\varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})}$$

objective data

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

How to perform this sum?

Problems if expressing this in Spark

- ML is too diverse; hard to express their computation in coarse-grained data transformations.

$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

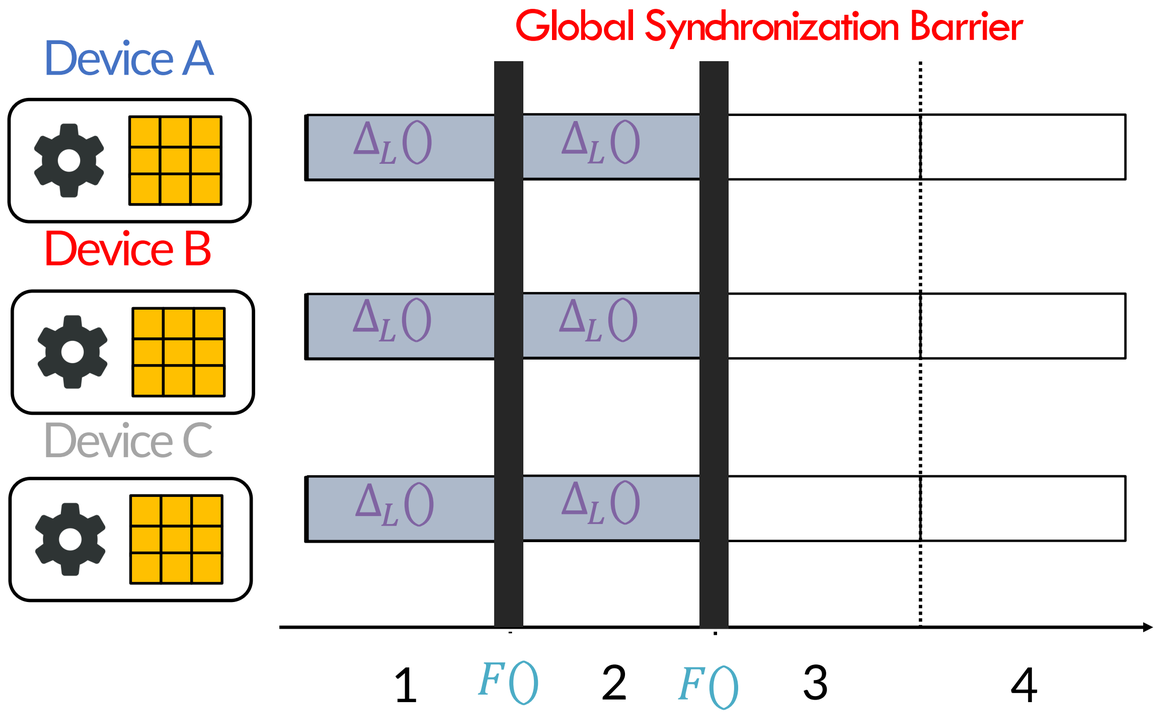
Problems if expressing this in Spark

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

- Very heavy communication per iteration
- Compute : communication = 1:10 in the era of 2012

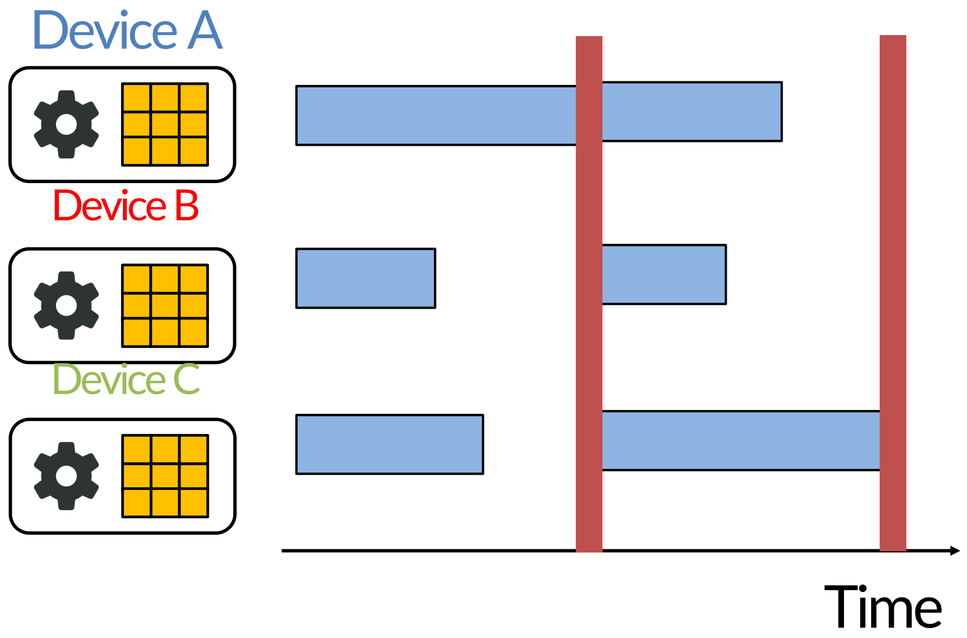
Consistency

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$



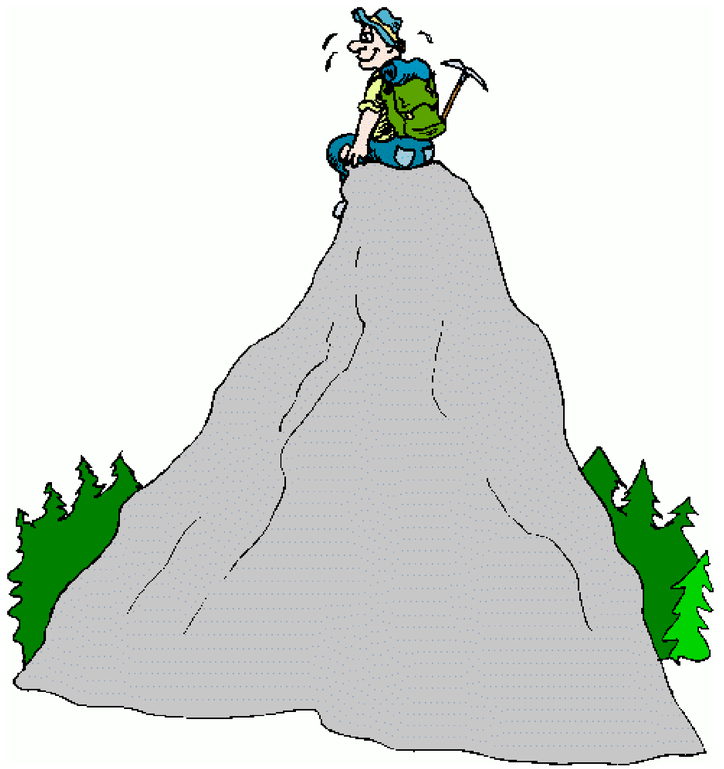
BSP's Weakness: Stragglers

- **BSP suffers from stragglers**
  - Slow devices (stragglers) force all devices to wait
  - More devices → higher chance of having a straggler
- **Stragglers are usually transient, e.g.**
  - Temporary compute/network load in multi-user environment
  - Fluctuating environmental conditions (temperature, vibrations)
- BSP's throughput is greatly decreased in large clusters/clouds, where stragglers are unavoidable

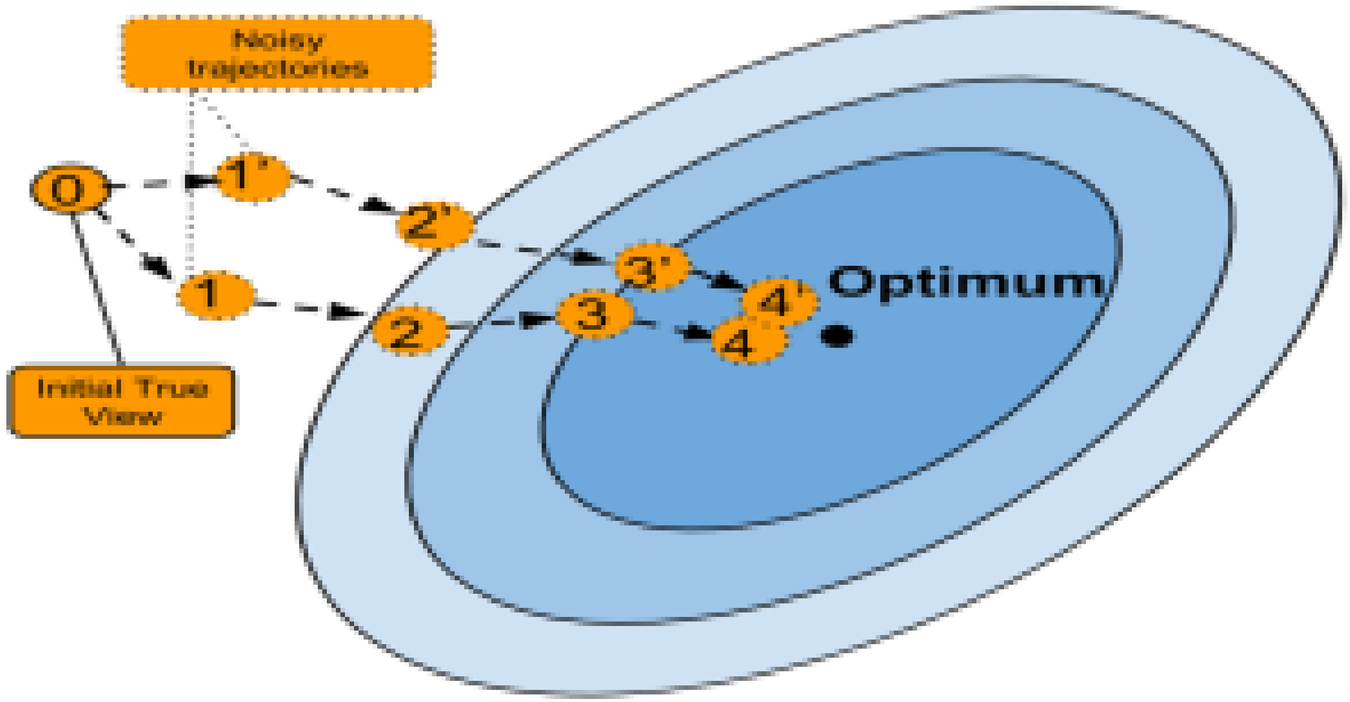


An interesting property of Gradient Descent (ascent)

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

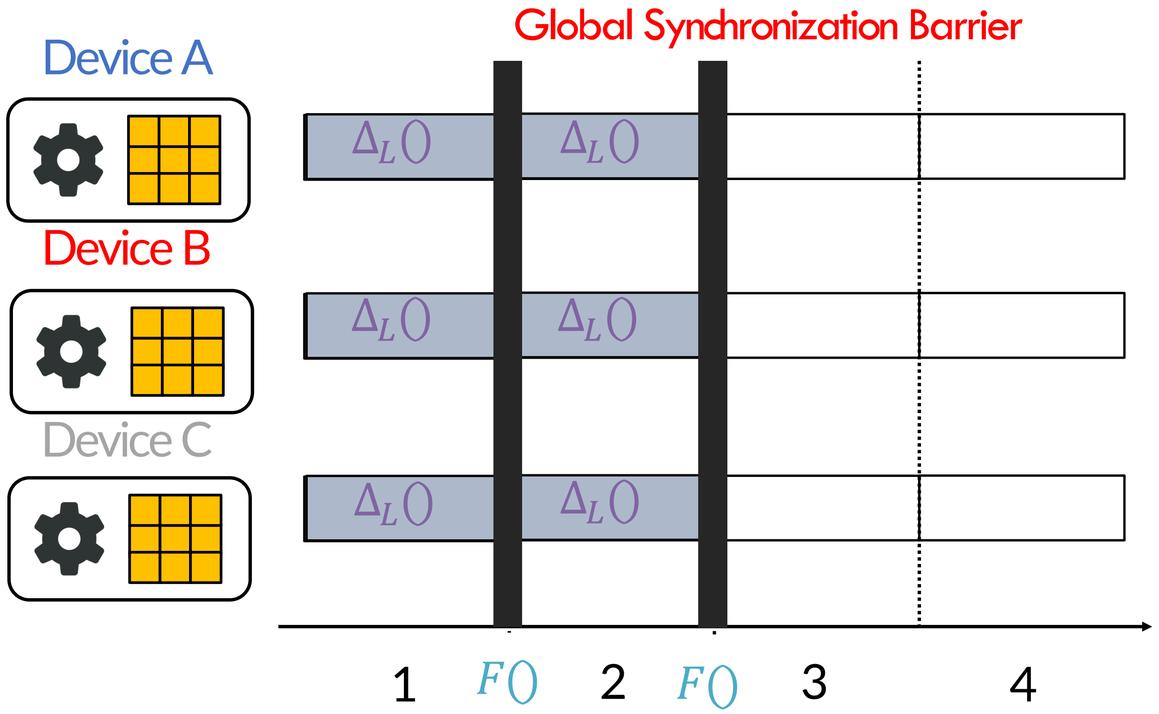


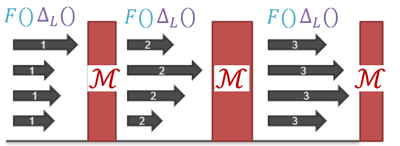
Machine Learning is Error-tolerant (under certain conditions)



Background: Strict Consistency

- **Baseline:** Bulk Synchronous Parallel (BSP)
  - MapReduce, Spark, many DistML Systems
- Devices compute updates $\Delta_L()$ between global barriers (iteration boundaries)
  - Messages $\mathcal{M}$ exchanged only during barriers
- **Advantage: Execution is serializable**
  - Same guarantees as sequential algo!
  - Provided that aggregation $F()$ is agnostic to order of messages $\mathcal{M}$ (e.g. in SGD)

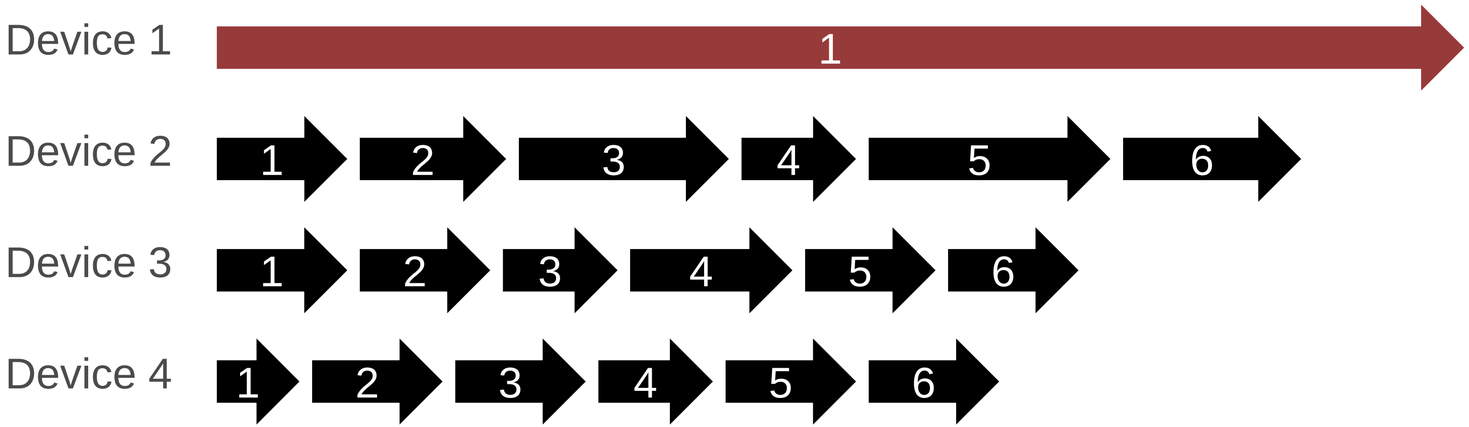




Background: Asynchronous Communication (No Consistency)

- **Asynchronous (Async):** removes all communication barriers
  - Maximizes computing time
  - Transient stragglers will cause messages to be extremely stale
    - Ex: Device 2 is at $t = 6$, but Device 1 has only sent message for $t = 1$

- **Some Async software:** messages can be applied while computing $F()$, $\Delta_L()$
  - Unpredictable behavior, can hurt statistical efficiency!

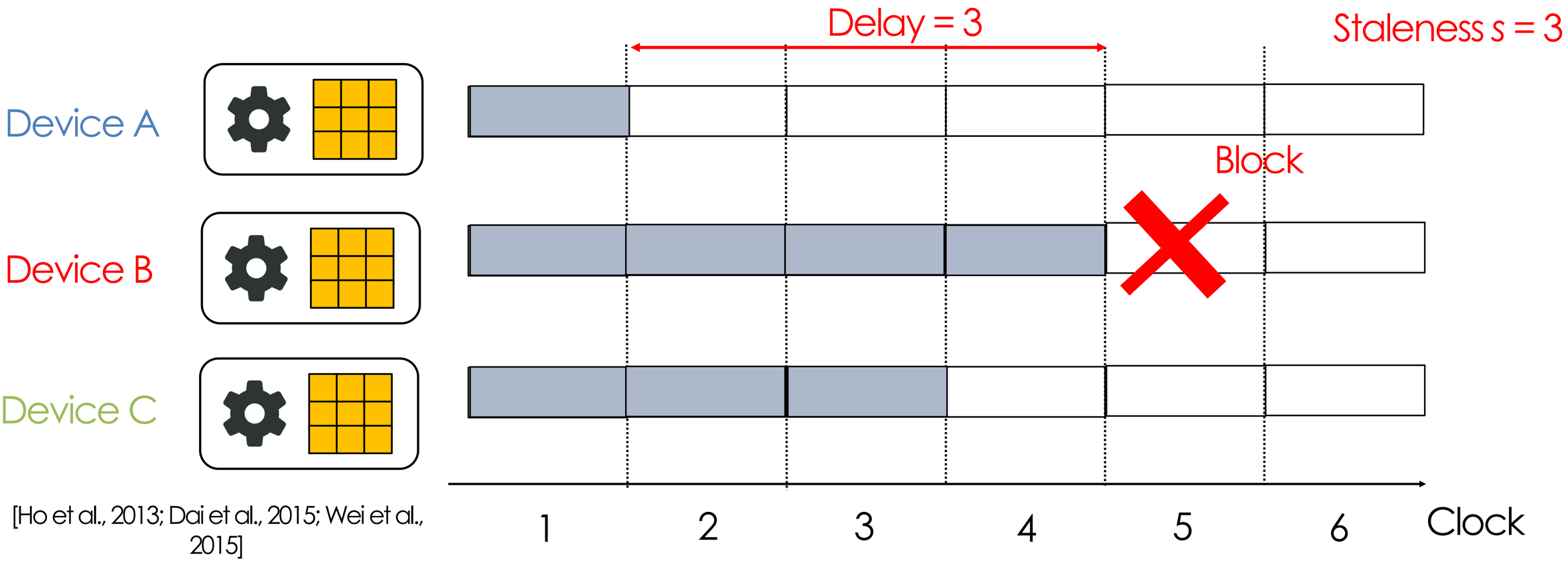


Background: Bounded Consistency

Bounded consistency models: Middle ground between BSP and fully-asynchronous (no-barrier)

e.g. Stale Synchronous Parallel (SSP): Devices allowed to iterate at different speeds

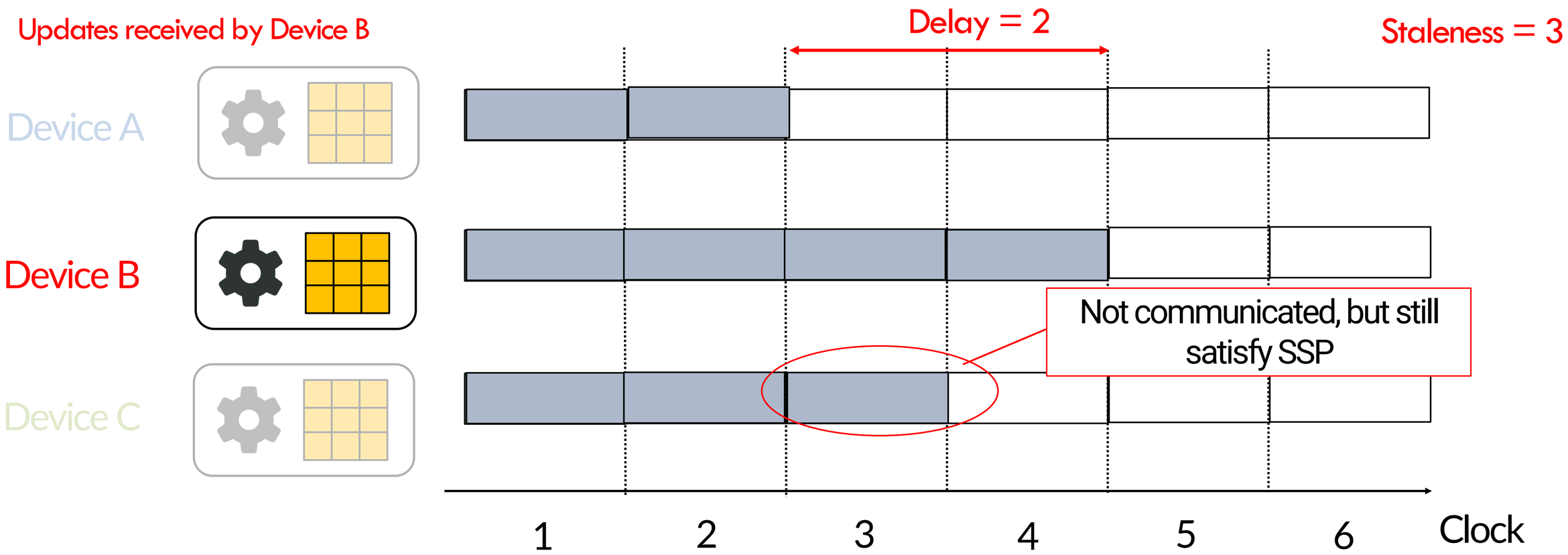
- Fastest & slowest device must not drift > s iterations apart (in this example, $s = 3$)
  - s is the maximum staleness



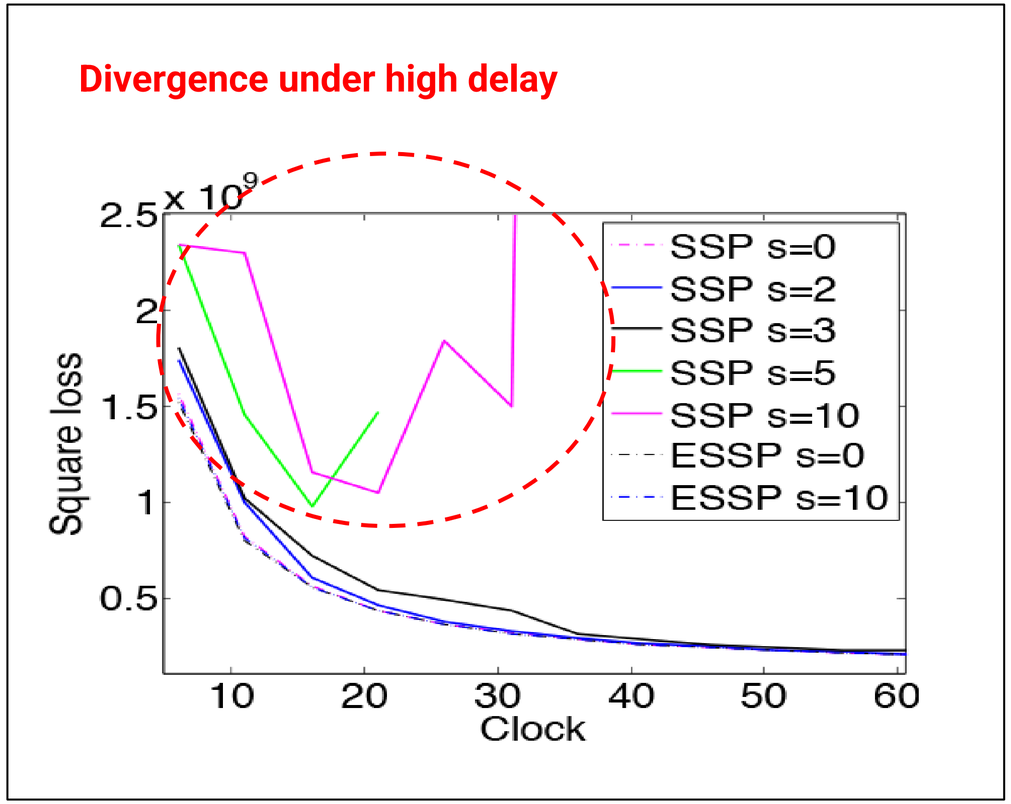
[Ho et al., 2013; Dai et al., 2015; Wei et al., 2015]

SSP: “Lazy” Communication

SSP: devices avoid communicating unless necessary
- i.e. when staleness condition is about to be violated
- Favors throughput at the expense of statistical efficiency



Impacts of Consistency/Staleness: Unbounded Staleness

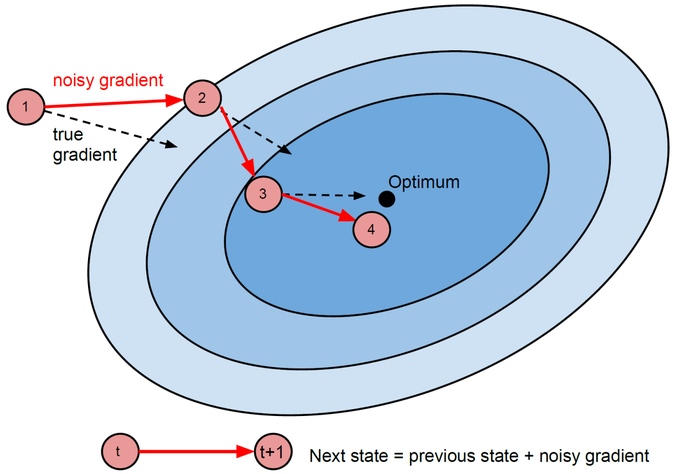


Theory: SSP Expectation Bound

- **Goal:** minimize convex $f(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x})$

  (Example: Stochastic Gradient)

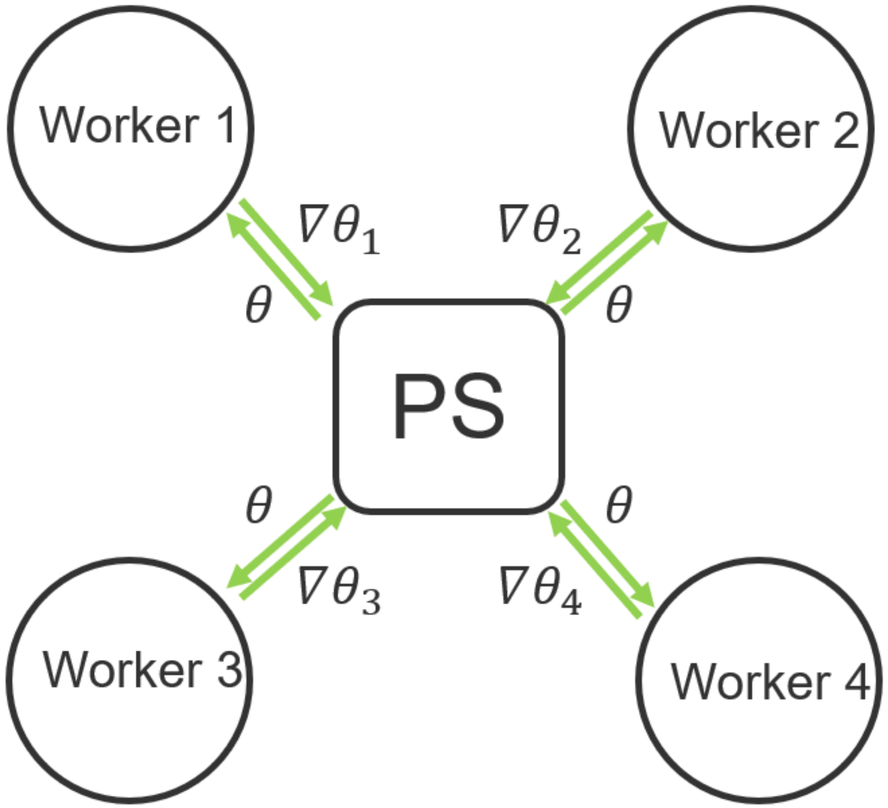
  - L-Lipschitz, problem diameter bounded by F^2
  - Staleness s, using P parallel devices
  - Use step size $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$
- **(E)SSP converges according to**
  - Where T is the number of iterations



Difference between SSP estimate and true optimum

$$R[\mathbf{X}] := \left[\frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{\mathbf{x}}_t)\right] - f(\mathbf{x}^*) \leq 4FL\sqrt{\frac{2(s+1)P}{T}}$$

Parameter Server Naturally emerges

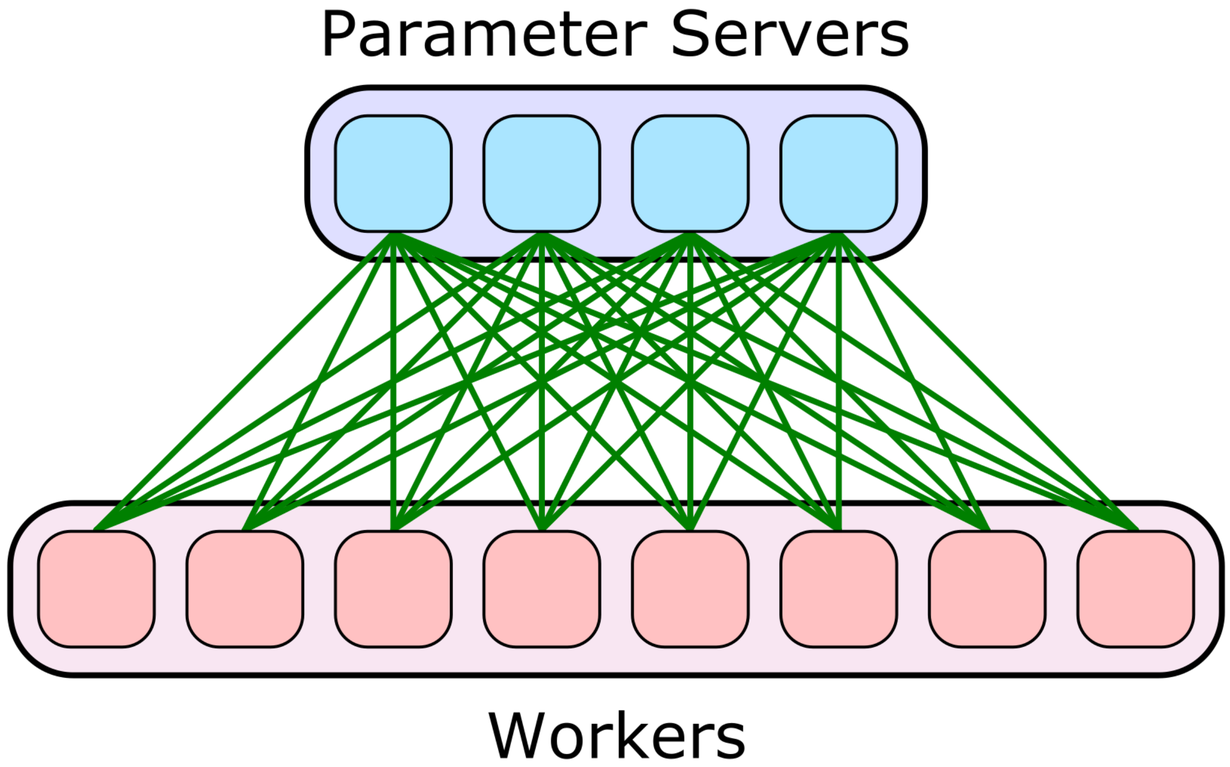


How to Implement Parameter Server?

- Key considerations:
  - Server: Communication bottleneck
  - Fault tolerance
  - Programming Model
  - Handling GPUs

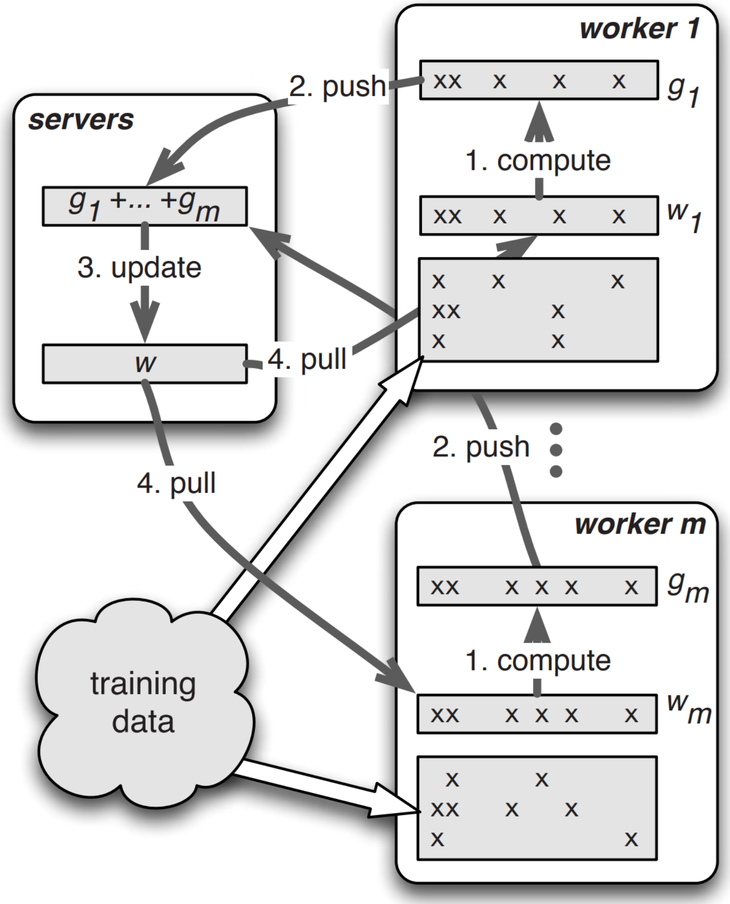
Parameter Server Implementation

- Sharded parameter server: sharded KV stores
  - Avoid communication bottleneck
  - Redundancy across different PS shards



Programming Model

- Client:
  - Push()
  - Pull()
  - Compute()
- Server:
  - Update()
- Very similar to the spirit of Map Reduce
- A lot of flexibility for users to customize
  - Recall Mapreduce vs. Spark



Summary: Parameter Server

- Why does it emerge?
  - Unification of iterative-convergence optimization algorithm
- What problems does it address and how?
  - Heavy communication, via flexible consistency
- Pros?
  - Cope well with iterative-convergent algo
- Cons?
  - Extension to GPUs?
  - Strong assumption on communication bottleneck

The Second Unified Component: Neural Networks

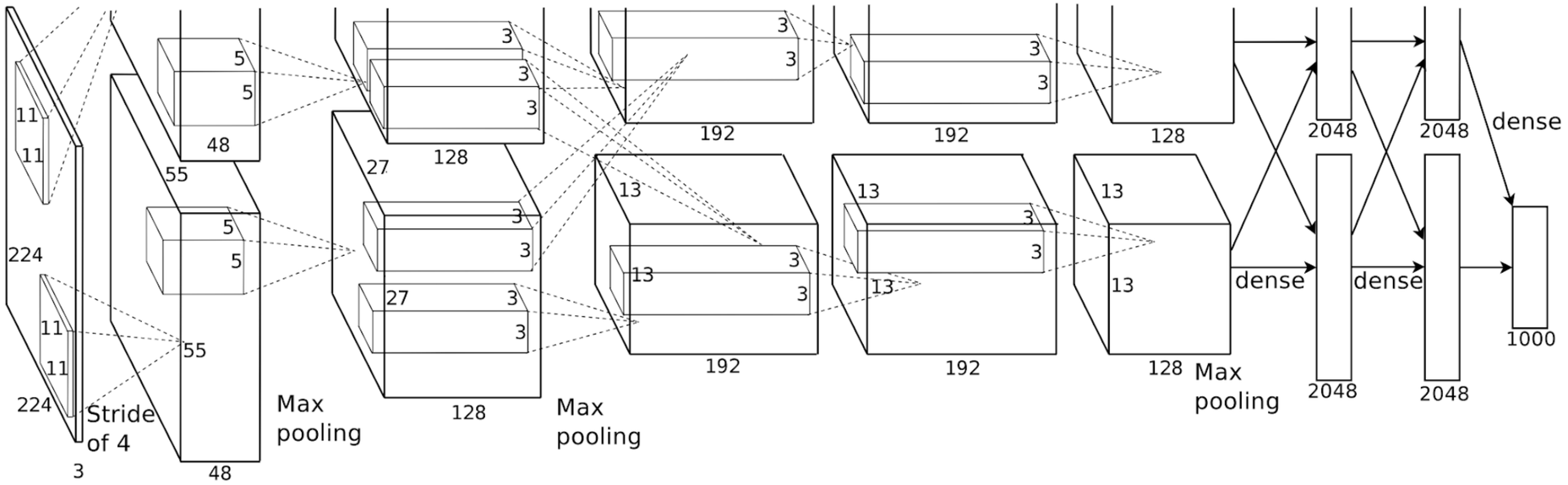


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Figure from AlexNet
[Krizhevsky et al., NeurIPS 2012], [Krizhevsky et al., preprint, 2014]

Deep learning Emerges

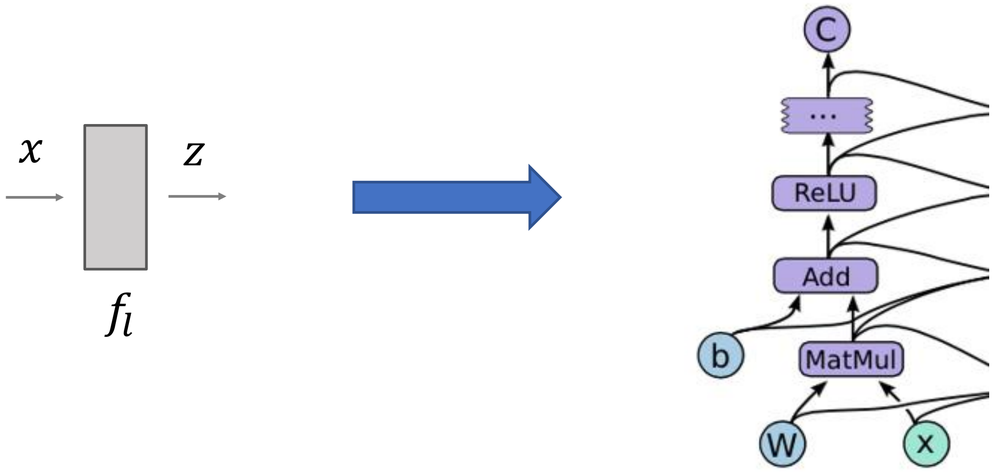
- Still iterative-convergent: because of using SGD
- GPU becomes a must
- Neural network architecture itself can be very diverse
  - But less diverse than the whole spectrum of all ML models
  - Still needs a sufficiently expressive lib to program various architectures
  - Map-reduce, Push/Pull are too coarse grained
- It starts with a relatively small model
  - Spark is too bulky
  - Spark op lib does not align well with neural network ops

Outline

- **Deep Learning as Dataflow Graphs**
- Auto-differentiable Libraries

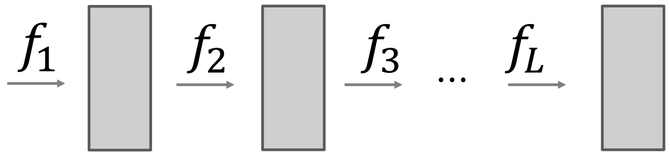
A Computational Layer in DL

- A layer in a neural network is composed of a few finer computational operations
  - A layer l has input x and output z, and transforms x into z following: $y = Wx + b, z = ReLU(y)$
  - Denote the transformation of layer l as f_l, which can be represented as a dataflow graphs: the input x flow though the layer



From Layers to Networks

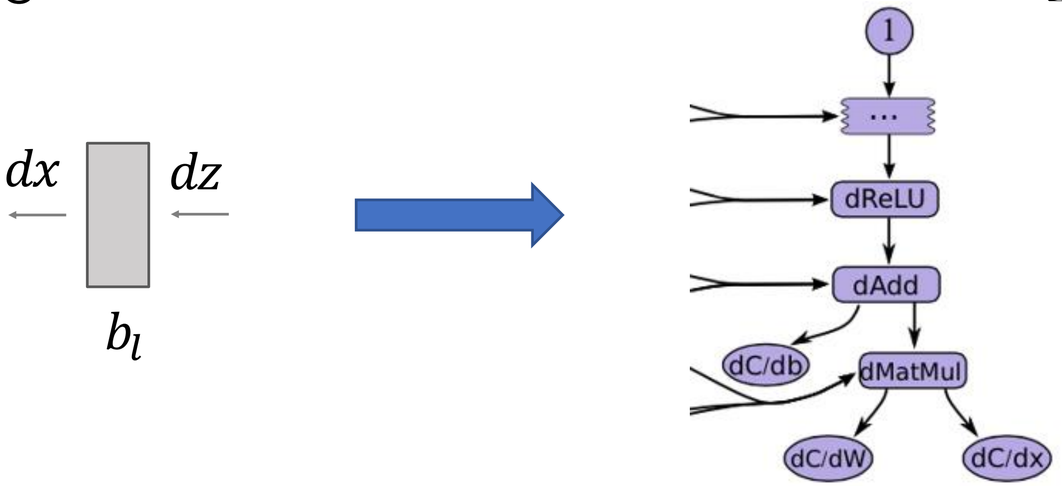
- A neural network is thus a few stacked layers $l = 1, \dots, L$, where every layer represents a function transform f_l
  - The forward computation proceeds by sequentially executing $f_1, f_2, f_3, \dots, f_L$



  - Training the neural network involves deriving the gradient of its parameters with a backward pass (next slides)

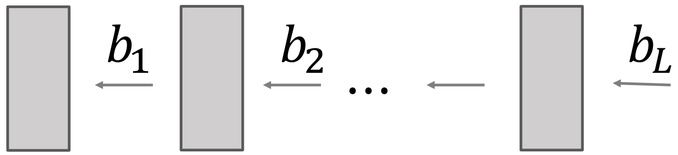
A Computational Layer in DL

- Denote the backward pass through a layer l as b_l
  - b_l derives the gradients of the input $x(\mathrm{d}x)$,given the gradient of z as $\mathrm{d}z$, as well as the gradients of the parameters $\mathbf{W}, \mathbf{b}$
  - $\mathrm{d}x$ will be the backward input of its previous layer $l-1$
  - Backward pass can be thought as a backward dataflow where the gradient flow through the layer



Backpropagation through a NN

- The backward computation proceeds by sequentially executing $b_L, b_{L-1}, b_{L-2}, \dots, b_1$



A Layer as a Dataflow Graph

- Give the forward computation flow, gradients can be computed by auto differentiation
  - Automatically derive the backward gradient flow graph from the forward dataflow graph

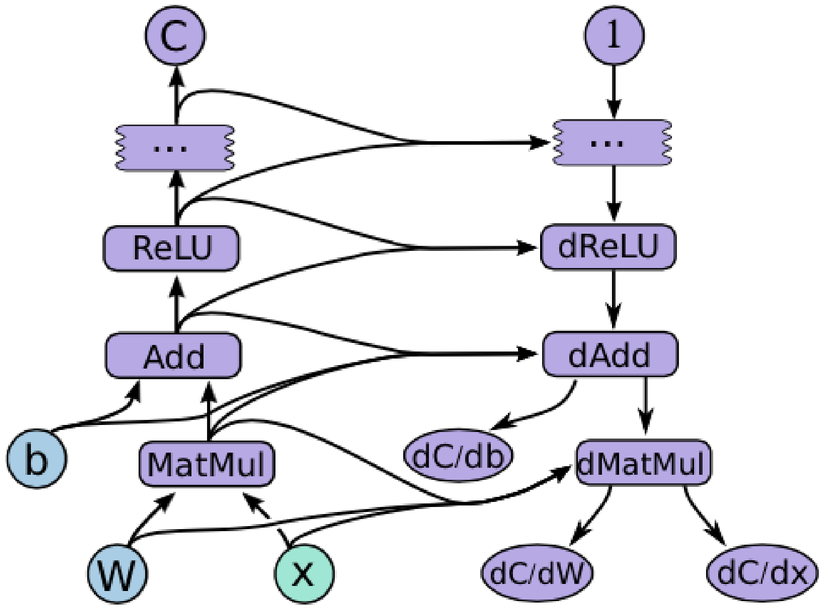
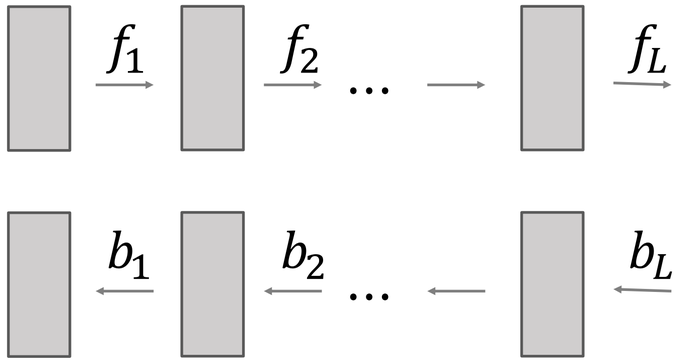


Photo from TensorFlow website

A Network as a Dataflow Graph

- Gradients can be computed by auto differentiation
  - Automatically derive the gradient flow graph from the forward dataflow graph



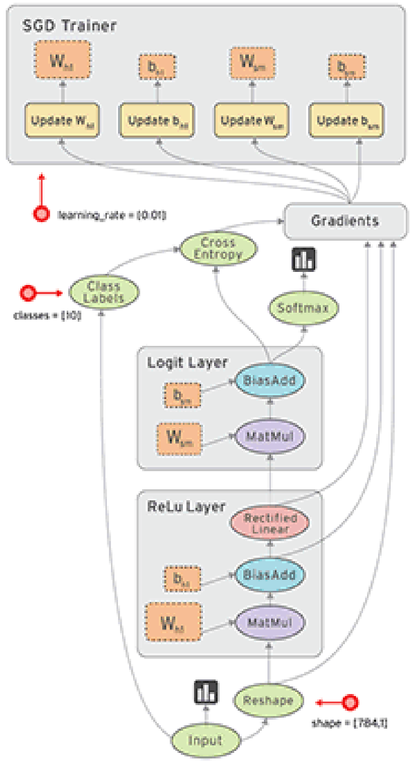


Photo from TensorFlow website

Dataflow Graph Programming Model Today

- Define a neural network
  - Define operations and layers: fully-connected? Convolution? Recurrent?
  - Define the data I/O: read what data from where?
  - Define a loss function/optimization objective: L2 loss? Softmax? Ranking Loss?
  - Define an optimization algorithm: SGD? Momentum SGD? etc
- Auto-differential Libraries will then take over
  - Connect operations, data I/O, loss functions and trainer.
  - Build forward dataflow graph and backward gradient flow graphs.
  - Perform training and apply updates

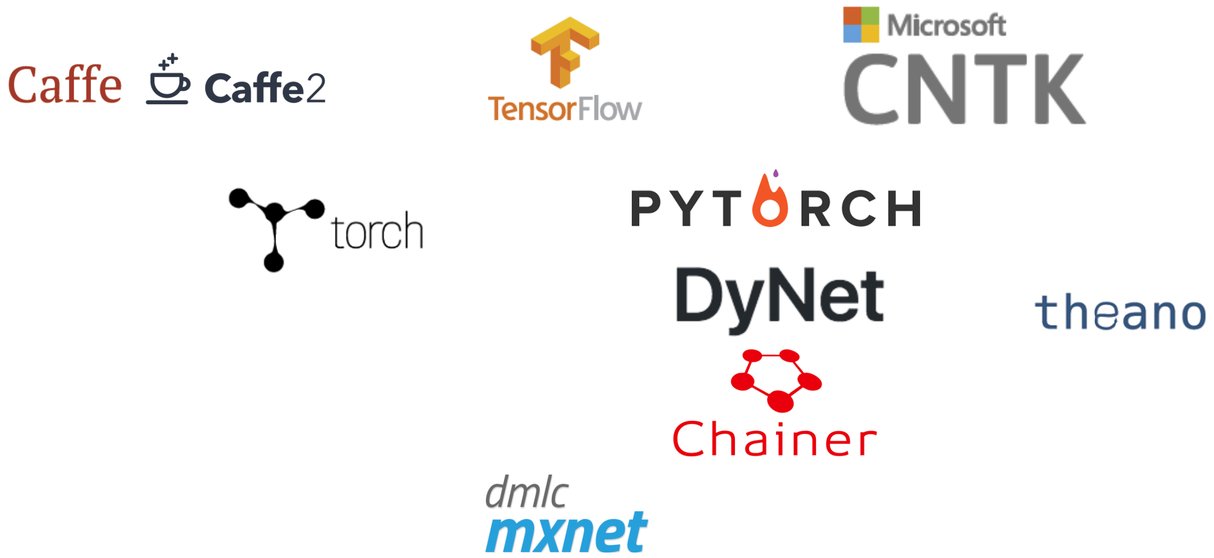
Compare this vs. Spark, parameter server, XGBoost?

Outline

- Deep Learning as Dataflow Graphs
- **Auto-differentiable Libraries**

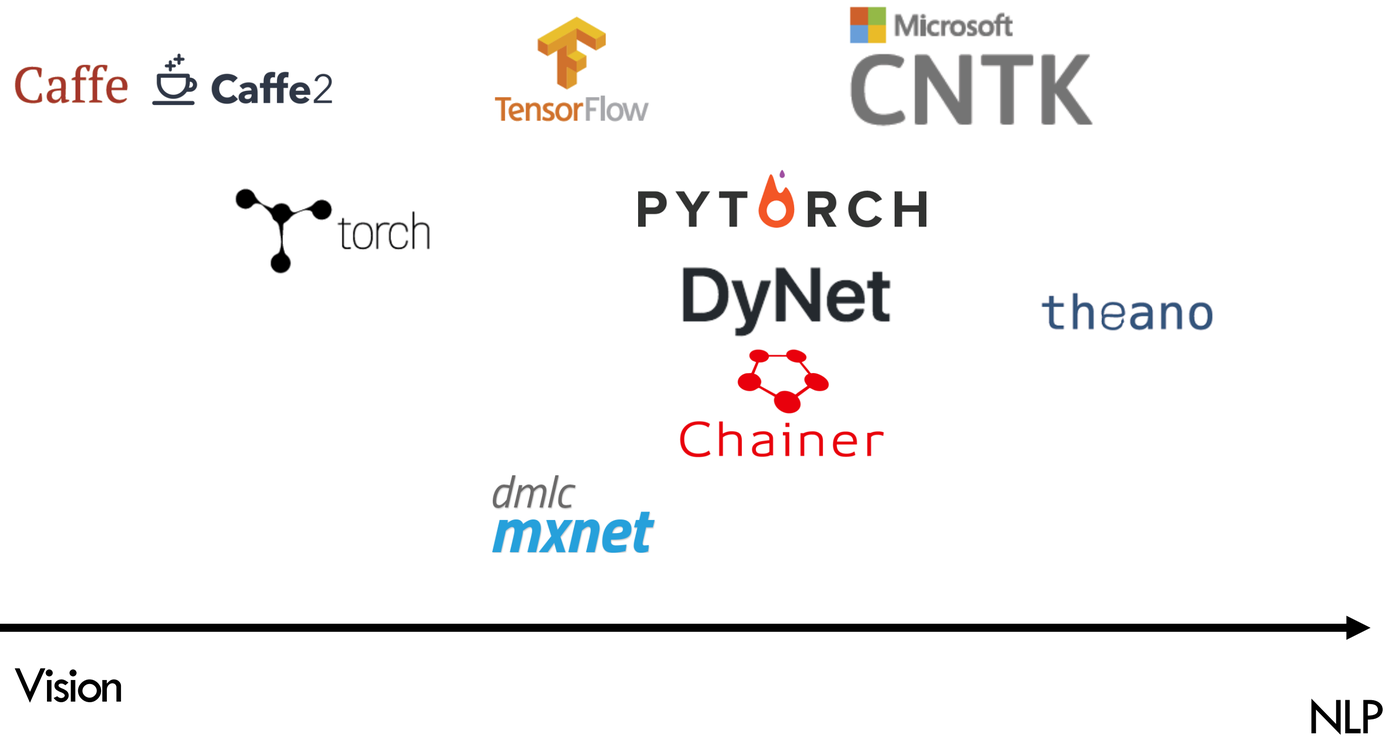
Auto-differential Libraries

- Auto-differential Library automatically derives the gradients following the back-propagation rule.
- A lot of auto-differentiation libraries have been developed:
- So-called Deep Learning toolkits



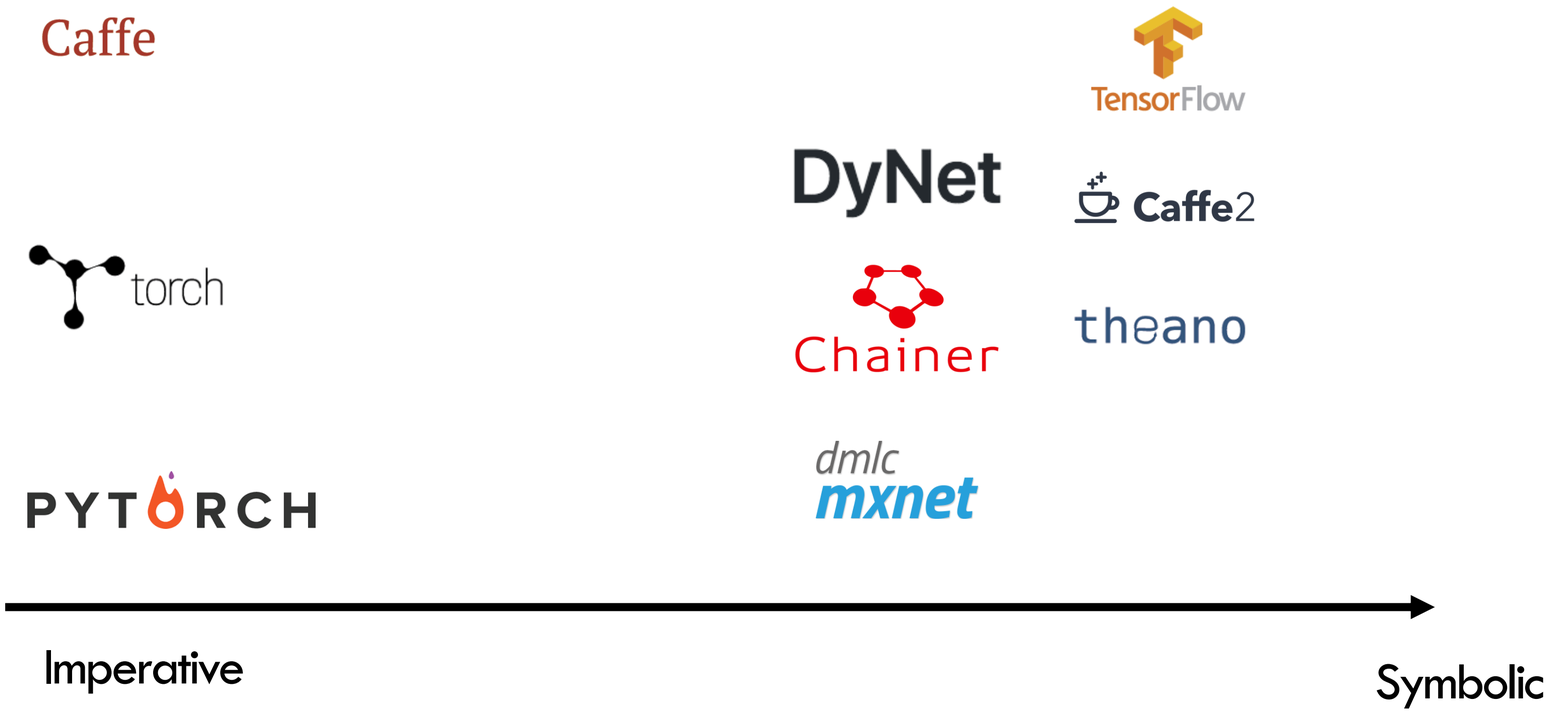
Deep Learning Toolkits

- They are roughly adopted by different domains



Deep Learning Toolkits

- They are also designed differently
  - Symbolic v.s. imperative programming



Deep Learning Toolkits

- Symbolic vs. imperative programming
  - Symbolic: write symbols to assemble the networks first, evaluate later
  - Imperative: immediate evaluation

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```

Symbolic

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

Imperative

Deep Learning Toolkits

- Symbolic
  - Good
    - easy to optimize (e.g. distributed, batching, parallelization) for developers
    - More efficient
  - Bad
    - The way of programming might be counter-intuitive
    - Hard to debug for user programs
    - Less flexible: you need to write symbols before actually doing anything
- Imperative:
  - Good
    - More flexible: write one line, evaluate one line
    - Easy to program and easy to debug: because it matches the way we use C++ or python
  - Bad
    - Less efficient
    - More difficult to optimize

Good and Bad of Dataflow Graphs

- Dataflow graphs seems to be a dominant choice for representing deep learning models
  - What's good for dataflow graphs
    - Good for **static** workflows: define once, run for arbitrary batches/data
    - Programming convenience: **easy to program once you get used to it**.
    - Easy to parallelize/batching for a fixed graph
    - Easy to optimize: a lot of off-the-shelf optimization techniques for graph
  - What's bad for dataflow graphs
    - Not good for **dynamic** workflows: need to define a graph for every training sample -> overheads
    - Hard to program dynamic neural networks: how can you define dynamic graphs using a language for static graphs? (e.g. LSTM, tree-LSTM).
    - Not easy for debugging.
    - Difficult to parallelize/batching across multiple graphs: every graph is different, no natural batching.

Static vs. Dynamic Dataflow Graphs

- Static Dataflow graphs
  - Define once, execute many times
    - For example: convolutional neural networks
  - Execution: Once defined, all following computation will follow the defined computation
  - Advantages
    - No extra effort for batching optimization, because it can be by nature batched
    - It is always easy to handle a static computational dataflow graphs in all aspects, because of its fixed structure
      - Node placement, distributed runtime, memory management, etc.
    - Benefit the developers

Static vs. Dynamic Dataflow Graphs

- Dynamic Dataflow graphs
  - When do we need?
    - In all cases that static dataflow graphs do not work well
  - Variably sized inputs
  - Variably structured inputs
  - Nontrivial inference algorithms
  - Variably structured outputs
  - Etc.

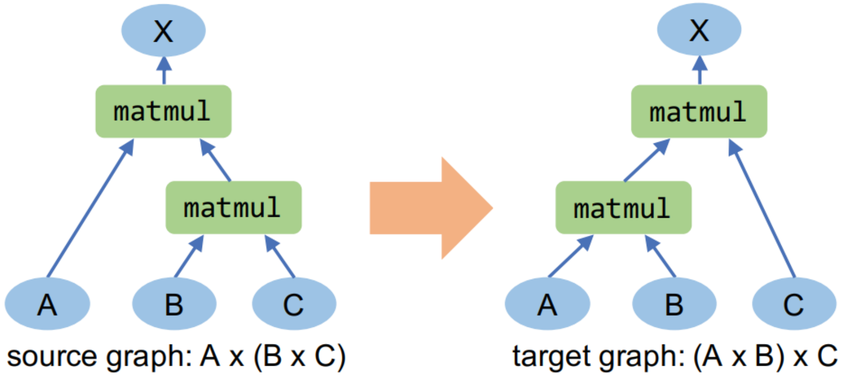
Static vs. Dynamic Dataflow Graphs

- Can we handle dynamic dataflow graphs? Using static methods (or declaration) will have a lot of problems
  - Difficulty in expressing complex flow-control logic
  - Complexity of the computation graph implementation
  - Difficulty in debugging

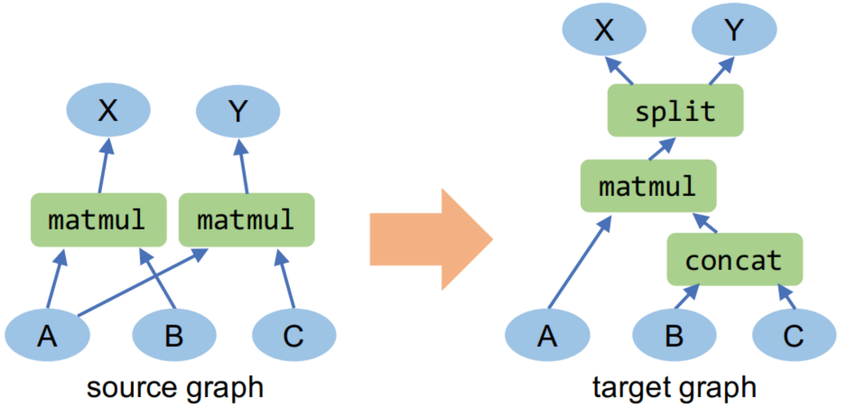
Questions

- Is CNN training static or dynamic graph?
- Is CNN inference static or dynamic graph?
- Is GPT-3 training static graph or dynamic?
- Is GPT-3 inference with batch size = 1 static or dynamic graph
- Is GPT-3 serving static or dynamic graph

DL Dataflow Graph Optimization (advanced)



(a) Associativity of matrix multiplication.



(b) Fusing two matrix multiplications using concatenation and split.

DL Graph Compilation (advanced)