https://hao-ai-lab.github.io/dsc204a-f25/

# DSC 204A: Scalable Data Systems
# Fall 2025


Staff

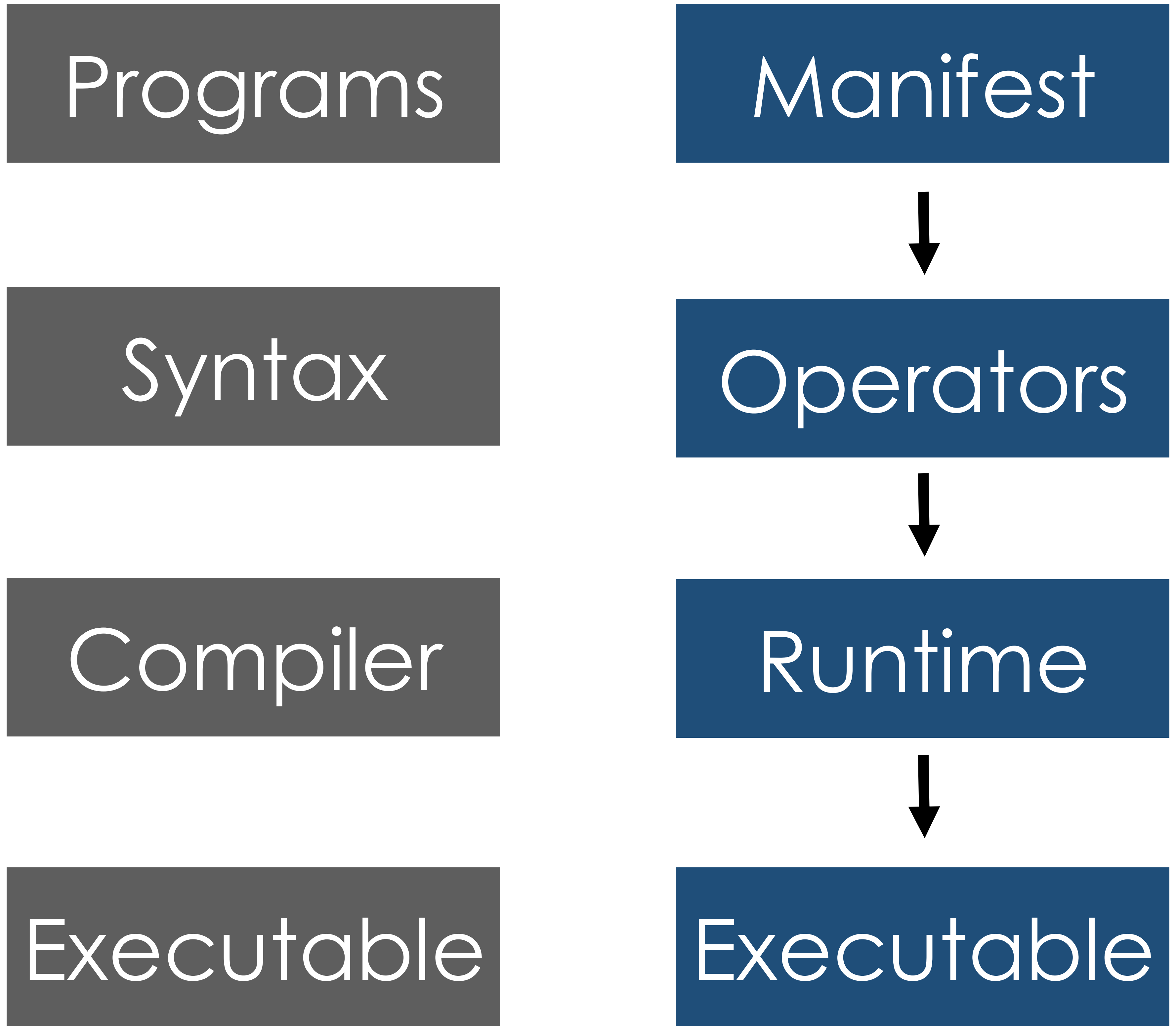
Instructor: Hao Zhang

TAs: Mingjia Huo, Yuxuan Zhang

@haozhangml @haoailab

haozhang@ucsd.edu

# After Spark:
# All Modern Data/ML Systems follow a similar architecture




A fixed set of operators

A trusted runtime with a small set of pre-loaded implementations

# After Spark: Many new systems

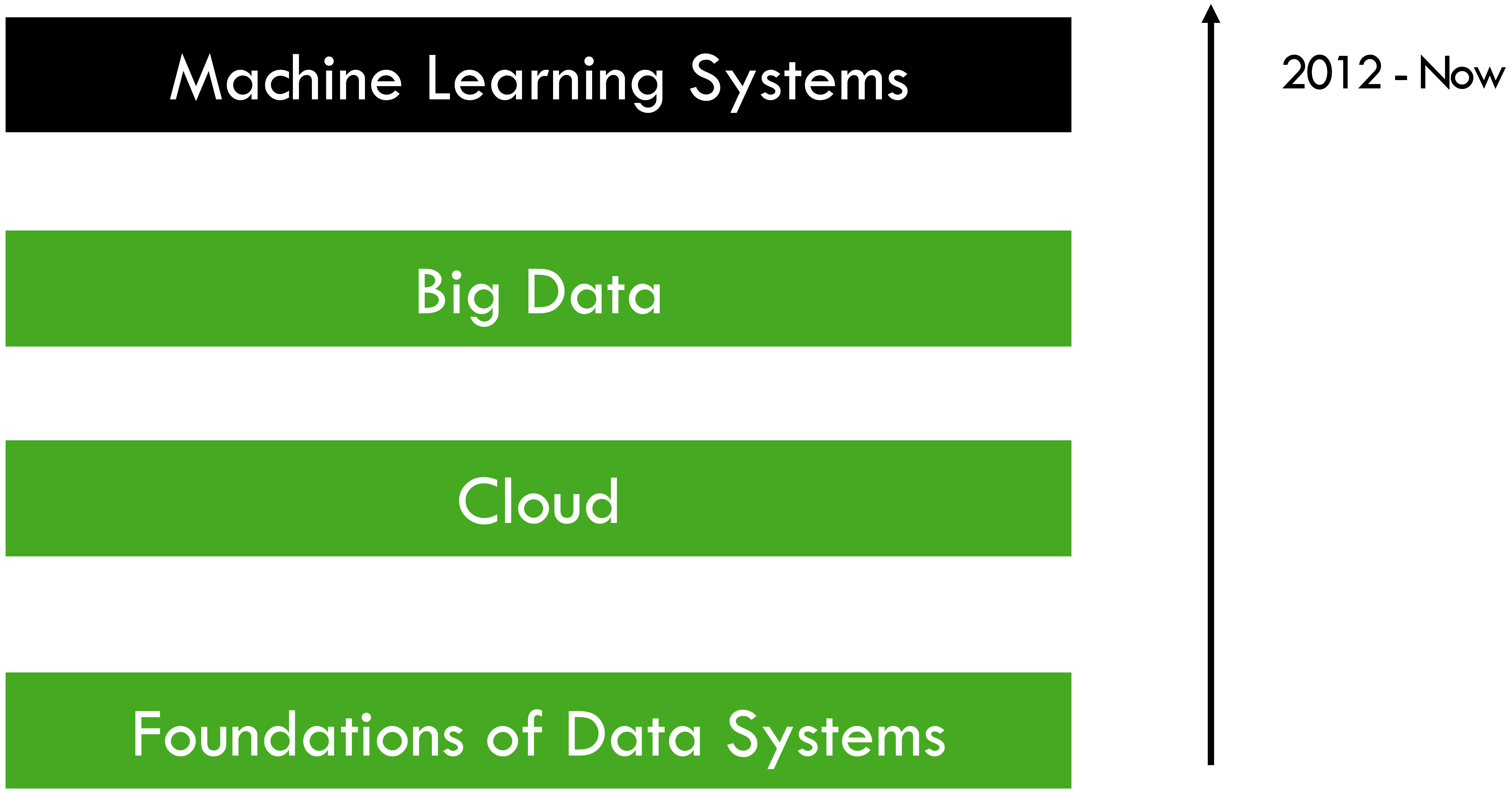
APACHE Spark™

GraphLab

Naiad

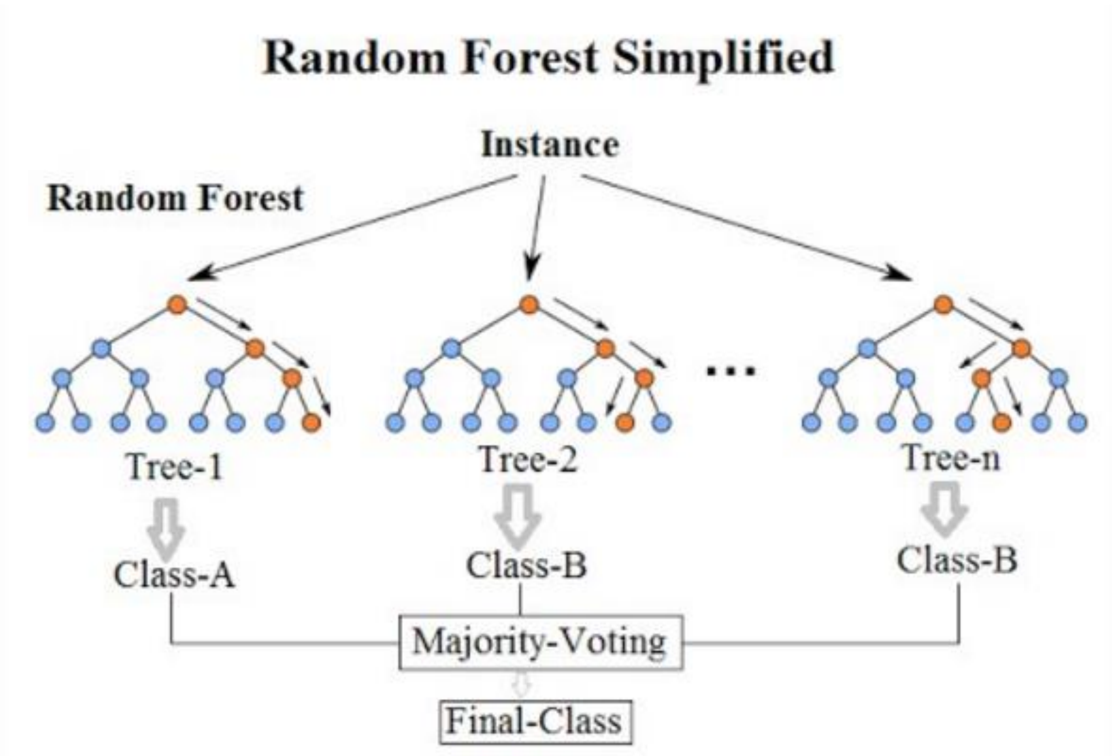
TensorFlow

# Where We Are


Machine Learning Systems
Big Data
Cloud
Foundations of Data Systems
2012 - Now

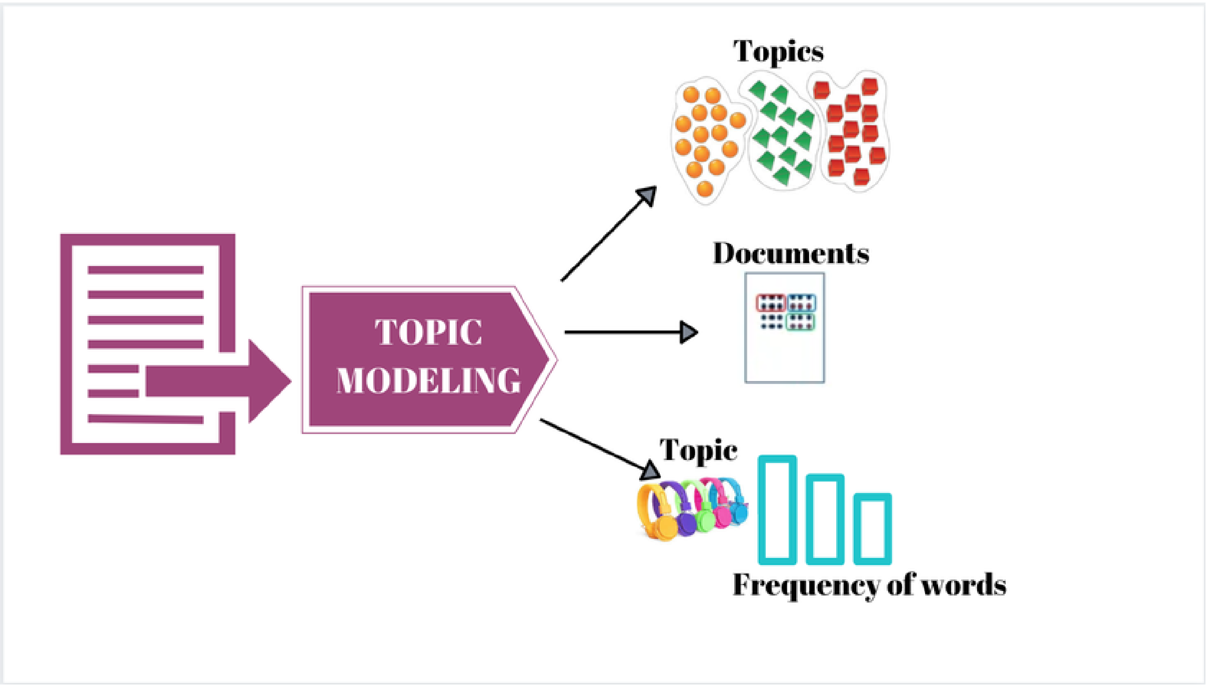
# ML Era (roughly starts from 2008, even before Spark has taken off)

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

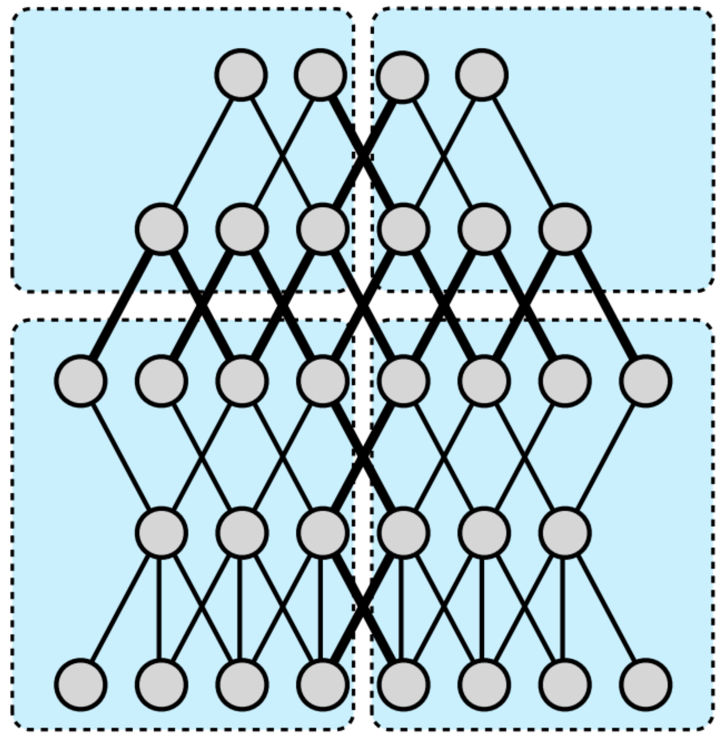



XGBOOST


Spark mllib




LDA



Torch (lua) / Theano / distbelief

# Diversity -> Good or Bad?

- 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
  - Pros:
    - A lot of opportunities: Gold mining era

# ML Systems Plan in DSC 204A

- ML System history
  - Parameter server for data parallelism
  - Deep Learning (Autodiff) libraries: tensorflow, pytorch, etc.
  - LLMs: Model Parallelism, training and inference

# ML System history

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

Ad-hoc: diverse model family, optimization algos, and data

Opt algo: iterative-convergent

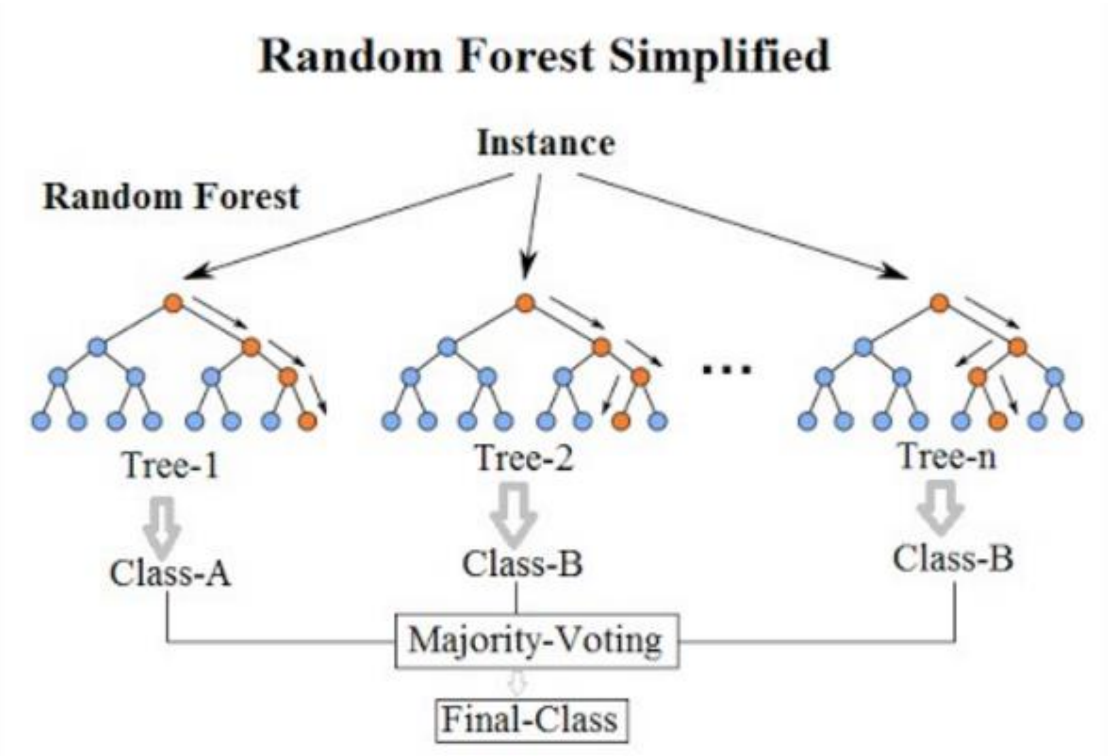
Model family: neural nets

Model: CNNs/transformers/GNNs

LLMs: transformer decoders

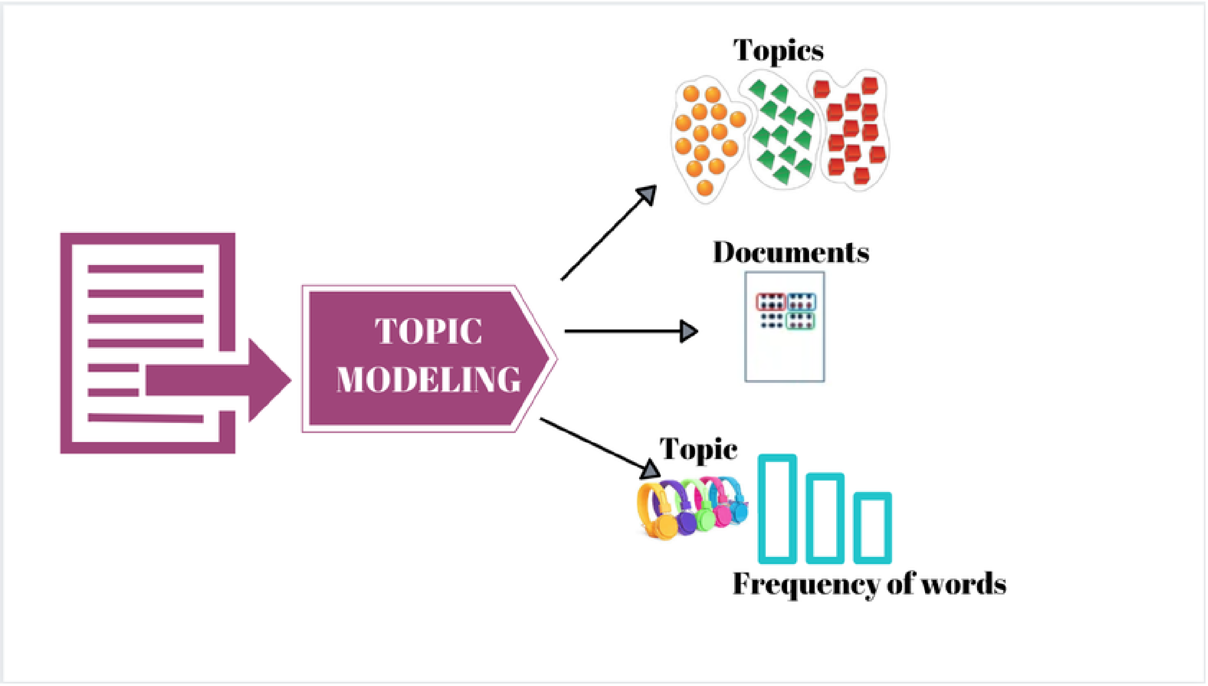
More and more **unified** yet scope becoming narrower and narrower

# The first Unified component: Iterative-convergence Algo

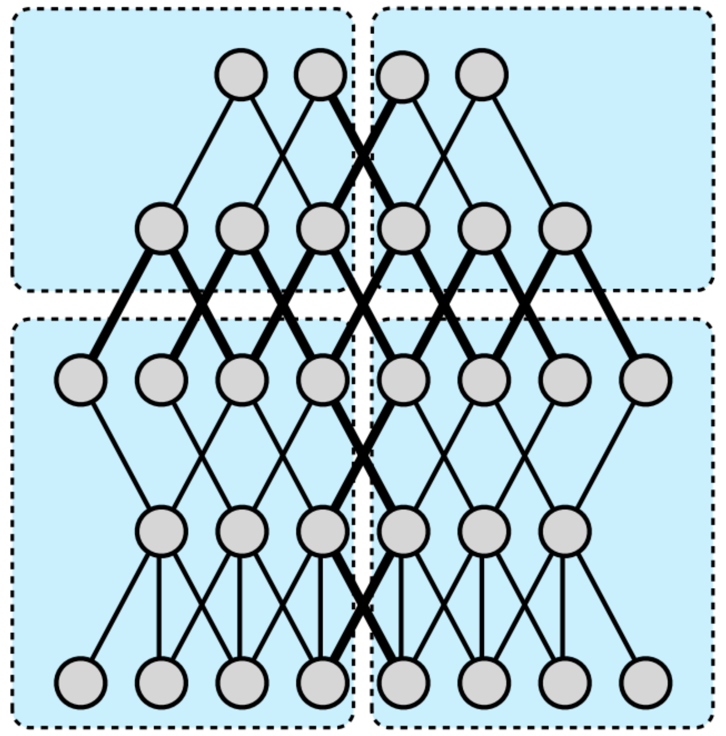



Gradient boosting tree


Coordinate descent




EM Algorithm



Gradient descent

# Example: Gradient Descent

Gradient / backward computation

Recall collective communication

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

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) |
| $sort(c : \mathrm{Comparator[K]})$ | : | $\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$ |
| $partitionBy(p : \mathrm{Partitioner[K]})$ | : | $\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$ |

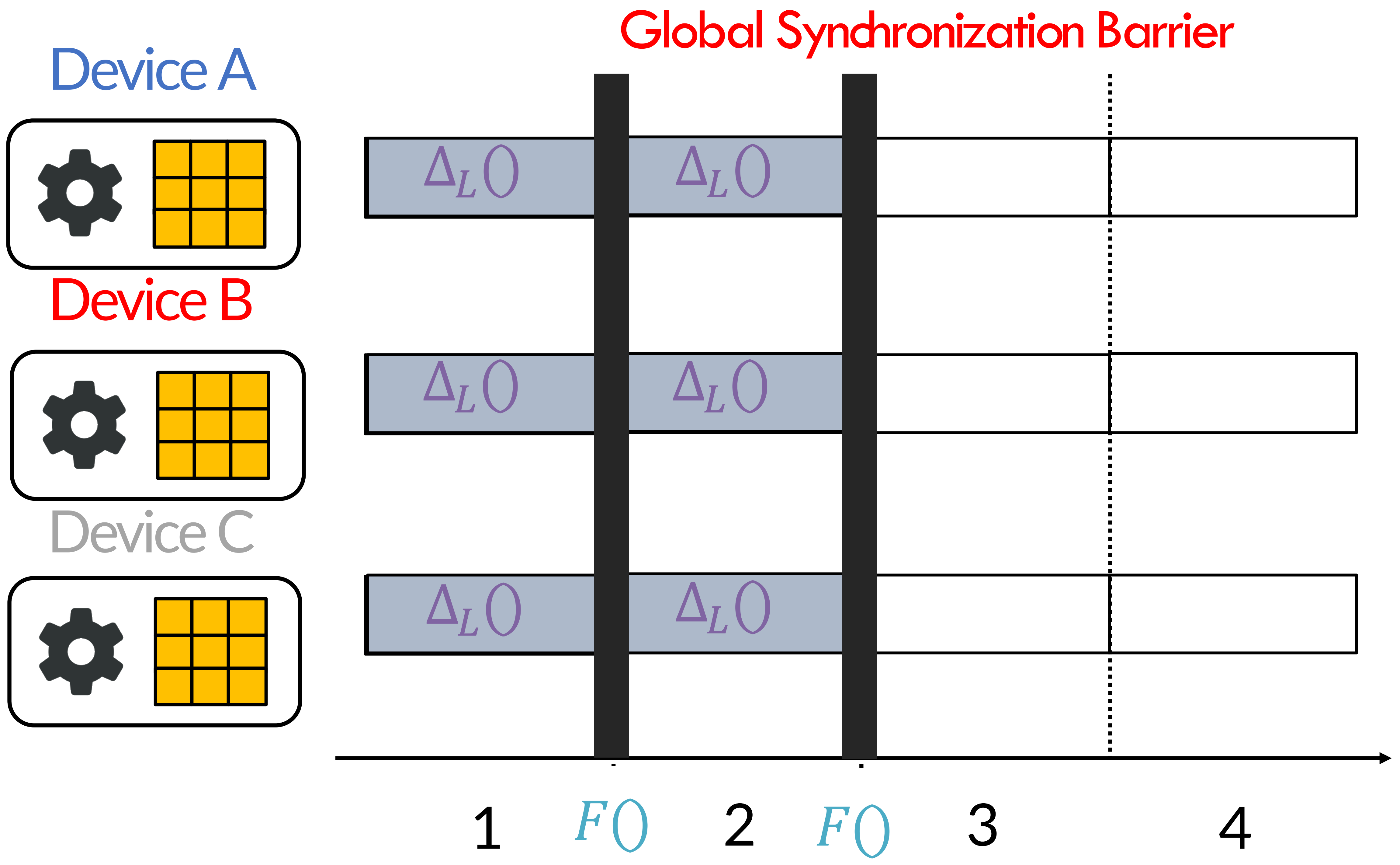
# Problems if expressing this in Spark

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

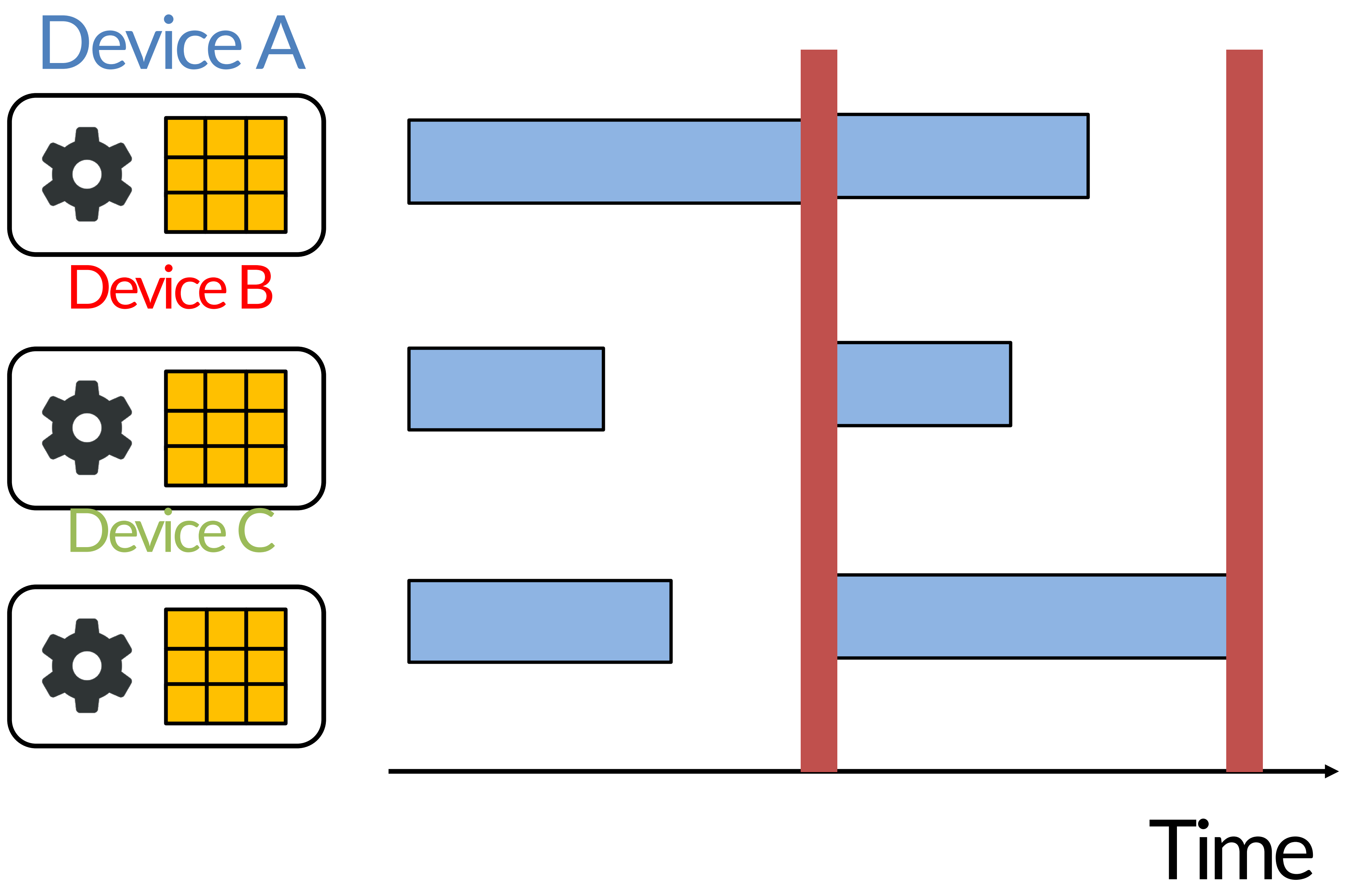
- Very heavy communication per iteration
- Compute time : communication time = 1:10 in the era of 2012

# Consistency

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

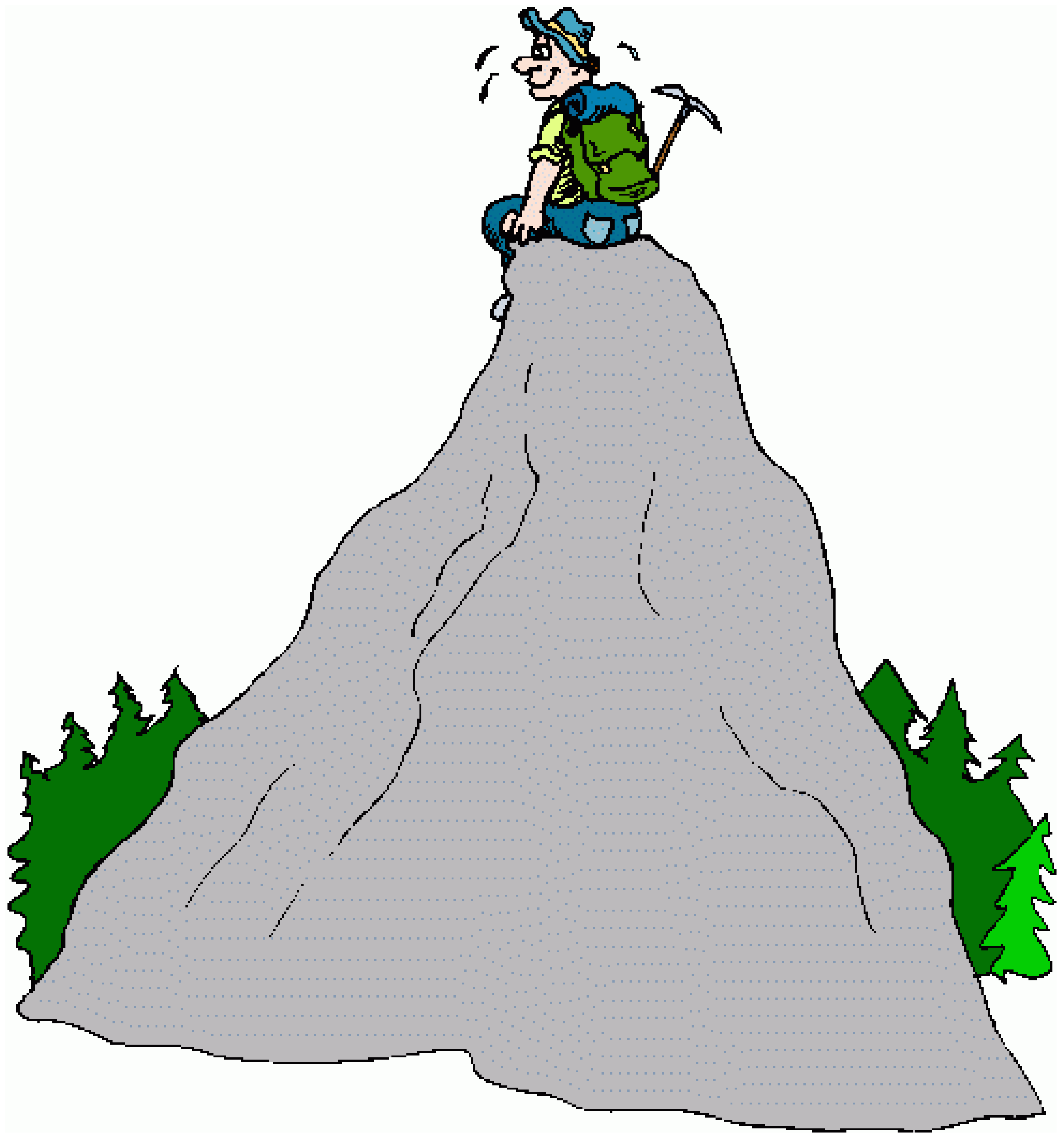


# BSP's Weakness: Stragglers

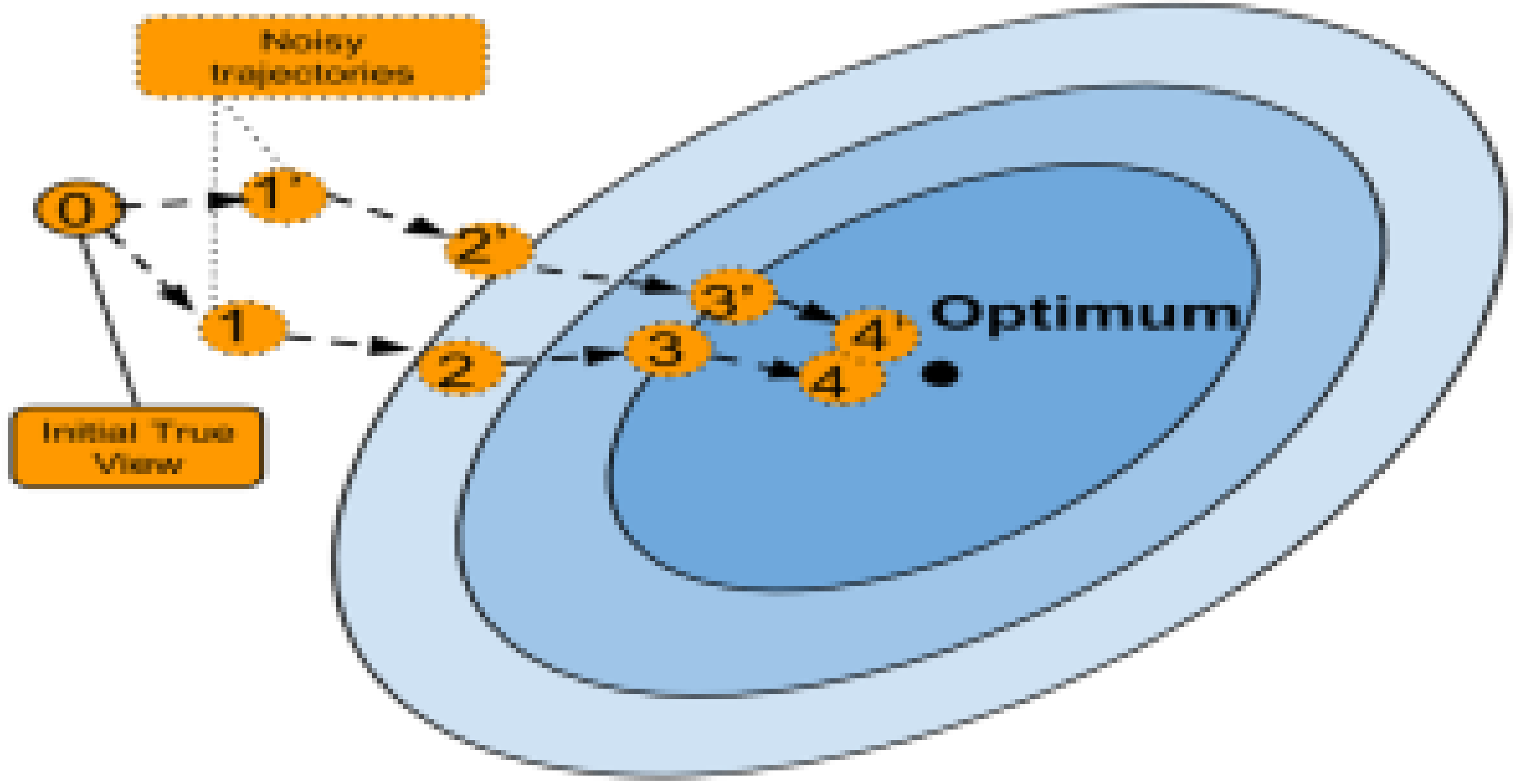


# An interesting property of Gradient Descent (ascent)

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

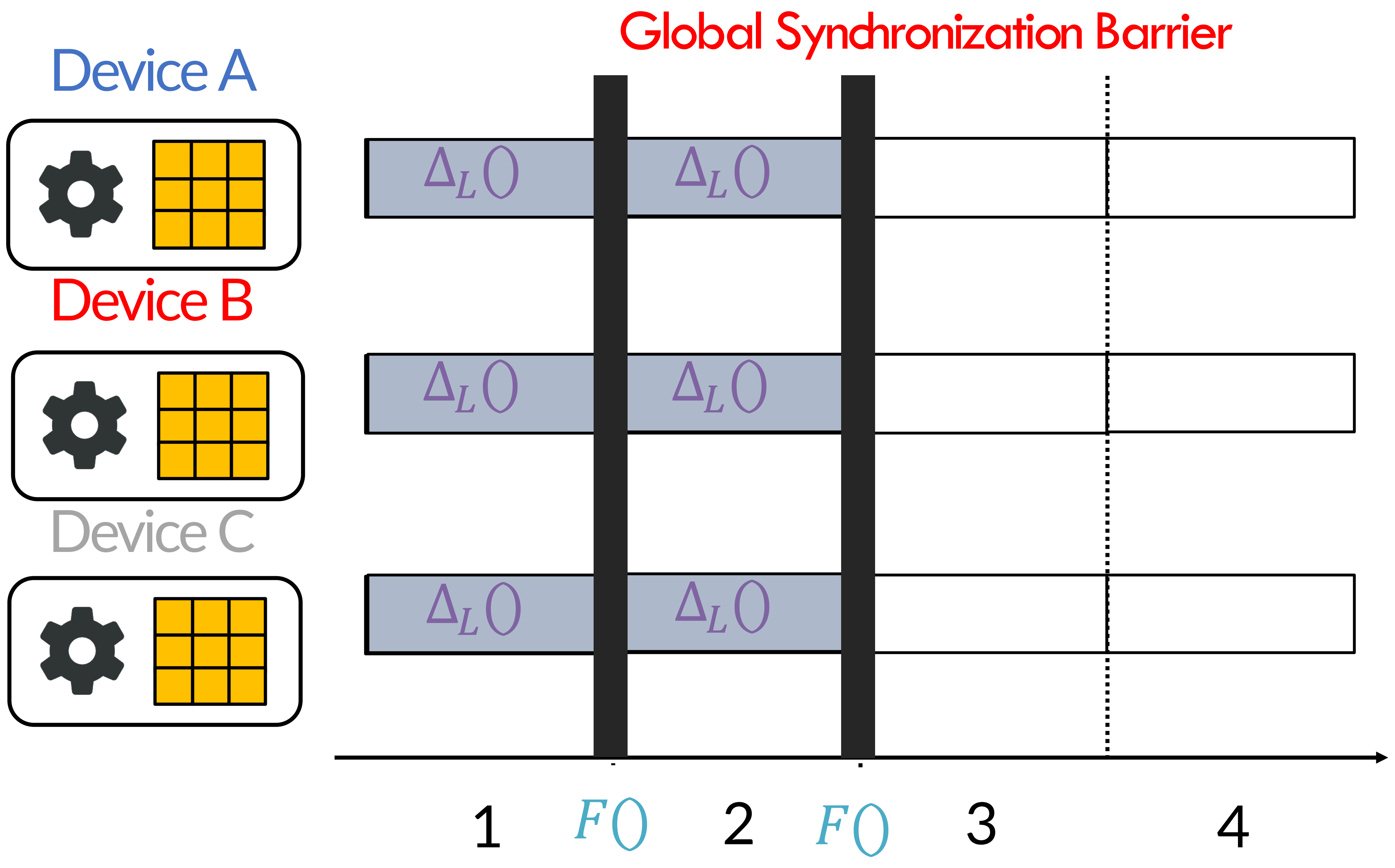


# Machine Learning is Error-tolerant (under certain conditions)

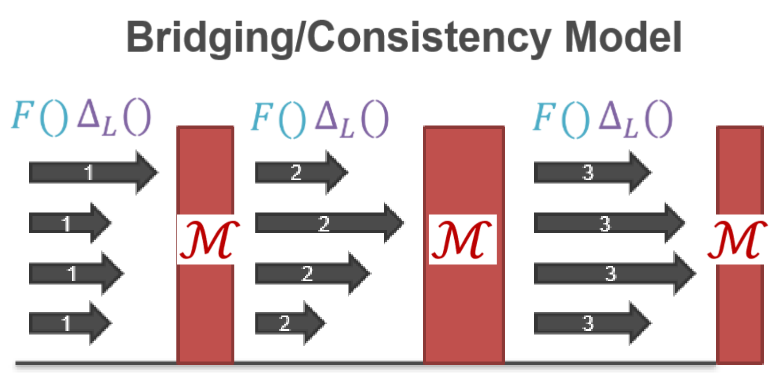


# Background: Strict Consistency

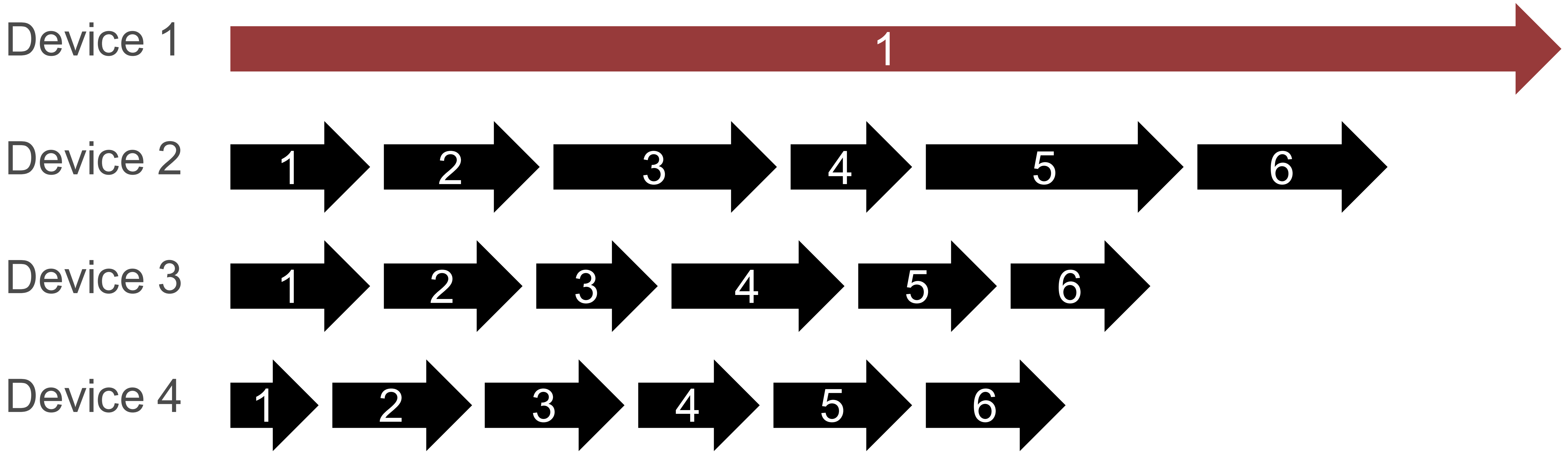
- **Baseline:** Bulk Synchronous Parallel (BSP)
  - MapReduce, Spark, many DistML Systems
- Devices compute updates $\Delta_L()$ between global barriers (iteration boundaries)
- **Advantage: Execution is serializable**
  - Same guarantees as sequential algo!



# Background: Asynchronous Communication (No Consistency)




- **Asynchronous (Async):** removes all communication barriers

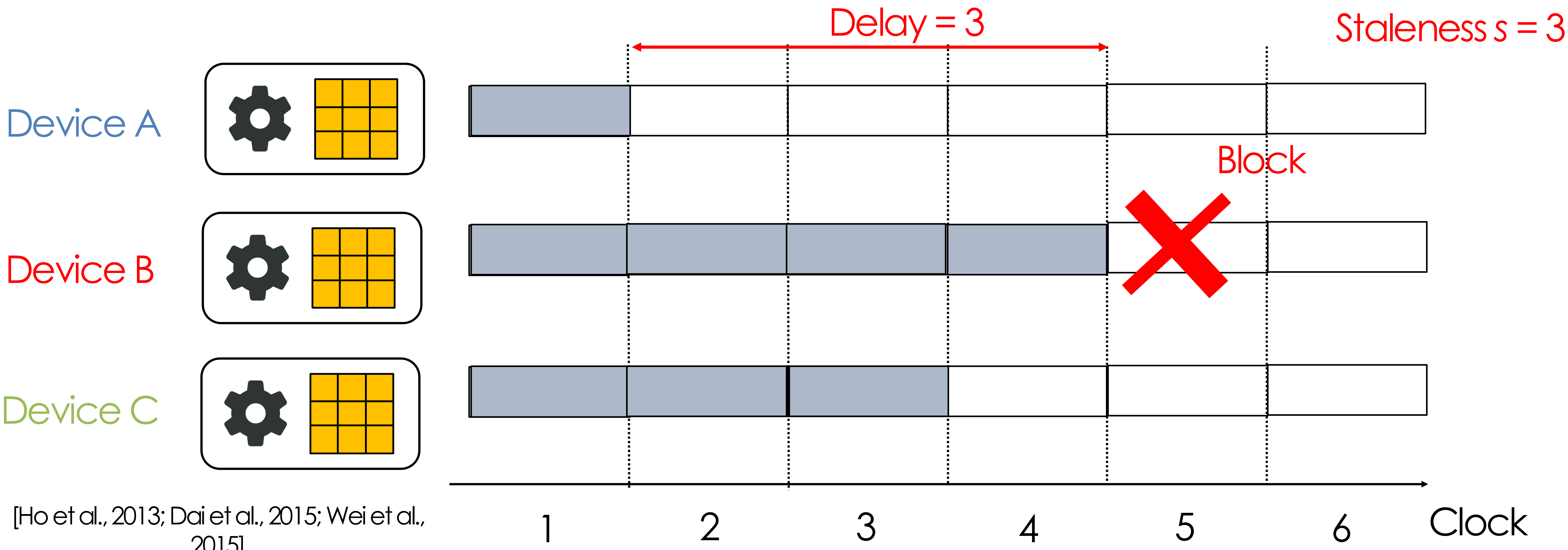


# Background: Bounded Consistency

**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

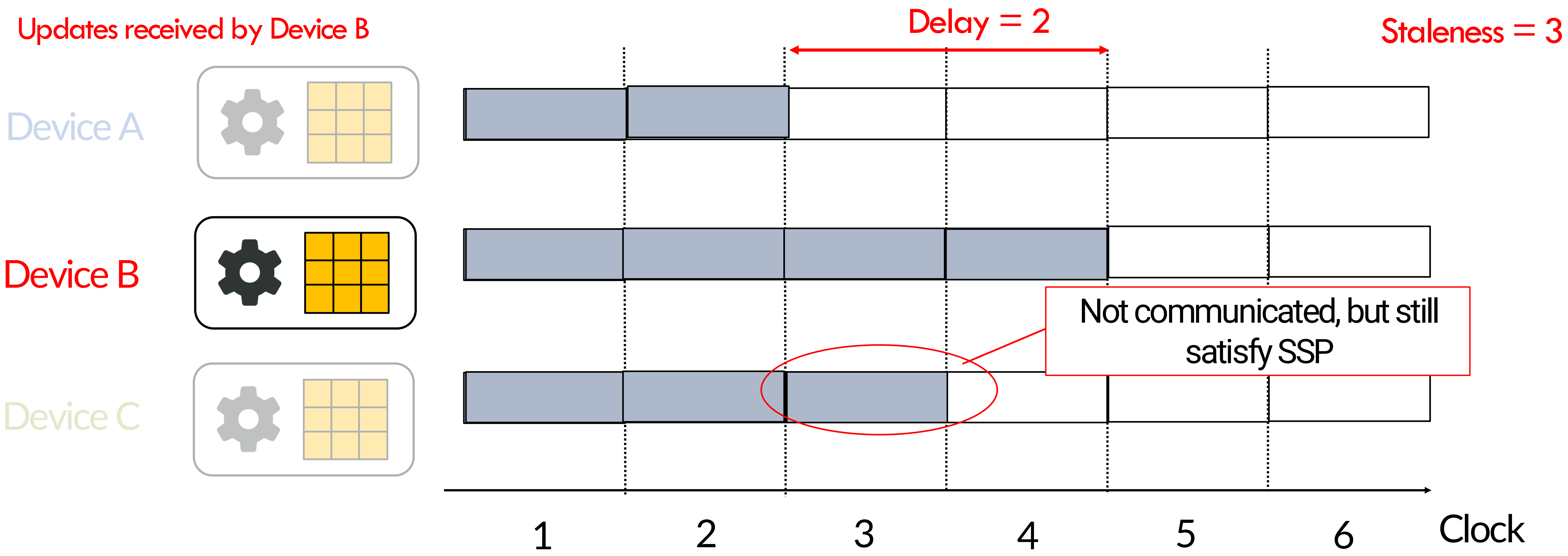
- Fastest & slowest device must not drift > $s$ iterations apart (in this example, $s = 3$)
  - $s$ is the maximum staleness



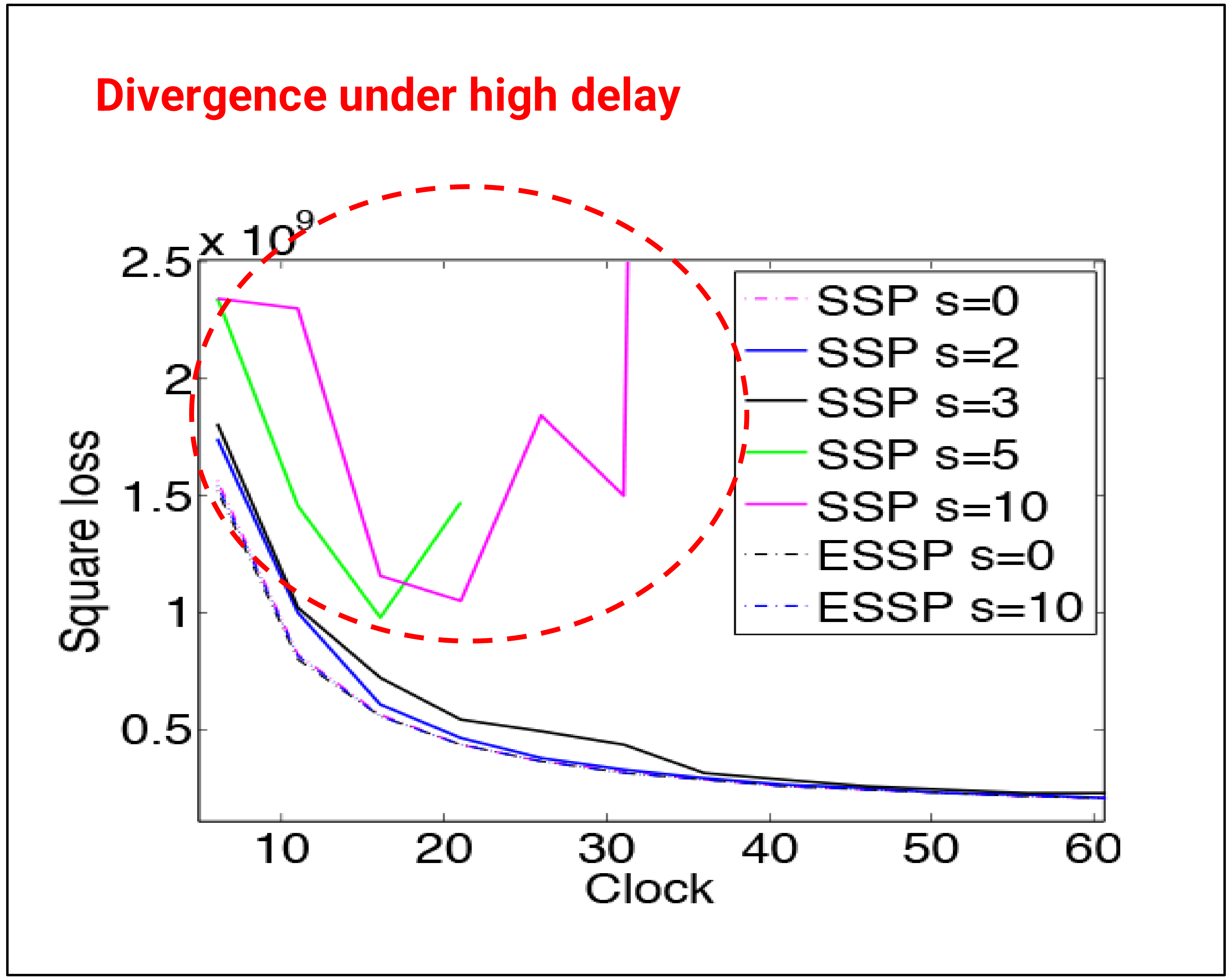

[Ho et al., 2013; Dai et al., 2015; Wei et al., 2015]

# SSP: "Lazy" Communication

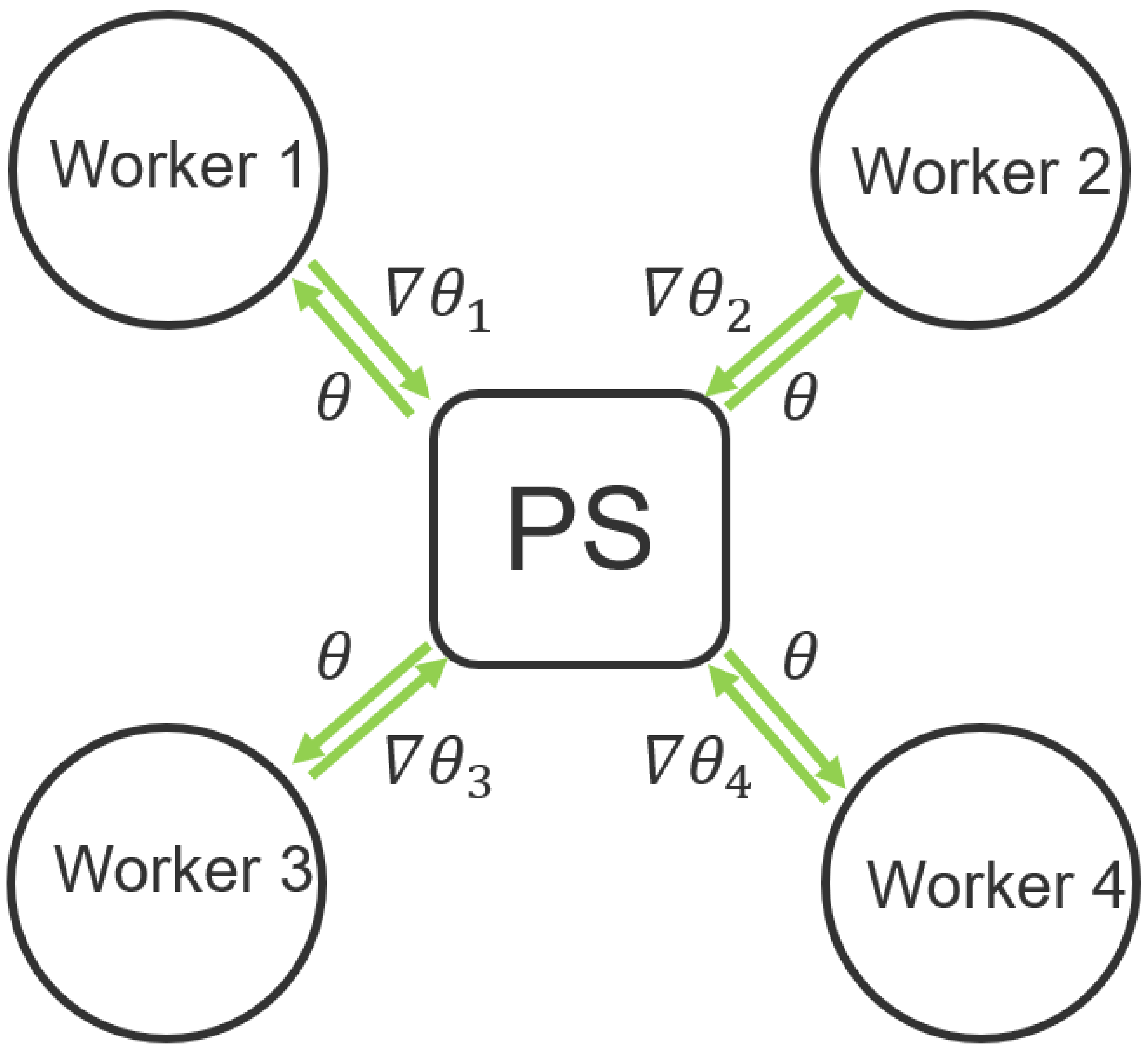
**SSP:** devices avoid communicating unless necessary
- i.e. when staleness condition is about to be violated
- Favors throughput at the expense of statistical efficiency



# Impacts of Consistency/Staleness: Unbounded Staleness

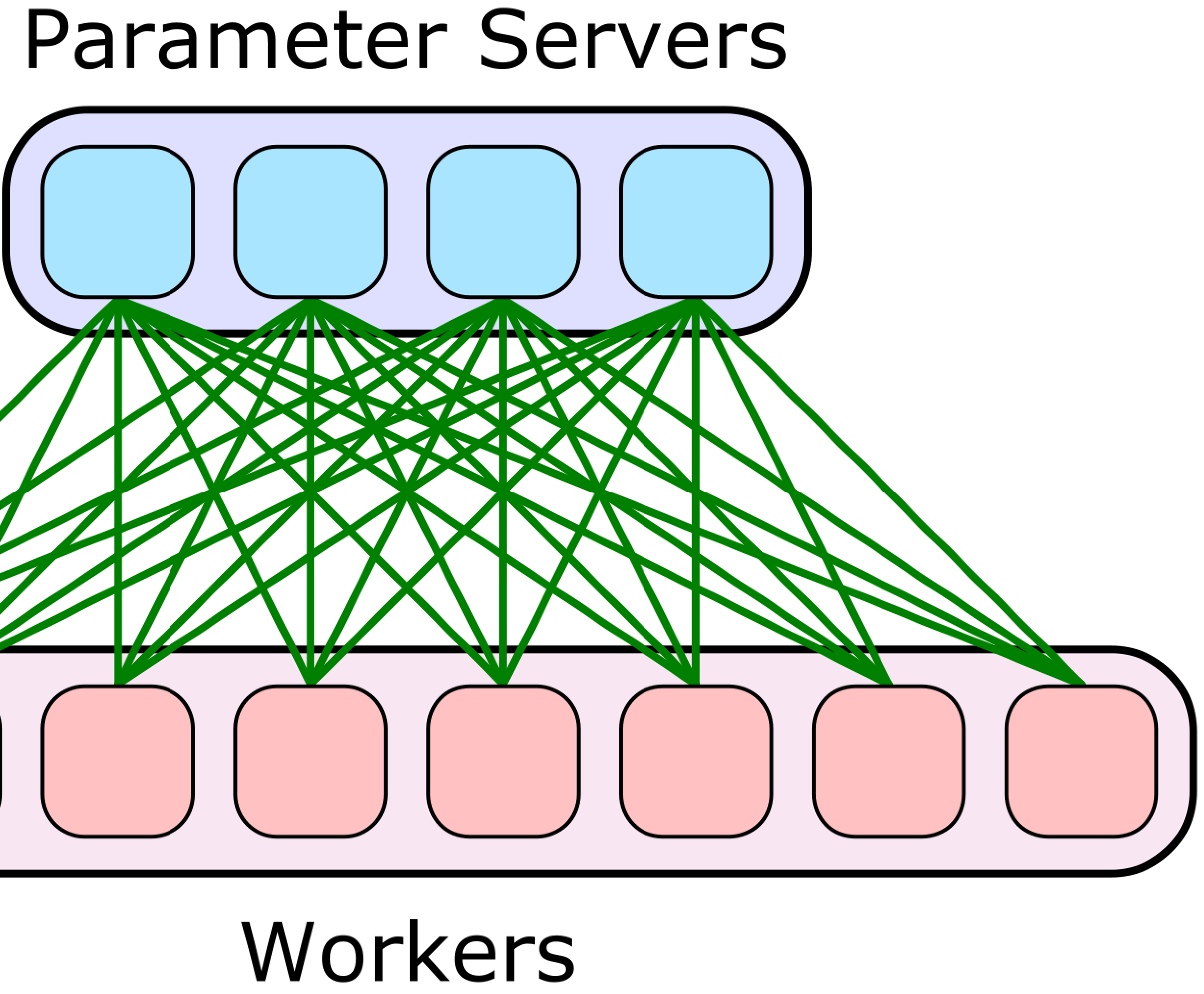


# Parameter Server Naturally emerges



# Parameter Server Implementation

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



# 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

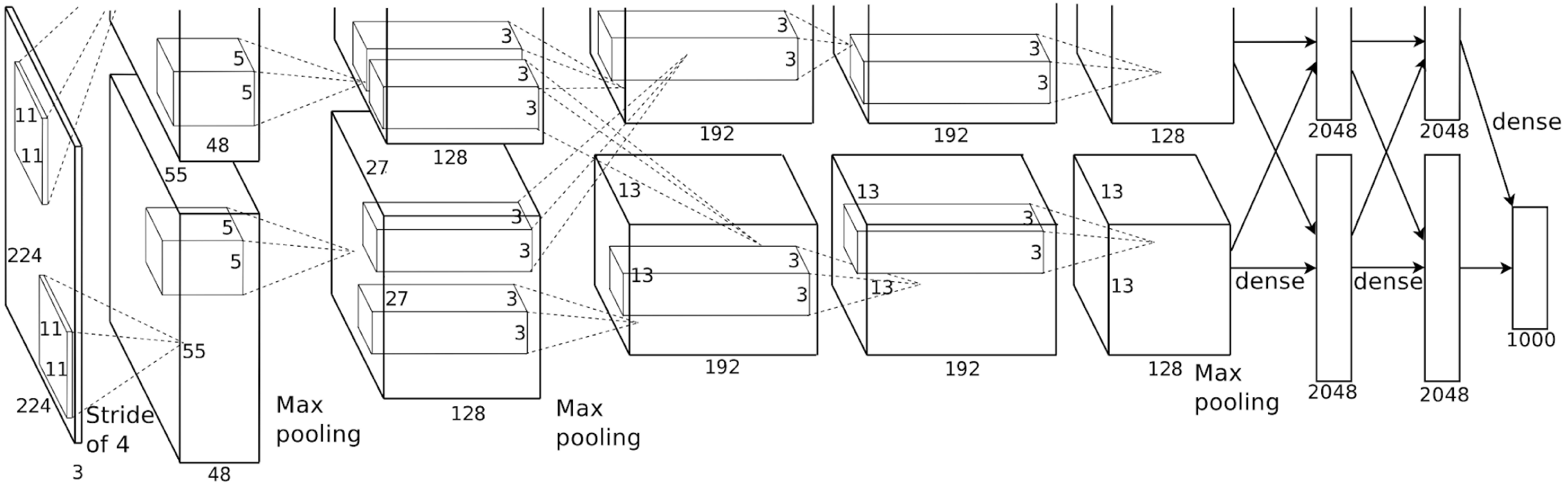



Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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, spark-defined data processing 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

# Deep Learning Libraries

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

# Recall our Goal

- Goal: we want to express as many as deep neural networks as possible using one set of programming interface by connecting math primitives
- What constitutes a model from math primitives?
  - Model and architecture: connecting math primitives
  - Objective function
  - Optimizer
  - Data

# Discussion: how we express computation in history
# Applications <-> System Design

| | | |
|---|---|---|
| Application | Data management<br>(OLTP) | Big data processing<br>(OLAP) |
| Systems | SQL<br>Query planner<br>Relational database<br>Storage | Spark/mapreduce<br>Dataflow, lineage<br>Data warehousing<br>Column storage |

# High-level Picture

Data

Model

Compute

❓ $\{x_i\}^n{}_{i=1}$

❓ Math primitives (mostly matmul)

❓ Make them run on (clusters of ) different kinds of hardware

❓ A repr that expresses the computation using primitives

# High-level Picture

Data

Model

Compute

✅ $\{x_i\}^n{}_{i=1}$

❓ Math primitives (mostly matmul)

❓Make them run on (clusters of ) different kinds of hardware

❓A repr that expresses the computation using primitives

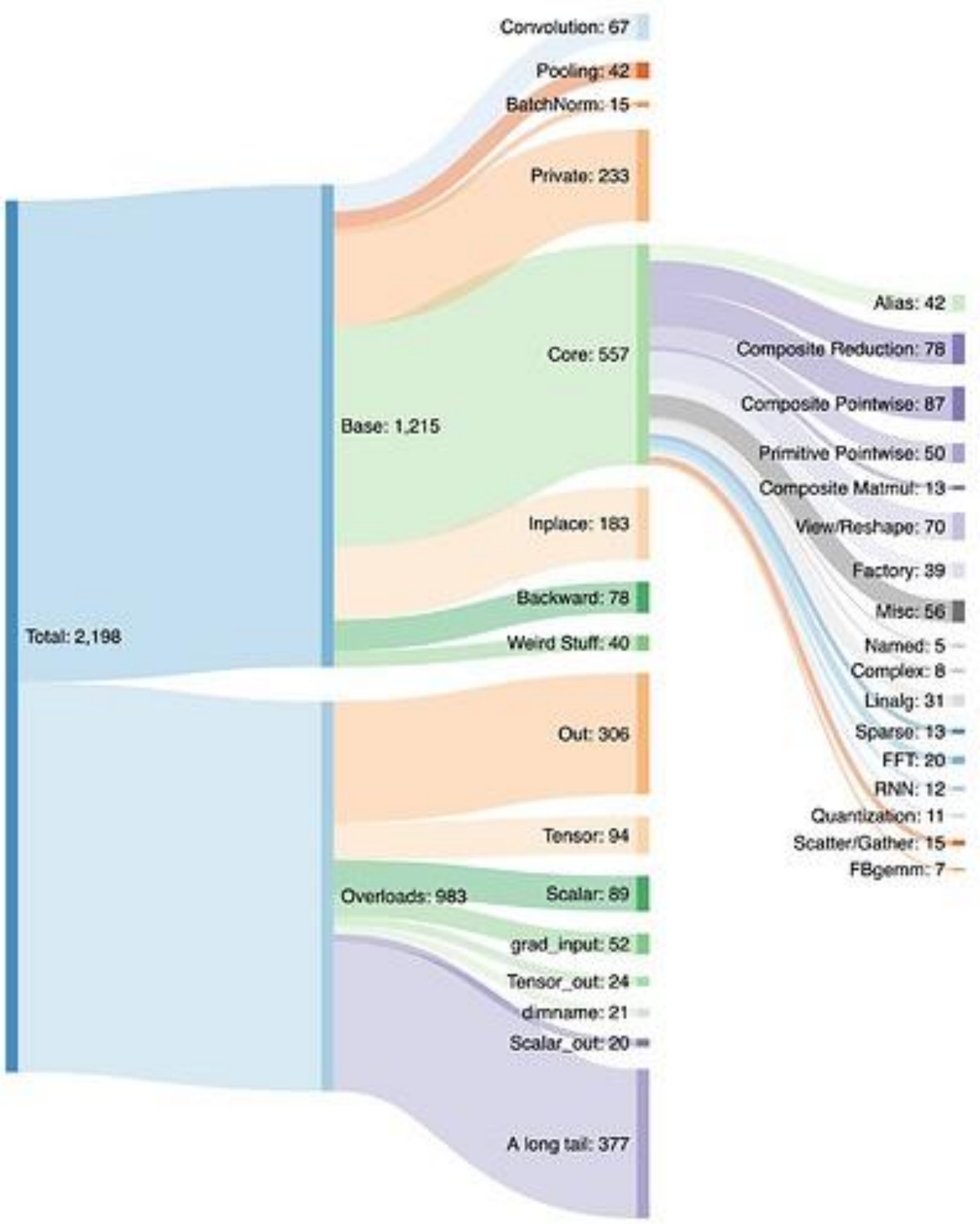
# Maybe?

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

- ML is mostly tensor operations and more diverse; hard to express their computation in coarse-grained data transformations.

# Operators

| API | Name inference rule |
|---|---|
| `Tensor.abs()`, `torch.abs()` | Keeps input names |
| `Tensor.abs_()` | Keeps input names |
| `Tensor.acos()`, `torch.acos()` | Keeps input names |
| `Tensor.acos_()` | Keeps input names |
| `Tensor.add()`, `torch.add()` | Unifies names from inputs |
| `Tensor.add_()` | Unifies names from inputs |
| `Tensor.addmm()`, `torch.addmm()` | Contracts away dims |
| `Tensor.addmm_()` | Contracts away dims |
| `Tensor.addmv()`, `torch.addmv()` | Contracts away dims |
| `Tensor.addmv_()` | Contracts away dims |
| `Tensor.align_as()` | See documentation |
| `Tensor.align_to()` | See documentation |
| `Tensor.all()`, `torch.all()` | None |
| `Tensor.any()`, `torch.any()` | None |
| `Tensor.asin()`, `torch.asin()` | Keeps input names |
| `Tensor.asin_()` | Keeps input names |
| `Tensor.atan()`, `torch.atan()` | Keeps input names |



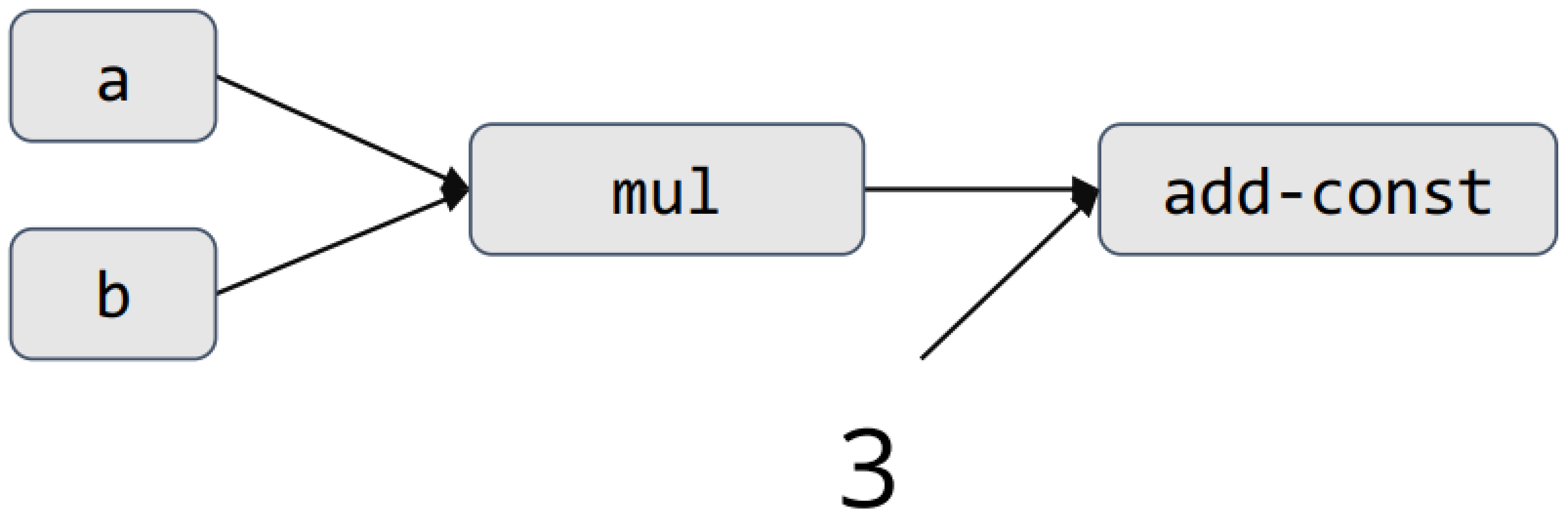
# High-level Picture

| Data | Model | Compute |
| --- | --- | --- |
| ✅ $\{x_i\}^n{}_{i=1}$ | ✅ Math primitives (mostly matmul) | ❓Make them run on (clusters of ) different kinds of hardware |

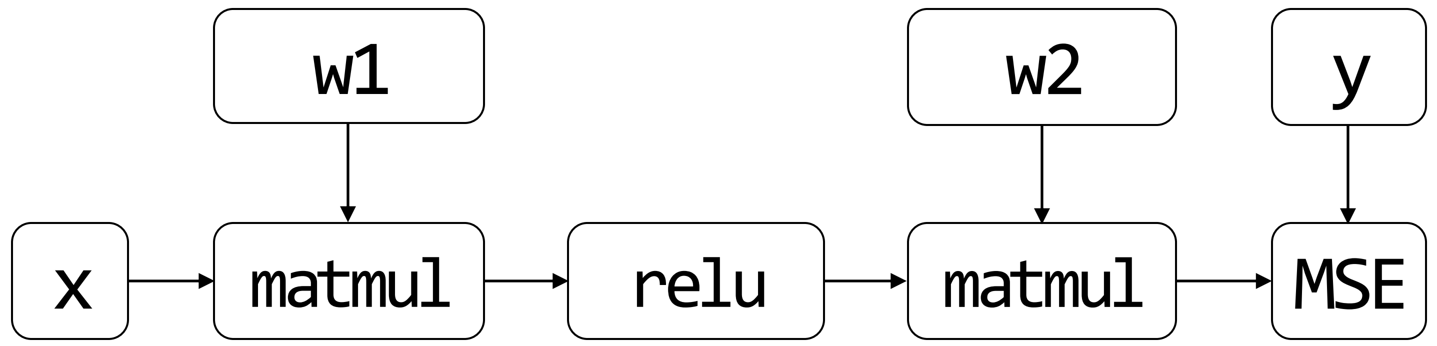
❓A repr that expresses the computation using primitives

# Computational Dataflow Graph

- Node: represents the computation (operator)
- Edge: represents the data dependency (data flowing direction)
- Node: also represents the *output tensor* of the operator
- Node: also represents an input constant tensor (if it is not a compute operator)




a x b + 3




$$L = \mathrm{MSE}(w_2 \cdot \mathrm{ReLU}(w_1 x),\, y)$$

# Case Study: TensorFlow Program

- In the next few slides, we will do a case study of a deep learning program using TensorFlow v1 style API (classic Flavor).
- Note that today most deep learning frameworks now use a different style, but share the same mechanism under the hood
- Think about abstraction and implementation when going through these examples

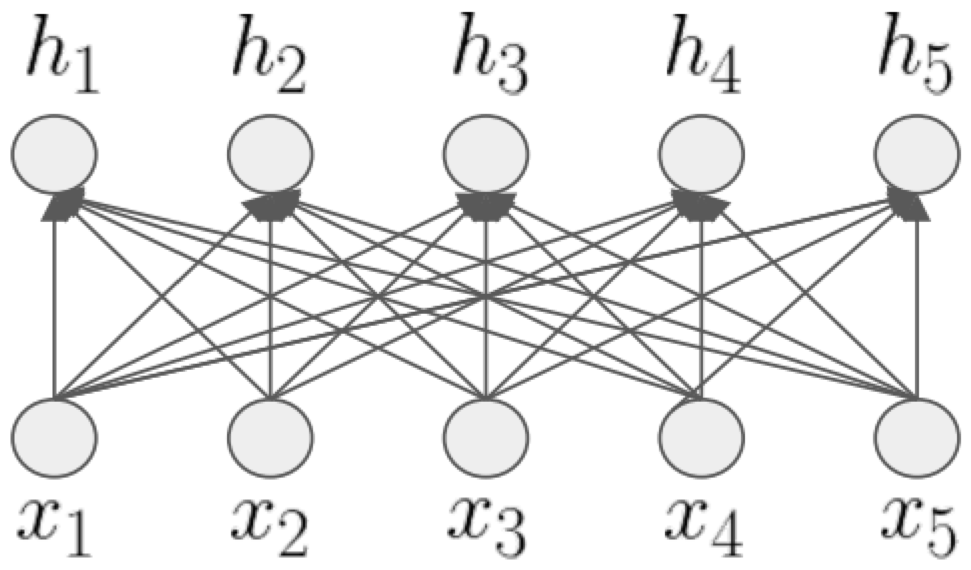
# One linear NN: Logistic Regression

**Input**

$$x_i = \begin{bmatrix} \text{pixel}_1 \\ \text{pixel}_2 \\ \cdots \\ \text{pixel}_m \end{bmatrix}$$

**One Linear Layer**

$$h_k = w_k^T x_i$$




**Softmax**

$$P(y_i = k | x_i) = \frac{\exp(h_k)}{\sum_{j=1}^{10} \exp(h_i)}$$

# Whole Program

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Forward Computation
**Declaration**

# Loss Function

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Loss function **Declaration**

$$P(\text{label} = k) = y_k$$

$$L(y) = \sum_{k=1}^{10} I(\text{label} = k) \log(y_i)$$

# Auto-diff

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Automatic Differentiation:
Next incoming topic

# SGD Update

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

SGD update rule

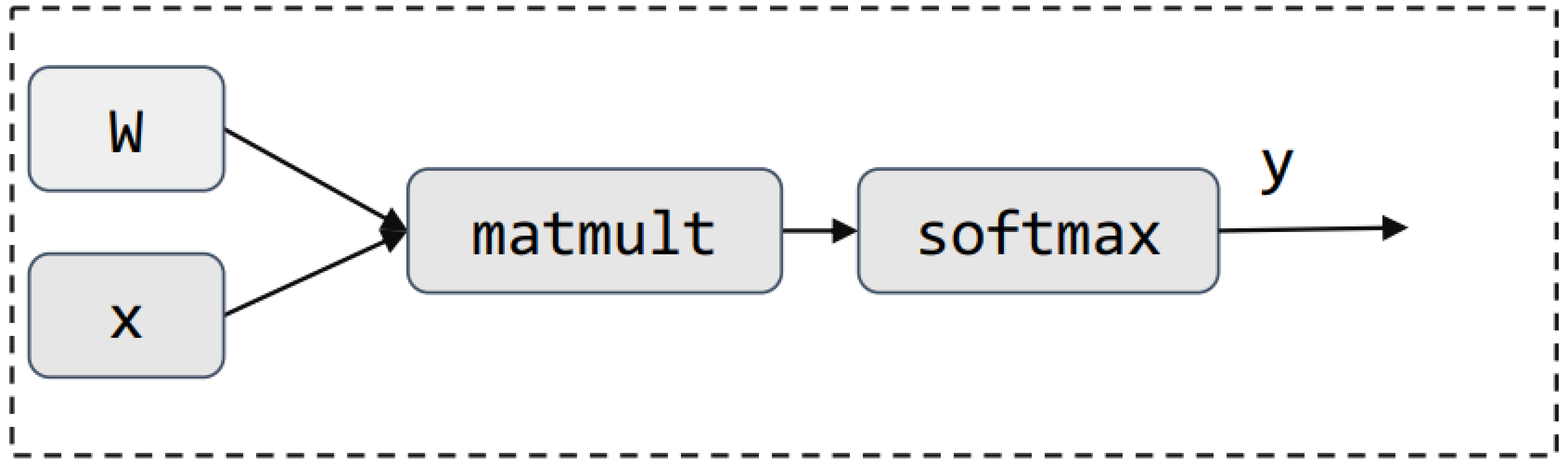
# Trigger the Execution

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Real execution happens here!

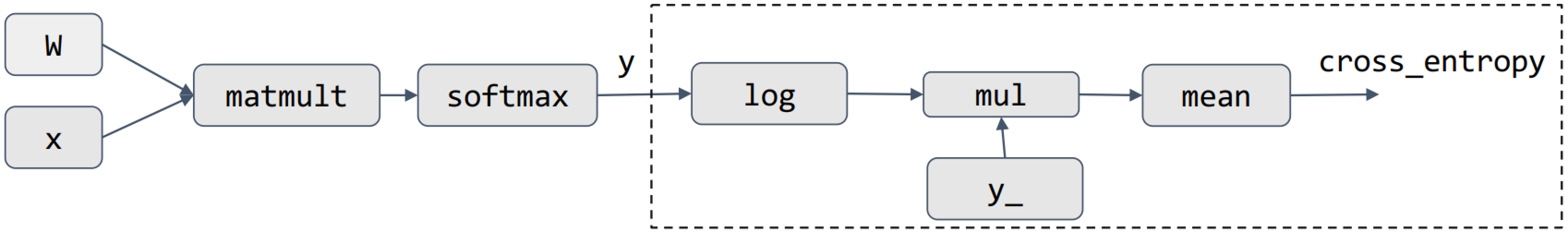
# What happens behind the Scene

```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
```



# What happens behind the Scene (Cond.)

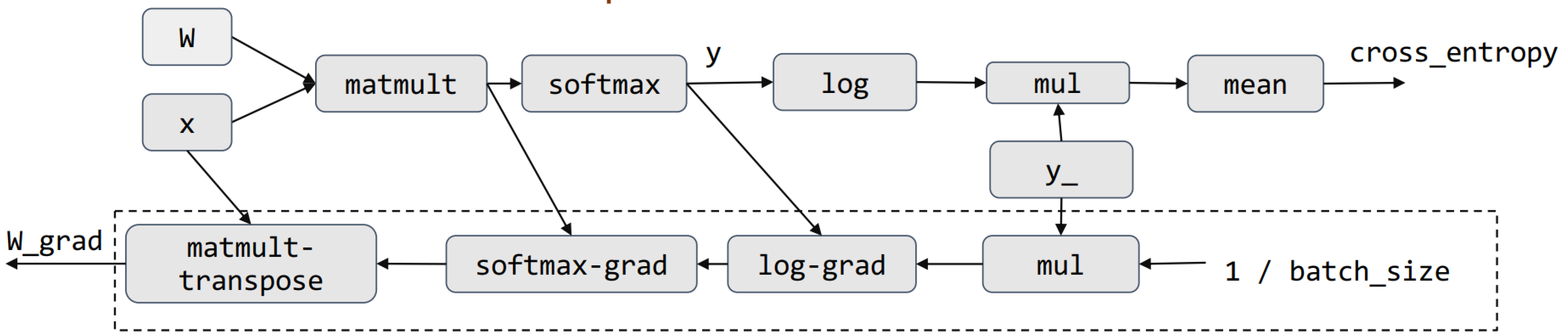
```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```



# What happens behind the Scene (Cond.)

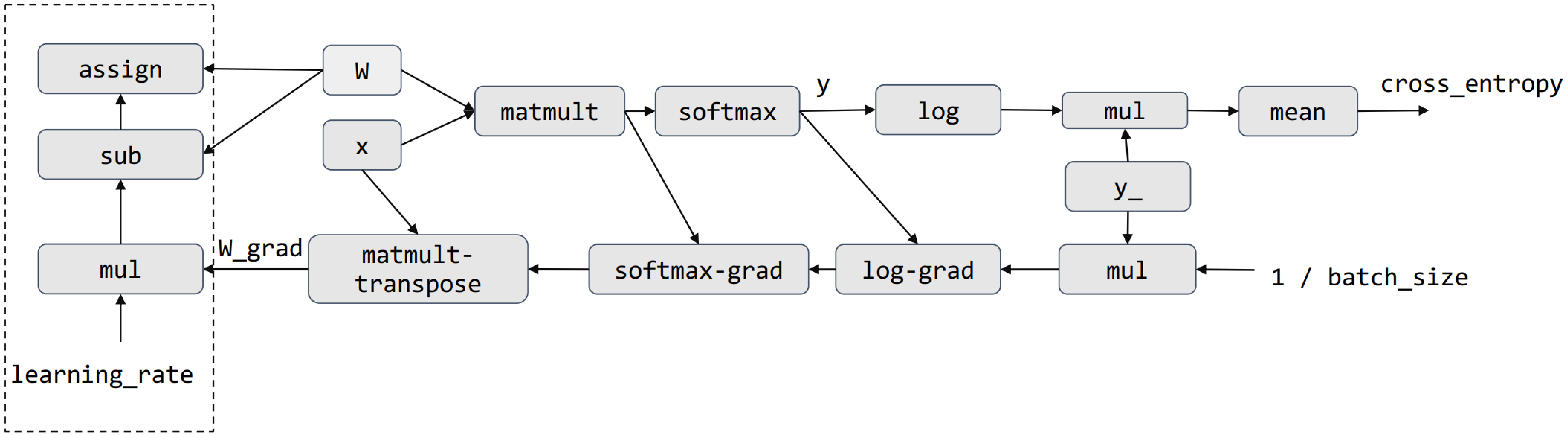
```
W_grad = tf.gradients(cross_entropy, [W])[0]
```

Automatic Differentiation, more details in follow up lectures



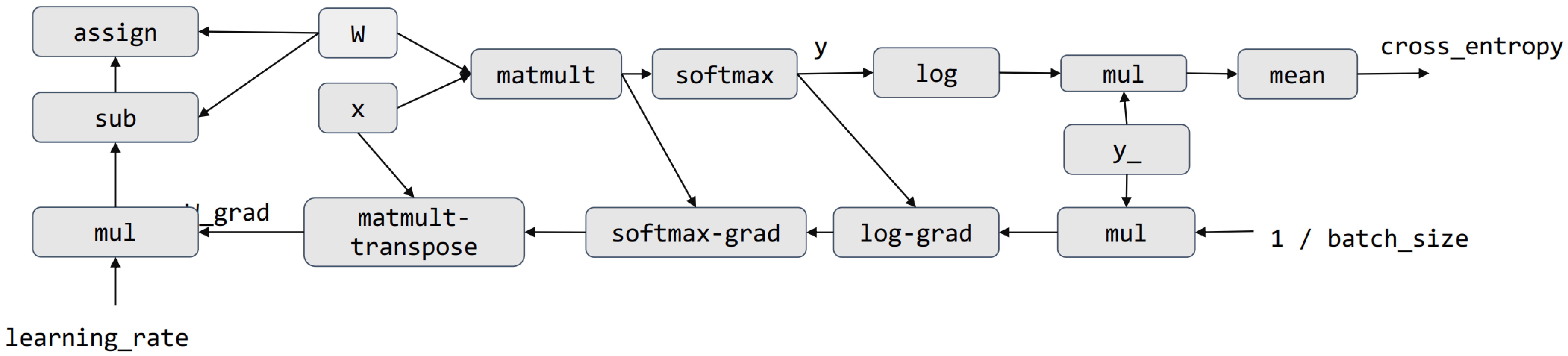
# What happens behind the Scene (Cond.)

```
sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```



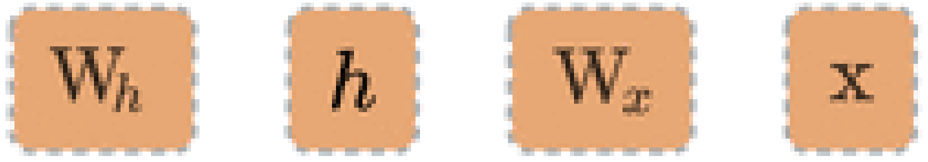
# Discussion

- What are the benefits for computational graph abstraction?
- What are possible implementations and optimizations on this graph?
- What are the cons for computational graph abstraction?



# A different flavor: PyTorch

## A graph is created on the fly




```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

# Topic: Symbolic vs. Imperative

- Symbolic vs. imperative programming
- Define-then-run vs. Define-and-run

```python
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

Symbolic

```python
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

Imperative

# Discussion: Symbolic vs. Imperative

- Symbolic
  - Good
    - easy to optimize (e.g. distributed, batching, parallelization) for developers
    - Much more efficient: can be 10x 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 (that's why we all like Python)
    - Easy to program and easy to debug
  - Bad
    - Less efficient
    - More difficult to optimize

# MCQ Time

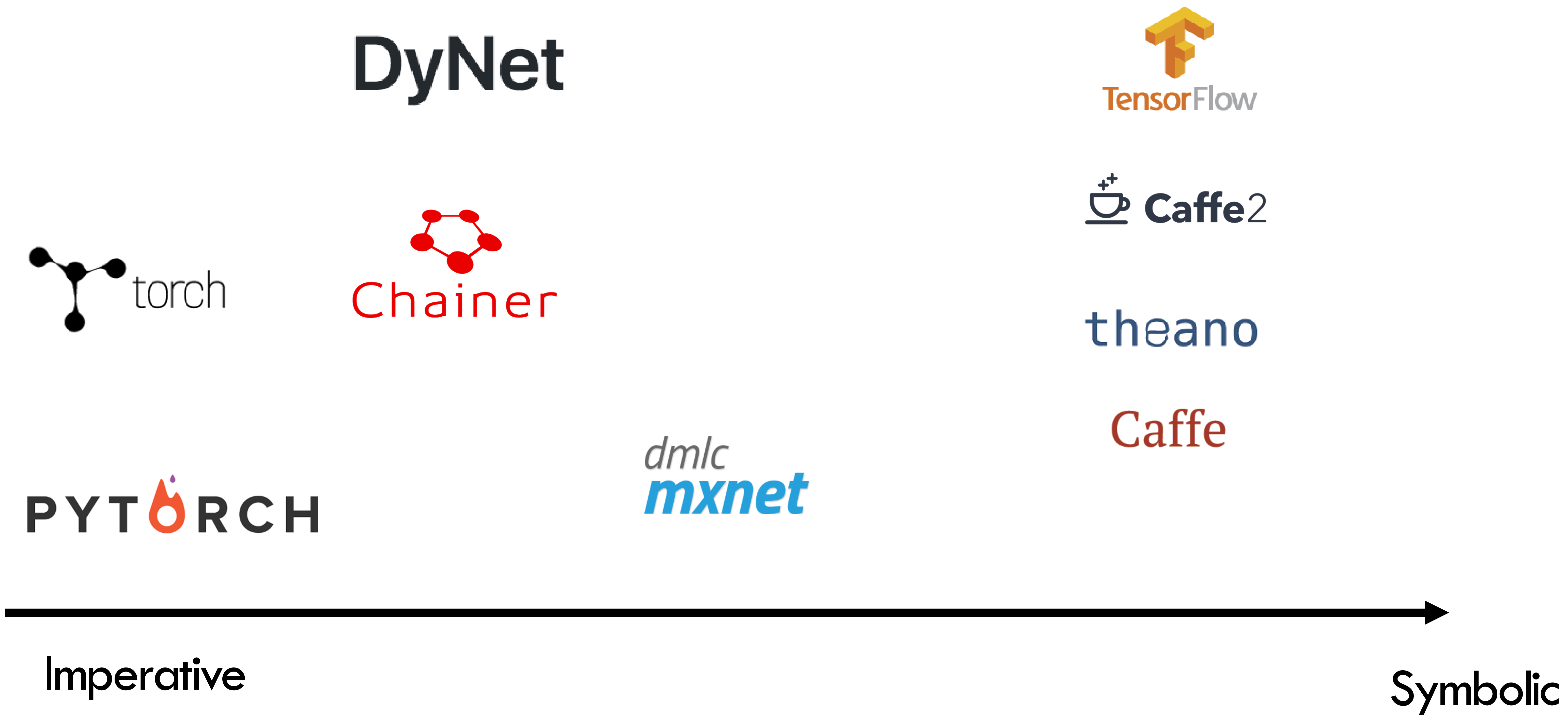
- Which category, symbolic vs. imperative, is the following PL belonging to?
  - C++
  - Python
  - SQL

# Something Interesting Here?

- Python is a *define-and-run* PL
- Tensorflow is *define-then-run* ML framework
- Tensorflow has Python as the primary interface language

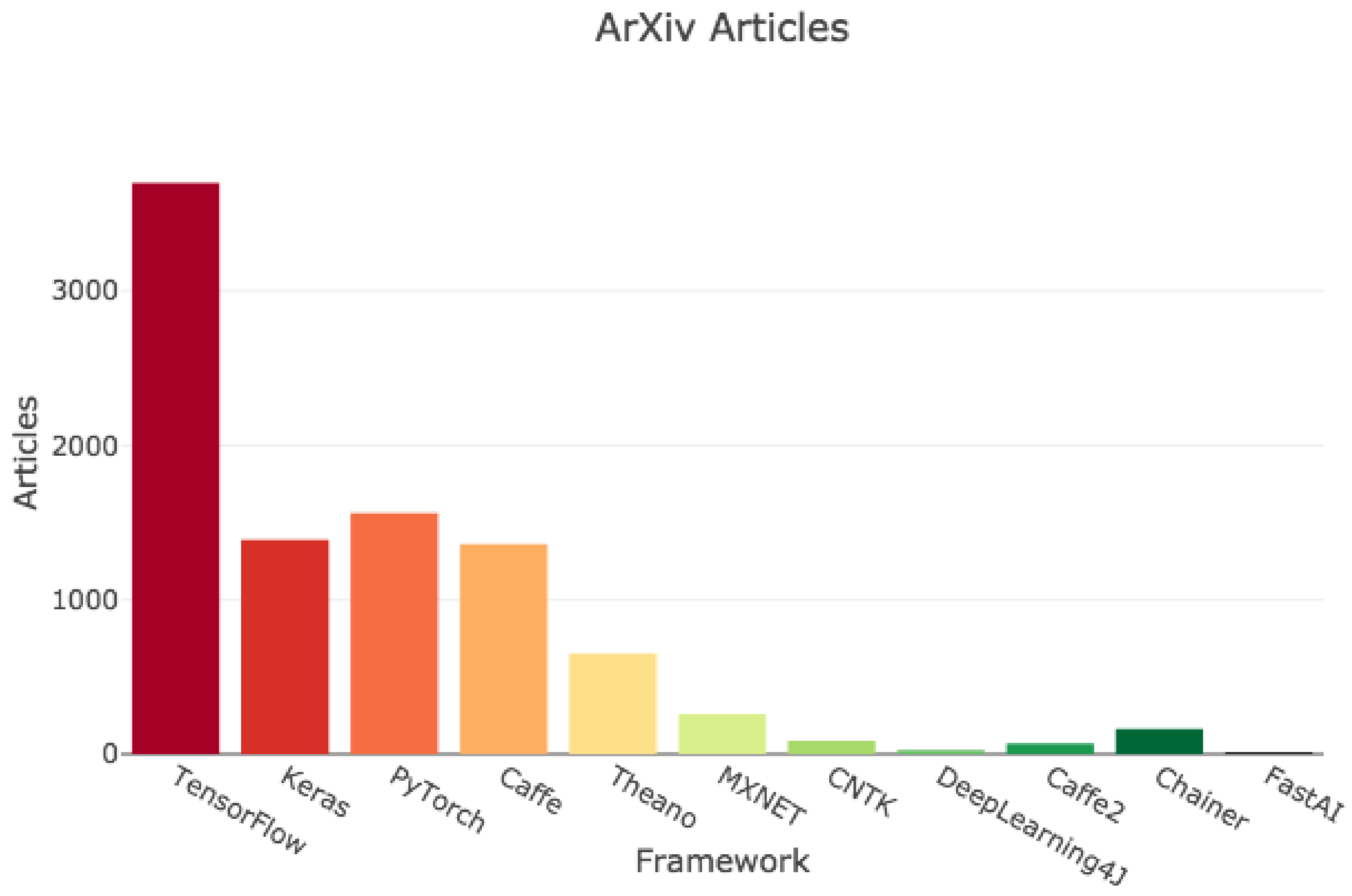
- You are indeed using a DSL built on top of Python
  - But PyTorch DSL is more *pythonic* than Tensorflow DSL.

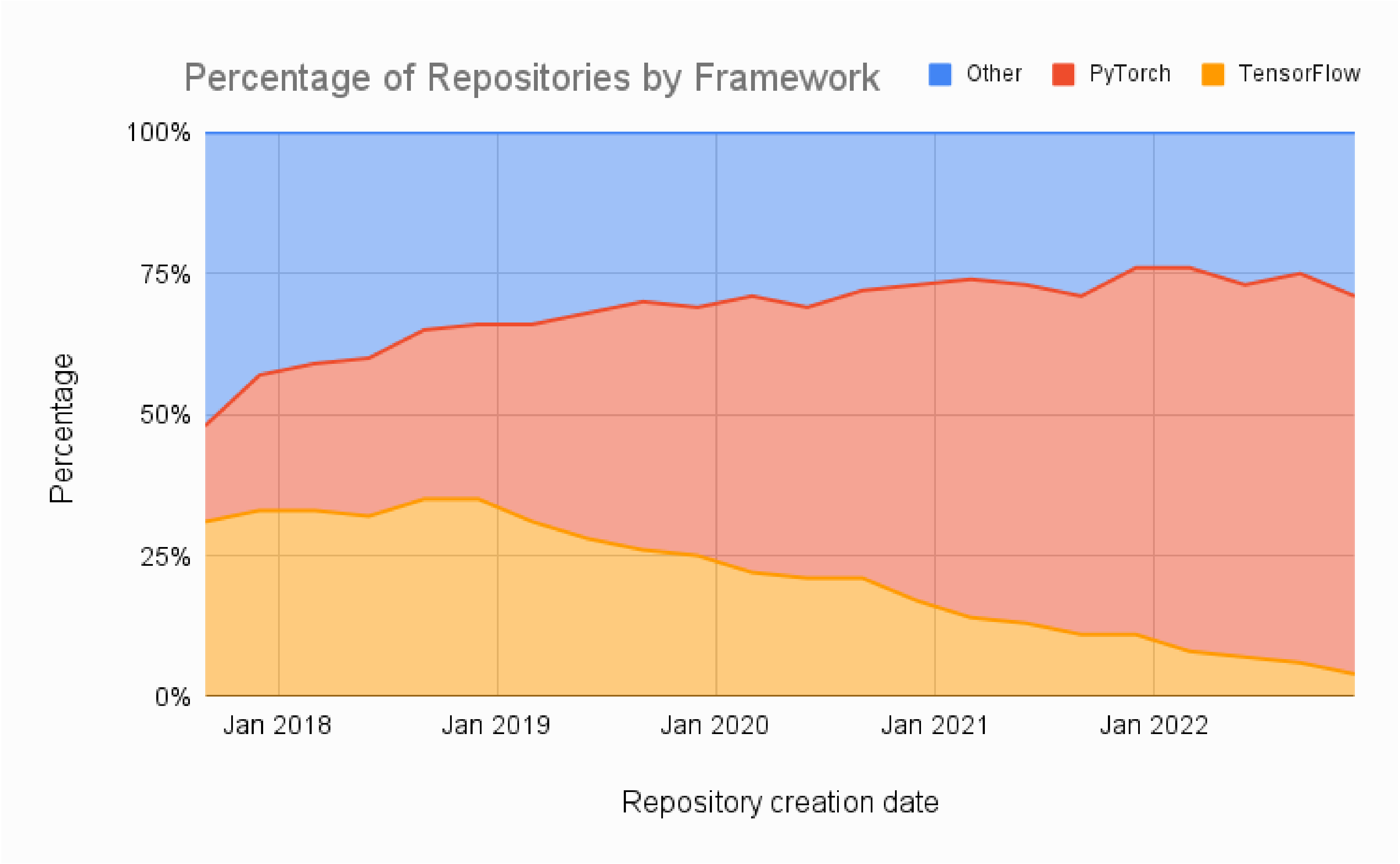
# Symbolic vs. Imperative (2016)

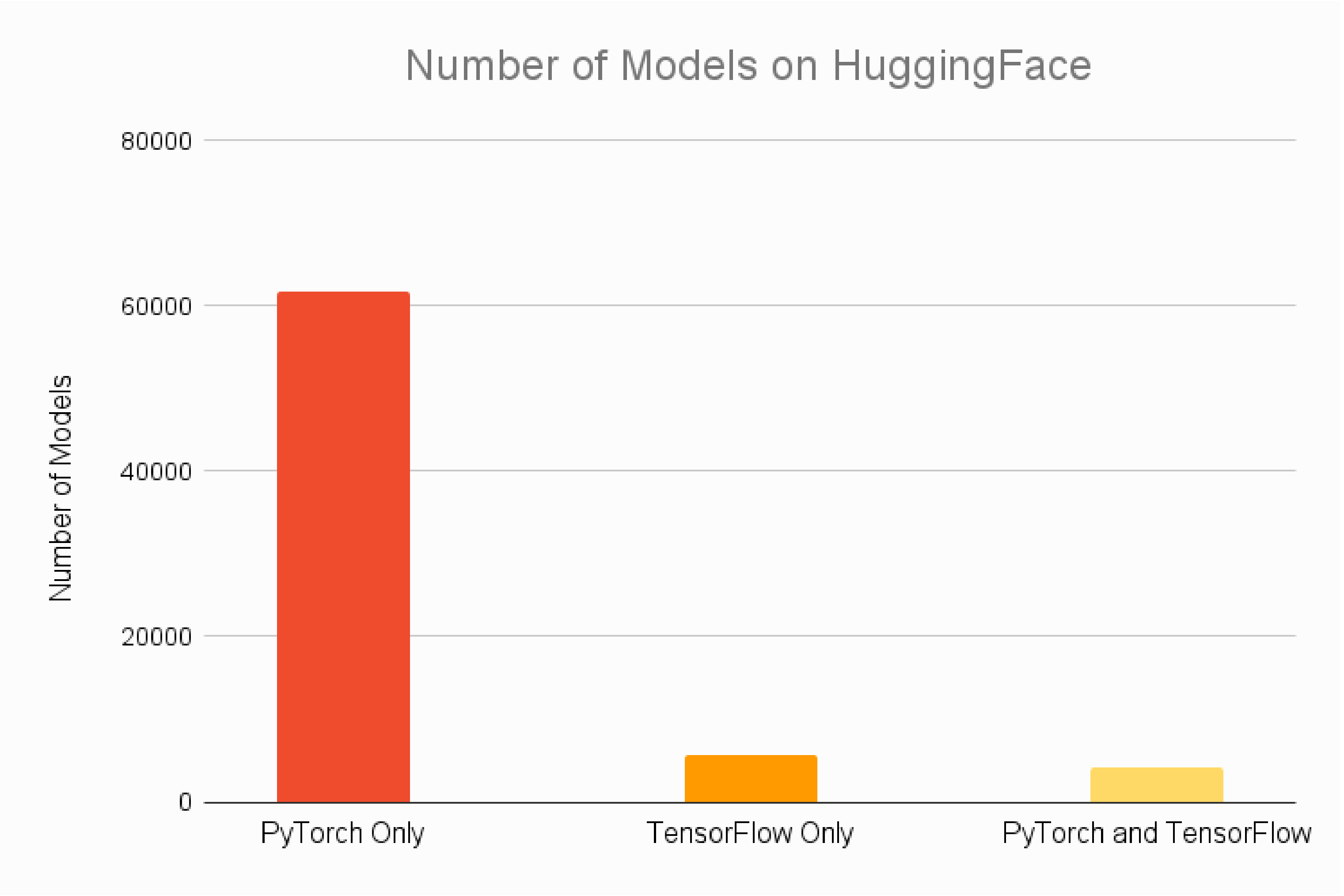


# Symbolic vs. Imperative (2024)

# Market size of frameworks



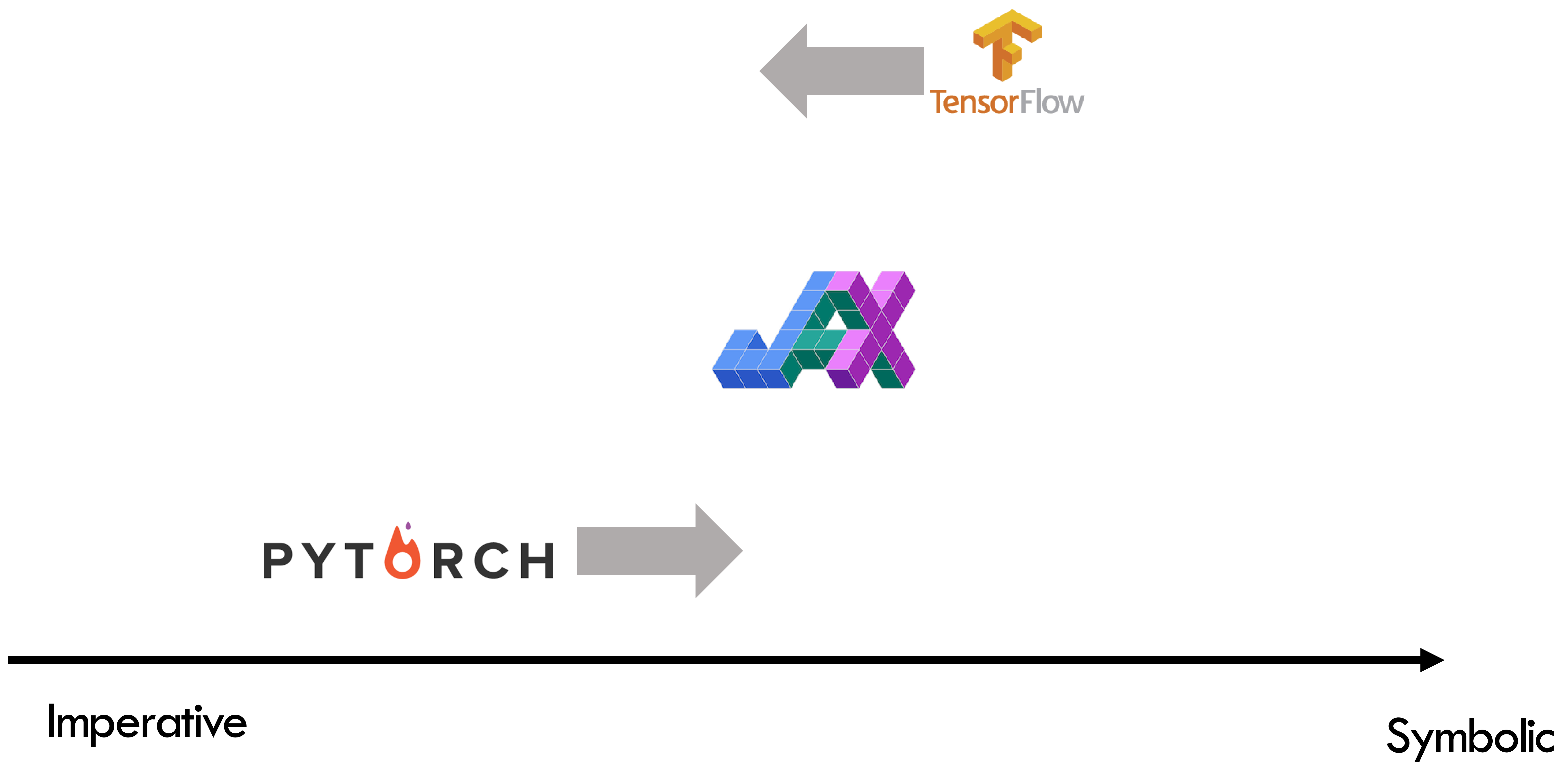



# After-class Question

Why PyTorch wins the market even if it was a later framework?

# Symbolic vs. Imperative (2024)



# Just-in-time (JIT) Compilation

- Ideally, we want define-and-run during ______
- We want define-then-run during _____
- Q: how can combine the best of both worlds?

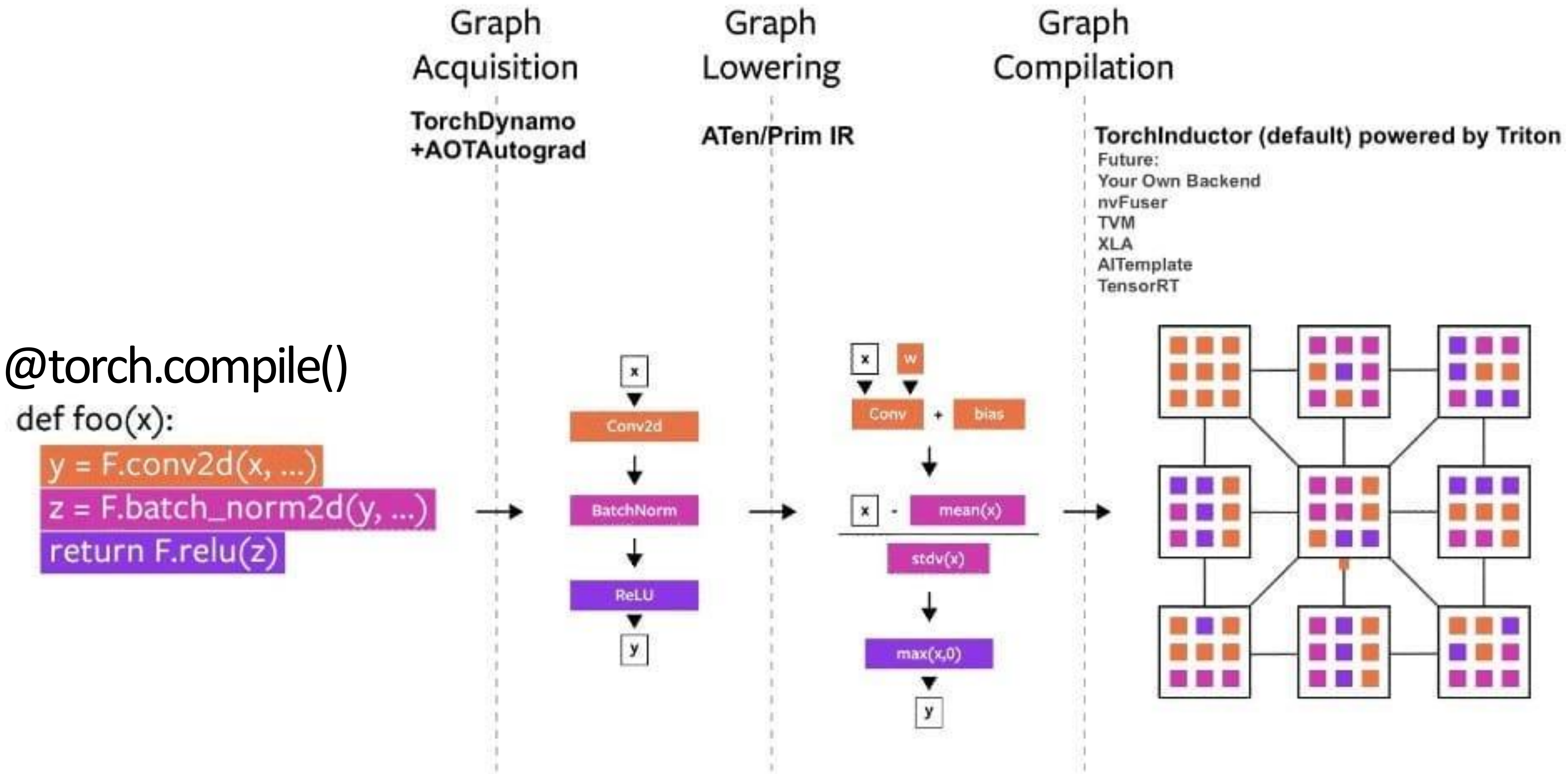
```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

**Dev mode**

```
@torch.compile()
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

**Deploy mode:**
**Decorate torch.compile()**

# What happens behind the scene



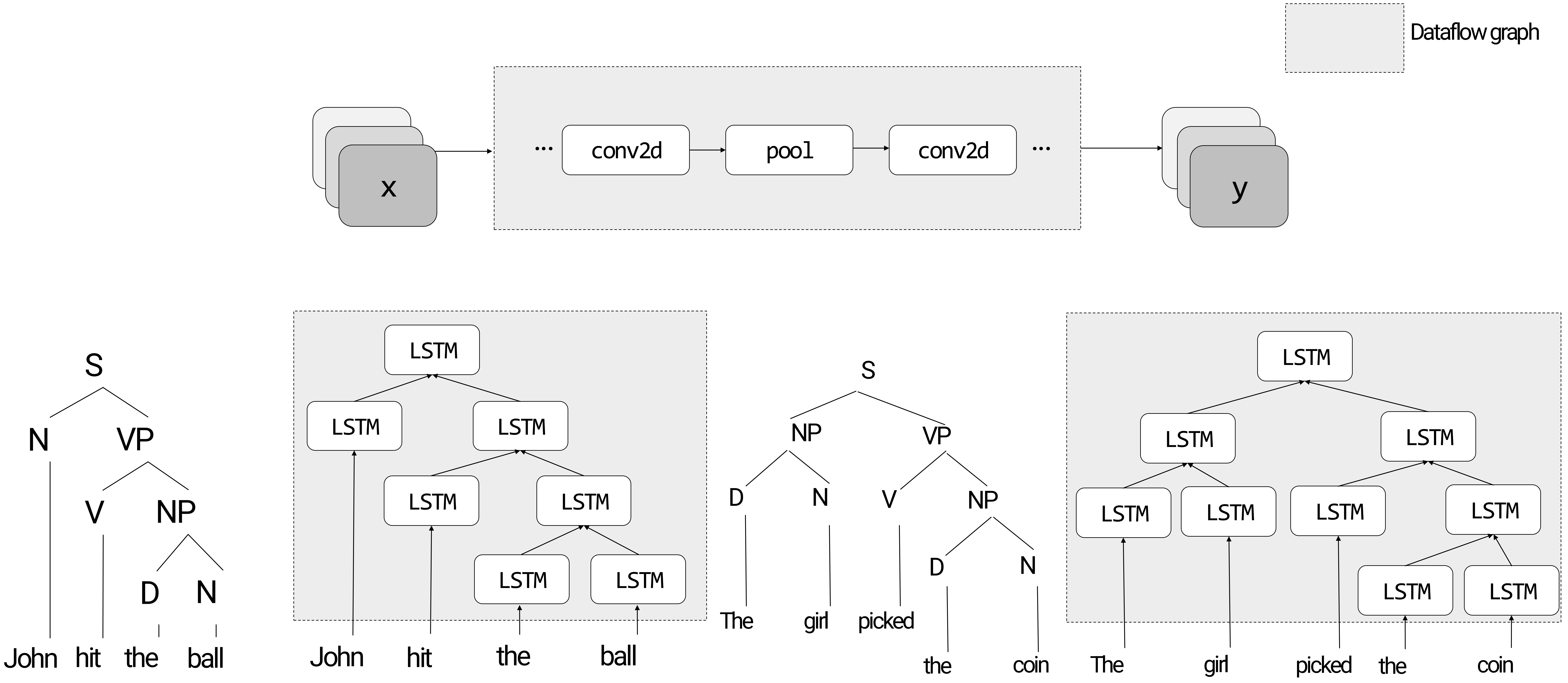

What is the problem of JIT?

Requirements for static graphs

# Q: What is the problem of JIT?

A: Requirements for static graphs

# Static Models vs. Dynamic Models



# High-level Picture

Data

Model

Compute

✅ $\{x_i\}^n{}_{i=1}$

✅ Math primitives (mostly matmul)

❓Make them run on (clusters of ) different kinds of hardware

❓A repr that expresses the computation using primitives

# Next class

A repr that expresses the computation using primitives

✅ A repr that expresses the forward computation using primitives

❓ A repr that expresses the backward computation using primitives

# Recap: how to take derivative?

Given $f(\theta)$, what is $\frac{\partial f}{\partial \theta}$ ?

$$\frac{\partial f}{\partial \theta} = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta)}{\epsilon}$$
$$\approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} + o(\epsilon^2)$$

**Problem:**
**slow:** evaluate f twice to get one gradient
**Error:** approximal and floating point has errors

# Instead, Symbolic Differentiation

Write down the formula, derive the gradient following PD rules

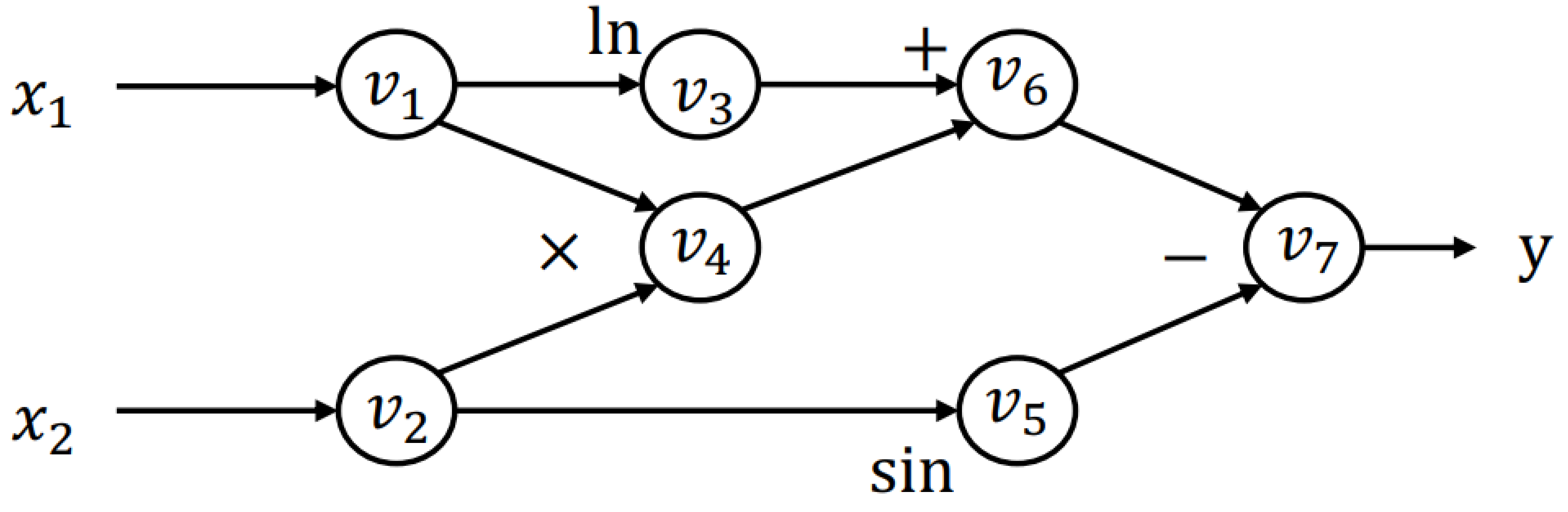
$$\frac{\partial(f(\theta) + g(\theta))}{\partial\theta} = \frac{\partial f(\theta)}{\partial\theta} + \frac{\partial g(\theta)}{\partial\theta}$$

$$\frac{\partial(f(\theta)g(\theta))}{\partial\theta} = g(\theta)\frac{\partial f(\theta)}{\partial\theta} + f(\theta)\frac{\partial g(\theta)}{\partial\theta}$$

$$\frac{\partial(f(g(\theta))}{\partial\theta} = \frac{\partial f(g(\theta))}{\partial g(\theta)}\frac{\partial g(\theta)}{\partial\theta}$$

# Map autodiff rules to computational graph

$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$



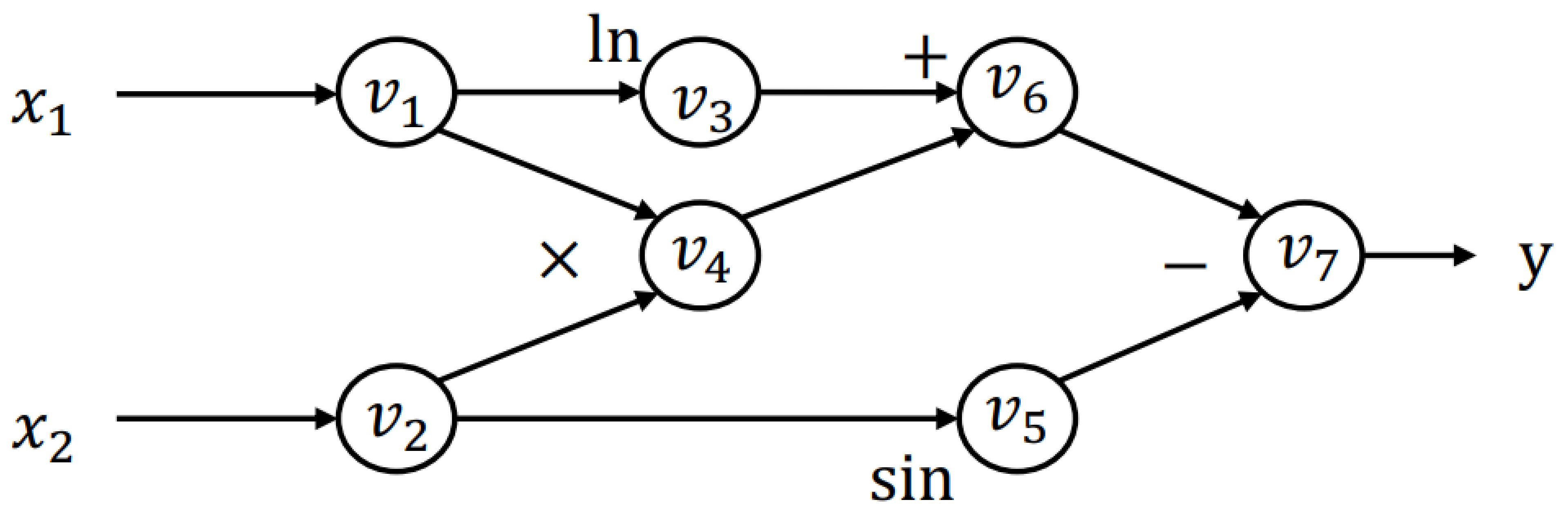

Forward evaluation trace

$v_1 = x_1 = 2$
$v_2 = x_2 = 5$
$v_3 = \ln v_1 = \ln 2 = 0.693$
$v_4 = v_1 \times v_2 = 10$
$v_5 = \sin v_2 = \sin 5 = -0.959$
$v_6 = v_3 + v_4 = 10.693$
$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$
$y = v_7 = 11.652$

- Q: Calculate the value of $\frac{\partial y}{\partial x_1}$
  - A: use PD and chain rules
- There are two ways of applying chain rules
  - Forward: from left (inside) to right (outside)
  - Backward: from right (outside) to left (inside)
  - Which one fits with deep learning?

# Forward Mode AD

$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$




Forward evaluation trace

$$
\begin{aligned}
v_1 &= x_1 = 2 \\
v_2 &= x_2 = 5 \\
v_3 &= \ln v_1 = \ln 2 = 0.693 \\
v_4 &= v_1 \times v_2 = 10 \\
v_5 &= \sin v_2 = \sin 5 = -0.959 \\
v_6 &= v_3 + v_4 = 10.693 \\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
y &= v_7 = 11.652
\end{aligned}
$$

- Define $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$
- We then compute each $\dot{v}_i$ following the forward order of the graph

$$
\begin{aligned}
\dot{v}_1 &= 1 \\
\dot{v}_2 &= 0 \\
\dot{v}_3 &= \dot{v}_1 / v_1 = 0.5 \\
\dot{v}_4 &= \dot{v}_1 v_2 + \dot{v}_2 v_1 = 1 \times 5 + 0 \times 2 = 5 \\
\dot{v}_5 &= \dot{v}_2 \cos v_2 = 0 \times \cos 5 = 0 \\
\dot{v}_6 &= \dot{v}_3 + \dot{v}_4 = 0.5 + 5 = 5.5 \\
\dot{v}_7 &= \dot{v}_6 - \dot{v}_5 = 5.5 - 0 = 5.5
\end{aligned}
$$

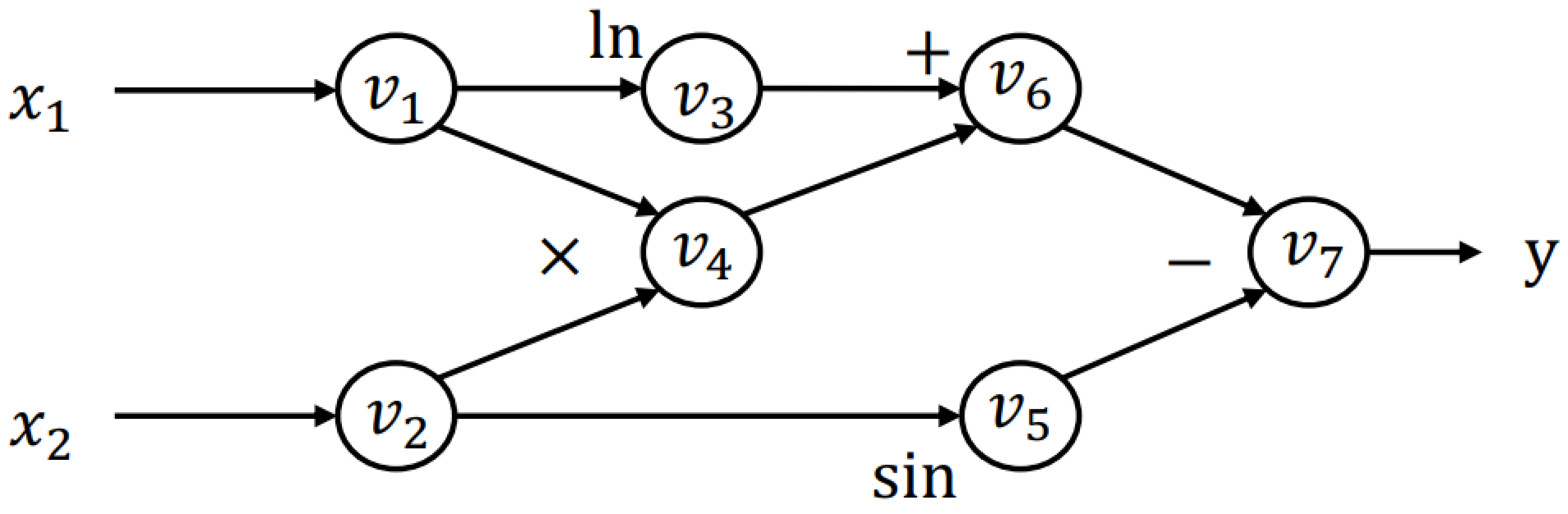
- Finally: $\frac{\partial y}{\partial x_1} = \dot{v}_7 = 5.5$

# Summary: Forward Mode Autodiff

- Start from the input nodes
- Derive gradient all the way to the output nodes
- Pros and Cons of FM Autodiff?
  - For $f: R^n \to R^k$, we need $n$ forward passes to get the grad w.r.t. each input
  - However, in ML: $k = 1$ mostly, and $n$ is very large

# Reverse Mode AD

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$




Forward evaluation trace

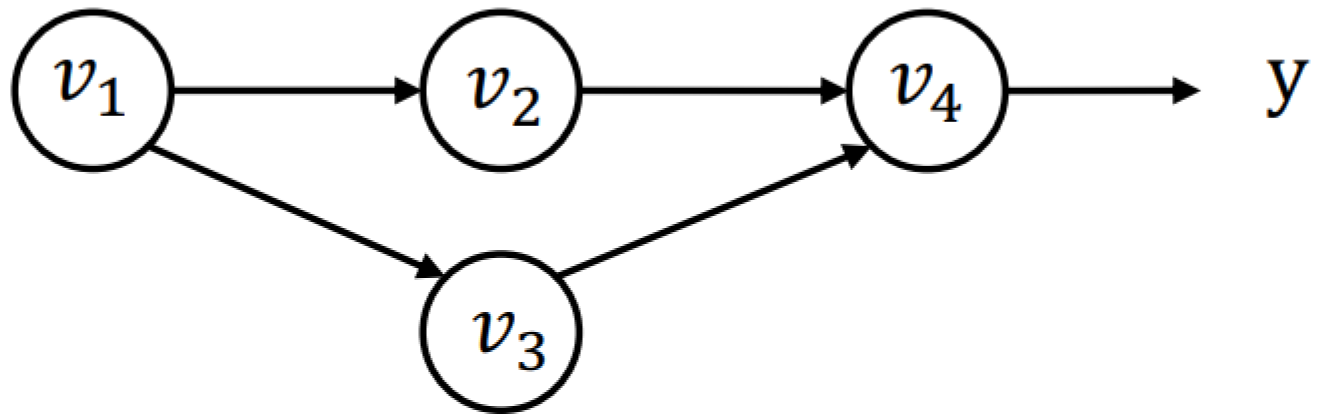
$$v_1 = x_1 = 2$$
$$v_2 = x_2 = 5$$
$$v_3 = \ln v_1 = \ln 2 = 0.693$$
$$v_4 = v_1 \times v_2 = 10$$
$$v_5 = \sin v_2 = \sin 5 = -0.959$$
$$v_6 = v_3 + v_4 = 10.693$$
$$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$$
$$y = v_7 = 11.652$$

- Define adjoint $\overline{v_i} = \frac{\partial y}{\partial v_i}$
- We then compute each $\bar{v}_i$ in the reverse topological order of the graph

$$\overline{v_7} = \frac{\partial y}{\partial v_7} = 1$$
$$\overline{v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7} \times 1 = 1$$
$$\overline{v_5} = \overline{v_7}\frac{\partial v_7}{\partial v_5} = \overline{v_7} \times (-1) = -1$$
$$\overline{v_4} = \overline{v_6}\frac{\partial v_6}{\partial v_4} = \overline{v_6} \times 1 = 1$$
$$\overline{v_3} = \overline{v_6}\frac{\partial v_6}{\partial v_3} = \overline{v_6} \times 1 = 1$$
$$\overline{v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_2} + \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_5} \times \cos v_2 + \overline{v_4} \times v_1 = -0.284 + 2 = 1.716$$
$$\overline{v_1} = \overline{v_4}\frac{\partial v_4}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_4} \times v_2 + \overline{v_3}\,\frac{1}{v_1} = 5 + \frac{1}{2} = 5.5$$

- Finally: $\frac{\partial y}{\partial x_1} = \bar{v}_1 = 5.5$

# Case Study




How to derive the gradient of $v_1$

$$\overline{v_1} = \frac{\partial y}{\partial v_1} = \frac{\partial f(v_2,v_3)}{\partial v_2}\frac{\partial v_2}{\partial v_1} + \frac{\partial f(v_2,v_3)}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}$$

For a $v_i$ used by multiple consumers:

$$\overline{v_i} = \sum_{j \in next(i)} \overline{v_{i\to j}} \quad \text{, where} \quad \overline{v_{i\to j}} = \overline{v_j}\frac{\partial v_j}{\partial v_i}$$

# Summary: Backward Mode Autodiff

- Start from the output nodes
- Derive gradient all the way back to the input nodes
- Discussion: Pros and Cons of FM Autodiff?
  - For $f: R^n \rightarrow R^k$, we need $k$ backward passes to get the grad w.r.t. each input
  - in ML: $k = 1$ and $n$ is very large
  - How about other areas?

# Back to Our Question

A repr that expresses the computation using primitives

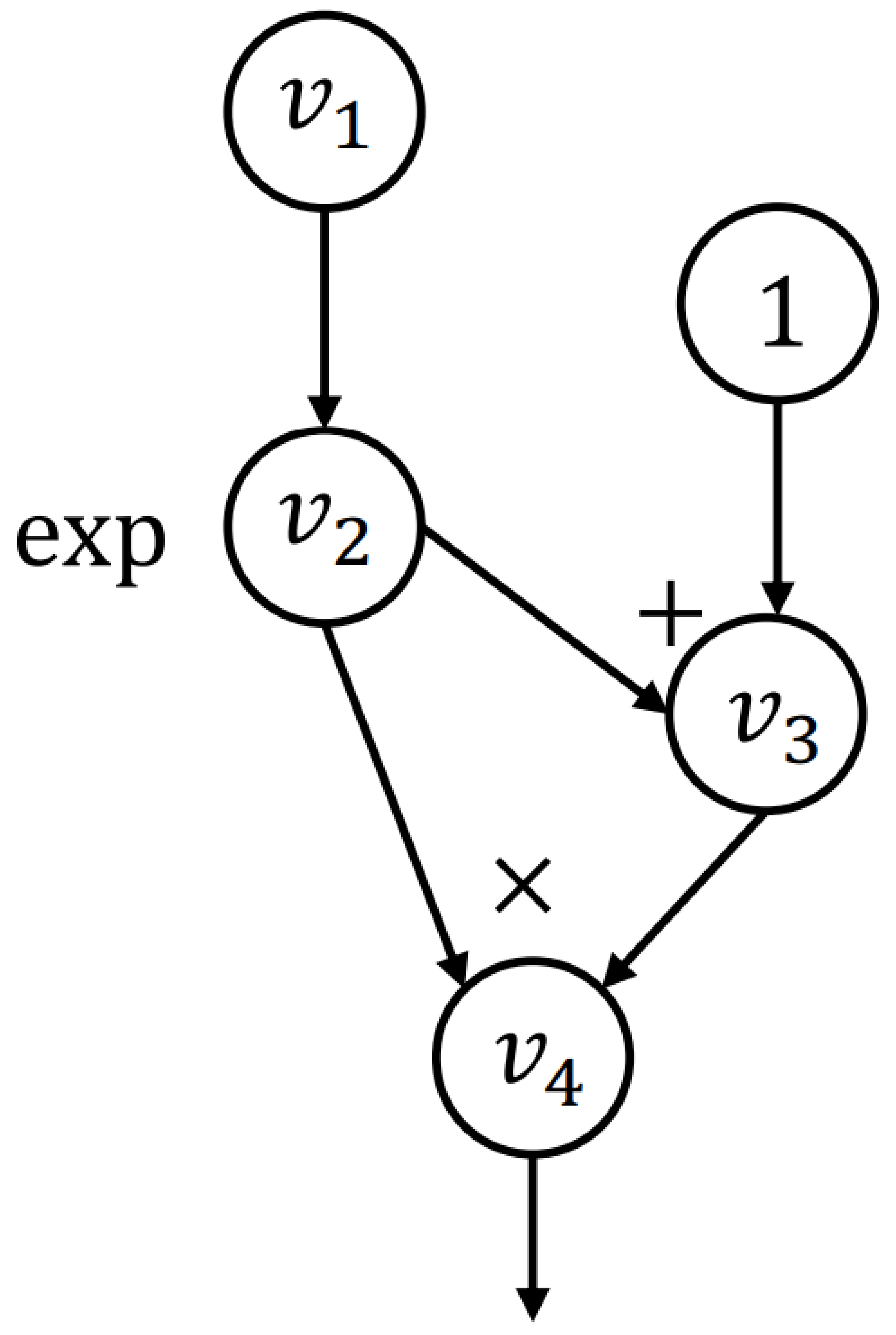
✅ A repr that expresses the forward computation using primitives

❓ A repr that expresses the backward computation using primitives

# Back to our question: Construct the Backward Graph

- How can we construct a computational graph that calculates the adjoint value?

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append  v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```




f: $(\exp(v_1)+1)\exp(v_1)$

# How to implement reverse Autodiff (aka. BP)

def gradient(out):
  node_to_grad = {out: [1]}
  for $i$ in reverse_topo_order(out):
    $\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = sum(node_to_grad[$i$])
    for $k \in inputs(i)$:
      compute $\overline{v_{k\to i}} = \overline{v_i}\ \frac{\partial v_i}{\partial v_k}$
      append $\overline{v_{k\to i}}$ to node_to_grad[$k$]
    return adjoint of input $\overline{v_{input}}$

Record all partial adjoints of a node

Sum up all partial adjoints to get the gradient

Compute and propagates partial adjoints to its inputs.

# Start from $v_4$

i = 4: $v_4 = sum([1]) = 1$

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
```
$\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = sum(node_to_grad[$i$])

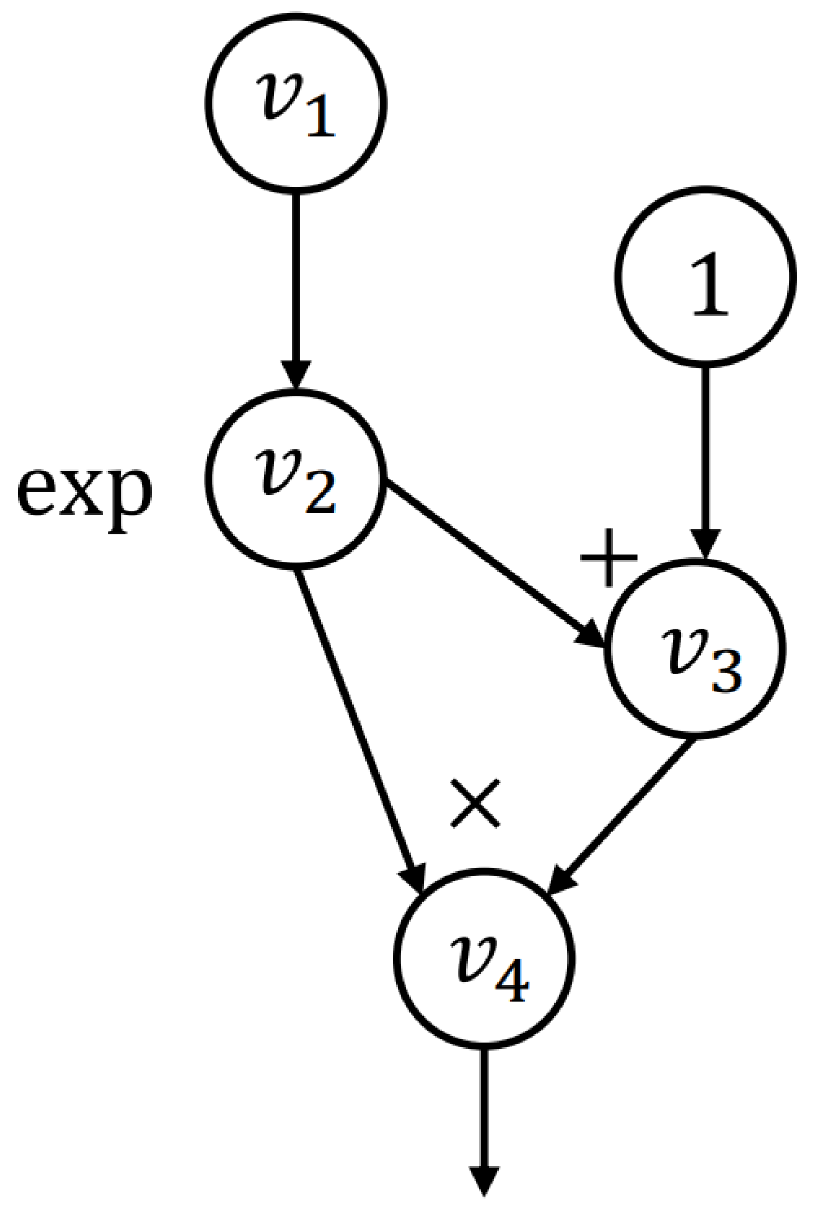
for $k \in inputs(i)$:

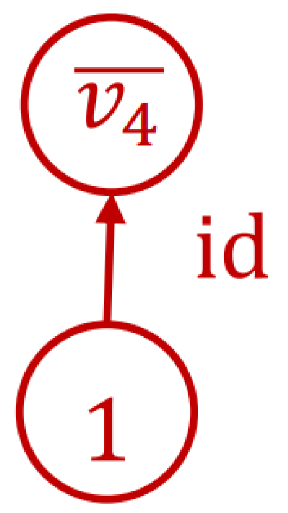
compute $\overline{v_{k\to i}} = \overline{v_i}\ \frac{\partial v_i}{\partial v_k}$

append $\overline{v_{k\to i}}$ to node_to_grad[$k$]

return adjoint of input $\overline{v_{input}}$

$i = 4$
node_to_grad: {
  4: [$\overline{v_4}$]
}



## $v_4$: Inspect $(v_2, v_4)$ and $(v_3, v_4)$

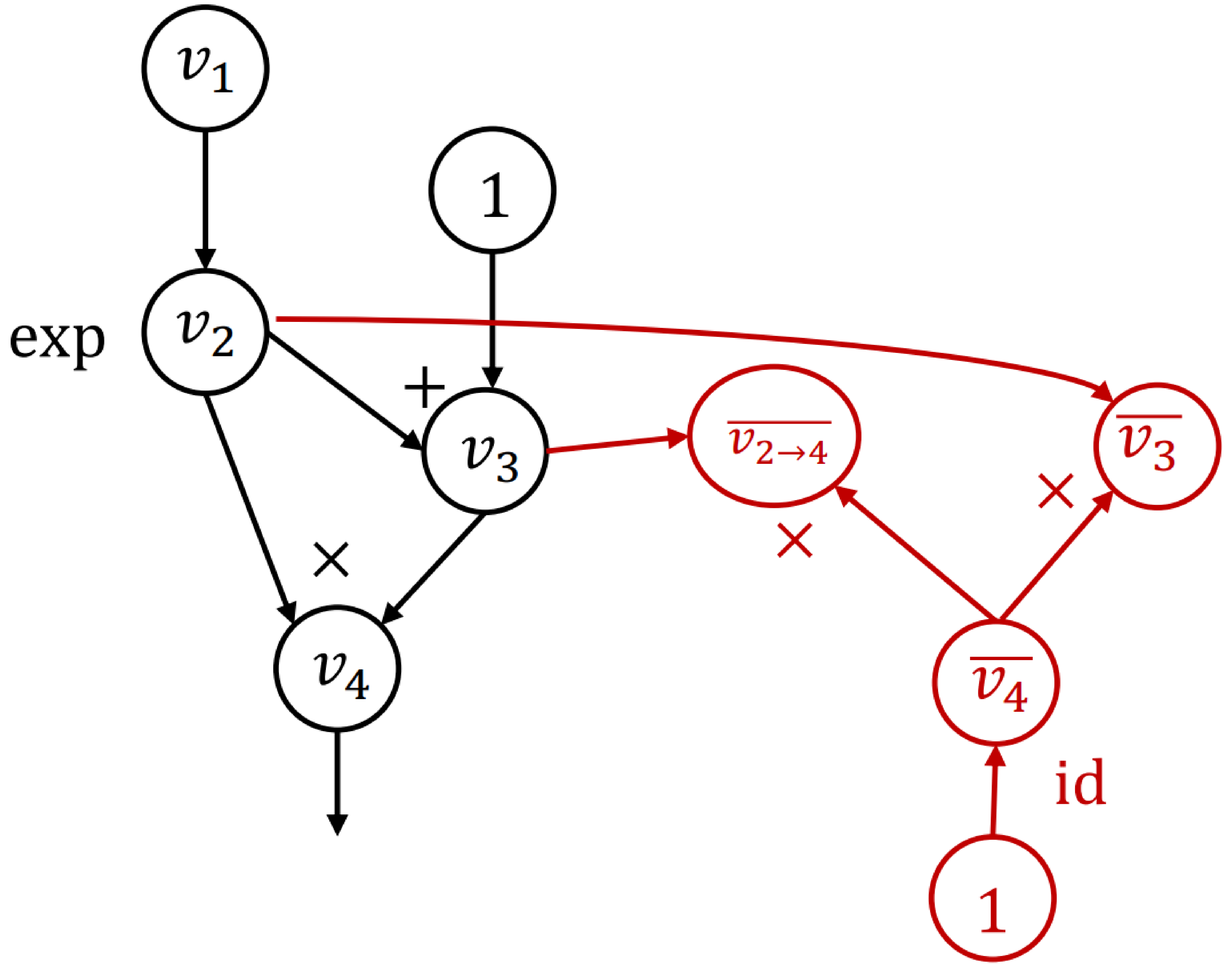
i=4: $\overline{v_4} = sum([1]) = 1$

k=2: $\overline{v_{2\to 4}} = \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_4} v_3$

k=3: $\overline{v_{3\to 4}} = \overline{v_4}\frac{\partial v_4}{\partial v_3} = \overline{v_4} v_2, \overline{v_{3\to 4}} = \overline{v_3}$

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = ∑_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]   ⇐
    return adjoint of input v̄_input
```

$i = 4$
node_to_grad: {
  2: $[\overline{v_{2\to 4}}]$
  3: $[\overline{v_3}]$
  4: $[\overline{v_4}]$
}



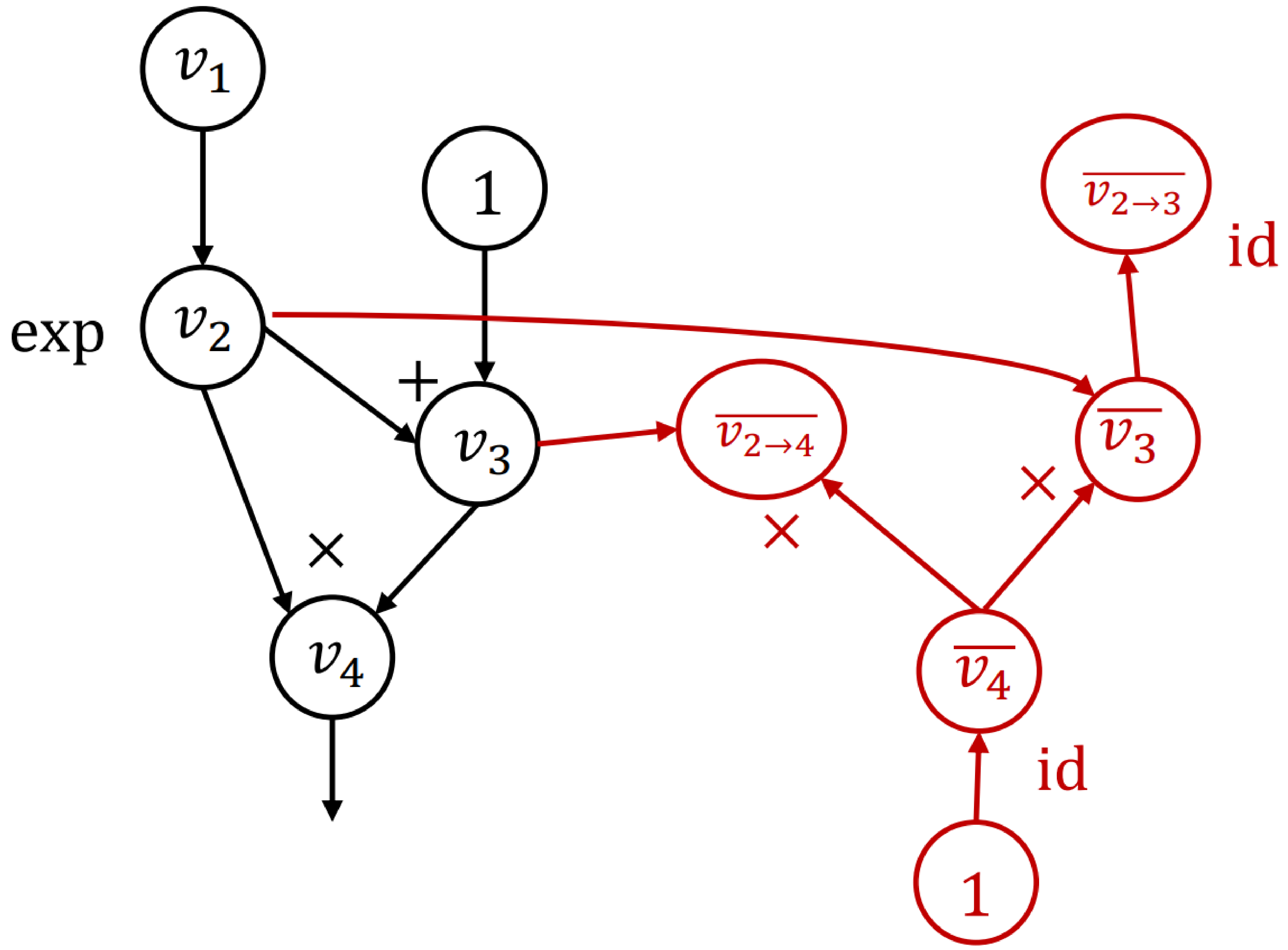
# Inspect $v_3$

i=3: $\overline{v_3}$ done!

k=2: $\overline{v_{2\to3}} = \overline{v_3}\frac{\partial v_3}{\partial v_2} = \overline{v_3}$

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```

```
i = 3
node_to_grad: {
  2: [v̄_{2→4}, v̄_{2→3}]
  3: [v̄_3]
  4: [v̄_4]
}
```

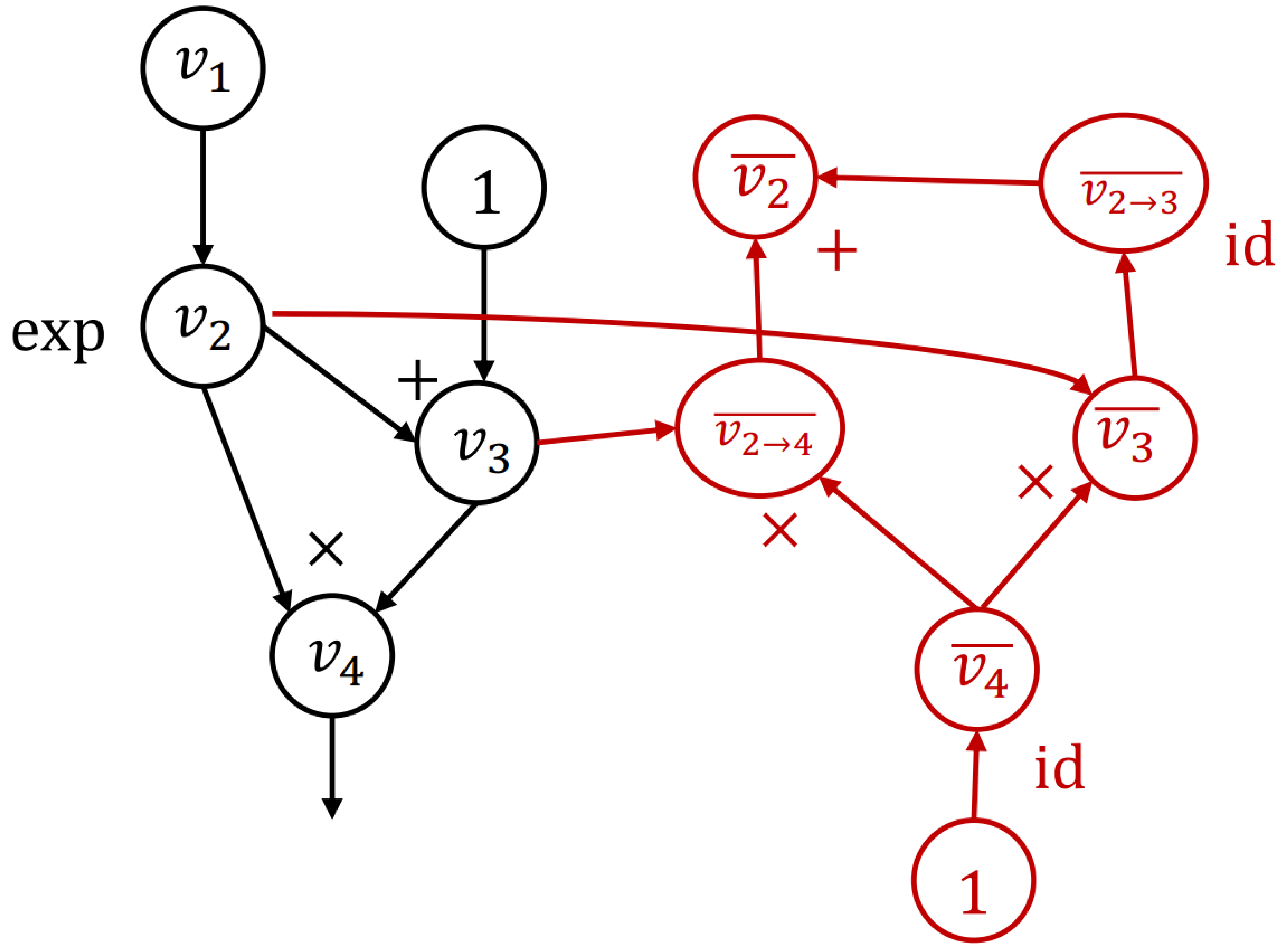


i=2: $\overline{v_2} = \overline{v_{2\to3}} + \overline{v_{2\to4}}$

# Inspect $v_2$

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append  v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```

$i = 2$
node_to_grad: {
  2: [$\overline{v_{2\to4}}, \overline{v_{2\to3}}$]
  3: [$\overline{v_3}$]
  4: [$\overline{v_4}$]
}



# Inspect $(v_1, v_2)$

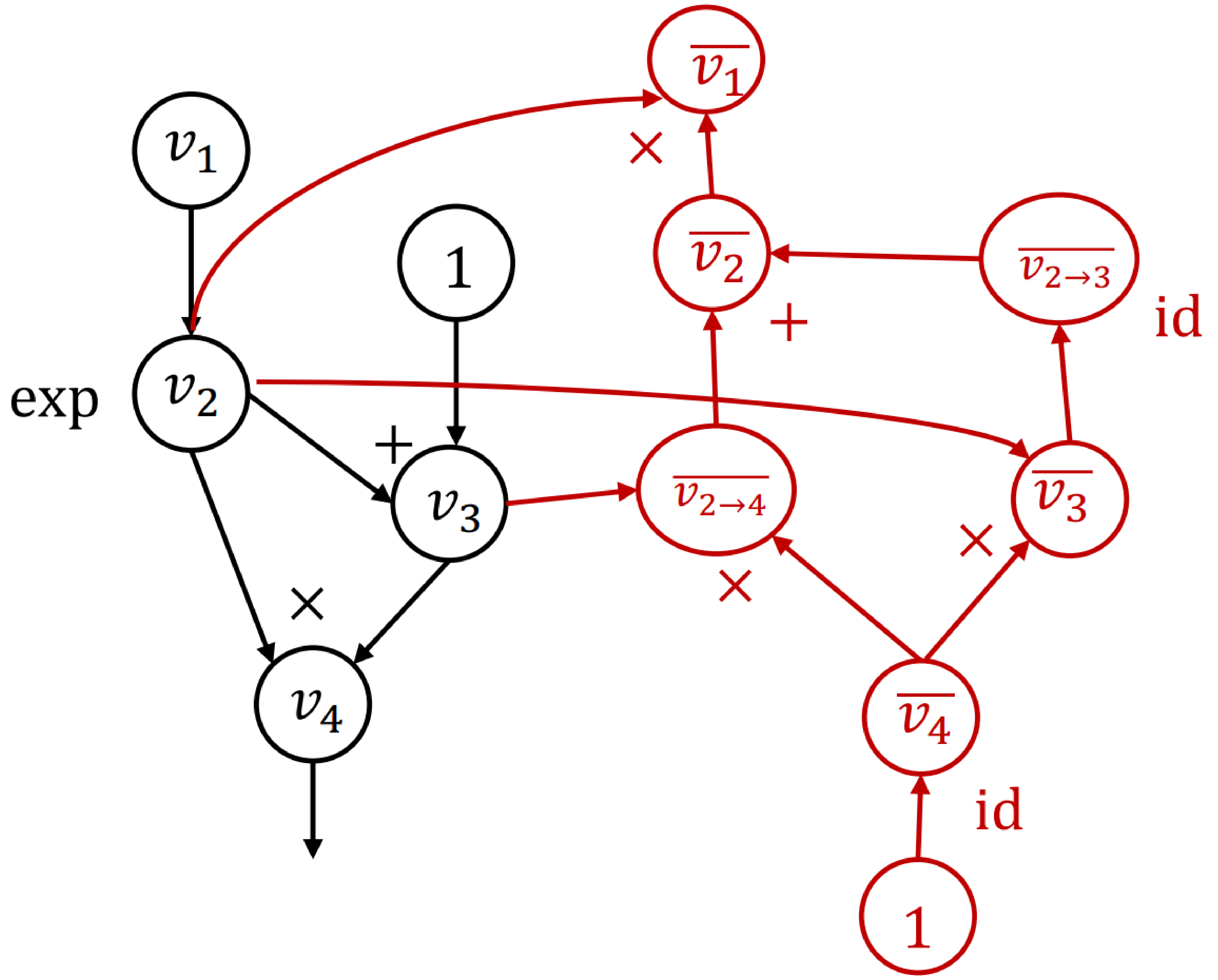
i=2: $\overline{v_2} = \overline{v_{2\to3}} + \overline{v_{2\to4}}$

k=1: $\overline{v_{1\to2}} = \overline{v_2}\frac{\partial v_2}{\partial v_1} = \overline{v_2}\exp(v_1)$,

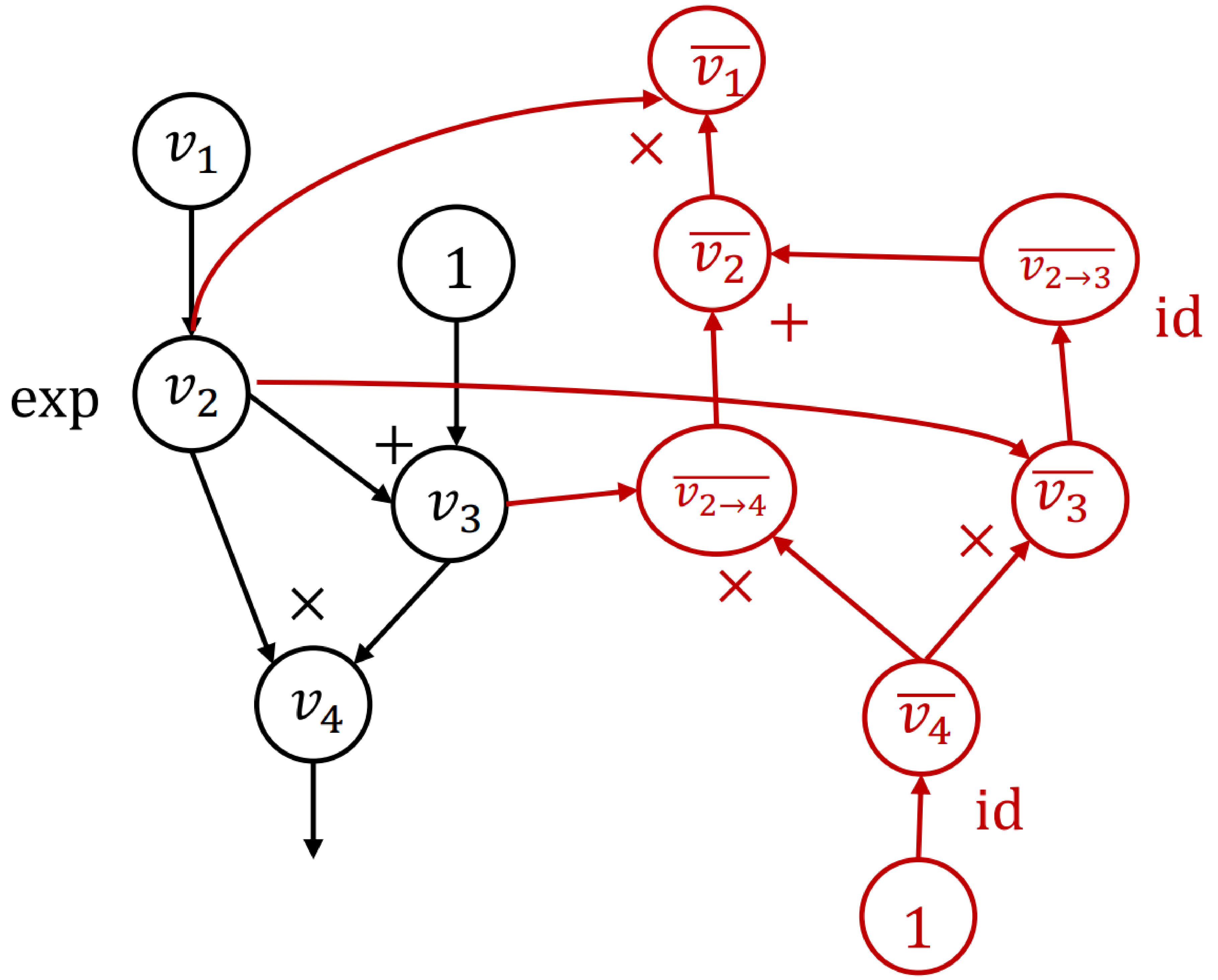
$\overline{v_1} = \overline{v_{1\to2}}$

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```

$i = 2$
node_to_grad: {
  1: $[\overline{v_1}]$
  2: $[\overline{v_{2\to4}}, \overline{v_{2\to3}}]$
  3: $[\overline{v_3}]$
  4: $[\overline{v_4}]$
}

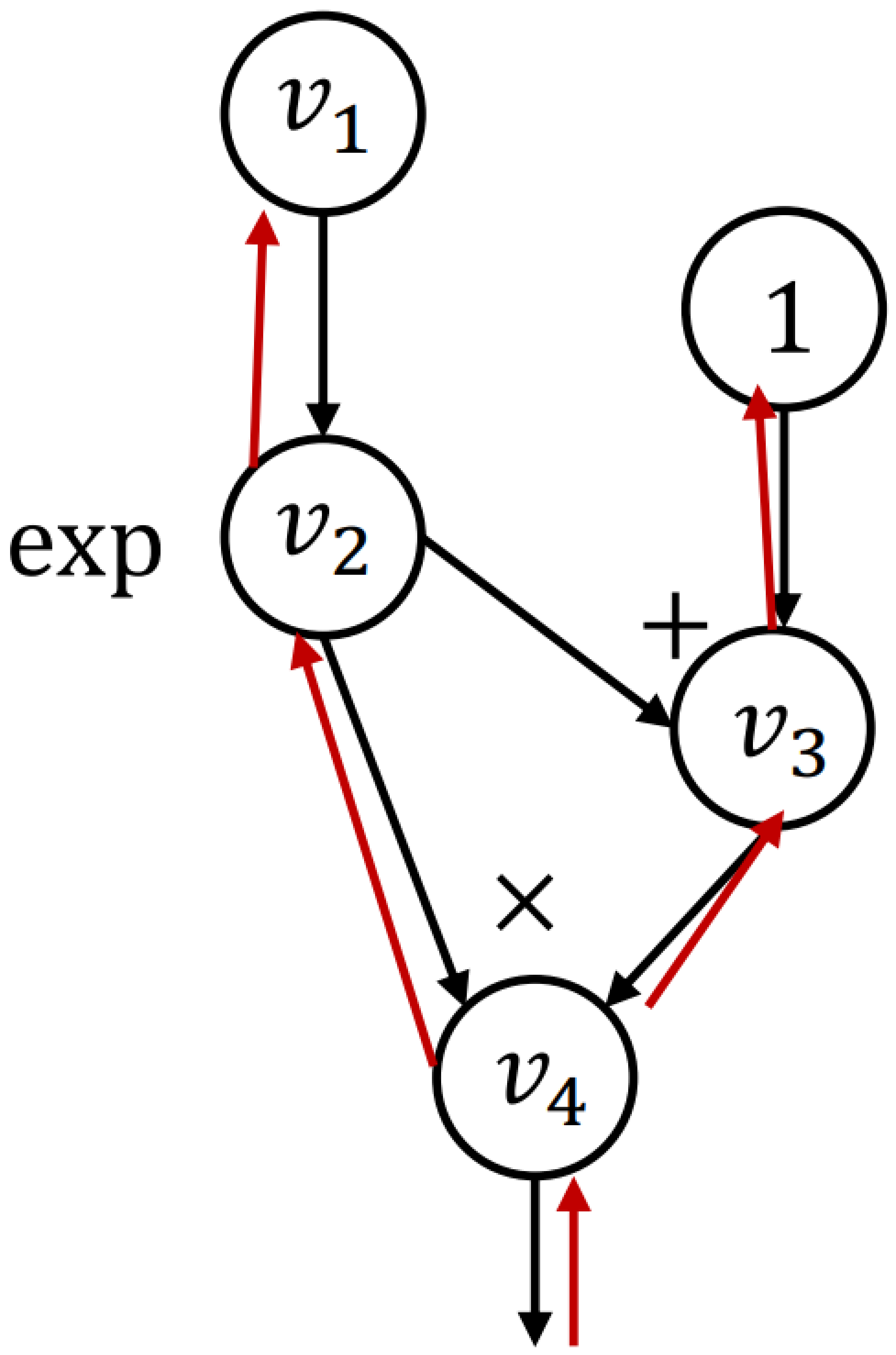


# Summary: Backward AD

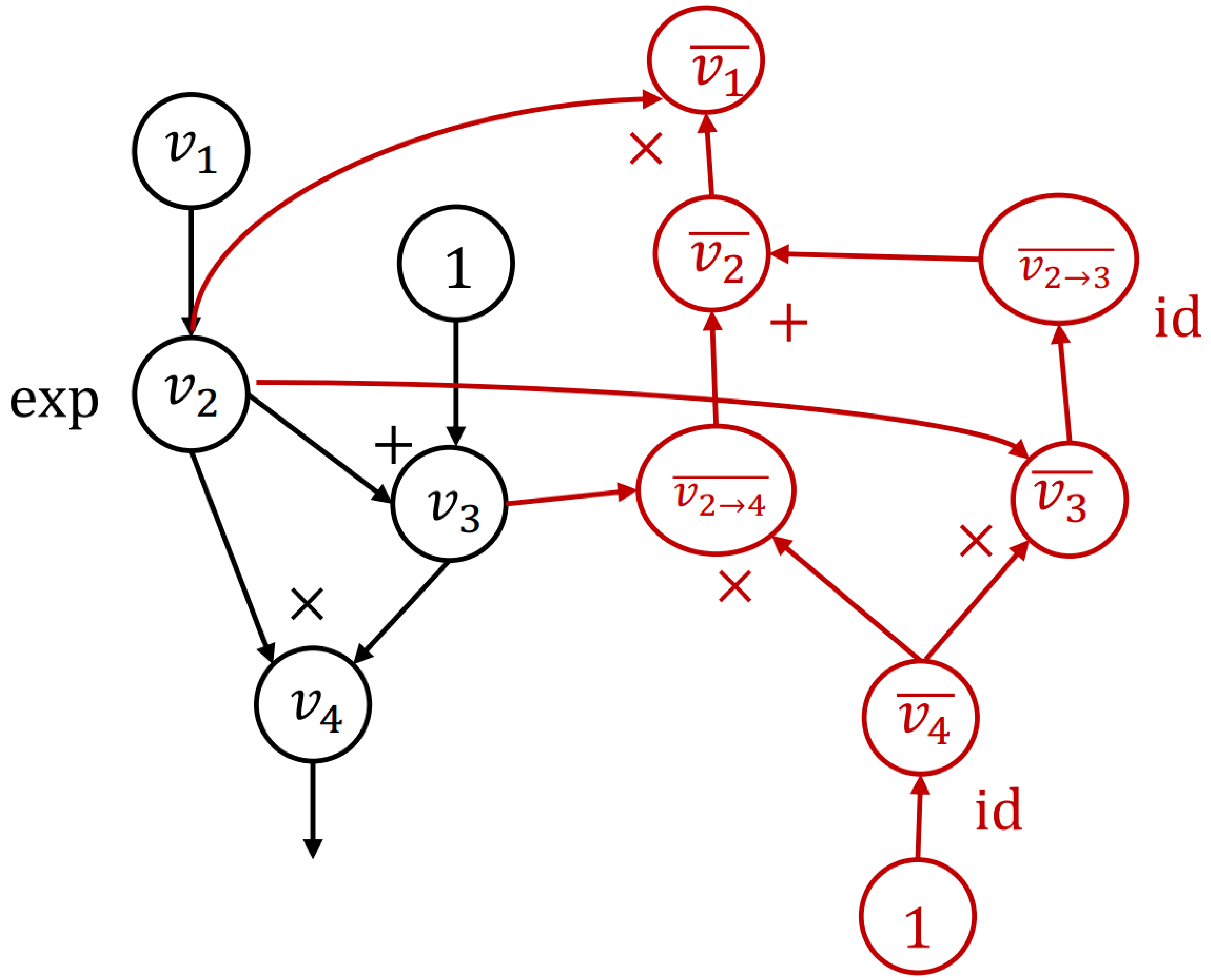



- Construct backward graph in a symbolic way (instead of concrete values)
- This graph can be reused by different input values

# Backpropagation vs. Reverse-mode AD




vs.

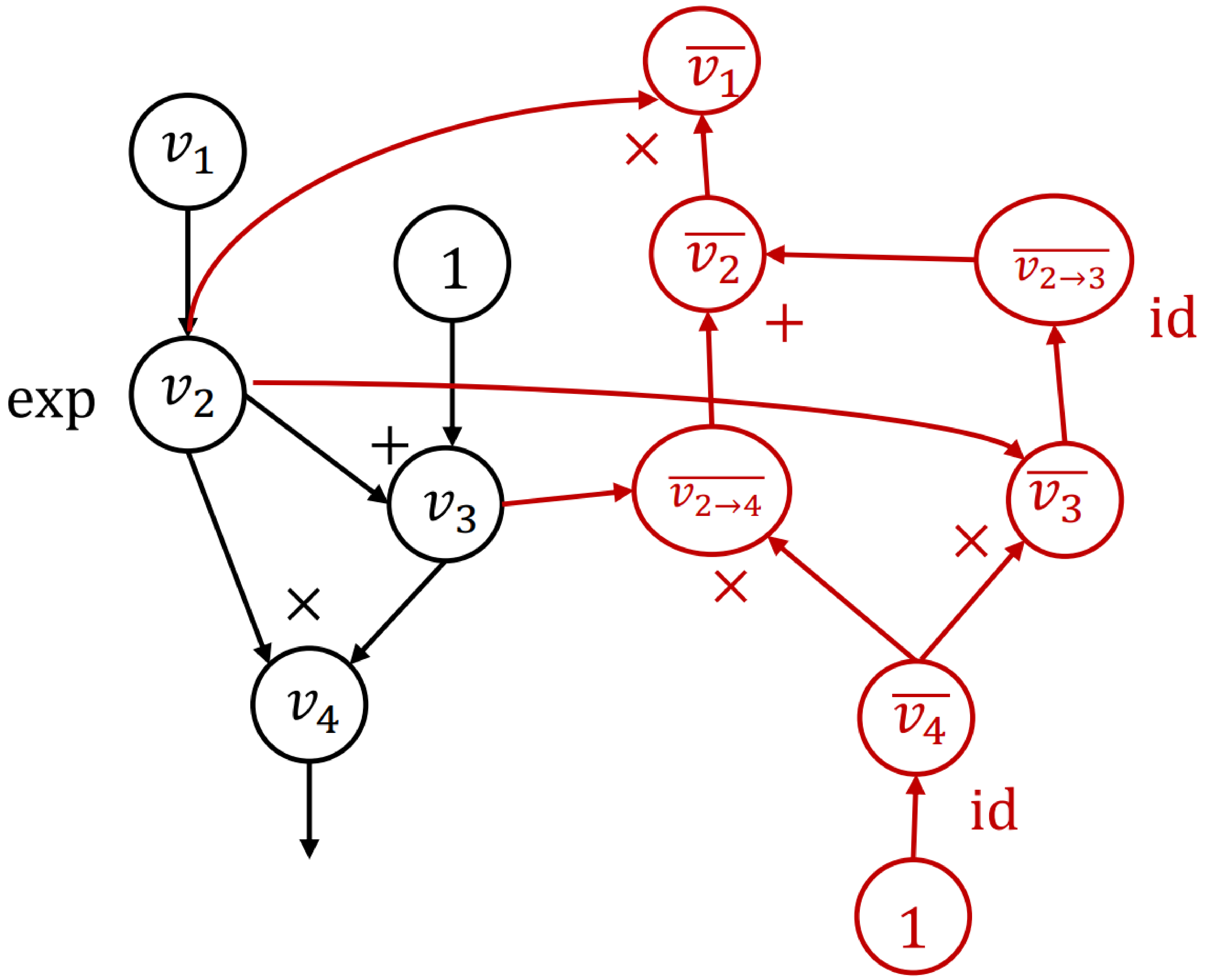



- Run backward through the forward graph
- Caffe/cuda-convnet

- Construct backward graph
- Used by TensorFlow, PyTorch

# Incomplete yet?

- What is the missing from the following graph for ML training?



# Recall Our Master Equation

$$\theta^{(t+1)} = f\left(\theta^{(t)},\, \nabla_L\left(\theta^{(t)},\, D^{(t)}\right)\right)$$



$$L = \mathrm{MSE}(w_2 \cdot \mathrm{ReLU}(w_1 x),\, y) \quad \theta = \{w_1, w_2\},\, D = \{(x, y)\}$$

$$f(\theta, \nabla_L) = \theta - \nabla_L$$

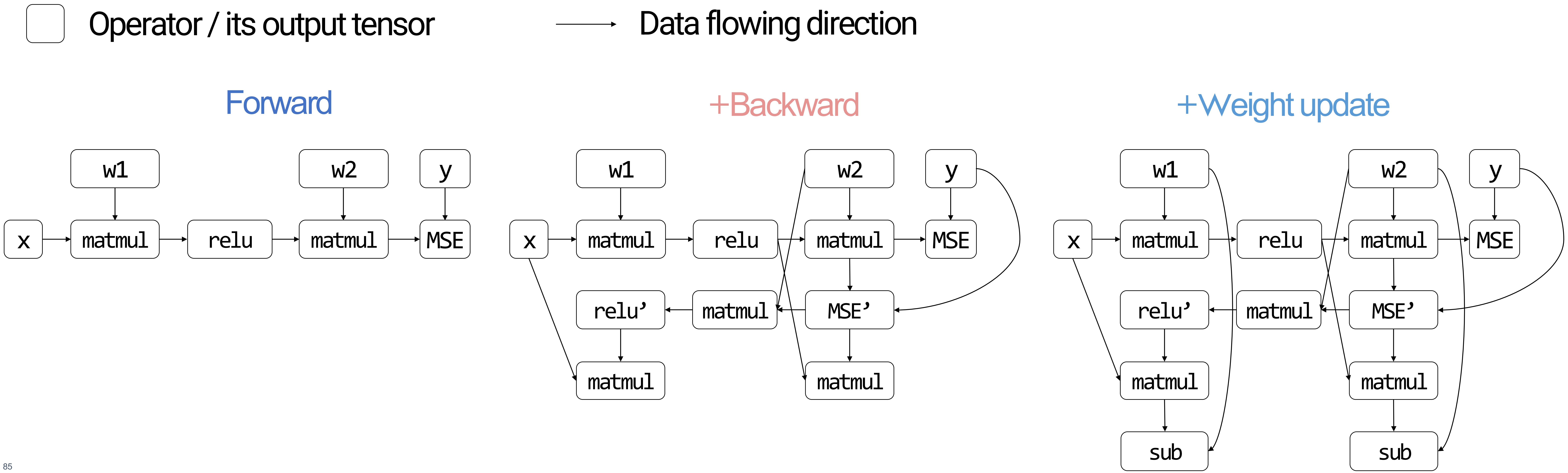
| Forward | Backward | Weight update |
| --- | --- | --- |
| $L(\cdot)$ | $\nabla_L(\cdot)$ | $f(\cdot)$ |

# Put in Practice

$$\theta^{(t+1)} = f\left(\theta^{(t)},\ \nabla_L\left(\theta^{(t)},\ D^{(t)}\right)\right)$$

$$L = \mathrm{MSE}(w_2 \cdot \mathrm{ReLU}(w_1 x),\ y) \quad \theta = \{w_1, w_2\},\ D = \{(x, y)\}$$

$$f(\theta, \nabla_L) = \theta - \nabla_L$$



# Homework: How to derive gradients for

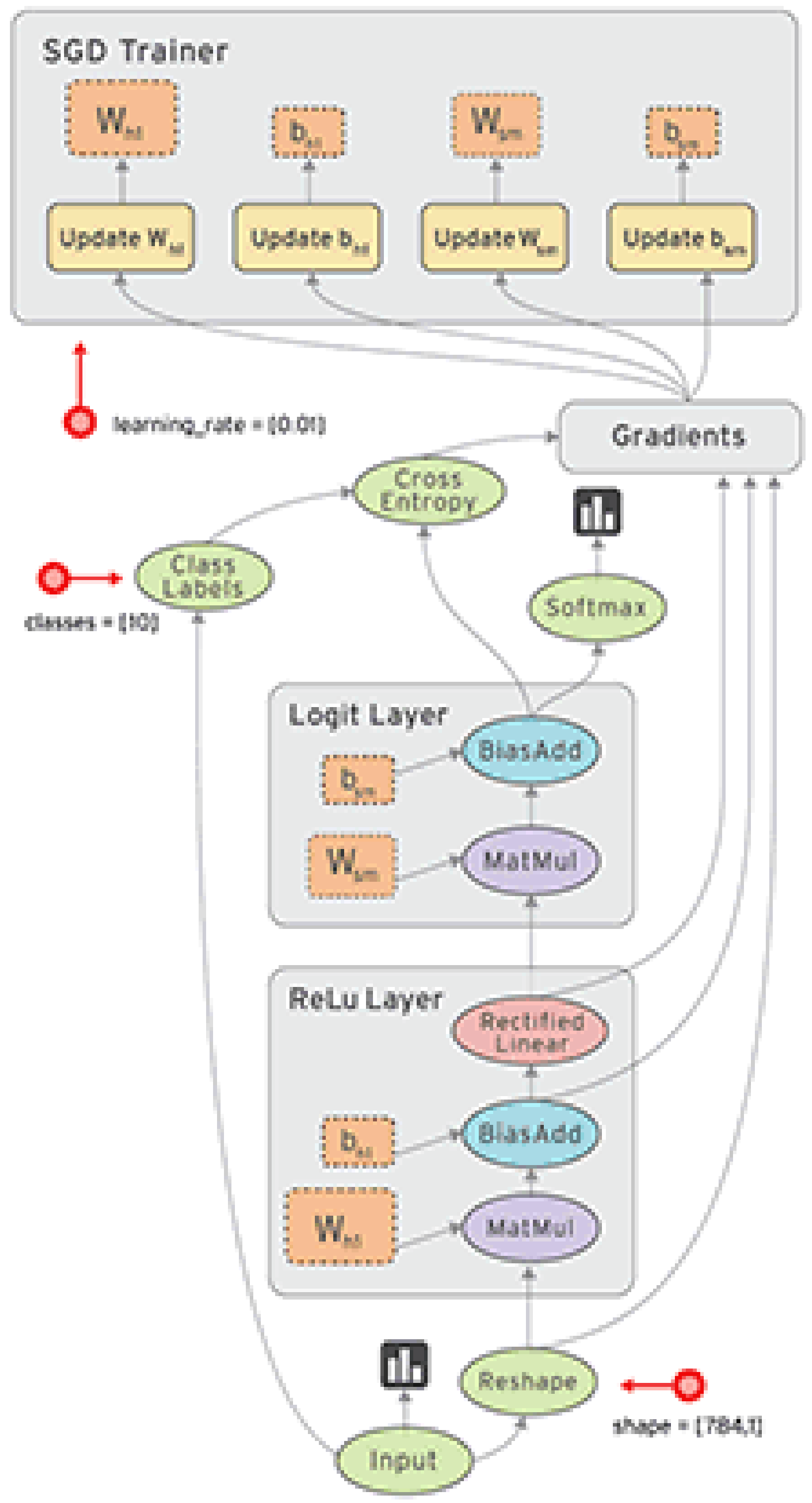
- Softmax cross entropy:

$$L = -\sum t_i \log(y_i)\,, y_i = softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum e^{x_d}}$$

# Today

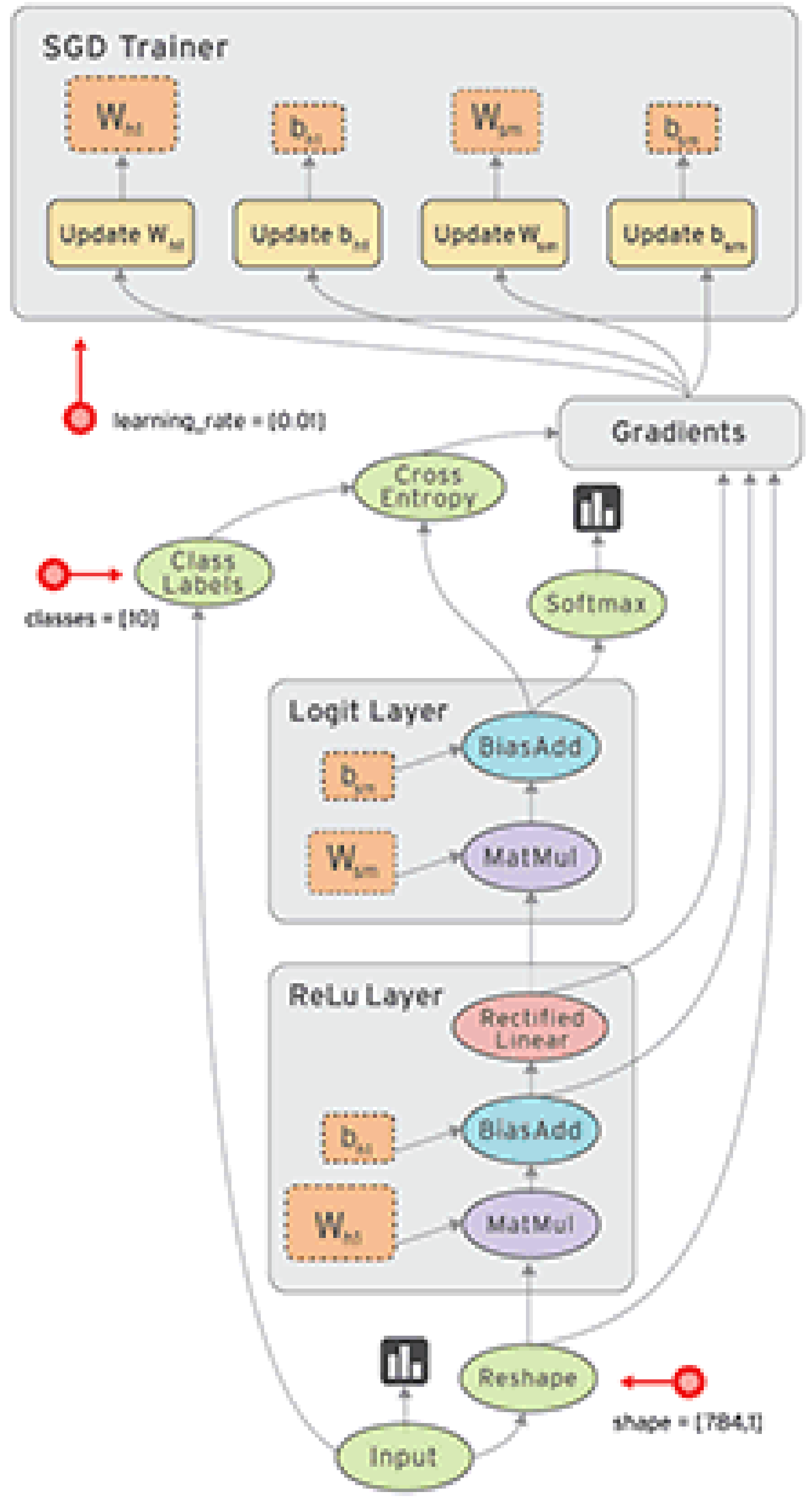
- Autodiff
- **Architecture Overview**

# MLSys' Grand problem




- Our system goals:
  - Fast
  - Scale
  - Memory-efficient
  - Run on diverse hardware
  - Energy-efficient
  - Easy to program/debug/deploy

# ML System Overview




Dataflow Graph

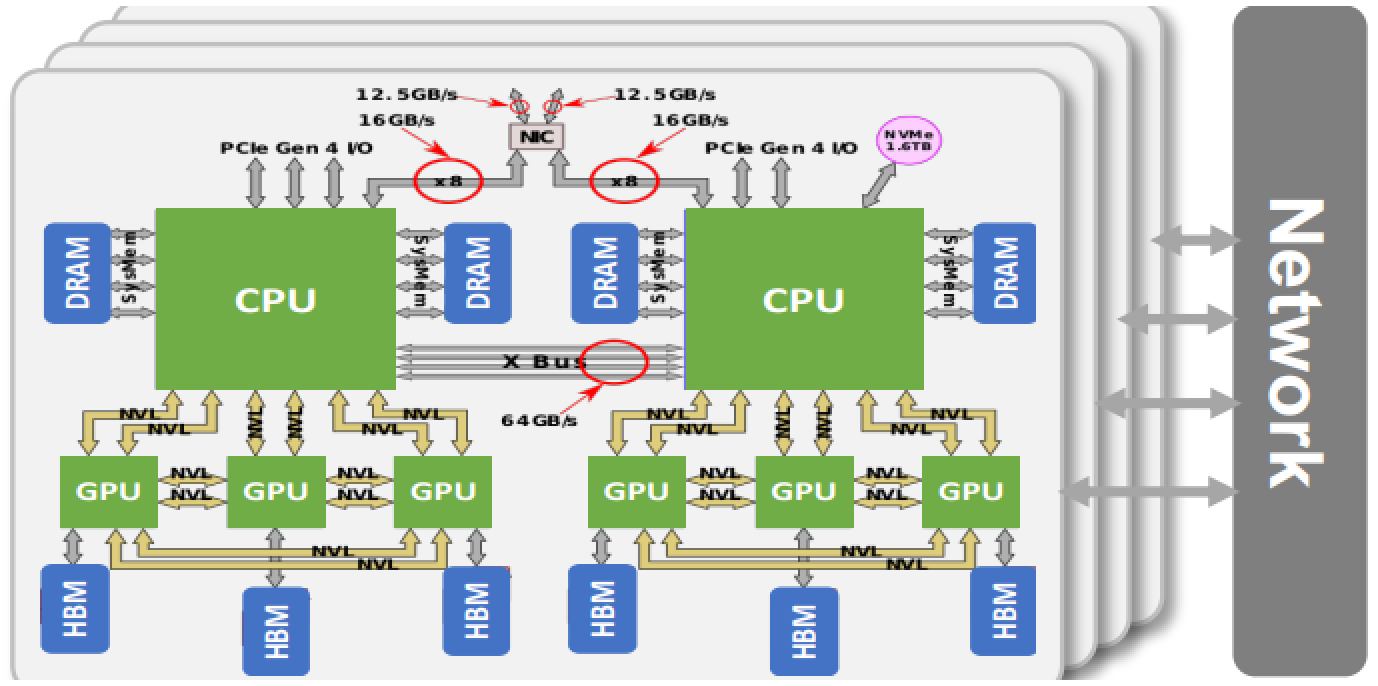
Autodiff

Graph Optimization

Parallelization

Runtime: schedule / memory

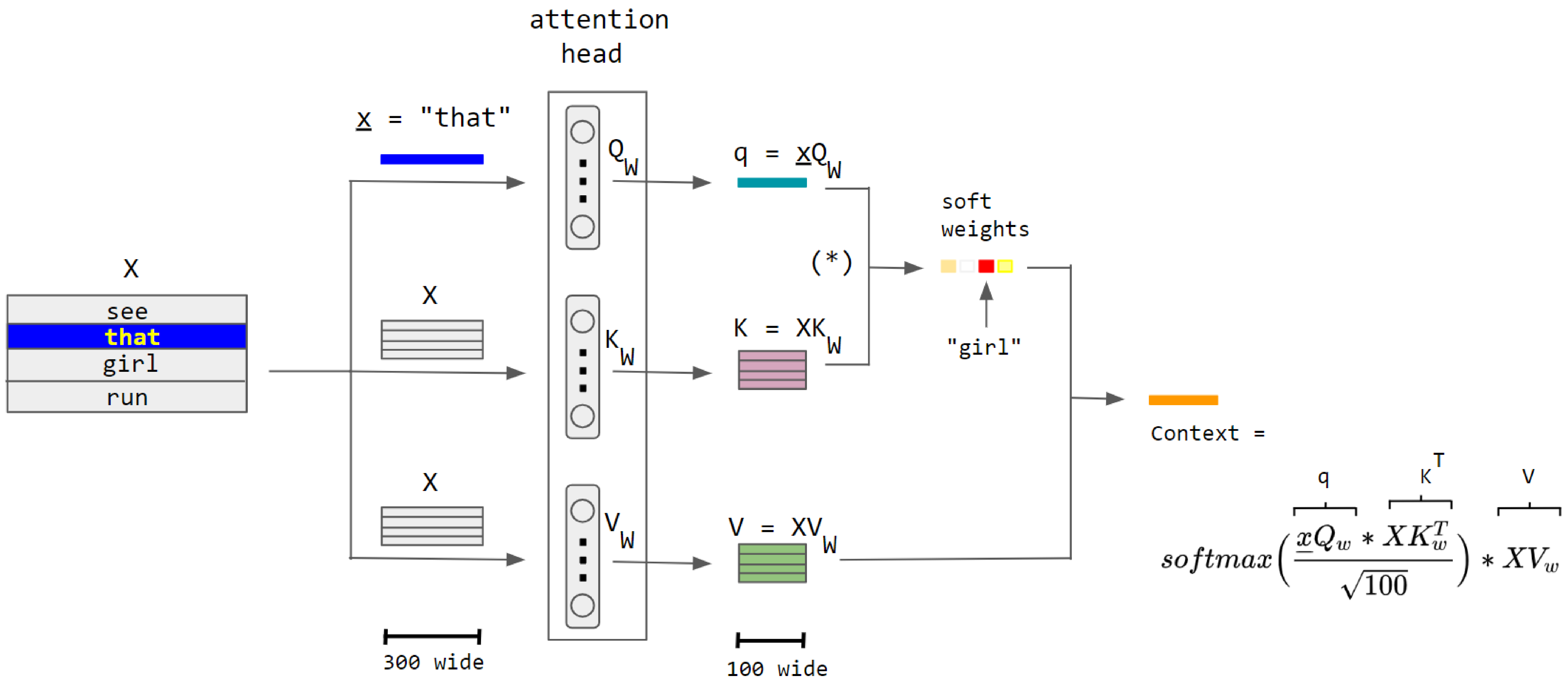
Operator optimization/compilation



# Graph Optimization

- Goal:
  - Rewrite the original Graph G to G'
  - G' runs faster than G

# Motivating Example: Attention




$$softmax\left(\frac{xQ_w * XK_w^T}{\sqrt{100}}\right) * XV_w$$

```
# Original
Q = matmul(W_q, h)
K = matmul(W_k, h)
V = matmul(W_v, h)

# Merged QKV
QKV = matmul(concat(W_q, W_k, W_v), h)
```

- Why merged QKV is faster?

# Arithmetic Intensity

$$AI = \#ops / \#bytes$$

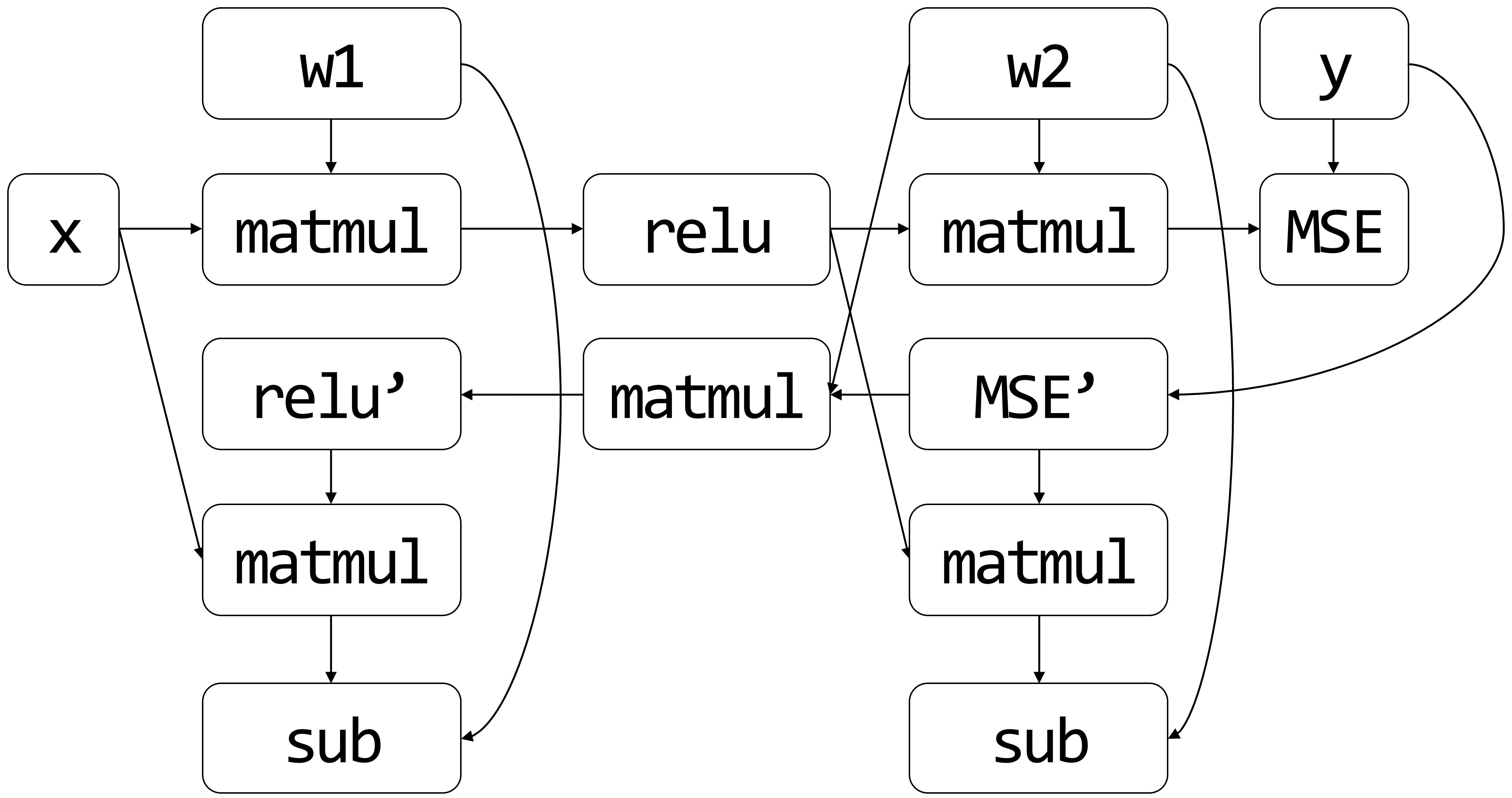
# How to perform graph optimization?

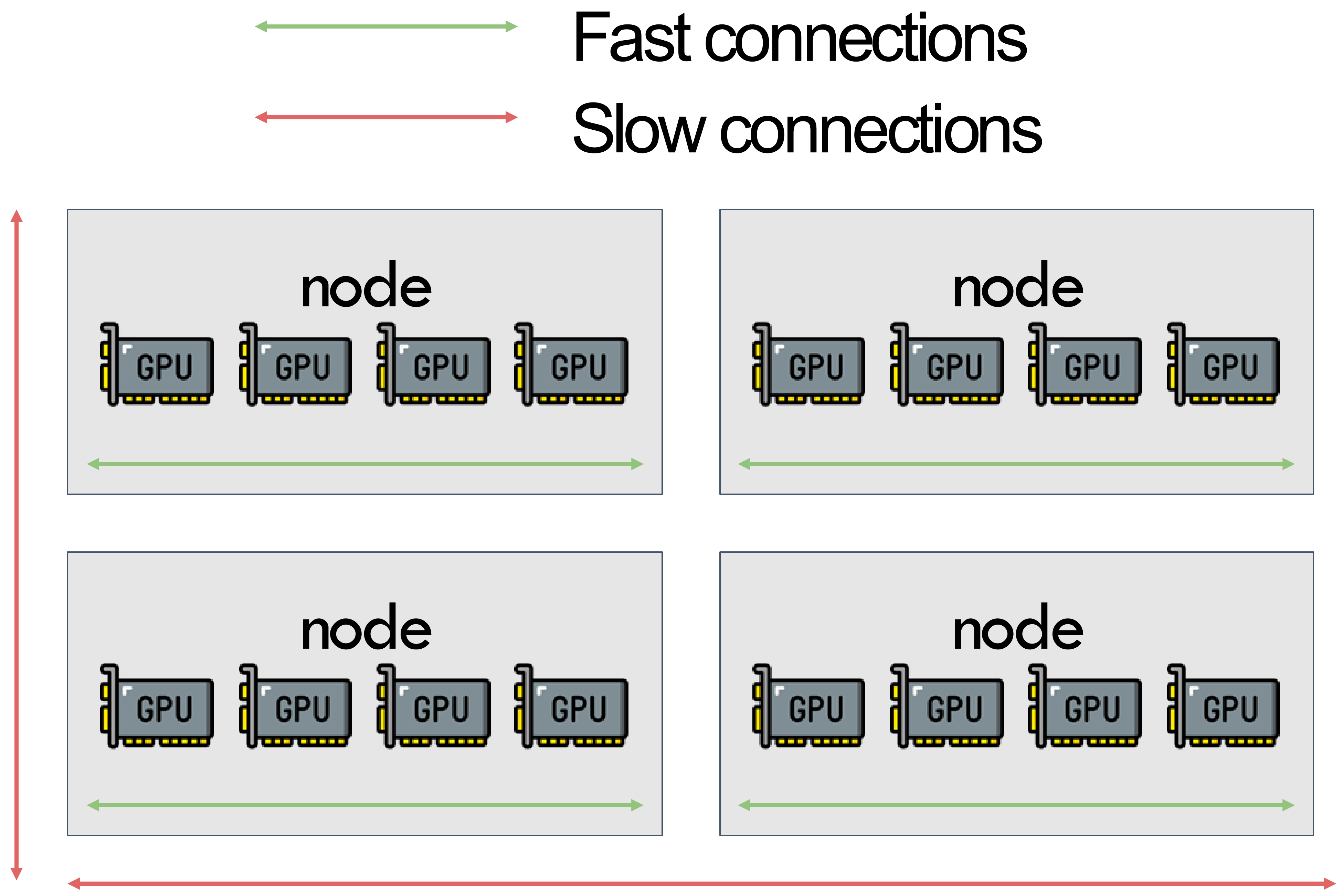
- Writing rules / template
- Auto discovery

# Parallelization

- Goal: parallelize the graph compute over multiple devices

How to partition the computational graph on the device cluster?



# Parallelization Problems

- How to partition
- How to communicate
- How to schedule
- Consistency
- How to auto-parallelize?

# Runtime and Scheduling

- Goal: schedule the compute/communication/memory in a way that
  - As fast as possible
  - Overlap communication with compute
  - Subject to memory constraints

# Operator Implementation

- Goal: get the fastest possible implementation of
  - Matmul
  - Conv2d?
- For different hardware: V100, A100, H100, phone, TPU
- For different precision: fp32, fp16, fp8, fp4
- For different shape: conv2d_3x3, conv2d_5x5, matmul2D, 3D, attention

# High-level Picture

Data

Model

Compute

✅ $\{x_i\}^n{}_{i=1}$

✅ Math primitives (mostly matmul)

❓Make them run on (clusters of ) different kinds of hardware

✅A repr that expresses the computation using primitives