https://haoailab.com/cse291-s26/

CSE/DSC 291: Deep Learning Systems
Spring 2026

LLM, diffusion, and case studies

Optimizations and Parallelization

Basics

Enrollment Update

Current enrollment:

- DSC: 70 (was 65)
- CSE: 88 (was 86)

- We are enrolling 20 more students by this week (based on FCFS)

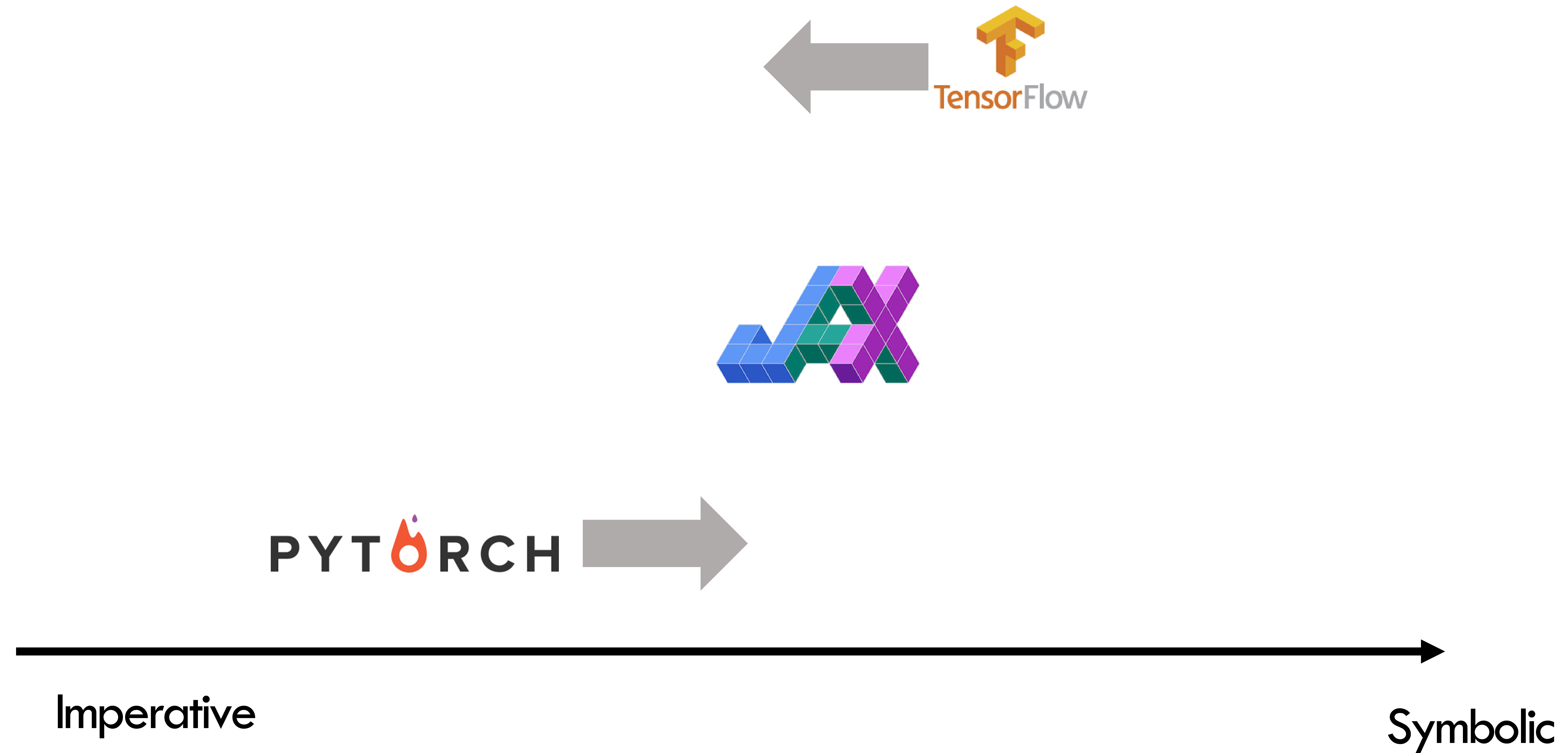
- For CSE Student: once we approve your enrollment on our side, CSE will grant you enrollment permission (maybe w/ a notification email), and you need to go ahead and manually enroll in the enrollment system.

- We will likely accommodate 180 students

Logistics

- PA1 was released, please start early
- Scribe duty: please sign up to make sure each lecture has equal # of scribers (TAs will adjust if you don't)

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?

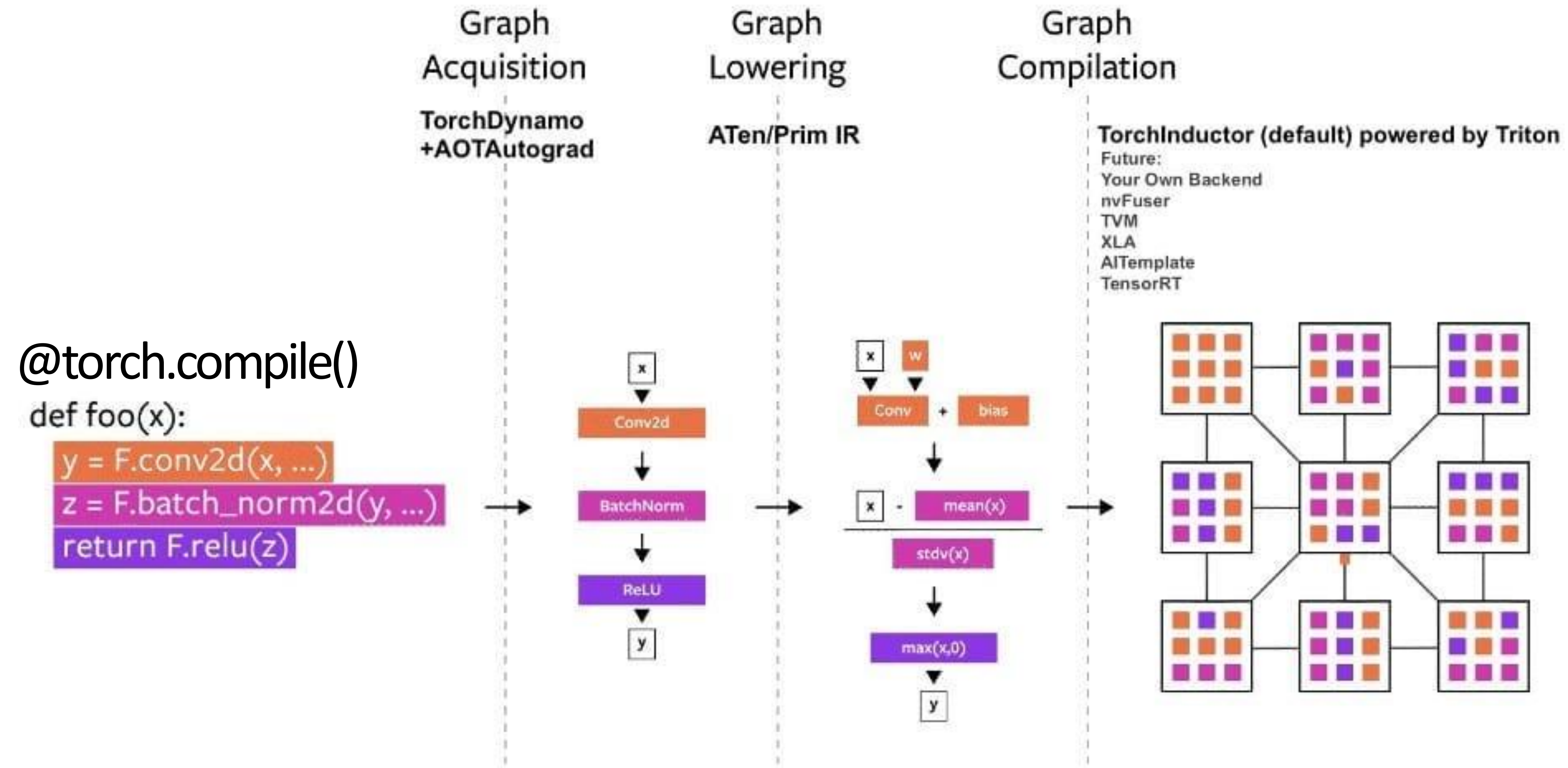
```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

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?

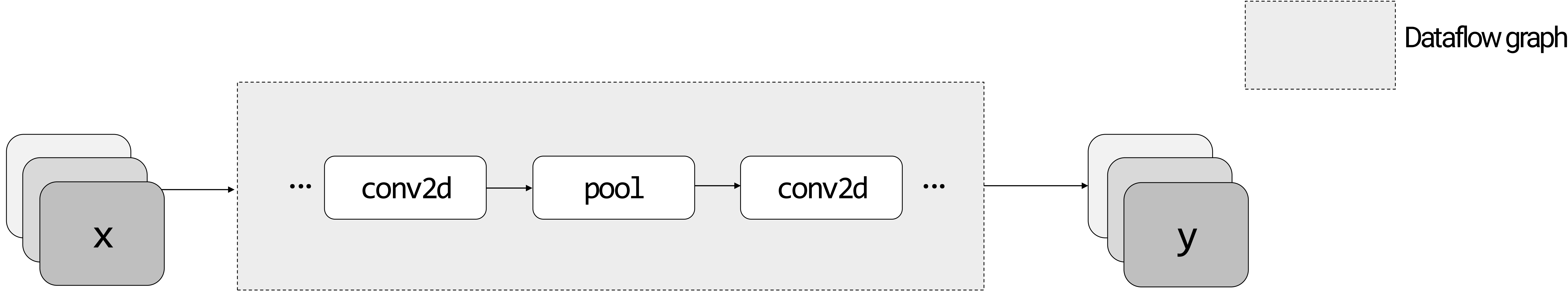
Requirements for static graphs

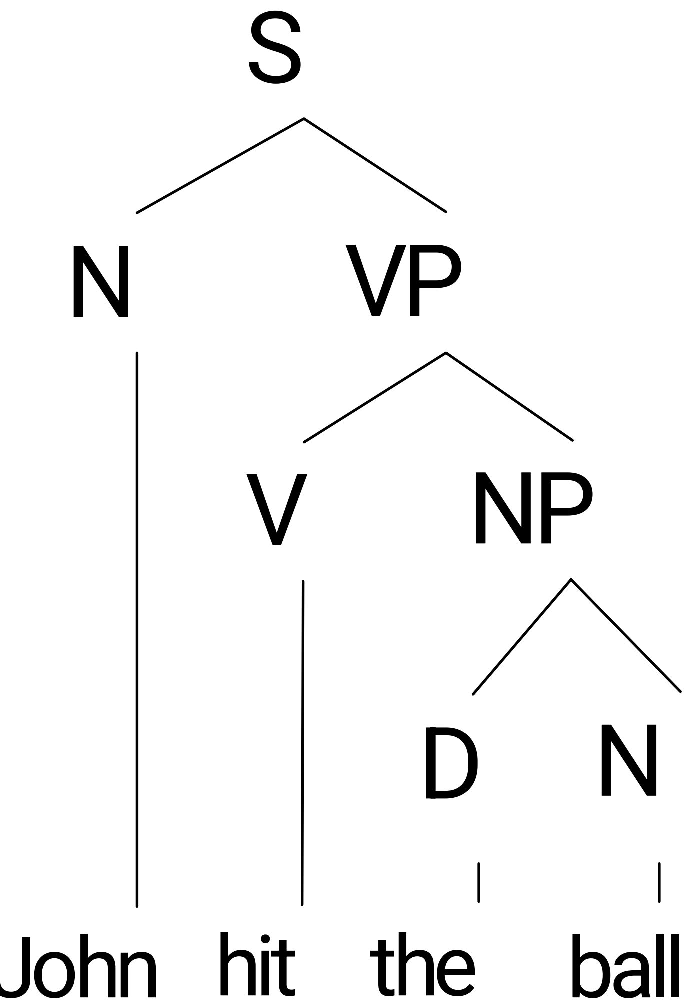
Q: What is the problem of JIT?

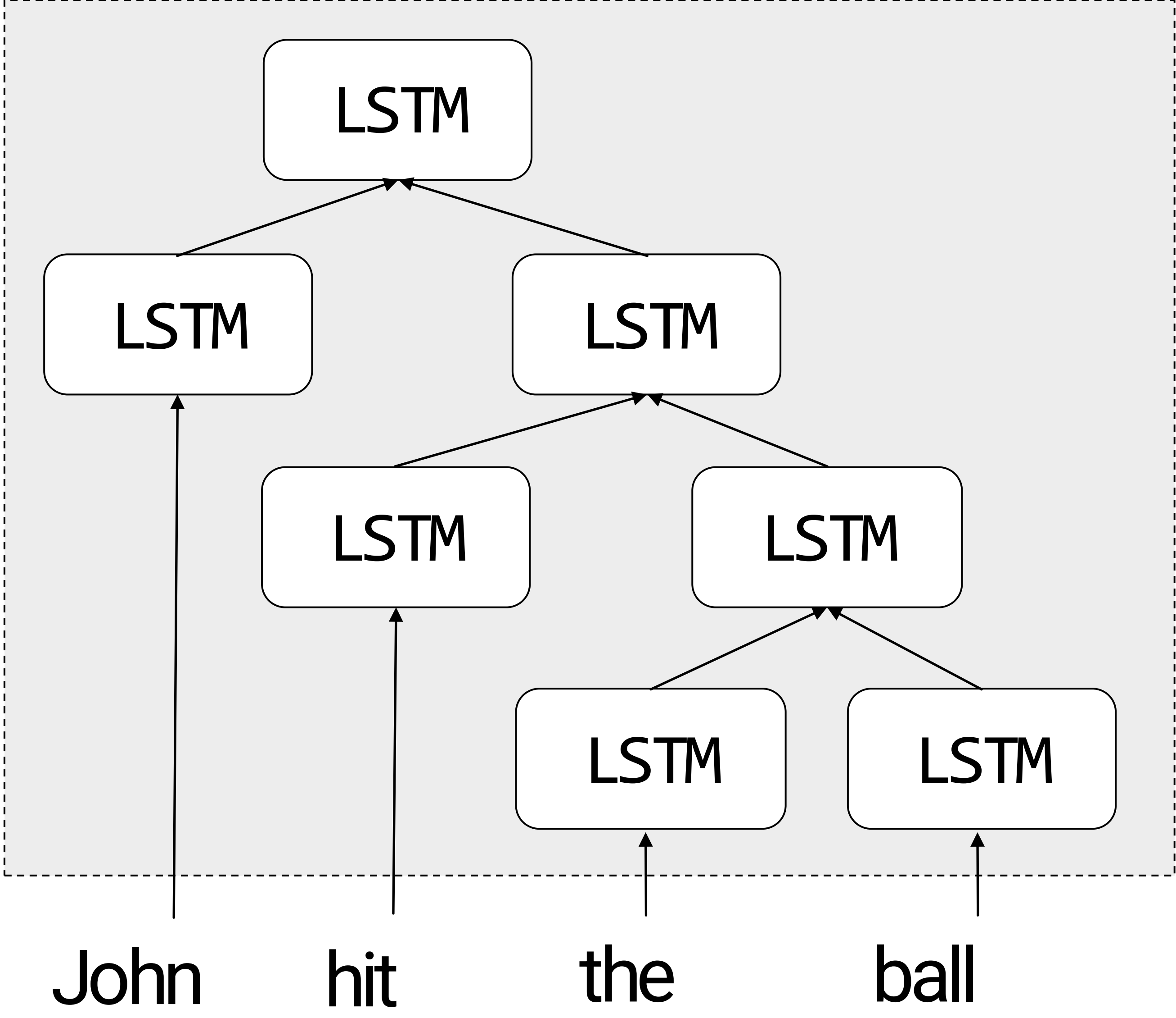
A: Requirements for static graphs

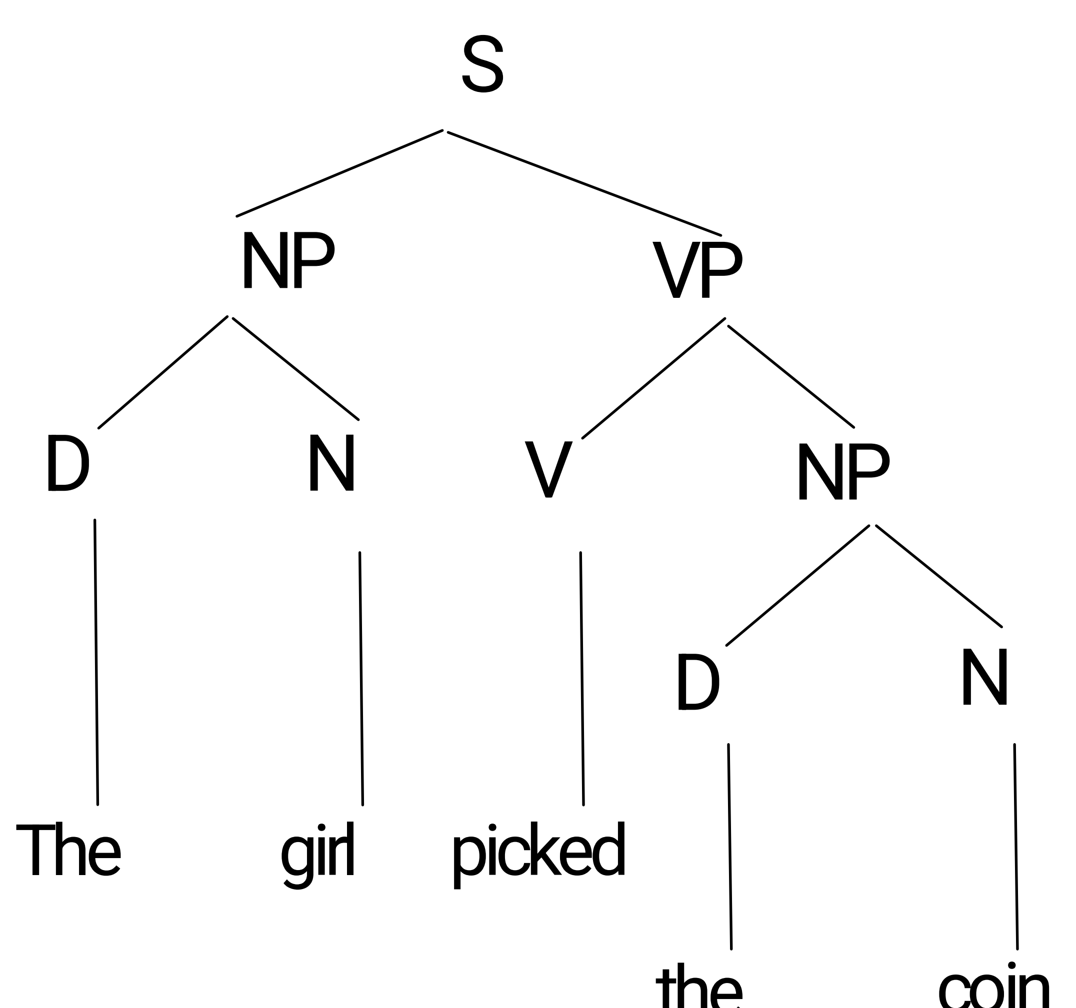
Static Models vs. Dynamic Models

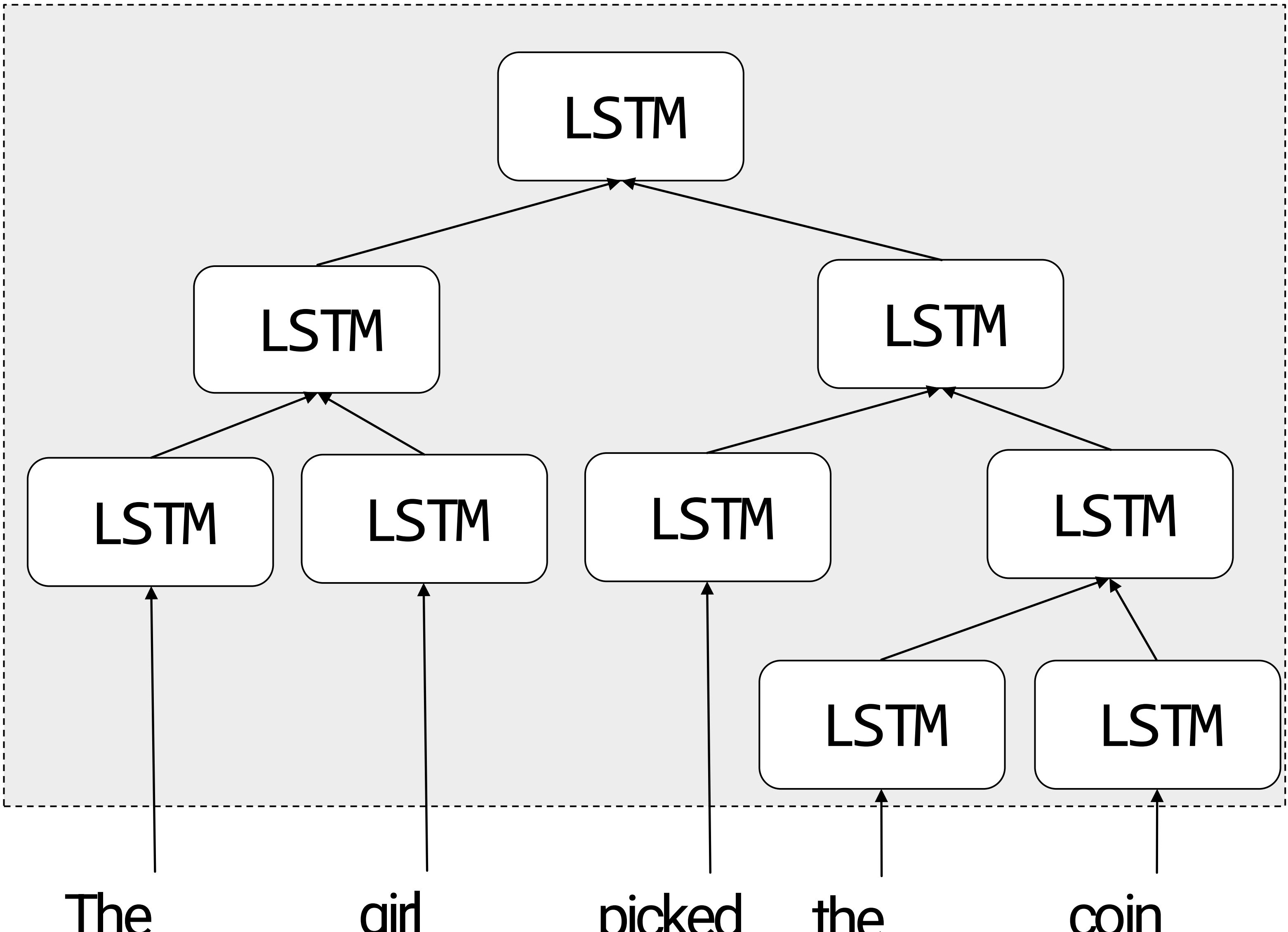
Dataflow graph











Static vs. Dynamic Dataflow Graphs

- Static Dataflow graphs
  - Define once, optimized once, execute many times
  - Execution: Once defined, all following computation will **follow** the defined computation

Static vs. Dynamic Dataflow Graphs

- Dynamic Dataflow Graphs
  - Difficulty in expressing complex flow-control logic
  - Complexity of the computation graph implementation
  - Difficulty in debugging

- Is LLM static or dynamic?

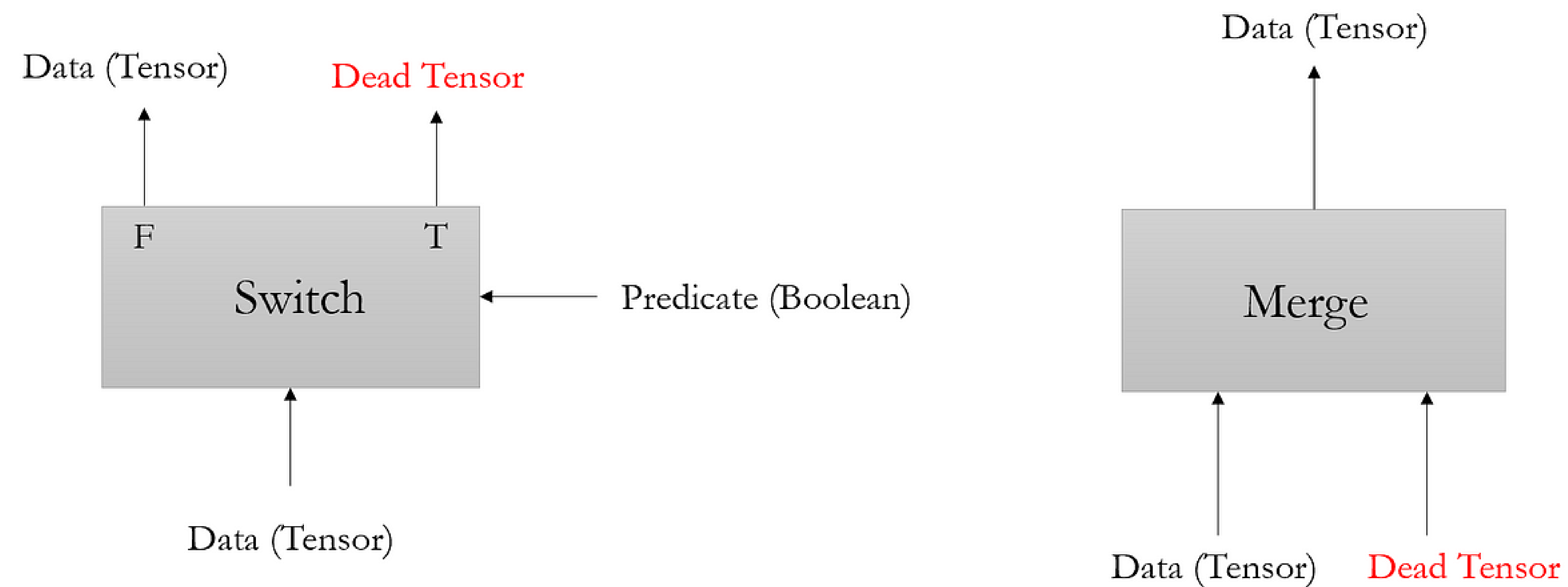
Open Research: How to Handle Dynamics?

Three ways:

- Just do Define-and-run and forget about JIT
  - As long as you do not care about performance…
- Introduce Control flow Ops
- Piecewise compilation and guards

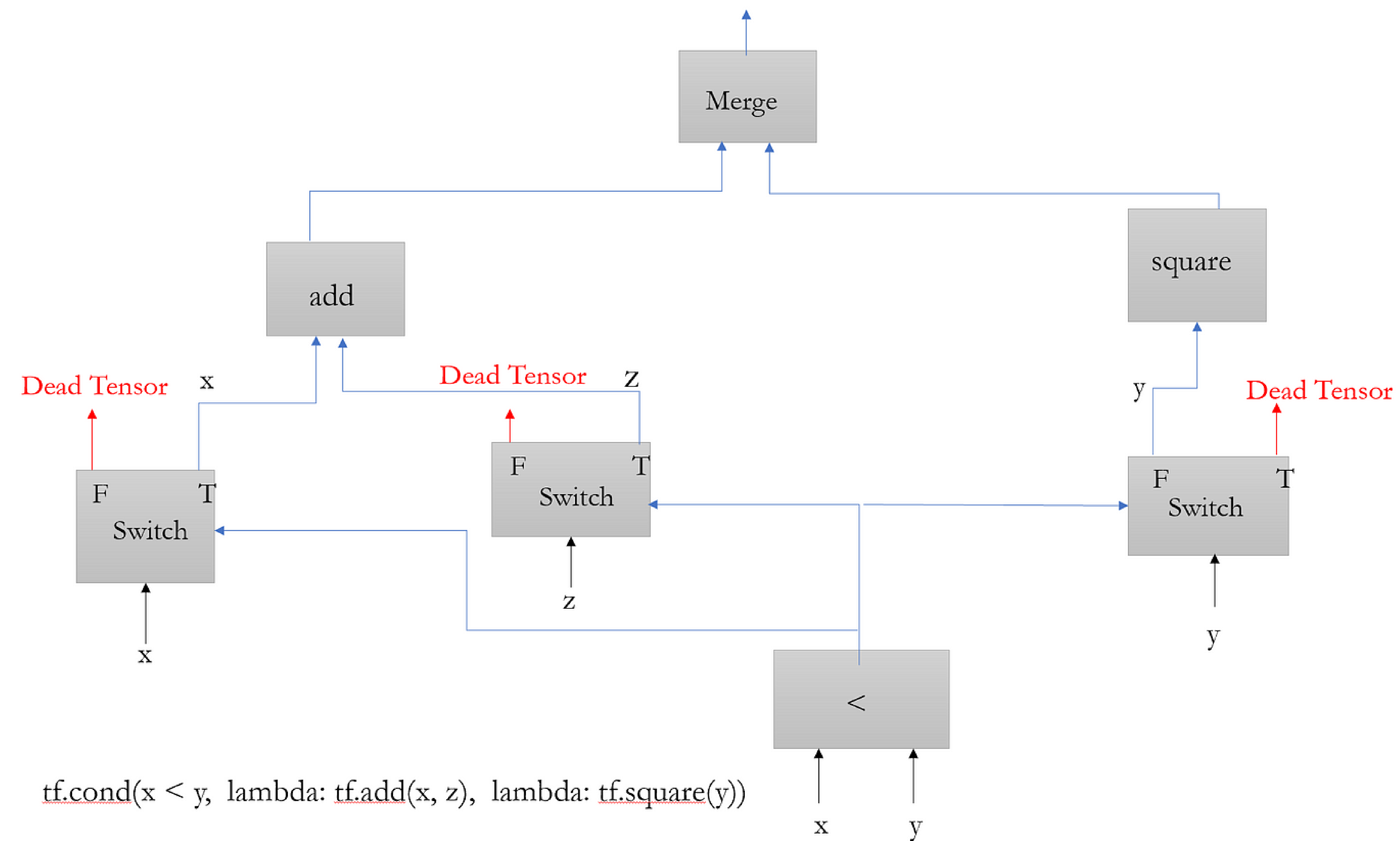
Control flow primitives

- Example primitive: Switch and Merge



Control flow primitives

- Example compute: tf.cond(x < y, lambda: tf.add(x, z), lambda: tf.square(y))



Control flow primitives

Control flow is natural idea in all PLs:

- if…then…,
- for,
- while

What is the potential problem of using control flow in dataflow graphs?

Piecewise Compilation

- Case 1: a graph accepting input shapes of [x, c1, c2]
  - c1, c2: constants
  - x: variable
  - Q: how to statically JIT this graph?

- Case 2: a graph with is static, then dynamic, then static.
  - Q: how to statically JIT this graph?

Summary

- We summarized our workload
  - Matmul + softmax + …
- Computational graphs
  - Nodes, edges
- Programming
  - Imperative vs. symbolic
  - Static vs. dynamic
  - JIT and its bottleneck

MCQ Time

You are a machine learning engineer at a company that is providing LLM endpoints to users. Your goal is running efficient inference for these LLMs. You are given a framework which has both symbolic and imperative APIs. While designing your system, would you:

A. Use symbolic mode for both testing and deployment of your system.
B. Use imperative mode for development and symbolic mode for deployment.
C. Use symbolic mode for development and imperative mode for deployment.
D. Use imperative mode for both testing and deployment of your system.

MCQ Time

Which of the following is not true about dataflow graphs?

A. Static dataflow graphs are defined once and executed many times

B. No extra effort is required for *batching* optimization of static dataflow graphs

C. Dynamic dataflow graphs are easy to debug

D. Define-and-run is a possible way to handle dynamic dataflow graphs

Where we are

Data

Model

Compute

❓ $\{x_i\}^n{}_{i=1}$

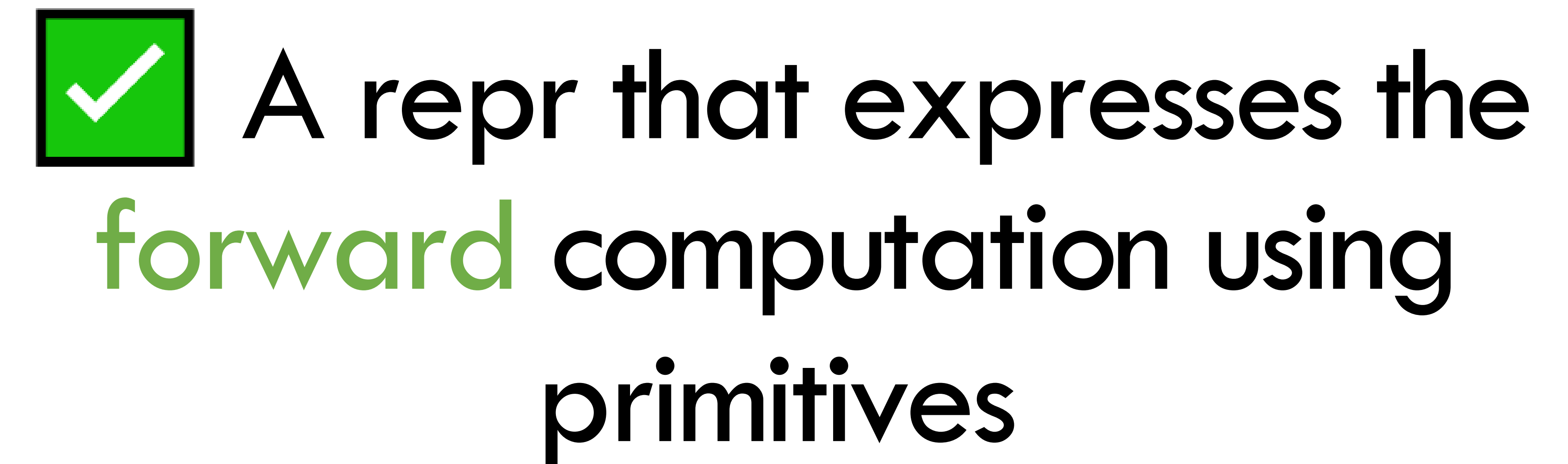
Math primitives (mostly matmul)

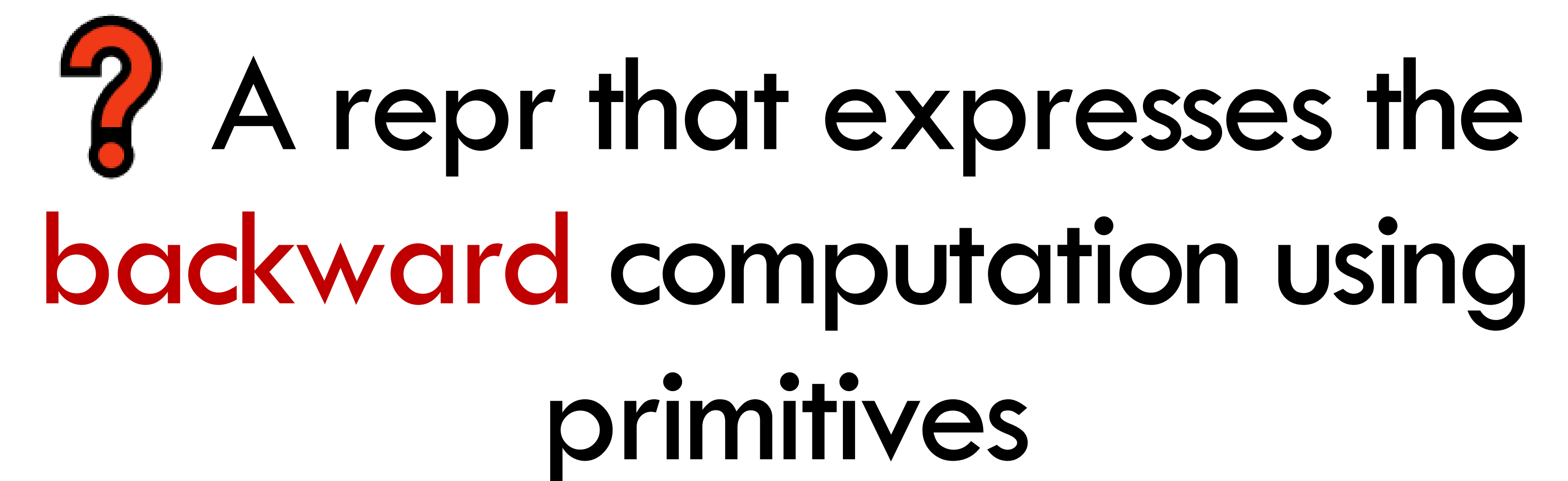
❓Make them run on (clusters of) different kinds of hardware

❓A repr that expresses the computation using primitives

Where we are

A repr that expresses the computation using primitives





Today's learning goals

- Autodiff
- MLSys architecture overview
  - Optimization opportunities
- Operator optimization: kick-starter

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

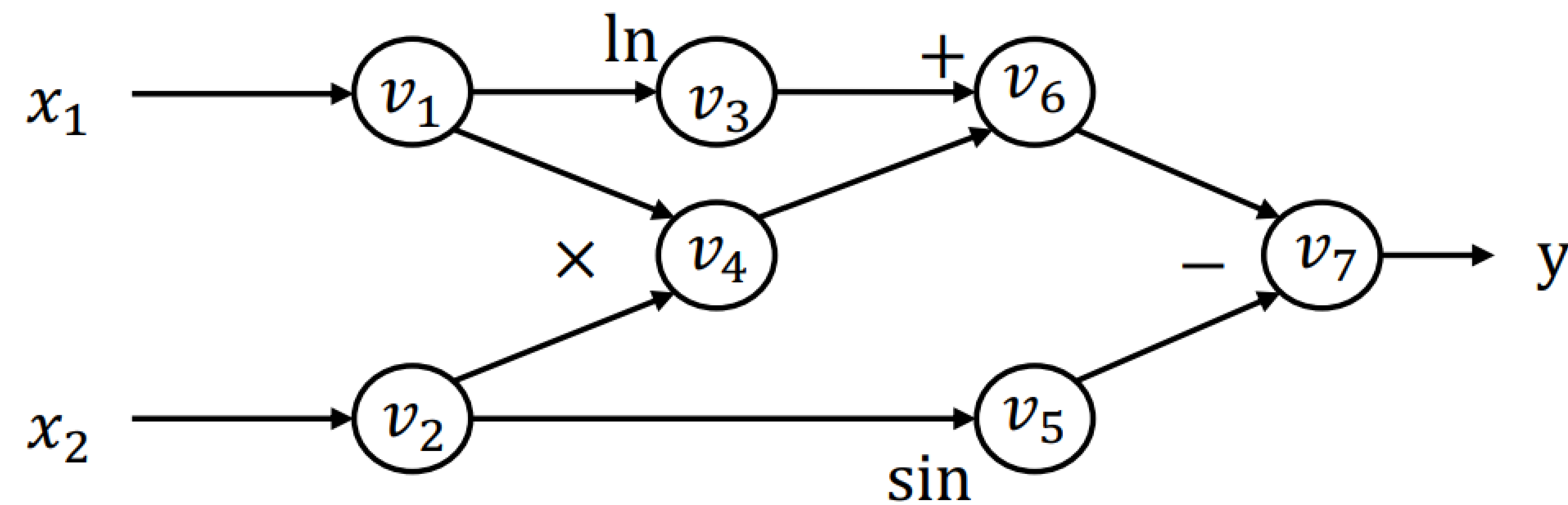
$$\frac{\partial(f(\theta) + g(\theta))}{\partial\theta} = \frac{\partial f(\theta)}{\partial\theta} + \frac{\partial g(\theta)}{\partial\theta}$$

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

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

Map autodiff rules to computational graph

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

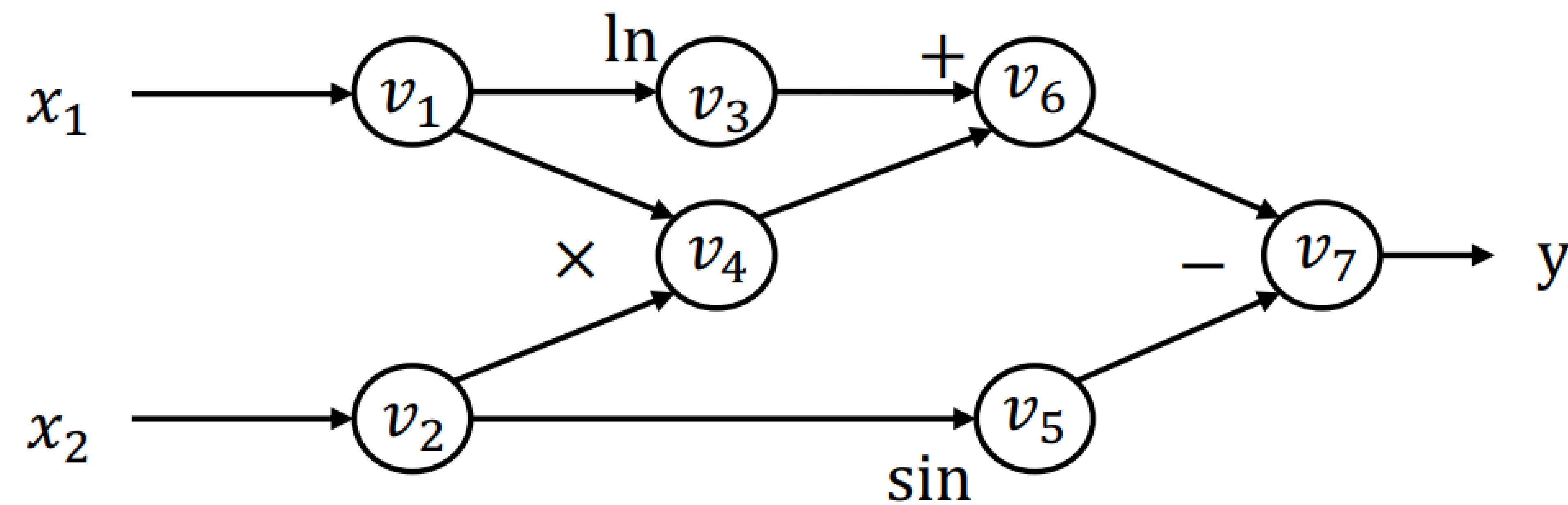


Forward evaluation trace

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

- 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



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

$$
\begin{aligned}
\dot{v_1} &= 1 \\
\dot{v_2} &= 0 \\
\dot{v_3} &= \dot{v_1}/v_1 = 0.5 \\
\dot{v_4} &= \dot{v_1} v_2 + \dot{v_2} v_1 = 1 \times 5 + 0 \times 2 = 5 \\
\dot{v_5} &= \dot{v_2} \cos v_2 = 0 \times \cos 5 = 0 \\
\dot{v_6} &= \dot{v_3} + \dot{v_4} = 0.5 + 5 = 5.5 \\
\dot{v_7} &= \dot{v_6} - \dot{v_5} = 5.5 - 0 = 5.5
\end{aligned}
$$

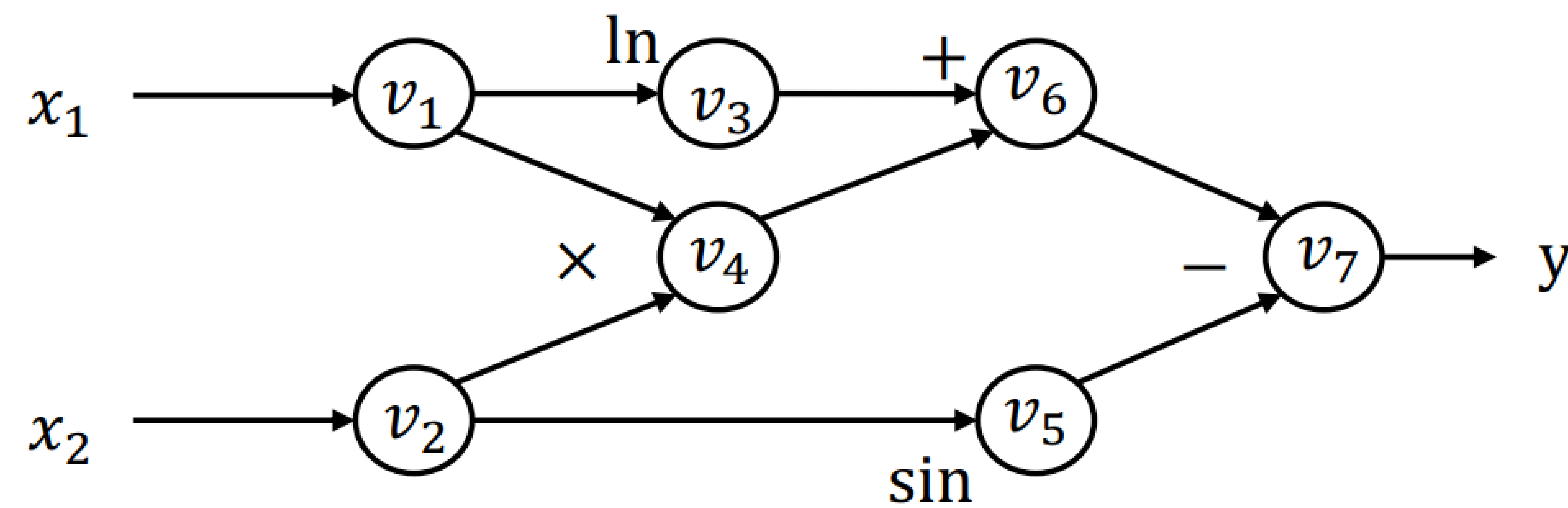
- Finally: $\frac{\partial y}{\partial x_1} = \dot{v_7} = 5.5$

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

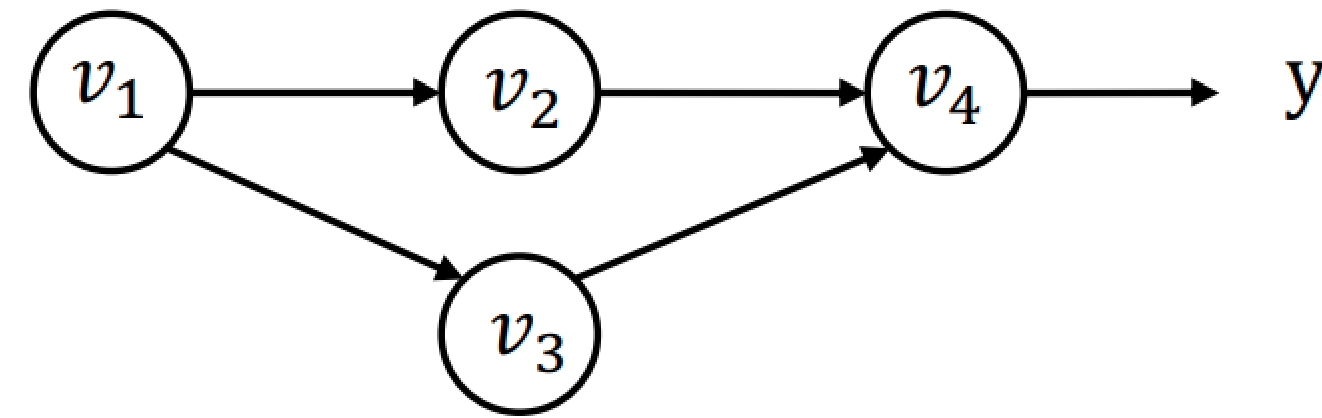
$$v_1 = x_1 = 2$$
$$v_2 = x_2 = 5$$
$$v_3 = \ln v_1 = \ln 2 = 0.693$$
$$v_4 = v_1 \times v_2 = 10$$
$$v_5 = \sin v_2 = \sin 5 = -0.959$$
$$v_6 = v_3 + v_4 = 10.693$$
$$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$$
$$y = v_7 = 11.652$$

- 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:

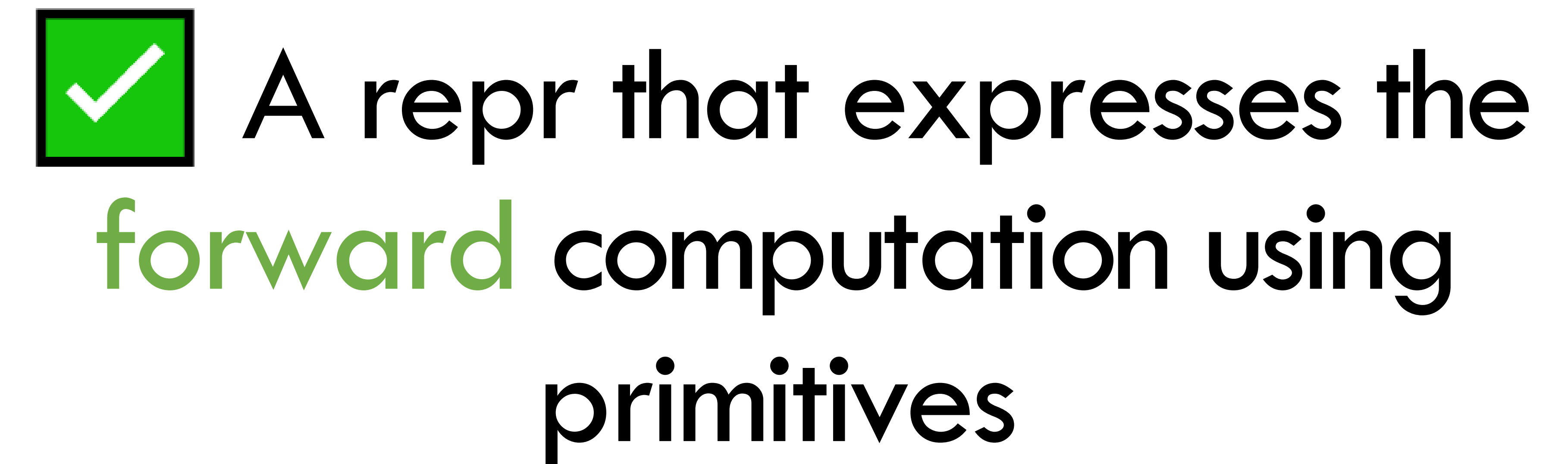
$$\overline{v_i} = \sum_{j \in next(i)} \overline{v_{i\to j}} \quad \text{, where} \quad \overline{v_{i\to j}} = \overline{v_j}\frac{\partial v_j}{\partial v_i}$$

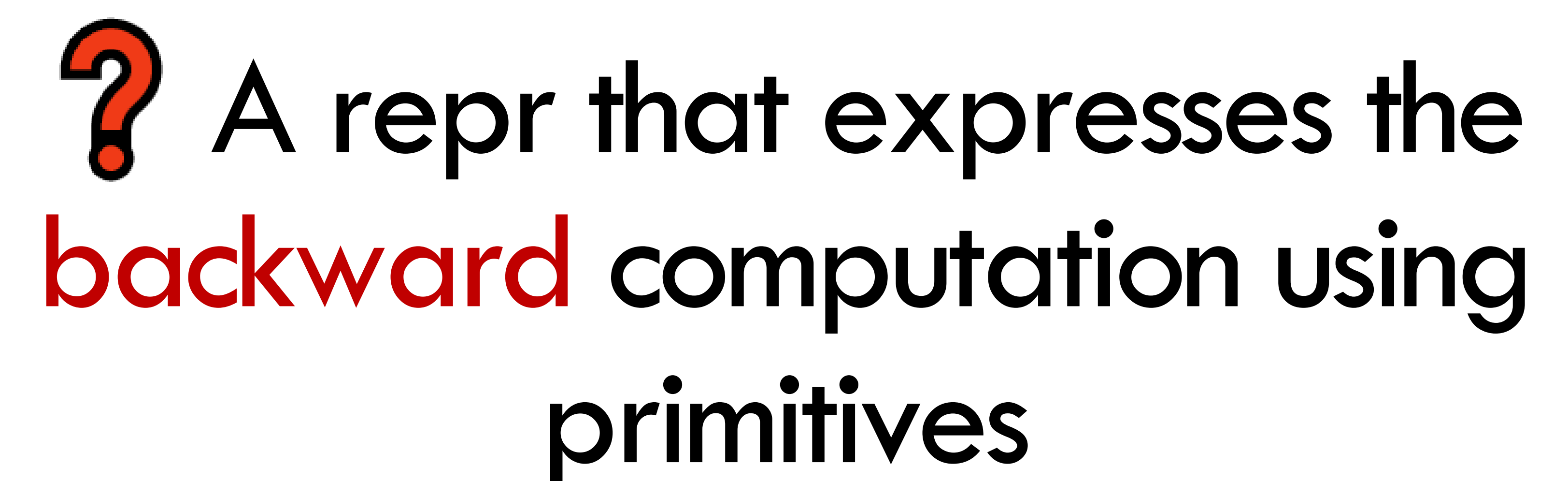
Summary: Backward Mode Autodiff

- 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





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
```

v_1, 1, exp v_2, $+$ v_3, $\times$ v_4

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):
    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
```

where $\overline{v_i} = \sum_j \overline{v_{i\to j}}$, $\overline{v_{k\to i}} = \overline{v_i}\,\frac{\partial v_i}{\partial v_k}$, and the returned value is $\overline{v_{input}}$.

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

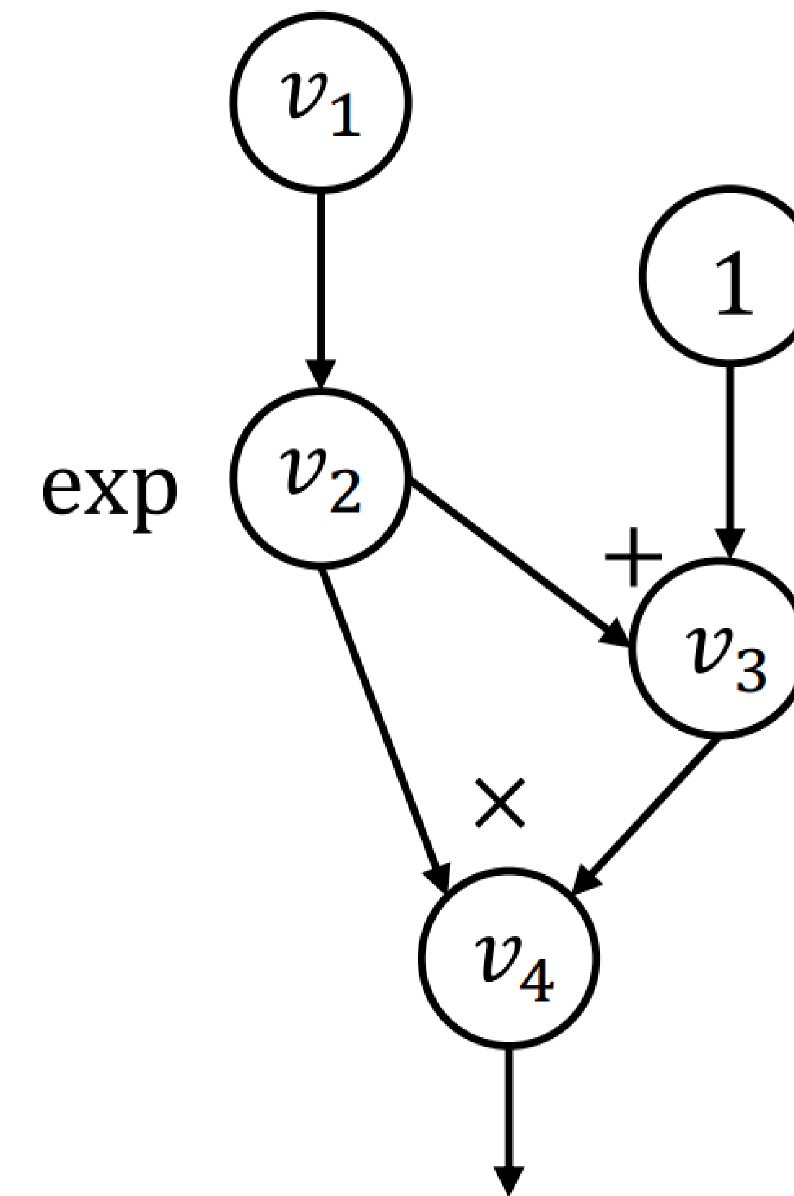
i = 4: $v_4 = sum([1]) = 1$

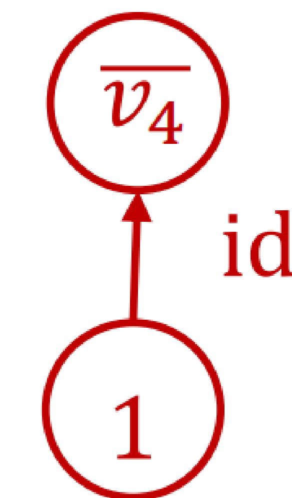
```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```

$\overline{v_i} = \sum_j \overline{v_{i \to j}}$ = sum(node_to_grad[i])

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

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





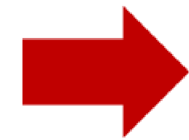
v_4: Inspect (v_2, v_4) and (v_3, v_4)

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

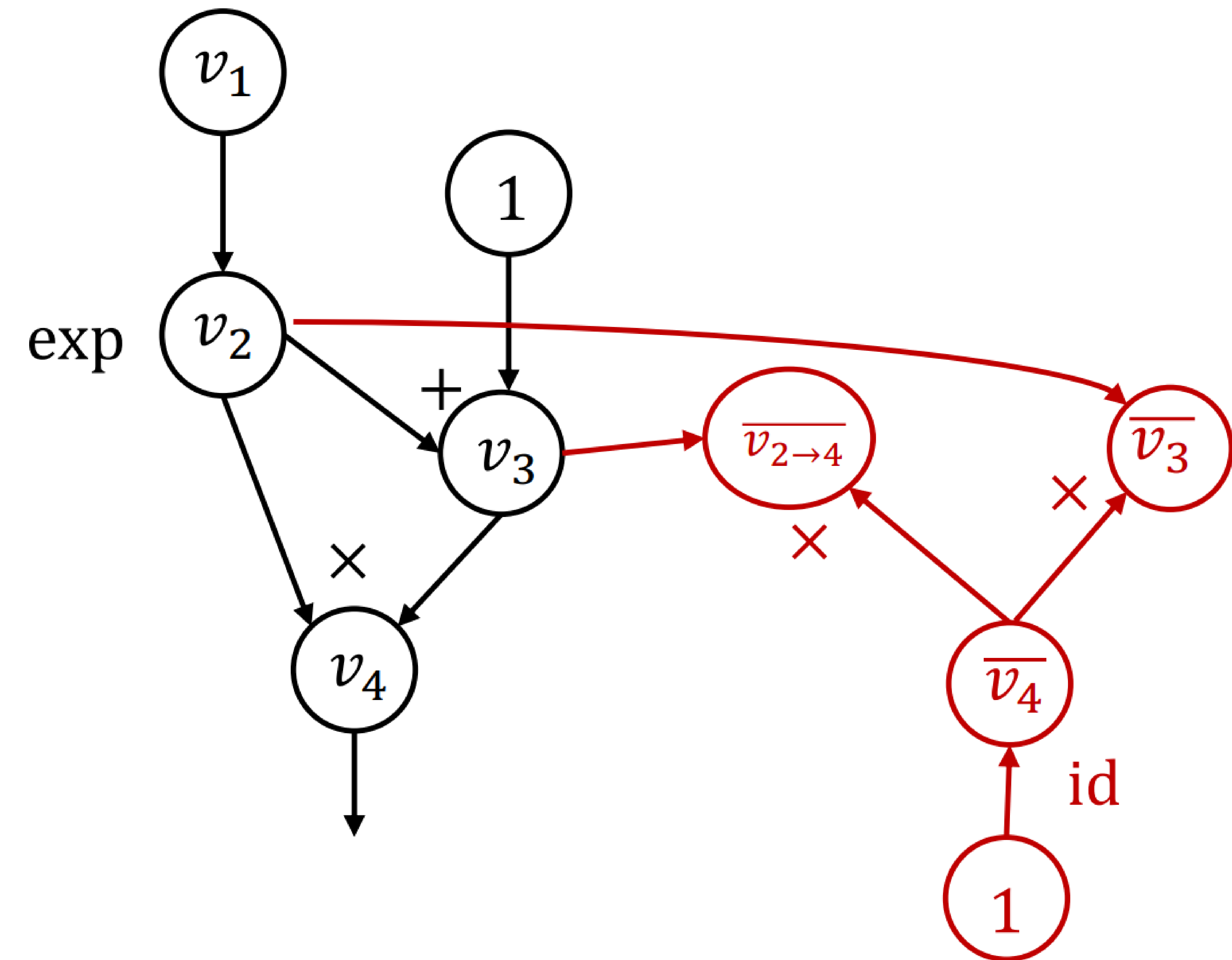
k=2: $\overline{v_{2\to4}} = \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_4} v_3$

k=3: $\overline{v_{3\to4}} = \overline{v_4}\frac{\partial v_4}{\partial v_3} = \overline{v_4} v_2, \overline{v_{3\to4}} = \overline{v_3}$

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

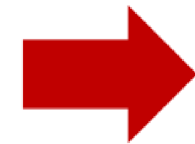


Inspect v_3

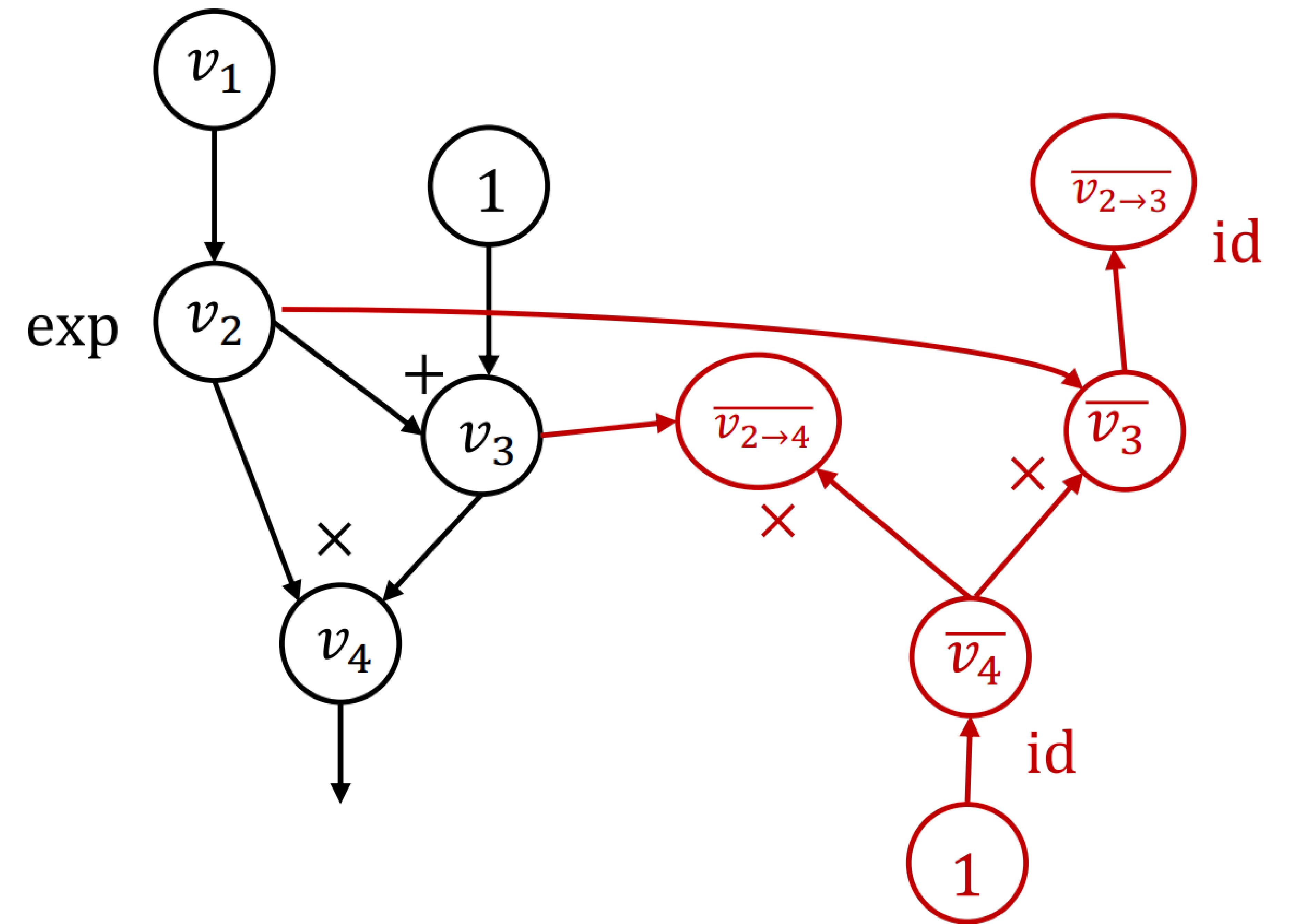
i=3: $\overline{v_3}$ done!

k=2: $\overline{v_{2\to 3}} = \overline{v_3}\frac{\partial v_3}{\partial v_2} = \overline{v_3}$

```
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: $[\overline{v_{2\to 4}}, \overline{v_{2\to 3}}]$
  3: $[\overline{v_3}]$
  4: $[\overline{v_4}]$
}

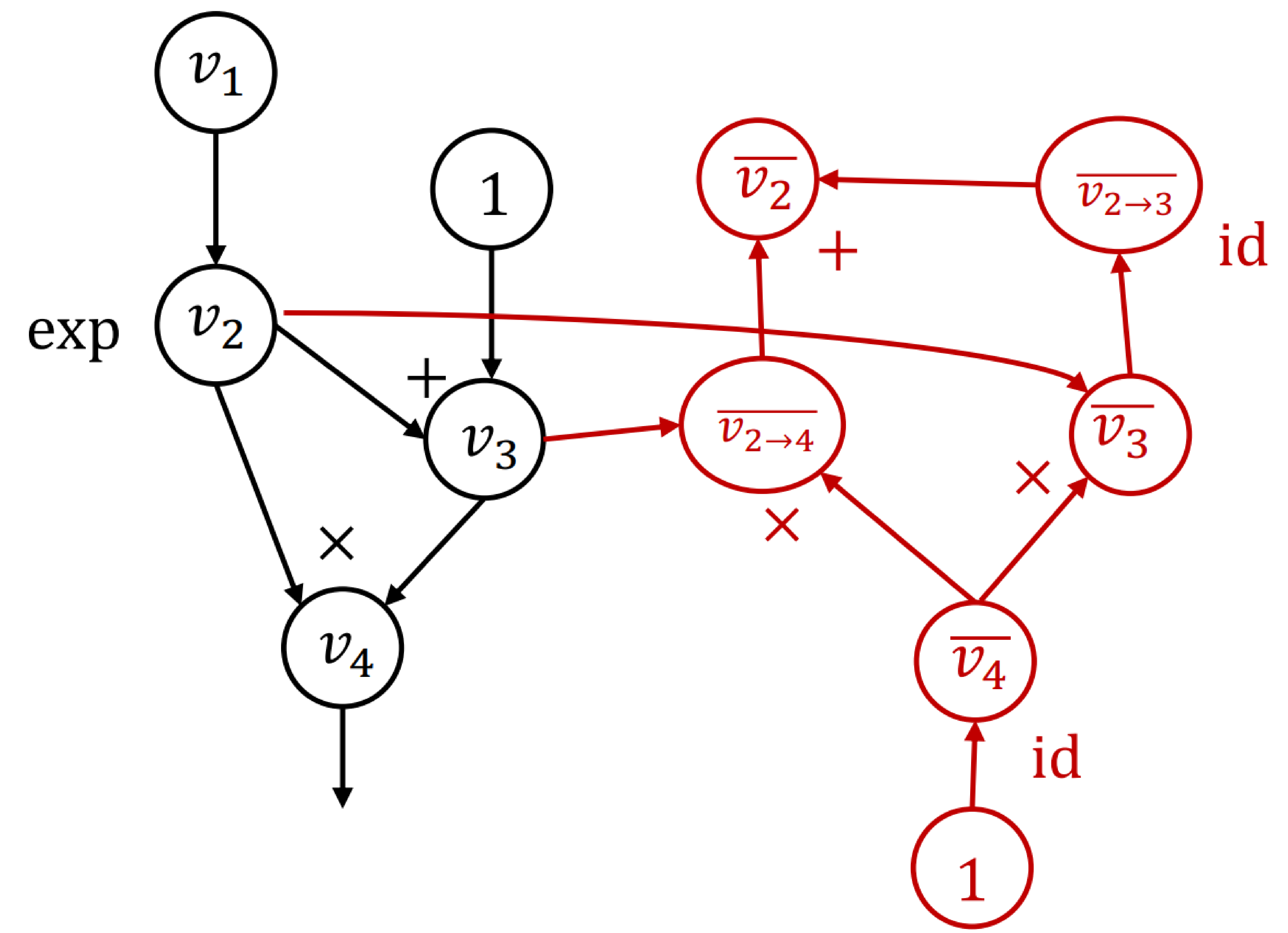


i=2: $\overline{v_2} = \overline{v_{2\to3}} + \overline{v_{2\to4}}$

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: $[\overline{v_4}]$
}



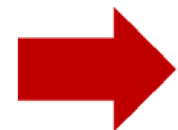
Inspect (v_1, v_2)

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

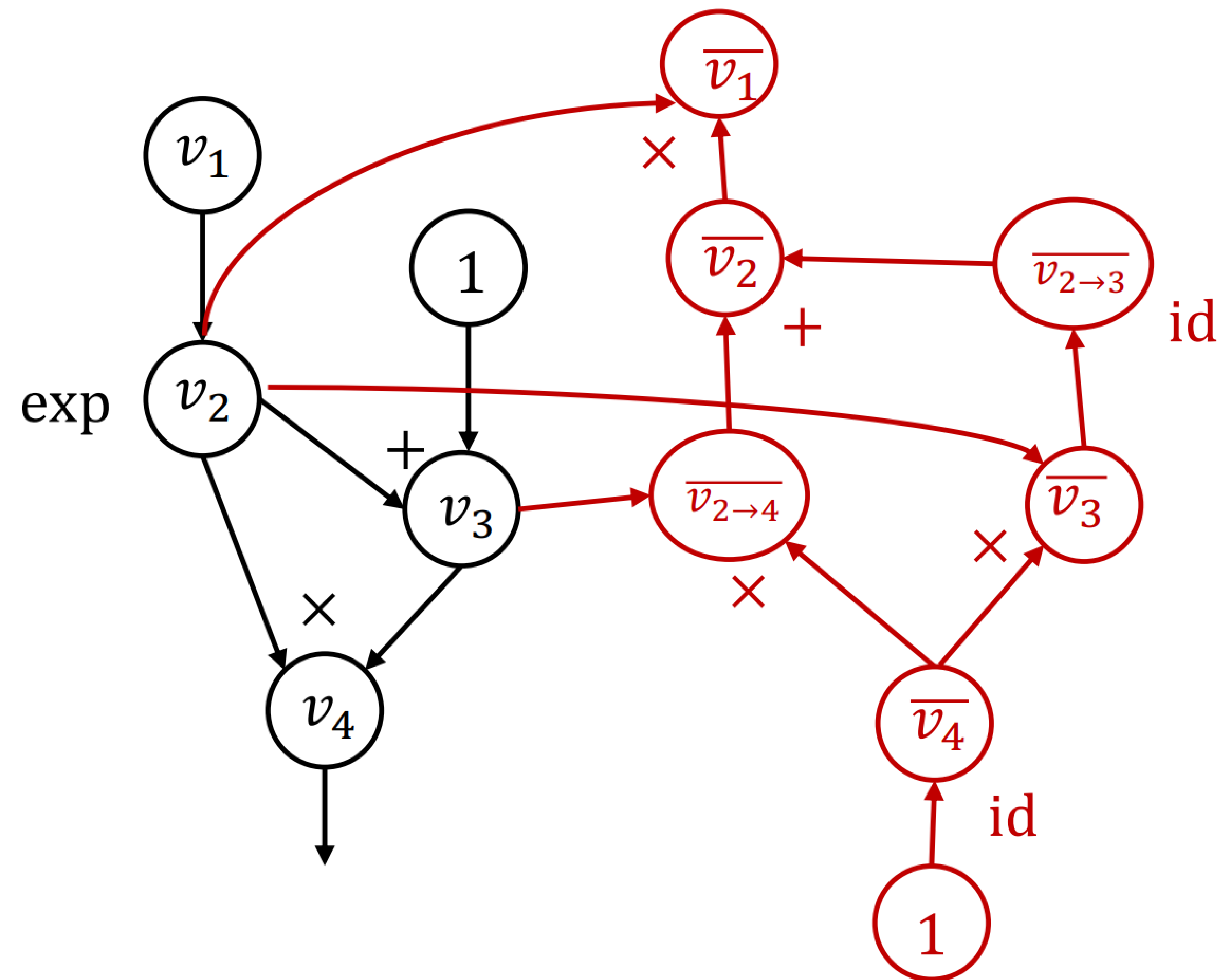
k=1: $\overline{v_{1\to2}} = \overline{v_2}\frac{\partial v_2}{\partial v_1} = \overline{v_2}\exp(v_1)$,

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

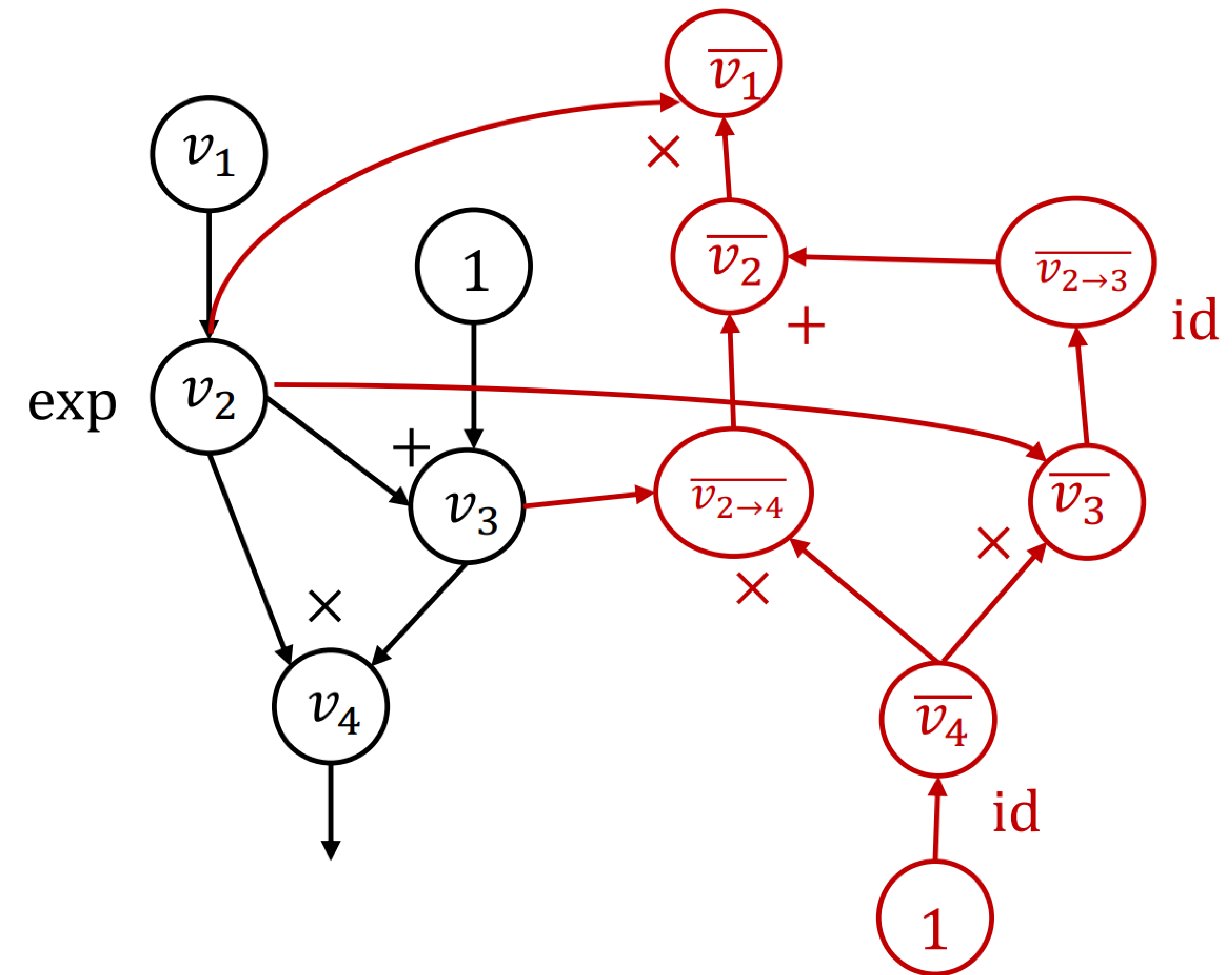
```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```



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

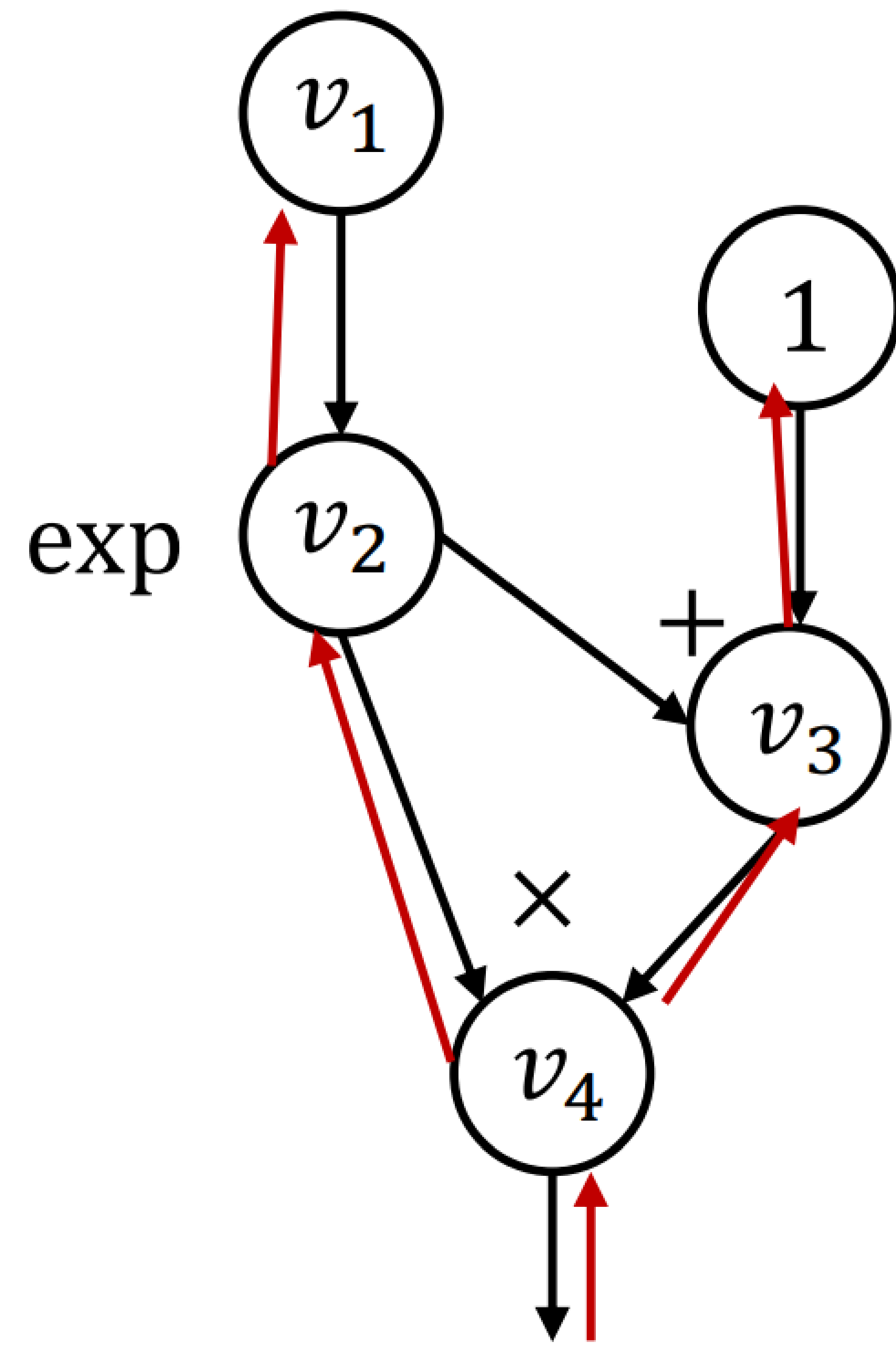


Summary: Backward AD

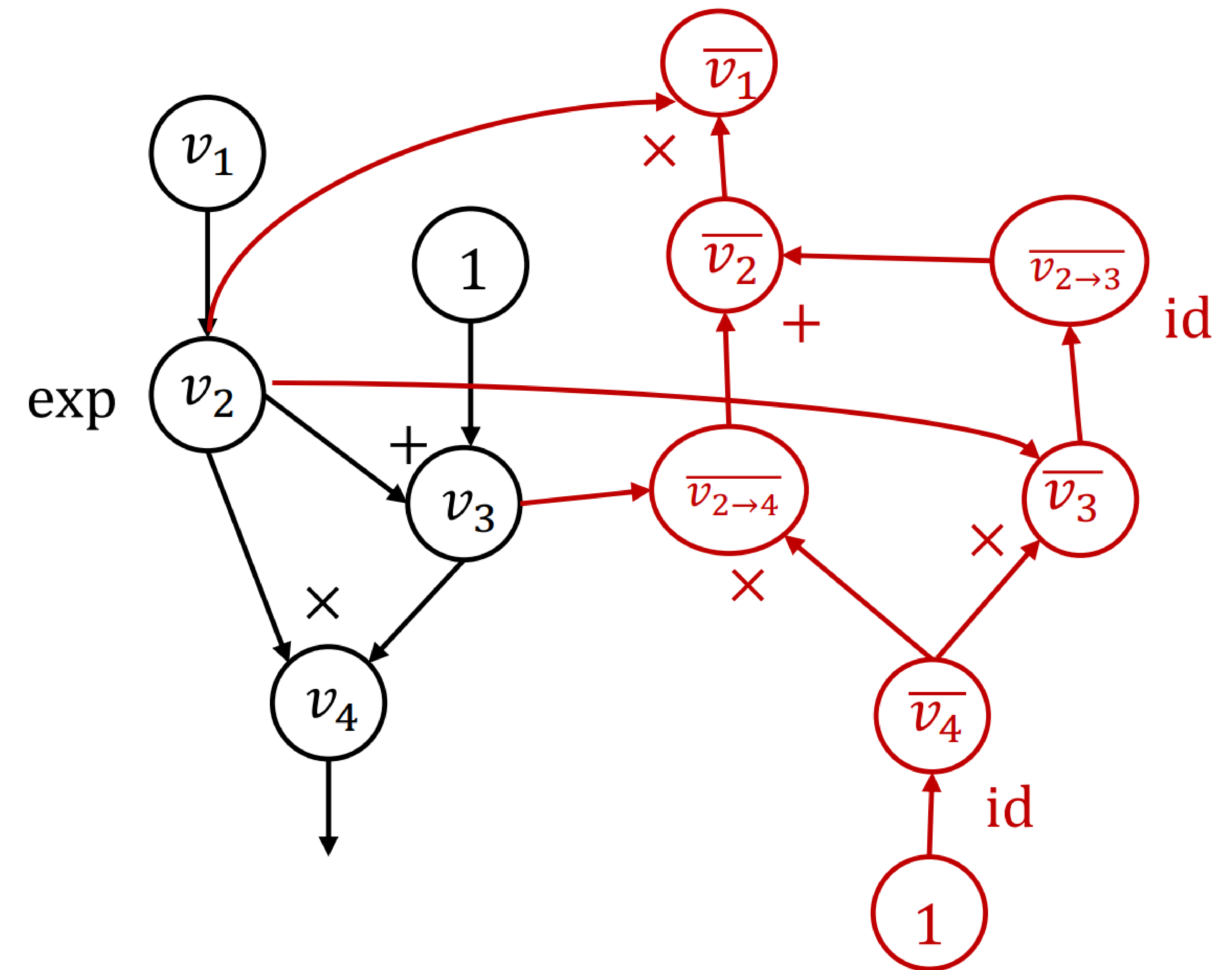


- Construct backward graph in a symbolic way (instead of concrete values)
- This graph can be reused by different input values

Backpropagation vs. Reverse-mode AD



vs.

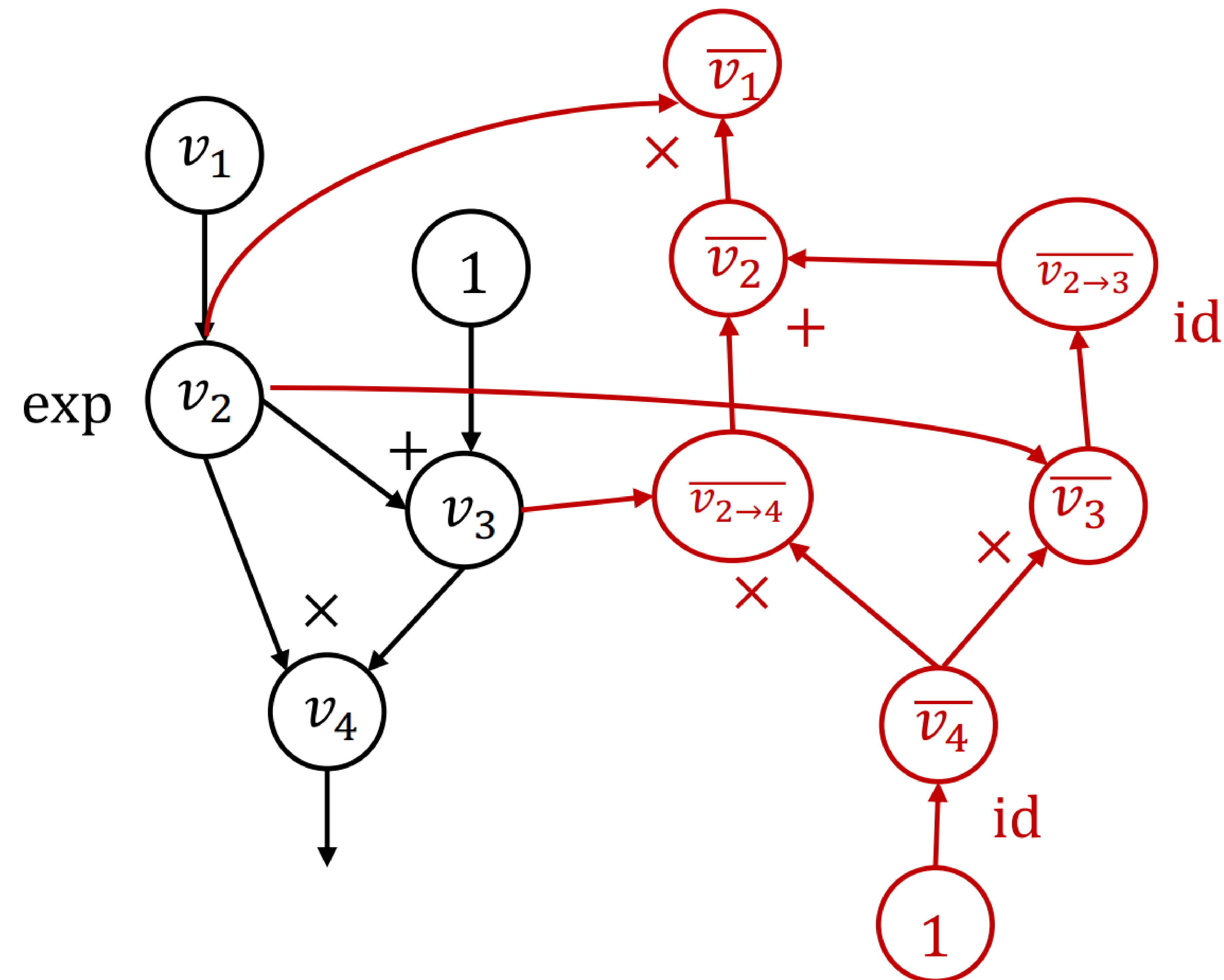


- Run backward through the forward graph
- Caffe/cuda-convnet

- 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

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

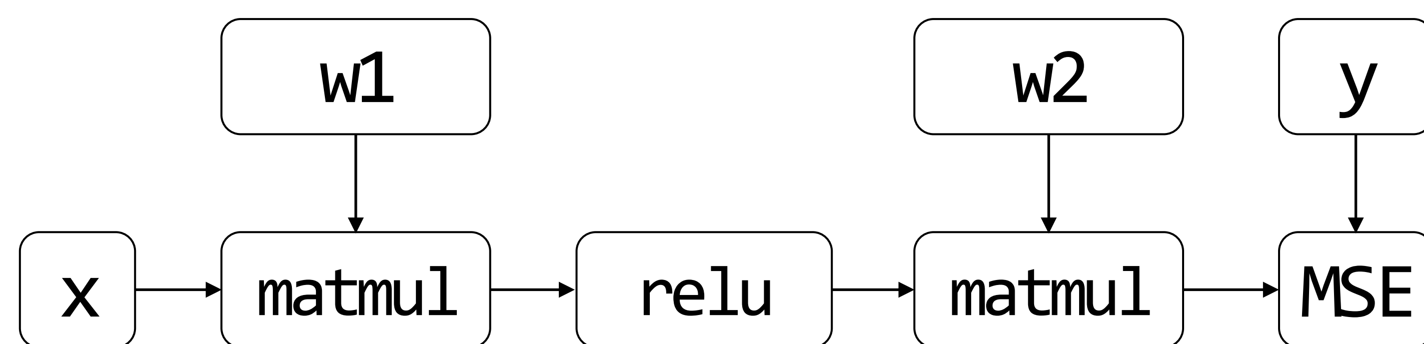
$$L = \mathrm{MSE}(w_2 \cdot \mathrm{ReLU}(w_1 x),\ y) \quad \theta = \{w_1, w_2\},\ D = \{(x, y)\}$$

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

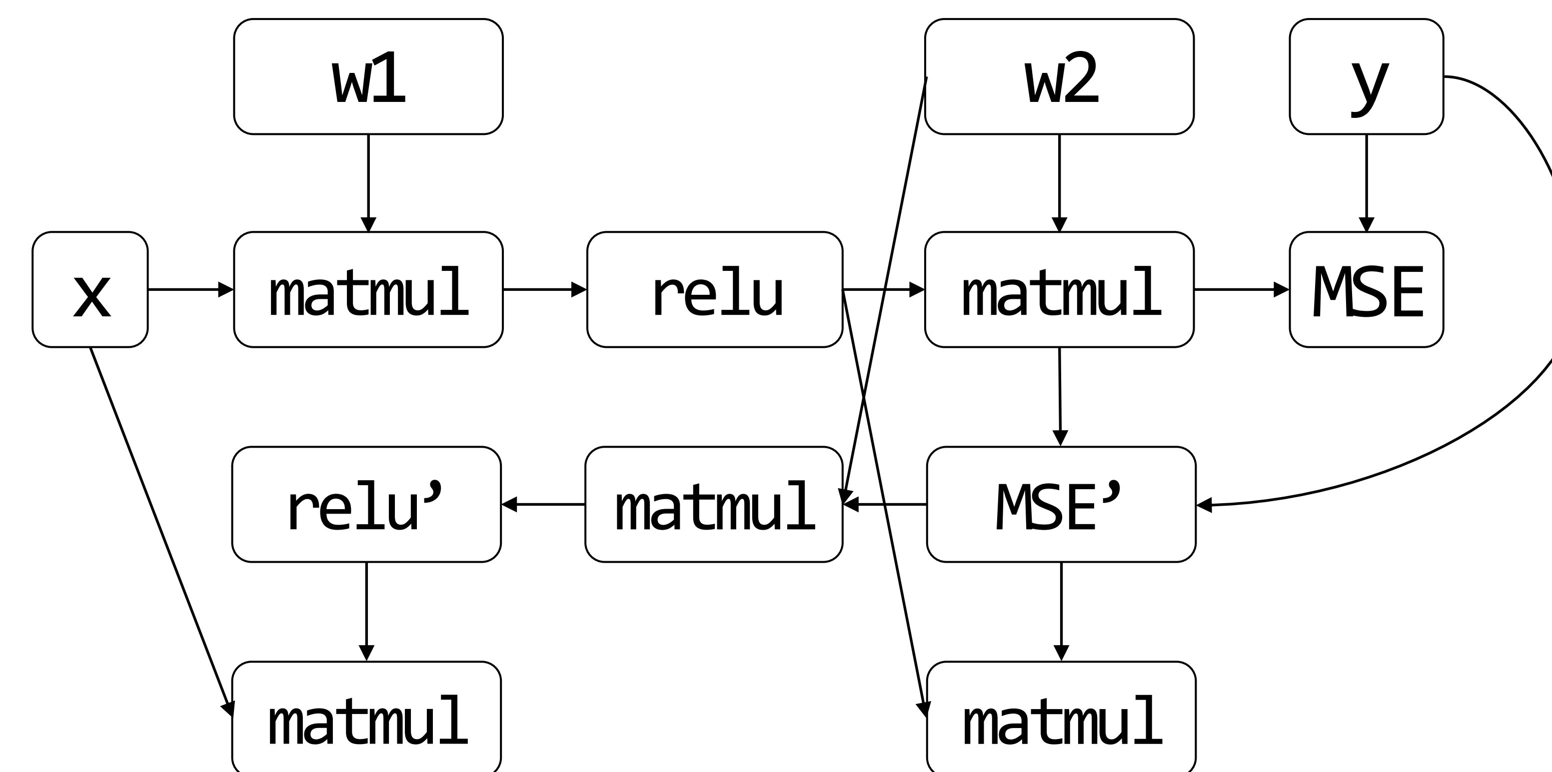
Operator / its output tensor

Data flowing direction

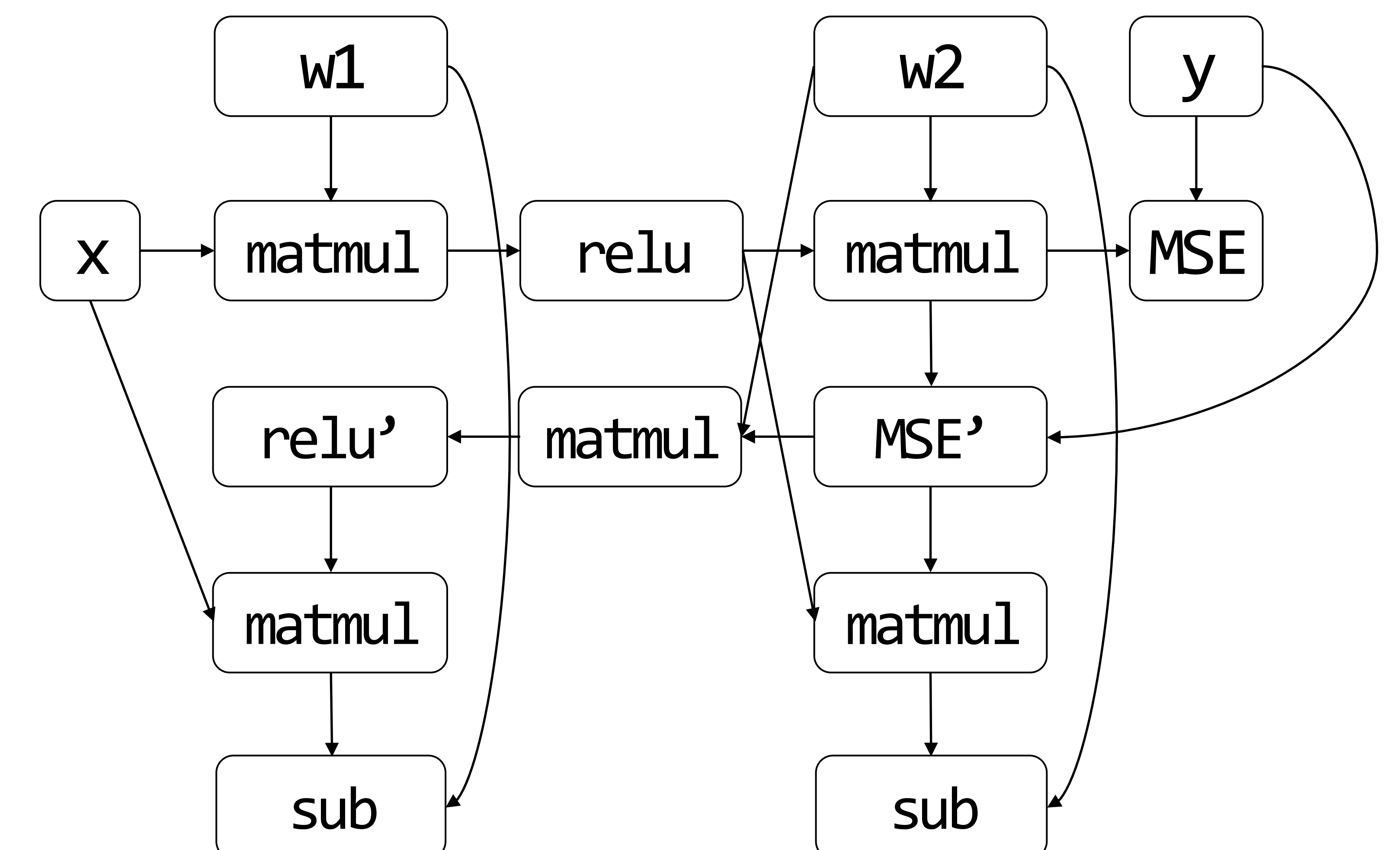
Forward



+Backward



+Weight update



Homework: How to derive gradients for

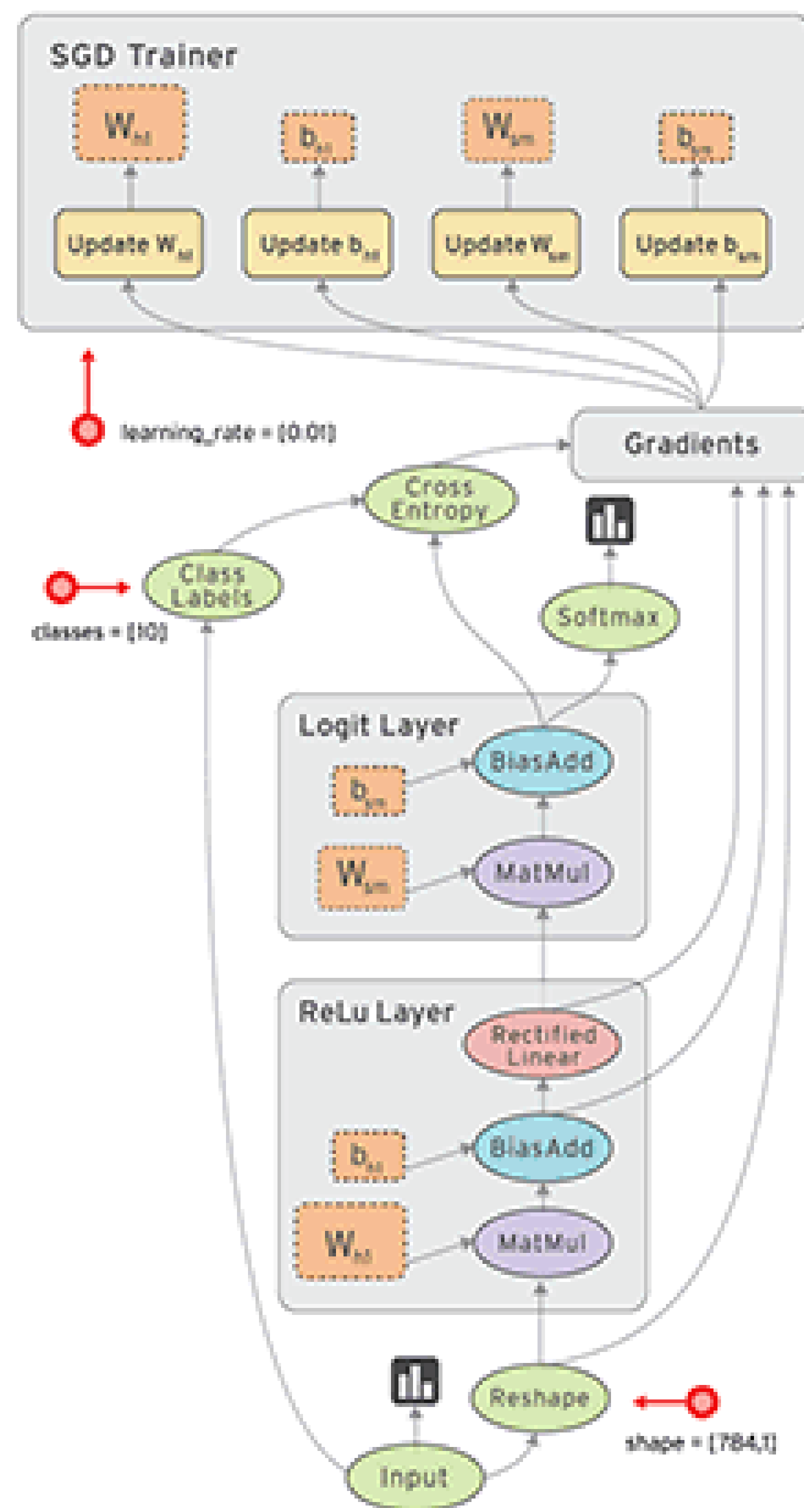
- Softmax cross entropy:

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

Today

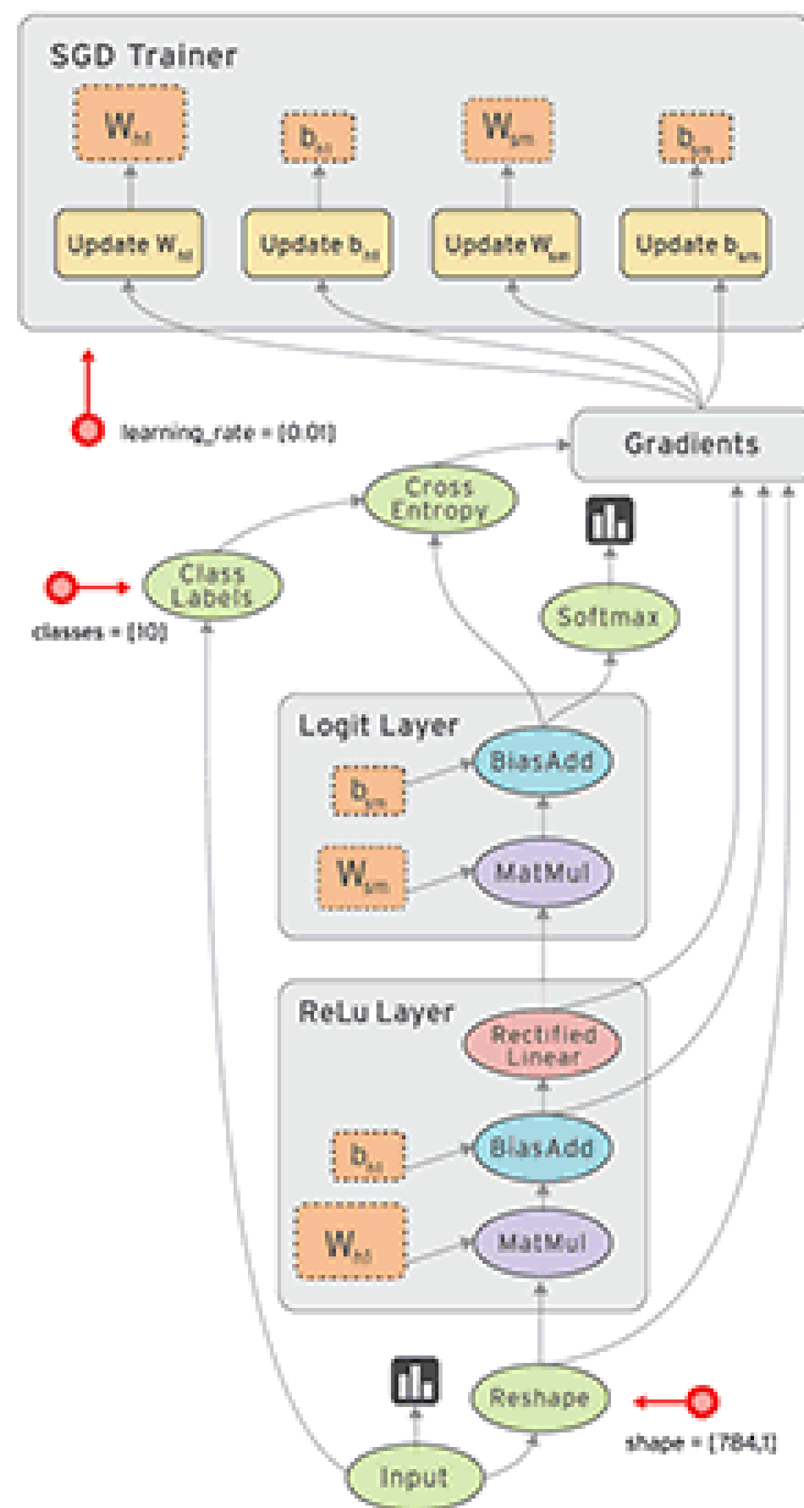
- Autodiff
- **Architecture Overview**

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

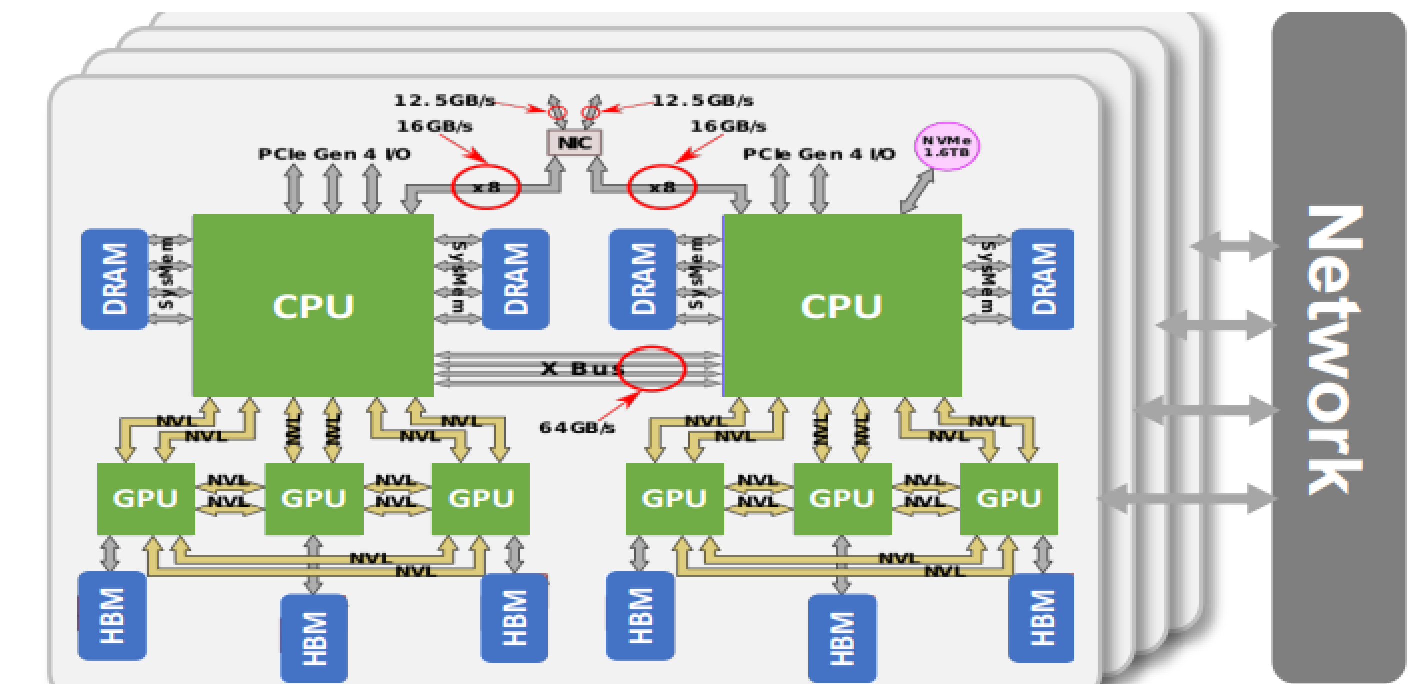
Autodiff

Graph Optimization

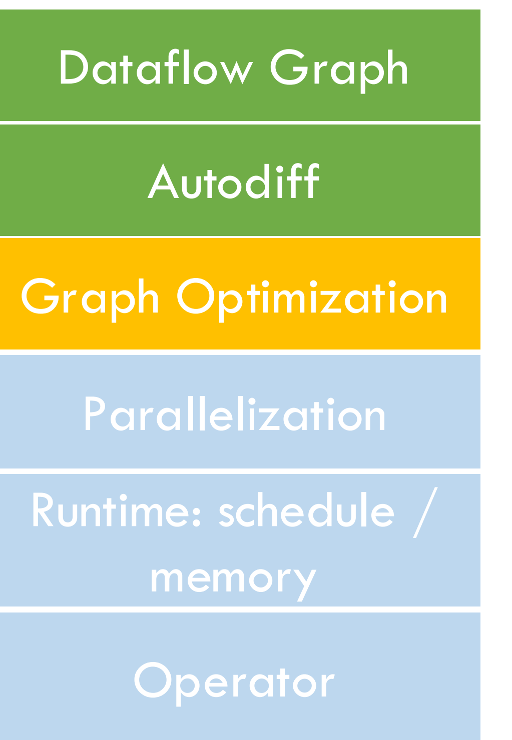
Parallelization

Runtime: schedule / memory

Operator optimization/compilation

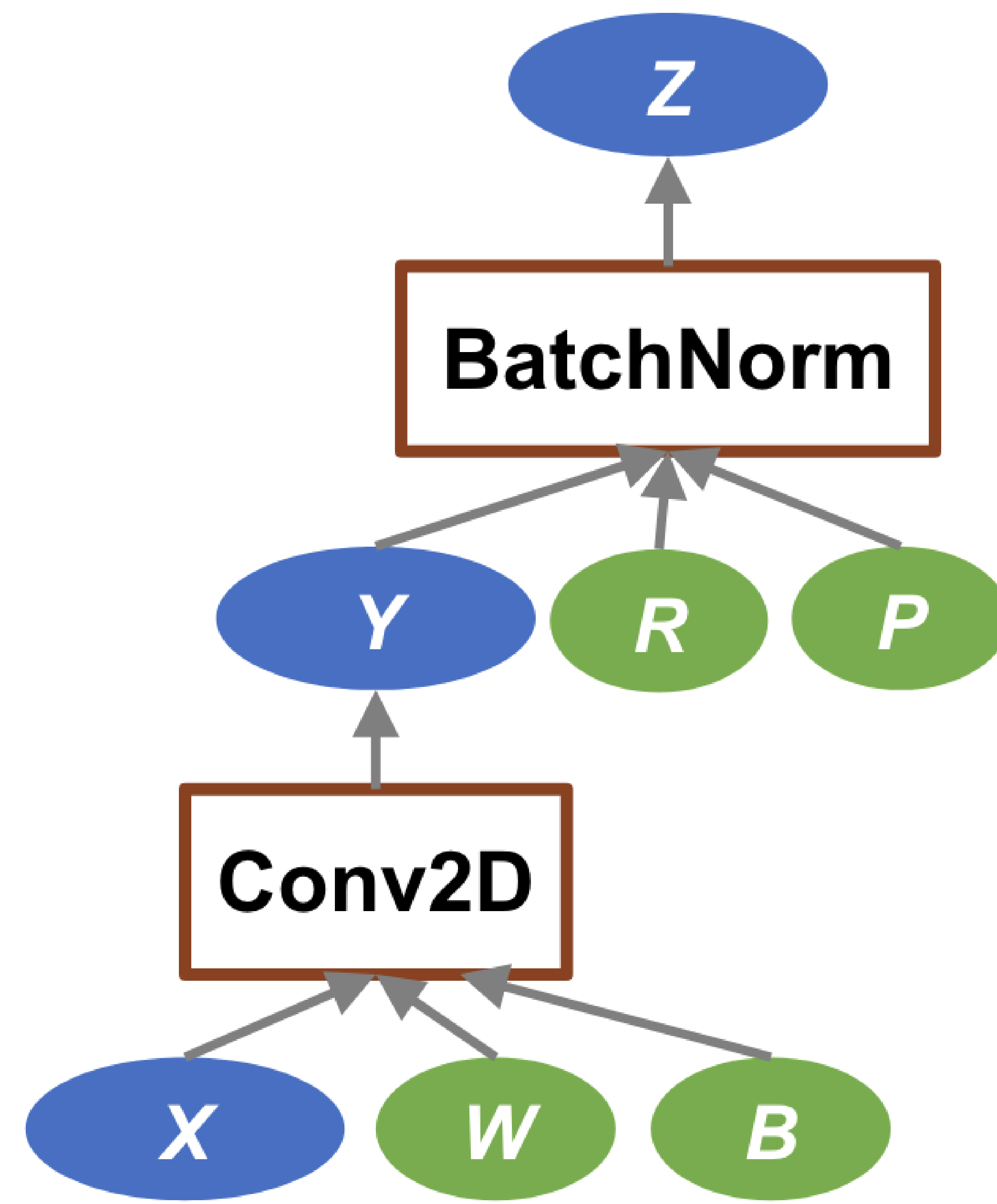


Graph Optimization



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

Motivating Example: ResNet

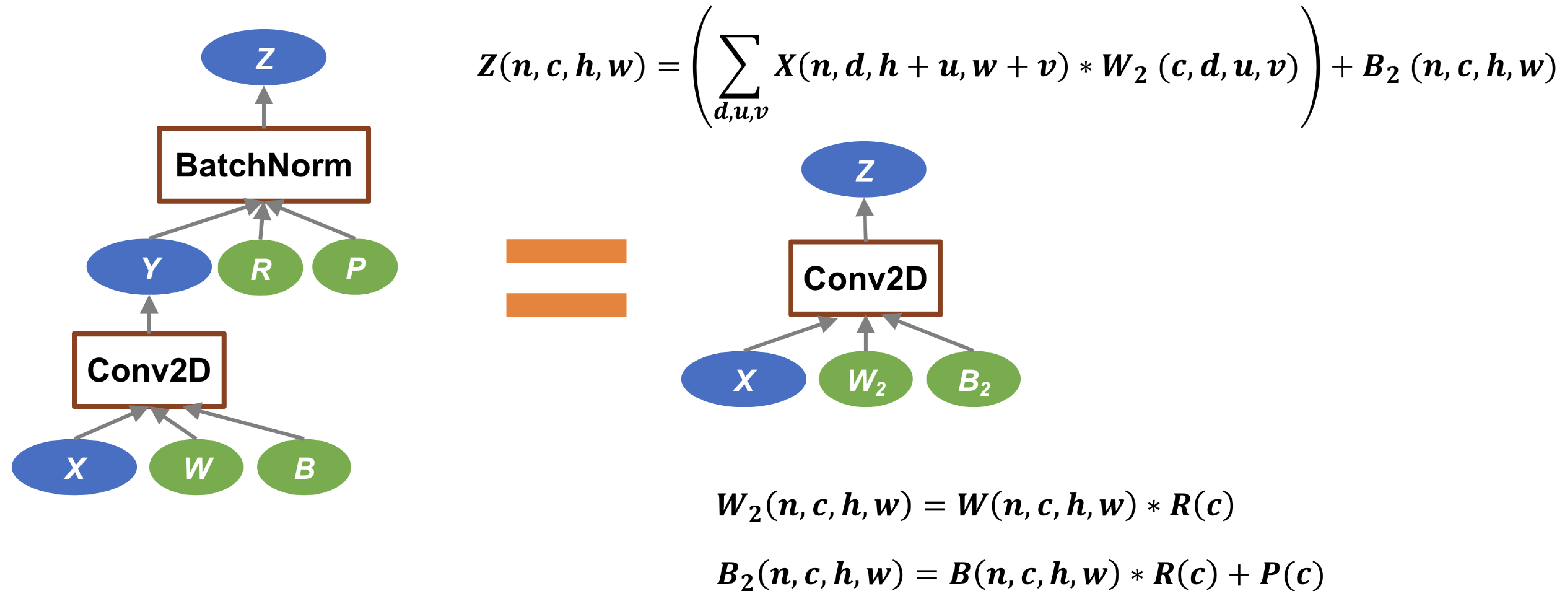


$$Z(n,c,h,w) = Y(n,c,h,w) * R(c) + P(c)$$

$$Y(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W(c,d,u,v)\right) + B(n,c,h,w)$$

Dataflow Graph
Autodiff
Graph Optimization
Parallelization
Runtime: schedule / memory
Operator

Motivating Example: ResNet



$$Z(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_2\ (c,d,u,v)\right) + B_2\ (n,c,h,w)$$

$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

$$B_2(n,c,h,w) = B(n,c,h,w) * R(c) + P(c)$$

- Why the fusion of conv2d & batchnorm is faster?