https://hao-ai-lab.github.io/cse234-w25/

CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

MLSys Basics

Logistics Update

- Enrollment:
  - We have ~60 vacancies now, CSE will process enrollments soon
  - On Thursday, we will enroll the next batch depending on availability.

Last week

- 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

Today

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

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

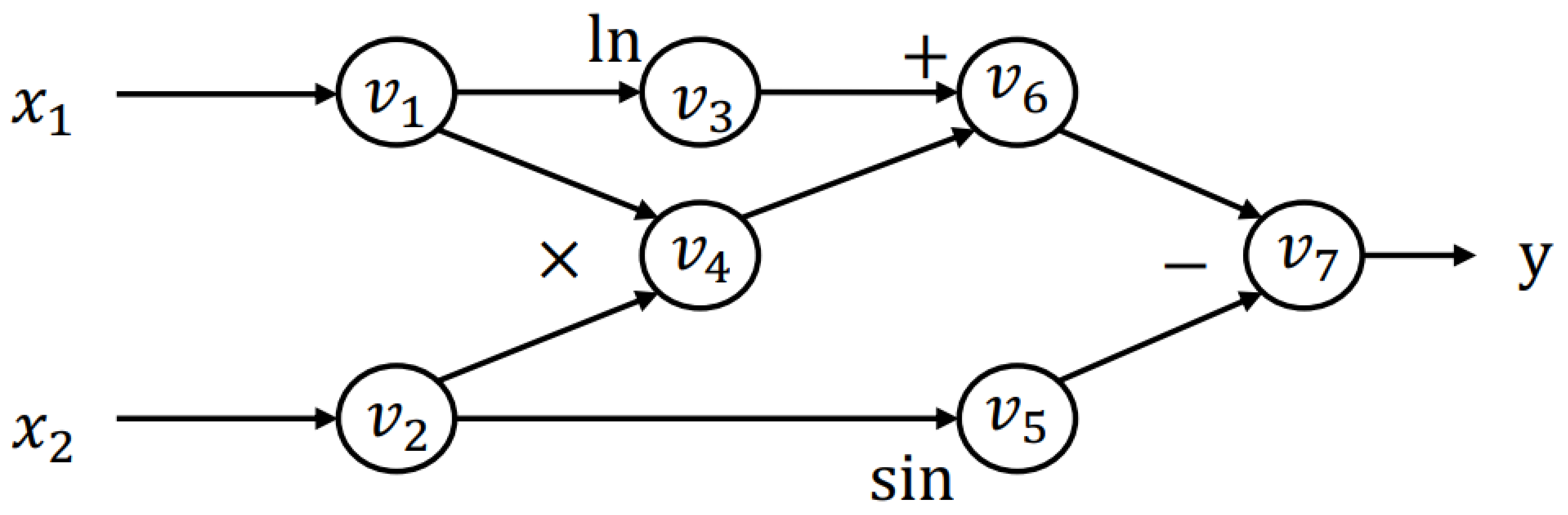
$$\frac{\partial(f(\theta) + g(\theta))}{\partial\theta} = \frac{\partial f(\theta)}{\partial\theta} + \frac{\partial g(\theta)}{\partial\theta}$$

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

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

Map autodiff rules to computational graph

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



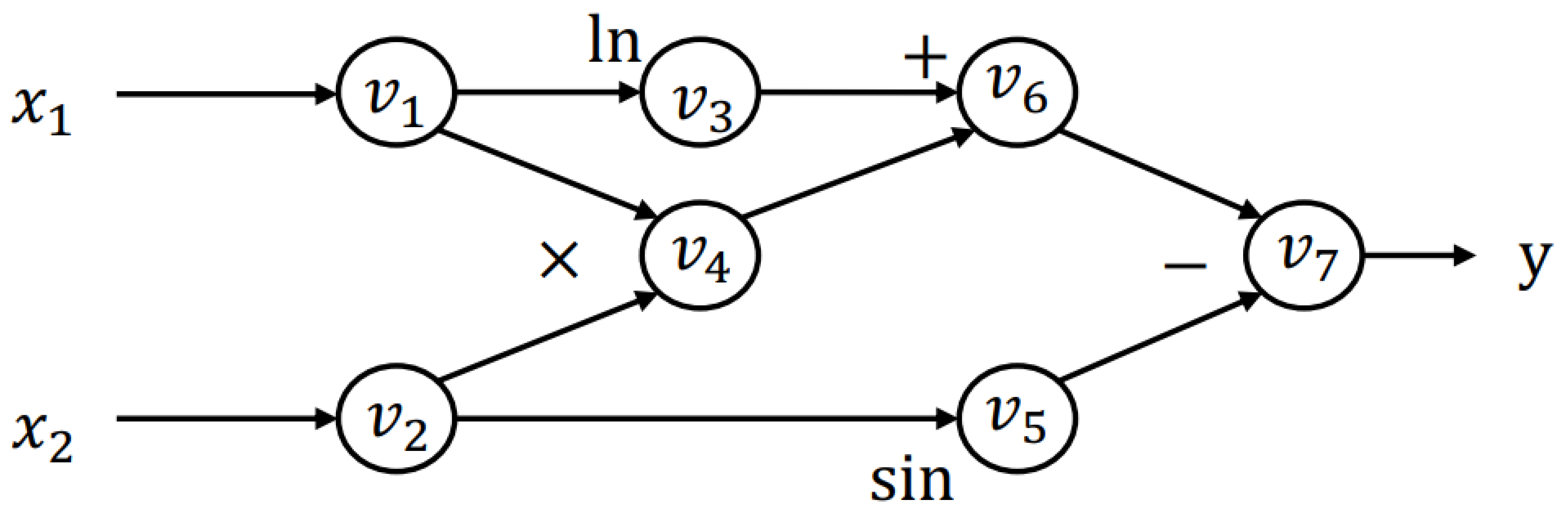
Forward evaluation trace

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

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

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

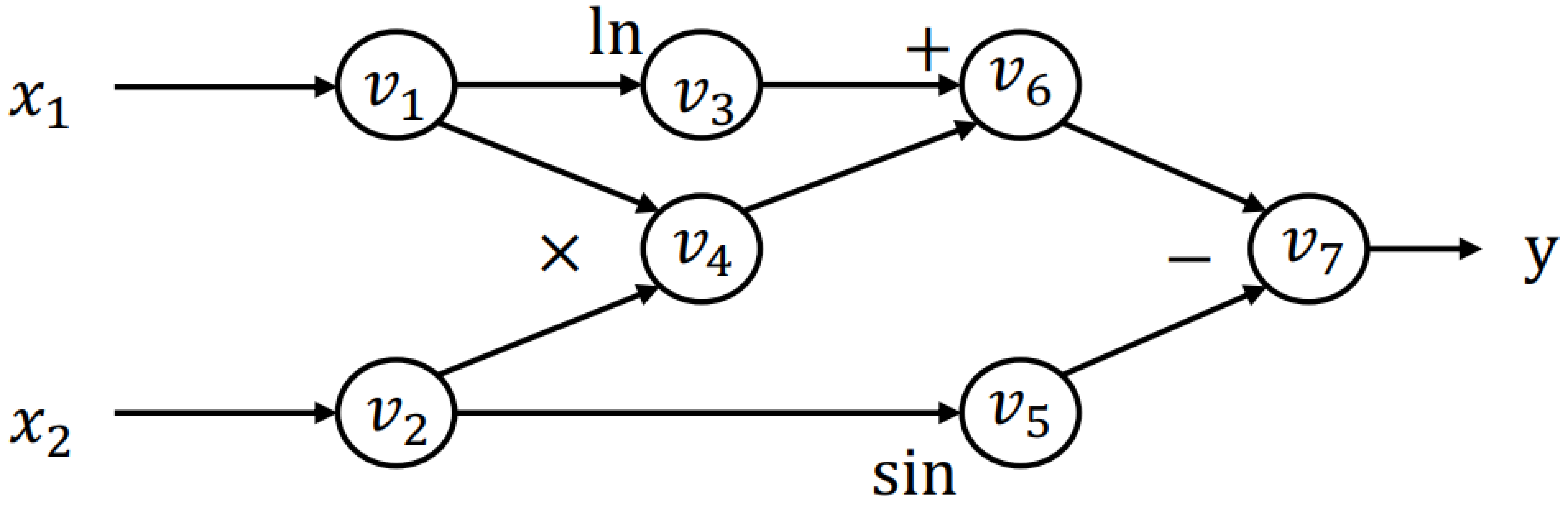
- Finally: $\frac{\partial y}{\partial x_1} = \dot{v_7} = 5.5$

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

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



Forward evaluation trace

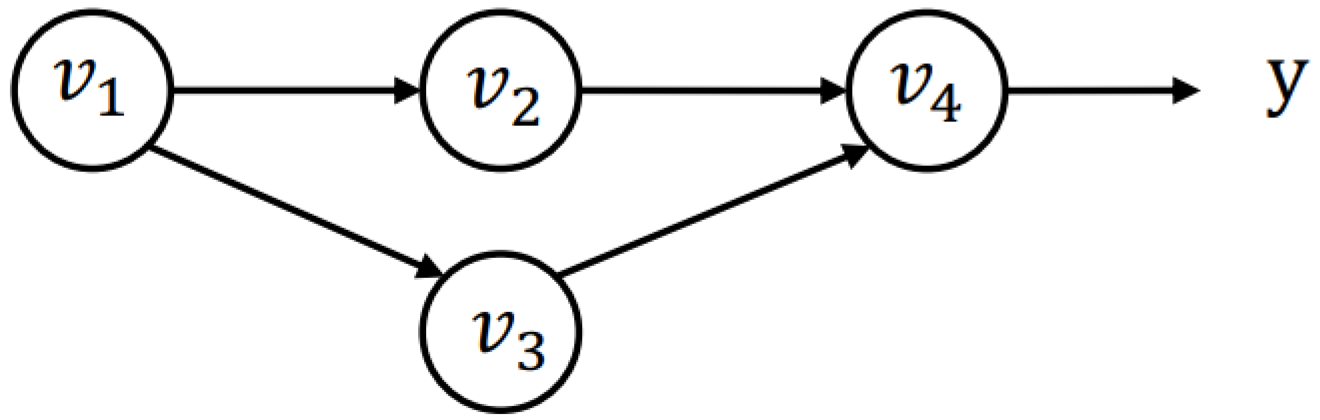
$$
\begin{aligned}
v_1 &= x_1 = 2 \\
v_2 &= x_2 = 5 \\
v_3 &= \ln v_1 = \ln 2 = 0.693 \\
v_4 &= v_1 \times v_2 = 10 \\
v_5 &= \sin v_2 = \sin 5 = -0.959 \\
v_6 &= v_3 + v_4 = 10.693 \\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
y &= v_7 = 11.652
\end{aligned}
$$

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

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

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

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:

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

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 $f: R^n \to R^k$, 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

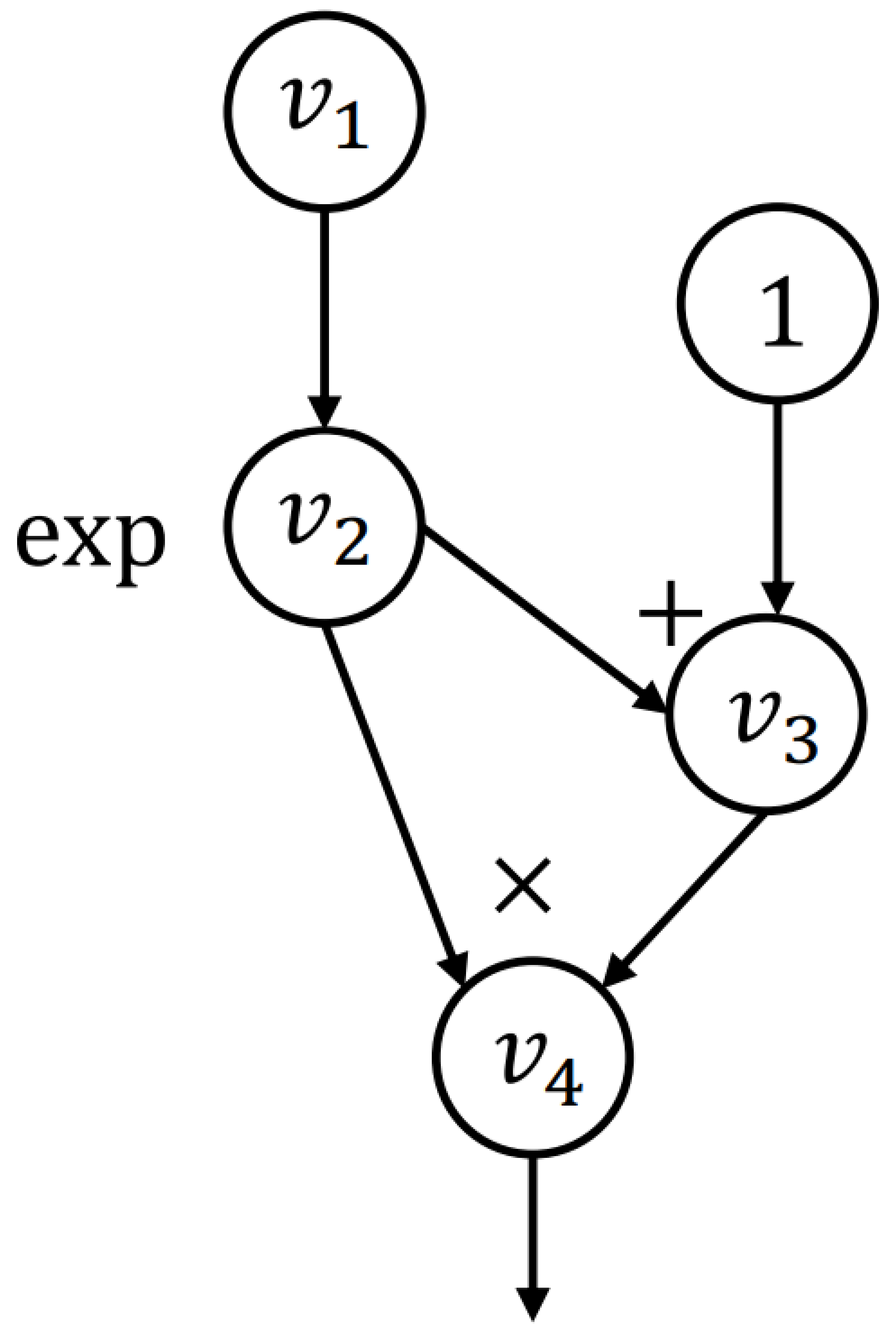
❓ A repr that expresses the backward computation using primitives

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
```

Where: $\overline{v_i} = \sum_j \overline{v_{i\to j}}$, $\overline{v_{k\to i}} = \overline{v_i}\,\frac{\partial v_i}{\partial v_k}$, return $\overline{v_{input}}$.



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

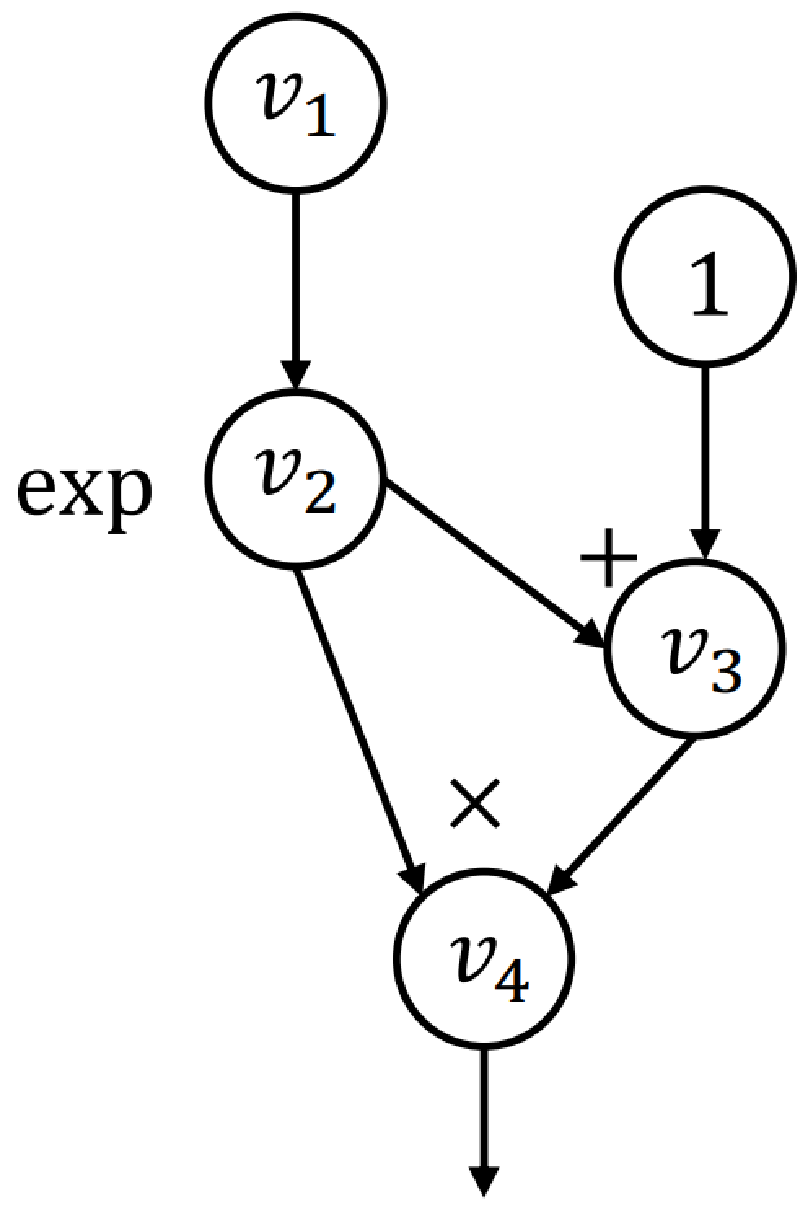
Sum up all partial adjoints to get the gradient

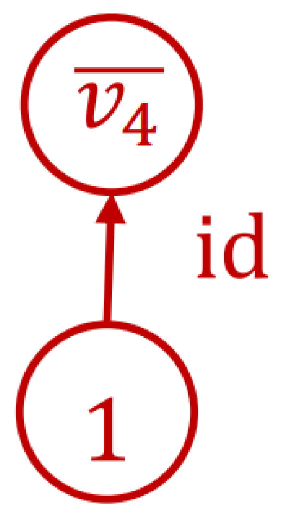
Compute and propagates partial adjoints to its inputs.

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):
    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: {
  4: $[\overline{v_4}]$
}

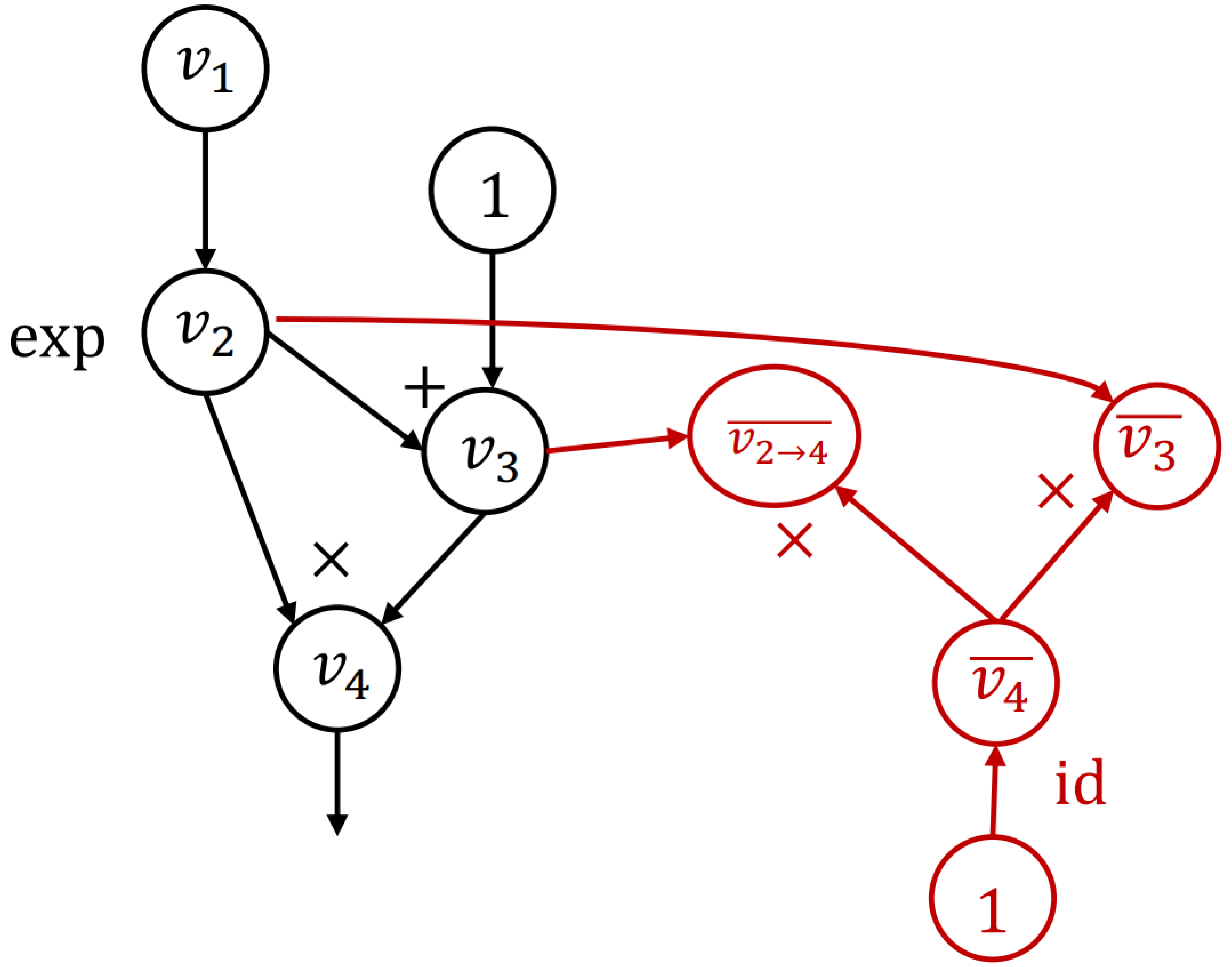
v_4: Inspect (v_2, v_4) and (v_3, v_4)

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):
    $\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = 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: {
  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):
```
$\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = 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 $\overline{v_{input}}$

$i = 3$

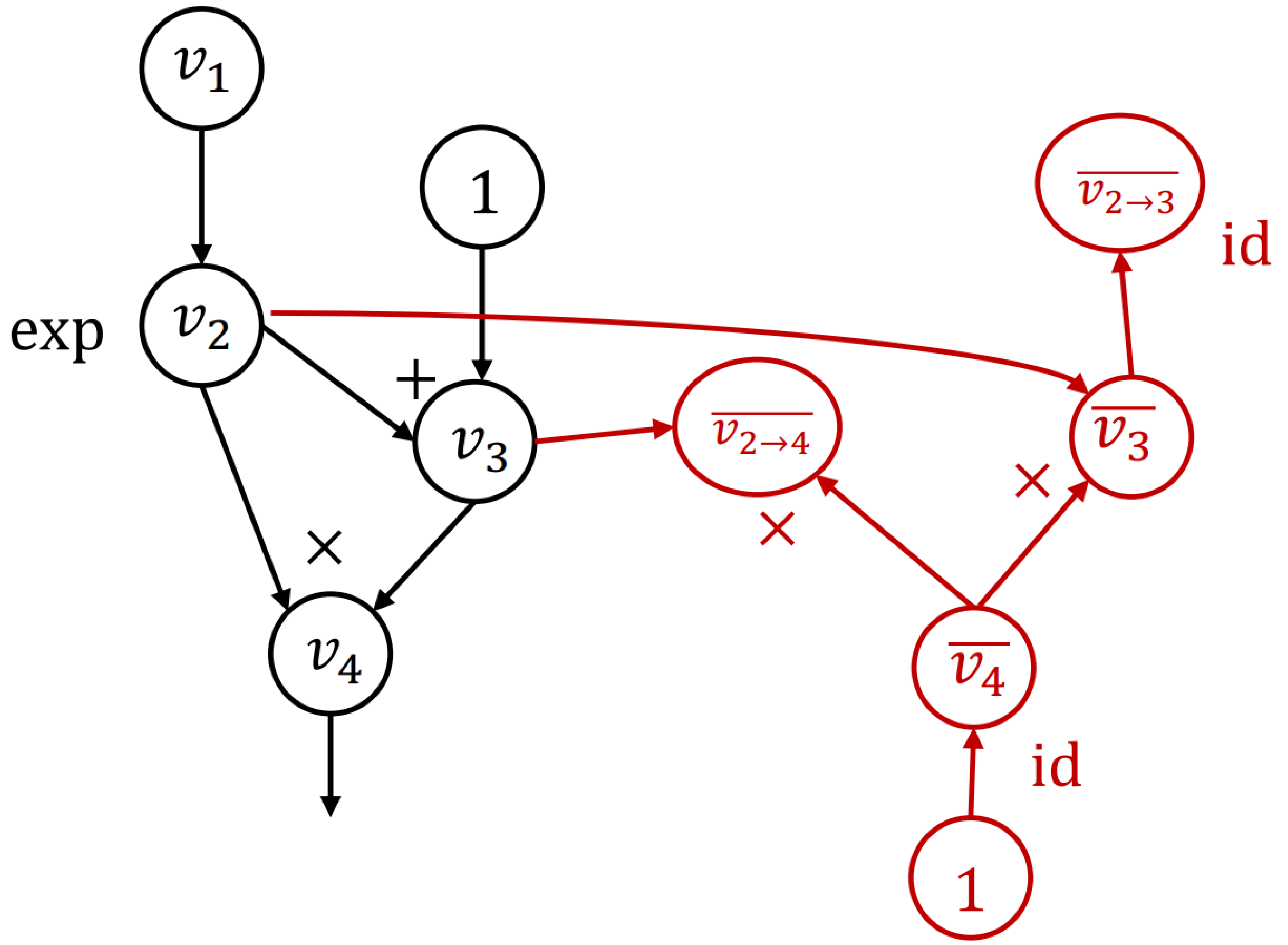
node_to_grad: {

2: $[\overline{v_{2\to4}}, \overline{v_{2\to3}}]$

3: $[\overline{v_3}]$

4: $[\overline{v_4}]$

}

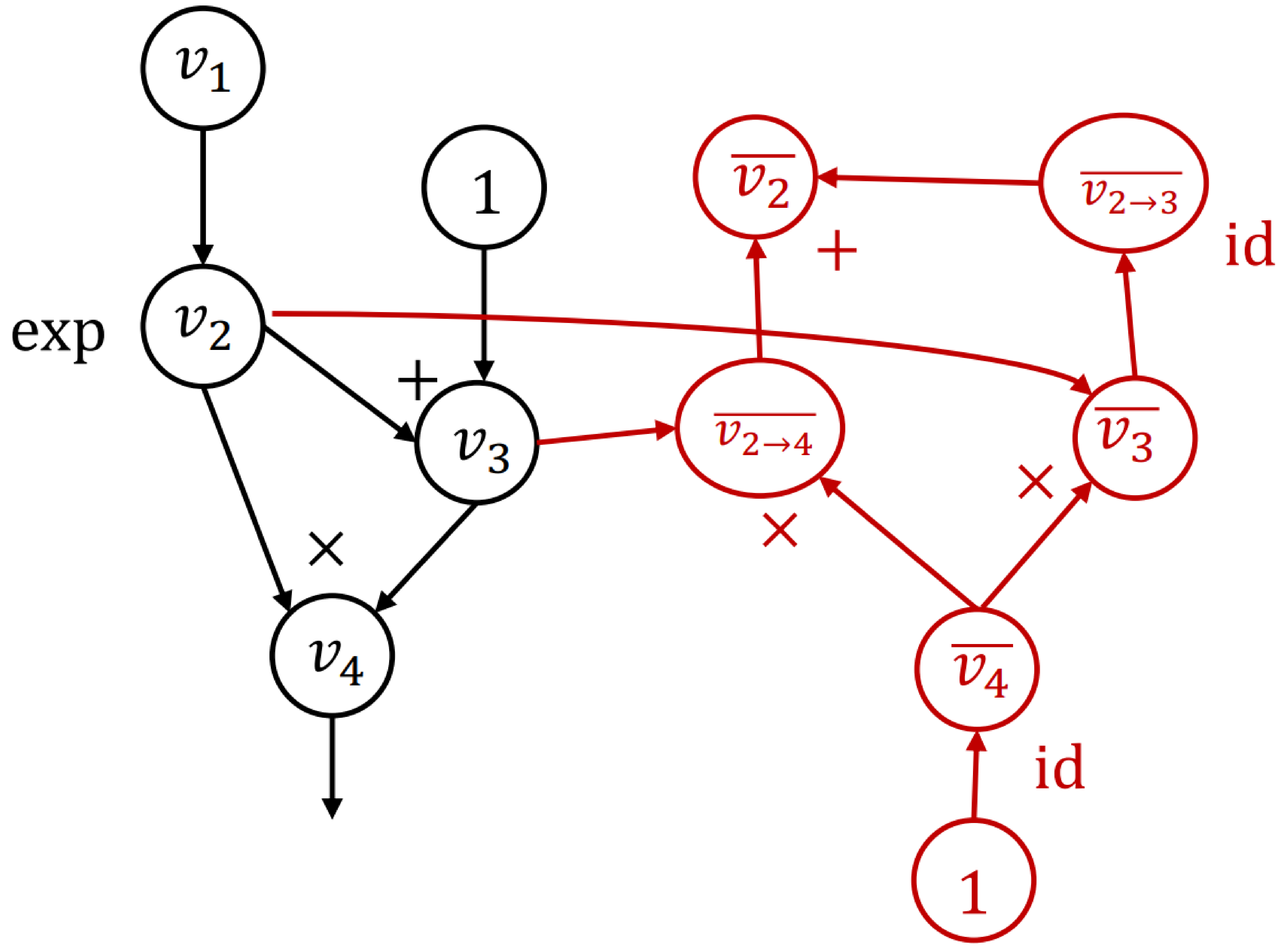


i=2: $\overline{v_2} = \overline{v_{2\to 3}} + \overline{v_{2\to 4}}$

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



Inspect (v_1, v_2)

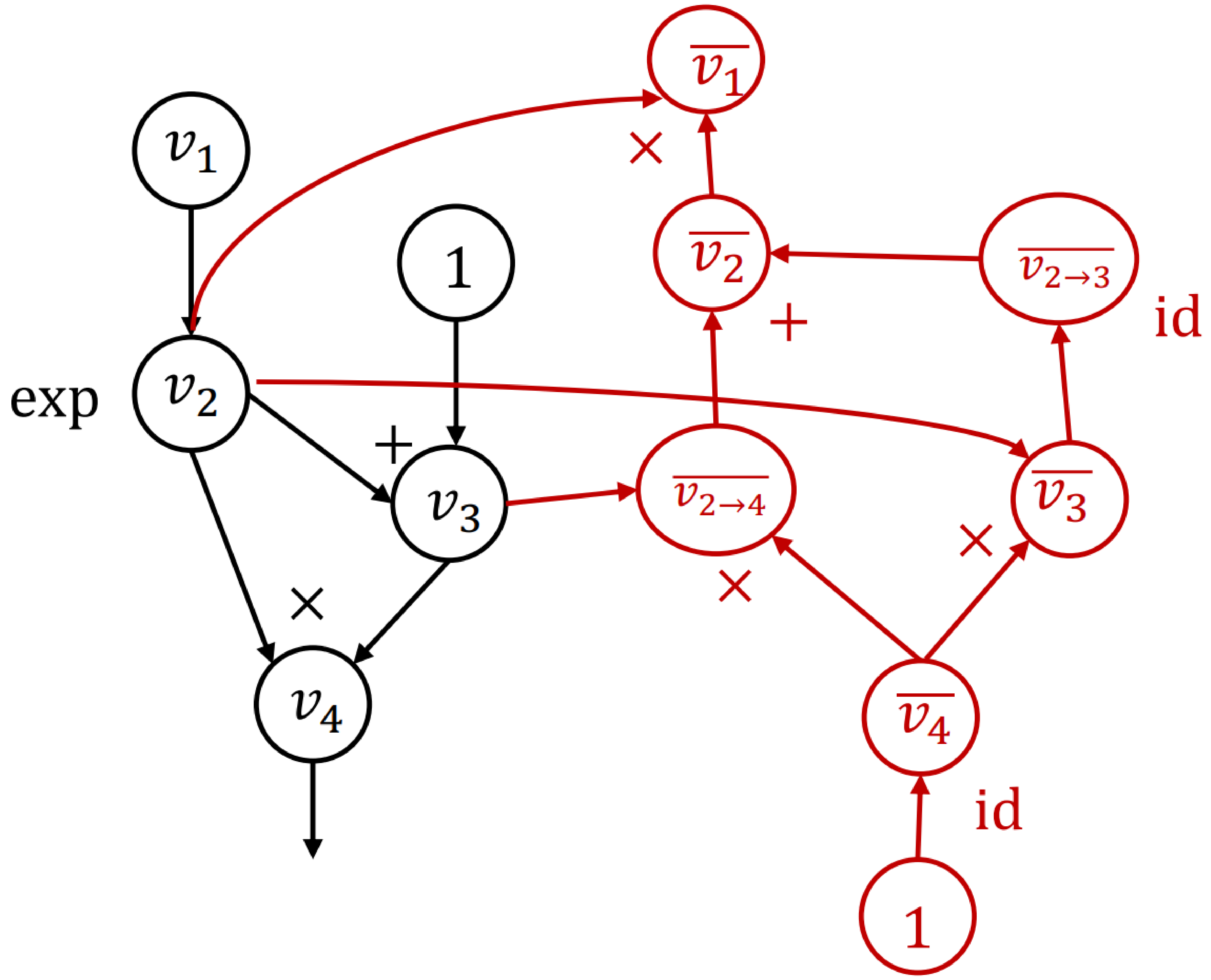
$$\text{i=2: } \overline{v_2} = \overline{v_{2\to3}} + \overline{v_{2\to4}}$$

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

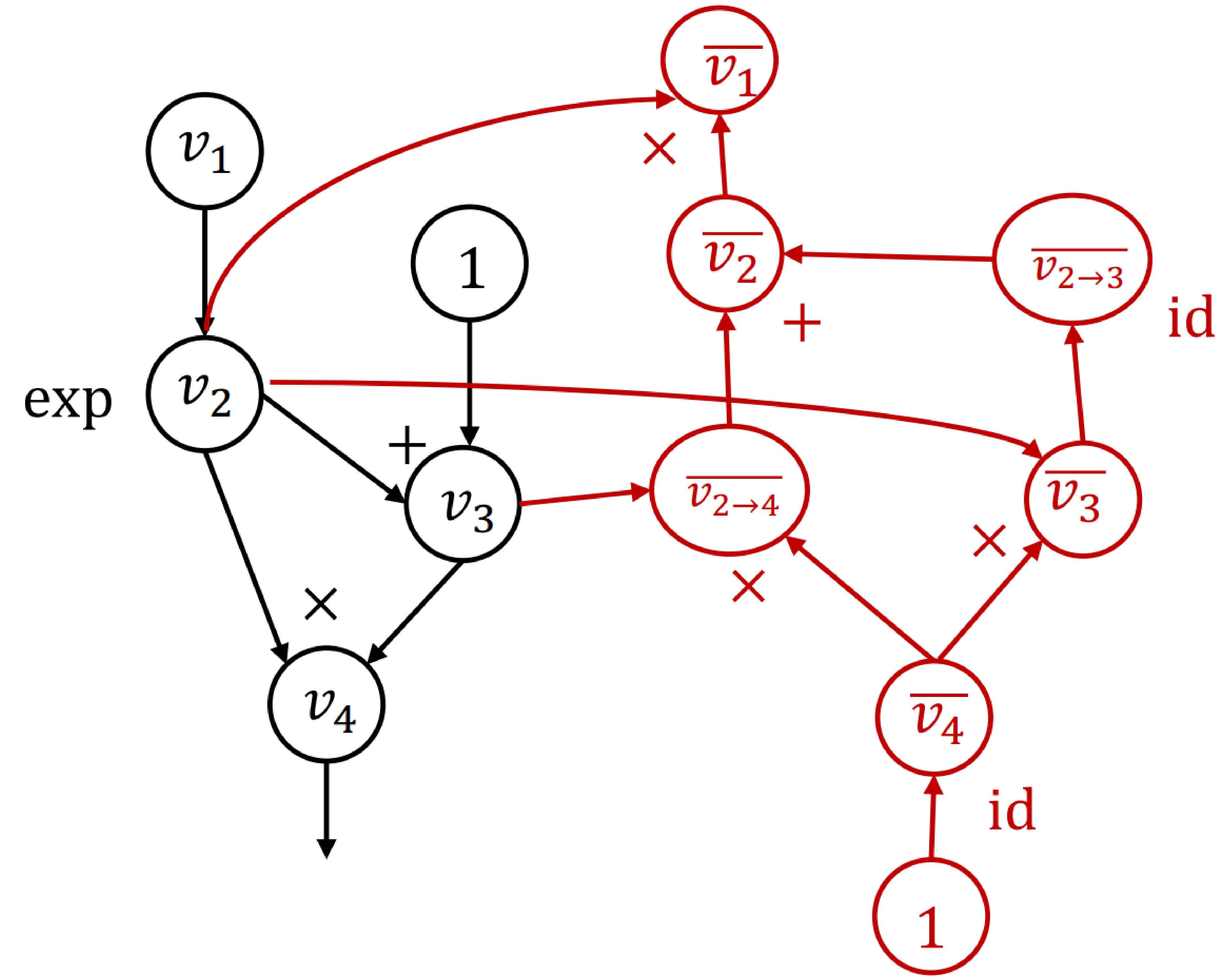
$$\overline{v_1} = \overline{v_{1\to2}}$$

```
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: $[\overline{v_4}]$
}

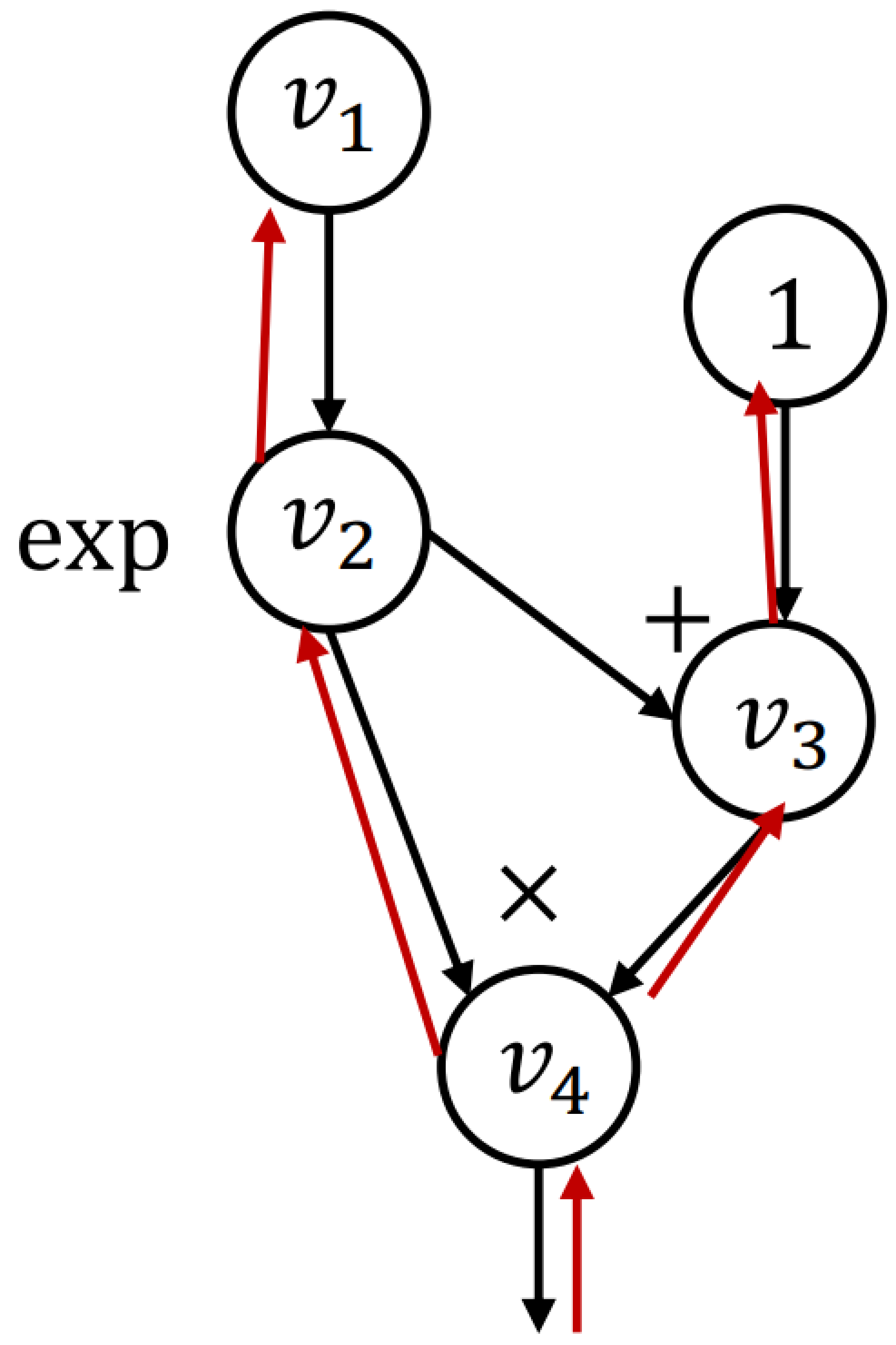


Summary: Backward AD

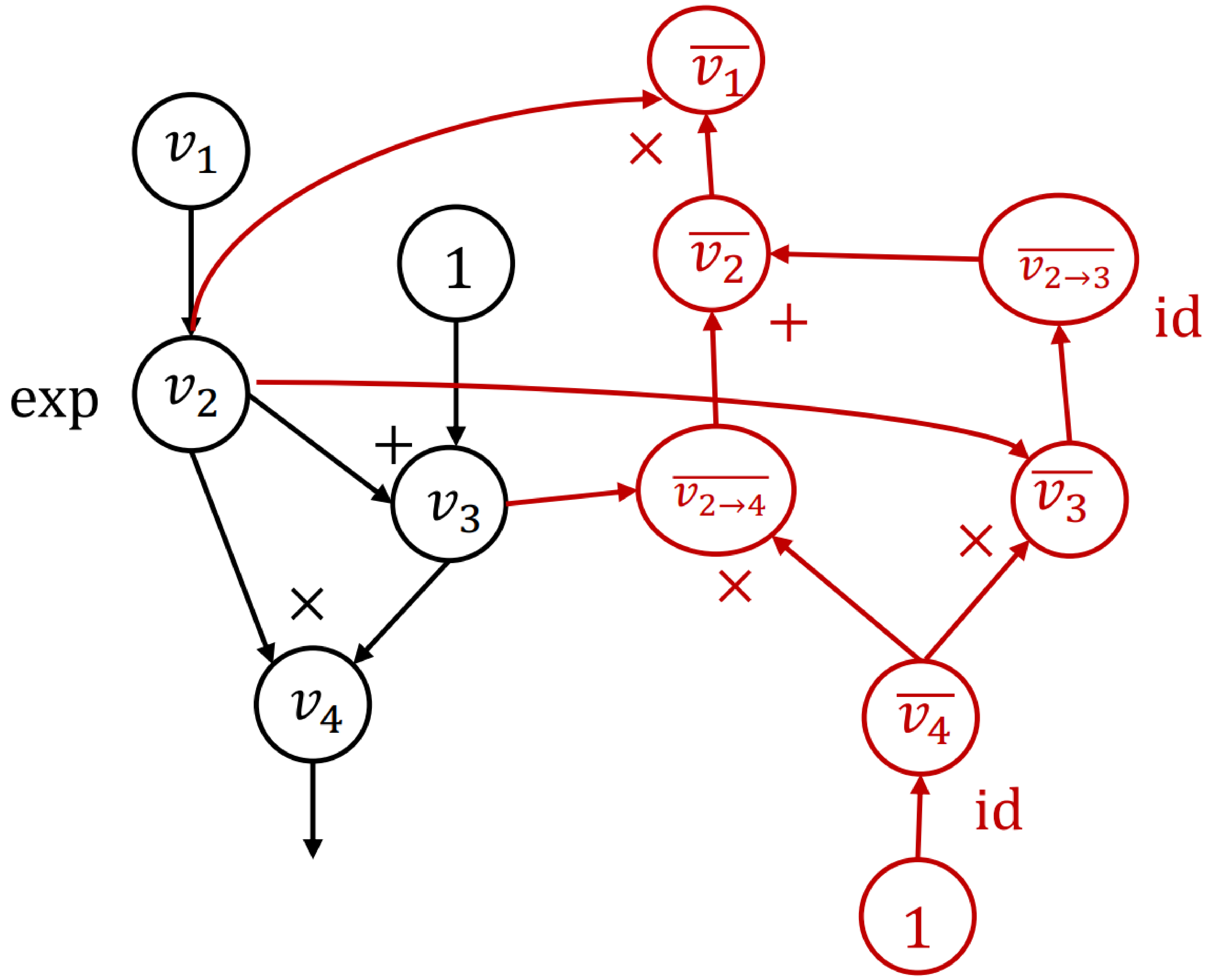


- Construct backward graph in a symbolic way (instead of concrete values)
- This graph can be reused by different input values

Backpropagation vs. Reverse-mode AD



vs.

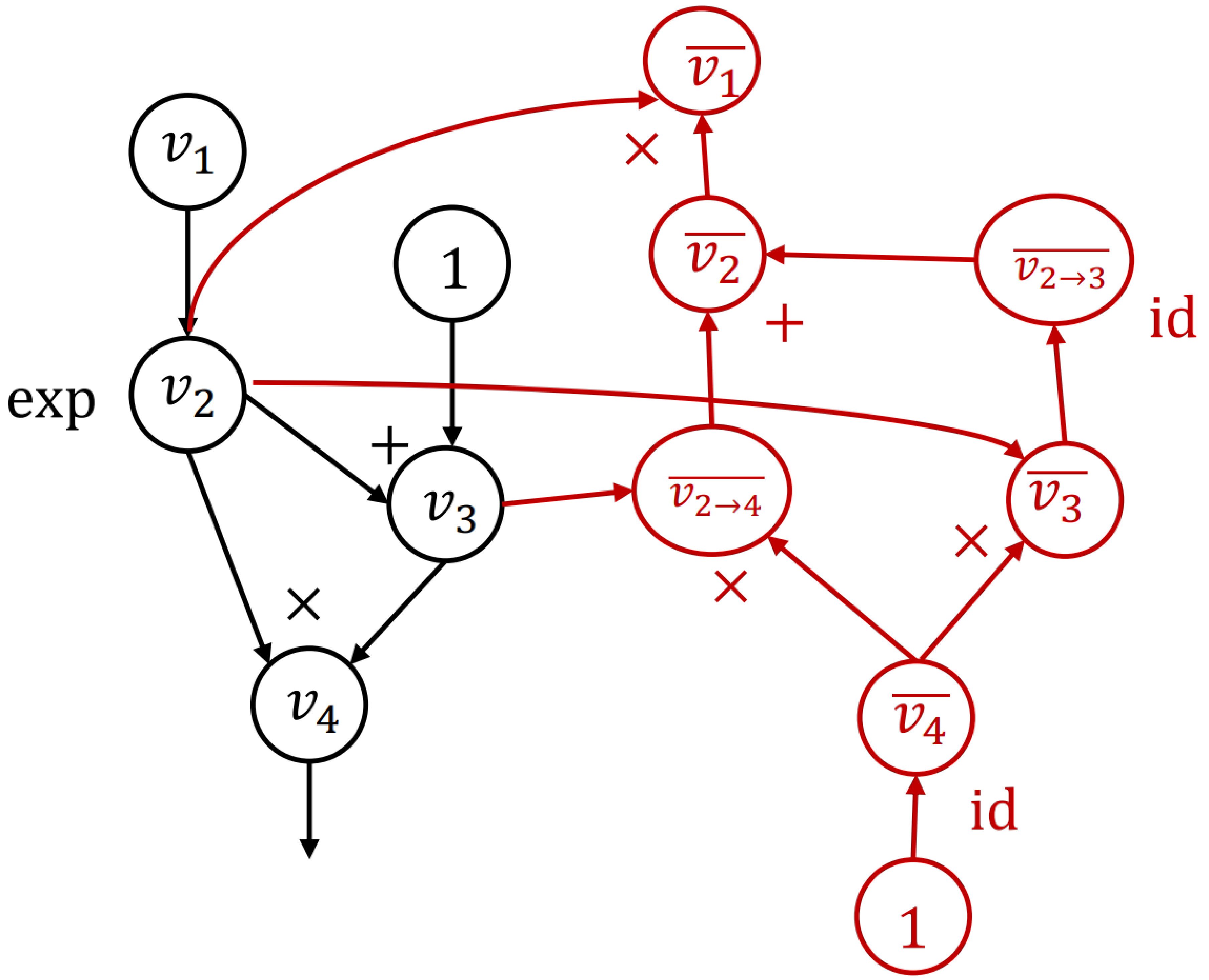


- Run backward through the forward graph
- Caffe/cuda-convnet

- 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
$L(\cdot)$	$\nabla_L(\cdot)$	$f(\cdot)$

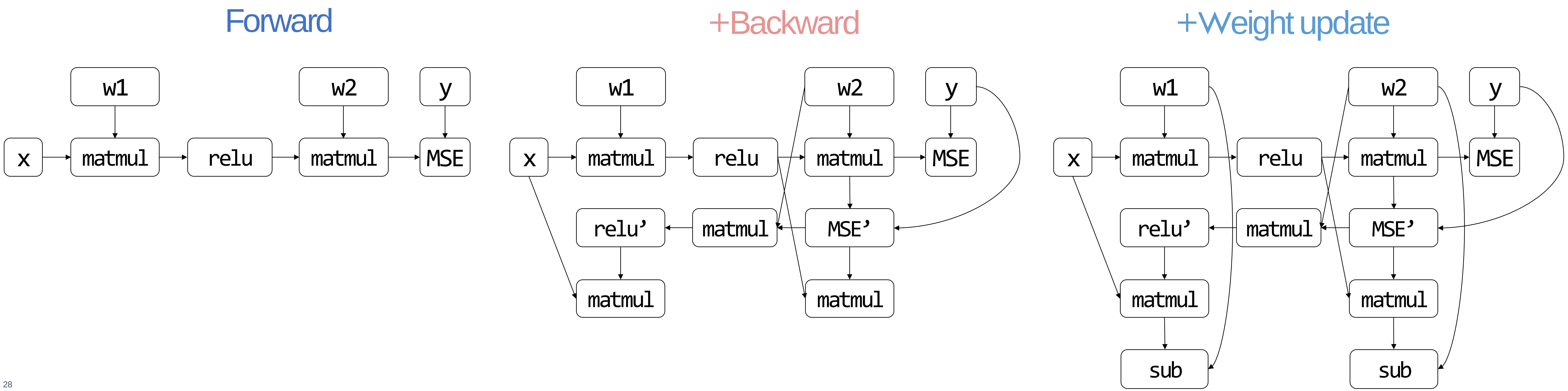
Put in Practice

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

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

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

Operator / its output tensor   Data flowing direction



Homework: How to derive gradients for

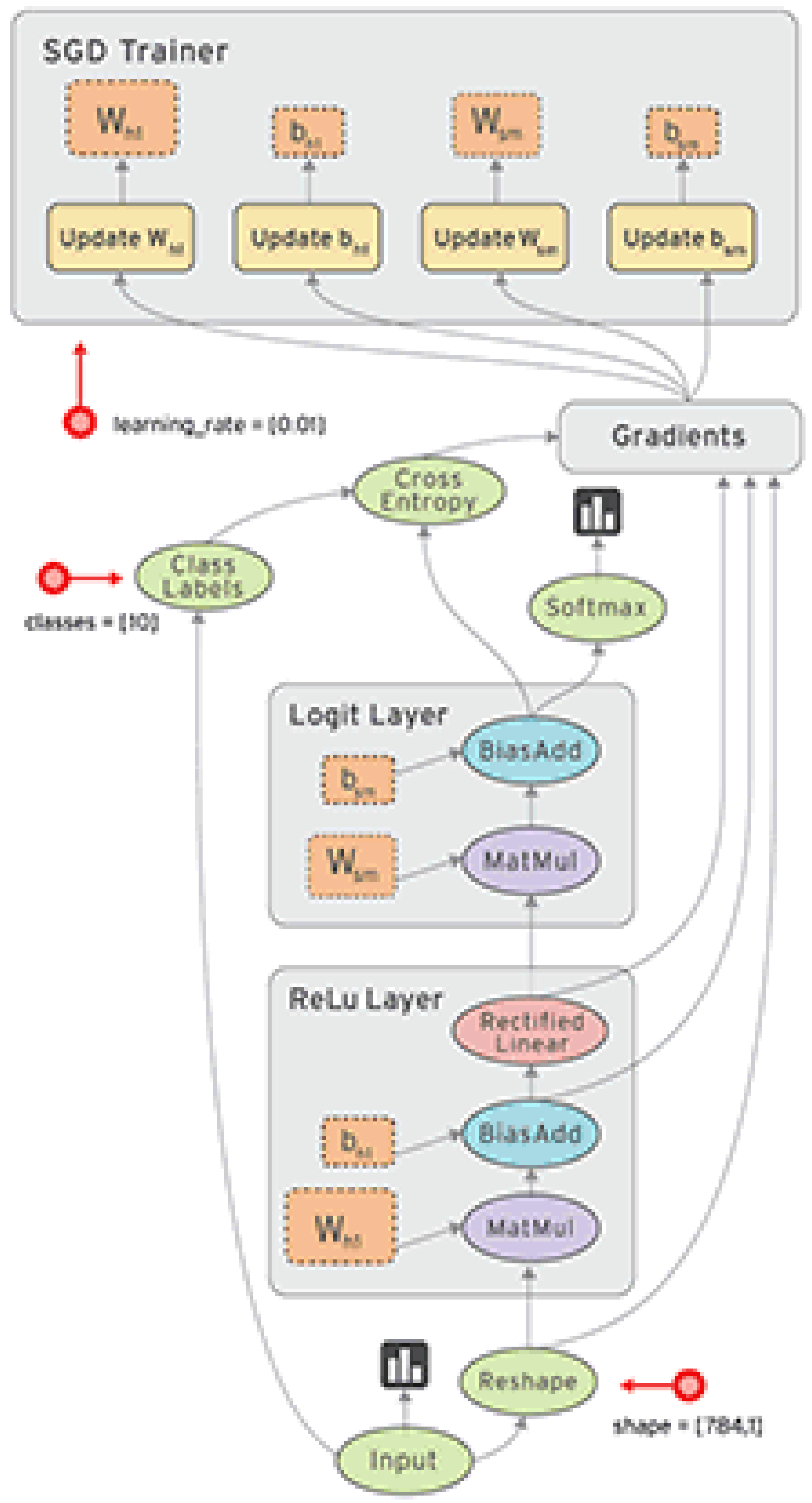
- Softmax cross entropy:

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

Today

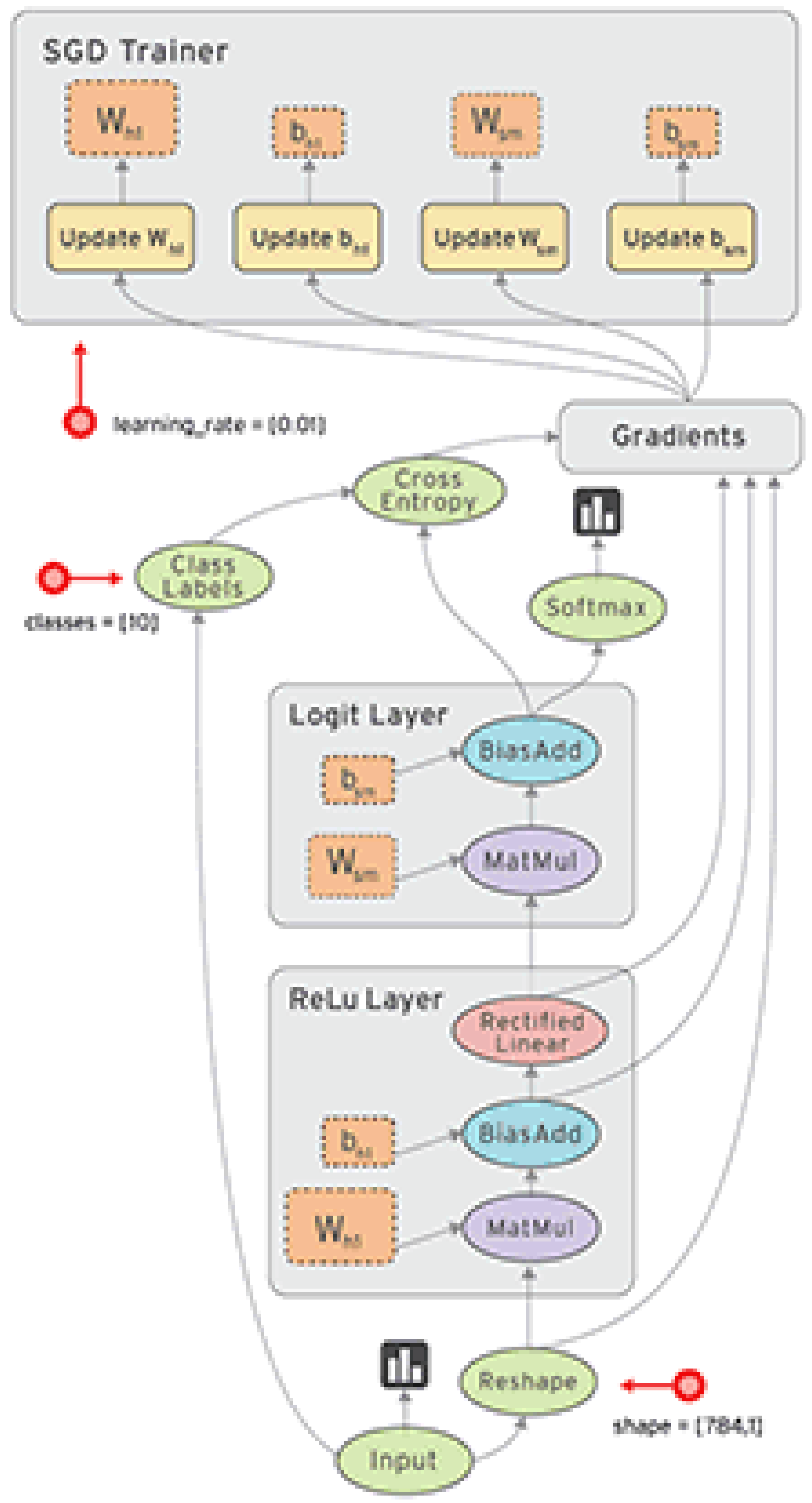
- Autodiff
- **Architecture Overview**

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

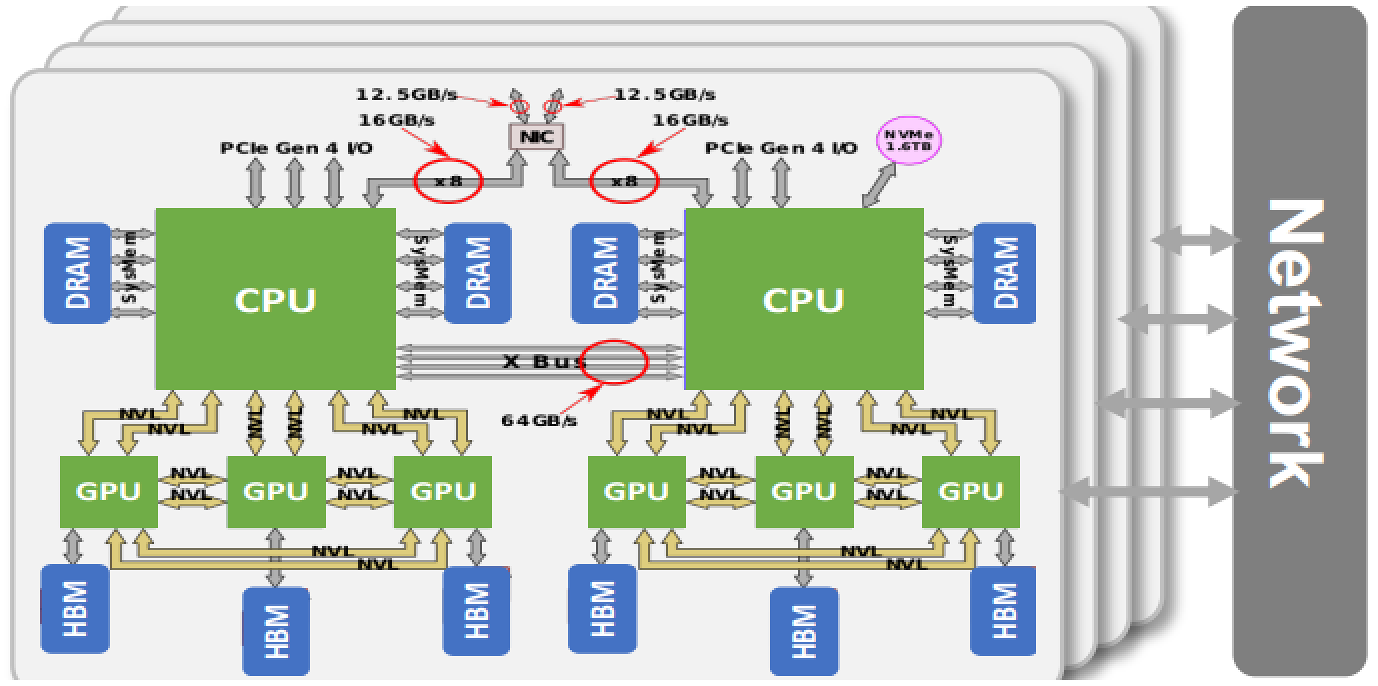
Autodiff

Graph Optimization

Parallelization

Runtime: schedule / memory

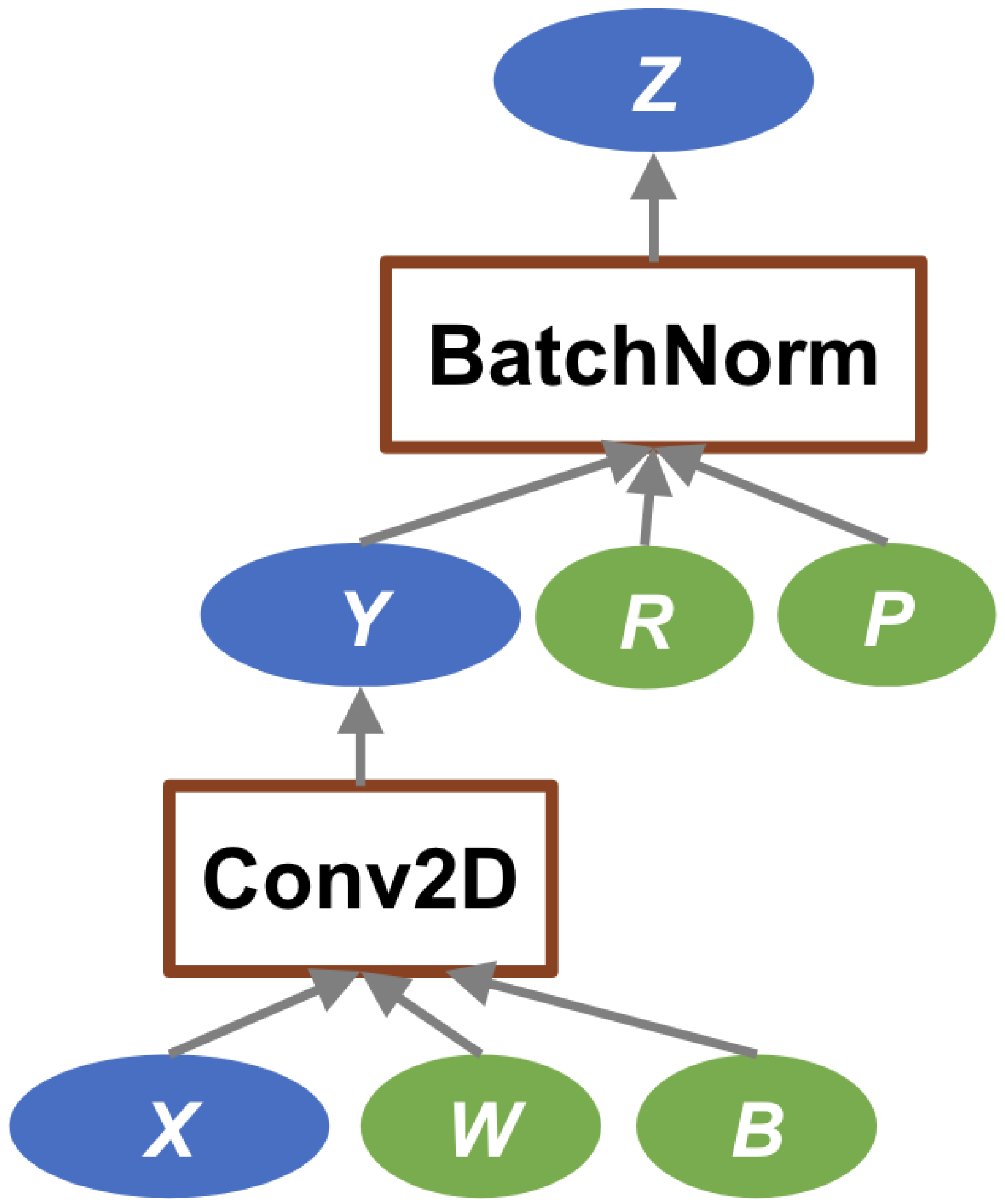
Operator optimization/compilation



Graph Optimization

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

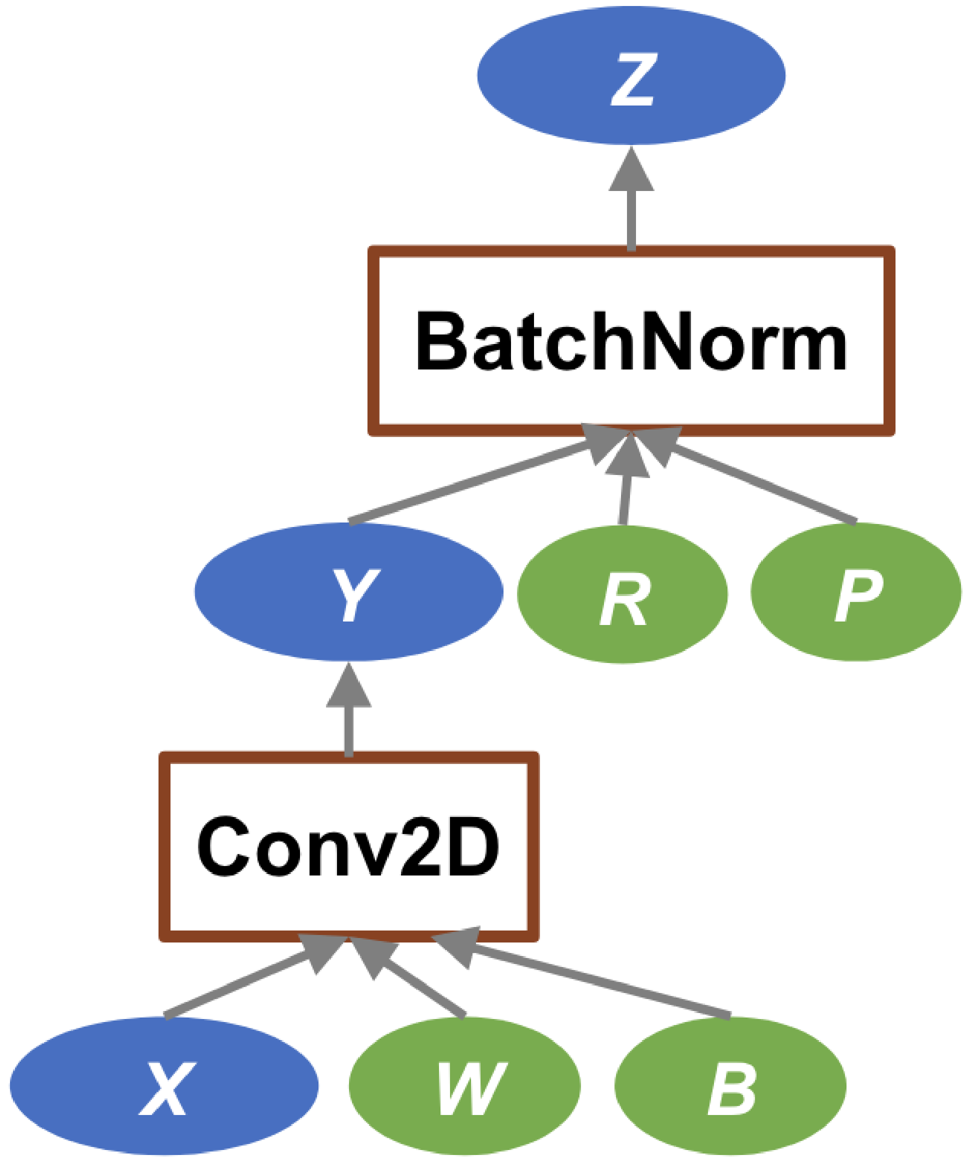
Motivating Example: ResNet



$$\boldsymbol{Z(n,c,h,w) = Y(n,c,h,w) * R(c) + P(c)}$$

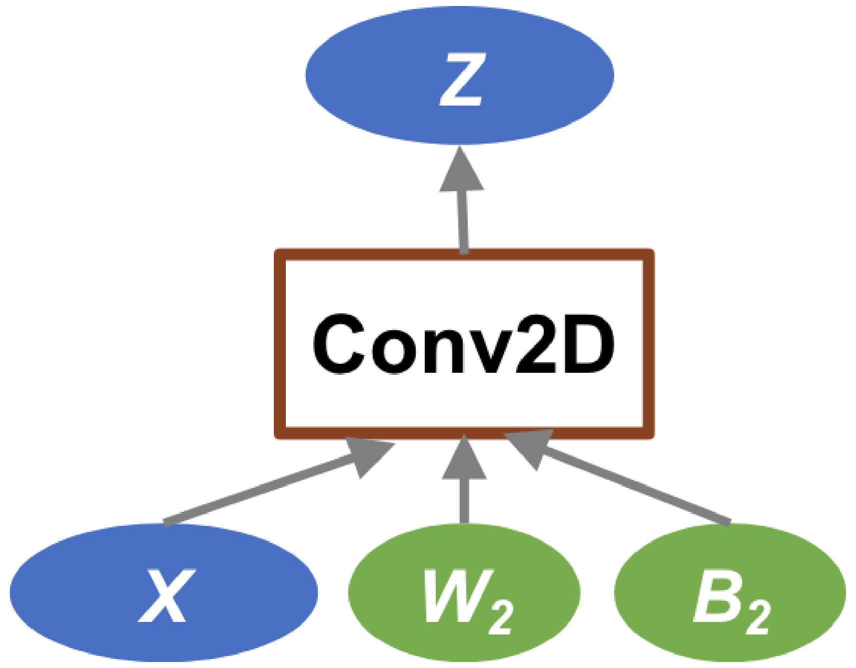
$$\boldsymbol{Y(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W(c,d,u,v)\right) + B(n,c,h,w)}$$

Motivating Example: ResNet



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

=

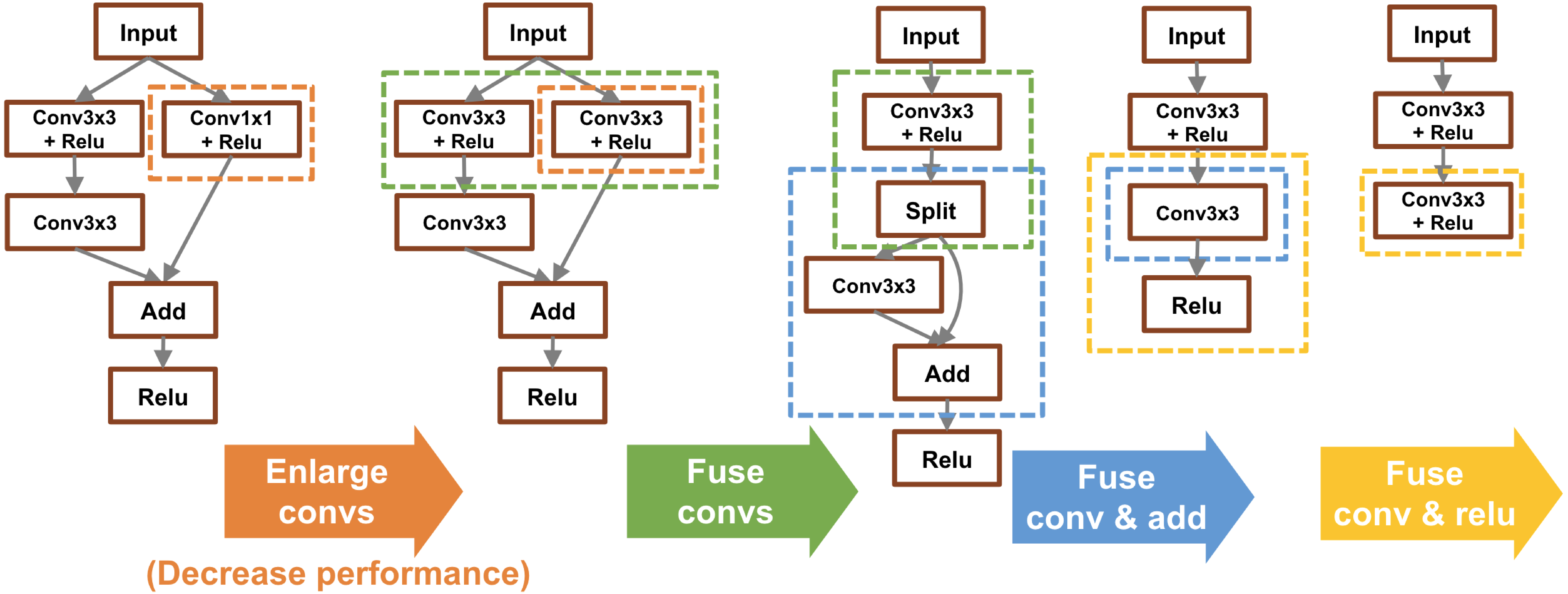


$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

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

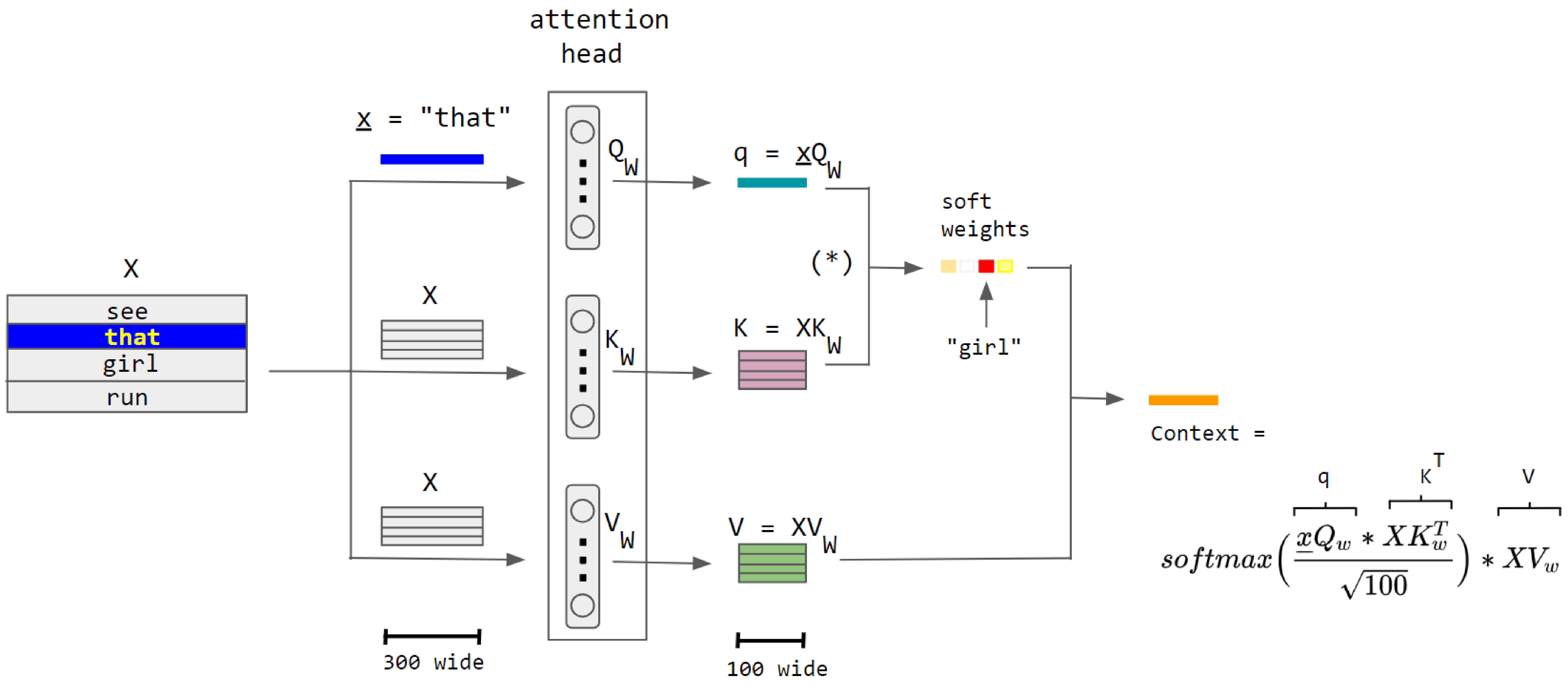
- Why the fusion of conv2d & batchnorm is faster?

Motivating Example: we can go further



- Does each step become faster than previous step?
- How does it perf on different hardware?

Motivating Example 2: Attention




```
# 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

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

Arithmetic intensity

```
void add(int n, float* A, float* B, float* C){
  for (int i=0; i<n; i++)
    C[i] = A[i] + B[i];
}
```

Two loads, one store per math op (arithmetic intensity = 1/3)

1. Read A[i]
2. Read B[i]
3. Add A[i]+B[i]
4. Store C[i]

Which program performs better? Program 1

```
void add(int n, float* A, float* B, float* C){
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}

float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Which program performs better? Program 2

```
float*  A, *B,  *C,  *D,  *E,  *tmp1, *tmp2;
//  assume arrays are  allocated here
//  compute  E  = D + ((A  + B) *  C)
add(n, A, B, tmp1);
mul(n,  tmp1,  C, tmp2);
add(n, tmp2,  D, E);



void fused(int  n, float*  A, float*  B, float*  C, float*  D,
  float*  E) {
  for  (int  i=0; i<n; i++)
    E[i]  = D[i]  + (A[i] + B[i])  *  C[i];
}
//  compute  E  = D + (A + B) *  C
fused(n,  A, B,C,   D, E);
```

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops
arithmetic intensity = 3/5

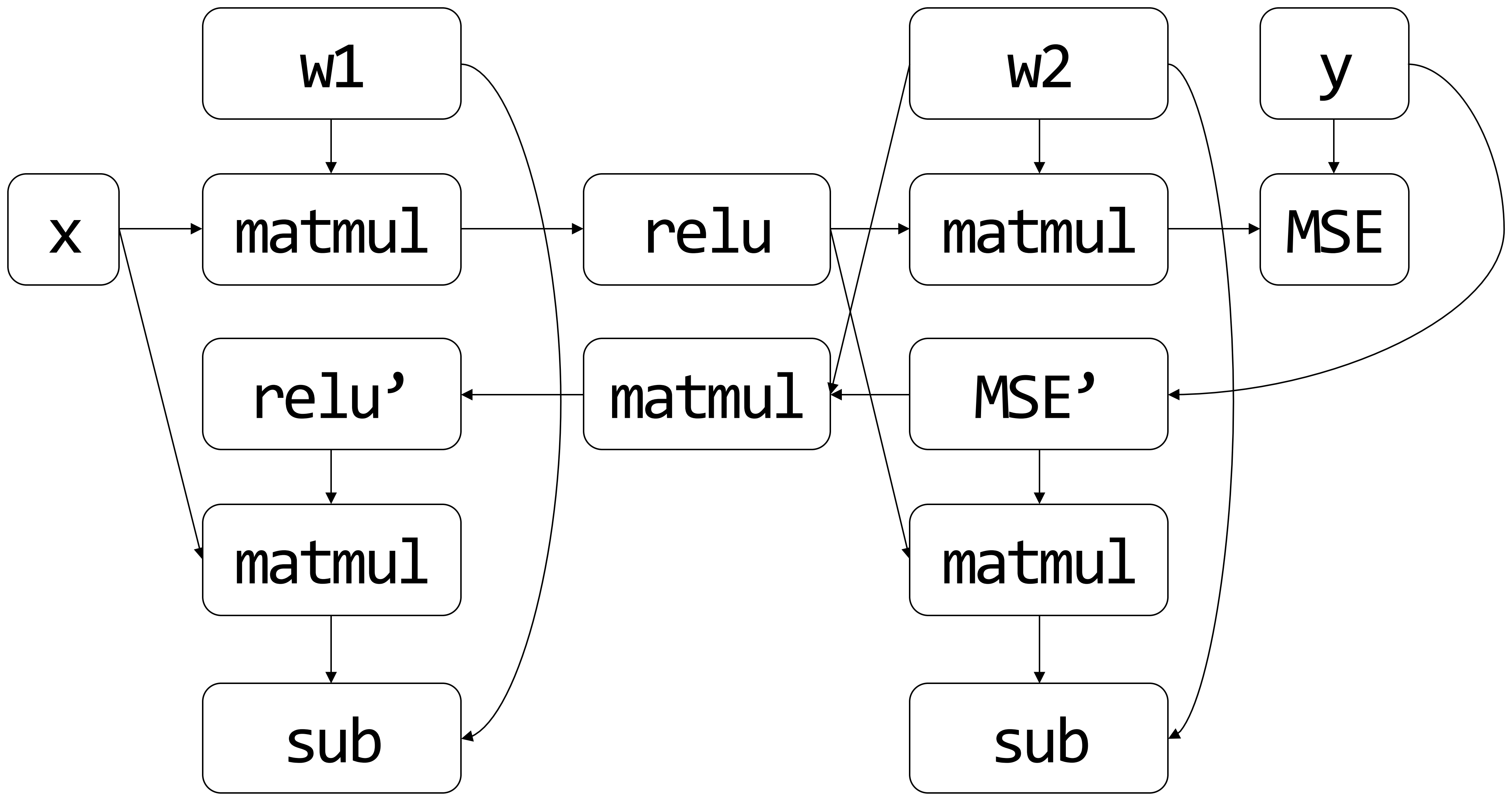
How to perform graph optimization?

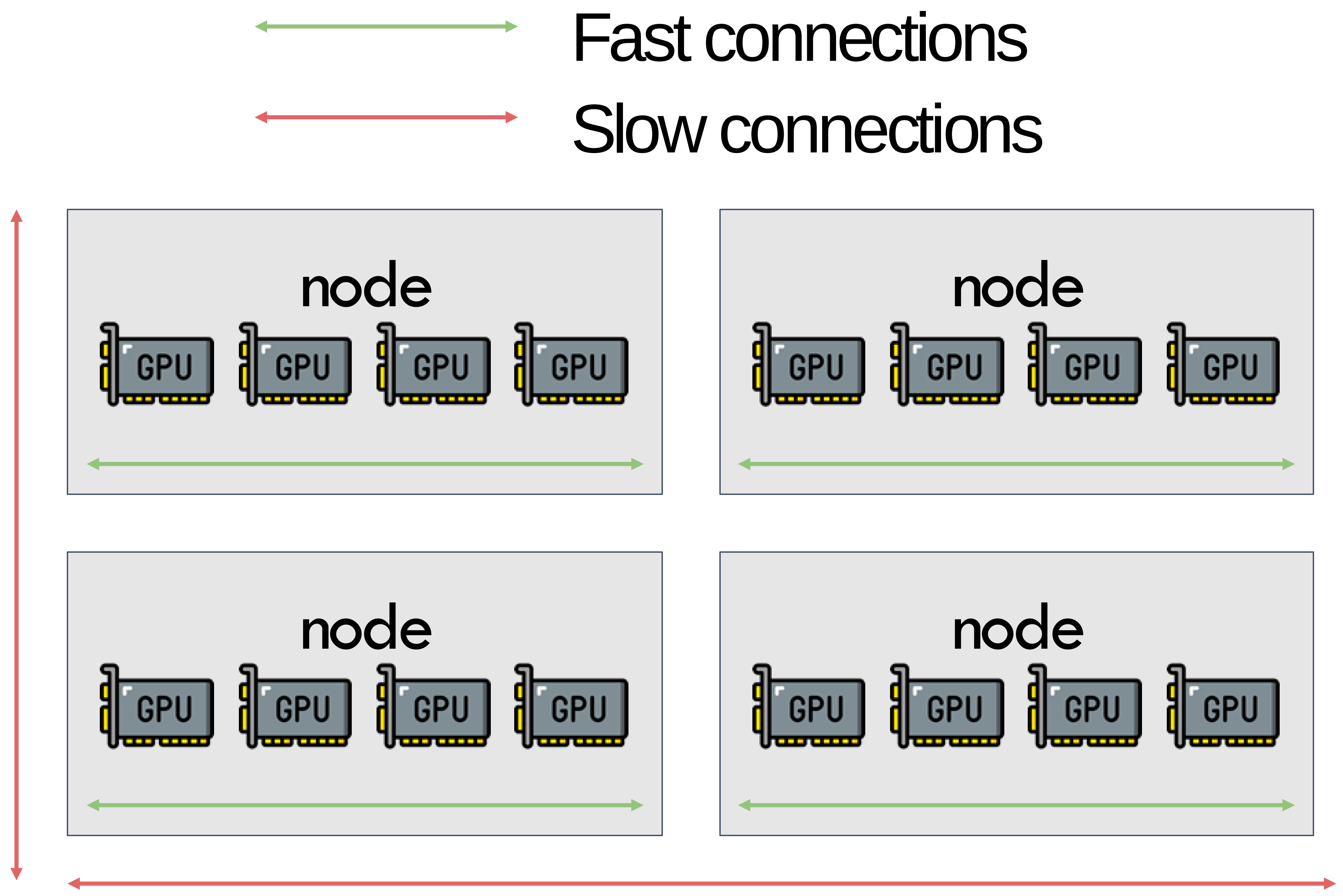
- Writing rules / template
- Auto discovery

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

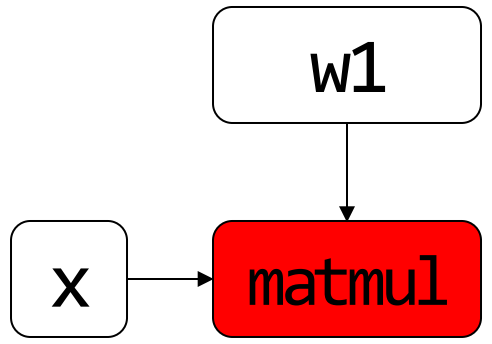
❓ $\{x_i\}^n{}_{i=1}$

✅ Math primitives (mostly matmul)

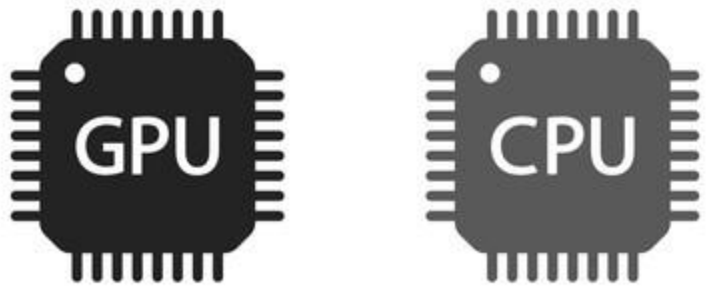
❓ Make them run on (clusters of) different kinds of hardware

✅ A repr that expresses the computation using primitives

Next: How to make operators run (fast) on devices?



- Dataflow Graph
- Autodiff
- Graph Optimization
- Parallelization
- Runtime: schedule / memory
- Operator optimization/compilation



Our Goal in This Layer: Maximize Arithmetic Intensity

max AI = #ops / #bytes

Next

- How we can make operator fast in general
- Case study: Matmul
- GPU architecture and programming

How we can make operators fast in general

- Vectorization
- Data layout
- Parallelization

Using vectorized operations: array add

Why vectorized is faster than unvectorized?

```
Float A[256], B[256], C[256]
For (int i = 0; i < 256; ++i) {
   C[i] = A[i] + B[i]
}
```

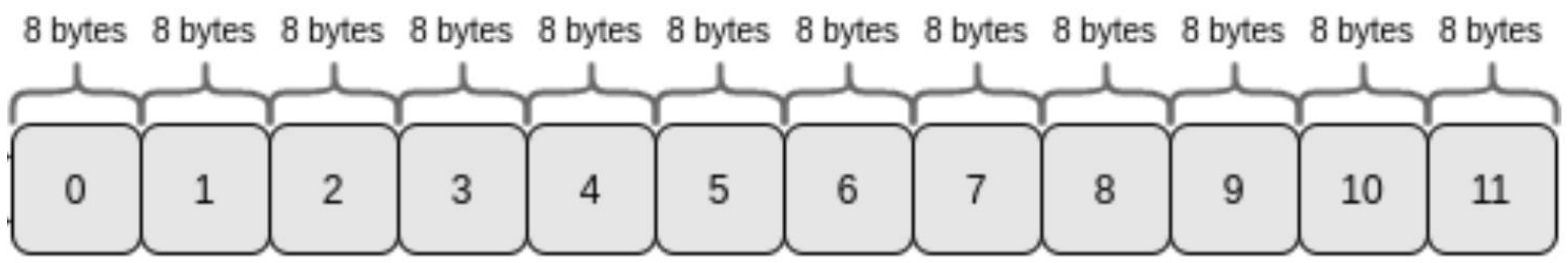
unvectorized

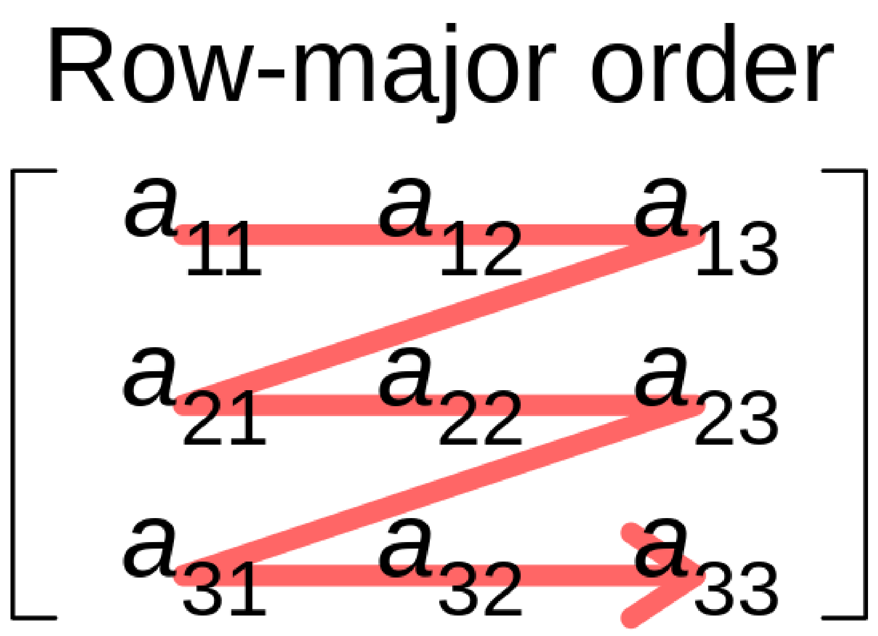
```
for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C + i* 4, c);
```

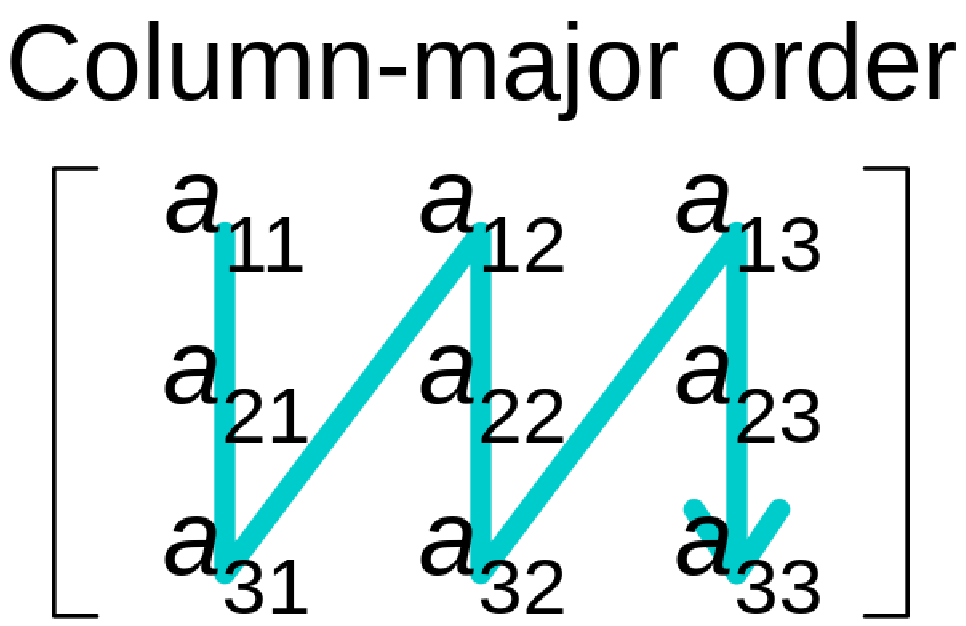
vectorized

Data Layout: make read/write faster

- How to store a matrix in memory
  - Data in memory are stored sequentially (no tensor awareness)
- Row Major: A[i, j] = A.data[i*A.shape[1] + j]
- Column major: A[i, j] = A.data[j*A.shape[0] + i]







Be aware of your data layout

Assuming row-major array

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

a [0] [0]	. . .	a [0] [N-1]	a [1] [0]	. . .	a [1] [N-1]	. . .	a [M-1] [0]	. . .	a [M-1] [N-1]

How to improve the above program?

MCQ Time

Data

? $\{x_i\}^n{}_{i=1}$

ML Systems Store Data in:

A. Row major

B. Col major

C. Strides format: A[i, j] = A.data[offset + i*A.strides[0] + j * A.strides[1]]

Strides in High-dimension

Offset: the offset of the tensor relative to the underlying storage

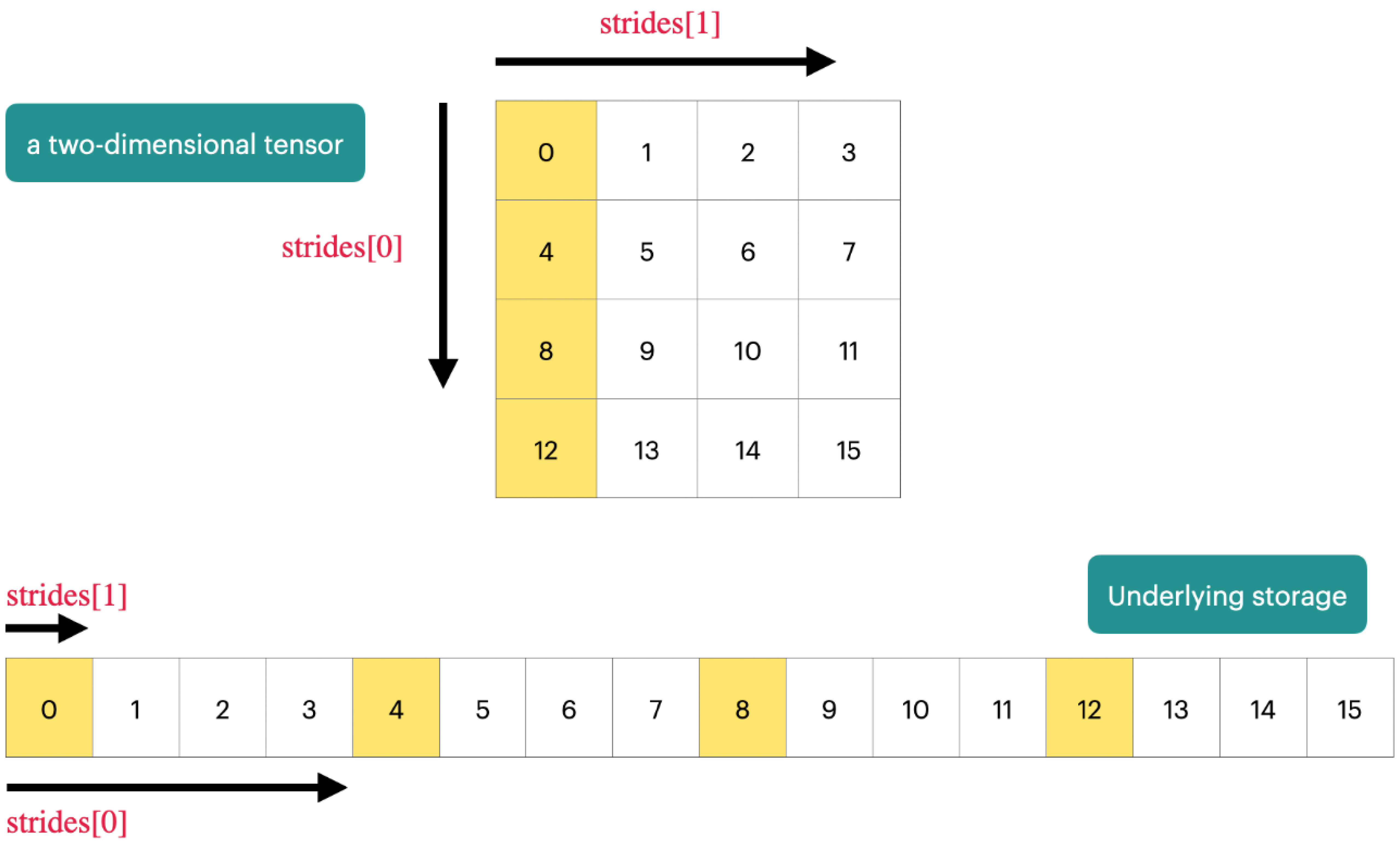
Strides: strides[i] indicates how many "elements" need to be skipped in memory to move "one unit" in the i-th dimension of the tensor

Python

```python
A[i0][i1][i2]... = A_internal[
    stride_offset
    + i0 * A.strides[0]
    + i1 * A.strides[1]
    + i2 * A.strides[2]
    + ...
    + in-1 * A.strides[n-1]
]
```

Strides format

- What we have when:
  - A.strides[0] = 1,
  - A.strides[1] = A.shape[0]?
- What we have when:
  - A.strides[0] = A.shape[1]
  - A.strides[1] = 1,
- Strides offers more flexibility



Questions

- If a tensor of shape [1, 2, 3, 4] is stored contiguous in memory following row Major, write down its strides?

```
torch.arange(0, 24).reshape(1, 2, 3, 4)
print(t)
# tensor([[[[ 0,  1,  2,  3],
#           [ 4,  5,  6,  7],
#           [ 8,  9, 10, 11]],

#          [[12, 13, 14, 15],
#           [16, 17, 18, 19],
#           [20, 21, 22, 23]]]])

print(t.stride())
# (24, 12, 4, 1)
```