https://hao-ai-lab.github.io/dsc291-s24/

DSC 291: ML Systems
Spring 2024

LLMs

Parallelization

Single-device Optimization

Basics

Next Quiz: Thursday (4/11)

- TA to help test IClicker

Two forms worth your attention

- Beginning of quarter survey
  - ?% have filled the survey – please fill to earn the 0.5%

In-Class Quiz

Consist of 2 Components:

- Attendance check-in on iClicker app
- 15 minute quiz on Gradescope (UCSD email)
  - Will go over quiz in class after

Need to complete both to get credit

Quiz will open at 5:00PM and close at 5:15PM.

Without checking in on iClicker you cannot get credit!

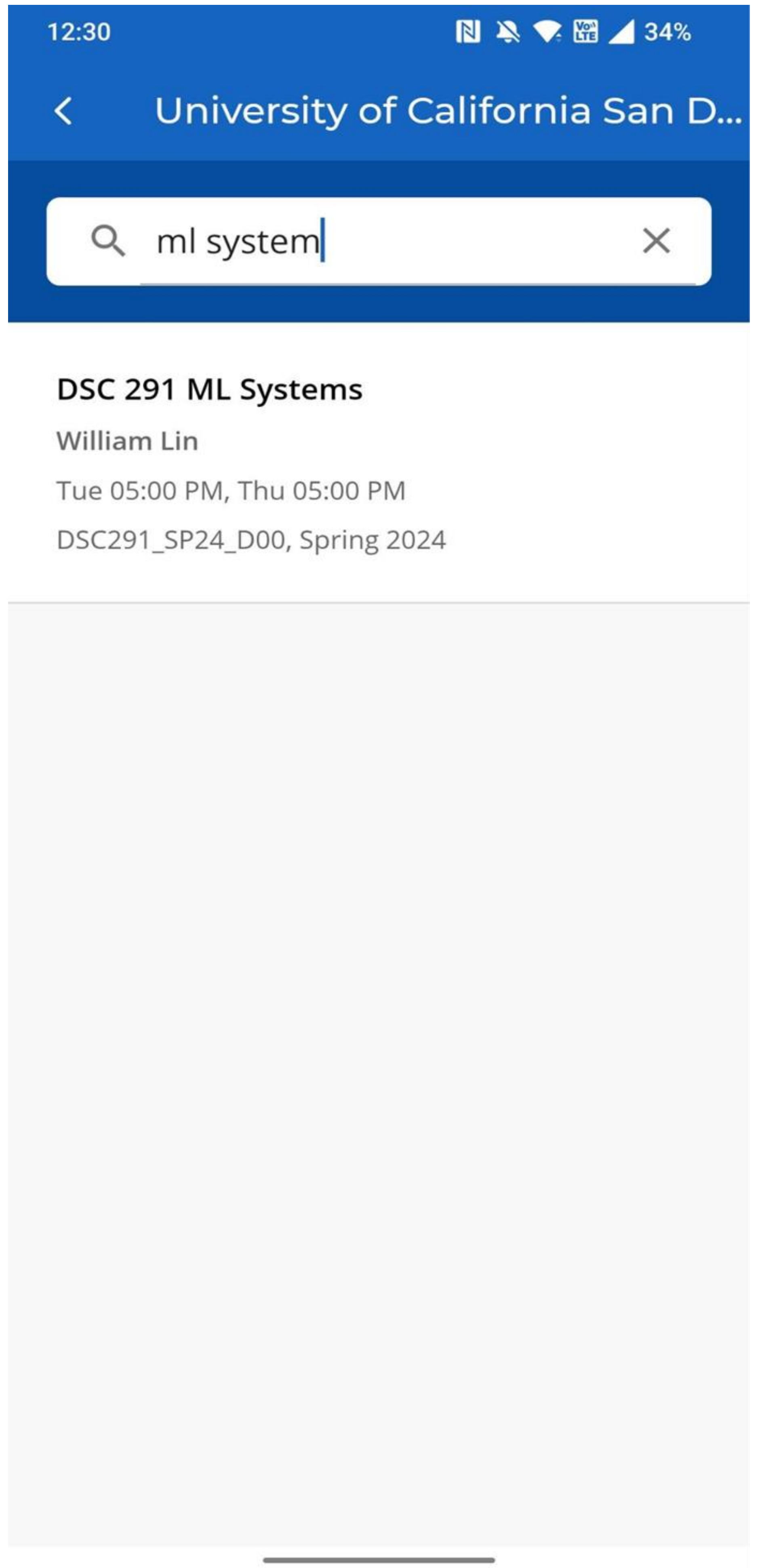
SCAN ME

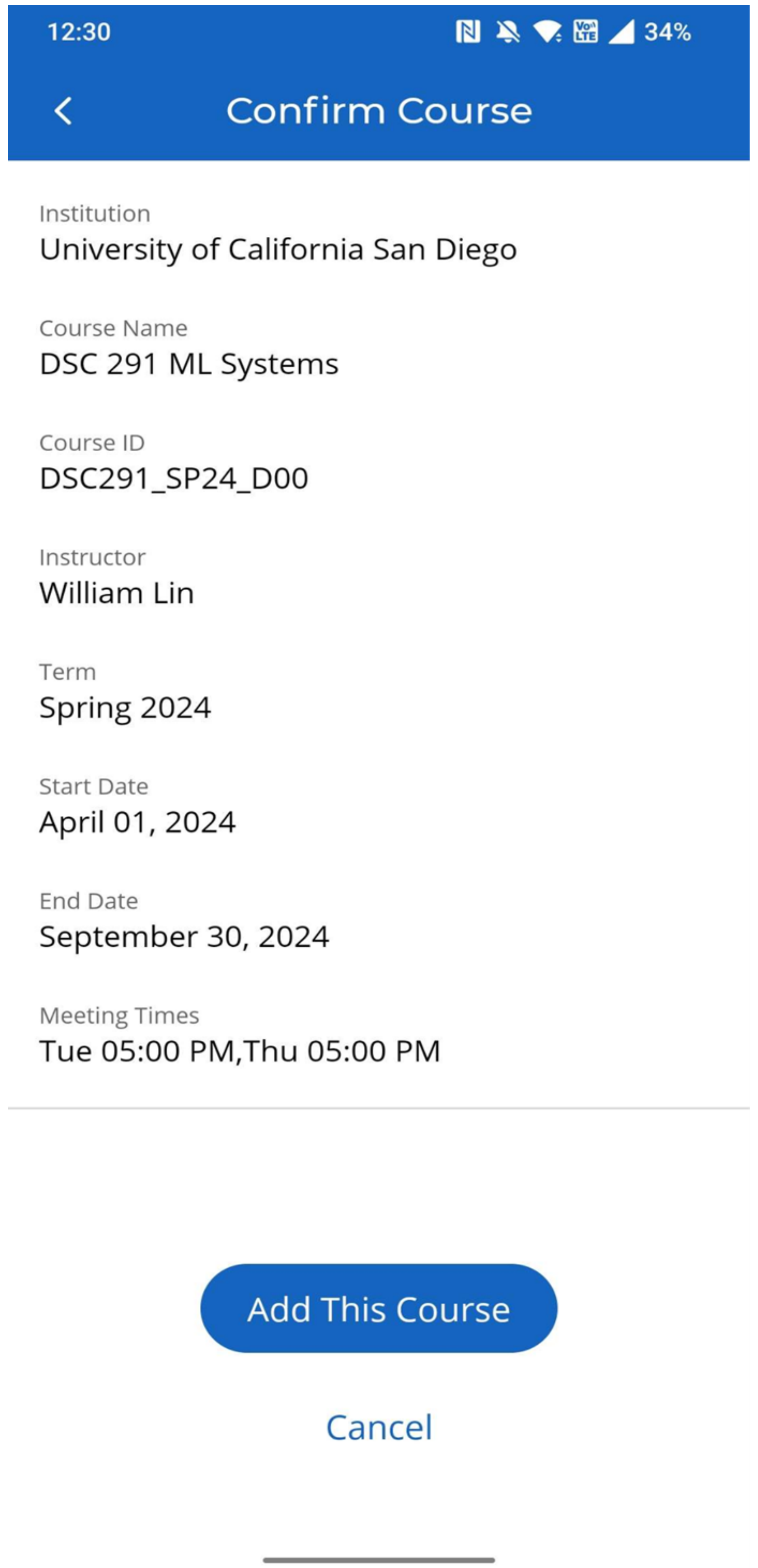
Try to check-in now

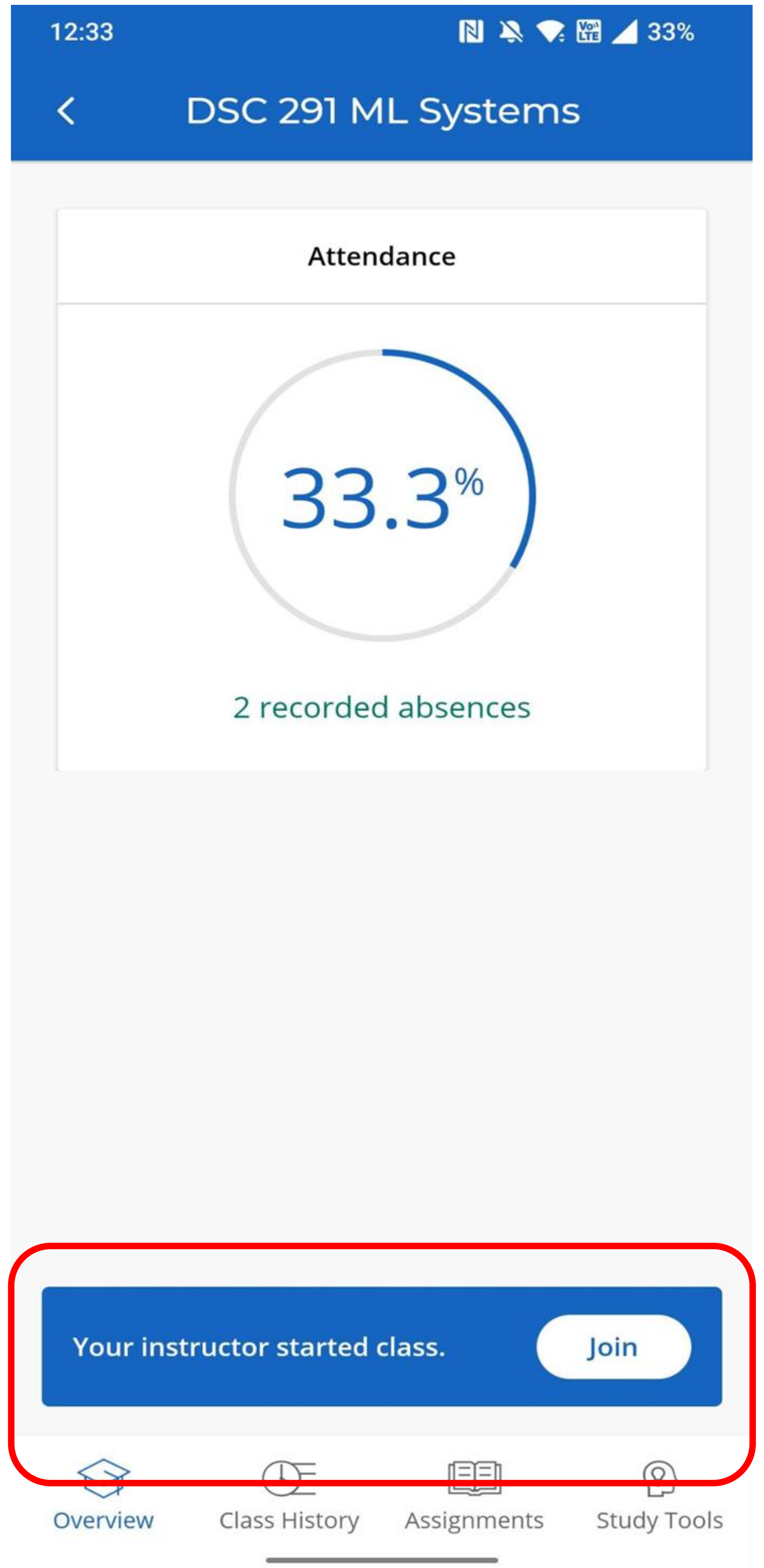
We are using iClicker App for attendance!

Try to check-in now

- Check-in to DSC 291 ML Systems on iclicker app







SCAN ME

Who originally developed PyTorch?

OpenAI

Google DeepMind

aws

Meta AI

Recap

- Understand our Workloads: Deep Learning
  - CNNs/RNNs/GNNs/Transformers/MoE
  - The most important operator: matmul
- Dataflow graph
  - Node: operator (e.g., matmul) and its output tensor
  - Edge: dataflowing directions and dependency
- Programming flavors
  - Define-then-run (Symbolic) and Define-and-run (Imperative)
  - Static and dynamic

Today

- Auto-differentiation
- Concurrent ML Systems architecture overview

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

Numerical differentiation: gradient checking

$$\hat{g}(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \cancel{+ o(\epsilon^2)}$$

1. Implement your own $\mathrm{g}(*)$
2. Substitute θ to get $\mathrm{g}(\theta)$
3. Compare $\hat{g}(\theta)$ and $\mathrm{g}(\theta)$
4. If $\hat{g}(\theta) - \mathrm{g}(\theta) > \delta$, your $\mathrm{g}(\theta)$ might be wrong!

Symbolic Differentiation

Write down the formula, derive the gradient following rules

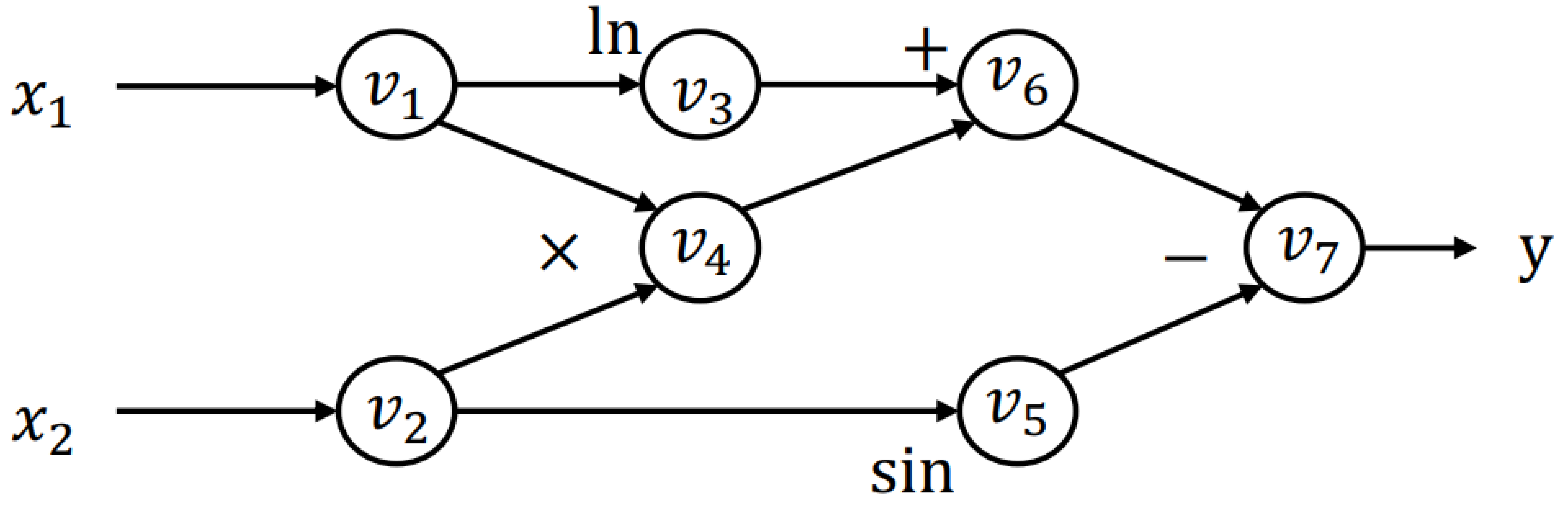
$$\frac{\partial(f(\theta)+g(\theta))}{\partial\theta}=\frac{\partial f(\theta)}{\partial\theta}+\frac{\partial g(\theta)}{\partial\theta}$$

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

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

Map autodiff rules to computational graph

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



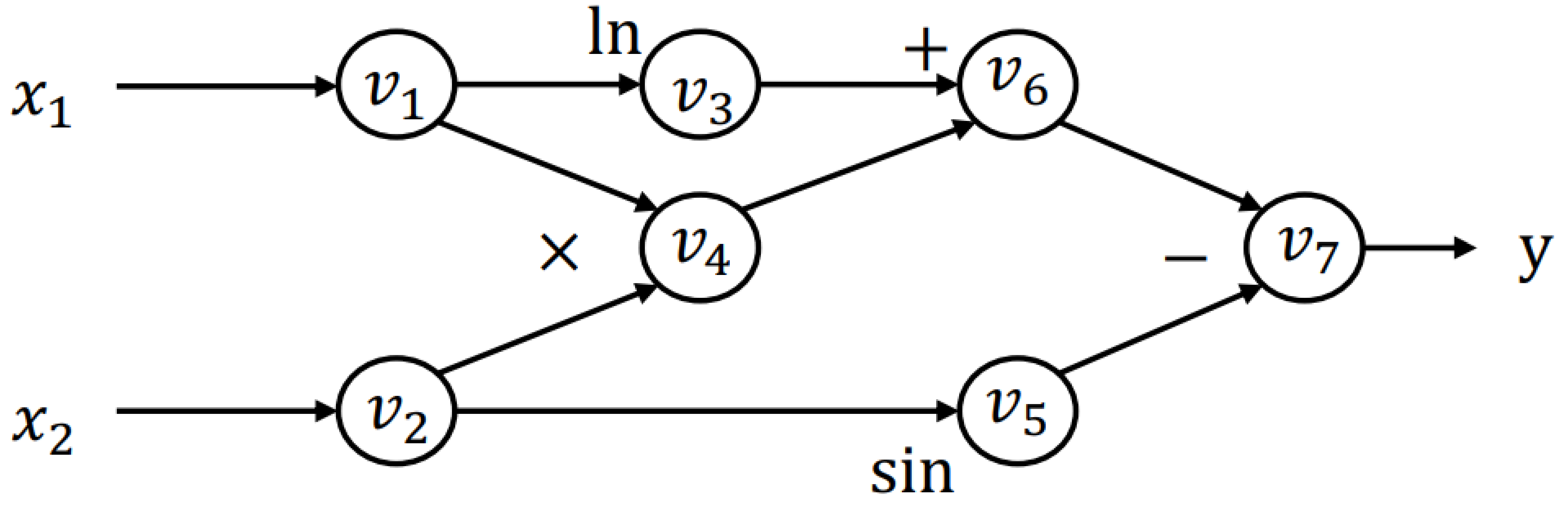
Forward evaluation trace

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

- High-level idea of autodiff:
  - Using chain rules
- There are two ways of autoidff
  - Forward mode autodiff
  - Backward mode autodiff
- Forward mode: Traverse the chain rule from inside to outside
- Backward mode: Traverse the chain rule from outside to inside

Forward Mode Autodiff

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



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

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

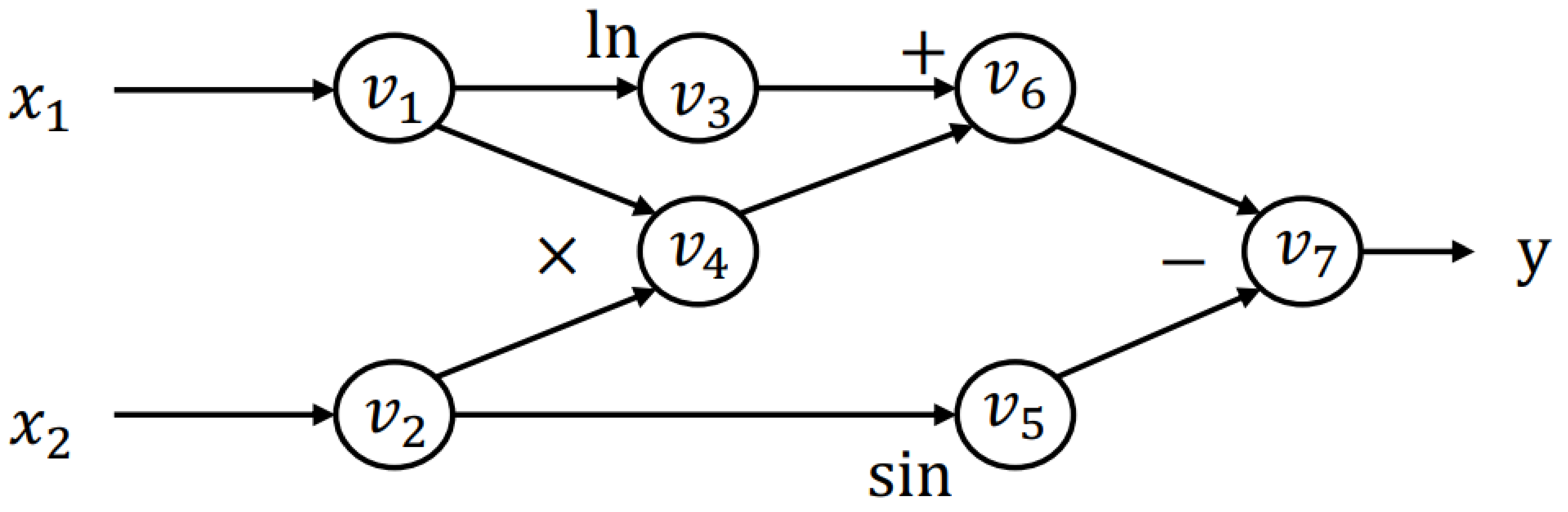
- Finally: $\frac{\partial y}{\partial x_1} = \dot{v}_7 = 5.5$

Summary: Forward Mode Autodiff

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

Reverse Mode Autodiff

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



Forward evaluation trace

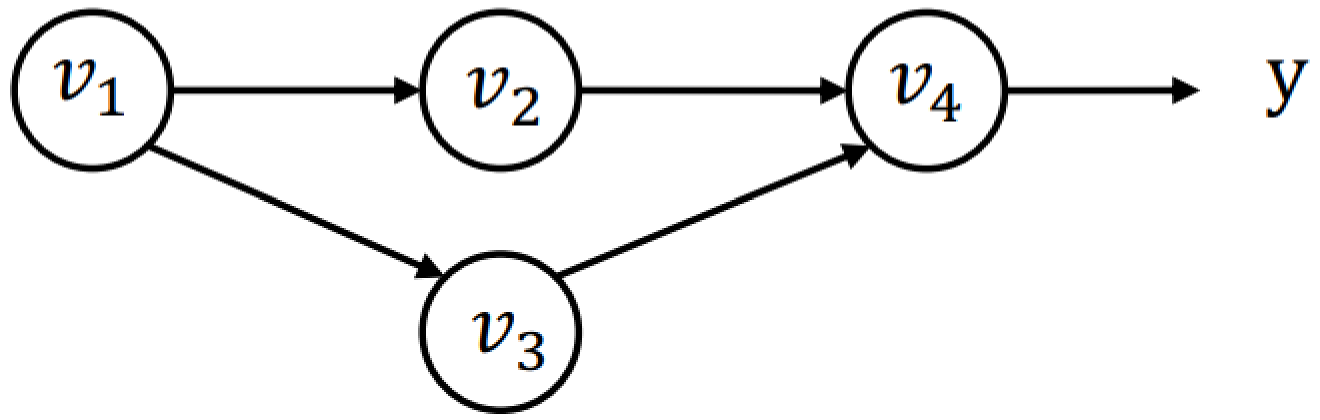
$$v_1 = x_1 = 2$$
$$v_2 = x_2 = 5$$
$$v_3 = \ln v_1 = \ln 2 = 0.693$$
$$v_4 = v_1 \times v_2 = 10$$
$$v_5 = \sin v_2 = \sin 5 = -0.959$$
$$v_6 = v_3 + v_4 = 10.693$$
$$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$$
$$y = v_7 = 11.652$$

- Define adjoint $\overline{v_i} = \frac{\partial y}{\partial x_i}$
- We then compute each $\bar{v}_i$ in the reserve 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:

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

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

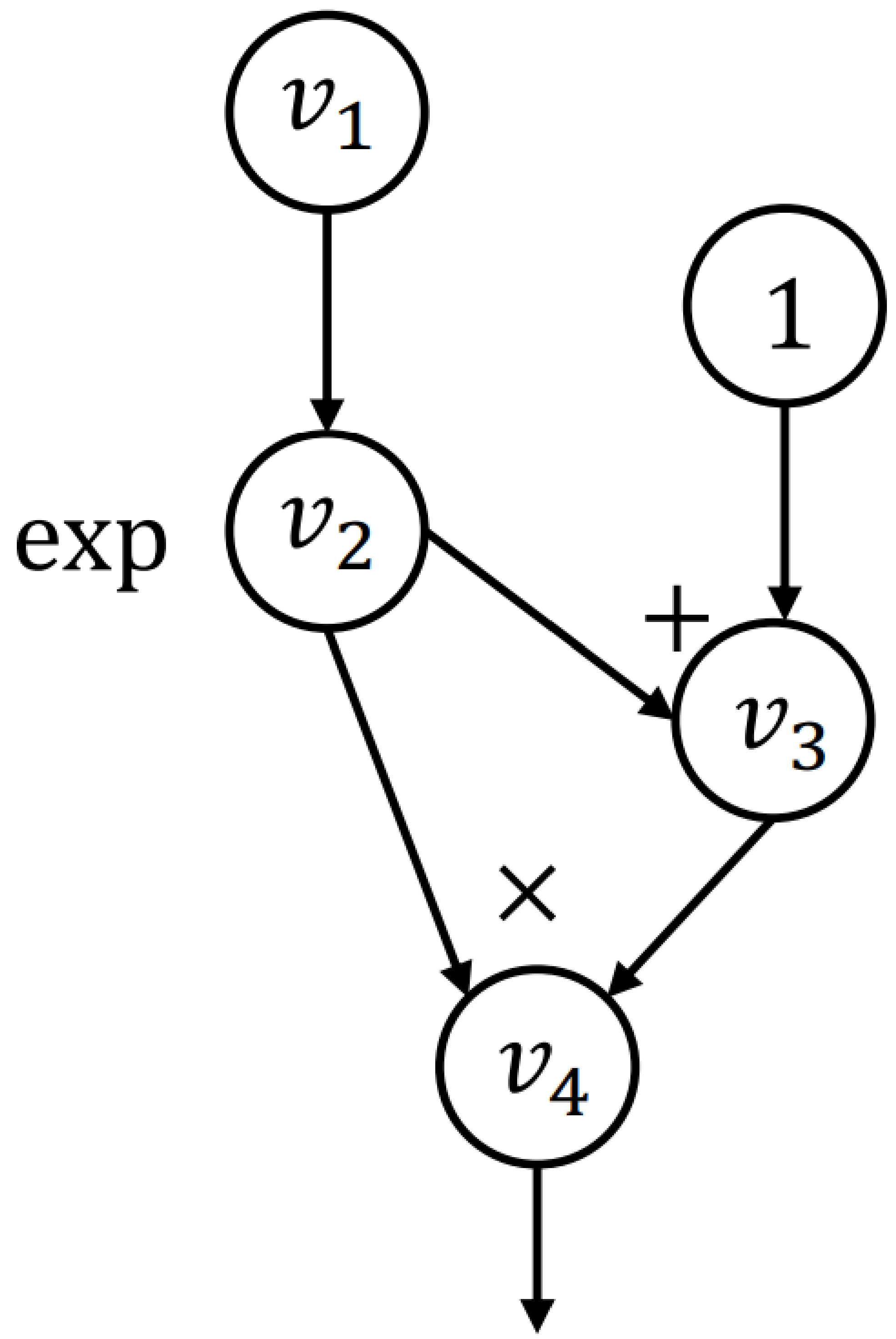
Backward Graph

- How can we construct a computational graph that calculates the adjust 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
```

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

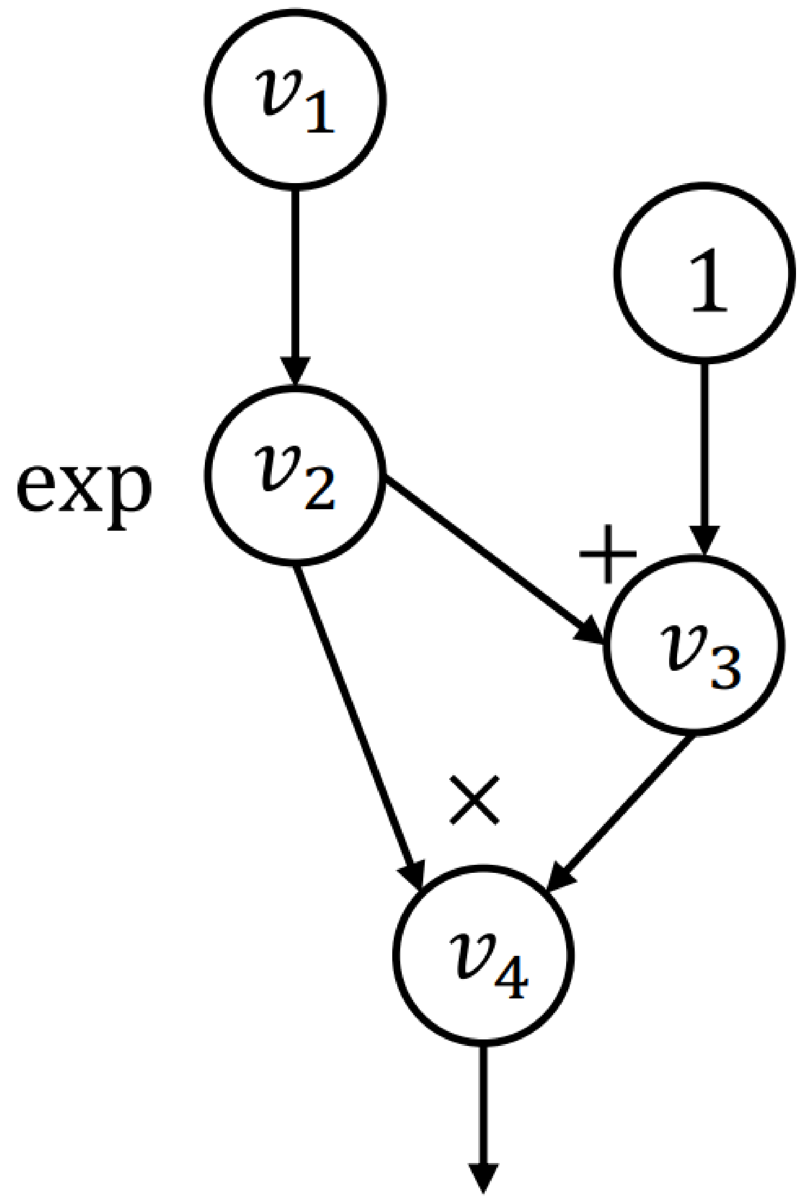
compute $\overline{v_{k\to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$

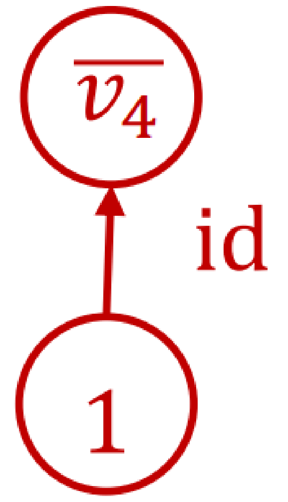


Idea: Just Express Grad Computation using Graph

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



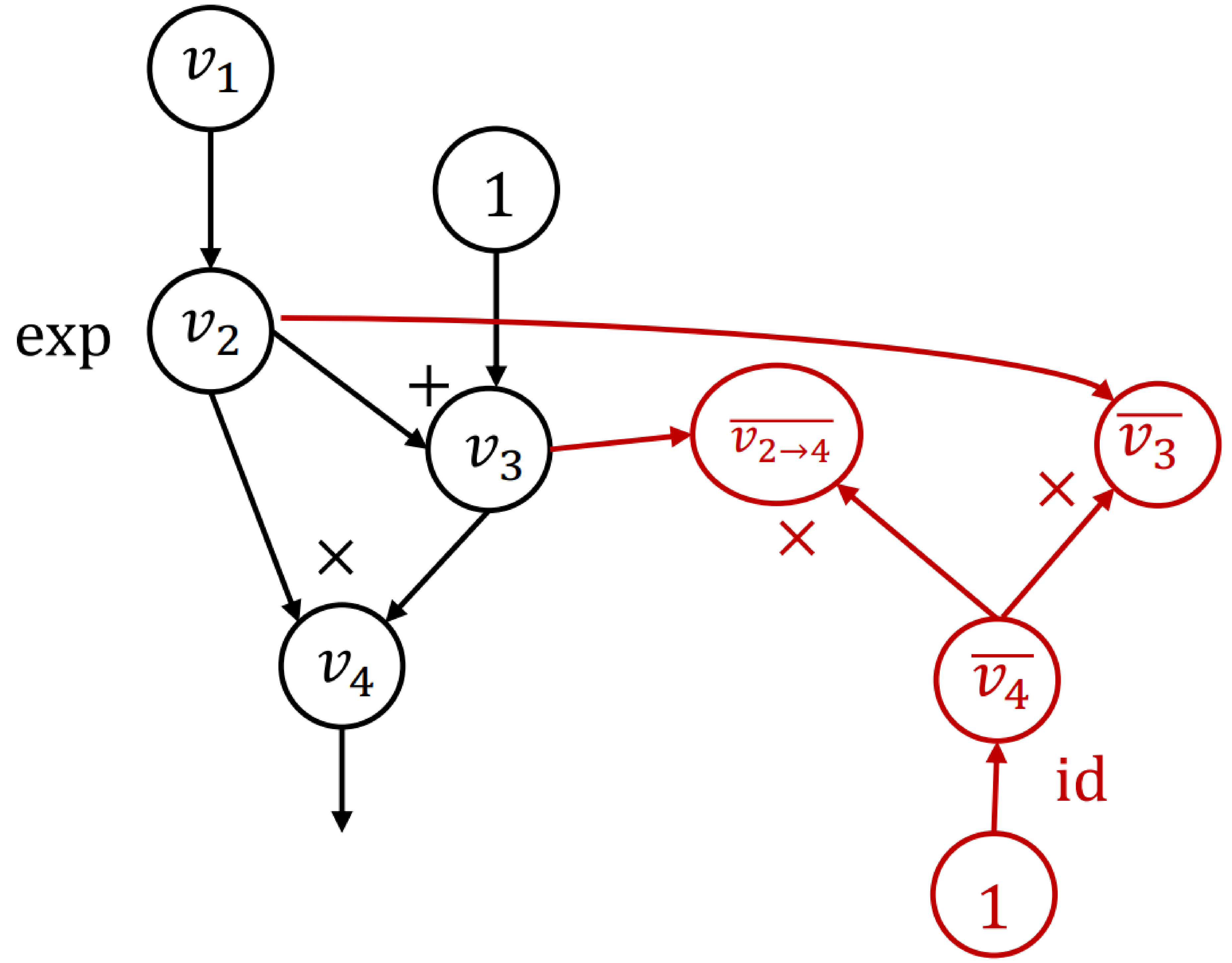


Inspect (v_2, v_4) and (v_3, v_4)

```
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]
}
```

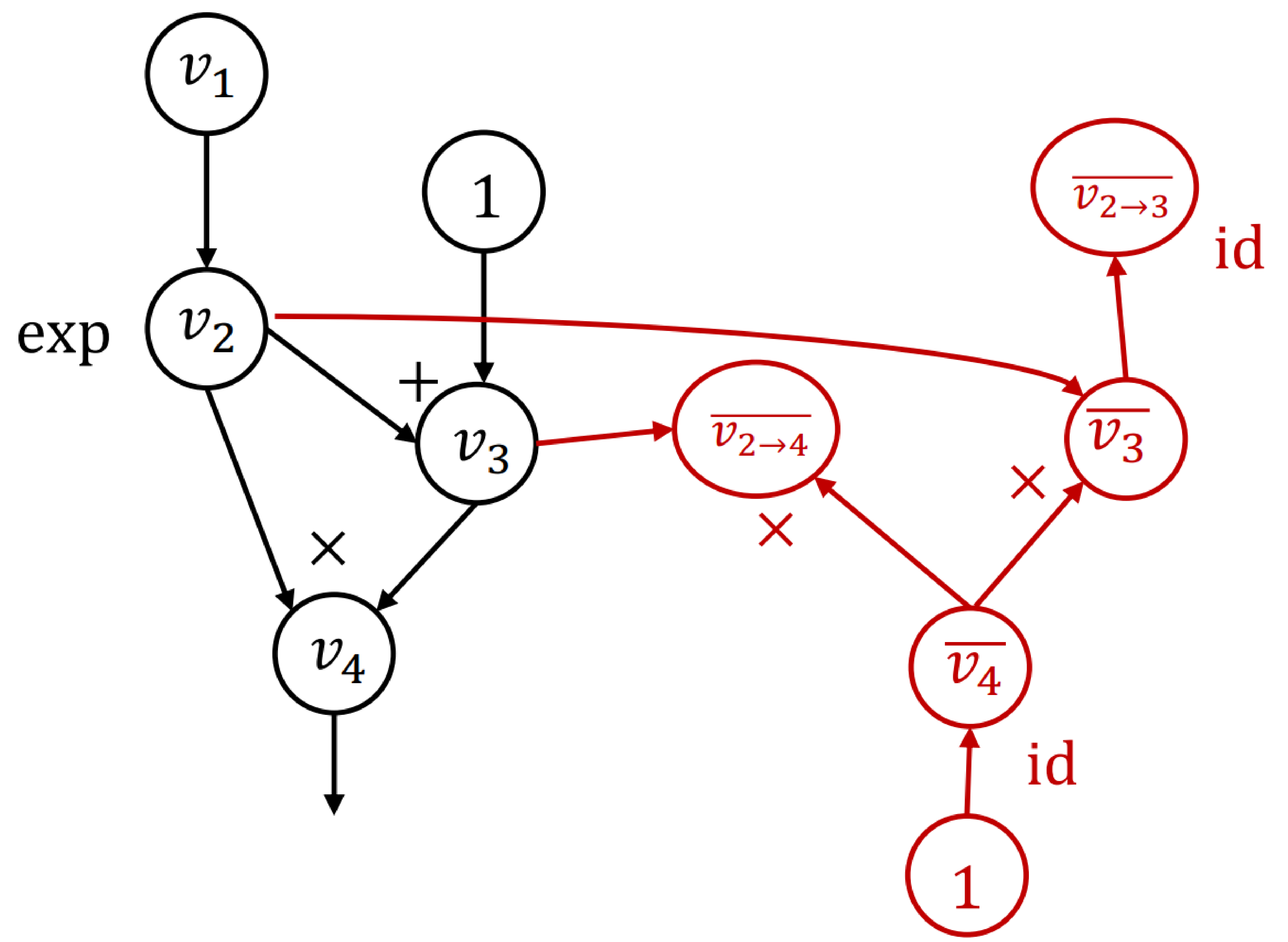


Inspect (v_2, v_3)

```
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]
}
```



Inspect v_2

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 $\overline{v_{input}}$

$i = 2$

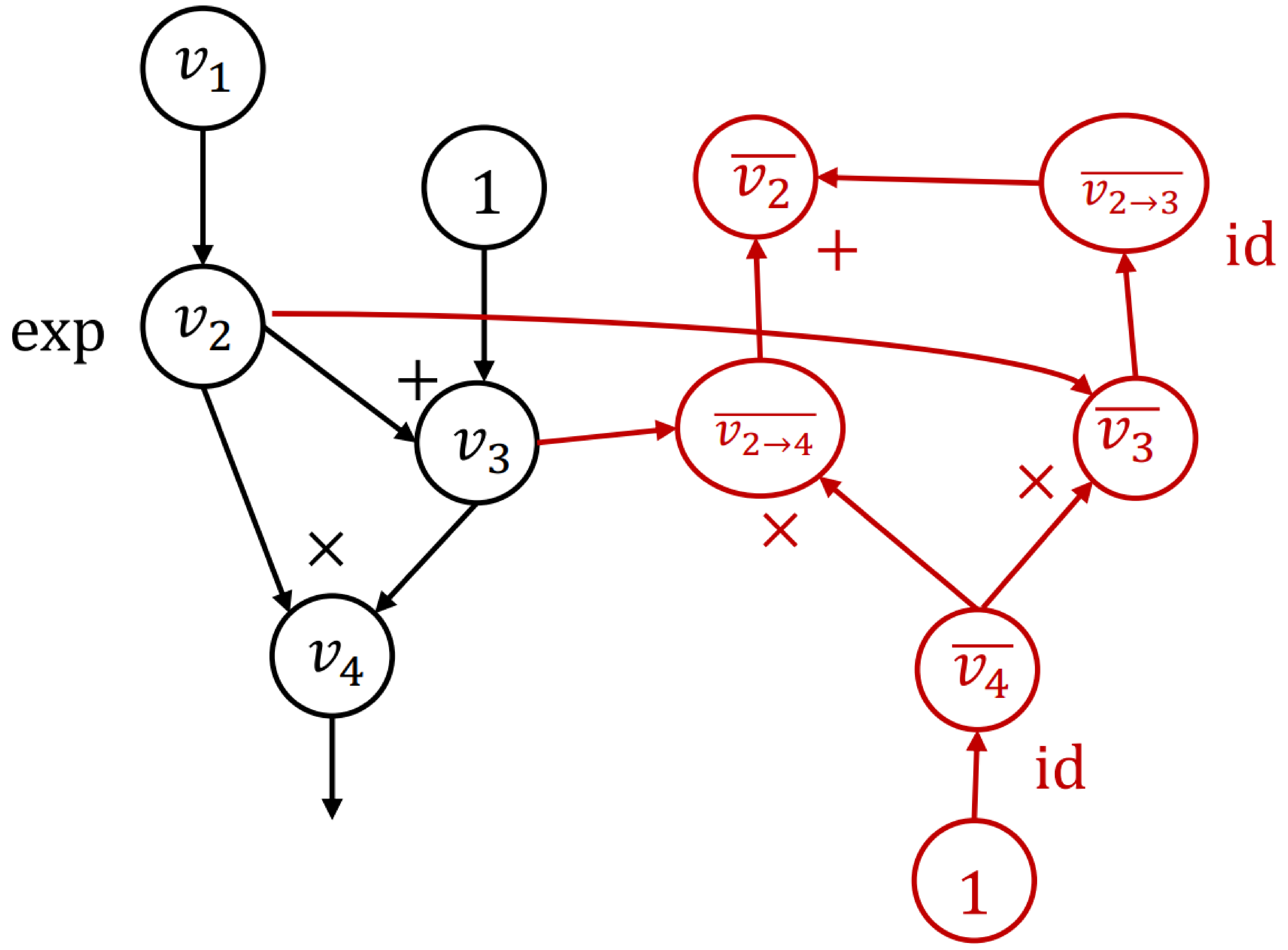
node_to_grad: {

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

  3: $[\overline{v_3}]$

  4: $[\overline{v_4}]$

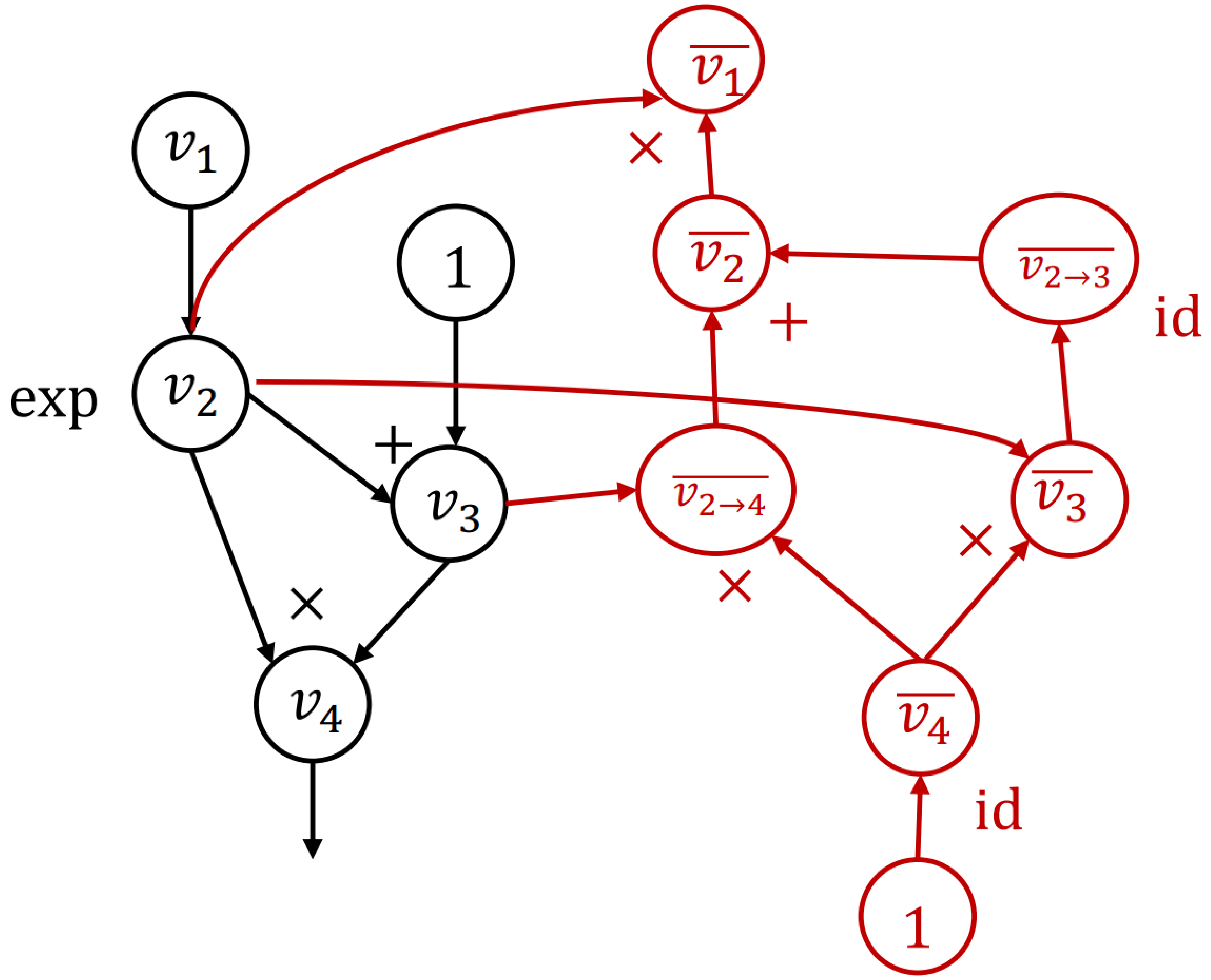
}



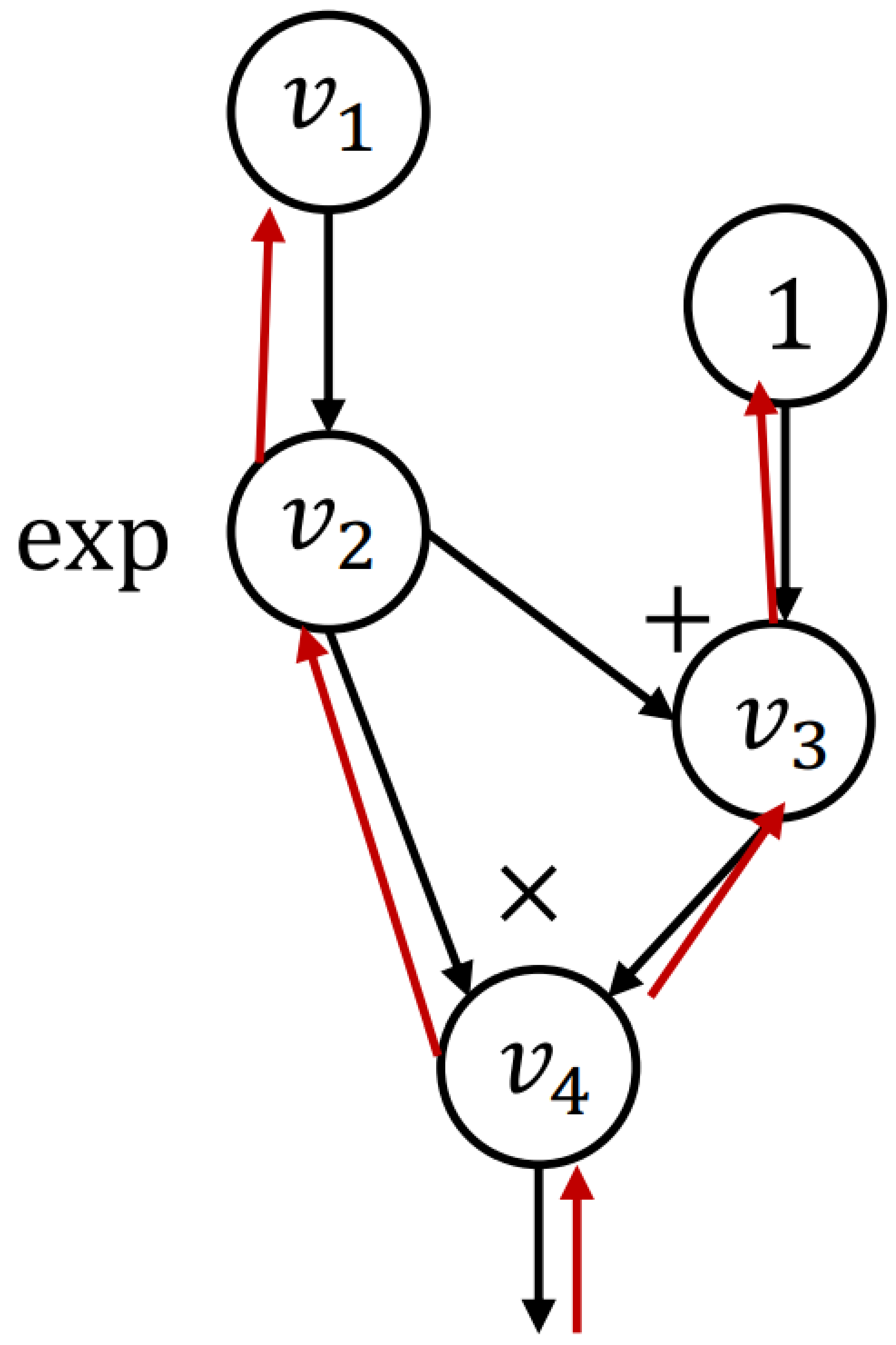
Inspect (v_1, v_2)

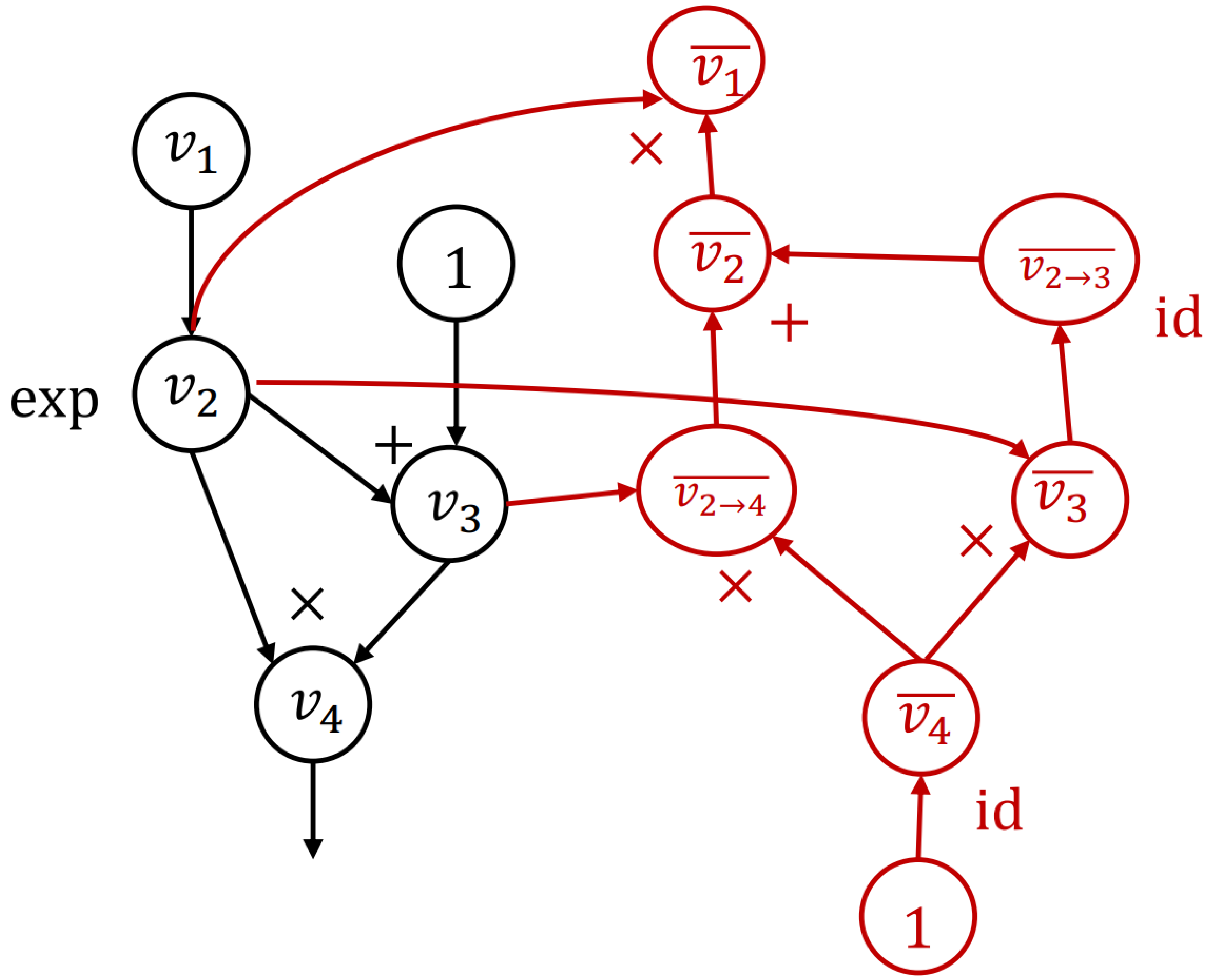
def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out):
    $\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = sum(node_to_grad[i])
    for $k \in inputs(i)$:
      compute $\overline{v_{k\to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$
      append $\overline{v_{k\to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

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



Summary



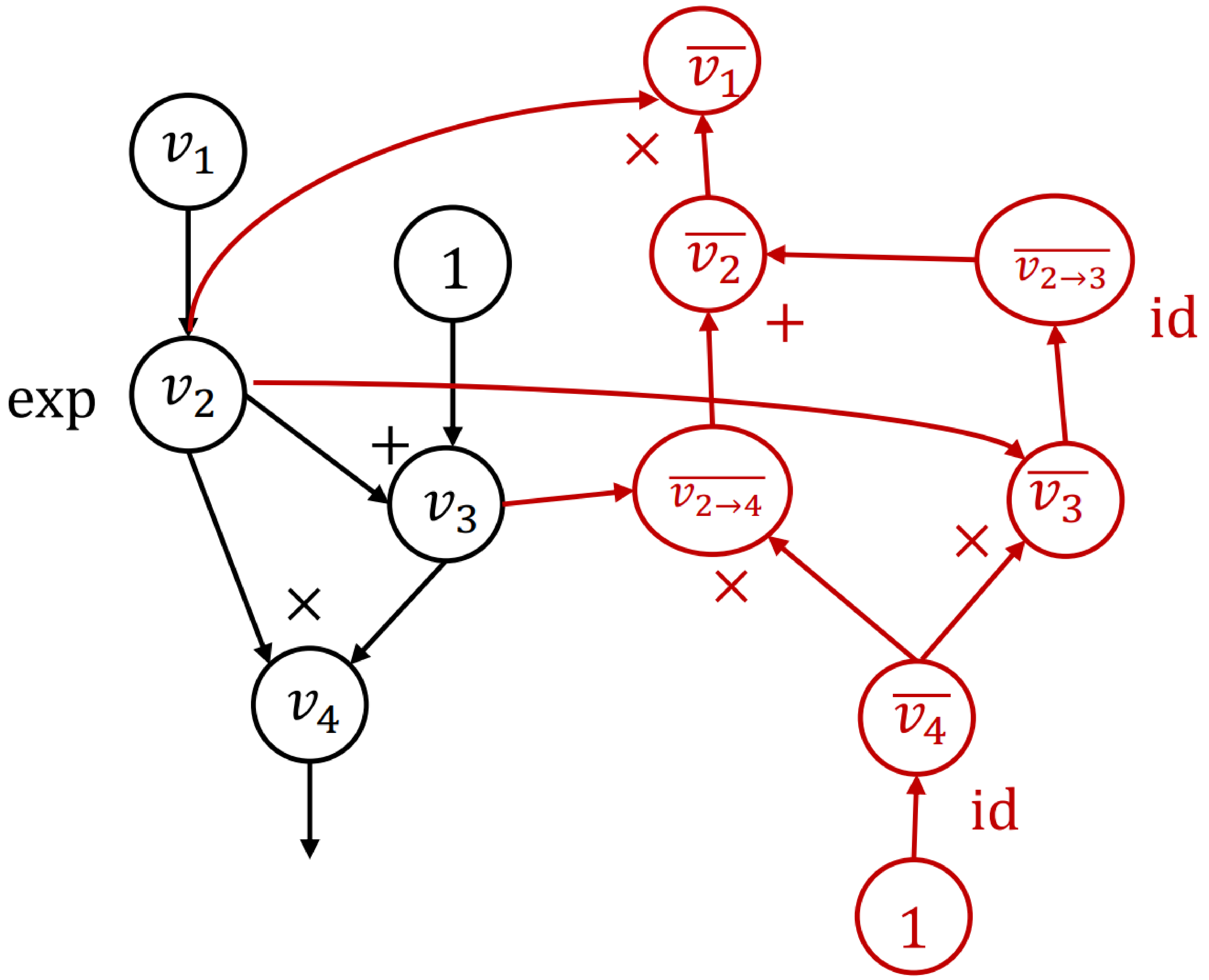


- Run backward through the forward graph
- Caffe/cuda-convnet

- Construct backward graph
- Used by TensorFlow, PyTorch

Incomplete yet?

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



Put in Practice

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

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

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

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

Practice: node are operators now

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

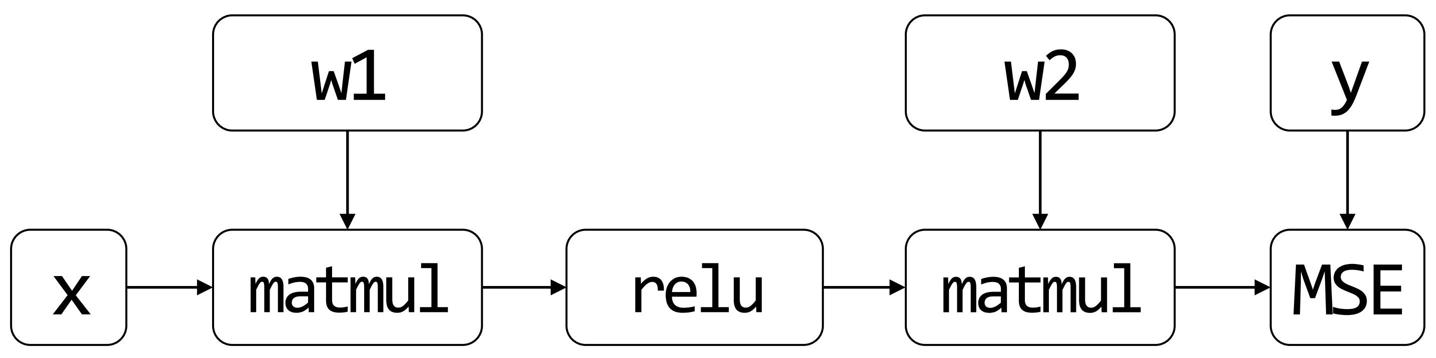
$$L = \mathrm{MSE}(w_2 \cdot \mathrm{ReLU}(w_1 x), y) \quad \theta = \{w_1, w_2\}, D = \{(x, y)\}$$

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

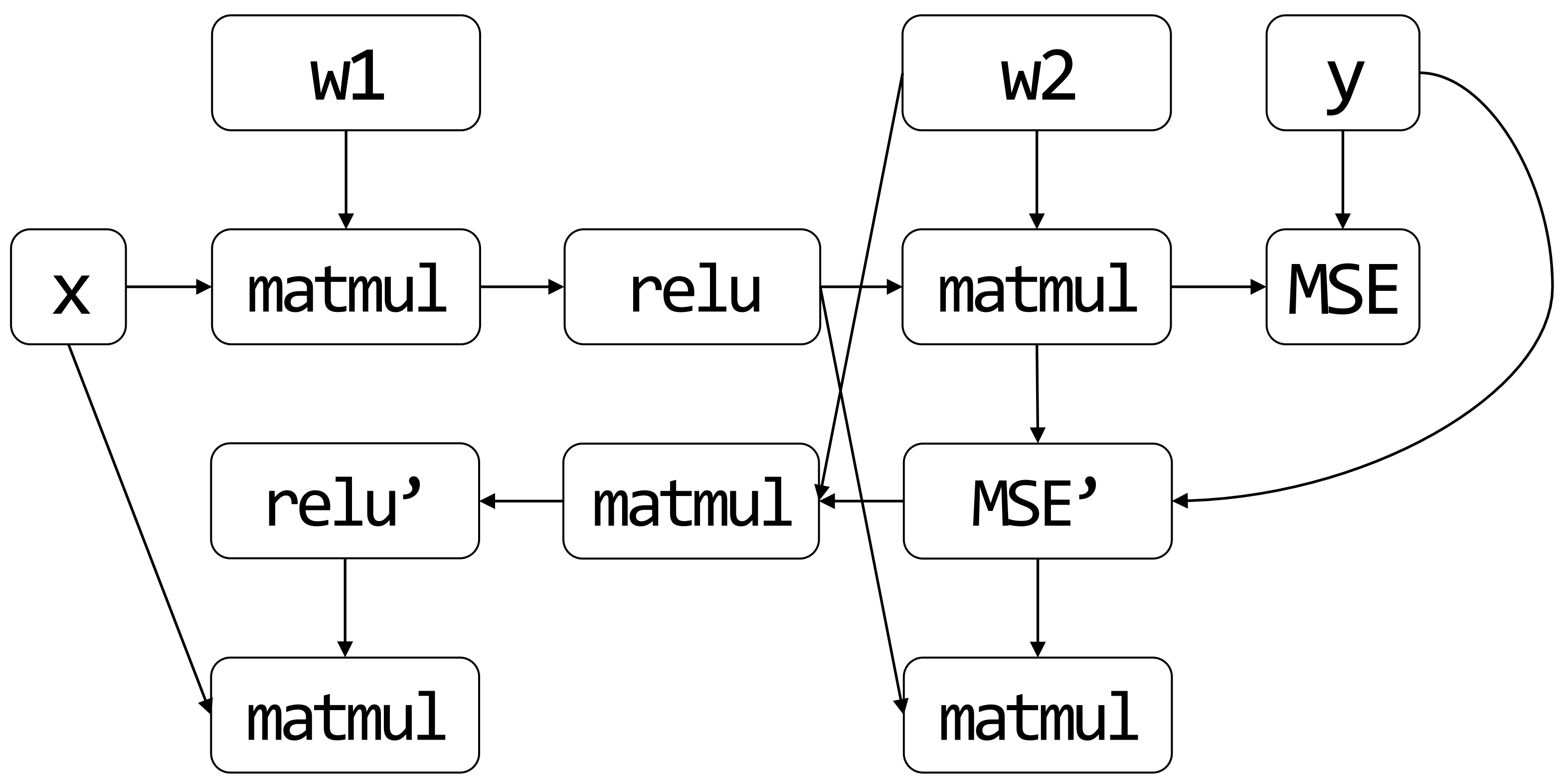
Operator / its output tensor

Data flowing direction

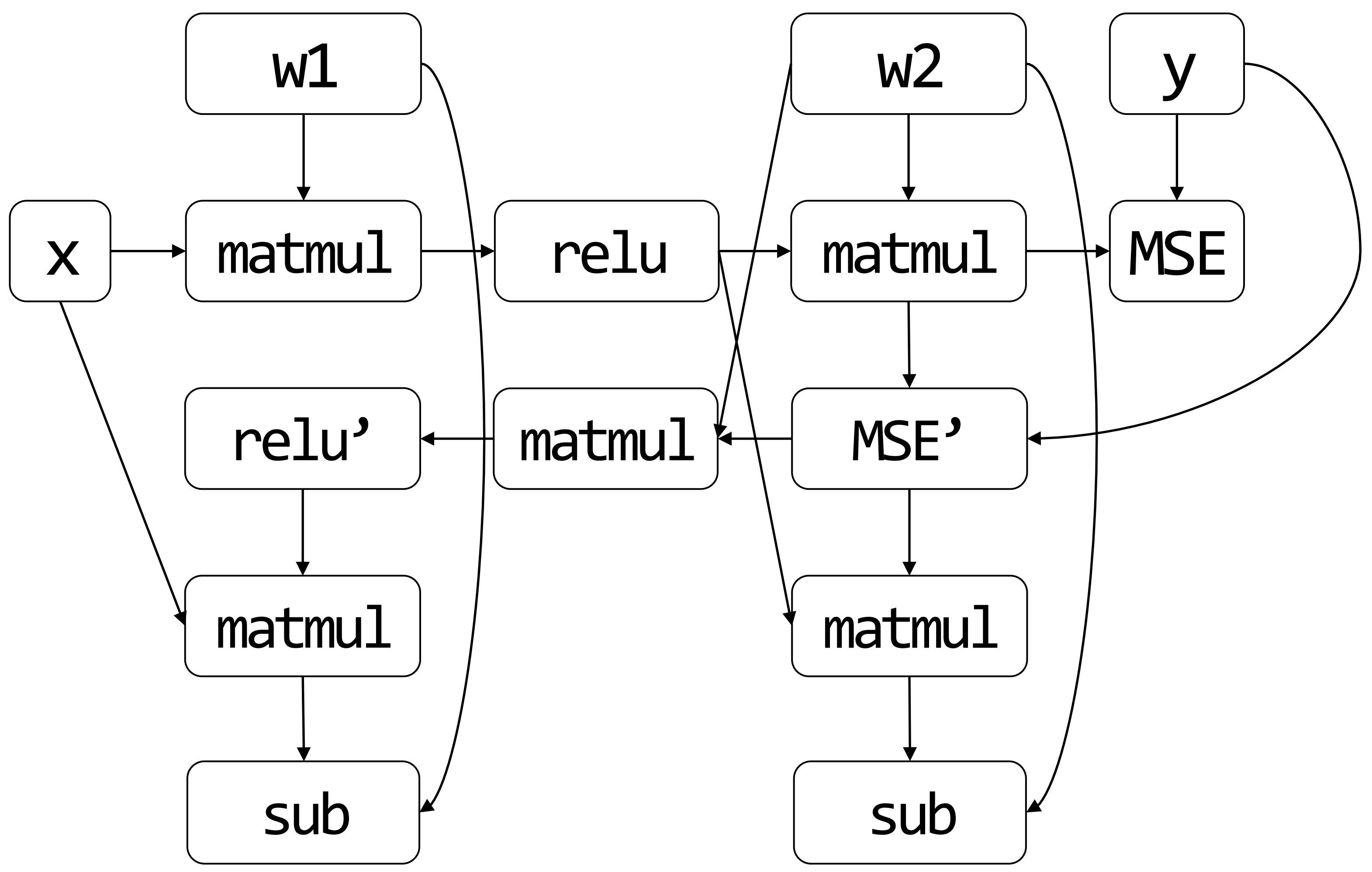
Forward



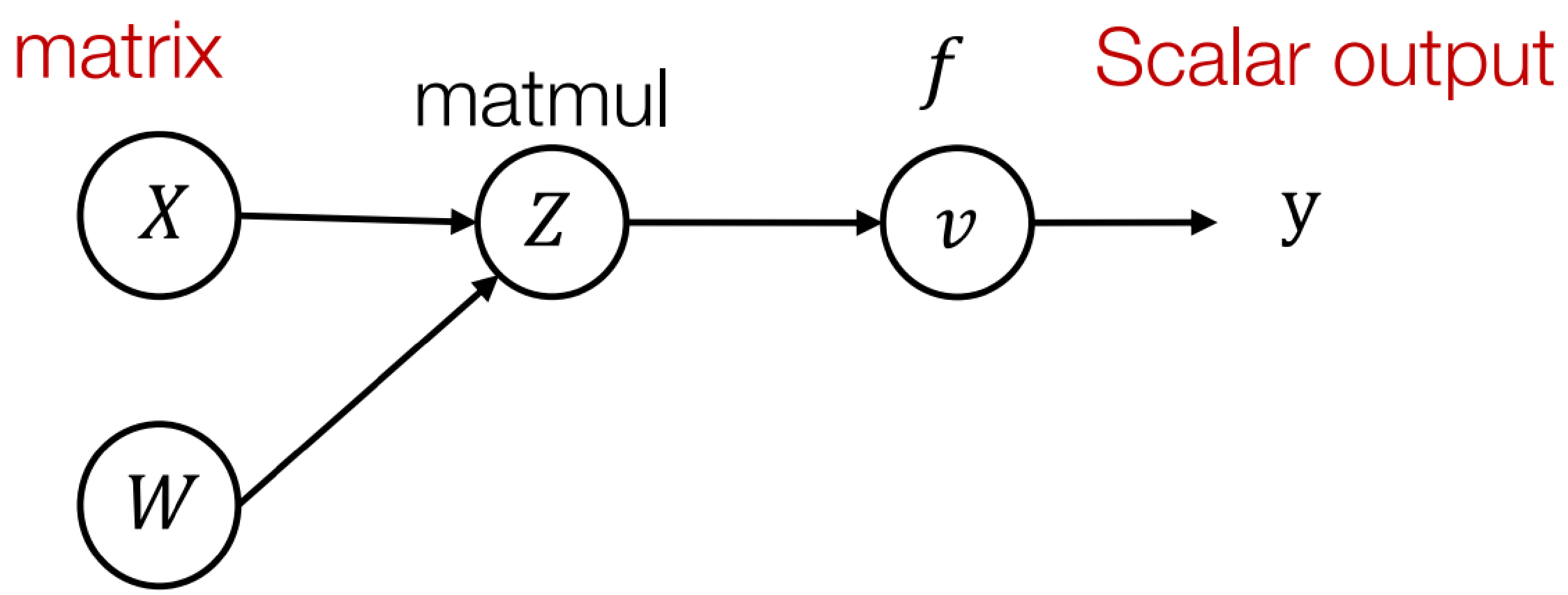
+Backward



+Weight update



1D -> 2D



Define adjoint for tensor values $\bar{Z} = \begin{bmatrix} \frac{\partial y}{\partial Z_{1,1}} & \cdots & \frac{\partial y}{\partial Z_{1,n}} \\ \cdots & \cdots & \cdots \\ \frac{\partial y}{\partial Z_{m,1}} & \cdots & \frac{\partial y}{\partial Z_{m,n}} \end{bmatrix}$

Forward evaluation trace

$$Z_{ij} = \sum_k X_{ik} W_{kj}$$
$$v = f(Z)$$

Reverse evaluation in scalar form

$$\overline{X_{i,k}} = \sum_j \frac{\partial Z_{i,j}}{\partial X_{i,k}} \bar{Z}_{i,j} = \sum_j W_{k,j} \bar{Z}_{i,j}$$

Forward matrix form

$$Z = XW$$
$$v = f(Z)$$

Reverse matrix form

$$\bar{X} = \bar{Z} W^T$$

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?

Homework: How to derive gradients for

- Softmax cross entropy:

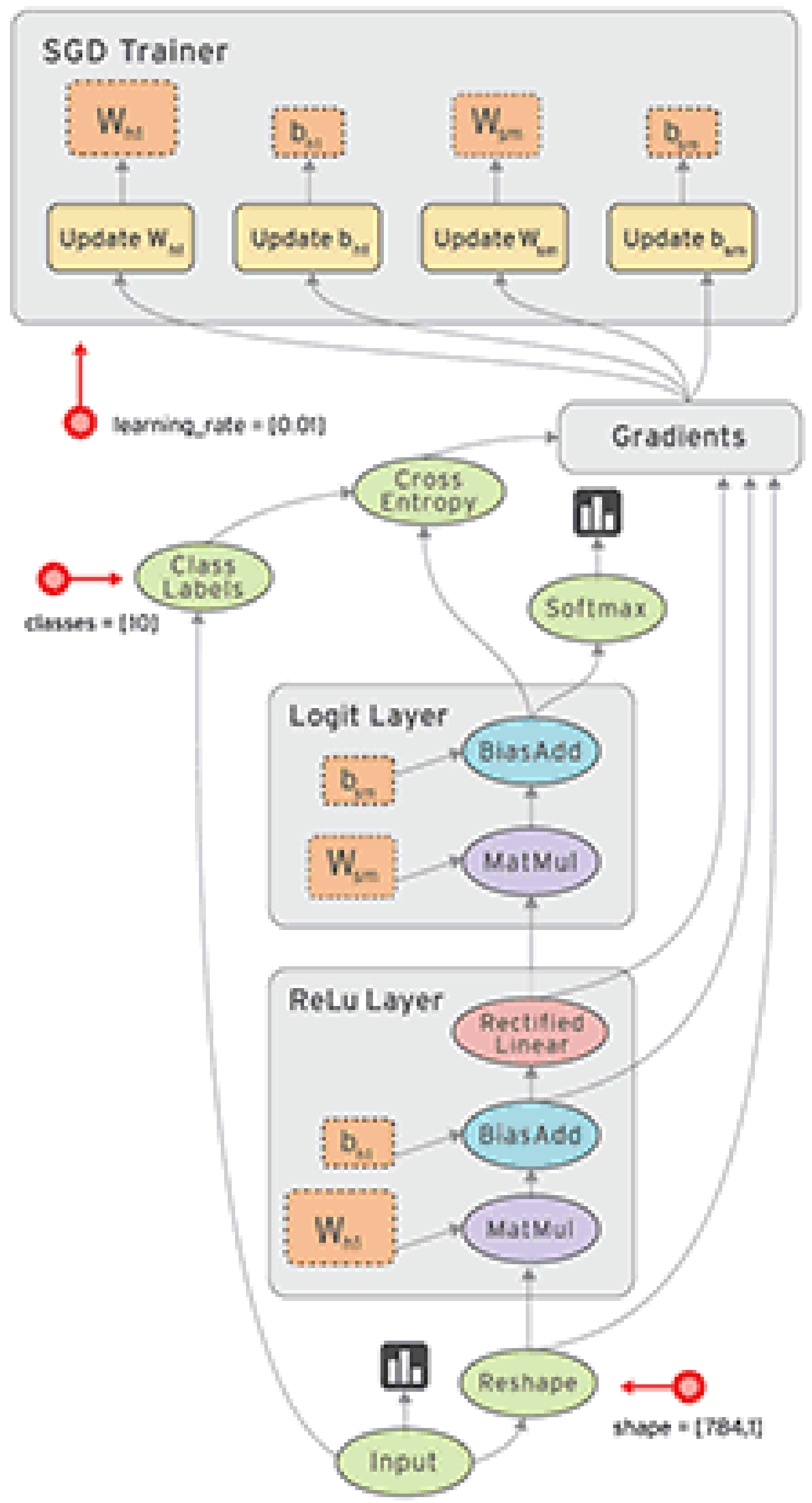
$$L = -\sum t_i \log(y_i)\,, y_i = softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum e^{x_d}}$$

- Sample $i \sim softmax(\boldsymbol{x})_i, z = f(i)$
  - How to derive $\frac{\partial f}{\partial x}$?

Today

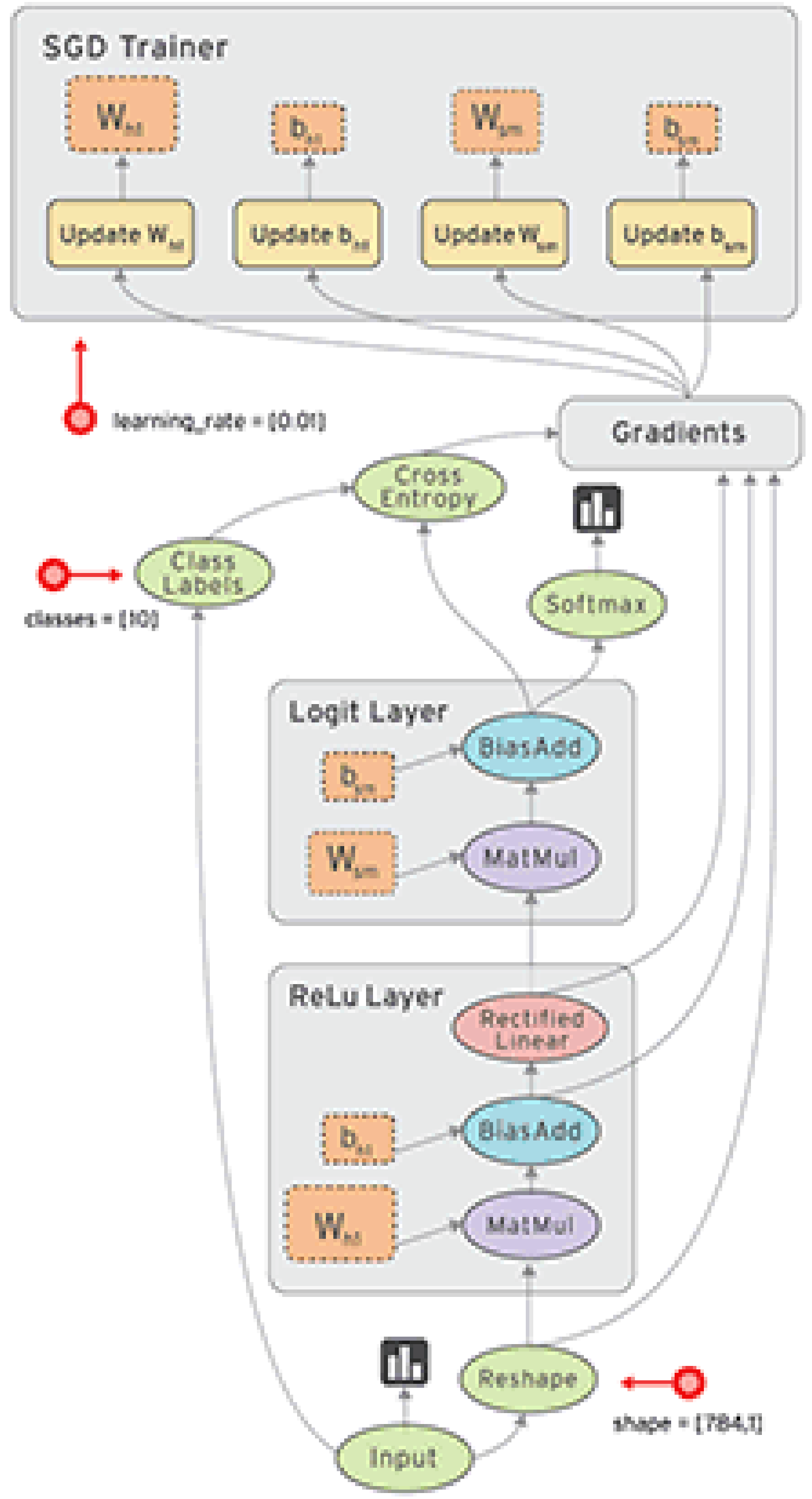
- Auto-differentiation
- **Concurrent ML Systems architecture overview**

Now we roughly have the problem



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

ML System Overview



Dataflow Graph

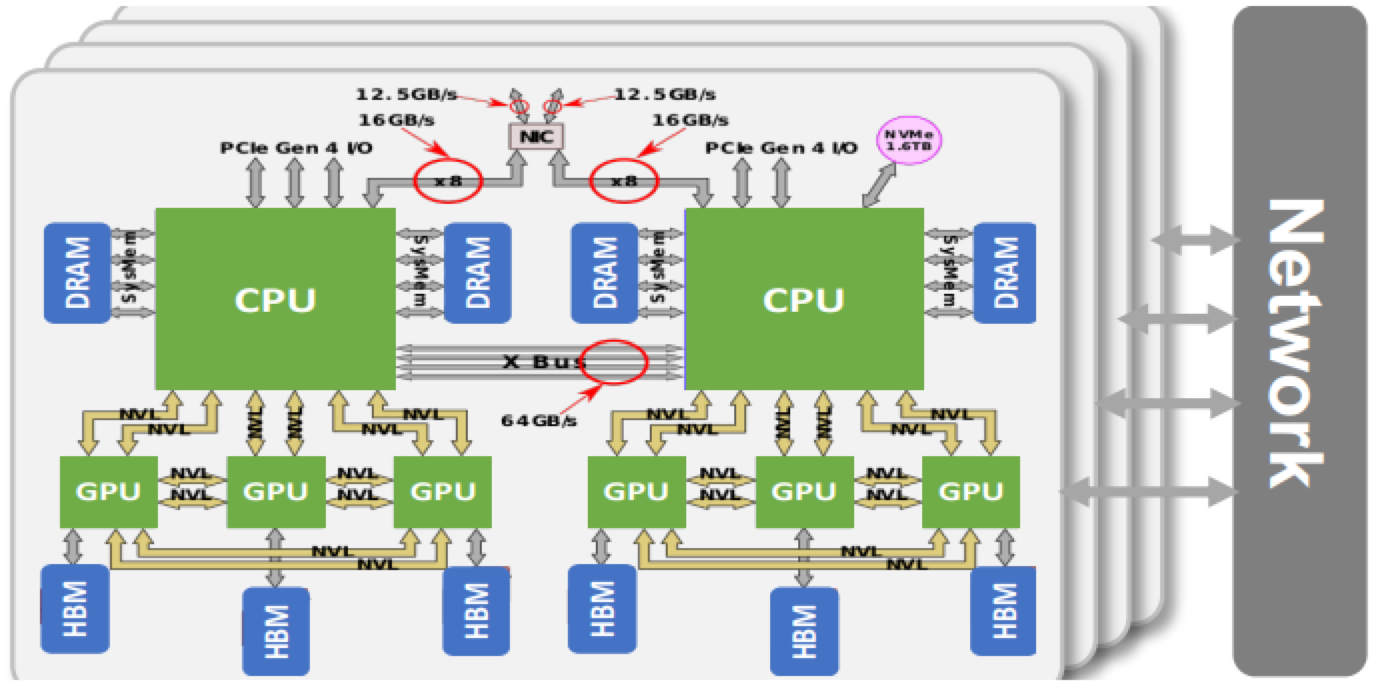
Autodiff

Graph Optimization

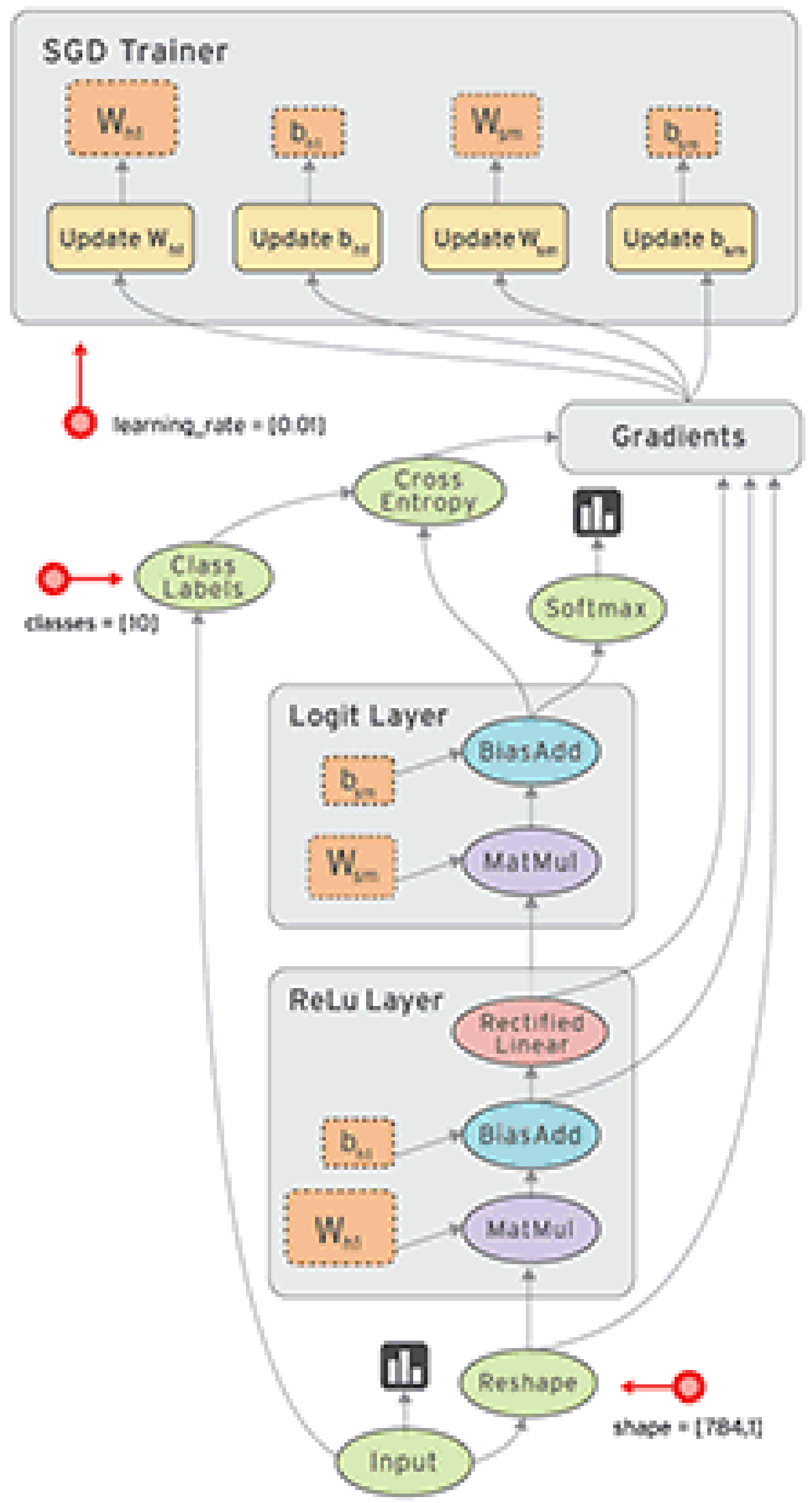
Parallelization

Runtime: schedule / memory

Operator optimization/compilation



ML System Overview



Dataflow Graph

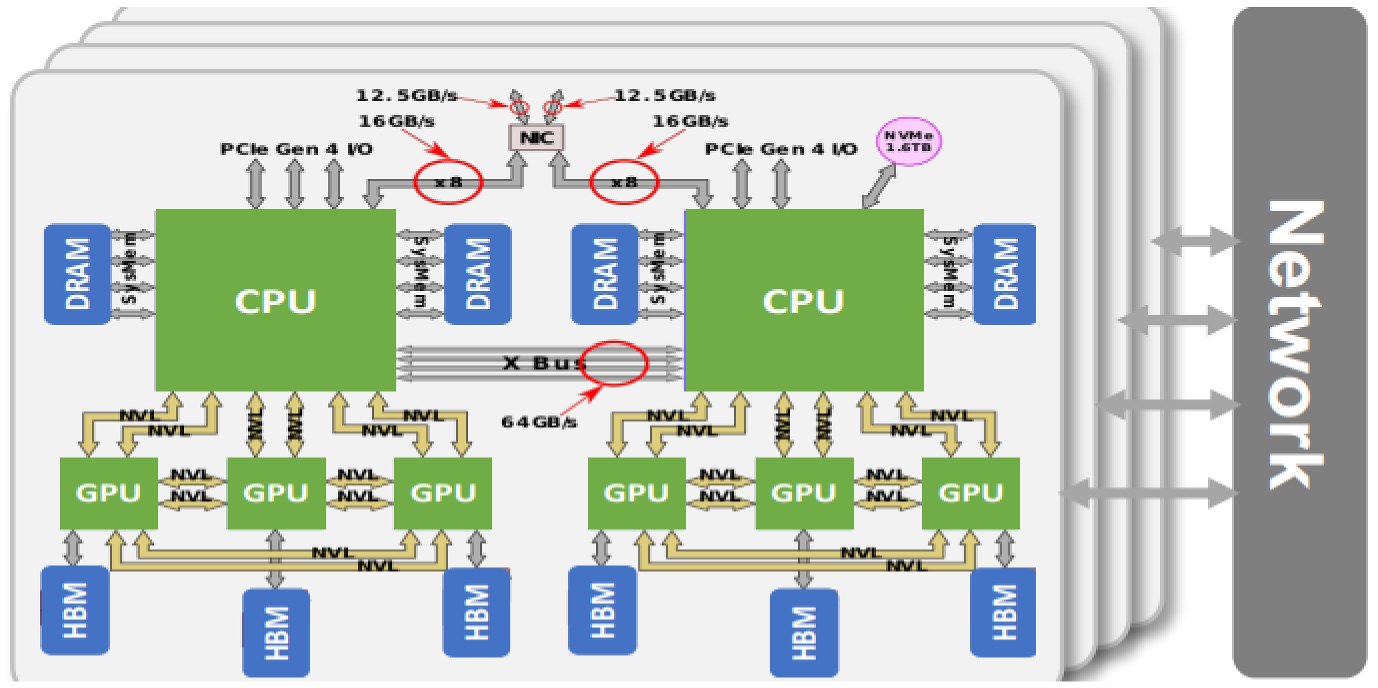
Autodiff

Graph Optimization

Parallelization

Runtime: schedule / memory

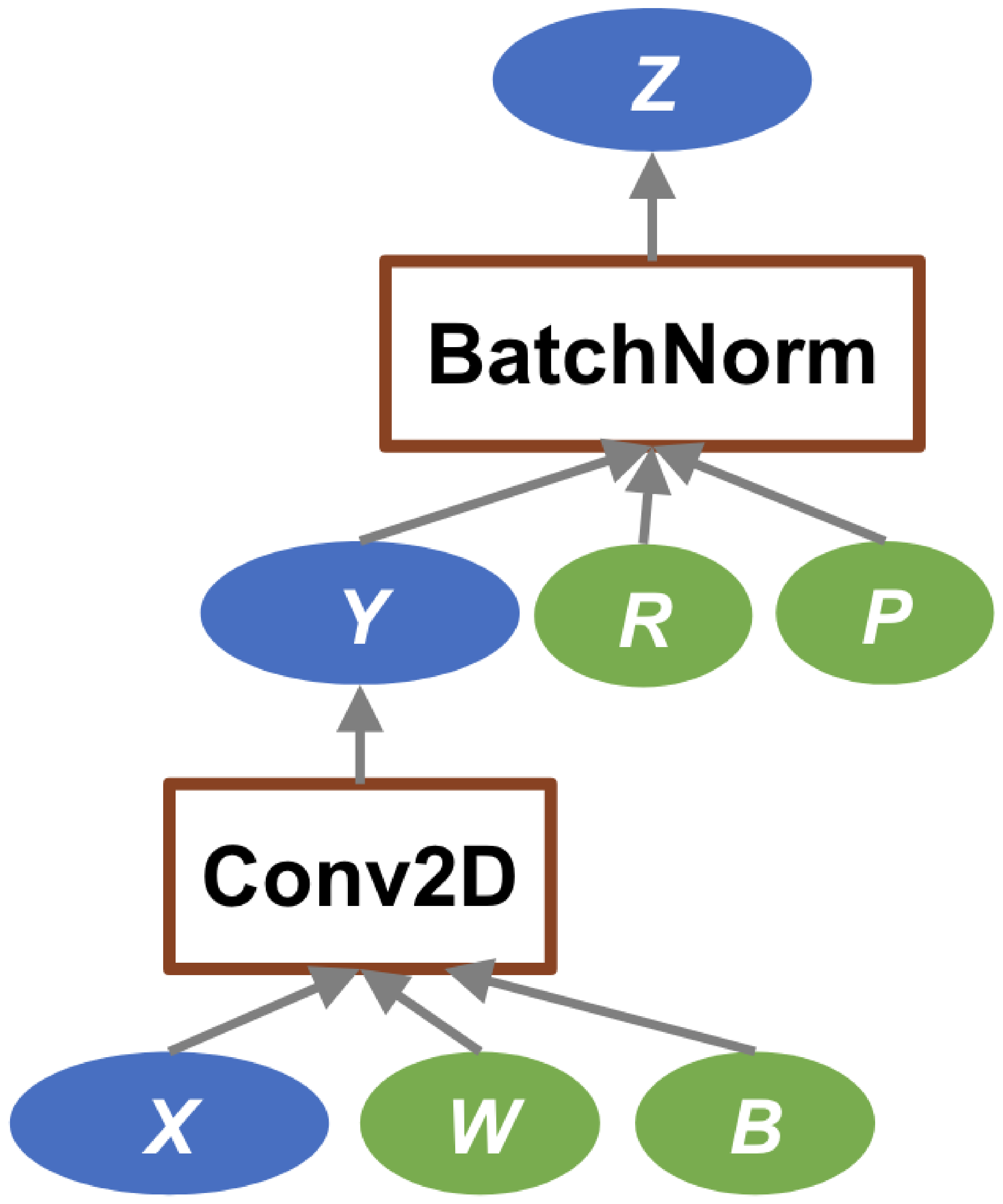
Operator optimization/compilation



Graph Optimization

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

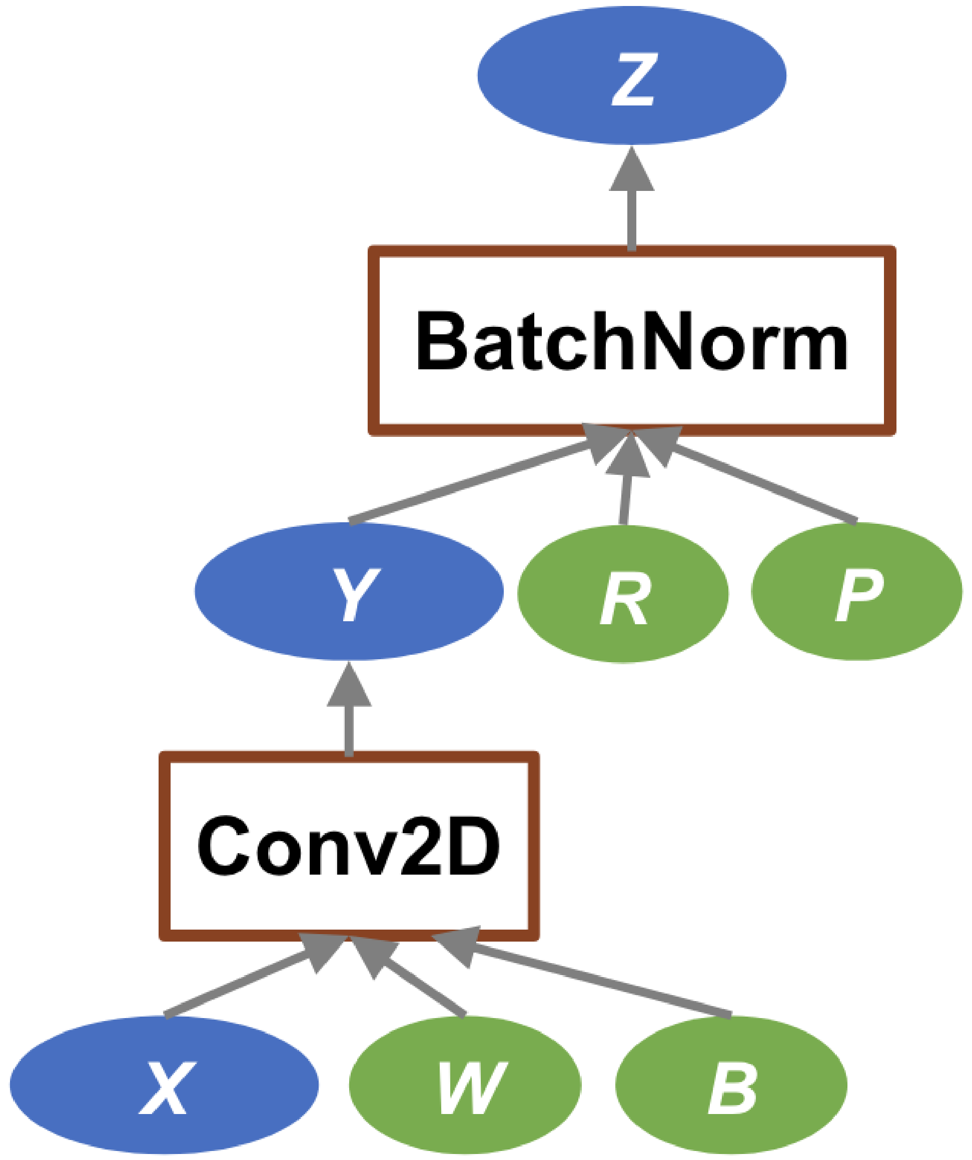
Motivating Example: ResNet



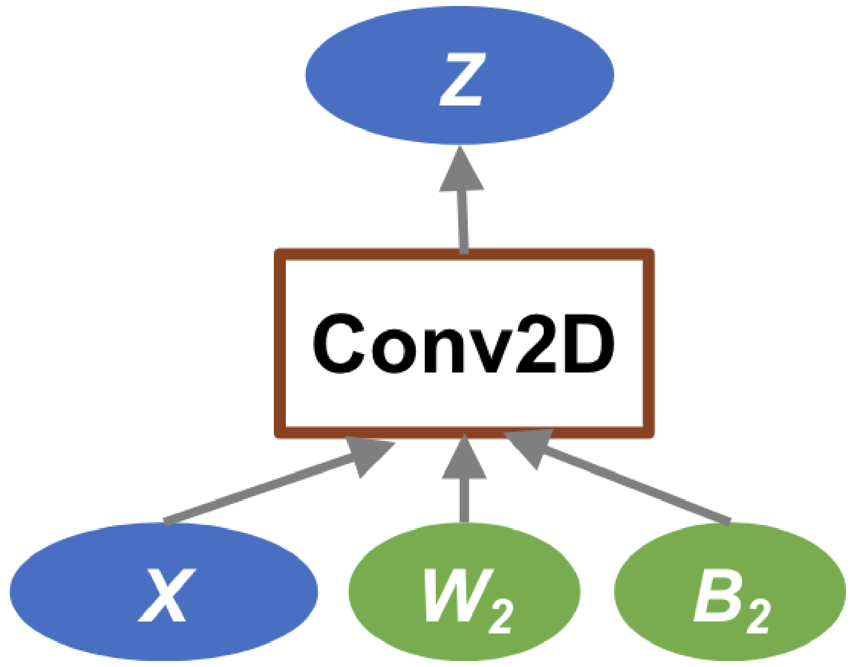
$$Z(n,c,h,w) = Y(n,c,h,w) * R(c) + P(c)$$

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

Motivating Example: ResNet



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

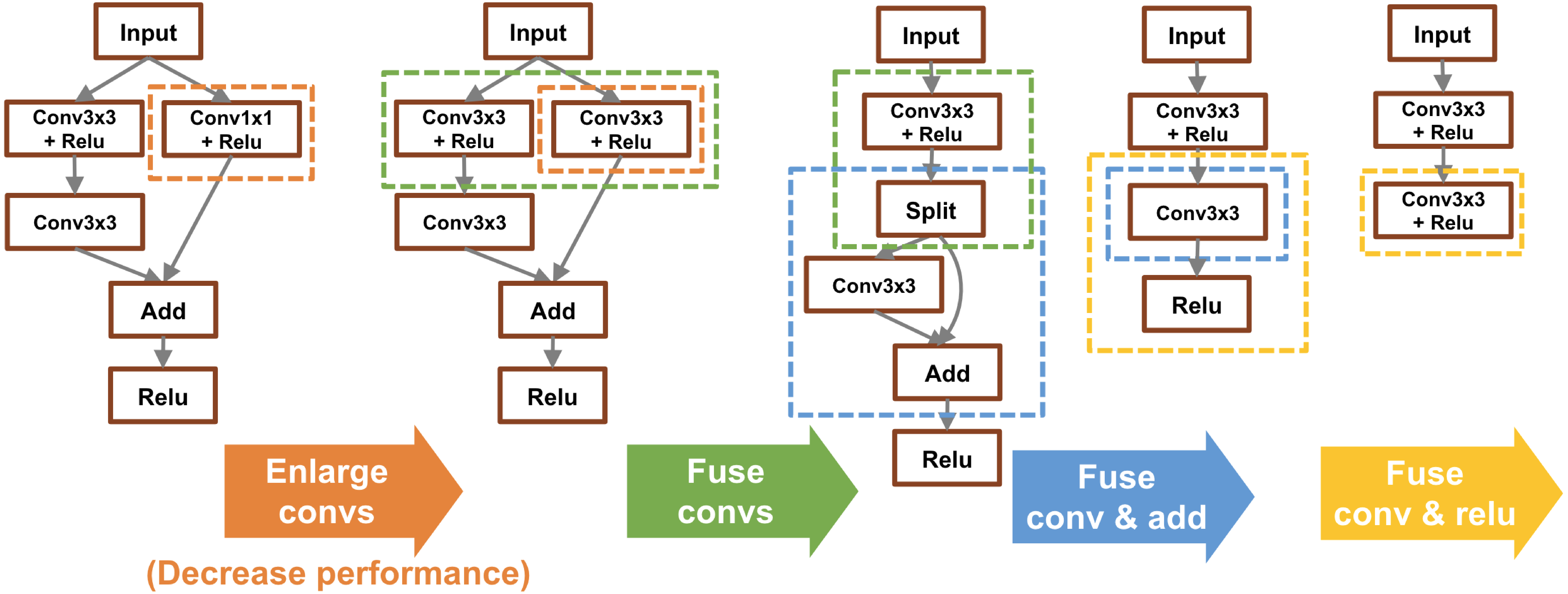


$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

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

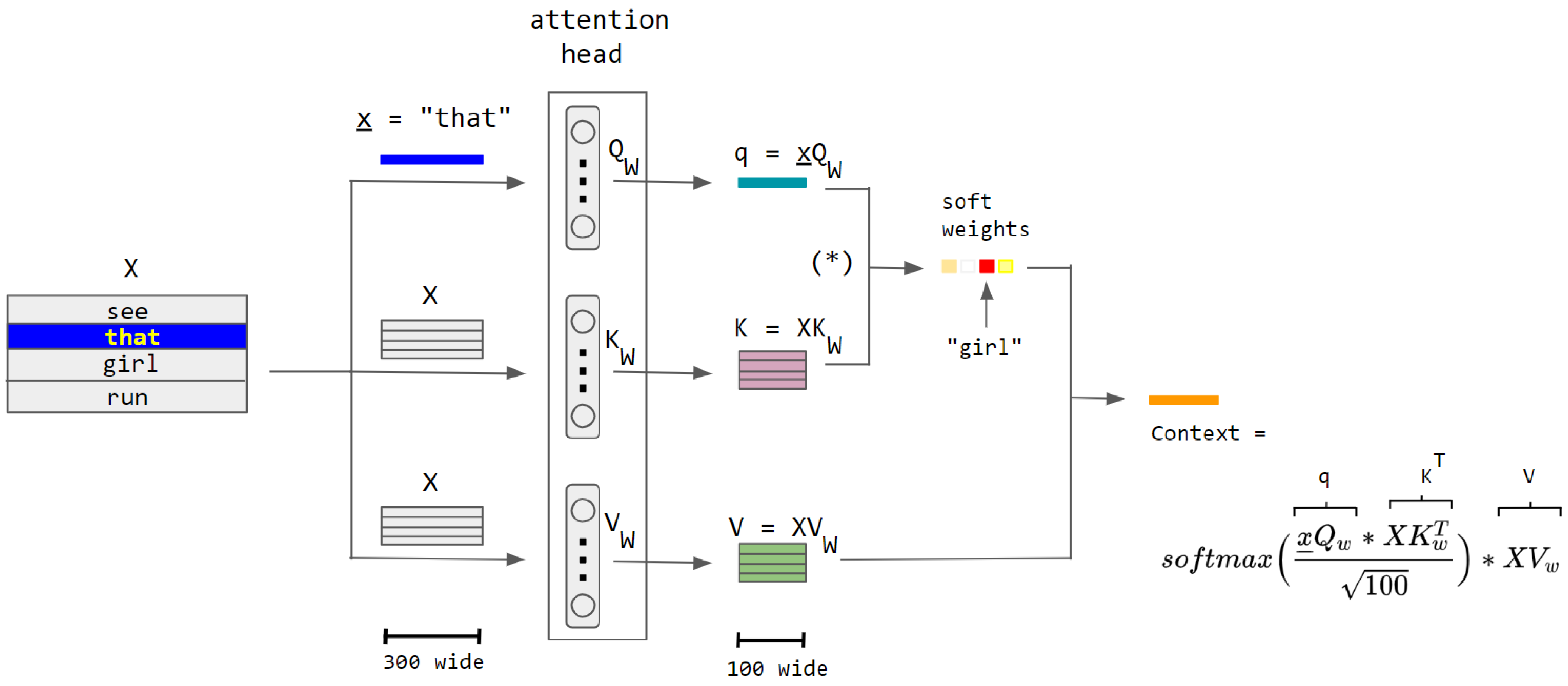
- Why the fusion of conv2d & batchnorm is faster?

Motivating Example: we can go further



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

Motivating Example 3: 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

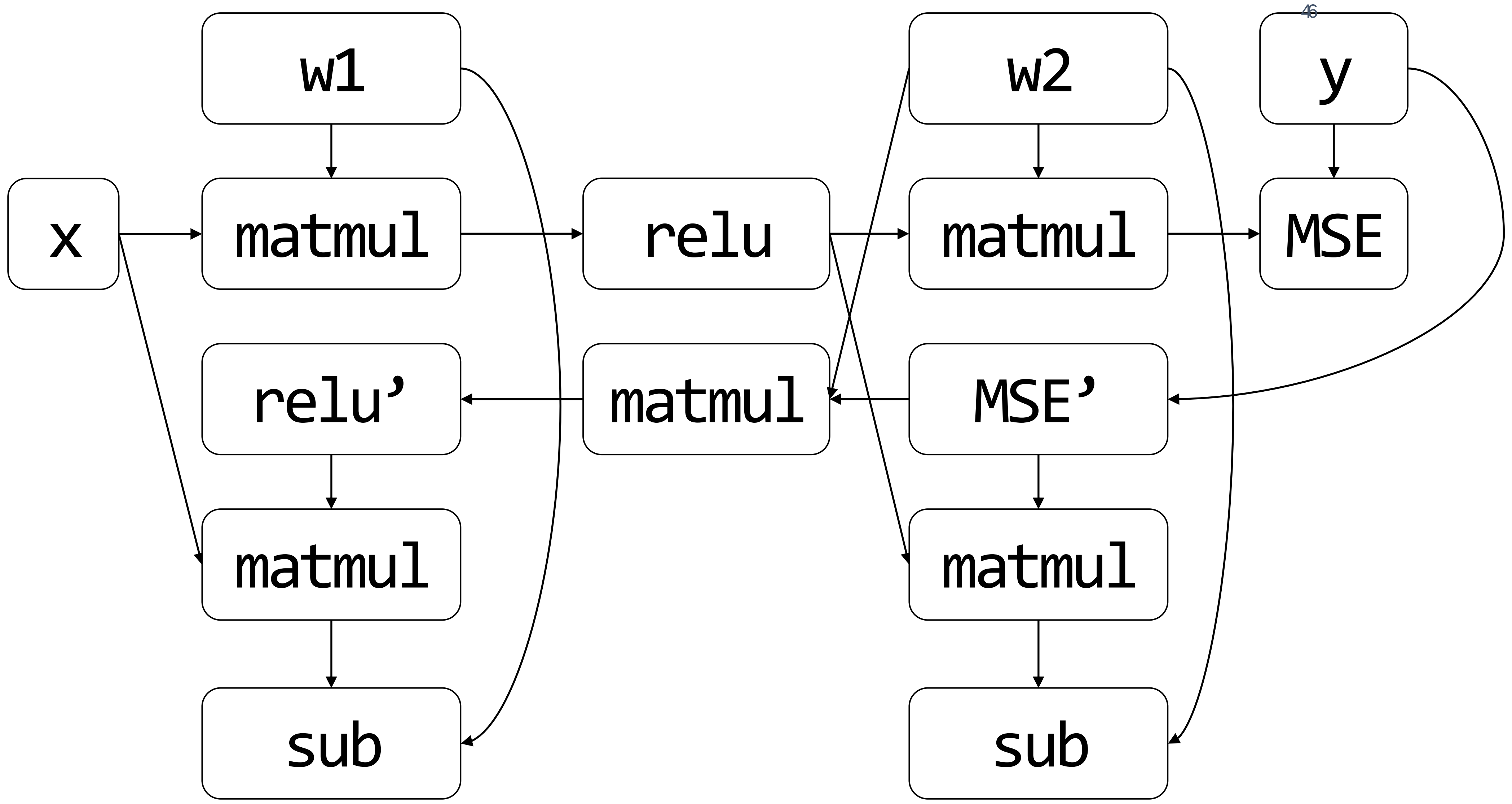
How to perform graph optimization?

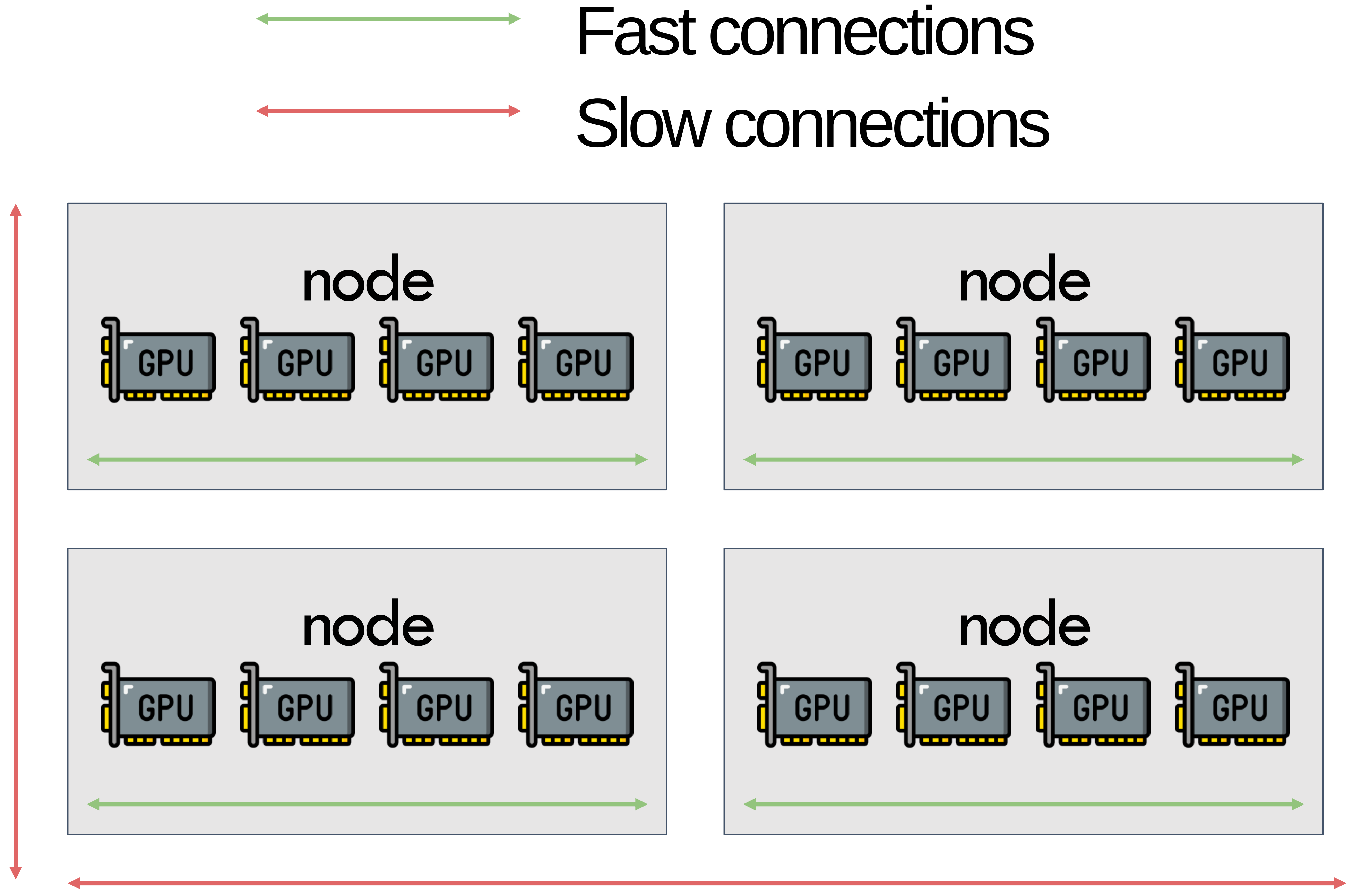
- Writing rules / template
- Auto discovery

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-parallel

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

Motivating Example: Schedule

Operator Implementation

- Goal: get the fastest possible implementation of
  - Matmul
  - Conv2d?
  - Etc
- 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