Data 188: Introduction to Deep Learning

Automatic Differentiation

Speaker: Eric Kim
Lecture 04 (Week 02)
2026-01-29, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Revision01: 2026-01-29

Announcements

- HW0 continues!
  - Due in 1 week
- HW1 will be released next week

Enrollment, Waitlist

- **If you're on the waitlist, and you're a DS senior that is graduating in SP26/SU26 and needs Data 188 to fulfill the MLDM requirement**: please email ds-advising@berkeley.edu to get into the course

- For more info, see Ed post: "Regarding Enrollment, Waitlists"

bCourses Course Participation Assignment

In accordance with federal requirements established by the Department of Education, we need to verify that students are participating in their courses by the end of the fourth week of classes.

An automatically-generated, ungraded assignment will be placed within your bCourses site to confirm the eligibility of your students to receive financial aid. Students will receive separate instructions to complete the 1 question assignment on academic integrity.

You can learn more about the requirement on the Eligibility for Financial Aid at UC Berkeley page.

Some Guidance

- So far, we've been doing a lot of matrix/vector calculations and derivations. For some of you, it may be somewhat unfamiliar and a lot to digest.

- Good news: this part (backprop) is the most math-heavy part of the course. After a few weeks, we'll move onto popular deep learning architectures and applications (ex: computer vision, natural language processing), where the complex Jacobian/partial-derivative stuff will largely disappear.
    - Notably, we'll begin using pytorch, a popular deep learning library, which abstracts a ton of the math away.

- I feel that it's important to learn this math-heavy part of the course, as it's a great way to learn what's going on "under the hood" of popular deep learning frameworks like pytorch and tensorflow.

- My thoughts: you can do it! This material is indeed challenging, and is deliberately meant to challenge you. It'll be 100% worth it!

Outline

General introduction to different differentiation methods

Reverse mode automatic differentiation

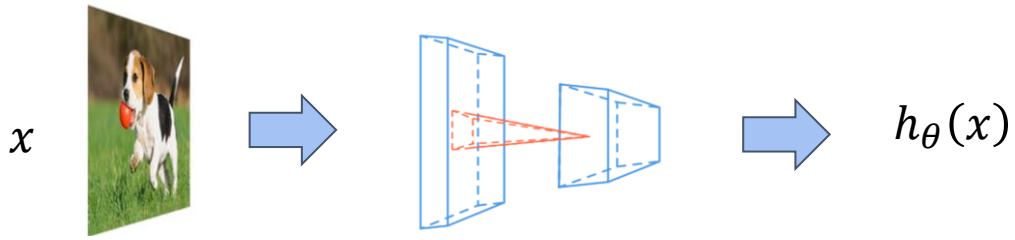
Outline

General introduction to different differentiation methods

Reverse mode automatic differentiation

How does differentiation fit into machine learning

1. The hypothesis class:



2. The loss function:

$$l(h_\theta(x), y) = -h_y(x) + \log \sum_{j=1}^{k} \exp\left(h_j(x)\right)$$

3. An optimization method:

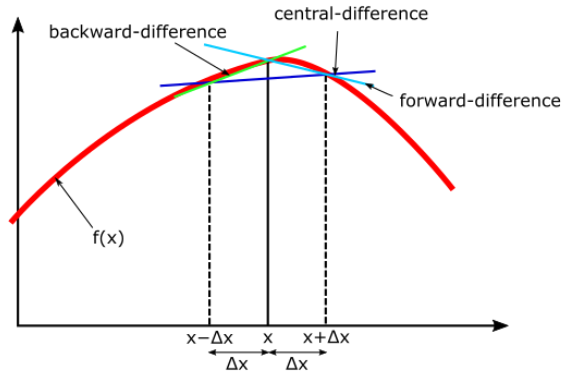
$$\theta := \theta - \frac{\alpha}{B} \sum_{i=1}^{B} \nabla_\theta \ell\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

Recap: every machine learning algorithm consists of three different elements.
Computing the loss function gradient with respect to hypothesis class parameters is the most common operation in machine learning

Numerical differentiation: finite differences

Directly compute the partial gradient by definition

$$\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon e_i) - f(\theta)}{\epsilon}$$



https://en.wikipedia.org/wiki/Finite_difference#Relation_with_derivatives

A more numerically accurate way to approximate the gradient

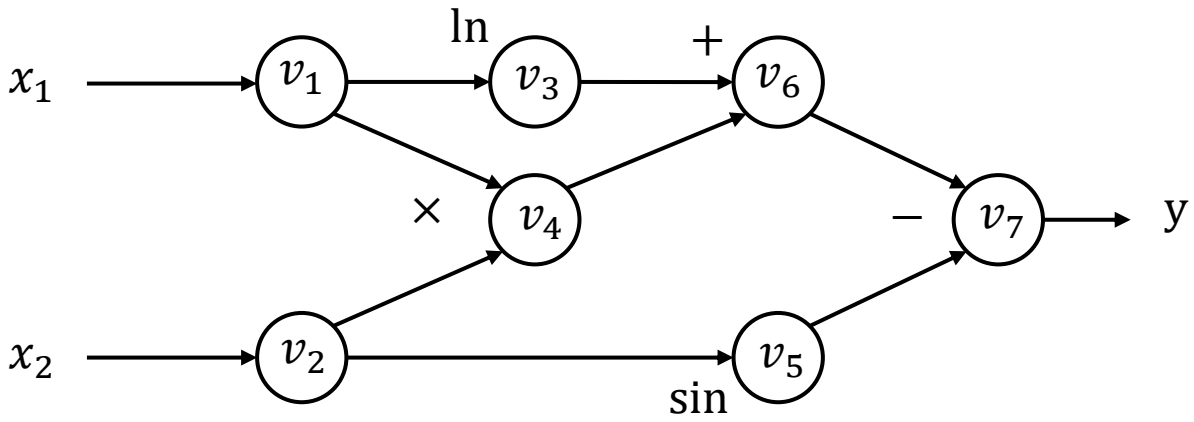
$$\frac{\partial f(\theta)}{\partial \theta_i} = \frac{f(\theta + \epsilon e_i) - f(\theta - \epsilon e_i)}{2\epsilon} + o(\epsilon^2)$$

Why not use this in deep learning? Numerical errors, less efficient to compute

(Still, useful for testing implementation correctness!)

Computational graph

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



Each node represent an (intermediate) value in the computation. Edges present input output relations.

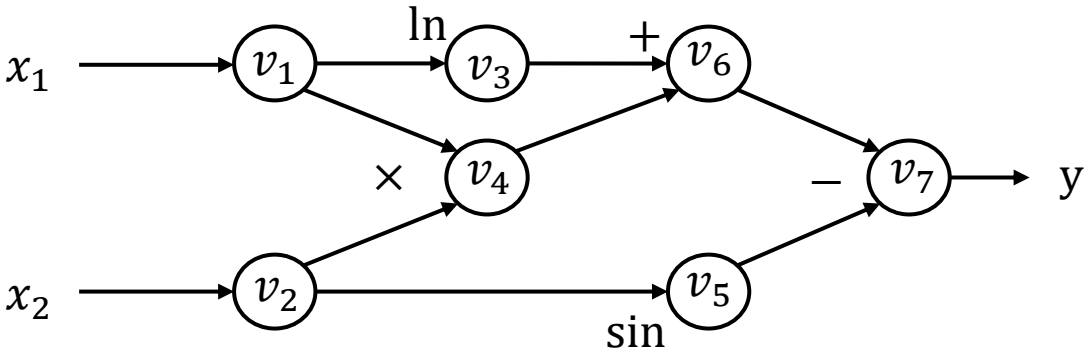
Forward evaluation trace

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

Example based on A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. JMLR

Forward mode automatic differentiation (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 $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$

We can then compute the $\dot{v}_i$ iteratively in the forward topological order of the computational graph

$$\dot{v}_3 = \frac{\partial v_3}{\partial x_1} = \frac{\partial v_3}{\partial v_1}\frac{\partial v_1}{\partial x_1} = \frac{\partial \ln(v_1)}{\partial v_1}\dot{v}_1 = \frac{\dot{v}_1}{v_1} = \frac{1}{2}$$

Chain rule

Forward AD trace

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

$$\dot{v}_4 = \frac{\partial v_4}{\partial x_1} = \frac{\partial (v_1 v_2)}{\partial x_1} = \frac{\partial (v_1)}{\partial x_1} v_2 + v_1 \frac{\partial (v_2)}{\partial x_1} = v_2 + 0 = 5$$

Product rule

Now we have $\frac{\partial y}{\partial x_1} = \dot{v}_7 = 5.5$

Limitation of forward mode AD

For $f: \mathbb{R}^n \to \mathbb{R}^k$, we need n forward AD passes to get the gradient with respect to each input.

We mostly care about the cases where $k = 1$ and large n .

To resolve the problem efficiently, we need to use another kind of AD.

Outline

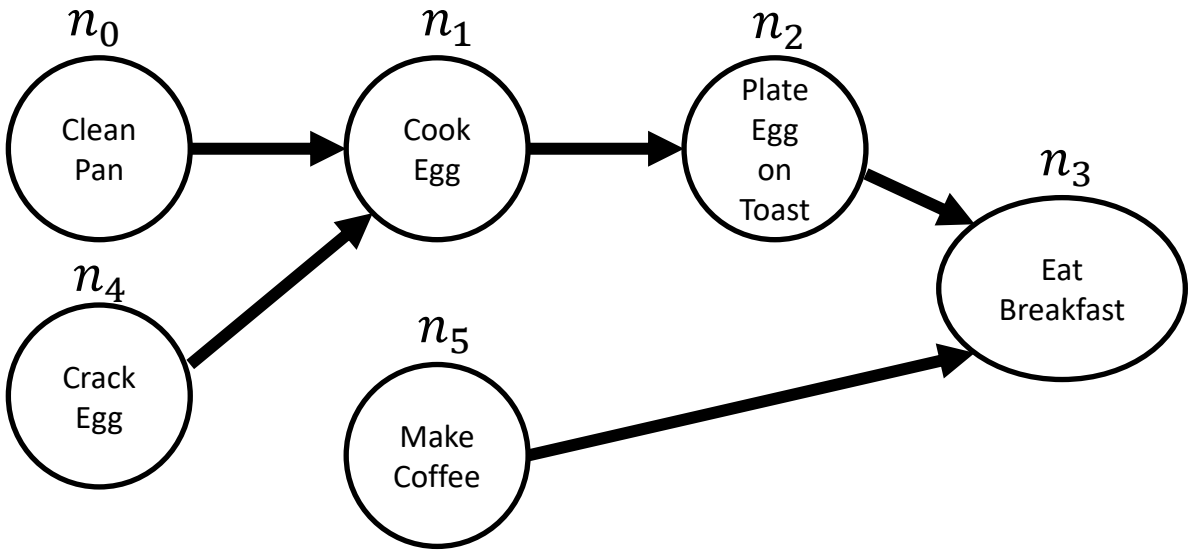
General introduction to different differentiation methods

Reverse mode automatic differentiation

Topological Sort

Definition: Given a directed acyclic graph (DAG), a topological sort of the graph is an ordering of the nodes such that every directed edge *(u,v)* from node *u* to node *v*, *u* comes before *v* in the ordering.

Example: given a dependency graph of tasks, the topological ordering tells you the correct order to complete the tasks, taking dependencies into account.



Topological orderings:

$$[n_0, n_4, n_1, n_2, n_5, n_3]$$

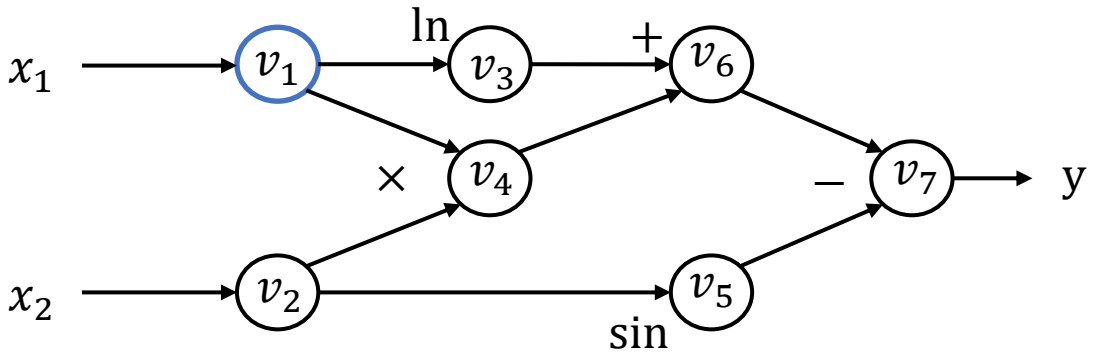
$$[n_4, n_0, n_1, n_5, n_2, n_3]$$

Note: there may be more than one valid topological ordering!

Topological Sort: Depth-first search (intuition)

There are several algorithms to compute topological ordering. Here is one! (Useful for HW1!)

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



Let's look at a **post-order depth-first traversal** starting from v1:

dfs_postorder(v1, visited)

dfs_postorder(v3, visited) = [v3, v6, v7]

dfs_postorder(v4, visited) = [v4]

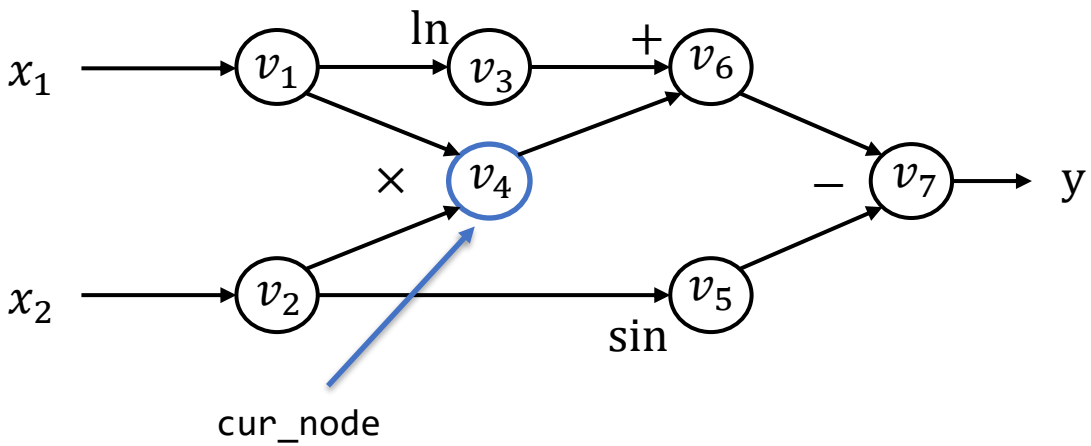
Output: [v1] + ([v4] + [v3, v6, v7]) = [v1, v4, v3, v6, v7]

Observation: This looks like part of a topological sort!

```
def dfs_postorder(node, visited: set, out: list):
    if node in visited:
        return
    for child in node.outgoing:
        dfs_postorder(child, visited, out)
    visited.add(node)
    # important: prepend!
    out.insert(0, node)
```

Topological Sort: Depth-first search (1/4)

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



visited = { }

L = [ ]

```
visited = set()
L = list()
Repeat until all nodes are visited:
    Let cur_node be some unvisited node.
    Call dfs_postorder(cur_node, visited, L)
Return L as topological ordering.
```

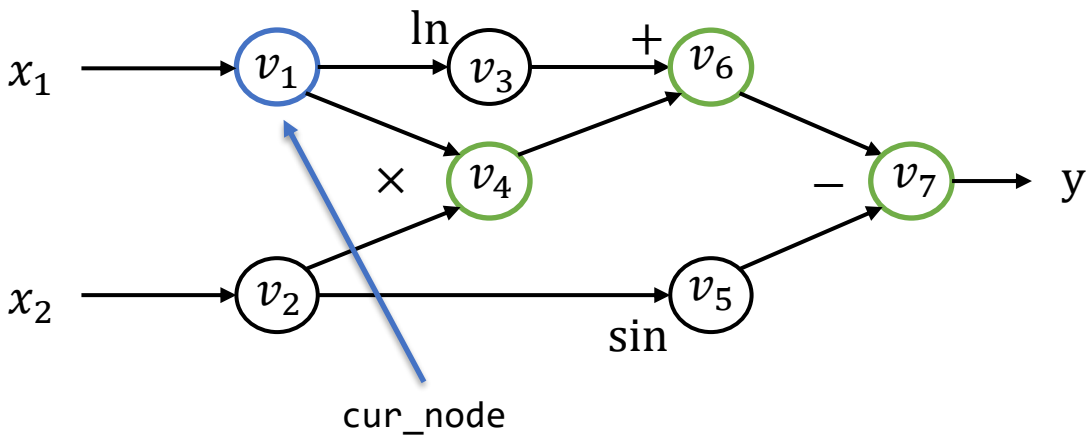
Tip: doesn't matter which unvisited node you pick. Neat algorithm property!

```
def dfs_postorder(node, visited: set, out: list):
    if node in visited:
        return
    for child in node.outgoing:
        dfs_postorder(child, visited, out)
    visited.add(node)
    # important: prepend!
    out.insert(0, node)
```

For more info about the algorithm, see Wikipedia "Topological sorting: Depth-first search"

Topological Sort: Depth-first search (2/4)

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



visited = { v_7, v_6, v_4 }

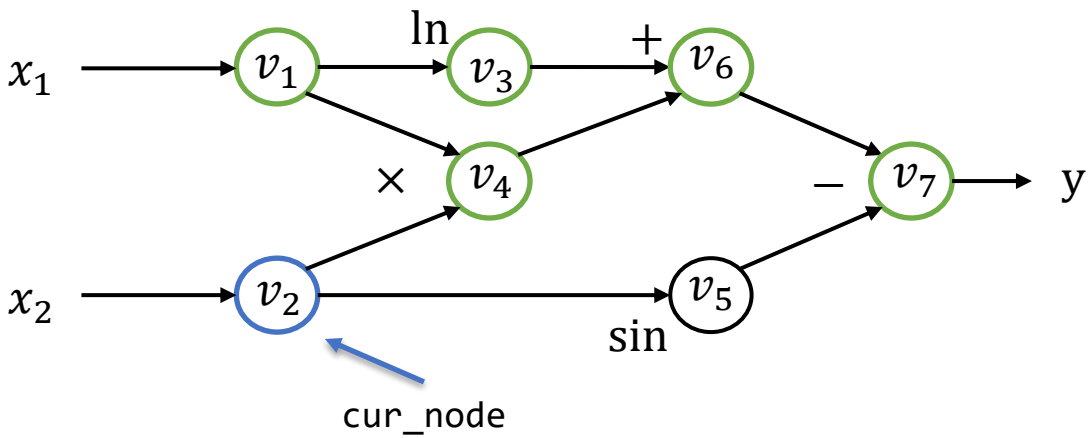
L = { v_4, v_6, v_7 }

```
visited = set()
L = list()
Repeat until all nodes are visited:
    Let cur_node be some unvisited node.
    Call dfs_postorder(cur_node, visited, L)
Return L as topological ordering.
```

```
def dfs_postorder(node, visited: set, out: list):
    if node in visited:
        return
    for child in node.outgoing:
        dfs_postorder(child, visited, out)
    visited.add(node)
    # important: prepend!
    out.insert(0, node)
```

Topological Sort: Depth-first search (3/4)

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



visited = { v_7, v_6, v_4, v_3, v_1 }

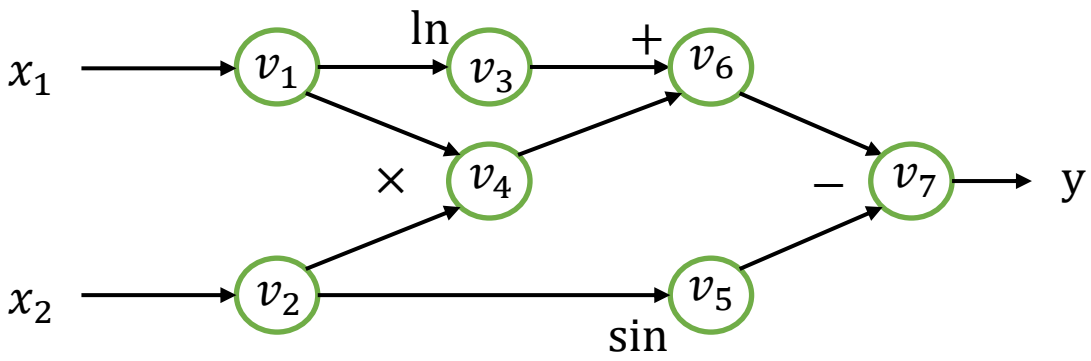
L = { v_1, v_3, v_4, v_6, v_7 }

```
visited = set()
L = list()
Repeat until all nodes are visited:
    Let cur_node be some unvisited node.
    Call dfs_postorder(cur_node, visited, L)
Return L as topological ordering.
```

```
def dfs_postorder(node, visited: set, out: list):
    if node in visited:
        return
    for child in node.outgoing:
        dfs_postorder(child, visited, out)
    visited.add(node)
    # important: prepend!
    out.insert(0, node)
```

Topological Sort: Depth-first search (4/4)

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



Done, no more nodes to process!

visited = { $v_7, v_6, v_4, v_3, v_1, v_5, v_2$ }

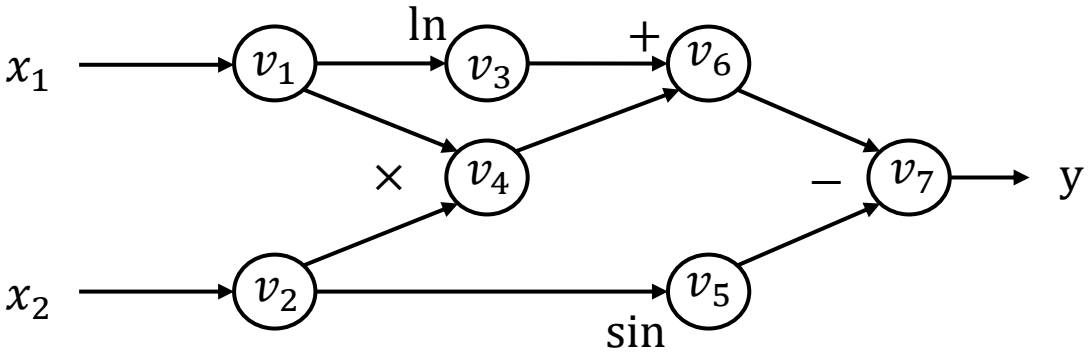
L = { $v_2, v_5, v_1, v_3, v_4, v_6, v_7$ }

```
visited = set()
L = list()
Repeat until all nodes are visited:
    Let cur_node be some unvisited node.
    Call dfs_postorder(cur_node, visited, L)
Return L as topological ordering.
```

```
def dfs_postorder(node, visited: set, out: list):
    if node in visited:
        return
    for child in node.outgoing:
        dfs_postorder(child, visited, out)
    visited.add(node)
    # important: prepend!
    out.insert(0, node)
```

Reverse mode automatic differentiation(AD)

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



Forward evaluation trace

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

By chain rule:
eg observe that:

$$\frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \frac{\partial v_y}{\partial v_6} = \overline{v_6}$$

Define adjoint $\overline{v_i} = \frac{\partial y}{\partial v_i}$

We can then compute the $\overline{v_i}$ iteratively in the **reverse** topological order of the computational graph

Reverse AD evaluation trace

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

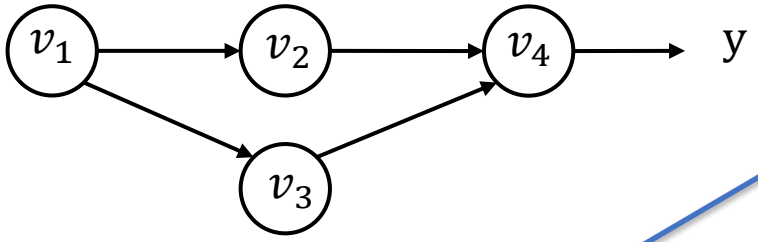
$$\overline{v_6} = \frac{\partial y}{\partial v_6} = \frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7}\frac{\partial (v_6 - v_5)}{\partial v_6} = \overline{v_7}(1) = 1$$
$$\overline{v_5} = \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_5} = \overline{v_7}\frac{\partial (v_6 - v_5)}{\partial v_5} = \overline{v_7}(-1) = -1$$
$$\overline{v_4} = \frac{\partial y}{\partial v_4} = \underbrace{\frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_6}}_{\overline{v_6}}\frac{\partial v_6}{\partial v_4} = \overline{v_6}\frac{\partial (v_3 + v_4)}{\partial v_4} = \overline{v_6}(1) = 1$$

Hint: this should remind you of our two-layer NN gradients abstraction...

Important: note the addition of multiple pathways (next slide!)

Multiple pathway case

v_1 is being used in multiple pathways (v_2 and v_3)



This is also known as the "total derivative":

$$\frac{\partial f(x(t),y(t))}{\partial t} = \frac{\partial f(x(t),y(t))}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f(x(t),y(t))}{\partial y}\frac{\partial y}{\partial t}$$

y can be written in the form of $y = f(v_2, v_3)$

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

Intuition: to calculate the impact that v_1 has on y ($\frac{\partial y}{\partial v_1}$), we can decompose it the sum of two contributions:

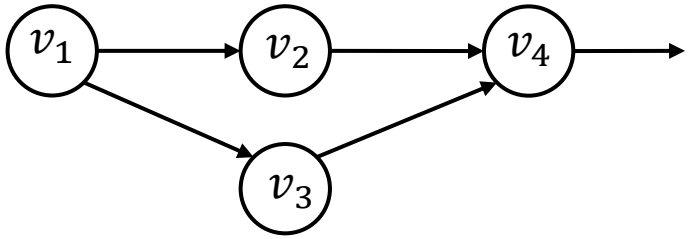
The impact that v_2 has on y ($\frac{\partial y}{\partial v_2}$), scaled by the impact that v_1 has on v_2 ($\frac{\partial v_2}{\partial v_1}$), aka: $\frac{\partial y}{\partial v_2}\cdot\frac{\partial v_2}{\partial v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1}$

The impact that v_3 has on y ($\frac{\partial y}{\partial v_3}$), scaled by the impact that v_1 has on v_3 ($\frac{\partial v_3}{\partial v_1}$), aka: $\frac{\partial y}{\partial v_3}\cdot\frac{\partial v_3}{\partial v_1} = \overline{v_3}\frac{\partial v_3}{\partial v_1}$

$$\Rightarrow \quad \overline{v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}$$

Partial Adjoints

v_1 is being used in multiple pathways (v_2 and v_3)



$$\overline{v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}$$

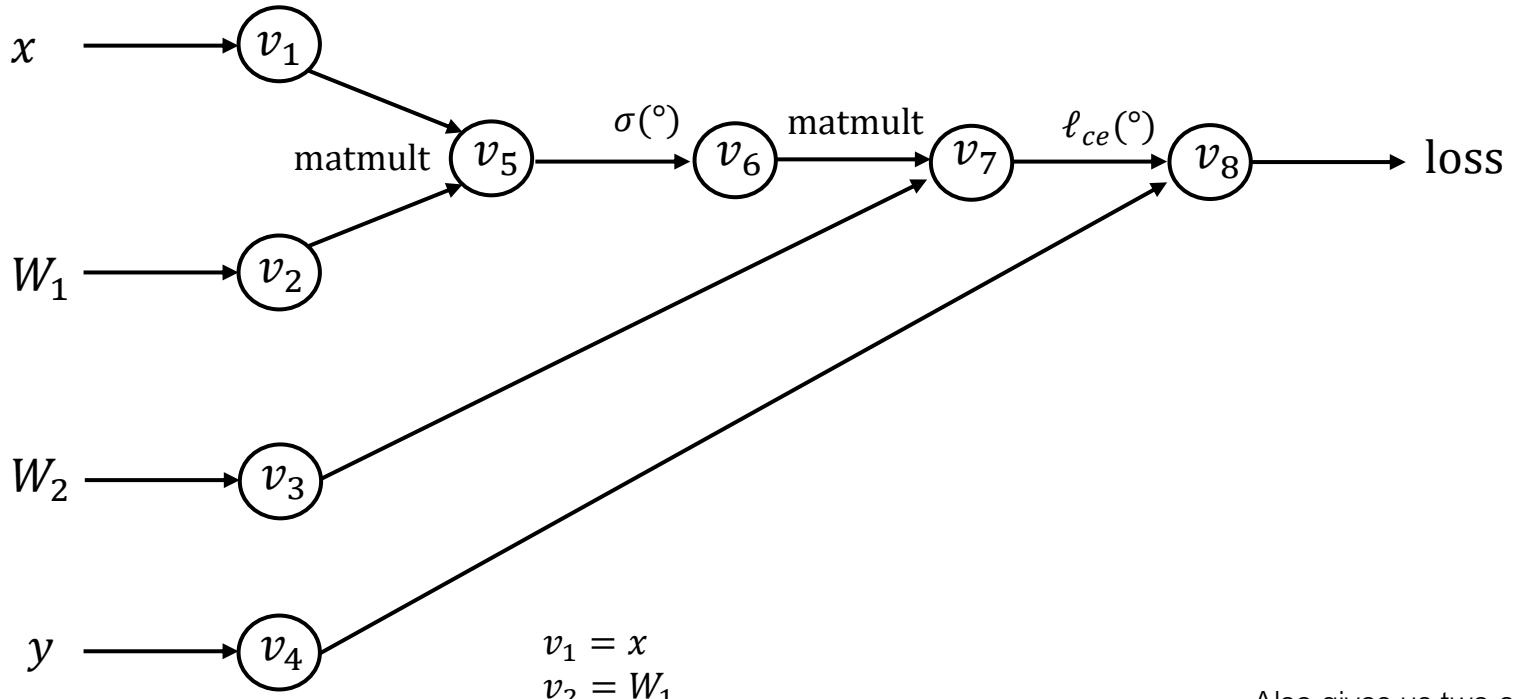
More generally, define "**partial adjoint**": $\overline{v_{i\to j}} = \overline{v_j}\,\frac{\partial v_j}{\partial v_i}$ for each input output node pair i and j

Then, the "full adjoint" is the sum of all outgoing partial adjoints: $\overline{v_i} = \sum_{j \in next(i)} \overline{v_{i\to j}}$

Reverse mode AD for deep learning (1/2)

$\text{loss} = \ell_{ce}(\sigma(xW_1)W_2, y)$

Define adjoint $\overline{v_i} = \frac{\partial loss}{\partial v_i}$



$v_1 = x$
$v_2 = W_1$
$v_3 = W_2$
$v_4 = y$
$v_5 = \text{matmul}(v_1, v_2)$
$v_6 = \sigma(v_5)$
$v_7 = \text{matmul}(v_6, v_3)$
$v_8 = \ell_{ce}(v_7, v_4)$

Running reverse mode AD gives us these very useful adjoints:

$$\overline{v_2} = \frac{\partial loss}{\partial v_2} = \frac{\partial loss}{\partial W_1}$$
$$\overline{v_3} = \frac{\partial loss}{\partial v_3} = \frac{\partial loss}{\partial W_2}$$

Also gives us two curious adjoints:

$$\overline{v_1} = \frac{\partial loss}{\partial v_1} = \frac{\partial loss}{\partial x}$$

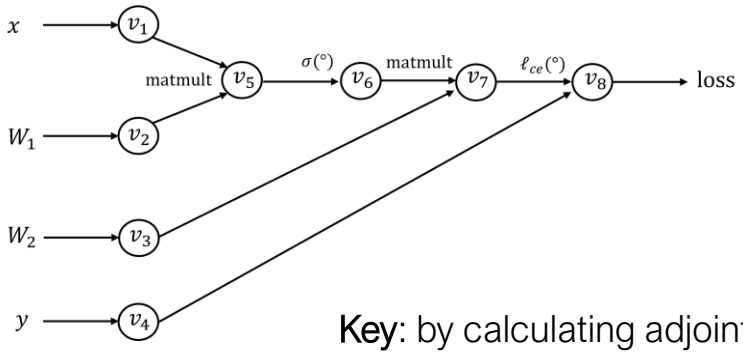
"How to adjust the input to minimize loss?"

$$\overline{v_4} = \frac{\partial loss}{\partial v_4} = \frac{\partial loss}{\partial y}$$

"How to adjust the ground-truth label to minimize loss?"

Reverse mode AD for deep learning (2/2)

$$\text{loss} = \ell_{ce}(\sigma(xW_1)W_2, y)$$



Key: by calculating adjoints in reverse-topological order, calculating $\overline{v_i}$ involves calculating a "local" gradient $\frac{\partial v_j}{\partial v_i}$, and multiplying by an already-computed $\overline{v_j}$ (eg by lookup)

$$v_1 = x$$
$$v_2 = W_1$$
$$v_3 = W_2$$
$$v_4 = y$$
$$v_5 = \text{matmul}(v_1, v_2)$$
$$v_6 = \sigma(v_5)$$
$$v_7 = \text{matmul}(v_6, v_3)$$
$$v_8 = \ell_{ce}(v_7, v_4)$$

Define adjoint $\overline{v_i} = \frac{\partial loss}{\partial v_i}$

$$\overline{v_8} = \frac{\partial loss}{\partial v_8} = 1$$

Recall: this is $\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2}$ from previous lecture!

$$\overline{v_7} = \frac{\partial loss}{\partial v_7} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7} = \overline{v_8}\frac{\partial v_8}{\partial v_7} = (1) * (\text{softmax}(z_2) - e_y)$$

$$\overline{v_3} = \frac{\partial loss}{\partial v_3} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_3} = \overline{v_7}\frac{\partial v_7}{\partial v_3} = \sigma(xW_1)^T(s - e_y) = \frac{\partial loss}{\partial W_2}$$

Recall: this is $\frac{\partial z_1 W_2}{\partial z_1}$ from previous lecture!

$$\overline{v_6} = \frac{\partial loss}{\partial v_6} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = (\text{softmax}(z_2) - e_y)(v_3^T)$$

$$\overline{v_5} = \frac{\partial loss}{\partial v_5} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6}\frac{\partial v_6}{\partial v_5} = \overline{v_6}\frac{\partial v_6}{\partial v_5} = (\text{softmax}(z_2) - e_y)(v_3^T) \circ \acute{\sigma}(z_0)$$

$$\overline{v_2} = \frac{\partial loss}{\partial v_2} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6}\frac{\partial v_6}{\partial v_5}\frac{\partial v_5}{\partial v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_2} = x^T(\text{softmax}(z_2) - e_y)W_2^T \circ \acute{\sigma}(xW_1) = \frac{\partial loss}{\partial W_1}$$

(optional, rewrite to clean things up)
$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

Operator abstraction

Let's define the Op ("Operator") interface:

```
class Op:
    def compute(self, *args: Tuple[NDArray]):
        """Calculate forward pass of operator.
        """
        raise NotImplementedError()

    def gradient(
        self, out_grad: "Value", node: "Value"
    ) -> Union["Value", Tuple["Value"]]:
        """Compute partial adjoint for each input value for a given output adjoint.
        Parameters
        ----------
        out_grad: Value
            The adjoint wrt to the output value.
        node: Value
            The value node of forward evaluation.
        Returns
        -------
        input_grads: Value or Tuple[Value]
            A list containing partial gradient adjoints to be propagated to
            each of the input node.
        """
        raise NotImplementedError()
```

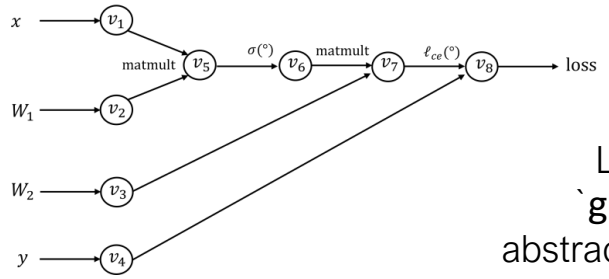
Tip: `out_grad` is $\frac{\partial loss}{\partial output}$, and `node` is typically used to fetch info like: input args for the Op.

`Op.gradient()`'s job is to calculate $\frac{\partial loss}{\partial input_i}$ for every Op input.

This is done by calculating: $\frac{\partial loss}{\partial output}\frac{\partial output}{\partial input_i}$ for each input.

Reverse mode AD vs `gradient()` (1/2)

$$\text{loss} = \ell_{ce}(\sigma(xW_1)W_2, y)$$



Let's connect our `gradient()` layer abstraction to these adjoints!

Define adjoint $\overline{v_i} = \frac{\partial loss}{\partial v_i}$

(optional, rewrite to clean things up)
$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\overline{v_8} = \frac{\partial loss}{\partial v_8} = 1$$

Recall: this is $\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2}$ from previous lecture!

$$\overline{v_7} = \frac{\partial loss}{\partial v_7} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7} = \overline{v_8}\frac{\partial v_8}{\partial v_7} = (1) * (\text{softmax}(z_2) - e_y)$$

$$\overline{v_3} = \frac{\partial loss}{\partial v_3} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_3} = \overline{v_7}\frac{\partial v_7}{\partial v_3} = \sigma(xW_1)^T(s - e_y) = \frac{\partial loss}{\partial W_2}$$

```
# forward
...
z0 = linear1.forward(x, W1)
z1 = elem_sigmoid.forward(z0)
z2 = linear2.forward(z1, W2)  # logits
loss_val = loss_ce.forward(z2, y)

# backward
dloss_dz2 = loss_ce.backward()

dloss_dz1, dloss_dW2 = linear2.gradient(dloss_dz2, ...)
dloss_dz0 = elem_sigmoid.backward(dloss_dz1, ...)
dloss_dx, dloss_dW1 = linear1.backward(dloss_dz0, ...)

# update params
W1 = W1 - stepsize * dloss_dW1
W2 = W2 - stepsize * dloss_dW2
```

`dloss_dW2` is $\overline{v_3}$

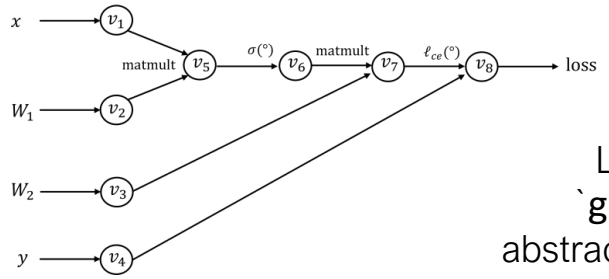
```
class MatMul(Op):
    def compute(self, lhs, rhs):
        return lhs @ rhs
    def gradient(self, out_grad, node):
        lhs, rhs = node.inputs
        # calculate dloss_dout @ dout/dlhs
        dloss_dlhs = out_grad @ rhs.T
        # calculate dloss_dout @ dout/drhs
        dloss_drhs = lhs.T @ out_grad
        return dloss_dlhs, dloss_drhs
```

Performant abstraction: for each layer, define `gradient()` function that returns (for each input):

$$\frac{\partial loss}{\partial input} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial input}$$

Reverse mode AD vs `gradient()` (2/2)

$$\text{loss} = \ell_{ce}(\sigma(xW_1)W_2, y)$$



Let's connect our `gradient()` layer abstraction to these adjoints!

```
# forward
...
z0 = linear1.forward(x, W1)
z1 = elem_sigmoid.forward(z0)
z2 = linear2.forward(z1, W2)  # logits
loss_val = loss_ce.forward(z2, y)

# backward
dloss_dz2 = loss_ce.backward()

dloss_dz1, dloss_dW2 = linear2.gradient(dloss_dz2, ...)
dloss_dz0 = elem_sigmoid.backward(dloss_dz1, ...)
dloss_dx, dloss_dW1 = linear1.backward(dloss_dz0, ...)

# update params
W1 = W1 - stepsize * dloss_dW1
W2 = W2 - stepsize * dloss_dW2
```

Define adjoint $\overline{v_i} = \frac{\partial loss}{\partial v_i}$

(optional, rewrite to clean things up)
$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\overline{v_8} = \frac{\partial loss}{\partial v_8} = 1$$

Recall: this is $\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2}$ from previous lecture!

$$\overline{v_7} = \frac{\partial loss}{\partial v_7} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7} = \overline{v_8}\frac{\partial v_8}{\partial v_7} = (1) * (\text{softmax}(z_2) - e_y)$$

$$\overline{v_3} = \frac{\partial loss}{\partial v_3} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_3} = \overline{v_7}\frac{\partial v_7}{\partial v_3} = \sigma(xW_1)^T(s - e_y) = \frac{\partial loss}{\partial W_2}$$

$$\overline{v_6} = \frac{\partial loss}{\partial v_6} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = (\text{softmax}(z_2) - e_y)(v_3^T)$$

`dloss_dz1` is $\overline{v_6}$

```
class MatMul(Op):
    def compute(self, lhs, rhs):
        return lhs @ rhs
    def gradient(self, out_grad, node):
        lhs, rhs = node.inputs
        # calculate dloss_dout @ dout/dlhs
        dloss_dlhs = out_grad @ rhs.T
        # calculate dloss_dout @ dout/drhs
        dloss_drhs = lhs.T @ out_grad
        return dloss_dlhs, dloss_drhs
```

Question: for the call to `linear2.gradient()`, which adjoint term is `out_grad`?

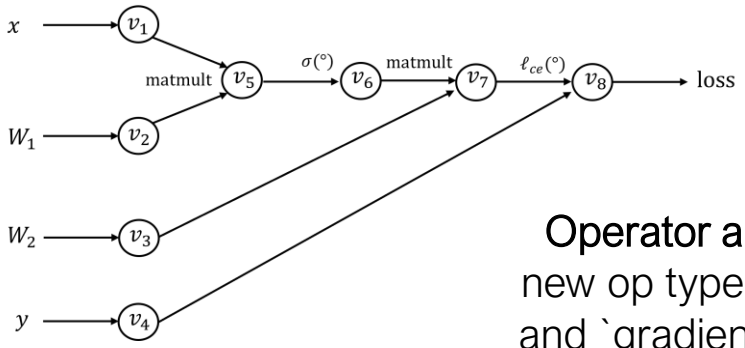
Answer: $\overline{v_7}$

Performant abstraction: for each layer, define `gradient()` function that returns (for each input):

$$\frac{\partial loss}{\partial input} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial input}$$

Operator abstraction and adjoints

$$\text{loss} = \ell_{ce}(\sigma(xW_1)W_2, y)$$



Operator abstraction: to define a new op type, implement `forward()` and `gradient()`, and it'll "just work"!

Ex: torch.nn.Linear, torch.nn.CrossEntropyLoss, torch.nn.Sigmoid

```
# forward
...
z0 = linear1.forward(x, W1)
z1 = elem_sigmoid.forward(z0)
z2 = linear2.forward(z1, W2)  # logits
loss_val = loss_ce.forward(z2, y)
# backward
dloss_dz2 = loss_ce.backward()
dloss_dz1, dloss_dW2 = linear2.gradient(dloss_dz2, ...)
dloss_dz0 = elem_sigmoid.backward(dloss_dz1, ...)
dloss_dx, dloss_dW1 = linear1.backward(dloss_dz0, ...)
# update params
W1 = W1 - stepsize * dloss_dW1
W2 = W2 - stepsize * dloss_dW2
```

Define adjoint $\overline{v_i} = \dfrac{\partial loss}{\partial v_i}$

(optional, rewrite to clean things up)
Let $z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\overline{v_8} = \frac{\partial loss}{\partial v_8} = 1$$

$$\overline{v_7} = \frac{\partial loss}{\partial v_7} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7} = \overline{v_8}\frac{\partial v_8}{\partial v_7} = (1) * (\text{softmax}(z_2) - e_y)$$

$$\overline{v_3} = \frac{\partial loss}{\partial v_3} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_3} = \overline{v_7}\frac{\partial v_7}{\partial v_3} = \sigma(xW_1)^T(s - e_y) = \frac{\partial loss}{\partial W_2}$$

$$\overline{v_6} = \frac{\partial loss}{\partial v_6} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = (\text{softmax}(z_2) - e_y)(v_3^T)$$

$$\overline{v_5} = \frac{\partial loss}{\partial v_5} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6}\frac{\partial v_6}{\partial v_5} = \overline{v_6}\frac{\partial v_6}{\partial v_5} = (\text{softmax}(z_2) - e_y)(v_3^T) \circ \acute{\sigma}(z_0)$$

$$\overline{v_2} = \frac{\partial loss}{\partial v_2} = \frac{\partial loss}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6}\frac{\partial v_6}{\partial v_5}\frac{\partial v_5}{\partial v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_2} = x^T(\text{softmax}(z_2) - e_y)W_2^T \circ \acute{\sigma}(xW_1) = \frac{\partial loss}{\partial W_1}$$

```
class MatMul(Op):
    def forward(self, lhs, rhs):
        return lhs @ rhs
    def gradient(self, out_grad, node):
        lhs, rhs = node.inputs
        # calculate dloss_dout @ dout/dlhs
        dloss_dlhs = out_grad @ rhs.T
        # calculate dloss_dout @ dout/drhs
        dloss_drhs = lhs.T @ out_grad
        return dloss_dlhs, dloss_drhs

...
```

```
class CrossEntropyLoss(Op):
    def forward(self, h, y):
        return ...
    def gradient(self, out_grad, node):
        ...
        return dloss_dh, dloss_dy

class Sigmoid(Op):
    def compute(self, x):
        return ...
    def gradient(self, out_grad, node):
        ...
        return dloss_dx
```

Tip: study these slides until you understand the connection between the code, Op.gradient(), and the math (adjoints).

Reverse AD algorithm (pseudocode)

```
def gradient(out):
    node_to_grad: dict[Node, list[ndarray]] = {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 node_to_grad
```

Dictionary that records a list of partial adjoints of each node

Sum up partial adjoints

(sum to handle the "multiple pathways" scenario)

"Propagates" partial adjoint to its input

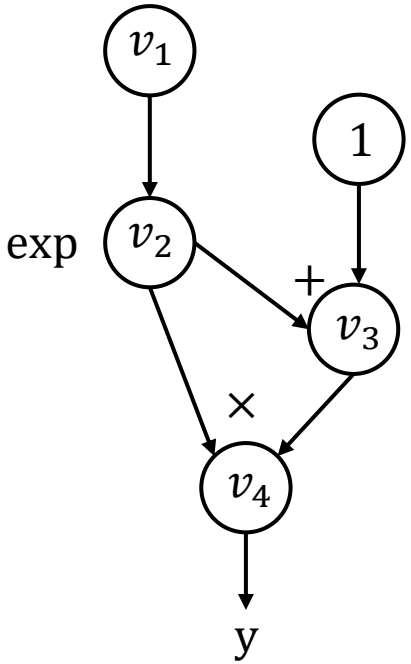
Reverse mode AD by extending computational graph (1/9)

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

Recall:
$\overline{v_i} = \frac{\partial y}{\partial v_i}$
$\overline{v_{i \to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$

```
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
```

Init to 1 because: $\frac{\partial y}{\partial y} = 1$



A naive implementation directly computes adjoint values. Totally works!
Interesting Idea: let's calculate the adjoint values by **augmenting** the computation graph!

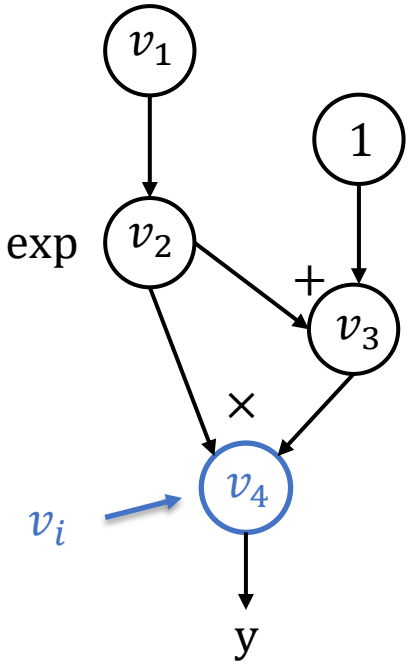
Reverse mode AD by extending computational graph (2/9)

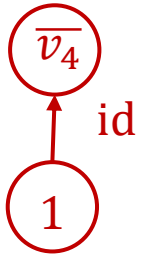
$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

$$\overline{v_4} = \frac{\partial y}{\partial v_4} = 1$$

Recall:
$$\overline{v_i} = \frac{\partial y}{\partial v_i}$$
$$\overline{v_{i \to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

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

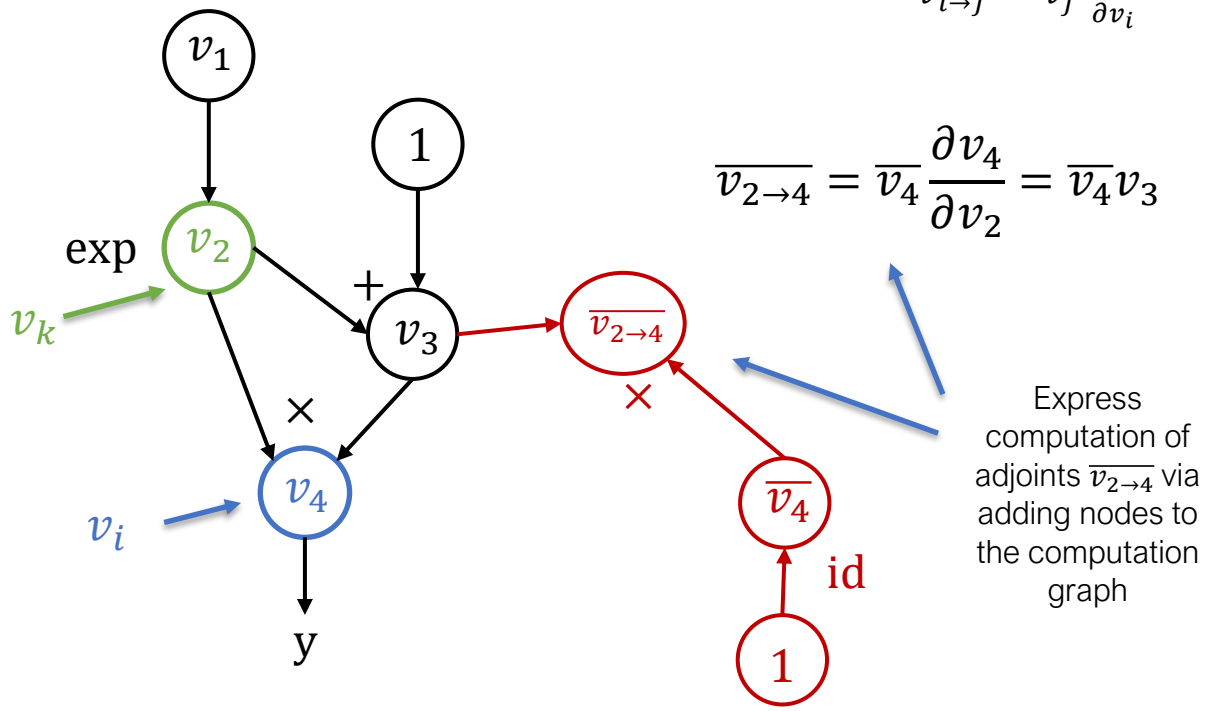
NOTE: id is identity function

Reverse mode AD by extending computational graph (3/9)

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

Recall:
$\overline{v_i} = \frac{\partial y}{\partial v_i}$
$\overline{v_{i \to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$

```
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
```



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

Express computation of adjoints $\overline{v_{2\to4}}$ via adding nodes to the computation graph

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

NOTE: id is identity function

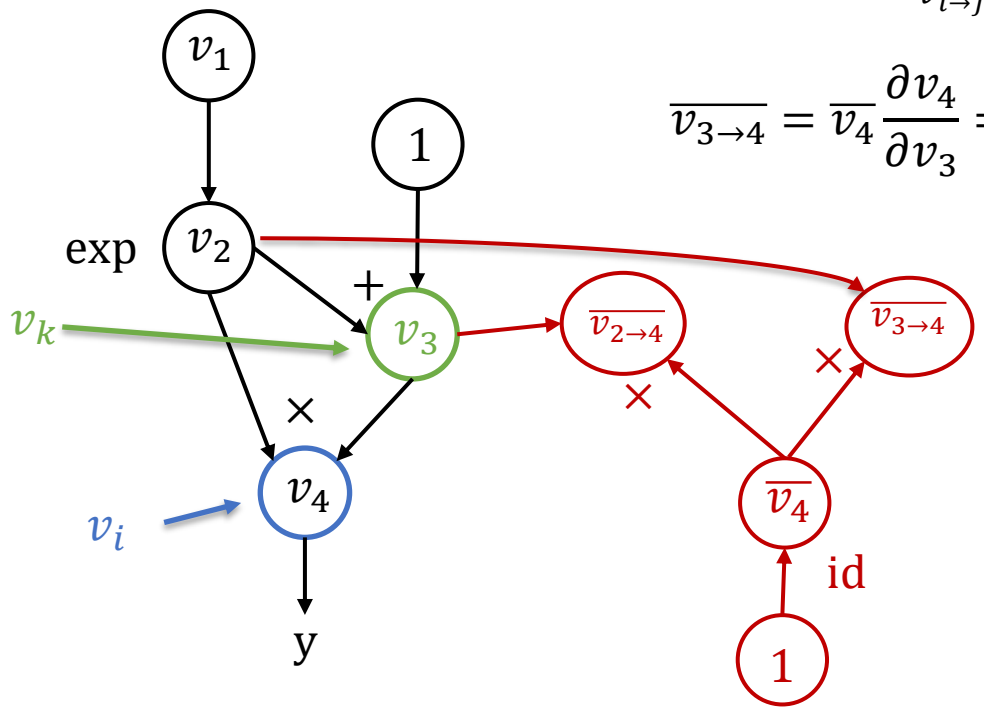
Reverse mode AD by extending computational graph (4/9)

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

Recall:
$\overline{v_i} = \frac{\partial y}{\partial v_i}$
$\overline{v_{i \to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$

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

```
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
```



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

NOTE: id is identity function

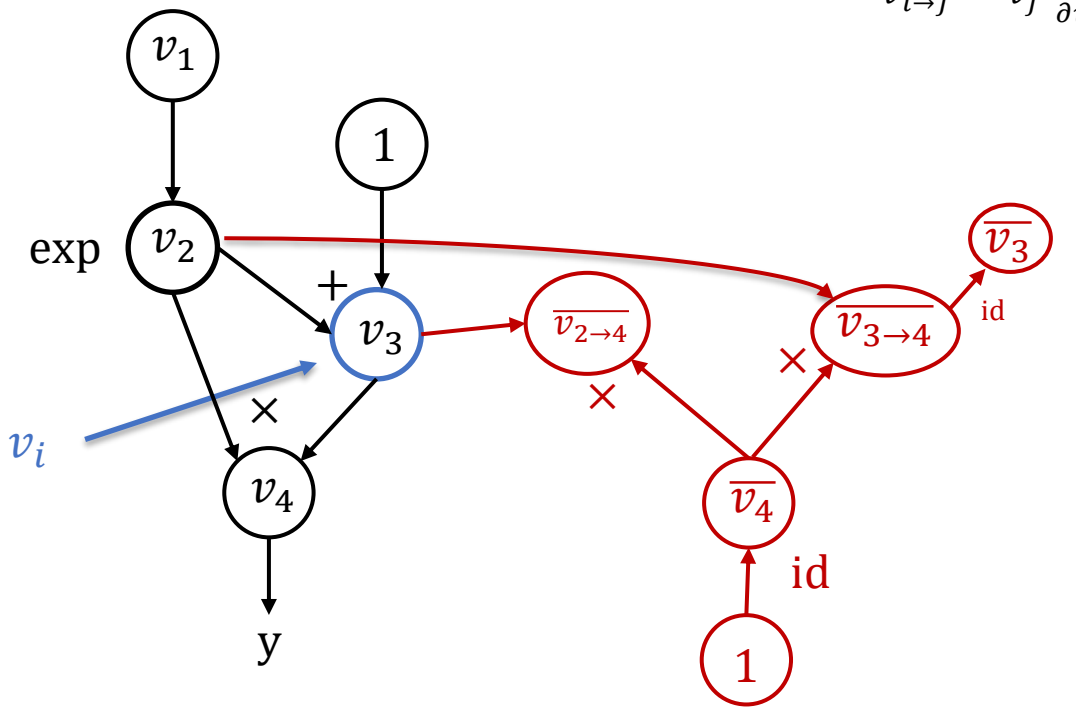
Reverse mode AD by extending computational graph (5/9)

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

$\overline{v_3} = \overline{v_{3\to4}}$

Recall:
$\overline{v_i} = \frac{\partial y}{\partial v_i}$
$\overline{v_{i\to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$

```
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: $[\overline{v_{2\to4}}]$
  3: $[\overline{v_{3\to4}}]$
  4: $[\overline{v_4}]$
}

NOTE: id is identity function

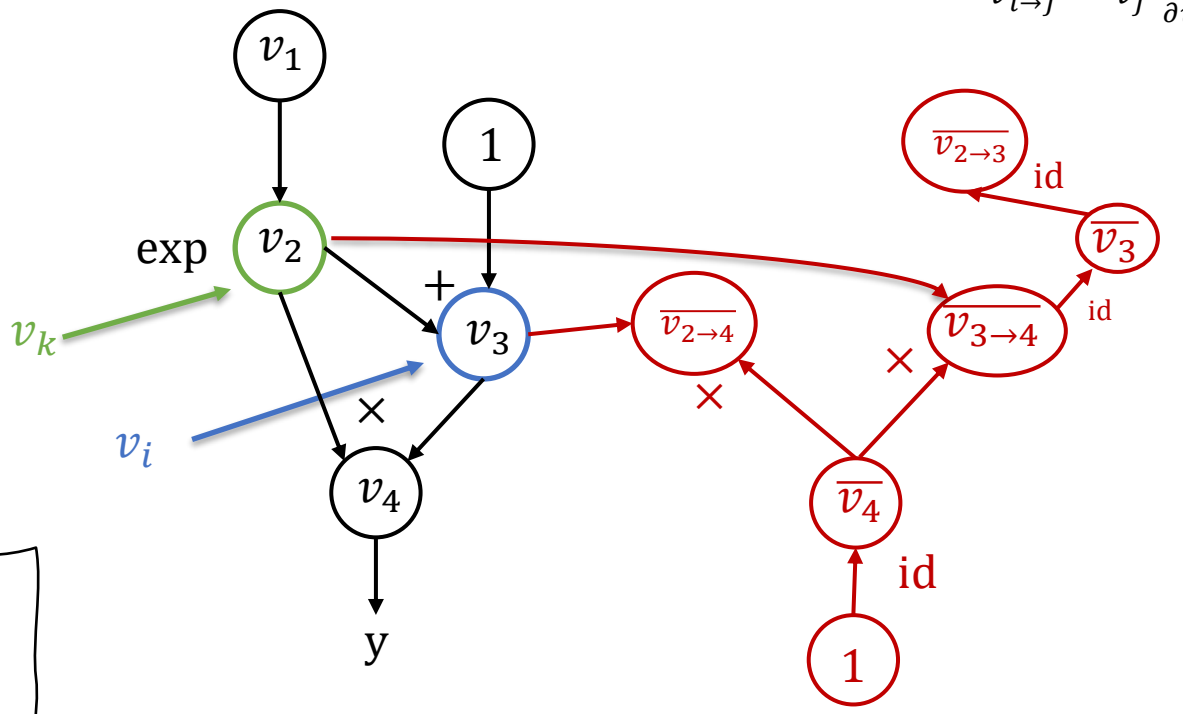
Reverse mode AD by extending computational graph (6/9)

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

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

Recall:
$$\overline{v_i} = \frac{\partial y}{\partial v_i}$$
$$\overline{v_{i\to j}} = \overline{v_j}\frac{\partial v_j}{\partial v_i}$$

```
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: $[\overline{v_{2\to4}}, \overline{v_{2\to3}}]$
  3: $[\overline{v_{3\to4}}]$
  4: $[\overline{v_4}]$
}

NOTE: id is identity function

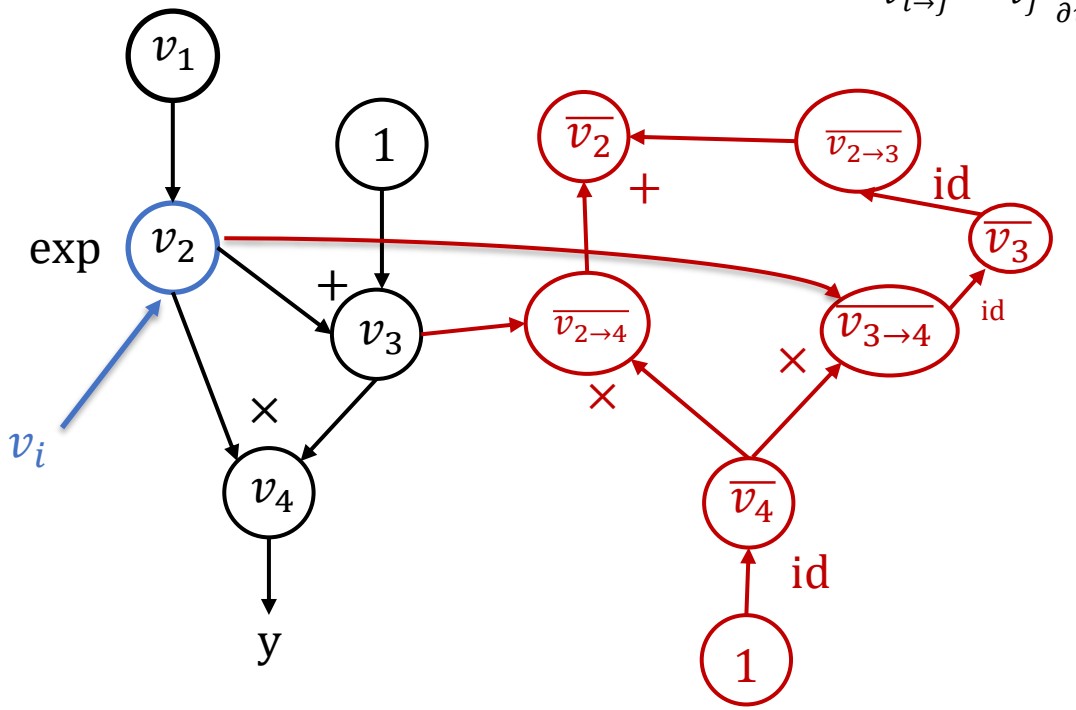
Reverse mode AD by extending computational graph (7/9)

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

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

Recall:
$\overline{v_i} = \frac{\partial y}{\partial v_i}$
$\overline{v_{i\to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$

```
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: $[\overline{v_{3\to4}}]$
  4: $[\overline{v_4}]$
}

NOTE: id is identity function

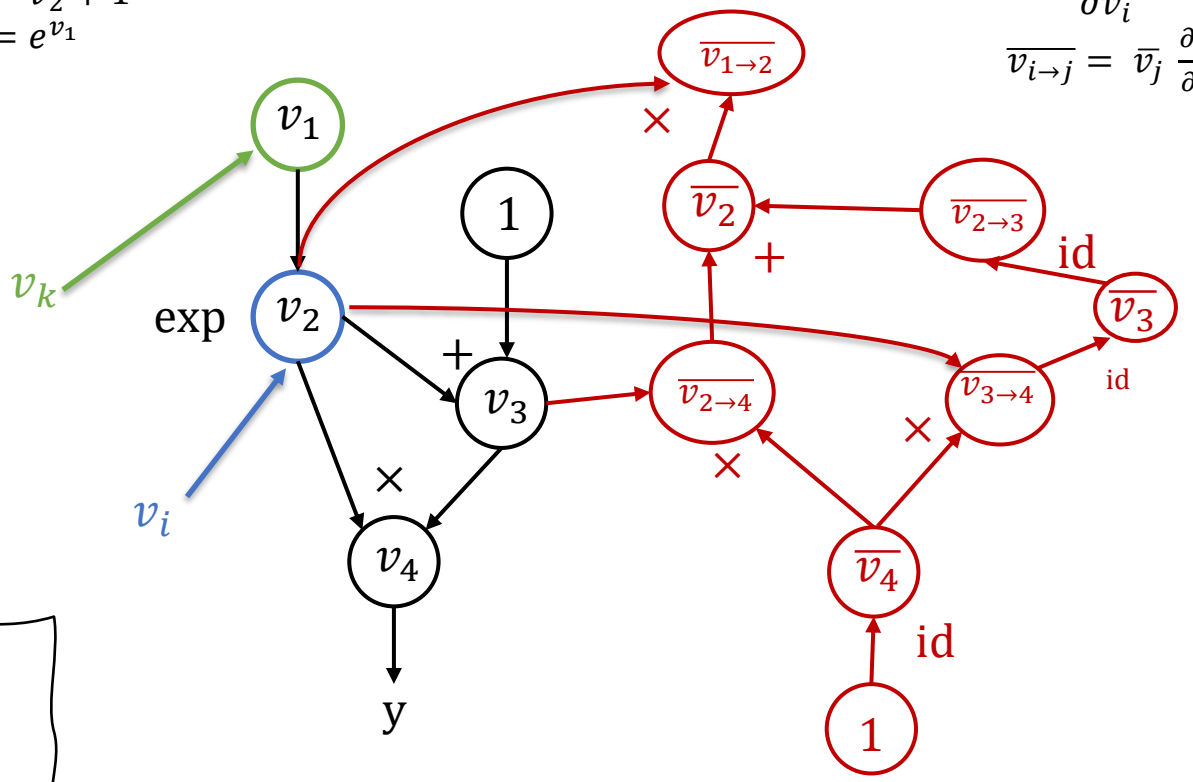
Reverse mode AD by extending computational graph (8/9)

```
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
```

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

$$\overline{v_{1\to 2}} = \overline{v_2}\frac{\partial v_2}{\partial v_1} = \overline{v_2} e^{v_1}$$

Recall:
$\bar{v}_i = \frac{\partial y}{\partial v_i}$
$\overline{v_{i\to j}} = \bar{v}_j \frac{\partial v_j}{\partial v_i}$



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

NOTE: id is identity function

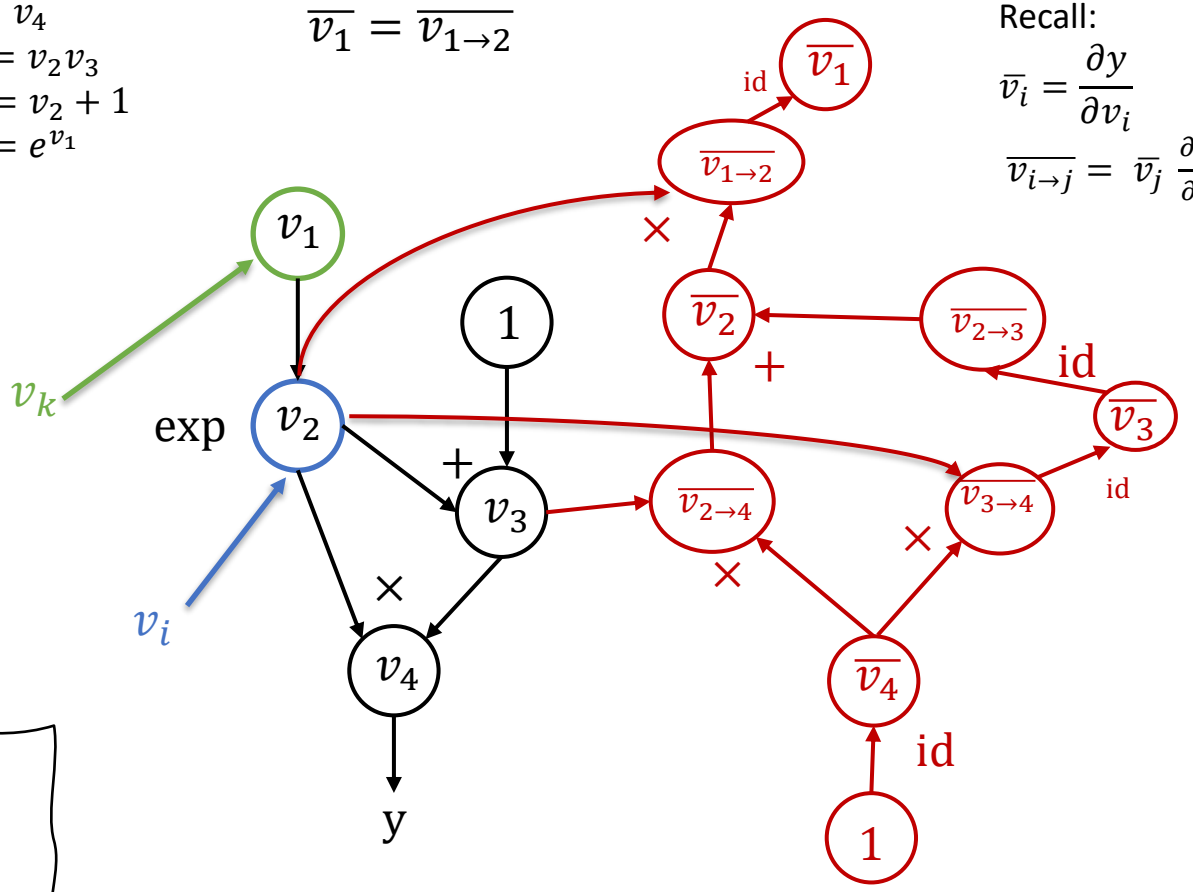
Reverse mode AD by extending computational graph (9/9)

```
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
```

$y = v_4$
$v_4 = v_2 v_3$
$v_3 = v_2 + 1$
$v_2 = e^{v_1}$

$\overline{v_1} = \overline{v_{1\to 2}}$

Recall:
$\overline{v_i} = \frac{\partial y}{\partial v_i}$
$\overline{v_{i\to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$



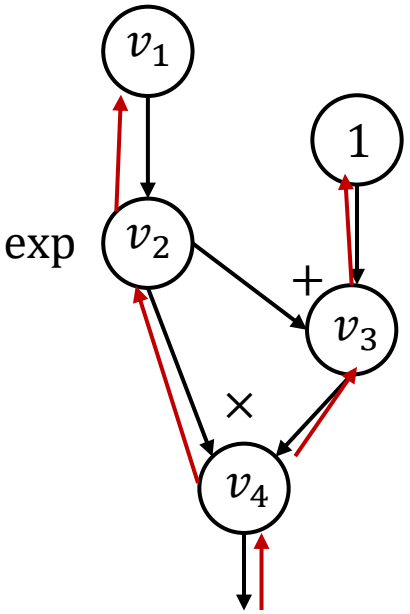
$i = 1$

```
node_to_grad: {
  1: [v̄_{1→2}]
  2: [v̄_{2→4}, v̄_{2→3}]
  3: [v̄_{3→4}]
  4: [v̄_4]
}
```

NOTE: id is identity function

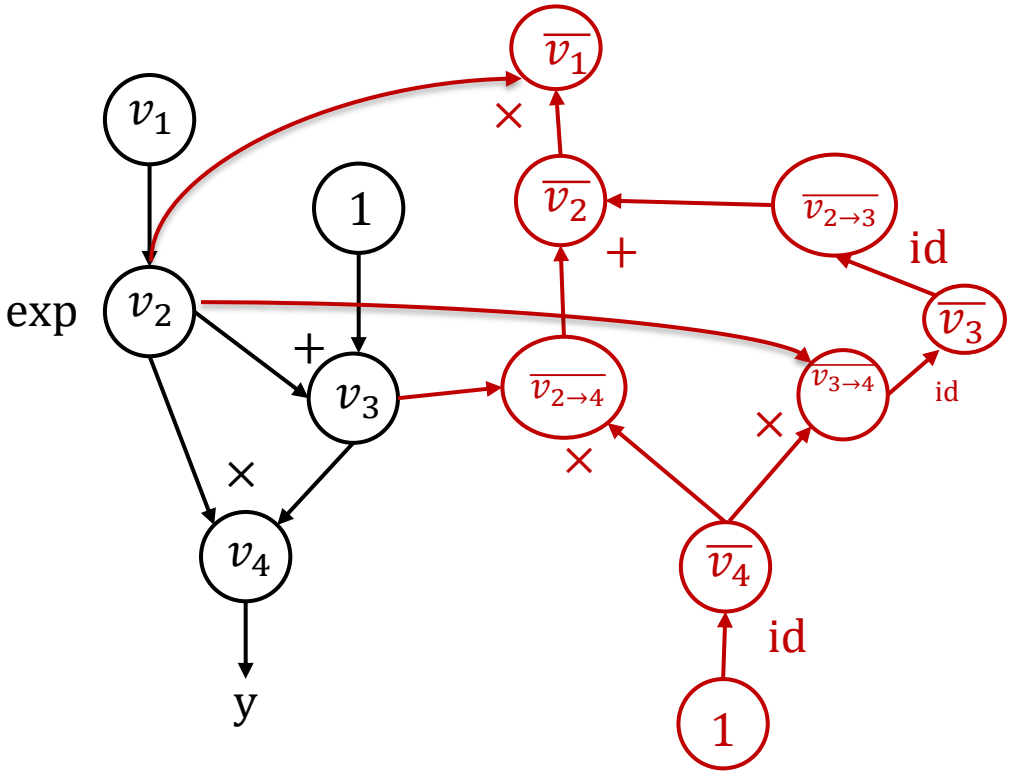
Reverse mode AD vs Backprop

Backprop



- Run backward operations the same forward graph
- Used in first generation deep learning frameworks (caffe, cuda-convnet)

Reverse mode AD by extending computational graph



- Construct separate graph nodes for adjoints
- Used by modern deep learning frameworks

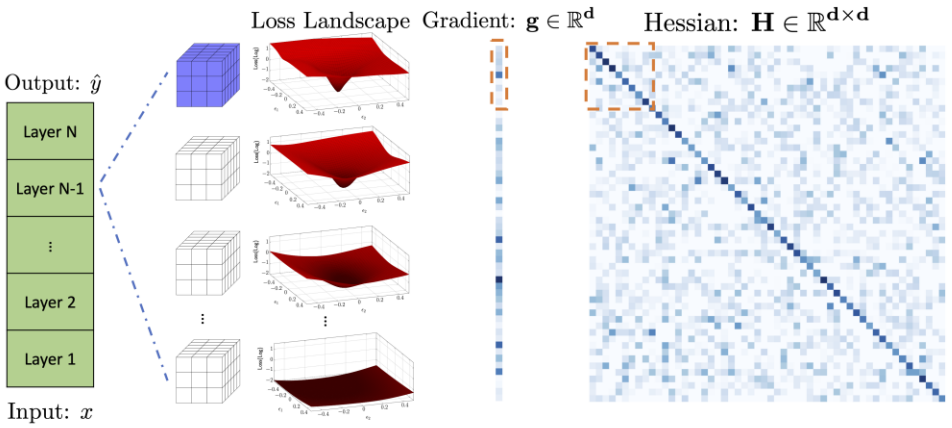
Pro: deep learning frameworks can apply optimizations for both forward and backward uniformly (ex: operator fusion)

Gradient of gradient: Second order methods

The result of reverse mode AD a computational graph.

We can extend that graph further by running reverse mode AD again to yield the gradient of the gradient (aka the second derivative!).

This can lead to **second-order optimization** methods (recall that SGD is a first-order optimization method)!



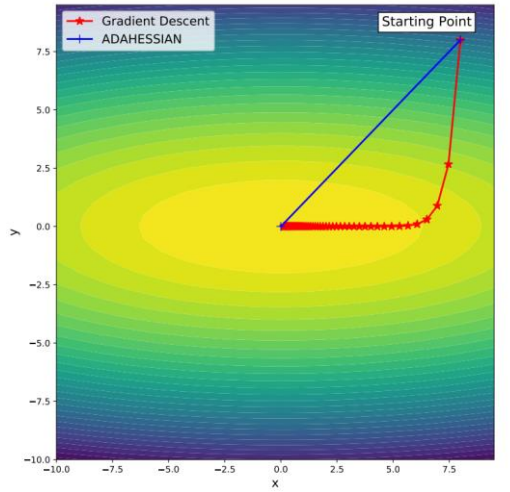


Figure 1: *The trajectory of gradient descent and* AdaHessian *on a simple 2D quadratic function* $f(x, y) = 10x^2 + y^2$. *Gradient descent converges very slowly, even though this problem has a reasonable condition number. However,* AdaHessian *converges to the optimum in just one step. This is because second order methods normalize the curvature difference between* x *and* y *axis by preconditioning the gradient vector before the weight update (by rescaling and rotating the gradient vector).*

(As of 2026) Although libraries like pytorch support calculating the gradient of the gradient, in practice second order methods aren't popular in deep learning due to its high computation cost. For an example, see "ADAHessian" (2021).