

# **15-442/15-642: Machine Learning Systems**

## **Automatic Differentiation**

Spring 2026

Tianqi Chen and Zhihao Jia  
Carnegie Mellon University

# Outline

General introduction to different differentiation methods

Reverse mode automatic differentiation

# Outline

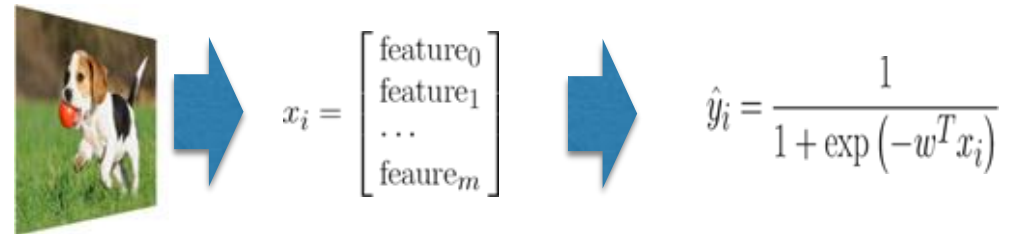
General introduction to different differentiation methods

Reverse mode automatic differentiation

# Recap: Elements of Machine Learning

- **Model(hypothesis) class**  
*A parameterized function that describes how do we map inputs to predictions*
- **Loss function**  
How “well” are we doing for a given set of parameters
- **Training (optimization) method**  
A procedure to find a set of parameters that minimizes the loss

Computing the loss function gradient with respect to hypothesis class parameters is the most common operation in machine learning



## Logistic regression model

$$L(w) = \sum_{i=1}^n l(y_i, \hat{y}_i) + \lambda \|w\|^2$$

## Regularized loss function

$$w \leftarrow w - \eta \nabla_w L(w)$$

## Stochastic gradient descent

# Numerical Differentiation

Directly compute the partial gradient by definition

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

A more numerically accurate way to approximate the gradient

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

Suffer from numerical error, less efficient to compute

# Numerical Gradient Checking

However, numerical differentiation is a powerful tool to check an implement of an automatic differentiation algorithm in unit test cases

$$\delta^T \nabla_{\theta} f(\theta) = \frac{f(\theta + \epsilon \delta) - f(\theta - \epsilon \delta)}{2\epsilon} + o(\epsilon^2)$$

Pick  $\delta$  from unit ball, check the above invariance.

# Symbolic Differentiation

Write down the formulas, derive the gradient by sum, product and chain rules

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

Naively do so can result in wasted computations

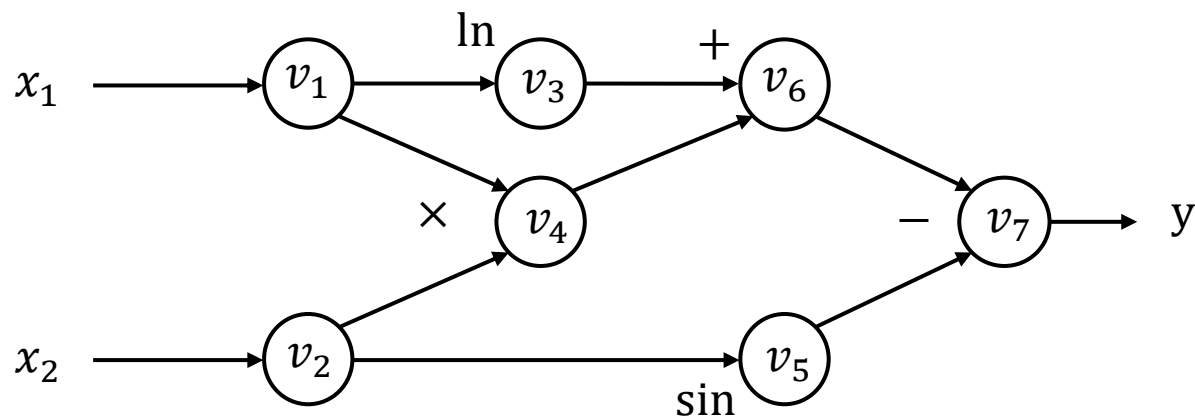
Example:

$$f(\theta) = \prod_{i=1}^n \theta_i \quad \frac{f(\theta)}{\partial\theta_k} = \prod_{j \neq k}^n \theta_j$$

Cost  $n(n-2)$  multiplies to compute all partial gradients

# Recap: Computational Graph

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



## Forward evaluation trace

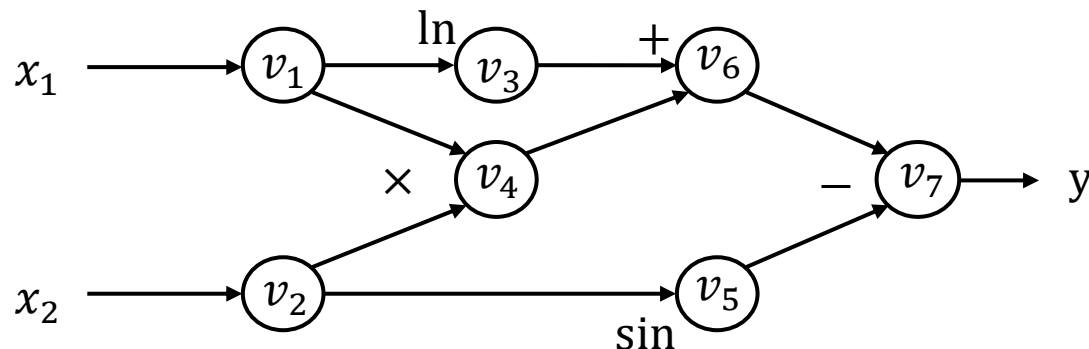
$$\begin{aligned} v_1 &= x_1 = 2 \\ v_2 &= x_2 = 5 \\ v_3 &= \ln v_1 = \ln 2 = 0.693 \\ v_4 &= v_1 \times v_2 = 10 \\ v_5 &= \sin v_2 = \sin 5 = -0.959 \\ v_6 &= v_3 + v_4 = 10.693 \\ v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\ y &= v_7 = 11.652 \end{aligned}$$

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



# Forward Mode Automatic Differentiation (AD)

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



Forward evaluation trace

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

**Define**  $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$

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

Forward AD trace

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

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

# Limitations of Forward Mode AD

- For  $f: \mathbb{R}^n \rightarrow \mathbb{R}^k$ , 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  $n$ .
- In order to resolve the problem efficiently, we need to use another kind of AD.

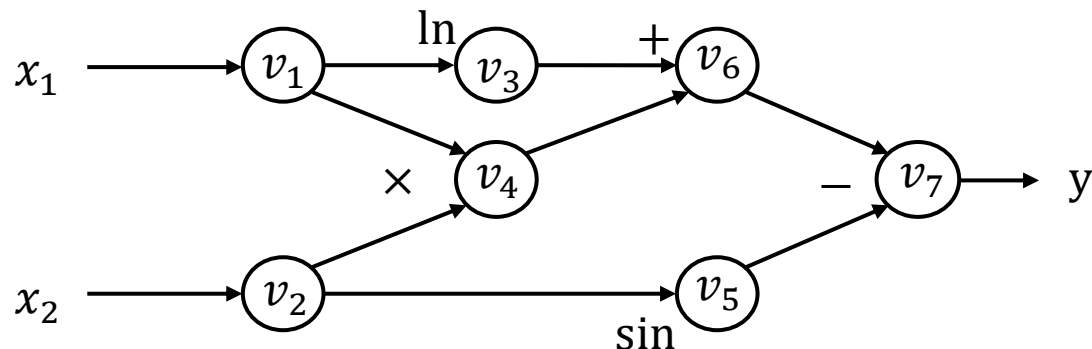
# Outline

General introduction to different differentiation methods

Reverse mode automatic differentiation

# Reverse Mode Automatic Differentiation(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 adjoint  $\bar{v}_i = \frac{\partial y}{\partial v_i}$

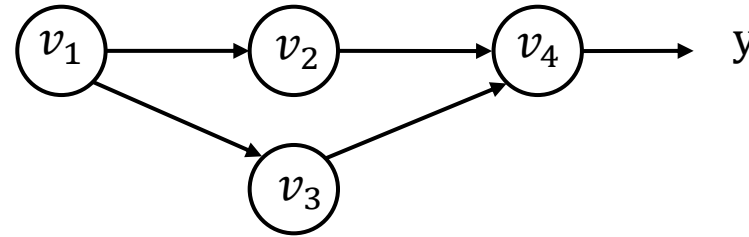
We can then compute the  $\bar{v}_i$  iteratively in the **reverse** topological order of the computational graph

Reverse AD evaluation trace

$$\begin{aligned}
 \bar{v}_7 &= \frac{\partial y}{\partial v_7} = 1 \\
 \bar{v}_6 &= \bar{v}_7 \frac{\partial v_7}{\partial v_6} = \bar{v}_7 \times 1 = 1 \\
 \bar{v}_5 &= \bar{v}_7 \frac{\partial v_7}{\partial v_5} = \bar{v}_7 \times (-1) = -1 \\
 \bar{v}_4 &= \bar{v}_6 \frac{\partial v_6}{\partial v_4} = \bar{v}_6 \times 1 = 1 \\
 \bar{v}_3 &= \bar{v}_6 \frac{\partial v_6}{\partial v_3} = \bar{v}_6 \times 1 = 1 \\
 \bar{v}_2 &= \bar{v}_5 \frac{\partial v_5}{\partial v_2} + \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_5 \times \cos v_2 + \bar{v}_4 \times v_1 = -0.284 + 2 = 1.716 \\
 \bar{v}_1 &= \bar{v}_4 \frac{\partial v_4}{\partial v_1} + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \bar{v}_4 \times v_2 + \bar{v}_3 \frac{1}{v_1} = 5 + \frac{1}{2} = 5.5
 \end{aligned}$$

# Derivation for the Multiple Pathway Case

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



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

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

Define partial adjoint  $\overline{v_{i \rightarrow j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}$  for each input output node pair  $i$  and  $j$

$$\overline{v_i} = \sum_{j \in \text{next}(i)} \overline{v_{i \rightarrow j}}$$

We can compute partial adjoints separately then sum them together

# Reverse AD Algorithm

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

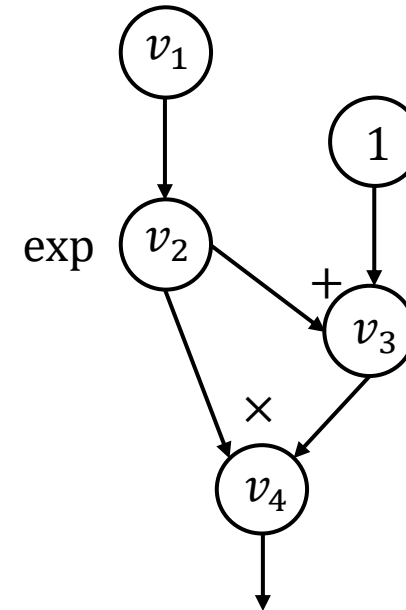
Dictionary that records a list of partial adjoints of each node

Sum up partial adjoints

“Propagates” partial adjoint to its input


# Reverse Mode AD by Extending Computational Graph

```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
         $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to  $\text{node\_to\_grad}[k]$   
    return adjoint of input  $\bar{v}_{input}$ 
```

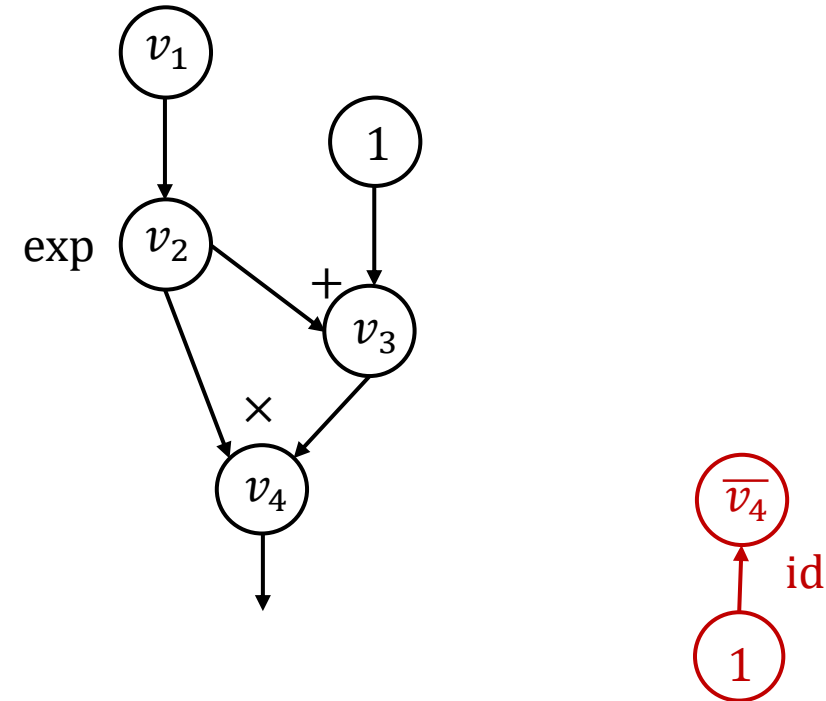


Our previous examples compute adjoint values directly by hand.  
How can we construct a computational graph that calculates the adjoint values?

# Reverse mode AD by extending computational graph

```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
          $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to  $\text{node\_to\_grad}[k]$   
    return adjoint of input  $\bar{v}_{\text{input}}$ 
```

```
i = 4  
node_to_grad: {  
    4: [ $\bar{v}_4$ ]  
}
```



NOTE: id is identity function

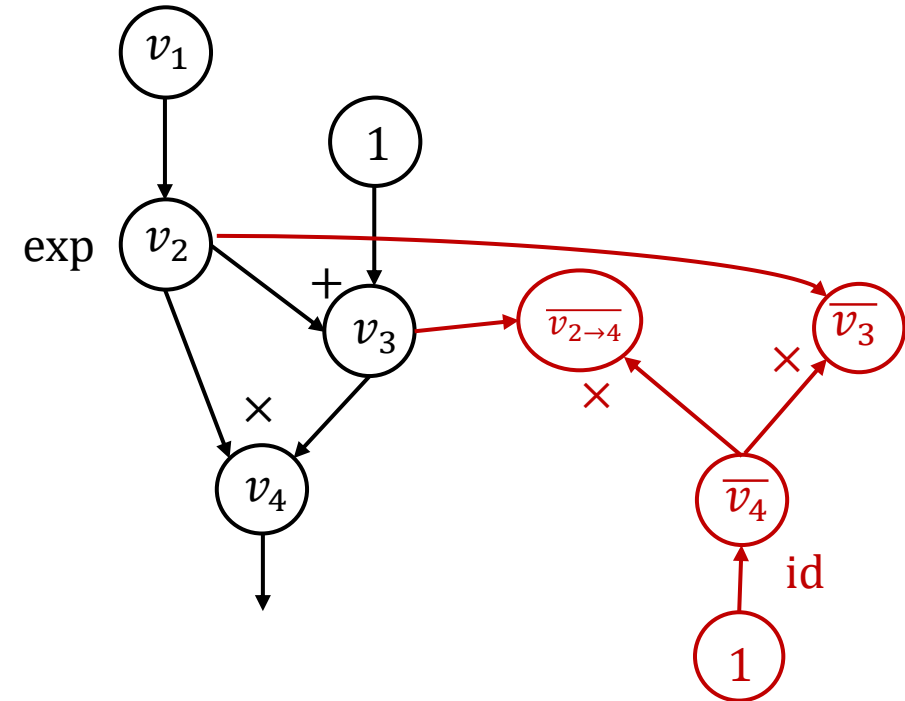


# Reverse Mode AD by Extending Computational Graph

```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
         $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to node_to_grad[k]  
    return adjoint of input  $\bar{v}_{input}$ 
```



```
i = 4  
node_to_grad: {  
    2: [ $\bar{v}_{2 \rightarrow 4}$ ]  
    3: [ $\bar{v}_3$ ]  
    4: [ $\bar{v}_4$ ]  
}
```



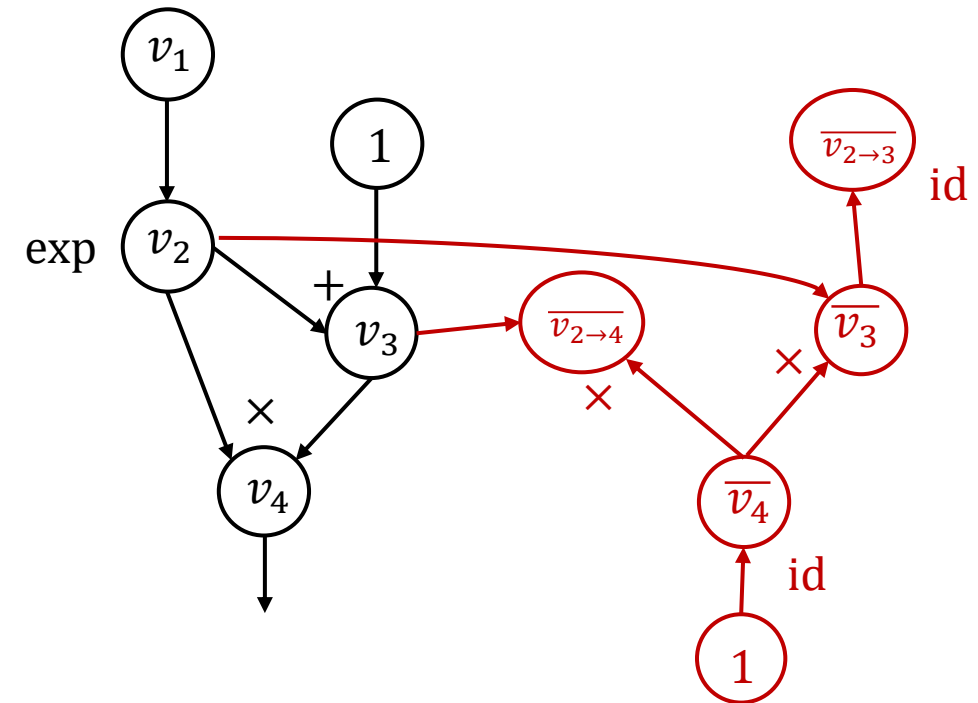
NOTE: id is identity function

# Reverse Mode AD by Extending Computational Graph

```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
         $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to  $\text{node\_to\_grad}[k]$   
    return adjoint of input  $\bar{v}_{\text{input}}$ 
```



```
i = 3  
node_to_grad: {  
    2: [ $\bar{v}_{2 \rightarrow 4}$ ,  $\bar{v}_{2 \rightarrow 3}$ ]  
    3: [ $\bar{v}_3$ ]  
    4: [ $\bar{v}_4$ ]  
}
```

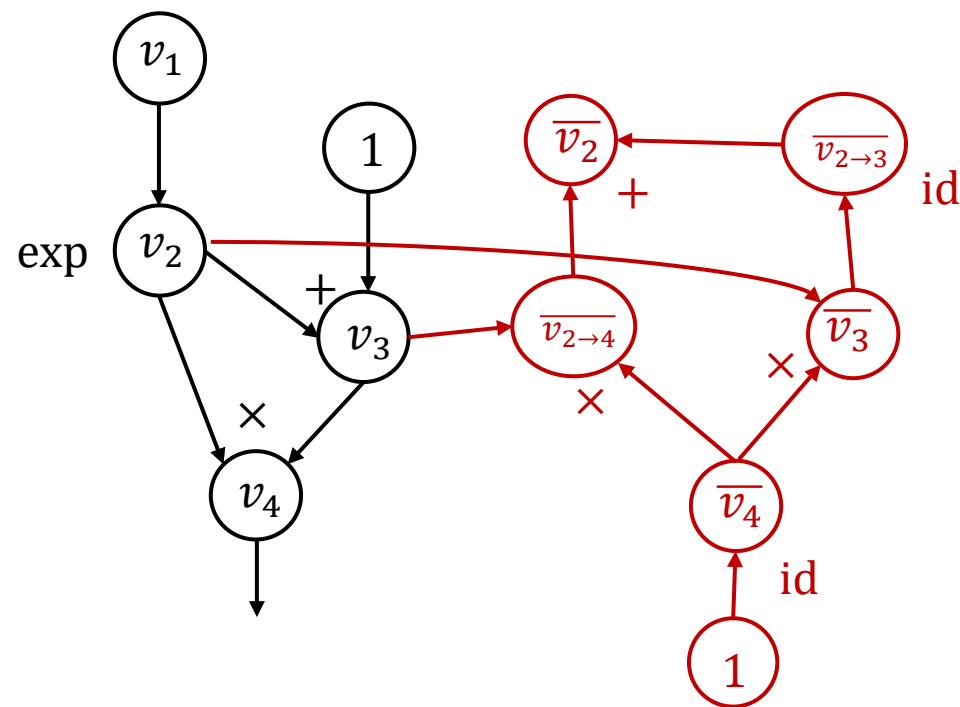


NOTE: id is identity function

# Reverse Mode AD by Extending Computational Graph

```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
        ➔  $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to  $\text{node\_to\_grad}[k]$   
    return adjoint of input  $\bar{v}_{\text{input}}$ 
```

```
i = 2  
node_to_grad: {  
    2: [ $\bar{v}_{2 \rightarrow 4}$ ,  $\bar{v}_{2 \rightarrow 3}$ ]  
    3: [ $\bar{v}_3$ ]  
    4: [ $\bar{v}_4$ ]  
}
```



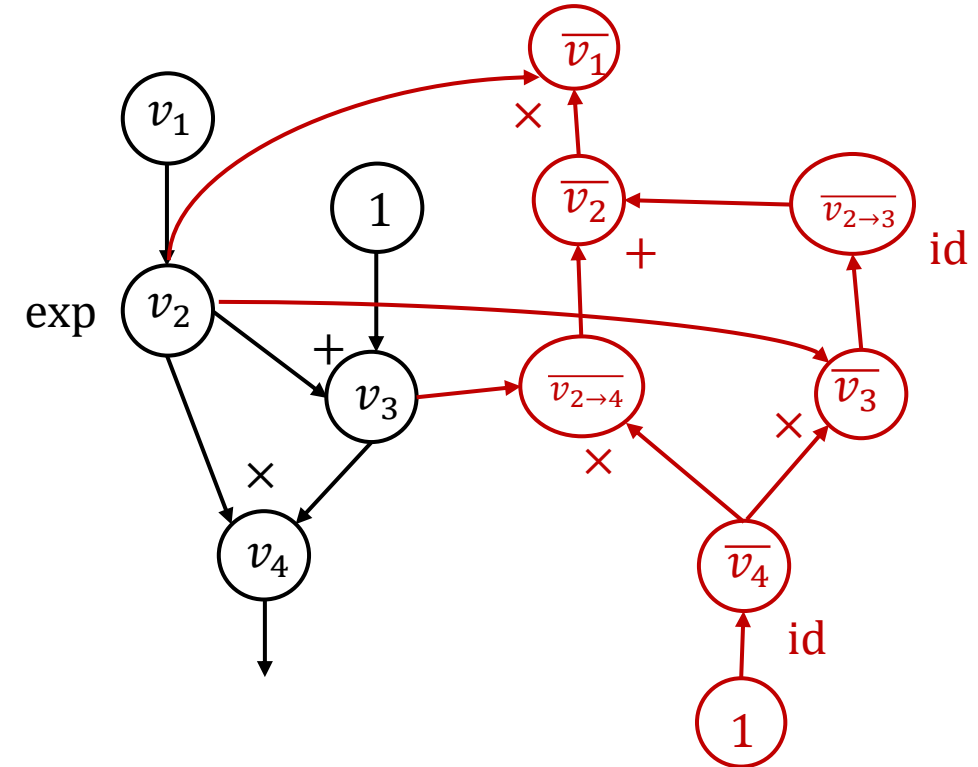
NOTE: id is identity function

# Reverse Mode AD by Extending Computational Graph

```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
         $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to node_to_grad[k]  
    return adjoint of input  $\bar{v}_{input}$ 
```



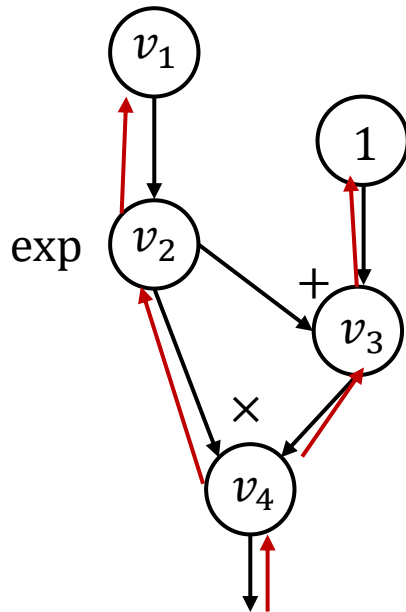
```
i = 2  
node_to_grad: {  
    1: [ $\bar{v}_1$ ]  
    2: [ $\bar{v}_{2 \rightarrow 4}$ ,  $\bar{v}_{2 \rightarrow 3}$ ]  
    3: [ $\bar{v}_3$ ]  
    4: [ $\bar{v}_4$ ]  
}
```



NOTE: id is identity function

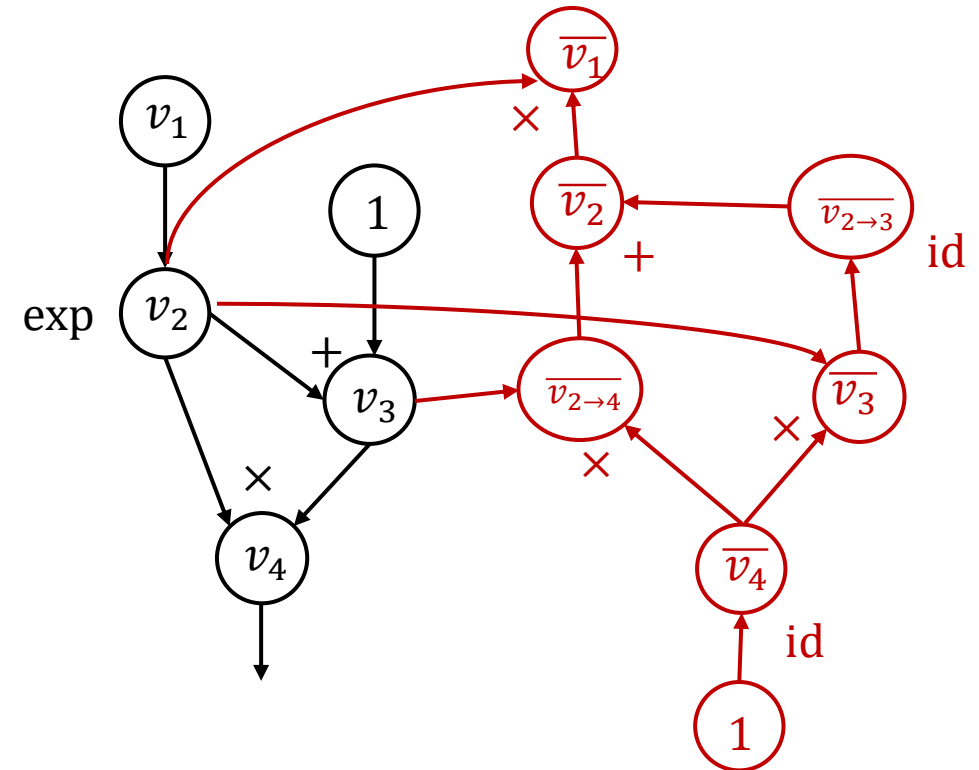
# Reverse Mode AD vs Backprop

**Backprop**



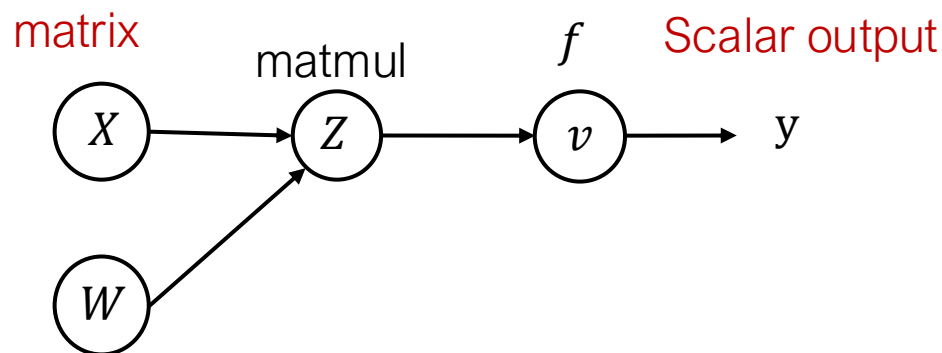
- Run backward operations the same forward graph
- Used in first generation deep learning frameworks (caffe, cuda-convnet)

**Reverse mode AD by extending computational graph**



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

# Reverse mode AD on Tensors



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

Forward evaluation trace

$$Z_{ij} = \sum_k X_{ik} W_{kj}$$

$$v = f(Z)$$

Reverse evaluation in scalar form

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

Forward matrix form

$$Z = XW$$

$$v = f(Z)$$

Reverse matrix form

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

# Reverse AD Algorithm

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

Dictionary that records a list of partial adjoints of each node

Sum up partial adjoints

"Propagates" partial adjoint to its input

# Discussions

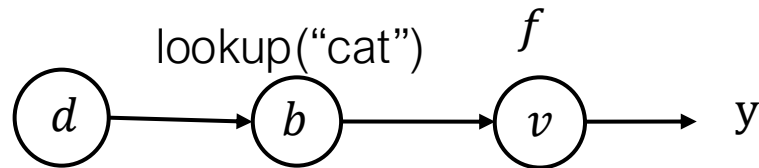
What are the pros/cons of backprop and reverse mode AD



# Handling Gradient of Gradient

- The result of reverse mode AD is still a computational graph
- We can extend that graph further by composing more operations and run reverse mode AD again on the gradient
- Part of homework 1

# Reverse Mode AD on Data Structures



**Define adjoint** data structure

$$\bar{d} = \{ \text{"cat"} : \frac{\partial y}{\partial a_0}, \text{"dog"} : \frac{\partial y}{\partial a_1} \}$$

Forward evaluation trace

$$\begin{aligned} d &= \{ \text{"cat"} : a_0, \text{"dog"} : a_1 \} \\ b &= d [\text{"cat"}] \\ v &= f(b) \end{aligned}$$

Reverse evaluation

$$\begin{aligned} \bar{b} &= \frac{\partial v}{\partial b} \bar{v} \\ \bar{d} &= \{ \text{"cat"} : \bar{b} \} \end{aligned}$$

- Key take away: Define “adjoint value” usually in the same data type as the forward value and adjoint propagation rule. Then the sample algorithm works.
- Do not need to support the general form in our framework, but we may support “tuple values”