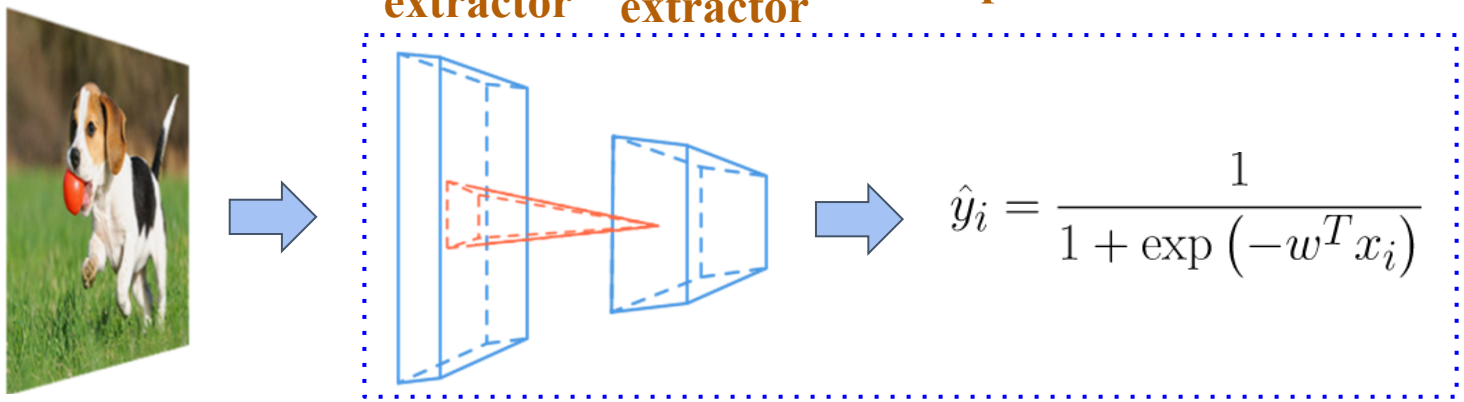
Lecture 4: Automatic Differentiation

CSE599G1: Spring 2017

Announcement

- Assignment 1 is out today, due in 2 weeks (Apr 20th, 5pm)

Model Training Overview



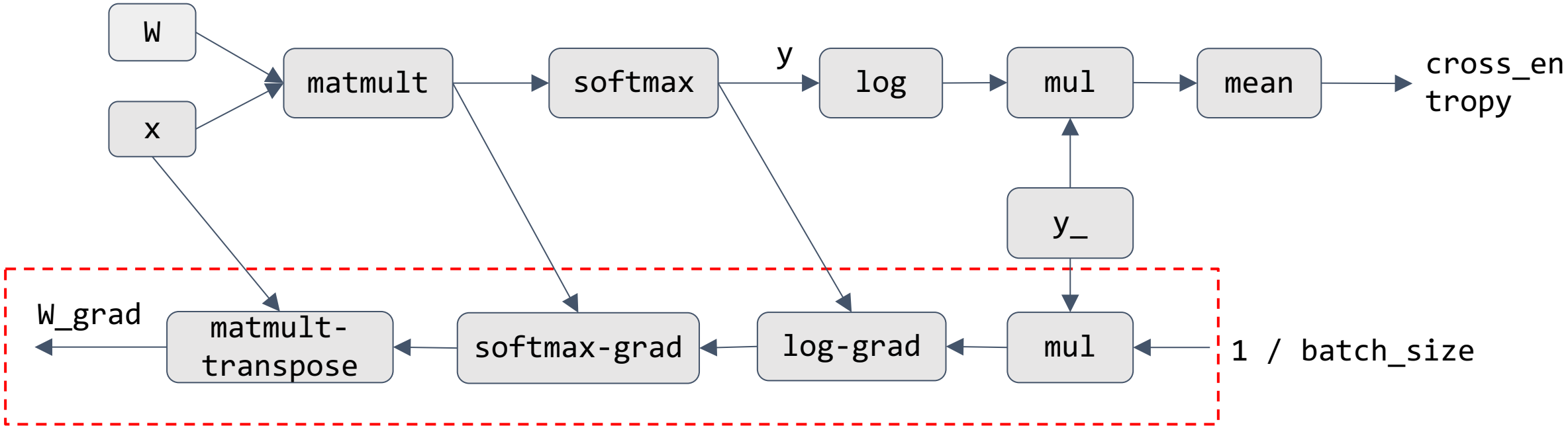
Objective

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

Training

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

Model Training Overview



Numerical Differentiation

- We can approximate the gradient using

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e_i}) - f(\boldsymbol{x})}{h}$$

- Reduce the truncation error by using center difference

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e_i}) - f(\boldsymbol{x} - h\boldsymbol{e_i})}{2h}$$

- Still suffer from rounding error.
- A powerful tool to check the correctness of implementation, usually use h = 1e-6.

Symbolic Differentiation

- Input formulae is a symbolic expression tree (computation graph).
- Implement differentiation rules, e.g., product rule, sum rule, chain rule

$$\frac{d(f+g)}{dx} = \frac{df}{dx} + \frac{dg}{dx} \qquad \frac{d(fg)}{dx} = \frac{df}{dx}g + f\frac{dg}{dx} \qquad \frac{d\big(h(x)\big)}{dx} = \frac{df\big(g(x)\big)}{dx} \cdot \frac{dg(x)}{x}$$

- For complicated functions, the resultant expression can be exponentially large.
- Wasteful to keep around intermediate symbolic expressions if we only need a numeric value of the gradient in the end.

$$y = \prod_{i=1}^{100} x_i \qquad \frac{\partial y}{\partial x_i} = \prod_{j \neq i} x_j$$

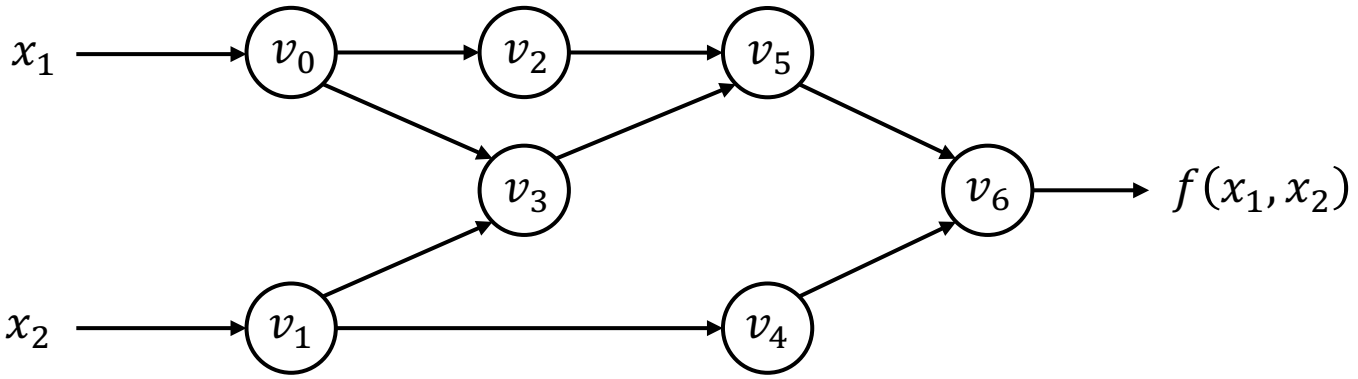
Automatic Differentiation (AutoDiff)

- **Intuition**: can we interleave symbolic differentiation and simplification?

- **Key idea**: apply symbolic differentiation at elementary operation level and keep intermediate results

AutoDiff in Forward Mode

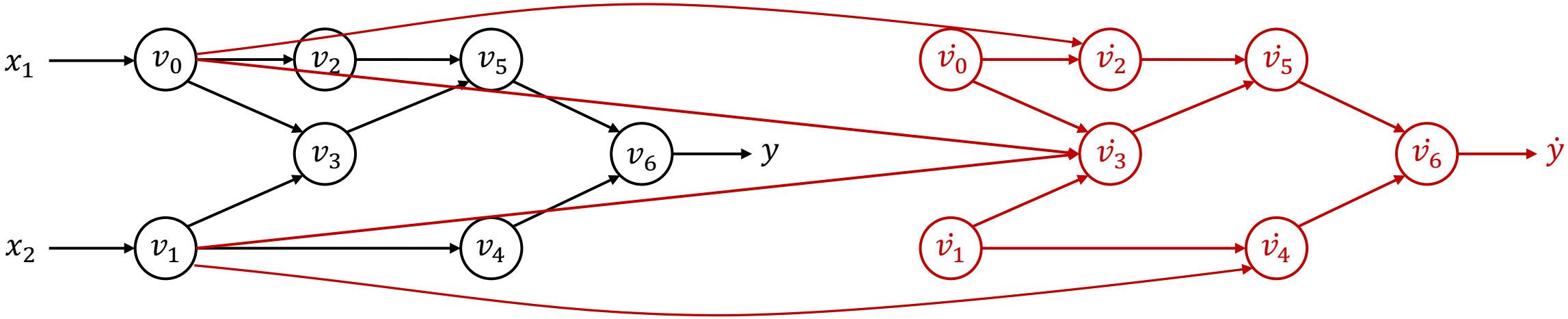
$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$



Each node is a (input/intermediate/output) variable. Computation graph (a DAG) with variable ordering from topological sort.

Forward Evaluation Trace

$$\begin{aligned}
v_0 &= x_1 && = 2 \\
v_1 &= x_2 && = 5 \\
v_2 &= \ln v_0 && = \ln 2 \\
v_3 &= v_0 \times v_1 && = 2 \times 5 \\
v_4 &= \sin v_1 && = \sin 5 \\
v_5 &= v_2 + v_3 && = 0.693 + 10 \\
v_6 &= v_5 - v_4 && = 10.693 + 0.959 \\
y &= v_6 && = 11.652
\end{aligned}$$



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

For every node, introduce a derivative node, $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$

Forward Evaluation Trace

$$\begin{aligned}
v_0 &= x_1 && = 2 \\
v_1 &= x_2 && = 5 \\
v_2 &= \ln v_0 && = \ln 2 \\
v_3 &= v_0 \times v_1 && = 2 \times 5 \\
v_4 &= \sin v_1 && = \sin 5 \\
v_5 &= v_2 + v_3 && = 0.693 + 10 \\
v_6 &= v_5 - v_4 && = 10.693 + 0.959 \\
y &= v_6 && = 11.652
\end{aligned}$$

Forward Derivative Trace

$$\begin{aligned}
\dot{v}_0 &= \dot{x}_1 && = 1 \\
\dot{v}_1 &= \dot{x}_2 && = 0 \\
\dot{v}_2 &= \dot{v}_0 / v_0 && = 1/2 \\
\dot{v}_3 &= \dot{v}_0 \times v_1 + v_0 \times \dot{v}_1 && = 1 \times 5 + 0 \times 2 \\
\dot{v}_4 &= \dot{v}_1 \times \cos v_1 && = 0 \times \cos 5 \\
\dot{v}_5 &= \dot{v}_2 + \dot{v}_3 && = 0.5 + 5 \\
\dot{v}_6 &= \dot{v}_5 - \dot{v}_4 && = 5.5 - 0 \\
\dot{y} &= \dot{v}_6 && = 5.5
\end{aligned}$$

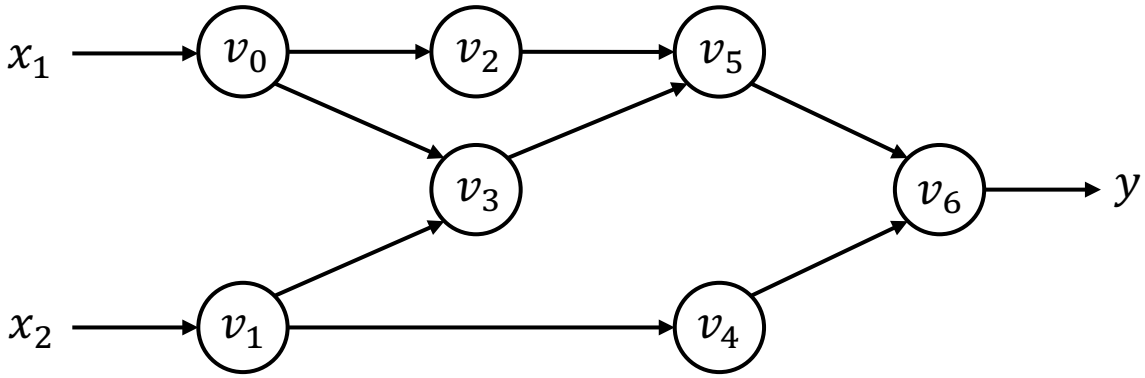
Now we have $\frac{\partial y}{\partial x_1}$.

Problem?

$f\colon \mathbb{R}^n \to \mathbb{R}^m$

- Needs n forward passes to get gradient wrt each inputs.
- However, DNN typically has a LARGE number of inputs (*weights considered as inputs too*), and a small number of outputs.
- Autodiff in **reverse mode** computes all gradients in m backward passes, and for many DNN, $m = 1$, so a single back pass (back propagation)!

AutoDiff in Reverse Mode



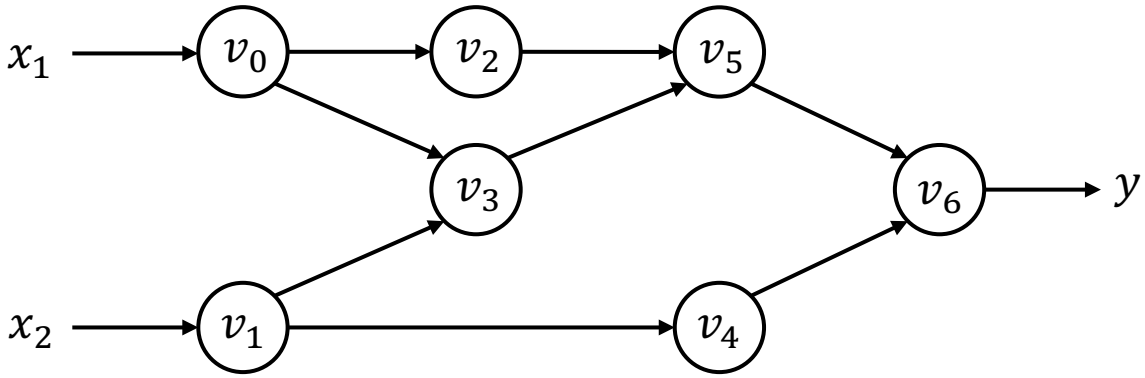
For each node v_i, introduce an **adjoint node**, i.e. derivative of output wrt this node,

$$\bar{v}_i = \frac{\sum_j \partial y_j}{\partial v_i}$$

Forward Evaluation Trace

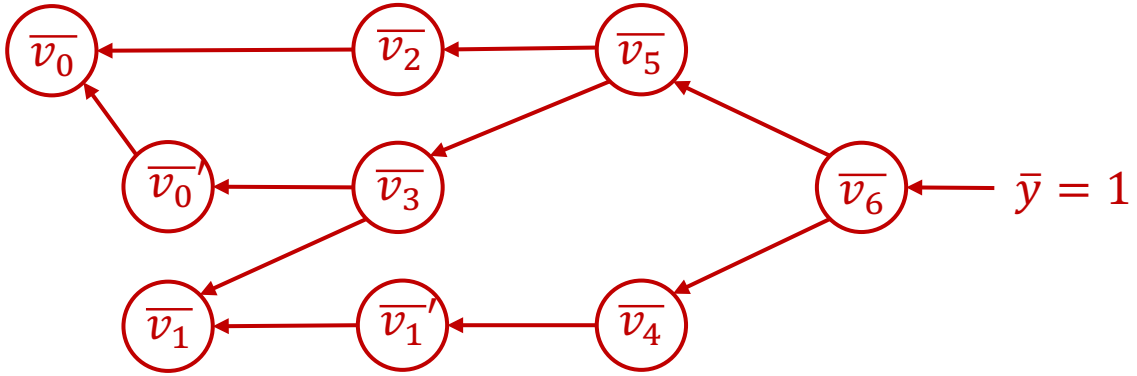
$$\begin{aligned}
v_0 &= x_1 && = 2 \\
v_1 &= x_2 && = 5 \\
v_2 &= \ln v_0 && = \ln 2 \\
v_3 &= v_0 \times v_1 && = 2 \times 5 \\
v_4 &= \sin v_1 && = \sin 5 \\
v_5 &= v_2 + v_3 && = 0.693 + 10 \\
v_6 &= v_5 - v_4 && = 10.693 + 0.959 \\
y &= v_6 && = 11.652
\end{aligned}$$

AutoDiff in Reverse Mode



Forward Evaluation Trace

$$
\begin{aligned}
v_0 &= x_1 && = 2 \\
v_1 &= x_2 && = 5 \\
v_2 &= \ln v_0 && = \ln 2 \\
v_3 &= v_0 \times v_1 && = 2 \times 5 \\
v_4 &= \sin v_1 && = \sin 5 \\
v_5 &= v_2 + v_3 && = 0.693 + 10 \\
v_6 &= v_5 - v_4 && = 10.693 + 0.959 \\
y &= v_6 && = 11.652
\end{aligned}
$$



Reverse Adjoint Trace

$$\overline{v_0} = \overline{v_0}' + \overline{v_2}\frac{\partial v_2}{\partial v_1} = \overline{v_0} + \frac{\overline{v_2}}{v_0} = 5.5$$

$$\overline{v_1} = \overline{v_1}' + \overline{v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_1} + \overline{v_3} \times v_0 = 1.716$$

$$\overline{v_0}' = \overline{v_3}\frac{\partial v_3}{\partial v_0} = \overline{v_3} \times v_1 = 5$$

$$\overline{v_1}' = \overline{v_4}\frac{\partial v_4}{\partial v_1} = \overline{v_4} \times \cos v_1 = -0.284$$

$$\overline{v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_3} = \overline{v_5} \times 1 = 1$$

$$\overline{v_3} = \overline{v_5}\frac{\partial v_5}{\partial v_3} = \overline{v_5} \times 1 = 1$$

$$\overline{v_4} = \overline{v_6}\frac{\partial v_6}{\partial v_4} = \overline{v_6} \times (-1) = -1$$

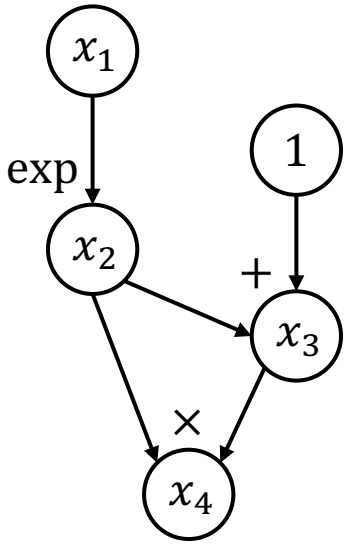
$$\overline{v_5} = \overline{v_6}\frac{\partial v_6}{\partial v_5} = \overline{v_6} \times 1 = 1$$

$$\overline{v_6} = \bar{y} = 1$$

AutoDiff (reverse) Algorithm

⇨

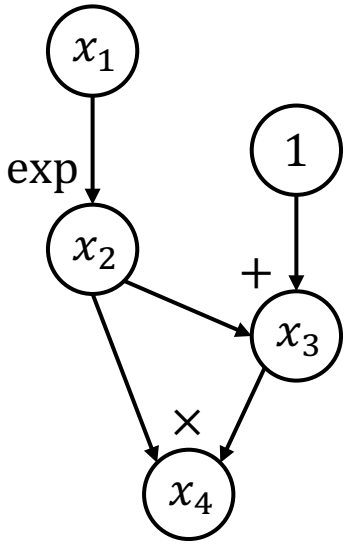
```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```



AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x_4: x̄_4
}
```

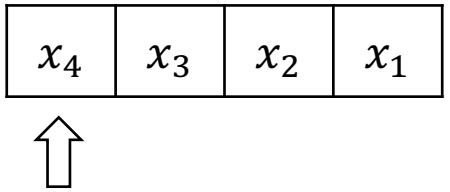


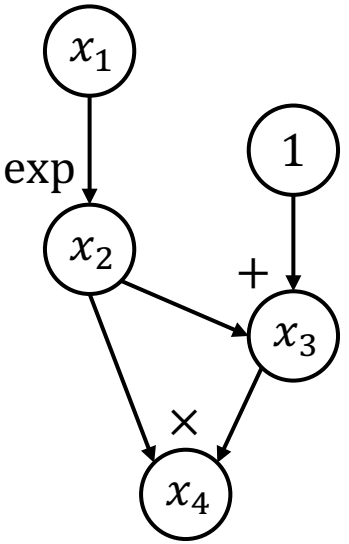
$\overline{x_4}$

AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x_4: x̄_4
}
```



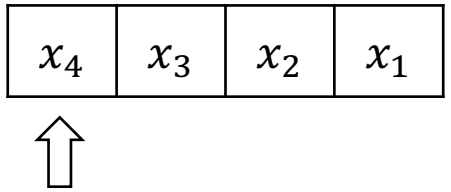


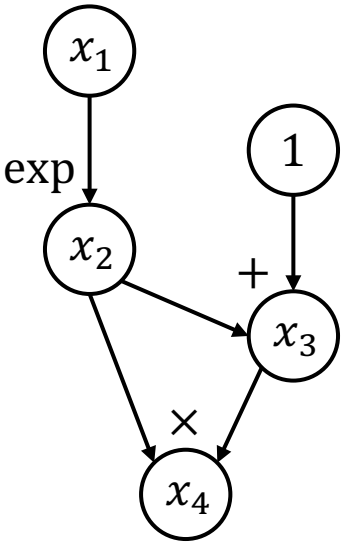
$\overline{x_4}$

AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x̄4
}
```



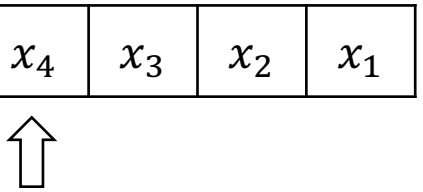


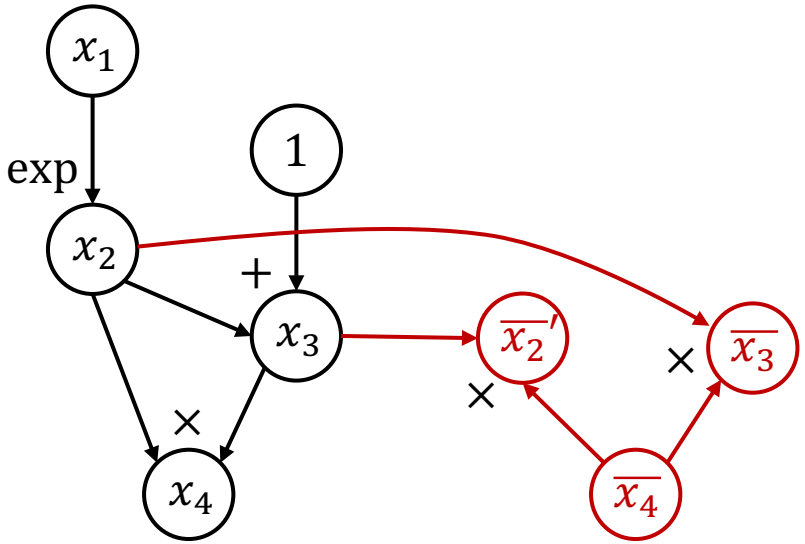
$\overline{x_4}$

AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x̄4
}
```

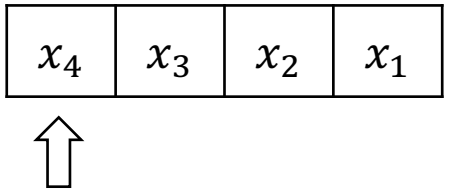


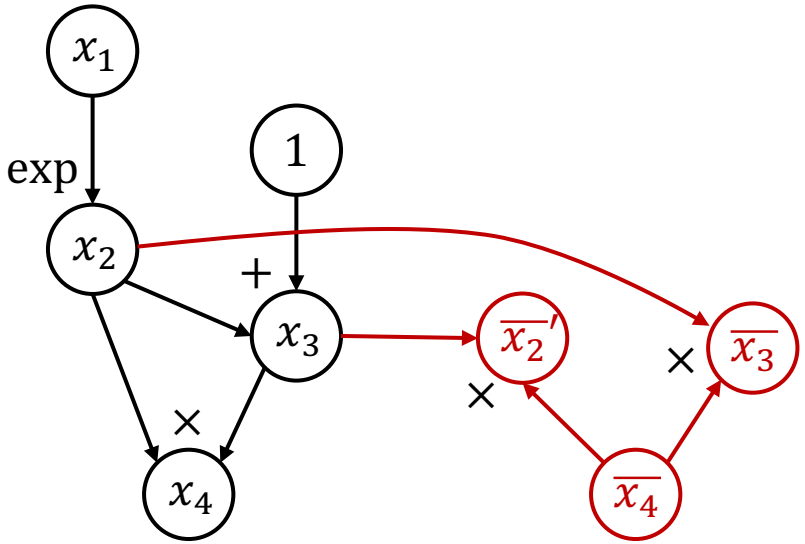


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar'
}
```

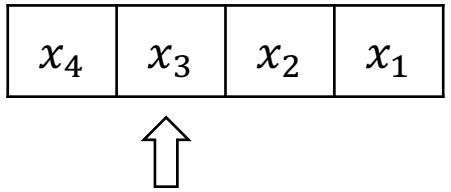


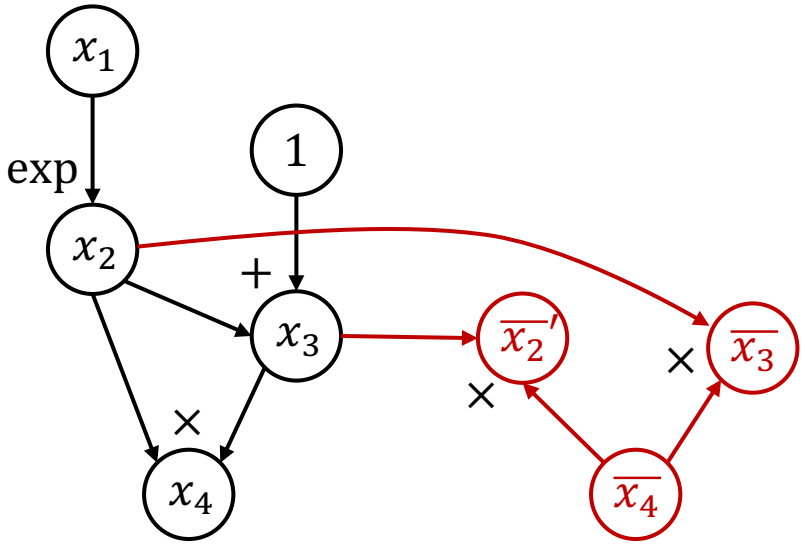


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar'
}
```

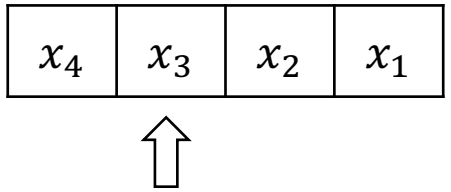


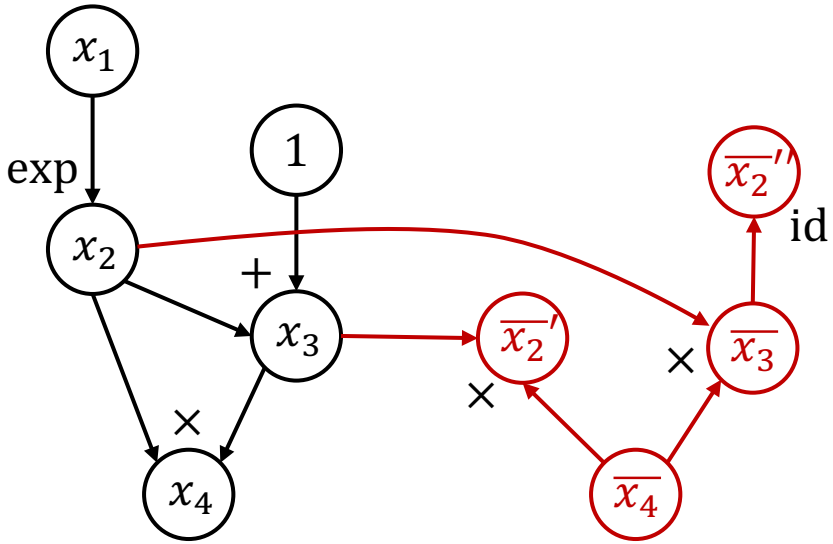


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar'
}
```

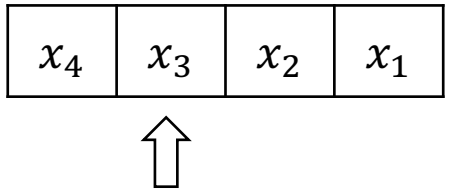


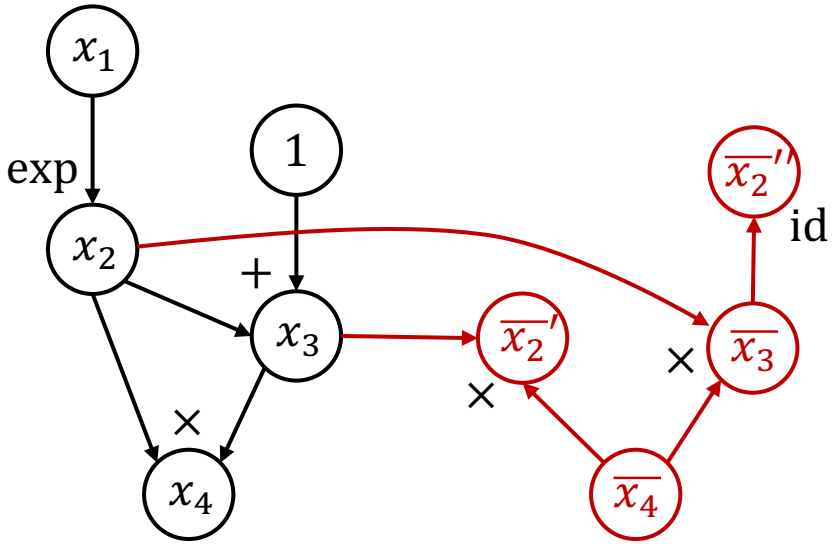


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar', x2_bar''
}
```

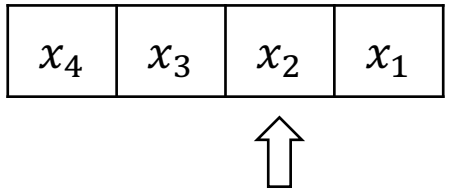


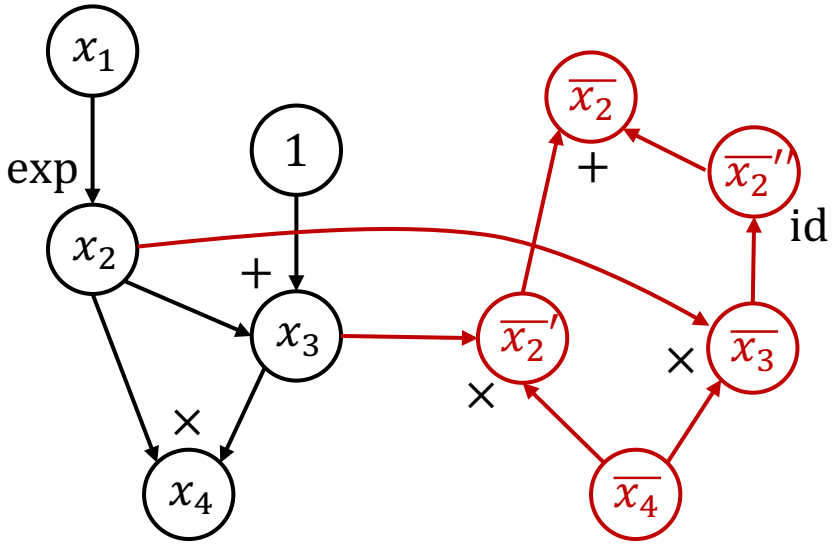


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar', x2_bar''
}
```

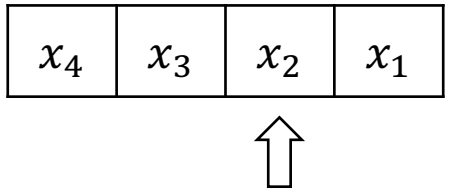


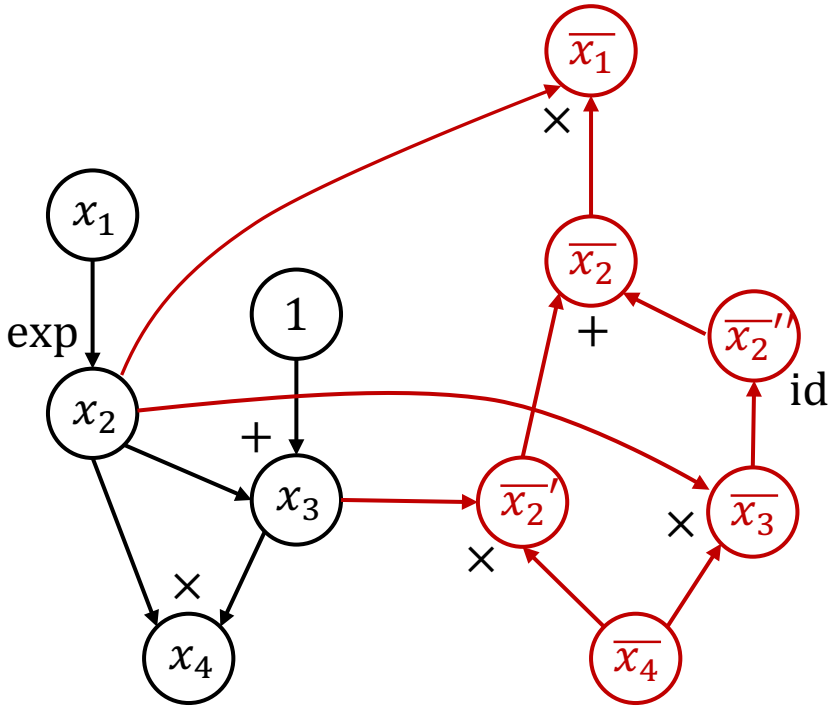


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar', x2_bar''
}
```

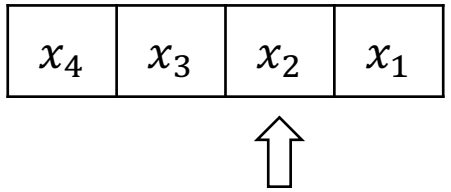


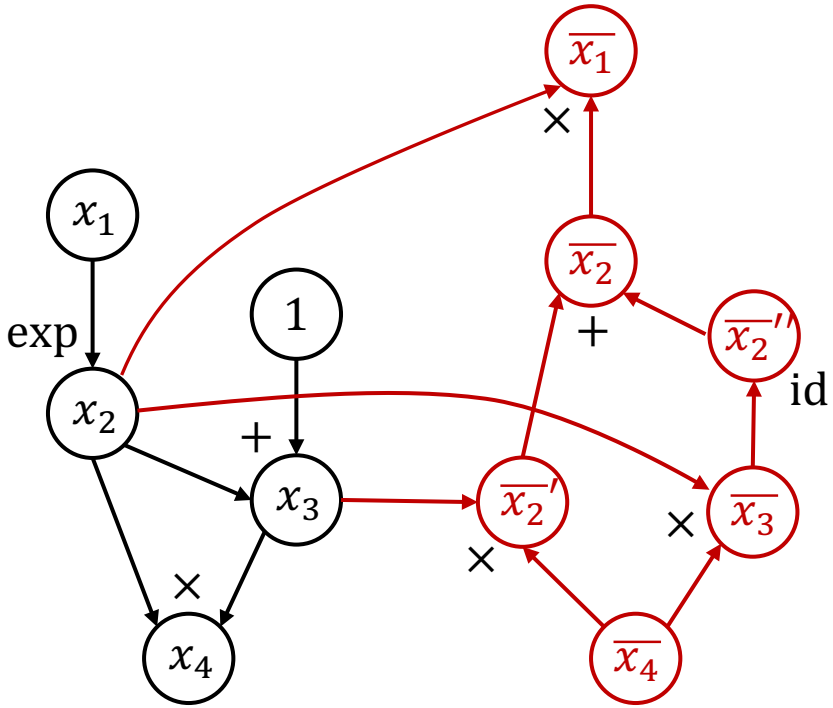


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar', x2_bar''
  x1: x1_bar
}
```



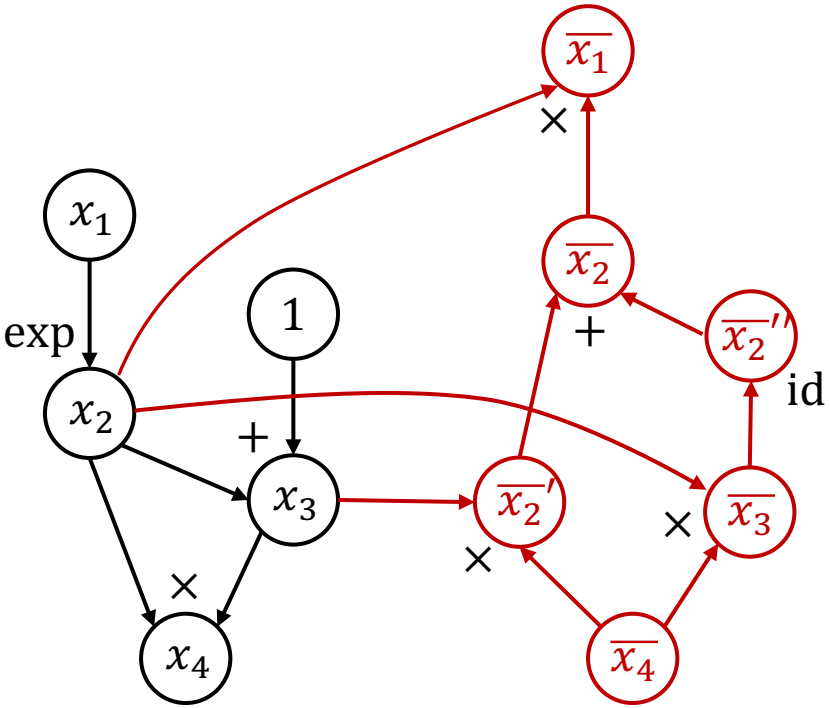


AutoDiff (reverse) Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given
node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```

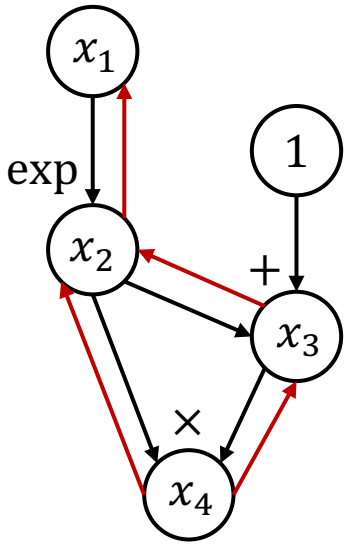
```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar', x2_bar''
  x1: x1_bar
}
```

x_4	x_3	x_2	x_1

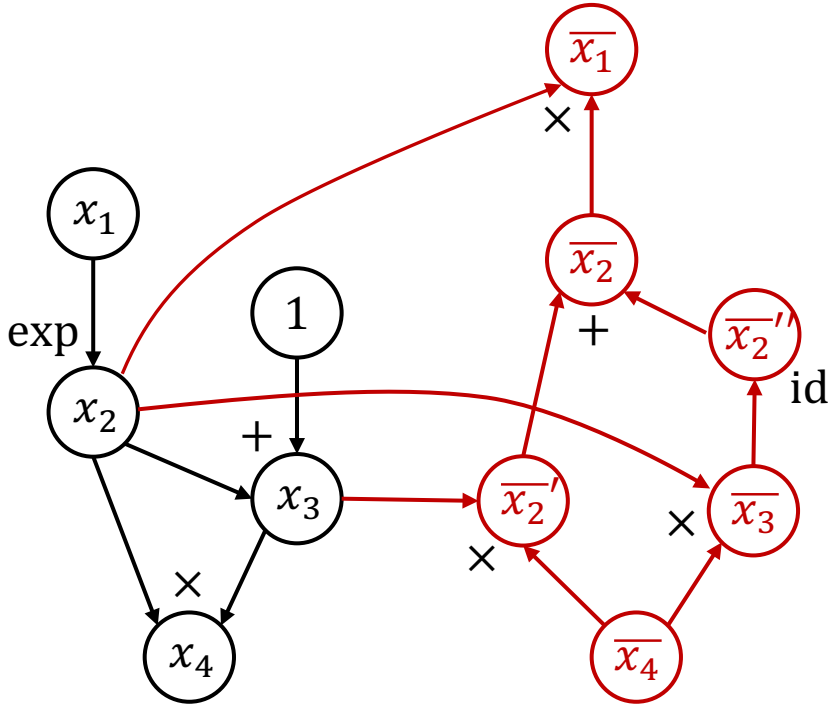


Backpropagation vs AutoDiff (reverse)

Backpropagation



AutoDiff (reverse)



Backpropagation vs AutoDiff (reverse)

- We can take derivative of derivative nodes in autodiff, while it's much harder to do so in backprop.
- In autodiff, there's only a forward pass (vs. forward-backward in backprop). So it's easier to apply graph and schedule optimization to a single graph.
- In backprop, all intermediate results might be used in the future, so we need to keep these values in the memory. On the other hand, in autodiff, we already know the dependencies of the backward graph, so we can have better memory optimization.

More about AutoDiff

- In neural network, people use more high level operator (composed operator), such as convolution, softmax, etc.

References

- Automatic differentiation in machine learning: a survey https://arxiv.org/abs/1502.05767
- CS231n backpropagation: http://cs231n.github.io/optimization-2/