15-884: Machine Learning Systems

Automatic Differentiation

Instructor: Tianqi Chen

Carnegie Mellon University
School of Computer Science

catalyst

Happy Lunar New Year

Good Job on the First Batch of Paper Reviews!

Some quick notes:

- Three short paragraphs are sufficient(key idea first paper, key idea second paper, comparison and future works)
- Type them out in any form(latex, google doc) and save as PDF
- Think them as quick notes you usually write for your own when revisiting the paper.

Example Review

- TensorFlow: The key idea in Tensorflow is the ability to use the computational graph as abstraction with state support. The optimizations can scale to multiple machines with automatic device placement.
- JANUS: JANUS combines the symbolic and static execution. The system speculatively generates a graph with assertions(fast path). When an assertion fails, the system fallbacks to imperative executions

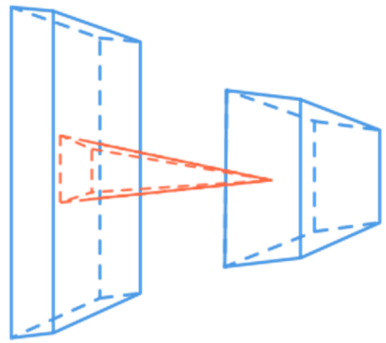
- Discussions: Weakness of Tensorflow, declarative(symbolic) form of the program makes it harder to debug. JANUS generates a fast path of symbolic programs via speculative execution. However, it would require a python interpreter to be available. We could possible use speculative execution techniques to speedup execution of JIT compilation of torchscript.

Model Training Overview

layer1 extractor

layer2 extractor

predictor



$$\hat{y}_i = \frac{1}{1 + \exp\left(-w^T x_i\right)}$$

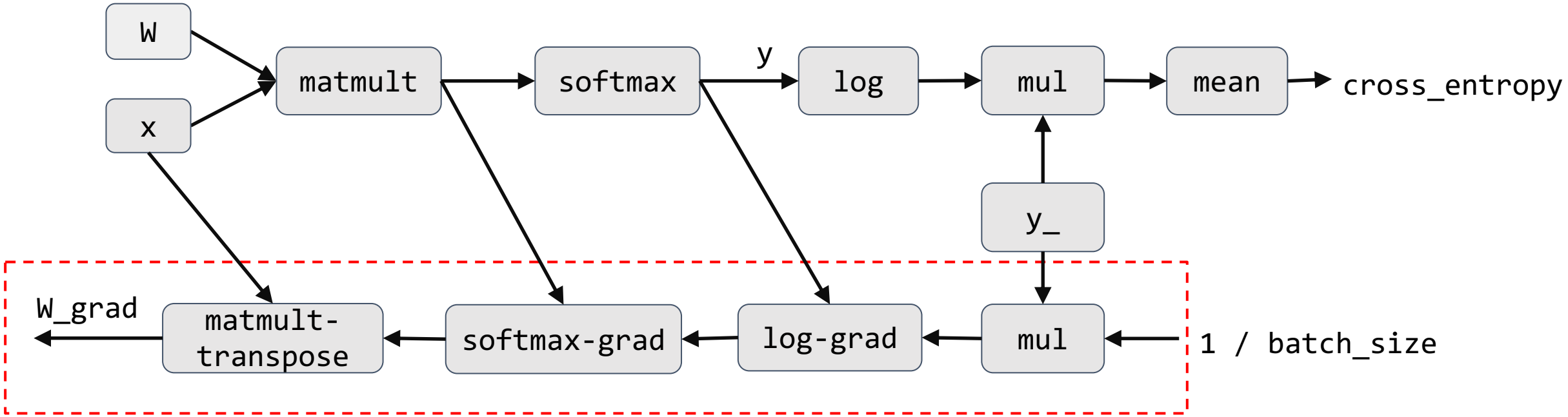
Objective

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

Training

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

Automatic Differentiation



Numerical Differentiation: Do it by Definition

Approximate the gradient as

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

Reduce error by using center difference

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}$$

Suffer from rounding error

A powerful tool to numerically **check** the gradient implementation (write unit test cases)

Numerical Gradient Checking

Invariance

Gradient value to be checked

Negligible small value

$$\frac{f(\mathbf{x}+h\delta)-f(\mathbf{x}-h\delta)}{2h} = \delta^T \nabla_{\mathbf{x}} f(\mathbf{x}) + O(h^2)$$

Choose small h

Randomly sample δ from unit ball, or pick $\mathbf{e}_i$

Symbolic Differentiation

- Input formula is a symbolic expression tree (computation graph).
- Implement 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.

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

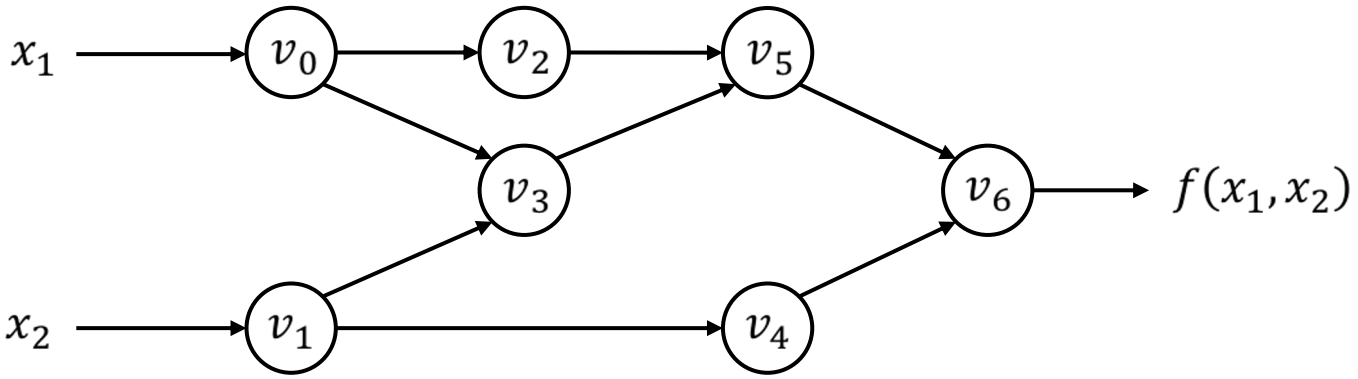
Automatic Differentiation (AutoDiff)

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

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

Forward Mode AutoDiff

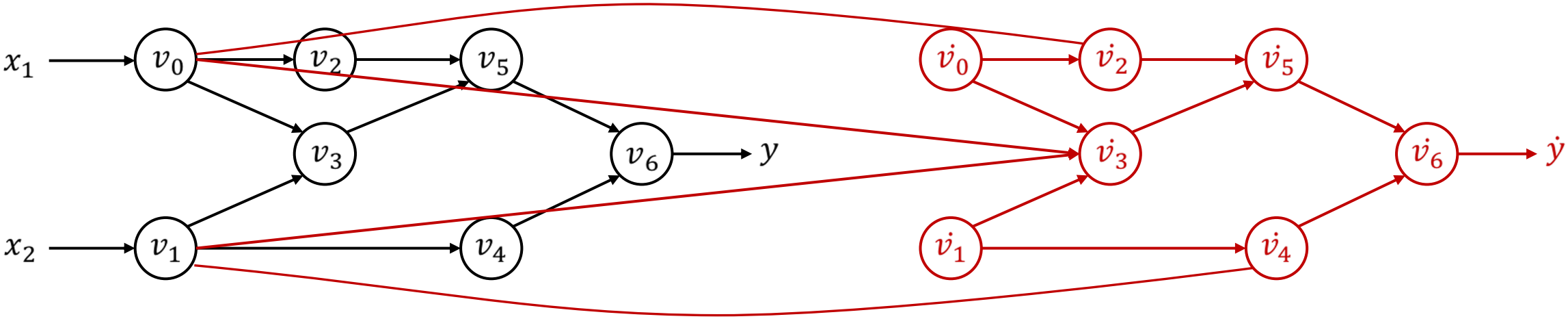
$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$



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

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

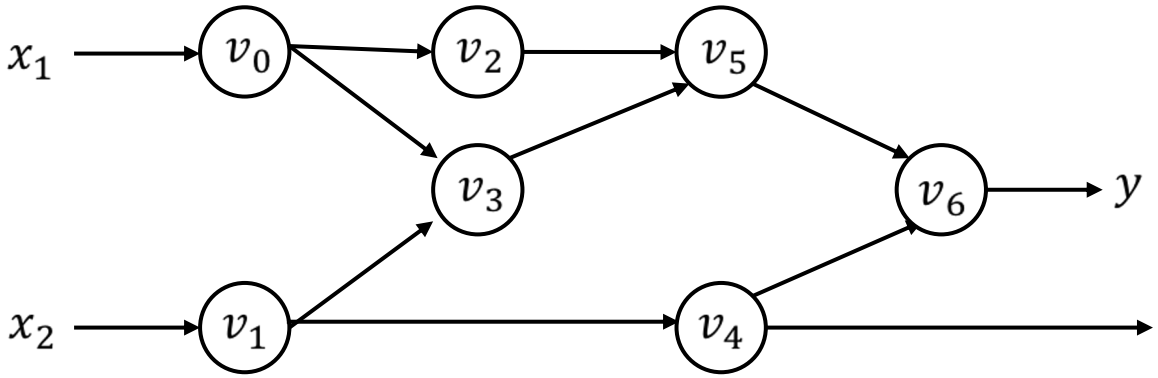
Now we have $\frac{\partial y}{\partial x_1}$.

Problem?

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

- Needs n forward passes to get gradient wrt to each input
- Deep learning models have large number of inputs(weights are also considered input), and one output(loss)
- Reverse mode AD come to the rescue

Reverse Mode AutoDiff



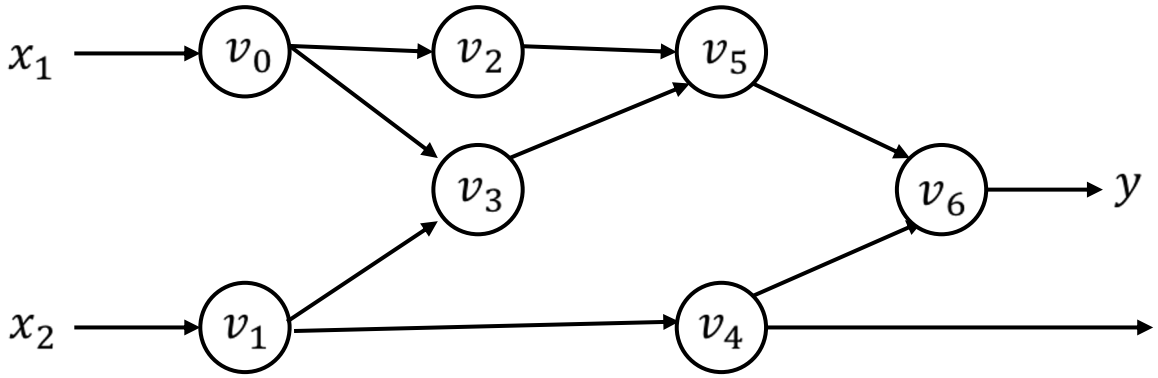
Forward Evaluation Trace

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

For each node v_i , introduce an **adjoint node** that corresponds to gradient of output wrt to this node

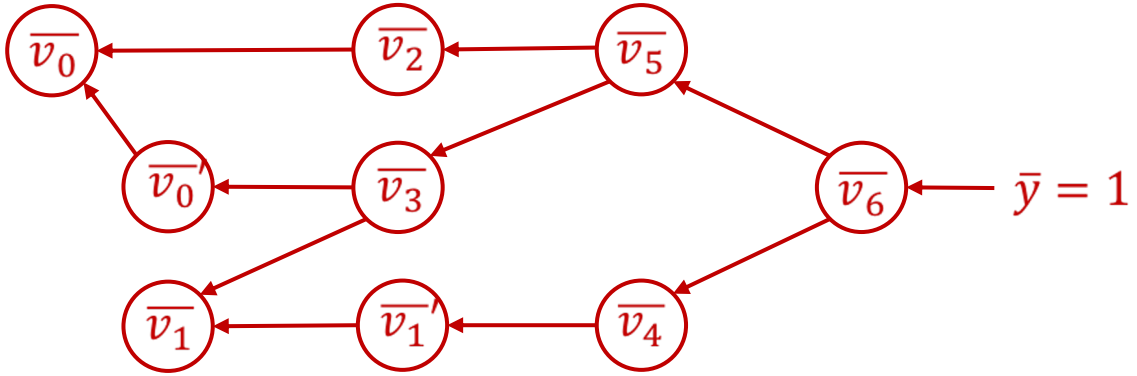
$$\bar{v_i} = \frac{\sum_j \partial y_j}{\partial v_i}$$

Reverse AD Example



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

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

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

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

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

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

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

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

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

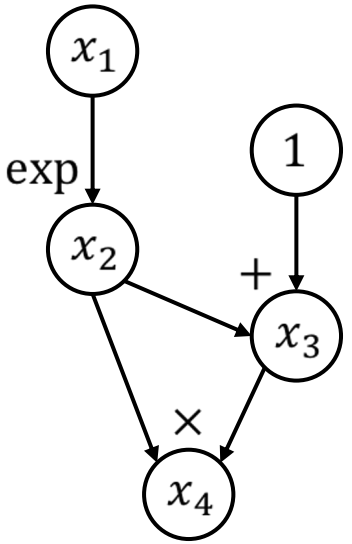
$$\overline{v_6} = \bar{y} = 1$$

Connect back to Computational Graph

Reverse Mode AutoDiff

⇨

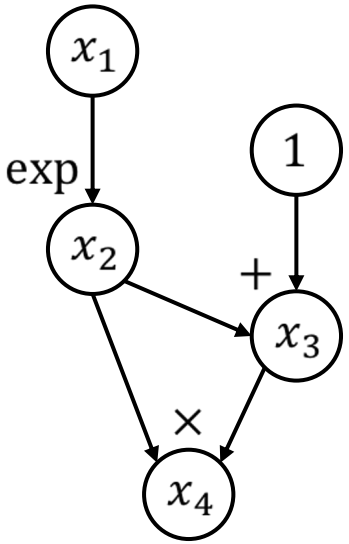
```
def gradient(out):
  node_to_grad[out] = 1
  for node in reverse_topo_order(out):
    grad ← sum partial adjoints from output edges
    input_grads ← calc partial adjoints for inputs given node.op and grad
    add input_grads to node_to_grad
  return node_to_grad
```



Reverse Mode AutoDiff

```
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
}
```

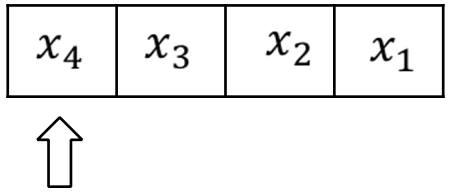


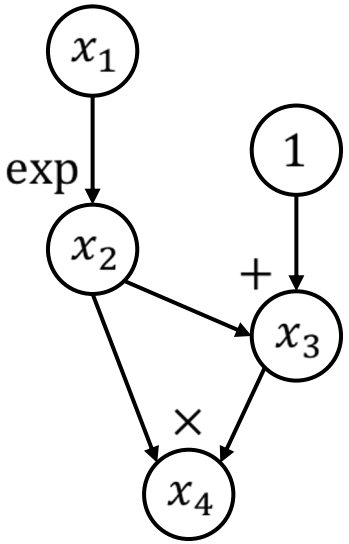
$\overline{x_4}$

Reverse Mode AutoDiff

```
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
}
```



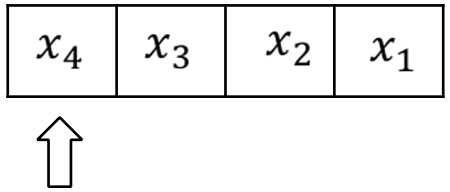


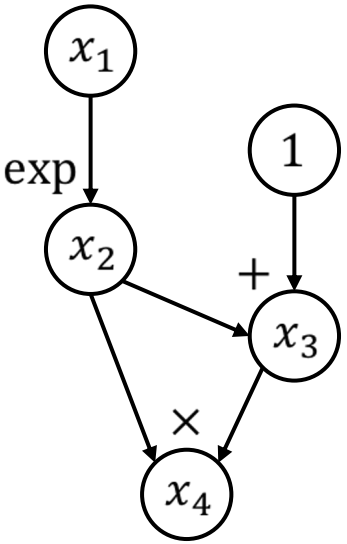
$\overline{x_4}$

Reverse Mode AutoDiff

```
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
}
```



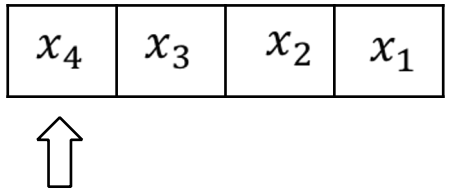


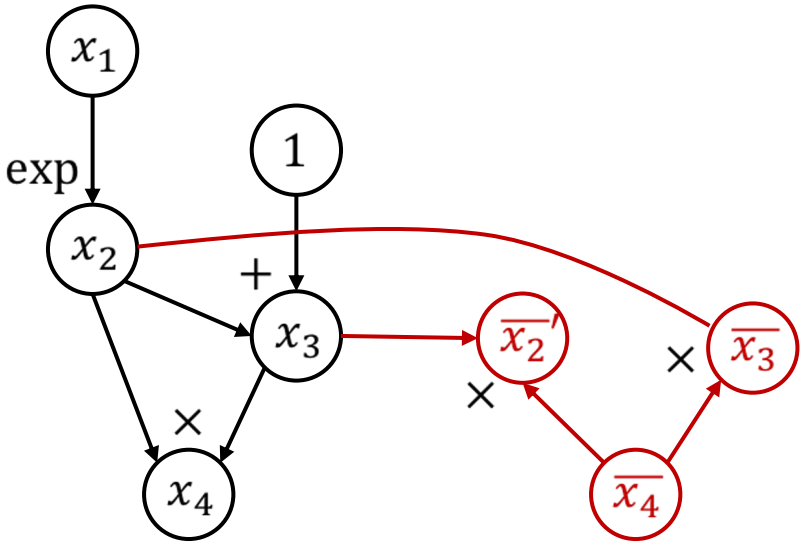
$\overline{x_4}$

Reverse Mode AutoDiff

```
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
}
```

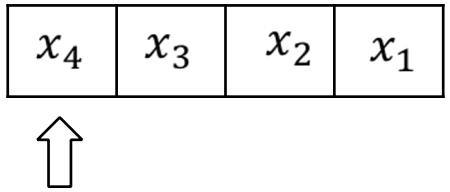


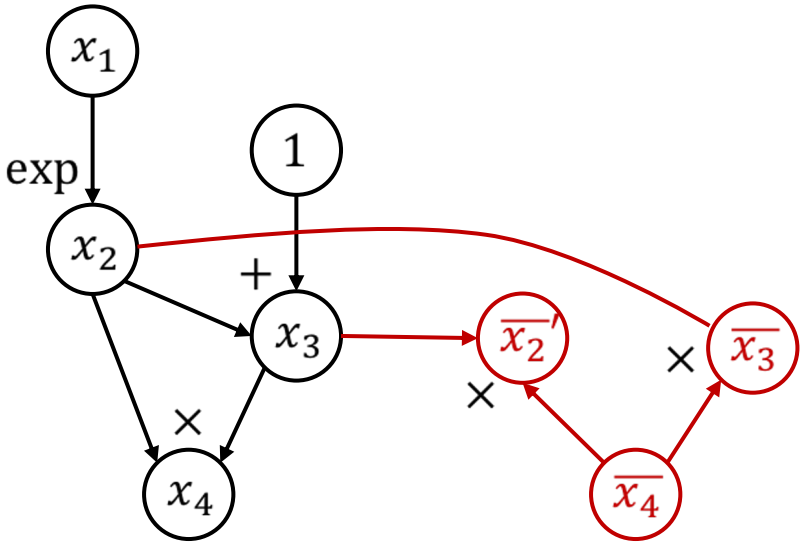


Reverse Mode AutoDiff

```
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'
}
```

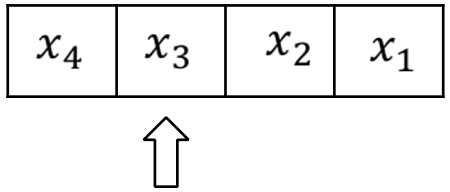


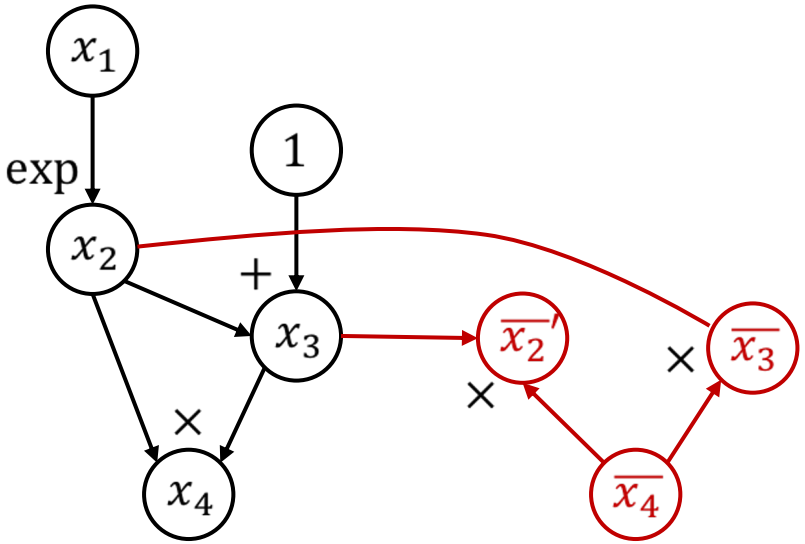


Reverse Mode AutoDiff

```
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'
}
```

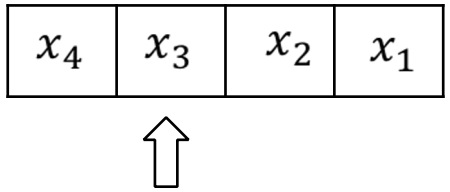


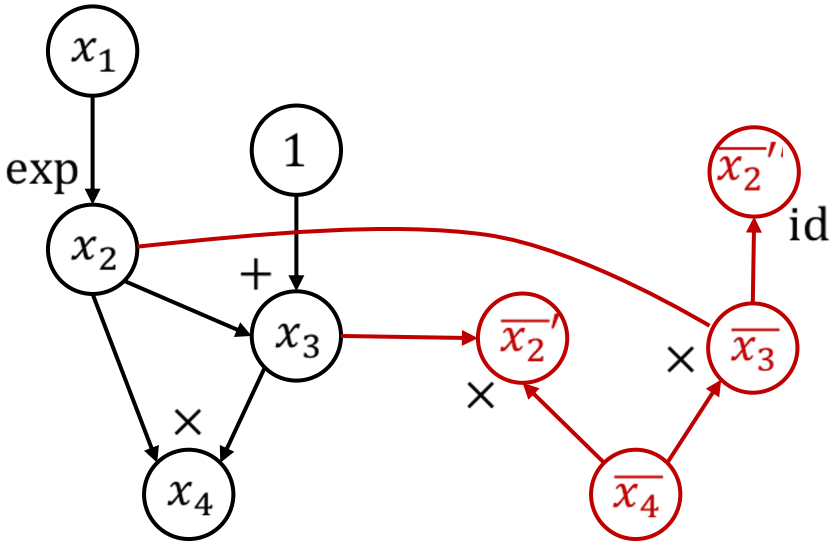


Reverse Mode AutoDiff

```
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'
}
```



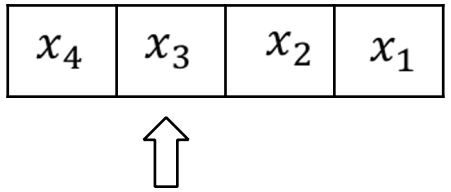


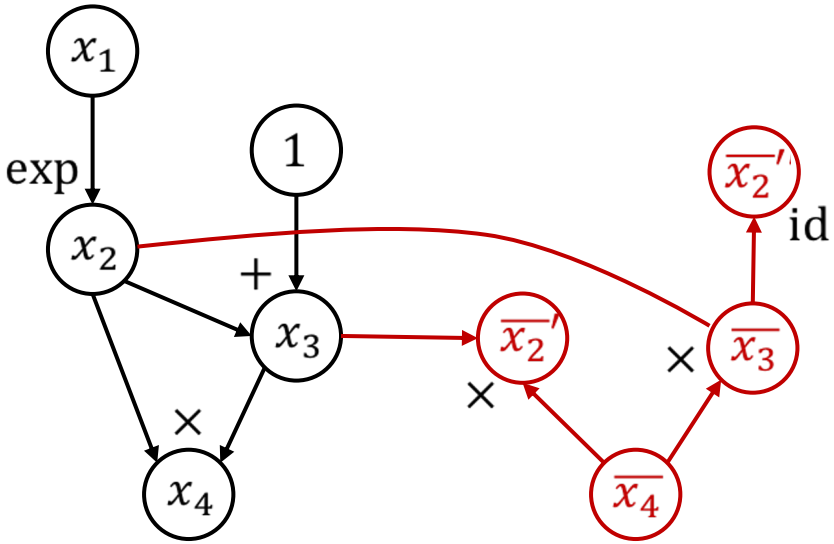
Reverse Mode AutoDiff

```
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''
}
```

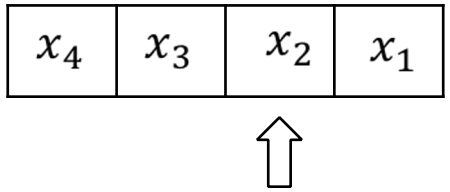


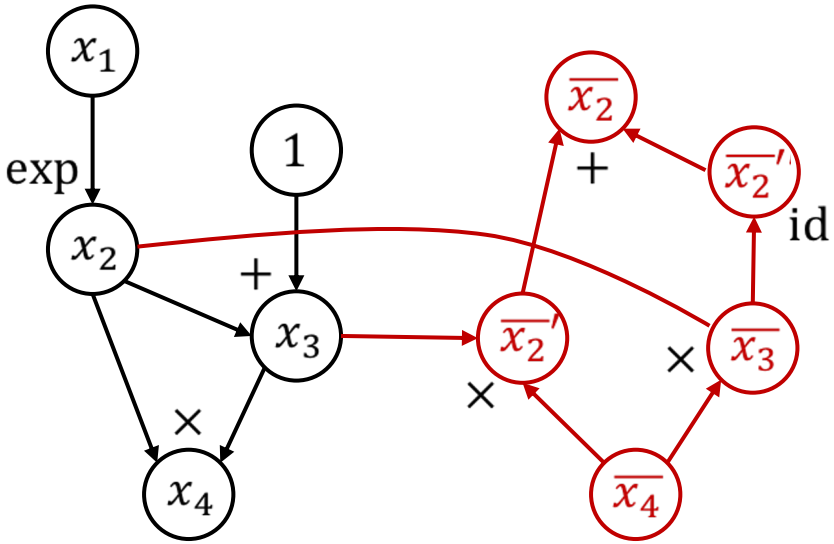


Reverse Mode AutoDiff

```
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''
}
```

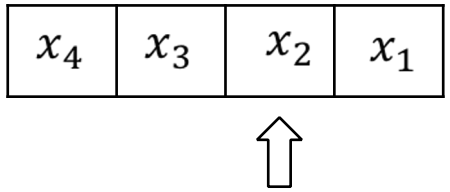


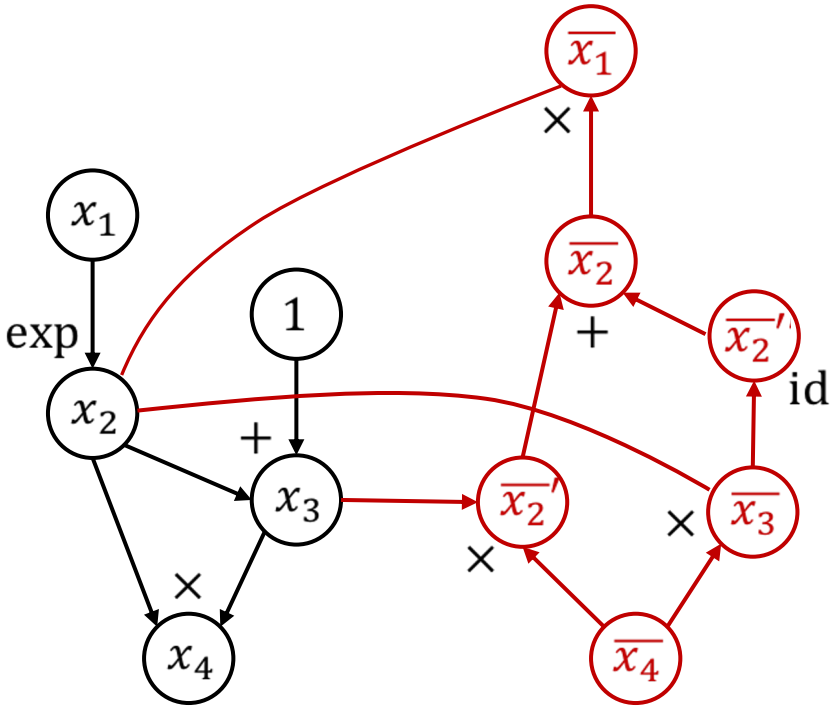


Reverse Mode AutoDiff

```
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''
}
```



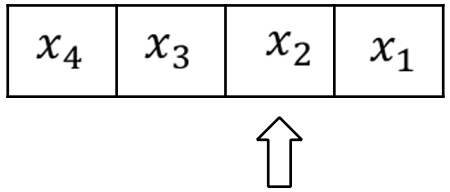


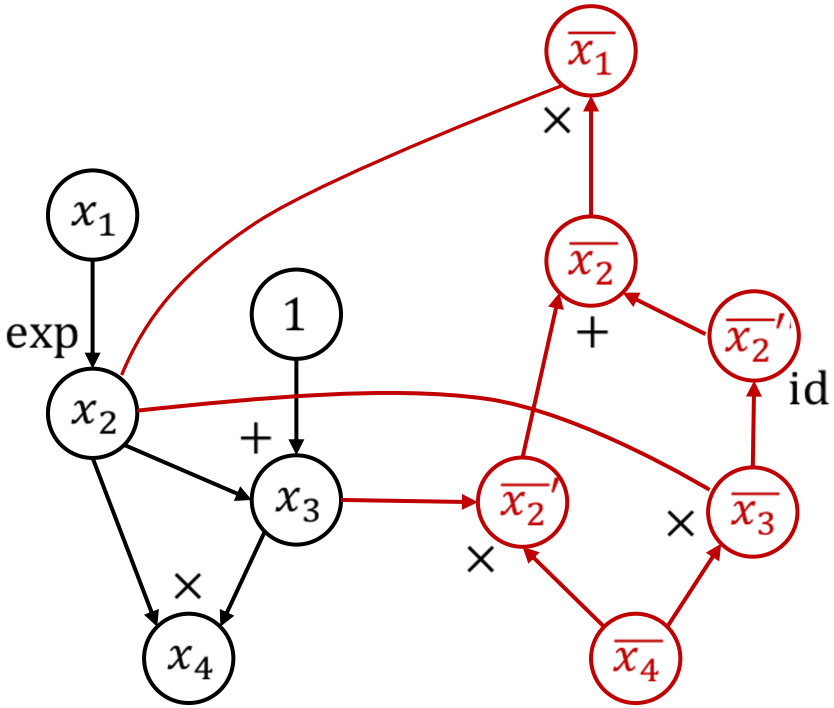
Reverse Mode AutoDiff

```
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
}
```

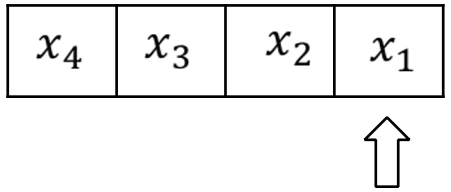


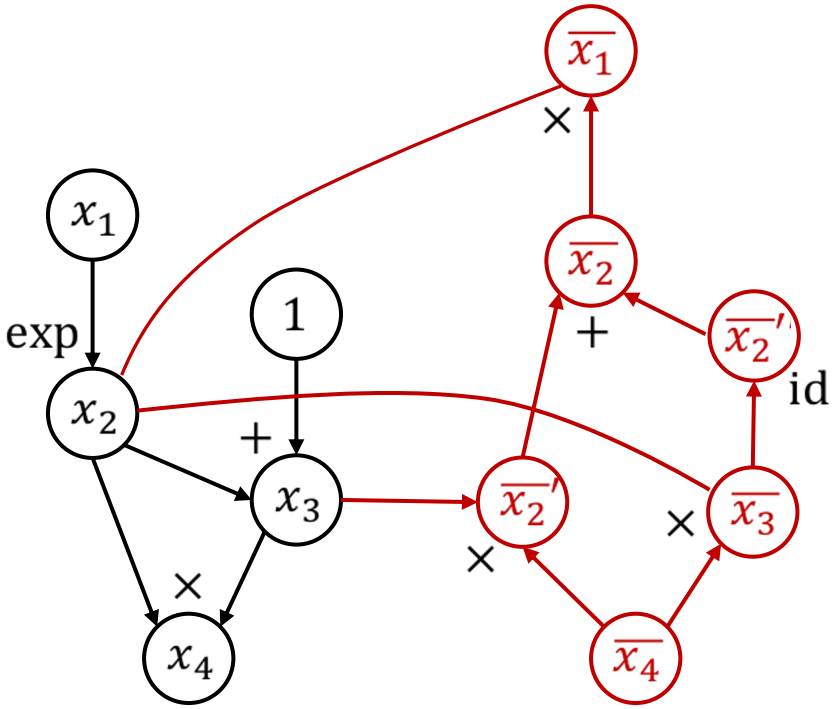


Reverse Mode AutoDiff

```
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
```

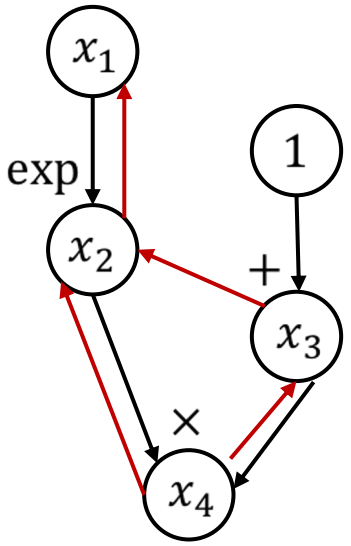
```
node_to_grad: {
  x4: x4_bar
  x3: x3_bar
  x2: x2_bar', x2_bar''
  x1: x1_bar
}
```



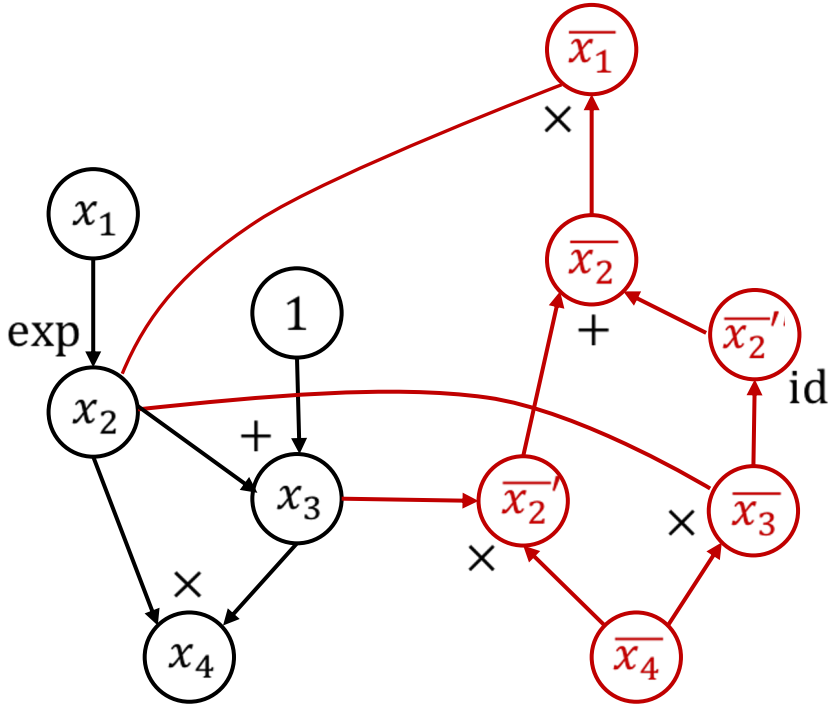


Backpropagation and AutoDiff (reverse)

Backpropagation



AutoDiff (reverse)



Discussion Items

What are the difference between backprop and reverse mode AD?

How to support second(higher) order gradients?

Backpropagation vs Reverse Mode AutoDiff

- 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.

AutoDiff on Composed Operators

- In neural network, people use more high-level operator (composed operator), such as batch-norm, softmax, etc.
- Solution0: Implement adjoint operator for those ops
- Solution1: Legalize the operator into smaller ops, then run AD

Advanced AD Topics

- Probabilistic decisions (policy gradients)
- Differentiate over sub-function calls
- Handling of data structure, states

GPU Memory is the Bottleneck in Training Big Models

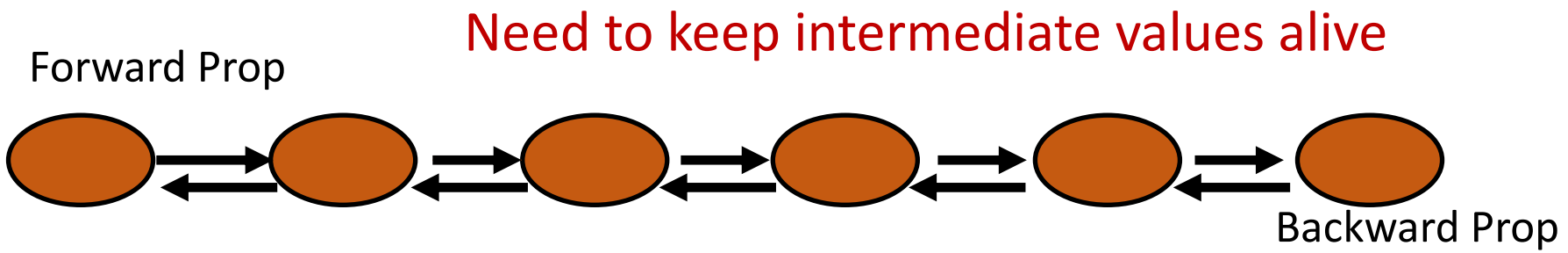
The maximum size of the model we can try is bounded by total RAM available of the GPU.

Extremely important for some accelerators with fast but small on-chip memory

Usually these models are also the state of the art.

Discussion: How can we train bigger models with limited GPU memory

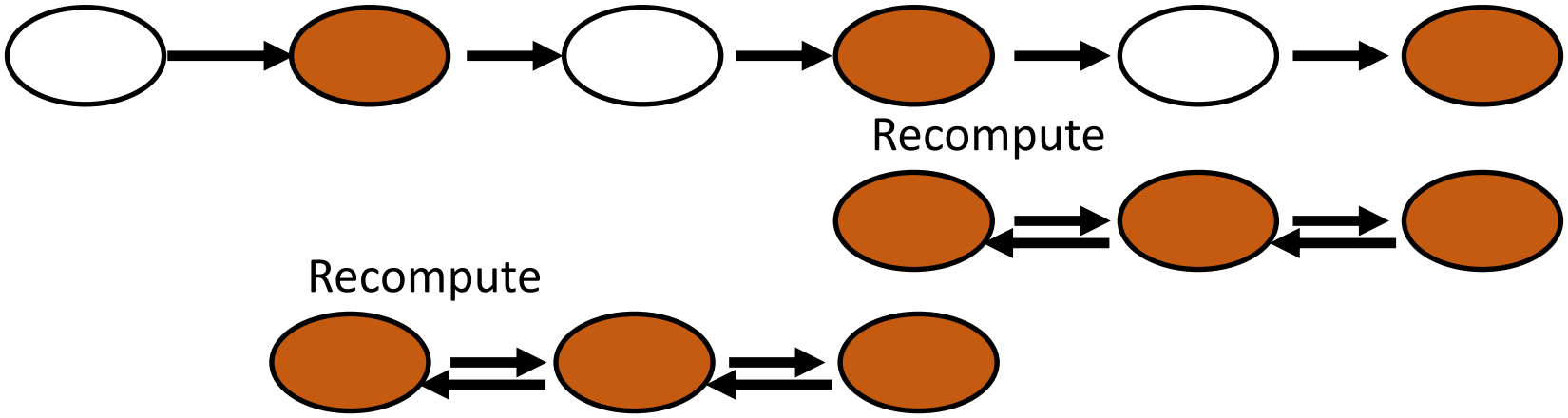
Cost of Training Deep Neural Network



Memory Cost = O(N)

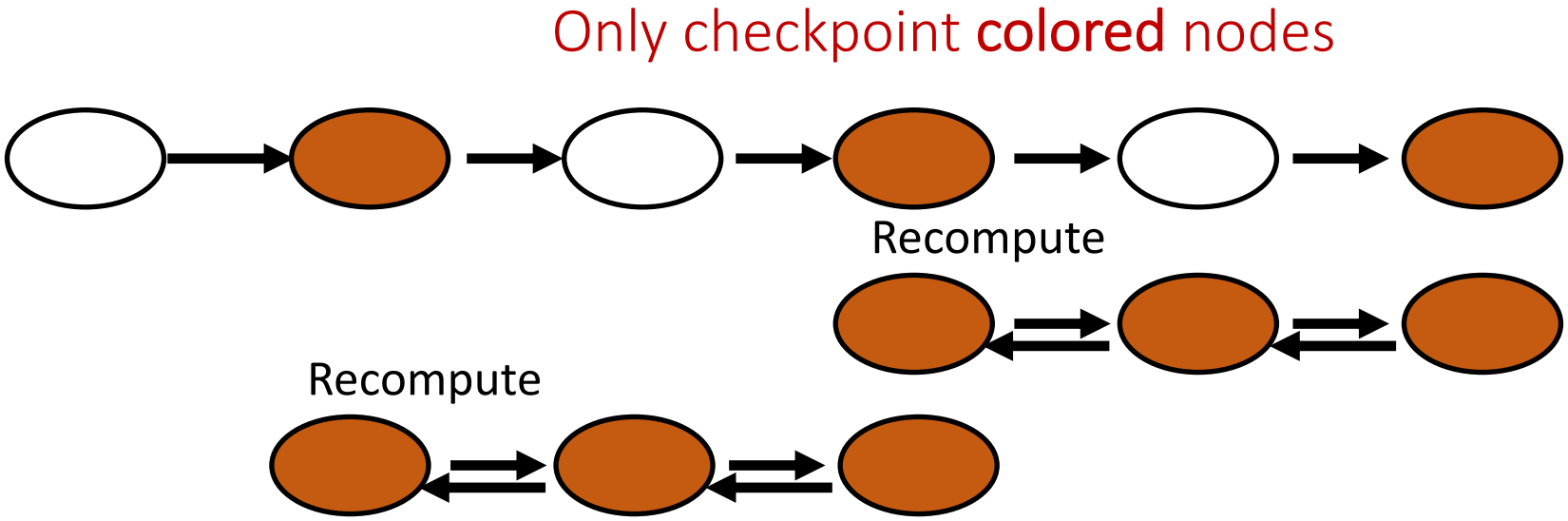
Gradient Checkpointing

Only store **colored** nodes



Memory Cost = Recompute Cost + Checkpoint Cost

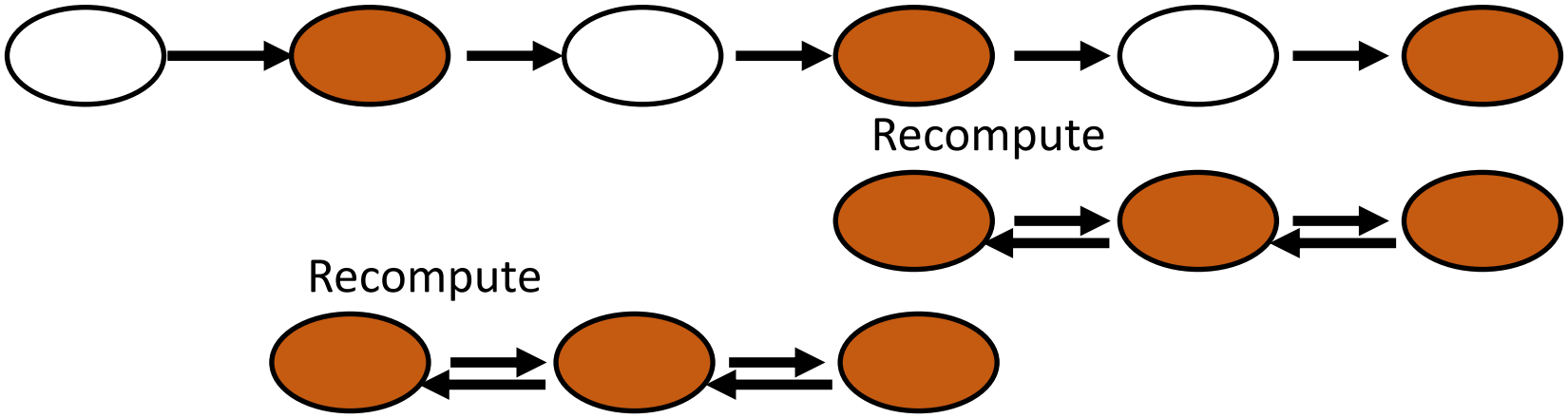
Gradient Checkpointing



Memory Cost = Recompute Cost + Checkpoint Cost

Gradient Checkpointing: Sublinear Memory Cost

Only checkpoint **colored** nodes



If we check point every K steps

on a N layer network

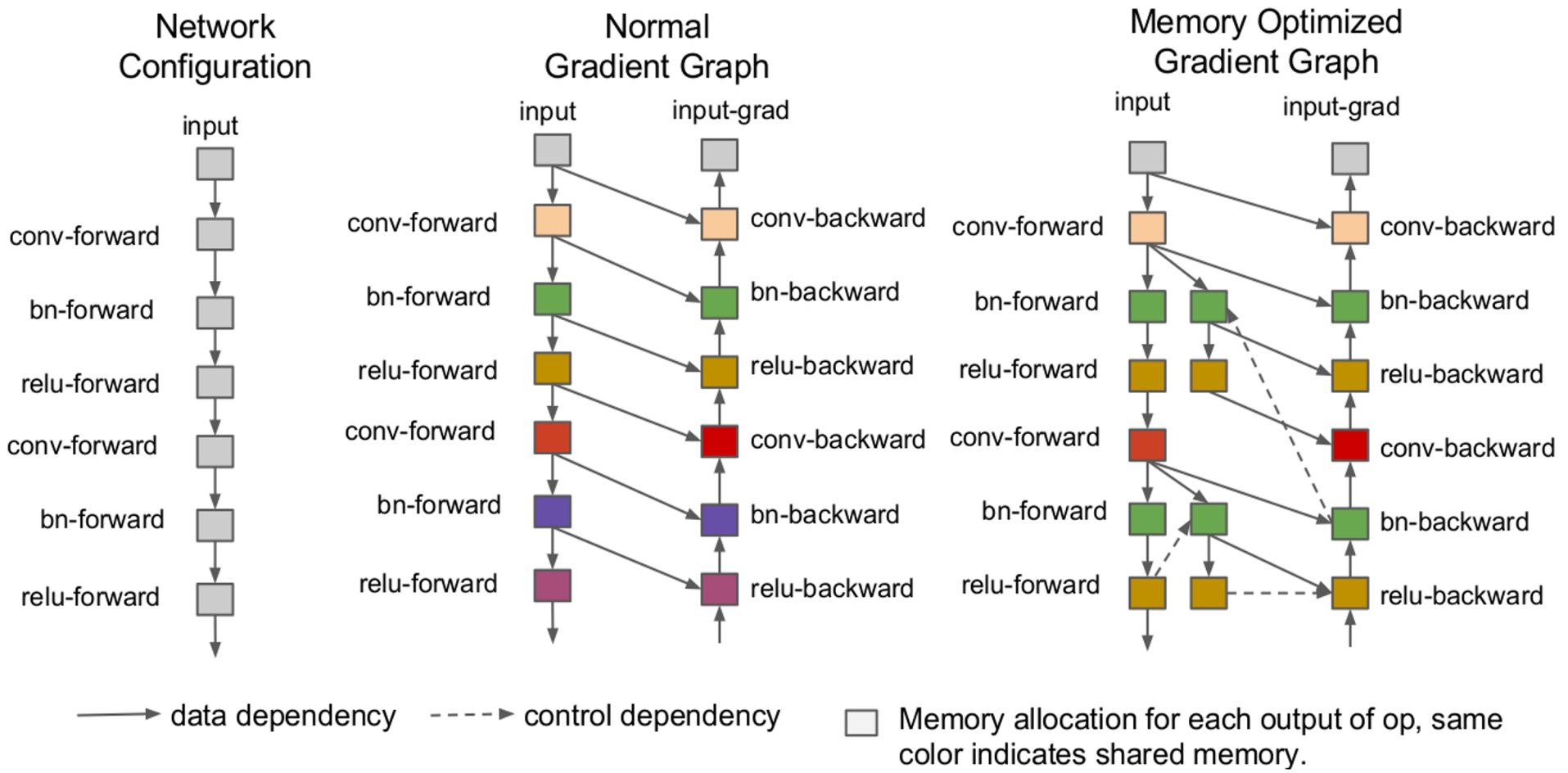
Memory Cost = O(K) + O(N/K)

Recompute Cost

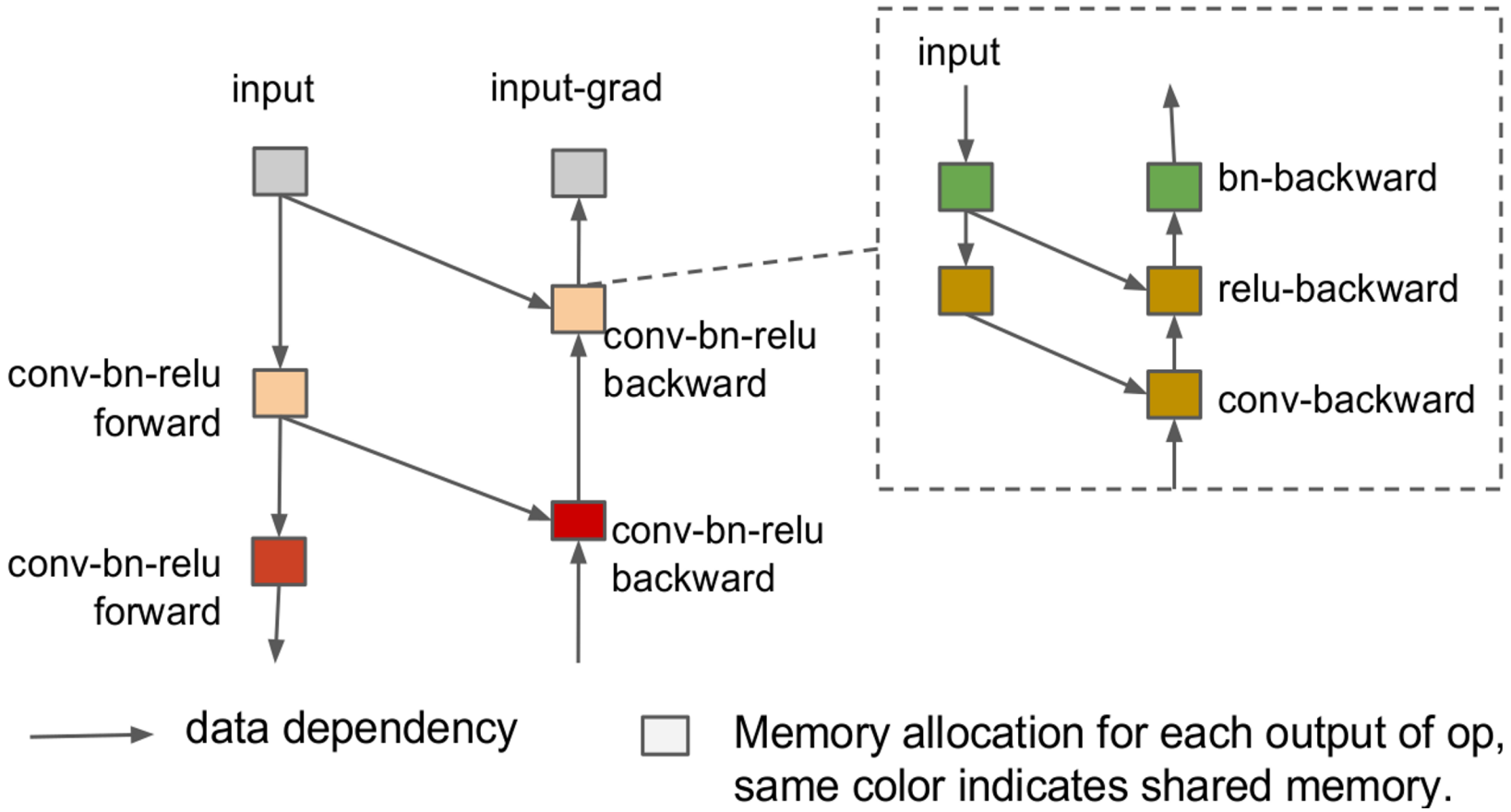
Checkpoint Cost

Pick K = $\sqrt{N}$

A Computational Graph View of Checkpointing



An Alternative View: Recursion



What are the limitations of the sqrt(n) checkpointing approach?

Alternative solutions to save memory?

Summary

- Reverse mode AutoDiff is the main algorithm for gradient computation
- Gradient checkpointing is extremely important, essential for big models like GPT-3
- Checkout more advanced topics in readings