CS 4803 / 7643: Deep Learning

Topics:

- Analytical Gradients
- Automatic Differentiation
  - Computational Graphs
  - Forward mode vs Reverse mode AD

Dhruv Batra

Georgia Tech

Administrativia

- HW1 Reminder
  - Due: 09/09, 11:59pm

Recap from last time

$\left[\min_{\vec{w}} L(\vec{w})\right]$

Strategy: **Follow the slope**

$\boxed{\text{Gradient Descent}}$

$L(\vec{w})$

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Gradient Descent

$$\min_{\vec{w}} L(\vec{w})$$

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

backprop

$\vec{w}^{(0)}$ = Initialise

for $t = 0, 1, 2, \ldots,$ exhausted

$$w^{(t+1)} = w^{(t)} - \boxed{\eta}\, \nabla_{\vec{w}} L(\vec{w})$$

0.001

step size

learning-rate

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

How do we compute gradients?

- Analytic or “Manual” Differentiation
- Symbolic Differentiation X
- Numerical Differentiation
- Automatic Differentiation
  - Forward mode AD
  - Reverse mode AD
    - aka “backprop”

$l_1 = x$

$l_{n+1} = 4l_n(1 - l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

Manual Differentiation

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

Coding

```
f(x):
  v = x
  for i = 1 to 3
    v = 4*v*(1 - v)
  return v
```

or, in closed-form,

```
f(x):
 return 64*x*(1-x)*((1-2*x)^2)
   *(1-8*x+8*x*x)^2
```

Coding

Symbolic Differentiation of the Closed-form

```
f'(x):
 return 128*x*(1 - x)*(-8 + 16*x)
   *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
   + 64*(1 - x)*((1 - 2*x)^2)*((1
   - 8*x + 8*x*x)^2) - (64*x*(1 -
   2*x)^2)*(1 - 8*x + 8*x*x)^2 -
   256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
   + 8*x*x)^2
```

f'(x₀) $= f'(x_0)$
Exact

Automatic Differentiation

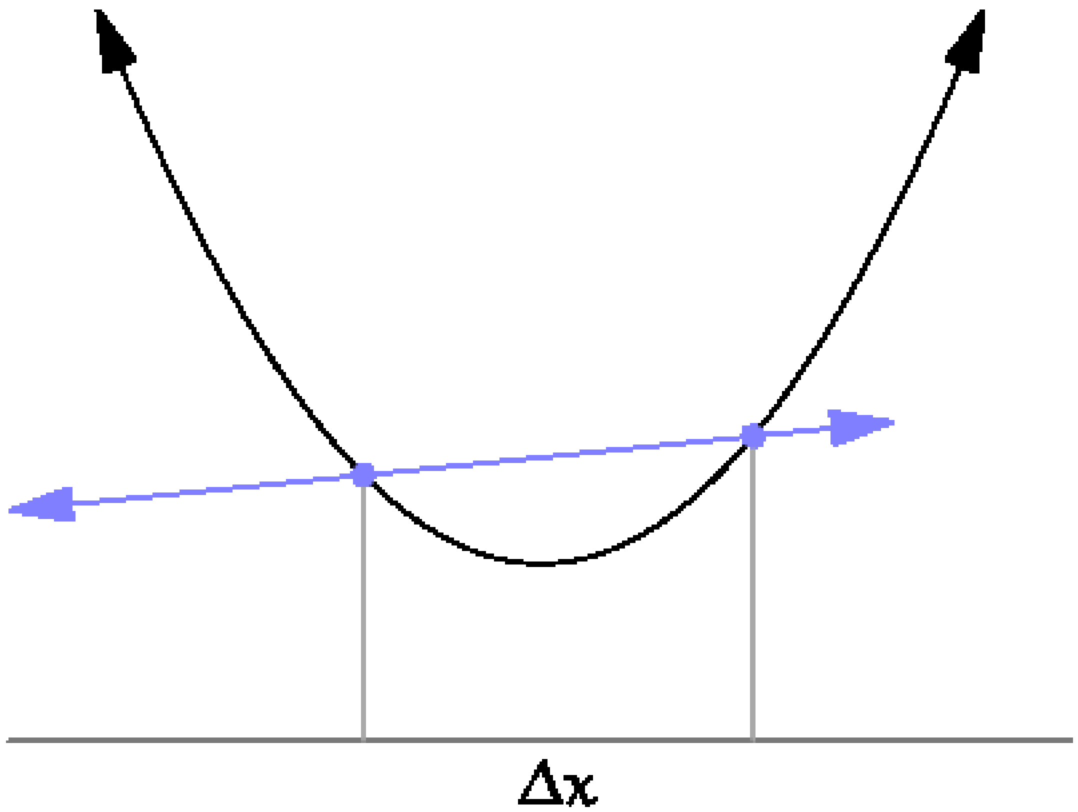
```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

f'(x₀) $= f'(x_0)$
Exact

Numerical Differentiation

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

f'(x₀) $\approx f'(x_0)$
Approximate


Δx

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?,
?,…]

Numerical vs Analytic Gradients

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Numerical gradient: slow :(, approximate :(, easy to write :)
Analytic gradient: fast :), exact :), error-prone :(

In practice: Derive analytic gradient, check your implementation with numerical gradient.
This is called a **gradient check.**

Plan for Today

- Analytical Gradients
- Automatic Differentiation
  - Computational Graphs
  - Forward mode vs Reverse mode AD

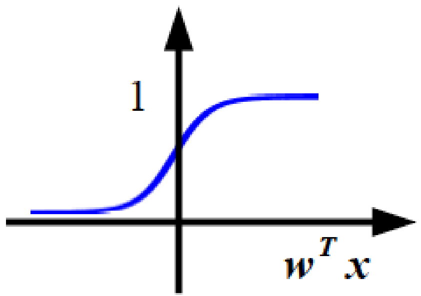
Example: Logistic Regression

Input: $\boxed{\boldsymbol{x} \in R^D}$

Binary label: $\boxed{y \in \{-1, +1\}}$

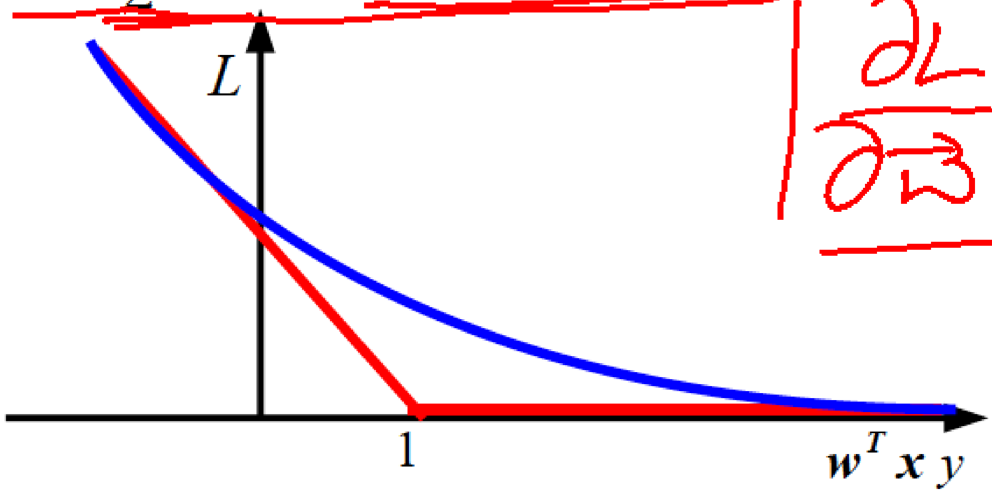
Parameters: $\boxed{\boxed{\boldsymbol{w}} \in R^D}$ $\vec{w}_1$ $\vec{w}_2$

Output prediction: $p(y=1|\boldsymbol{x}) = \dfrac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}}$



Loss: $\boxed{L = \frac{1}{2}\|\boldsymbol{w}\|^2 - \lambda \log(p(y|\boldsymbol{x}))}$

$$\boxed{\frac{\partial L}{\partial \vec{w}}}$$



Log Loss

Vector/Matrix Derivatives Notation

$x, y \in \mathbb{R}$

$\vec{x} \in \mathbb{R}^d \quad \vec{y} \in \mathbb{R}^c$

$X, Y \in \mathbb{R}^{m \times n}$

$$\begin{array}{c|ccc}
y \backslash x & S & V & M \\
\hline
S & \frac{\partial y}{\partial x} & \frac{\partial y}{\partial \vec{x}} & \frac{\partial y}{\partial X} \\
V & \frac{\partial \vec{y}}{\partial x} & \frac{\partial \vec{y}}{\partial \vec{x}} & \text{tensors} \\
M & \frac{\partial Y}{\partial x} & &
\end{array}$$

$$\frac{\partial \vec{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \vdots \\ \frac{\partial y_c}{\partial x} \end{bmatrix}_{c \times 1}$$

num = dim 1 → rows

$$\frac{\partial L}{\partial \vec{w}} \quad \boxed{\frac{\partial y}{\partial \vec{x}}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_d} \end{bmatrix}$$

den = dim 2 → col

Vector/Matrix Derivatives Notation

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{array}{cc} & j \\ i & \begin{bmatrix} \frac{\partial y_i}{\partial x_1} & \cdots & \frac{\partial y_i}{\partial x_j} & \cdots & \frac{\partial y_i}{\partial x_d} \\ & & \frac{\partial y_c}{\partial x_j} & & \end{bmatrix}_{c \times d} \end{array}$$

$$\underbrace{\hspace{6em}}_{\text{Jacobian}}$$

Vector Derivative Example

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x \\ x^2 \end{bmatrix} \qquad \frac{\partial y}{\partial x} = \begin{bmatrix} 1 \\ 2x \end{bmatrix}$$

$$y = \vec{w}^T \vec{x} = \sum_i w_i x_i$$

$$\frac{\partial y}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_d} \end{bmatrix}$$

$$= \begin{bmatrix} w_1 & w_2 & \cdots & w_d \end{bmatrix}$$

$$= w^T$$

$$\frac{\partial \left(\sum_i \boxed{w_i} x_i \right)}{\partial x_1}$$

$$\frac{\partial (\vec{w}^T \cancel{\vec{x}})}{\partial \vec{x}} = w^T$$

Vector Derivative Example

$$\frac{\partial(\vec{w}^T A \vec{w})}{\partial \vec{w}} = \underline{2\vec{w}^T [A]}$$

$$\vec{y} = A\vec{x}$$

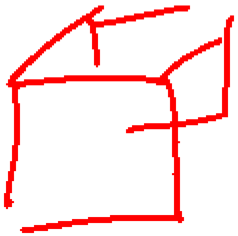
$$\frac{\partial \vec{y}}{\partial \vec{x}} = A \qquad y_i = \sum_j a_{ij} x_j$$

$$i \left[\frac{\partial y_i}{\partial x_j} \right] = \left[\cdots a_{ij} \right] \qquad \frac{\partial y_i}{\partial x_j} = a_{ij}$$

Extension to Tensors

$$\underline{X} \in \mathbb{R}^{d_1 \times \dots \times d_n}$$

$$\underline{Y} \in \mathbb{R}^{c_1 \times \dots \times c_n}$$



$$\frac{\partial Y[i_1, \dots, i_m]}{\partial X[j_1, \dots, j_n]}$$

$$y\text{-vec} = Y(:)$$

$$x\text{-vec} = X(:)$$

$$\frac{\partial \overrightarrow{y\text{-vec}}}{\partial \overrightarrow{x\text{-vec}}} = \begin{bmatrix} & (\; , \;) & \\ (\; , \;) & & (\;) \end{bmatrix}$$

Chain Rule: Composite Functions

$$f(g(x)) = (f \circ g)(x)$$

$L(\vec{w})$

$$f(x) = g_\ell(g_{\ell-1} \cdots g_1(x))$$

$$= (g_\ell \circ g_{\ell-1} \circ \cdots g_1)(x)$$

Chain Rule: Scalar Case

$\underline{x} \in \mathbb{R}^1 \xrightarrow[g_1()]{} z \in \mathbb{R}^1 \xrightarrow[g_2()]{} \underline{y} \in \mathbb{R}^1$

$y = g_2(\underbrace{g_1(x)}_{z})$

$$\frac{\partial y}{\partial \underline{x}} = \frac{\partial y}{\partial \underline{z}} \cdot \frac{\partial z}{\partial \underline{x}}$$

Scalar mult.

Chain Rule: Vector Case

$$\vec{x} \in \mathbb{R}^d \xrightarrow[:\mathbb{R}^d \to \mathbb{R}^m]{g_1(\,)} \vec{z} \in \mathbb{R}^m \xrightarrow[:\mathbb{R}^m \to \mathbb{R}^c]{g_2(\,)} \vec{y} \in \mathbb{R}^c$$

$$\underbrace{\left[\frac{\partial \vec{y}}{\partial \vec{x}}\right]}_{\boxed{J_{g_2 \circ g_1}}} = \underbrace{\left[\frac{\partial \vec{y}}{\partial \vec{z}}\right]}_{J_{g_2}} \cdot \left[\frac{\partial \vec{z}}{\partial \vec{x}}\right]$$

Matrix $\underbrace{\text{Mult}}_{J_{g_1}}$

Chain Rule: Jacobian view

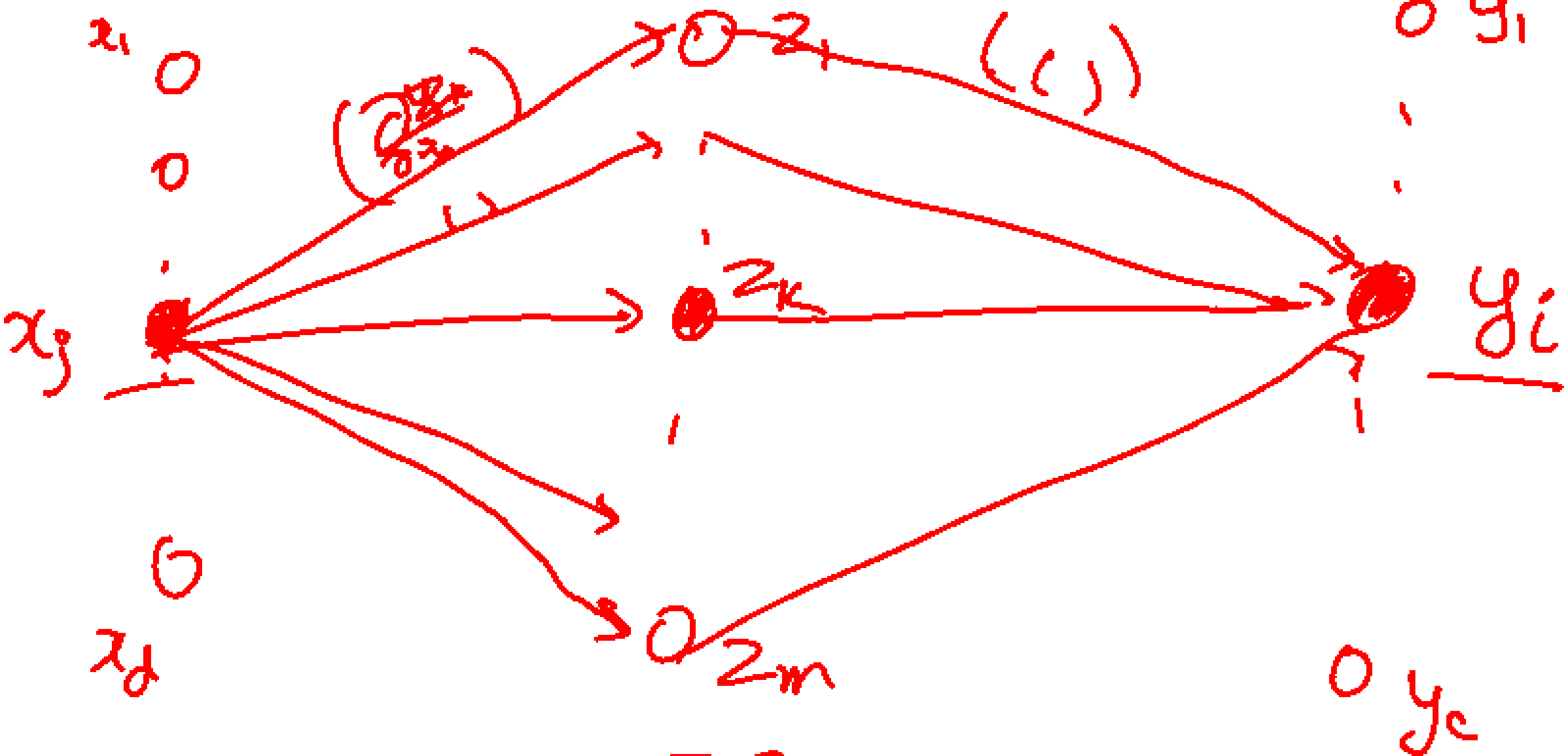
$$\frac{\partial \vec{y}}{\partial \vec{x}} \qquad\qquad \frac{\partial \vec{y}}{\partial \vec{z}} \qquad\qquad \frac{\partial \vec{z}}{\partial \vec{x}}$$

$$\begin{array}{c c} & j \\ i & \begin{bmatrix} \vdots \\ \frac{\partial y_i}{\partial x_j} \end{bmatrix} \end{array} = \begin{array}{c c} i & \begin{bmatrix} \cdots & \frac{\partial y_i}{\partial z_k} & \cdots \end{bmatrix} \end{array} \begin{array}{c} j \\ \begin{bmatrix} \vdots \\ \frac{\partial z_k}{\partial x_j} \\ \vdots \end{bmatrix} \end{array}$$

$$\frac{\partial y_i}{\partial x_j} = \sum_{\boxed{k}} \frac{\partial y_i}{\partial z_k} \frac{\partial z_k}{\partial x_j}$$

Chain Rule: Graphical view

$\vec{x} \in \mathbb{R}^d \longrightarrow z \in \mathbb{R}^m \longrightarrow y \in \mathbb{R}^C$



$$\frac{\partial y_i}{\partial x_j} = \boxed{\sum_{k \text{ on path}}} \frac{\partial y_i}{\partial z_k} \frac{\partial z_k}{\partial x_j}$$

Chain Rule: Cascaded

$$x = \underline{h}^0 \xrightarrow{g_1(\cdot)} \vec{h}^1 \in \mathbb{R}^d \xrightarrow{g_2(\cdot)} \vec{h}^{(2)} \in \mathbb{R}^d \cdots \xrightarrow{g_\ell(\cdot)} \vec{h}^{(\ell)} \to L$$

$$\frac{\partial L}{\partial \vec{h}^{(1)}} = \boxed{\frac{\partial L}{\partial \vec{h}^{\ell}}}\boxed{\frac{\partial \vec{h}^{(\ell)}}{\partial \vec{h}^{(1)}}} = \boxed{\frac{\partial L}{\partial \vec{h}^{(\ell)}}}\frac{\partial \vec{h}^{(\ell)}}{\partial \vec{h}^{\ell-1}} \cdot \frac{\partial \vec{h}^{\ell-1}}{\partial \vec{h}^{\ell-2}} \cdots \frac{\partial \vec{h}^{(2)}}{\partial \vec{h}^{(1)}}$$

$$[\,] = [\,][\,]_{d\times d} = \underbrace{[\,]_{1\times d}[\,]_{d\times d}}_{O(d^2)} \qquad \underbrace{[\,]_{d\times d}[\,]_{d\times d}}_{O(\ell d^3)}$$

Chain Rule: How should we multiply?

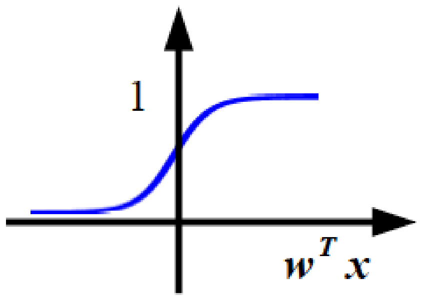
Example: Logistic Regression

Input: $\boldsymbol{x} \in R^D$

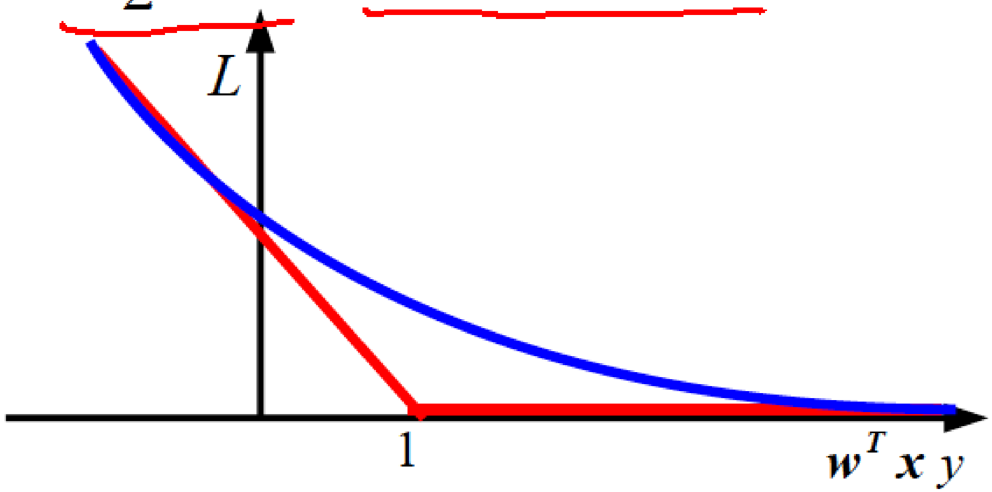
Binary label: $y \in \{-1, +1\}$

Parameters: $\boldsymbol{w} \in R^D$

Output prediction: $p(y=1|\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{w}^T \boldsymbol{x}}}$



Loss: $L = \frac{1}{2}||\boldsymbol{w}||^2 - \lambda \log(p(y|\boldsymbol{x}))$



Log Loss

Logistic Regression Derivatives

$$L(w) = -\log\left(\frac{1}{1+e^{-w^T x}}\right) = \log\underbrace{\left(1+e^{\overbrace{-w^T x}^{s}}\right)}_{p}$$

$$\left[\ \frac{\partial L}{\partial \vec{w}} = \left(\frac{1}{p}\right)\left(e^{-w^T x}\right)\cdot\left(-x^T\right)\ \right]$$

$$[\qquad\qquad]$$

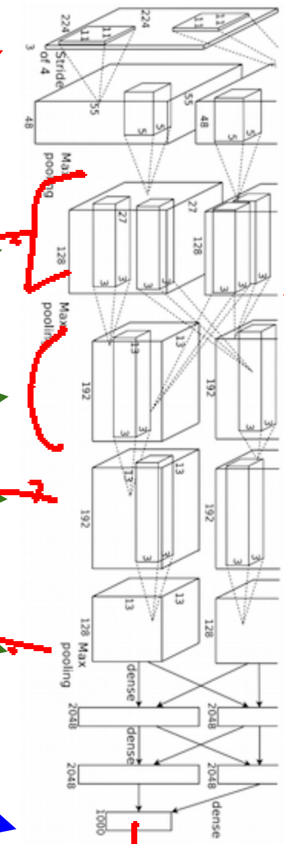
Logistic Regression Derivatives

Convolutional network (AlexNet)

input image

weights

loss



$$\frac{\partial L}{\partial w}$$

$L(\ \ )$

Neural Turing Machine

input image

loss

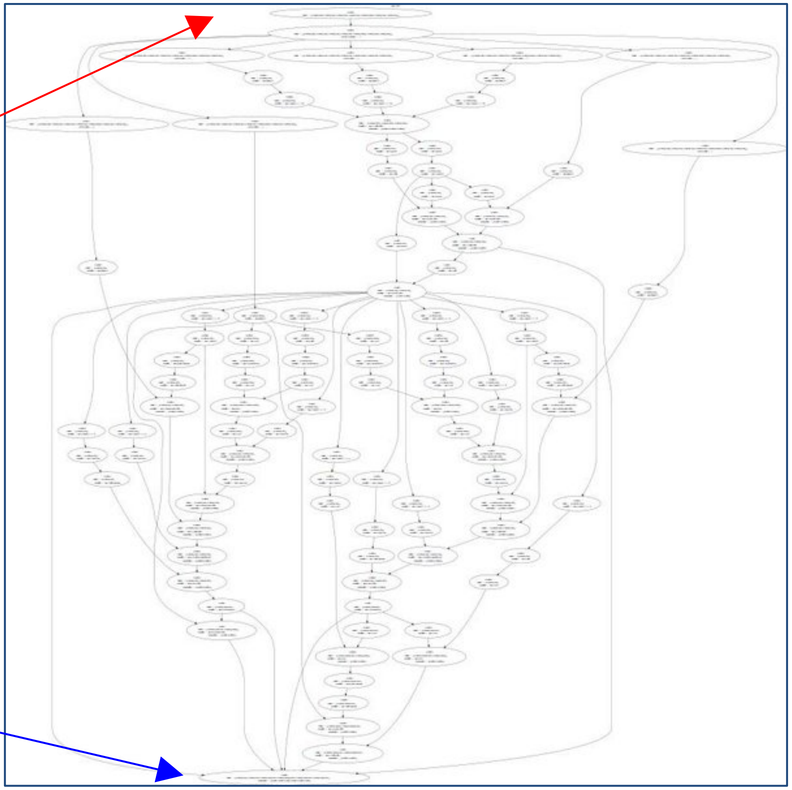


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

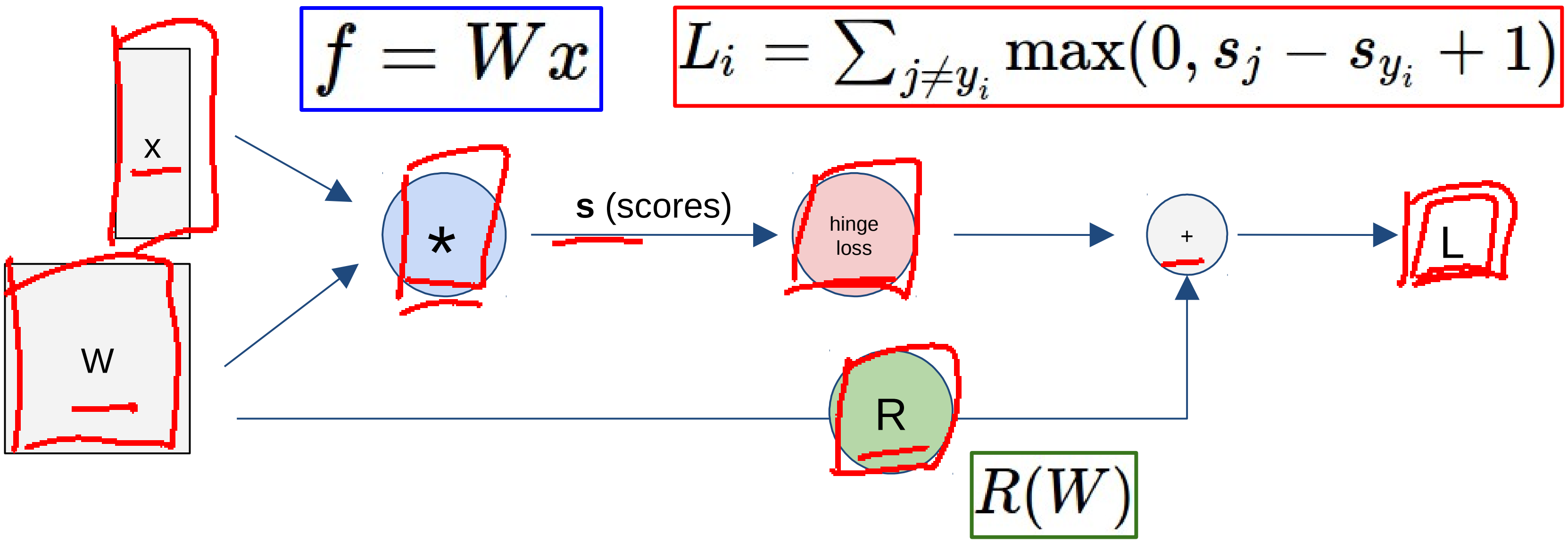
How do we compute gradients?

- Analytic or “Manual” Differentiation
- Symbolic Differentiation
- Numerical Differentiation
- Automatic Differentiation
  - Forward mode AD
  - Reverse mode AD
    - aka “backprop”

Deep Learning = Differentiable Programming

- Computation = Graph
  - Input = Data + Parameters
  - Output = Loss
  - Scheduling = Topological ordering
- Auto-Diff
  - A family of algorithms for implementing chain-rule on computation graphs

Computational Graph



Neural Network Computation Graph

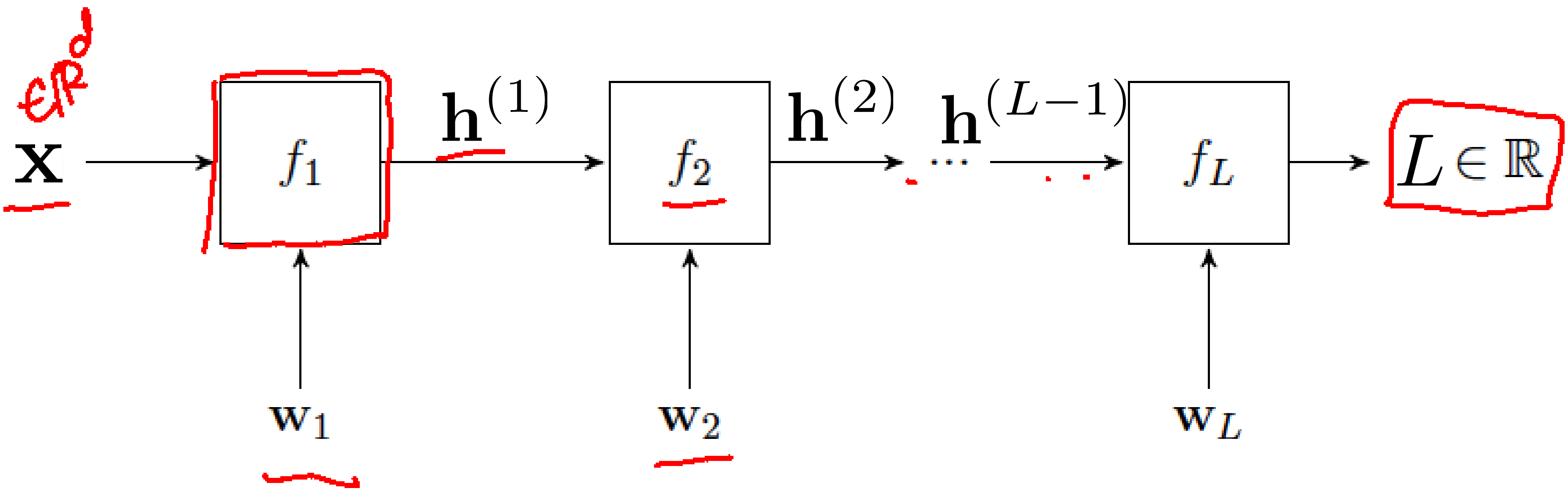
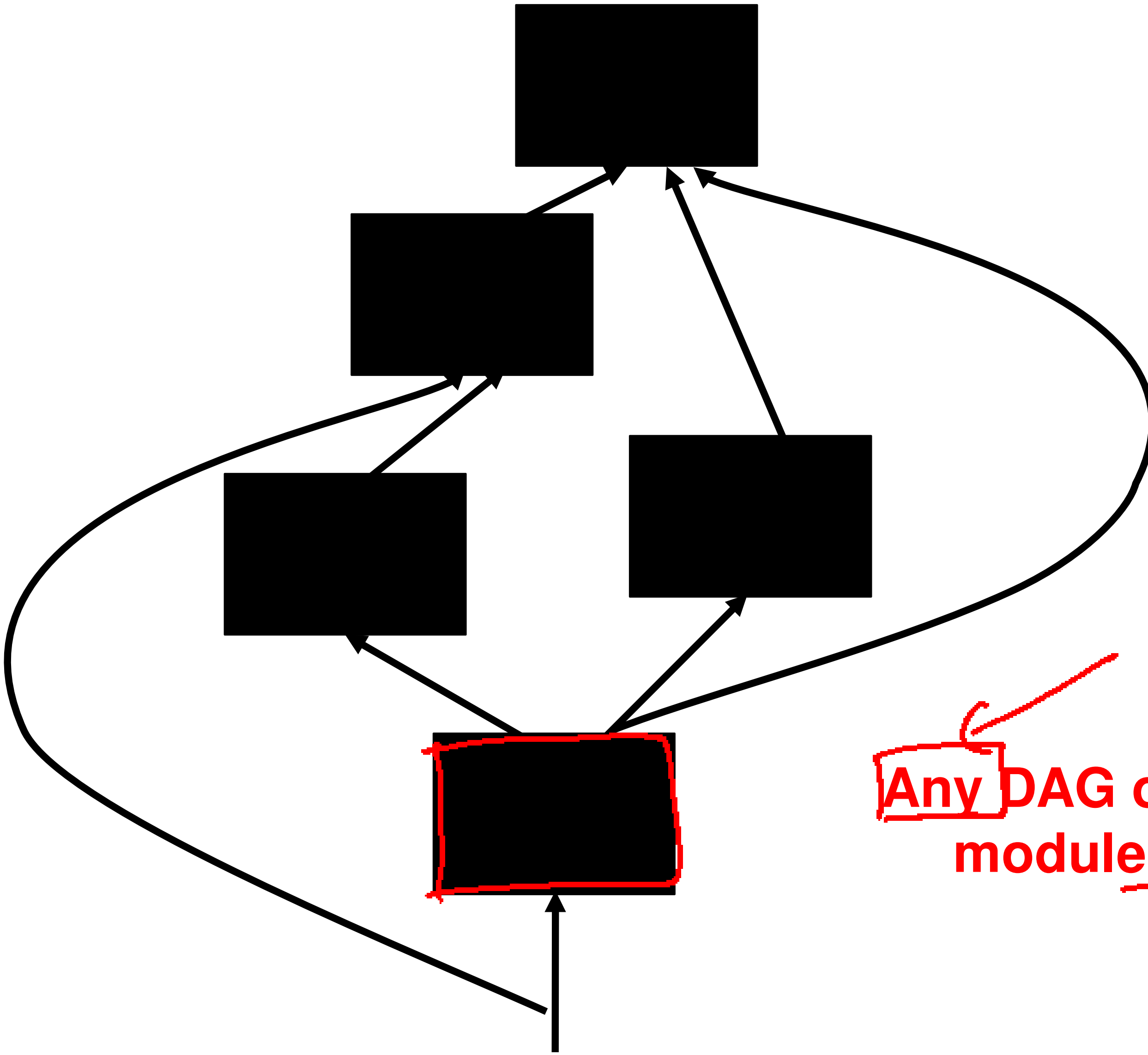


Figure Credit: Andrea Vedaldi

Computational Graph

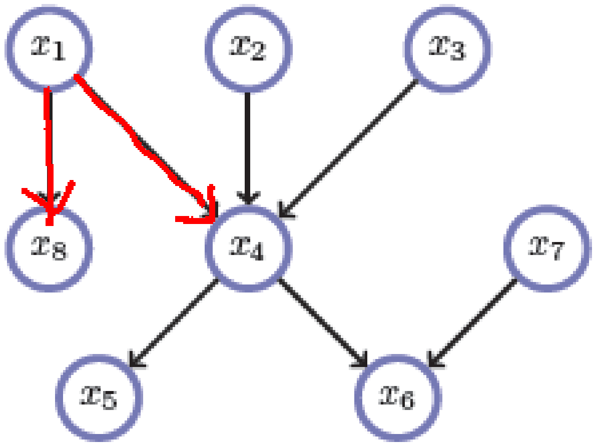


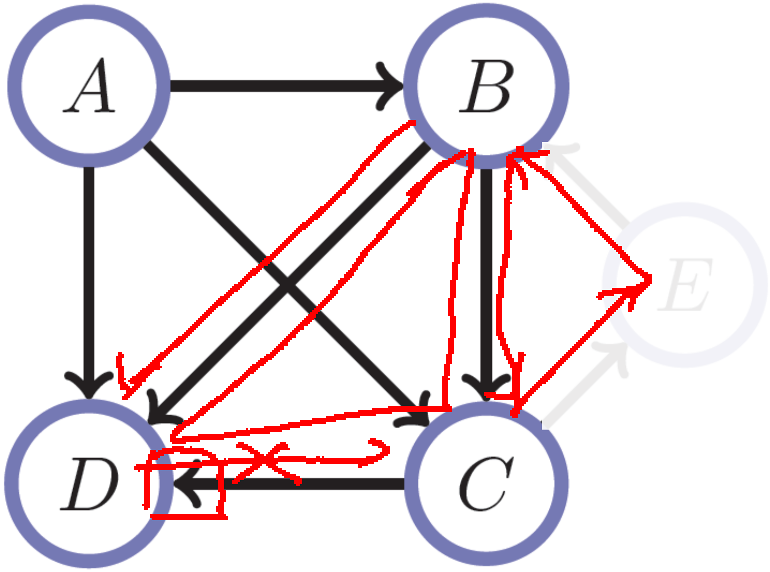
Any DAG of differentiable modules is allowed!

Directed Acyclic Graphs (DAGs)

- Exactly what the name suggests
  - Directed edges
  - No (directed) cycles
  - Underlying undirected cycles okay

DAG ✓





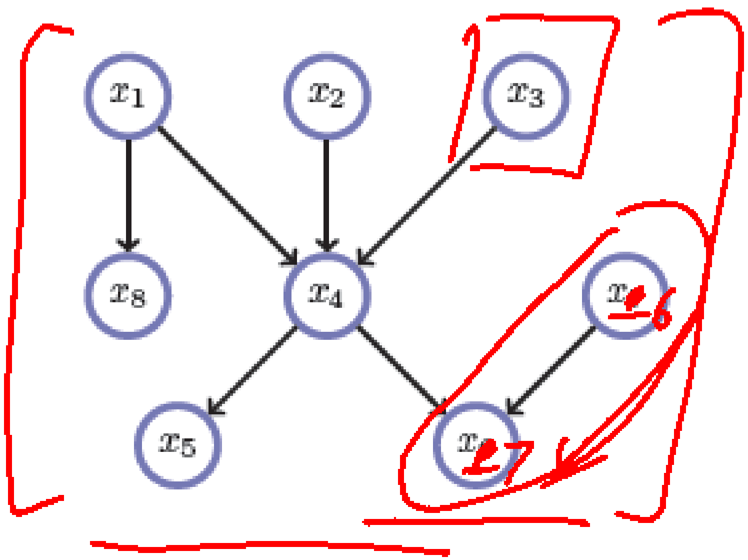
Directed Acyclic Graphs (DAGs)

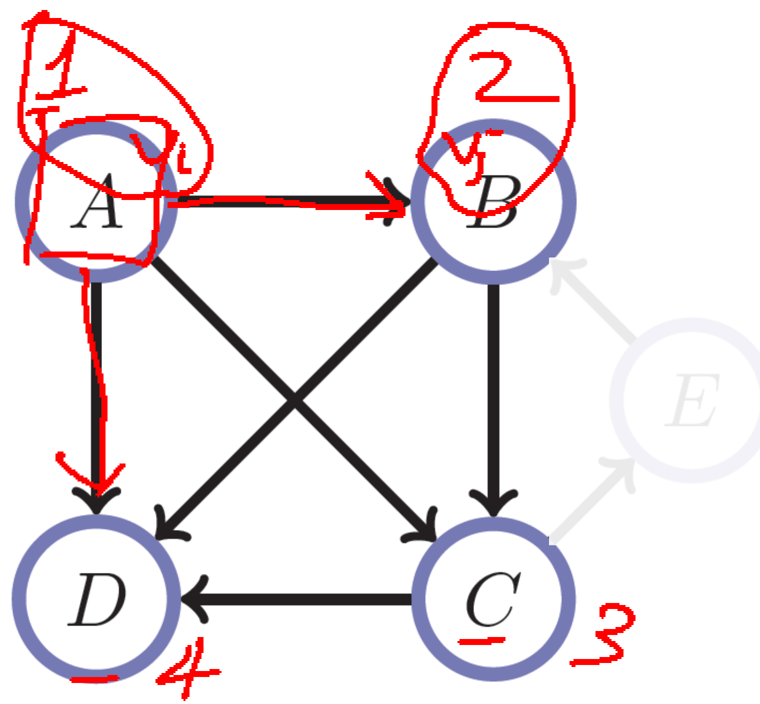
- Concept
  - Topological Ordering

$\exists$ bijection $\sigma : V \to \{1, \ldots, n\}$

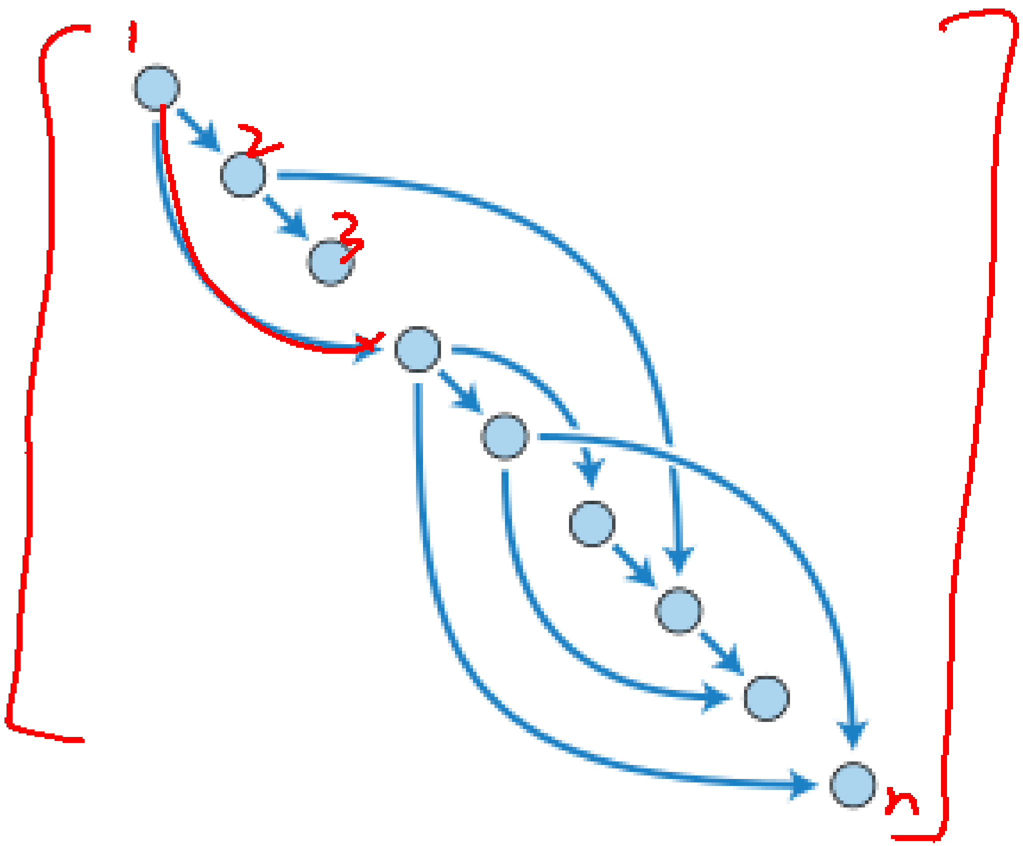
s.t. $\forall (v_i, v_j) \in E$

$\sigma(v_i) < \sigma(v_j)$





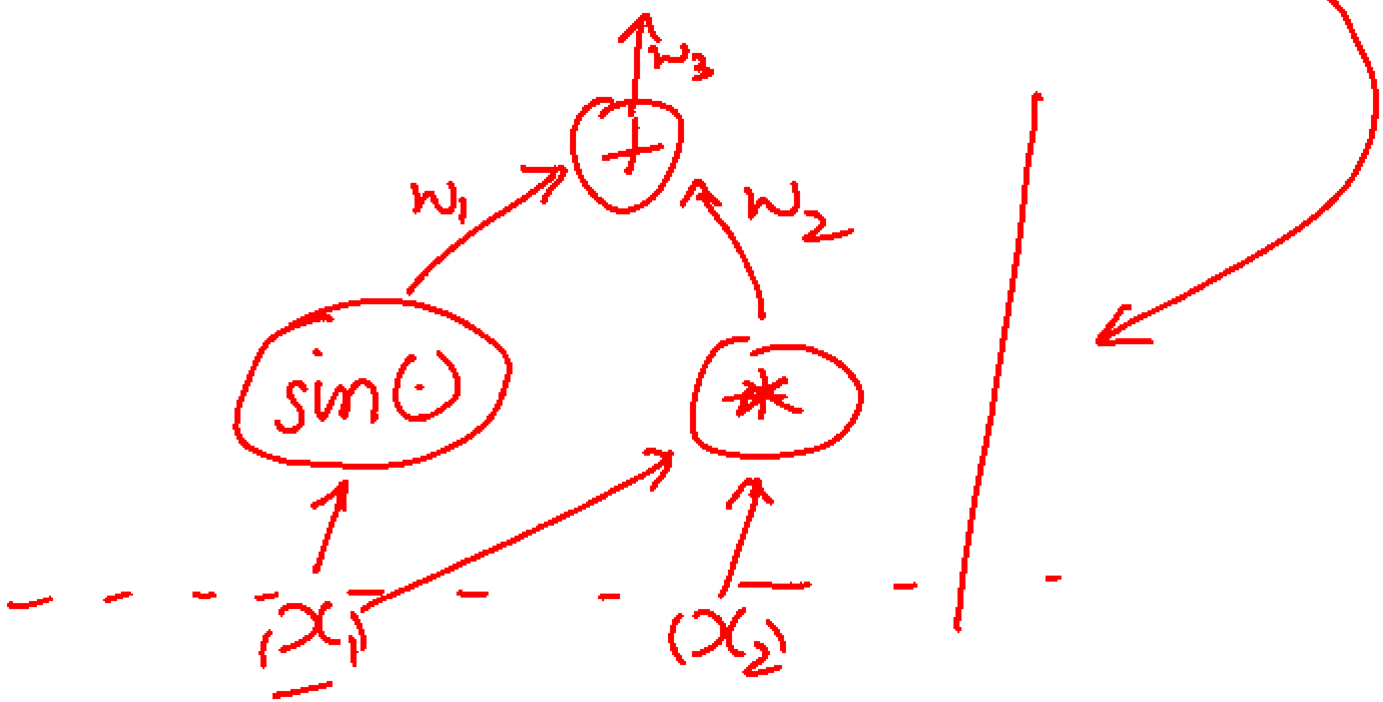
Directed Acyclic Graphs (DAGs)



Computational Graphs

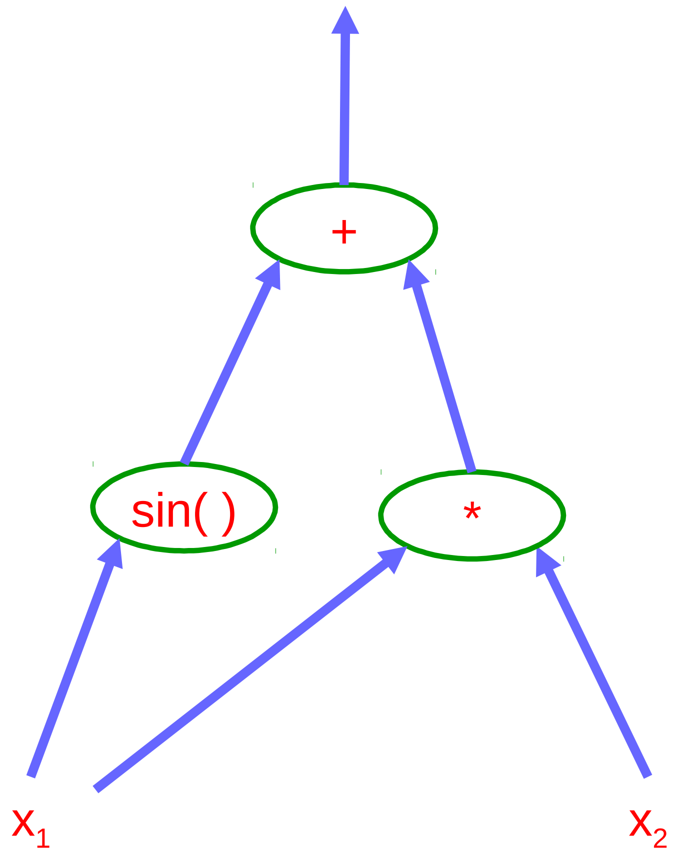
- Notation

$$f(x_1, x_2) = \overbrace{x_1 x_2}^{w_2} + \overbrace{\sin(x_1)}^{w_1}$$



Example

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

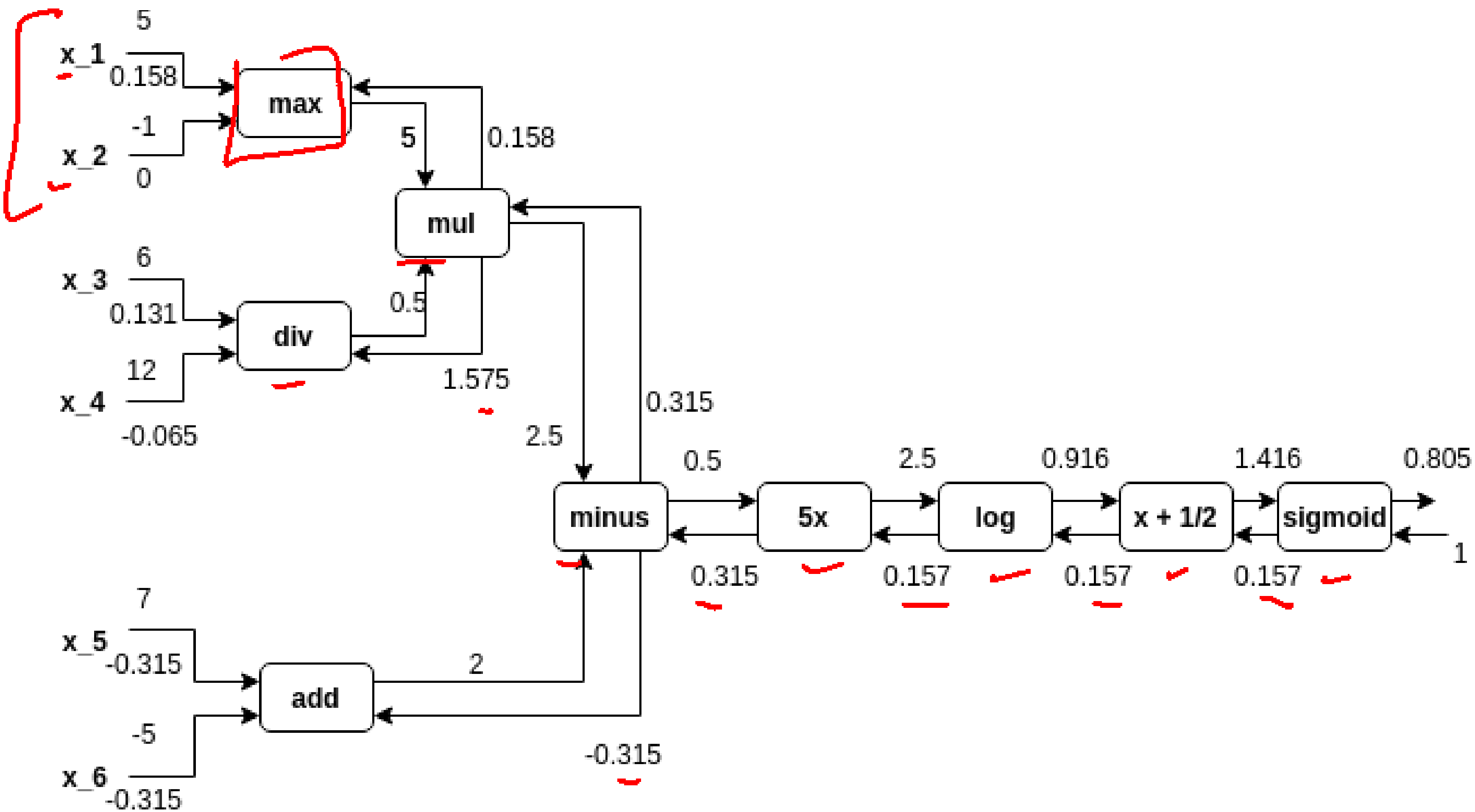


HW0

$$f(\mathbf{x}) = \sigma\left(\log\left(5\left(\max\{x_1, x_2\} \cdot \frac{x_3}{x_4} - (x_5 + x_6)\right)\right) + \frac{1}{2}\right)$$

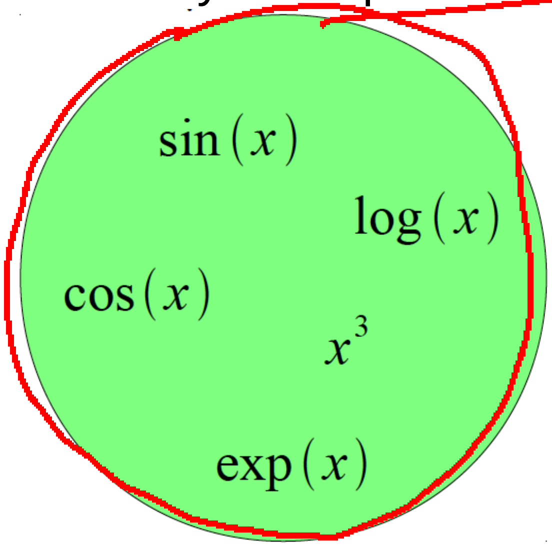
HW0 Submission by Samyak Datta

$$f(\mathbf{x}) = \sigma\left(\log\left(5\left(\max\{x_1, x_2\} \cdot \frac{x_3}{x_4} - (x_5 + x_6)\right)\right) + \frac{1}{2}\right)$$



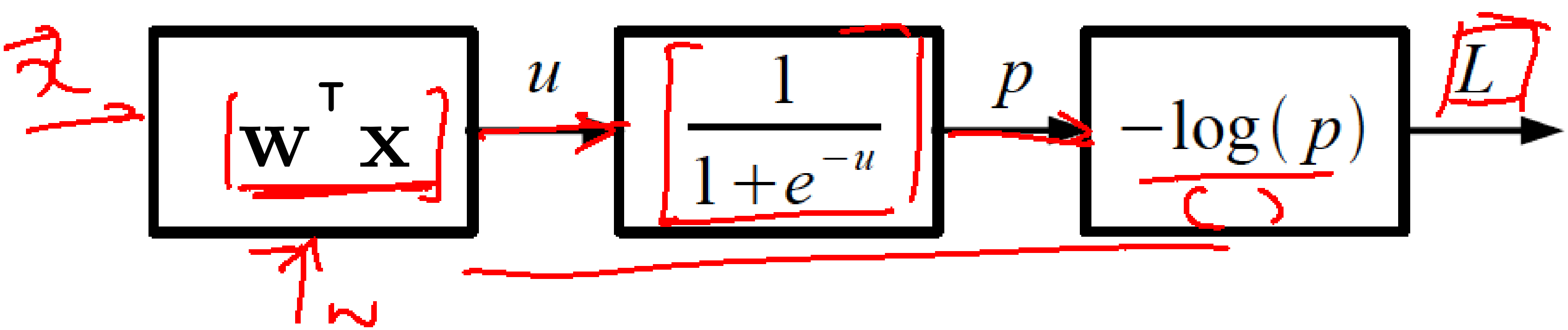
Logistic Regression as a Cascade

Given a library of simple functions



Compose into a complicate function

$$-\log\left(\frac{1}{1+e^{-\mathbf{w}^{\intercal}\mathbf{x}}}\right)$$



Deep Learning = Differentiable Programming

- Computation = Graph
  - Input = Data + Parameters
  - Output = Loss
  - $\boxed{\text{Scheduling}}$ = Topological ordering

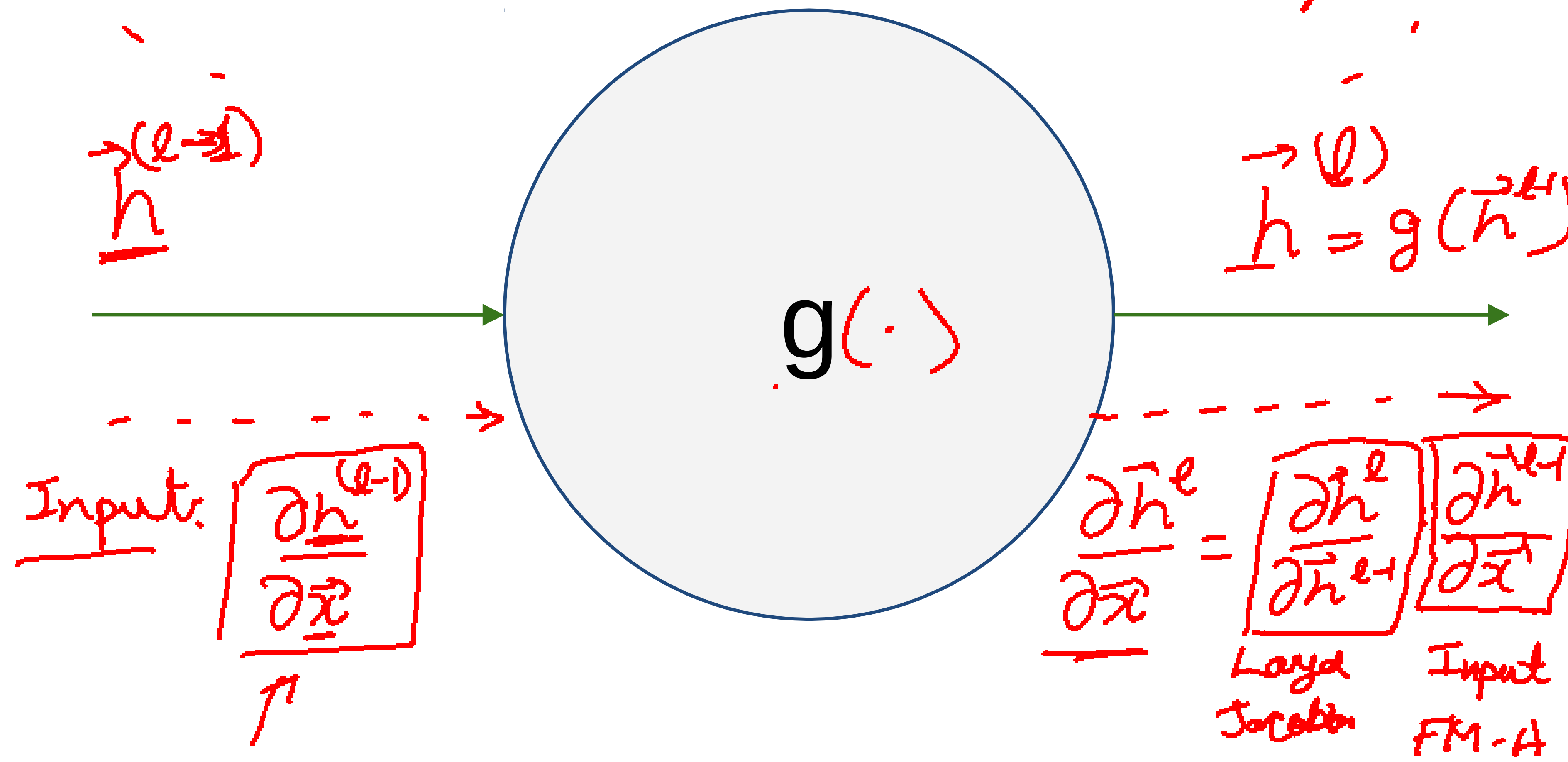
- Auto-Diff
  - A family of algorithms for implementing chain-rule on computation graphs

Forward mode vs Reverse Mode AD

- Key Computations

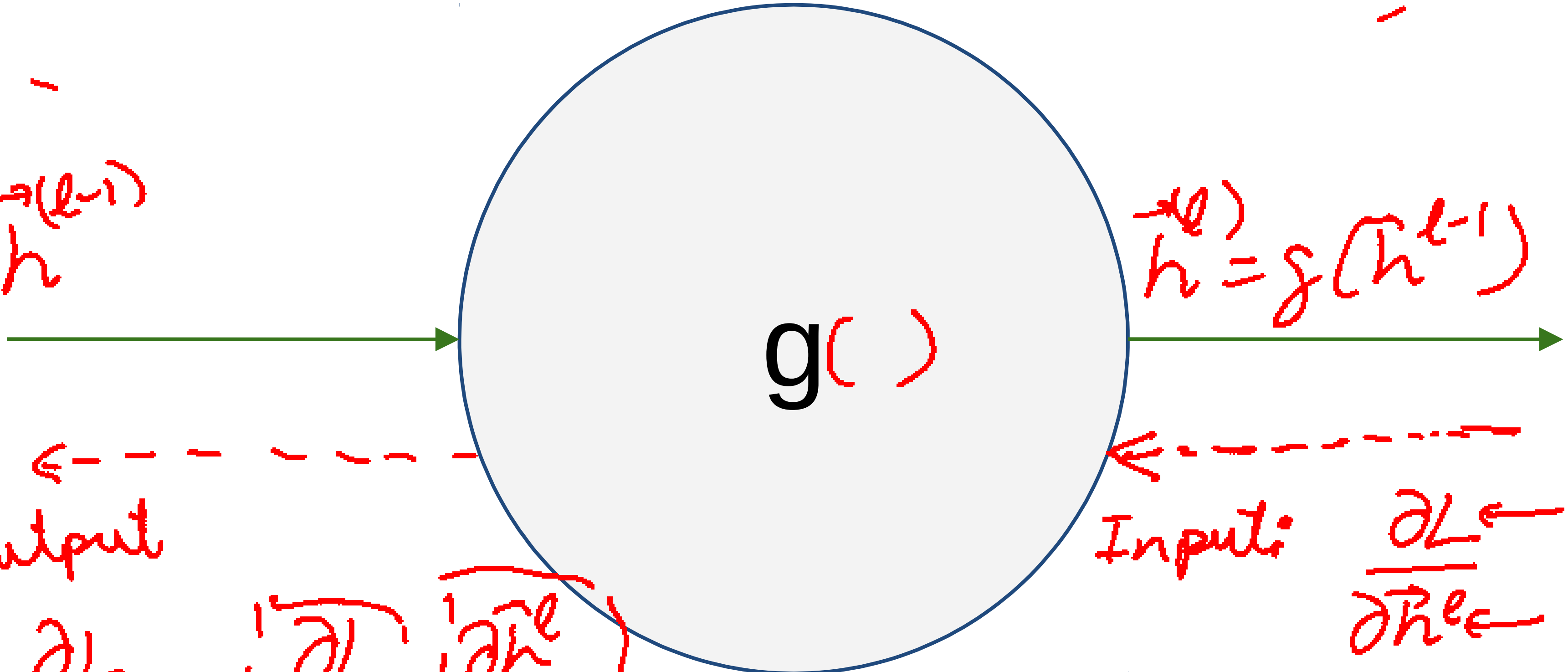
Forward mode AD

Goal: $\frac{\partial L}{\partial x}$ $\boxed{\frac{\partial L}{\partial x}}$



Reverse mode AD

Goal: $\frac{\partial L}{\partial x}$



$\vec{h}^{(l-1)}$

$\vec{h}^{(l)} = g(\vec{h}^{l-1})$

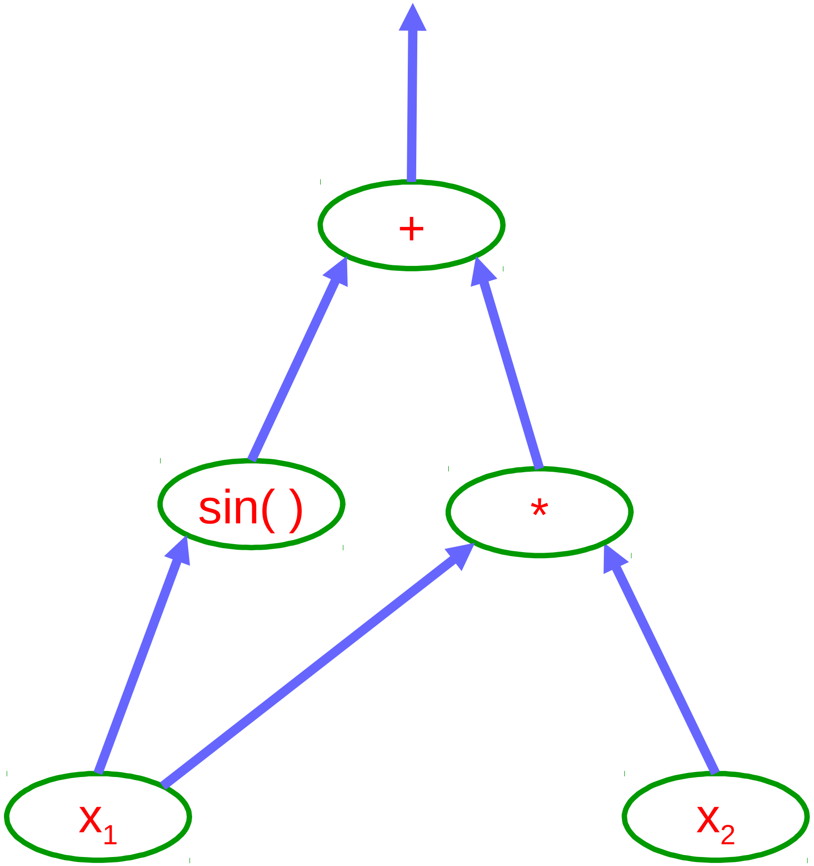
Output

Input: $\frac{\partial L}{\partial \vec{h}^{l}}$

$$\frac{\partial L}{\partial \vec{h}^{l-1}} = \underbrace{\frac{\partial L}{\partial \vec{h}^{l}}}_{\text{Input}} \underbrace{\frac{\partial \vec{h}^{l}}{\partial \vec{h}^{l-1}}}_{\text{Layer Jacobian}}$$

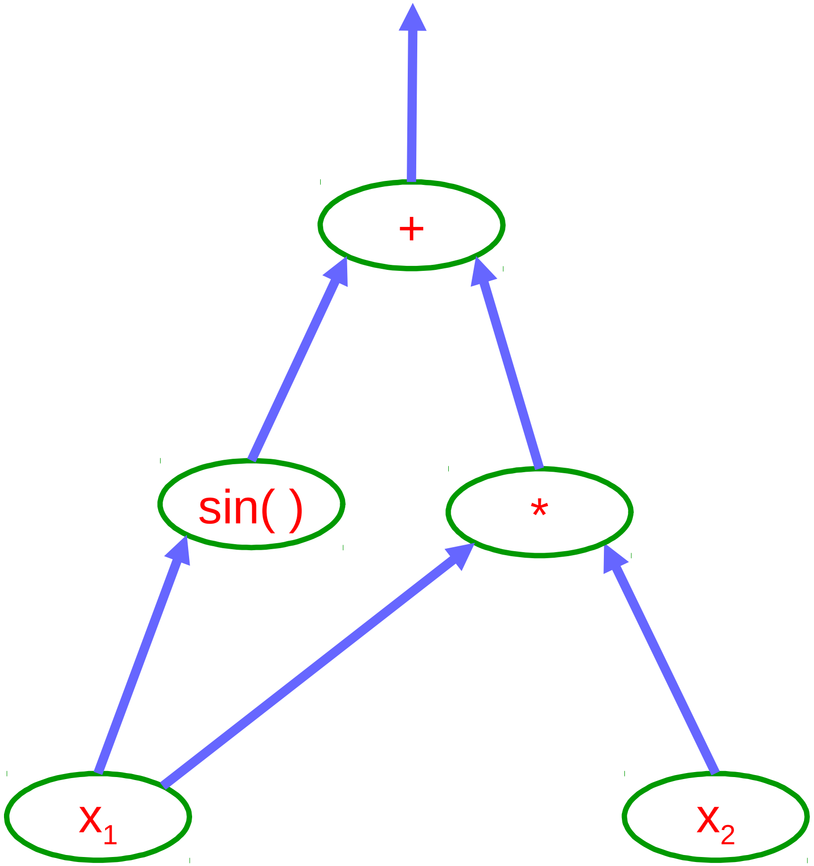
Example: Forward mode AD

$$f(x_1, x_2) = \underline{\sin(x_1)} + \underline{x_1 x}_2$$



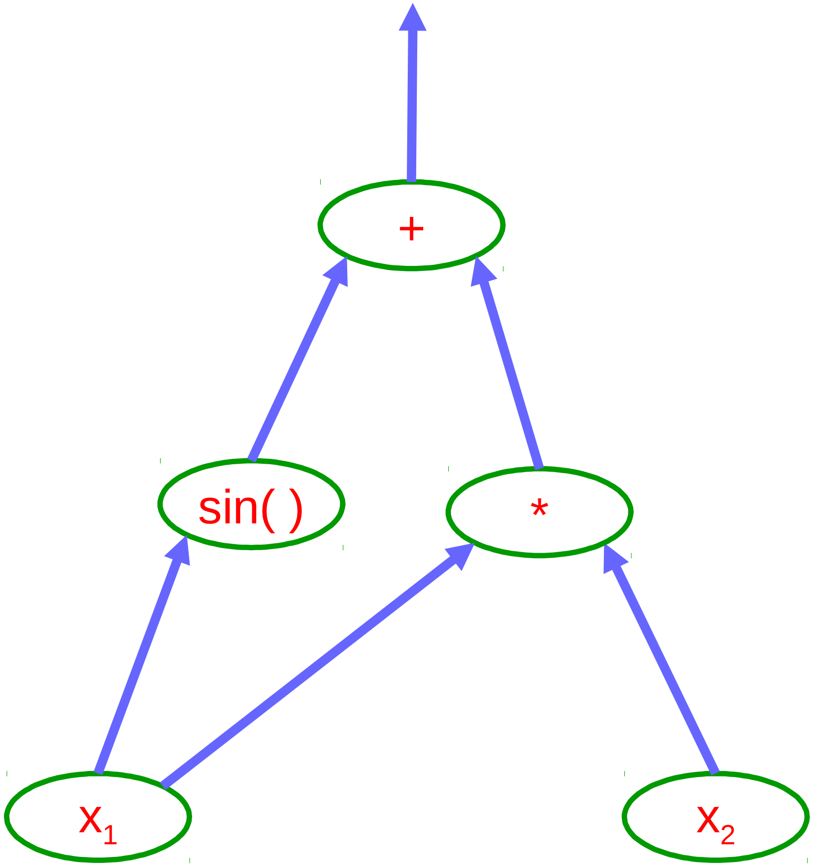
Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$



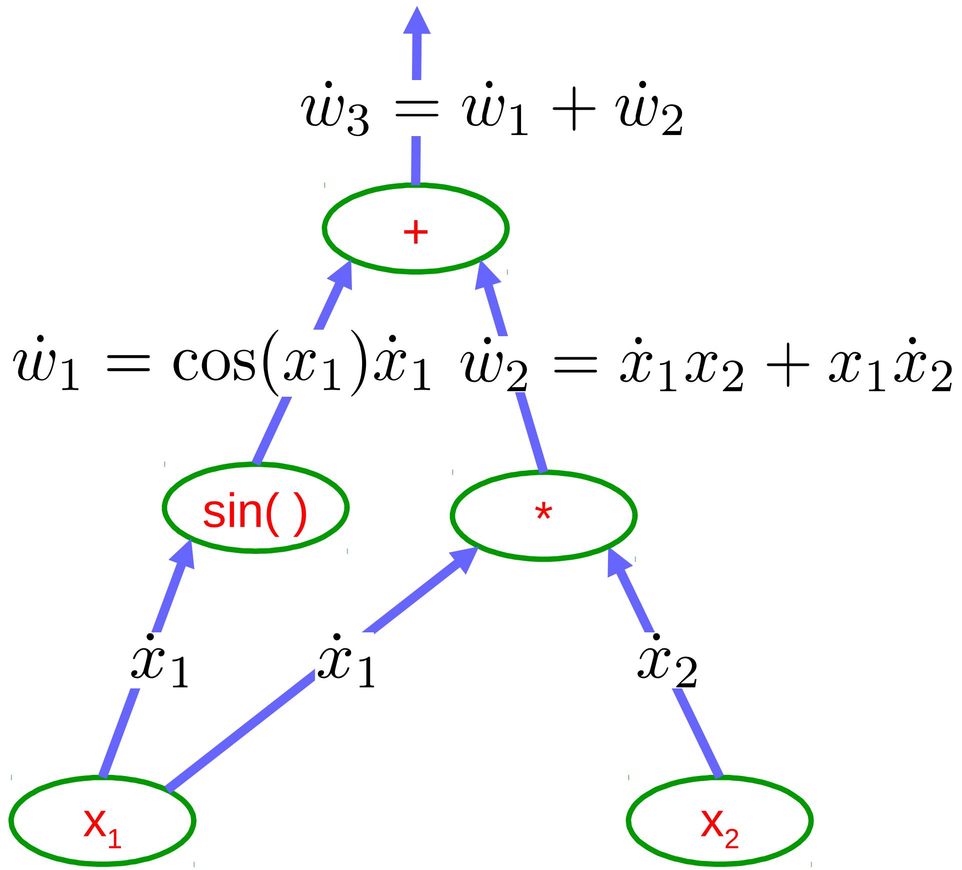
Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$



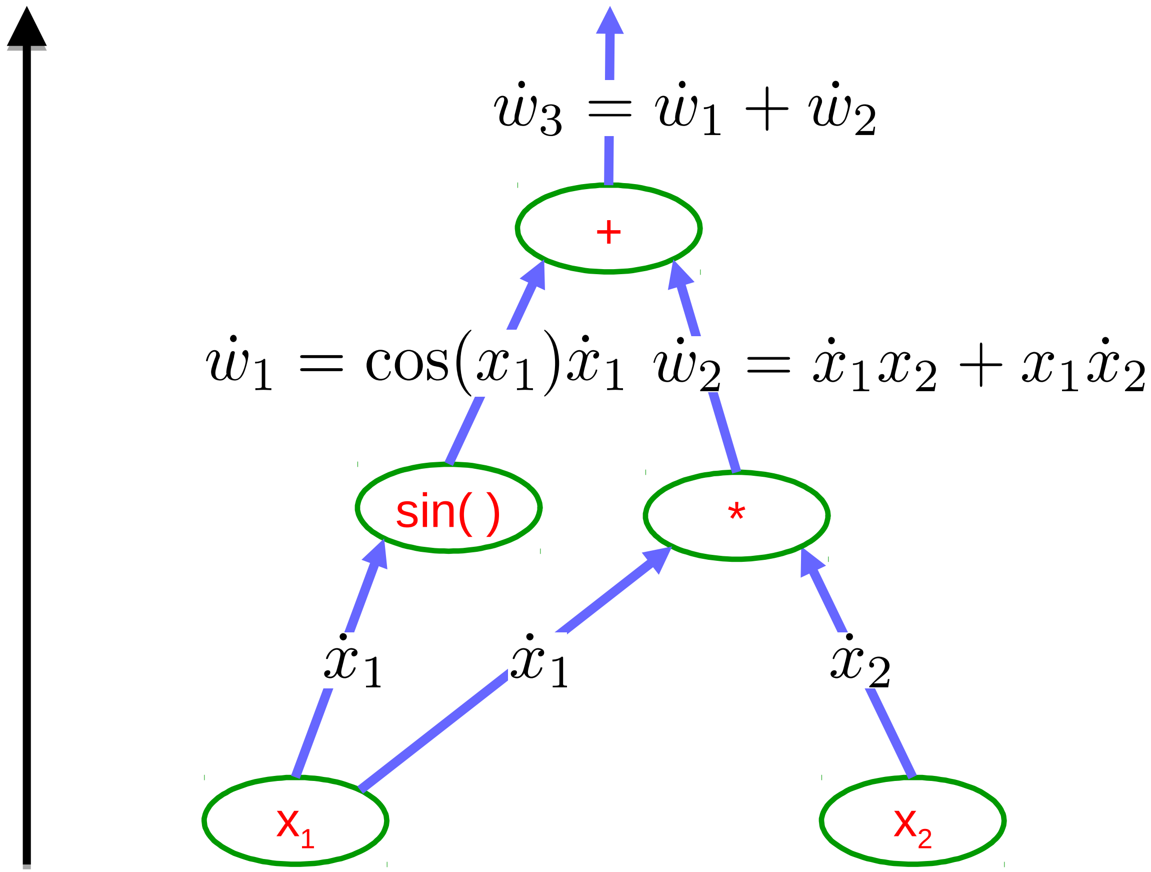
Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$



Example: Forward mode AD

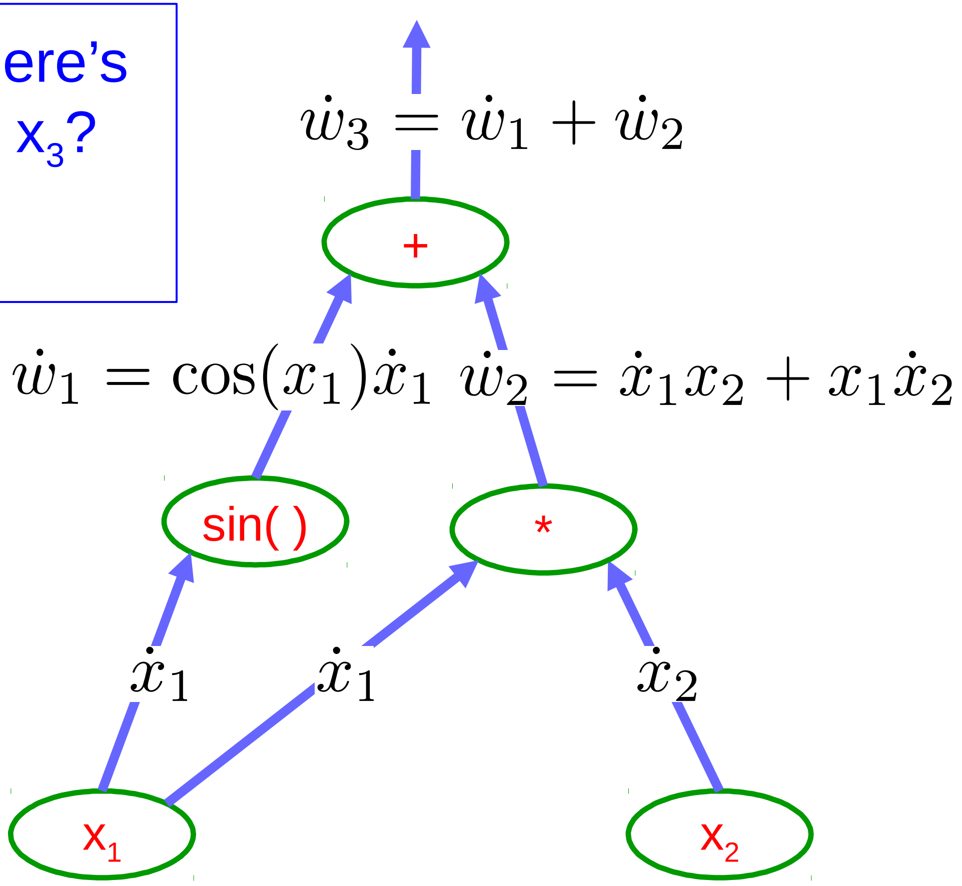
$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$



Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

Q: What happens if there's another input variable x_3?

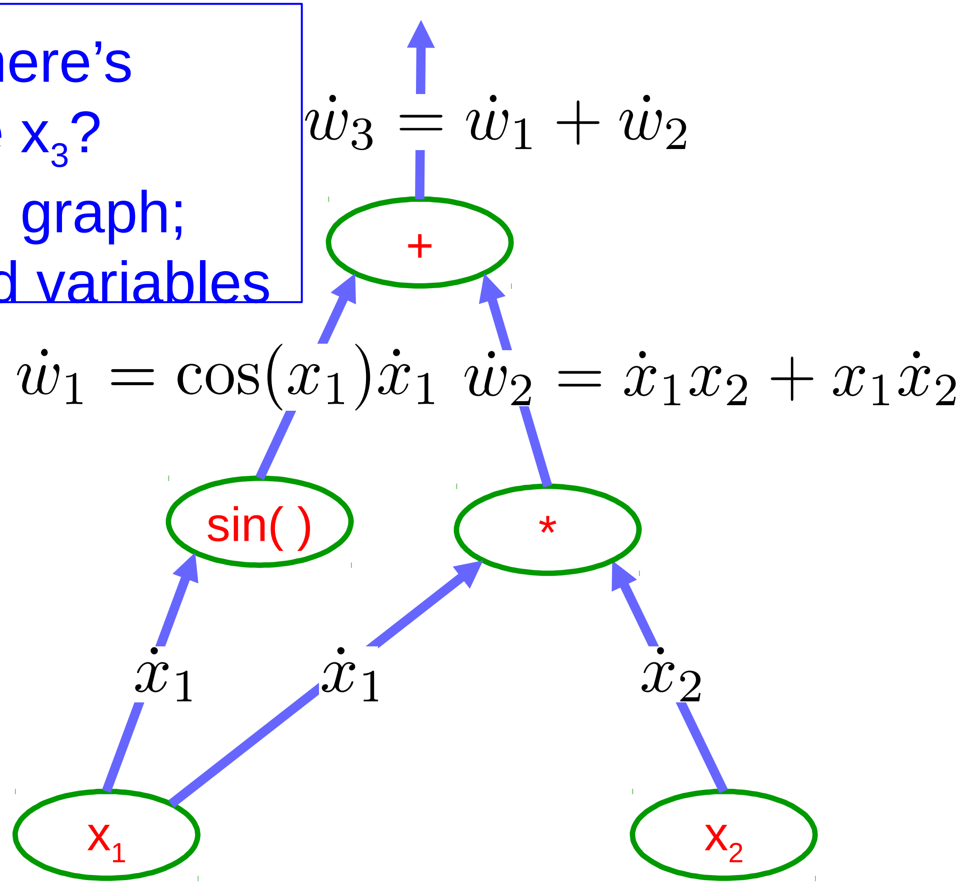


Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

Q: What happens if there's another input variable x_3?

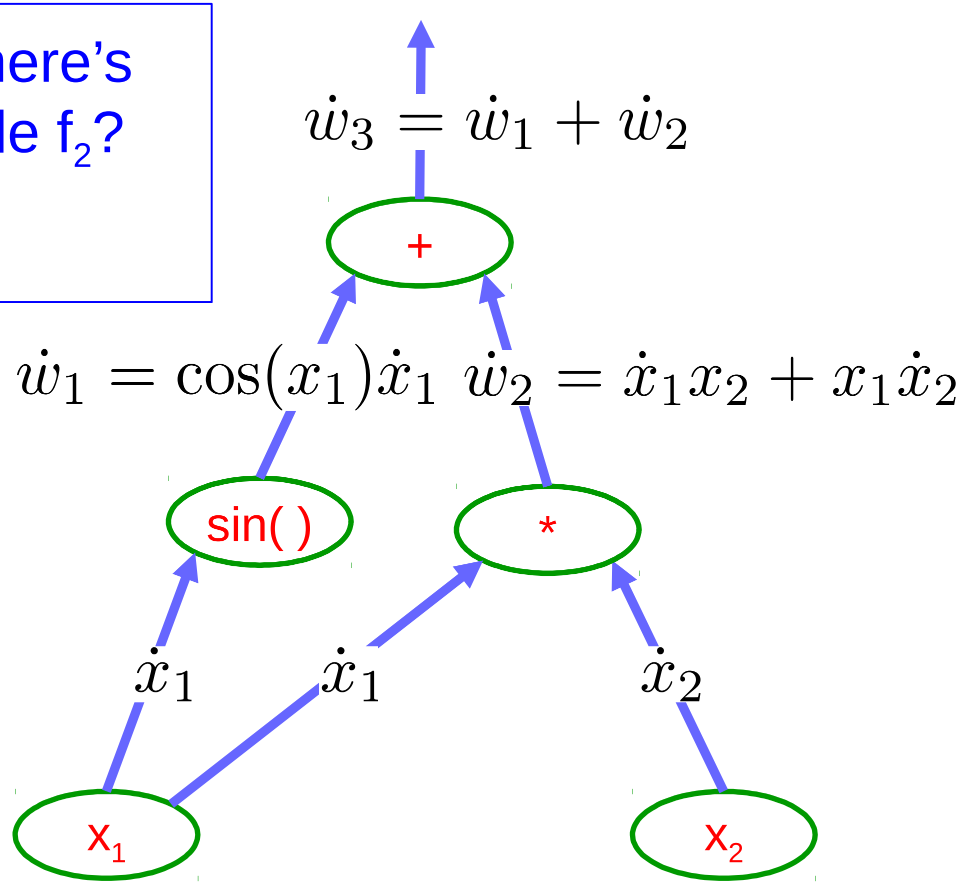
A: more sophisticated graph; d "forward props" for d variables



Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

Q: What happens if there's another output variable f_2?



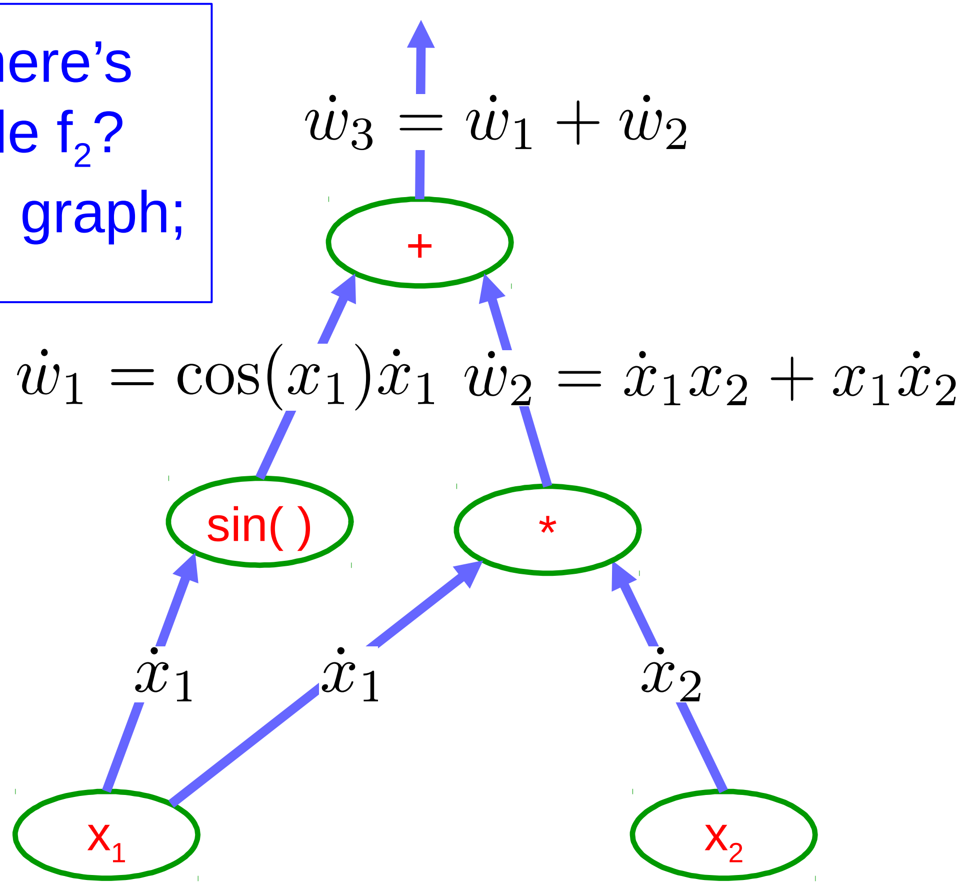
Example: Forward mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

Q: What happens if there's another output variable f_2?

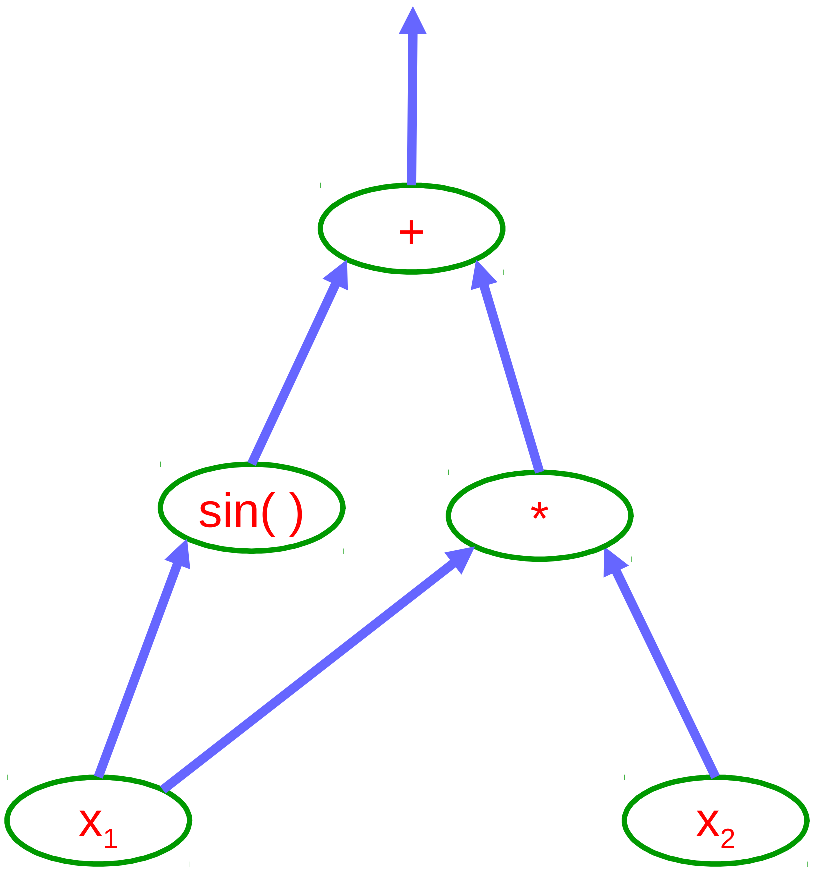
A: more sophisticated graph;

single "forward prop"



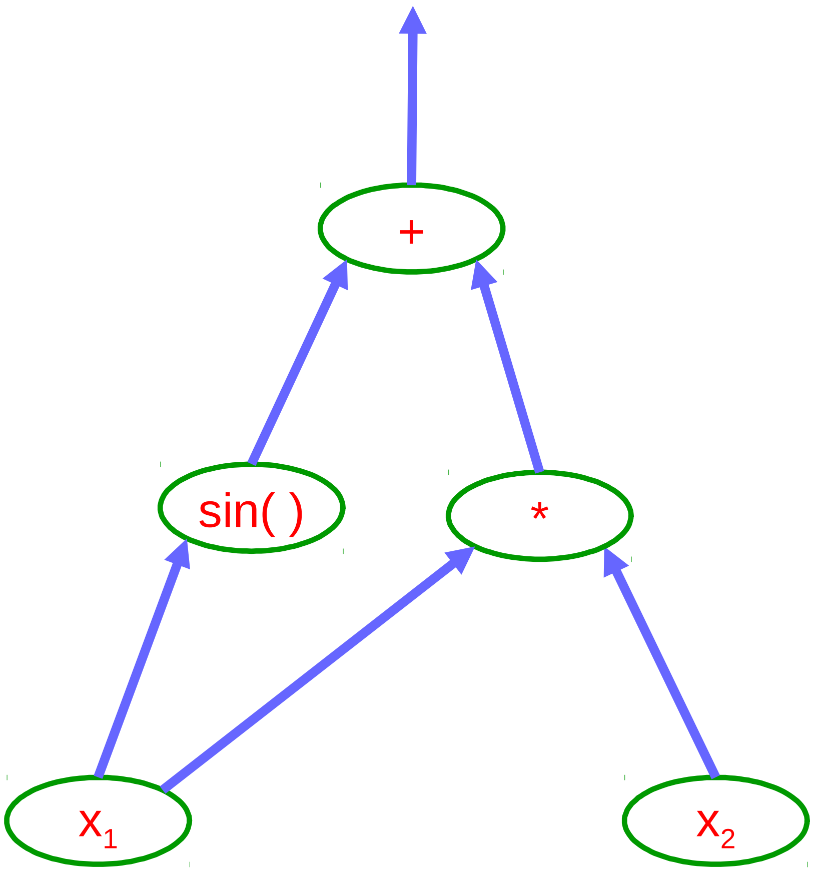
Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$



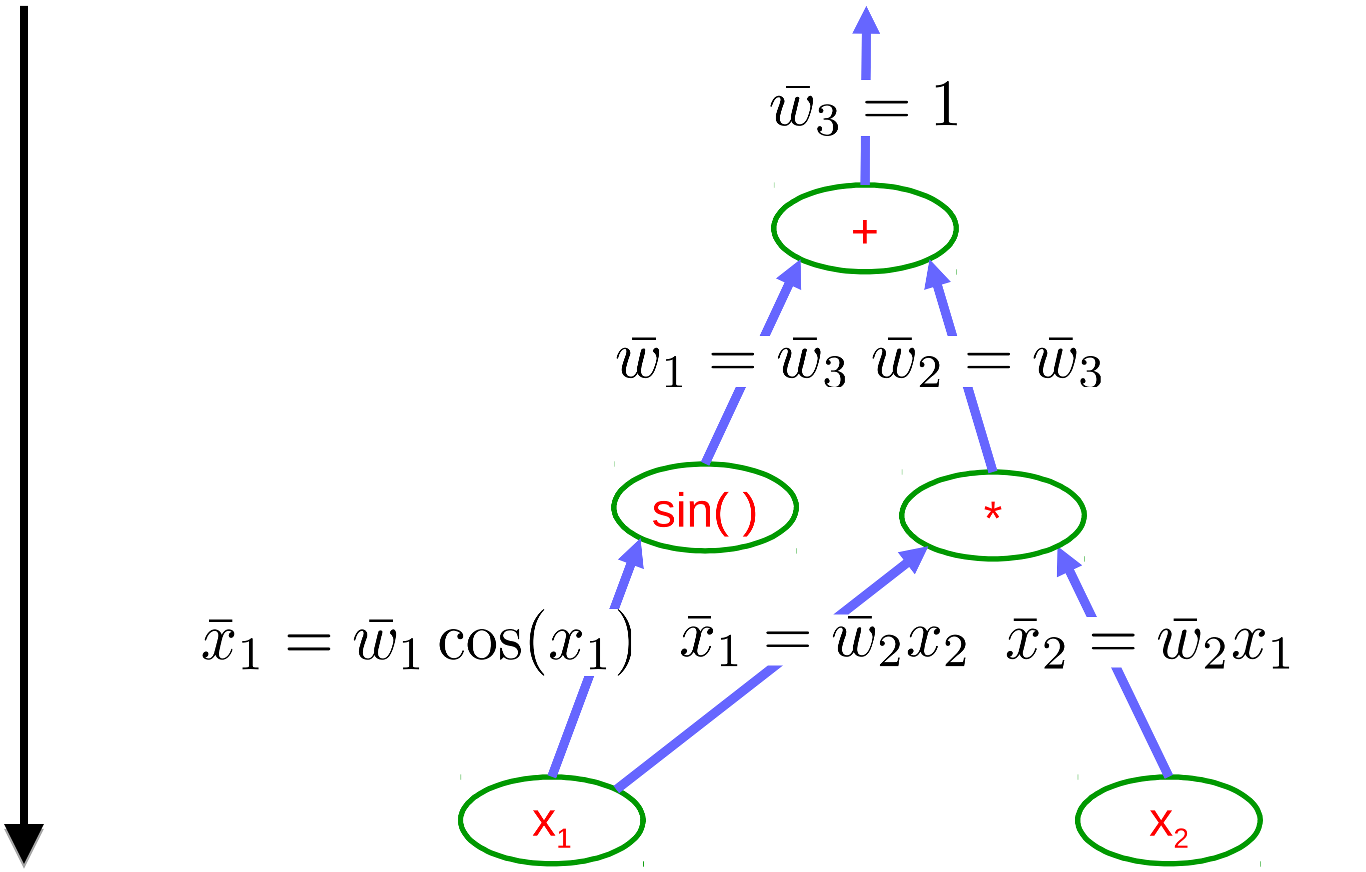
Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

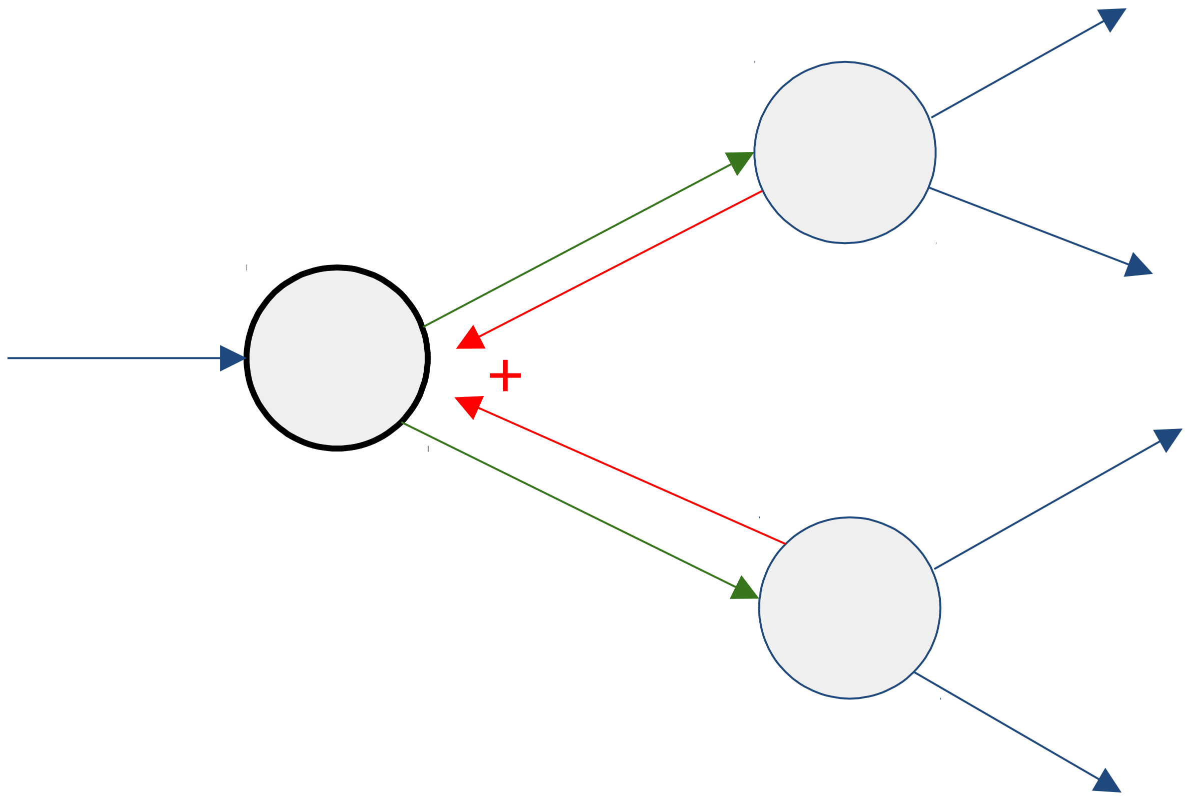


Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$



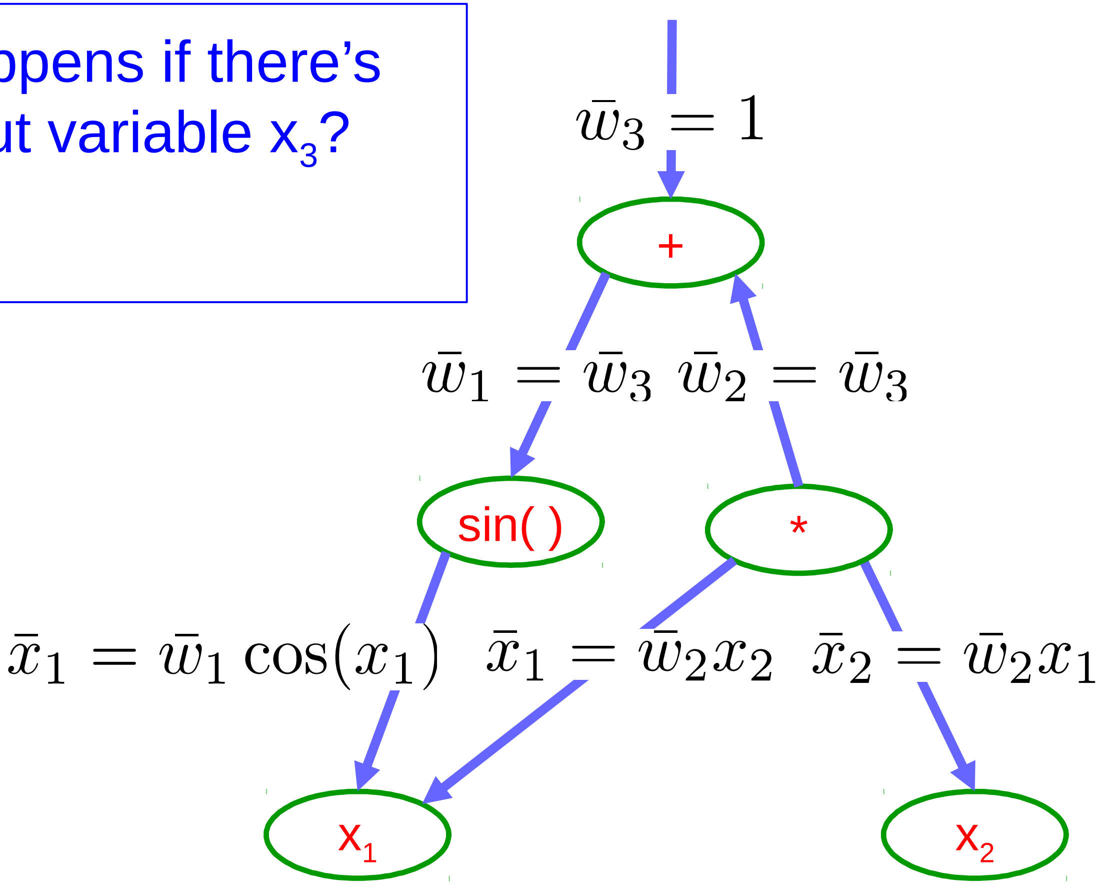
Gradients add at branches



Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

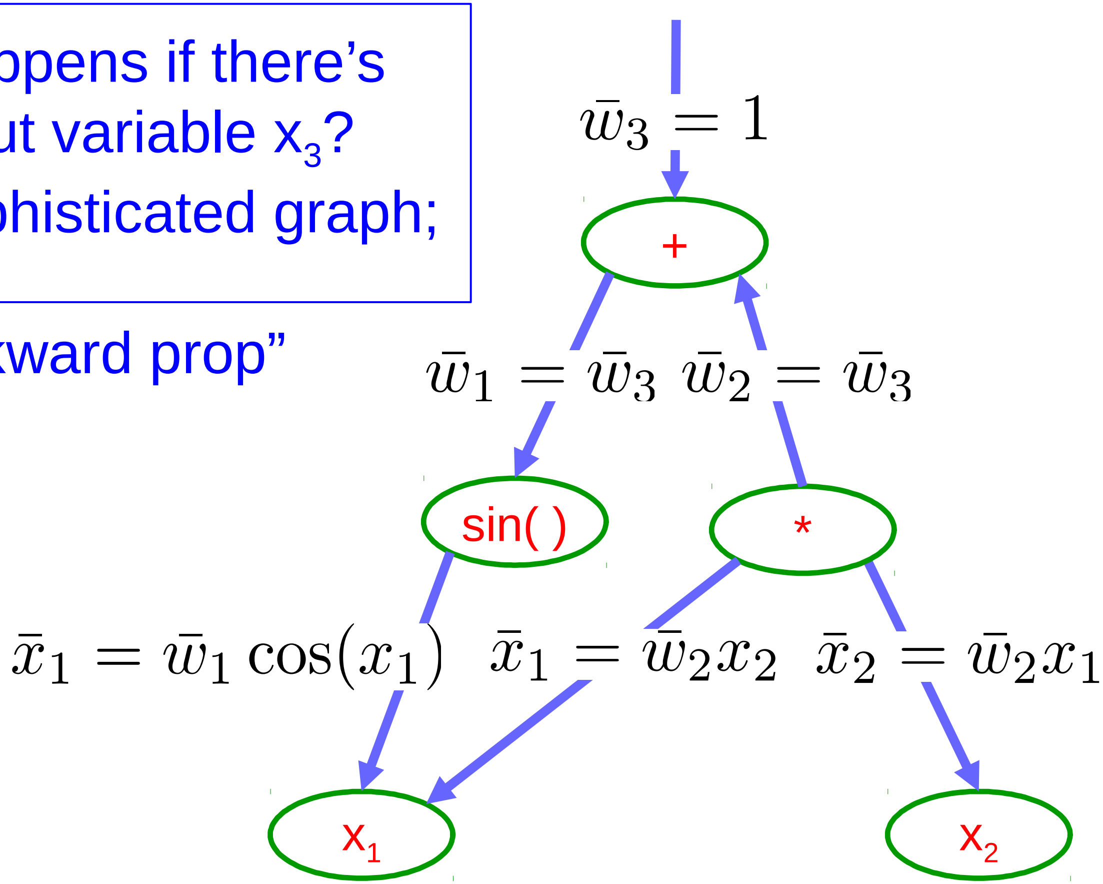
Q: What happens if there's another input variable x_3?



Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

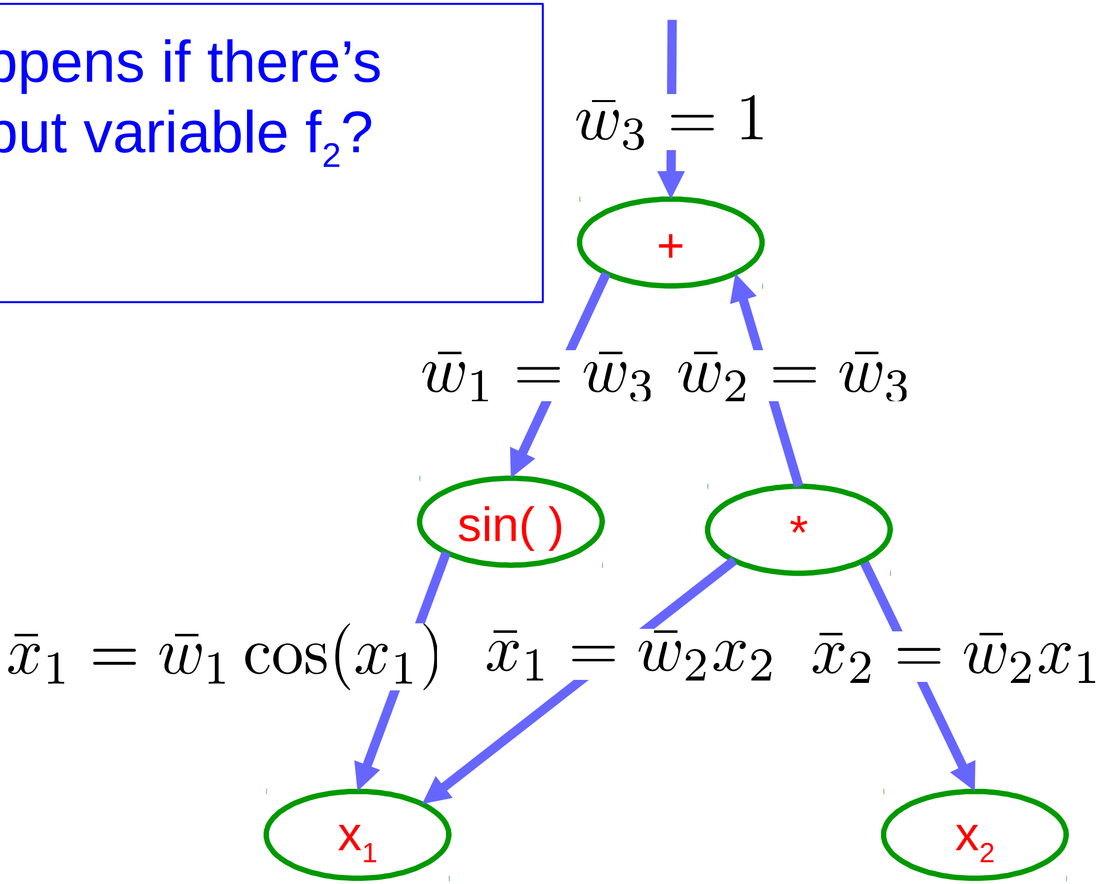
Q: What happens if there's another input variable x_3?
A: more sophisticated graph; single "backward prop"



Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

Q: What happens if there's another output variable f_2?

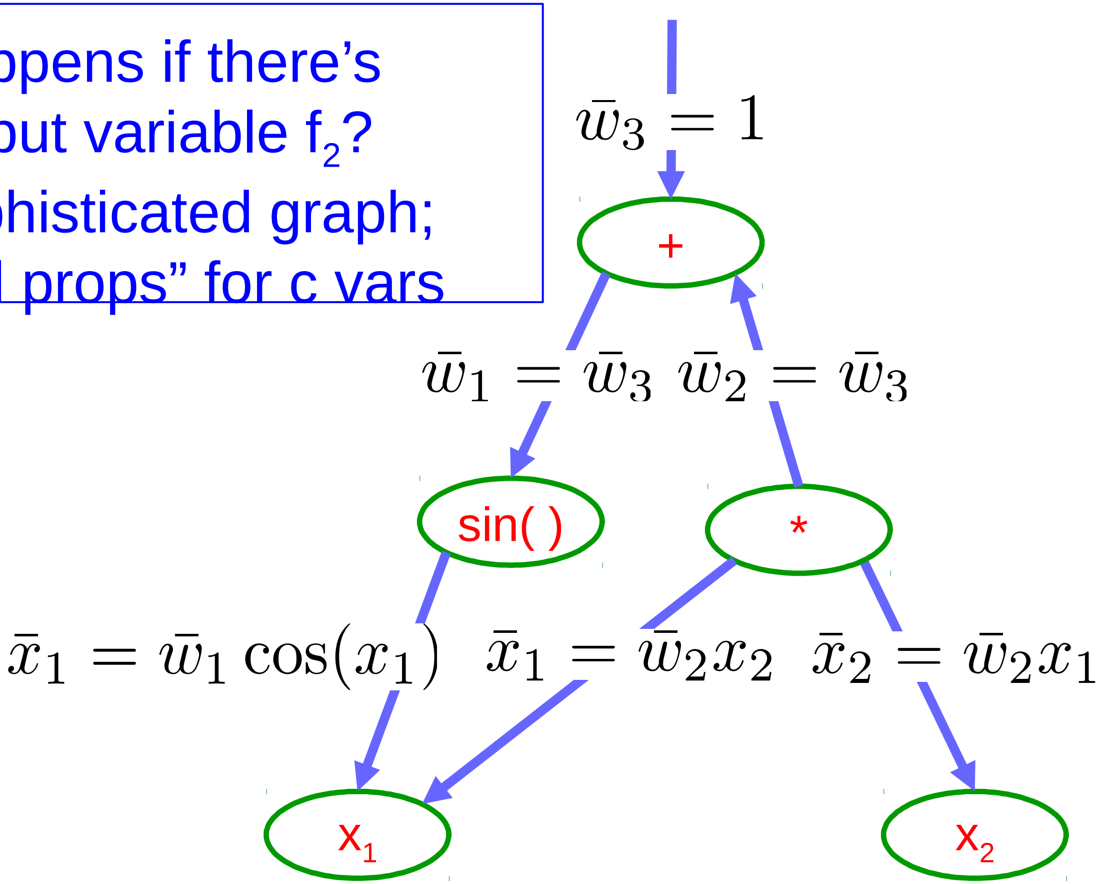


Example: Reverse mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

Q: What happens if there's another output variable f_2?

A: more sophisticated graph; c "backward props" for c vars

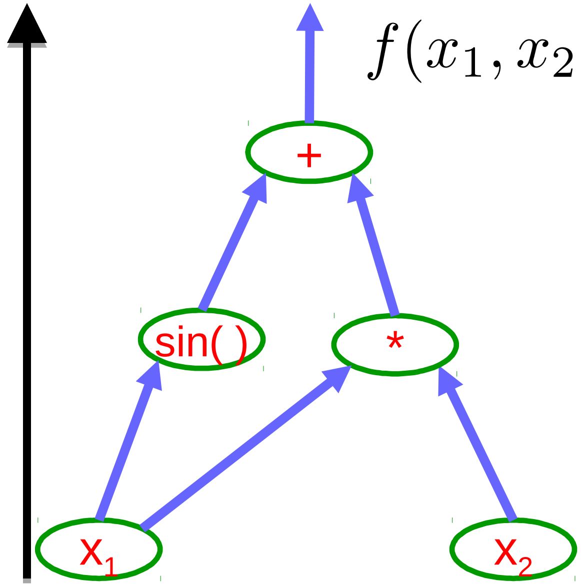


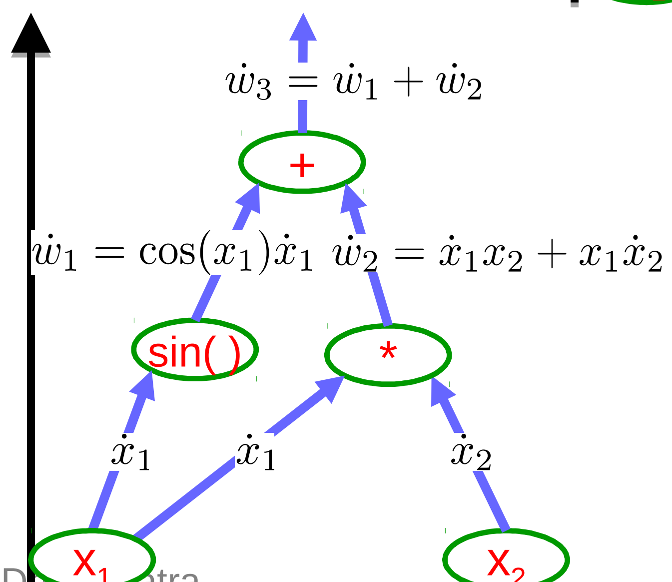
Forward mode vs Reverse Mode

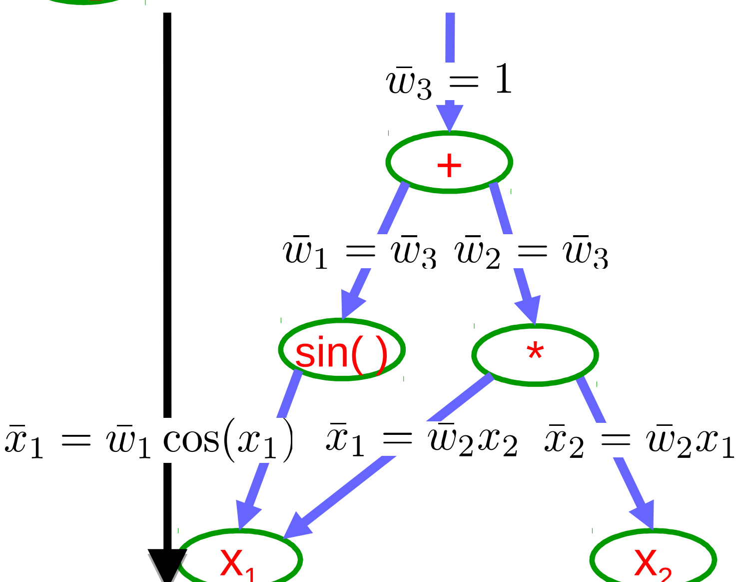
- x ✉ Graph ✉ L
- Intuition of Jacobian

Forward Pass vs
Forward mode AD vs Reverse Mode AD

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

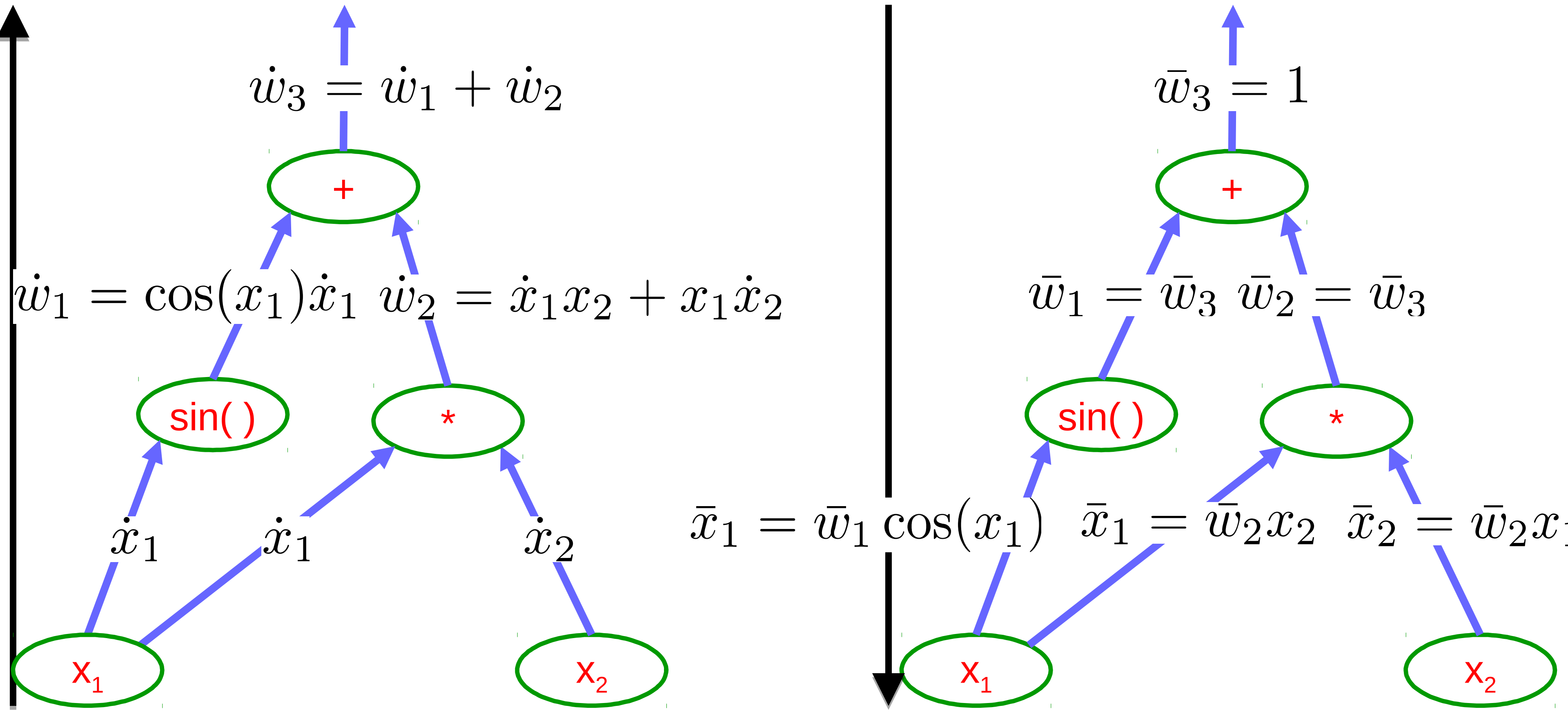






Forward mode vs Reverse Mode

- What are the differences?



Forward mode vs Reverse Mode

- What are the differences?

- Which one is faster to compute?
  - Forward or backward?

Forward mode vs Reverse Mode

- What are the differences?

- Which one is faster to compute?
  - Forward or backward?

- Which one is more memory efficient (less storage)?
  - Forward or backward?