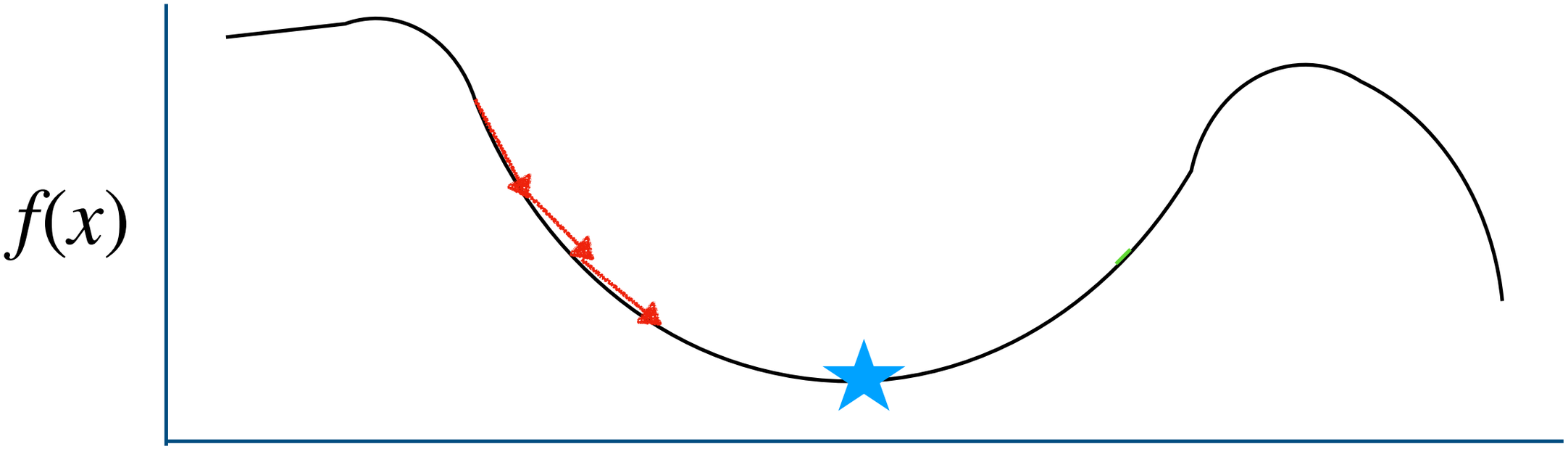
Lecture 2
Backpropagation and Automatic Differentiation

Gradient Descent

$$\min_{\mathbf{x}} f(\mathbf{x})$$



Gradient

Negative Gradient gives the direction of steepest descent

$$f : \mathscr{R}^D -> \mathscr{R} \qquad \nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_D} \end{bmatrix}$$

$$\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{z}) = \begin{bmatrix} \frac{\partial f(\mathbf{x}, \mathbf{z})}{\partial \mathbf{x}_i} \\ \vdots \\ \frac{\partial f(\mathbf{x}, \mathbf{z})}{\partial \mathbf{x}_D} \end{bmatrix}$$

Gradient Descent

$$\min_{\mathbf{x}} f(\mathbf{x})$$

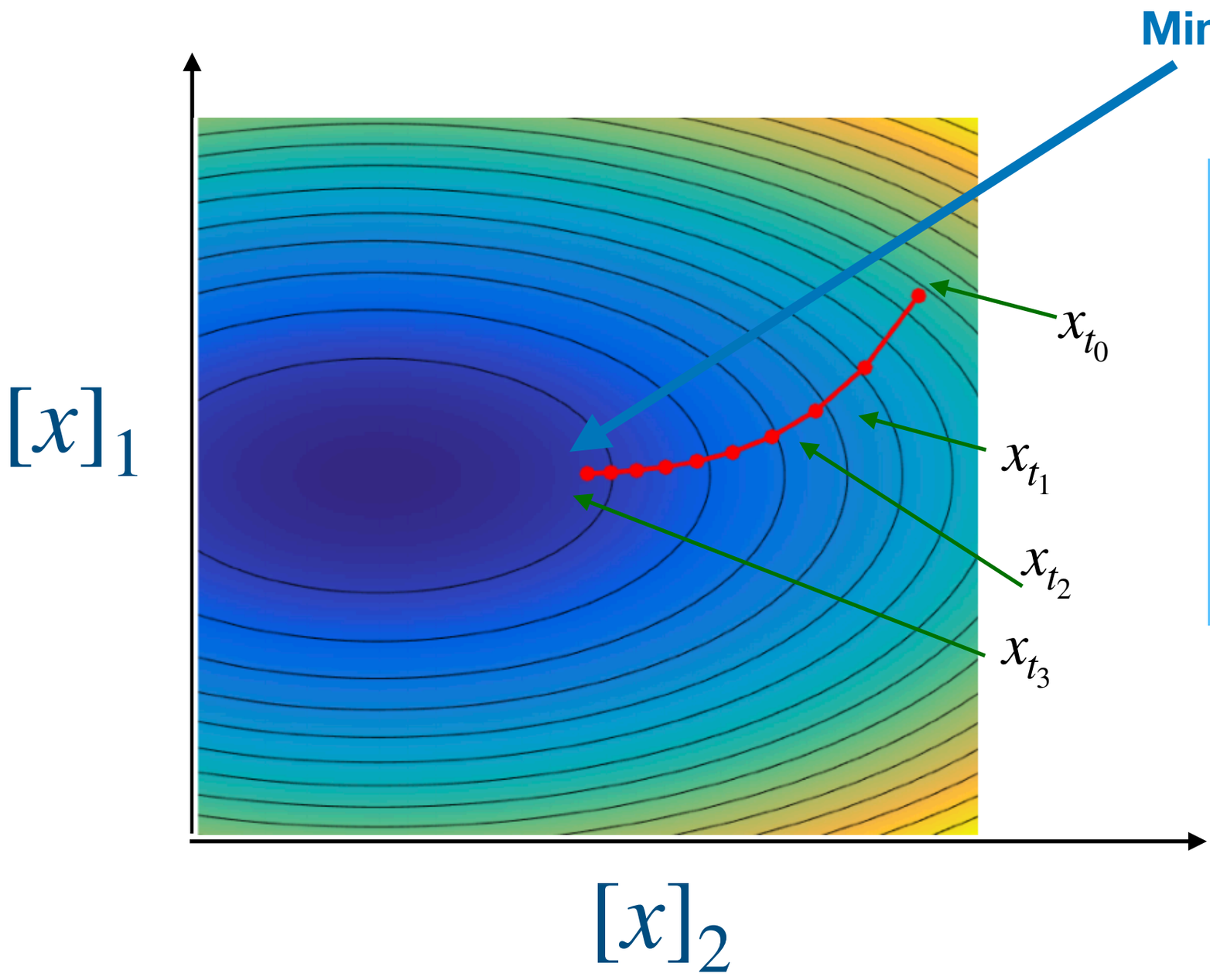
Initialize $\mathbf{x}_0$

Iterate $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t)$

Stopping Crit. $|f(\mathbf{x}_{t+1}) - f(\mathbf{x}_t)| < tolerance$

Gradient Descent in 2D

Visualization in 2 dimensions using contours

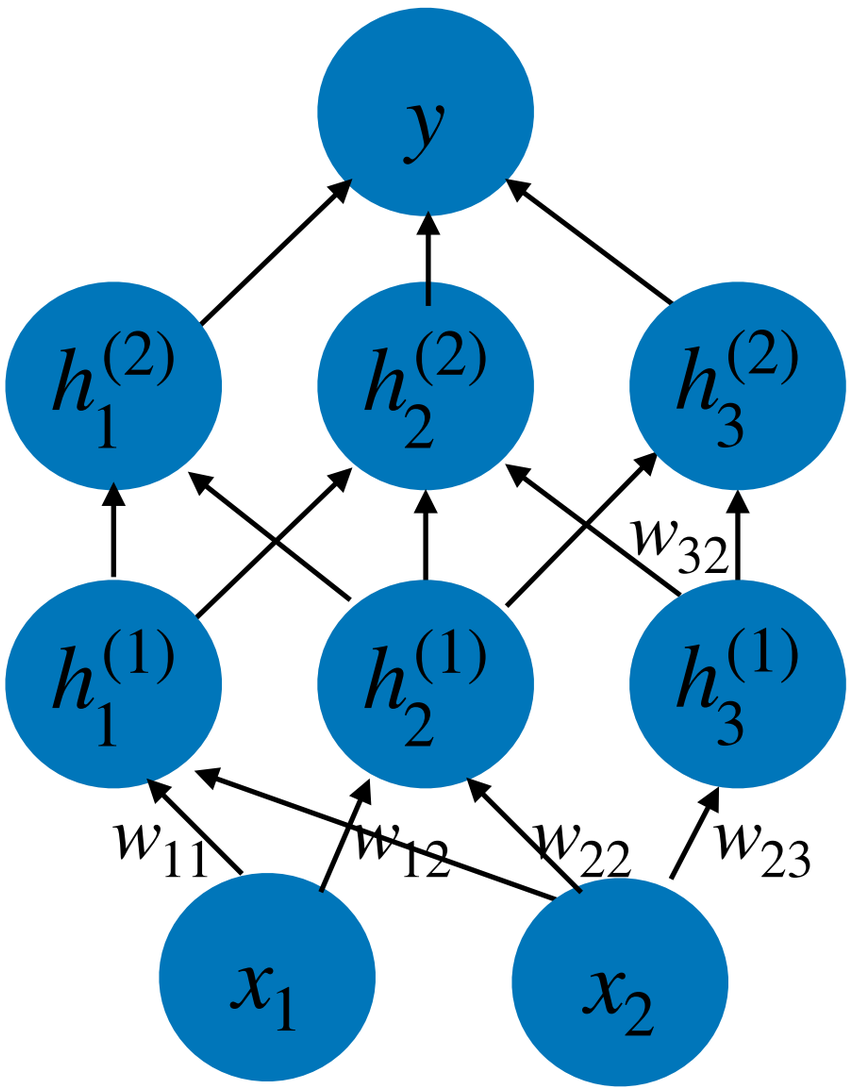


Initialize $\mathbf{x}_0$

Iterate $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t)$

Stopping Crit. $|f(\mathbf{x}_{t+1}) - f(\mathbf{x}_t)| < \epsilon$

Gradient Descent for NN



W_3

W_2

W_1

$$f_{W_1,W_2,W_3}(x) = W_3\rho(W_2\rho(W_1x))$$

W_i **Matrix of parameters at layer I**

ρ **Pointwise non-lineartiy**

X **Data Matrix N samples x 2 features**

$$w = [flat(W_1), flat(W_2), flat(W_3)]$$

All Parameters (flattend)

$$f_w(x)$$

Gradient Descent for NN

Empiricial Risk Minimization $$w^* = \arg\min_w \frac{1}{n} \sum_{i=1}^{n} l(f_w(x_i), y_i)$$

e.g. $$l(f_w(x), y) = \frac{1}{2}(f_w(x) - y)^2$$

$$\mathscr{L}(X, Y, w) = \frac{1}{n} \sum_{i=1}^{n} l(f_w(x), y) = \frac{1}{2n} \|Y - f_w(X)\|^2$$

Gradient of objective respect to weights

$$\nabla_w \mathscr{L}(X, Y, w)$$

w **All Parameters of Model**

X, Y **Data Matrix and Labels**

Gradient based learning

Gradient Descent (GD)

Gradient of full objective

$$\nabla_w \mathscr{L}(X, Y, w)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_w \mathscr{L}(\mathbf{X}, \mathbf{Y}, \mathbf{w}_t)$$

Stochastic GD (SGD)

Gradient of loss w.r.t 1 sample

$$\nabla_w l(x, y, w)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_w l(\mathbf{x}, y, \mathbf{w}_t)$$

Mini-batch SGD

Gradient of loss w.r.t sub-sample

$$X_n \subset X$$

$$\nabla_w \mathscr{L}(X_n, Y_n, w)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_w \mathscr{L}(\mathbf{X}_n, \mathbf{Y}_n, \mathbf{w}_t)$$

Gradient-Based Optimization in ML

- Gradient-based optimization are critical in machine learning and especially in deep learning

- Deriving gradients becomes tedious as the number of components and their complexity grows $f_{W_1,W_2,W_3}(x) = W_3\rho(W_2\rho(W_1x))$

- Changes to the model require re-deriving gradients

Computing the Gradient

$$\nabla_W \mathscr{L}(X, Y, w) = \begin{bmatrix} \dfrac{\partial \mathscr{L}(X, Y, w)}{\partial w_{11}^1} \\ \vdots \\ \dfrac{\partial \mathscr{L}(X, Y, w)}{\partial w_{KJ}^I} \end{bmatrix}$$

- Finite differences

$$[\nabla_W \mathscr{L}(\mathbf{X}, \mathbf{Y}, \mathbf{w})]_1 =$$

$$\frac{\partial \mathscr{L}(\mathbf{X}, \mathbf{Y}, \mathbf{w})}{\partial w_{11}^1} \approx \frac{\mathscr{L}(X, Y, w_{11}^1 + \epsilon, \ldots, w_{kj}^i, \ldots, w_{KJ}^I) - \mathscr{L}(X, Y, w_{11}^1 - \epsilon, \ldots, w_{kj}^i, \ldots, w_{KJ}^I)}{2\epsilon}$$

What's wrong with this method of estimating the gradient

Speed of Finite Difference

$$\frac{\partial \mathscr{L}(\mathbf{X}, \mathbf{Y}, \mathbf{w})}{\partial w_{11}^1} \approx \frac{\mathscr{L}(X, Y, w_{11}^1 + \epsilon, \ldots, w_{kj}^i, \ldots, w_{KJ}^I) - \mathscr{L}(X, Y, w_{11}^1 - \epsilon, \ldots, w_{kj}^i, \ldots, w_{KJ}^I)}{2\epsilon}$$

- Requires 2 forward for each component i of $\nabla_w \mathscr{L}(\mathbf{X}, \mathbf{Y}, \mathbf{w})$

For d parameters $2d$ forward passes (calls to the objective func)

Automatic Differentiation

- Automatic differentiation is a general term for a system that computes the gradients without needing closed form expressions

- Backpropagation is largely synonymous with a specific form of reverse mode auto differentiation

Chain Rule

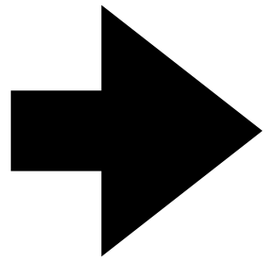
- Consider $z(x) = f(g(x))$ $\quad f, g : \mathcal{R} \rightarrow \mathcal{R}$

- The chain rule in one dimension

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \qquad y = g(x) \text{ and } z = f(y)$$

Chain Rule Example

$$z(x) = log(x)^2$$
$$h(x) = f(g(x))$$



$$g(x) = log(x)$$
$$f(y) = y^2$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = (2 * \log(x)) * (\frac{1}{x}) = \frac{2\log(x)}{x}$$

Automatic Differentiation

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

Simplified expression

$$\frac{\partial z}{\partial x} = \frac{2\log(x)}{x}$$

Procedural

1. $$\frac{\partial z}{\partial y} = 2 * \log(x)$$

2. $$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}(\frac{1}{x})$$

Multivariable Calculus Review

- Gradient

- Jacobian

Multivariable Calculus Review

- Gradient — when vector input and scalar output $f : \mathscr{R}^D -> \mathscr{R}$

$$\nabla f(\mathbf{w}) = \begin{bmatrix} \frac{\partial f(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial f(\mathbf{w})}{\partial w_D} \end{bmatrix} = \left(\frac{\partial f}{\partial \mathbf{w}} \right)^T$$

- Jacobian — vector input and vector output $f : \mathscr{R}^D -> \mathscr{R}^M$

$$J_g(w) = \frac{\partial f(\mathbf{w})}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial f_1}{\partial w_1} \cdots \frac{\partial f_1}{\partial w_D} \\ \vdots \\ \frac{\partial f_M}{\partial w_1} \cdots \frac{\partial f_M}{\partial w_D} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{w})^T \\ \vdots \\ \nabla f_M(\mathbf{w})^T \end{bmatrix}$$

Multivariable chain rule warm up

Function of two variables $h(g(x), f(x))$

$$\frac{\partial h(g(x), f(x))}{\partial x} = \frac{\partial h}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial h}{\partial f}\frac{\partial f}{\partial x}$$

Chain Rule Vector Valued f

- Consider $f(g(\mathbf{x}))$

$$\mathbf{x} \in R^n \ , \ g : \mathscr{R}^n \rightarrow \mathscr{R}^m, \ f : \mathscr{R}^m \rightarrow \mathscr{R}$$

- The chain rule in multiple dimension $y = g(\mathbf{x})$

$$\frac{\partial f}{\partial x_i} = \sum_j \frac{\partial f}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad \rightarrow (\nabla_{\mathbf{y}} f)^T \begin{bmatrix} \frac{\partial g_1}{\partial x_j} \\ \vdots \\ \frac{\partial g_M}{\partial x_j} \end{bmatrix}$$

$$\nabla_{\mathbf{x}} f(\mathbf{x})^T = \nabla_{\mathbf{y}} f(\mathbf{y})^T \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

Jacobian

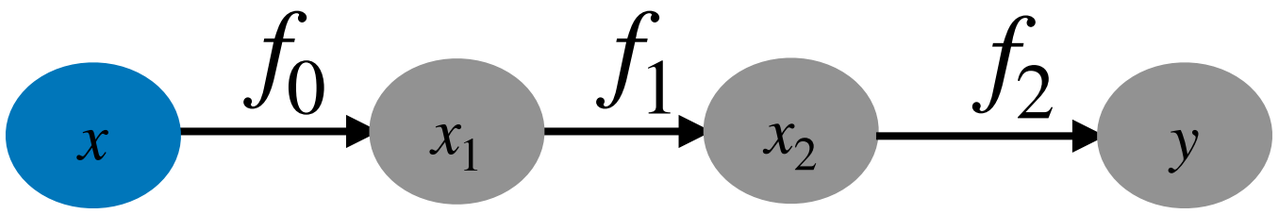
Vector - Jacobian product

Computation Graphs

$\mathbf{x} \in R^n$, $f : \mathscr{R}^m \rightarrow \mathscr{R}$

$$f(\mathbf{x}) = f_2(f_1(f_0(\mathbf{x})))$$

$$x_1 = f_0(\mathbf{x}_0)$$
$$x_2 = f_1(\mathbf{x}_1)$$
$$y = f_2(\mathbf{x}_2)$$



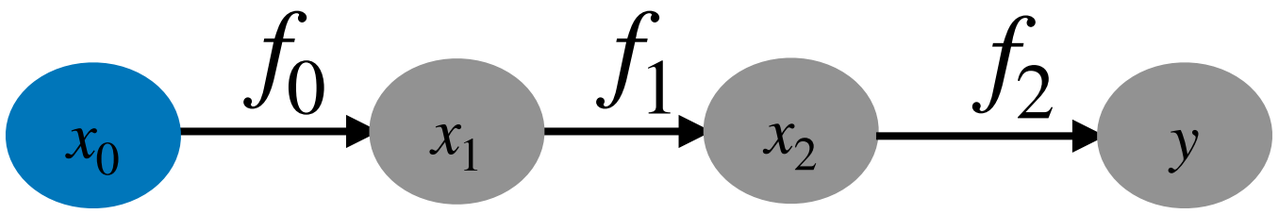
- Nodes are input or computed variables
- Non-leaf nodes are obtained by operations dependent only on parent nodes
- Note several valid alternative ways to formalize computation graphs exist

Computation Graphs

$$\mathbf{x}_0 \in R^n \ , f : \mathscr{R}^m \rightarrow \mathscr{R}$$

$$f(\mathbf{x}_0) = f_2(f_1(f_0(\mathbf{x}_0)))$$

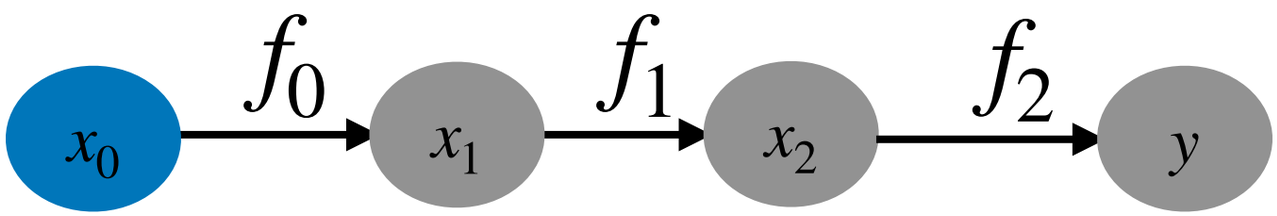
$$x_1 = f_0(\mathbf{x}_0)$$
$$x_2 = f_1(\mathbf{x}_1)$$
$$y = f_2(\mathbf{x}_2)$$



$$\frac{\partial y}{\partial \mathbf{x}_0} = \frac{\partial y}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0} = \frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2}\frac{\partial f_1(\mathbf{x}_1)}{\partial \mathbf{x}_1}\frac{\partial f_0(\mathbf{x}_0)}{\partial \mathbf{x}_0}$$

$1 \times M_3$ $M_3 \times M_2$ $M_2 \times M_1$

Forward and Backward Differentiation



$$\frac{\partial y}{\partial \mathbf{x}_0} = \frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2} \frac{\partial f_1(\mathbf{x}_1)}{\partial \mathbf{x}_1} \frac{\partial f_0(\mathbf{x}_0)}{\partial \mathbf{x}_0}$$

$$1 \times M_3 \quad M_3 \times M_2 \quad M_2 \times M_1$$

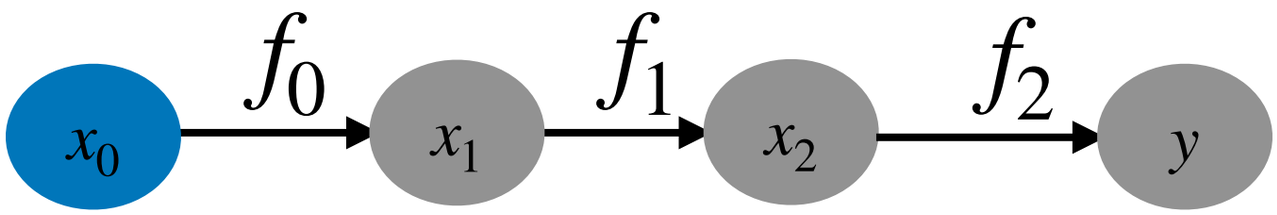
Take $M = M_3 = M_2 = M_1$

Forward Mode AutoDiff ← $M^3 + M^2$ **Ops** $O(M^3)$

Reverse Mode AutoDiff (Backprop) → $M^2 + M^2$ **Ops** $O(M^2)$

Multiply this way

Reverse Mode AD



Forward Pass

$$x_1 = f_0(\mathbf{x}_0)$$
$$x_2 = f_1(\mathbf{x}_1)$$
$$y = f_2(\mathbf{x}_2)$$

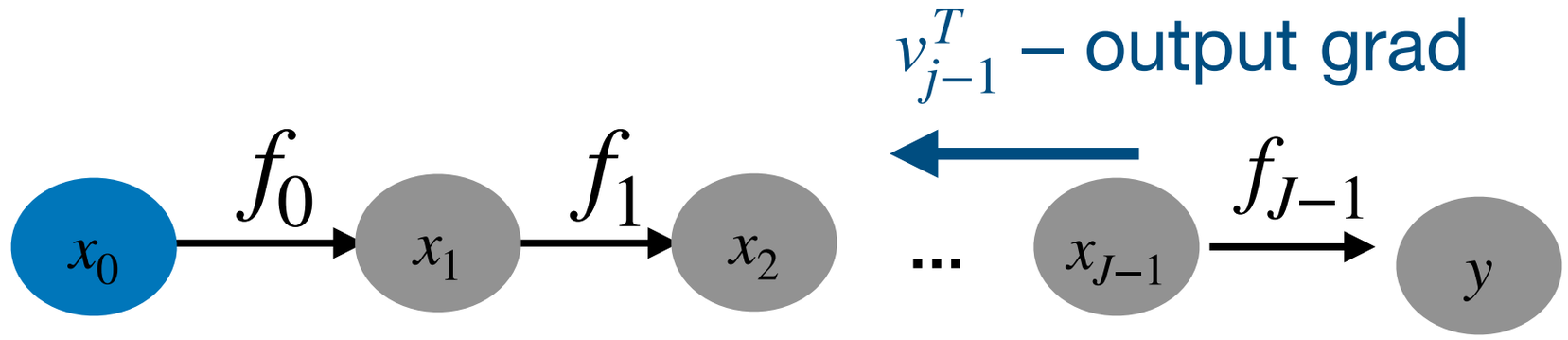
Backward Pass

$$\frac{\partial y}{\partial \mathbf{x}_2} = \frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2}$$

$$\frac{\partial y}{\partial \mathbf{x}_1} = \frac{\partial y}{\partial \mathbf{x}_2}\frac{\partial f_1(\mathbf{x}_1)}{\partial \mathbf{x}_1}$$

$$\frac{\partial y}{\partial \mathbf{x}_0} = \frac{\partial y}{\partial \mathbf{x}_1}\frac{\partial f_0(\mathbf{x}_0)}{\partial \mathbf{x}_0}$$

Reverse Mode AD



Reverse Mode AD for chain graph and scalar output

$x_0 \leftarrow x$

for $j = 0$ to $J - 1$:

Forward Pass

$x_{k+1} \leftarrow f_k(x_k)$

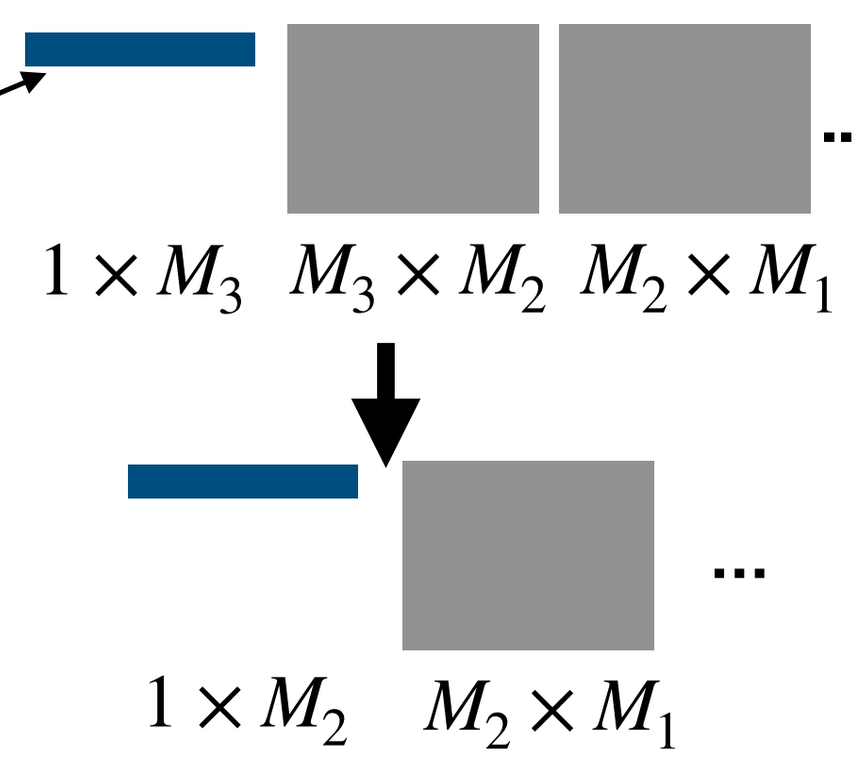
$v_J \leftarrow \nabla f_{J-1}(x_{J-1})$

for $j = J - 1$ to 1 :

Backward Pass

$v_{j-1} \leftarrow v_j^T J_{f_{j-1}}(x_{j-1})$

$\nabla_{x_0} y = v_0^T$



Reverse Mode AD for MLP

Terminology: feedforward networks with fully connected layers -> Multilayer Perceptrons (MLP)

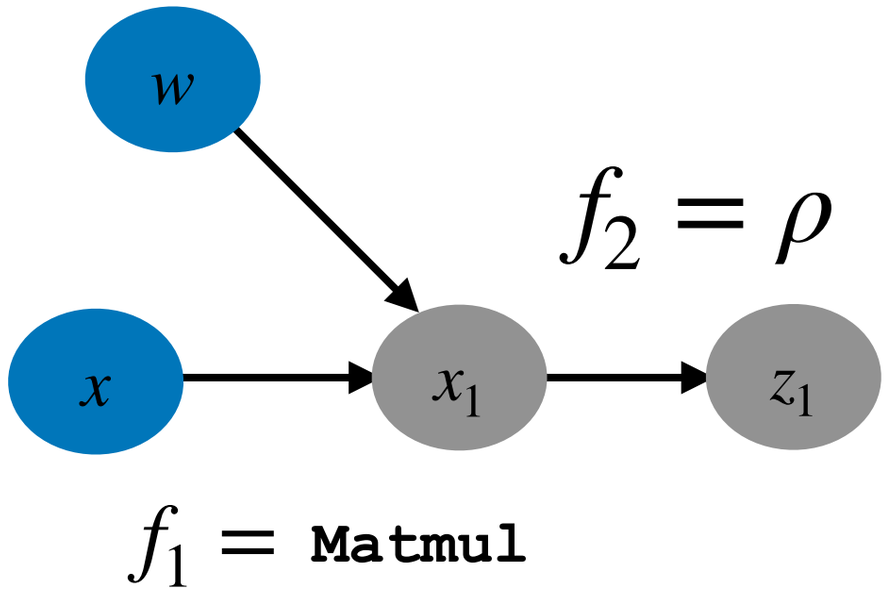
$$f(\mathbf{x}) = f_1(f_0(\mathbf{x}, \mathbf{w}))$$

$$x_1 = f_0(\mathbf{x}_0, \mathbf{w})$$

$$x_2 = f_1(\mathbf{x}_1)$$

Leaf nodes

Non-Leaf nodes



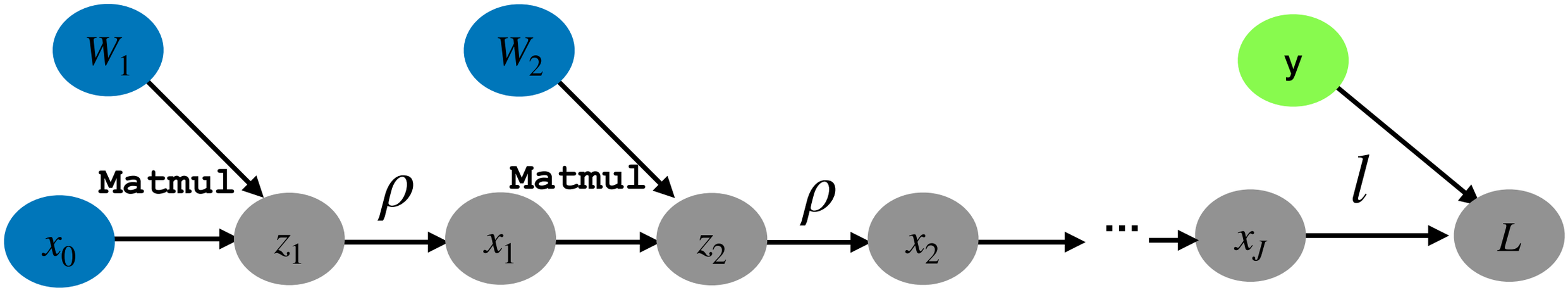
Reverse Mode AD for MLP

$$f_{W_1,W_2,W_3,\cdots,W_J}(x) = \rho W_J \cdots W_3 \rho(W_2 \rho(W_1 x))$$

x_{i-1} **Input layer i**

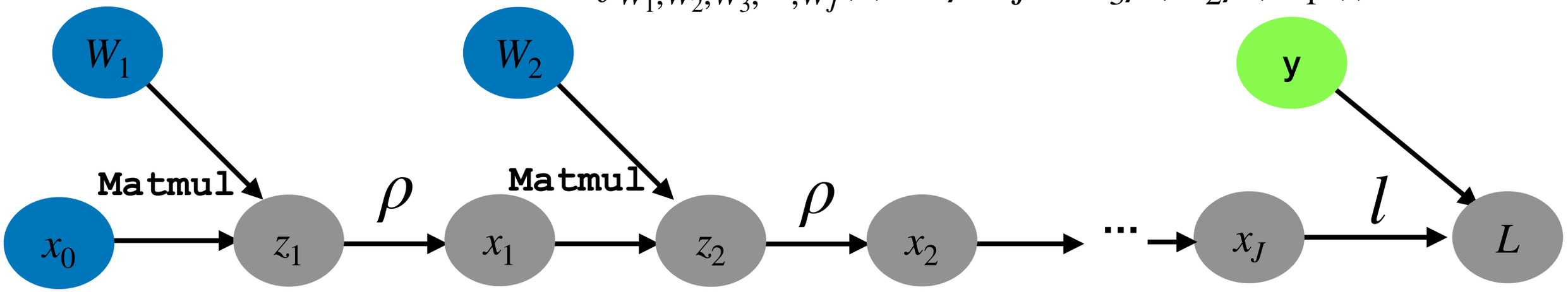
z_i **Pre-activation**

x_i **Post-activation**



Reverse Mode AD for MLP

$$f_{W_1,W_2,W_3,\cdots,W_J}(x) = \rho W_J \cdots W_3 \rho(W_2 \rho(W_1 x))$$



We want $\nabla_{W_J} L, \cdots \nabla_{W_1} L$

x_{i-1}	**Input layer i**
z_i	**Pre-activation**
x_i	**Post-activation**

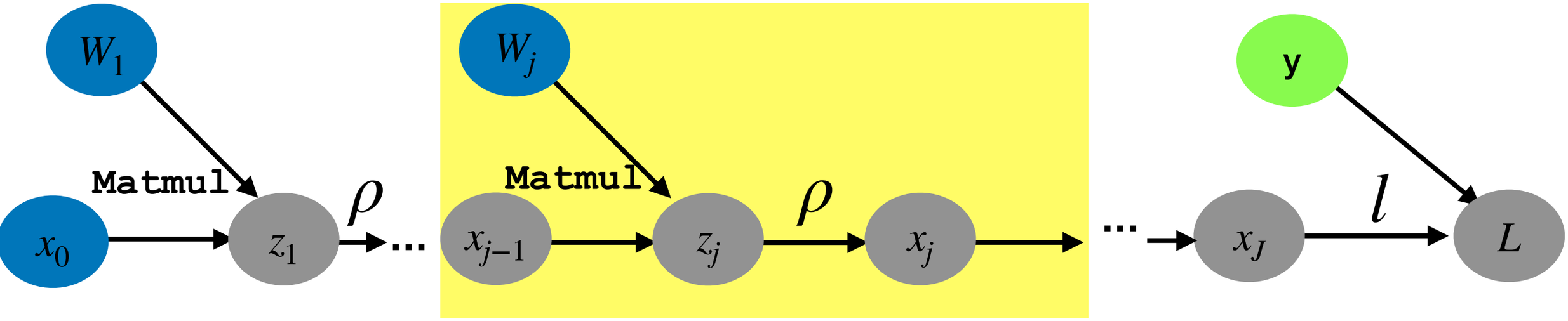
Forward Pass

$x_0 \leftarrow x$
for $j = 0$ to $J-1$:
$z_{j+1} \leftarrow W_{j+1} x_j$
$x_{j+1} \leftarrow \rho(z_{j+1})$
$L = l(x_J, y)$

$$\frac{\partial \mathbf{z}_{j+1}}{\partial \mathbf{x}_j} = W_{j+1} \qquad \frac{\partial \mathbf{z}_j}{\partial \mathbf{W}_j} = ?$$

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{z}_j} = diag(\rho'(z_j))$$

Reverse Mode AD for MLP



$$f_{W_1,W_2,W_3,\cdots,W_J}(x) = \rho W_J \ldots W_3 \rho(W_2 \rho(W_1 x))$$

We want $\nabla_{W_J} L, \cdots \nabla_{W_1} L$

Forward Pass

$x_0 \leftarrow x$

for $j = 0$ to $J-1$:

$z_{j+1} \leftarrow W_{j+1} x_j$

$x_{j+1} \leftarrow \rho(z_{j+1})$

$L = l(x_J, y)$

VJPs (Vector Jacobian Products)

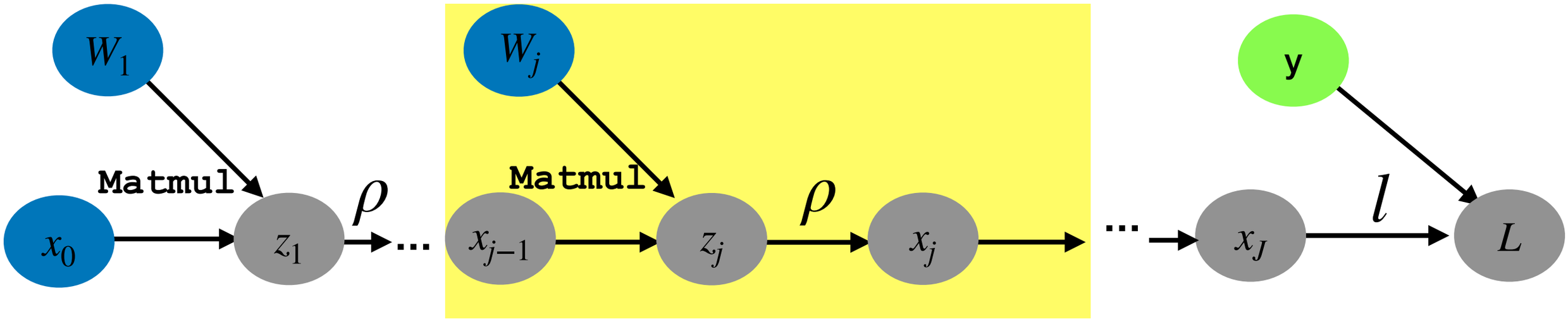
$$\frac{\partial L}{\partial \mathbf{x}_J} = \frac{\partial l(x_J, y)}{\partial \mathbf{x}_J}$$

$$\frac{\partial L}{\partial \mathbf{z}_j} = \frac{\partial L}{\partial \mathbf{x}_j}\frac{\partial \mathbf{x}_j}{\partial \mathbf{z}_j} = \frac{\partial L}{\partial \mathbf{x}_j} diag(\rho'(\mathbf{z}_j)) = \frac{\partial L}{\partial \mathbf{x}_j} \circ \rho'(\mathbf{z}_j)$$

$$\frac{\partial L}{\partial \mathbf{x}_{j-1}} = \frac{\partial L}{\partial \mathbf{z}_j}\frac{\partial \mathbf{z}_j}{\partial \mathbf{x}_{j-1}} = \frac{\partial L}{\partial \mathbf{z}_j}\mathbf{W}_j$$

$$\frac{\partial L}{\partial \mathbf{W}_j} = \frac{\partial L}{\partial \mathbf{z}_j}\frac{\partial \mathbf{z}_j}{\partial \mathbf{W}_j} = (\frac{\partial L}{\partial \mathbf{z}_j})^T \mathbf{x}_{j-1}^T$$

Reverse Mode AD for MLP



We want $\nabla_{W_J} L, \cdots \nabla_{W_1} L$

Forward Pass

$$x_0 \leftarrow x$$
for $j = 0$ to $J-1$:
$$z_{j+1} \leftarrow W_{j+1} x_j$$
$$x_{j+1} \leftarrow \rho(z_{j+1})$$
$$J = l(x_J, y)$$

Backward Pass

$$v = \nabla_{x_J} L = \nabla_{x_J} l(x_J, y)$$
for $j = J-1$ to 1 :
$$v \leftarrow \nabla_{z_j} L = v \circ \rho'(z_j)$$
$$\nabla_{W_j} L = v x_{j+1}^T$$
$$v \leftarrow \nabla_{x_j} L = W_j^T v$$

Note: here we keep everything as column vectors

Speed for MLP

Forward Pass

$$x_0 \leftarrow x$$
$$\text{for } j = 0 \text{ to } J-1:$$
$$\boxed{z_{j+1} \leftarrow W_{j+1} x_j}$$
$$x_{j+1} \leftarrow \rho(z_{j+1})$$
$$L = l(x_J, y)$$

Backward Pass

$$v = \nabla_{x_J} L = \nabla_{x_J} l(x_J, y)$$
$$\text{for } j = J-1 \text{ to } 1:$$
$$v \leftarrow \nabla_{z_j} L = v \circ \rho'(z_j)$$
$$v \leftarrow \boxed{\begin{array}{l} \nabla_{W_j} L = v x_{j+1}^T \\ \nabla_{x_j} L = W_j^T v \end{array}}$$

- Finite difference requires 2*D forward passes, with D parameters
- Reverse Mode AD, often ~ 2x forward pass
- Forward Mode AD speed / forward pass would increase with width

Group Activity

- In a group work through the following - 20 minutes

- Consider the following: $y = \mathbf{w}_2^T \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b})$

  - Draw the computation graph

    - Nodes as input or computed variables

  - Find systematically the expressions for $\frac{\partial y}{\partial \mathbf{w}_2}, \frac{\partial y}{\partial \mathbf{W}_1}, \frac{\partial y}{\partial \mathbf{b}}$

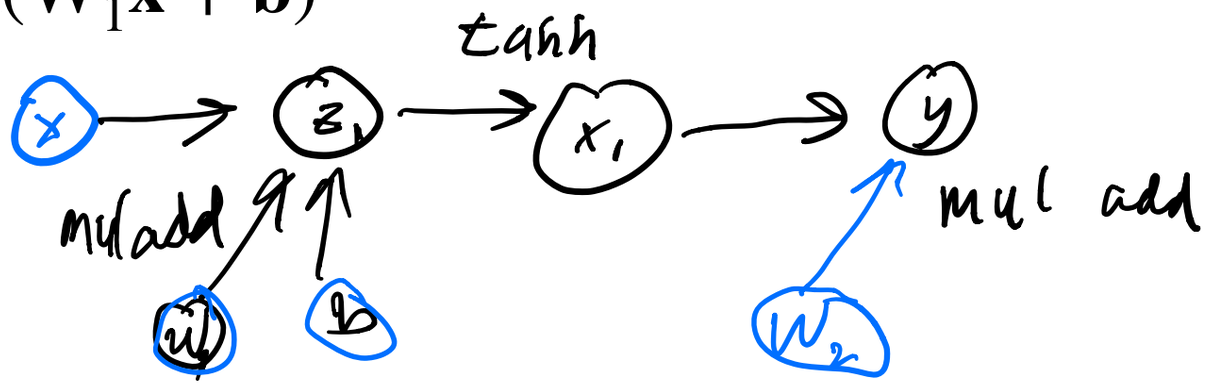
$$\mathbf{W}_1 = \begin{bmatrix} -1 & 1 \\ 0.5 & 0.5 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note

$$\tanh'(x) = 1 - \tanh^2(x)$$

Problem

$$y = \mathbf{w}_2^T \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b})$$



Note: there are different valid graphs depending how you define the primitive ops

$y = w_2^T x_1$

$y = w_2^T x_1$

$x_1 = \tanh(z_1)$

$z_1 = W_1 x + b$

$\frac{\partial y}{\partial w_2} = x_1^T = \tanh(W_1 x + b)^T$

$\frac{\partial y}{\partial x_1} = w_2^T$

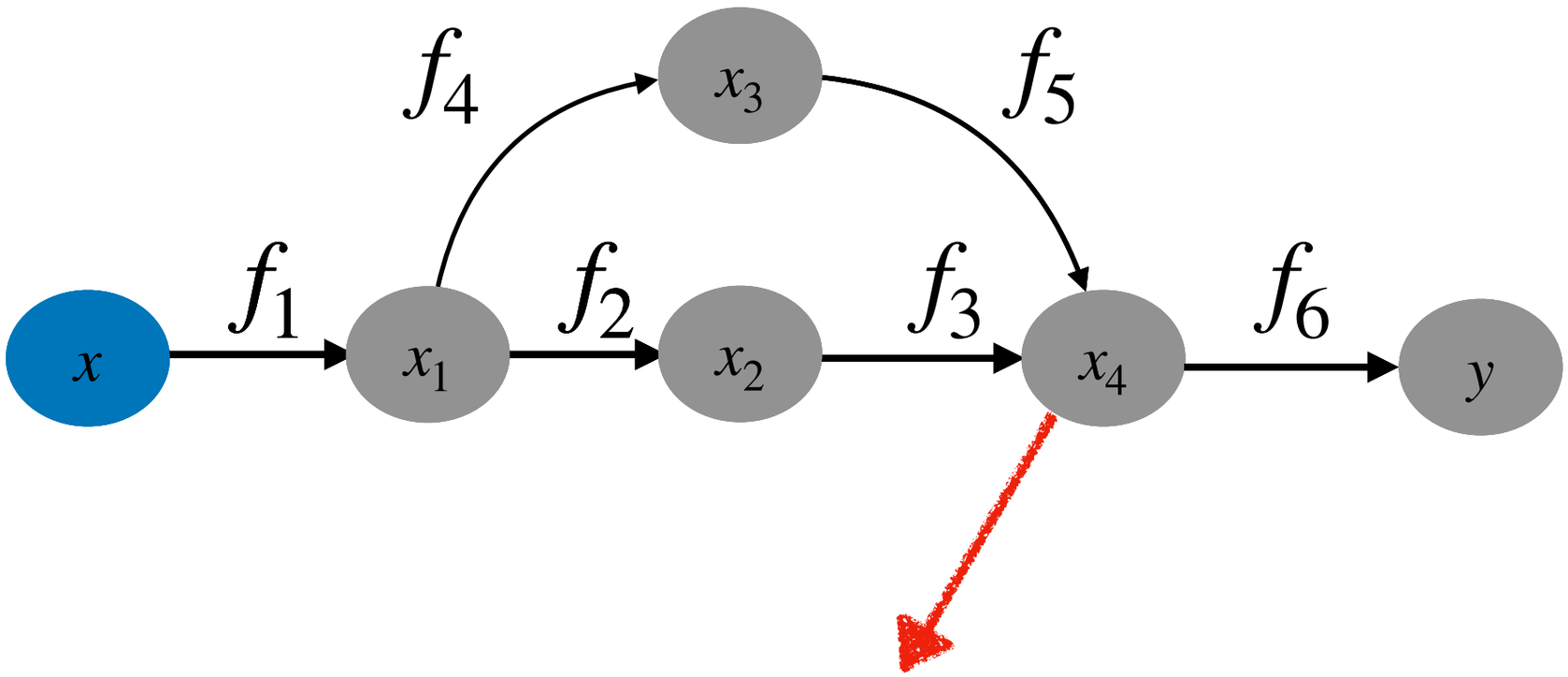
$\frac{\partial y}{\partial z_1} = w_2^T \operatorname{diag}(\tanh'(z_1)) = w_2^T \odot \tanh'(z_1)^T$

$\frac{\partial y}{\partial W_1} = x^T \frac{\partial y}{\partial z} = x^T w_2^T \odot \tanh'(z_1)^T$

$= x^T w_2^T \odot (1 - \tanh^2(W_1 x + b))^T$

$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial z_1} \cdot 1 = w_2^T \odot (1 - \tanh^2(W_1 x + b))^T$

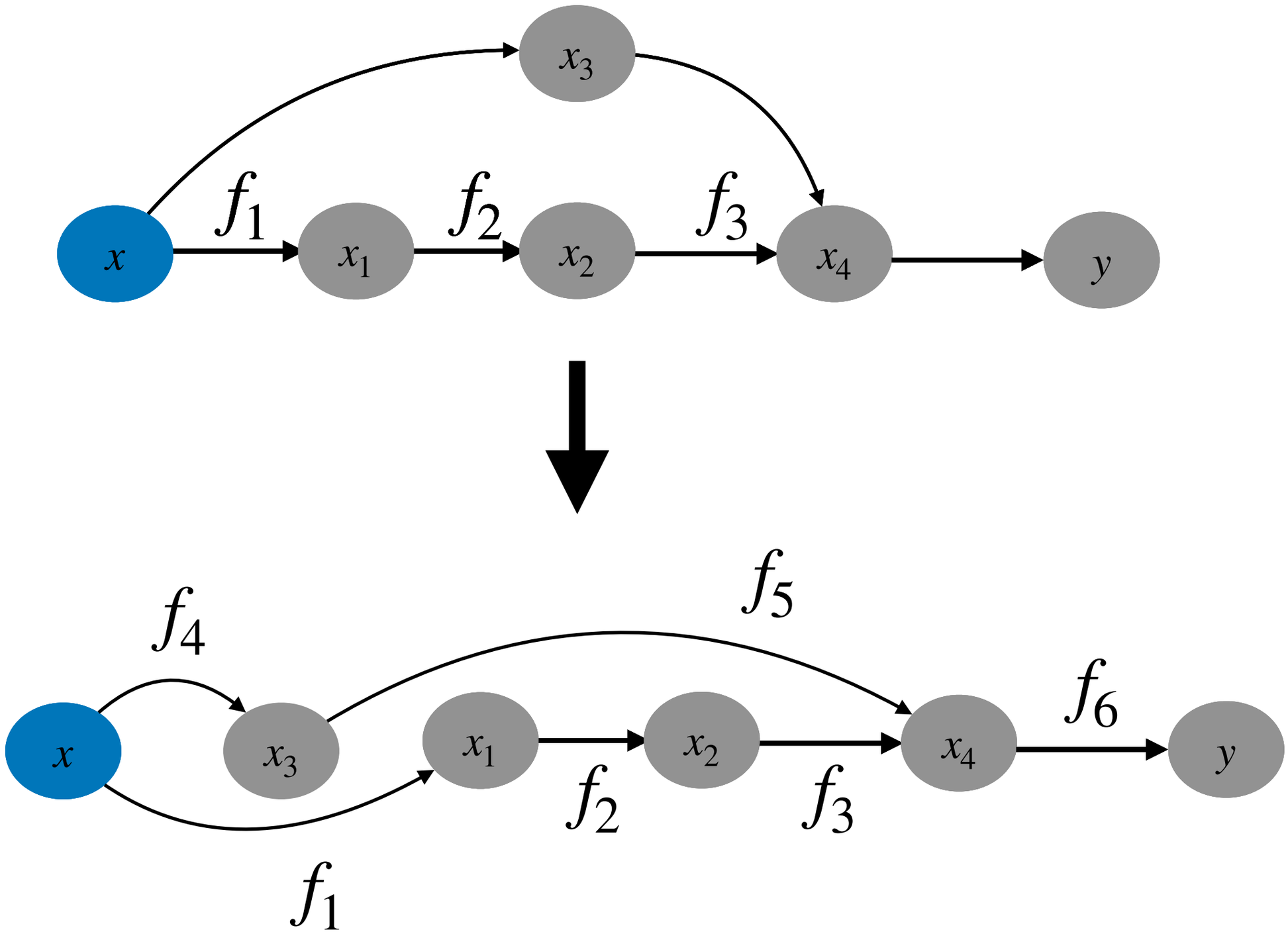
More Complex Graphs



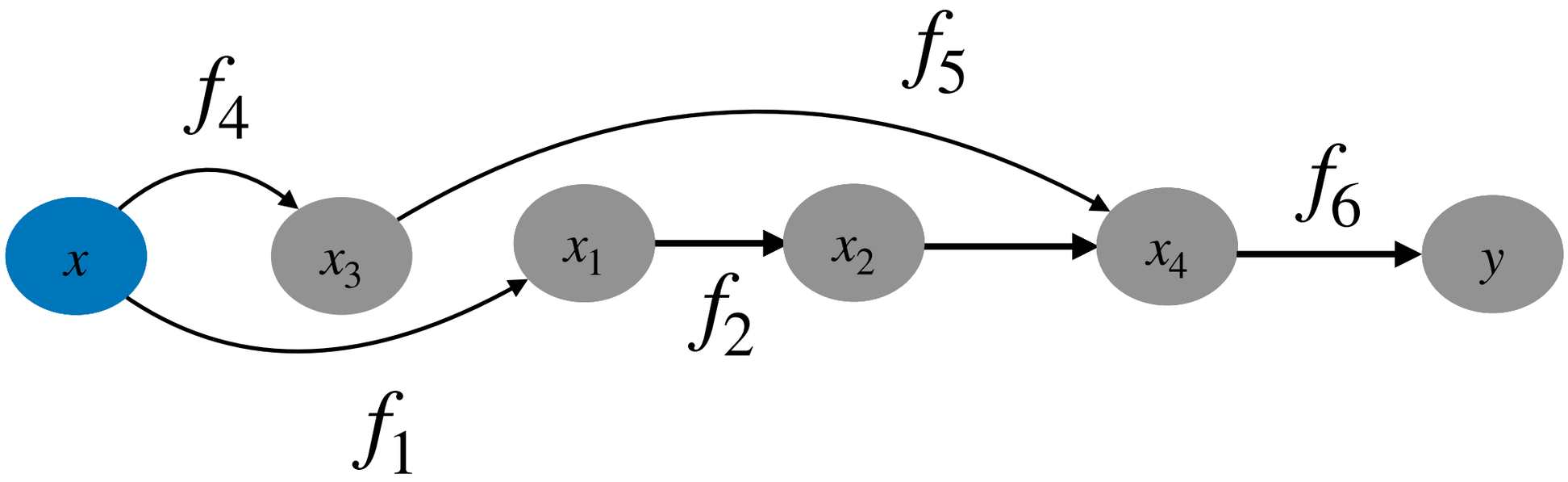
$$\frac{\partial y}{\partial \mathbf{x}_0} = \frac{\partial y}{\partial \mathbf{x}_4}\left(\frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3}\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_1}\right)\frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}$$

$$\frac{\partial y}{\partial \mathbf{x}_j} = \sum_{s \in Child(j)} \frac{\partial y}{\partial \mathbf{x}_s}\frac{\partial \mathbf{x}_s}{\partial \mathbf{x}_j}$$

Topological Sort



Reverse AD over General Graph



For last node $x_J := L$

Forward Pass

$x_1, \cdots, x_J \leftarrow$ topological sort$(Graph)$

for $j = 1$ to J :

$x_i \leftarrow f_i(\text{Parent}_1(x_i), \ldots, \text{Parent}_K(x_i))$

Backward Pass

$\nabla_{x_J} L = \nabla_{x_J} x_J = 1$

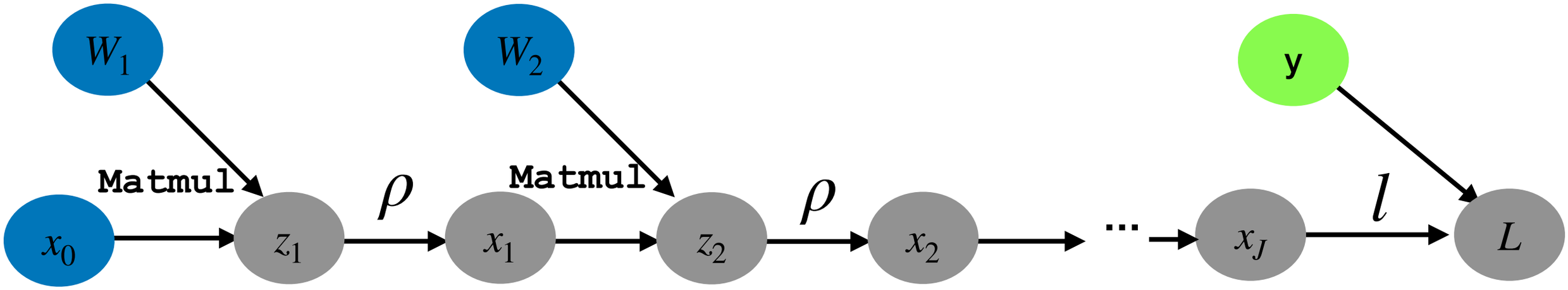
for $j = J - 1$ to 1 :

$$(\nabla_{x_j} L)^T \leftarrow \sum_{k \in Child(x_i)} (\nabla_{x_k} L)^T \frac{\partial x_k}{\partial x_j}$$

Recap of Terminology

- Backpropagation is how we compute the gradient
    - It is not Gradient Descent, which is how we optimize the objective
- Automatic Differentiation
    - More general than backprop
    - Backprop is essentially reverse mode AD for scalar output
- Autograd
    - Specific package implementing Auto Differentiation
    - Predecessor of torch autograd

Vanishing Gradients



$$\frac{\partial L}{\partial \mathbf{x}_1} = (\frac{\partial L}{\partial \mathbf{x}_J})^T \Pi_{j=J..2} diag(\rho'(\mathbf{x}_{j-1}))W_j$$

- Has been observed for feedforward nets that signal degrades with depth
- This makes adapting lower layers difficult
- Dependence on distribution of initial weights
- RNNs (same)

$$\frac{\partial L}{\partial \mathbf{z}_j} = \frac{\partial L}{\partial \mathbf{x}_j} diag(\rho'(\mathbf{z}_j))$$

$$\frac{\partial L}{\partial \mathbf{x}_{j-1}} = \frac{\partial L}{\partial \mathbf{z}_j} \mathbf{W}_j$$

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010.

Sepp Hochreiter Master's Thesis

Vanishing and Exploding Gradients

- For many decades this was believed to be the main issue in training deep networks
- Recurrent networks did not work at all until circa 2014, attributed to vanshining/exploding gradients problem

(Optional for now)

Vanishing and Exploding Gradients

- Various ways to address this
  - Most of these will be discussed in more detail in future lectures
- Gradient clipping for exploding gradients
- Shortcut connections (LSTMs and ResNets)
- Normalization techniques
- Better activation selection
- Initialization
- Trying to keep matrices orthogonal
- Alternatives to gradient based learning (bypass bprop)

(Optional for now)

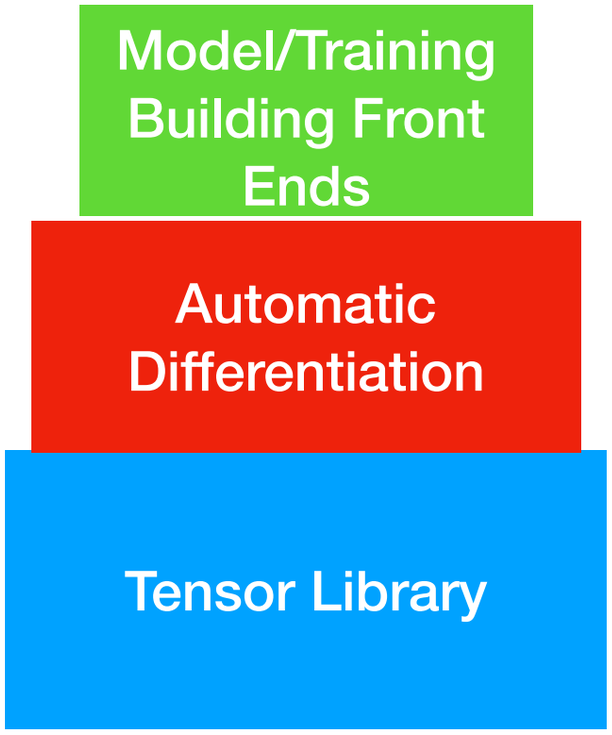
Deep Learning Frameworks

Theano (deprecated)

Tensorflow

MxNet

Pytorch



Deep Learning Frameworks (Pytorch)

Tensor Library

- Built on tensor libraries (similar to numpy)

- Backends to operations on GPU

DL Frameworks

- Built on tensor libraries (similar to numpy)
- Backends to operations on GPU

```
import torch
import math


dtype = torch.float
device = torch.device("cpu")

# Randomly initialize weights
a = torch.randn((5,5), device=device, dtype=dtype)
b = torch.randn((5,5), device=device, dtype=dtype)
```

```
print(a)
```

```
tensor([[-1.7594,  0.9069, -0.9061, -0.2046,  0.2450],
        [ 0.9503, -0.0539, -1.6430,  0.5169, -1.0768],
        [ 0.0174, -1.6540, -0.4759, -0.8193,  1.6534],
        [ 0.0829,  0.8142,  0.5722, -0.7830,  1.7930],
        [-0.5130, -1.2160,  0.9049,  1.1782,  0.3043]])
```

```
torch.matmul(a,b)
```

```
tensor([[ 0.0532,  0.7206, -0.6999, -0.0081,  2.1033],
        [-0.1231,  0.7151, -0.0372, -1.7649,  2.9037],
        [ 3.4586, -0.9548, -2.3225,  0.7618, -1.9983],
        [-0.2156, -0.4302, -0.9443, -0.3182, -0.9874],
        [ 3.4791, -1.0321,  0.9711,  1.1428, -2.7500]])
```

DL Frameworks

```
import torch
import math


dtype = torch.float
device = torch.device("cpu")

# Randomly initialize weights
a = torch.randn((5,5), device=device, dtype=dtype)
b = torch.randn((5,5), device=device, dtype=dtype)
```

- Backends to operations on GPU

```
torch.matmul(a,b)
```

```
tensor([[ 1.0293,  1.8288,  2.4954,  0.6034, -0.1730],
        [-2.0367,  1.1742, -0.6395,  1.0939,  1.1971],
        [ 4.1902,  0.3434,  2.3410, -0.9291,  0.3311],
        [ 0.7546, -2.8378, -0.4807, -4.2653,  2.4706],
        [-0.6787,  0.0684,  0.1628, -2.2533,  0.8050]])
```

```
a = a.cuda(); b = b.cuda()
torch.matmul(a,b)
```

```
tensor([[ 1.0293,  1.8288,  2.4954,  0.6034, -0.1730],
        [-2.0367,  1.1742, -0.6395,  1.0939,  1.1971],
        [ 4.1902,  0.3434,  2.3410, -0.9291,  0.3311],
        [ 0.7546, -2.8378, -0.4807, -4.2653,  2.4706],
        [-0.6787,  0.0684,  0.1628, -2.2533,  0.8050]], device='cuda:0')
```

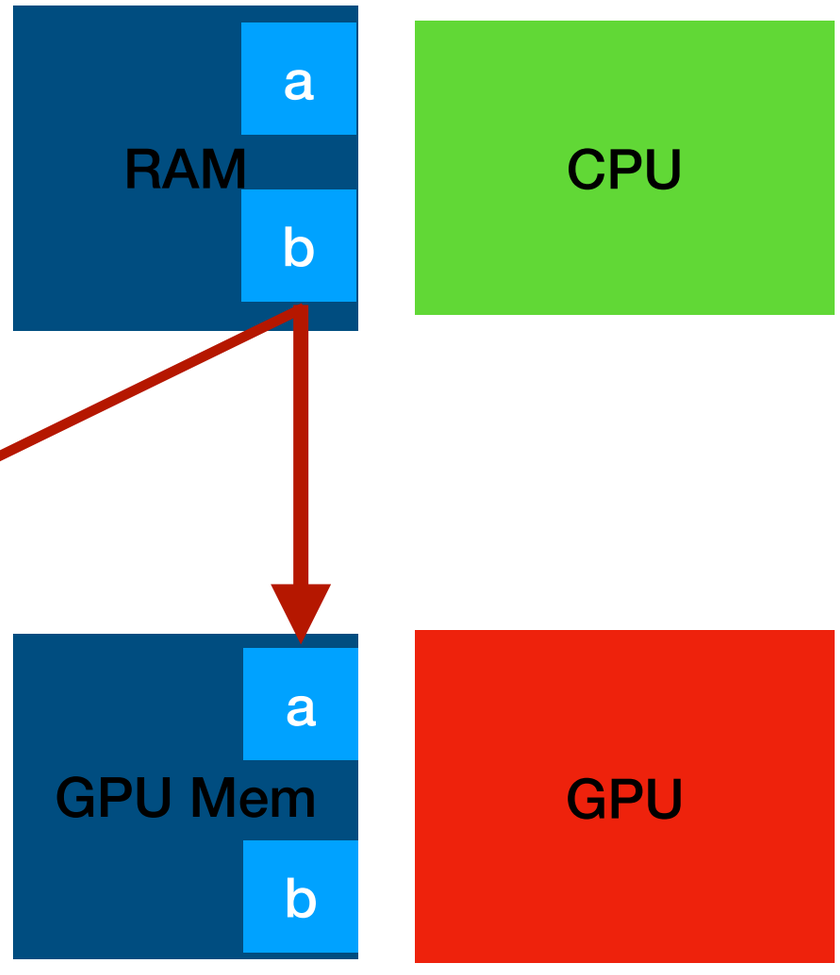
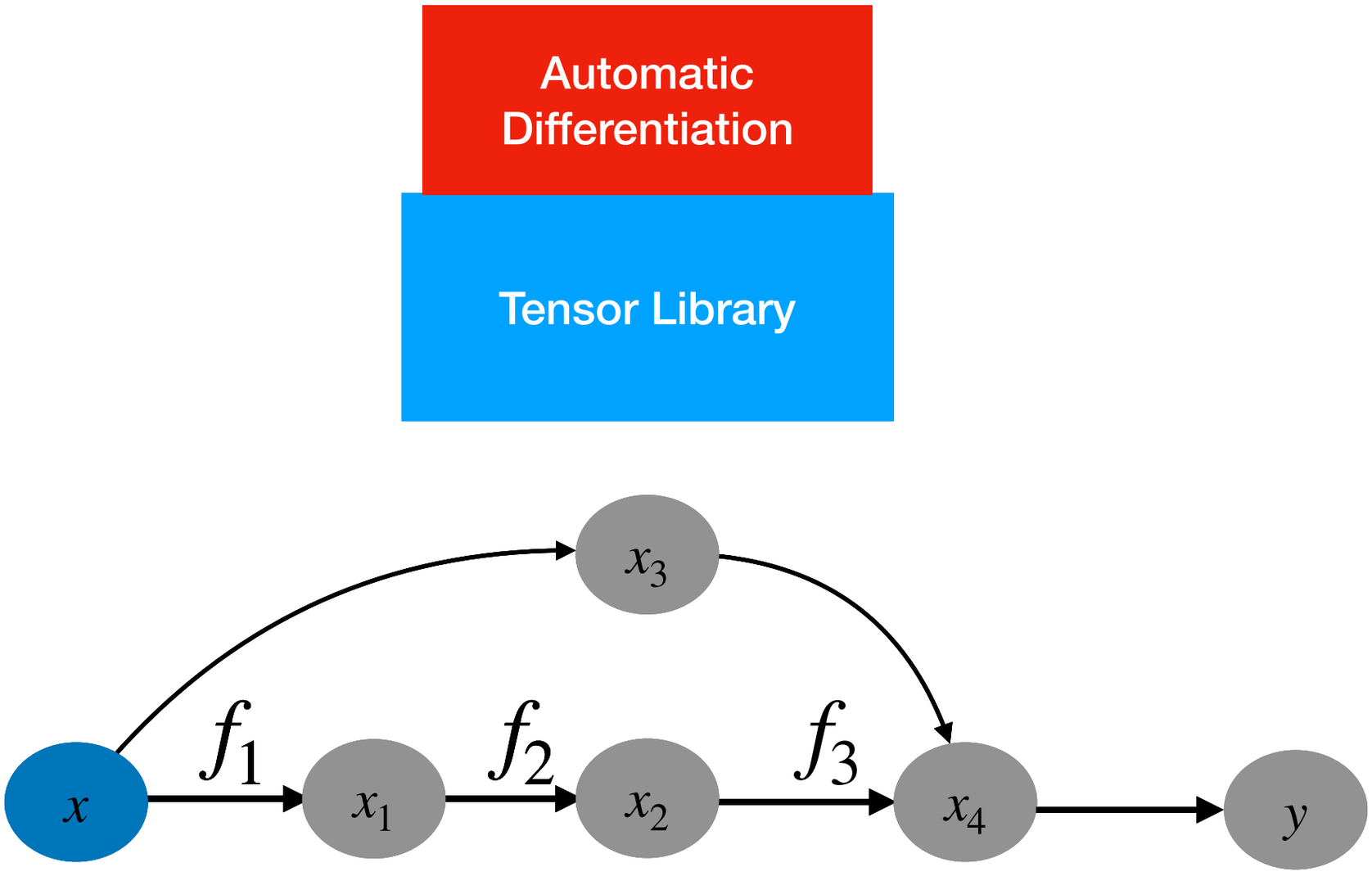


Image from Nvidia

DL Frameworks: Autodiff

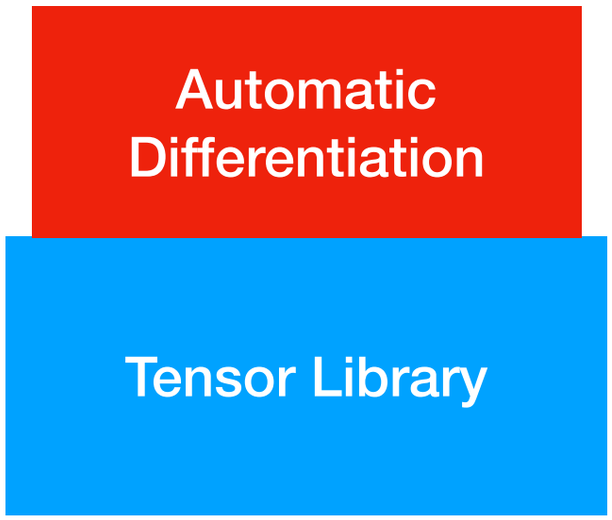


- Automate the construction of computation graph and the backward pass

DL Frameworks: Autodiff

- Frameworks allow to define primitives and optimize their forward and backward computation
- Optimized primitives can be chained together to form complex models

DL Frameworks: Autodiff

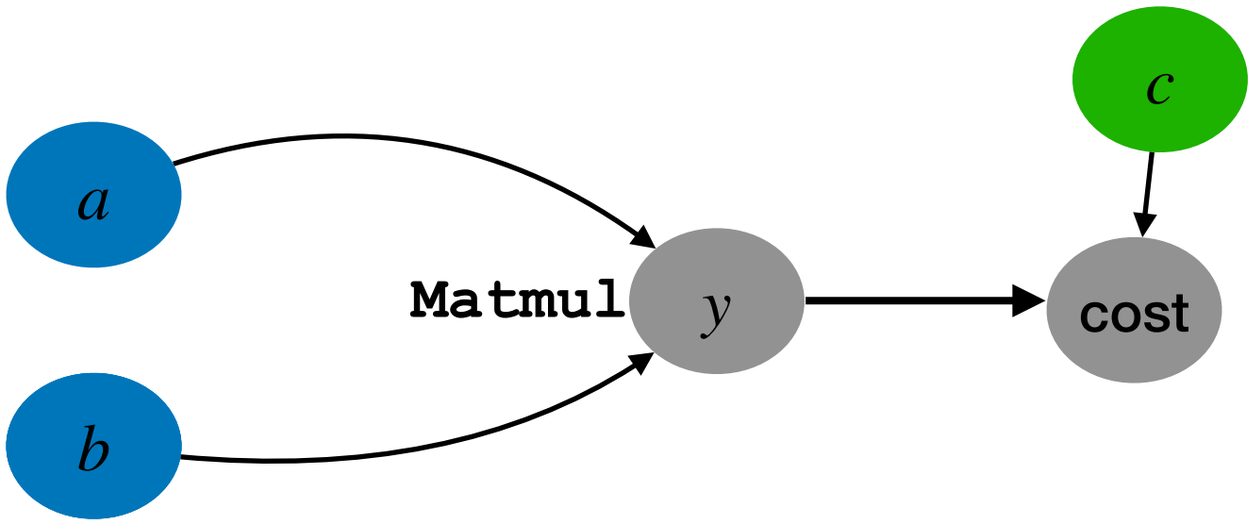


- Automatic Differentiation tools
  - Only need to specify the forward pass behaviour if using predefined primitives
  - Obtain computation graph ahead of time (theano, tensorflow v1, mxnet v1)
  - On the fly (pytorch, mxnet gluon, tensorflow v2)

On the Fly Construction (Tracing)

- Each torch tensor created with `requires_grad=True` will be recognized by torch autograd for building computation graphs
- Graph is constructed on the fly by storing for each node a reference to the parent nodes and functions applied

Torch Autograd




```
# Randomly initialize weights
a = torch.randn((5,5), device=device, dtype=dtype)
b = torch.randn((5,5), device=device, dtype=dtype)
```

```
a.requires_grad = True
b.requires_grad = True

y = torch.matmul(a,b)

#create target
c = torch.randn((5,5), device=device, dtype=dtype)

cost = torch.sum(y - c)**2

cost.backward()
```

```
b.grad
```

```
tensor([[ 334.9134,  334.9134,  334.9134,  334.9134,  334.9134],
        [ 797.6003,  797.6003,  797.6003,  797.6003,  797.6003],
        [   4.5179,    4.5179,    4.5179,    4.5179,    4.5179],
        [-272.1082, -272.1082, -272.1082, -272.1082, -272.1082],
        [ -86.4887,  -86.4887,  -86.4887,  -86.4887,  -86.4887]])
```

Torch Autograd

```
import math

#Initialize
weights = torch.randn(784, 200) / math.sqrt(784)
weights.requires_grad_()
bias = torch.zeros(200, requires_grad=True)

weights2 = torch.randn(200, 10) / math.sqrt(784)
weights2.requires_grad_()
bias2 = torch.zeros(10, requires_grad=True)



def log_softmax(x):
    return x - x.exp().sum(-1).log().unsqueeze(-1)

def model(x):
  out = torch.relu_(torch.matmul(x,weights)+ bias)
  return log_softmax(torch.matmul(out,weights2)+ bias2)

def nll(input, target):
    return -input[range(target.shape[0]), target].mean()
```

```
#create some synthetic data
data = torch.randn(5,784)
targets = torch.randint(10,(5,))

output = nll(model(data),targets)


output.backward()

print(weights2.grad)
print(weights.grad)
print(bias2.grad)
```

```
tensor([[ 0.0547,  0.0627, -0.1317,  ..., -0.2960,  0.0523,  0.0480],
        [-0.1298, -0.1856,  0.0612,  ...,  0.0288,  0.0437,  0.0387],
        [-0.0018, -0.2831,  0.0786,  ..., -0.0708,  0.0482,  0.0423],
        ...,
        [ 0.0493, -0.1256,  0.0723,  ..., -0.0816,  0.0481, -0.1414],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.0028,  0.0036,  0.0033,  ..., -0.0255,  0.0030,  0.0021]])
tensor([[-0.0038,  0.0009,  0.0361,  ...,  0.0192,  0.0000, -0.0374],
        [-0.0011,  0.0247,  0.0149,  ...,  0.0123,  0.0000, -0.0109],
        [-0.0022, -0.0043,  0.0247,  ...,  0.0110,  0.0000, -0.0288],
        ...,
        [-0.0004, -0.0021, -0.0210,  ..., -0.0080,  0.0000, -0.0026],
        [-0.0002,  0.0062, -0.0057,  ..., -0.0009,  0.0000, -0.0016],
        [-0.0002, -0.0006,  0.0069,  ...,  0.0015,  0.0000, -0.0051]])
tensor([-0.0973, -0.0715, -0.0702,  0.0799,  0.0851,  0.0686,  0.1351, -0.1351,
         0.1036, -0.0982])
```

Barebones Autograd Implementations

Mathieu Blondel

https://github.com/mblondel/teaching/blob/main/autodiff-2020/autodiff.py

Andrei Karpathy

https://github.com/karpathy/micrograd

Matt Johnson

https://github.com/mattjj/autodidact

DL Frameworks: Autograd and Pytorch

- Mini-Autograd from Mathieu Blondel:
  https://github.com/mblondel/teaching/blob/main/autodiff-2020/autodiff.py

```
def evaluate_dag(sorted_nodes):
  for node in sorted_nodes:
    if node.value is None:
      values = [p.value for p in node.parents]
      node.value = node.func(*values)
  return sorted_nodes[-1].value
```

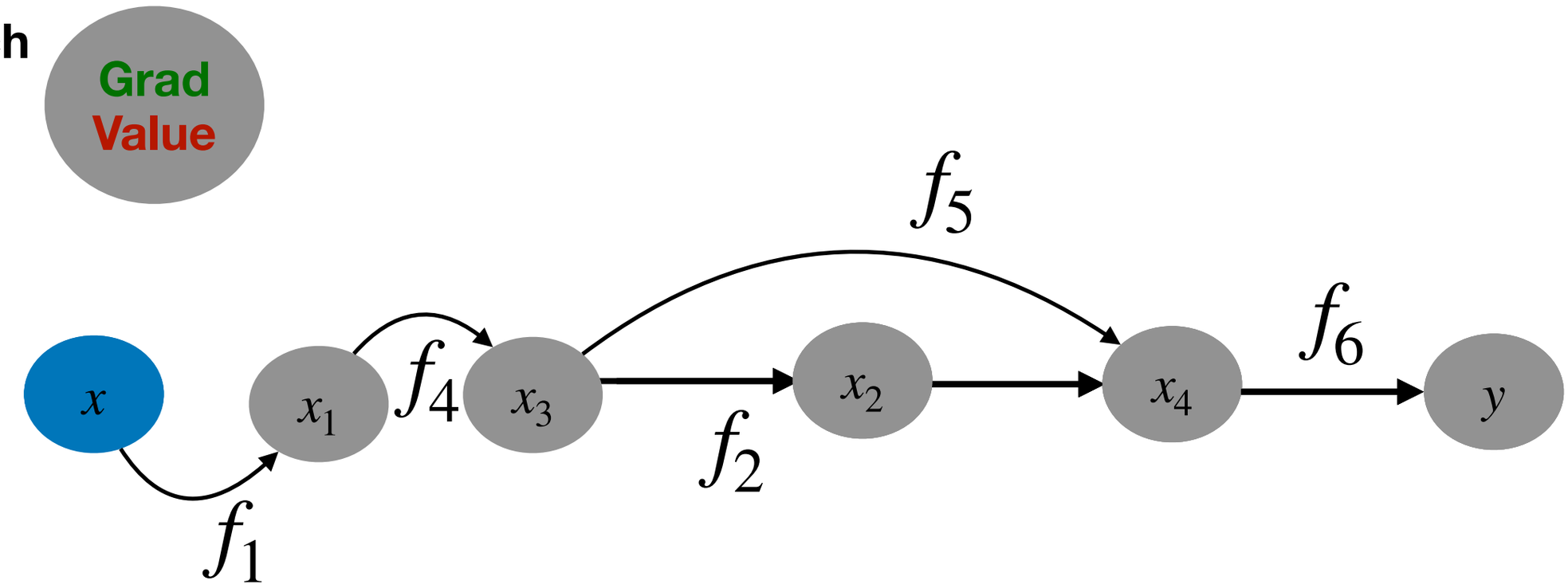
Forward Pass

$x_1, \cdots, x_J \leftarrow$ topological sort$(Graph)$

for $j = 1$ to J :

$x_i \leftarrow f_i(\text{Parent}_1(x_i), \ldots, \text{Parent}_K(x_i))$

Store at each node



DL Frameworks: Autograd and Pytorch

```
def backward_diff_dag(sorted_nodes):
  value = evaluate_dag(sorted_nodes)

  # Initialize recursion.
  sorted_nodes[-1].grad = 1.0

  for node_k in reversed(sorted_nodes):
    if not node_k.parents:
      # We reached a node without parents.
      continue

    # Values of the parent nodes.
    values = [p.value for p in node_k.parents]

    # A list of size len(values) containing the vjps.
    vjps = node_k.func.make_vjp(*values)(node_k.grad)

    for node_j, vjp in zip(node_k.parents, vjps):
      node_j.grad += vjp

return sorted_nodes
```

Backward Pass

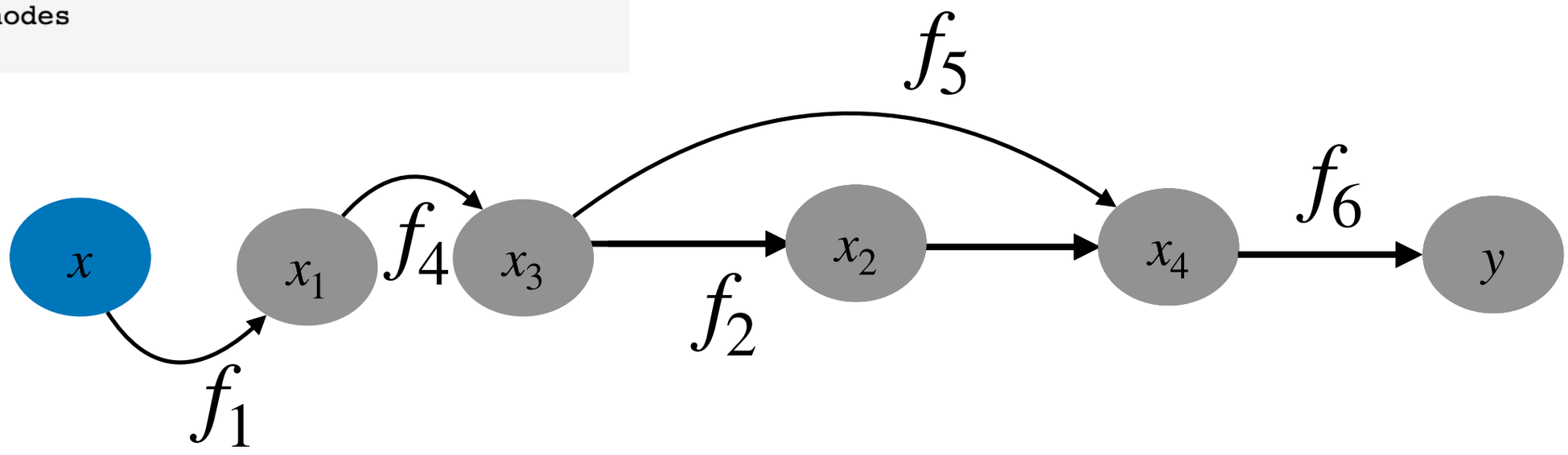
$$\nabla_{x_J} L = \nabla_{x_J} x_J = 1$$

$$\text{for } j = J-1 \text{ to } 1:$$

$$(\nabla_{x_j} L)^T \leftarrow \sum_{k \in Child(x_i)} (\nabla_{x_k} L)^T \frac{\partial x_k}{\partial x_j}$$

Store at each node

Grad
Value



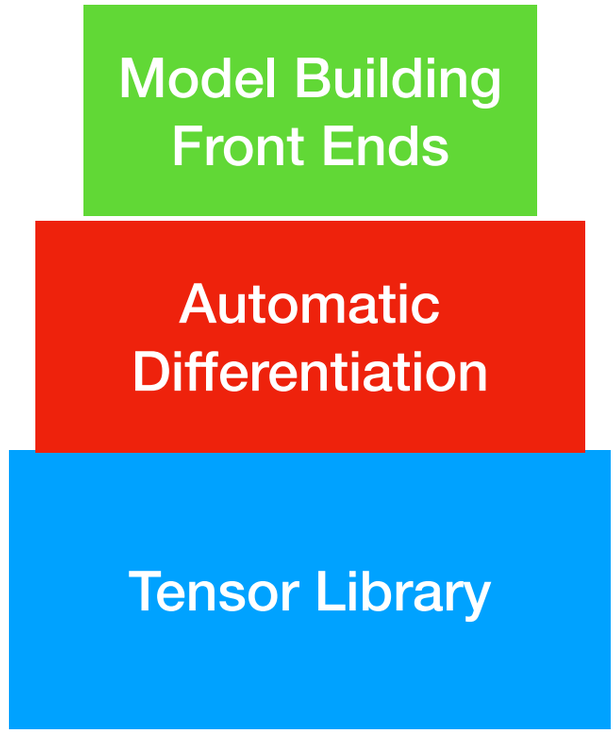
DL Frameworks: Model Building

torch.nn.module

- Simple ways to track and manipulate all parameters of large models

- Allows to easily build and plug and play layers
  - Easily specify the parameters and initialize them
  - Describe the forward pass behaviour

- Designed to work well with training pipelines



DL Frameworks: Model Building

torch.nn.module

```
import torch.nn as nn
import torch.nn.functional as F



class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.linear1 = nn.Linear(784,200)
        self.linear2 = nn.Linear(200,10)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        return self.linear2(x)

loss = nn.CrossEntropyLoss()
model = NN()
for param in model.parameters():
  print(param.data.shape)
```

```
torch.Size([200, 784])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
```

- Provides commonly used module that build on top of each other
- Tracks parameters initialized in __init__
- Specify forward pass behaviour
- Put all parameter tensors on GPU with one call

```
#move all parameters to gpu()
model.cuda()
model.cpu()
```

DL Frameworks: Model Building

```
#create some synthetic data
data = torch.randn(5,784)
targets = torch.randint(10,(5,))
output = loss(model(data),targets)

output.backward()
```

```
for param in model.parameters():
  print(param.grad)
```

```
tensor([[ 1.2310e-02, -2.8742e-02,  8.1141e-03,  ..., -1.9237e-02,
          5.6233e-03,  4.2168e-02],
        [ 1.6153e-02, -6.5609e-03, -9.9466e-05,  ..., -1.3851e-02,
          6.2818e-03, -1.7617e-02],
        [-3.9199e-03,  2.6342e-02,  8.1523e-04,  ...,  1.4414e-02,
          2.5767e-04, -6.8926e-03],
        ...,
        [-1.2751e-02, -3.1583e-03, -1.3319e-02,  ..., -2.6294e-02,
         -1.8542e-02, -2.2516e-02],
        [ 8.7694e-04,  2.5848e-02,  1.4042e-03,  ...,  1.2563e-02,
          3.9489e-03, -1.5012e-02],
        [ 1.0114e-02, -2.3434e-02, -1.5364e-05,  ..., -1.5404e-02,
          4.7295e-03, -5.5517e-03]])
```

References

- Differential Programming by Gabriel Peyre
- Automatic Differentiation Slides by Roger Grosse
- Autodiff Slides and Code from Mathieu Blondel