

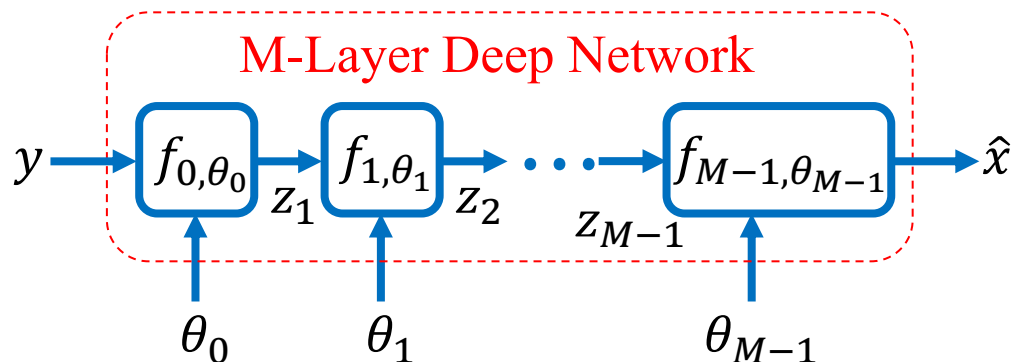
# Optimization of Deep Functions

- Intro to Deep Learning
- Deep Learning Structures
- Gradient of a Deep Network
- Forward and Back Propagation

# Deep Learning Background

- Shallow learning:
  - Inference function formed by 1 or 2 layers.
  - For many years, this was thought to be enough [Cybenko 1989].
- Deep learning:
  - What is it?
    - Uses many layers of inference functions in a hierarchal structure.
    - Typically 10 to 100 layers.
  - Why do people use it?
    - Over the last decade, there has been overwhelming empirical evidence that deep learning dramatically outperforms shallow learning on a wide range of real applications.
    - Intuitively, it is easier to represent complex phenomena using a hierarchy of inference functions.
  - If it's so great, then why did it take so long?
    - It was difficult to implement and no one was sure it would work, but now its easy to implement with modern software tools.
    - Must train all layers jointly  $\Rightarrow$  requires automatic differentiation
    - Many more parameters to model and the problem of “vanishing gradient”
    - Requires more training data.

# Deep Learning Structure



- Inference function is hierarchical.

- For  $M = 4$ , we have that  $\theta = [\theta_0, \theta_1, \theta_2, \theta_3]$ , and

$$\hat{x} = f_{\theta}(y) = f_{3, \theta_3} \left( f_{2, \theta_2} \left( f_{1, \theta_1} \left( f_{0, \theta_0}(y) \right) \right) \right) \text{ (this is a mess)}$$

- Instead write

$$f_{\theta} y = f_{3, \theta_3} f_{2, \theta_2} f_{1, \theta_1} f_{0, \theta_0} y = \left[ \prod_{m=0}^{M-1} f_{m, \theta_m} \right] y$$

# Remember the Loss Gradient

- To compute the loss gradient

$$\nabla L_{MSE}(\theta) = \frac{-2}{K} \sum_{k=0}^{K-1} (x_k - f_{\theta}(y_k))^t \nabla f_{\theta}(y_k)$$

$\epsilon_k^t$   
 error  
 vector

adjoint  
 gradien  
 $t$

$1 \times P$   
 Loss gradient

$1 \times N_x$   
 error vector

$N_x \times P$   
 function gradient  
 $\frac{\partial [f_{\theta}(y_k)]_i}{\partial \theta_j}$

$= \frac{-2}{K} \sum_{k=0}^{K-1}$

- For each training pair, we need to:
  - Compute the error vector
  - Multiply by the adjoint gradient  $\Leftarrow$  **This is the hard part!**
  - Sum them up

# The Adjoint of the Loss Gradient

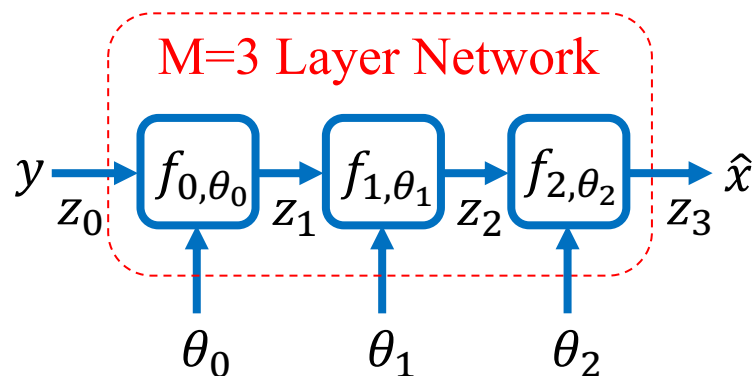
- To compute the loss gradient

$$[\nabla L_{MSE}(\theta)]^t = \frac{-2}{K} \sum_{k=0}^{K-1} [\nabla f_{\theta}(y_k)]^t \cdot (x_k - f_{\theta}(y_k))$$

$$\begin{array}{c}
 \text{Loss gradient} \\
 P \times 1
 \end{array}
 = \frac{-2}{K} \sum_{k=0}^{K-1}
 \begin{array}{c}
 P \times N_x \\
 \text{adjoint function} \\
 \text{gradient} \\
 \frac{\partial [f_{\theta}(y_k)]_j}{\partial \theta_i}
 \end{array}
 \begin{array}{c}
 N_x \times 1 \\
 \text{error vector}
 \end{array}$$

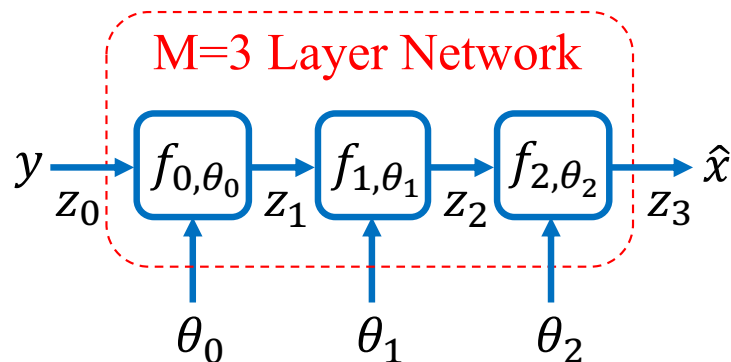
- The adjoint gradient of the loss function:
  - Input and output are exchanged
  - Reverses flow of network

# Example Deep Network



- For  $M = 3$ ,
  - Inference function  $f_{\theta} y = f_{2, \theta_2} f_{1, \theta_1} f_{0, \theta_0} y$
  - Parameter vectors is  $\theta = [\theta_0, \theta_1, \theta_2]$
  - Hidden states  $z_{m+1} = f_{\theta_m} z_m$  with dimension  $z_m \in \mathfrak{R}^{N_m}$
- How do we compute the gradient of this inference function?

# Gradients of Deep Network



- By the chain rule:

- Gradient with respect to  $z_0$

$$\nabla_{z_0} f_{\theta} = [\nabla_{z_2} f_2][\nabla_{z_1} f_1][\nabla_{z_0} f_0]$$

- Adjoint gradient with respect to  $z_0$

$$[\nabla_{z_0} f_{\theta}]^t = [\nabla_{z_0} f_0]^t [\nabla_{z_1} f_1]^t [\nabla_{z_2} f_2]^t$$

- Gradient with respect to  $\theta_0$

$$\nabla_{\theta_0} f_{\theta} = [\nabla_{z_2} f_2][\nabla_{z_1} f_1][\nabla_{\theta} f_0]$$

- Adjoint gradient with respect to  $\theta_0$

$$[\nabla_{\theta_0} f_{\theta}]^t = [\nabla_{\theta} f_0]^t [\nabla_{z_1} f_1]^t [\nabla_{z_2} f_2]^t$$

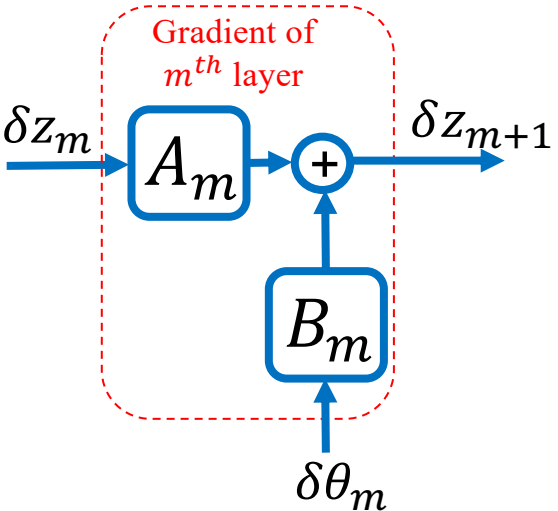
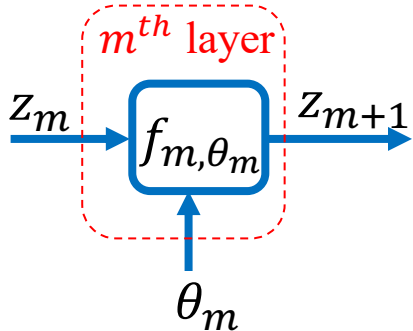
# Gradients for Each Node

- Input gradient of layer:

$$A_m \leftarrow \nabla_z f_{\theta_m}(z_m) = \begin{matrix} \text{index } j \\ \text{input gradient} \\ \frac{\partial [f_{\theta_m}]_i}{\partial [z_m]_j} \\ \text{index } i \end{matrix}$$

- Parameter gradient of layer:

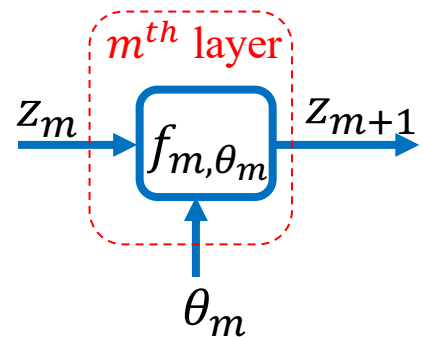
$$B_m \leftarrow \nabla_{\theta} f_{\theta_m}(z_m) = \begin{matrix} \text{index } j \\ \text{parameter gradient} \\ \frac{\partial [f_{\theta_m}]_i}{\partial [\theta_m]_j} \\ \text{index } i \end{matrix}$$



# Adjoint Gradients for Each Node

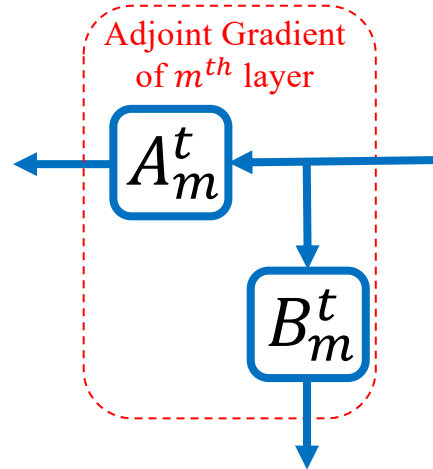
- Input gradient of layer:

$$A_m^t \leftarrow [\nabla_z f_{\theta_m}(z_m)]^t =$$

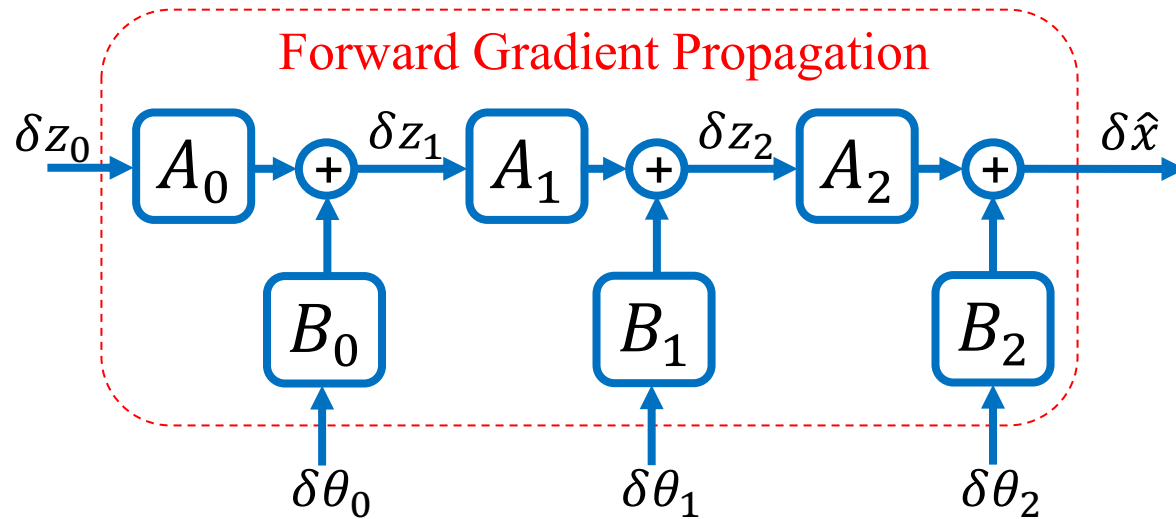


- Parameter gradient of layer:

$$B_m^t \leftarrow [\nabla_{\theta} f_{\theta_m}(z_m)]^t =$$



# Forward Propagation of Gradient



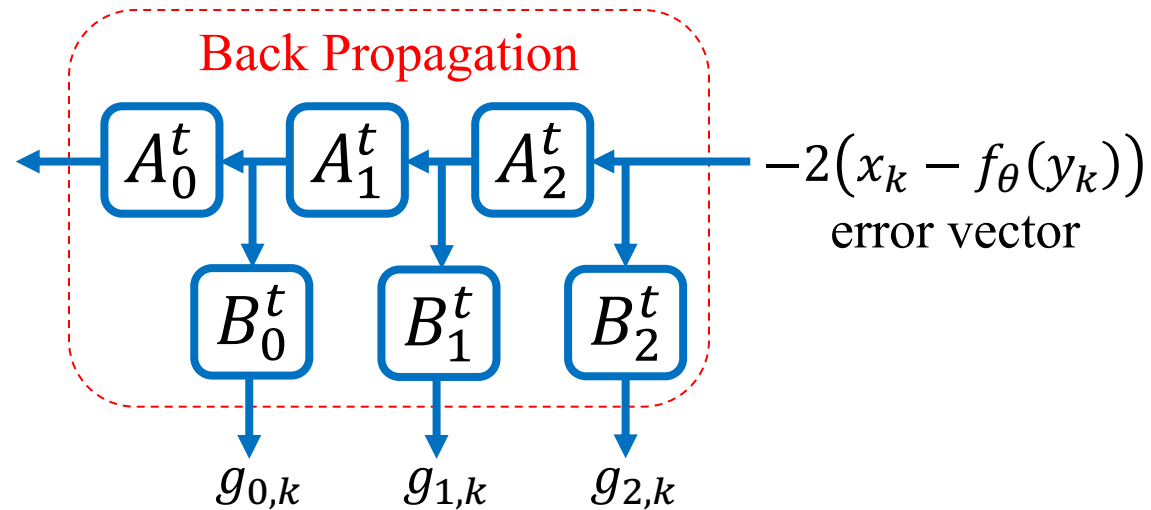
- By the chain rule, the gradients with respect to inputs are

$$\begin{aligned}\nabla_{z_2} f_{\theta} &= A_2 \\ \nabla_{z_1} f_{\theta} &= A_2 A_1 \\ \nabla_{z_0} f_{\theta} &= A_2 A_1 A_0\end{aligned}$$

- By the chain rule, the gradients with respect to parameters are

$$\begin{aligned}\nabla_{\theta_2} f_{\theta} &= B_2 \\ \nabla_{\theta_1} f_{\theta} &= A_2 B_1 \\ \nabla_{\theta_0} f_{\theta} &= A_2 A_1 B_0\end{aligned}$$

# Back Propagation of Adjoint Gradient



- Compute loss gradient by reversing network flow:

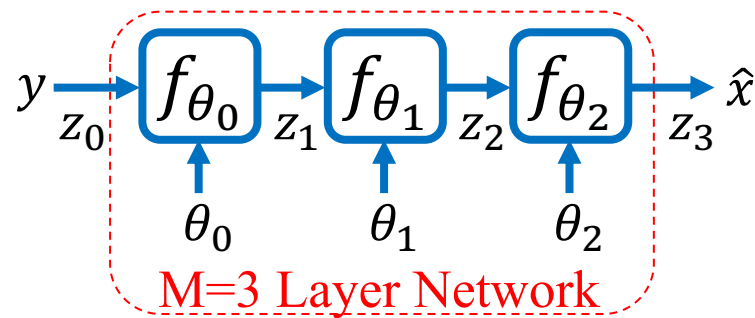
- For each training vector,

$$g_{m,k} = \nabla_{\theta_m} \|x_k - f_\theta(y_k)\|^2 = -2(x_k - f_\theta(y_k))^t \nabla f_\theta(y_k)$$

- So then the gradient of the loss function is given by

$$\nabla_{\theta_m} L(\theta) = g_m = \frac{1}{K} \sum_{k=0}^{K-1} g_{m,k}$$

# BP Step 1: Forward Propagate States



- Forward propagation pseudo code:

```
z0 ← y
For (m = 0 to M - 1) {
  zm+1 ← fθm zm
}
x̂ ← zM
```

*Forward Propagation*

# BP Step 2: Back Propagation

## ■ Backpropagation pseudo code

For( $m = 0$  to  $M - 1$ )  $g_m \leftarrow 0$

For ( $k = 0$  to  $K - 1$ ) {

$\epsilon_M \leftarrow -2(x_k - f_\theta(y_k))$

For( $m = M - 1$  to 0) {

$A \leftarrow \nabla_z f_{\theta_m}(z_m)$

$B \leftarrow \nabla_\theta f_{\theta_m}(z_m)$

$\epsilon_m \leftarrow A^t \epsilon_{m+1}$

$g_m \leftarrow g_m + B^t \epsilon_{m+1}$

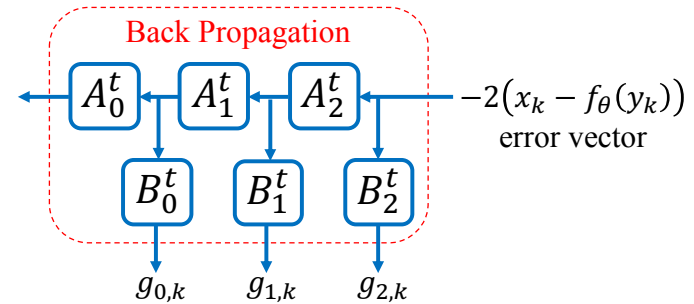
}

}

For( $m = 0$  to  $M - 1$ )  $g_m \leftarrow g_m / K$

Return( $g_m$ )

*Back Propagation*



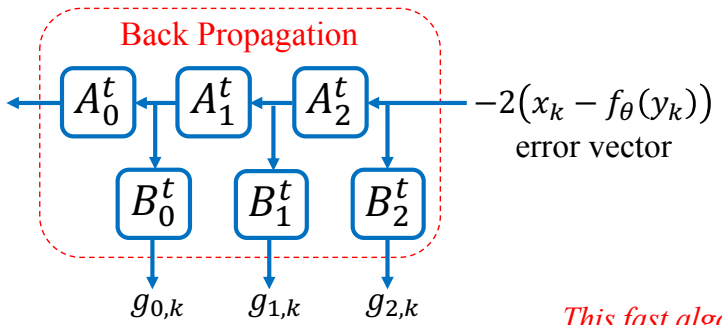
Results in

$$g_m = \nabla_{\theta_m} L(\theta)$$

where

$$L(\theta) = \frac{1}{K} \sum_{k=0}^{K-1} \|x_k - f_\theta(y_k)\|^2$$

# Back Propagation: Computational Efficiency



*This fast algorithm is sometimes referred to as adjoint differentiation.*

- Order of computation matters!!
  - Left to Right: Dense matrix products  $\Rightarrow \mathcal{O}(N^3K)$
  - Right to Left: Vector matrix products  $\Rightarrow \mathcal{O}(N^2K)$

$$g_o = \begin{matrix} P_0 \times 1 \\ \text{Loss gradient} \end{matrix} = \frac{1}{K} \sum_{k=0}^{K-1} \begin{matrix} P_0 \times N_1 \\ \text{parameter gradient} \\ B_0^t \end{matrix} \begin{matrix} N_1 \times N_2 \\ \text{input gradient} \\ A_1^t \end{matrix} \begin{matrix} N_2 \times N_3 \\ \text{input gradient} \\ A_2^t \end{matrix} \begin{matrix} N_3 \times 1 \\ \text{error vector} \end{matrix}$$

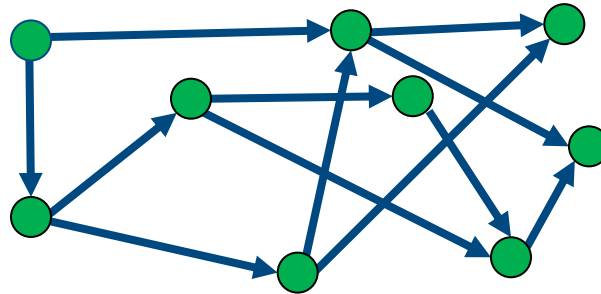
$\xrightarrow{\text{left to right}}$   
 $\xleftarrow{\text{right to left}}$

# GD on Acyclic Graph Structures

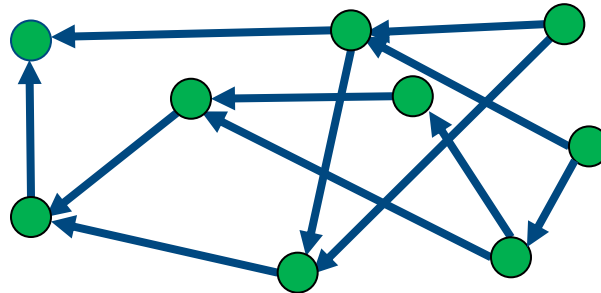
- Definition of DAG
- General Node Structure
- Back Propagation on Graphs

# Define DAG

- Directed Acyclic Graph (DAG)
  - A directed graph with no loops.
  - Starting at any node, you will eventually hit a termination

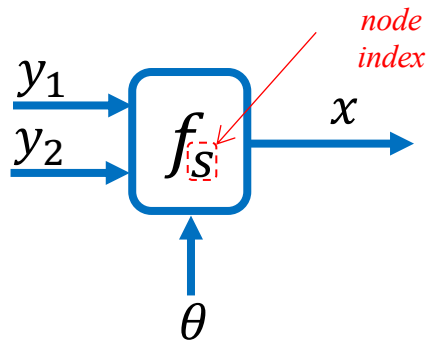


- If you reverse the arrows (i.e., flow), it's still a DAG



# Define DL Node

- Assume that DL network is a DAG.
  - Then w.o.l.o.g for a fan-in of 2, each nodes is



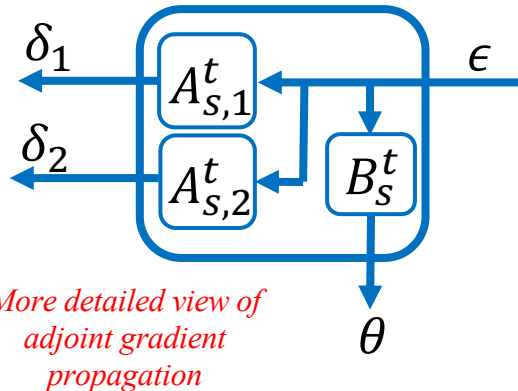
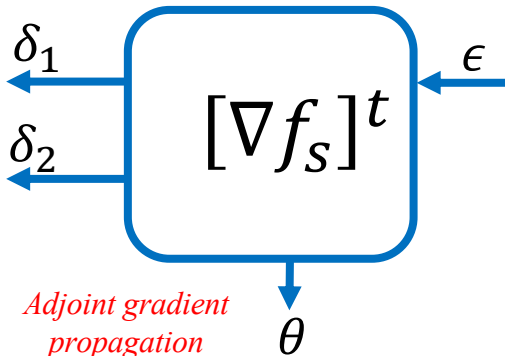
$$x = f_{s,\theta}(y_1, y_2)$$

$$A_{s,1} = \nabla_{y_1} f_s$$

$$A_{s,2} = \nabla_{y_2} f_s$$

$$B_s = \nabla_{\theta} f_s$$

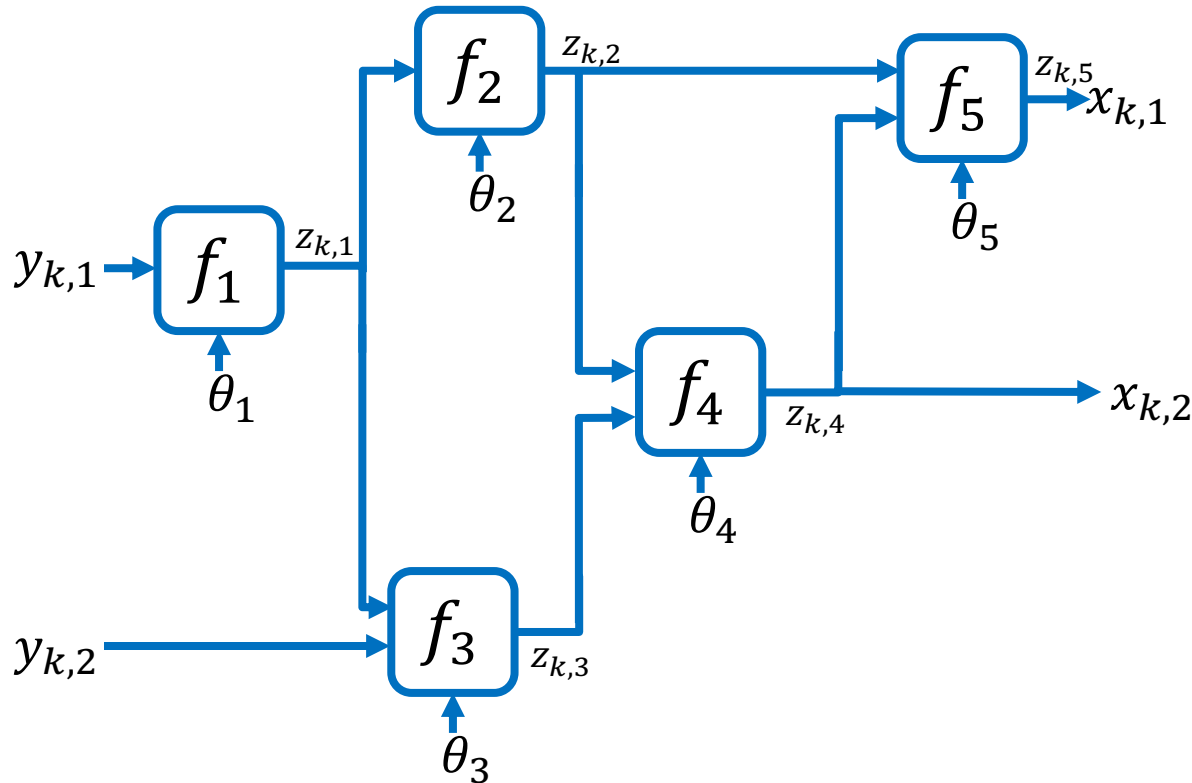
- And the back propagation is



# Step 1: Forward Propagation for DAG

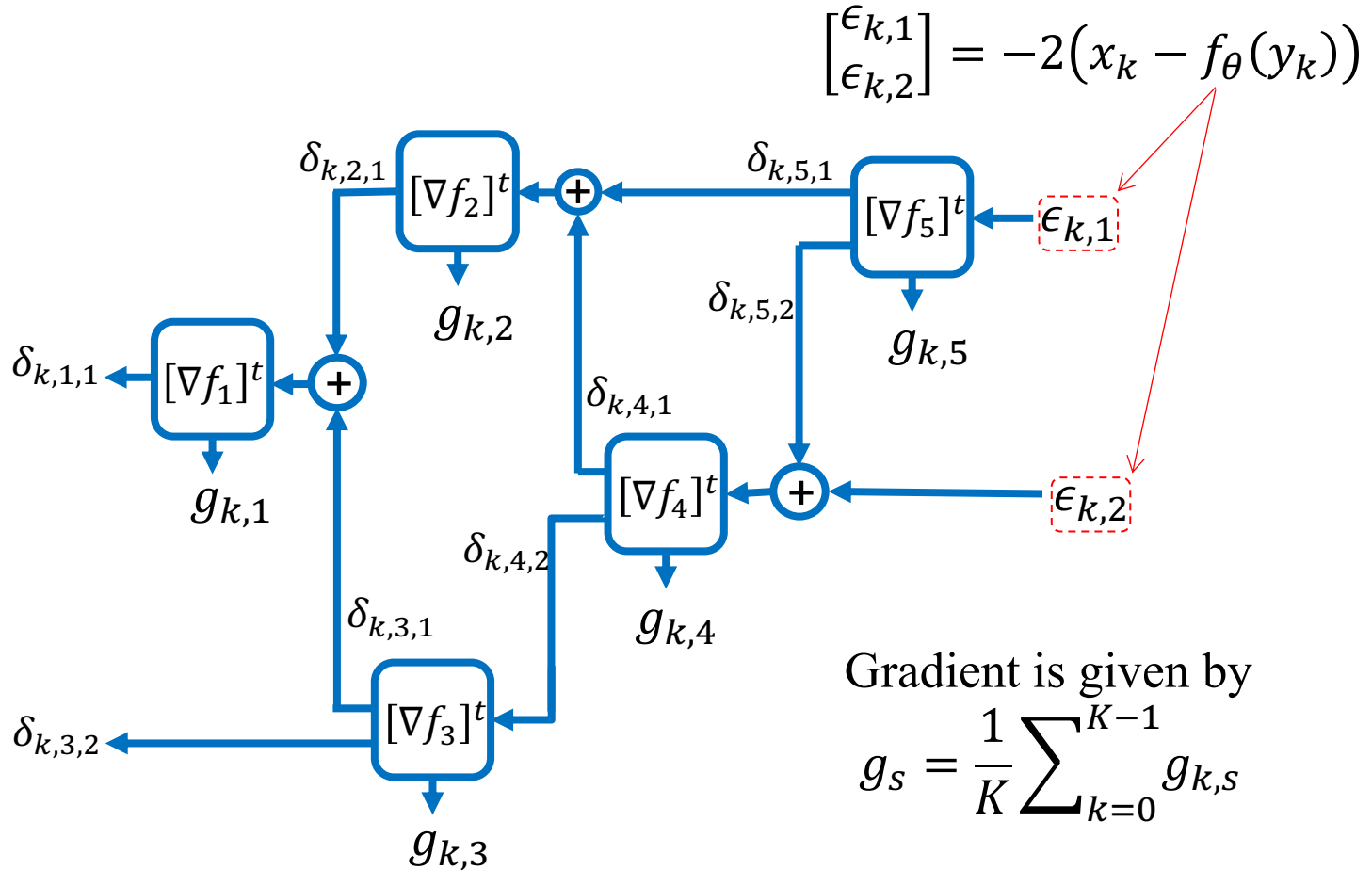
- Forward propagation.

$$\begin{bmatrix} x_{k,1} \\ x_{k,2} \end{bmatrix} = f_{\theta} \left( \begin{bmatrix} y_{k,1} \\ y_{k,2} \end{bmatrix} \right)$$



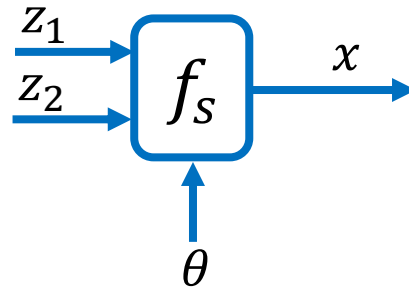
# Step 2: Back Propagation for DAG

- Back propagation using adjoint gradients.



# Implementation of Each Node

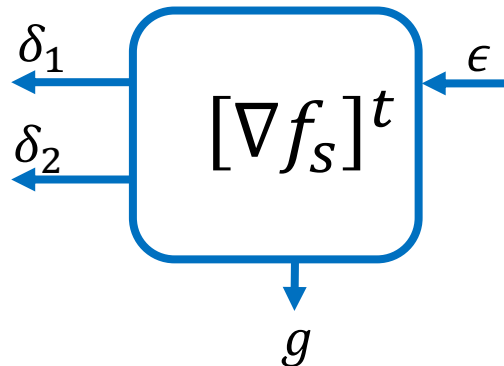
- For each node you need two functions:
  - Function for forward propagation



$$x \leftarrow f(z_1, z_2, \theta)$$

*You need a software implementation of this forward function.*

- Function for adjoint gradient (i.e., back propagation)



$$[\delta_1, \delta_2, g] \leftarrow G(\epsilon, z_1, z_2, \theta)$$

*You need a software implementation of this function that multiplies  $\epsilon$  by the adjoint gradient.*

Very important: You don't need to ever compute  $[\nabla f_S]^t$ !!

# General Loss Functions

- The Weakness of MSE Loss Functions
- MAE and Cross Correlation Loss Functions
- Back Propagating General Loss Functions

# Weakness of MSE Loss

- MSE loss is given by

$$L_{MSE}(\theta) = \frac{1}{K} \sum_{k=0}^{K-1} \|x_k - f_{\theta}(y_k)\|^2$$

- Excessively weights outliers

- If training sample is wrong, then prediction error can be very large.
- If a few bad training samples can dramatically degrade results.

$$\begin{aligned} & [\nabla L_{MSE}(\theta)]^t \quad \begin{array}{c} \text{adjoint gradient} \\ \text{of function} \end{array} \\ & = \frac{-2}{K} \sum_{k=0}^{K-1} [\nabla f_{\theta}(y_k)]^t (x_k - f_{\theta}(y_k)) \quad \begin{array}{c} \text{prediction} \\ \text{error} \end{array} \end{aligned}$$

# A General Class of Loss Functions

- General loss functions

$$L(\theta) = \frac{1}{K} \sum_{k=0}^{K-1} \rho(x_k, f_{\theta}(y_k))$$

– where  $\rho$  is a distortion function s.t.  $\forall a, b, \rho(a, b) \geq \rho(a, a)$ .

- Typical choices include

– Mean Squared Error (MSE)

- $\rho(a, b) = \|a - b\|^2 = \sum_i (a_i - b_i)^2$  for  $\forall a, b \in \mathfrak{R}^{N_x}$

– Mean Absolute Error (MAE)

- $\rho(a, b) = \|a - b\|_1 = \sum_i |a_i - b_i|$  for  $\forall a, b \in \mathfrak{R}^{N_x}$

– Multicategory Cross Entropy (assumes  $b_i > 0$ ,  $\sum_i b_i = 1$ , and  $a_i \geq 0$ )

- $\rho(a, b) = -\sum_i a_i \log b_i$  for  $\forall a, b \in \mathcal{S}^{N_x}$

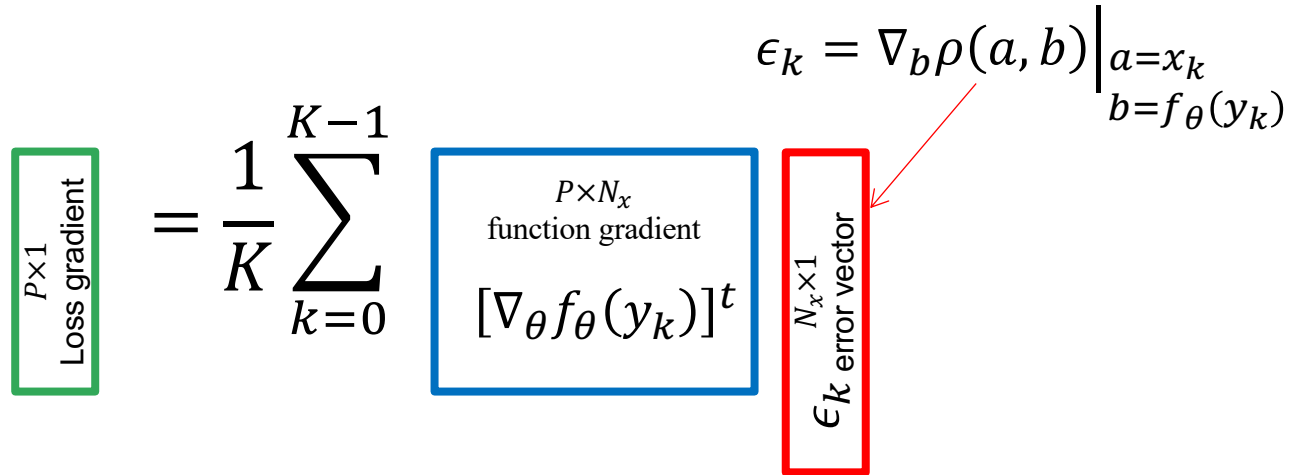
# Gradient for General for Loss

- Alternative loss functions

$$L(\theta) = \frac{1}{K} \sum_{k=0}^{K-1} \rho(x_k, f_{\theta}(y_k))$$

Results in gradient

$$[\nabla L(\theta)]^t = \frac{1}{K} \sum_{k=0}^{K-1} [\nabla_{\theta} f_{\theta}(y_k)]^t \epsilon_k$$



# Error Vector for General Loss Functions

$$L(\theta) = \frac{1}{K} \sum_{k=0}^{K-1} \rho(x_k, f_{\theta}(y_k))$$

General loss functions

$$\epsilon_k = \nabla_b \rho(a, b) \Big|_{\substack{a=x_k \\ b=f_{\theta}(y_k)}}$$

General error function

- MSE

Loss:  $\rho(a, b) = \|a - b\|^2$

Error:  $\epsilon_k^t = -2(x_k - f_{\theta}(y_k))$

- MAE

Loss:  $\rho(a, b) = \|a - b\|_1$

Error:  $\epsilon_k^t = -\text{sign}(x_k - f_{\theta}(y_k))$

- Cross entropy\*

Loss:  $\rho(a, b) = -\sum_i a_i \log b_i$

Error:  $\epsilon_k^t = -\frac{x_k}{f_{\theta}(y_k)}$

(Assumes  $b_i > 0$ ,  $\sum_i b_i = 1$ , and  $a_i \geq 0$ )

*Point-wise division*

\*The software utility includes softmax preprocessing: <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>

# Gradient for Cross Entropy Loss

■ Paradox:

- Let  $f_{\theta}(y) = \theta$  for  $\sum_i \theta_i = 1$ , and for all  $i$  let  $x_i = a$ , then for cross-entropy we have that:

$$L(\theta) = -\sum_i x_i \log \theta_i = -\sum_i a \log \theta_i \quad \text{for } \sum_i \theta_i = 1$$

So that

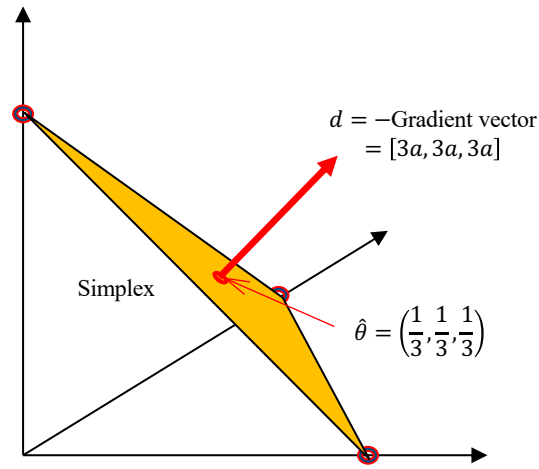
$$[\nabla_{\theta} L(\theta)]_i = [\nabla_{\theta} \rho(x, \theta)]_i = -\frac{x_i}{\theta_i} = -\frac{a}{\theta_i}$$

- Paradox: Even when  $\hat{\theta}_i = \frac{x_i}{\sum_i x_i} = \frac{1}{3}$ , the gradient is not zero!

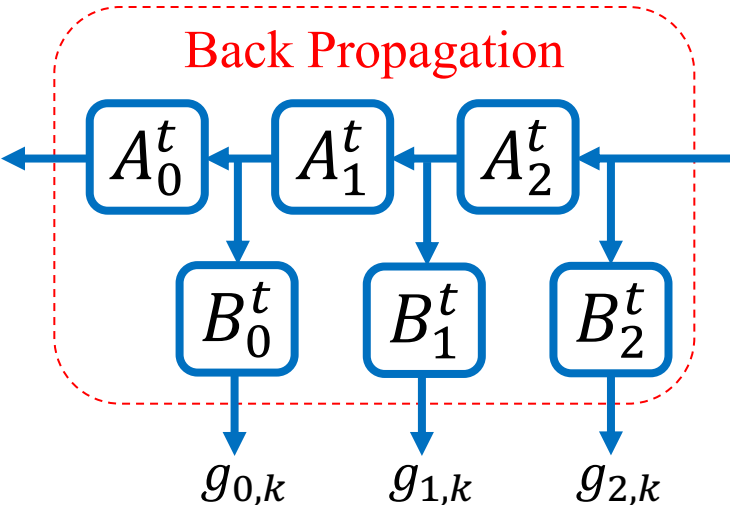
$$d = -\nabla_{\theta} L(\hat{\theta}) = [3a, 3a, 3a]$$

- How can this be??

3D Example:  
Gradient is perpendicular  
to constraint space!



# Back Propagation for Typical Loss Functions



General:  $\epsilon_k = \nabla_b \rho(a, b) |_{a=x_k, b=f_\theta(y_k)}$

MSE:  $\epsilon_k^t = -2(x_k - f_\theta(y_k))$

MAE:  $\epsilon_k^t = -\text{sign}(x_k - f_\theta(y_k))$

CE:  $\epsilon_k^t = -\frac{x_k}{f_\theta(y_k)}$

- Loss gradient given by

$$\nabla_{\theta_m} L(\theta) = \frac{1}{K} \sum_{k=0}^{K-1} g_{m,k}$$

# DAG Back Propagation for General Loss

- Back propagation using adjoint gradients.

