

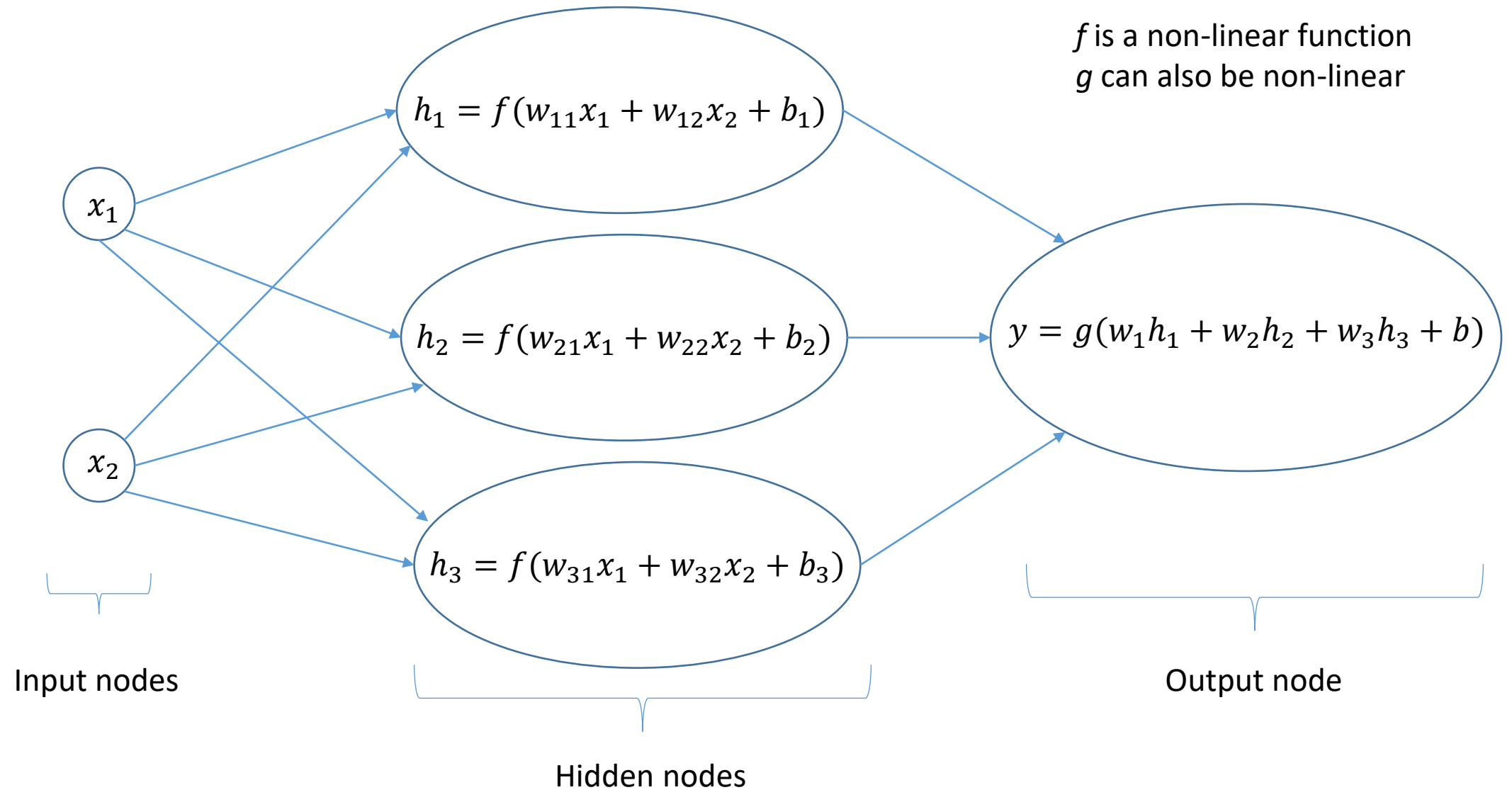
# Introduction to Neural Networks

Computing Science  
University of Alberta  
Nilanjan Ray

# Agenda

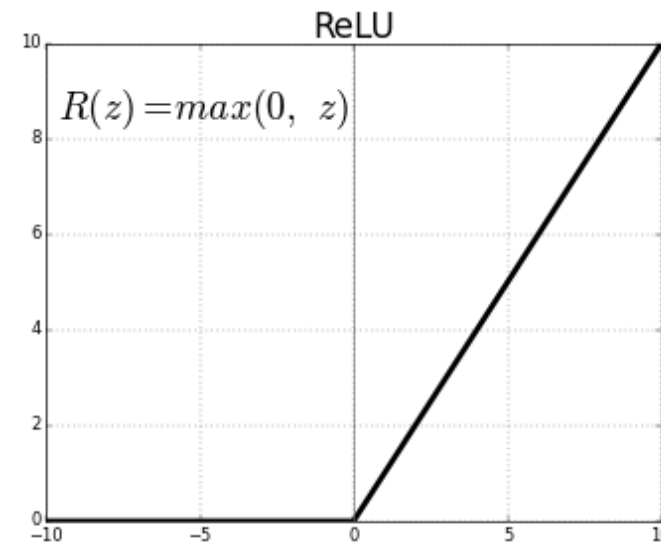
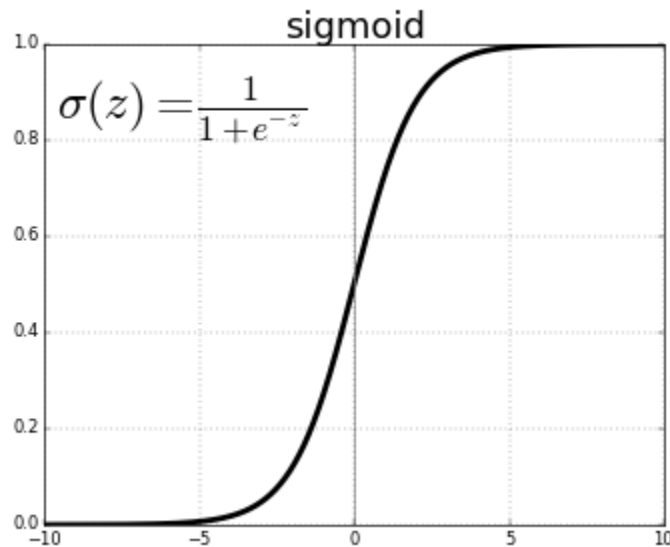
- What is a Neural Net?
  - Neural net as a computational graph
- Approximating “XOR” function with neural net
- Understanding backpropagation
- Universal function approximation by a neural net

# Feed forward neural network



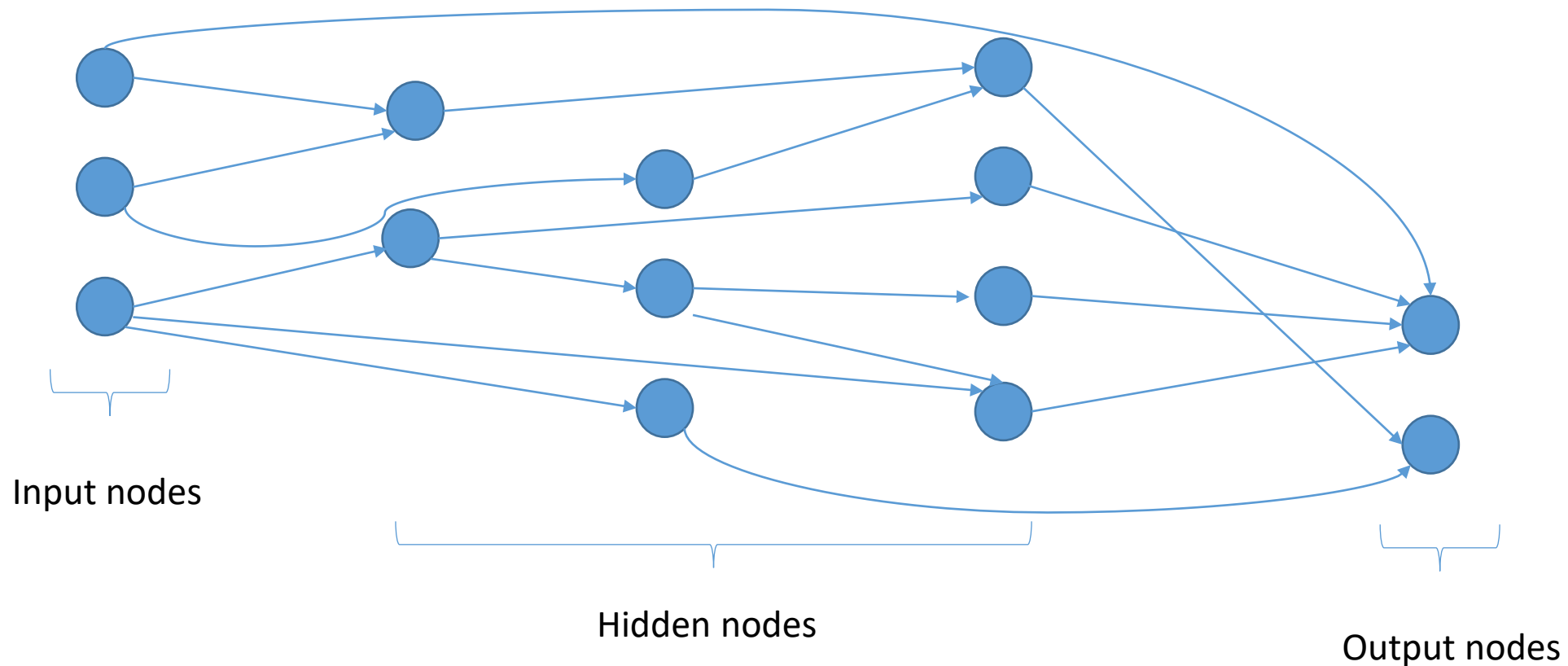
# Feed forward net: non-linear functions

- Non-linear functions at hidden nodes are known as “activation function”
  - Sigmoid, ReLU, ELU, ....



Why activation functions are non-linear?

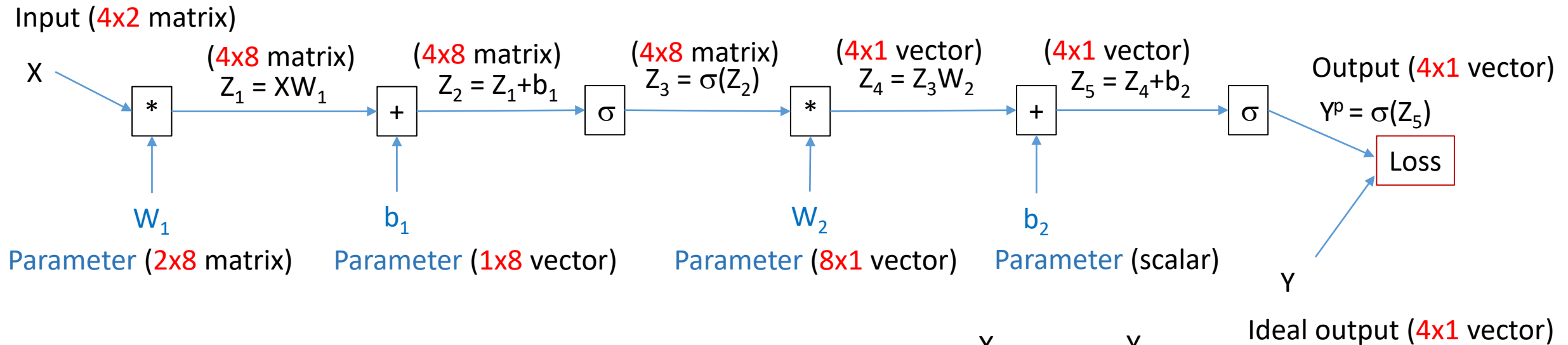
# Feedforward net in general: Directed acyclic graph



# What's the big deal about neural net?

- Mathematically very rich: it can approximate any function
- It is biologically inspired: (loosely) resembles brain connections
- Computationally:
  - Simple: matrix-vector multiplication and point-wise non-linear function
  - Highly parallelizable: cuBLAS, GEMM, Batched GEMM!
- Excellent **empirical** results on “generalization capability” over variety of applications!

# Neural network as a computational graph



$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

Sigmoid function;  
applied **pointwise**  
to a vector or  
matrix input

This network is  
trying to learn  
XOR function

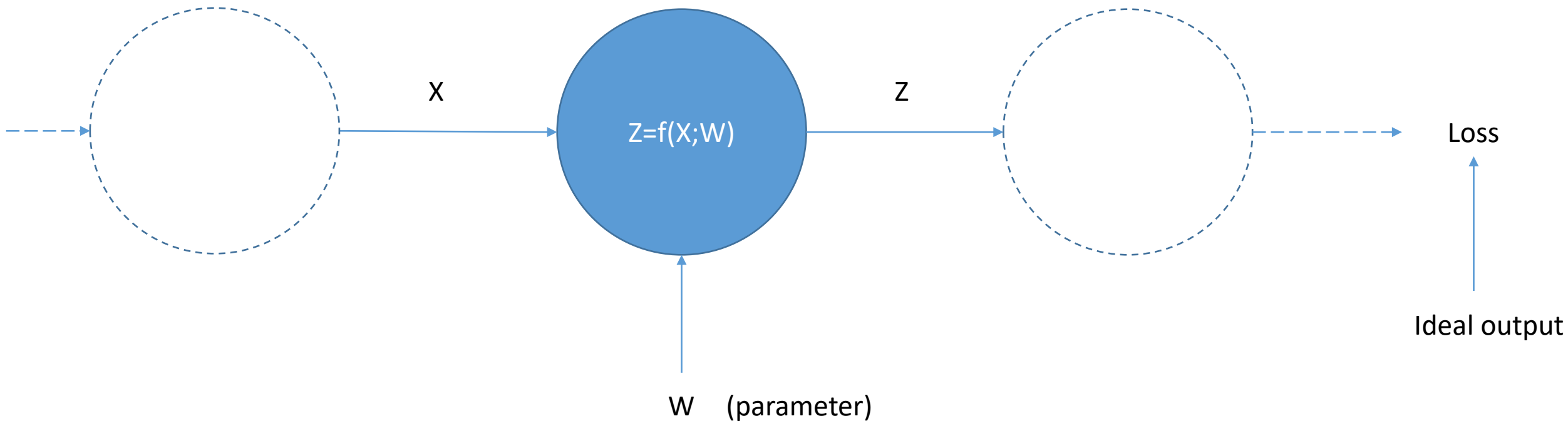
X		Y
1	0	1
0	0	0
0	1	1
1	1	0

# How does PyTorch optimize parameters?

- By **gradient descent** PyTorch adjusts network parameters to reduce the value of the loss function.
- But how?
  - Answer: **Backpropagation**
- **Let us learn how to do backpropagation on a computational graph!**



# Chain rule of derivative for a computational node

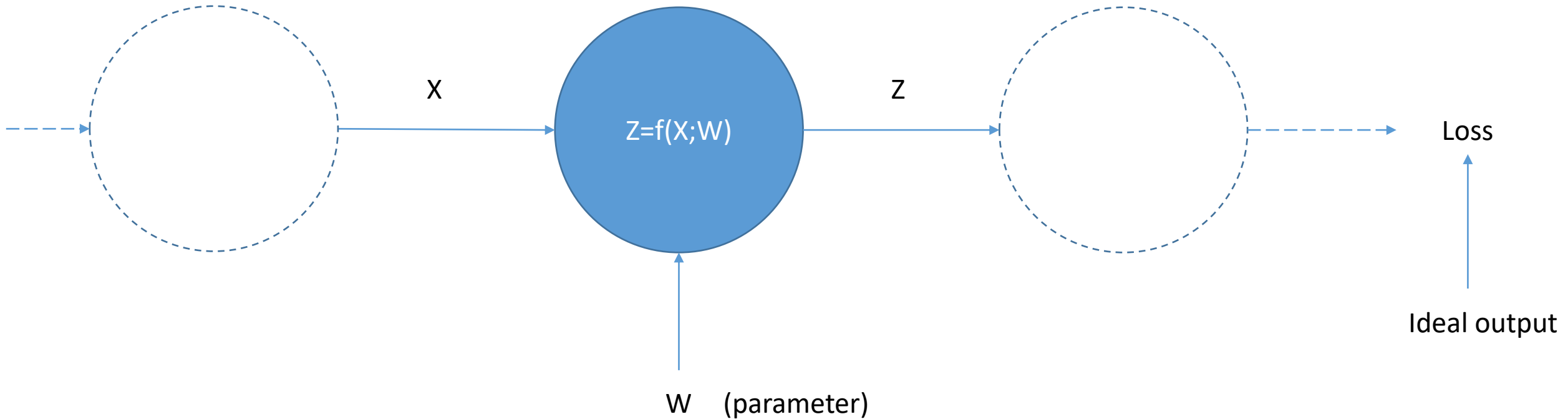


If  $X, Z, W$  are all scalars, then usual chain rule of derivative applies:

$$\frac{\partial(\text{Loss})}{\partial X} = \frac{\partial Z}{\partial X} \frac{\partial(\text{Loss})}{\partial Z}$$

$$\frac{\partial(\text{Loss})}{\partial W} = \frac{\partial Z}{\partial W} \frac{\partial(\text{Loss})}{\partial Z}$$

# Chain rule of derivative...



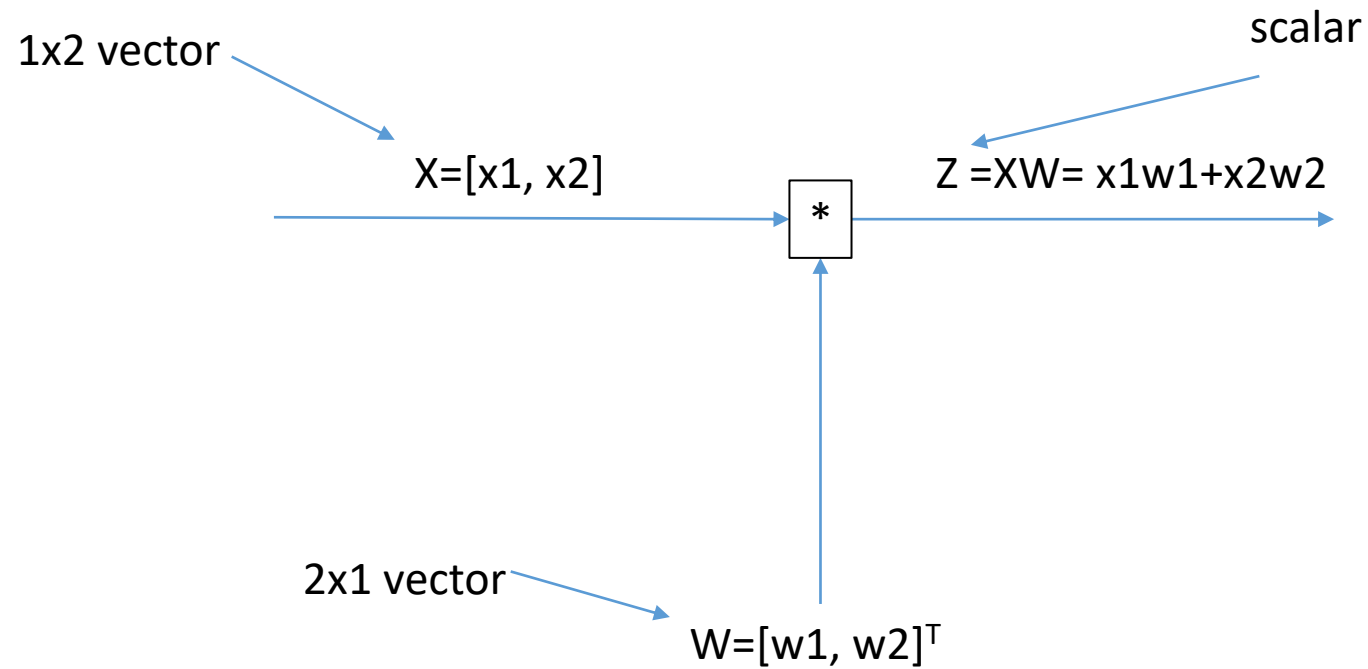
If  $X$ ,  $Z$ ,  $W$  are **matrices or vectors**, then :

$$\nabla_X(\text{Loss}) = \left( \frac{\partial Z}{\partial X} \right) * \nabla_Z(\text{Loss})$$

$$\nabla_W(\text{Loss}) = \left( \frac{\partial Z}{\partial W} \right) * \nabla_Z(\text{Loss})$$

“\*” refers to matrix vector multiplication

# Example 1

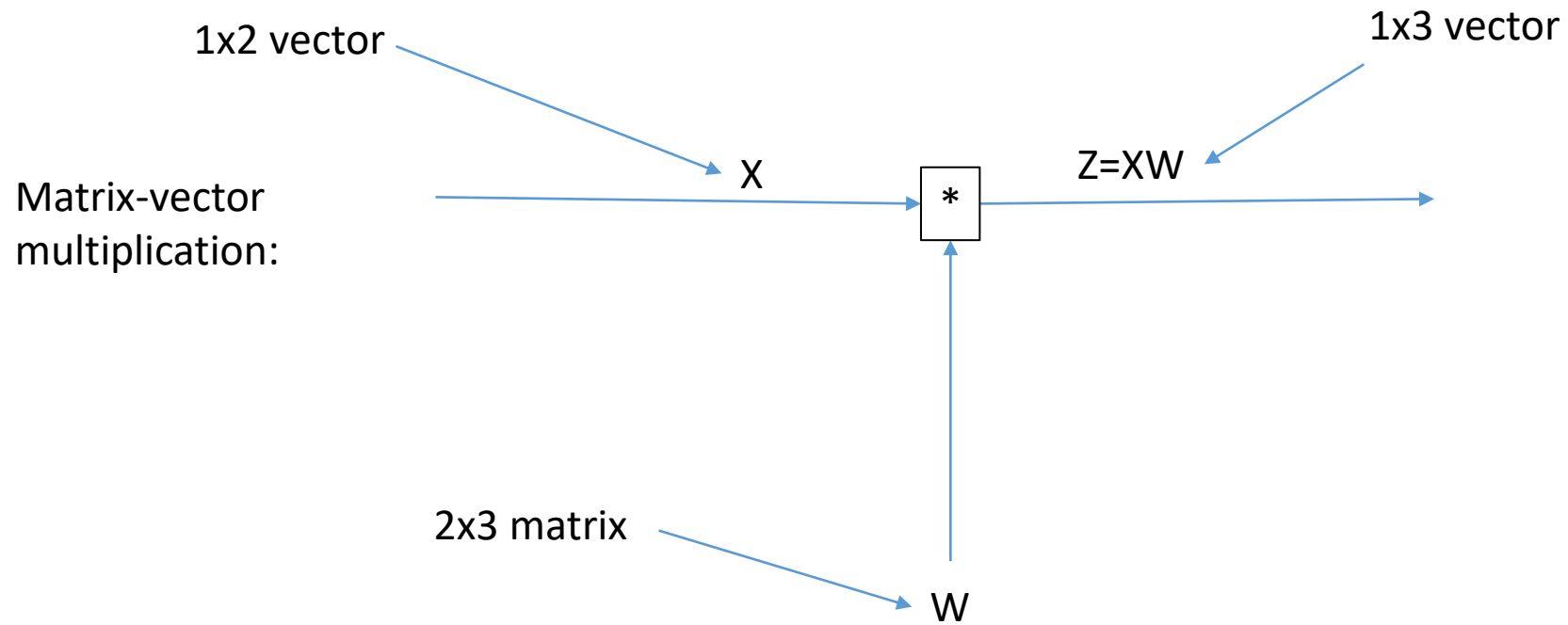


Chain rules:  $\nabla_X(\text{Loss}) = W^T \frac{\partial(\text{Loss})}{\partial Z} = [w1 \quad w2] \frac{\partial(\text{Loss})}{\partial Z}$

$$\nabla_W(\text{Loss}) = X^T \frac{\partial(\text{Loss})}{\partial Z} = \begin{bmatrix} x1 \\ x2 \end{bmatrix} \frac{\partial(\text{Loss})}{\partial Z}$$

Why?

# Example 2



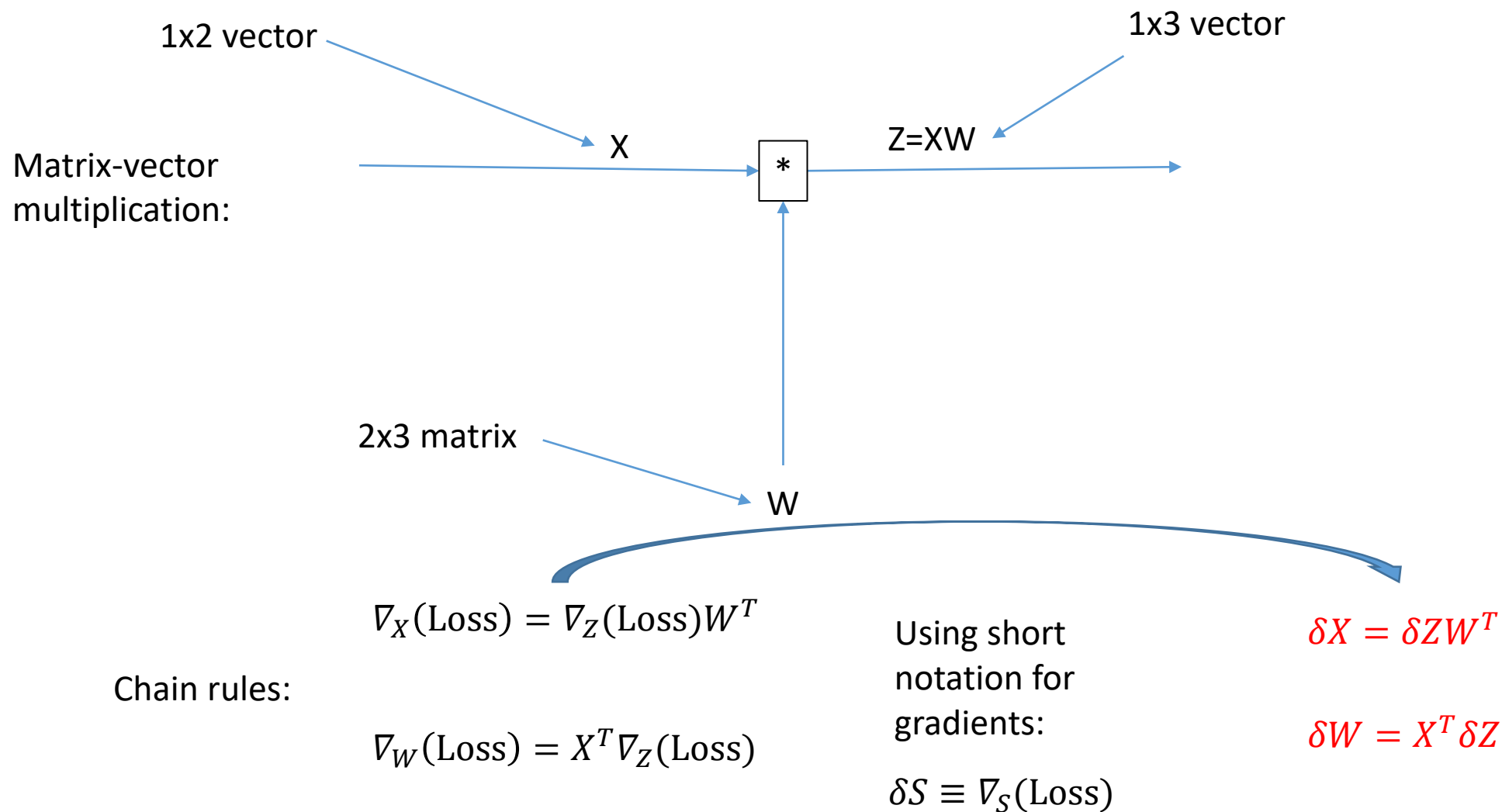
$$\nabla_X(\text{Loss}) = \nabla_Z(\text{Loss})W^T$$

Chain rules:

$$\nabla_W(\text{Loss}) = X^T \nabla_Z(\text{Loss})$$

Why?

# Backprop derivation



# Backprop derivation...

The diagram illustrates the backpropagation derivation through several steps, with annotations explaining each part:

- Step 1:**  $\delta X_i = \sum_k \underbrace{\frac{\partial Z_k}{\partial X_i} \frac{\partial(\text{Loss})}{\partial Z_k}}_{\text{Chain rule of derivative}} = \sum_k \underbrace{\frac{\partial}{\partial X_i} \left[ \sum_j X_j W_{jk} \right]}_{\text{Substitute } Z_k} \delta Z_k = \sum_k W_{ik} \delta Z_k$ 
  - An arrow points from  $\delta X_i$  to the text:  $i^{\text{th}}$  component of  $\delta X$  vector.
  - An arrow points from the chain rule term to the text: Chain rule of derivative.
  - An arrow points from the substitution term to the text: Substitute  $Z_k$ .
  - An arrow points from  $\delta Z_k$  to the text:  $k^{\text{th}}$  component of  $\delta Z$  vector.
- Step 2:** A large blue arrow points from the scalar equation to the matrix equation.
- Step 3:**  $\delta X = \delta Z W^T$  (written in red).
  - An arrow points from the text "Because," to the equation  $\frac{\partial}{\partial X_i} \left[ \sum_j X_j W_{jk} \right] = W_{ik}$ .
  - An arrow points from the text "Writing in matrix-vector multiplication form" to the final equation  $\delta X = \delta Z W^T$ .

# Backprop derivation...

$$\delta W_{ij} = \sum_k \underbrace{\frac{\partial Z_k}{\partial W_{ij}} \frac{\partial(\text{Loss})}{\partial Z_k}}_{\text{Chain rule of derivative}} = \sum_k \frac{\partial}{\partial W_{ij}} \underbrace{\left[ \sum_m X_m W_{mk} \right]}_{\text{Substitute } Z_k} \delta Z_k = X_i \delta Z_j$$

$(i,j)^{\text{th}}$  component of  $\delta W$  matrix

Chain rule of derivative

Substitute  $Z_k$

Because,

$$\frac{\partial}{\partial W_{ij}} \left[ \sum_m X_m W_{mk} \right] = \begin{cases} X_i, & \text{if } i = m \text{ and } j = k, \\ 0, & \text{otherwise.} \end{cases}$$

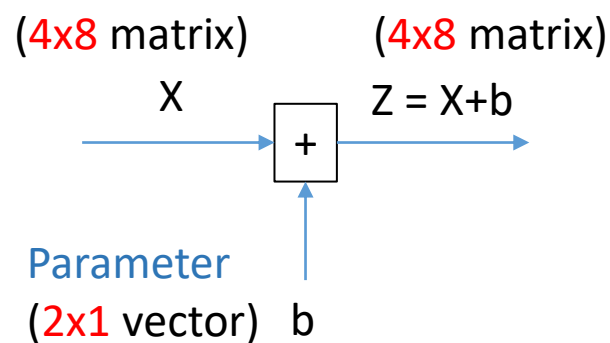
$k^{\text{th}}$  component of  $\delta Z$  vector

Writing in matrix-vector multiplication form

$$\delta W = X^T \delta Z$$

# Backprop derivation...

“Broadcast” addition:



$$\delta X_{i,j} = \sum_k \sum_l \frac{\partial Z_{k,l}}{\partial X_{i,j}} \delta Z_{k,l} = \sum_k \sum_l \frac{\partial}{\partial X_{i,j}} [X_{k,l} + b_k] \delta Z_{k,l} = \delta Z_{i,j} \quad \Rightarrow \quad \delta X = \delta Z$$

Chain rule

Substitute  $Z_{k,l}$

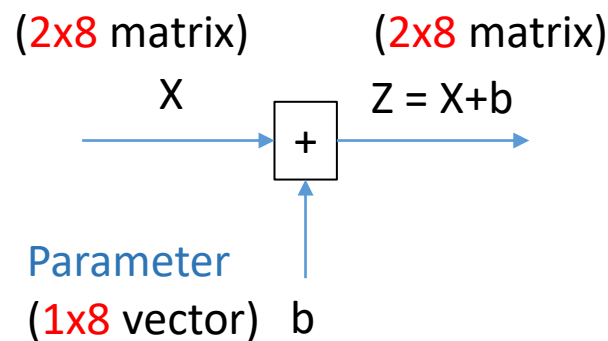
Because,

$$\frac{\partial}{\partial X_{i,j}} [X_{k,l} + b_k] = \begin{cases} 1, & \text{if } i = k \text{ and } j = l, \\ 0, & \text{otherwise.} \end{cases}$$



# Backprop derivation for broadcast addition

“Broadcast” addition:



$$\delta b_i = \sum_k \sum_l \frac{\partial Z_{k,l}}{\partial b_i} \delta Z_{k,l} = \sum_k \sum_l \frac{\partial}{\partial b_i} [X_{k,l} + b_l] \delta Z_{k,l} = \sum_k \delta Z_{k,i}$$

Chain rule

Substitute  $Z_{k,l}$

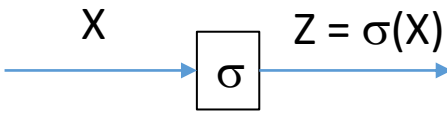
Because,

$$\frac{\partial}{\partial b_i} [X_{k,l} + b_l] = \begin{cases} 1, & \text{if } i = l, \\ 0, & \text{otherwise.} \end{cases}$$

$$\delta b = \sum_k \delta Z_{k,:}$$

# Backprop derivation for activation function

Non-linear function:  
(applied **pointwise**)



Using chain rule:  $\delta X_{i,j} = \frac{dZ_{i,j}}{dX_{i,j}} \delta Z_{i,j} = \frac{d\sigma(X_{i,j})}{dX_{i,j}} \delta Z_{i,j}$

If the non-linear  
function is sigmoid,

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

$$\frac{d\sigma}{da} = \frac{\exp(-a)}{(1 + \exp(-a))^2} = \frac{1}{1 + \exp(-a)} \left( 1 - \frac{1}{1 + \exp(-a)} \right) = \sigma(a)(1 - \sigma(a))$$

$$\delta X_{i,j} = \sigma(X_{i,j})(1 - \sigma(X_{i,j}))\delta Z_{i,j}$$

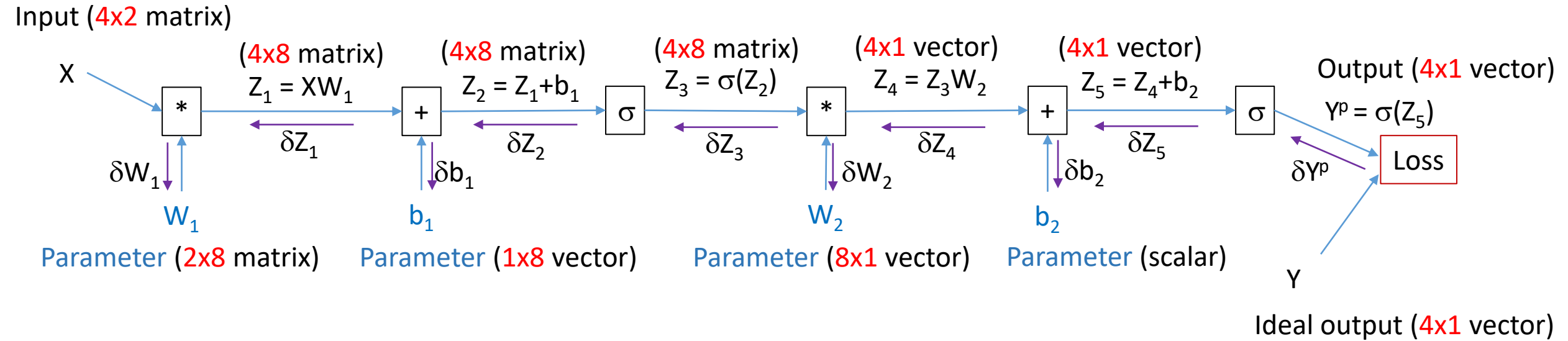
# Backprop derivation for loss function

Euclidean loss function:  $Loss(Y^p, Y) = \frac{1}{2} \|Y^p - Y\|^2 = \frac{1}{2} \sum_i (Y_i^p - Y_i)^2$

$i^{\text{th}}$  component of  $\delta Y^p$  vector:  $\delta Y_i^p = \frac{\partial}{\partial Y_i^p} Loss(Y^p, Y) = \frac{\partial}{\partial Y_i^p} \frac{1}{2} \sum_k (Y_k^p - Y_k)^2 = Y_i^p - Y_i$

Using vector notation:  $\delta Y^p = Y^p - Y$

# Apply chain rule to XOR neural network



Chain rule of derivatives:

$$\delta Y^p = Y^p - Y$$

$$\delta Z_5 = \sigma(Z_5)(1 - \sigma(Z_5))\delta Y^p$$

$$\delta Z_4 = \delta Z_5$$

$$\delta Z_3 = \delta Z_4 W_2^T$$

$$\delta Z_2 = \sigma(Z_2)(1 - \sigma(Z_2))\delta Z_3$$

$$\delta Z_1 = \delta Z_2$$

Gradient of "Loss" with respect to input signals



Propagates backward

$$\delta W_2 = Z_3^T \delta Z_4$$

$$\delta b_2 = \sum_k (\delta Z_5)_k$$

$$\delta W_1 = X^T \delta Z_1$$

$$\delta b_1 = \sum_k (\delta Z_2)_{k,:}$$

Gradient of "Loss" with respect to parameters

New notation:  
 $\delta S \equiv \nabla_S(\text{Loss})$


# Backprop to train a neural net

Initialize all parameters of the neural network

Initialize learning rate variable  $lr$

Iterate:

If loading the whole training data, do it **once** outside the “Iterate” loop, to be efficient



(Load Data): Get training data batch

(Forward pass): Compute  $Z_1, Z_2, \dots, Y^p$

(Backward pass): Compute gradients  $\delta Y^p, \delta Z_5, \dots, \delta Z_1, \delta W_2, \delta W_1, \delta b_2, \delta b_1$

(Gradient descent to update parameters):  $W_2 \leftarrow W_2 - lr * \delta W_2, \quad b_2 \leftarrow b_2 - lr * \delta b_2, \dots,$

(Diagnostics): Compute “Loss” from time to time to check if it is decreasing

“Learn\_XOR\_manualBP.ipnyb” implements this learning algorithm

# PyTorch magic!

- Fortunately, PyTorch can automatically compute derivatives by chain rules!
- Also, it has several optimizers that can use these derivatives in the gradient descent optimization method.
- Look at “Learn\_XOR.ipnyb” and “Learn\_XOR\_with\_LBFGS.ipnyb”

# Universal function approximation

- A neural network with a single hidden layer can approximate “any” function!
  - Wikipedia has a clear statement:  
[https://en.wikipedia.org/wiki/Universal\\_approximation\\_theorem](https://en.wikipedia.org/wiki/Universal_approximation_theorem)
- A non-technical explanation of universality theorem:
  - <http://neuralnetworksanddeeplearning.com/chap4.html>

Why do we then even need multiple layers and why even deep nets?