Introduction to Neural Networks and Backpropagation

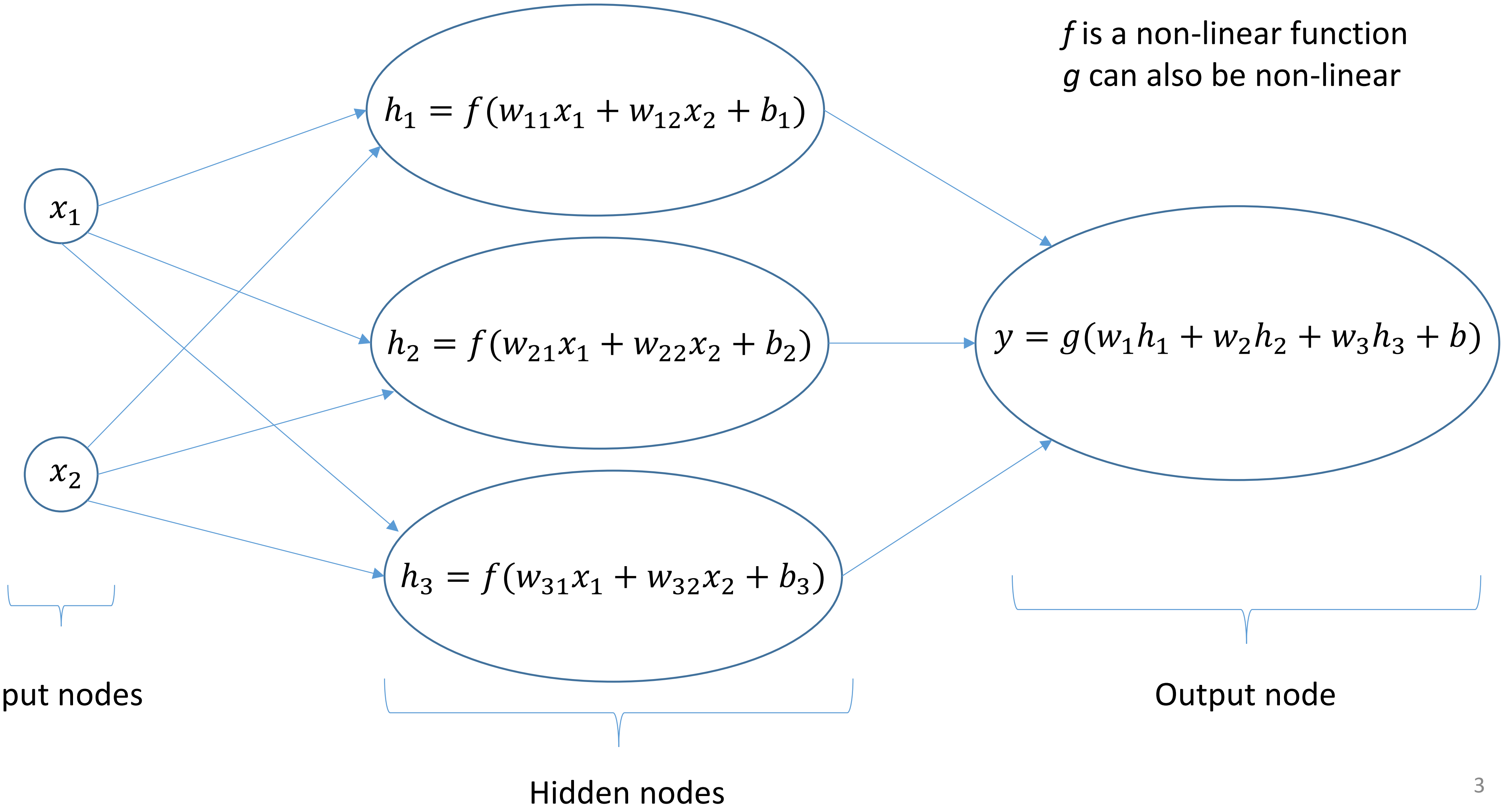
Computing Science
University of Alberta
Nilanjan Ray

Agenda

- What is a Neural Net?
  - Neural net as a computational graph
- Approximating “XOR” function with neural net
- Applying a neural net to classify MNIST
- Universal function approximation by a neural net
- (re)Introduction to gradient descent optimization
- Chain rule of derivatives
- Understanding backpropagation algorithm

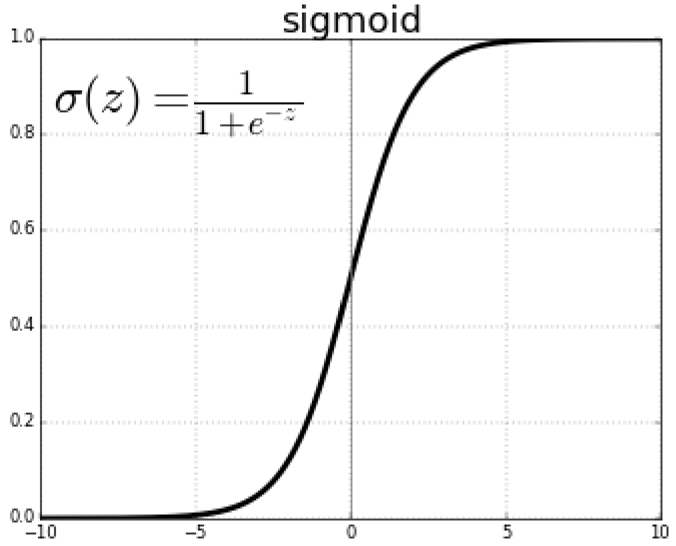
Feed forward neural network

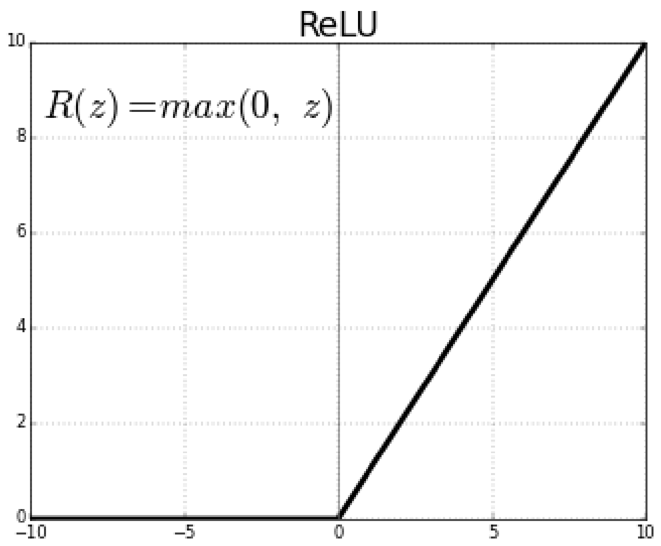
f is a non-linear function
g can also be non-linear



Feed forward net: non-linear functions

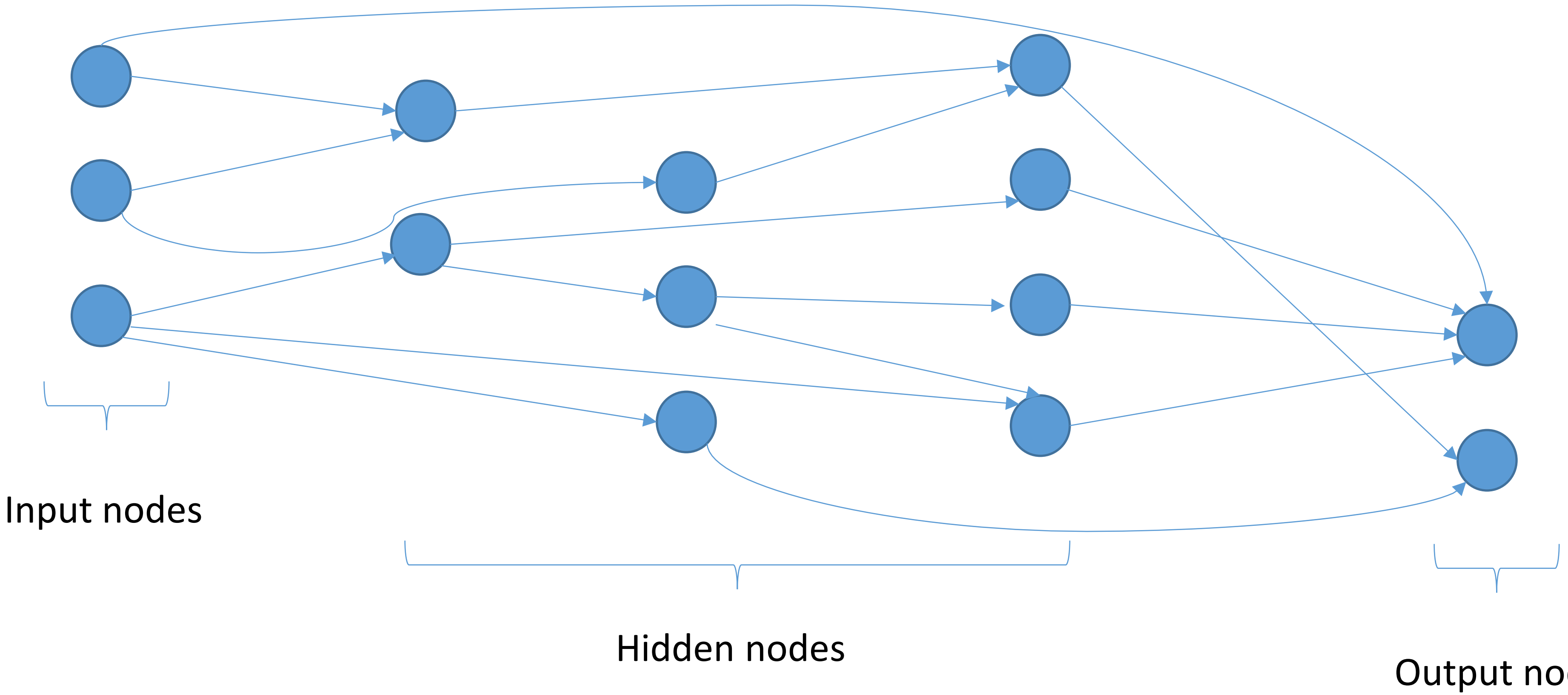
- Non-linear functions at hidden nodes are known as "activation function"
  - Sigmoid, ReLU, ELU, ….





Why activation functions are non-linear?

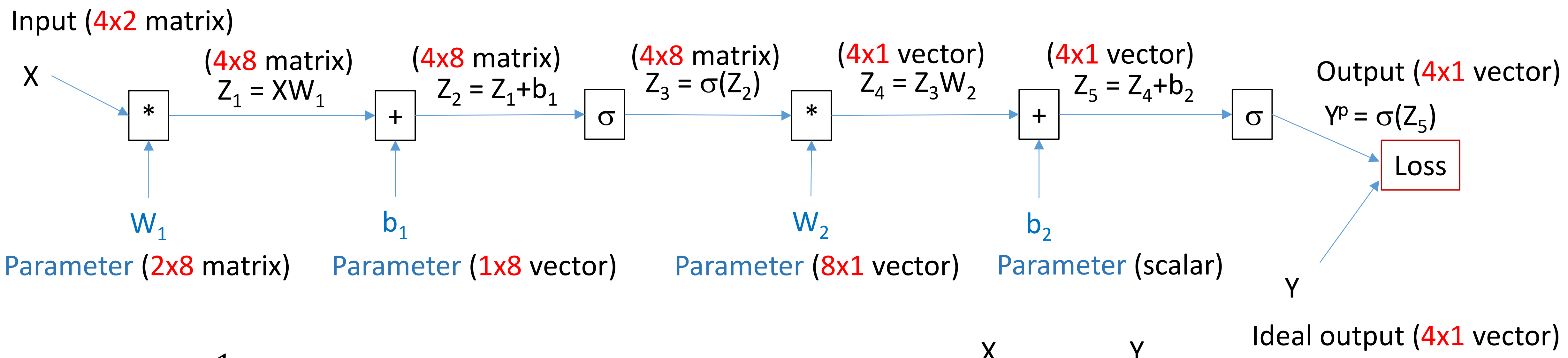
Feedforward net in general: Directed acyclic graph



What’s the big deal about neural net?

- Mathematically 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 paralleizable: 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

See Learn_XOR.ipynb

How does PyTorch optimize parameters?

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

Using PyTorch to Learn XOR

Learning algorithm has this basic structure as we have already seen in logistic regression

- Define an architecture for the neural network and instantiate it
- Instantiate an optimizer to adjust the parameters of the neural net
- Iterate
  - Load data (X, y)
  - Do a forward pass, i.e., compute output of neural net: $f(X;\theta)$
  - Do a backward pass:
    - Compute loss $L(f(X;\theta), y)$. The function L (loss) measures discrepancy between ground truth annotation y and the output of the neural net $f(X;\theta)$
    - Adjust parameters θ of the neural network to reduce loss value
  - Diagnostic: from time to time print loss value

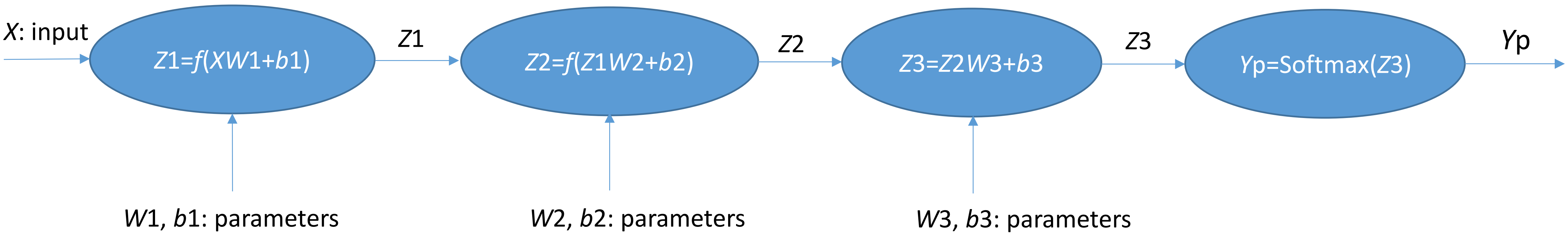
MNIST classification problem

Small 28 pixels-by-28 pixels images of hand written digits

The visual recognition problem definition:
to recognize the digit from an image

Pixel values (feature)				Digit: 1-hot vector		
x_1	x_2	…	x_{784}	y_1	…	y_{10}
0.1	0.3	…	0.0	0	…	1
0.2	0.1	…	0.5	1		0
…	…	…	…	…	…	…
…	…	…	…	…	…	…
0.0	0.98	…	0.8	0	…	1
0.5	0.25	…	0.36	?	…	?
0.1	0.95	…	0.1	?	…	?

NN Architecture for MNIST Classification



Activation function, f is ReLU in our implementation

Learning MNIST NN with Backprop and SGD

Initialize all parameters of the neural network
Initialize learning rate variable *lr*

Iterate:

(Load Data): Get training data batch *X*

(Forward pass): Compute *Z*1, *Z*2, Z3, *Y*p

(Compute loss): Compute a suitable loss between ideal output *Y* and output of NN *Yp*

(Backward pass): Ask PyTorch optimizer to adjust neural network parameters

(Diagnostics): Compute loss on training and validation sets

MNIST_NN.ipynb

Neural Net as Universal Function Approximator

- http://neuralnetworksanddeeplearning.com/index.html

Gradient Descent: PyTorch under the hood

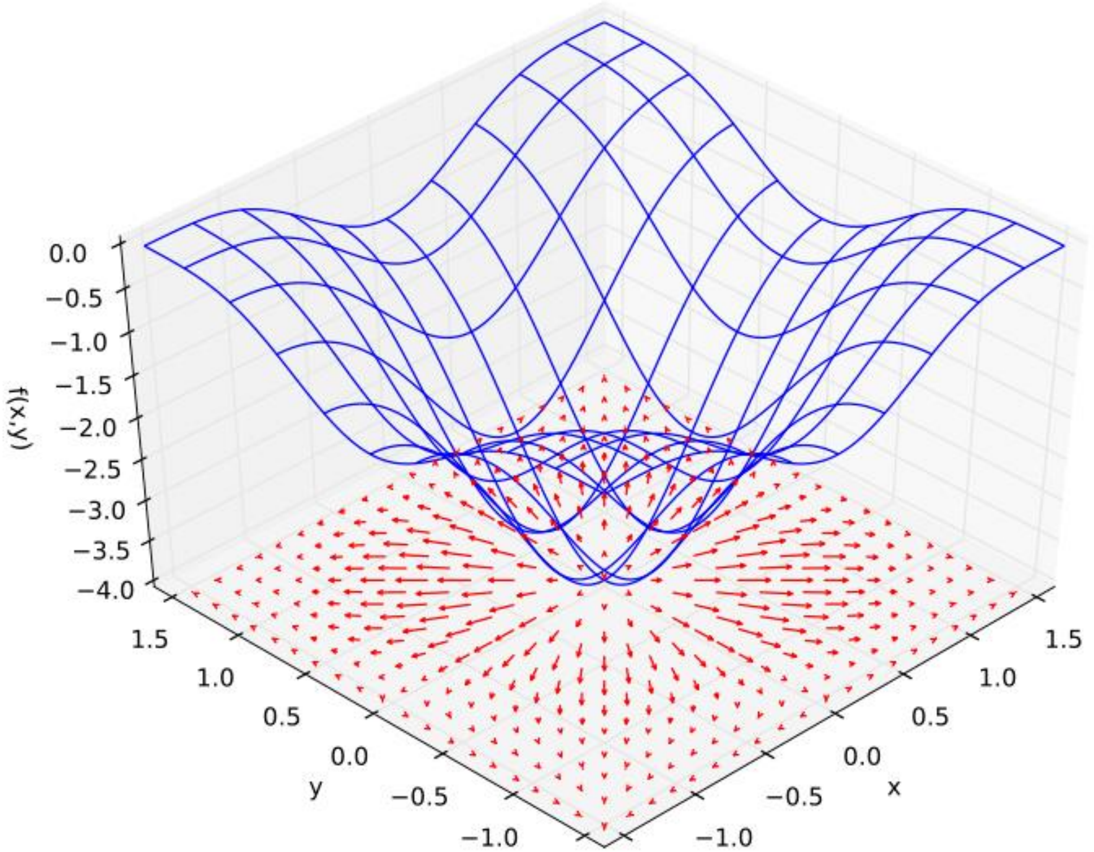
- How does PyTorch optimizes parameters of a model to reduce loss value?
  - Using gradient descent
- We will apply GD to multiple linear regression
- Then we will move on to using it for a neural net
  - We must learn how to use the chain rule of differentiation

Gradient of a function

Example:

$$f(x,y) = -(cos^2 x + cos^2 y)^2$$

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 4\left(cos^2(x) + cos^2(y)\right)\cos(x)\sin(x) \\ 4\left(cos^2(x) + cos^2(y)\right)\cos(y)\sin(y) \end{bmatrix}$$



Note 1: *f* is a function of two variables,
so gradient of *f* is a two dimensional vector

Note 2: Gradient (vector) of *f* points toward the steepest ascent for *f*

Note 3: At a (local) minimum of *f* its gradient becomes a zero vector

Example source: Wikipedia

Gradient descent optimization

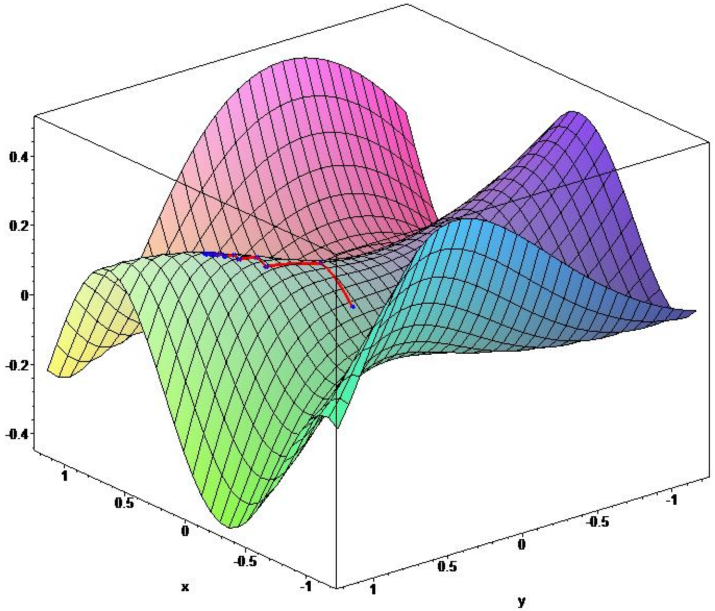
Start at an initial guess for the optimization variable: $\mathbf{x}_0$

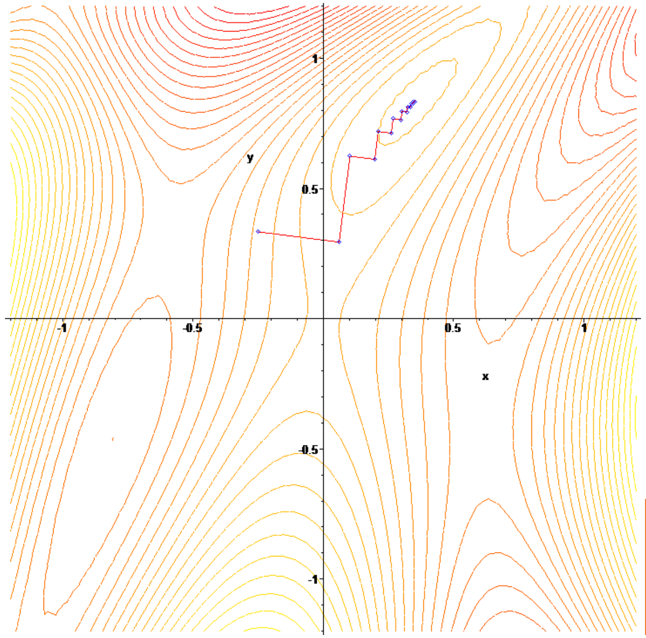
Iterate until gradient magnitude becomes too small: $\mathbf{x}^{t+1} = \mathbf{x}^t - \alpha \nabla f(\mathbf{x}^t)$

Gradient descent algorithm

α is called the step-length.

f(x, y)





Gradient descent creates a zig-zag path leading to a local minimum of f

Look at GradientDescentDemo.ipynb

Picture source: Wikipedia

PyTorch optimizer uses GD

Let's try our own gradient descent for multiple linear regression

Gradient of loss function for multiple linear regression: $\boxed{\nabla_W L = (X^T X + \gamma I)W - X^T Y}$

$$\boxed{\nabla_b L = \sum_{i=1}^{n} (y_i^p - y_i)}$$

Exercise: write GD for MNIST multiple linear regression

For implementation of this GD, look at MNIST_Multiple_Linear_Regression.ipynb

How do we apply GD for learning a neural net?

We need to compute gradient of the loss function with respect to all parameters in a neural net:

$$\delta\theta_i \equiv \nabla_{\theta_i} L(y^p, y)$$

Parameter in the i^{th} layer

Ground truth/tag

Output (aka prediction) from neural net

Once we have this loss gradient, we can adjust parameters using gradient descent rule:

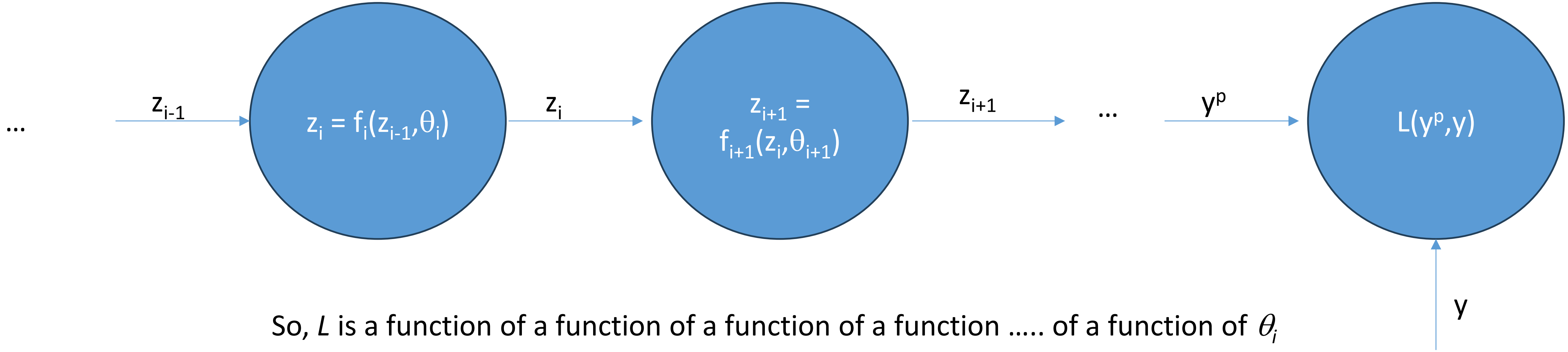
$$\theta_i = \theta_i - \alpha\delta\theta_i$$

Learning rate/step size

How do we apply GD for learning a neural net?

We need to compute gradient of the loss function with respect to all parameters in a neural net:

$$\delta\theta_i \equiv \nabla_{\theta_i} L(y^p, y)$$



So, L is a function of a function of a function of a function ..... of a function of θ_i

Therefore, we need chain rule of derivative to compute $\delta\theta_i$

To apply chain rule of derivative in a neural net…

- We need to understand chain rule of derivative for multivariate functions: Jacobian vector product
- We also need to understand the notion of a computational graph and how to apply Jacobian vector product to a computational graph
- These components will lead us to the well acclaimed backpropagation algorithm for learning parameters of a neural net using GD

Example gradient computations

- Let's consider the following function of four variables:

$$f(x_1, x_2, x_3, x_4) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 + 2x_3)^4 + 10(x_1 - x_4)^4$$

- Let's compute derivative (gradient) of this function at

$$[x_1, x_2, x_3, x_4] = [3, -1, 0, 1]$$

- Cross-verify PyTorch partial derivative computations with math formulas
- Gradient descent optimization

Look into Understanding_chain_rule.ipynb

Chain rule of derivatives

- Let consider the same function as before:

$$f(x_1, x_2, x_3, x_4) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 + 2x_3)^4 + 10(x_1 - x_4)^4$$

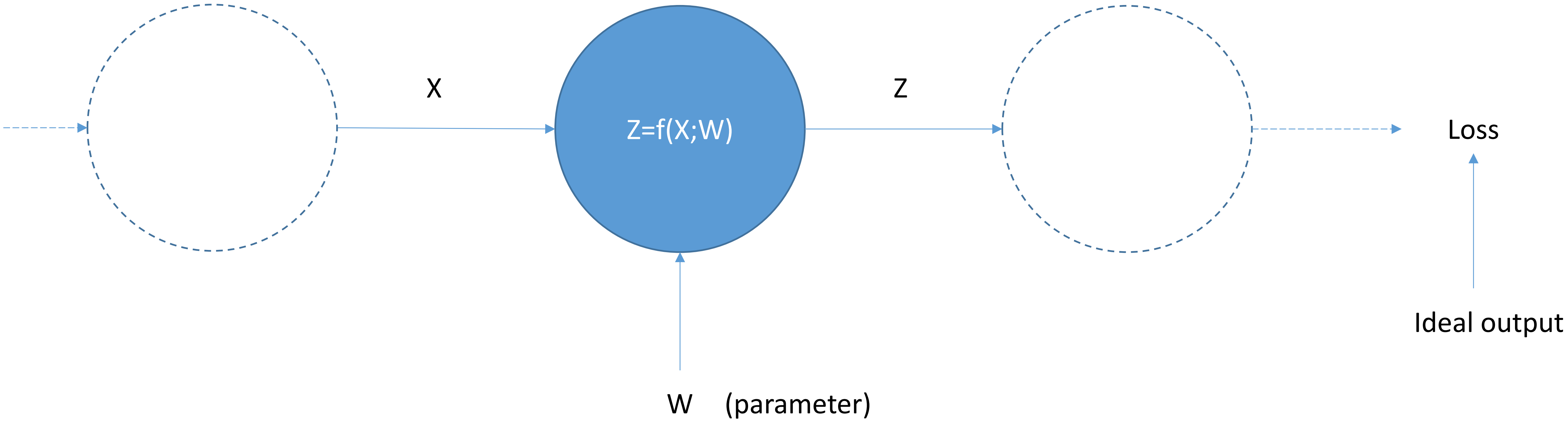
- But this time *x* is a (vector-valued) function of two variables z_1 and z_2:

$$\begin{gathered} x_1 = z_1 - z_2, \\ x_2 = z_1^2, \\ x_3 = z_2^2, \\ x_4 = z_1^2 + z_1 z_2 \end{gathered}$$

- Let's compute gradient of *f* with respect to *z* using chain rule: Jacobian vector product

Look into Understanding_chain_rule.ipynb

Chain rule of derivative for a computational node



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

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

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

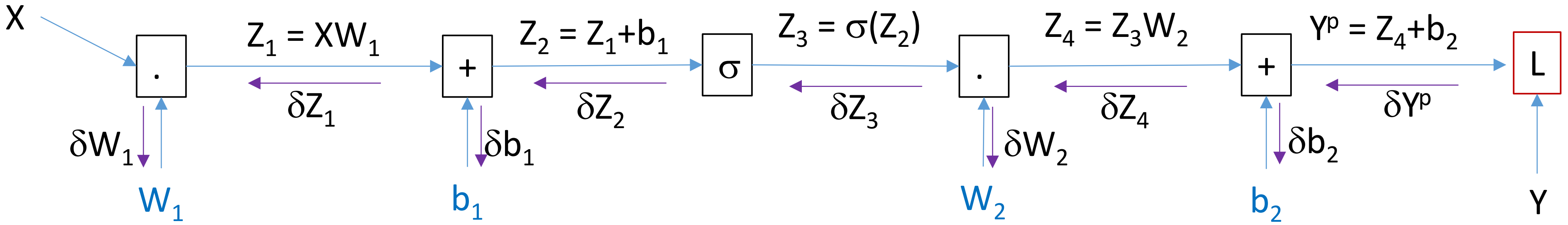
OK, let's apply chain rule to a computational graph where all variables and parameters are scalars

So, our scalar neural net is: $$Y^p = \sigma(XW_1 + b_1)W_2 + b_2$$

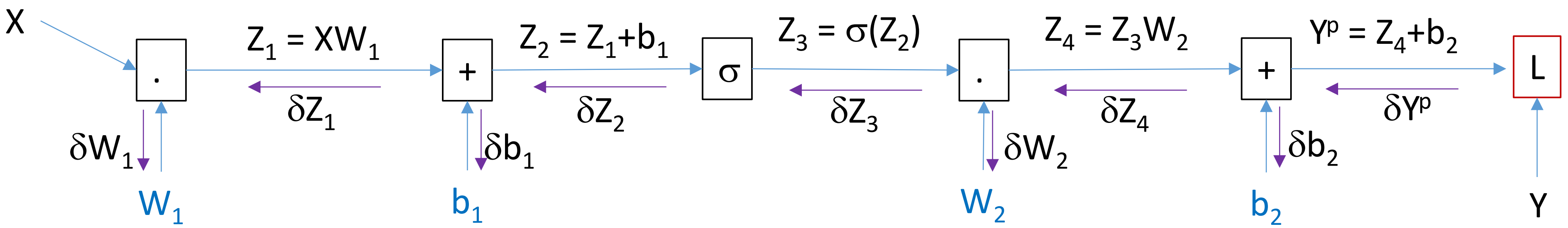
with a square (aka Euclidean) loss function: $$L(Y^p, Y) = \frac{1}{2}(Y^p - Y)^2$$

As usual, X is the input, Y^p is the output, and W_1, b_1, W_2, b_2 are parameters of the neural net, σ is a non-linear function.

The computational graph for this scalar neural net is (also showing loss gradient symbols):



Chain rule for a scalar neural net…



1 $$\delta Y^p \equiv \frac{\partial L}{\partial Y^p} = \frac{\partial}{\partial Y^p}\left[\frac{1}{2}(Y^p - Y)^2\right] = y^p - y$$

2 $$\delta Z_4 \equiv \frac{\partial L}{\partial Z_4} = \frac{\partial Y^p}{\partial Z_4}\frac{\partial L}{\partial Y^p} = \delta Y^p$$

Because $Y^p = Z_4 + b_2, \dfrac{\partial Y^p}{\partial Z_4} = 1$

3 $$\delta Z_3 \equiv \frac{\partial L}{\partial Z_3} = \frac{\partial Z_4}{\partial Z_3}\frac{\partial L}{\partial Z_4} = W_2 \delta Z_4$$

Because $Z_4 = Z_3 W_2, \dfrac{\partial Z_4}{\partial Z_3} = W_2$

4 $$\delta Z_2 \equiv \frac{\partial L}{\partial Z_2} = \frac{\partial Z_3}{\partial Z_2}\frac{\partial L}{\partial Z_3} = \sigma'(Z_2)\delta Z_3$$

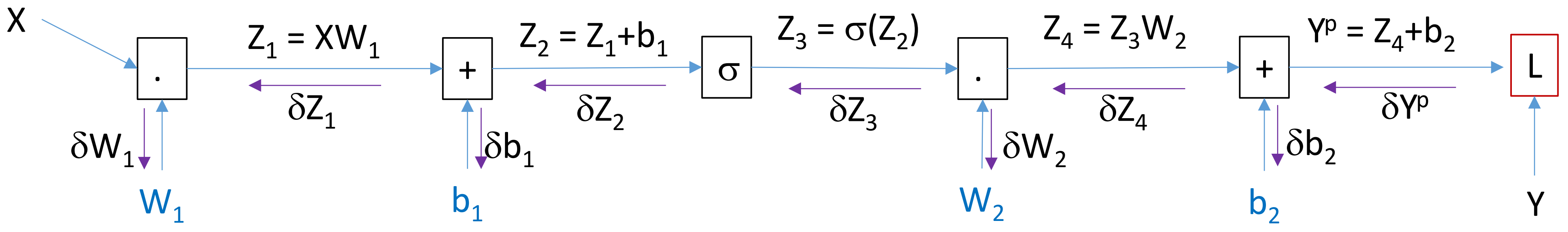
Because $Z_3 = \sigma(Z_2), \dfrac{\partial Z_3}{\partial Z_2} = \sigma'(Z_2)$

5 $$\delta Z_1 \equiv \frac{\partial L}{\partial Z_1} = \frac{\partial Z_2}{\partial Z_1}\frac{\partial L}{\partial Z_2} = \delta Z_2$$

Because $Z_2 = Z_1 + b_1, \dfrac{\partial Z_2}{\partial Z_1} = 1$

But we need loss derivatives with respect to parameters…

Loss derivatives w.r.t. parameters



1 $$\delta W_1 \equiv \frac{\partial L}{\partial W_1} = \frac{\partial Z_1}{\partial W_1}\frac{\partial L}{\partial Z_1} = X\delta Z_1$$

Because $Z_1 = XW_1, \frac{\partial Z_1}{\partial W_1} = X$

2 $$\delta b_1 \equiv \frac{\partial L}{\partial b_1} = \frac{\partial Z_2}{\partial b_1}\frac{\partial L}{\partial Z_2} = \delta Z_2$$

Because $Z_2 = Z_1 + b_1, \frac{\partial Z_2}{\partial b_1} = 1$

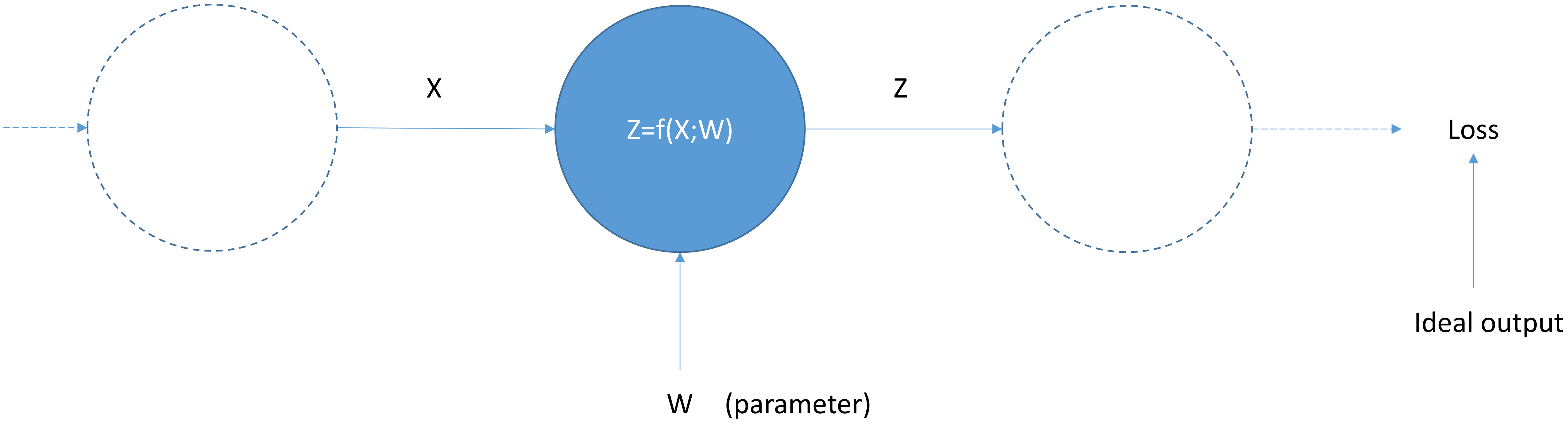
3 $$\delta W_2 \equiv \frac{\partial L}{\partial W_2} = \frac{\partial Z_4}{\partial W_2}\frac{\partial L}{\partial Z_4} = Z_3\delta Z_4$$

Because $Z_4 = Z_3W_2, \frac{\partial Z_4}{\partial W_2} = Z_3$

4 $$\delta b_2 \equiv \frac{\partial L}{\partial b_2} = \frac{\partial Y^p}{\partial b_2}\frac{\partial L}{\partial Y^p} = \delta Y^p$$

Because $Y^p = Z_4 + b_2, \frac{\partial Y^p}{\partial b_2} = 1$

Chain rule of derivative…



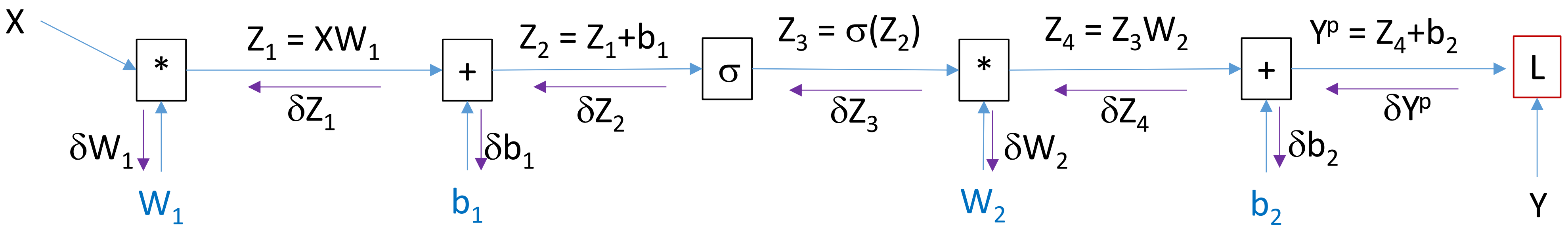
If X, Z, W are matrices or vectors, then :

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

"*" refers to matrix vector multiplication

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

Chain rule for a (general) neural net



1. $$\delta Y^p \equiv \frac{\partial L}{\partial Y^p} = \frac{\partial}{\partial Y^p}\left[\frac{1}{2}||Y^p - Y||^2\right] = y^p - y$$

2. $$\delta Z_4 \equiv \frac{\partial L}{\partial Z_4} = \frac{\partial Y^p}{\partial Z_4}\frac{\partial L}{\partial Y^p} = \delta Y^p$$

Because $Y^p = Z_4 + b_2, \dfrac{\partial Y^p}{\partial Z_4} = 1$

3. $$\delta Z_3 \equiv \frac{\partial L}{\partial Z_3} = \frac{\partial Z_4}{\partial Z_3}\frac{\partial L}{\partial Z_4} = \delta Z_4 W_2^T$$

Because $Z_4 = Z_3 W_2, \dfrac{\partial Z_4}{\partial Z_3} = W_2^T$

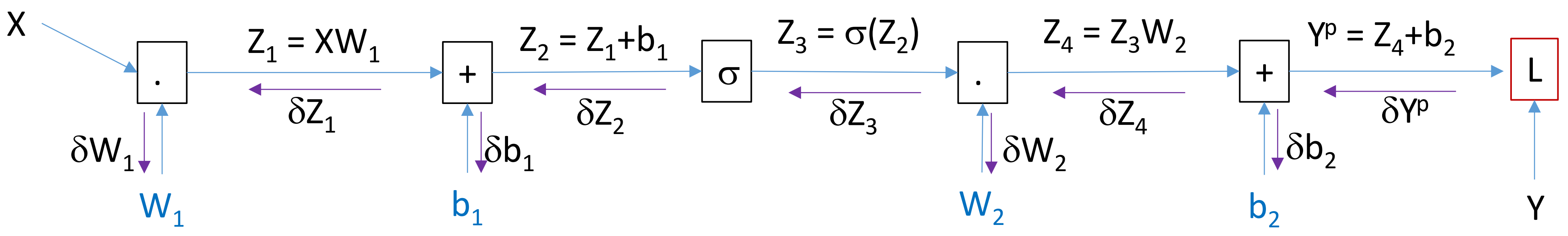
4. $$\delta Z_2 \equiv \frac{\partial L}{\partial Z_2} = \frac{\partial Z_3}{\partial Z_2}\frac{\partial L}{\partial Z_3} = \sigma'(Z_2).\,\delta Z_3$$

Because $Z_3 = \sigma(Z_2), \dfrac{\partial Z_3}{\partial Z_2} = \sigma'(Z_2)$

5. $$\delta Z_1 \equiv \frac{\partial L}{\partial Z_1} = \frac{\partial Z_2}{\partial Z_1}\frac{\partial L}{\partial Z_2} = \delta Z_2$$

Because $Z_2 = Z_1 + b_1, \dfrac{\partial Z_2}{\partial Z_1} = 1$

Loss derivatives w.r.t. matrix or vector parameters



1

$$\delta W_1 \equiv \frac{\partial L}{\partial W_1} = \frac{\partial Z_1}{\partial W_1}\frac{\partial L}{\partial Z_1} = X^T \delta Z_1$$

Because $Z_1 = XW_1, \frac{\partial Z_1}{\partial W_1} = X^T$

2

$$\delta b_1 \equiv \frac{\partial L}{\partial b_1} = \frac{\partial Z_2}{\partial b_1}\frac{\partial L}{\partial Z_2} = \sum_k (\delta Z_2)_{k,:}$$

Because $Z_2 = Z_1 + b_1, \frac{\partial Z_2}{\partial b_1} = [1, \ldots, 1]$

3

$$\delta W_2 \equiv \frac{\partial L}{\partial W_2} = \frac{\partial Z_4}{\partial W_2}\frac{\partial L}{\partial Z_4} = Z_3^T \delta Z_4$$

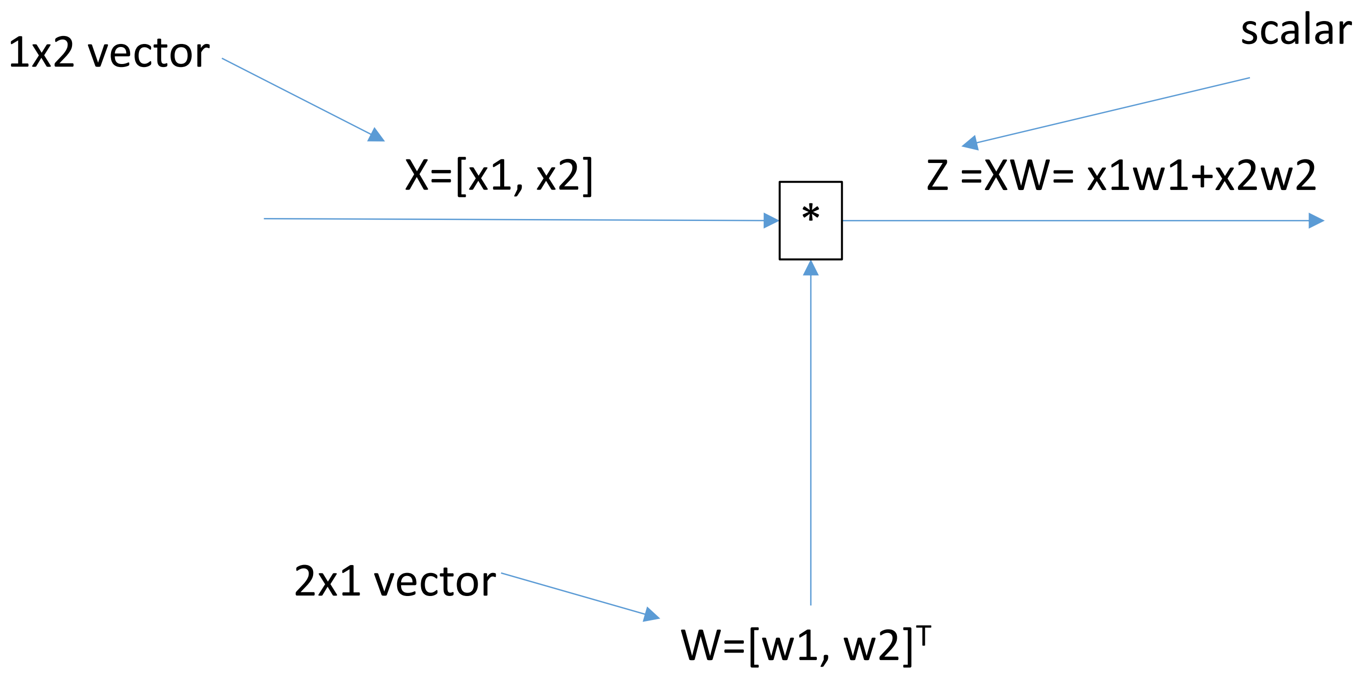
Because $Z_4 = Z_3 W_2, \frac{\partial Z_4}{\partial W_2} = Z_3^T$

4

$$\delta b_2 \equiv \frac{\partial L}{\partial b_2} = \frac{\partial Y^p}{\partial b_2}\frac{\partial L}{\partial Y^p} = \sum_k (\delta Y^p)_{k,:}$$

Because $Y^p = Z_4 + b_2, \frac{\partial Y^p}{\partial b_2} = [1, \ldots, 1]$

Example 1



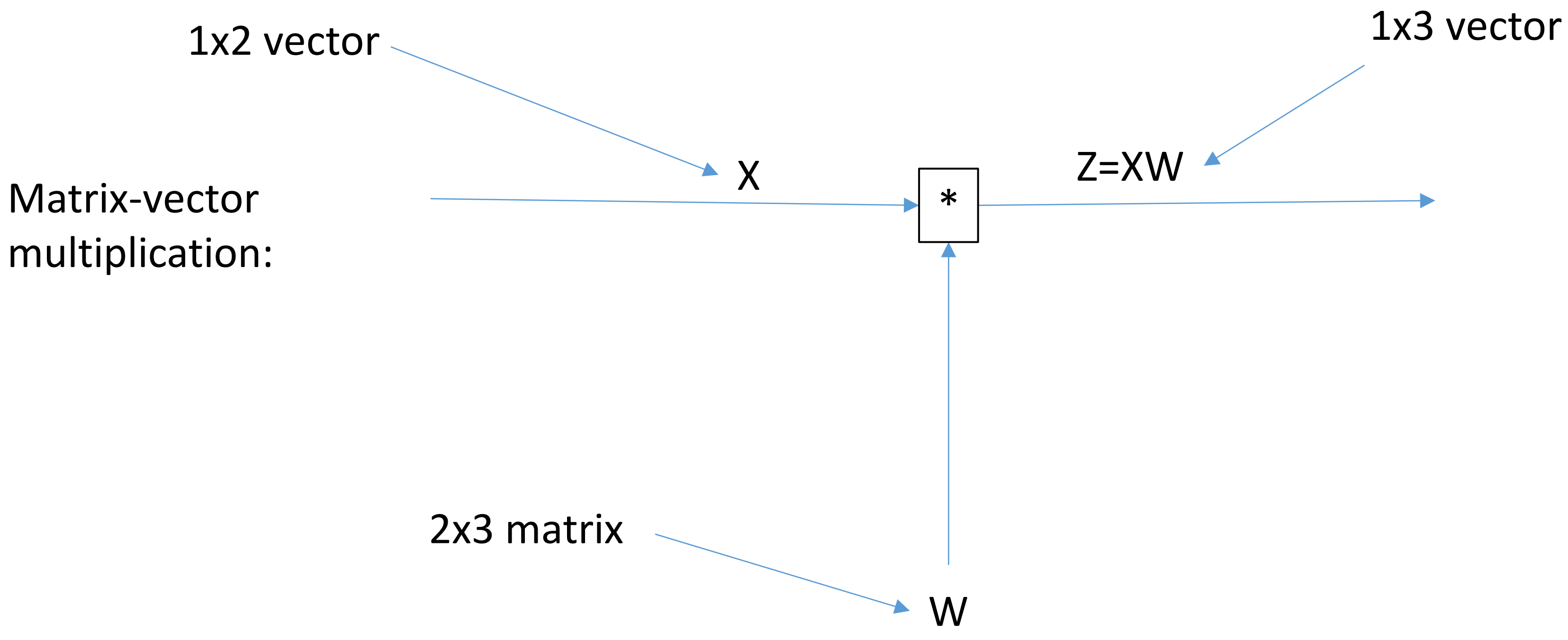
Chain rules:

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

Why?

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

Example 2



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

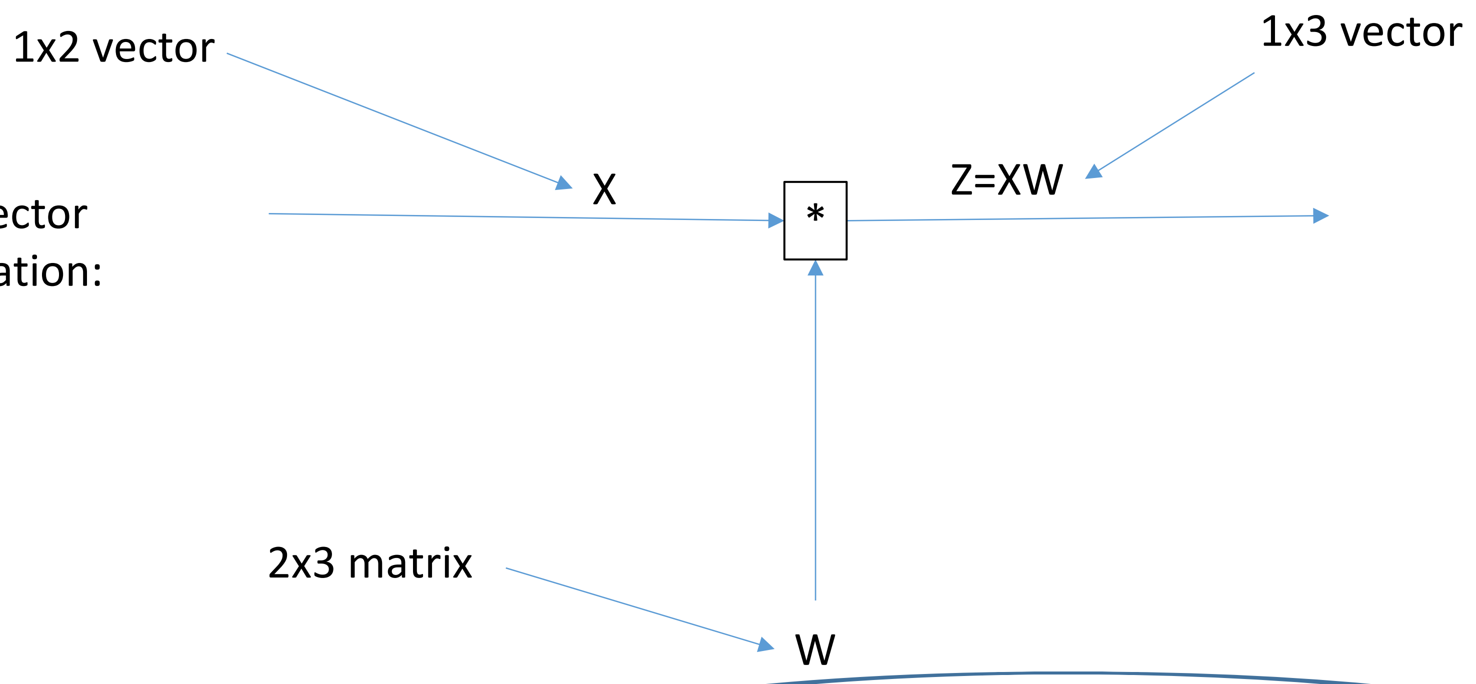
Chain rules:

Why?

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

Backprop derivation

Matrix-vector multiplication:



Chain rules:

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

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

Using short notation for gradients:

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

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

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

Backprop derivation…

$$\delta X_i = \underbrace{\sum_k \frac{\partial Z_k}{\partial X_i} \frac{\partial(\text{Loss})}{\partial Z_k}}_{\text{Chain rule of derivative}} = \sum_k \underbrace{\frac{\partial}{\partial X_i}\left[\sum_j X_j W_{jk}\right]}_{\text{Substitute } Z_k} \delta Z_k = \sum_k W_{ik} \delta Z_k \Rightarrow \delta X = \delta Z W^T$$

i^{th} component of δX vector

Chain rule of derivative

Substitute Z_k

k^{th} component of δZ vector

Because,

$$\frac{\partial}{\partial X_i}\left[\sum_j X_j W_{jk}\right] = W_{ik}$$

Writing in matrix-vector multiplication form

Backprop derivation...

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

$(i,j)^{th}$ component of δW matrix

k^{th} component of δZ vector

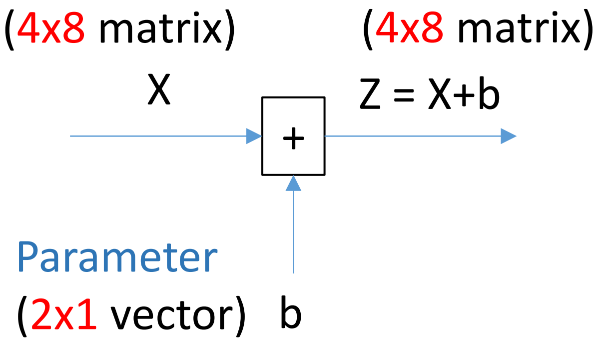
Because,

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

Writing in matrix-vector multiplication form

Backprop derivation…

"Broadcast" addition:



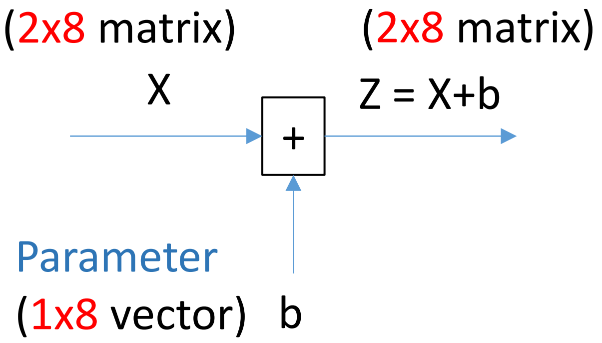
$$\delta X_{i,j} = \underbrace{\sum_k \sum_l \frac{\partial Z_{k,l}}{\partial X_{i,j}} \delta Z_{k,l}}_{\text{Chain rule}} = \sum_k \sum_l \frac{\partial}{\partial X_{i,j}} \underbrace{\left[X_{k,l} + b_k\right]}_{\text{Substitute } Z_{k,l}} \delta Z_{k,l} = \delta Z_{i,j} \quad \Rightarrow \quad \delta X = \delta Z$$

Because,

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

Backprop derivation for broadcast addition

"Broadcast" addition:



$$\delta b_i = \underbrace{\sum_k \sum_l \frac{\partial Z_{k,l}}{\partial b_i}}_{\text{Chain rule}} \delta Z_{k,l} = \sum_k \sum_l \frac{\partial}{\partial b_i} \underbrace{[X_{k,l} + b_l]}_{\text{Substitute } Z_{k,l}} \delta Z_{k,l} = \sum_k \delta Z_{k,i} \Longrightarrow \delta b = \sum_k \delta Z_{k,:}$$

Because,

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

Backprop derivation for activation function

Non-linear function:
(applied pointwise)

$$X \longrightarrow \boxed{\sigma} \longrightarrow Z = \sigma(X)$$

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

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

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

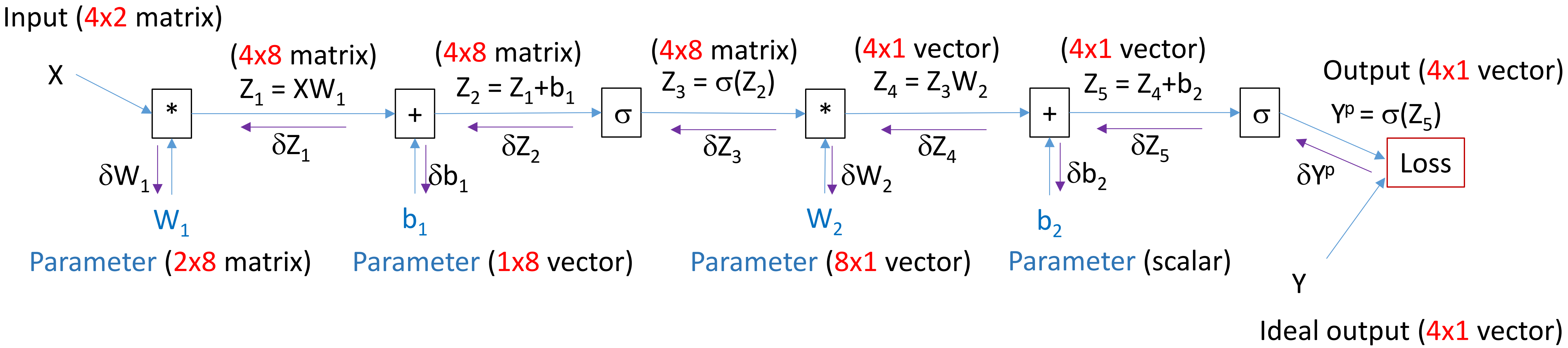
Backprop derivation for loss function

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

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

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

Apply chain rule to XOR neural network



Chain rule of derivatives:

$$\left.\begin{aligned}
\delta Y^p &= Y^p - Y \\
\delta Z_5 &= \sigma(Z_5).\left(1-\sigma(Z_5)\right).\delta Y^p \\
\delta Z_4 &= \delta Z_5 \\
\delta Z_3 &= \delta Z_4 W_2^T \\
\delta Z_2 &= \sigma(Z_2).\left(1-\sigma(Z_2)\right).\delta Z_3 \\
\delta Z_1 &= \delta Z_2
\end{aligned}\right\}$$

Gradient of "Loss" with respect to input signals

Propagates backward

$$\left.\begin{aligned}
\delta W_2 &= Z_3^T \delta Z_4 \\
\delta b_2 &= \sum_k (\delta Z_5)_k \\
\delta W_1 &= X^T \delta Z_1 \\
\delta b_1 &= \sum_k (\delta Z_2)_{k,:}
\end{aligned}\right\}$$

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,…,Y^p

(Backward pass): Compute gradients δY^p, δZ_5,…, δZ_1, δW_2, δW_1, δb_2, δb_1

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

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

"Learn_XOR_manualBP.ipnyb" implements this learning algorithm

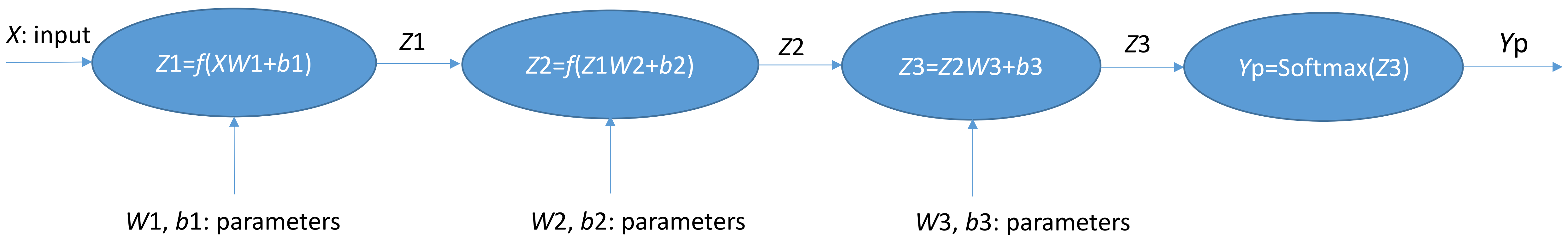
MNIST classification problem

Small 28 pixels-by-28 pixels images of hand written digits

The visual recognition problem definition:
to recognize the digit from an image

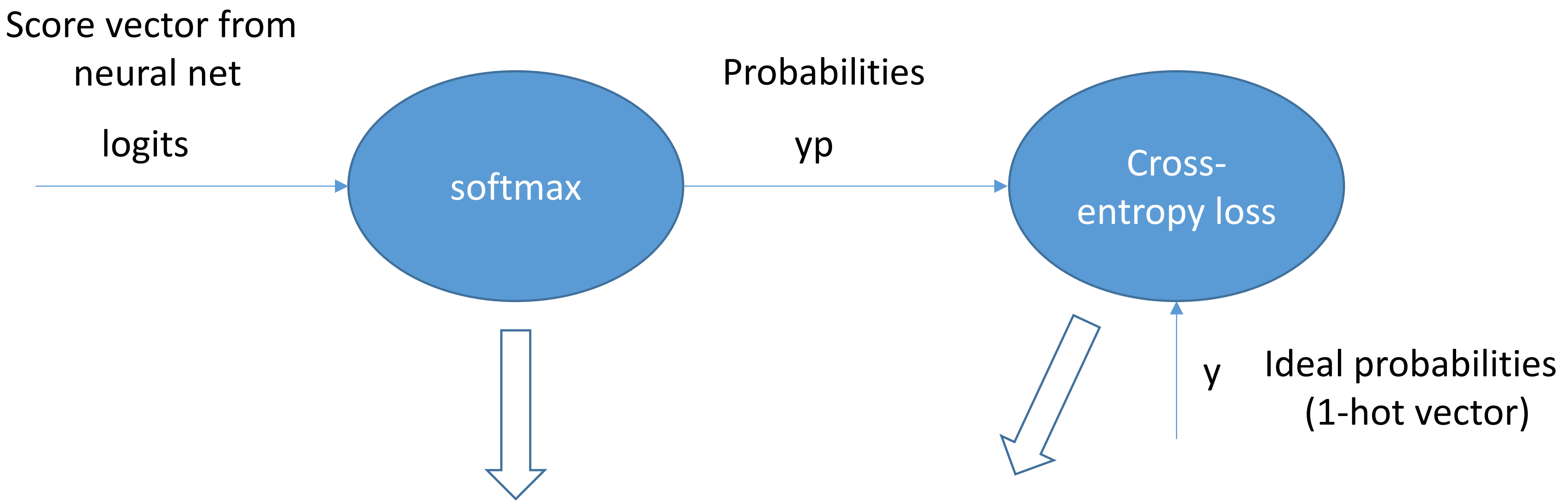
Pixel values (feature)				Digit: 1-hot vector		
x_1	x_2	…	x_{784}	y_1	…	y_{10}
0.1	0.3	…	0.0	0	…	1
0.2	0.1	…	0.5	1		0
…	…	…	…	…	…	…
…	…	…	…	…	…	…
0.0	0.98	…	0.8	0	…	1
0.5	0.25	…	0.36	?	…	?
0.1	0.95	…	0.1	?	…	?

NN Architecture for MNIST Classification



Activation function, *f* is ReLU in our implementation

Softmax and cross-entropy loss

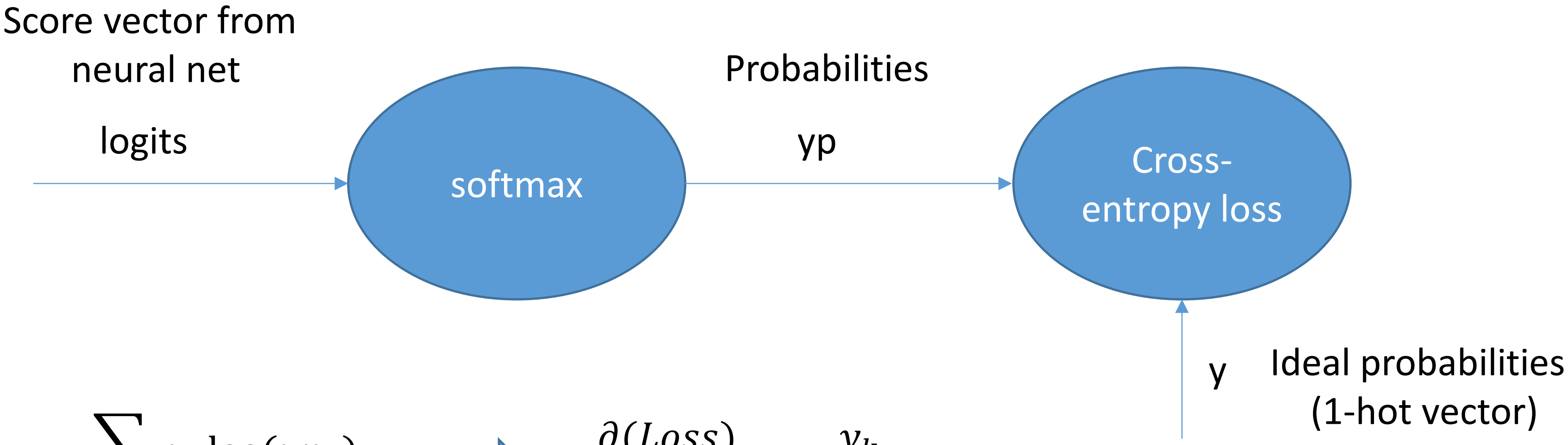


$$yp_i = \frac{\exp(logits_i)}{\sum_k \exp(logits_k)}$$

$$Loss = -\sum_k y_k \log(yp_k)$$

To backpropagate error, we need to compute: $\delta(logits)_i \equiv \frac{\partial(Loss)}{\partial(logits)_i}$

Softmax and cross-entropy loss: backprop



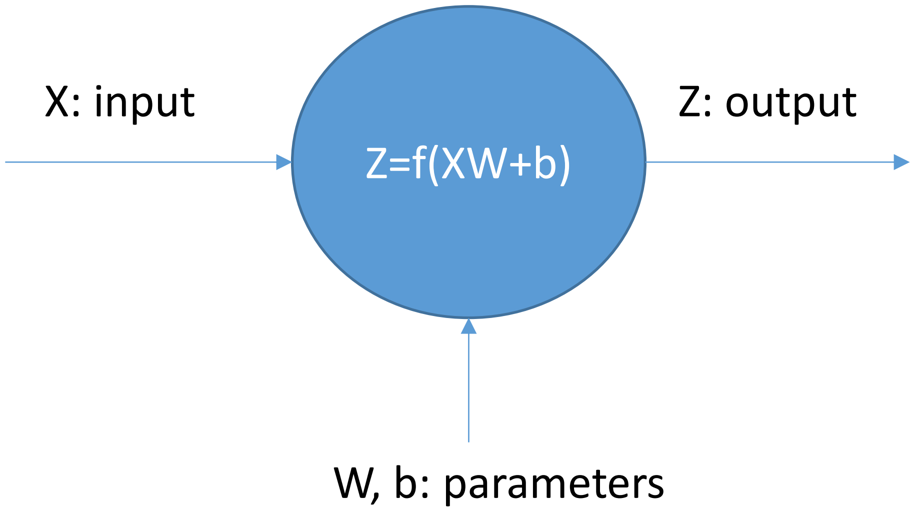
$$Loss = -\sum_k y_k \log(yp_k) \Longrightarrow \frac{\partial(Loss)}{\partial(yp)_k} = -\frac{y_k}{yp_k}$$

$$yp_i = \frac{\exp(logits_i)}{\sum_k \exp(logits_k)} \Longrightarrow \frac{\partial(yp)_k}{\partial(logits)_i} = \begin{cases} yp_i(1 - yp_i), & \text{if } i = k, \\ -yp_i yp_k, & \text{otherwise.} \end{cases}$$

Using the above two results in the chain rule, $\delta(logits)_i \equiv \dfrac{\partial(Loss)}{\partial(logits)_i} = \sum_k \dfrac{\partial(yp)_k}{\partial(logits)_i} \dfrac{\partial(Loss)}{\partial(yp)_k} = yp_i - y_i$

What if, instead of cross-entropy, we used L2 loss along with softmax?

Backprop across a neural net layer



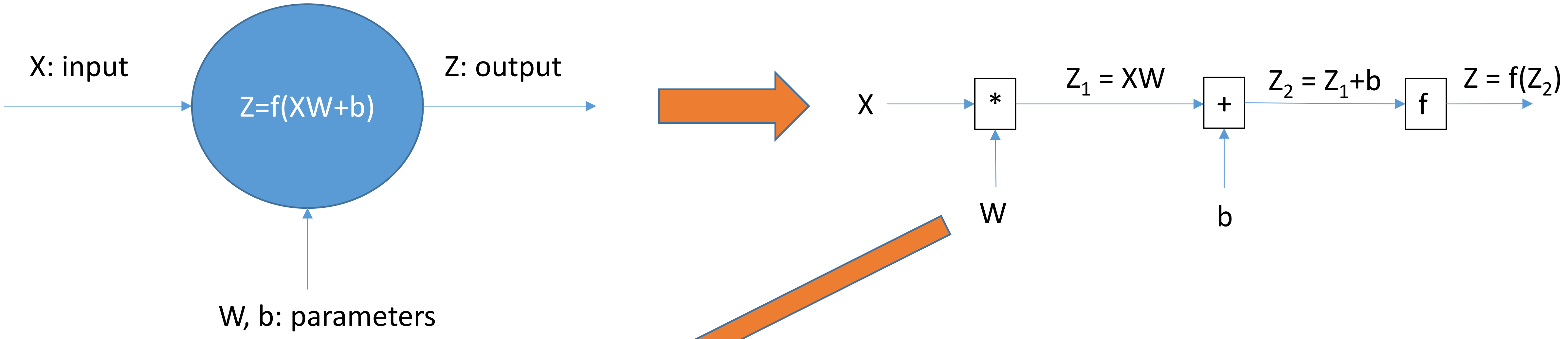
Pointwise multiplication

Backprop rules:

$$\delta X = [f'(XW + b) \cdot \delta Z]W^T \qquad \delta W = X^T[f'(XW + b) \cdot \delta Z] \qquad \delta b = \sum_l [f'(XW + b) \cdot \delta Z]_{l,:}$$

Matrix multiplication

Backprop across a neural net layer: derivation



Apply chain rule to computational nodes:

$$\delta Z_2 = f'(Z_2)\delta Z$$

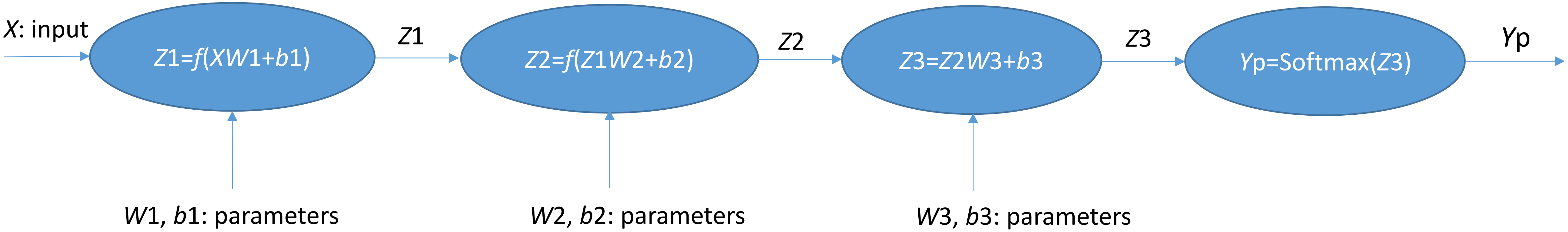
$$\delta Z_1 = \delta Z_2$$

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

$$\delta X = [f'(XW + b) \cdot \delta Z]W^T$$

Similarly, we can derive backprop rules for δW and δb

NN for MNIST Classification: Gradients and Manual Backprop



Backprop:

$\delta Z3 = Yp\text{-}Y$

$\delta Z2 = (\delta Z3)W3^T$

$\delta Z1 = [f'(Z1W2+b2).\delta Z2]W2^T$

$$\delta W3 = (Z2^T)\delta Z3$$

$$\delta W2 = Z1^T[f'(Z1W2 + b2) \cdot \delta Z2]$$

$$\delta W1 = X^T[f'(XW1 + b1) \cdot \delta Z1]$$

$$\delta b3 = \sum_l [\delta Z3]_{l,:}$$

$$\delta b2 = \sum_l [f'(Z1W2 + b2) \cdot \delta Z2]_{l,:}$$

$$\delta b1 = \sum_l [f'(XW1 + b1) \cdot \delta Z1]_{l,:}$$

MNIST_NN_manualBP.ipynb