KOREATECH 한국기술교육대학교

Learning & Autograd

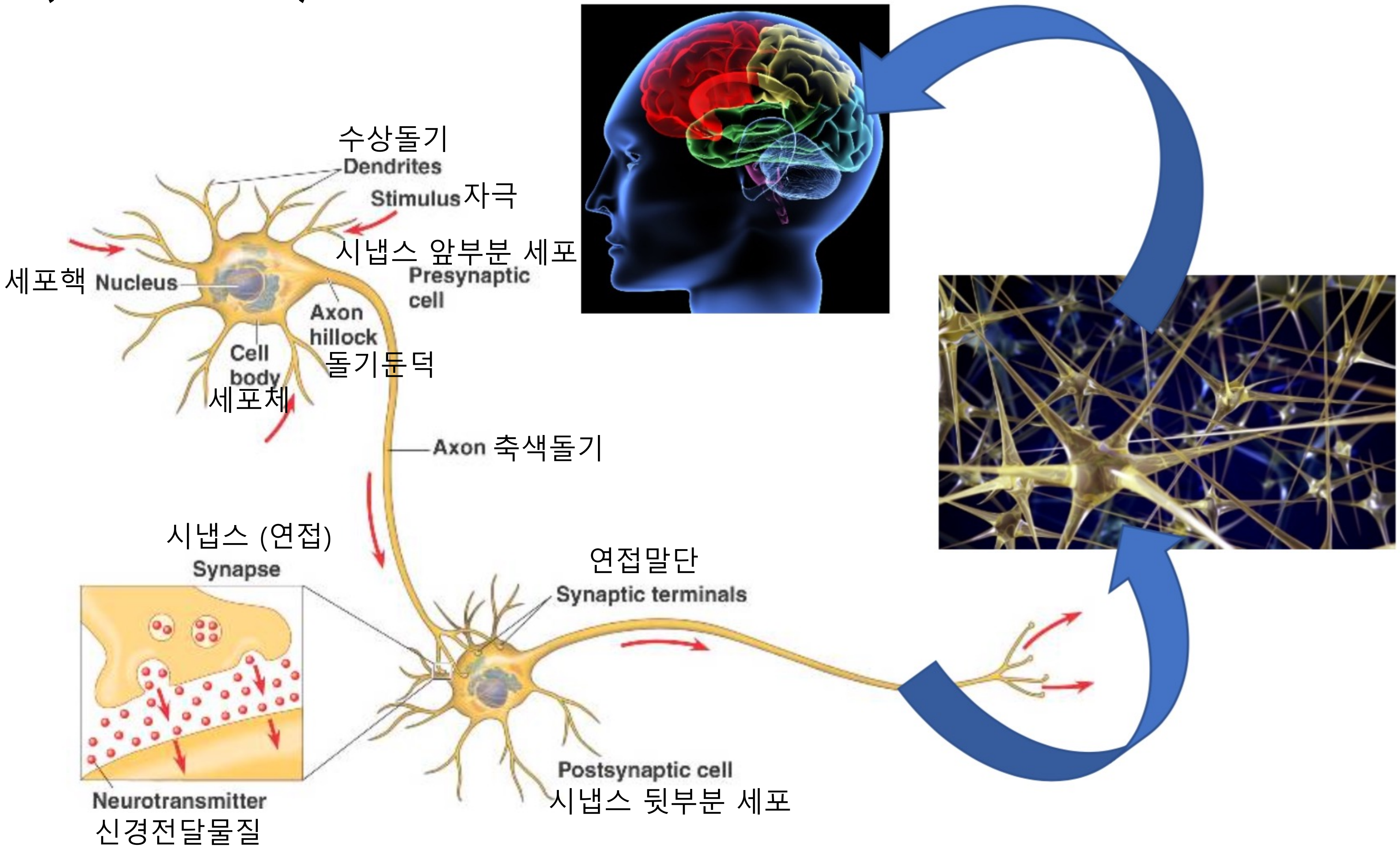
Sept. 2023

http://link.koreatech.ac.kr

Artificial Neuron

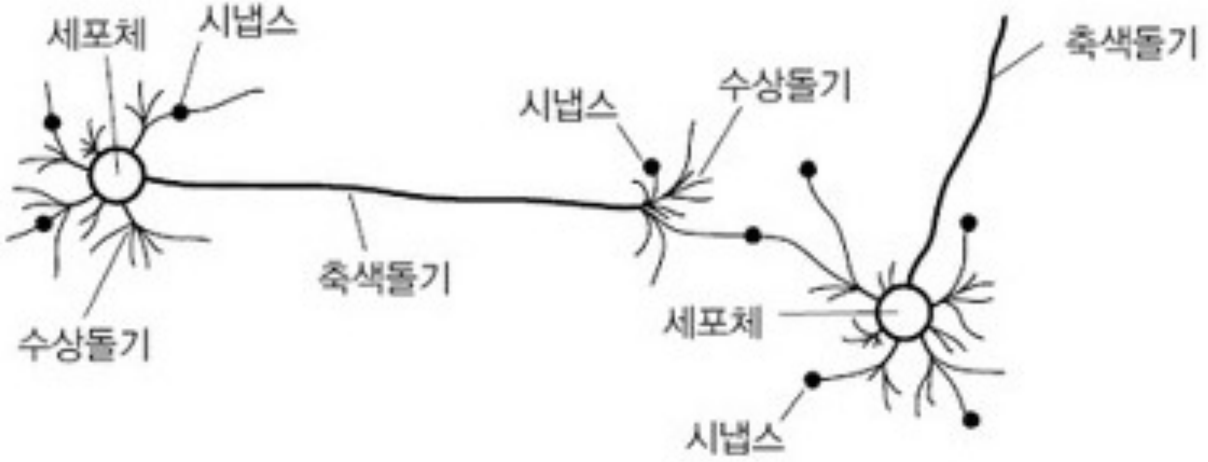
Human Brain

◈Neuron (신경 세포)

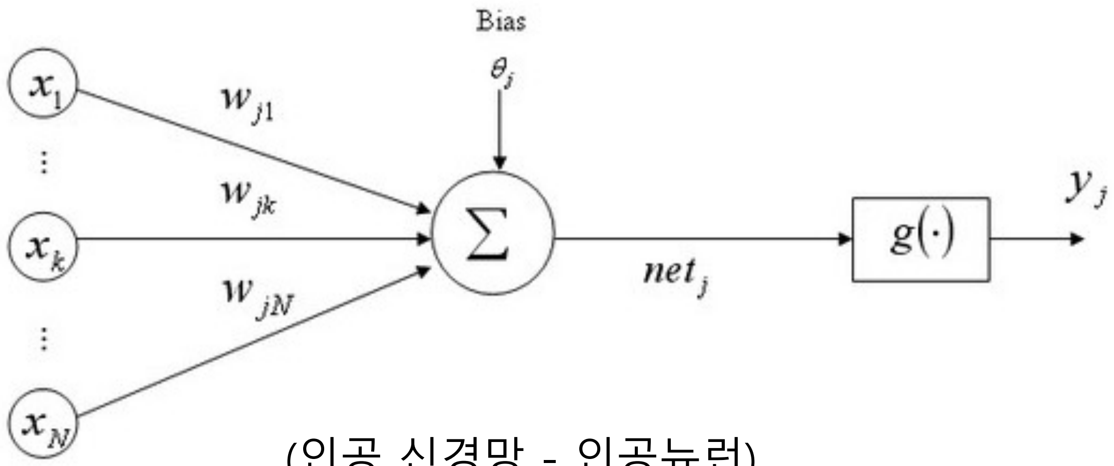


Human Brain

◈Neuron



(생물학적 신경망 - 뉴런)



(인공 신경망 - 인공뉴런)

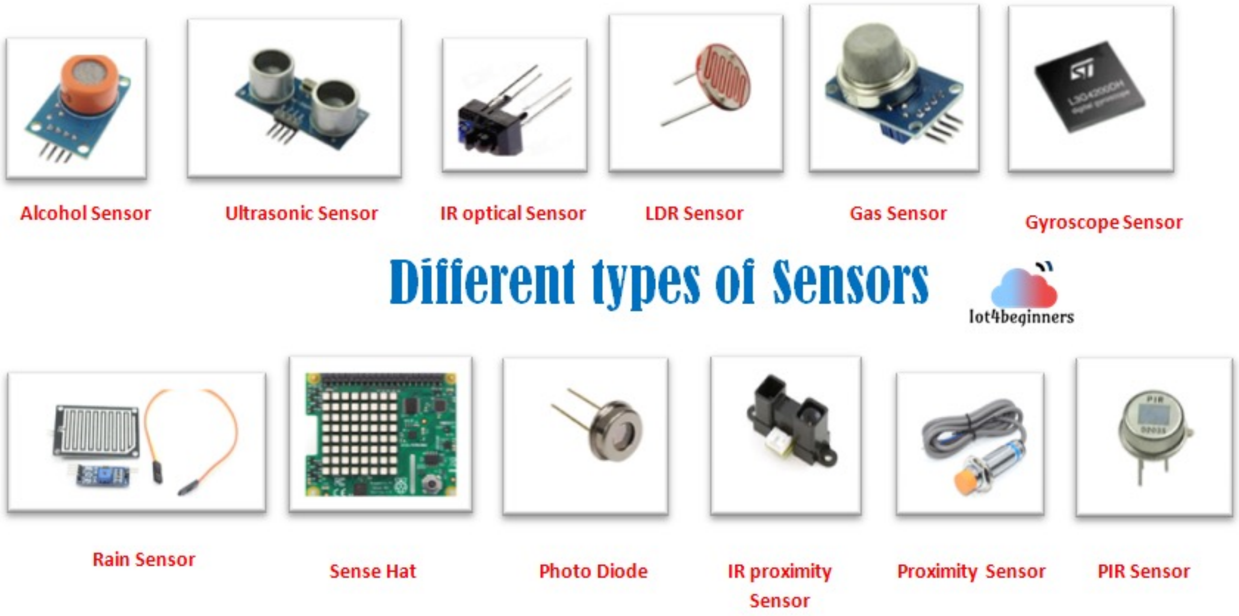
생물학적 신경망	인공 신경망
Dendrites (수상돌기)	Inputs
Cell Body (세포체)	Neuron
Axon (축색돌기)	Outputs

Artificial Neuron

◈Simple data representation

– Measurements by new two sensors

- X = [[0.5, 0.9], [14.0, 12.0], [15.0, 13.6], [28.0, 22.8], [11.0, 8.1], [8.0, 7.1], [3.0, 2.9], [-4.0, 0.1], [6.0, 5.3], [13.0, 12.0], [21.0, 19.9], [-1.0, 1.5]]
- Dataset shape: ***N × F = 12 × 2 → (12, 2)***



– Temperatures (measured at the same times of X measurement)

- y = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4, 29.1]
- This is a kind of target 'LABEL'
- Target dataset shape: ***(12,)***

Artificial Neuron

◈Simple data representation → Simple Dataset (1/2)

```
class SimpleDataset(Dataset):
  def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)

    X = [[0.5, 0.9], [14.0, 12.0], [15.0, 13.6],
         [28.0, 22.8], [11.0, 8.1], [8.0, 7.1],
         [3.0, 2.9], [4.0, 0.1], [6.0, 5.3],
         [13.0, 12.0], [21.0, 19.9], [-1.0, 1.5]]

    y = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4, 29.1]

    self.X = torch.tensor(X, dtype=torch.float, device=device)
    self.y = torch.tensor(y, dtype=torch.float, device=device)
    self.y = self.y * 0.01
```

Artificial Neuron

◈Simple data representation → Simple Dataset (2/2)

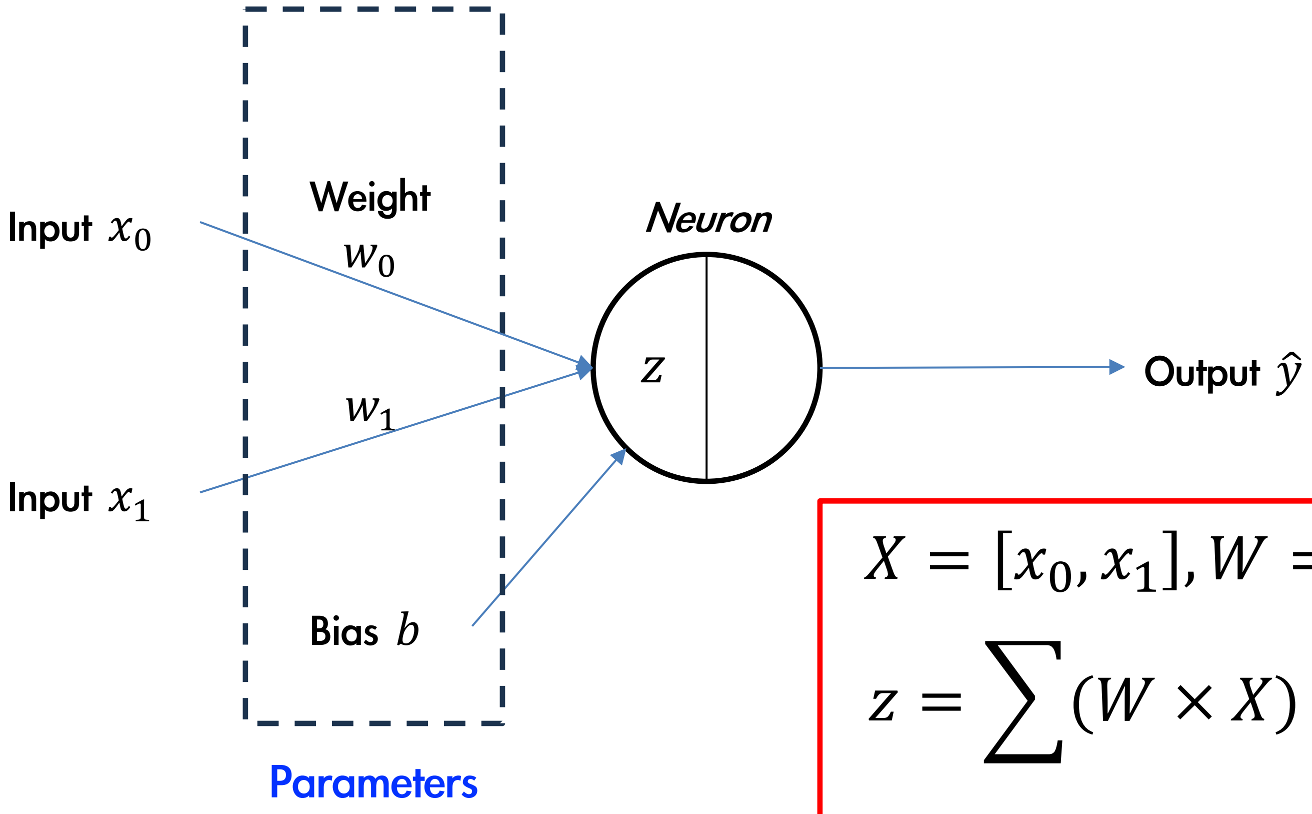
```python
class SimpleDataset(Dataset):
  ...

  def __len__(self):
    return len(self.X)

  def __getitem__(self, idx):
    return {'input': self.X[idx], 'target': self.y[idx]}

  def __str__(self):
    str = "Data Size: {0}, Input Shape: {1}, Target Shape: {2}".format(
      len(self.X), self.X.shape, self.y.shape
    )
    return str
```

Artificial Neuron

◈Artificial Neuron

– A mathematical function conceived as a model of biological neurons

[Weight Multiply & Bias Sum]



$$X = [x_0, x_1], W = [w_0, w_1]$$

$$z = \sum (W \times X) + b$$

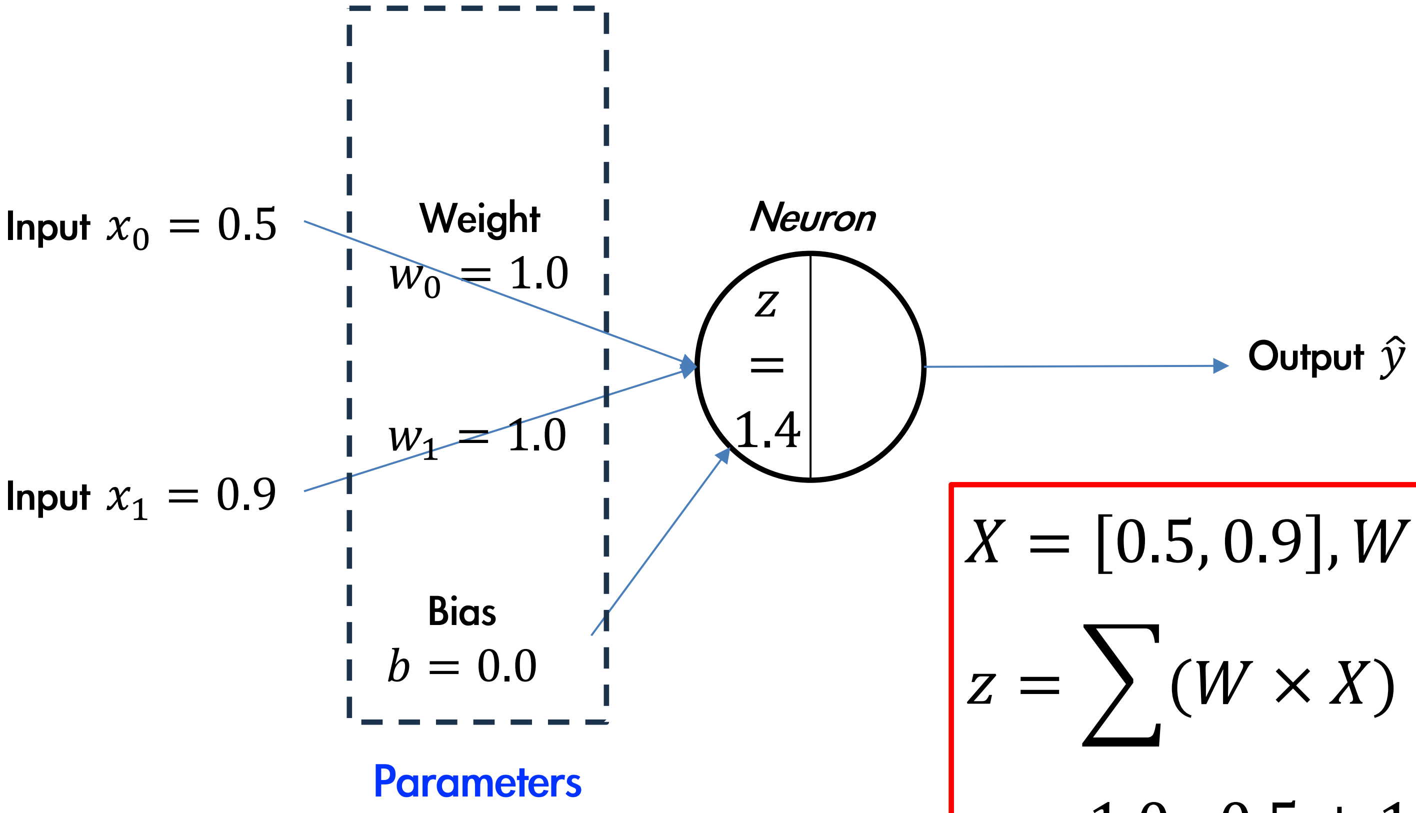
Affine Sum

$$= w_0 \cdot x_0 + w_1 \cdot x_1 + b$$

Artificial Neuron

◈Artificial Neuron

[Weight Multiply & Bias Sum]



$$X = [0.5, 0.9], W = [1.0, 1.0]$$

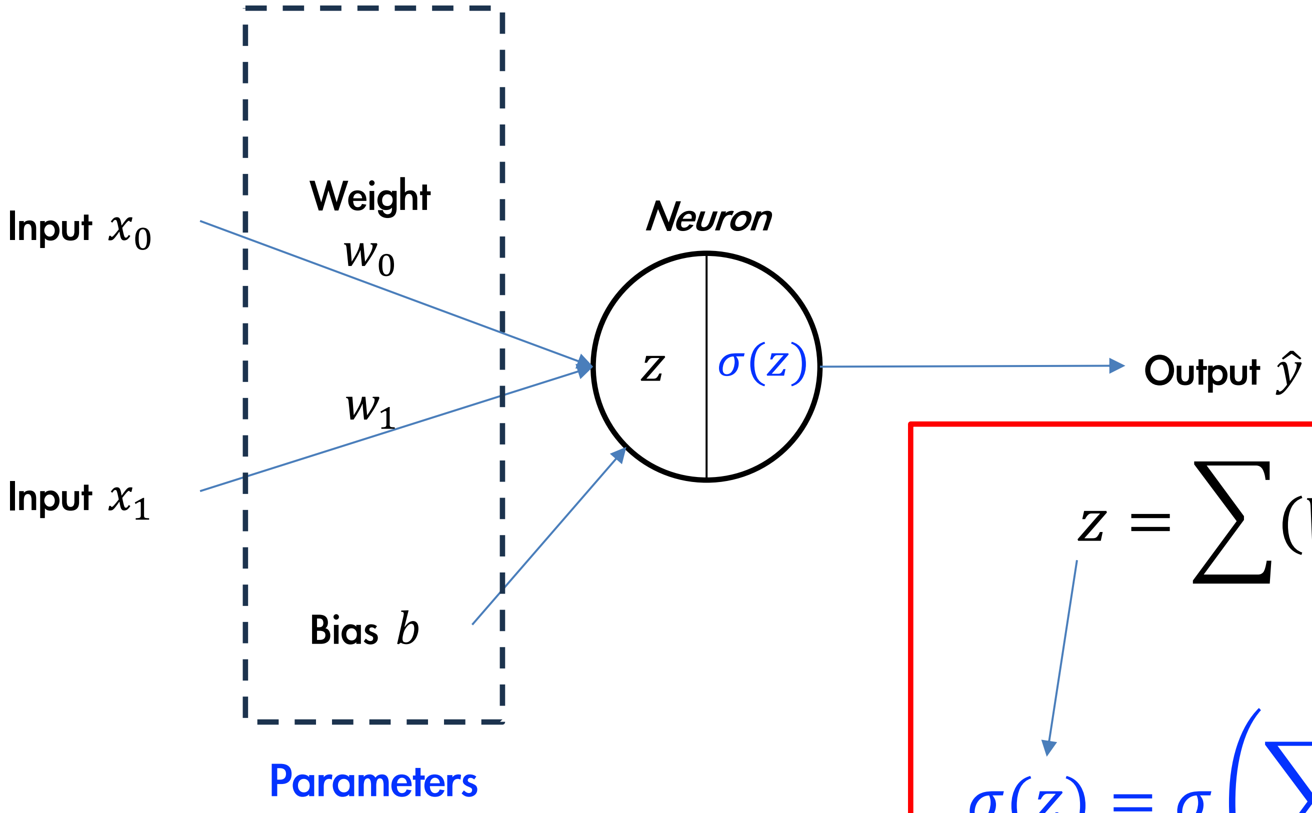
$$z = \sum (W \times X) + b$$

$$= 1.0 \cdot 0.5 + 1.0 \cdot 0.9 + 0.0 = 1.4$$

Artificial Neuron

◈Artificial Neuron

[Activation function (σ)]



$$z = \sum (W \times X) + b$$

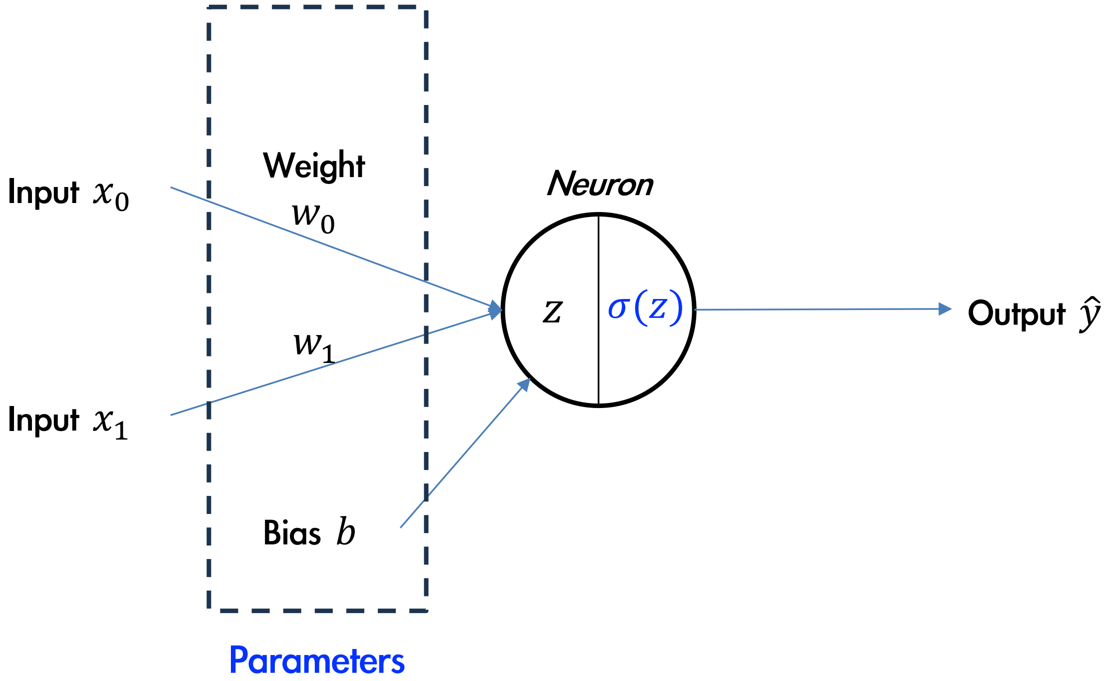
$$\sigma(z) = \sigma\left(\sum (W \times X) + b\right)$$

Artificial Neuron

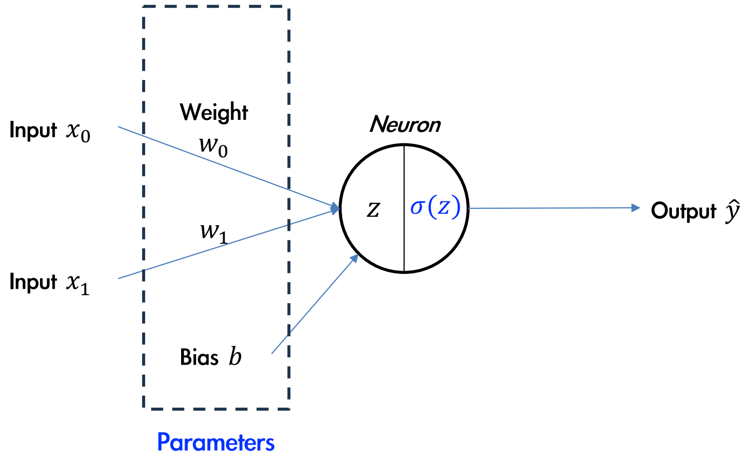
◈ Artificial Neuron

[Activation function (σ)]

- A function that transforms input signals into output signals
- It transforms or squashes input signals
- The main goal of it is to introduce non-linear properties into the neural network
  - ➢ Non-linear mappings applied to inputs can capture important properties of the input
- Activation function examples
  - ➢ Sigmoid
  - ➢ Tanh
  - ➢ ReLU
  - ➢ LeakyReLU
  - ➢ ELU
  - ➢ …



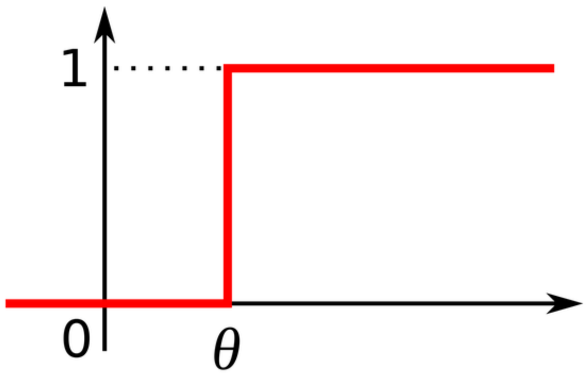
Artificial Neuron



◈Artificial Neuron

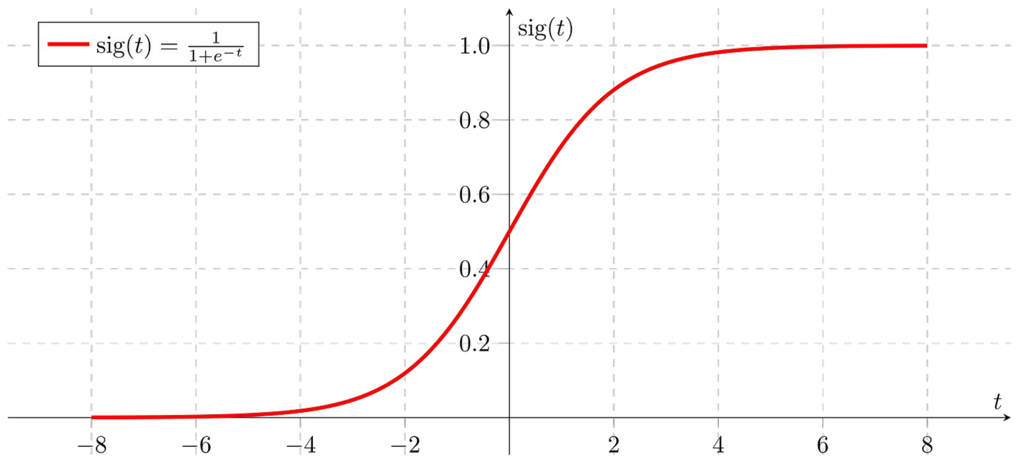
[Activation function (σ)]

Binary Step Function
(Perceptron, 1957)



$$\sigma(z) = \begin{cases} 1 & if\ z \geq \theta \\ 0 & otherwise \end{cases}$$

Sigmoid (or Logistic Activation)



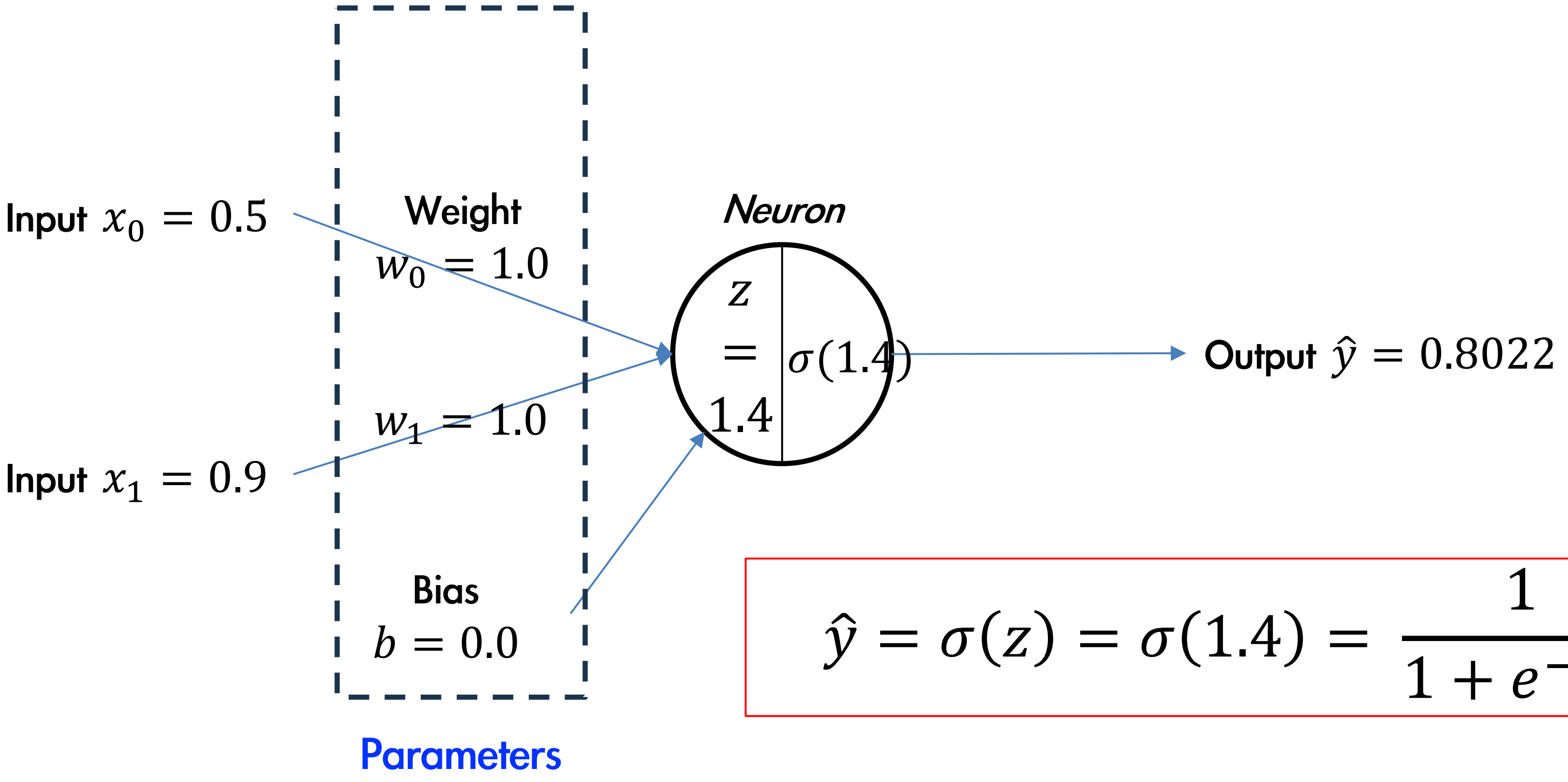
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

[Activation functions] - https://en.wikipedia.org/wiki/Activation_function

Artificial Neuron

◈Artificial Neuron

[Activation function (σ)]

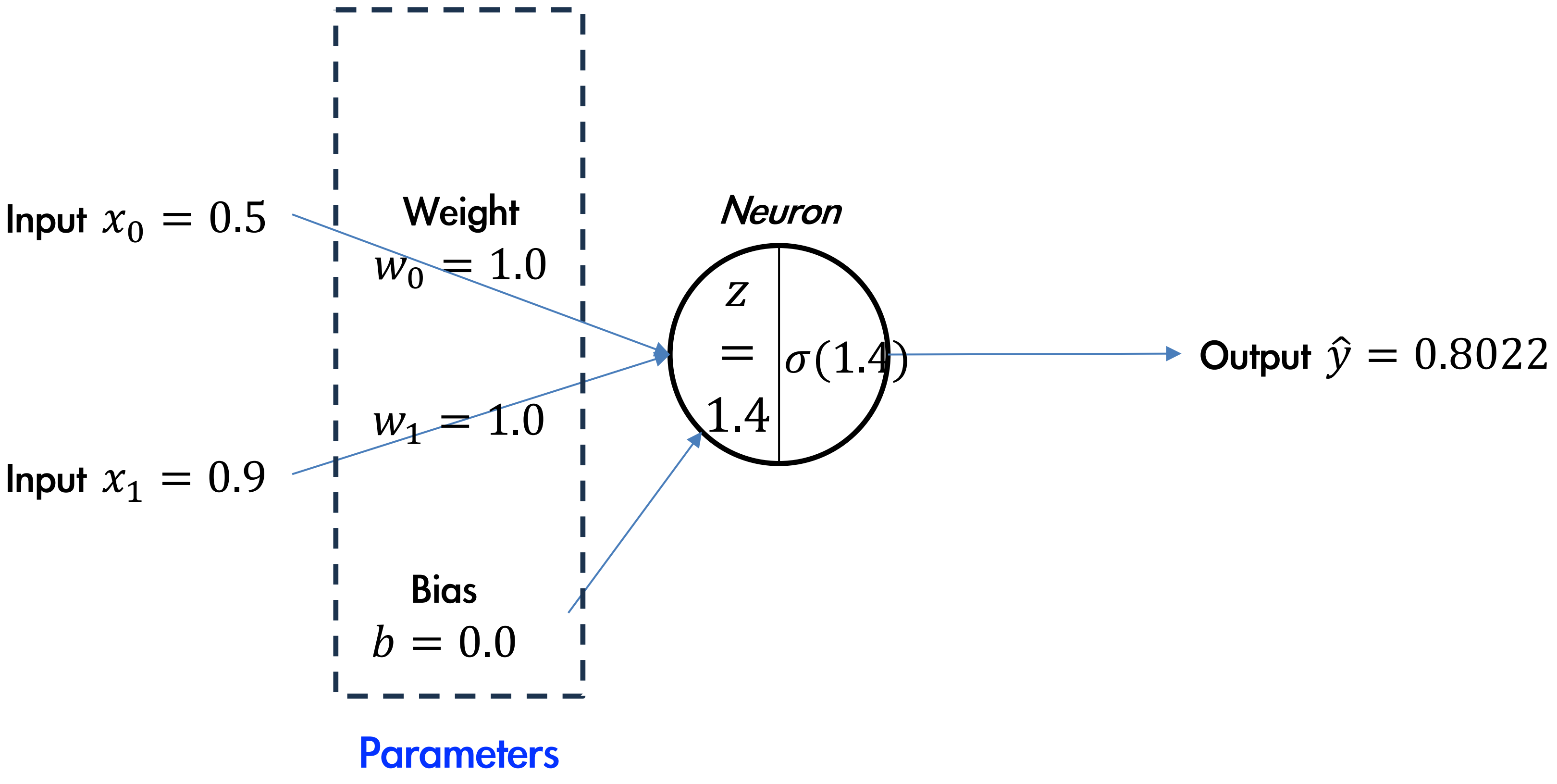


$$\hat{y} = \sigma(z) = \sigma(1.4) = \frac{1}{1 + e^{-1.4}} \approx 0.8022$$

Artificial Neuron

◈Artificial Neuron

$$\hat{y} = \sigma(z) = \sigma\left(\sum(W \times X) + b\right) = \frac{1}{1 + e^{-(\sum(W \times X)+b)}}$$

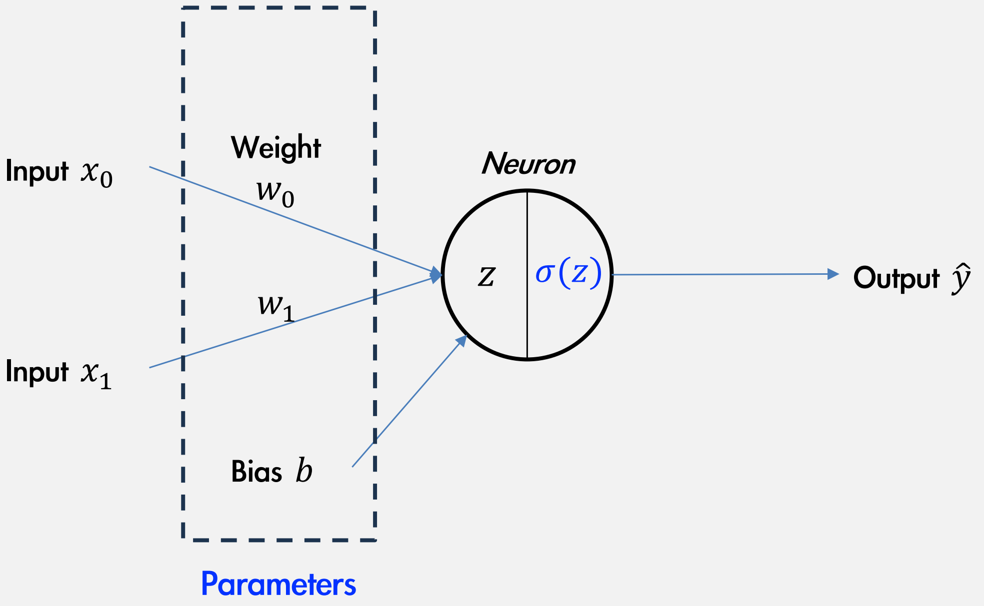


link_dl/_01_code/_04_learning_and_autograd/a_single_neuron.py

Artificial Neuron

◈ Artificial Neuron

```python
def model(X, W, b):
    print(X.shape)  # >>> torch.Size([12, 2])
    print(W.shape)  # >>> torch.Size([2])
    print(b.shape)  # >>> torch.Size([1])

    u = torch.sum(X * W, dim=1) + b
    # u.shape: torch.Size([12])

    z = activate(u)

    return z

def activate(u):
    return torch.sigmoid(u)
```



$$X = [x_0, x_1], W = [w_0, w_1]$$

$$z = \sum (W \times X) + b$$

$$= w_0 \cdot x_0 + w_1 \cdot x_1 + b$$

Affine Sum

Artificial Neuron

◈Artificial Neuron

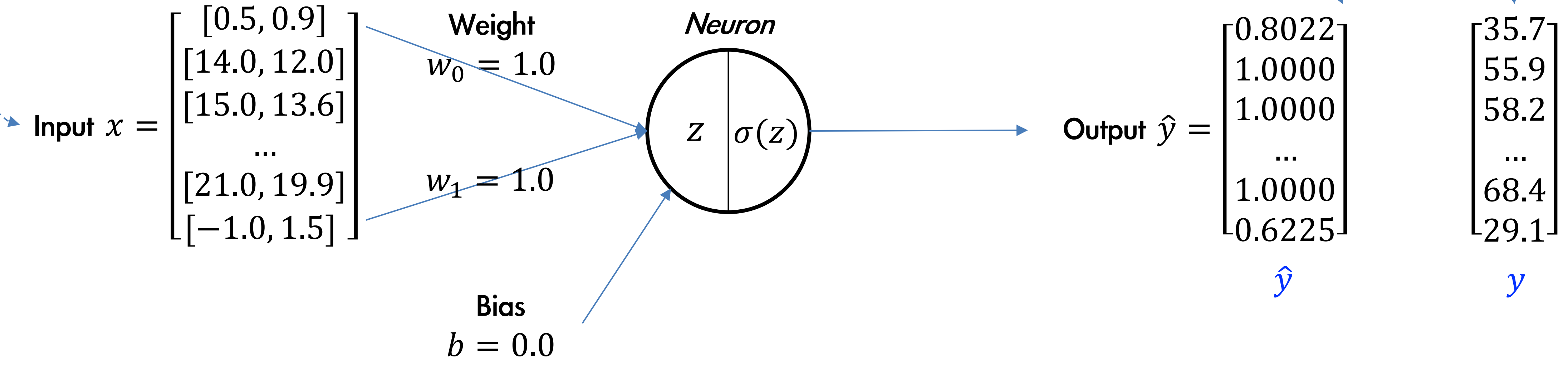
```
def main():
    W = torch.ones((2,))
    b = torch.zeros((1,))

    simple_dataset = SimpleDataset()
    train_data_loader = DataLoader(dataset=simple_dataset, batch_size=len(simple_dataset))
    batch = next(iter(train_data_loader))

    y_pred = model(batch["input"], W, b)
    print(y_pred.shape) # >>> torch.Size([12])
    print(y_pred)       # >>> tensor([0.8022, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
                        #             0.9973, 0.0198, 1.0000, 1.0000, 1.0000, 0.6225])

if __name__ == "__main__":
    main()
```

Learning with Gradient Descent

Loss Function and Our Goal Setup

◈Predicted Output & Target Output

- Feature Data
  - X = [[0.5, 0.9], [14.0, 12.0], [15.0, 13.6], [28.0, 22.8], [11.0, 8.1], [8.0, 7.1], [3.0, 2.9], [-4.0, 0.1], [6.0, 5.3], [13.0, 12.0], [21.0, 19.9], [-1.0, 1.5]]
- Target output (label)
  - y = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4, 29.1]
- Predicted output (by artificial neuron)
  - $\hat{y}$ = [0.8022, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9973, 0.0198, 1.0000, 1.0000, 1.0000, 0.6225]



Loss Function and Our Goal Setup

◈ Loss (or Error) for m data samples

– MSE (Mean Square Error)

$$L(W,b) = \frac{1}{m}\sum(\hat{y}-y)^2 = \frac{1}{m}\sum\left(\sigma\left(\sum(W\times X)+b\right)-y\right)^2$$

– RMSE (Root Mean Square Error)

$$L(W,b) = \sqrt{\frac{1}{m}\sum(\hat{y}-y)^2} = \sqrt{\frac{1}{m}\sum\left(\sigma\left(\sum(W\times X)+b\right)-y\right)^2}$$

– MAE (Mean Absolute Error)

$$L(W,b) = \frac{1}{m}\sum|\hat{y}-y| = \frac{1}{m}\sum\left|\sigma\left(\sum(W\times X)+b\right)-y\right|$$

$$\begin{array}{cc} & \hat{y} \\ \text{Output } z = & \begin{bmatrix}0.8022\\1.0000\\1.0000\\\dots\\1.0000\\0.6225\end{bmatrix}\end{array} \qquad \begin{array}{c} y \\ \begin{bmatrix}35.7\\55.9\\58.2\\\dots\\68.4\\29.1\end{bmatrix}\end{array}$$

MSE

$$L(W,b) = \frac{1}{m}\sum(\hat{y}-y)^2$$

$$= \frac{1}{12}\sum\left(\begin{bmatrix}0.8022\\1.0000\\1.0000\\\dots\\1.0000\\0.6225\end{bmatrix} - \begin{bmatrix}35.7\\55.9\\58.2\\\dots\\68.4\\29.1\end{bmatrix}\right)^2$$

$$= 2673.9028$$

Loss Function and Our Goal Setup

◈ Loss (or Error) for m data samples

```
def loss_fn(y_pred, y):
    loss = torch.square(y_pred - y).mean()
    assert loss.shape == () or loss.shape == (1,)
    return loss

def main():
    W = torch.ones((2,))
    b = torch.zeros((1,))

    simple_dataset = SimpleDataset()
    train_data_loader = DataLoader(dataset=simple_dataset, batch_size=len(simple_dataset))
    batch = next(iter(train_data_loader))
    y_pred = model(batch["input"], W, b)
    print(y_pred.shape) # >>> torch.Size([12])
    print(y_pred)       # >>> tensor([0.8022, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
                        #             0.9973, 0.0198, 1.0000, 1.0000, 1.0000, 0.6225])
    loss = loss_fn(y_pred, batch["target"])
    print(loss)         # >>> tensor(0.2254)

if __name__ == "__main__":
    main()
```

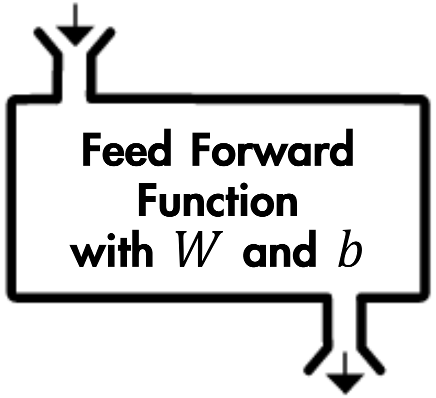
Loss Function and Our Goal Setup

◈Our goal

Find W^* and b^* such that

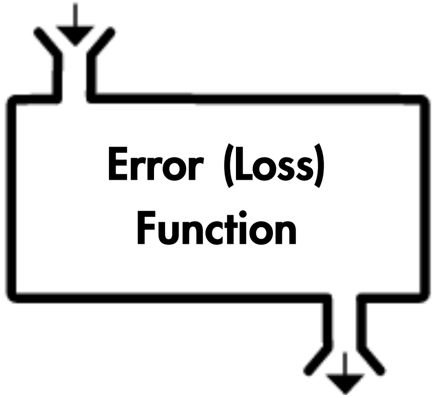
$$W^*, b^* = argmin_{W,b} L(W,b) = argmin_{W,b} \frac{1}{m} \sum (\hat{y} - y)^2$$

Input x



Output $\hat{y}$

W, b



$L(W,b)$

Our Goal

$$\hat{y} = \sigma(z) = \sigma\left(\sum (W \times X) + b\right)$$

$$L(W,b) = \frac{1}{m} \sum (\hat{y} - y)^2$$

$$W^*, b^* = argmin_{W,b} L(W,b)$$

[source]: http://www.drcruzan.com/TheDerivative.html

Gradient Descent

◇Derivative (도함수) of a function

$$f'(x) = \frac{df}{dx} = \text{Derivative of a function } f$$

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

$$f'(x_0) = \left.\frac{df}{dx}\right|_{x=x_0} = \lim_{\Delta x \to 0} \frac{f(x_0+\Delta x) - f(x_0)}{\Delta x}$$

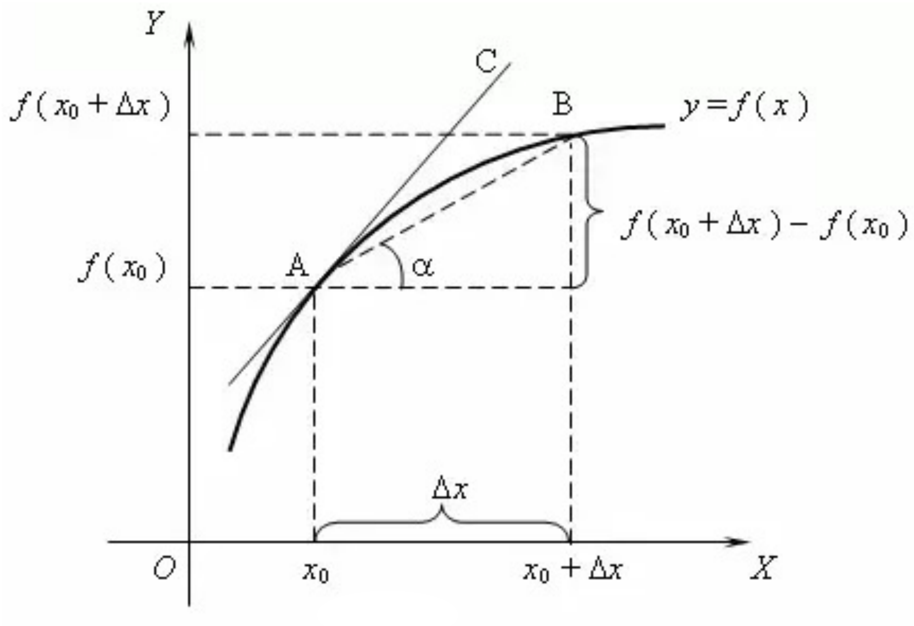
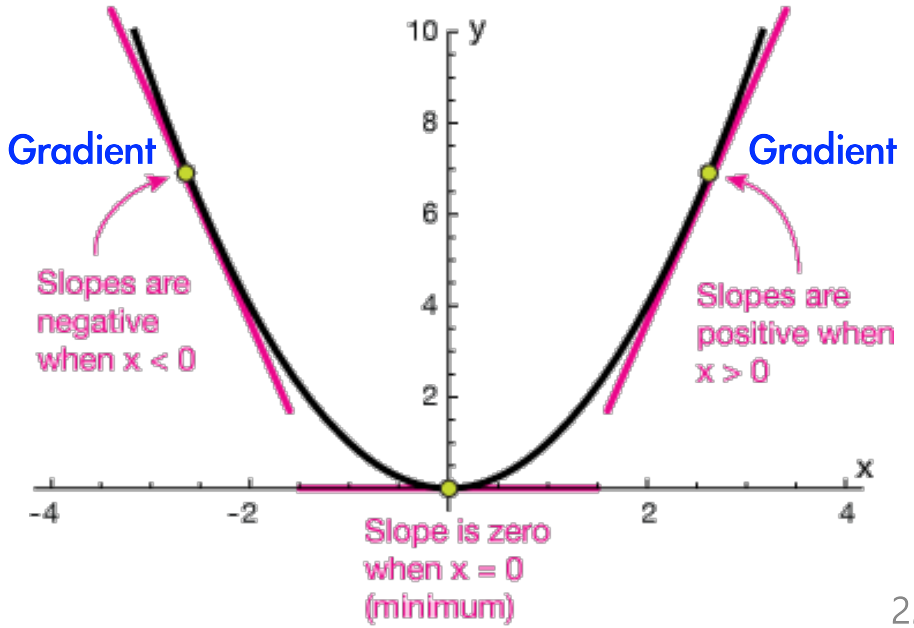


Fig. 1

$f = x^2$ ➡

$$\begin{aligned} f'(x) &= \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx} \\ &= \lim_{dx \to 0} \frac{(x+dx)^2 - x^2}{dx} \\ &= \lim_{dx \to 0} \frac{x^2 + 2xdx + dx^2 - x^2}{dx} \\ &= \lim_{dx \to 0} 2x + dx \\ &= 2x \end{aligned}$$

$$f(x) = x^2 \to f'(x) = 2x$$



Gradient Descent

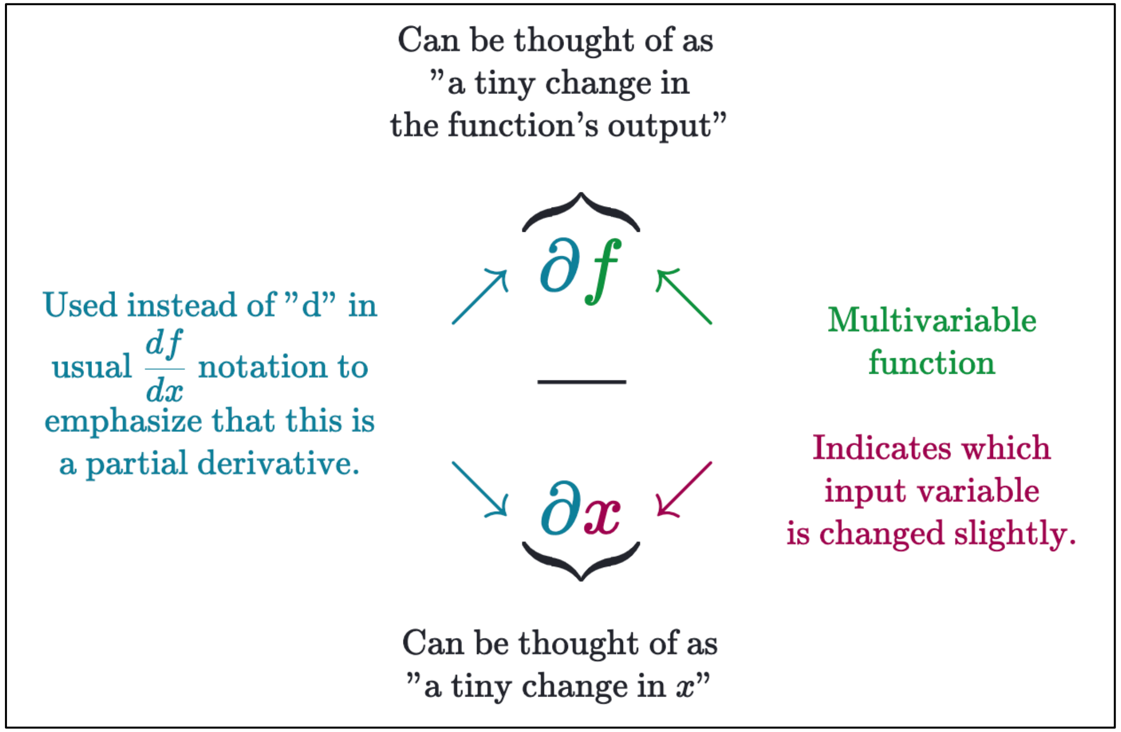
◇ Partial derivative of a multi-variable function $f(x, y, \ldots)$

– For a multi-variable function, like $f(x, y) = x^2 y$, partial derivatives are as follows:

$$\frac{\partial f}{\partial x} = \underbrace{\frac{\partial}{\partial x} x^2 y}_{\substack{\text{Treat } y \text{ as constant;} \\ \text{take derivative.}}} = 2xy \qquad \frac{\partial f}{\partial y} = \underbrace{\frac{\partial}{\partial y} x^2 y}_{\substack{\text{Treat } x \text{ as constant;} \\ \text{take derivative.}}} = x^2 \cdot 1$$

$$f_x = \frac{\partial f}{\partial x} = lim_{h \to 0} \frac{f(x+h, y) - f(x, y)}{h}$$

$$f_y = \frac{\partial f}{\partial y} = lim_{h \to 0} \frac{f(x, y+h) - f(x, y)}{h}$$



Gradient Descent

◈ Gradient by partial derivative of a multi-variable function $f(x, y, \ldots)$

– Gradient of a (scalar-valued) multi-variable function $f(x, y, \ldots)$, denoted ∇f, is the collection of all its partial derivatives into a vector

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \vdots \end{bmatrix}$$

– this means ∇f is a vector-valued function

Gradient Descent

- **Gradient** by partial derivative of a <u>multi-variable</u> function $f(x, y, \dots)$
  - Example differential operators

Name	Symbol	Example
Derivative	$\frac{d}{dx}$	$\frac{d}{dx}(x^2) = 2x$
Partial derivative	$\frac{\partial}{\partial x}$	$\frac{\partial}{\partial x}(x^2 - xy) = 2x - y$
Gradient	∇	$\nabla(x^2 - xy) = \begin{bmatrix} 2x - y \\ -x \end{bmatrix}$

Gradient Descent

Our Goal

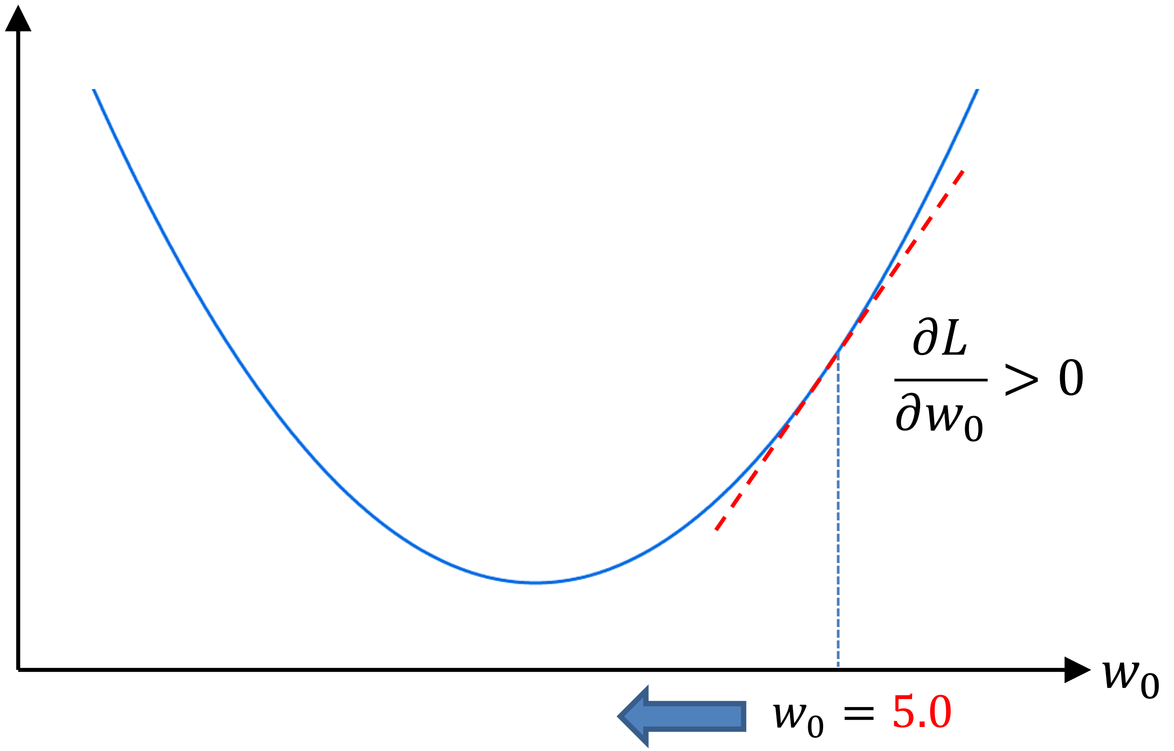
$$w_0^*, w_1^*, b^* = argmin_{w_0,w_1,b} L(w_0, w_1, b)$$

◈Gradient Descent Method (경사하강법)

$$L(W,b) = \frac{1}{m}\sum(\hat{y} - y)^2$$

$$= \frac{1}{m}\sum\left(\sigma\left(\sum(W \times X) + b\right) - y\right)^2$$

$$= \frac{1}{m}\sum(\sigma(x_0 \cdot w_0 + x_1 \cdot w_1 + b) - y)^2$$



$$w_0 = w_0 - \gamma \frac{\partial L}{\partial w_0}$$

Using Partial Derivative

where γ is the learning rate

Gradient Descent

Our Goal

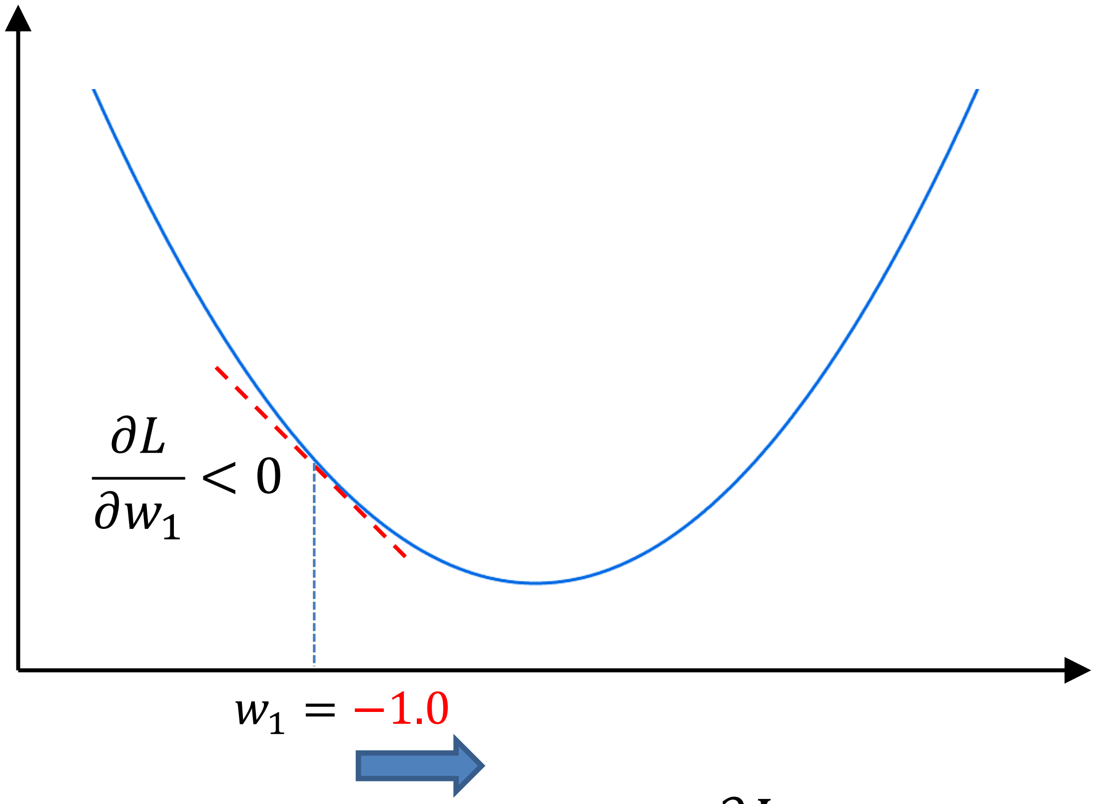
$$w_0^*, w_1^*, b^* = argmin_{w_0,w_1,b} L(w_0, w_1, b)$$

◈Gradient Descent Method

$$L(W,b) = \frac{1}{m}\sum(\hat{y}-y)^2$$

$$= \frac{1}{m}\sum\left(\sigma\left(\sum(W \times X) + b\right) - y\right)^2$$

$$= \frac{1}{m}\sum(\sigma(x_0 \cdot w_0 + x_1 \cdot w_1 + b) - y)^2$$



$$w_1 = w_1 - \gamma \frac{\partial L}{\partial w_1}$$

Using Partial Derivative

where γ is the learning rate

Gradient Descent

◈Gradient Descent Method

Find W^* and b^* such that

$$W^*, b^* = argmin_{W,b} L(W,b) = argmin_{W,b} \frac{1}{m} \sum (\hat{y} - y)^2$$

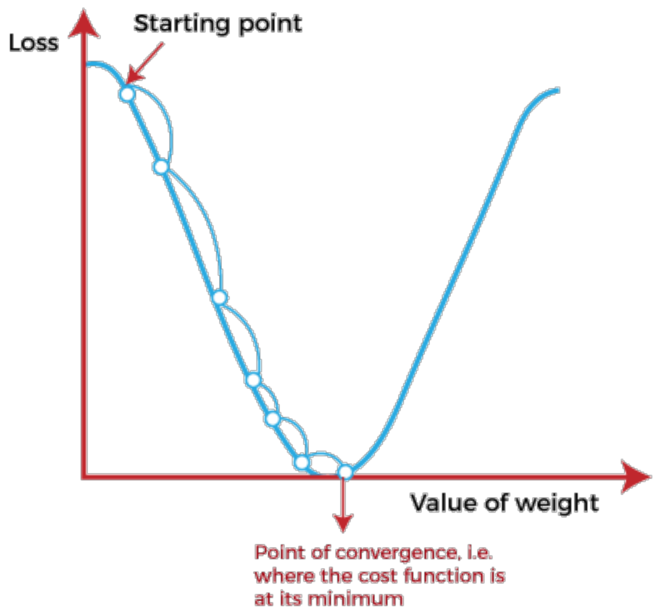
Solution:

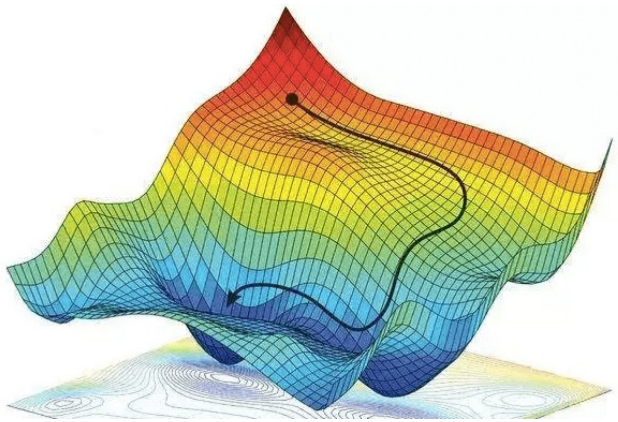
$$w_0 = w_0 - \gamma \frac{\partial L}{\partial w_0}$$

$$w_1 = w_1 - \gamma \frac{\partial L}{\partial w_1}$$

$$b = b - \gamma \frac{\partial L}{\partial b}$$

where γ is the learning rate





<u>Repeat the above equations until the error value L becomes enough small</u>

Gradient Descent

◈Gradient Descent Method

Find W^* and b^* such that

$$W^*, b^* = argmin_{W,b} L(W, b) = argmin_{W,b} \frac{1}{m} \sum (\hat{y} - y)^2$$

The magnitude of each component of ∇L is telling you that

<u>**"how sensitive the loss function L is to the current weight and bias"**</u>

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b} \end{bmatrix} \longrightarrow \boxed{\begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix} - \gamma \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b} \end{bmatrix}}$$

$$\left\{ \begin{aligned} w_0 &= w_0 - \gamma \frac{\partial L}{\partial w_0} \\ w_1 &= w_1 - \gamma \frac{\partial L}{\partial w_1} \\ b &= b - \gamma \frac{\partial L}{\partial b} \end{aligned} \right.$$

Gradient Descent

◈Gradient Descent Method - Pseudo Code

Gradient descent

Input: Function f to minimize.

Initialization: initial weight vector $w^{(0)}$ ← Weights & Bias

Parameters: step size $\eta > 0$. ← Learning Rate

While *not converge* do

- $w^{(k+1)} \leftarrow w^{(k)} - \eta \nabla f(w^{(k)})$
- $k \leftarrow k + 1$.

Output: $w^{(k)}$.

Backpropagation

Backpropagation

◈ How to calculate gradient for a multi-variable function

$f(x, y, \ldots)$

Numerical Differentiation

$$f'(x) = \frac{\partial f}{\partial x} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} = \frac{a(x+\Delta x)^n - a(x)^n}{\Delta x}$$

$$f'(x) = \frac{\partial f}{\partial x} \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2 \times \Delta x} = \frac{a(x+\Delta x)^n - a(x-\Delta x)^n}{2 \times \Delta x}$$

$$f_x = \frac{\partial f}{\partial x} = lim_{h \to 0} \frac{f(x+h, y) - f(x, y)}{h}$$

$$f_y = \frac{\partial f}{\partial y} = lim_{h \to 0} \frac{f(x, y+h) - f(x, y)}{h}$$

Differentiation Formulas

$$\frac{d}{dx} k = 0$$

$$\frac{d}{dx}[f(x) \pm g(x)] = f'(x) \pm g'(x)$$

$$\frac{d}{dx}[k \cdot f(x)] = k \cdot f'(x)$$

$$\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$$

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

$$\frac{d}{dx} x^n = nx^{n-1}$$

Chain Rule & Backpropagation

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial i} \cdot \frac{\partial i}{\partial x}$$

Backpropagation

◈ Differentiation formulas of a <u>multi-variable</u> function $f(x, y, \dots)$

– Multiplication function of two numbers

$$f(x, y) = xy \longrightarrow \frac{\partial f}{\partial x} = y,\ \frac{\partial f}{\partial y} = x \longrightarrow \Delta f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [y, x]$$

- For example,
  - $x = 4, y = -3$ then $f(x, y) = -12$
  - the derivative on x and y: $\frac{\partial f}{\partial x} = -3 \quad \frac{\partial f}{\partial y} = 4$
  - Interpretation
    - » If we increase the value of x by a tiny amount, the effect on the output would be to decrease it by -3
    - » If we increase the value of y by a tiny amount, the effect on the output would be to increase it by 4

Backpropagation

◈ Differentiation formulas of a <u>multi-variable</u> function $f(x, y, \dots)$

– Addition function of two numbers

$$f(x,y) = x + y \longrightarrow \frac{\partial f}{\partial x} = 1, \frac{\partial f}{\partial y} = 1 \longrightarrow \Delta f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [1,1]$$

- For example,
  - $x = 4, y = -3$ then $f(x,y) = 1$
  - the derivative on x and y: $\dfrac{\partial f}{\partial x} = 1 \quad \dfrac{\partial f}{\partial y} = 1$
  - Interpretation
    - » This makes sense, since increasing either x, y would increase the output by the same amount, and the rate of that increase would be independent of what the actual values of x, y

Backpropagation

◈Differentiation formulas of a <u>multi-variable</u> function $f(x, y, \dots)$

– 'Max' function of two numbers

$$f(x,y) = \max(x,y) \longrightarrow \begin{aligned} \Delta f &= \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [1,0] \; if \; x \geq y \\ \Delta f &= \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [0,1] \; if \; x < y \end{aligned}$$

- For example,
  - ➢ $x = 4$, $y = 2$ then $f(x, y) = 4$ and it is not sensitive of y
  - ➢ the derivative on x and y: $\frac{\partial f}{\partial x} = 1 \quad \frac{\partial f}{\partial y} = 0$
  - ➢ Interpretation
    - » if we increase x by a tiny amount, the function would increase the output by the same amount
    - » if we increase y by a tiny amount, the function would keep outputting 4, and therefore the gradient is zero: there is no effect

Backpropagation

◈Chain Rule (연쇄 법칙)

연쇄 법칙

위키백과, 우리 모두의 백과사전.

연쇄법칙은 두 함수를 합성한 합성 함수의 도함수에 관한 공식이다.

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

라이프니츠 표기를 쓰면 다음과 같다.

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

$$f(x) = (x^3 + x^2 + x + 1)^3$$

⇩

$$g(x) = x^3 + x^2 + x + 1$$
$$f(g(x)) = g(x)^3$$

⇩

$$f\big(g(x)\big)' = f'(g(x))g'(x)$$

⇩

$$\begin{aligned}\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx} &= f'(g(x))g'(x) \\ &= 3g(x)^2 \cdot (3x^2 + 2x + 1) \\ &= 3(x^3 + x^2 + x + 1)^2(3x^2 + 2x + 1)\end{aligned}$$

$$(f \circ g)'(x) = f'(g(x)) \times g'(x)$$
$$(f \circ g \circ h)'(x) = f'(g(h(x))) \times g'(h(x)) \times h'(x)$$
$$(f \circ g \circ h \circ i)'(x) = f'(g(h(i(x)))) \times g'(h(i(x))) \times h'(i(x)) \times i'(x)$$

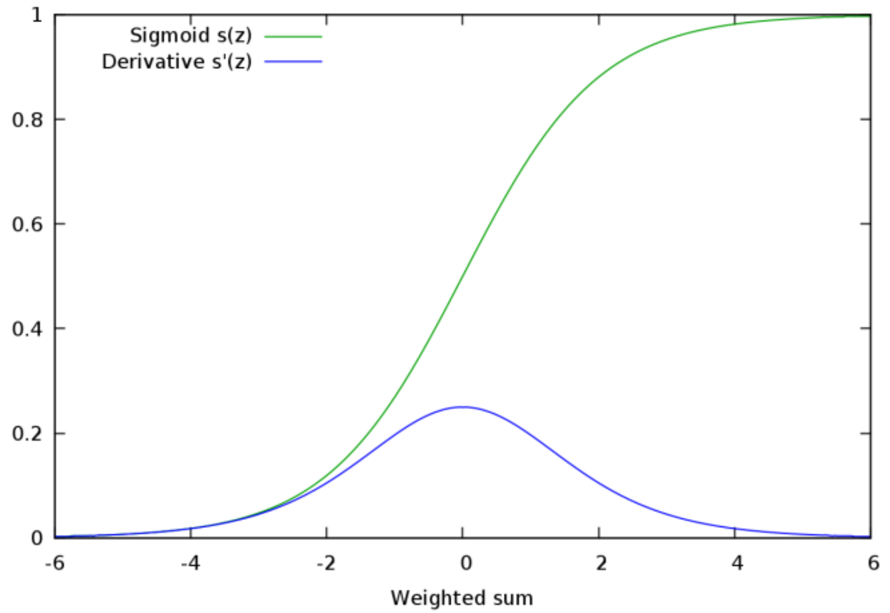
$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx}$$

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{di} \cdot \frac{di}{dx}$$

$f(x) = \frac{1}{x} \Rightarrow f'(x) = -\frac{1}{x^2}$

Backpropagation



◈Chain Rule

– Derivative of sigmoid function

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$\sigma(z) = \frac{1}{k(z)} \quad \text{where} \quad k(z) = 1 + e^{-z}$$

by chain rule $\frac{d\sigma}{dz} = \frac{d\sigma}{dk} \cdot \frac{dk}{dz} = -\frac{1}{k(z)^2}(-e^{-z})$

$$= -\frac{1}{(1+e^{-z})^2}(-e^{-z}) = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - \frac{1+e^{-z}-1}{1+e^{-z}}$$

$$= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = \sigma(z)(1-\sigma(z))$$

$\boxed{\sigma(z) = \frac{1}{1+e^{-z}}}$ $\xrightarrow{\text{Derivative of sigmoid function}}$ $\boxed{\frac{d\sigma}{dz} = \sigma(z)(1-\sigma(z))}$

Backpropagation

◈Chain Rule

$$L = \frac{1}{m}\sum l(w_0, w_1, b)$$

⇩

$$\frac{dL}{dx} = \frac{1}{m}\sum \frac{dl(w_0, w_1, b)}{dx}$$

$$l(w_0, w_1, b) = (\hat{y} - y)^2 \quad \text{where} \quad \hat{y} = \sigma(z)$$

$$= (\sigma(z) - y)^2 \quad \text{where} \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$= \left(\frac{1}{1 + e^{-z}} - y\right)^2 \quad \text{where} \quad z = x_0 \cdot w_0 + x_1 \cdot w_1 + b$$

$$= \left(\frac{1}{1 + e^{-(x_0 \cdot w_0 + x_1 \cdot w_1 + b)}} - y\right)^2$$

$$\frac{\partial l}{\partial w_0} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_0} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot x_0$$

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot x_1$$

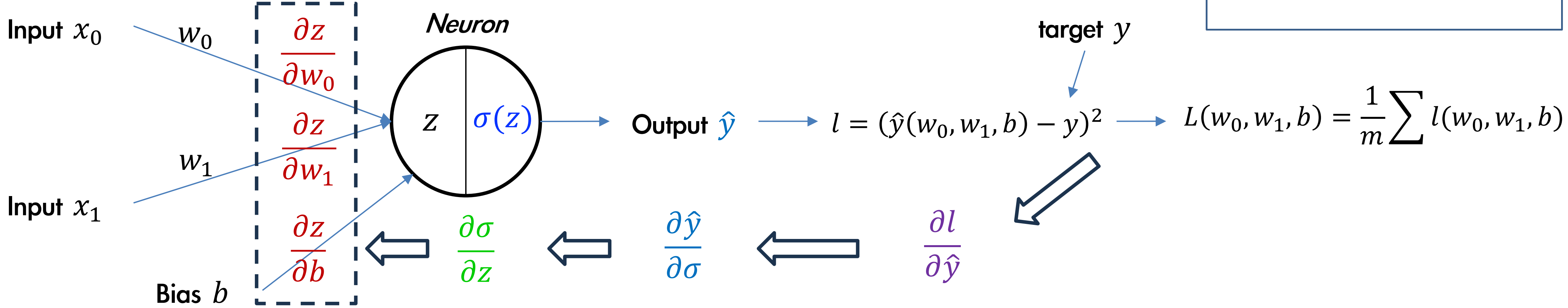
$$\frac{\partial l}{\partial b} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial b} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot 1.0$$

Backpropagation

$$L = \frac{1}{m}\sum l(w_0, w_1, b)$$

$$\frac{dL}{dx} = \frac{1}{m}\sum \frac{dl(w_0, w_1, b)}{dx}$$

◈Backpropgation



$$\frac{\partial l}{\partial w_0} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_0} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot x_0$$

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot x_1$$

$$\frac{\partial l}{\partial b} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial b} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot 1.0$$

Backpropagation

◈Backpropagation Implementation

– Chain Rule $f(x, y, z) = (x + y)z$ ➡ $f = qz$ where $q = x + y$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} = z \cdot 1 = z$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} = z \cdot 1 = z$$

$$\frac{\partial f}{\partial z} = q = x + y$$

```
def backward(x, y, z):
    q = x + y
    f = q * z

    df_dq = z             # -4
    df_dz = q             # 3
    df_dx = z * 1.0       # -4
    df_dy = z * 1.0       # -4
    return df_dx, df_dy, df_dz

df_dx, df_dy, df_dz = \
    backward(x=-2, y=5, z=-4)
```

$\boxed{[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}]}$ is the sensitivity of the variables x, y, z on f

Backpropagation

◈Backpropagation Implementation

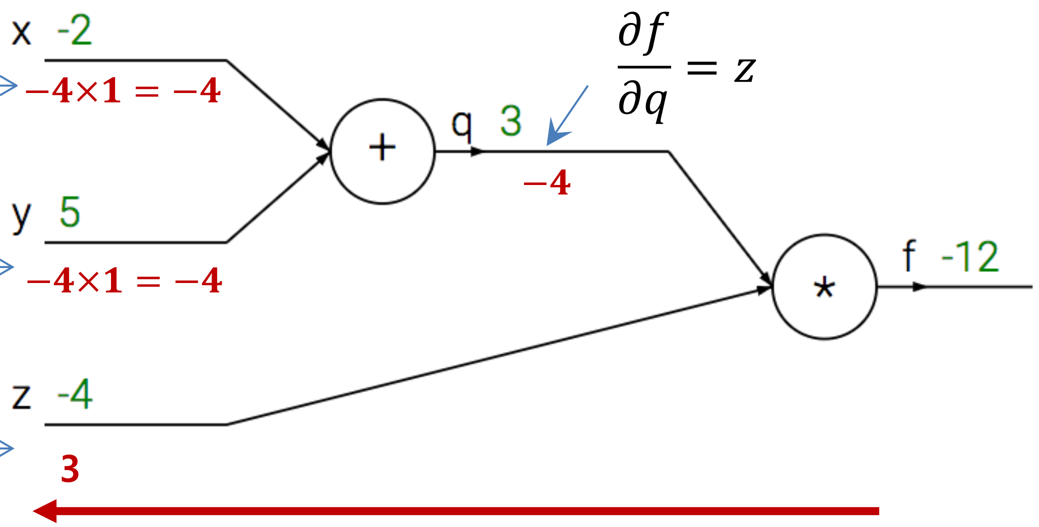
– Computational graph (Visual representation of the computation)

$$f(x, y, z) = (x + y)z \quad \Rightarrow \quad f = qz \text{ where } q = x + y$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y}$$

$$\frac{\partial f}{\partial z}$$



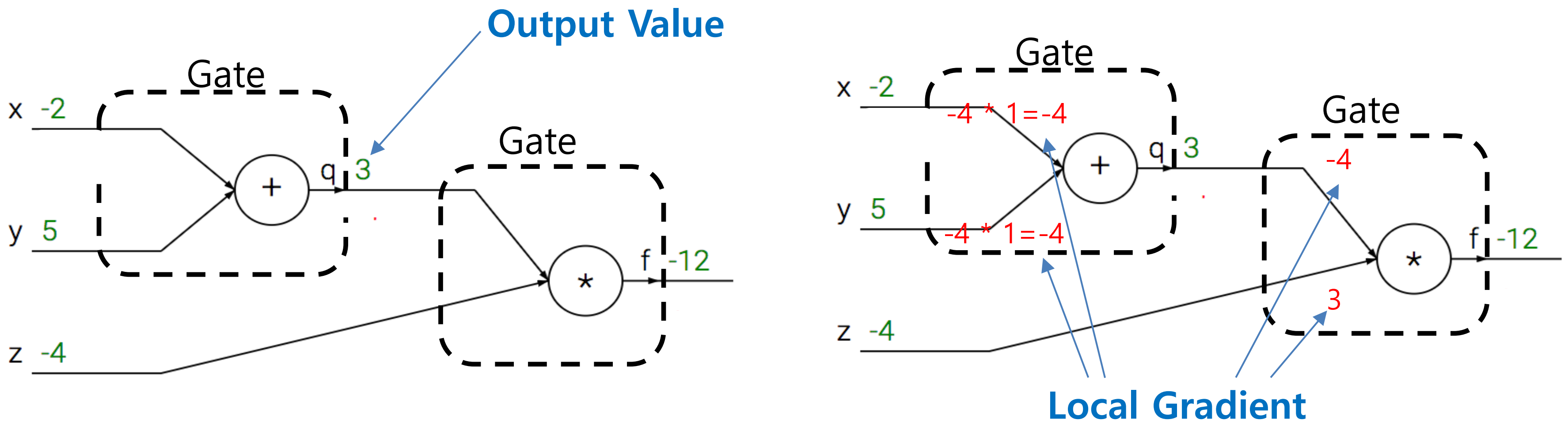

```
def backward(x, y, z):
    q = x + y
    f = q * z

    df_dq = z            # -4
    df_dz = q            # 3
    df_dx = z * 1.0      # -4
    df_dy = z * 1.0      # -4
    return df_dx, df_dy, df_dz

df_dx, df_dy, df_dz = \
    backward(x=-2, y=5, z=-4)
```

- Backward pass via computational graph
  - ➢ performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients all the way to the inputs of the graph

◈Backpropagation Implementation

Backpropagation is a beautifully local process!

Each gate produce its output and local gradient completely independently without being aware of any of the details of the full graph



Backpropagation

◈Training with Backpropagation (1/3)

```
def gradient(W, b, X, y):
    # W.shape: (2,), b.shape: (1,), X.shape: (12, 2), y.shape: (12)
    y_pred = model(X, W, b)
    dl_dy = 2 * (y_pred - y)
    dl_dy = dl_dy.unsqueeze(dim=-1)                              # dl_dy.shape: (12, 1)

    dy_df = 1.0

    z = torch.sum(X * W, dim=-1) + b                             # z.shape: (12,)
    ds_dz = activate(z) * (1.0 - activate(z))
    ds_dz = ds_dz.unsqueeze(dim=-1)                              # ds_dz.shape: (12, 1)

    W_grad = torch.mean(dl_dy * dy_df * ds_dz * X, dim=0)        # W_grad.shape: (2,)
    b_grad = torch.mean(dl_dy * dy_df * ds_dz * 1.0, dim=0)      # b_grad.shape: (1,)

    return W_grad, b_grad
```

$$\frac{\partial l}{\partial w_0} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_0} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot x_0$$

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot x_1$$

$$\frac{\partial l}{\partial b} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial b} = 2(\hat{y} - y) \cdot 1.0 \cdot \sigma(z)(1 - \sigma(z)) \cdot 1.0$$

Backpropagation

◈Training with Backpropgation (2/3)

```
def learn(W, b, train_data_loader):
  MAX_EPOCHS = 20_000
  LEARNING_RATE = 0.01

  for epoch in range(0, MAX_EPOCHS):
    batch = next(iter(train_data_loader))
    y_pred = model(batch["input"], W, b)
    loss = loss_fn(y_pred, batch["target"])

    W_grad, b_grad = gradient(W, b, batch["input"], batch["target"])

    if epoch % 100 == 0:
      print("[Epoch:{0:6,}] loss:{1:8.5f}, w0:{2:6.3f}, w1:{3:6.3f}, b:{4:6.3f}".format(
        epoch, loss.item(), W[0].item(), W[1].item(), b.item()
      ), end=", ")
      print("W.grad: {0}, b.grad:{1}".format(W_grad, b_grad))

    W = W - LEARNING_RATE * W_grad
    b = b - LEARNING_RATE * b_grad
```

Backpropagation

◈Training with Backpropgation (3/3)

```
def main():
    W = torch.ones((2,))
    b = torch.zeros((1,))

    simple_dataset = SimpleDataset()
    train_data_loader = DataLoader(dataset=simple_dataset, batch_size=len(simple_dataset))
    batch = next(iter(train_data_loader))

    y_pred = model(batch["input"], W, b)
    print(y_pred.shape) # >>> torch.Size([12])
    print(y_pred)       # >>> tensor([0.8022, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
                        #             0.9973, 0.0198, 1.0000, 1.0000, 1.0000, 0.6225])
    loss = loss_fn(y_pred, batch["target"])
    print(loss)         # >>> tensor(0.2254)

    learn(W, b, train_data_loader)

if __name__ == "__main__":
    main()
```

Backpropagation with PyTorch Autograd

Autograd

◈Pytorch Autograd

- Creating Tensors with requires_grad=True enables autograd
- Operations on Tensors with requires_grad=True cause PyTorch to build a computational graph
- grad attribute is created, but it is None
- Since w is created by the user, its grad_fn is None

- We will not want gradients (of loss) with respect to just constant (or gathered data)

- x was created as a result of an operation, so it has grad & grad_fn attributes
- grad_fn references a Function that has created the Tensor

```
w = torch.ones(3, requires_grad=True)
print(w)
# >>> tensor([1., 1., 1.], requires_grad=True)
print(w.grad, w.grad_fn)
# >>> None None

c = torch.tensor([2])

x = w + c
print(x)
# >>> tensor([3., 3., 3.], grad_fn=<AddBackward0>)
print(x.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

# Do more operations on x
y = x * 3
print(y)
# >>> tensor([27., 27., 27.], grad_fn=<MulBackward0>)
print(y.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

z = y.mean()   # Make the output scalar
print(z)    # >>> tensor(27., grad_fn=<MeanBackward0>)
print(z.shape) # >>> torch.Size([])
print(z.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>
```

link_dl/_01_code/_04_learning_and_autograd/b_autograd_1.py

Autograd

◈Pytorch Autograd

y was created as a result of an operation, so it has grad & grad_fn attributes.

z was once more created as a result of 'mean' operator, so it still keeps grad & grad_fn attributes.

```
w = torch.ones(3, requires_grad=True)
print(w)
# >>> tensor([1., 1., 1.], requires_grad=True)
print(w.grad, w.grad_fn)
# >>> None None

c = torch.tensor([2])

x = w + c
print(x)
# >>> tensor([3., 3., 3.], grad_fn=<AddBackward0>)
print(x.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

# Do more operations on x
y = x * 3
print(y)
# >>> tensor([27., 27., 27.], grad_fn=<MulBackward0>)
print(y.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

z = y.mean()  # Make the output scalar
print(z)    # >>> tensor(27., grad_fn=<MeanBackward0>)
print(z.shape) # >>> torch.Size([])
print(z.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>
```

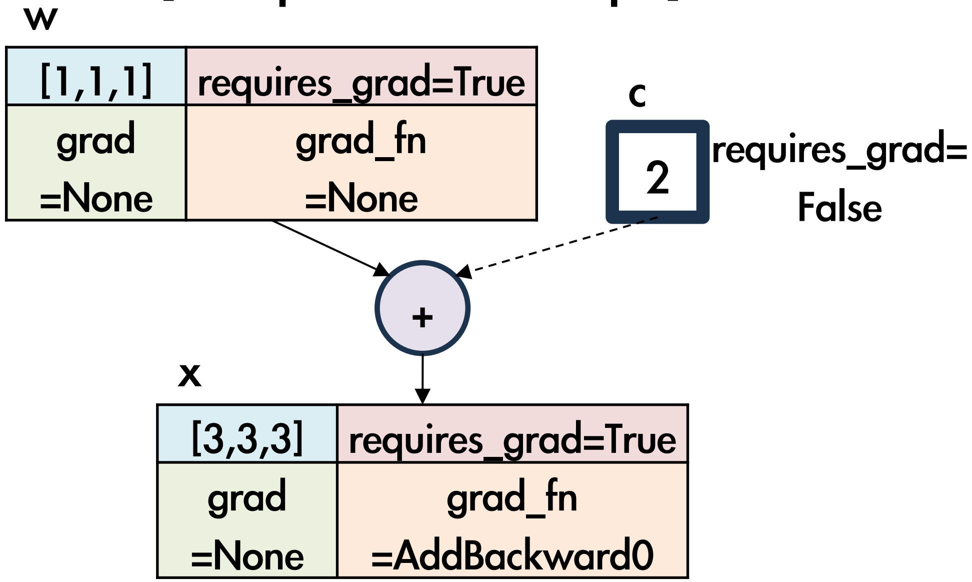
link_dl/_01_code/_04_learning_and_autograd/b_autograd_1.py

Autograd

◈Computational Graph

Every operation on a tensor with requires_grad=True will add to the computational graph, and the resulting tensors will also have requires_grad=True

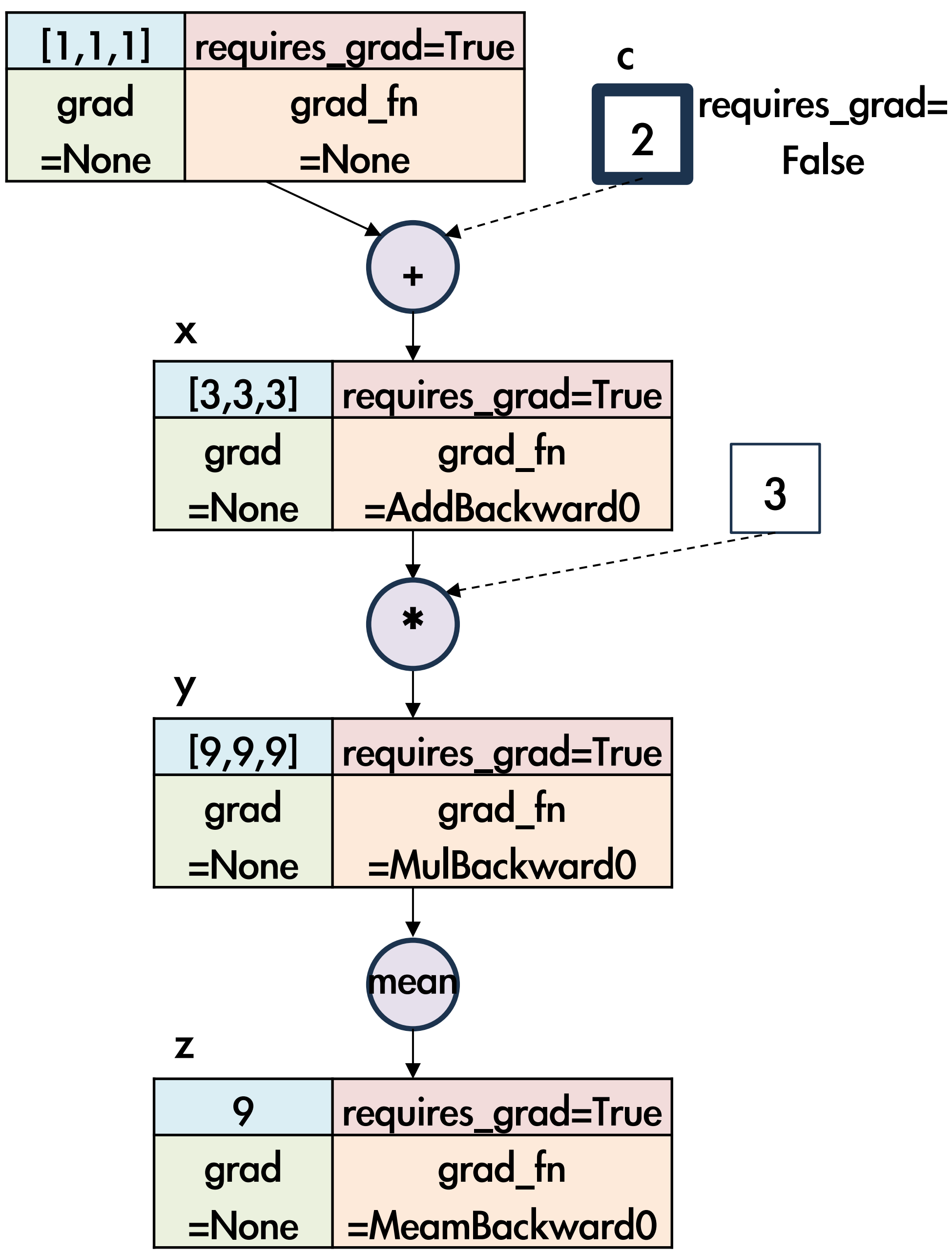
[Computational Graph]



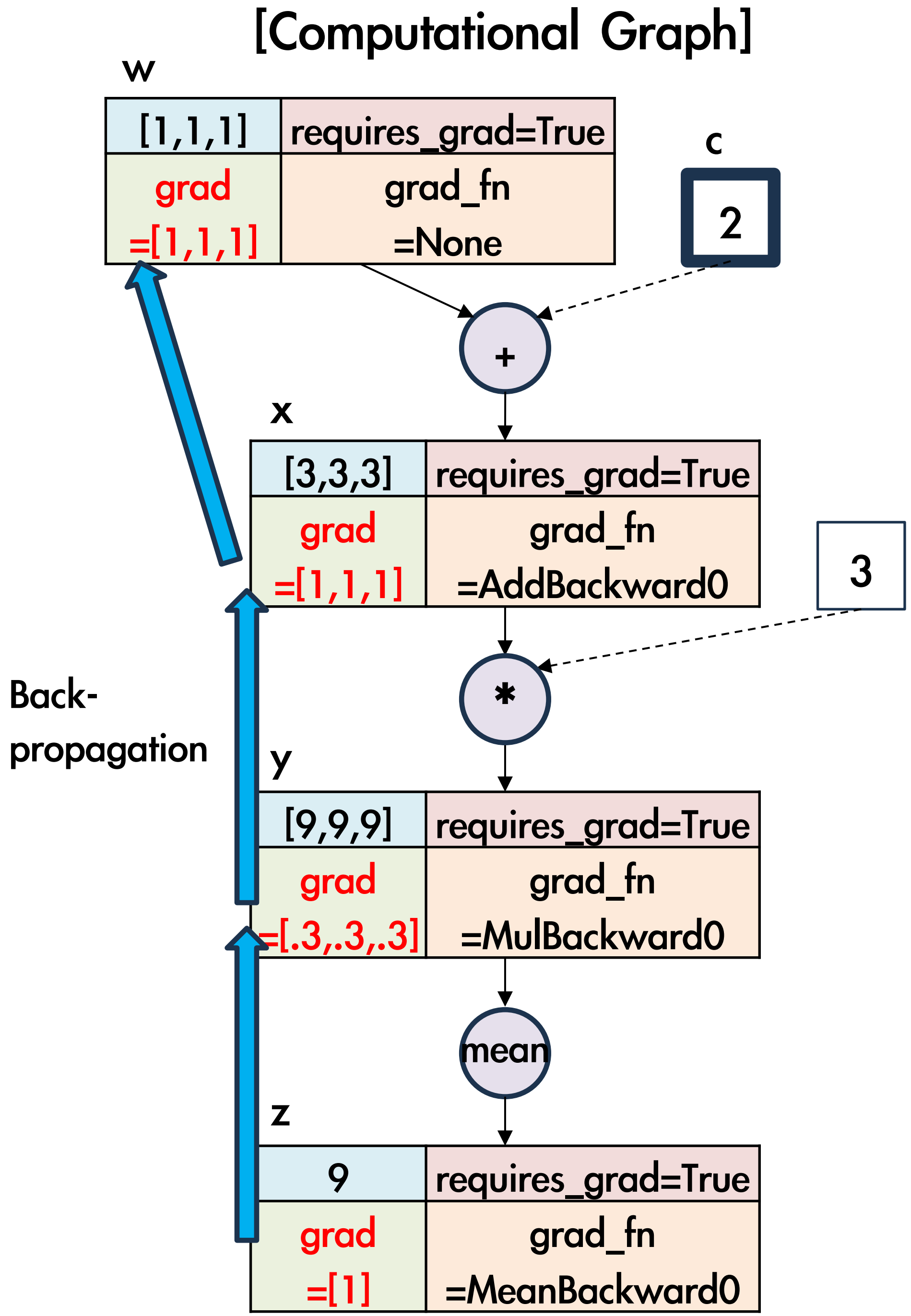

```python
w = torch.ones(3, requires_grad=True)
print(w)
# >>> tensor([1., 1., 1.], requires_grad=True)
print(w.grad, w.grad_fn)
# >>> None None

c = torch.tensor([2])

x = w + c
print(x)
# >>> tensor([3., 3., 3.], grad_fn=<AddBackward0>)
print(x.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

# Do more operations on x
y = x * 3
print(y)
# >>> tensor([27., 27., 27.], grad_fn=<MulBackward0>)
print(y.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

z = y.mean()   # Make the output scalar
print(z)    # >>> tensor(27., grad_fn=<MeanBackward0>)
print(z.shape) # >>> torch.Size([])
print(z.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>
```

link_dl/_01_code/_04_learning_and_autograd/b_autograd_1.py

[Computational Graph]




```python
w = torch.ones(3, requires_grad=True)
print(w)
# >>> tensor([1., 1., 1.], requires_grad=True)
print(w.grad, w.grad_fn)
# >>> None None

c = torch.tensor([2])

x = w + c
print(x)
# >>> tensor([3., 3., 3.], grad_fn=<AddBackward0>)
print(x.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

# Do more operations on x
y = x * 3
print(y)
# >>> tensor([27., 27., 27.], grad_fn=<MulBackward0>)
print(y.grad_fn)
# >>> <AddBackward0 object at 0x11b082ef0>

z = y.mean()  # Make the output scalar
print(z)    # >>> tensor(27., grad_fn=<MeanBackward0>)
print(z.shape) # >>> torch.Size([])
print(z.grad_fn)
# >>> <MeanBackward0 object at 0x11b082ef0>
```



Autograd

◆ **tensor.backward()**

– Backpropate to all tensors with requires_grad=True

– After backward finishes, gradients are accumulated into w.grad, x.grad, y.grad and z.grad and <u>the graph is destroyed</u>

```
...
...
z.backward()

print(w.grad)
# >>> tensor([1., 1., 1.])
```

Autograd

w

[1,1,1]	requires_grad=True
grad =[1,1,1]	grad_fn =None

x

[3,3,3]	requires_grad=True
grad =[1,1,1]	grad_fn =AddBackward0

y

[9,9,9]	requires_grad=True
grad =[.3,.3,.3]	grad_fn =MulBackward0

z

9	requires_grad=True
grad =[1]	grad_fn =MulBackward0

◈ tensor.backward()

- By default, you're not allowed to print the gradients for intermediate tensors

```
...
...
z.backward()

print(z.grad)
# >>> UserWarning: The .grad attribute of a Tensor tha
t is not a leaf Tensor is being accessed.
print(y.grad)
# >>> UserWarning: The .grad attribute of a Tensor tha
t is not a leaf Tensor is being accessed.
print(x.grad)
# >>> UserWarning: The .grad attribute of a Tensor
that is not a leaf Tensor is being accessed.
print(w.grad)
# >>> tensor([1., 1., 1.])
```

Autograd

w

[1,1,1]	requires_grad=True
grad =[1,1,1]	grad_fn =None

x

[3,3,3]	requires_grad=True
grad =[1,1,1]	grad_fn =AddBackward0

y

[9,9,9]	requires_grad=True
grad =[.3,.3,.3]	grad_fn =MulBackward0

z

9	requires_grad=True
grad =[1]	grad_fn =MulBackward0

◈ tensor.backward()

– If you indeed want the Tensor.grad field to be populated for a non-leaf Tensor, use Tensor.retain_grad() on the non-leaf Tensor

```
...
...
y.retain_grad()

z = y.mean()   # Make the output scalar
print(z)     # >>> tensor(27., grad_fn=<MeanBackward0>)
print(z.shape) # >>> torch.Size([])
print(z.grad_fn)
# >>> <MeanBackward0 object at 0x11b082ef0>

z.backward()

print(y.grad)  # dz/dy
# >>> tensor([0.3333, 0.3333, 0.3333])
```

Autograd

- Stop a tensor from tracking history
  - Tensor.requires_grad_()
    - changes a tensor's gradient requirement
      - .requires_grad_(True)
      - .requires_grad_(False)

```
a = torch.randn(2, 2)
print(a.requires_grad)   # >>> False
b = ((a * 3) / (a - 1))
print(b.grad_fn)         # >>> None
#b.backward()
# >>> RuntimeError: element 0 of tensors does not
require grad and does not have a grad_fn

a.requires_grad_(True)

print(a.requires_grad)   # >>> True

c = (a * a).sum()
print(c.grad_fn)
# >>> <SumBackward0 object at 0x116773d60>

c.backward()

print(a.grad)
# >>>
tensor([[3.9463, 0.2887],
        [0.5109, 3.0314]])
```

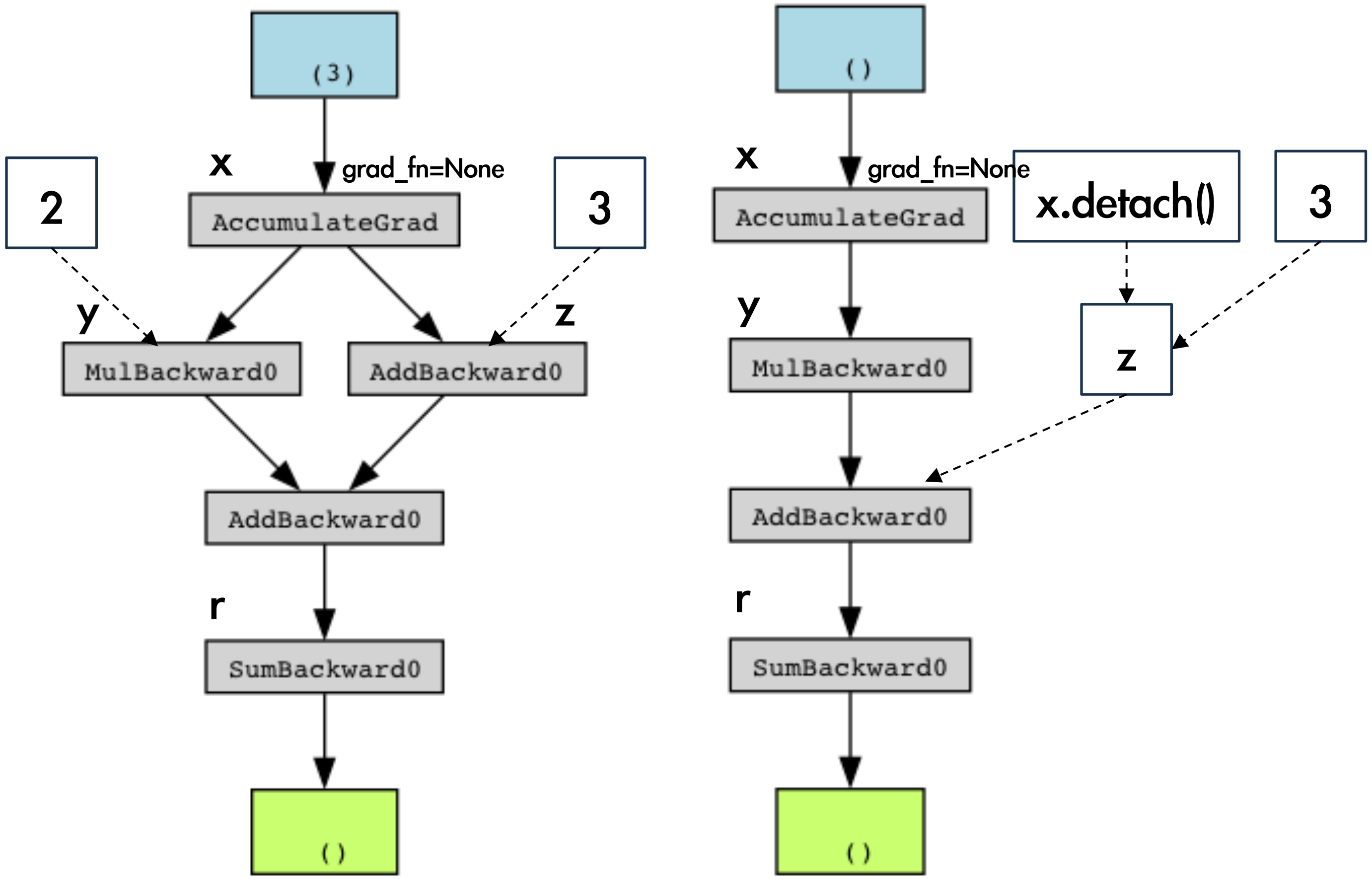
Autograd

- Stop a tensor from tracking history
  - Tensor.detach()
    - Returns a new Tensor, detached from the current computational graph
      - get a new Tensor with the same content but no gradient computation

```python
a = torch.randn(2, 2, requires_grad=True)

print(a.requires_grad)    # >>> True

# b is a new tensor detached from the current
# computational graph
b = a.detach()

print(b.requires_grad)    # >>> False

print(b is a)             # >>> False
```

Autograd

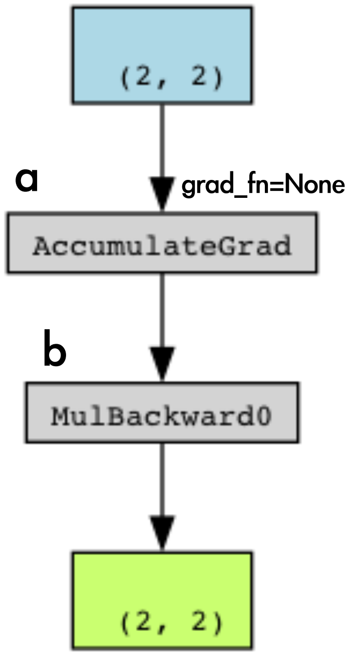
- Stop a tensor from tracking history
  - Tensor.detach()

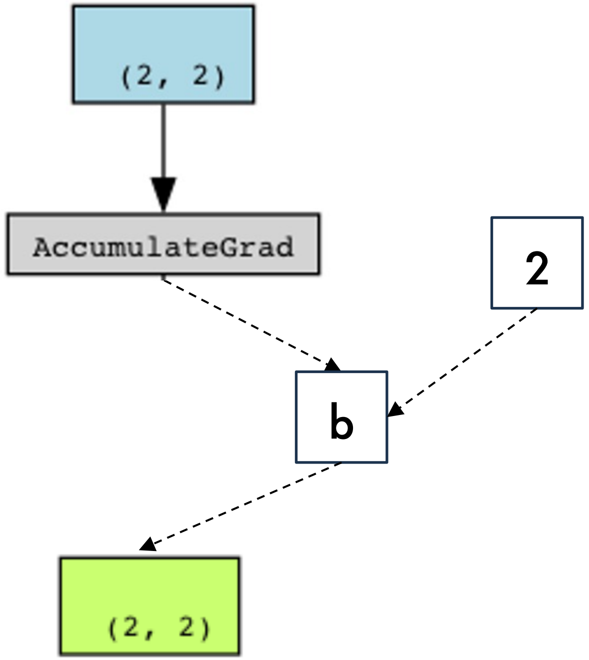



```python
# pip install graphviz
# pip install torchviz
from torchviz import make_dot

x = torch.ones(3, requires_grad=True)
y = 2 * x
z = 3 + x
r = (y + z).sum()
make_dot(r).render("torchviz_1", format="png")

# Detach
x = torch.ones(3, requires_grad=True)
y = 2 * x
z = 3 + x.detach()
r = (y + z).sum()
make_dot(r).render("torchviz_2", format="png")
```

Autograd

- Stop a tensor from tracking history
  - wrap in with torch.no_grad()
    - prevent gradient calculations for the wrapped parts of code to save memory and computational resources
    - or using the @torch.no_grad on function def.






```python
a = torch.randn(2, 2, requires_grad=True)
b = a * 2
make_dot(b).render("torchviz_3", format="png")

a = torch.randn(2, 2, requires_grad=True)
print(a.requires_grad)          # >>> True

# Tell PyTorch not to build a graph
# for the wrapped operations
with torch.no_grad():
    print(a.requires_grad)     # >>> True
    b = a * 2

print(a.requires_grad)     # >>> True

print(b.requires_grad)     # >>> False

make_dot(b).render("torchviz_4", format="png")
```

Gradient Descent with Autograd

```
weights = torch.ones(4, requires_grad=True)

for epoch in range(3):
    # just a dummy example
    output = (weights * 3).sum()
    print("[output {0}]:".format(epoch), output)
    output.backward()

    print("weights.grad:", weights.grad)

    # optimize model by gradient descent method
    with torch.no_grad():
        weights -= 0.1 * weights.grad
        # 'empty gradients' is important!
        # It affects the final weights & output
        weights.grad.zero_()

    print("weights:", weights)

output = (weights * 3).sum()
print("\n[output final]:", output)
```

- Empty gradients
  - Tensor.backward() **accumulates the gradient** for this tensor into the .grad attribute
    - We need to be careful during optimization!!!
  - Tensor.grad.zero_() empty the gradients before a new optimization step!

Tell PyTorch not to build a graph for these operations

Make gradient step on weights

Set gradients to zero

Gradient Descent with Autograd

```
weights = torch.ones(4, requires_grad=True)

for epoch in range(3):
    # just a dummy example
    output = (weights * 3).sum()
    print("[output {0}]:".format(epoch), output)
    output.backward()

    print("weights.grad:", weights.grad)

    # optimize model by gradient descent method
    with torch.no_grad():
        weights -= 0.1 * weights.grad
        # 'empty gradients' is important!
        # It affects the final weights & output
        weights.grad.zero_()

    print("weights:", weights)

output = (weights * 3).sum()
print("\n[output final]:", output)
```

- Empty gradients
  - Tensor.backward() **accumulates the gradient** for this tensor into the .grad attribute
    - We need to be careful during optimization !!!
  - Tensor.grad.zero_() empty the gradients before a new optimization step!

```
# >>>
[output 0]: tensor(12., grad_fn=<SumBackward0>)
weights.grad: tensor([3., 3., 3., 3.])
weights: tensor([0.7000, 0.7000, 0.7000, 0.7000], requires_g
rad=True)
[output 1]: tensor(8.4000, grad_fn=<SumBackward0>)
weights.grad: tensor([3., 3., 3., 3.])
weights: tensor([0.4000, 0.4000, 0.4000, 0.4000], requires_g
rad=True)
[output 2]: tensor(4.8000, grad_fn=<SumBackward0>)
weights.grad: tensor([3., 3., 3., 3.])
weights: tensor([0.1000, 0.1000, 0.1000, 0.1000], requires_g
rad=True)
[output final]: tensor(1.2000, grad_fn=<SumBackward0>)
```

Gradient Descent with Autograd

```
weights = torch.ones(4, requires_grad=True)


for epoch in range(3):
    # just a dummy example
    output = (weights * 3).sum()
    print("[output {0}]:".format(epoch), output)
    output.backward()

    print("weights.grad:", weights.grad)

    # optimize model by gradient descent method
    with torch.no_grad():
        weights -= 0.1 * weights.grad
        # 'empty gradients' is important!
        # It affects the final weights & output
        weights.grad.zero_()

    print("weights:", weights)

output = (weights * 3).sum()
print(”\n[output final]:", output)
```

```
weights = torch.ones(4, requires_grad=True)
optimizer = torch.optim.SGD([weights], lr=0.1)

for epoch in range(3):
    # just a dummy example
    output = (weights * 3).sum()
    print("[output {0}]:".format(epoch), output)
    output.backward()

    print("weights.grad:", weights.grad)

    # optimize model by gradient descent method
    optimizer.step()
    optimizer.zero_grad()  # empty gradients

    print("weights:", weights)

output = (weights * 3).sum()
print(”\n[output final]:", output)
```

Gradient Descent with Autograd

```
weights = torch.ones(4, requires_grad=True)
optimizer = torch.optim.SGD([weights], lr=0.1)

for epoch in range(3):
    # just a dummy example
    output = (weights * 3).sum()
    print("[output {0}]:".format(epoch), output)
    output.backward()

    print("weights.grad:", weights.grad)

    # optimize model by gradient descent method
    optimizer.step()
    optimizer.zero_grad()  # empty gradients


    print("weights:", weights)

output = (weights * 3).sum()
print(”\n[output final]:", output)
```

◈Update the tensors

– torch.optim.Optimizer
  - It helps in updating the parameters during performing the gradient descent
    - ➢ torch.optim.Optimizer.SGD
    - ➢ torch.optim.Optimizer.RMSprop
    - ➢ …

    We will study later

– torch.optim.Optimizer.step()
  - This is called after computing gradients using backpropagation
  - This updates the parameters of a neural network model using gradient descent during training

– torch.optim.Optimizer.zero_grad()
  - empty the gradients

```
# >>>
(Same output! As the previous run)
```

Gradient Descent with Autograd
- Measurements by new two sensors & Temperature -

Examples: Gradient Descent with Autograd

◈Example of Autograd at Gradient Descent

```python
def learn(W, b, train_data_loader):
  MAX_EPOCHS = 20_000
  LEARNING_RATE = 0.01

  for epoch in range(0, MAX_EPOCHS):
    batch = next(iter(train_data_loader))
    y_pred = model(batch["input"], W, b)
    loss = loss_fn(y_pred, batch["target"])

    loss.backward()

    if epoch % 100 == 0:
      print("[Epoch:{0:6,}] loss:{1:8.5f}, w0:{2:6.3f}, w1:{3:6.3f}, b:{4:6.3f}".format(
        epoch, loss.item(), W[0].item(), W[1].item(), b.item()
      ), end=", ")
      print("W.grad: {0}, b.grad:{1}".format(W.grad, b.grad))

    with torch.no_grad():
      W -= LEARNING_RATE * W.grad
      b -= LEARNING_RATE * b.grad
      W.grad = None
      b.grad = None
```

Examples: Gradient Descent with Autograd

◈Example of Autograd with Step() at Gradient Descent

```
def learn(W, b, train_data_loader):
  MAX_EPOCHS = 20_000
  LEARNING_RATE = 0.01

  from torch import optim
  optimizer = optim.SGD([W, b], lr=LEARNING_RATE)

  for epoch in range(0, MAX_EPOCHS):
    batch = next(iter(train_data_loader))
    y_pred = model(batch["input"], W, b)
    loss = loss_fn(y_pred, batch["target"])

    loss.backward()

    if epoch % 100 == 0:
      print("[Epoch:{0:6,}] loss:{1:8.5f}, w0:{2:6.3f}, w1:{3:6.3f}, b:{4:6.3f}".format(
        epoch, loss.item(), W[0].item(), W[1].item(), b.item()
      ), end=", ")
      print("W.grad: {0}, b.grad:{1}".format(W.grad, b.grad))

    optimizer.step()
    optimizer.zero_grad()
```

Examples: Gradient Descent with Autograd

◈Example of Autograd main()

```
def main():
    W = torch.ones((2,), requires_grad=True)
    b = torch.zeros((1,), requires_grad=True)

    simple_dataset = SimpleDataset()
    train_data_loader = DataLoader(dataset=simple_dataset, batch_size=len(simple_dataset))

    learn(W, b, train_data_loader)

if __name__ == "__main__":
    main()
```