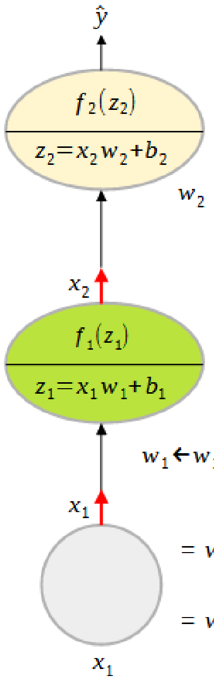
MX-AI



$$\frac{\partial L}{\partial w_2}$$

$$w_2 \leftarrow w_2 - \alpha \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$= w_2 - \alpha \underbrace{(\hat{y}-y) \cdot \frac{\partial f_2(z_2)}{\partial z_2}}_{e_2} \cdot x_2$$

$$= w_2 - \alpha \cdot e_2 \cdot f_2' \cdot x_2$$

$$w_1 \leftarrow w_1 - \alpha \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial x_2} \cdot \frac{\partial x_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

$$= w_1 - \alpha \underbrace{(\hat{y}-y) \cdot \frac{\partial f_2(z_2)}{\partial z_2} \cdot w_2}_{e_1} \cdot \frac{\partial f_1(z_1)}{\partial z_1} \cdot x_1$$

$$= w_1 - \alpha \cdot e_1 \cdot f_1' \cdot x_1$$

3. Error Backpropagation

Part 1: Backpropagation along single path

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

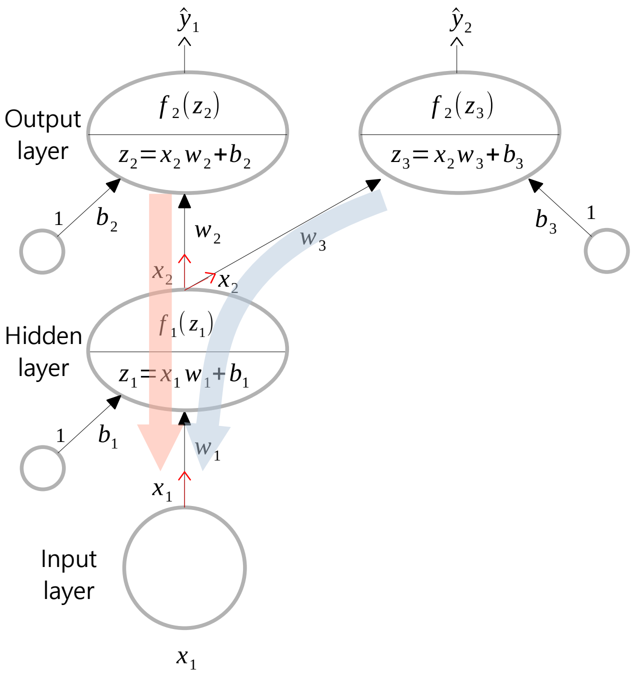
MX-AI

Error Backpropagation (BP)

[MXDL-3-01]
1. Backpropagation along single path
2. Interpretation of BP

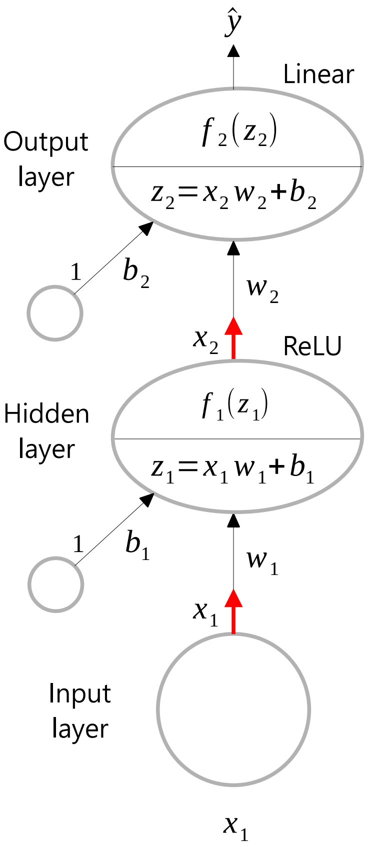
[MXDL-3-02]
3. Backpropagation along multiple paths
4. Activation function and Vanishing Gradient
5. Numerical differentiation and Automatic differentiation

[MXDL-3-03]
6. Automatic differentiation – Forward derivative trace
7. Automatic differentiation – Backward derivative trace
8. Tensorflow: GradientTape( )



MX-AI

■ Backpropagation along single path



$L = \frac{1}{2}(\hat{y} - y)^2$: Loss function

$\hat{y} = f_2(z_2) = f_2(x_2 w_2 + b_2)$

$x_2 = f_1(z_1) = f_1(x_1 w_1 + b_1)$

(f_1, f_2 : activation function)

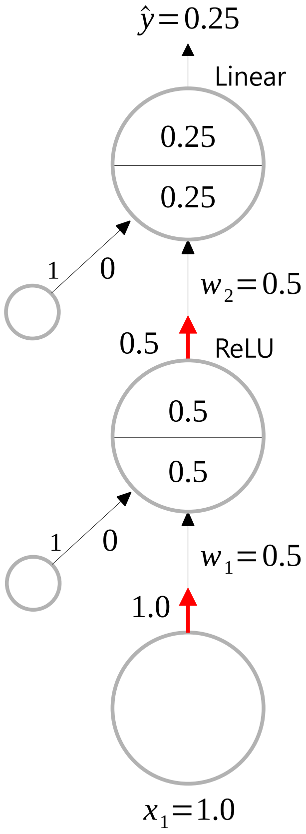
$w_2 \leftarrow w_2 - \alpha \frac{\partial L}{\partial w_2}$: Gradient Descent

$$w_2 \leftarrow w_2 - \alpha \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2} = w_2 - \alpha (\hat{y} - y) \cdot \frac{\partial f_2(z_2)}{\partial z_2} \cdot x_2$$

$w_1 \leftarrow w_1 - \alpha \frac{\partial L}{\partial w_1}$: Gradient Descent

$$w_1 \leftarrow w_1 - \alpha \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial x_2} \cdot \frac{\partial x_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

[Update w_2]



$y = 1$, $\alpha = 0.1$

$b_1 = b_2 = 0$

$f_2(x) = x$, $f_1(x) = ReLU(x)$

$$L = \frac{1}{2}(0.25 - 1)^2 = 0.2813$$

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = -0.75$$

$$\frac{\partial \hat{y}}{\partial z_2} = \frac{\partial f_2(z_2)}{\partial z_2} = \frac{\partial z_2}{\partial z_2} = 1$$

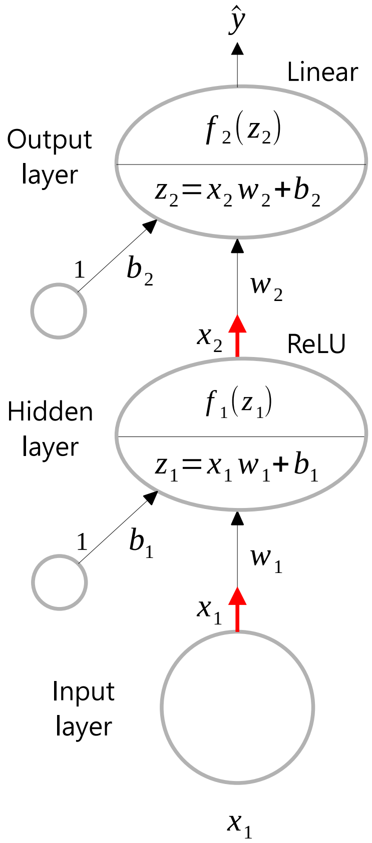
$$\frac{\partial z_2}{\partial w_2} = \frac{\partial (x_2 w_2 + b_2)}{\partial w_2} = x_2$$

$$w_2 = 0.5 - 0.1(-0.75 \times 1 \times 0.5) = 0.5375$$

* w_2 is updated to 0.5375.

MX-AI

■ Backpropagation along single path



$$L=\frac{1}{2}(\hat{y}-y)^2$$: Loss function

$$\hat{y}=f_2(z_2)=f_2(x_2w_2+b_2)$$

$$x_2=f_1(z_1)=f_1(x_1w_1+b_1)$$

(f_1, f_2 : activation function)

$$w_2 \leftarrow w_2-\alpha\frac{\partial L}{\partial w_2}$$: Gradient Descent

$$w_2 \leftarrow w_2-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial w_2}=w_2-\alpha(\hat{y}-y)\cdot\frac{\partial f_2(z_2)}{\partial z_2}\cdot x_2$$

$$w_1 \leftarrow w_1-\alpha\frac{\partial L}{\partial w_1}$$: Gradient Descent

$$w_1 \leftarrow w_1-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial x_2}\cdot\frac{\partial x_2}{\partial z_1}\cdot\frac{\partial z_1}{\partial w_1}$$

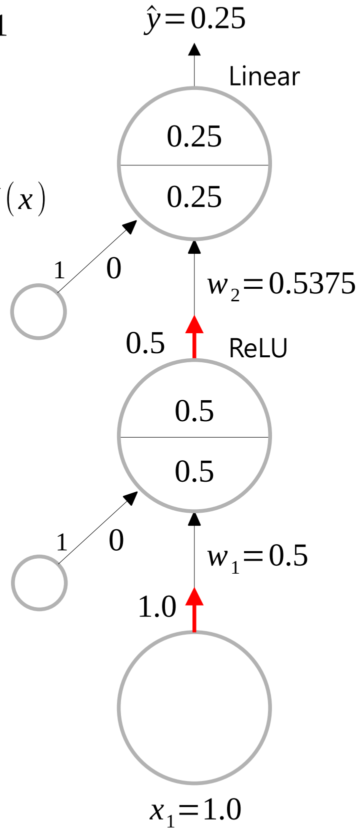
$$y=1,\ \alpha=0.1$$

$$b_1=b_2=0$$

$$f_1(x)=x$$

$$f_2(x)=ReLU(x)$$

[Update w_1]



$$L=\frac{1}{2}(0.25-1)^2=0.2813$$

$$\frac{\partial L}{\partial \hat{y}}=\hat{y}-y=-0.75$$

$$\frac{\partial \hat{y}}{\partial z_2}=\frac{\partial f_2(z_2)}{\partial z_2}=\frac{\partial z_2}{\partial z_2}=1$$

$$\frac{\partial z_2}{\partial x_2}=\frac{\partial(x_2w_2+b_2)}{\partial w_2}=w_2=0.5375$$

$$\frac{\partial x_2}{\partial z_1}=\frac{\partial(f_1(z_1))}{\partial z_1}=1 \quad (z_1>0)$$

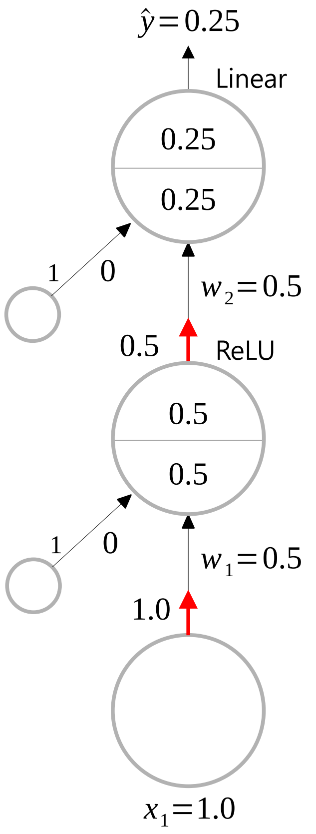
$$\frac{\partial z_1}{\partial w_1}=\frac{\partial(x_1w_1+b_1)}{\partial w_1}=x_1=1.0$$

* w_1 is updated to 0.5403

$$w_1=0.5-0.1(-0.75\times1\times0.5375\times1\times1)=0.5403$$

Backpropagation along single path

[before update]

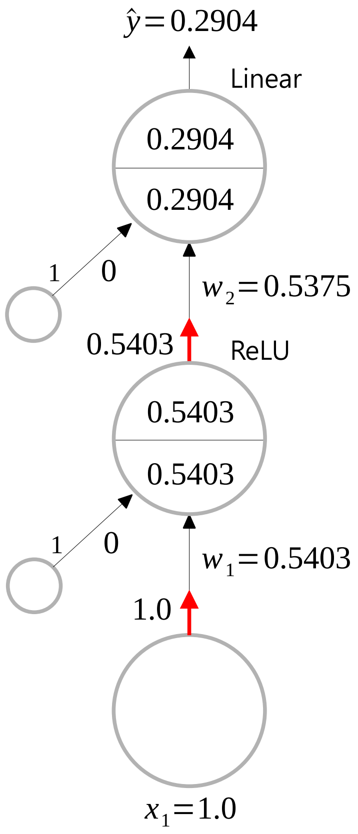


$$L = \frac{1}{2}(0.25-1)^2 = 0.2813$$

1 update

(b has not been updated.)

[after update]

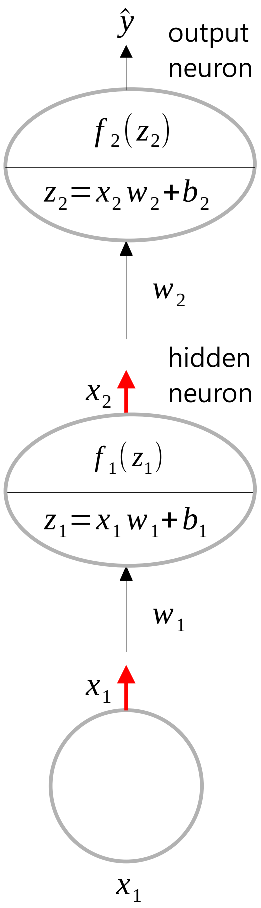


$$L = \frac{1}{2}(0.2904-1)^2 = 0.2518$$

- After updating once, the loss was reduced.
  0.2813 → 0.2518
- $\hat{y}$ got closer to y=1.
  0.25 → 0.2904

MX-AI

Backpropagation along single path – Interpretation



$$\frac{\partial L}{\partial w_2}$$

$$w_2 \leftarrow w_2 - \alpha \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$= w_2 - \alpha \underbrace{(\hat{y} - y)}_{e_2} \cdot \frac{\partial f_2(z_2)}{\partial z_2} \cdot x_2$$

$$= w_2 - \alpha \cdot e_2 \cdot f_2{}' \cdot x_2$$

$$w_1 \leftarrow w_1 - \alpha \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial x_2} \cdot \frac{\partial x_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

$$= w_1 - \alpha \underbrace{(\hat{y} - y) \cdot \frac{\partial f_2(z_2)}{\partial z_2} \cdot w_2}_{e_1} \cdot \frac{\partial f_1(z_1)}{\partial z_1} \cdot x_1$$

$$= w_1 - \alpha \cdot e_1 \cdot f_1{}' \cdot x_1$$

e_1 corresponds to the magnitude of the error propagated to lower layer. The error of a lower layer cannot be measured directly because the desired output of the lower layer is unknown. However, if this value is used as a proxy for the error, w_1 can be updated.

Error Back Propagation algorithm

w_2 is updated more as the error increases, the gradient of f_2 increases, and the input value increases. The larger these values, the more w must change to minimize L. In other words, w is currently far away from the target point and the value is greatly wrong.

1) If the error is 0, that is, if the output value of the output neuron is equal to the desired output value, then w_2 is correct and will not be updated.

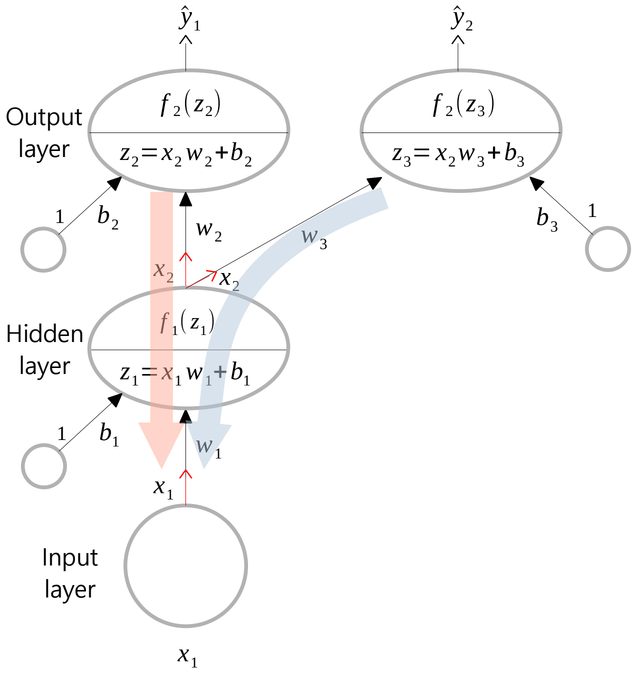
2) If the hidden neuron's output value is 0, the connection weight between the two neurons is not updated. This is because the hidden neuron did not contribute at all to the output neuron outputting an incorrect result.

3) The derivative of f_2 tells you the output's sensitivity to change with respect to a change in its input, z_2.

If z changes slightly but the output changes significantly, the stability of this network decreases. Therefore, if w is at the point where the derivative of f2 is large, it is better to change w significantly in order to quickly escape from this point.

For example, if f_2 is sigmoid and z is 0, the derivative of f_2 has a maximum value of 0.25. And f_2 becomes 0.5. If this output is used for binary classification, the output 0.5 will not help at all. In binary classification, the closer the output is to 0 or 1, the better.

MX-AI



3. Error Backpropagation

Part 2: Backpropagation along multiple paths

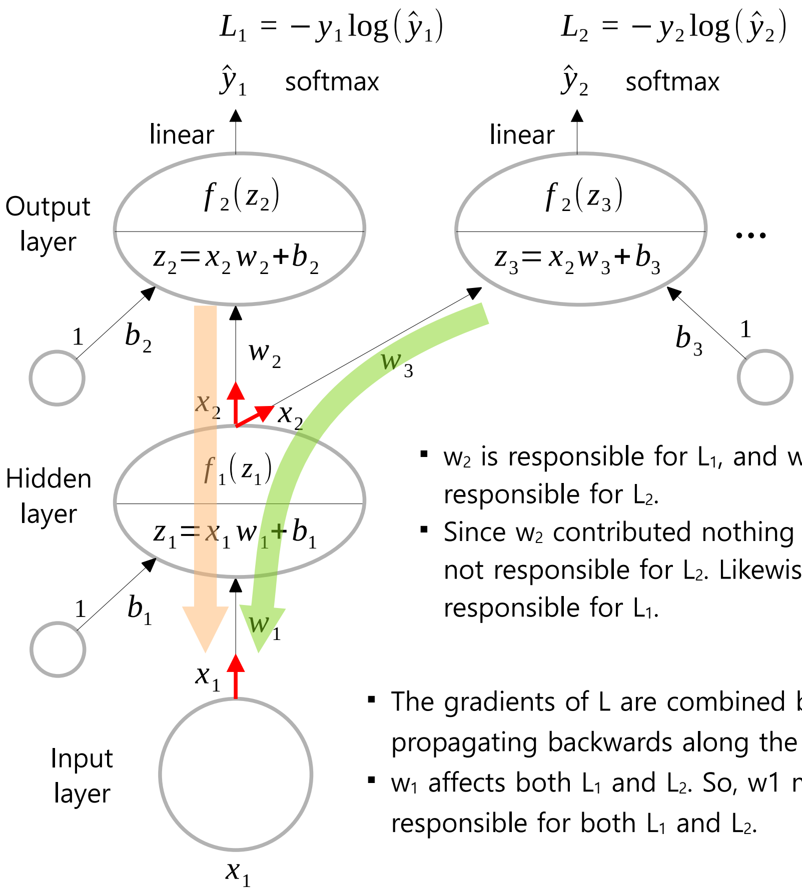
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

■ Backpropagation along multiple paths

Example of multiclass classification



- w_2 is responsible for L_1, and w_3 is responsible for L_2.
- Since w_2 contributed nothing to L_2, w_2 is not responsible for L_2. Likewise, w_3 is not responsible for L_1.

- The gradients of L are combined by propagating backwards along the two paths.
- w_1 affects both L_1 and L_2. So, w1 must be responsible for both L_1 and L_2.

$$L = L_1 + L_2 + \ldots = -\sum_{k=1}^{C} y_{i,k} \log(\hat{y}_{i,k})$$

$$w_2 \leftarrow w_2 - \alpha \frac{\partial L_1}{\partial w_2} \qquad \left(\frac{\partial L_2}{\partial w_2} = 0\right)$$

$$w_2 \leftarrow w_2 - \alpha \frac{\partial L_1}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$w_3 \leftarrow w_3 - \alpha \frac{\partial L_2}{\partial w_3} \qquad \left(\frac{\partial L_1}{\partial w_3} = 0\right)$$

$$w_3 \leftarrow w_3 - \alpha \frac{\partial L_2}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3}$$

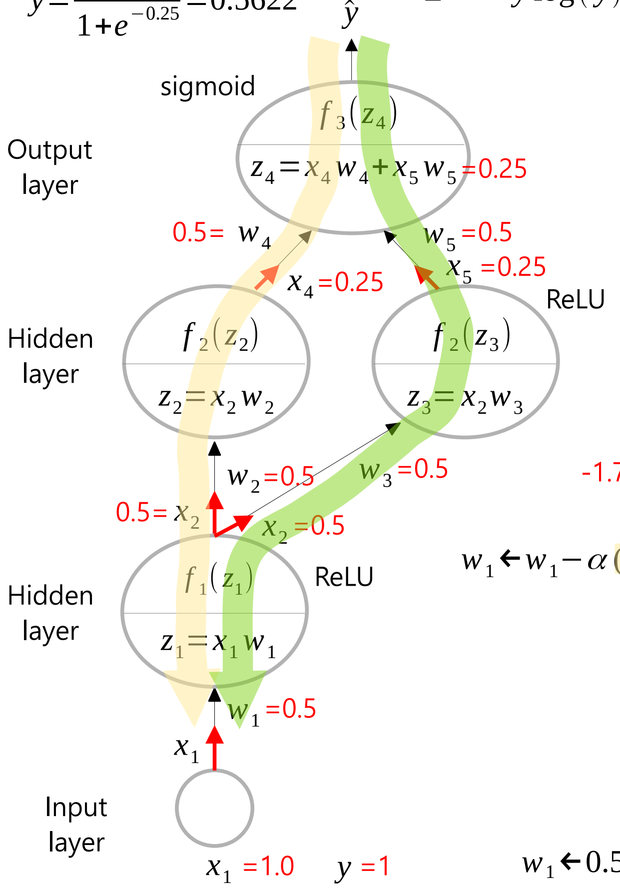
$$w_1 \leftarrow w_1 - \alpha \frac{\partial (L_1 + L_2)}{\partial w_1}$$

$$w_1 \leftarrow w_1 - \alpha \left[\frac{\partial L_1}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_2} \cdot \overset{w_2}{\frac{\partial z_2}{\partial x_2}} \cdot \frac{\partial x_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} + \frac{\partial L_2}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_3} \cdot \overset{w_3}{\frac{\partial z_3}{\partial x_2}} \cdot \frac{\partial x_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}\right]$$

MX-AI

■ Backpropagation along multiple paths

$$\hat{y}=\frac{1}{1+e^{-0.25}}=0.5622 \qquad L=-y\log(\hat{y})-(1-y)\log(1-\hat{y})=0.5759$$



$$\frac{\partial L}{\partial \hat{y}}=-\frac{y}{\hat{y}}+\frac{1-y}{1-\hat{y}}=-1.7787$$

$$\frac{\partial \hat{y}}{\partial z_4}=\hat{y}(1-\hat{y})=0.2461$$

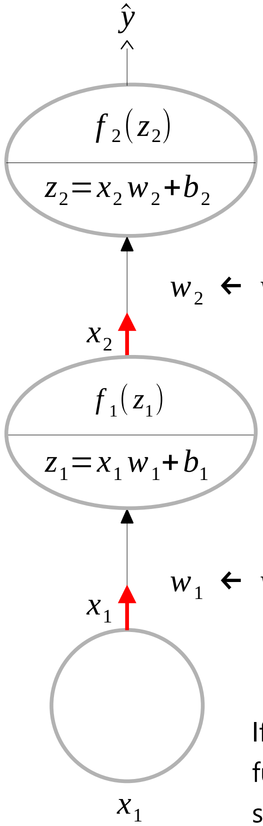
$$w_1 \leftarrow w_1-\alpha\left(\overset{-1.7787}{\frac{\partial L}{\partial \hat{y}}}\cdot\overset{0.2461}{\frac{\partial \hat{y}}{\partial z_4}}\cdot\overset{w_4}{\frac{\partial z_4}{\partial x_4}}\cdot\overset{1}{\frac{\partial x_4}{\partial z_2}}\cdot\overset{w_2}{\frac{\partial z_2}{\partial x_2}}\cdot\overset{1}{\frac{\partial x_2}{\partial z_1}}\cdot\overset{x_1}{\frac{\partial z_1}{\partial w_1}}+\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_4}\cdot\underset{w_5}{\frac{\partial z_4}{\partial x_5}}\cdot\underset{1}{\frac{\partial x_5}{\partial z_3}}\cdot\underset{w_3}{\frac{\partial z_3}{\partial x_2}}\cdot\frac{\partial x_2}{\partial z_1}\cdot\frac{\partial z_1}{\partial w_1}\right)$$

$$w_1 \leftarrow 0.5-0.1(-0.1094-0.1094)=0.5219$$

```
import numpy as np
import tensorflow as tf

x = np.array([[1.0]])
y = np.array([[1.0]])

w0 = tf.Variable(np.array([[0.5]]))
w1 = tf.Variable(np.array([[0.5, 0.5]]))
w2 = tf.Variable(np.array([[0.5], [0.5]]))
parameters = [w0, w1, w2]

def bce(y, y_hat):
    return -y * tf.math.log(y_hat) - \
                (1. - y) * tf.math.log(1. - y_hat))
def predict(x):
    h1 = tf.nn.relu(tf.matmul(x, parameters[0]))
    h2 = tf.nn.relu(tf.matmul(h1, parameters[1]))
    return tf.sigmoid(tf.matmul(h2, parameters[2]))

with tf.GradientTape() as tape:
    loss = bce(y, predict(x))

grads = tape.gradient(loss, parameters)

for i, p in enumerate(parameters):
    p.assign_sub(0.1 * grads[i])

print(w2.numpy())
print(w1.numpy())
print(w0.numpy())

[[0.51094559]
 [0.51094559]]
[[0.51094559 0.51094559]]
[[0.52189117]] ← w1
```


MX-AI

■ Activation function and Vanishing Gradient



error | differential value of the activation function (f_2) | output of lower neuron

$$w_2 \leftarrow w_2-\alpha(\hat{y}-y)\cdot\frac{\partial f_2(z_2)}{\partial z_2}\cdot x_2$$

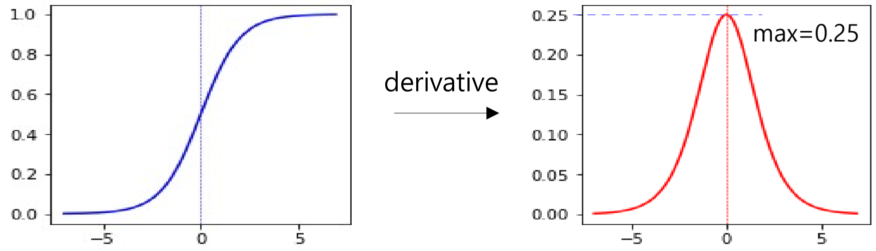
As we go down to lower layers, the derivatives of the activation function continue to be multiplied.

$$w_1 \leftarrow w_1-\alpha\underbrace{(\hat{y}-y)\cdot\frac{\partial f_2(z_2)}{\partial z_2}\cdot w_2}_{e_1}\cdot\frac{\partial f_1(z_1)}{\partial z_1}\cdot x_1$$

If there are five hidden layers, the activation functions of the hidden layers are all sigmoid, and if the differential values are all 0.2, then 0.2 is multiplied 5 times.

$$0.2^5=3.2\times10^{-4}$$

▪ sigmoid : $f(x)=\frac{1}{1+e^{-x}} \equiv \sigma(x)$ $\qquad \frac{d}{dx}f(x)=\sigma(x)\cdot(1-\sigma(x))$

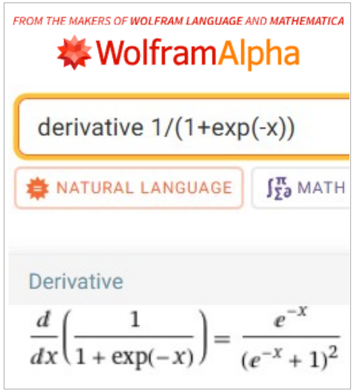


- The maximum value of the derivative of sigmoid is 0.25. When using a sigmoid activation function in a multi-layer neural network, the amount of back-propagated error decreases exponentially because values smaller than 0.25 are multiplied multiple times as it goes down to the lower layers. This is called **vanishing gradient** problem.
- The maximum value of the derivative of tanh is 0.25. This is larger than that of sigmoid but smaller than 1, so the vanishing gradient problem still occurs.
- To alleviate this problem, ReLU is widely used in hidden layers. ReLU is a non-linear function that combines two linear functions. The gradient of ReLU is either 0 or 1. If it is 1, then the gradient propagates well to the lower layer. If it is 0, the gradient will not propagate. However, in this case, the output value of some neurons in the hidden layer become 0, causing some of the dropout and regularization effects that we will look at later, which can prevent overfitting.
- In the worst case, if the output of all neurons in the hidden layer becomes 0, the gradient cannot propagate to lower layers. This is called the **dying ReLU** problem. This problem may appear if the learning rate, alpha is large or the bias is large and negative. This is because they make the weights negative and the output of ReLU 0. To alleviate this problem, You need to lower the learning rate and prevent the bias from becoming too large. You can also try using the Leaky ReLU or softplus activation function.

MX-AI

■ Numerical Differentiation and Automatic Differentiation

- Finding the gradient of the loss L requires a differentiation algorithm.



Differentiation

- Symbolic differentiation ← It manipulates mathematical expressions.
- Numerical differentiation ←

$$\frac{df}{dx} \approx \frac{f(x+h)-f(x)}{h}$$ (Forward difference)

$$\frac{df}{dx} \approx \frac{f(x+h)-f(x-h)}{2h}$$ (Center difference)

← Finding the approximate differential value of a function at point x. (h = small value)

- Automatic differentiation

Finding the exact differential value of a function at point x.

  - Forward derivative trace ← Computes the partial derivative of a variable during the forward pass of a function. Because it is slow, it is not suitable for gradient-based optimization.
  - Reverse derivative trace (Backward) ← After evaluating the function, it calculates the partial derivatives of the variables during the backward pass. This mode is faster than forward mode.
tensorflow's GradientTape() works this way.

MX-AI

Numerical Differentiation

$$y=f(x_1,x_2)=\log(x_1)+x_1\cdot x_2-\sin(x_2) \qquad (x_1,x_2)=(2,5)$$

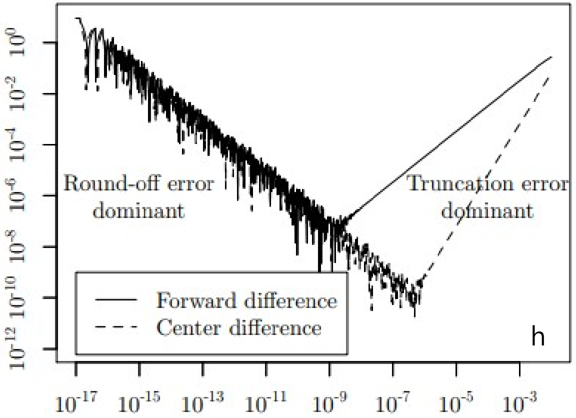
- Forward difference approximation

$$\frac{\partial y}{\partial x_1}\approx\frac{f(x_1+h,x_2)-f(x_1,x_2)}{h} \qquad \frac{\partial y}{\partial x_2}\approx\frac{f(x_1,x_2+h)-f(x_1,x_2)}{h}$$

- Center difference approximation

$$\frac{\partial y}{\partial x_1}\approx\frac{f(x_1+h,x_2)-f(x_1-h,x_2)}{2h} \qquad \frac{\partial y}{\partial x_2}\approx\frac{f(x_1,x_2+h)-f(x_1,x_2-h)}{2h}$$

$$f(x)=64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

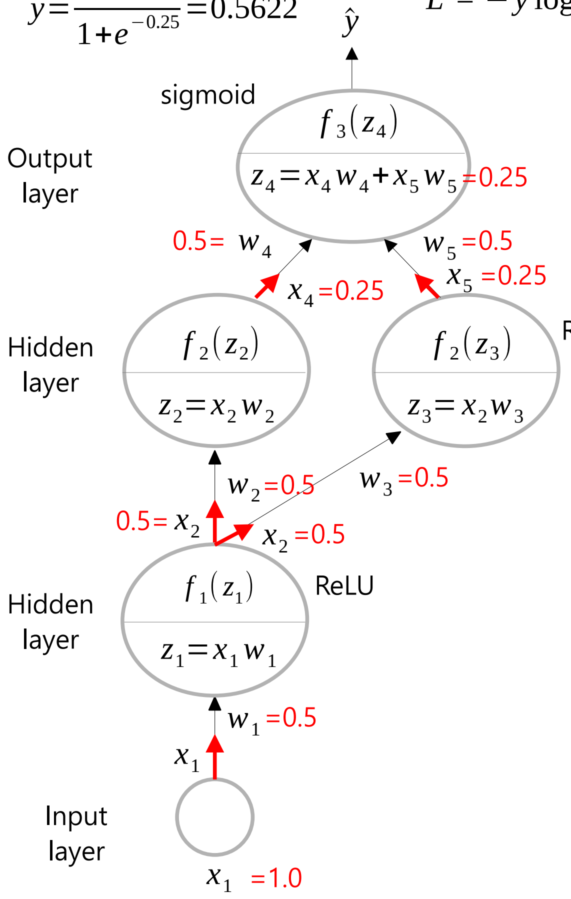



```
import numpy as np

# Define a function
def f(x1, x2):
    return np.log(x1) + x1 * x2 - np.sin(x2)

x1 = 2.0
x2 = 5.0
h = 1e-4  # small value

# center difference
# Compute the gradients at (x1, x2) = (2, 5)
dx1 = (f(x1 + h, x2) - f(x1 - h, x2)) / (2.0 * h)
dx2 = (f(x1, x2 + h) - f(x1, x2 - h)) / (2.0 * h)

print('dx1 = {:.4f}, dx2 = {:.4f}'.format(dx1, dx2))

Results:

dx1 = 5.5000, dx2 = 1.7163

(if h = 1e-15, dx1 = 4.441, dx2 = 1.776)
```

- Source : Baydin, et, al., 2018, Automatic Differentiation in Machine Learning: a Survey (Figure 3)

Updating parameters using numerical differentiation

$$\hat{y}=\frac{1}{1+e^{-0.25}}=0.5622 \qquad L=-y\log(\hat{y})-(1-y)\log(1-\hat{y})$$



$$\frac{\partial L}{\partial w_1}\approx\frac{L(w_1+h)-L(w_1-h)}{2h}$$

$$w_1\leftarrow 0.5-0.1\frac{\partial L}{\partial w_1}=0.5219$$

Using automatic differentiation by tensorflow's GradientTape()

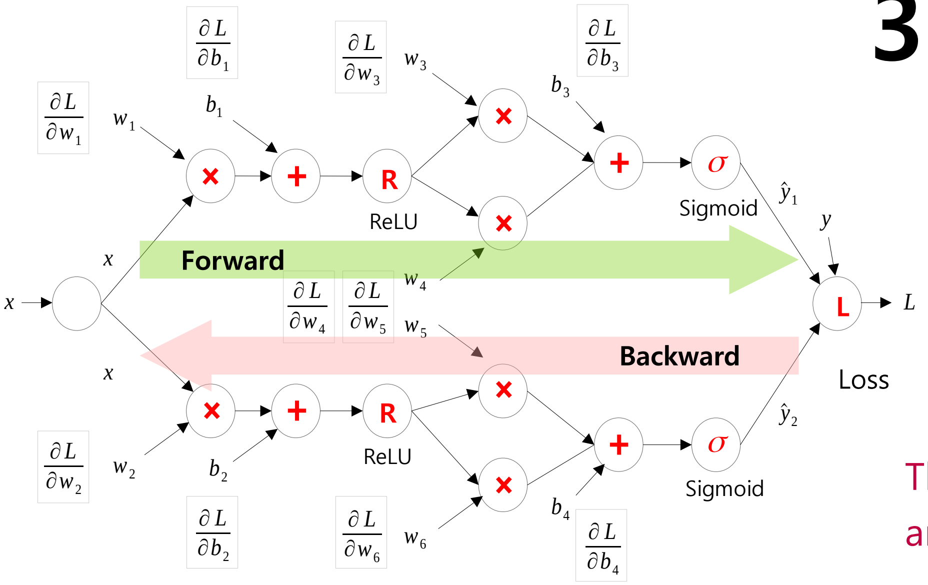
```
[[0.51094559]
 [0.51094559]]
[[0.51094559 0.51094559]]
[[0.52189117]]  ← w1
```

Using numerical differentiation

```
[[0.51094559]
 [0.51094559]]
[[0.51094559 0.51094559]]
[[0.52189117]]  ← w1
```

```
import numpy as np
x = np.array([[1.0]]);   y = np.array([[1.0]]);  h = 1e-4
w0 = np.array([[0.5]])
w1 = np.array([[0.5, 0.5]])
w2 = np.array([[0.5], [0.5]])
parameters = [w0, w1, w2]

def sigmoid(x): return 1. / (1. + np.exp(-x))
def relu(x):    return np.maximum(0, x)
def bce(y, y_hat):
    return -np.mean(y*np.log(y_hat) + (1.-y)*np.log(1.-y_hat))

def predict(x):
    h1 = relu(np.dot(x, parameters[0]))
    h2 = relu(np.dot(h1, parameters[1]))
    return sigmoid(np.dot(h2, parameters[2]))

p_gradients = []
for p in parameters:
    grad = np.zeros_like(p)
    for row in range(p.shape[0]):
        for col in range(p.shape[1]):
            p_org = p[row, col]
            p[row, col] = p_org + h
            L1 = bce(y, predict(x))

            p[row, col] = p_org - h
            L2 = bce(y, predict(x))
            grad[row, col] = (L1 - L2) / (2. * h)
            p[row, col] = p_org
    p_gradients.append(grad)

for i in range(len(parameters)):
    parameters[i] -= 0.1 * p_gradients[i]

print(parameters[2]); print(parameters[1]); print(parameters[0])
```

This is time-consuming because the predict() function must be run every time each parameter element changes.



3. Error Backpropagation

Part 3: Automatic Differentiation

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

■ Forward derivative trace

$$y = f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$$

Computational graph

$$(x_1, x_2) = (2,\ 5)$$

$$\frac{\partial y}{\partial x_1} = \frac{1}{x_1} + x_2 = 5.5$$ ← Symbolic differentiation

▪ Define

$$\dot{y} = \frac{\partial y}{\partial x_1}$$

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1}$$

▪ **Forward derivative trace**

$$\dot{v}_1 = \frac{\partial v_1}{\partial x_1} = \frac{\partial x_1}{\partial x_1} = 1$$

$$\dot{v}_2 = \frac{\partial v_2}{\partial x_1} = 0$$

$$\dot{v}_3 = \frac{\partial v_3}{\partial x_1} = \frac{\partial}{\partial x_1}\log(v_1) = \frac{\dot{v}_1}{v_1} = \frac{1}{2}$$

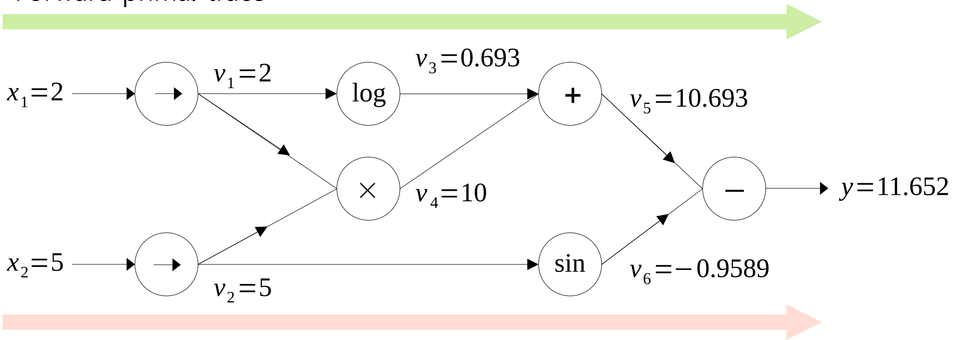
$$\dot{v}_4 = \frac{\partial v_4}{\partial x_1} = \frac{\partial}{\partial x_1}(v_1 v_2) = \dot{v}_1 v_2 + v_1 \dot{v}_2 = 1 \times 5 + 2 \times 0 = 5$$

▪ Source : Baydin, et, al., 2018, Automatic Differentiation in Machine Learning: a Survey (Table 2)

○ : Node → primitive arithmetic operation (+, -, x, /, sin, cos, log, …)

— : Edge → value or data (scalar, vector, matrix … : tensor)

Forward primal trace



Forward derivative trace

$$\dot{v}_5 = \frac{\partial v_5}{\partial x_1} = \frac{\partial}{\partial x_1}(v_3 + v_4) = \dot{v}_3 + \dot{v}_4 = 0.5 + 5 = 5.5$$

$$\dot{v}_6 = \frac{\partial v_6}{\partial x_1} = \frac{\partial}{\partial x_1}\sin(v_2) = \dot{v}_2 \cos(v_2) = 0$$

$$\dot{y} = \frac{\partial y}{\partial x_1} = \frac{\partial}{\partial x_1}(v_5 - v_6) = \dot{v}_5 - \dot{v}_6 = 5.5$$ ← This is the same as the result of symbolic differentiation.

- This method is time consuming because it requires the forward derivative trace for all variables.
- An improvement over this problem is the reverse derivative trace method, which we will look at on the next page.

MX-AI

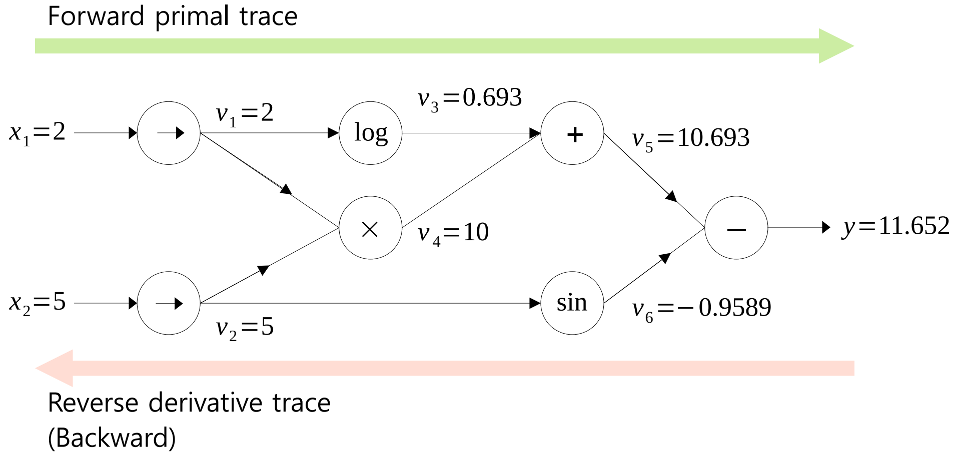
Reverse derivative trace

- AD in the reverse accumulation mode corresponds to a generalized backpropagation algorithm, in that it propagates derivatives backward from a given output. [Baydin, et, al., 2018. 3.2 Reverse Mode.]

$$y = f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$$

$$(x_1, x_2) = (2, 5)$$

Computational graph



- Define

$$\frac{\partial y}{\partial v_i} = \bar{v}_i$$

$$v_1 = x_1$$

$$\bar{v}_1 = \bar{x}_1 = \frac{\partial y}{\partial x_1}$$

- **Reverse derivative trace**

$$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \frac{\partial}{\partial v_5}(v_5 - v_6) = 1$$

$$\bar{v}_6 = \frac{\partial y}{\partial v_6} = \frac{\partial}{\partial v_6}(v_5 - v_6) = -1$$

$$\bar{v}_3 = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_3} = \bar{v}_5 \frac{\partial}{\partial v_3}(v_3 + v_4) = 1$$

$$\bar{v}_4 = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5 \frac{\partial}{\partial v_4}(v_3 + v_4) = 1$$

$$\bar{v}_1 = \bar{v}_3 \frac{\partial}{\partial v_1}\log(v_1) + \bar{v}_4 \frac{\partial}{\partial v_1}(v_1 v_2) = \bar{v}_3 \frac{1}{v_1} + \bar{v}_4 v_2 = 5.5$$

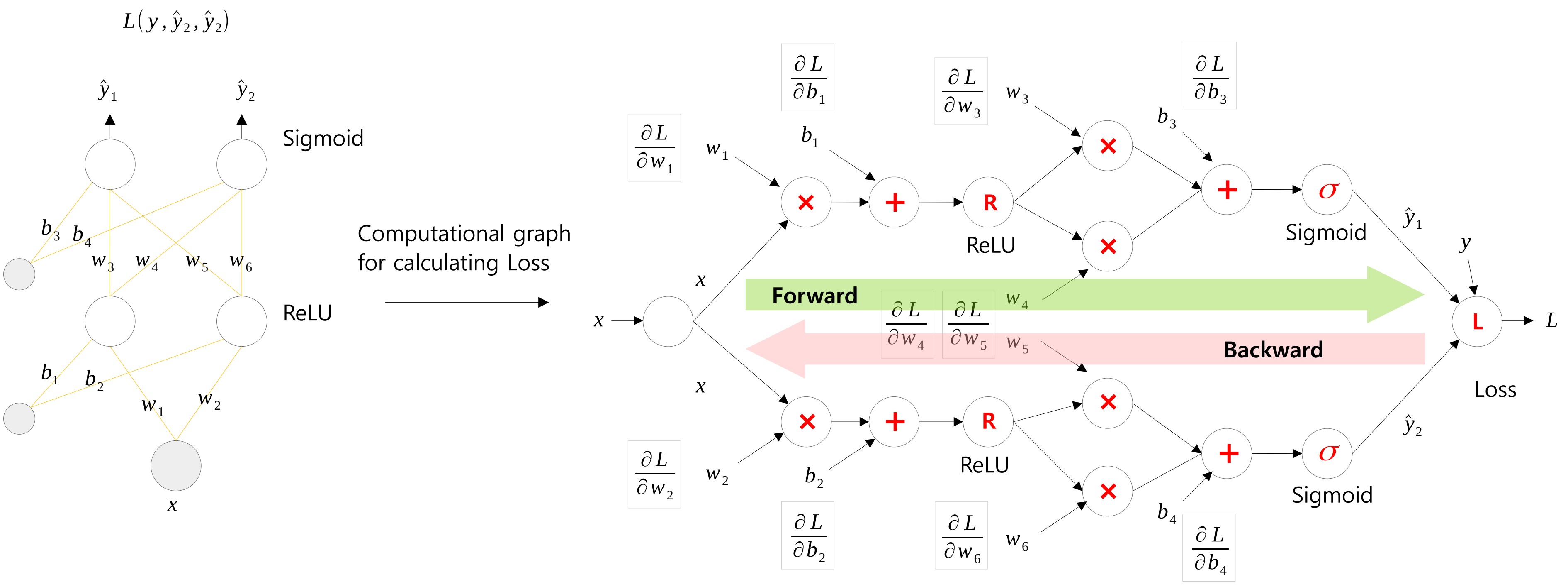
This is the same as the result of forward derivative trace

$$\bar{v}_2 = \bar{v}_4 \frac{\partial}{\partial v_2}(v_1 v_2) + \bar{v}_6 \frac{\partial}{\partial v_2}\sin(v_2) = \bar{v}_4 v_1 + \bar{v}_6 \cos(v_2) = 2 - 0.2837 = 1.7163$$

- This method allows you to find the partial derivatives with respect to x1 and x2 all at once in a single trace. Therefore, this method is faster than the forward derivative trace method.
- This is used for error backpropagation in Tensorflow, PyTorch, etc.

- Source : Baydin, et, al., 2018, Automatic Differentiation in Machine Learning: a Survey (Table 3)

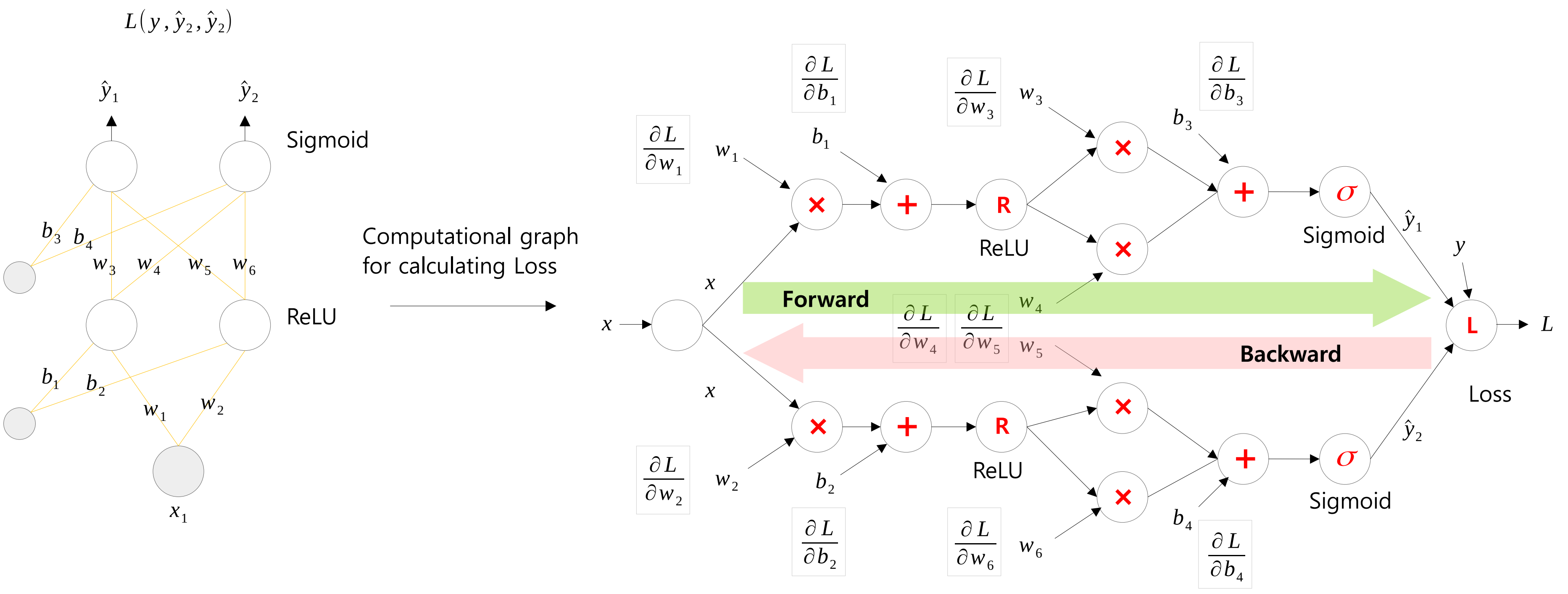
■ Automatic differentiation in neural network



▪ Reference : https://mehta-rohan.com/writings/blog_posts/autodiff.html

MX-AI

■ Automatic differentiation in neural network



▪ Reference : https://mehta-rohan.com/writings/blog_posts/autodiff.html

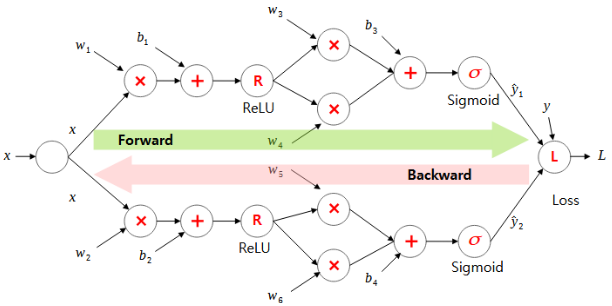
MX-AI

Tensorflow's GradientTape( )

```
import numpy as np
import tensorflow as tf

# Set the parameters as variables in TensorFlow.
x1 = tf.Variable(2.)
x2 = tf.Variable(5.)

# Perform automatic differentiation.
# Forward pass
with tf.GradientTape() as tape:
    y = tf.math.log(x1) + x1 * x2 - tf.math.sin(x2)

# backward pass
dx1, dx2 = tape.gradient(y, [x1, x2])

print('∂y/∂x1 =', dx1.numpy())
print('∂y/∂x2 =', dx2.numpy())

Results:

∂y/∂x1 = 5.5
∂y/∂x2 = 1.7163378
```

$$y = f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$$

$$(x_1, x_2) = (2, 5)$$



https://www.tensorflow.org/guide/autodiff

Introduction to gradients and automatic differentiation

Automatic Differentiation and Gradients

Automatic differentiation is useful for implementing machine learning algorithms such as backpropagation for training neural networks.

In this guide, you will explore ways to compute gradients with TensorFlow, especially in eager execution.

Computing gradients

To differentiate automatically, TensorFlow needs to remember what operations happen in what order during the **forward pass**. Then, during the **backward pass**, TensorFlow traverses this list of operations in reverse order to compute gradients.

Gradient tapes

TensorFlow provides the tf.GradientTape API for automatic differentiation; that is, computing the gradient of a computation with respect to some inputs, usually tf.Variables. TensorFlow "records" relevant operations executed inside the context of a tf.GradientTape onto a "tape". TensorFlow then uses that tape to compute the gradients of a "recorded" computation using reverse mode differentiation.