# DeepLearning Basis 정리

2017272043 이성진

**Logistic Regression** : $\hat{y}$가 0or1로 분류되는 이산 분류 모델에서 사용하는 회귀 개념

**train_sample** :{ $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \dots (x^{(m)}, y^{(m)})$ } → m개의 train_sample, 위첨자 ( )는 sample number 뜻함

Input = **train_set_x** ($x_1^{(1)}, x_2^{(1)} \dots x_n^{(1)}$ : ***n* 차원을 가지는 특징벡터**)

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & \ddots & & \\ \vdots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots \\ x_n^{(1)} & & & \ddots \end{bmatrix}$$

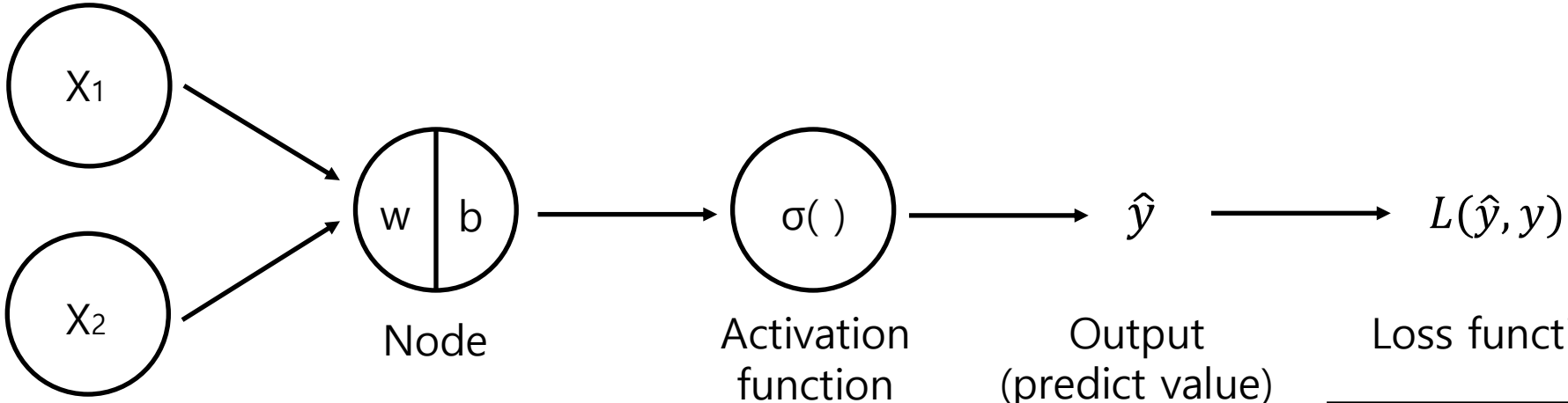
각 train_sample의
feature 개수 (n)
아래첨자 number로 표시

Train_set의
Train_sample 수 (m)

**train_set_y** (0or1의 값을 가지는 true lable)

$$Y = \begin{bmatrix} y^{(1)}, & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

**Logistic Regression Computed Diagram**




Input
(1st train_sample Feature들)

Node

Activation function
(= sigmoid)

Output
(predict value)

Loss function

Perceptron : 신경망의 기본 단위

Input : 입력
Node : 원 (=Neuron)
→ : 신호의 흐름
Activation Function : 활성화 함수
Output : 출력(=prediction value)
Loss function : 손실 함수

**용어 개념 정리**

Input(train_set) : m개의 train sample을 인가하여 model을 학습시킨다

Output(predict_value, $\hat{y}$) : m개의 train sampl에 대해 학습된 model에서 예측한 값 → Y에 가깝게 해야 함

W(weight) : 가중치, 각 node에서 train_sample의 $x_1, x_2 \dots x_n$에 곱해진다 (기울기 개념)

B(bias) : 바이어스, 각 node에서 train_sample의 $x_1, x_2 \dots x_n$에 더해진다 (절편 개념)

Activation function : Logistic Regression는 sigmoid function 사용 → σ(z) : z→-∞ : 0에 수렴, z→∞ : 1에 수렴 (이진분류에 사용)

Loss function : $L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$ → 하나의 전역 최솟값을 가지는 Loss function 정의

참 값 Y와 예측 값 $\hat{Y}$ 사이에 대한 오차 측정 (Error function이라고도 함)

train_sample 각각에 대해 정의되어 예측이 얼마나 잘 되었는지 측정 → 작을수록 예측 잘 된 것 (minimizing한다)

Cost function : $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y) = -\frac{1}{m}\sum_{i=1}^{m}(y \log \hat{y} + (1-y) \log(1-\hat{y}))$

Loss function의 평균

train_set 전체에 대해 예측이 얼마나 잘 되었는지 측정

*Loss function 정의에 대한 타당성 증명*

*증명)*

$L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$

$if)\ y = 1$

$L(\hat{y}, y) = -\log \hat{y} \rightarrow for\ minimizing\ \log \hat{y} \uparrow \rightarrow \hat{y} \uparrow$ ($0 \leq \hat{y} \leq 1$ ∵ sigmoid) $\rightarrow \hat{y} \fallingdotseq 1$

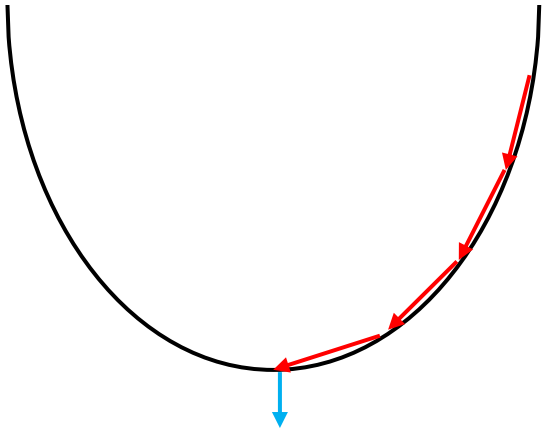
$if)\ y = 0$

$L(\hat{y}, y) = -\log(1-\hat{y}) \rightarrow for\ minimizing\ \log(1-\hat{y}) \uparrow \rightarrow \hat{y} \downarrow$ ($0 \leq \hat{y} \leq 1$ ∵ sigmoid) $\rightarrow \hat{y} \fallingdotseq 0$

*결론)*

∴ 참 값 y와 예측 값 $\hat{y}$ 이 같아진다 + 하나의 전역 최솟값을 가지는 볼록함수이다 = Gradinet descent 적용 가능

**Cost function** : $\mathrm{J(w,b)} = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y) = -\frac{1}{m}\sum_{i=1}^{m}(y \log \hat{y} + (1-y)\log(1-\hat{y}))$ → 최솟값을 가져야 함 (minimizing)




전역 최솟값 또는
그 근사치로 수렴

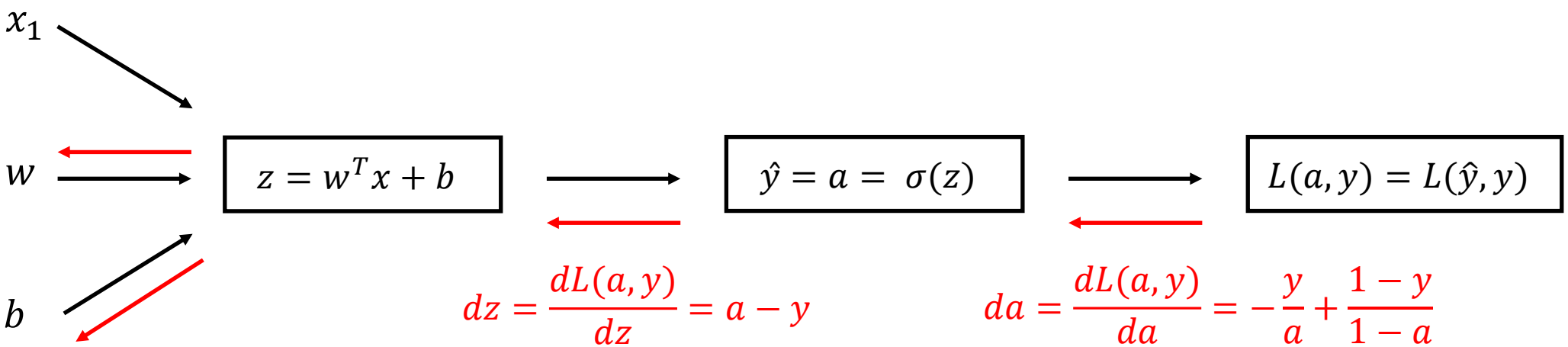
→

Gradient descent

$\mathrm{J(w,b)}$의 임의의 점에서 시작하여 $\mathrm{J(w,b)}$의 전역 최솟값으로
또는 그 근사치로 수렴하게 만드는 방법
Gradient descent가 한번 진행될 때마다 w와 b를 update함
이 때 w와 b의 update 크기를 Learning rate라고 함
Learning rate의 크기가 작을수록 정확해지지만 학습 시간 ↑
Learning rate의 크기가 클수록 학습 시간 ↓하지만 수렴하지 않고 발산가능
적절한 Learning rate를 적용시켜서 model을 학습시켜야 함

$w, b$ update
$w := w - \alpha dw$
$b := b - \alpha db$ ($\alpha$ : learning rate)

**Back Propagation** : 각 단계의 Gradient 계산




$$dw = \frac{\partial L(a,y)}{\partial w} = xdz$$

$$db = \frac{\partial L(a,y)}{\partial b} = dz$$

# Vectorization

Vectorization : Python code에서 for문으로 각 test_sample의 feature 계산하면 러닝 시간 ↑
Vectorization을 이용해서 계산하면 학습속도 향상 → code에서 numpy 사용 및 transpose(전치) 개념 도입

<러닝 시간 비교 Code>

```python
#%% 러닝 시간 측정
import numpy as np
import time

# Vectorized version
a = np.array([1,2,3,4])

a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b) # c = a*b
toc = time.time()

print(c)
print("Vectorized version" + str(1000*(toc-tic))+"ms")

# Non-vectorized version
c = 0
tic = time.time()

for i in range(1000000) :
    c += a[i]*b[i]
toc = time.time()

print(c)
print("for loop :" + str(1000*(toc-tic))+"ms")
```

<속도 비교 결과>

```
In [4]: runcell('러닝 시간 측정', 'C:/Users/nelab_under/.spyder-py3/temp.py')
249995.59803334533
Vectorized version0.9405612945556641ms
249995.59803334234
for loop :532.9718589782715ms
```

for문으로 계산했을 때보다 Vectorization을 도입해서 계산했을 때 러닝 시간이 500배 이상 빨라진 것을 볼 수 있음

→ Data 양이 증가하고 code가 길어지고 epoch가 증가할 수록 Vectorization으로 계산하는 것이 훨씬 빠르고 유리하다

**Broadcasting** : Python에서 자동으로 matrix를 확장시켜서 계산해주는 것

장점 : 언어의 표현성 생성, 언어의 유동성(단순히 한 줄의 코드로도 많은 기능 구현 가능)
단점 : 유동성이 큰 만큼 Broadcasting의 구조와 특성에 미숙하면 감지하기 힘들거나 이상한 버그 생성 가능 → 디버깅 어려워짐

(4x1) matrix에 실수를 더하면 python은 자동으로 실수를 (4x1) matrix form으로 확장시켜서 계산 → 결과값 : (4x1) matrix

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

(mxn) matrix에 (1xn) matrix를 더하면 (1xn) matrix를 m번 반복시켜서 계산 → 결과값 : (mxn) matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

(mxn) matrix에 (mx1) matrix를 더하면 (mx1) matrix를 n번 반복시켜서 계산 → 결과값 : (mxn) matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

## Broadcasting

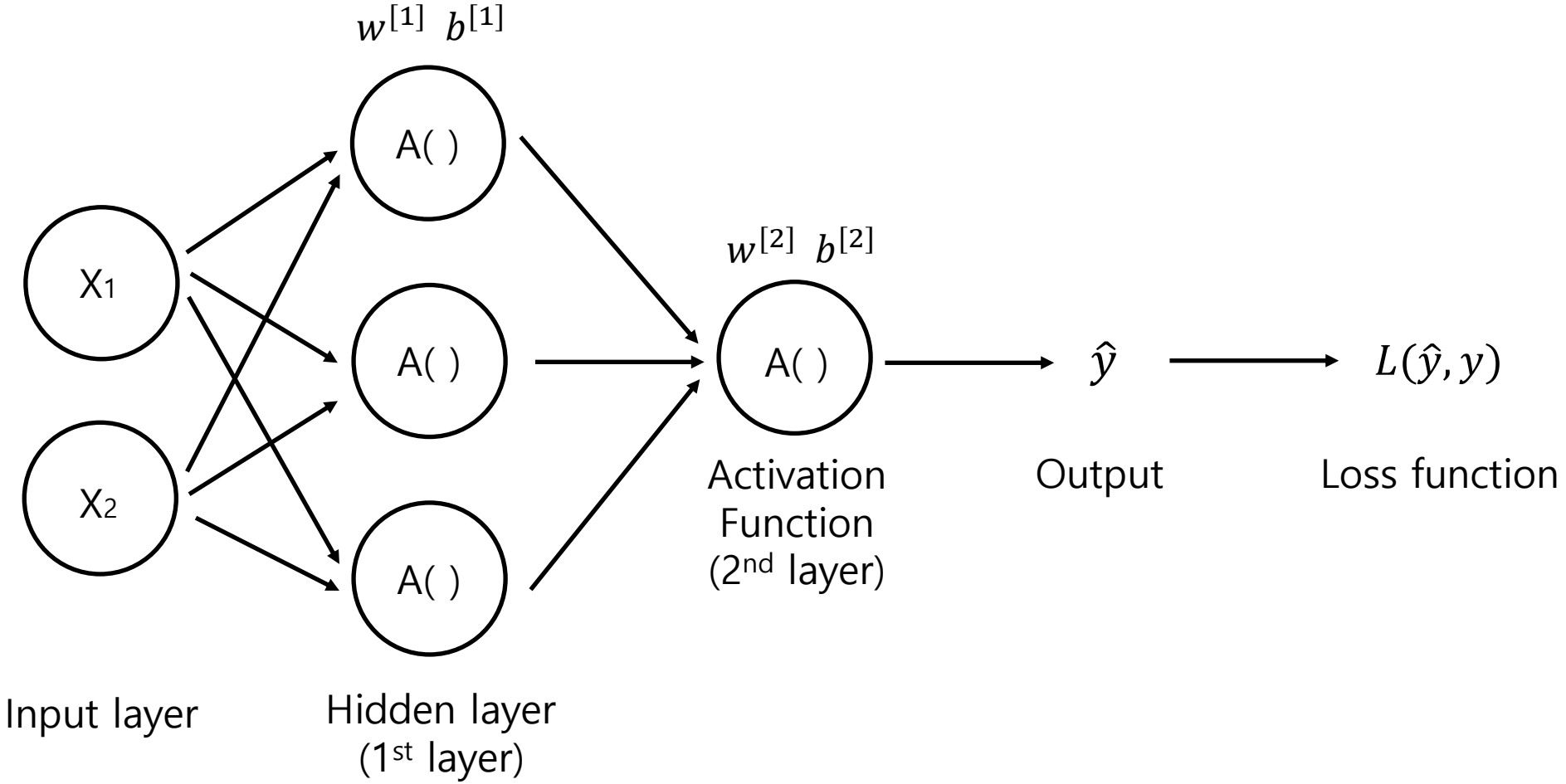
Vector(matrix) vs Rank 1 array

Ex) a = np.random.randn(5) → print(a.shape) → (5, ) → ???
(5, ) : Rank 1 array 데이터 구조 → row vector도 아니고 column vector도 아님
print(a.T) → a의 Transpose는 a와 같음 (전치가 안됨) ∵ Rank 1 array
print(np.dot(a, a.T)) → Broadcasting X, 바깥쪽 matrix가 확장되지 않고 결과값이 단 하나의 값으로 나옴

Sol) a = np.random.randn(5,1) → print(a.shape) → (5,1) → 5x1 구조를 정확히 지정해줌으로써 직관적으로 생성
print(a) : 5x1 matrix
print(a.T) : 1x5 matrix (transpose)
print(np.dot(a, a.T)) → Vector produc에서 Broadcasting에 의해 바깥쪽 a.T가 확장되어 5x5 matrix 결과값 생성

- Rank 1 array를 의도치 않게 생성했을 시 reshape( , )을 통해 원하는 form의 matrix로 지정해준다
- operator '*' 는 element wise product : 요소별 (원소별) 곱셈 → Broadcasting 불가능
- np.dot( , ) 은 Matrix product : 행렬 곱 → Broadcasting 가능

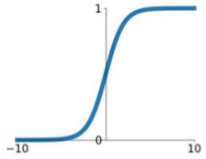
**Multiple Layer** → 관례적으로 Input layer는 count하지 않음, 위첨자 [ ]는 layer number를 뜻함



# Activation Functions

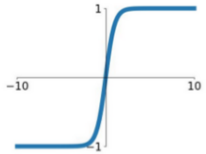
**Sigmoid**

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



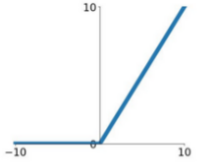
**tanh**

$\tanh(x)$



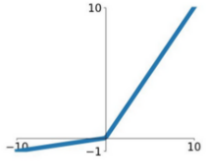
**ReLU**

$\max(0, x)$



**Leaky ReLU**

$\max(0.1x, x)$

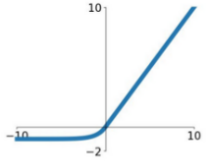


**Maxout**

$\max(w_1^T x + b_1, w_2^T x + b_2)$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid $g(z) = \sigma(z) \rightarrow g(z)' = g(z)\big(1 - g(z)\big)$

Hyperbolic tangent $g(z) = \tanh(z) \rightarrow g(z)' = 1 - g(z)^2$

ReLU $g(z) = \max(0, z) \rightarrow g(z)' = \begin{matrix} 0\ (z < 0) \\ 1\ (z > 0) \end{matrix}$

Leaky ReLU $g(z) = \max(0.1z, z) \rightarrow g(z)' = \begin{matrix} 0.1\ (z < 0) \\ 1\ (z > 0) \end{matrix}$

Linear activation function을 사용하면 신경망은 Input의 선형식만을 출력 → 층이 얼마나 많든, 출력은 은닉층이 얻는 것과 같음 = 은닉층은 쓸모가 없음 (단, 회귀문제에 대한 머신러닝 기법에서는 출력값인 $\hat{y}$ 이 실수일 때 사용)

∴ Non-linear activation function 사용

## Forward Propagation

$$z^{[1]} = w^{[1]}\, x + b$$
$$A^{[1]} = g^{[1]}\,(z^{[1]})$$
$$z^{[2]} = w^{[2]}\, A^{[1]} + b$$
$$A^{[2]} = g^{[2]}\,(z^{[2]})$$

## Back Propagation

$$dz^{[1]} = w^{[2]T}\, dz^{[2]} * g^{[1]\prime} z^{[1]}$$
$$dw^{[1]} = \frac{1}{m} dz^{[1]}\, X^{T}$$
$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True})$$
$$dz^{[2]} = A^{[2]} - Y$$
$$dw^{[2]} = \frac{1}{m} dz^{[2]}\, A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

## Dimension

$$z^{[1]} = (n^{[1]}, 1)$$
$$z^{[2]} = (n^{[2]}, 1)$$
$$w^{[2]} = (n^{[1]}, n^{[x]})$$
$$w^{[2]} = (n^{[2]}, n^{[1]})$$
$$dz^{[1]} = w^{[2]T}\, dz^{[2]} * g^{[1]\prime} z^{[1]}$$
$$= (n^{[1]}, n^{[2]})\text{x}(n^{[2]}, 1) * (n^{[1]}, 1)$$
$$= (n^{[1]}, 1)$$

**Initialization**

$w^{[1]} = np.random.randn\left(n^{[1]}, n^{[x]}\right) * 0.01$ → Gaussian random variable에 0.01을 곱해서 매우 작은 값으로 만든다
(w가 커지면 z가 매우 커지거나 작아지므로 gradient 값에 문제 발생)
(ex. Sigmoid or tanh에서 gradient = 0 → gradient descent 속도 ↓)
(따라서 Learning 속도 ↓ 문제 발생)

Zero Initializing하지 않는 이유?
→ w를 Zero Initializing 하면 epoch를 몇으로 하든지, 은닉층이 몇 개인지 상관없이 Symmetric proble이 발생함
Symmetric problem을 수학적 귀납법으로 계산하면 각 training마다 hidden unit이 항상 같은 함수를 계산

$b^{[1]} = np.zeros\left(n^{[1]}, 1\right)$ → b는 zero Initializing해도 Symmetric problem 발생 X = Symmetric braking problem