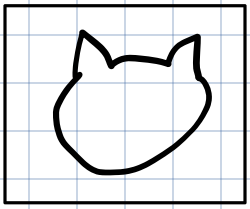
Week 2. Neural Network Basics

Essential Techniques for Neural Network
① Do not explicit for-loops to loop over your entire training set.
② Computation of neural network can be organized as Forward & backward propagation

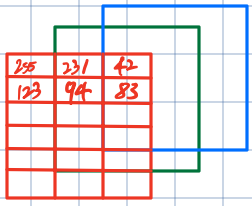
1. Logistic Regression as a Neural Network

- Binary Classification



label
y { 1 (cat)
{ 0 (noncat)

- how an image is represented in computer
  ⇒ computer stores image as 3 separate matrices (red, green, blue)
  ∴ if image is 64 X 64 px ⇒ 64×64 matrix X3



input image

unroll all the pixel values

$x =$ [255, 231, ⋮, 202, 255, 134, ⋮, 94, 255, 134, ⋮, 142]

feature vector

(dimension = 64×64×3
$n = n_x = 12288$)

* Notation
(x, y) : a single training sample
x : feature vector, $x \in \mathbb{R}^{n_x}$
y : label, $y \in \{0, 1\}$

training set : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \cdots (x^{(m)}, y^{(m)})\}$
m training samples

stack training inputs $x_{(1)}, x_{(2)} \cdots$

in columns

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad n_x \text{ (height of feature vector)}, \quad m \text{ (\# of training samples)}$$

$m \times n_x$

$$Y = [y^{(1)}, y^{(2)}, y^{(3)} \cdots, y^{(m)}]$$

$$Y = \mathbb{R}^{1 \times m} \Rightarrow Y.shape = (1, m)$$

$$\Rightarrow X.shape = (n_x, m)$$

• Logistic Regression

- Algorithm for **binary classification**

- Given x, $x \in \mathbb{R}^{n_x}$ e.g. cat vs noncat picture
- Want $\hat{y} = P(y=1|x) \approx y$ e.g. probability of given image is a cat picture
- Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$ (real number)

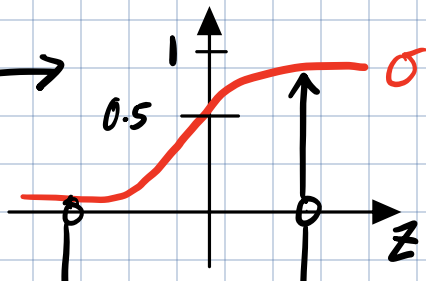
⇒ Given an input x and the parameters w and b,
how do we generate output $\hat{y}$?

① linear function of the input x: $\hat{y} = w^T x + b$

→ But, $\hat{y}$ should be $0 \le \hat{y} \le 1$

② Apply **sigmoid function**!: $\hat{y} = \sigma(w^T x + b)$

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad (z \in \mathbb{R})$$



If z is large: $\sigma(z) \approx \frac{1}{1+0} = 1$

If z is small (large negative): $\sigma(z) = \frac{1}{1+\text{big num}} \approx 0$

∴ Goal: Learn parameters w & b → $\hat{y}$ becomes a good estimate

of the chance of $y=1$

* Notation (Alternative notation)

$x_0 = 1,\ x \in \mathbb{R}^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$, $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \\ \\ \\ \\ \} w \end{matrix}$

how to know best parameters w & b?

• Logistic Regression

– $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$ ($z^{(i)} = w^T x^{(i)} + b$)

Given $\{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

* $A^{(i)}$: i-th sample

– Loss (error) function: measure how well algorithm is doing on a single training sample

↳ cost of a single training sample

= how good output $\hat{y}$ is close to true label y

= minimize the loss (error) function!

① Squared Error: $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

⇒ it's non-convex: have many local optima so the algorithm cannot find global optimum

⇒ it makes gradient descent not work well

② $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$

intuitive look

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

  Want large $\log \hat{y}$ ⇒ want large $\hat{y}$ ⇒ $\hat{y} \approx 1$ ($0 \le \hat{y} \le 1$)

- $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$

Want large $\log(1-\hat{y})$ ⇒ want small $\hat{y}$ ⇒ $\hat{y} \approx 0 \ (0 \le \hat{y} \le 1)$

- Cost function : measure how well algorithm is doing on entire training set
  ↳ cost of parameters
  = average of loss function applied to each of m training samples

⇒ find best parameters w & b that minimizes cost function J.

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

- Logistic Regression cost function
  - Why do we use $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$ as a cost function?

Given x, $x \in \mathbb{R}^{n_x}$
Want $\hat{y} = P(y=1|x) \approx y$ ⇒ $\hat{y}$ = y가 1일 확률
Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$ (real number)

binary classification

If $y=1$: $P(y|x) = \hat{y}$
If $y=0$: $P(y|x) = 1-\hat{y}$
} $P(y|x)$

⇒ Let's summarize these 2 equations into a single equation.

$$P(y|x) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$$

↳ how? because this equation returns exactly what we wanted!

If $y=1$: $P(y|x) = \hat{y}$
If $y=0$: $P(y|x) = (1-\hat{y})$

⇒ What about log?

∵ log function is strictly monotonically increasing function,

Maximizing $\log p(y|x)$ gives you similar result of maximizing $p(y|x)$

$$\log P(y|x) = y \log \hat{y} - (1-y) \log(1-\hat{y}) = -L(\hat{y}, y)$$

↳ increasing probability ↑ in other algorithms, but logistic algorithm wants to minimize the loss function

minimize loss function = maximize log of probability

- Overall cost function on the m training samples

$$P(\text{labels in training set}) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$$

↳ samples are iid

⇒ to carry out maximum likelihood estimation, find out parameter that maximizes chance of observations of training set

$$\log(P(\text{labels in training set})) = \log\left(\prod_{i=1}^{m} P(y^{(i)}|x^{(i)})\right)$$

$$= \sum_{i=1}^{m} \log(P(y^{(i)}|x^{(i)}))$$

$$\log(P(y^{(i)}|x^{(i)})) = -L(\hat{y}^{(i)}, y^{(i)})$$

$$= -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

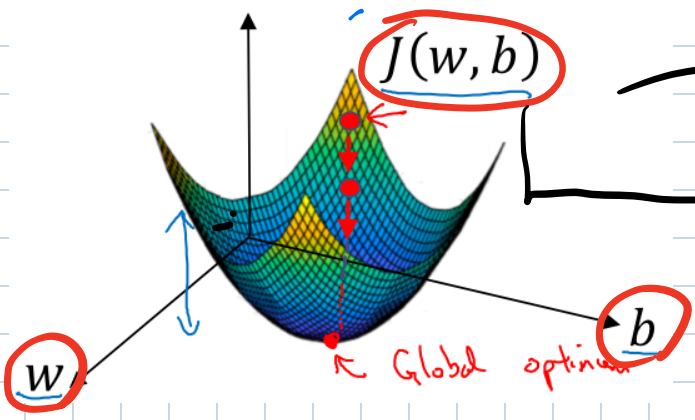
We want to minimize the cost, as maximize the likelihood ⇒ removed (-) sign) and $\frac{1}{m}$ for better scale

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

⇒ By minimizing $J(w,b)$, carry out maximum likelihood estimation over the regression model (under assumption training samples are iid)

• Gradient descent

- how to train (learn) w and b on the training set?



→ Illustration of Gradient descent

→ convex function

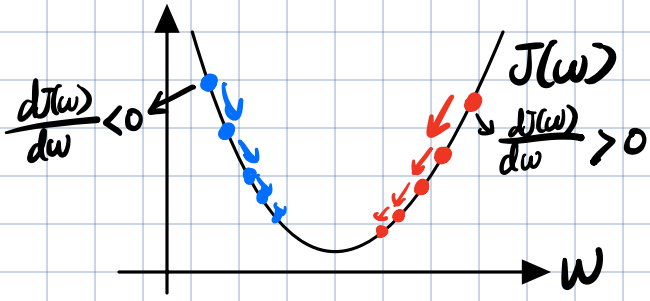
Goal: Find the value of w & b that minimize $J(w,b)$

① Set initial value to $(0,0)$ ⇒ 보통은 random value 한다.

② Take a step to the steepest direction

③ Iterate step ② until it converges to the global optimum

- Let's simplify $J(w,b) \rightarrow J(w)$ (3D → 2D)



∴ Repeat {

$w := w - \alpha \frac{dJ(w)}{dw}$ ← `dw in python

}

⇒ In cost function...

$J(w,b)$

$w := w - \alpha \frac{dJ(w,b)}{dw}$

$b := b - \alpha \frac{dJ(w,b)}{db}$

Implementation

$J(w,b)$

$w := w - \alpha dw$

$b := b - \alpha db$

* Notation

:= : update

α : learning rate (step size)

$\frac{dJ(w)}{dw}$: derivative = slope of the function, used for update of the change of w

$\partial(d)$: used for derivative of function with 2 or more variables

• Computation Graph

- Computation of neural network [forward propagation → Computes output.
  back propagation → Compute gradients (derivatives)

- Try compute

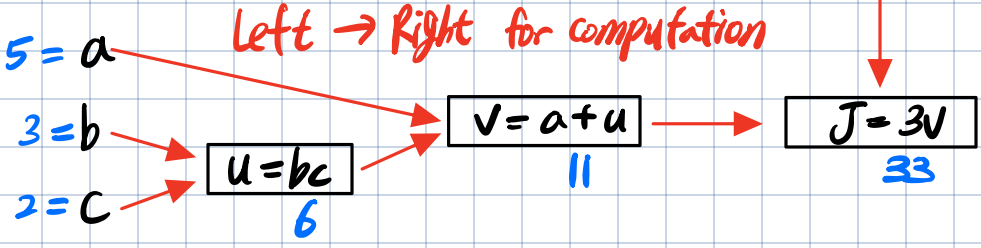
$J(a,b,c) = 3(a+bc)$ ⇒ In LR, J is cost function to minimize

① Compute u, $u = bc$

② Compute v, $v = a+u$

③ compute J, $J = 3v$

Computation graph
⇒ distinguish specific output that you want to optimize



• Derivatives with a computation graph

- Let's compute derivative of J with respect to V

⇒ How would the value of J change if we change the value of V?

$b = 3 \rightarrow 3.001$

$u = b \cdot c = 6 \rightarrow 6.002$ (×2)

$\therefore \frac{dJ}{db} = \frac{dJ}{du}\cdot\frac{du}{db} = 6$ (3, 2)

J = 3V

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$ (×3)

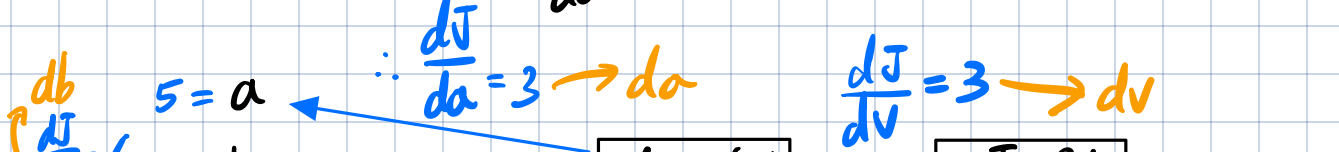
$\therefore \frac{dJ}{da} = 3$

J = 3V

$V = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$ (×3)

$\therefore \frac{dJ}{dv} = 3$



$\frac{dJ}{db}=6$ $3=b$ $\frac{dJ}{dc}=9$ $2=c$ dc

$u=bc$ (6) $\rightarrow$ $v=a+u$ (11) $\rightarrow$ $J=3v$ (33)

$\frac{dJ}{du}=3 \rightarrow du$

$\frac{dJ}{dc}=\frac{dJ}{du}\cdot\frac{du}{dc}=9$ ($\frac{dJ}{du}$: 3, $\frac{du}{dc}$: 3)

$\therefore \frac{du}{db}=2$

$u=6 \rightarrow 6.001$

$v=11 \rightarrow 11.001$ (x1)

$J=33 \rightarrow 33.003$ (x3)

$\therefore \frac{dJ}{du}=3=\frac{dJ}{dv}\cdot\frac{dv}{du}$ ($\frac{dJ}{dv}$: 3, $\frac{dv}{du}$: 1)

- Chain rule

⇒ if you change a → change v → change J

a를 변화했을 때 J의 변화량 = a를 변화시켰을 때 v의 변화량 × v를 변화시켰을 때 J의 변화량

$$\frac{dJ}{da}=\frac{dJ}{dv}\times\frac{dv}{da}$$

* Notation

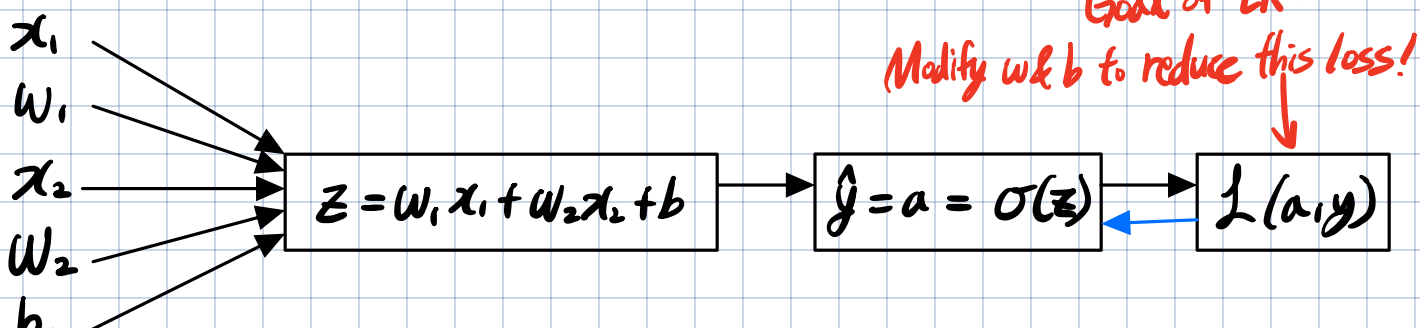
$\frac{d\text{final output Var}}{d\text{var}}$ ⇒ dvar : derivative of the final output with respect to intermediate quantities.

• Logistic Regression Gradient Descent

- Logistic Regression Recap

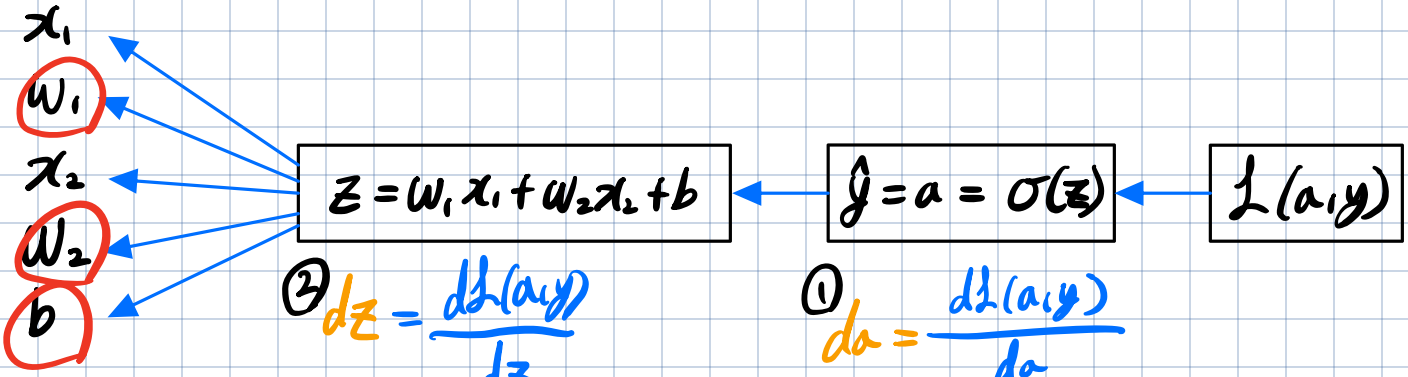
$$z=w^Tx+b$$
$$\hat{y}=a=\sigma(z)$$
$$\mathcal{L}(a,y)=-(y\log(a)+(1-y)\log(1-a))$$

- Computation graph of LR (with 2 features)



- Backpropagation of LR

⇒ Compute derivative of $\mathcal{L}(a,y)$



② $dz = \frac{d\mathcal{L}(a,y)}{dz}$

$= \frac{d\mathcal{L}(a,y)}{da} \cdot \frac{da}{dz}$

$= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \cdot a(1-a)$

$= a - y$

① $da = \frac{d\mathcal{L}(a,y)}{da}$

$= -\frac{y}{a} + \frac{1-y}{1-a}$

*$y = a \log x \rightarrow y' = \frac{a}{x}$

③ How much do we need to change w and b?

$\frac{\partial \mathcal{L}}{\partial w_1} = dw_1 = x_1 dz$

$\frac{\partial \mathcal{L}}{\partial w_2} = dw_2 = x_2 dz$

$db = dz$

- Perform gradient descent to a single training sample...

① Compute $dz \rightarrow dw_1, dw_2, db$

② Update the values

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

⇒ What about m training samples?

• Gradient descent on m example

- Cost function

$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y)$ → average of individual losses

$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$

$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$ → average of individual derivatives

↳ overall gradient

$dw_1^{(i)} - (x^{(i)}, y^{(i)})$

- Let's implement algorithm

① Initialization

$J=0,\ dw_1=0,\ dw_2=0,\ db=0$ → accumulative var ∴ no (i)

$dw_1 = \frac{\partial J}{\partial w_1}$

② Calculate derivatives

For $i=1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} - (1-y^{(i)}) \log(1-a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$ ← 계산할 때 i-th training sample

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$] 2 features

$db \mathrel{+}= dz^{(i)}$

$J /= m$

$dw_1 /= m;\ dw_2 /= m,\ db /= m$

③ Implement 1 step of gradient descent

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

④ Repeat multiple steps of gradient descent

- Weakness of this implement: 2 for loops over m training samples & n features

⇒ efficiency ↓

⇒ Use vectorization for deep learning with large data set.

• Vectorization

- to avoid explicit for loops
- What is vectorization?

$z = w^T x + b \Rightarrow w = \begin{bmatrix} \vdots \end{bmatrix}, w \in \mathbb{R}^{n_x} \quad x = \begin{bmatrix} \vdots \end{bmatrix}, x \in \mathbb{R}^{n_x}$ (features)

Non-vectorized

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

⇒ Slow

Vectorized

```
z = np.dot(w, x) + b
```

($w^T x$)

⇒ Just compute $w^T x$ directly (Fast)

↳ Single Instruction Multiple Data

- CPU & GPU have parallelized command (SIMD)

⇒ built-in function like np.dot can compute much faster

• More Vectorization Examples

- Whenever possible, avoid explicit for-loops.

e.g. $u = Av$ (vector = matrix × vector) — matrix multiply

$u_i = \sum_j A_{ij} v_j$

Non-vectorized

```
u = np.zeros((n,1))
for i ...
    for j ...
        u[i] += A[i][j] * v[j]
```

(2 for-loops)

Vectorized

```
u = np.dot(A, v)
```

- Say you need to apply the exponential operation on every element of a matrix/vector

$$v=\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \quad u=\begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

Non-vectorized

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

Vectorized

```
u = np.exp(v)
```

- Let's apply to gradient descent to remove a for-loop.

⇒ Remove one for-loop (②)

<Non-vectorized>

$J=0,\ dw_1=0,\ dw_2=0,\ db=0$

① For $i=1$ to m

$z^{(i)}=w^T x^{(i)}+b$

$a^{(i)}=\sigma(z^{(i)})$

$J\ +=-[y^{(i)}\log a^{(i)}-(1-y^{(i)})\log(1-a^{(i)})]$

$dz^{(i)}=a^{(i)}-y^{(i)}$

② $dw_1\ +=x_1^{(i)}dz^{(i)}$

$dw_2\ +=x_2^{(i)}dz^{(i)}$

① $db\ +=dz^{(i)}$

$J/=m,\ dw_1/=m,\ dw_2/=m,\ db/=m$

② for $j=1\sim n_x$ $dw_j\ +=\cdots$

2 features

feature 수만큼 for loop 필요 ($dw_1, dw_2\cdots$)

<Vectorized>

$dw=np.zeros((n_x,1))$

$dw\ +=x^{(i)}dz^{(i)}$

dw/m

• Vectorizing Logistic Regression

- Forward propagation

$$z^{(1)}=w^T x^{(1)}+b \quad a^{(1)}=\sigma(z^{(1)})$$

$$z^{(2)}=w^T x^{(2)}+b \quad a^{(2)}=\sigma(z^{(2)})$$

$$z^{(3)}=w^T x^{(3)}+b \quad a^{(3)}=\sigma(z^{(3)})$$

$$\begin{bmatrix} z^{(1)} & z^{(2)} & z^{(3)} \end{bmatrix}$$

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$, $X \in \mathbb{R}^{n_x}$

← Stack feature vector x horizontally

⇒ how to compute $z^{(1)}, z^{(2)}, z^{(3)} \cdots$ all in one-line code?

make a row vector...

$[z^{(1)} z^{(2)} \cdots z^{(m)}] = w^T X + [b\ b \cdots]$ ⇒ $w^T \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$

(row vector; $(1,m)$; row vector)

$= [w^T x^{(1)} + b \cdots w^T x^{(m)} + b]$

($(1,m)$ vector)

⇒ Just as X was obtained by stacking training samples horizontally,

Define Z as...

$Z = [z^{(1)} z^{(2)} \cdots z^{(m)}] = w^T X + [b\ b \cdots b]$

$(1,m)$

$= np.dot(w^T, X) + b$

* broadcasting

- $b \in \mathbb{R} \Rightarrow [b, b, b \cdots b]$: python automatically change real number b into a vector/matrix when real num + vector/matrix

⇒ how to compute $a^{(1)}, a^{(2)}, a^{(3)} \cdots a^{(m)}$ all in one-line code?

stack $a^{(i)}$ horizontally and define as A

$$A = [a^{(1)}, a^{(2)} \cdots a^{(m)}] = \sigma(Z)$$

- **Vectorizing Logistic Regression's Gradient Computation**

  - Back propagation

    $dz^{(1)} = a^{(1)} - y^{(1)}$ $\quad dz^{(2)} = a^{(2)} - y^{(2)}$

    $dZ = [dz^{(1)}\ dz^{(2)} \cdots dz^{(m)}] = [a^{(1)} - y^{(1)} \cdots a^{(m)} - y^{(m)}] = A - Y$ $(1 \times m)$

    $A = [a^{(1)} \cdots a^{(m)}]$

    $Y = [y^{(1)} \cdots y^{(m)}]$

  - Vectorizing db

$db = 0$
$db \mathrel{+}= dz^{(1)}$
$db \mathrel{+}= dz^{(2)}$
$\vdots$
$db \mathrel{+}= dz^{(m)}$
$db \mathrel{/}= m$

$$\Rightarrow db = \frac{1}{m}\sum_{i=1}^{m} \underbrace{dz^{(i)}}_{\text{row vector}} = \frac{1}{m}\,\text{np.sum}(dz)$$

- Vectorizing dw

$dw = 0$
$dw \mathrel{+}= x^{(1)} dz^{(1)}$
$dw \mathrel{+}= x^{(2)} dz^{(2)}$
$\vdots$
$dw \mathrel{+}= x^{(m)} dz^{(m)}$
$dw \mathrel{/}= m$

$$\Rightarrow dw = \frac{1}{m}\begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & & | \end{bmatrix}\begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

$$= \frac{1}{m}\left[x^{(1)}dz^{(1)} + \cdots + x^{(m)}dz^{(m)}\right] \quad (n_x, 1)$$

$$= \frac{1}{m} X dz^{T}$$

<Non-vectorized>

$J=0,\ dw_1=0,\ dw_2=0,\ db=0$
For $i=1$ to m
$z^{(i)} = w^T x^{(i)} + b$
$a^{(i)} = \sigma(z^{(i)})$
$J \mathrel{+}= -\left[y^{(i)} \log a^{(i)} - (1-y^{(i)}) \log(1-a^{(i)})\right]$
$dz^{(i)} = a^{(i)} - y^{(i)}$
$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$
$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$
$db \mathrel{+}= dz^{(i)}$
$J \mathrel{/}= m,\ dw_1 \mathrel{/}= m,\ dw_2 \mathrel{/}= m,\ db \mathrel{/}= m$

<Vectorized>

$J=0,\ db=0$
$dw = \text{np.zeros}((n_x, 1))$
$Z = \text{np.dot}(w.T, X) + b$
$dz = A - Y$
$dw = \frac{1}{m} X dz^T$
$db = \frac{1}{m}\,\text{np.sum}(dz)$
$w := w - \alpha\, dw$
$b := b - \alpha\, db$

→ a single step of gradient descent

⇒ you have to use for loop to specify # of iteration

• Broadcasting in Python

- example: Calculate % of colories from carb, protein, and fat from each of the food without for-loop.

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

$= A(3,4)$

① Sum down the columns

```
cal = A.sum(axis=0)  (1,4)
```

* np axis
0 : column (세로)
1 : row (가로)

② Divide each of the 4 columns by their corresponding sum

```
percentage = 100*A/cal.reshape((1,4))
```

→ 행렬의 차원이 확실하지 않을 때

- how can you divide 3x4 matrix by 1x4 matrix?
  ⇒ python automatically expands 1x4 to 3x4

e.g.

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

$$\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix} \rightarrow = \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100&100&100\\200&200&200\end{bmatrix} = \begin{bmatrix}101&102&103\\104&105&106\end{bmatrix}$$

- General Principle of broadcasting

(m, n) + - * / (1, n) or (m, 1)
matrix
↓ (m, n) ↓ (m, n) ⇒ element-wise calculation

(m, 1) + - * / Real# ⇒ element-wise calculation

(1,n)

↓ copying n times

(m,1) of Real num
(1,n)

- A note on python/numpy vectors
  - When construct vectors never use rank 1 array!

```
a = np.random.randn(5)
print(a.shape) ⇒ (5,)
```

→ rank 1 array (neither row/column vector)

⇒ a.T does not change the shape

⇒ np.dot(a,a) returns a real number

  - Proper way to construct vectors

```
a = np.random.randn(5,1)
a.shape = (5,1)
```

→ 5x1 matrix (column vector)

⇒ a.T returns 1x5 matrix

⇒ np.dot(a,a) returns outer product (matrix)

  - Tip:
    - Use assert to ensure dimension of vectors

```
Assert(a.shape == (5,1))
```

    - when you got rank 1 array, Use reshape

```
a = a.reshape((5,1))
```