

# Deep Learning with Python

Jongwon Seok  
Changwon National University





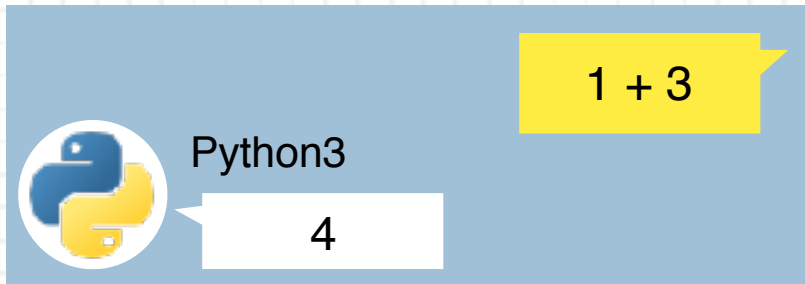
# 01.

## Python 기초

What is the relative number of presentations that will target the audience as a logical Expression unnecessary information is released or increased alignment will fail.

# Python 기초

- 인간과 대화를 하듯 동적으로 반응하는 언어 파이썬



컴파일러 대신 인터프리터가  
코드를 한 줄 씩 번역하는 방식

- 정적 언어 (static language)와 동적 언어 (dynamic language)

## 정적 언어

```
int i = 50;  
float j = 5.0f;
```

변수 선언시 변수의 자료형을 선택,  $i + j$   
? 형변환을 해줘야 한다



## 동적 언어

```
i = 50  
j = 5.0  
c = 'a'
```

자동으로 인터프리터가 자료형을 잡아주  
고,  $i + j$  계산 시 자동으로  
자료형을 변환한다.

# Python 기초 : 산술연산

- 산술 연산자 :  
덧셈, 뺄셈, 곱셈, 나눗셈 등의 계산을 수행하는 연산자

연산자	설명	예시	결과
+	더하기	$5 + 3$	8
-	빼기	$5 - 3$	2
*	곱하기	$5 * 3$	15
/	나누기 (부동소수점)	$5 / 3$	1.6667
//	나누기 (정수)	$5 // 3$	1
%	나눈 나머지	$5 \% 3$	2
**	거듭제곱	$5 ** 3$	125

# Python 기초 : 자료형

- 데이터 타입 :  
여러 종류의 데이터를 식별하는 분류

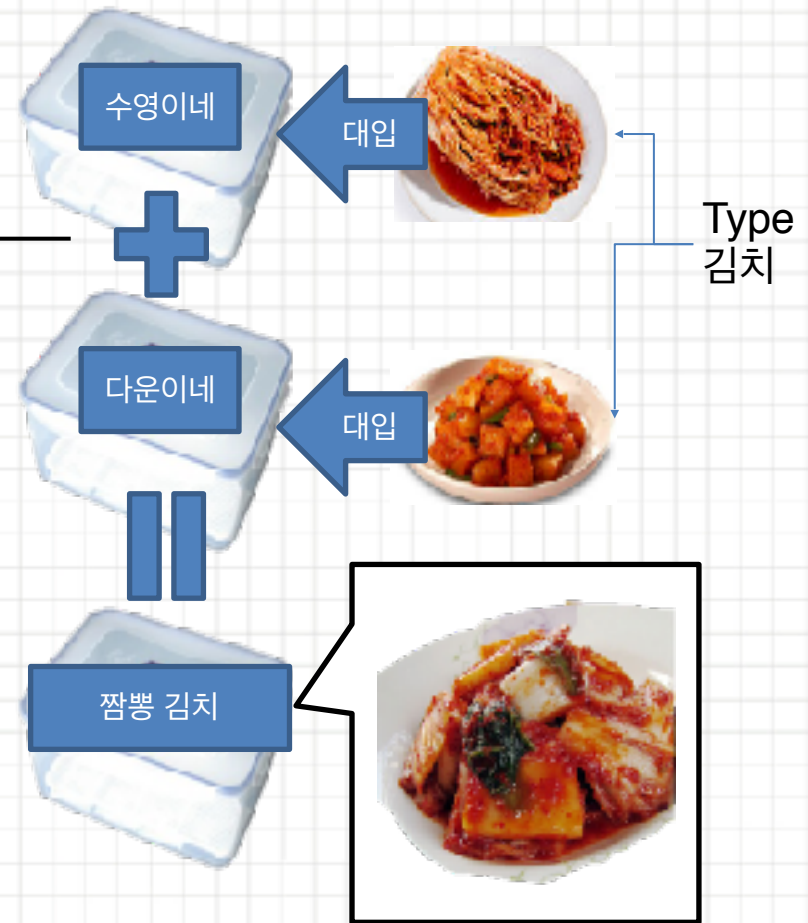
타입	설명
참/거짓 (Boolean)	True 혹은 False
정수 (Integer)	1, 900927 같은 숫자
부동소수점 (Float)	0.5 같이 소수점이 있는 숫자
문자열 (String)	텍스트 문자

# Python 기초 : 변수

- 파이썬 변수 :  
가변하는 데이터가 포함된 객체의 참조

```
>>> x = 10
>>> print(x)
10
>>> x = 100
>>> print(x)
100
>>> y = 3.14
>>> x * y
314.0
>>> type(x * y)
<class 'float'>
```

※파이썬은 동적 언어로  
변수의 자료형 지정없이 자동으로 결정된다



# Python 기초 : 변수

- 파이썬 변수 :  
가변하는 데이터가 포함된 객체의 참조
- 파이썬 변수명으로 사용 가능한 문자
  - : 소문자 (a~z)
  - : 대문자 (A~Z)
  - : 숫자 (0~9)
  - : 언더스코어 (\_)
- 사용 불가능한 변수명  
(숫자로 시작할 수 없고, 파이썬 예약어를 변수로 사용할 수 없다)

: 1  
: 2a  
: 3\_

← 숫자로 시작

→ 파이썬 예약어

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

# Python 기초 : 리스트

- **리스트 :**

여러 데이터를 순차적으로 파악하는데 유용한 시퀀스

---

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> print(a)
```

```
[1, 2, 3, 4, 5]
```

```
>>> len(a)
```

```
5
```

```
>>> a[1]
```

```
2
```

```
>>> a[0 : 2]
```

```
[1, 2]
```

```
>>> str = 'abcde'
```

```
>>> str[0]
```

```
'a'
```

```
>>> str[-3:-1]
```

```
'cd'
```

리스트 인덱싱

0	1	2	3	4
a	b	c	d	e
-5	-4	-3	-2	-1



# Python 기초 : 딕셔너리

- 딕셔너리 :

순서와 오프셋 없이 고유 키와 해당 키의 값으로 데이터를 파악하는데 유리한 시퀀스

---

```
>>> me = {'height' : 180}
>>> me['height']
180
>>> me['weight'] = 70
>>> print(me)
{'weight' : 70, 'height' : 180}
>>> me.keys()
dict_keys(['weight', 'height'])
>>> me.values()
dict_values([180, 70])
>>> num = [1, 2, 3, 4, 5]
>>> str = ['a', 'b', 'c', 'd', 'e']
>>> dict = {'Num' : num, 'Str' : str}
>>> dict
{'Num': [1, 2, 3, 4, 5], 'Str': ['a', 'b', 'c', 'd', 'e']}
```

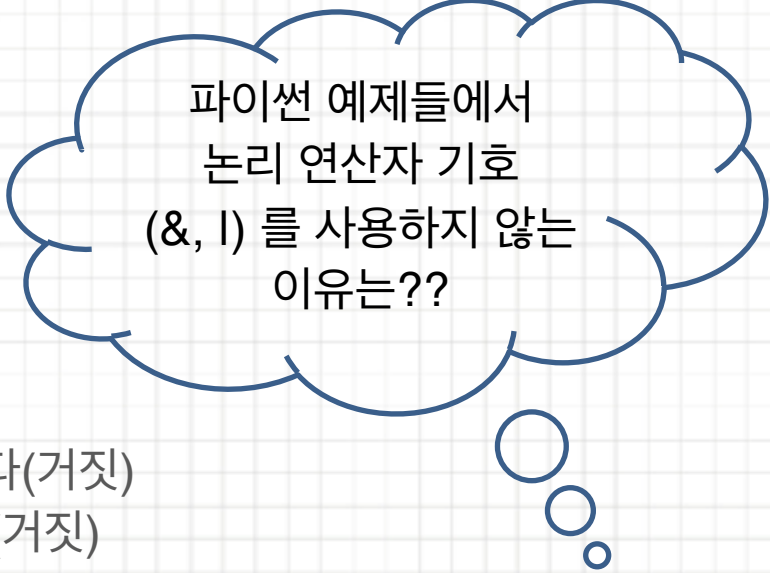
# Python 기초 : bool

- **bool :**

True(참) / False(거짓) 을 구분하는 자료형, and / or / not 연산자와 사용 가능

---

```
>>> hungry = True    #배 고프다(참)
>>> sleepy = False   #졸리다 (거짓)
>>> type(hungry)
bool
>>> not hungry
>>> hungry != hungry
False
>>> hungry and sleepy #배 고프다(참) 그리고 졸리다(거짓)
>>> hungry & sleepy   #배 고프다(참) 그리고 졸리다(거짓)
False
>>> hungry or sleepy #배 고프다(참) 또는 졸리다(거짓)
>>> hungry | sleepy   #배 고프다(참) 또는 졸리다(거짓)
True
```



파이썬 예제들에서  
논리 연산자 기호  
(&, |) 를 사용하지 않는  
이유는??

# Python 기초 : bool

- **bool :**

True(참) / False(거짓) 을 구분하는 자료형, and / or / not 연산자와 사용 가능

## OR 진리표

True	or	True	True
True	or	False	True
False	or	True	True
False	or	False	False

## NOT 진리표

not	True	False
not	False	True

## AND 진리표

True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

# Python 기초 : if 문

- **if 와 else :**  
조건이 참(True)인지 거짓(False)인지 확인하는 선언문
- 

```
>>> hungry = True
>>> if hungry:
>>>     print("I'm hungry")
I'm hungry
>>> hungry = False
>>> if hungry:
>>>     print("I'm hungry")
>>> else:
>>>     print("I'm not hungry")
>>>     print("I'm not sleepy")
I'm not hungry
I'm not sleepy
```

# Python 기초 : for 문

- **for문 :**  
반복(루프) 처리에 사용하는 문법
- 

```
>>> for i in [1, 2, 3]:  
    print(i)
```

1

2

3     #리스트 안의 원소를 하나씩 출력함

# Python 기초 : 함수

- 함수 :  
특정 기능을 수행하는 일련의 명령들을 묶어 정의 가능
- 

```
>>> def hello():  
    print("Hello World!")
```

```
>>> hello()  
Hello World!
```

```
>>> def hello(object)  
    print("Hello " + object + "!")  
>>> hello("Modu_Lab")  
Hello Modu_Lab!
```

# Python 기초 : 클래스

- **class :**  
개발자의 독자적인 자료형과 전용 함수 정의 가능
- 

class 클래스 이름:

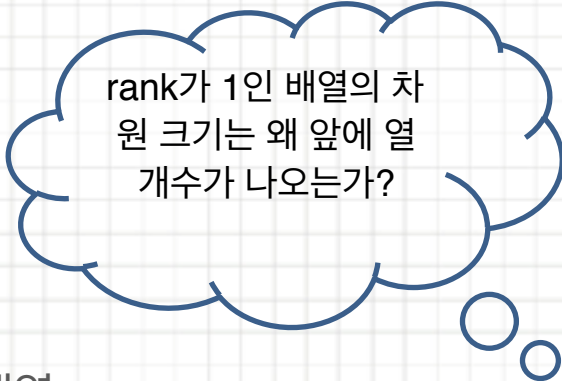
```
def __init__(self, 인수, ): #생성자, 클래스 초기화 방법 정의
def 메서드 이름 1(self, 인수, ): #메서드1
def 메서드 이름 2(self, 인수, ): #메서드2
```

# Python 기초 : Numpy

- **Numpy :**  
계산과학분야에 이용될때 핵심 역할을 하는 파이썬 라이브러리
- 

```
>>> import numpy as np
>>> a = np.array([1, 2, 3]) # rank가 1인 배열 생성 // rank는 배열이 몇 차원인지 의미
>>> print (type(a))        # 출력 "<type 'numpy.ndarray'>"
<class 'numpy.ndarray'>
>>> print (a.shape)         # 출력 "(3,)" // shape는 각 차원의 크기를 알려줌
>>> print (a[0], a[1], a[2]) # 출력 "1 2 3"
>>> a[0] = 5                # 요소를 변경
>>> print (a)               # 출력 "[5, 2, 3]"

>>> b = np.array([[1,2,3],[4,5,6]]) # rank가 2인 배열 생성
[[1, 2, 3]
 [4, 5, 6]]
>>> print (b.shape)         # 출력 "(2, 3)" // 2행 3열의 배열
>>> print (b[0, 0], b[0, 1], b[1, 0]) # 출력 "1 2 4"
```



rank가 1인 배열의 차  
원 크기는 왜 앞에 열  
개수가 나오는가?



# Python 기초 : Numpy 산술연산

- **Numpy 산술연산:**

```
>>> x = np.array([1.0, 2.0, 3.0])
```

```
>>> y = np.array([2.0, 4.0, 6.0])
```

```
>>> print(x+y)
```

```
[ 3.  6.  9.]
```

```
>>> print(x-y)
```

```
[-1. -2. -3.]
```

```
>>> print(x*y)
```

```
[ 2.  8. 18.]
```

```
>>> print(x/y)
```

```
[ 0.5  0.5  0.5]
```

```
>>> print(x/2.0)
```

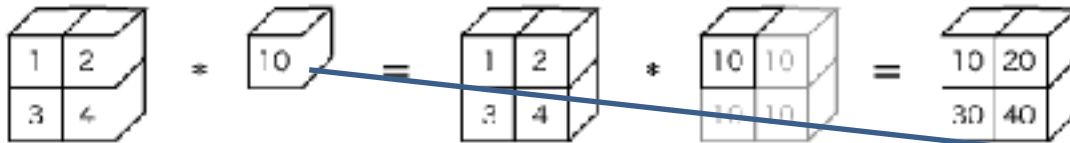
```
[ 0.5  1.  1.5]
```

# Python 기초 : Numpy 브로드캐스트

- 브로드캐스트 :

넘파이에서 형상이 다른 배열을 계산하기 위해서 지원하는 기능

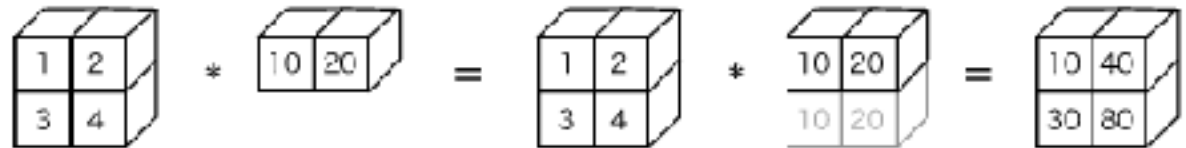
그림 1-1 브로드캐스트의 예 : 스칼라값인 10이 2×2 행렬로 확대된다.



스칼라 값 :  
크기만 있고, 방향(위치)  
정보가 없는 데이터

```
>>> A = np.array([[1, 2], [3, 4]])  
>>> B = np.array([10, 20])  
>>> A * B  
array([[10, 40],  
       [30, 80]])
```

그림 1-2 브로드캐스트의 예 2



# Python 기초 : Numpy 원소접근

- 배열 원소 접근 :  
문자열과 마찬가지로 원소에 인덱싱과 슬라이싱이 가능
- 

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# 슬라이싱을 이용하여 첫 두 행과 1열, 2열로 이루어진 부분배열을 만들어 봅시다;
# b는 shape가 (2,2)인 배열이 됩니다:
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
```

# 슬라이싱된 배열은 원본 배열과 같은 데이터를 참조합니다, 즉 슬라이싱된 배열을 #수정하면 원본 배열 역시 수정됩니다.

```
print a[0, 1] # 출력 "2"
b[0, 0] = 77 # b[0, 0]은 a[0, 1]과 같은 데이터입니다
print a[0, 1] # 출력 "77"
```

# Python 기초 : Matplotlib



## 02.

---

# Perceptron

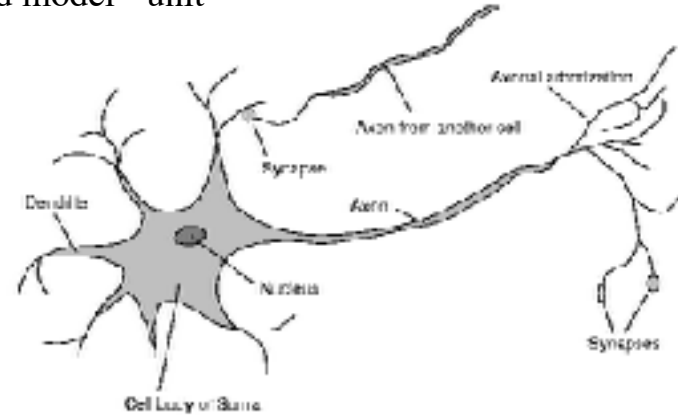
What is the relative number of presentations that will target the audience as a logical  
Expression unnecessary information is released or increased alignment will fail.

# ANN(Artificial Neural Network)

## Neurons vs. Units

- Each element of NN is a node called unit
- Units are connected by links
- Each link has

Real neuron is far away from our simplified model - unit



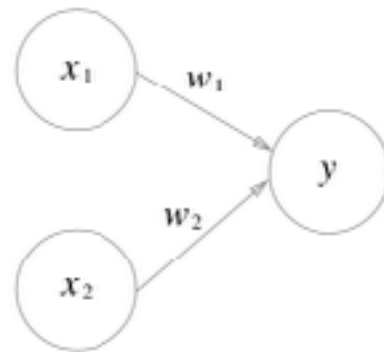
## History

- 1943: McCulloch & Pitts show that **neurons** can be combined to construct a **Turing machine** (using ANDs, Ors, & NOTs)
- 1958: **Rosenblatt** shows that **perceptrons** will converge if what they are trying to learn can be represented
- 1969: **Minsky & Papert** showed the **limitations** of perceptrons, killing research for a decade
- 1985: **backpropagation** algorithm revitalizes the field

# 퍼셉트론

## 퍼셉트론이란?

- 다수의 신호를 입력으로 받아 하나의 신호를 출력한다
- 퍼셉트론은 신호 1/0 값을 가질 수 있고 1은 신호가 흐른다, 0은 신호가 흐르지 않는다는 의미



- 입력이 2개인 퍼셉트론
- $x_1, x_2$ 는 입력 신호
- $y$ 는 출력 신호
- $w_1$ 과  $w_2$ 는 가중치(weight)
- 원을 뉴런 또는 노드라고 함

- 입력 신호가 뉴런에 보내질 때 가가이 가중치가 곱해집니다
- 뉴런에서 보내온 신호 
$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leq \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases}$$
 이 때에만 1을 출력하고, 이 때 한계를 임계값이라고 하며  $\theta$  라고 함

- 아래는  $\theta$ 를  $-b$ (편향) 
$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$
 이 \* 가중치 + 편향  $\geq 0$  이면 1의 값을 가짐

# 퍼셉트론

## 퍼셉트론과 논리회로

$x_1$	$x_2$	$y$
0	0	0
1	0	0
0	1	0
1	1	1

$x_1$	$x_2$	$y$
0	0	1
1	0	1
0	1	1
1	1	0

$x_1$	$x_2$	$y$
0	0	0
1	0	1
0	1	1
1	1	1

### AND, NAND, OR 게이트

- AND, NAND, OR 게이트를 구성하기 위해서는 적절한 ( $w_1, w_2, \theta$ )를 설정
- AND : if ( $w_1, w_2, \theta$ ) = (0.5, 0.5, -0.7),  $x_1 = 1, x_2 = 1$ 일 때에만  $w_1 * x_1 + w_2 * x_2 + b \geq 0$  를 만족
- NAND : if ( $w_1, w_2, \theta$ ) = (-0.5, -0.5, -0.7), 조건식 만족
- OR : if ( $w_1, w_2, \theta$ ) = (0.5, 0.5, -0.2) , 조건식 만족
- 다른 것은 매개변수의 값 뿐임. 즉, 똑같은 구조에서 매개변수만 바꾸는 것임
- 기계학습 문제는 이 매개변수의 값을 정하는 작업을 컴퓨터가 자동으로 하도록 함.

- **학습**  $w$ 는 입력신호( $x$ )가 결과에 주는 영향력을 조정하는 변수이고, 편향은 뉴런이 얼마나 쉽게 활성화 (결과로 1을 출력) 하느냐를 조정하는 매개변수 이다.

예를 들어 편향의 값이 -10 이면 **sum**(입력 신호 \* 가중치) 값이 10이 넘어야 활성화 된다는 의미임



# 퍼셉트론

## 퍼셉트론 구현

```
import numpy as np

def AND(x1, x2, w1 = 0.5, w2 = 0.5, b = -0.7):
    x = np.array([x1, x2])
    w = np.array([w1, w2])
    return 1 if np.sum(x*w) + b >= 0 else 0

def NAND(x1, x2, w1 = -0.5, w2 = -0.5, b = 0.7):
    x = np.array([x1, x2])
    w = np.array([w1, w2])
    return 1 if np.sum(x*w) + b >= 0 else 0

def OR(x1, x2, w1 = 0.5, w2 = 0.5, b = -0.2):
    x = np.array([x1, x2])
    w = np.array([w1, w2])
    tmp = np.sum(x*w) + b
    if tmp <= 0:
        return 0
    else:
        return 1

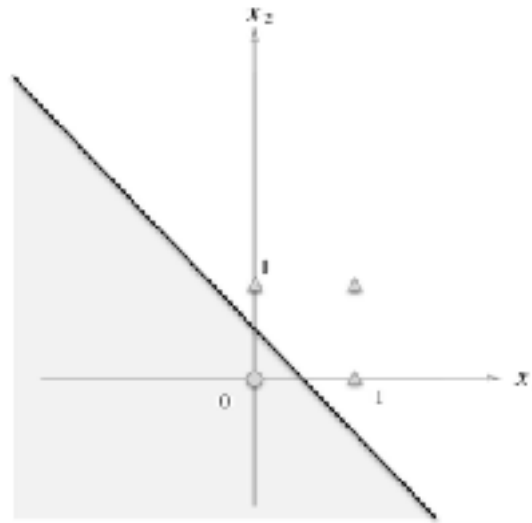
result = AND(0, 0)
result = NAND(0, 0)
result = OR(0, 0)
print(result)
```

# 퍼셉트론

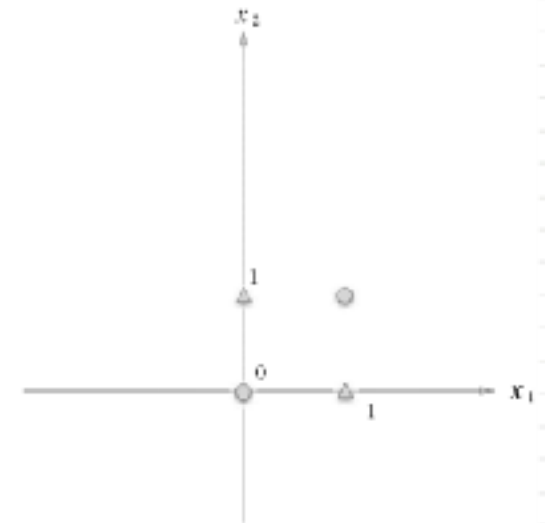
## 퍼셉트론의 한계

$x_1$	$x_2$	$y$
0	0	0
1	0	1
0	1	1
1	1	0

XOR



OR

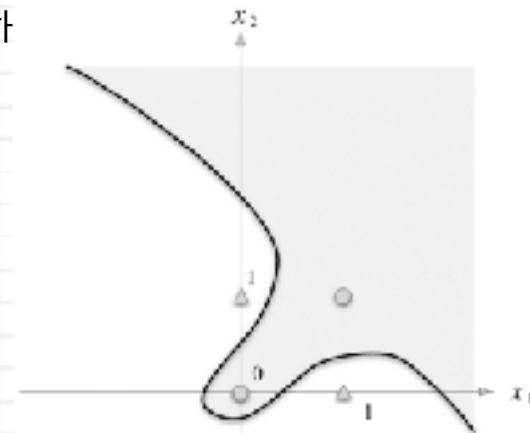


XOR

- XOR 게이트는 AND, NAND, OR 과 같이 단일 퍼셉트론을 이용하
- 예를 들어, ( $w_1$ ,

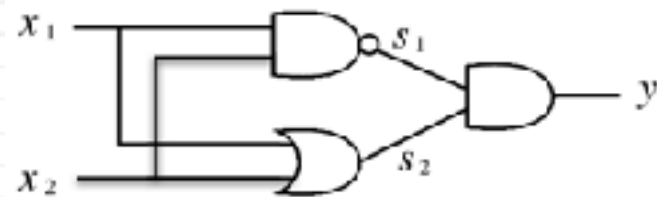
$$y = \begin{cases} 0 & (-0.5 + x_1 + x_2 \leq 0) \\ 1 & (-0.5 + x_1 + x_2 > 0) \end{cases}$$

- 하나의 직선을 이용하여 구분할 수 없음
- XOR의 경우 다음과 같이 비선형으로 나누어야 함.



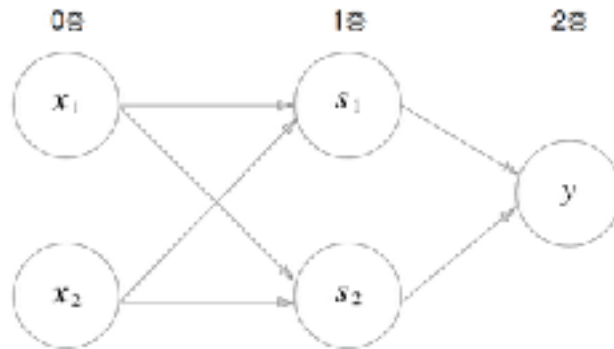
# 퍼셉트론

## XOR 문제 : 층을 하나 더!! (Multi-layer Perceptron)



$x_1$	$x_2$	$s_1$	$s_2$	$y$
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

$$\text{XOR}(x_1, x_2) = \text{AND}(\text{NAND}(x_1, x_2), \text{OR}(x_1, x_2))$$



```
def XOR(x1, x2):  
    s1 = NAND(x1, x2)  
    s2 = OR(x1, x2)  
    return AND(s1, s2)
```

- Layer 0 : 입력값  $x_1, x_2$ , Layer 1 : NAND, OR 게이트 결과 그리고 Layer 2 : AND 게이트 결과
- 0층의 두 뉴런이 입력 신호를 받아 1층의 뉴런으로 신호를 보낸다.
- 1층의 뉴런이 2층의 뉴런으로 신호를 보내고, 2층의 뉴런은 이 신호를 바탕으로  $y$ 를 출력

다층 퍼셉트론은 단층 퍼셉트론으로는 구현하지 못하는 것을 층을 하나 늘리는 것으로 해결

# 03.

---

## Neural Networks

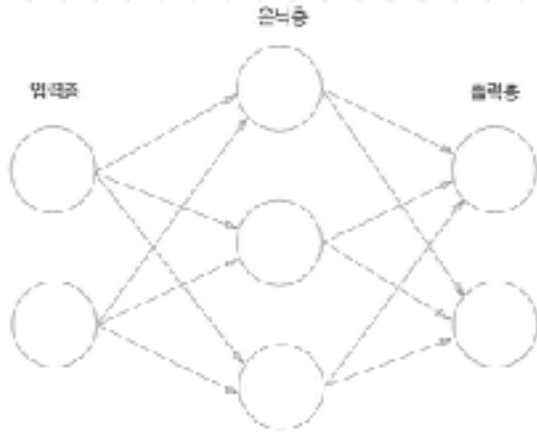
### - Forward Propagation

What is the relative number of presentations that will target the audience as a logical Expression unnecessary information is released or increased alignment will fail.



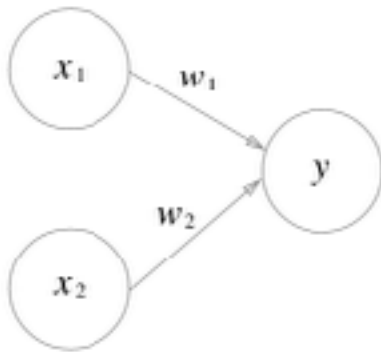
# Neural Network :Forward Propagation

## Example of NN



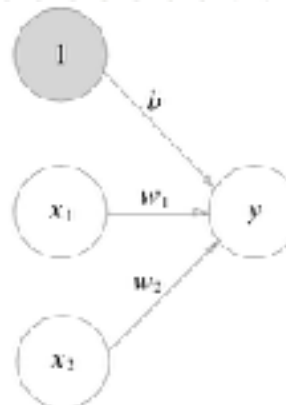
- 입력층(Input Layer)
- 은닉층(Hidden Layer)
- 출력층(Output Layer)
- 3 Layer 혹은 가중치를 가지는 Hidden과 Output만을 고려해 2 Layer NN이라고 하기도 한다.

Re



$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

Perceptron



bias를 고려한 Perceptron

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

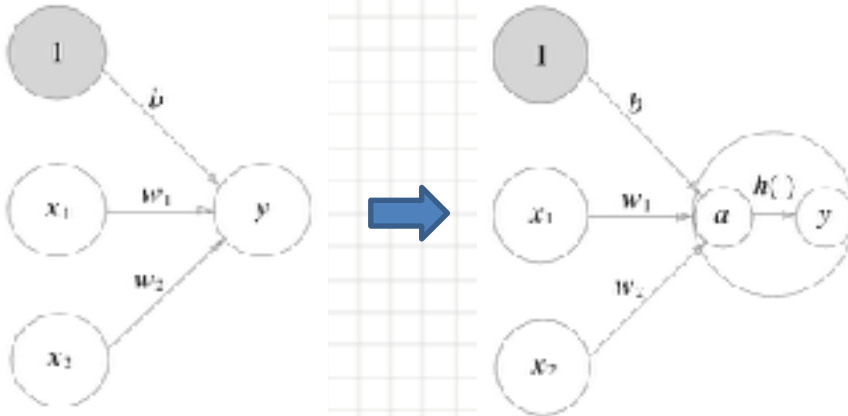
$h(x)$ 함수를 도입

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

$$y = h(b + w_1x_1 + w_2x_2)$$

# Neural Network : Forward Propagation

## Activation Function(활성화 함수)

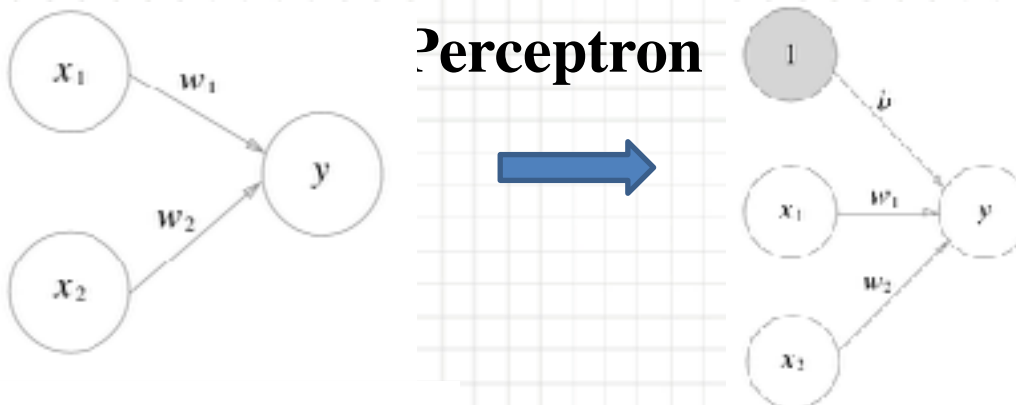


- 가중치가 곱해진 입력 신호와 bias의 총합을 계산하고 이를 a라고 하면, a를 함수 h에 넣어 y를 출력

$$a = b + w_1x_1 + w_2x_2$$

$$y = h(a)$$

## Perceptron



$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

h(x)함수를 도입

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

$$y = h(b + w_1x_1 + w_2x_2)$$

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

bias를 고려한 Perceptron

# Neural Network : Forward Propagation

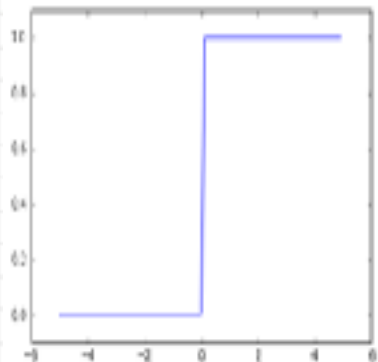
## Various Activation Functions

### < Step Function >

```
import numpy as np
import matplotlib.pyplot as plt

def step_function(x):
    return np.array(x > 0, dtype = np.int)
```

```
x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1)
plt.show()
```

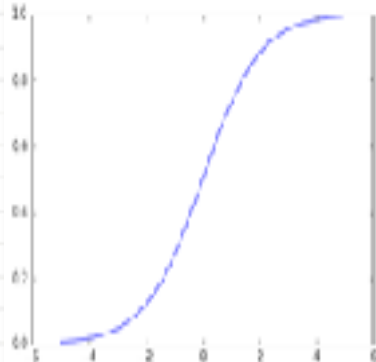


### < Sigmoid Function >

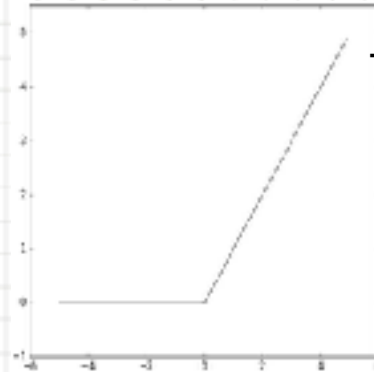
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

### < ReLU Function >

```
def ReLU(x):
    return np.maximum(0, x)
```



$$h(x) = \frac{1}{1 + \exp(-x)}$$



$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

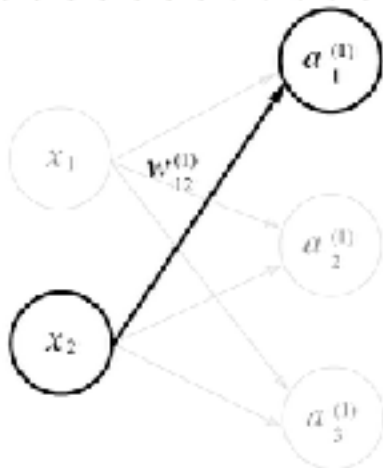
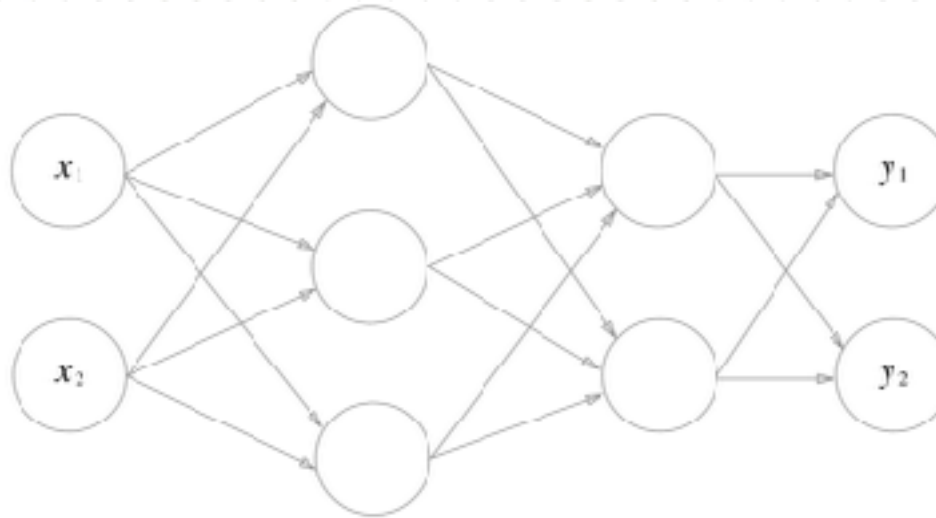
## -ReLU(Rectified Linear Unit)

- Sigmoid : 부드러운 곡선
- 계단함수는 0을 경계로 출력이 갑자기 바뀌게 됨
- sigmoid 함수는 신경망 학습에서 아주 중요한 역할을 함.
- 계단 함수가 0과 1 중 하나의 값만 반환하는 반면 sigmoid 함수는 연속적인 값들을 반환
- 둘 다 입력이 작을 때에는 출력이 0에 가까워지고, 입력이 커지면 출력이 1에 가까워짐
- 입력이 아무리 작거나 커도 출력의 범위는 [0, 1] 임
- 둘 다 비선형 함수

**Nonlinear !  
Why?**

# Neural Network : Forward Propagation

## Implementation of 3 Layered NN



1번 은닉층 : 3개, 2번 은닉층 : 2개, 출력층 : 2개

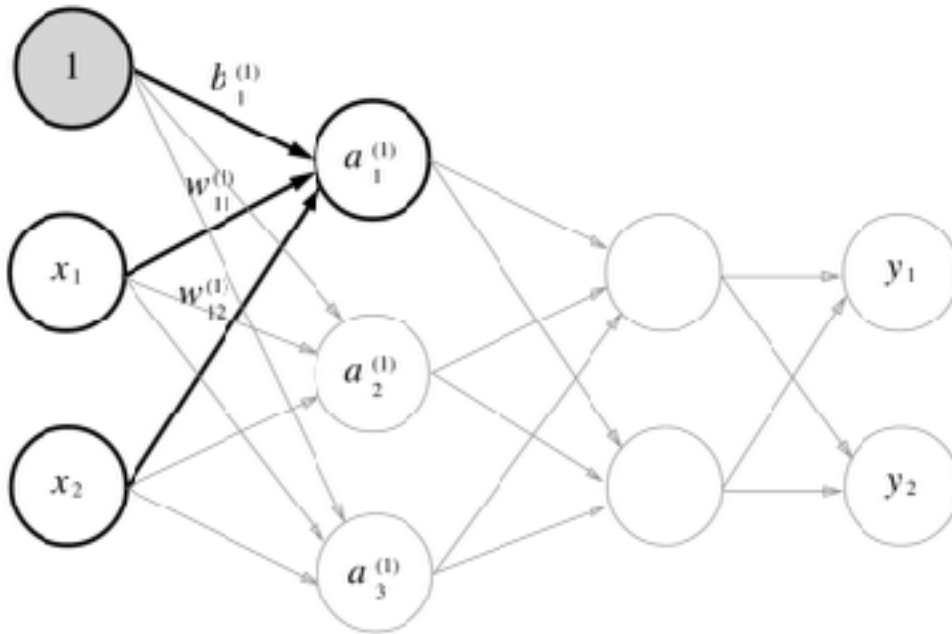
1층의 가중치  
 $w_{12}^{(1)}$   
앞 층의 2번째 뉴런  
다음 층의 1번째 뉴런

- 가중치와 은닉층의 뉴런에는 첨자로 (1)을 붙임  
: 1층의 가중치, 1층의 뉴런을 의미함  
:  $w_{12}^{(1)}$ 는 (1)층으로 향하는 가중치이고  
destination : 1 / source : 2



# Neural Network : Forward Propagation

## Implementation of 3 Layered NN



입력층에서 1층으로 신호 전달

```
import numpy as np
```

```
x = np.array([1.0, 0.5]) # 1 x 2 벡터
```

```
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]]) # 2 x 3 매트릭스
```

```
B1 = np.array([0.1, 0.2, 0.3]) # 1 x 3 벡터
```

```
# np.dot(x, W1) → (1 x 2) · (2 x 3) ⇒ 1 x 3
```

```
A1 = np.dot(x, W1) + B1
```

```
print(A1)
```

$$\mathbf{A}^{(1)} = \mathbf{XW}^{(1)} + \mathbf{B}^{(1)}$$

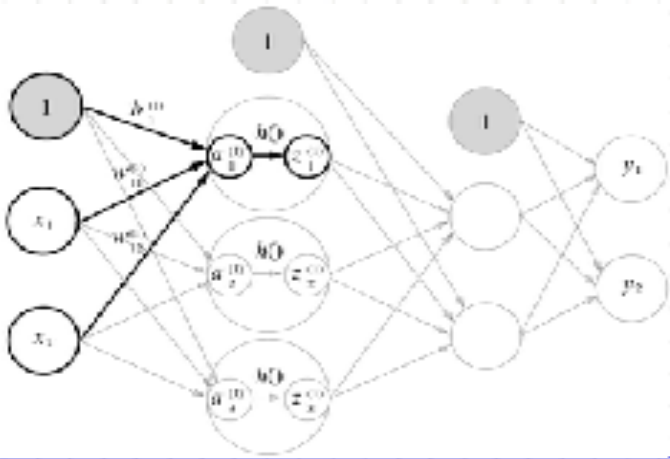
$$\mathbf{A}^{(1)} = (a_1^{(1)} \ a_2^{(1)} \ a_3^{(1)}), \mathbf{X} = (x_1 \ x_2), \mathbf{B}^{(1)} = (b_1^{(1)} \ b_2^{(1)} \ b_3^{(1)})$$

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}$$

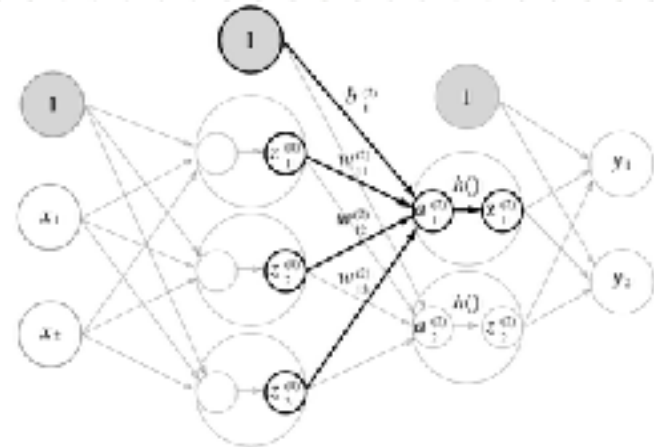
$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix}$$

# Neural Network :Forward Propagation

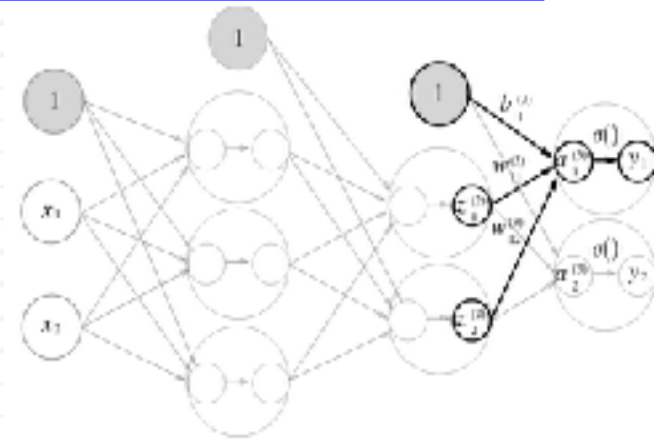
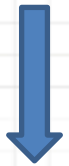
## Implementation of 3 Layered NN



```
x = np.array([1.0, 0.5]) # 1 x 2 벡터
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
# 2 x 3 매트릭스
B1 = np.array([0.1, 0.2, 0.3]) # 1 x 3 벡터
# np.dot(x, W1) → (1 x 2) · (2 x 3) => 1 x 3
A1 = np.dot(x, W1) + B1
Z1 = sigmoid(A1)
```



```
W2 = np.array([ [0.1, 0.4], [0.2, 0.5], [0.3, 0.6] ])
B2 = np.array([0.1, 0.2])
A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```



```
W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])
A3 = np.dot(Z2, W3) + B3
Y = A3
```

# Neural Network :Forward Propagation

## Implementation of 3 Layered NN

```
import numpy as np
```

```
def init_network():
```

```
    network = {}
```

```
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
```

```
    network['b1'] = np.array([0.1, 0.2, 0.3])
```

```
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
```

```
    network['b2'] = np.array([0.1, 0.2])
```

```
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
```

```
    network['b3'] = np.array([0.1, 0.2])
```

```
    return network
```

```
def identical(x):
```

```
    return x
```

```
def sigmoid(x):
```

```
    return 1 / (1 + np.exp(-x))
```

```
def forward(network, x):
```

```
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
```

```
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
```

```
    a1 = np.dot(x, W1) + b1
```

```
    z1 = sigmoid(a1)
```

```
    a2 = np.dot(z1, W2) + b2
```

```
    z2 = sigmoid(a2)
```

```
    a3 = np.dot(z2, W3) + b3
```

```
    y = identical(a3)
```

```
    return y
```

```
network = init_network()
```

```
x = np.array([1.0, 0.5])
```

```
y = forward(network, x)
```

```
print(y)
```

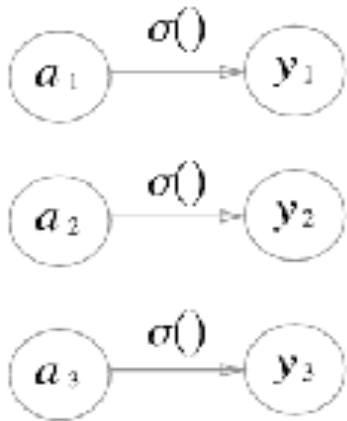
```
=> [0.31682708 0.69627909]
```

# Neural Network : Forward Propagation

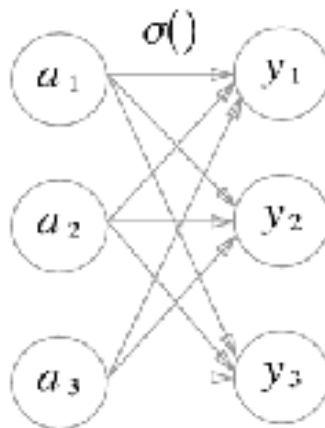
## Implementation of 3 Layered NN : 출력층 설계

- Regression -> identity function, 입력을 그대로 출력
- Classification -> softmax function

<Identity function>



<softmax function>



$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} \rightarrow \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i + \log C)} = \frac{\exp(a_k + \log C)}{\sum_{i=1}^n \exp(a_i + \log C)} = \frac{\exp(a_k + C')}{\sum_{i=1}^n \exp(a_i + C')}$$

```
import numpy as np
```

```
a = np.array([0.3, 2.9, 4.0])
exp_a = np.exp(a)
sum_exp_a = np.sum(exp_a)
y = exp_a / sum_exp_a
print(y)
=> [ 0.01821127  0.24519181  0.73659691]
```

- exponential 연산 시 큰 값이 들어오면 오버플로우 발생  
ex) `np.exp(1000)`

```
import numpy as np
```

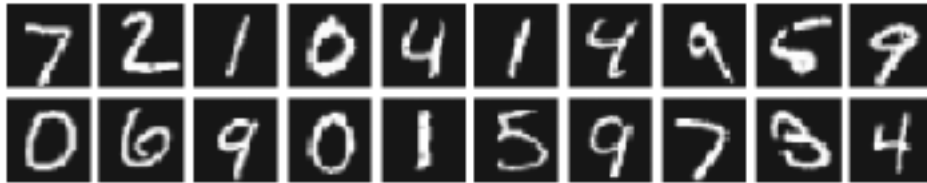
```
def softmax(a):
    exp_a = np.exp(a - np.max(a))
    sum_exp_a = np.sum(exp_a)
    return exp_a / sum_exp_a
```

```
a = np.array([0.3, 2.9, 4.0])
rst = softmax(a)
print(rst)
=> [ 0.01821127  0.24519181  0.73659691]
print(np.sum(rst))
=> 1.0
```

# Neural Network :Forward Propagation

## MNIST 데이터를 이용한 필기체 숫자 인식

- 0부터 9까지 숫자 이미지
- 훈련 이미지가 60,000장, 시험 이미지가 10,000장
- 28 x 28 크기의 gray image(1채널)이며, 각 픽셀은 0에서 255까지 값을 가짐



```
import sys, os
sys.path.append(os.pardir) # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=True)

img = x_train[0]
label = t_train[0]
print(label) # 5

plt.imshow(img.reshape(28, 28), cmap='Greys', interpolation='nearest')
plt.show()
```

# Neural Network :Forward Propagation

## MNIST 데이터를 이용한 필기체 숫자 인식

```
import sys, os
sys.path.append(os.pardir) # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=True)

img = x_train[0]
label = t_train[0]
print(label) # 5

plt.imshow(img.reshape(28, 28), cmap='Greys', interpolation='nearest')
plt.show()
```

- load\_mnist 함수로 MNIST 데이터 셋을 읽음
- load\_mnist가 MNIST 데이터를 받아오므로 최초는 인터넷으로 받아온 뒤 그 이후에는 로컬에 저장된 파일([pickle 파일](#))을 읽어온다.
- load\_mnist 함수는 읽은 MNIST 데이터를 '(훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)' 형식으로 반환
- 인수로는 normalize, flatten, one\_hot\_label 세 가지를 설정. 세 인수 모두 bool 값을 가짐.
  - 1) normalize : 입력 이미지의 픽셀을 0.0 ~ 1.0 사이로 정규화 할지(True) 아니면 기존 그대로 0 ~ 255 값을 사용할 지(False)
  - 2) flatten : 입력 이미지를 1차원 배열로 변환할지(True), 1x28x28 로 유지할 지(False)
  - 3) one\_hot\_label : 원 핫 인코딩 형태로 저장할 지 결정
    - ☞ 원-핫 인코딩 : 정답을 뜻하는 원소만 1, 나머지는 0의 값을 가진다.  
[0,0,0,1,0,0,0]

# Neural Network :Forward Propagation

## MNIST 필기체 숫자 인식 : 신경망의 추론처리(forward)

- MNIST 데이터 셋을 가지고 추론을 수행하는 신경망
- 하나의 이미지는  $28 \times 28 = 784$ 개의 입력층을 가지고, 출력층은 10개의 뉴런(0 ~9)으로 구성
- 은닉층은 총 2개로 배치, 임의로 첫번째 은닉층은 50개의 뉴런, 두번째 은닉층은 100 개의 뉴런 배치
- `init_network()` 에서는 pickle 파일인 `sample_weight.pkl`에 저장된 '학습된 가중치 매개변수'를 읽어옴
- 이 파일에는 가중치와 bias 매개변수가 딕셔너리 변수로 저장
- `sigmoid()` 함수와 `softmax()` 함수를 이용

```
import sys, os
import pickle
import numpy as np

sys.path.append(os.pardir)
from dataset.mnist import load_mnist

def get_data():
    # normalize = True로 설정하여 각 픽셀의 범위를 [0, 1.0]으로 정규화 시킴
    (x_train, t_train), (x_test, t_test) = \
        load_mnist(flatten=True, normalize=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open('sample_weight.pkl', 'rb') as f:
        network = pickle.load(f)
    return network

def softmax(a):
    exp_a = np.exp(a - np.max(a))
    sum_exp_a = np.sum(exp_a)
    return exp_a / sum_exp_a

def sigmoid(a):
    return 1 / (1 + np.exp(-a))
```

```
def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y

x, t = get_data()
network = init_network()

accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p = np.argmax(y) # 확률이 가장 높은 원소의 인덱스를 얻는다.
    if p == t[i]:
        accuracy_cnt += 1

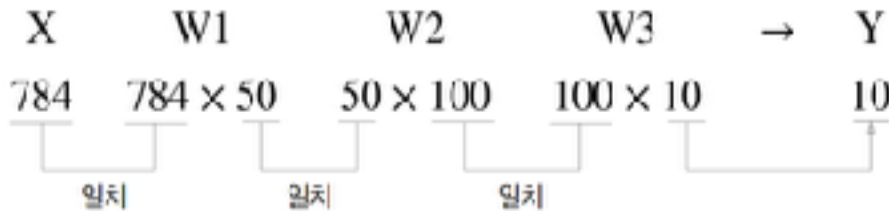
print('Accuracy : ' + str(float(accuracy_cnt) / len(x)))
# Accuracy: 0.9352
```



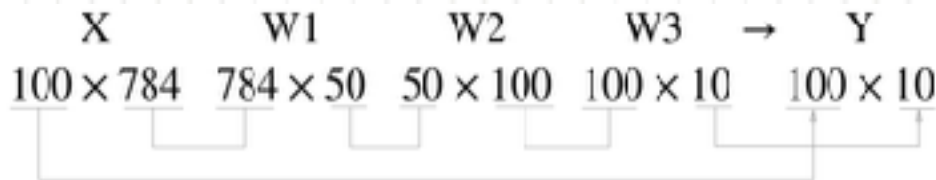
# Neural Network : Forward Propagation

## MNIST 필기체 숫자 인식 : Batch Process

- Network Size



- Batch Process : 100개를 묶어서 배치처리 예



```
x, t = get_data()
network = init_network()
```

```
batch_size = 100 # 배치의 크기
accuracy_cnt = 0
```

```
t1 = time.time()
for i in range(0, len(x), batch_size):
    x_batch = x[i:i + batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1) # 확률이 가장 높은 원소의 인덱스를 얻는다.
    accuracy_cnt += np.sum(p == t[i:i + batch_size])
```

```
t2 = time.time()
print('Accuracy : ' + str(float(accuracy_cnt) / len(x)))
```

```
print('Execution Time : ' + str(t2 - t1))
```

- 작은 배열을 여러 번 계산하는 것 보다 큰 배열로 한번에 계산하는 것이
- Numpy 라이브러리에서 큰 배열을 효율적으로 처리
- I/O 에 비용이 많이 들어 I/O 횟수를 줄이고 CPU/GPU로 순수 계산을 수행하는 것이 훨씬 효율적
- 10000개를 한꺼번에 배치 처리하면?



# 04.

---

## Neural Networks

### - Loss Function

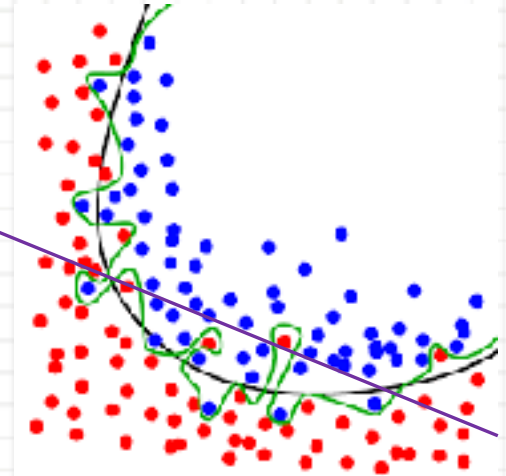
What is the relative number of presentations that will target the audience as a logical Expression unnecessary information is released or increased alignment will fail.



# Neural Network : Loss Function

## Data in Deep Learning

- 데이터 주도 학습



**Underfitting & Overfitting**

- End-to-end machine learning

## Training vs. Test Data in Machine Learning

- Training data : 학습을 통해 최적 매개변수를 찾는다
- Test data : 훈련한 모델의 능력을 평가

\* **Overfitting** : 특정 데이터셋에 과도하게 최적화된 경우

# Neural Network : Loss Function

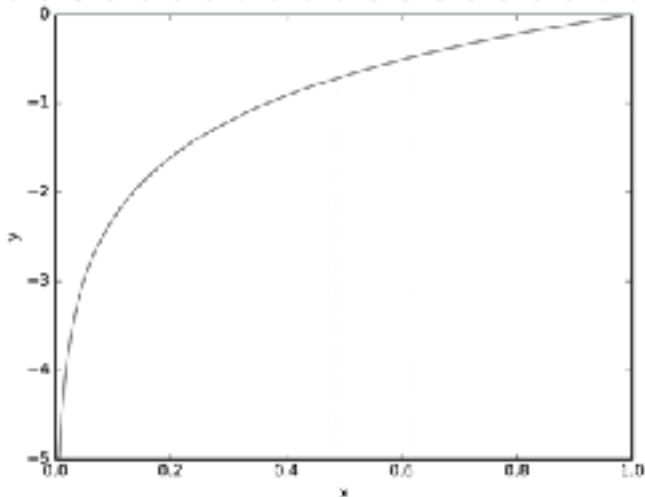
## Loss Function

- 최적의 매개변수를 찾아내기 위한 지표
- Mean Square Error

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

- Cross Entropy Error

$$E = - \sum_k t_k \log y_k$$



- $y = \log(x)$
- $x=1$ 에 근접하면,  $y=0$ 에 가까워진다.
- $x=0$ 에 근접하면,  $y$ 값은 점점 작아진다.
- 정답에 해당하는 출력이 커질수록 0에 가까워짐
- 정답에 해당하는 출력이 작아질수록 오차는 커진다

```
def mean_squared_error(y, t):  
    return 0.5*np.sum((y-t)**2)
```

```
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]  
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]  
d = mean_squared_error(np.array(y), np.array(t))  
print(d)
```

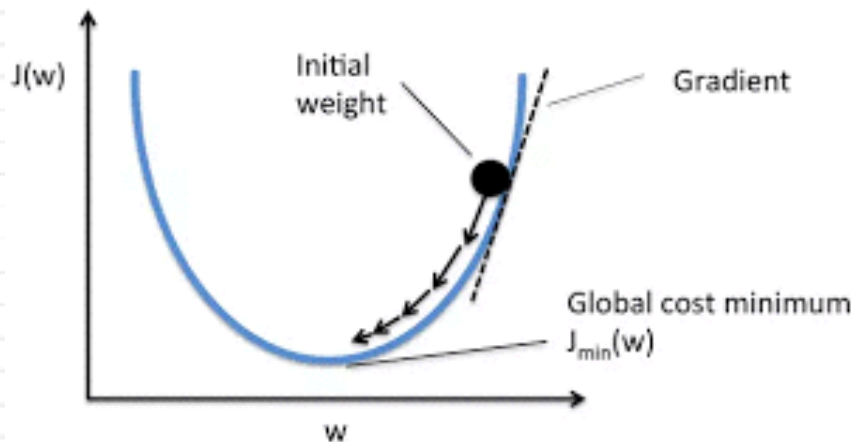
```
def cross_entropy_error(y, t):  
    delta = 1e-7  
    return -np.sum(t*np.log(y+delta))
```

```
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]  
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]  
d = cross_entropy_error(np.array(y), np.array(t))  
print(d)
```

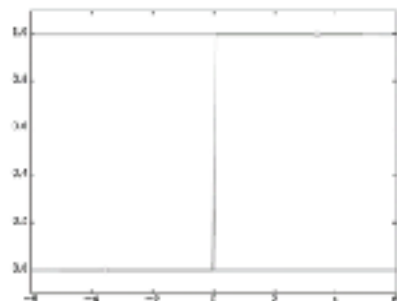
# Neural Network : Loss Function

## Why we have to set “Loss Function”?

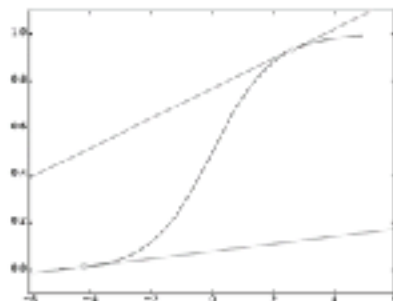
- 최적의 매개변수를 찾을 때, 손실함수가 최소가 되게 하는 매개변수를 찾으면 됨
- 손실함수의 미분 : 손실함수의 변화량을 반영
  - 미분이 음수이면 매개변수를 양의 방향으로 변화시켜 손실함수를 줄인다
  - 미분이 양수이면 매개변수를 음의 방향으로 변화시켜 손실함수를 줄인다
  - 미분이 0 이면 손실함수의 값은 달라지지 않으므로 멈춘다.



계단 함수



시그모이드 함수



- Sigmoid는 미분값이 0이 아니고 연속적으로 변한다.
- 계단함수는 0점을 제외한 미분값이 0이 된다.
- 매개변수의 작은 변화가 주는 효과를 계단함수가 사라지게 한다 (미분하면 0이 되므로)

# Neural Network : Loss Function

## 수치미분

- 미분은 한 순간의(아주 작은 한 구간) 변화량을 의미

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

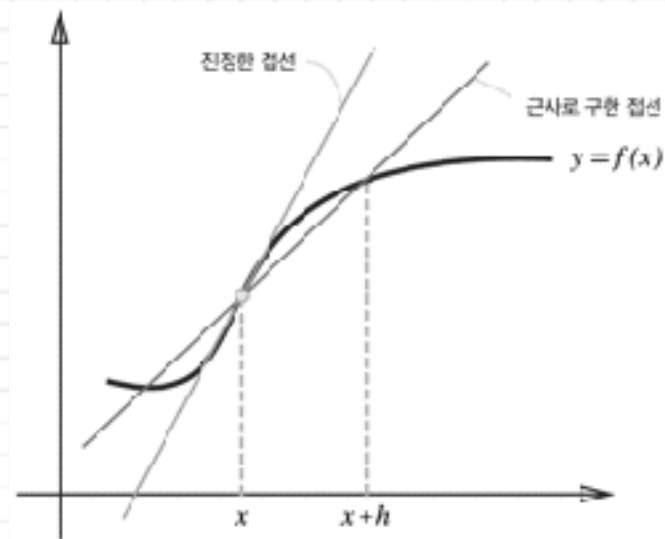
: Computer Program으로 해결 ⇨ 수치미분(numerical differentiation)

```
def numerical_diff(f, x):  
    h = 1e-50  
    return (f(x+h) - f(x)) / h
```

h가 너무 작으면, computer에서  
반올림오차(rounding error)가 생김

- 실제 미분(접선)과 수치미분(근사접선)은 다르다.
  - 오차를 줄이기 위해 중심(중앙) 차분을 사용  
:중앙차분 ⇨  $f(x+h)$ 와  $f(x-h)$ 사이의 차분을 구한다.
- 수정된 최종 수치미분 함수

```
def numerical_diff(f, x):  
    h = 1e-4 # 0.0001  
    return (f(x+h) - f(x-h)) / (2*h)
```



# Neural Network : Loss Function

## 수치미분의 예

- $y = 0.01x^2 + 0.1x$
- 함수를 그리고 접선을 표시

```
def function_1(x):  
    return 0.01*x**2 + 0.1*x
```

< lambda 함수 란? >  
익명 함수 또는 함수의 축약형이다..  
표현식 : lambda 인수 : <구문>  
ex) lambda x : x + 1  
ex) g = lambda x, y : x \* y  
g(2,3)  
6

```
import numpy as np  
import matplotlib.pyplot as plt
```

```
def numerical_diff(f, x):  
    h = 1e-4 # 0.0001  
    return (f(x+h) - f(x-h)) / (2*h)
```

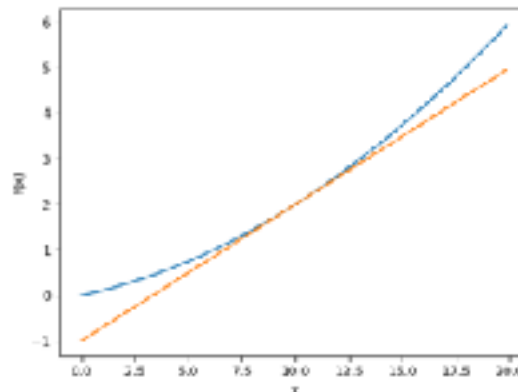
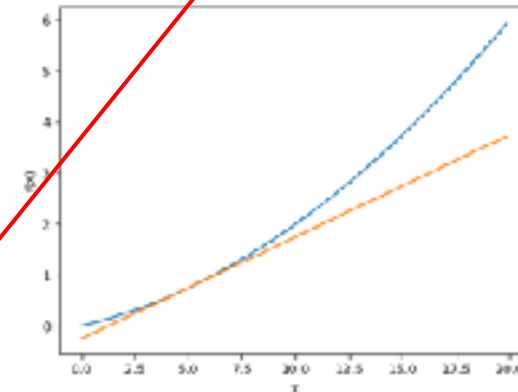
```
def function_1(x):  
    return 0.01*x**2 + 0.1*x
```

```
def tangent_line(f, x):  
    d = numerical_diff(f, x)  
    print(d)  
    n = f(x) - d*x # n:절편(y = f(x) = dx + n)  
    return lambda t: d*t + n # y = dx + n
```

```
x = np.arange(0.0, 20.0, 0.1)  
y = function_1(x)  
plt.xlabel("x")  
plt.ylabel("f(x)")
```

```
tf = tangent_line(function_1, 5)  
y2 = tf(x)
```

```
plt.plot(x, y)  
plt.plot(x, y2)  
plt.show()
```



# Neural Network : Loss Function

## 수치 편미분의 예

$$f(x_0, x_1) = x_0^2 + x_1^2$$

$\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}$  을 구한다.

```
def function_2(x):  
    return x[0]**2 + x[1]**2 # 또는 return np.sum(x**2)
```

```
def function_tmp1(x0):  
    return x0*x0 + 4.0**2.0
```

```
def function_tmp2(x1):  
    return 3.0**2.0 + x1*x1
```

```
from mpl_toolkits.mplot3d import Axes3D  
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib import cm  
from matplotlib.ticker import LinearLocator, FormatStrFormatter
```

```
fig = plt.figure()  
ax = fig.gca(projection='3d')
```

```
def function_2(x):  
    return x[0]**2 + x[1]**2 # 또는 return np.sum(x**2)
```

*# Make data.*

```
X = np.arange(-3.0, 3.0, 0.1)
```

```
Y = np.arange(-3.0, 3.0, 0.1)
```

```
X, Y = np.meshgrid(X, Y)
```

```
x = [X, Y]
```

```
Z = function_2(x)
```

*# Plot the surface.*

```
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False)
```

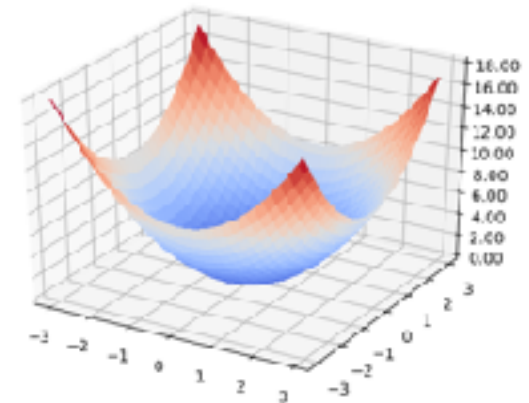
```
ax.zaxis.set_major_locator(LinearLocator(10))
```

```
ax.zaxis.set_major_formatter(FormatStrFormatter('%0.02f'))
```

*# Add a color bar which maps values to colors.*

```
#fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
plt.show()
```



$$f(x_0, x_1) = x_0^2 + x_1^2$$

# Neural Network : Loss Function

## 수치 편미분의 예

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
def _numerical_gradient_no_batch(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성

    for idx in range(x.size):
        tmp_val = x[idx]

        # f(x+h) 계산
        x[idx] = float(tmp_val) + h
        fxh1 = f(x)

        # f(x-h) 계산
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # 값 복원

    return grad
```

```
def numerical_gradient(f, X):
    if X.ndim == 1:
        return _numerical_gradient_no_batch(f, X)
    else:
        grad = np.zeros_like(X)

        for idx, x in enumerate(X):
            grad[idx] = _numerical_gradient_no_batch(f, x)

    return grad
```

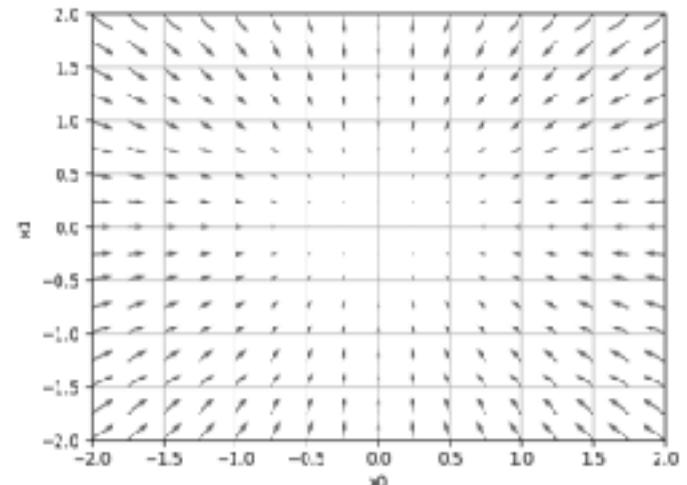
```
def function_2(x):
    if x.ndim == 1:
        return np.sum(x**2)
    else:
        return np.sum(x**2, axis=1)
```

```
x0 = np.arange(-2, 2.5, 0.25)
x1 = np.arange(-2, 2.5, 0.25)
X, Y = np.meshgrid(x0, x1)
```

```
X = X.flatten()
Y = Y.flatten()
```

```
grad = numerical_gradient(function_2, np.array([X, Y]) )
```

```
plt.figure()
plt.quiver(X, Y, -grad[0], -grad[1], angles="xy", color="#666666")
plt.xlim([-2, 2])
plt.ylim([-2, 2])
plt.xlabel('x0')
plt.ylabel('x1')
plt.grid()
plt.legend()
plt.draw()
plt.show()
```





# Neural Network : Loss Function

## 신경망에서의 기울기

- 신경망 학습에서도 기울기를 구한다. : 신경망의 기울기  $\Rightarrow$  가중치 매개변수에 대한 손실함수의 기울기
- 예를 들어 2x3 행렬의 가중치  $W$ , 손실함수  $L$  인 신경망의 경우

$$W = \begin{pmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \end{pmatrix}$$

$\frac{\partial L}{\partial W}$ 의 형상과  $W$ 의  
형상의 같다

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \frac{\partial L}{\partial W_{12}} & \frac{\partial L}{\partial W_{13}} \\ \frac{\partial L}{\partial W_{21}} & \frac{\partial L}{\partial W_{22}} & \frac{\partial L}{\partial W_{23}} \end{pmatrix}$$

$\frac{\partial L}{\partial W_{11}} \Rightarrow W_{11}$ 이 변했을 때, 손실함수  $L$ 이 얼마나 변화하느냐?

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient
```

```
class simpleNet:
```

```
    def __init__(self):
```

```
        self.W = np.random.randn(2,3) # 정규분포로 초기화
```

```
    def predict(self, x):
```

```
        return np.dot(x, self.W)
```

```
    def loss(self, x, t):
```

```
        z = self.predict(x)
```

```
        y = softmax(z)
```

```
        loss = cross_entropy_error(y, t)
```

```
        return loss
```

```
x = np.array([0.6, 0.9])
```

```
t = np.array([0, 1])
```

```
net = simpleNet()
```

```
f = lambda w: net.loss(x, t)
```

```
dW = numerical_gradient(f, net.W)
```

```
print(dW)
```

```
def numerical_gradient(f, x):
```

```
    h = 1e-4 # 0.0001
```

```
    grad = np.zeros_like(x)
```

```
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
```

```
    while not it.finished:
```

```
        idx = it.multi_index
```

```
        tmp_val = x[idx]
```

```
        x[idx] = float(tmp_val) + h
```

```
        fxh1 = f(x) # f(x+h)
```

```
        x[idx] = tmp_val - h
```

```
        fxh2 = f(x) # f(x-h)
```

```
        grad[idx] = (fxh1 - fxh2) / (2 * h)
```

```
        x[idx] = tmp_val # 값 복원
```

```
        it.iternext()
```

```
    return grad
```

$$\frac{\partial L}{\partial W} =$$

```
[[ 0.37337484  0.19236272 -0.56660066]
 [ 0.56060691  0.28929408 -0.84990099]]
```

< Numpy.nditer 다차원 배열 순회 >

- 다차원 배열을 다차원 iterator로 변환
- Iterator에는 다양한 method 존재
- it.multi\_index: 다차원 배열의 인덱스를 tuple로 반환 받는다.
- 2x3 행렬의 경우 : (0,0), (0,1), (0,2), (1,0), (1,1), (1,2)를 차례로 받는다.
- For문 등을 사용하지 않아도 다차원 배열을 간단히 순회할 수 있다.

# Neural Network : Loss Function

## MNIST 데이터로 평가

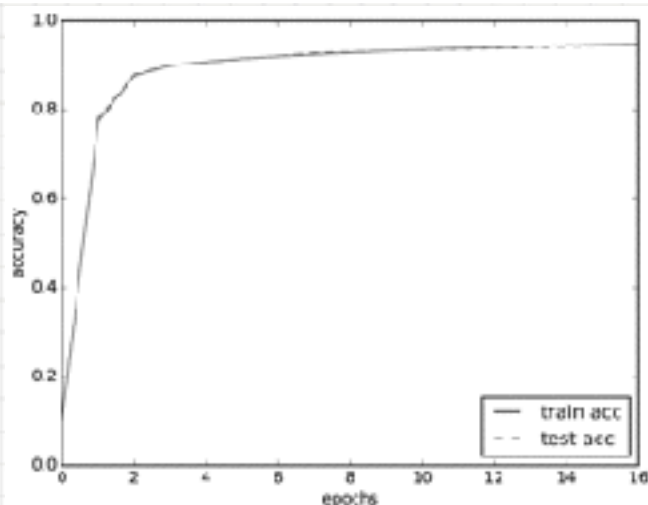
```
import sys, os
sys.path.append(os.pardir) # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼파라미터
iters_num = 10000 # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100 # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []
```



```
# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    # grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)

    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    # 1에폭당 정확도 계산
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# 그래프 그리기
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

# 05.

---

## Neural Networks

### - Backpropagation

What is the relative number of presentations that will target the audience as a logical Expression unnecessary information is released or increased alignment will fail.



# Neural Network :Backpropagation

## Recapitulation

- Training(학습): 훈련 데이터로부터 가중치 매개변수의 최적 값을 자동으로 획득
  - Loss Function(손실 함수)
    - ✓ 신경망 학습의 지표
    - ✓ 일반적인 손실함수: MSE(평균 제곱 오차), Cross-entropy(교차 엔트로피 오차)
    - ✓ 손실 함수의 결과값을 가장 작게 만드는 것이 목표
      - Gradient Descent Method(경사 하강법)
        - ◆ 목표 달성을 위한 기법
        - ◆ 기울기를 활용해 손실 함수의 최솟값을 찾고 싶다
        - ◆ 손실 함수의 기울기를 수치 미분을 사용해 구현
          - 수치 미분의 장, 단점
            - 장점: 단순, 구현이 쉬움
            - 단점: 계산 시간이 오래 걸림
            - 해결방안: 오차역전파법(**backpropagation**)

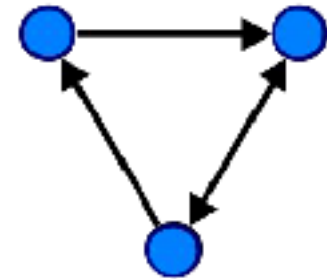
# Neural Network : Backpropagation

## Computational Graph(계산 그래프)

- 목표 : 계산 그래프를 이용해 오차역전파법을 시각적으로 이해

- Computational Graph

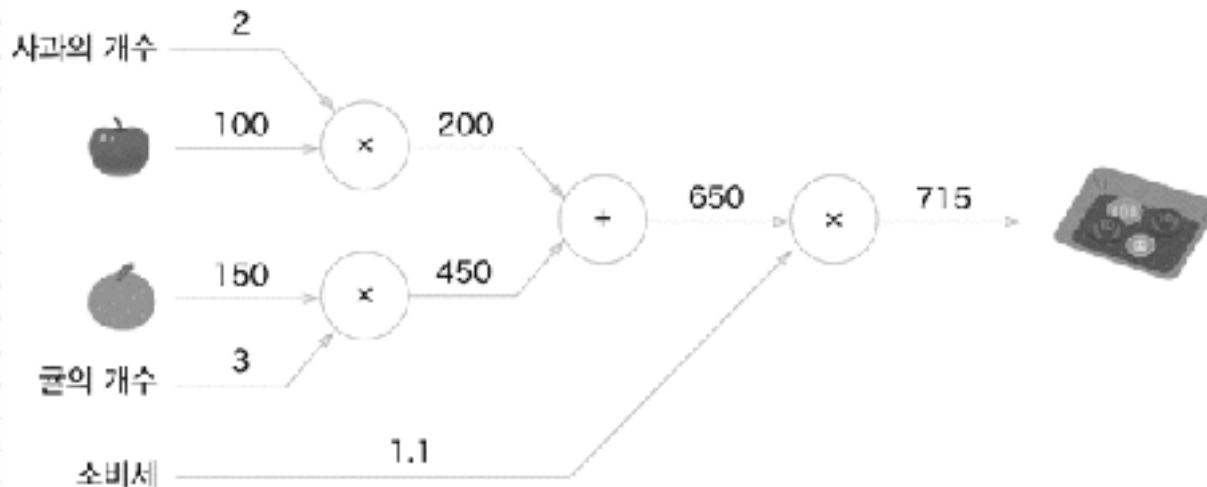
- 그래프 자료구조
- Node(노드) : 원, 연산자
- Edge(에지) : 원 사이의 화살표, 입출력 방향



3개의 node와 3개의 edge를 가진 그래프

- 문제풀이

- 문제 2. 당신은 슈퍼에서 사과를 2개, 귤을 3개 샀습니다. 사과는 1개에 100원, 귤은 1개에 150원입니다. 소비세가 10%일 때 지불 금액을 구하시오.



# Neural Network : Backpropagation

## Computational Graph(계산 그래프)

### - Computational Graph 풀이 방법

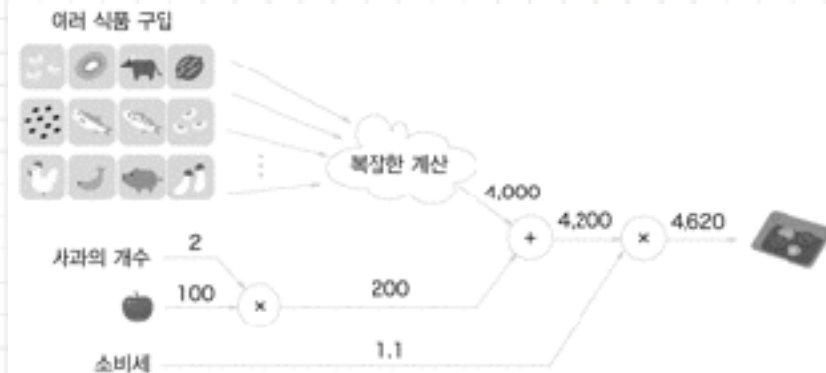
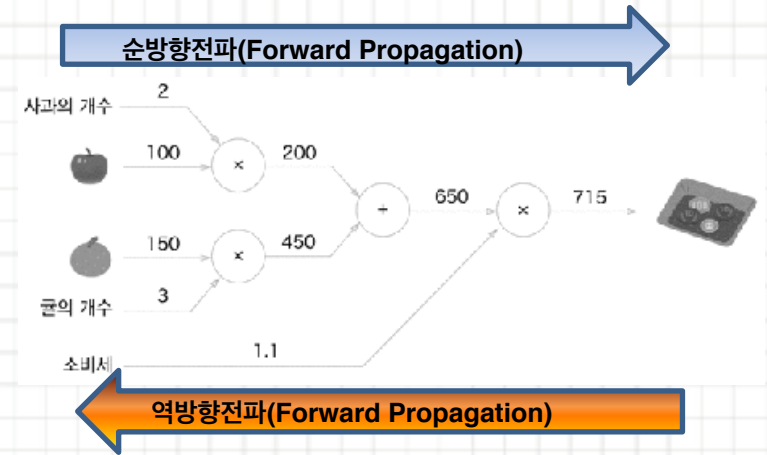
- Draw “Computational Graph”
- 계산을 왼쪽(입력)에서 오른쪽(출력)으로 진행
- Edge(에지) : 원 사이의 화살표, 입출력 방향

### - Local Computation(국소적 계산)

- Computational Graph의 특징: 국소적 계산을 전파 → 최종 결과 도출
- Local(국소적): 자신과 직접 관계된 작은 범위
- 각 node에서의 계산은 국소적 계산이다.

### - Why Computational Graph ?

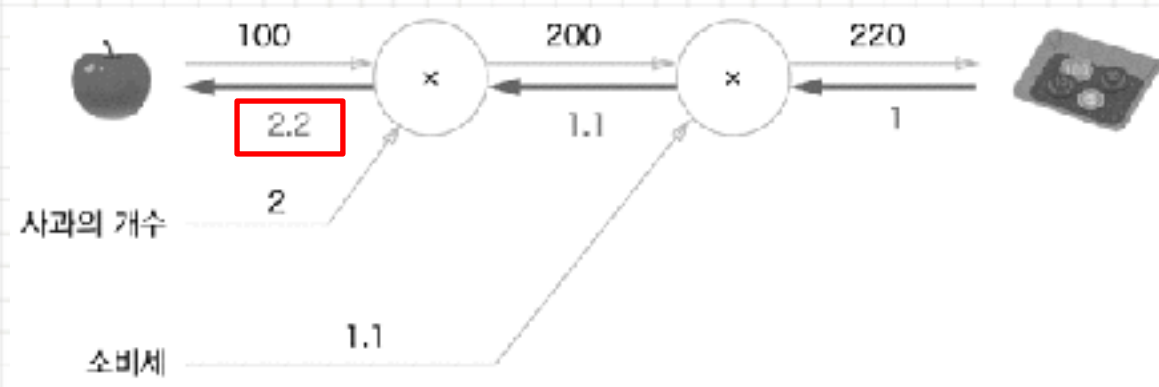
- 국소적 계산의 이점으로 문제를 단순화
- 계산의 중간 결과를 모두 보관
- **Backpropation(역전파)**를 통한 미분의 효율적인 계산



# Neural Network : Backpropagation

## Computational Graph(계산 그래프)

- 사과 가격이 올랐을 때 최종 지불 금액의 변화량은?
  - '사과 가격에 대한 지불 금액의 미분'을 구하는 문제
  - $x$  = 사과 값,  $L$  = 지불 금액일 때,  $\frac{\partial L}{\partial x}$ 을 구하는 문제
  - 계산 그래프의 역전파를 통해 구할 수 있다.



해석) 사과 1개의 가격: 100원 → 101원  
지불 금액 : 220원 → 222.2원



# Neural Network : Backpropagation

## Chain Rule(연쇄법칙)

- Backpropagation(역전파)은 '국소적인 미분'을 순방향과는 반대로 전달
- '국소적 미분'을 전달하는 원리는 연쇄법칙을 따름
- $y = f(x)$  계산의 backpropagation
- Computational procedure of Backpropagation
  - 신호  $E$ 에 node의 국소적 미분  $\frac{\partial L}{\partial x}$ 을 곱한 후, 다음 node로 전달
  - 국소적 미분: forward propagation(순전파) 때,  $y = f(x)$  계산의 미분



: 목표 미분 값의 효율적 도출 → 연쇄법칙으로 설명

- 합성함수 : 여러 함수로 구성된 함수
- 연쇄법칙
  - 합성 함수의 미분에 대한 성질
  - 원리: 합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.

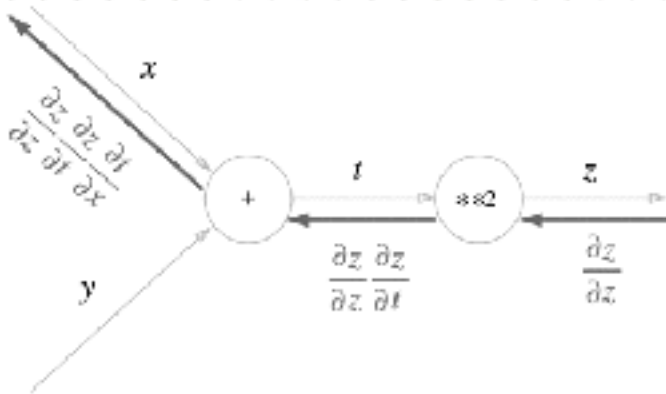
$$\begin{aligned} z &= t^2 & \frac{\partial z}{\partial t} &= 2t \\ t &= x + y & \frac{\partial t}{\partial x} &= 1 \end{aligned} \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \cdot 1 \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x + y)$$



# Neural Network : Backpropagation

## Chain Rule(연쇄법칙)

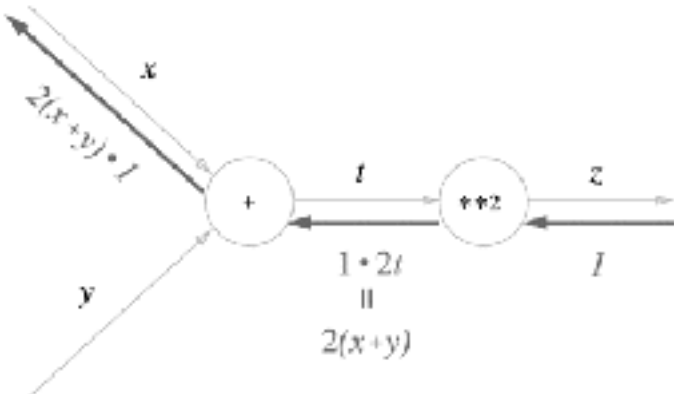
- 계산 그래프로 연쇄법칙 나타내기



- 맨 왼쪽 역전파는 연쇄법칙에 따라 'x에 대한 z의 미분'이 된다.
- 역전파가 하는 일은 연쇄법칙의 원리와 같다.

$$\boxed{\frac{\partial z}{\partial z} \frac{\partial z}{\partial t} \frac{\partial t}{\partial x}} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = \frac{\partial z}{\partial x}$$

- 연쇄법칙과 계산 그래프

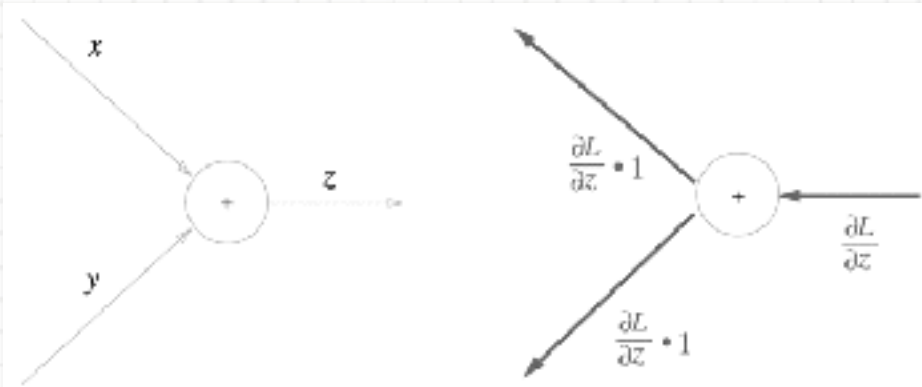


$$\begin{aligned} z &= t^2 & \frac{\partial z}{\partial t} &= 2t \\ t &= x + y & \frac{\partial t}{\partial x} &= 1 \end{aligned}$$

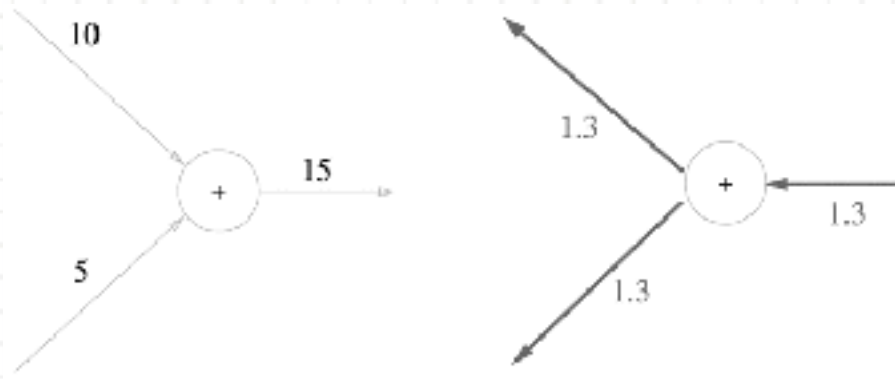
# Neural Network : Backpropagation

## Backpropagation(역전파) : 덧셈 노드

- $L = \text{최종 출력값}$   $\frac{\partial z}{\partial x} = 1$
- $z = x + y$   $\frac{\partial z}{\partial y} = 1$



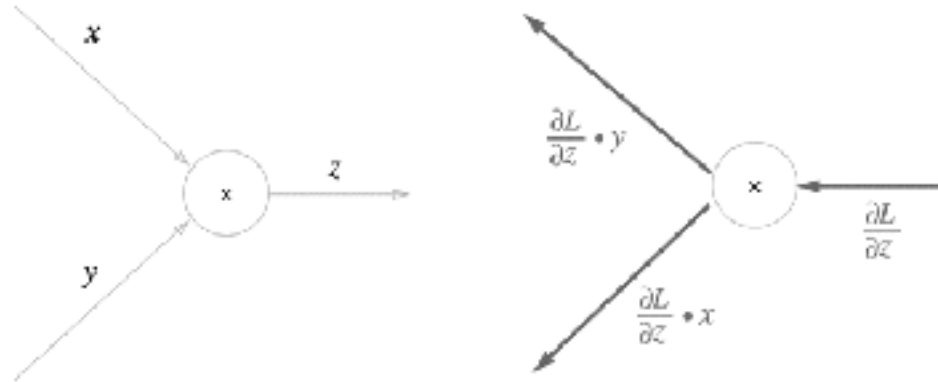
- 덧셈 노드의 역전파
  - 가정) 상류에서 1.3 값이 입력됨.
  - 덧셈 노드는 역전파는 입력된 값을 그대로 다음 노드로 전파함.



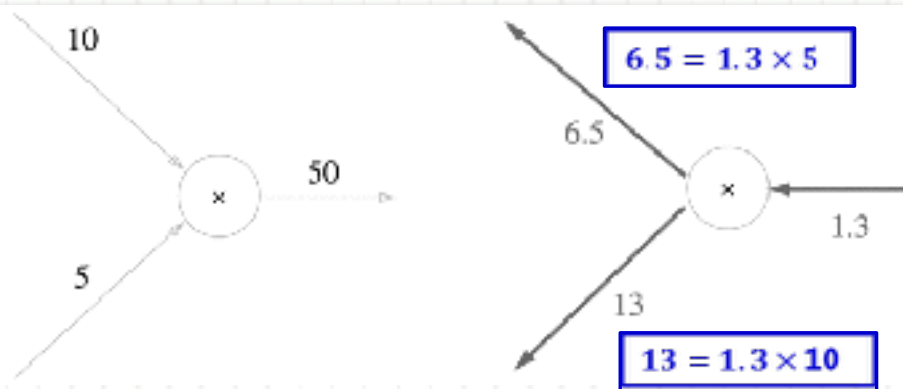
# Neural Network : Backpropagation

## Backpropagation(역전파) : 곱셈 노드

- $L = \text{최종 출력값}$   $\frac{\partial z}{\partial x} = y$
- $z = xy$   $\frac{\partial z}{\partial y} = x$

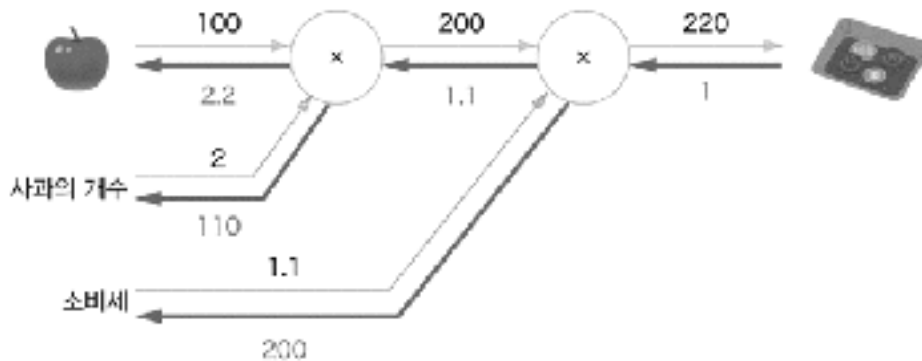


- 곱셈 노드의 역전파
  - 상류의 값에 순전파의 입력 신호를 '서로 바꾼 값'을 곱해서 하류로 보냄
  - 순방향 입력 신호 값이 필요

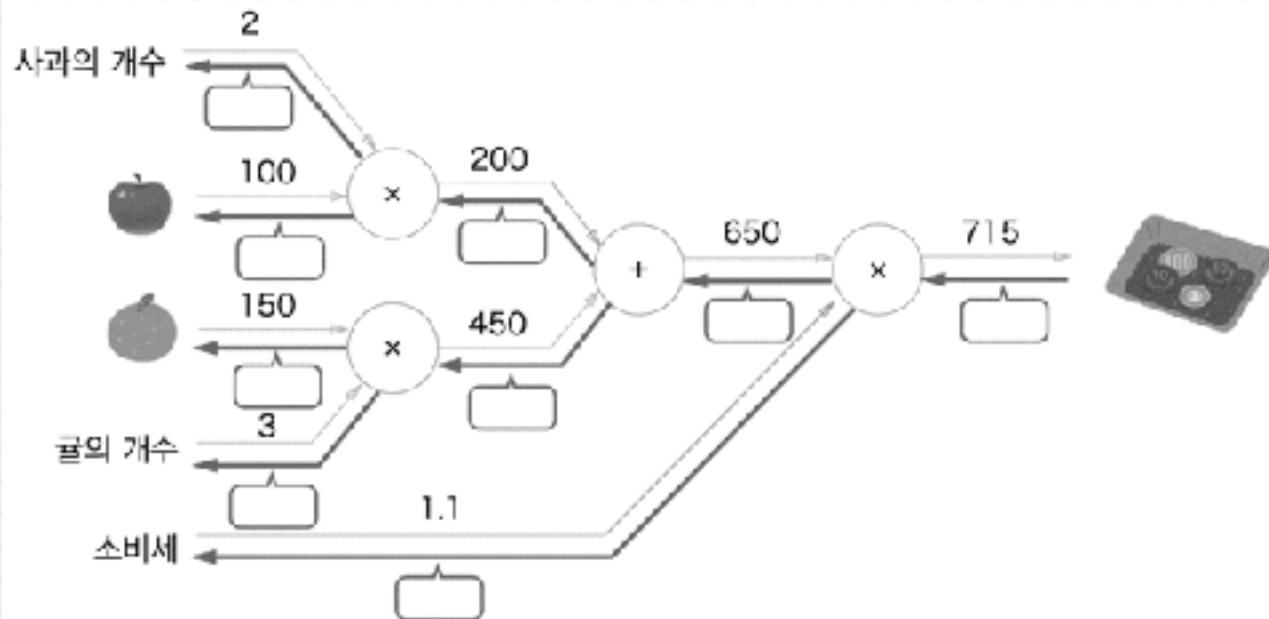


# Neural Network : Backpropagation

## 사과 쇼핑의 역전파 예



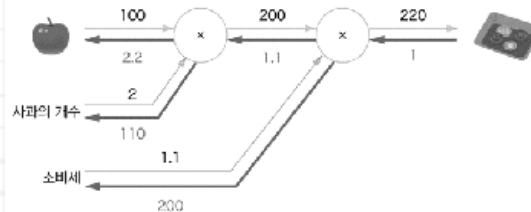
해석) 소비세와 사과 가격이 같은 양만큼 오르면  
최종 금액에는 소비세가 200 크기로, 사과  
가격이 2.2 크기로 영향을 준다.  
단위에 주의.



# Neural Network : Backpropagation

## Backpropagation : 단순한 계층 구현(사과 쇼핑)

- 곱셈 계층 : MulLayer class
- 덧셈 계층 : AddLayer class



: 두개의 MulLayer 필요

```
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y
        return out

    def backward(self, dout):
        dx = dout * self.y # x와 y를 바꾼다.
        dy = dout * self.x
        return dx, dy

class AddLayer:
    def __init__(self):
        pass

    def forward(self, x, y):
        out = x + y
        return out

    def backward(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

```
apple = 100
apple_num = 2
tax = 1.1

mul_apple_layer = MulLayer() #첫 번째 MulLayer
mul_tax_layer = MulLayer() #두 번째 MulLayer

# forward
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)
print(" price: ", int(price))

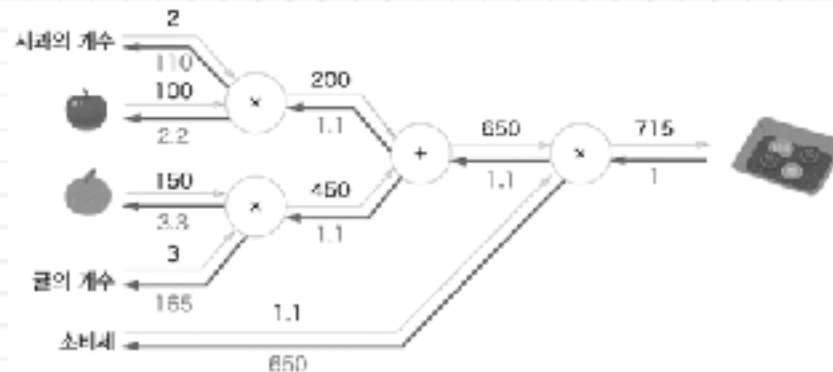
# backward
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dTax:", dtax)
```

# Neural Network : Backpropagation

## Backpropagation : 단순한 계층 구현(사과&오렌지 쇼핑)

- 곱셈 계층 : MulLayer class
- 덧셈 계층 : AddLayer class



: 세 개의 MulLayer  
: 한 개의 AddLayer

```
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1
```

```
# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()
```

```
# forward
apple_price = mul_apple_layer.forward(apple, apple_num) # (1)
orange_price = mul_orange_layer.forward(orange, orange_num) # (2)
all_price = add_apple_orange_layer.forward(apple_price, orange_price) # (3)
price = mul_tax_layer.forward(all_price, tax) # (4)
print("price:", int(price))
```

```
# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice) # (4)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price) # (3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price) # (2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price) # (1)

print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dOrange:", dorange)
print("dOrange_num:", int(dorange_num))
print("dTax:", dtax)
```

# Neural Network : Backpropagation

## Backpropagation 실제 구현 : Activation Function

- ReLU 계층  $y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$  : 미분  $\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}$



: ReLU 계층의 Computational Graph

- Sigmoid 계층  $y = \frac{1}{1 + \exp(-x)}$  : 미분  $\frac{\partial y}{\partial x} = y(1 - y)$

```
class Relu:
    def __init__(self):
        self.mask = None
```

```
    def forward(self, x):
        self.mask = (x <= 0)
        print(self.mask)
        out = x.copy()
        out[self.mask] = 0
```

```
    return out
```

```
    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout
```

```
    return dx
```

< mask >

- True/False로 된 numpy 배열을 유지

```
x = np.array([1, 2, 3, -5, 10])
mask = (x <= 0)
x[mask] = 100
print(mask)
print(x)
```

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

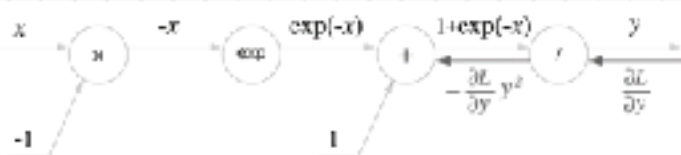
        # dy/dx = y(1-y)

    return dx
```

# Neural Network : Backpropagation

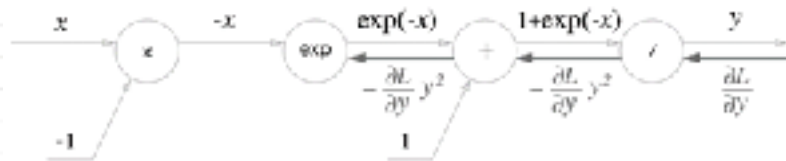
## Backpropagation 실제 구현 : Activation Function

- Sigmoid 계층  $y = \frac{1}{1 + \exp(-x)}$  : 미분  $\Rightarrow \frac{\partial L}{\partial y} y(1-y)$



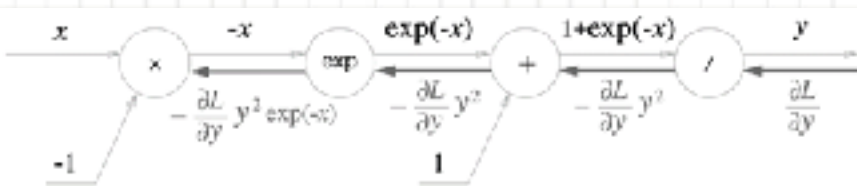
Computational Graph(forward:순전파)

1 단계 : '/' node 미분 즉,  $y = \frac{1}{x}$  을 미분하면  $\Rightarrow \frac{\partial y}{\partial x} = -\frac{1}{x^2} = -y^2$



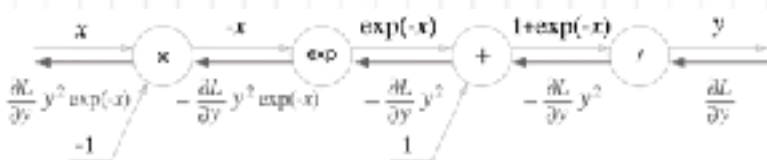
Sigmoid layer의 Computational Graph

2 단계 : '+' node는 여과 없이 하류로 보낸다



$$\begin{aligned} \frac{\partial L}{\partial y} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1 + \exp(-x))^2} \exp(-x) \\ &= \frac{\partial L}{\partial y} \frac{1}{1 + \exp(-x)} \frac{\exp(-x)}{1 + \exp(-x)} \\ &= \frac{\partial L}{\partial y} y(1-y) \end{aligned}$$

3 단계 : 'exp' node 미분 즉,  $y = \exp(x)$  을 미분하면  $\Rightarrow \frac{\partial y}{\partial x} = \exp(x)$



4 단계 : 'x' node는 forward 때의 값을 "서로 바꿔" 곱한다 : -1을 곱한다

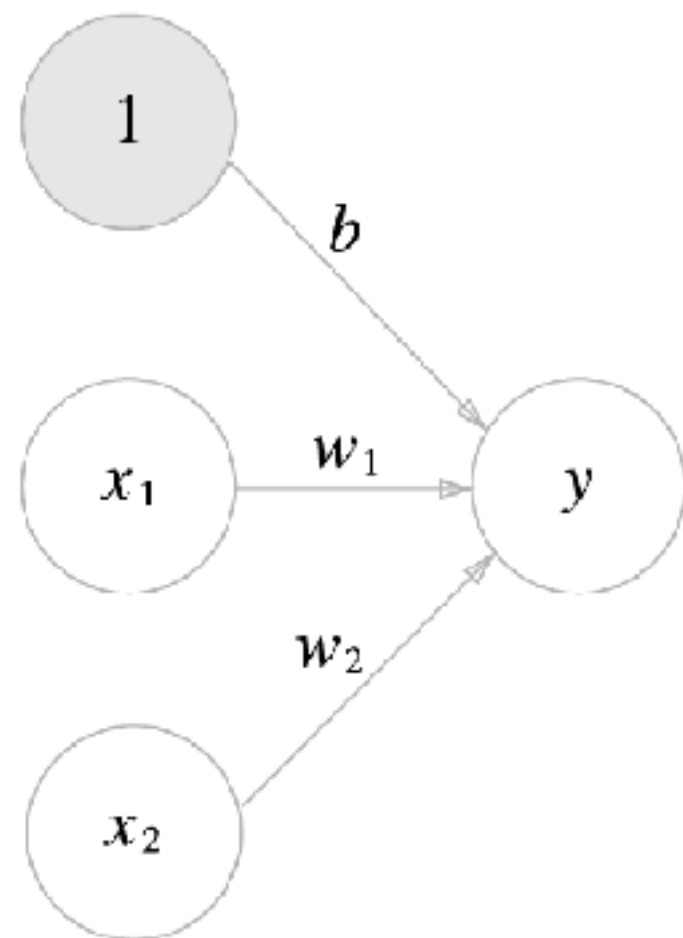


05-1.

다시 보기

What is the relative number of presentations that will target the audience as a logical Expression unnecessary information is released or increased alignment will fail.

## 단층 퍼셉트론

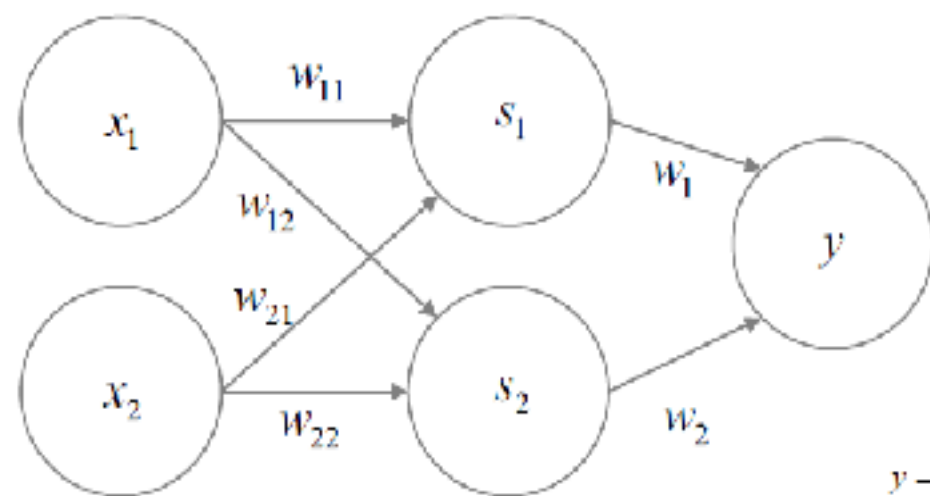


$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

$$y = Wx + b = [w_1 \quad w_2 \quad b] \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

분류문제를 행렬연산으로 해결가능(가중치/편향값을 알고 있다면)

# 다층 퍼셉트론



$$y = \begin{cases} 0 & w_1(w_{11}x_1 + w_{21}x_2) + w_2(w_{12}x_1 + w_{22}x_2) \leq 0 \\ 1 & w_1(w_{11}x_1 + w_{21}x_2) + w_2(w_{12}x_1 + w_{22}x_2) > 0 \end{cases}$$

$$y = [w_1 \ w_2] \begin{bmatrix} s_1 \\ s_2 \end{bmatrix}, S = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \rightarrow y = [w_1 \ w_2] \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

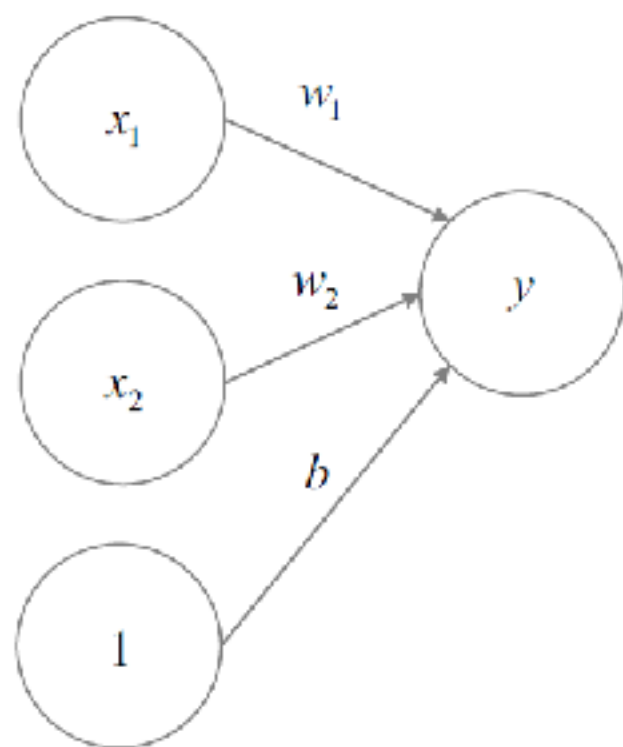
$$y = W_{s \rightarrow y} W_{x \rightarrow s} x$$

층(layer)간 연산은 단순한 행렬연산( $y = Wx + b$ )로 계산 가능

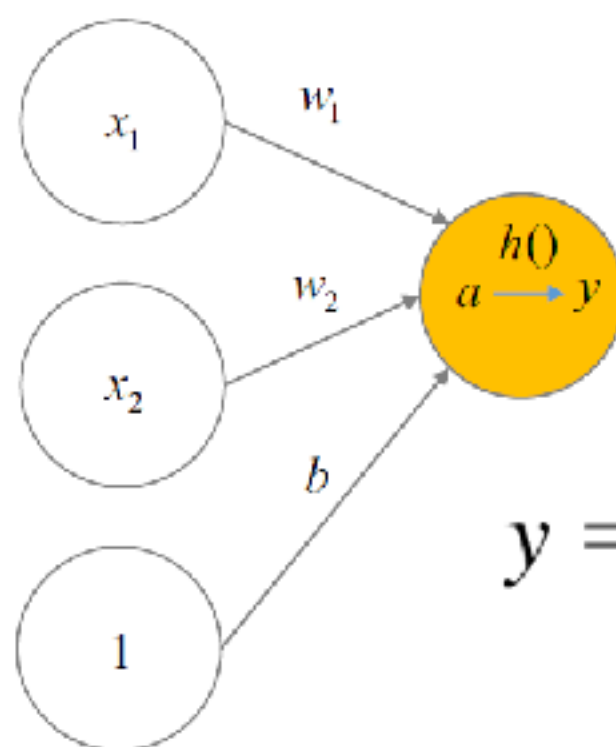
복잡한 분류 문제도 행렬연산으로 가능(가중치/편향값을 알고 있다면)

# 신경망(neural network)

다층 퍼셉트론 & (활성화함수 or 소프트맥스)



$$y = \begin{cases} 0 & b + w_1x_1 + w_2x_2 \leq 0 \\ 1 & b + w_1x_1 + w_2x_2 > 0 \end{cases}$$



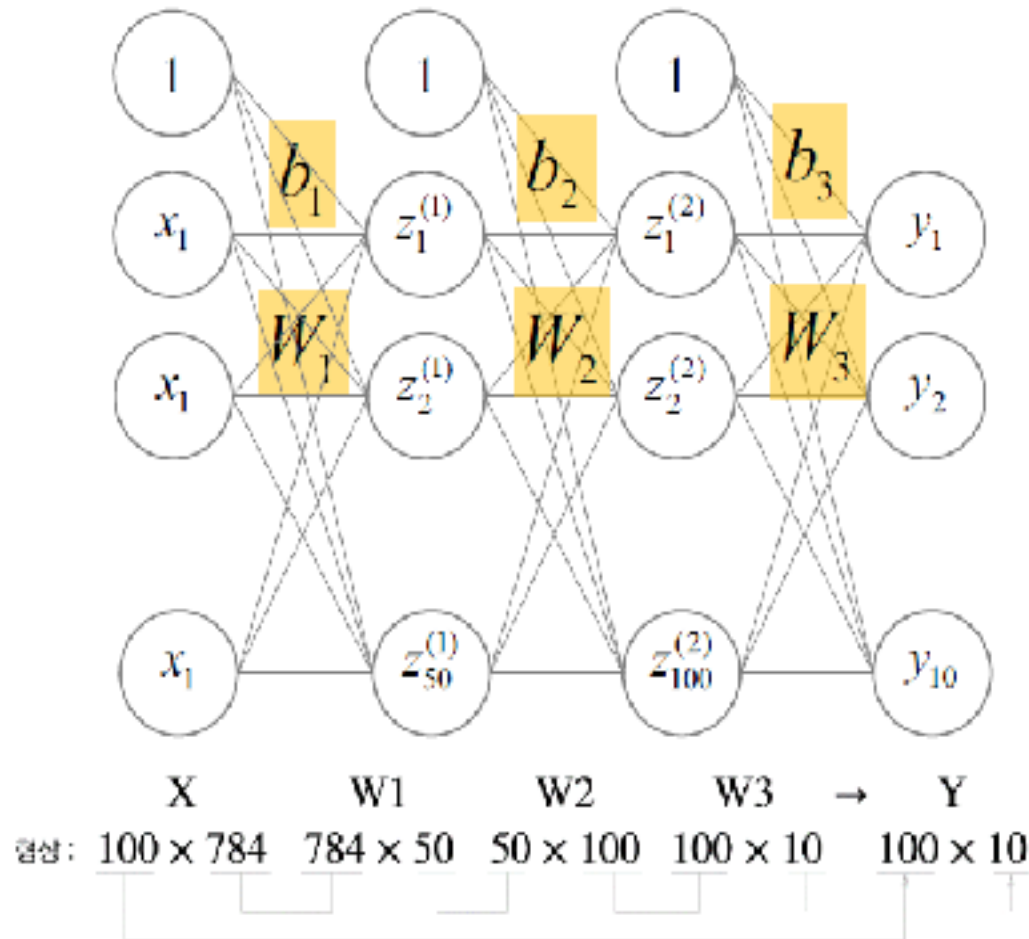
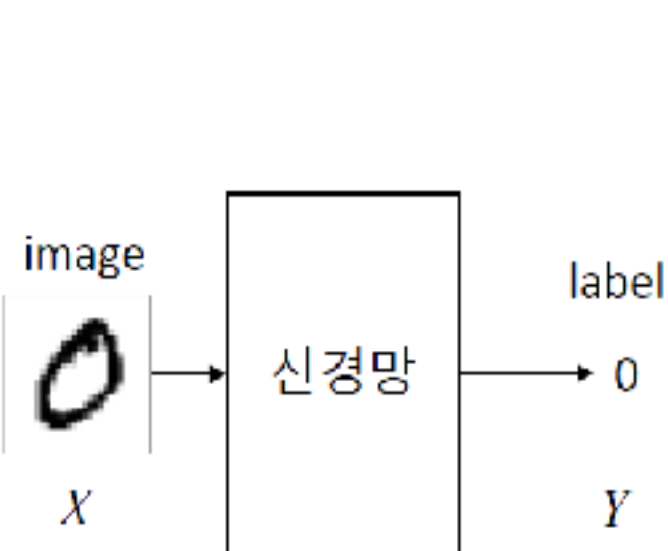
$$y = h(a)$$

$$y = \begin{cases} 0 & h(b + w_1x_1 + w_2x_2) \leq 0 \\ 1 & h(b + w_1x_1 + w_2x_2) > 0 \end{cases}$$

One layer!

Sigmoid / ReLU 모두 연속함수(미분가능)

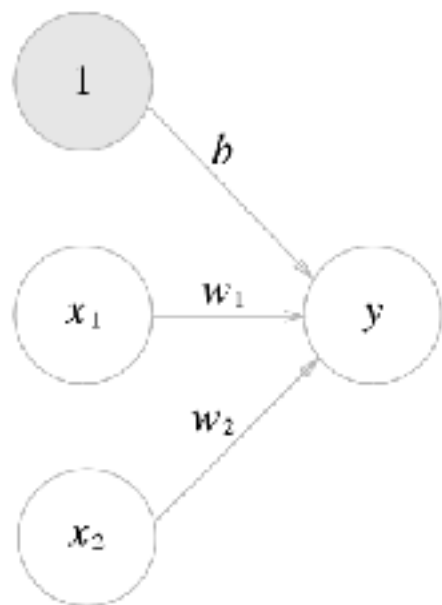
# 신경망(neural network) MNIST db case (숫자 이미지 입력 > 숫자 분류)



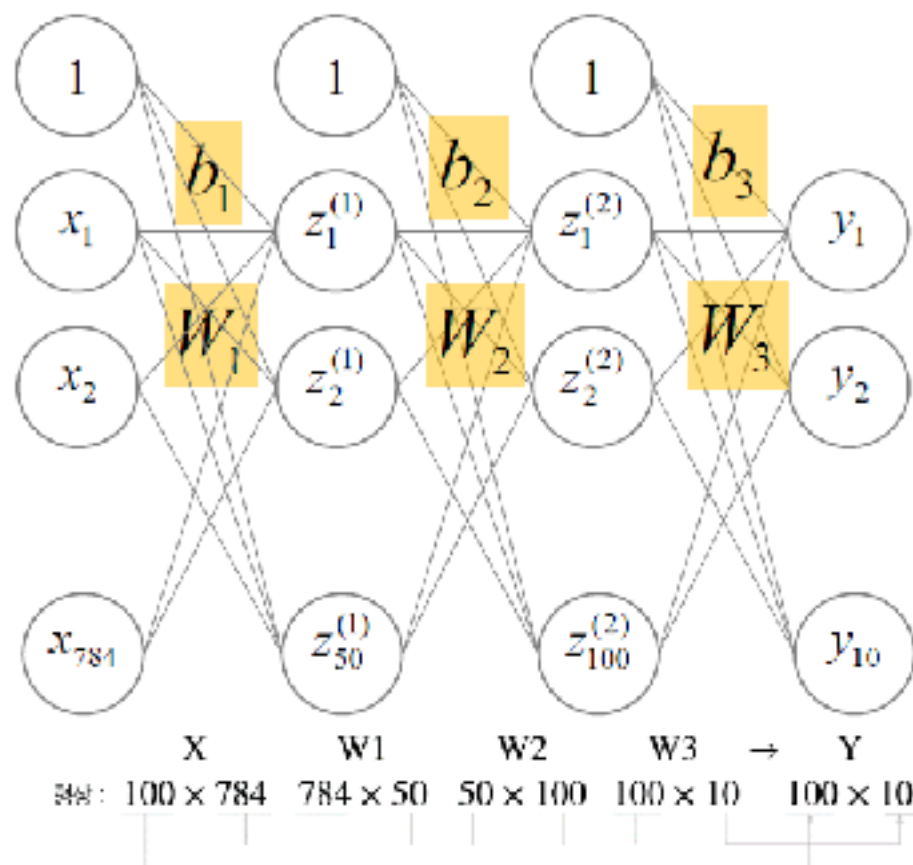
입력과 출력에 따라 신경망을 설계하면 분류기가 된다.  
(여전히, 가중치(w)/편향치(b)/적합한 레이어/노드수를 안다면)

# 데이터 기반 학습

- 최적화 목표= 손실 함수를 정의하고 최소가 되는 방향
- 매개변수를 바꾸는 방향=손실 함수의 매개변수 미분 방향



versus

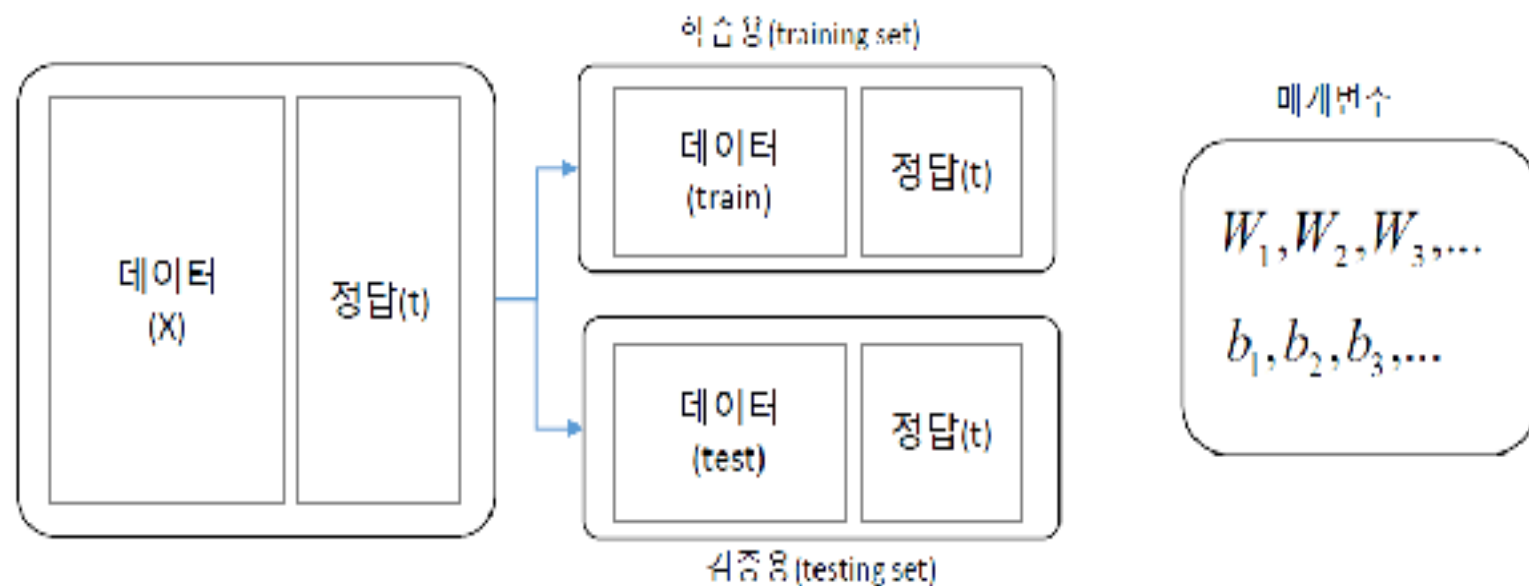


3개

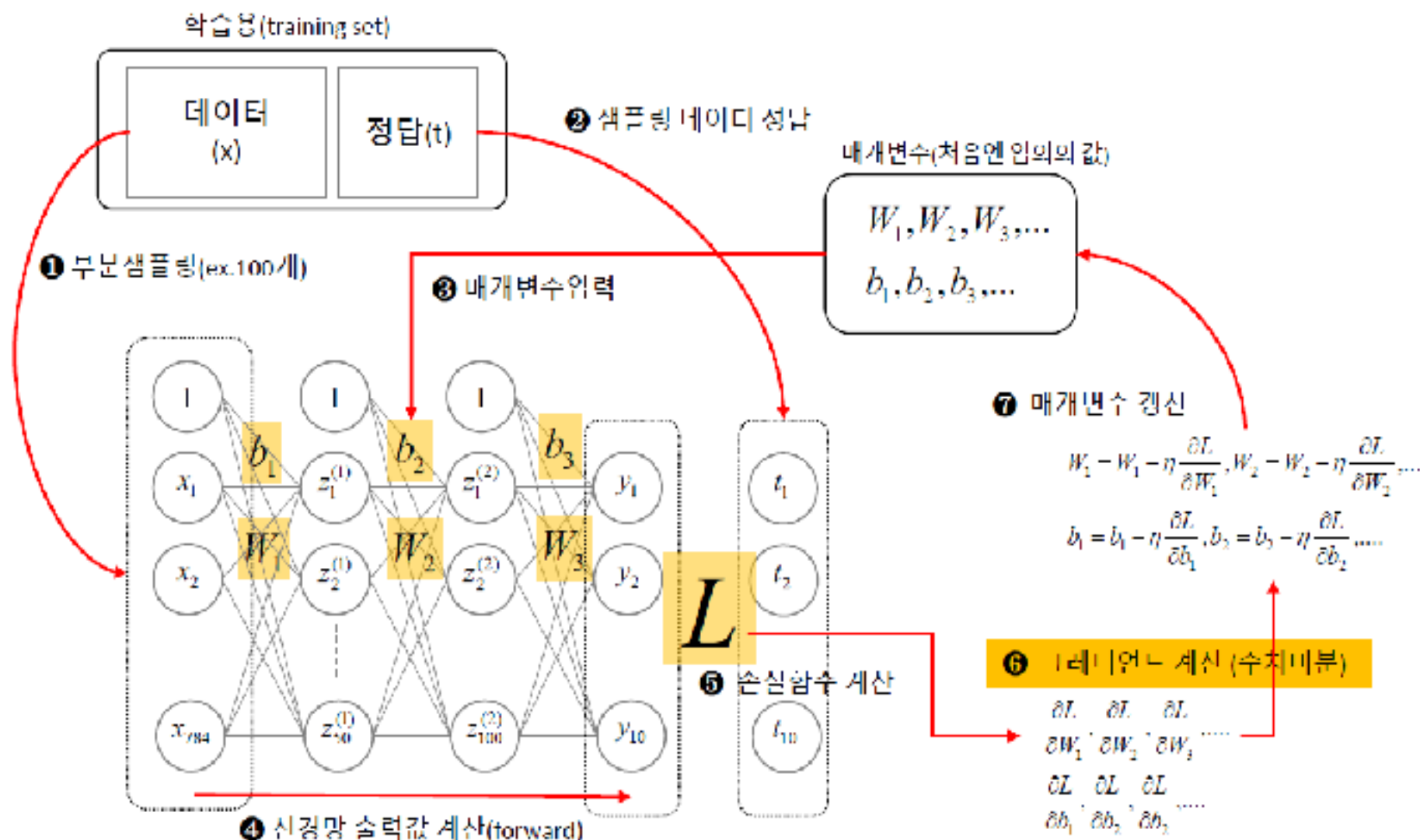
45,360개

- 데이터 기반으로 매개변수( $w, b$  등)을 알아내는 작업 = 학습
- 기본 아이디어 = 프로그램으로 매개변수들을 움직여 최선의 파라미터 값 정하기

## 데이터 기반 학습 알고리즘



# 데이터 기반 학습 알고리즘 + 수치미분



①~⑦ 과정 반복



# 데이터 기반 학습 알고리즘 + 오류역전파

학습용 (training set)

데이터  
(x)

정답(t)

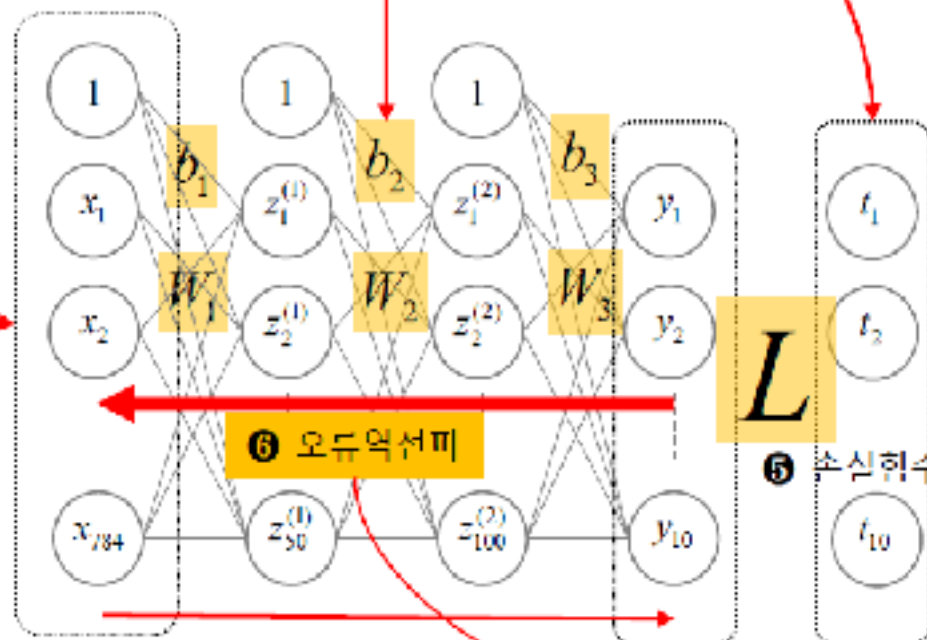
② 샘플링 데이터 정답

매개변수(처음엔 임의의 값)

$W_1, W_2, W_3, \dots$   
 $b_1, b_2, b_3, \dots$

① 부분샘플링(ex.100개)

③ 매개변수 입력



⑦ 매개변수 갱신

$$W_1 - W_1 \eta \frac{\partial L}{\partial W_1}, W_2 - W_2 \eta \frac{\partial L}{\partial W_2}, \dots$$

$$b_1 - b_1 \eta \frac{\partial L}{\partial b_1}, b_2 - b_2 \eta \frac{\partial L}{\partial b_2}, \dots$$

⑤ 손실함수 계산

④ 신경망 출력값 계산(forward)

⑥ 오류역전파

①~⑦ 과정 반복