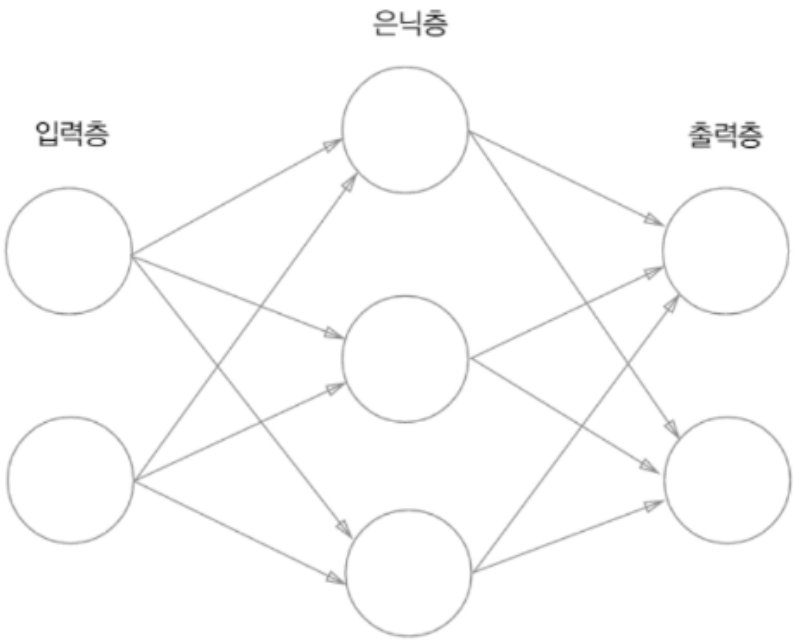
Chapter 3. 신경망

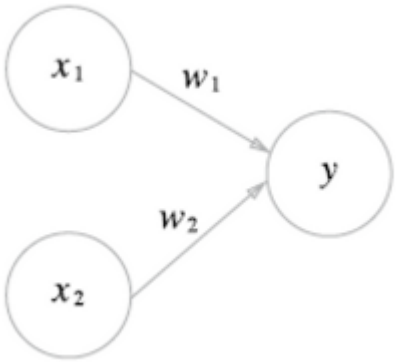
- 가중치 매개변수의 적절한 값을 데이터로부터 자동으로 학습하는 능력

3-1. 퍼셉트론에서 신경망

- 신경망을 그림으로 나타내는 아래와 같고, 가장 왼쪽 줄이 **입력층,**
  맨 오른쪽 줄 **출력층**, 중간 줄 **은닉층**



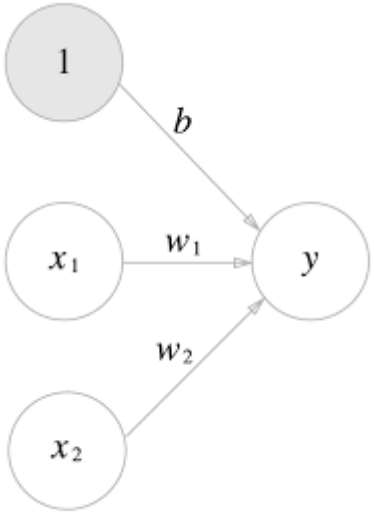
- x1과 x2 라는 두 신호를 받아 y를 출력하는 퍼셉트론



$$y = \begin{cases} 0 & (b + w_1 x_1 + w_2 x_2 \leq 0) \\ 1 & (b + w_1 x_1 + w_2 x_2 > 0) \end{cases}$$

b는 **편향**을 나타내는 매개변수, 뉴런이 얼마나 쉽게 활성화되느냐를 제어

w1, w2는 각 신호의 **가중치**를 나타내는 매개변수로, 각 신호의 영향력을 제어



[편향을 명시한 퍼셉트론]

- 가중치가 b이고 입력이 1인 뉴런이 추가됨
- 퍼셉트론의 동작은 x1, x2, 1이라는 3개의 신호가 뉴런에 입력도어, 각 신호에 가중치를 곱한 후 다음 뉴런에 전달됨
- 뉴런에서 신호들의 값을 해 합이 0을 넘으면 1을 출력하고, 그렇지않으면 0을 출력함

- 조건 분기의 동작(0을 넘으면 1을 출력하고, 그렇지 않으면 0을 출력)을 하나의 함수인 h(x)로 나타내고, 입력이 0을 넘으면 1을 돌려주고 그렇지 않으면 0을 돌려줌

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

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

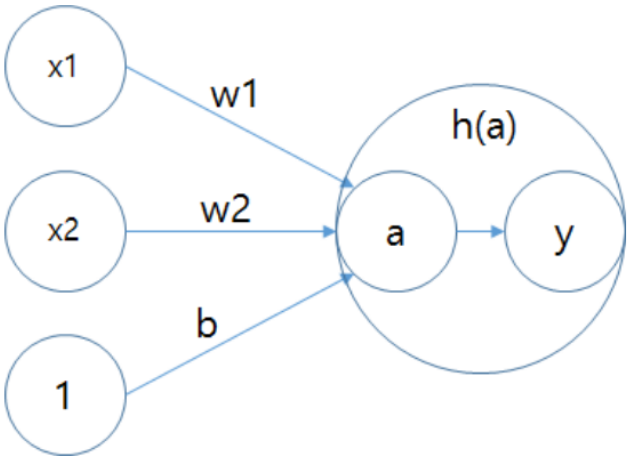
활성화 함수의 등장

- h(x) 함수는 입력 신호의 총합을 출력 신호로 변환하는 함수를 **활성화 함수(activation function)**
- 활성화 함수는 입력 신호의 총합이 활성화를 일으키는지 정하는 역할을 함
- 가중치가 곱해진 입력 신호의 총합을 계산하고, 그 합을 활성화 함수에 입력해 결과를 내는 2단계로 처리됨

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

가중치가 달린 입력 신호와 편향의 총합을 계산(a)

a를 함수 h()에 넣어 y를 출력하는 흐름



- 가중치 신호를 조합하면 a라는 노드가 되고, 활성화 함수 h()를 통과하여 y라는 노드로 변환 (a와 y의 원은 노드)

- **단순 퍼셉트론**은 단층 네트워크에서 계단 함수(임계값을 경계로 출력이 바뀌는 함수)를 활성화 함수로 사용한 모델을 가리키고, **다층 퍼셉트론**은 신경망(여러 층으로 구성되고 시그모이드 함수 등의 매끈한 활성화 함수를 사용하는 네트워크)를 가리킴

3-2. 활성화 함수

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

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

- 위의 활성화 함수는 임계값을 경계로 출력이 바뀜 ⇒ `계단 함수(setp function`

  ⇒ 퍼셉트론에서는 활성화 함수로 계단 함수를 이용한다

`시그모이드 함수` (sigmoid function)

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

exp(-x) 는 e^(-x) 를 뜻하고, e는 자연상수로 2.7172..의 값을 갖는 실수

시그모이드 함수는 '함수'로 입력을 주면 출력을 돌려주는 변환기

시그모이드 함수에 1.0과 2.0을 입력하면 h(1.0) = 0.731, h(2.0) = 0.880 특정 값 출력

- 신경망에서 활성화 함수로 시그모이드 함수를 사용하여 신호를 변환하고, 변환된 신호를 다음 뉴런에 전달함
- 퍼셉트론과 신경망의 주된 차이는 '활성화 함수' 뿐

계단 함수 구현하기

```
def step_function(x):
    if x>0:
        return 1
    else:
        return 0
```

- 해당 함수는 인수로 실수(부동소수점)만 받아들임
step_function(3.0)은 가능하지만, 넘파이 배열을 인수로 넣을 수 없음
function(np.array([1.0, 2.0])) 이 안되, 넘파이 배열을 지원하도록 수정

```
def setp_function(x):
    y = x > 0
    return y.astype(np.int)
```

```
import numpy as np
x = np.array([-1.0, 1.0, 2.0])
print(x)

y = x>0
print(y)
```

```
[-1.  1.  2.]
[False  True  True]
```

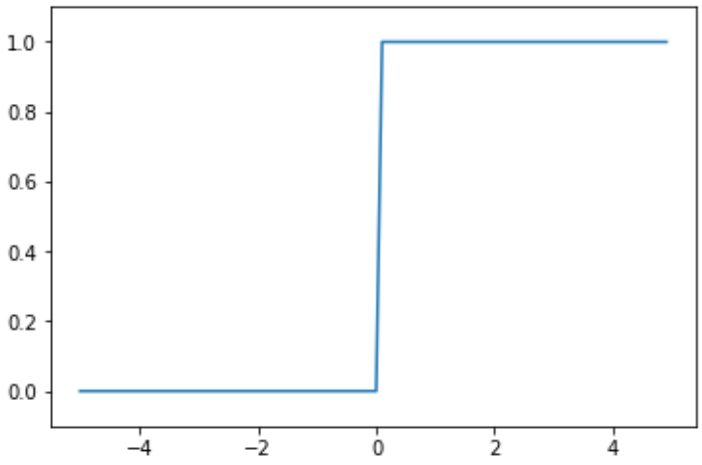
```
y = y.astype(np.int)
print(y)
```

```
[0 1 1]
```

- 파이썬에서는 bool을 int로 변환하면 True는 1로 False는 0으로 변환됨

계단 함수의 그래프

```
import numpy as np
import matplotlib.pylab 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)  # y축의 범위
plt.show()
```



- np.arange(-5.0, 5.0, 0.1)은 -5.0에서 5.0 전까지 0.1 간격의 넘파이 배열 생성
즉, [-5.0, -4.9, ... 4.9]
- 계단 함수는 0을 경계로 출력이 0에서 1(또는 1에서 0으로) 바뀜

시그모이드 함수 구현

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

```
x = np.array([-1.0, 1.0, 2.0])
print(sigmoid(x))
```

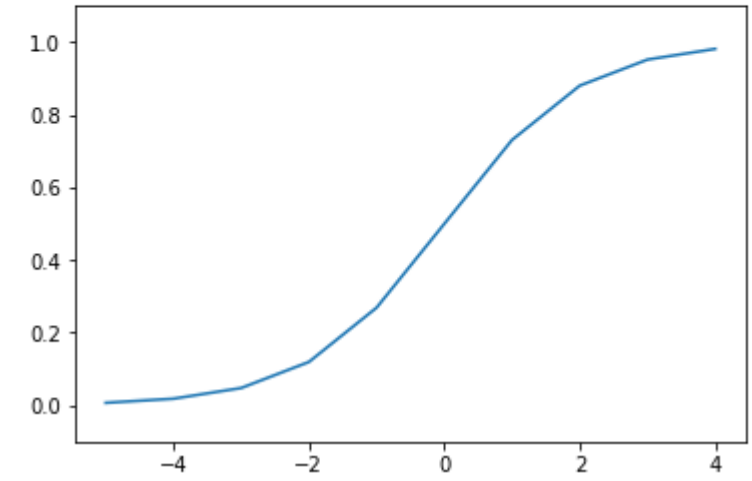
```
[0.26894142 0.73105858 0.88079708]
```

- 넘파이의 브로드캐스트 기능으로 넘파이 배열과 스칼라값의 연산을 넘파이 배열의 원소 각각과 스칼라값의 연산으로 바꿔 수행

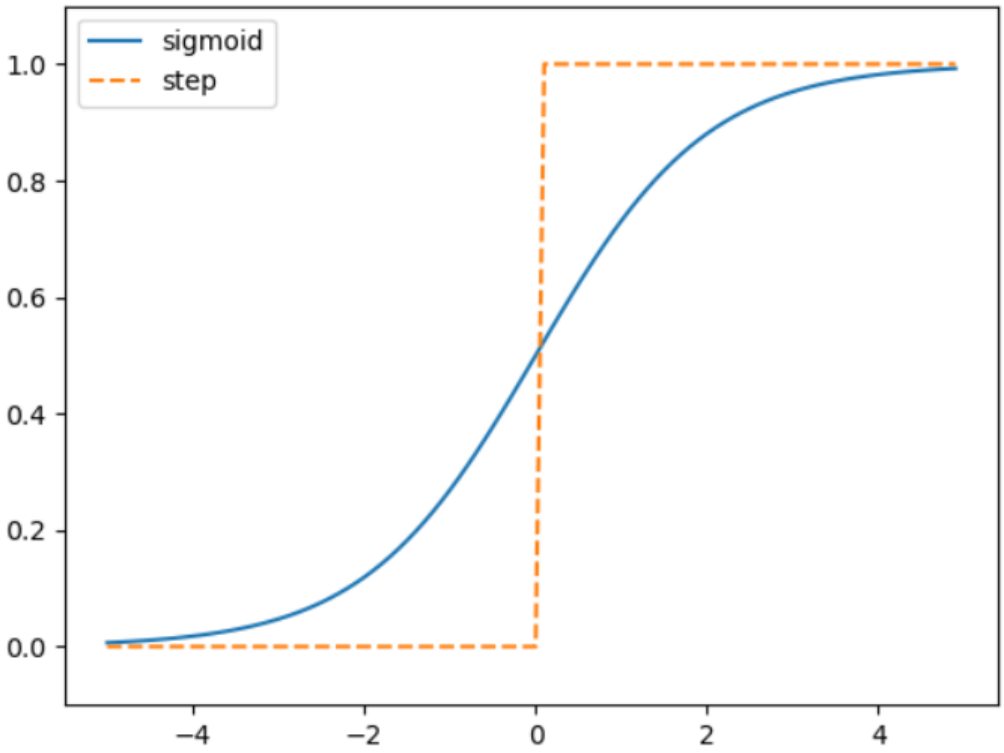
```
# 넘파이의 브로드캐스트
t = np.array([1.0, 2.0, 3.0])
1.0 + t
print(1.0+t)
```

```
[2. 3. 4.]
```

```python
x = np.arange(-5.0, 5.0, 1.0)
y = sigmoid(x)

plt.plot(x,y)
plt.ylim(-0.1, 1.1) # y축 범위 지정
plt.show()
```



시그모이드 함수와 계단 함수 비교



- 시그모이드 함수는 부드러운 곡선이며, 입력에 따라 출력이 **연속적으로 변화**

- 계단 함수는 **0**을 경계로 출력이 바뀜
- 계단 함수가 0과 1 중 하나의 값만 돌려주는 반면, 시그모이드 함수는 실수(0.731, 0.880 등)을 돌려줌
- 퍼셉트론에서는 뉴런 사이에 0 혹은 1이 흘렀다면, 신경망에서는 연속적인 실수가 흐름
- 두 함수는 모두 입력이 작을 때의 출력은 0에 가깝고(혹은 0이고), 입력이 커지면 출력이 1에 가까워지는(혹은 1이 되는) 구조
- 계단 함수와 시그모이드 함수는 입력이 중요하면 큰 값을 출력하고, 입력이 중용지 않으면 작은 값을 출력함
- 입력이 아무리 작거나 커도 출력은 0에서 1사이임

비선형 함수

- 계단 함수와 시그모이드 함수의 공통점은 **비선형 함수**
- 시그모이드 함수는 곡선, 계단 함수는 계단처럼 구부러진 직선으로 나타나며 비선형 함수로 분류

> **함수** : 어떤 값을 입력하면 그에 따른 값을 돌려주는 '변환기'
>
> 이 함수에 무엇을 입력했을 때 출력이 입력의 상수배 만큼 변함 → **선형 함수**
>
> **f(x) = a*x + b (곧은 1개의 직선)**
>
> **비선형 함수**는 '선형이 아닌' 함수로, 1개로는 그릴 수 없는 함수

- 신경망에서는 활성화 함수로 비선형 함수를 사용함
- 선형 함수는 층을 아무리 깊게 해도 '은닉층이 없는 네트워크'와 같은 기능

ReLU 함수

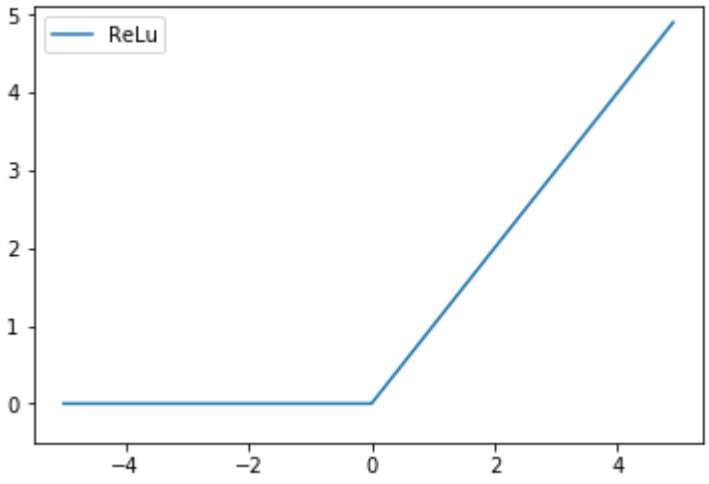
- 시그모이드 함수는 신경망 분야에서 오래전부터 이용해왔으나, 최근에는 **ReLU(Rectified Linear Unit, 렐루)** 함수를 주로 이용함

- ReLU는 입력이 0을 넘으면 입력을 그대로 출력하고, 0 이하면 0을 출력함

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

ReLu 함수 구현

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def ReLu(x):
5      return np.maximum(0,x)
6
7  x = np.arange(-5.0, 5.0, 0.1)
8  y = ReLu(x)
9
10 plt.plot(x,y, label='ReLu')
11 plt.ylim(-0.5, 5.1)
12 plt.legend()
13 plt.show()
```



3-3. 다차원 배열의 계산

다차원 배열

- '숫자의 집합', 숫자가 한 줄로 늘어선 것이나 직사각형으로 늘어놓은 것, 3차원으로 늘어놓은 것이나 N차원으로 나열하는 것

다차원 배열

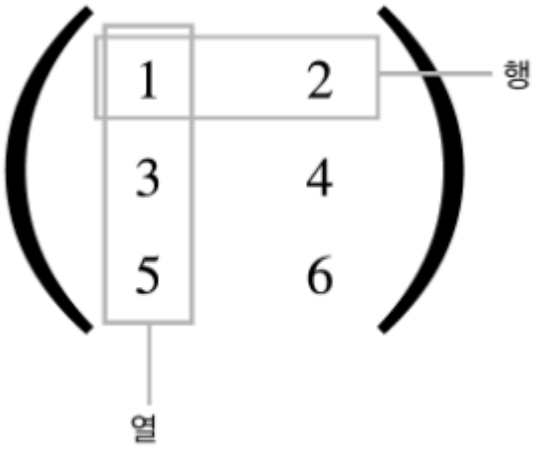
```python
import numpy as np


A = np.array([1,2,3,4])
print(A)
print(np.ndim(A))
print(A.shape)
print(A.shape[0])
```

```
[1 2 3 4]
1
(4,)
4
```

```python
B = np.array([[1,2], [3,4], [5,6]])
print(B)
print(np.ndim(B))
print(B.shape)
```

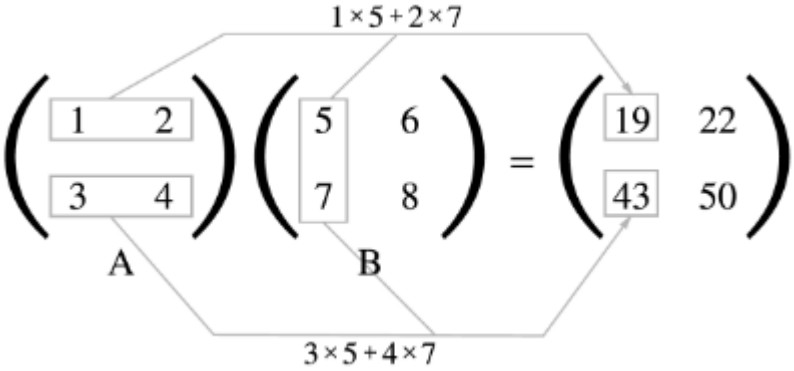
```
[[1 2]
 [3 4]
 [5 6]]
2
(3, 2)
```

- '3x2 배열' 인 B로 처음 차원에는 원소가 3개, 다음 차원에는 원솩 2개 있음
- 처음 차원은 0번째 차원, 다음 차원은 1번째 차원으로 대응
- 2차원 배열은 행렬(matrix)라고 부르고, 배열의 가로 방향 **행**(row), 세로 방향 **열(column)**



2차원 배열(행렬)의 행(가로)과 열(세로)

행렬의 곱



왼쪽 행렬의 행 (가로) 와 오른쪽 행렬의 열(세로)을 원소별로 곱하고 그 값을 더해서 계산

계산 결과는 새로운 다차원 배열의 원소가 됨

행렬의 곱

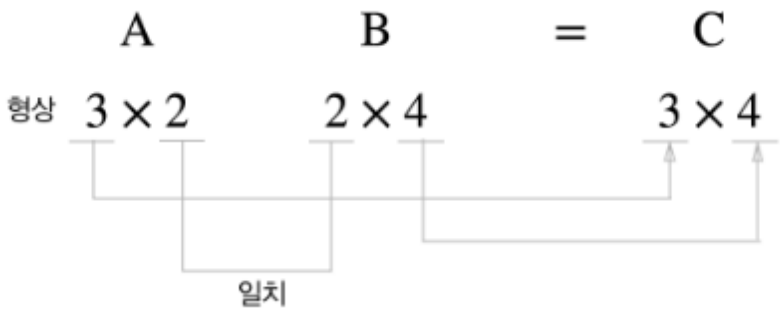
```python
A = np.array([[1,2], [3,4]])
print(A)
print(A.shape)

B = np.array([[5,6], [7,8]])
print(B)
print(B.shape)

print(np.dot(A,B))
```

```
[[1 2]
 [3 4]]
(2, 2)
[[5 6]
 [7 8]]
(2, 2)
[[19 22]
 [43 50]]
```

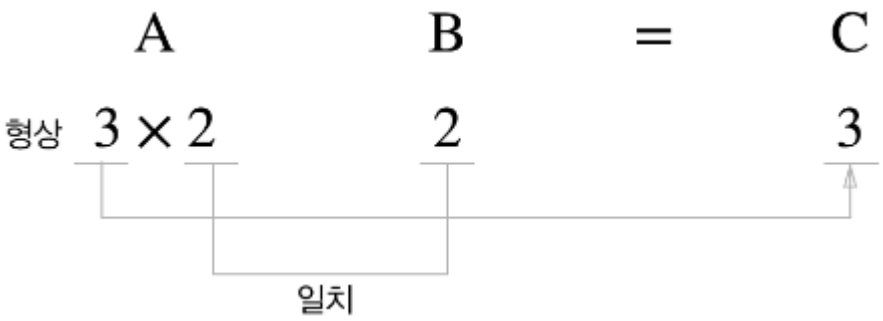
```
:  1 C = np.array([[1,2], [3,4]])
   2 print(C.shape)
   3 print(A.shape)
   4
   5 print(np.dot(A,C))
```

```
(2, 2)
(2, 3)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-44-5ace185acd1c> in <module>()
      3 print(A.shape)
      4
----> 5 print(np.dot(A,C))

ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

- 행렬 A의 1번째 차원(dim 1)과 행렬 C의 0번째 차원(dim 0)의 원소 수가 다르다는 오류 (차원의 인덱스는 0부터 시작)
- 다차원 배열을 곱하려면 두 행렬의 대응하는 차원의 원소 수를 일치시켜야 함



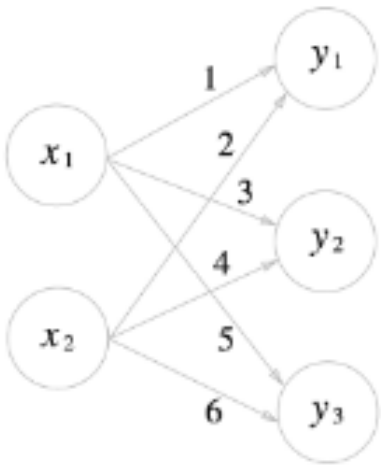
- 행렬 A와 B의 대응하는 차원의 원소수가 같아야 함




```
: 1 A = np.array([[1,2], [3,4], [5,6]])
  2 print(A.shape)
  3 B = np.array([7,8])
  4 print(B.shape)
  5
  6 print(np.dot(A,B))
```

```
(3, 2)
(2,)
[23 53 83]
```

신경망에서의 행렬 곱



$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

X W = Y

2 2 × 3 3

일치

```python
X = np.array([1,2])
print(X.shape)

W = np.array([[1,3,5], [2,4,6]])
print(W)

y = np.dot(X,W)
print(y)
```

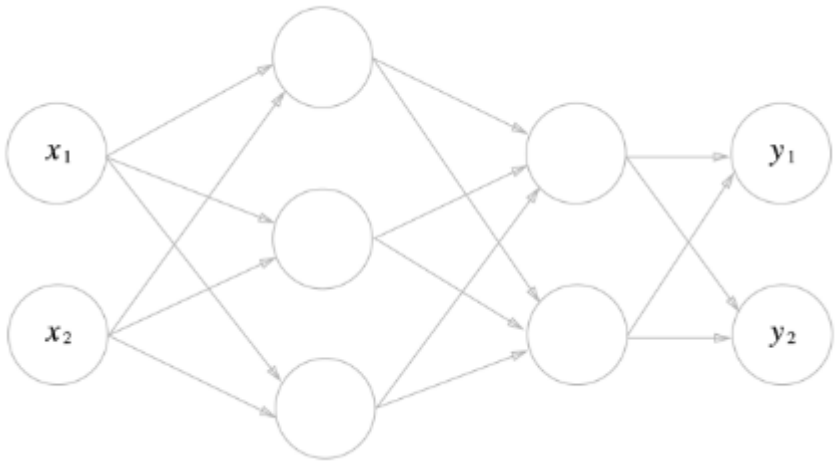
```
(2,)
[[1 3 5]
 [2 4 6]]
[ 5 11 17]
```

- 다차원 배열의 스칼라곱을 구해주는 np.dot 함수를 이용하면 결과 Y를 계산

3-4. 3층 신경망 구현하기

- 3층 신경망 :

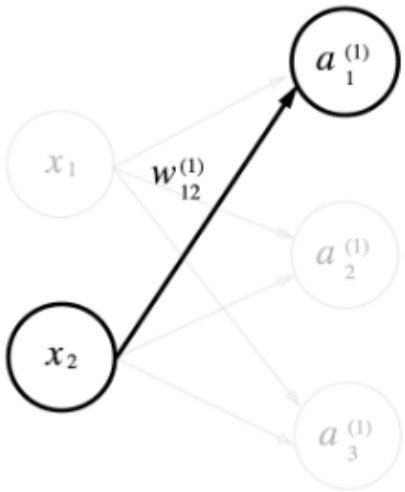
  0층 - 입력층 (2개), 1층 - 첫 번째 은닉층 (3개), 2층 - 두 번째 은닉층 (2개), 3층 - 출력층 (2개)

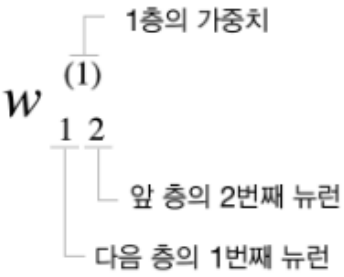


표기법

- 신경망에서의 계산을 행렬 계산으로 정리할 수 있음

아래의 그림은 입력층의 뉴런 x_2에서 다음 층의 뉴런 $a_1^{(1)}$으로 향하는 선 위에 가중치를 표시하고 있다.



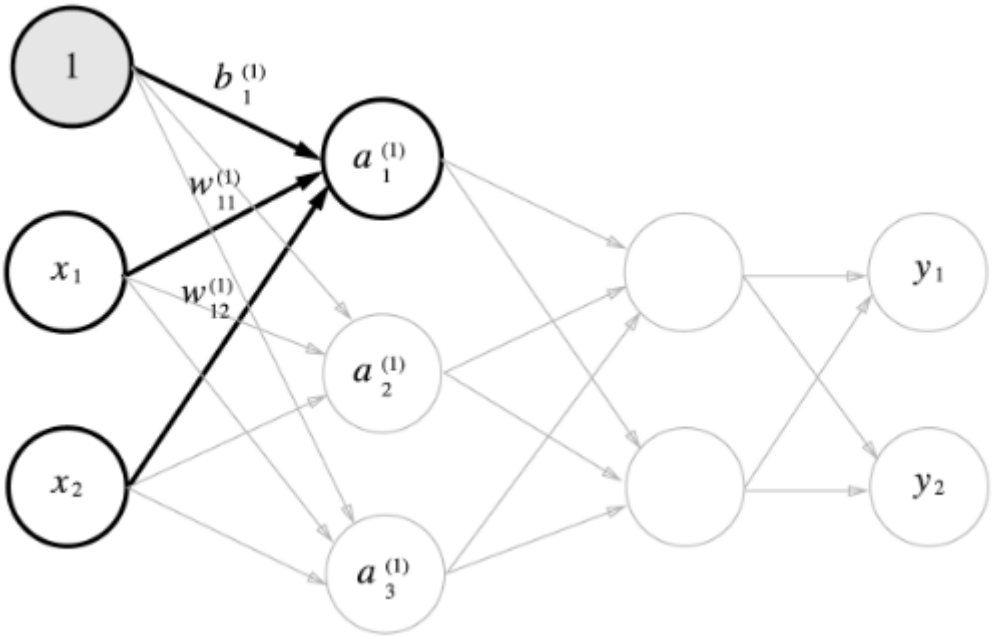


- 가중치와 은닉층 뉴런의 오른쪽 위에 '(1)' 은 1층의 가중치, 1층의 뉴런임을 뜻하는 번호
- 가중치의 오른쪽 아래의 두 숫자는 차례로 **다음 층 뉴런**과 **앞 층 뉴런**의 인덱스 번호

  예) w12(1) : 앞 층의 2번째 뉴런(x2)에서 다음 층의 1번째 뉴런(a1(1)) 으로 향할 때의 가중치
- 가중치 오른쪽 아래의 인덱스 번호는 '다음 층 번호, 앞 층 번호' 순으로 적음

각 층의 신호 전달 구현

- 입력층에서 '1층의 첫 번째 뉴런' 으로 가는 신호



- 편향(bias) 뉴런 추가

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

a1(1) 은 가중치를 곱한 신호 두 개와 편향을 합함

행렬의 곱을 이용하면 1층의 '가중치 부분'을 간소화 가능

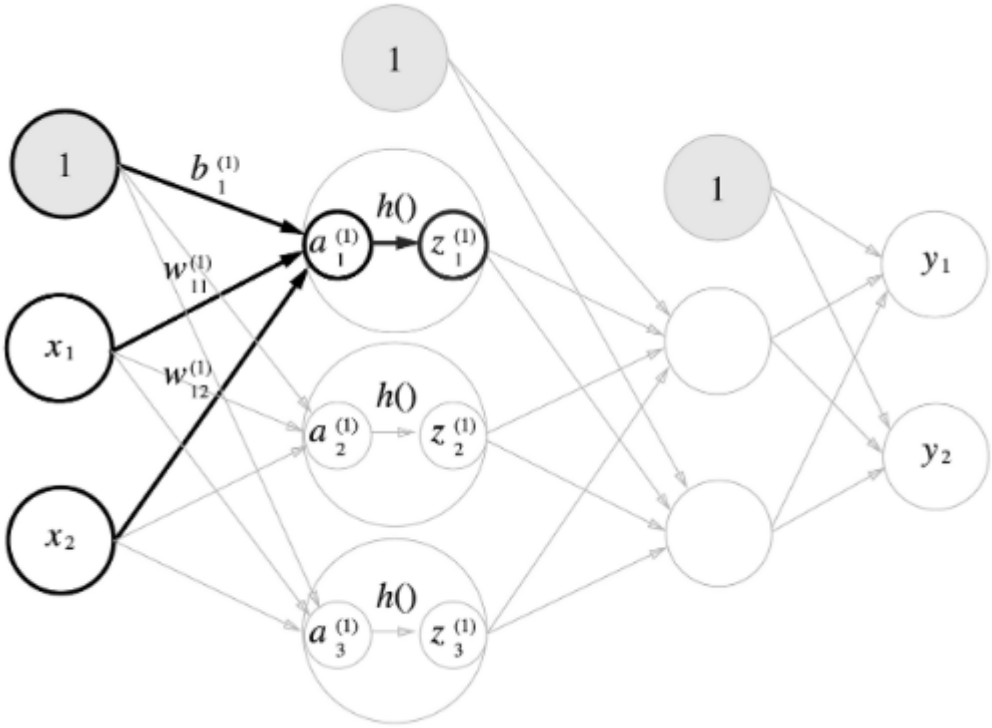
$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

$$A^{(1)} = \left(a_1^{(1)} a_2^{(1)} a_3^{(1)}\right)$$
$$X = (x_1 x_2)$$
$$B^{(1)} = \left(b_1^{(1)} b_2^{(1)} b_3^{(1)}\right)$$
$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

다차원 배열을 이용하여 3층 신경망 구현

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

print(W1.shape)
print(X.shape)
print(B1.shape)

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

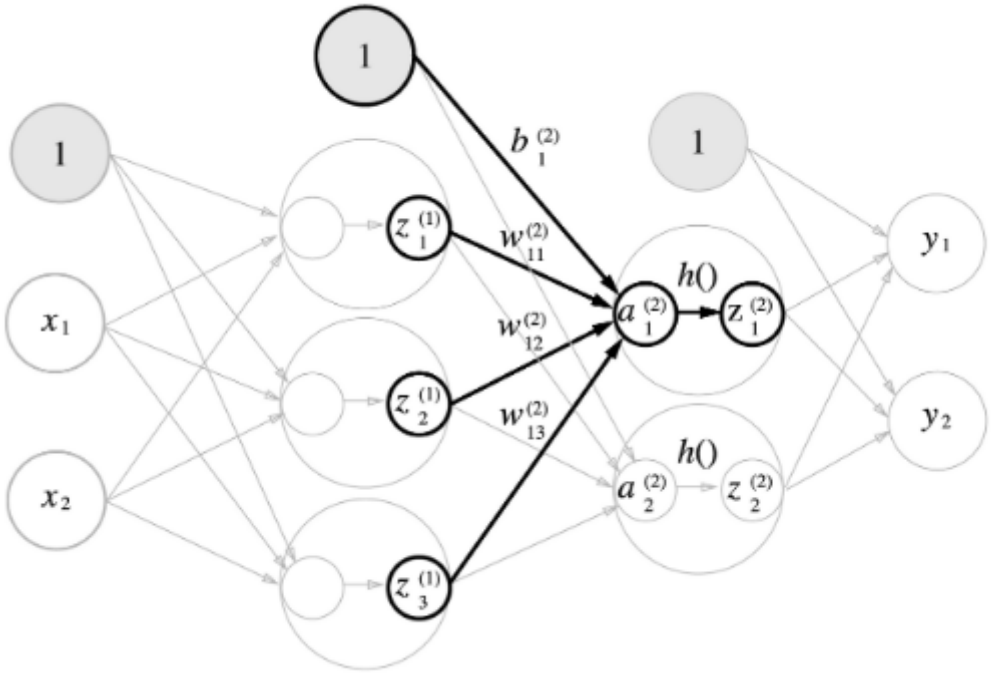
```
(2, 3)
(2,)
(3,)
[0.3 0.7 1.1]
```



[입력층에서 1층으로의 신호 전달]

```
:  1  Z1 = sigmoid(A1)
   2
   3  print(A1)
   4  print(Z1)
```

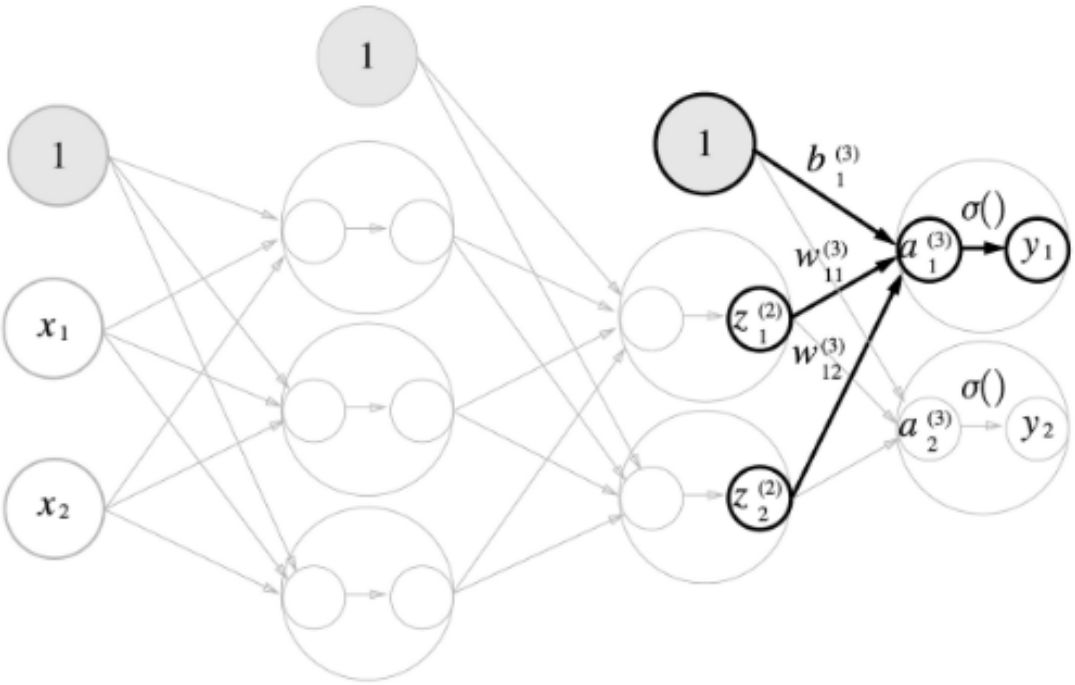
```
[0.3 0.7 1.1]
[0.57444252 0.66818777 0.75026011]
```



[1층에서 2층으로의 신호 전달]

```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

print(Z1.shape)
print(W2.shape)
print(B2.shape)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)

print(A2)
print(Z2)
```

```
(3,)
(3, 2)
(2,)
[0.51615984 1.21402696]
[0.62624937 0.7710107 ]
```



[2층에서 출력층으로의 신호 전달]

```python
def identity_function(x):
    return x

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 = identity_function(A3)  # 혹은 y = A3
```

- 항등 함수인 identity_function()을 정의, 출력층의 활성화 함수로 이용
- 항등 함수는 입력을 그대로 출력하는 함수
- 출력층의 활성화 함수를 σ()로 표시하여 은닉층의 활성화 함수 h()와 다르게 명시함

- 출력층의 활성화 함수는 풀고자 하는 문제의 성질에 맞게 정함
- 회귀에는 항등 함수, 2클래스 분류에는 시그모이드 함수, 다중 클래스 분류에는 소프트맥스 함수를 사용하는 것이 일반적

3층 신경망 구현 정리

```python
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 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 = identity_function(a3)

    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
print(y)
```

```
[0.31682708 0.69627909]
```

- init_network() 와 forward() 라는 함수 정의

- init_network() 함수는 가중치와 편향을 초기화 하고, 딕셔너리 변수인 network에 저장

- 딕셔너리 변수 network에는 각 층에 필요한 매개변수(가중치와 편향)을 저장

- forward() 함수는 입력 신호를 출력으로 변환하는 처리 과정을 구현

- 함수 이름이 forward 인 것은 시놓가 순방향(입력에서 출력 방향)으로 전달되는 순전파를 구현했기 때문임

3-5. 출력층 설계하기

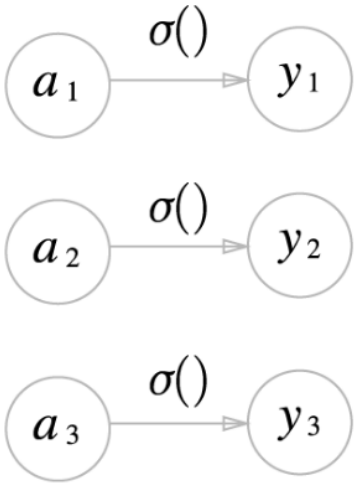
- 신경망은 분류와 회귀 모두에 이용할 수 있음
- 어떤 문제냐에 따라서 출력층에서 사용하는 활성화 함수가 달라짐

- 회귀에는 **항등 함수**, 분류에는 **소프트맥스 함수** 사용

> 기계학습 문제는 **분류(classification)** 와 **회귀(regression)** 으로 나뉨
> 분류는 데이터가 어느 클래스(class)에 속하는지에 대한 문제이며,
> 회귀는 입력 데이터에서 (연속적인) 수치를 예측하는 문제

`항등 함수와 소프트맥스 함수 구현`

- 항등 함수(identity function)는 입력을 그대로 출력함
- 입력과 출력이 항상 같다는 뜻의 항동
- 출력층에서 항등 함수를 이용하면 입력 신호가 그대로 출력 신호가 됨
- 항등 함수에 의한 변환은 은닉층에서 확성화 함수와 마찬가지로 화살표로 그림



- **분류** 에서 사용하는 `소프트맥스 함수(softmax function)` 식

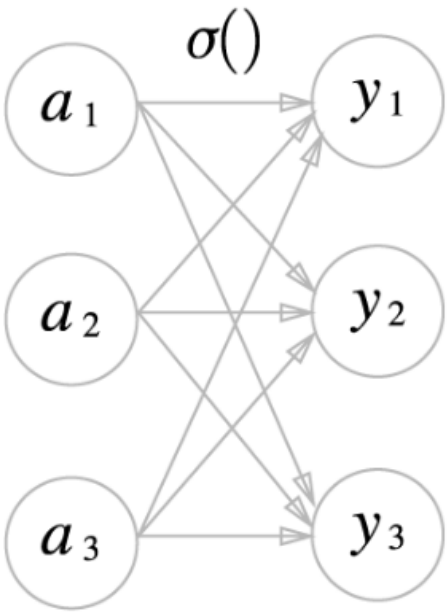
$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

exp(x) 는 e^x 를 뜻하는 지수 함수(exponential function) , e는 자연상수

n은 출력층의 뉴런 수, yk 는 k번째 출력

소프트맥스 함수의 분자는 입력 신호 ak의 지수 함수,

분모는 모든 입력 신호의 지수함수의 합



소프트맥스 함수 구현

```python
a = np.array([0.3, 2.9, 4.0])

exp_a = np.exp(a) # 지수함수
print(exp_a)

sum_exp_a = np.sum(exp_a)
print(sum_exp_a)

y = exp_a / sum_exp_a
print(y)
```

```
[ 1.34985881 18.17414537 54.59815003]
74.1221542101633
[0.01821127 0.24519181 0.73659691]
```

```python
def softmax(a):
    exp_a = np.exp(a)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y

print(softmax(a))
```

```
[0.01821127 0.24519181 0.73659691]
```

소프트맥스 함수 구현 시 주의점

- softmax() 함수 구현시, 컴퓨터로 계산 할때는 오버플로 문제가 발생할 수 있음
- 소프트맥스 함수는 지수 함수를 사용하는데, 지수 함수는 아주 큰 값이라, 이를 나눗셈 하면 수치가 '불안정' 해짐
- 컴퓨터는 수(number)를 4바이트나 7바이트와 같이 크기가 유한한 데이터로 다루기 때문에, 표현할 수 있는 수의 범위가 한정되어 너무 큰 값은 표현할 수 없다는 문제가 발생함

  ⇒ 오버 플로(overflow)

[소프트맥스 함수 개선 수식]

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

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

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

- C라는 임의의 정수를 분자와 분모 양쪽에 곱하고, C를 지수 함수 exp()에 옮겨 logC로 만듦

- logC를 C’ 라는 개솔운 기호로 바꿈

- 소프트맥스의 지수 함수를 계산할 때, 어떤 정수를 더하거나 빼도 결과는 달라지지 않음

- C’에 어떤 값을 대입해도 상관 없지만, 오버 플로를 막을 목적으로 입력 신호 중 최댓값을 이용하는 것이 일반적

softmax 함수 개선 수식

```
a = np.array([1010, 1000, 990])
print(np.exp(a)/np.sum(np.exp(a)))

c = np.max(a)
print(a-c)

print(np.exp(a-c)/np.sum(np.exp(a-c)))
```

```
[nan nan nan]
[  0 -10 -20]
[9.99954600e-01 4.53978686e-05 2.06106005e-09]
```

```
/home/yeoai/.local/lib/python2.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning: overflow encountered in exp

/home/yeoai/.local/lib/python2.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in divide
```

- 입력 신호 중 최댓값(c)을 빼주면 올바르게 계산할 수 있음

```
def softmax(a):
    c = np.max(a)
    exp_a = np.exp(a-c) # 오버플로 대책
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y

print(softmax(a))
```

```
[9.99954600e-01 4.53978686e-05 2.06106005e-09]
```

소프트맥스 함수의 특징

softmax() 함수를 사용한 신경망 출력

```
:
1 a = np.array([0.3, 2.9, 4.0])
2 y = softmax(a)
3 print(y)
4 print(np.sum(y))
```

```
[0.01821127 0.24519181 0.73659691]
1.0
```

- 소프트맥스 함수의 출력은 0에서 1.0 사이의 실수
- 소프트맥스 함수 출력의 총합은 1 ⇒ 소프트맥스 함수의 출력을 '확률'로 해석

  예를 들어 위 그림에서 y[0]의 0.018(1.8%), y[1]의 확률은 0.245(24.5%), y[2]의 확률은 0.737(73.7%)로 해석 가능

  ⇒ 이 결과 확률들로부터 '2번째 원소의 확률이 가장 높으니, 답은 2번째 클래스'

  혹은 '74%의 확률로 2번째 클래스, 25%의 확률로 1번째 클래스, 1%의 확률로 0번째 클래스'
- 주의점은 소프트맥스 함수를 적용해도 각 원소의 대소 관계는 변하지 않음

  지수 함수 y=exp(x)가 **단조 증가 함수** 이기 때문

  예를 들어 a에서 가장 큰 원소는 2번째 원소, y에서 가장 큰 원소도 2번째 원소임

> 💡 **단조 증가 함수**
>
> 정의역 원소 a,b가 a≤b 일때, f(a) ≤f(b)가 성립하는 함수

출력층의 뉴런 수 정하기

- 출력층의 뉴런 수는 풀려는 문제에 맞게 적절히 정함
- 분류에서는 분류하고 싶은 클래스 수로 설정하는 것이 일반적

  예를 들어, 입력 이미지를 숫자 0부터 9 중 하나로 분류하는 문제면, 출력층의 뉴런은 10개

- 출력층 뉴런은 위에서부터 차례로 0,1, … 9 에 대응
- 뉴런의 회색 농도가 해당 뉴런의 출력 값의 크기를 의미

  여기서는 색이 가장 짙은 y2 뉴런이 가장 큰 값을 출력함

3-6. 손글씨 숫자 인식

- 신경망 구조를 **손글씨 숫자 분류**에 적용
- 해당 추론 과정은 신경망의 `순전파(forward propagation)`
- 기계학습과 마찬가지로 신경망도 두 단계를 거쳐 문제를 해결함

  (1) 훈련 데이터(학습 데이터)를 사용해 가중치 매개변수 학습

  (2) 추론 단계에서 앞서 학습한 매개변수를 사용하여 입력 데이터 분류

`MNIST 데이터셋`

- 0부터 9까지의 숫자 이미지
- 훈련 이미지 60,000장, 시험 이미지 10,000장
- 훈련 이미지를 사용하여 모델을 학습하고, 학습한 모델로 시험 이미지를 얼마나 정확하게 분류하는지 평가함
- 28X28 크기의 회색조 이미지(1채널), 각 픽셀은 0에서 255까지의 값을 취함

- 각 이미지에는 그 이미지가 실제 의미하는 숫자가 레이블로 붙어 있음

MNIST

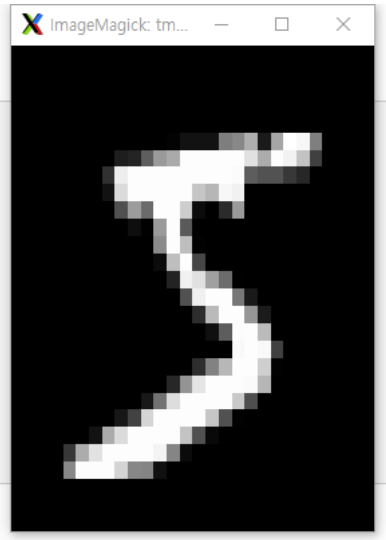
```
import tensorflow as tf

(x_train, t_train), (x_test, t_test) = tf.keras.datasets.mnist.load_data()

print(x_train.shape)
print(t_train.shape)
print(x_test.shape)
print(t_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

- (훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)

```
import sys, os
import numpy as np
import tensorflow as tf
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = tf.keras.datasets.mnist.load_data()
img = x_train[0]
label = t_train[0]
print(label)

print(img.shape)
img_show(img)
```

```
5
(28, 28)
```



신경망의 추론 처리

- MNIST 데이터셋을 가지고 추론을 수행하는 신경망 구현
- 해당 신경망은 입력층 뉴런을 784개, 출력층 뉴런을 10개로 구성함
- 입력층 뉴런이 784개인 이뉴는 이미지 크기가 28X28(784) 이기 때문이고,
  출력층 뉴런이 10개인 이유는 문제가 0에서 9까지의 숫자를 구분하는 문제이기 때문

- 은닉층은 총 두 개로, 첫 번째 은닉층에는 50개의 뉴런을, 두 번째 은닉층에는 100개의 뉴런을 배치 ⇒ 여기서의 50과 100은 임의로 정한 값

MNIST

```
import tensorflow as tf

(x_train, t_train), (x_test, t_test) = tf.keras.datasets.mnist.load_data()

print(x_train.shape)
print(t_train.shape)
print(x_test.shape)
print(t_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

- (훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)

```
import sys, os
import numpy as np
import tensorflow as tf
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = tf.keras.datasets.mnist.load_data()
img = x_train[0]
label = t_train[0]
print(label)

print(img.shape)
img_show(img)
```

```
5
(28, 28)
```

```
import pickle5 as pickle

def get_data():
    (x_train, t_train), (x_test, t_test) = tf.keras.datasets.mnist.load_data()
    return x_test, t_test

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

def predict(network, x):

    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    x = x.reshape(-1,784)

    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)
    if p == t[i]:
        accuracy_cnt += 1
print('Accuracy :' + str(float(accuracy_cnt) / len(x)))
```

```
<ipython-input-6-233ffe07162b>:2: RuntimeWarning: overflow encountered in exp
  return 1 / (1 +np.exp(-x))
```

```
Accuracy :0.9207
```

-> 올바르게 분류한 비율 0.92 (92%)

- 위에서 처리하지는 않았으나 데이터를 특정 범위로 변환하는 처리인 `정규화(Normalization)`와 신경망의 입력 데이터에 특정 변환을 가하는 것을 `전처리(pre-processing)` 이라고 함
- 현업에서는 데이터 전체의 분포를 고려해 전처리 하는데, 예를 들어 데이터 전체 평균과 표준편차를 이용하여 데이터들이 0을 중심으로 분포하도록 이동하거나, 데이터의 확산 범위를 제한하는 정규화를 수행함

  또는, 전체 데이터를 균일하게 분포시키는 데이터 백색화(`Whitening`) 등이 있음

`배치 처리`

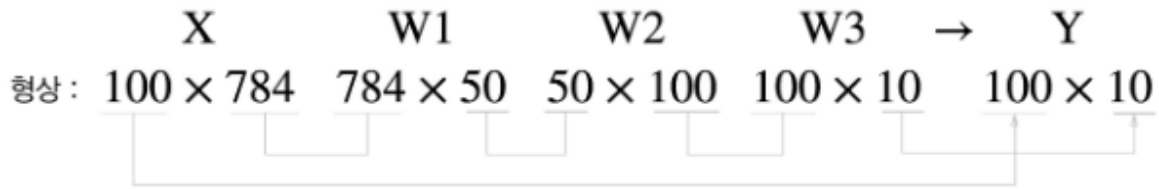
[신경망 각 층의 배열 형상의 추이]

- 원소 784개로 구성된 1차원 배열(28x28인 2차원 배열)이 입력되어, 마지막에는 원소가 10개인 1차원 배열이 출력되는 흐름으로, 이미지 데이터를 1장만 입력했을 때의 처리 흐름

[배치 처리를 위한 배열들의 형상 추이]

- 이미지 100개를 묶어 predict() 함수에 한번에 넘기면,

x의 형상을 100X784로 바꿔서 100장 분량의 데이터를 하나의 입력 데이터로 표현하면 됨



- 입력 데이터의 형상은 100X784, 출력 데이터의 형상은 100X10
- 100장 분량 입력 데이터의 결과가 한번에 출력됨
- x[0]과 y[0] 에는 0번째 이미지와 추론 결과가 저장되는 식

- 하나로 묶은 입력 데이터 ⇒ **`배치(batch)`**

`배치 처리` 는 이미지 1장당 처리 시간을 대폭 줄여줌

(1) 수치 계산 라이브러리 대부분이 큰 배열을 효율적으로 처리할 수 있도록 고도로 최적화 되어 있음
(2) 커다란 신경망은 데이터 전송이 병목으로 작용하는 경우가 있는데, 배치 처리를 하면서 버스에 주는 부하를 줄임
[느린 I/O 를 통해 데이터를 읽는 횟수가 줄어, 빠른 CPU나 GPU로 순수 계산을 수행하는 비율이 높아짐]

즉, 배치 처리를 수행함으로써 큰 배열로 이뤄진 계산을 하게 되는데,
컴퓨터에서는 큰 배열을 한꺼번에 계산하는 것이 분할된 작은 배열을 여러 번 계산하는 것보다 빠름

`배치 처리 구현`

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

batch_size = 100 #배치 크기
accuracy_cnt = 0

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)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

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

```
Accuracy :0.9207
```

```
<ipython-input-6-233ffe07162b>:2: RuntimeWarning: overflow encountered in exp
  return 1 / (1 +np.exp(-x))
```

- range() 함수가 반환하는 반복자를 바탕으로 x[i:i+batch_size] 에서 입력 데이터를 묶음
- x[i:i+batch_size]는 입력 데이터의 i번째부터 i+batch_size번까지 데이터를 묶음
- argmax()로 최댓밧의 인덱스를 가져오면서, axis=1 인수를 추가하여, 100x10의 배열 중 1번째 차원을 구성하는 각 원소에서 (1번째 차원을 축으로) 최댓값의 인덱스를 찾도록 함
- 인덱스가 0부터 시작하므로, 0번째 차원이 가장 처음 차원임)

```
x = np.array([[0.1, 0.8, 0.1], [0.3, 0.1, 0.6], [0.2, 0.5, 0.3], [0.8, 0.1, 0.1]])
y = np.argmax(x, axis=1)

print(y)

```

```
[1 2 1 0]
```

```
y = np.array([1,2,1,0])
t = np.array([1,2,0,0])

print(y==t)

print(np.sum(y==t))
```

```
[ True  True False  True]
3
```

3-7. 정리

- 신경망은 각 층의 뉴런들이 다음 층의 뉴런으로 신호를 전달한다는 점에서 퍼셉트론과 같음
- 하지만 다음 뉴런으로 갈 때 신호를 변화시키는 활성화 함수에 큰 차이가 있음
- 신경망에서는 매끄럽게 변화하는 시그모이드 함수를, 퍼셉트론에서는 갑자기 변화하는 계단 함수를 활성화 함수로 사용함

- 신경망에서는 활성화 함수로 시그모이드 함수와 ReLU 함수 같은 매끄럽게 변화하는 함수를 이용함
- 넘파이의 다차원 배열을 잘 사용하면 신경망을 효율적으로 구현할 수 있음
- 기계학습 문제는 크게 회귀와 분류로 나눌 수 있음
- 출력층의 활성화 함수로는 회귀에서는 주로 항등 함수를, 분류에서는 주로 소프트맥스 함수를 이용함
- 분류에서는 출력층의 뉴런 수를 분류하려는 클래스 수와 같게 설정함
- 입력 데이터를 묶은 것을 배치라고 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 훨씬 빠르게 얻을 수 있음