CH 3. 신경망

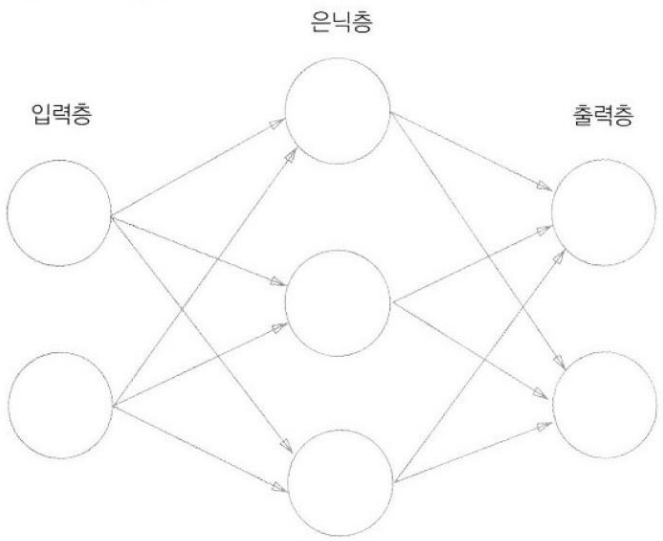
DSL 9기 유희조

Contents

1. 퍼셉트론에서 신경망으로

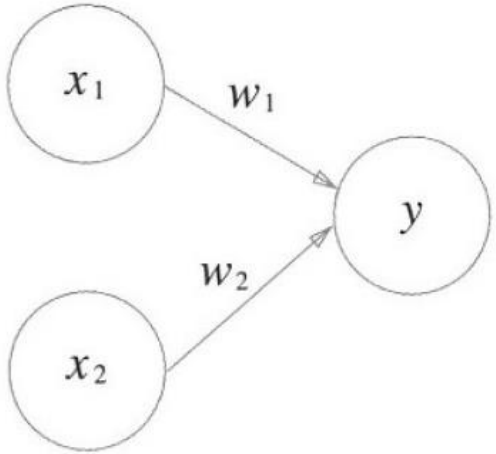
퍼셉트론 복습

신경망



2층 신경망
(가중치를 갖는 층의 개수 기준)

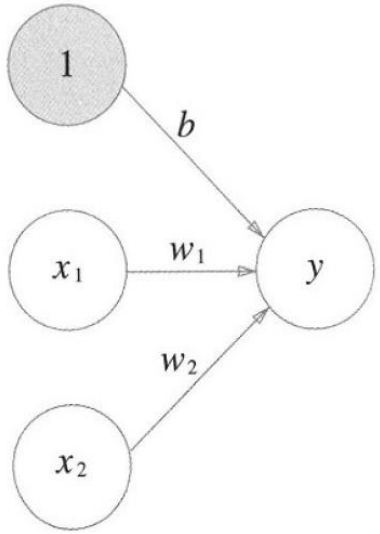
퍼셉트론



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

b (편향) : 뉴런이 얼마나 쉽게 활성화되느냐
w(가중치) : 각 신호의 영향력 제어

편향을 명시한 퍼셉트론



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

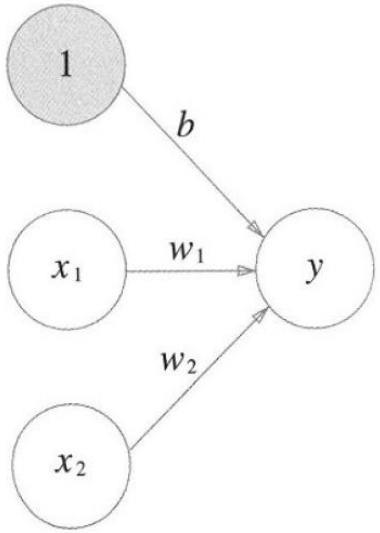
$$h(x) = \begin{cases} 0\ (x \le 0) \\ 1\ (x > 0) \end{cases}$$

활성화 함수

1. 퍼셉트론에서 신경망으로

활성화 함수

편향을 명시한 퍼셉트론

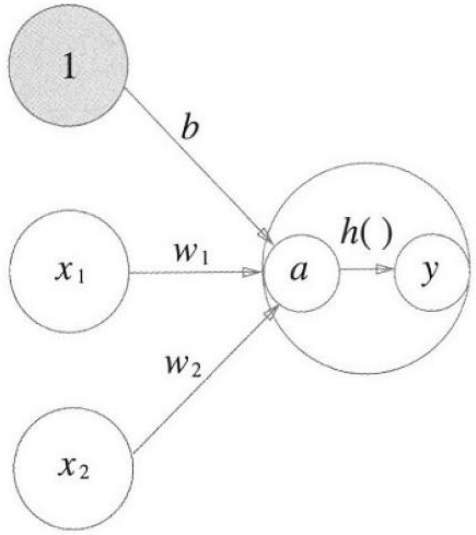


$$y = h(b + w_1x_1 + w_2x_2)$$

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

활성화 함수
: 입력 신호의 총합을 출력 신호로 변환

활성화 함수의 처리 과정



$$a = b + w_1x_1 + w_2x_2$$

$$y = h(a)$$

2. 활성화 함수

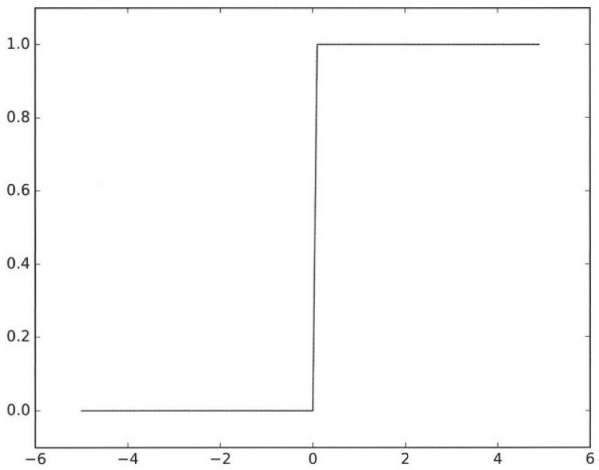
시그모이드 함수

$$a = b + w_1 x_1 + w_2 x_2$$

$$y = h(a)$$

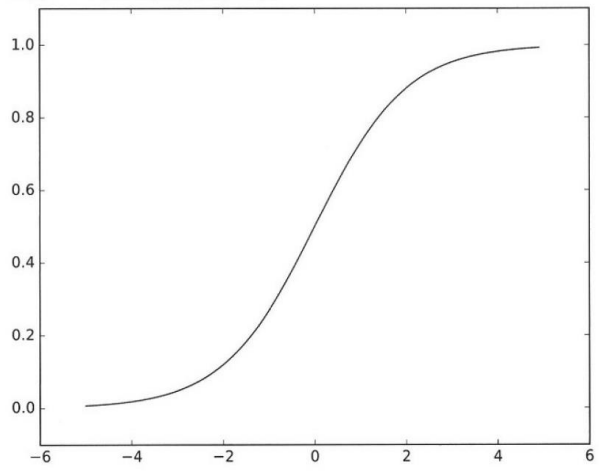
퍼셉트론 – 계단 함수

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



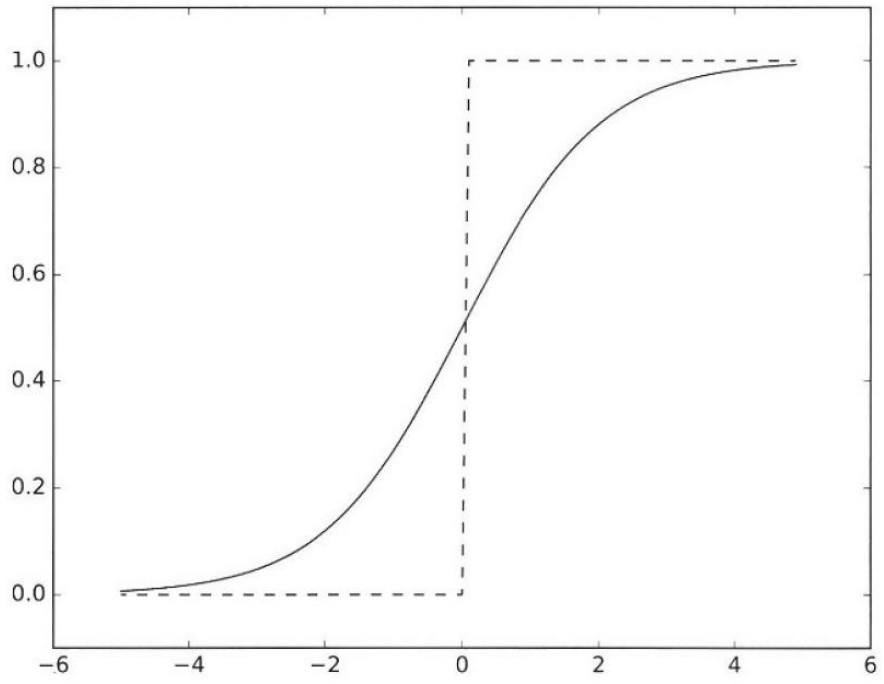
신경망 – 시그모이드 함수

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



2. 활성화 함수

시그모이드 함수 vs 계단 함수



계단 함수 (점선)
시그모이드 함수 (실선)

	계단 함수	시그모이드 함수
공통점	• 입력이 작을 때 0 또는 0에 가까운 값 출력 • 입력이 커질 때 1 또는 1에 가까운 값 출력 • 입력이 아무리 작거나 커도 출력은 0에서 1 사이 • 비선형 함수	
차이점	0을 경계로 출력이 갑자기 바뀜	입력에 따라 출력이 연속적으로 변화(매끄러움)
	0과 1 중 하나만 출력	연속적인 실수

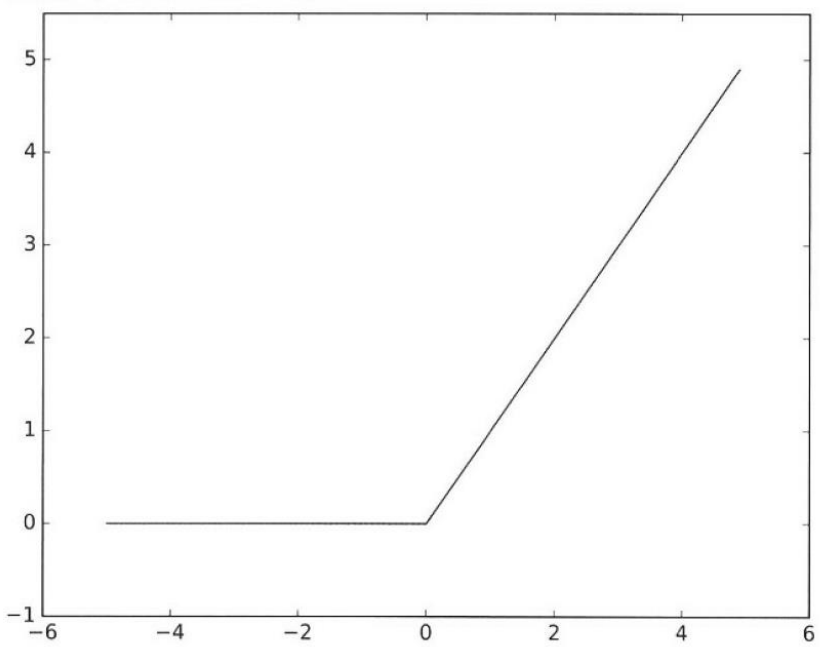
신경망의 활성화 함수는 비선형 함수만 가능!

$h(x) = cx$

$Y(x) = h(h(h(x)) = c^3x$

2. 활성화 함수

ReLU 함수



ReLU 함수

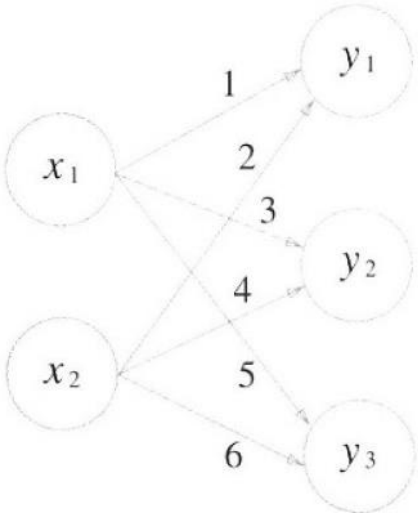
: 입력이 0을 넘으면 그 입력을 그대로 출력,
0 이하면 0을 출력

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

3. 다차원 배열의 계산

신경망에서의 행렬 곱

행렬의 곱으로 신경망 계산 수행



X1가중치 $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$
X2가중치

$$X \quad W \quad = \quad Y$$
$$2 \quad 2 \times 3 \quad 3$$

일치

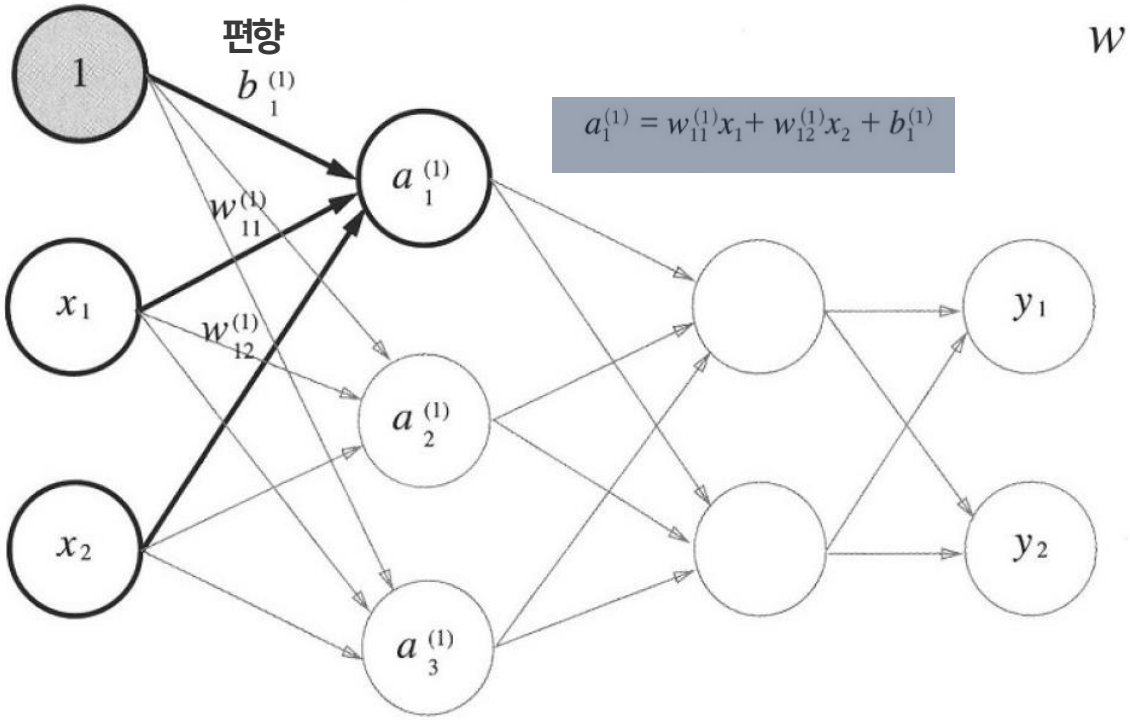
```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
 [2 4 6]]
>>> W.shape
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17]
```

다차원배열의스칼라곱구하기

4. 3층 신경망 구현하기

각 층의 신호 전달 구현하기

3층 신경망



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

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

1층

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

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

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

4. 3층 신경망 구현하기

각 층의 신호 전달 구현하기

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

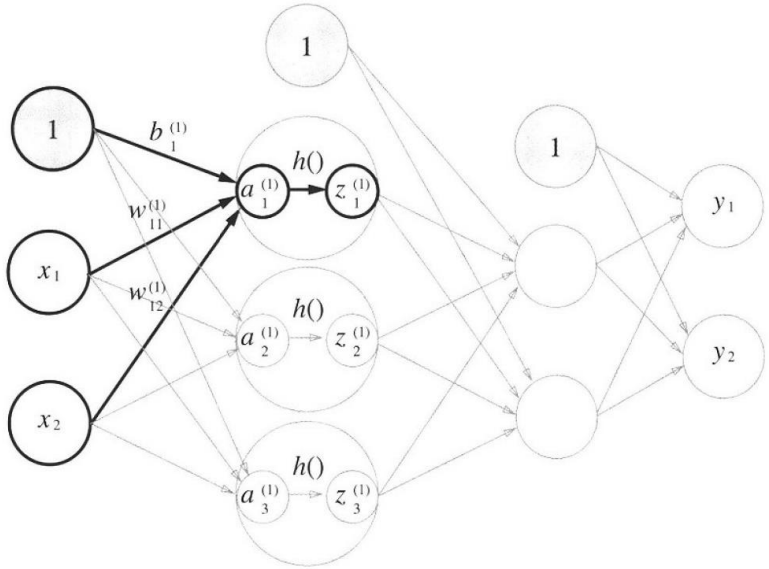
$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

```
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) # (2, 3)
print(X.shape)  # (2,)
print(B1.shape) # (3,)

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

4. 3층 신경망 구현하기

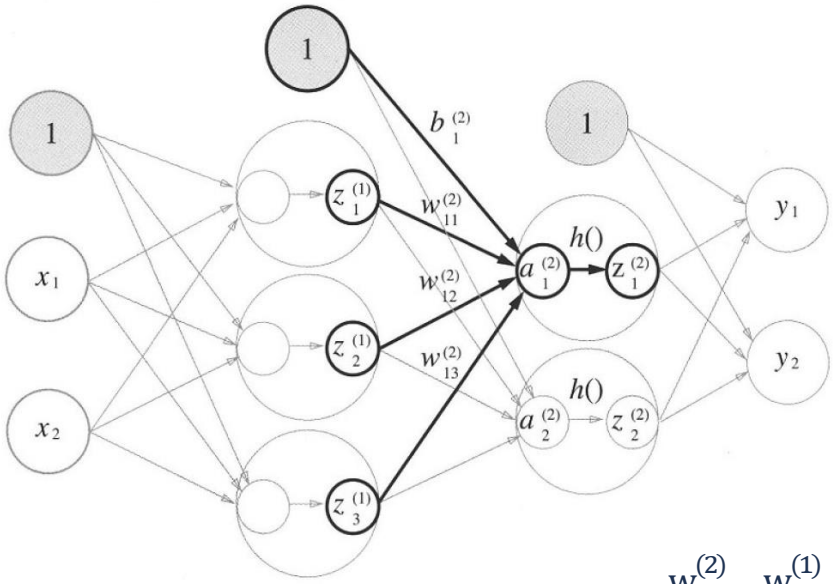
각 층의 신호 전달 구현하기

입력층에서 1층으로의 신호 전달




```
Z1 = sigmoid(A1)

print(A1) # [0.3, 0.7, 1.1]
print(Z1) # [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) # (3,)
print(W2.shape) # (3, 2)
print(B2.shape) # (2,)

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

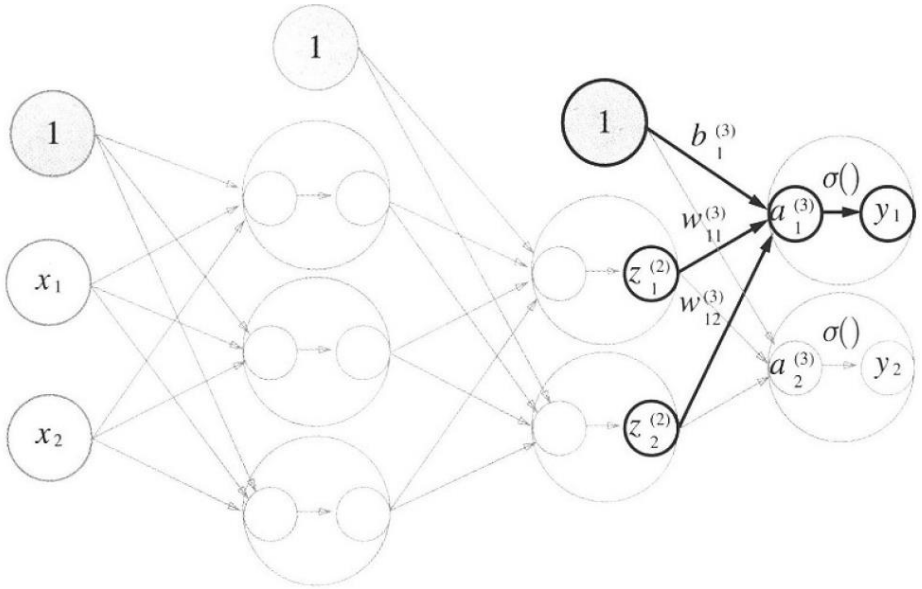
$$W2 = \begin{matrix} w^{(2)}_{11} & w^{(1)}_{21} \\ w^{(2)}_{12} & w^{(2)}_{22} \\ w^{(2)}_{13} & w^{(2)}_{23} \end{matrix}$$

$$B2 = b^{(2)}_1 \quad b^{(2)}_2$$

4. 3층 신경망 구현하기

각 층의 신호 전달 구현하기

2층에서 출력층으로의 신호 전달




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

항등함수

```
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
```

$$W3=\begin{matrix} w^{(3)}_{11} & w^{(3)}_{21} \\ w^{(3)}_{12} & w^{(3)}_{22} \end{matrix} \qquad B3=b^{(3)}_1 \quad b^{(3)}_2$$

4. 3층 신경망 구현하기

신호 전달 구현 정리

가중치와 편향 초기화
딕셔너리 변수 network에 저장

```
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
```

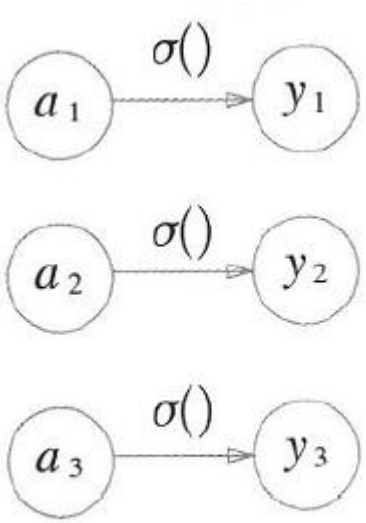
입력 신호를 출력으로 변환하는 처리 과정 구현

```
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]
```

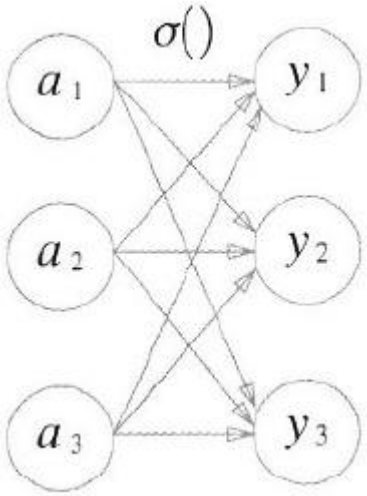
5. 출력층 설계하기

항등 함수와 소프트맥스 함수 구현하기

회귀 – 항등함수



분류 – 소트프맥스 함수



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

n : 출력층의 뉴런 수
y_k : k번째 출력
a_k : 입력 신호

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

    return y
```

5. 출력층 설계하기

소프트맥스 함수 구현 시 주의점

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

임의의 정수 C 곱하기

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

C 지수함수 안으로 이동

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

logC를 C'로 치환

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # 소프트맥스 함수의 계산
array([ nan,  nan,  nan])         # 제대로 계산되지 않는다.
>>>
>>> c = np.max(a)                 # c = 1010 (최댓값)
>>> a - c
array([  0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([  9.99954600e-01,   4.53978686e-05,   2.06106005e-09])
```

```
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
```

5. 출력층 설계하기

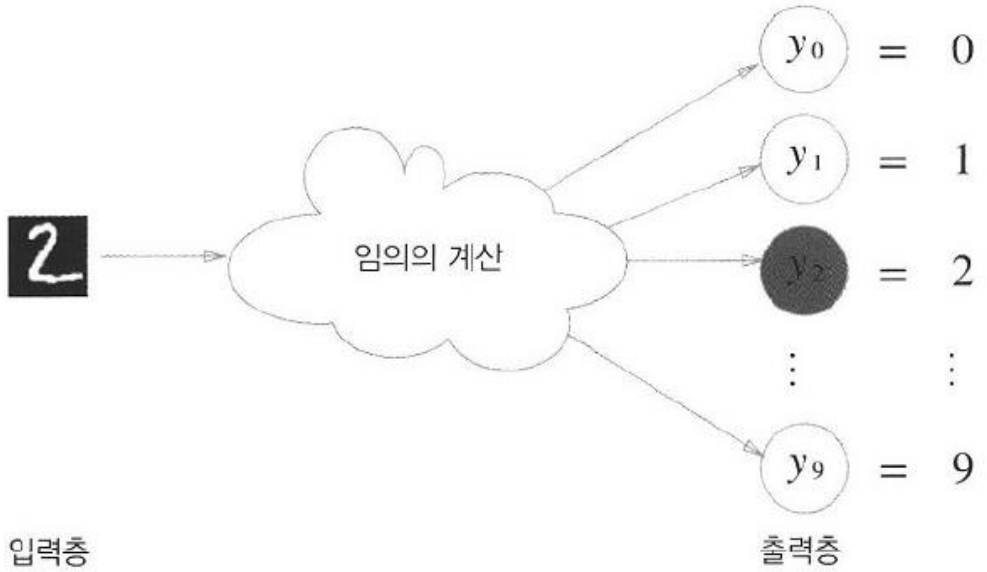
소프트맥스 함수의 특징 | 출력층의 뉴런 수 정하기

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

출력 : 0과 1 사이

출력 총합 = 1

출력층의 뉴런 수 정하기



6. 손글씨 숫자 인식

MNIST 데이터셋

0~9 숫자 이미지
28x28 크기

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from dataset.mnist import load_mnist

# 처음 한 번은 몇 분 정도 걸립니다.
(x_train, t_train), (x_test, t_test) = \
    load_mnist(flatten=True, normalize=False)

# 각 데이터의 형상 출력
print(x_train.shape) # (60000, 784)
print(t_train.shape) # (60000,)
print(x_test.shape)  # (10000, 784)
print(t_test.shape)  # (10000,)
```

- Flatten = True : 입력 이미지를 1차원 배열로 저장
- Normalize = False : 입력 이미지의 픽셀을 원래 값 그대로 (0~255) 유지

6. 손글씨 숫자 인식

신경망의 추론 처리

```
def get_data():
    (x_train, t_train), (x_test, t_test) = \
        load_mnist(normalize=True, flatten=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 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
```

- 입력층 뉴런 784개 (이미지 크기 28x28)
- 출력층 뉴런 10개(0~9까지의 숫자 구분)
- 은닉층 2개

```
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)))
```

For문 돌며 x에 저장된 이미지 데이터 한 장씩 꺼내 predict() 함수로 분류

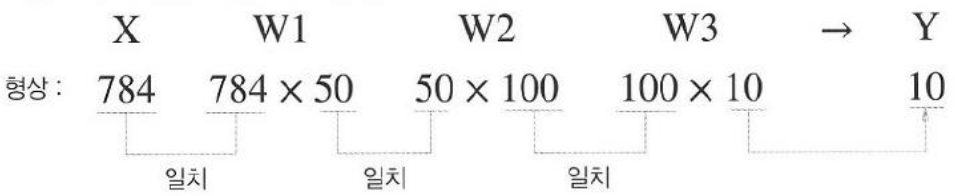
Predict() : 각 레이블의 확률을 넘파이 배열로 반환

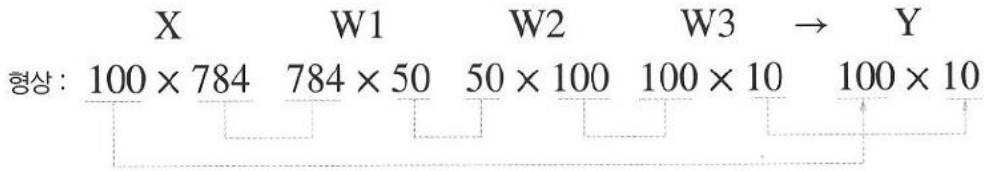
신경망이 예측한 답변과 정답 레이블 비교하여 맞힌 숫자 세기

6. 손글씨 숫자 인식

배치 처리

```
>>> x, _ = get_data()
>>> network = init_network()
>>> W1, W2, W3 = network['W1'], network['W2'], network['W3']
>>>
>>> x.shape
(10000, 784)
>>> x[0].shape
(784,)
>>> W1.shape
(784, 50)
>>> W2.shape
(50, 100)
>>> W3.shape
(100, 10)
```





6. 손글씨 숫자 인식

소프트맥스 함수의 특징 | 출력층의 뉴런 수 정하기

```
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)))
```

입력 데이터 묶기

```
>>> y = np.array([1, 2, 1, 0])
>>> t = np.array([1, 2, 0, 0])
>>> print(y==t)
[True True False True]
>>> np.sum(y==t)
3
```

Summary

- 신경망에서는 활성화 함수로 시그모이드 함수나 ReLU 함수와 같은 비선형 함수를 이용한다.
- 넘파이의 다차원 배열을 통해 신경망을 효율적으로 구현할 수 있다.
- 출력층의 활성화 함수로는 회귀의 경우 주로 항등 함수를, 분류의 경우 소프트맥스 함수를 이용한다.
- 분류에서는 출력층의 뉴런 수를 분류하려는 클래스의 수와 같게 설정한다.
- 입력 데이터를 묶은 것을 배치라고 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 훨씬 빠르게 얻을 수 있다.

E.O.D