Deep Learning from Scratch

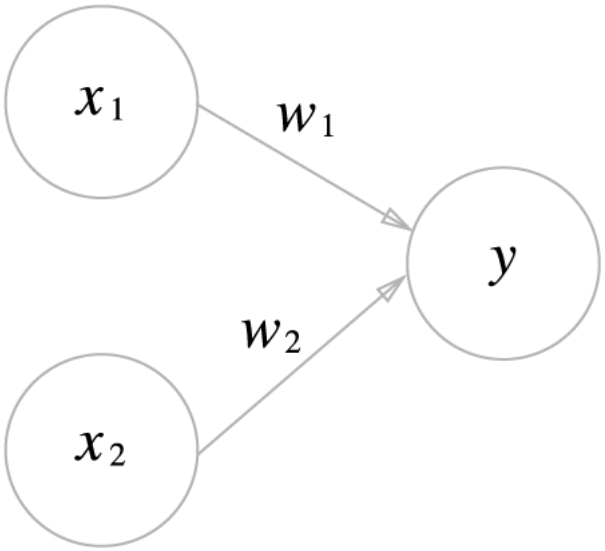
조용군

iceman4u@naver.com

퍼셉트론

퍼셉트론(Perceptron)

- ✓ **인공신경망의 한 종류, linear classifier**
- ✓ **1957, 프랑크 로젠블라트(Frank Rosenblatt)**
- ✓ **다수의 입력 신호, 하나의 출력 신호**
  - 신호는 0 또는 1
- ✓ **용어**
  - 뉴런(neuron), 노드
  - 가중치(weight)
  - 임계값(threshold)



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

퍼셉트론 이용한 단순 논리회로 구성

✓ **AND 게이트**

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

✓ **NAND 게이트**

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

✓ **OR 게이트**

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

✓ **위 게이트를 퍼셉트론으로 구현**

- w_1, w_2, θ 값 결정
- (0.5, 0.5, 0.7)
- (-0.5, -0.5, -0.7)
- (?, ?, ?)

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

퍼셉트론 간단 구현

✓ AND 게이트

```
def AND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.7
    tmp = w1*x1 + w2*x2
    if tmp <= theta:
        return 0
    else:
        return 1


>>> AND(0, 0)
>>> AND(1, 0)
>>> AND(0, 1)
>>> AND(1, 1)
```

바이어스(bias) 도입

✓ θ 를 $-b$로 치환

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

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

✓ **가중치**

- 각 입력 신호가 결과에 주는 영향력(중요도) 조절하는 매개변수

✓ **바이어스**

- 뉴런이 얼마나 쉽게 활성화(결과로 1을 출력)하느냐를 조정하는 매개변수

바이어스 고려한 논리회로 구현

✓ AND 게이트

```
import numpy as np

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


for xs in [(0, 0), (1, 0), (0, 1), (1, 1)]:
        y = AND(xs[0], xs[1])
        print(str(xs) + " -> " + str(y))
```

바이어스 고려한 논리회로 구현

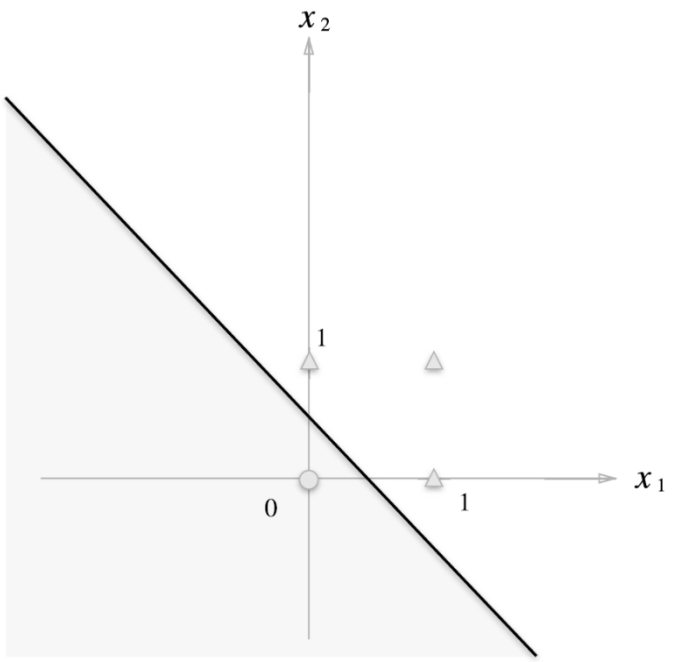
- ✓ NAND 게이트

```
import numpy as np

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

바이어스 고려한 논리회로 구현

✓ **OR 게이트**

```
import numpy as np

def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([1.0, 1.0])
    b = -0.5
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```



단층 퍼셉트론의 학습

1. Initialize the weights and the threshold. Weights may be initialized to 0 or to a small random value. In the example below, we use 0.
2. For each example j in our training set D, perform the following steps over the input $\mathbf{x}_j$ and desired output d_j:

   a. Calculate the actual output:

   $$\begin{aligned} y_j(t) &= f[\mathbf{w}(t) \cdot \mathbf{x}_j] \\ &= f[w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}] \end{aligned}$$

   b. Update the weights:

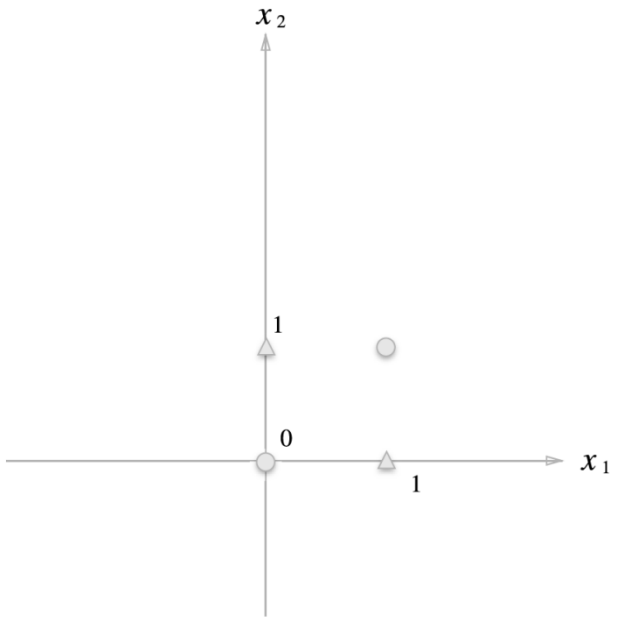
   $$w_i(t+1) = w_i(t) + (d_j - y_j(t))x_{j,i}$$, for all features $0 \leq i \leq n$.

3. For offline learning, the step 2 may be repeated until the iteration error $\frac{1}{s}\sum_{j=1}^{s}|d_j - y_j(t)|$ is less than a user-specified error threshold γ, or a predetermined number of iterations have been completed.

퍼셉트론의 한계

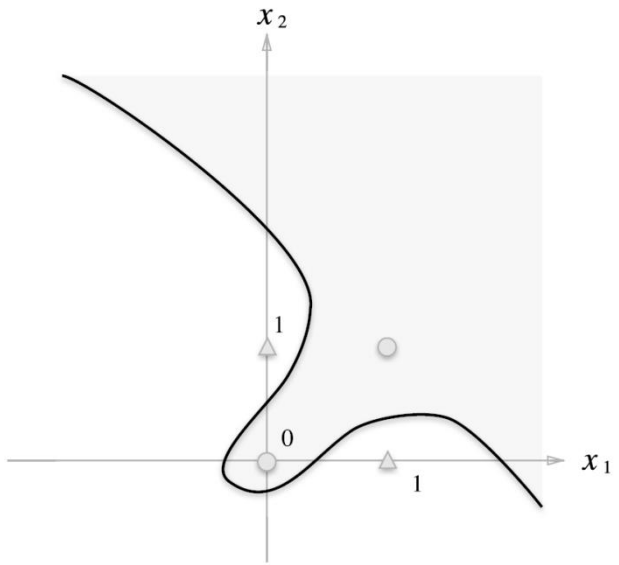
✓ **XOR 문제**

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



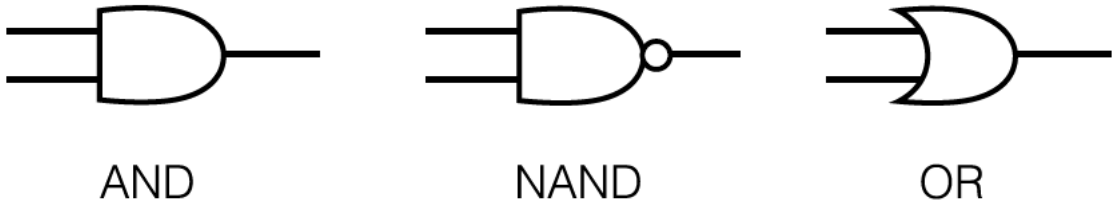
퍼셉트론의 한계

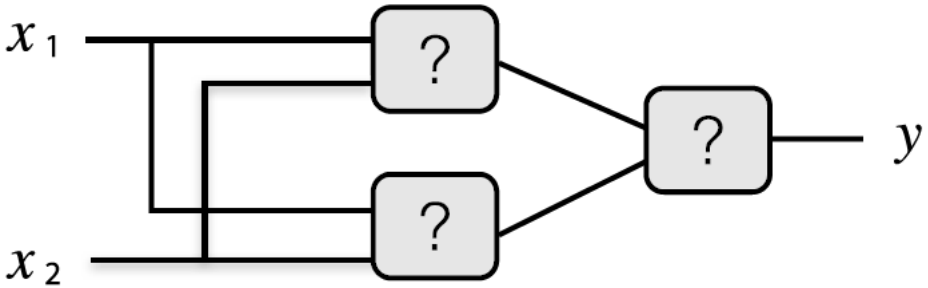
- ✓ 퍼셉트론은 직선 하나로 두 개의 영역으로 나눔
- ✓ 직선 하나로 출력을 구분할 수 없는 XOR 게이트를 단순 퍼셉트론(SLP)으로는 해결 못함



XOR

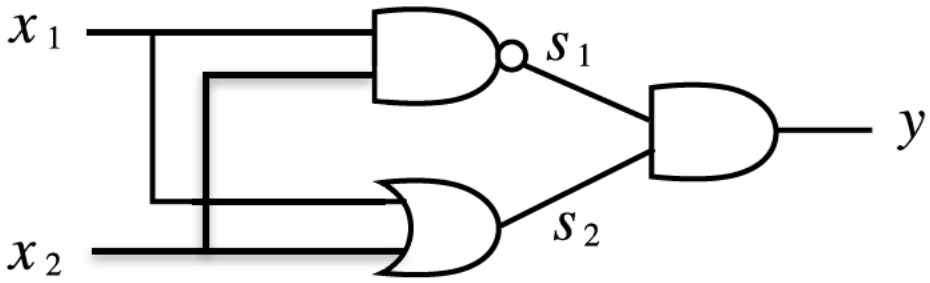
✓ XOR 게이트를 AND, NAND, OR 게이트로 표현





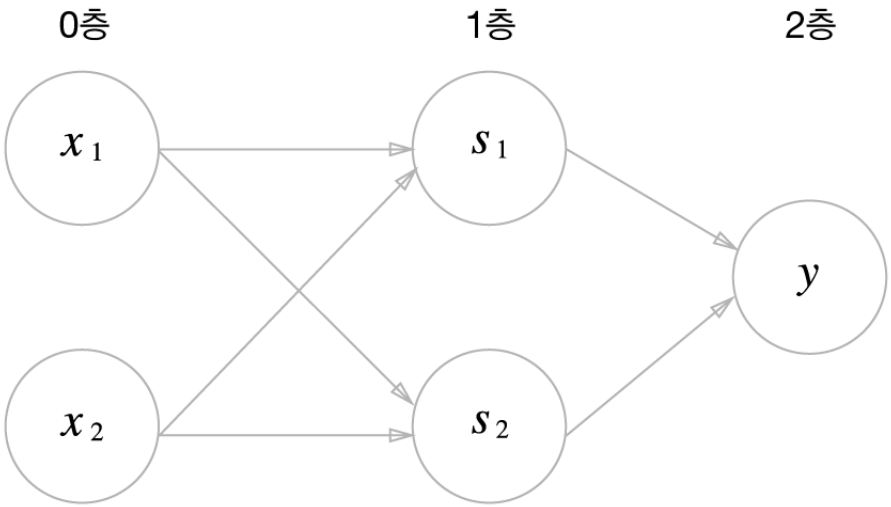
XOR

✓ XOR 게이트를 AND, NAND, OR 게이트로 표현



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

다층 퍼셉트론(MLP)




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

for xs in [(0, 0), (1, 0), (0, 1), (1, 1)]:
    y = XOR(xs[0], xs[1])
    print(str(xs) + " -> " + str(y))
```

정리

- ✓ 퍼셉트론은 입출력을 갖춘 단순 인공신형망의 한 종류
- ✓ 퍼셉트론에서는 가중치와 바이어스를 매개변수로 설정
- ✓ 퍼셉트론으로 논리 회로를 표현 가능
- ✓ XOR 게이트는 단순 퍼셉트론으로는 표현 불가능
- ✓ 2층 퍼셉트론을 이용하면 XOR 게이트 표현 가능
- ✓ 단층 퍼셉트론은 직선형 영역만 표현(구별)가능하고, 다층 퍼셉트론은 비선형 영역도 표현 가능
- ✓ 다층 퍼셉트론은 (이론상) 컴퓨터 표현 가능

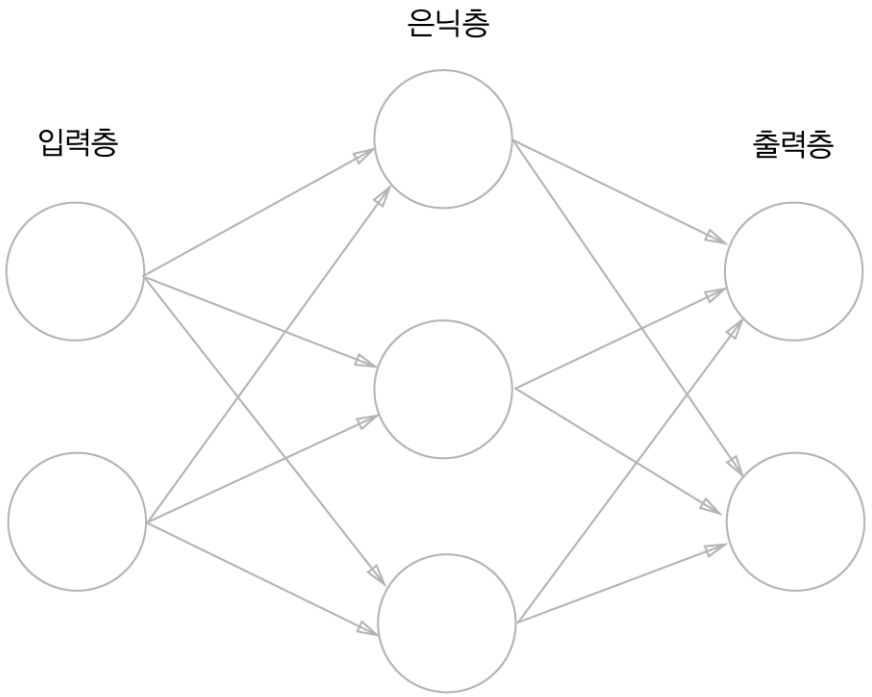
신경망, 신경회로망, 인공신경망

퍼셉트론에서 신경망으로

✓ **신경망**

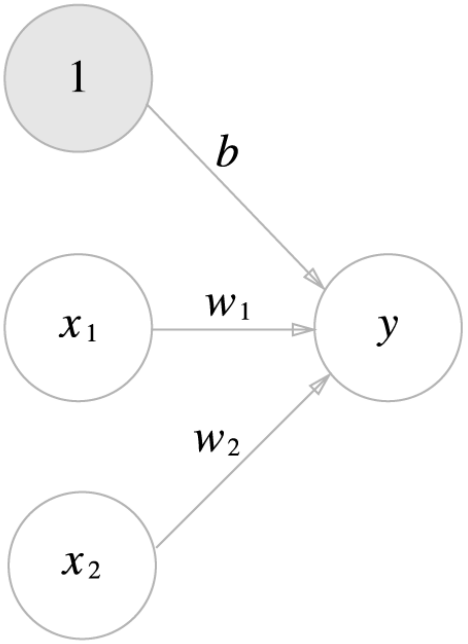
- 적절한 가중치 값을 데이터로부터 자동으로 설정
- 학습

- 입력층
- 출력층
- 은닉층(hidden layer)
  - 1개: 얕은 신경망
  - 2개 이상: 깊은 신경망



바이어스를 고려한 퍼셉트론

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



$$y = h(b + w_1 x_1 + w_2 x_2)$$

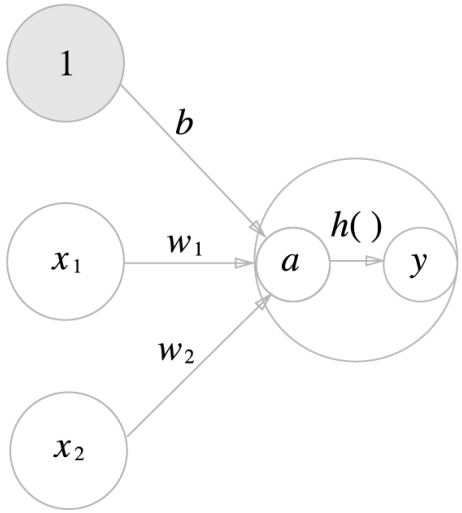
$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

활성화 함수 등장

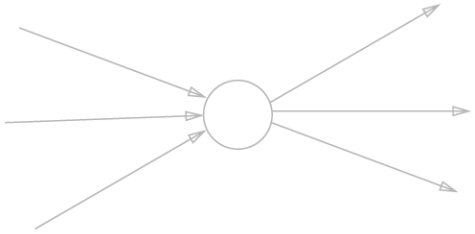
- ✓ **활성화 함수(activation function, h)**
  - 입력 신호의 총합(net)을 출력 신호로 변환하는 함수

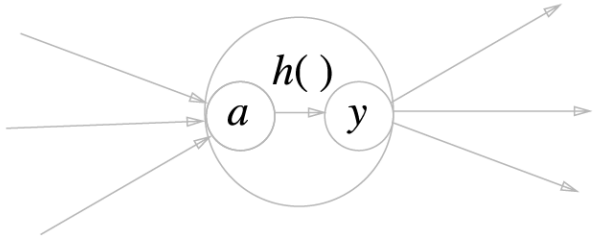
$$a = b + w_1 x_1 + w_2 x_2$$

$$y = h(a)$$



- ✓ **뉴런(노드) 표기 방법**

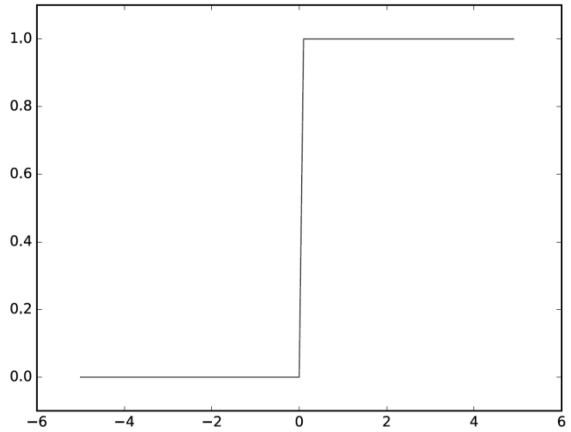


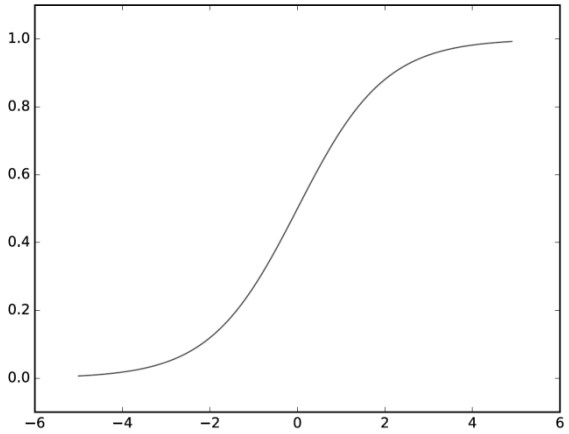


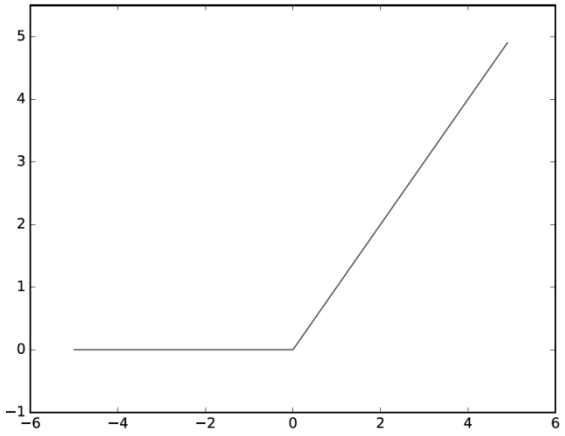
활성화 함수

✓ **종류**

- 계단 함수(step function)
- 시그모이드 함수(sigmoid function)
- ReLU 함수(Rectified Linear Unit function)







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

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

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

활성화 함수

✓ 계단 함수 구현

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

```
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 지정
plt.show()
```

활성화 함수

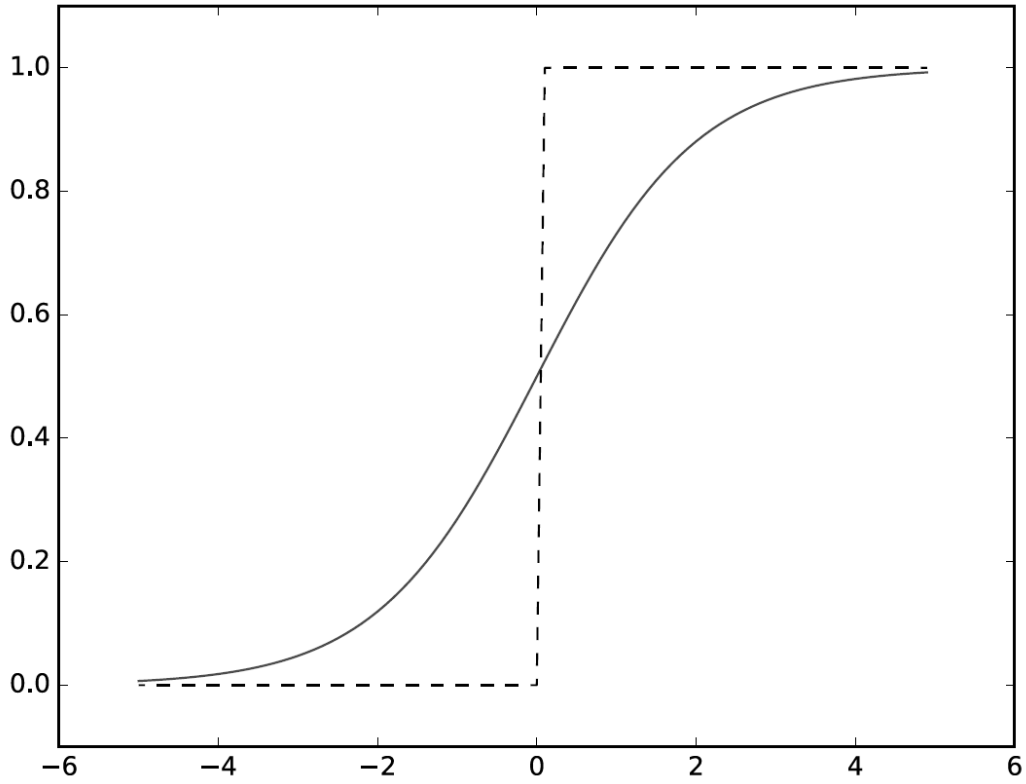
✓ 시그모이드 함수 구현

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

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

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

X = np.arange(-5.0, 5.0, 0.1)
Y = sigmoid(X)
plt.plot(X, Y)
plt.ylim(-0.1, 1.1)
plt.show()
```

시그모이드 함수와 계단 함수 비교

- ✓ **차이점**
  - 모양
  - 출력값
- ✓ **공통점**
  - 작은 입력 ➔ 출력: 0
  - 큰 입력 ➔ 출력: 1
  - 출력값 범위
  - 비선형



비선형 함수

- ✓ **신경망에서 활성화 함수는 비선형 함수이어야 함**
  - 다층 신경망에서는 필수

- ✓ **이유?**
  - 활성화 함수가 선형이라면
  - 즉

$$h(x) = cx$$

  - 3계층 신경망의 출력, y

$$y = h\Big(h\big(h(x)\big)\Big) = c\big(c(cx)\big) = c^3x = ax$$

  - 3계층 신경망을 1계층 신경망, 즉 은닉층이 없는 신경망으로 표현 가능

  - 은닉층을 사용하는 이점이 사라짐

활성화 함수

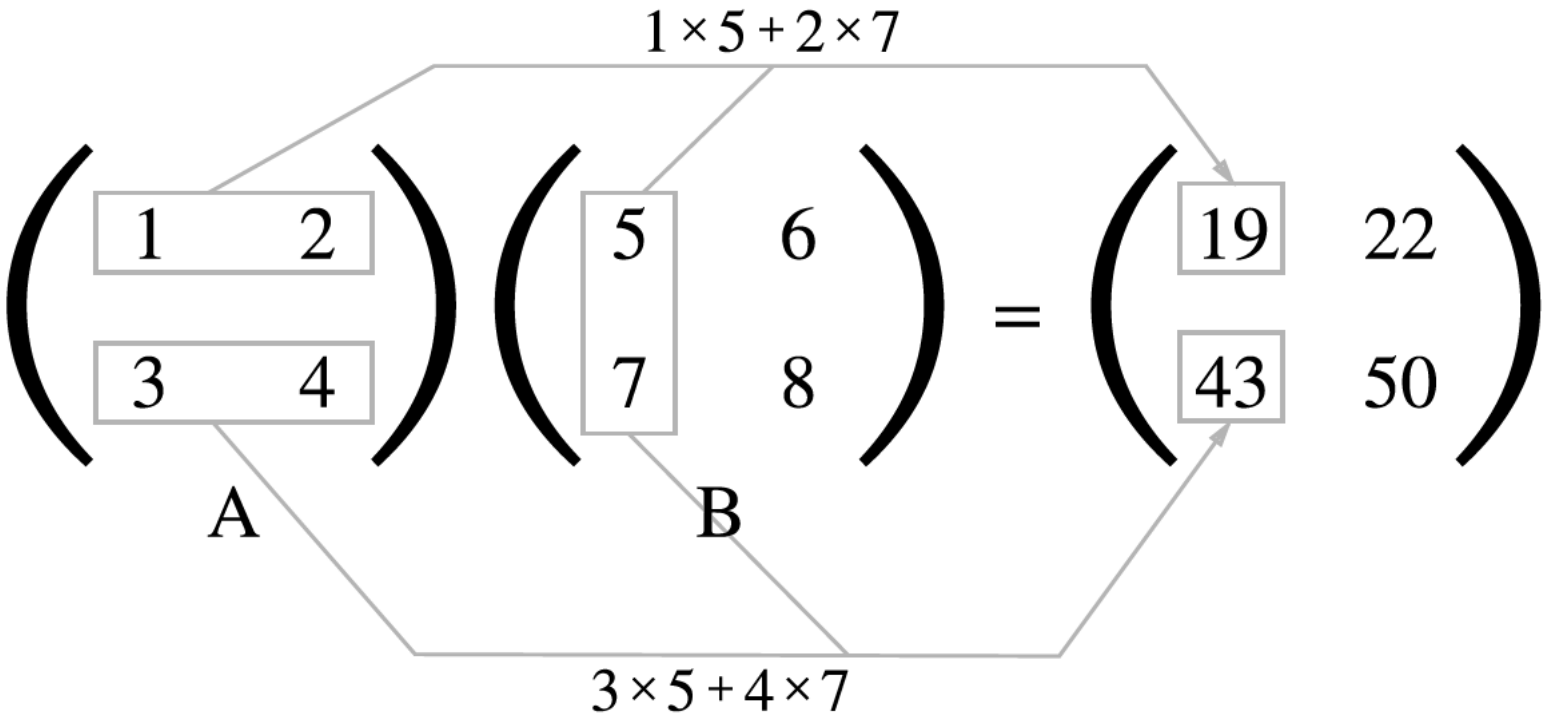
✓ **ReLU 함수 구현**

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

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

def relu(x):
    return np.maximum(0, x)

x = np.arange(-5.0, 5.0, 0.1)
y = relu(x)
plt.plot(x, y)
plt.ylim(-1.0, 5.5)
plt.show()
```

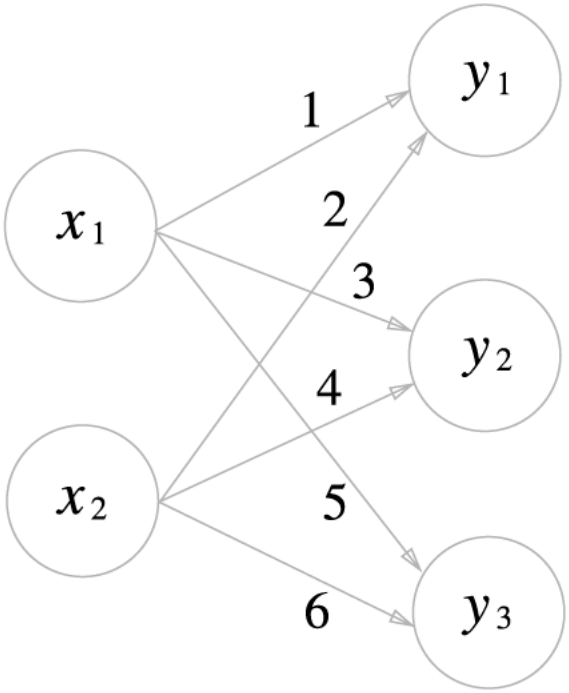
행렬의 내적(행렬 곱)

- ✓ np.dot(A, B)



신경망의 내적

✓ 신경망의 출력을 행렬 연산으로



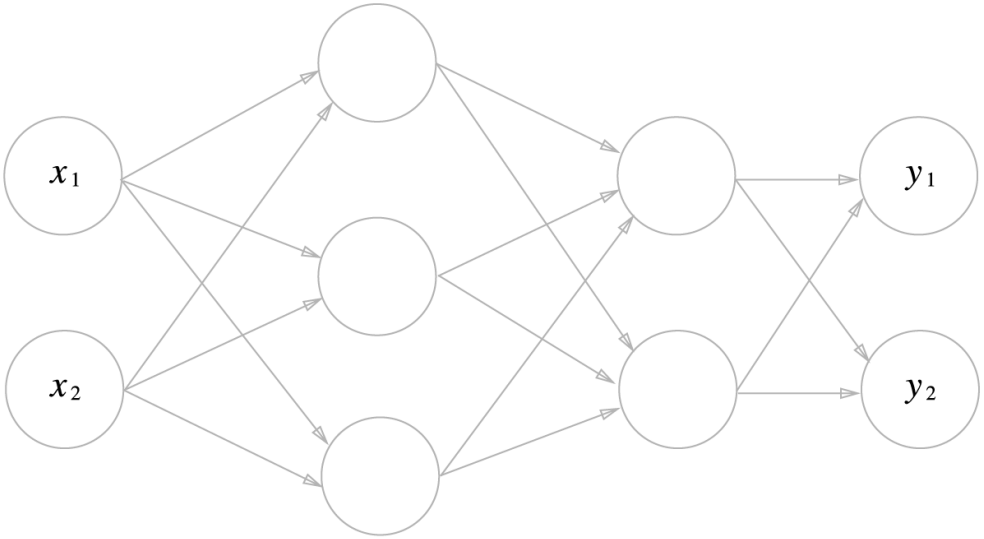
$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

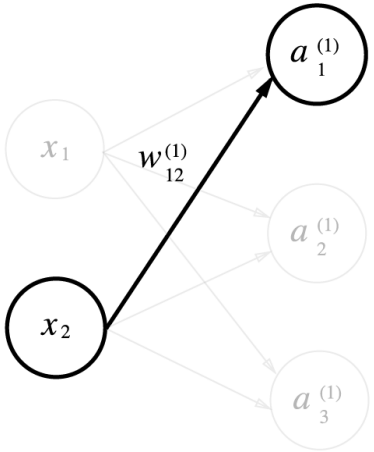
$$X \quad W = Y$$

$$2 \quad 2 \times 3 \quad 3$$

일치

3층 신경망



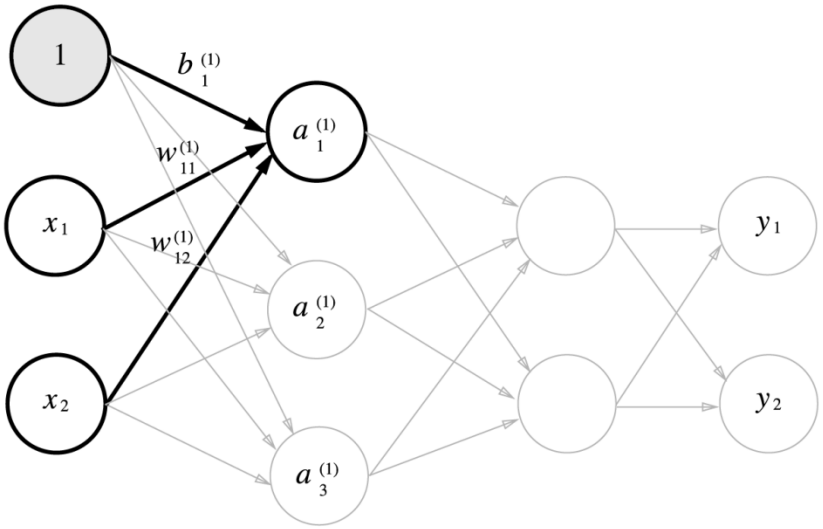


$W^{(1)}_{12}$

- 1층의 가중치
- 앞 층의 2번째 뉴런
- 다음 층의 1번째 뉴런

각 층의 신호 전달 구현(신경망의 순방향)

✓ 입력층에서 1층의 첫 번째 뉴런으로 가는 신호



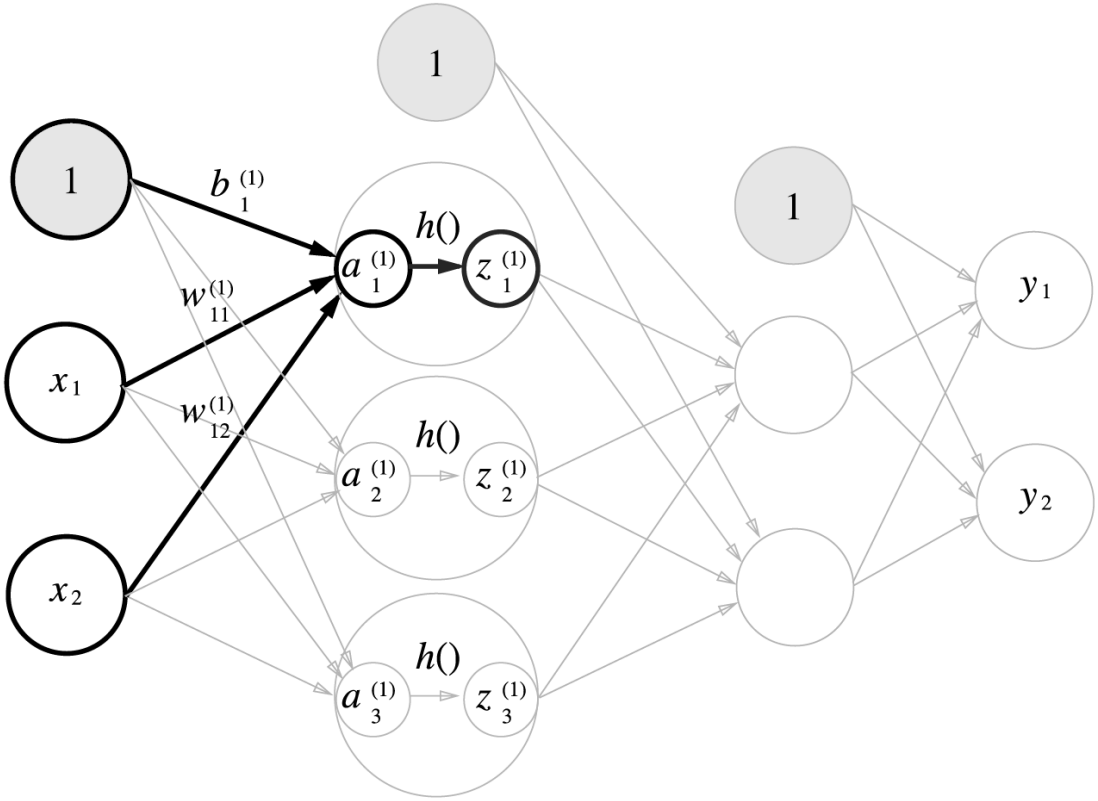
$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}$$

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

각 층의 신호 전달 구현

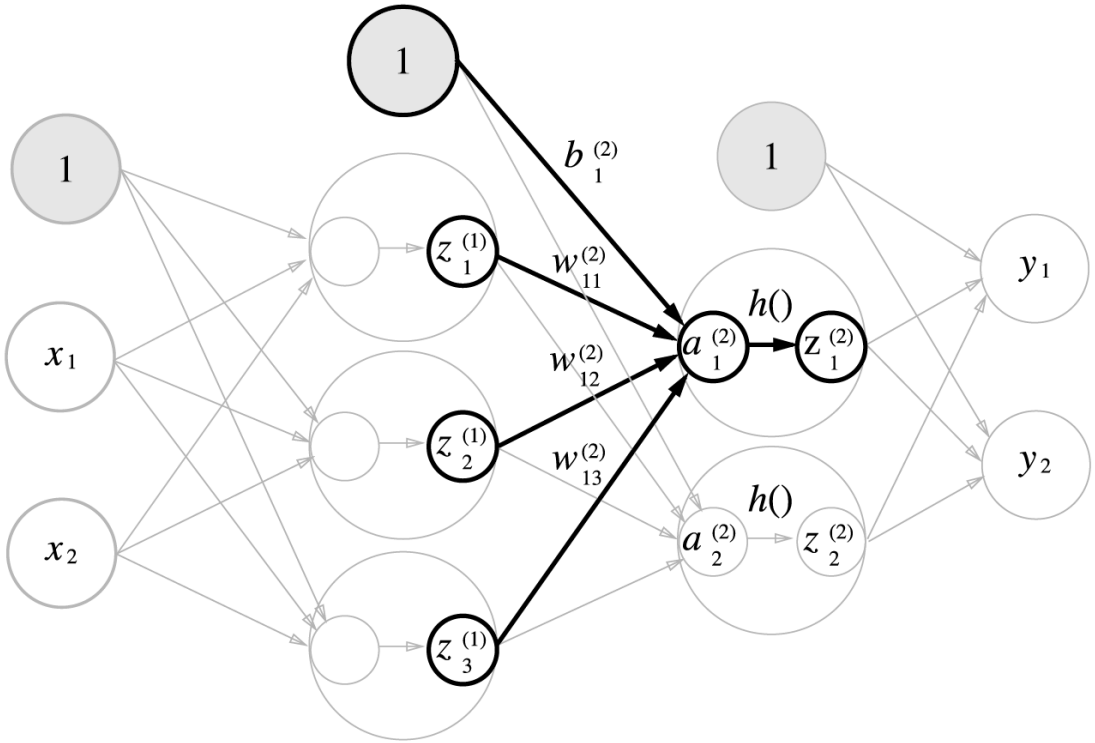
✓ **입력층에서 1층으로의 신호 전달**

- $h(x)$ 는 시그모이드 함수



각 층의 신호 전달 구현

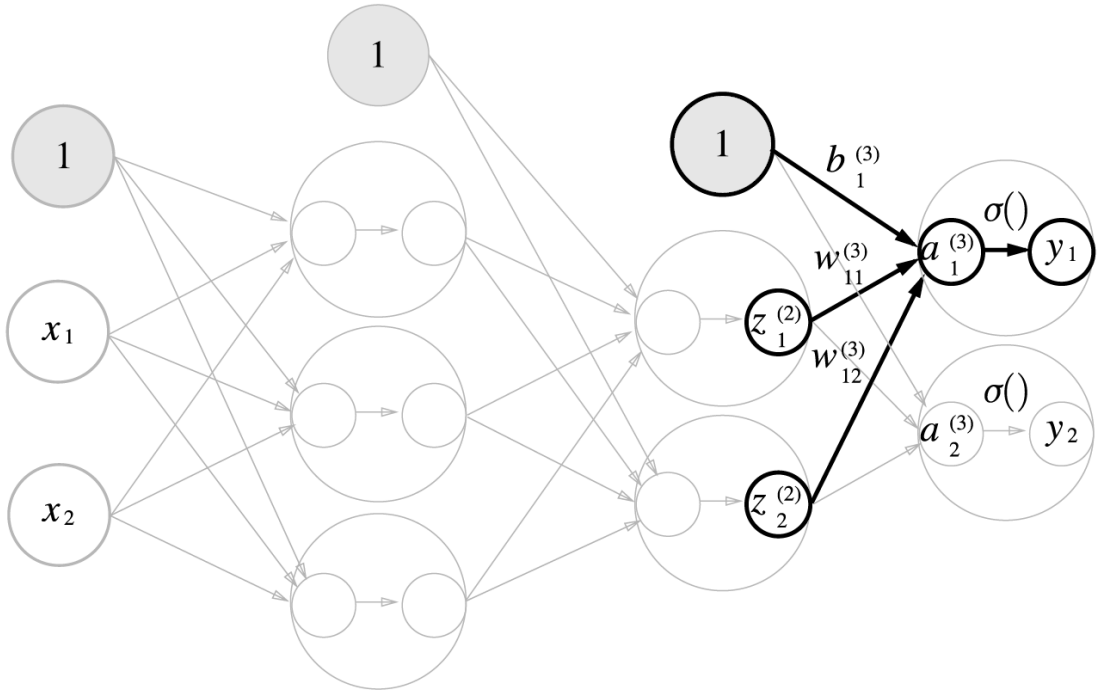
✓ 1층에서 2층으로의 신호 전달



각 층의 신호 전달 구현

✓ **2층에서 출력층으로의 신호 전달**

- 출력층의 활성화 함수로 항등 함수 사용



신경망 순방향 전달(순전파)

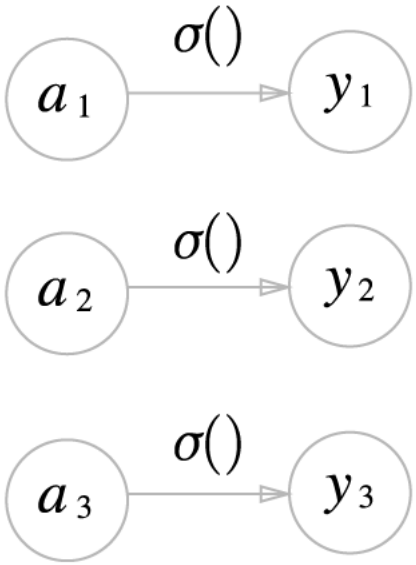
- ✓ 구현 정리

출력층 설계하기

- ✓ **신경망 출력층의 활성화 함수**
  - 회귀 문제
    - 항등 함수
  - 분류 문제
    - 2 클래스 분류: 시그모이드 함수
    - 다중 클래스 분류: 소프트맥스 함수

출력층 설계하기

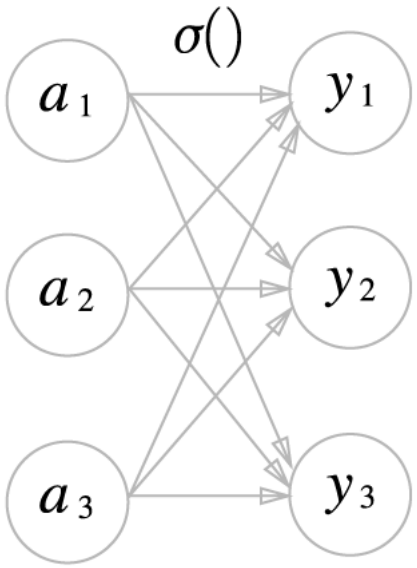
- ✓ **항등 함수(identity function)**
  - 입력을 그대로 출력



출력층 설계하기

✓ **소프트맥스(softmax) 함수**

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




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

    return y
```

출력층 설계하기

✓ **소프트맥스 함수 구현 시 주의점**

- 오버플로우(overflow) 문제
  - 지수 값이 쉽게 커지기 때문
- 소프트맥스 함수 개선

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

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

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

- 지수 함수 계산 시 어떤 수를 더하거나 빼도 결과는 똑같음을 보여줌

출력층 설계하기

✓ **개선된 소프트맥스 함수**

- C' 에 어떤 값도 괜찮으나 오버플로우를 방지할 목적이라면
  - 입력 신호 중 최댓값으로 설정

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

>>> c = np.max(a)
>>> a - c
>>> np.exp(a - c) / np.sum(np.exp(a -c))

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

출력층 설계하기

✓ 소프트맥스 함수의 특징

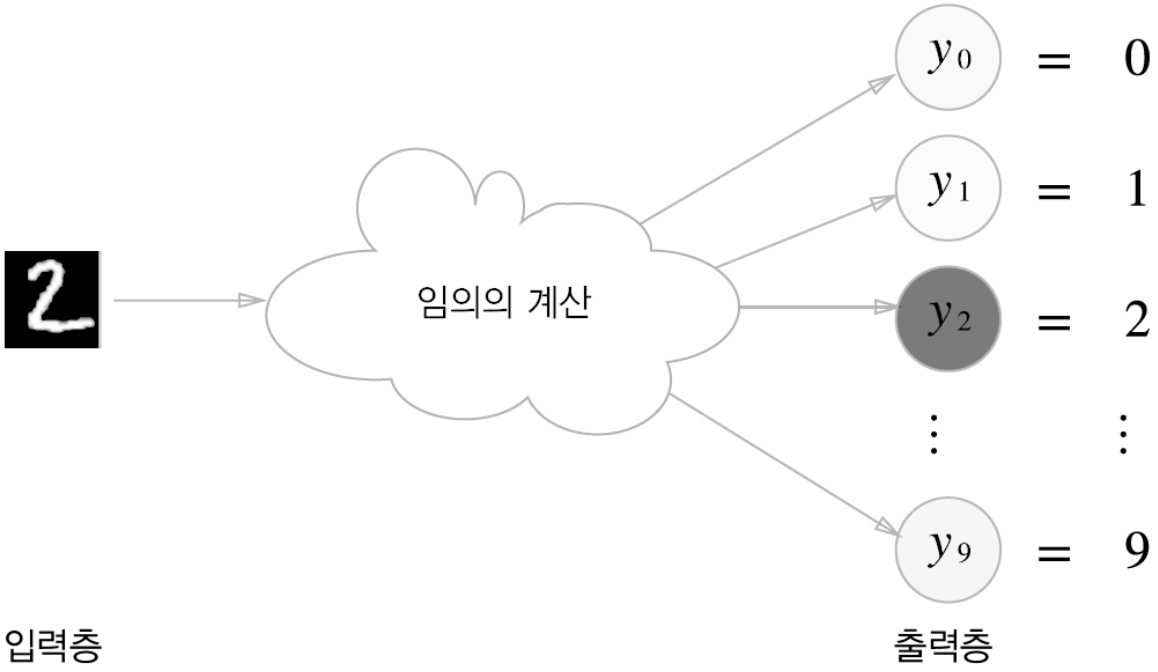
- 출력값 범위
  - [0, 1.0)
- 출력의 총합
  - 1

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

- 출력을 확률로 해석할 수 있음
  - "2번째 원소의 확률이 가장 높으니, 답은 2번째 클래스다."
  - "74%의 확률로 2번째 클래스, 25%의 확률로 1번째 클래스, 1%의 확률로 0번째 클래스다"
  - 문제를 확률적(통계적)으로 대응할 수 있음
- 대소 관계가 변하지 않음
  - 신경망으로 분류 시 출력층의 소프트맥스 함수 생략 가능

출력층 설계하기

- ✓ **출력층의 뉴런 수 정하기**
  - 문제에 맞게 적절히 설정
    - 예) 숫자 이미지를 0부터 9 중 하나로 분류



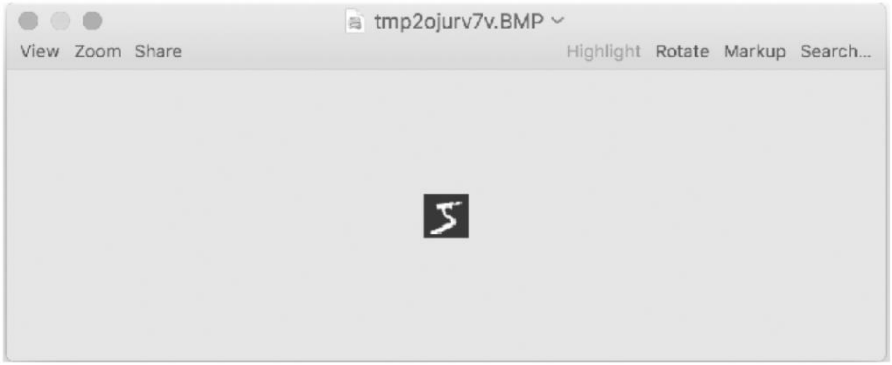
손글씨 숫자 인식

- ✓ **이미 학습된 매개변수(신경망의 지식 표현) 사용**
  - 학습 과정 생략
  - 추론 과정만: 순전파(forward propagation)

- ✓ **MNIST 데이터셋**
  - http://yann.lecun.com/exdb/mnist/
  - 0~9 숫자 이미지
  - 훈련 이미지: 60,000 장
  - 시험 이미지: 10,000 장
  - 크기: 28x28, 회색조(monochrome)
  - 각 픽셀 값: 0~255
  - 이미지가 실제 의미하는 숫자가 레이블로 붙어 있음

손글씨 숫자 인식

- ✓ **mnist.load_minst**
  - MNIST 데이터 ➔ (훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)
  - normalize
    - 입력 이미지 픽셀 값을 [0.0 ~ 1.0] 값으로 정규화 여부
  - flatten
    - 입력 이미지를 1차원 배열로 변환 여부
  - one_hot_label(원-핫 인코딩)
    - 정답에 해당하는 원소만 1
    - 나머지는 모두 0인 배열
- ✓ **mnist_show.py**



손글씨 숫자 인식

- ✓ **손글씨 숫자 인식을 위한 신경망 설계**
  - 입력층 뉴런: 784개
    - 이미지 크기: 28 x 28 = 784
  - 출력층 뉴런: 10개
    - 0 ~ 9 숫자 구분
  - 첫 번째 은닉층 뉴런: 50개
  - 두 번째 은닉층 뉴런: 100개
- ✓ **nerualnet_mnist.py**
- ✓ **sample_weight.pkl**
  - 학습된 가중치 매개변수: weight, bias

손글씨 숫자 인식

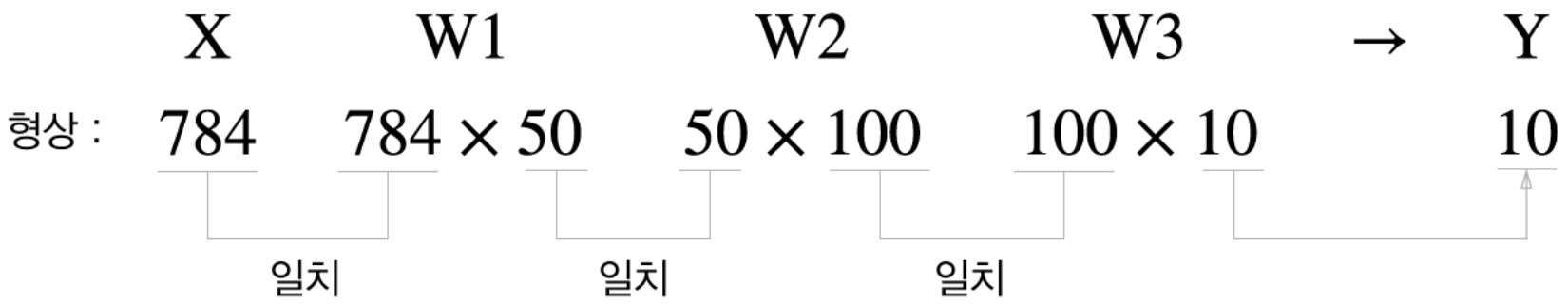
✓ **정확도 평가**

```
>>> 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 인덱스

        if p == t[i]:
            accuracy_cnt += 1

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

손글씨 숫자 인식

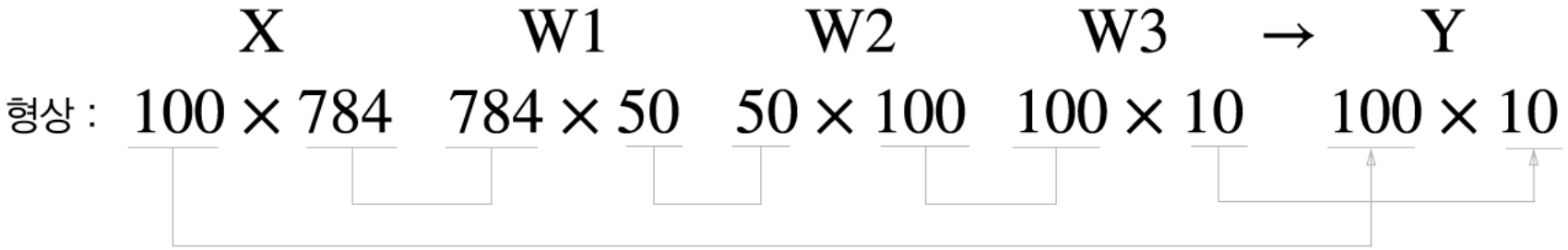
✓ **신경망 각 층의 배열 형상**

```
>>> x, _ = get_data()
>>> network = init_network()
>>> W1, W2, W3 = network['W1'], network['W2'], network['W3']

>>> x.shape
>>> x[0].shape
>>> W1.shape
>>> W2.shape
>>> W3.shape
```



손글씨 숫자 인식

- ✓ **배치 처리(batch processing)**
  - 이미지 여러 장을 입력하는 경우
  - 100장



  - 데이터를 효율적이고 빠르게 처리 가능
    - 수치 계산 라이브러리가 큰 배열을 효율적으로 처리할 수 있도록 고도로 최적화되었기 때문
    - 배치 처리를 통해 I/O 횟수가 줄어들어 빠른 CPU 또는 GPU로 순수 계산하는 비율이 높아짐

손글씨 숫자 인식

✓ 배치 처리(batch processing)

```
>>> 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)))
```

정리

- ✓ 신경망에서는 활성화 함수로 시그모이드 함수와 ReLU 함수 같은 매끄럽게 변화하는 함수를 이용
- ✓ NumPy의 다차원 배열을 잘 사용하면 신경망을 효율적으로 구현 가능
- ✓ 기계학습 문제는 크게 회귀와 분류로 나눌 수 있다
- ✓ 출력층의 활성화 함수로는 회귀에서 주로 항등 함수를, 분류에서는 주로 소프트맥스 함수를 이용
- ✓ 분류에서는 출력층의 뉴런 수를 분류하려는 클래스 수와 같게 설정
- ✓ 입력 데이터를 묶은 것을 배치라 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 훨씬 빠르게 얻을 수 있다.