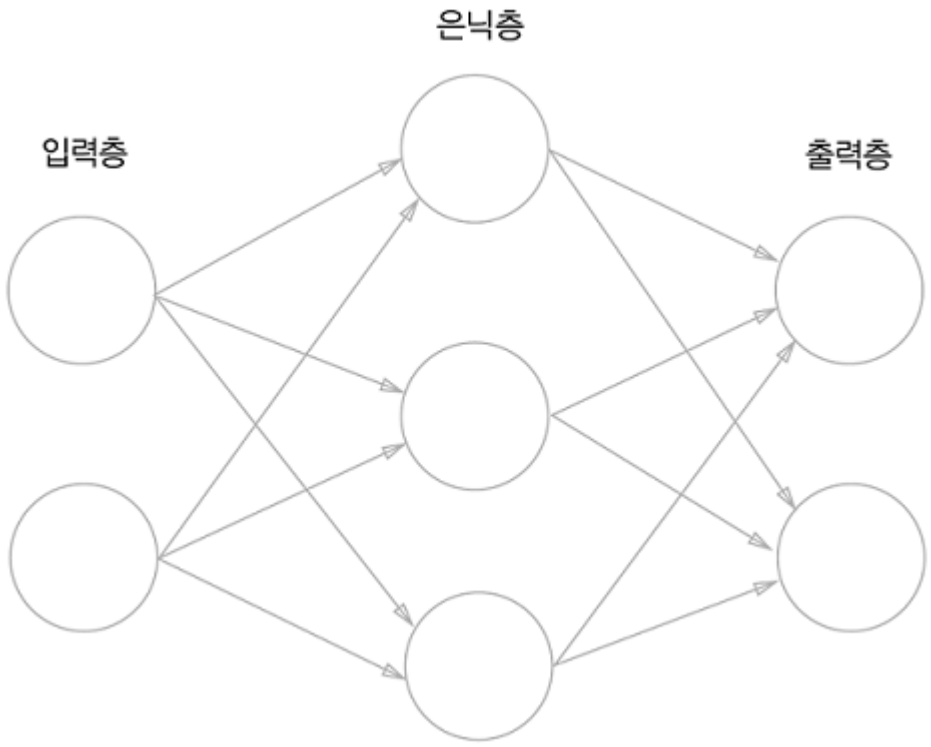
밑바닥부터 시작하는 딥러닝 CHATER 3

신경망이란?

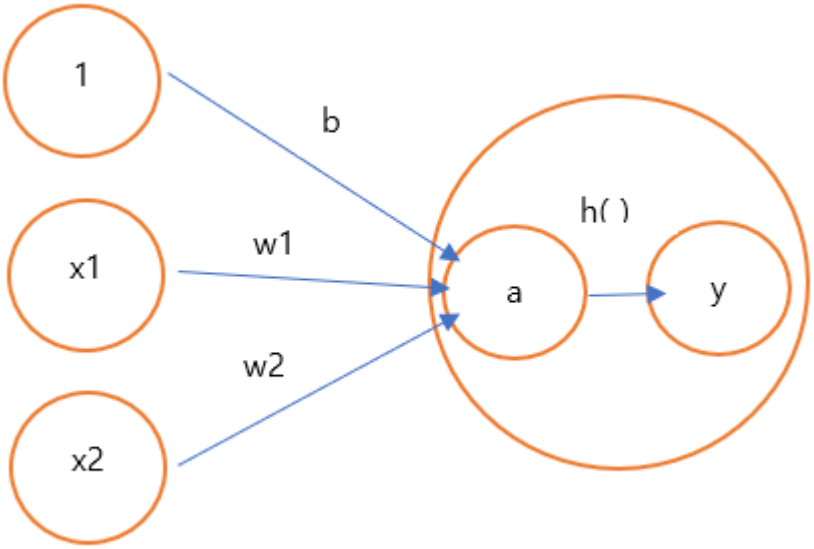


입력층이나 출력층과 달리 은닉층의 뉴런은 사람 눈에는 보이지 않는다.

0층을 입력층, 1층을 은닉층, 2층을 출력층이라 한다.

위의 그림 속 신경망은 3층으로 구성되어있지만 가중치를 갖는 층이 2개이기 때문에 *2층 신경망* 이라고 한다.

활성화 함수(activateion function) : 입력 신호의 총합을 출력 신호로 변환하는 함수. 변환된 신호를 다음 뉴런에 전달한다. 입력 신호의 총합이 활성화를 일으키는지를 정하는 역할을 한다.



> a = b+w1×1+w2×2 #가중치가 달린 입력 신호와 편향의 총합
>
> y=h(a) #a를 함수 h()에 넣어 y를 출력

단순 퍼셉트론 : 단층 네트워크에서 *계단 함수* (임계값을 경계로 출력이 바뀌는 함수)를 활성화 함수로 사용한 모델

다층 퍼셉트론 : *신경망* (여러 층으로 구성되고 시그모이드 함수 등의 매끈한 활성화 함수를 사용하는 네트워크)

활성화 함수의 종류

시그모이드 함수(sigmoid function)

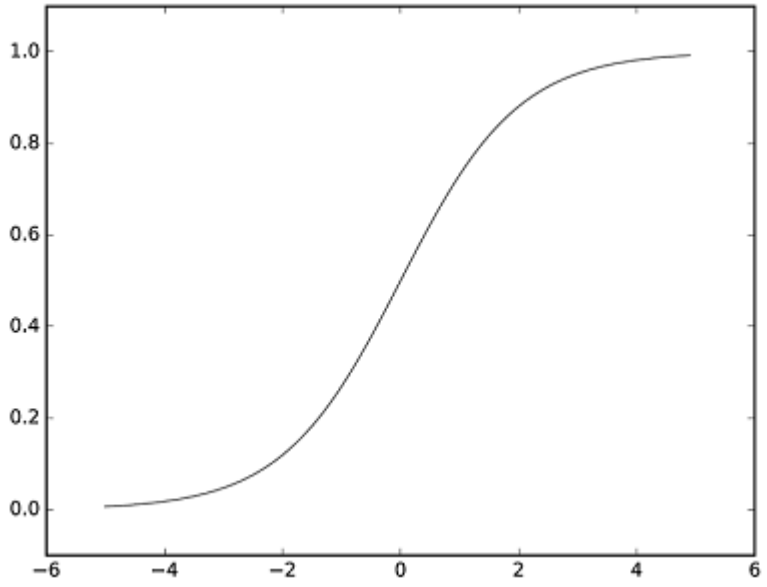
$$h(x) = \frac{1}{1 + \exp(-x)}$$

신경망에서는 활성화 함수로 시그모이드 함수를 이용하여 신호를 변환하고 그 변환된 신호를 다음 뉴런에 전달한다.

- 시그모이드 함수 구현하기브로드캐스트 기능: 넘파이 배열과 스칼라값의 연산을 넘파이 배열의 원소 각각과 스칼라값의 연산으로 바꿔 수행한다.

```
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)
#브로드캐스트
#넘파이 배열과 스칼라값의 연산을 넘파이 배열의 원소 각각과 스칼라값의 연산으로 바꿔 수행

plt.plot(x, y)
plt.ylim(-0.1, 1.1)
plt.show()
```

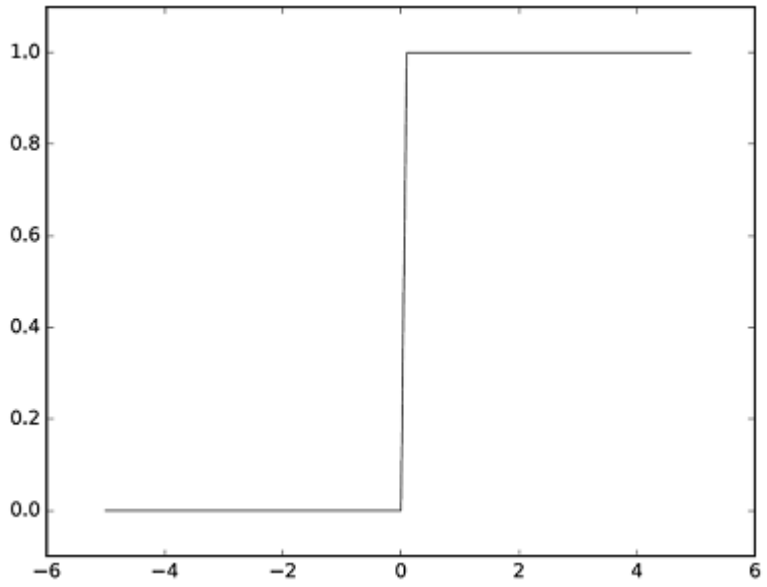


계단 함수

- 계단 함수 구현하기

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

def step_function(x):
    return np.array(x>0, dtype=np.int)
    ##numpy 배열을 인수로 넣을 수 있게 하는 방법
    #x = np.array([-1.0, 1.0, 2.0])
    #y = x>0
    #y를 출력하면 0보다 큰 x값은 True로, 0보다 작거나 같은 값은 Fals
```

```
e로 나온다.
    #booleaan값을 int형으로 변환시키면 True는 0, False는 1이다.

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()
```



계단함수 : 0과 1 중 하나의 값만 돌려준다. -> 뉴런 사이에 0 혹은 1이 흐른다.

시그모이드 함수 : 실수를 돌려준다. -> 연속적인 실수가 흐른다.

- **계단 함수와 시그모이드 함수의 공통점**
  - 입력이 중요하면 큰 값을 출력하고 입력이 중요하지 않으면 작은 값을 출력한다. (입력이 작을 때의 출력은 0에 가깝고 혹은 0이고, 입력이 커지면 출력이 1에 가까워지는 혹은 1이 되는 구조이다.)
  - (입력이 아무리 작거나 커도) 출력은 0에서 1사이이다.
  - 비선형 함수(직선 1개로는 그릴 수 없는 함수)이다.

```
        - 선형함수의 문제: 층을 아무리 깊게 해도 '은닉층이 없는 네트워크'로도 똑같은 기능을 할 수 있다. 즉 선형 함수를 이용해서는 여러 층으로 구성하는 이점을 살릴 수 없다.
```

예) h(x) = cx를 활성화 함수로 사용한 3층 네트워크를 나타내면

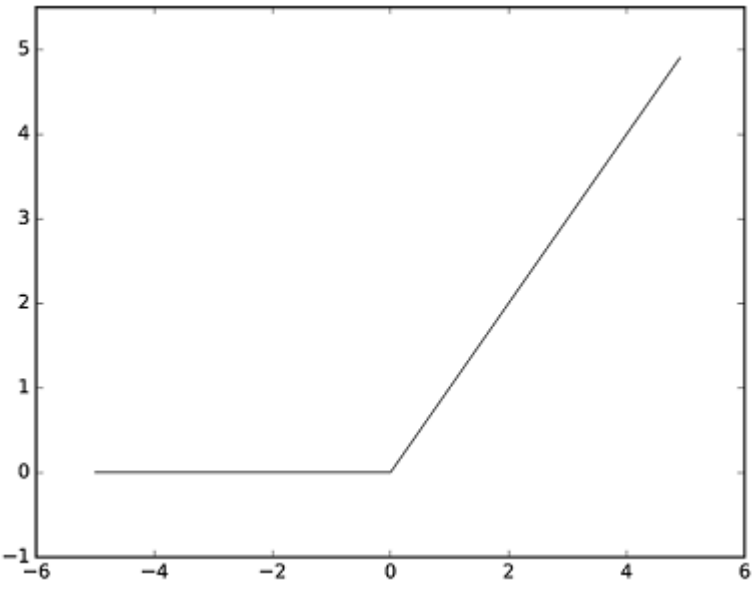
y(x) = h(h(h(x)))가 된다. 하지만 y(x) = ax와 똑같은 식이다. a=x^3일 뿐.

ReLU함수(rectified linear unit)

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

입력이 0을 넘으면 그 입력을 그대로 출력하고 0이하이면 0을 출력하는 함수

```
import numpy as np
import matplotlib.pyplot 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.show()
```

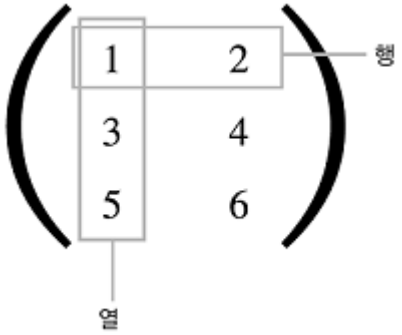


다차원 배열의 계산

`np.ndim()` : 배열의 차원 수 확인

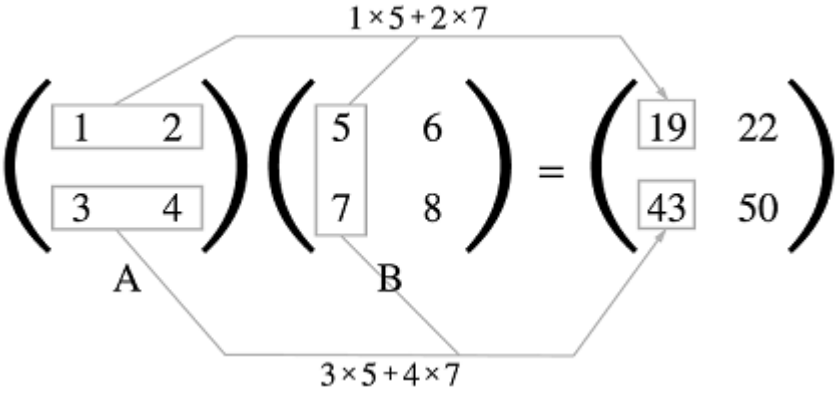
`배열.shape` : 배열의 형상 확인

```
import numpy as np

#1차원 배열
A = np.array([1,2,3,4])
np.ndim(A)
A.shape #튜플로 반환

#2차원 배열
B = np.array([[1,2], [3,4], [5,6]])
np.ndim(B)
B.shape
```



2차원 배열 은 특히 *행렬* 이라 부르고 배열의 가로 방향을 *행* , 세로 방향을 *열* 이라고 한다.

행렬의 곱

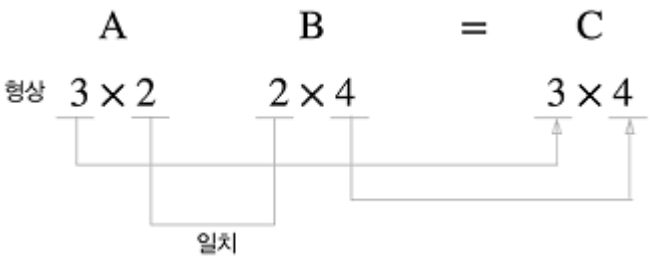



```
#위 그림을 파이썬으로 구현
A = np.array([[1,2], [3,4]]) #2*2행렬
```

```
B = np.array([[5,6], [7,8]]) #2*2행렬
np.dot(A, B)
```

`np.dot()` : 입력이 1차원 배열이면 벡터를, 2차원 배열이면 행렬 곱을 계산한다.

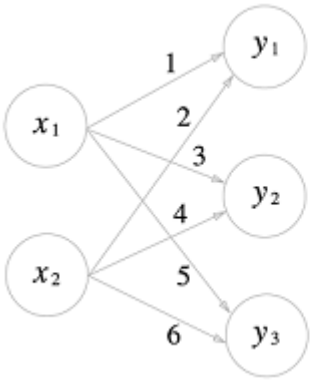
`np.dot(A, B)`와 `np.dot(B, A)`는 다른 값이 될 수 있다.



행렬A의 1번째 차원의 원소 수(열 수) 와 *행렬 B의 0번째 차원의 원소 수(행 수)* 가 같아야 한다.

```
A = np.array([[1,2,3], [4,5,6]]) #2*3행렬
A.shape

B = np.array([[1,2], [3,4], [5,6]]) #3*2행렬
B.shape

np.dot(A, B)
```

신경망에서의 행렬 곱



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

X W = Y

2 2 × 3 3

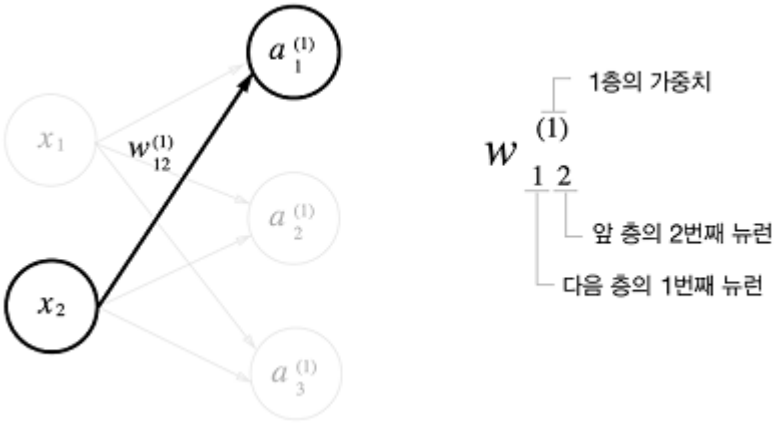
일치

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

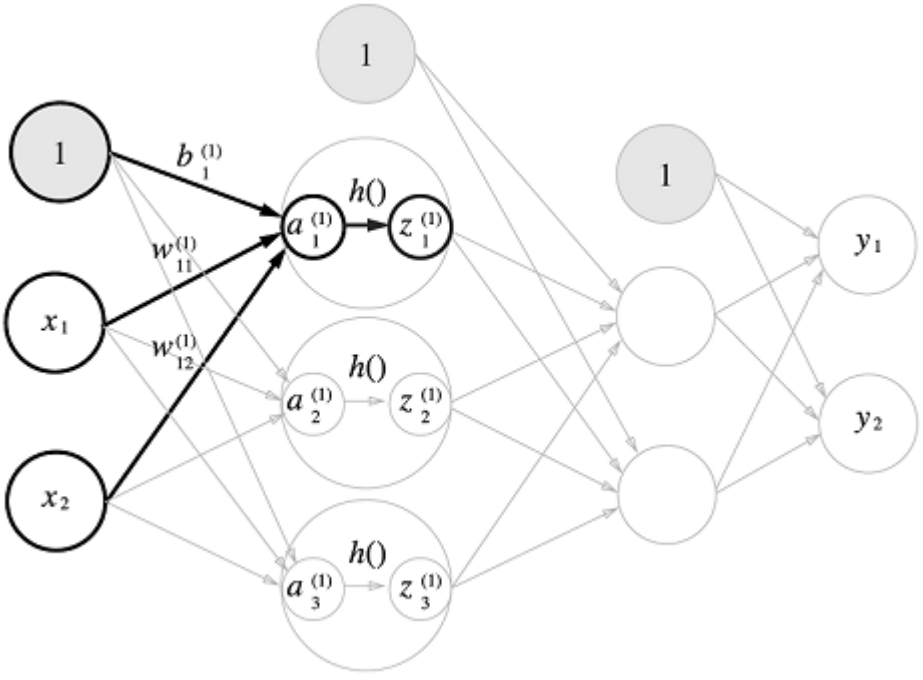
```
W = np.array([[1,3,5], [2,4,6]])
print(W)
W.shape #(2,3)

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

3층 신경망 구현하기

표기법



각 층의 신호 전달 구현하기



은닉층에서의 가중치 합(가중 신호와 편향의 총합)을 a로 표기하고 활성화 함수 h( )로 변환된 신호를 z로 표현한다.

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

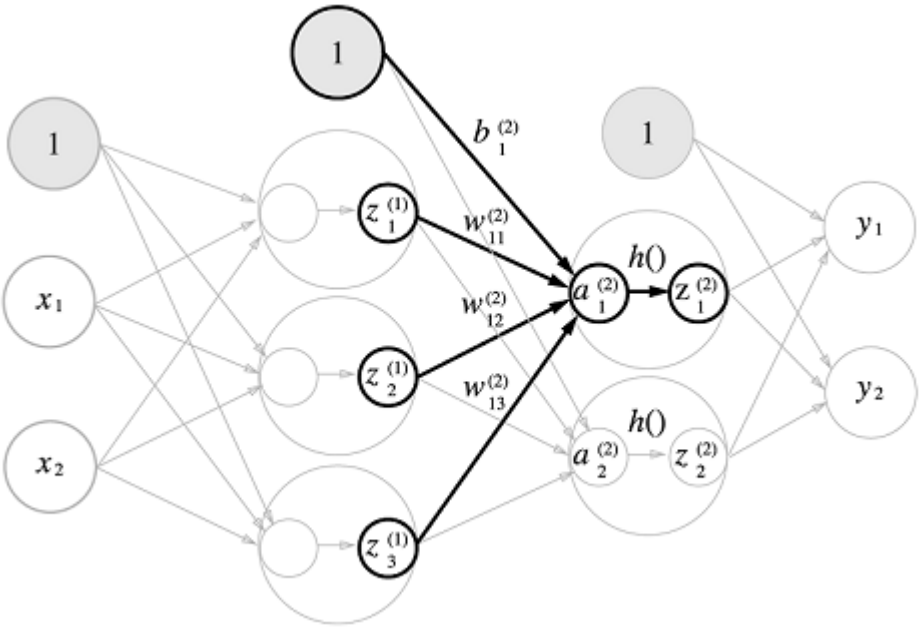
1층의 '가중치 부분'을 행렬식으로 간소화하면

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

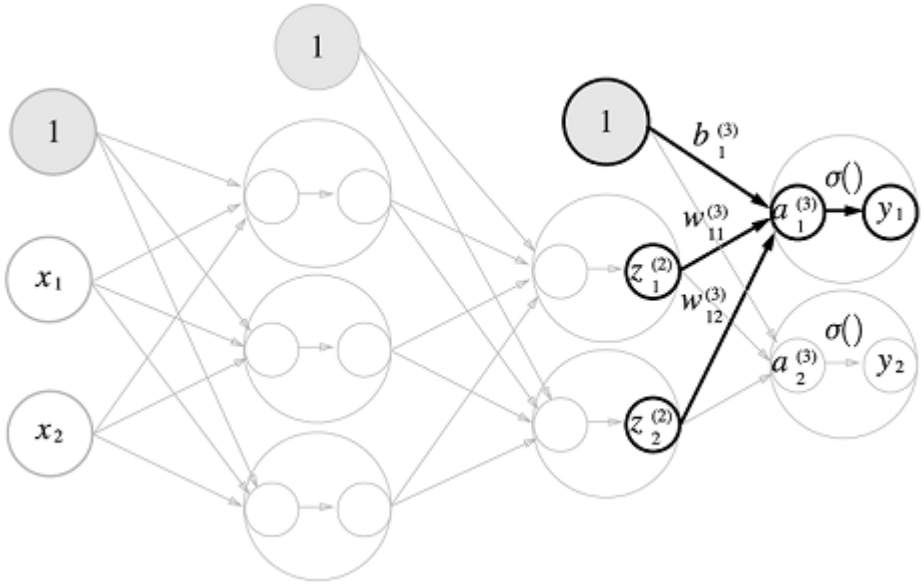
$$A^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix} \quad X = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$$

$$B^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix} \quad W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

```
#입력층에서 1층으로의 신호 전달
X = np.array([1.0, 0.5]) #1*2
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]]) #2*3
B1 = np.array([0.1, 0.2, 0.3]) #1*3

A1 = np.dot(X, W1) + B1 #1*3
#print(A1)

#활성화 함수로 시그모이드 함수를 사용하기로 했을 때
Z1 = sigmoid(A1) #1*3
#print(Z1)
```




```
#1층에서 2층으로의 신호 전달
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]]) #3*2
B2 = np.array([0.1, 0.2]) #1*2

A2 = np.dot(Z1, W2) + B2 #1*2
Z2 = sigmoid(A2) #1*2
```




```
#2층에서 출력층으로의 신호 전달
#출력층의 활성화 함수를 identity_function으로 정의
```

```
def identity_function(x):
  return x

W3 = np.array([[0.1, 0.3], [0.2, 0.4]]) #2*2
B3 = np.array([0.1, 0.2]) #1*2

A3 = np.dot(Z2, W3) + B3 #1*2
Y = identity_function(A3)
```

일반적으로 *회귀* 에서는 출력층의 활성화 함수를 *항등함수* 로, *2클래스 분류* 에서는 *시그모이드 함수* 로, *다중 클래스 분류* 에서는 *소프트맥스 함수* 로 사용한다.

3층 신경망 구현 정리

```
def init_network():
  network = {}
  network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.
6]]) #2*3
  network['b1'] = np.array([0.1, 0.2, 0.3]) #1*3
  network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3,
0.6]]) #3*2
  network['b2'] = np.array([0.1, 0.2]) #1*2
  network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]]) #2*2
  network['b3'] = np.array([0.1, 0.2]) #1*2

  return network

#신호가 순방향(입력에서 출력 방향)으로 전달됨(순전파)임을 알리기 위함이
다.
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]) #1*2
y = forward(network, x)
print(y)
```

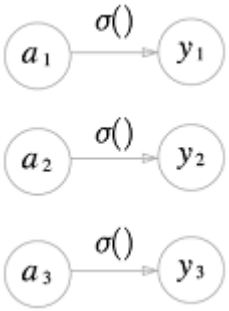
출력층 설계하기

기계학습 문제는 **분류(classification)**와 **회귀(regression)**로 나뉜다.

일반적으로 **회귀**에는 **항등함수**를 **분류**에는 **소프트맥스 함수**를 출력층의 활성화 함수로 사용한다.

3.4.1 항등 함수와 소프트맥스 함수 구현하기

- 항등함수(identity function): 입력을 그대로 출력



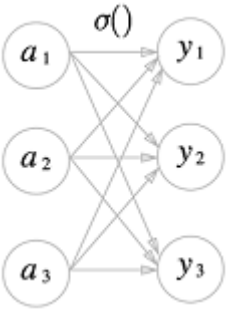
- 소프트맥스 함수(softmax function):

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

소프트맥스 함수의 분자는 입력 신호의 지수함수, 분모는 모든 입력 신호의 지수 함수의 합으로 구성된다.

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

```
  return y
```



소프트맥스 함수 구현 시 주의점

지수함수를 사용하는 소프트맥스 함수는 '오버플로'의 문제가 발생해 수치가 '불안정'해질 수 있는 문제점이 있다.

- 오버플로(overflow) : 표현할 수 있는 수의 범위가 한정되어 너무 큰값은 표현할 수 없다.
- 소프트맥스 함수 구현 개선

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

  - > 소프트맥스의 지수 함수를 계산할 때 어떤 정수를 더해도(혹은 빼도) 결과는 바뀌지 않는다.
  - > C '에 어떤 값을 대입해도 상관없지만 오버플로를 막기 위해 입력 신호 중 최댓값을 이용하는 것이 일반적이다.

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

```
    return y
```

소프트맥스 함수의 특징

- 출력값은 0에서 1.0 사이의 실수이다.
- 출력의 총합은 1이다. -> **확률**로 해석가능
- 지수함수가 단조 증가 함수 이기 때문에 소프트맥스 함수를 적용해도 각 원소의 대소 관계는 변하지 않는다. 결과적으로 신경망으로 분류할 때는 출력층의 소프트맥스 함수를 생략해도 된다.

출력층의 뉴런 수 정하기

출력층의 뉴런 수는 풀려는 문제에 맞게 적절히 정해야 한다. *분류* 에서는 분류하고 싶은 클래스 수로 설정하는 것이 일반적이다.

손글씨 숫자 인식

- 신경망의 문제 해결 단계1. 학습 : 훈련 데이터(학습 데이터)를 사용해 가중치 매개변수를 학습한다.2. 추론 : 학습한 매개변수를 사용하여 입력 데이터를 분류한다.

신경망의 순전파(forward propagation): 이미 학습된 매개변수를 사용하여 입력 데이터를 분류하는 추론 과정

MNIST 데이터셋

- 28*28 크기의 회색조 이미지(1채널)
- 각 픽셀은 0~255까지의 값을 취한다.
- 각 이미지에 실제 의미하는 숫자가 레이블로 붙어 있다.

밑바닥부터 시작하는 딥러닝 깃허브

```
import sys, os
#sys.path.append(os.pardir)
#from dataset.mnist import load_mnist
from mnist import load_mnist #mnist.py만을 다운받았다면

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=T
rue, normalize=False)
#normalize: 입력 이미지의 픽셀값을 0.0~1.0 사이의 값으로 정규화할지
```

```
정함
#flatten: 입력 이미지를 1차원 배열로 만들지를 정함
#(False: 1*28*28 3차원 배열로 True: 784개의 원소로 이루어진 1차원 배열로)
#one-hot-label: 원-핫 인코딩 형태로 저장할지를 정함
#(False: 숫자 형태의 레이블을 저장 True: 레이블을 원-핫 인코딩하여 저장)
```

```
import sys, os
sys.path.append(os.pardir)
from dataset.mnist import load_mnist
import numpy as np
from PIL import Image



def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    #Image.fromarray: 넘파이로 저장된 이미지 데이터를 PIL용 데이터 객체로 변환해야
    pil_img.show()

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

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

print(img.shape)
img = img.reshape(28, 28)
#flatten=True는 1차원 넘파이 배열로 저장되어 있다
#이미지를 표시할 때 원래 형상인 28*28크기로 다시 변형해야
print(img.shape)
img_show(img)
```

```
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=T
rue, normalize=False)
```

신경망의 추론 처리

```
import numpy as np
import pickle
from dataset.mnist import load_mnist
from common.functions import sigmoid, softmax


def get_data():
    (x_train, t_train), (x_test, t_test) = load_mnist(norma
lize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    #sample_weight.pkl파일에 저장된 학습된 가중치 매개변수를 읽는다
    #가중치와 편향 매개변수가 딕셔너리 변수로 저장되어 있다.
    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['W
3']
    b1, b2, b3 = network['b1'], network['b2'], network['b
3']

    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)
```

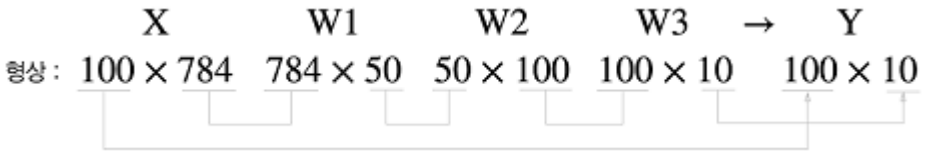
```
    return y

#신경망의 정확도(분류가 얼마나 올바른가)를 평가
x, t = get_data()
network = init_network()

accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i]) #각 레이블의 확률을 넘파이 배열로 반환
    p = np.argmax(y) #배열에서 값이 가장 큰(확률이 가장 높은) 원소의 인덱스를 구함
    if p == t[i]:
        accuracy_cnt += 1

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

배치 처리

이미지 여러 장을 한꺼번에 입력하는 경우

배치(batch): 하나로 묶은 입력 데이터

(이미지 100개를 묶어 predict() 함수에 한 번에 넘긴 경우)



x[0]와 y[0]에는 0번째 이미지와 그 추론 결과가, x[1]과 y[1]에는 1번째의 이미지와 그 결과가 저장된다.

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

batch_size = 100 #배치 크기
accuracy_cnt = 0

#range: 0부터 len(x)까지 batch_size간격으로 증가하는 리스트 반환
```

```
for i in range(0, len(x), batch_size):
  x_batch = x[i:i+batch_size] #0~100, 100~200, ...
  y_batch = predict(network, x_batch) #각 레이블의 확률을 넘파이 배열로 반환
  p = np.argmax(y_batch, axis=1) #배열에서 값이 가장 큰(확률이 가장 높은) 원소의 인덱스를 구함
  #axis=1: 1번째 차원을 구성하는 각 원소에서 최댓값의 인덱스를 찾도록
  accuracy_cnt += np.sum(p == t[i:i+batch_size])

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