밑바닥부터 시작하는 딥러닝

03. 신경망

PPT made by. 차승철
Template designed By. 스마일게이트

INDEX

03. 신경망

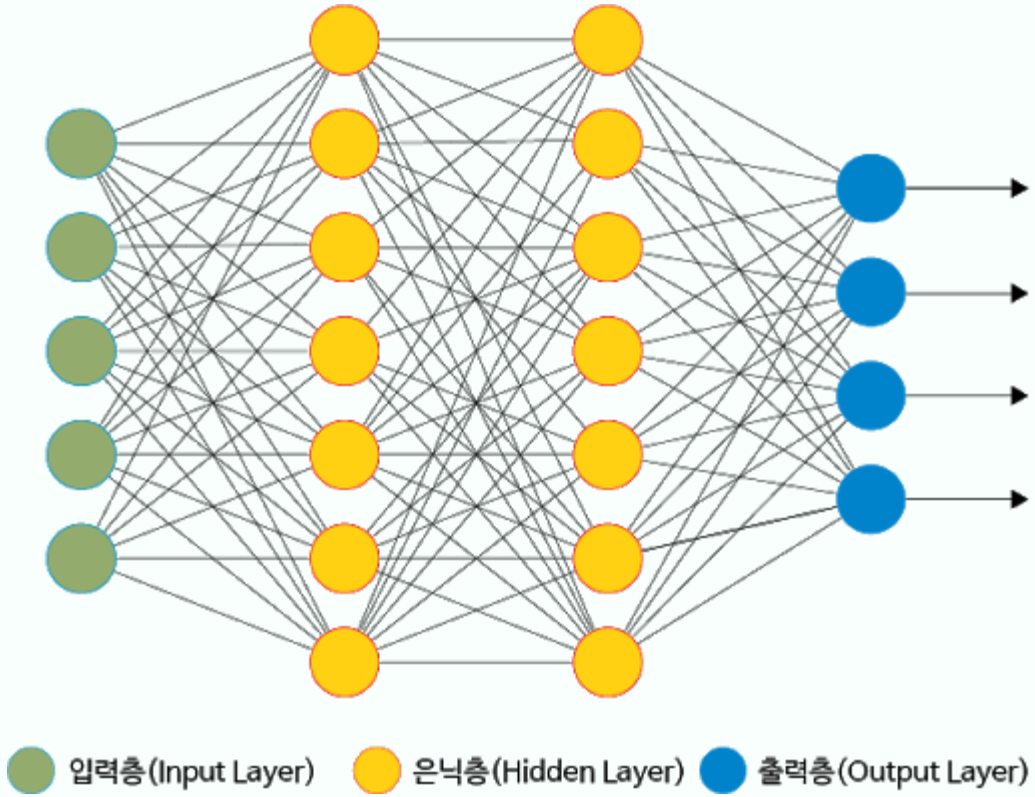
- 신경망이란
- 활성화 함수
- 다차원 배열 계산
- 3층 신경망
- 출력층 설계
- 손글씨 숫자 인식

Ch.03
신경망

1 신경망이란?

01 신경망이란?

퍼셉트론과의 차이를 통해 알아보는 신경망의 정의와 특성

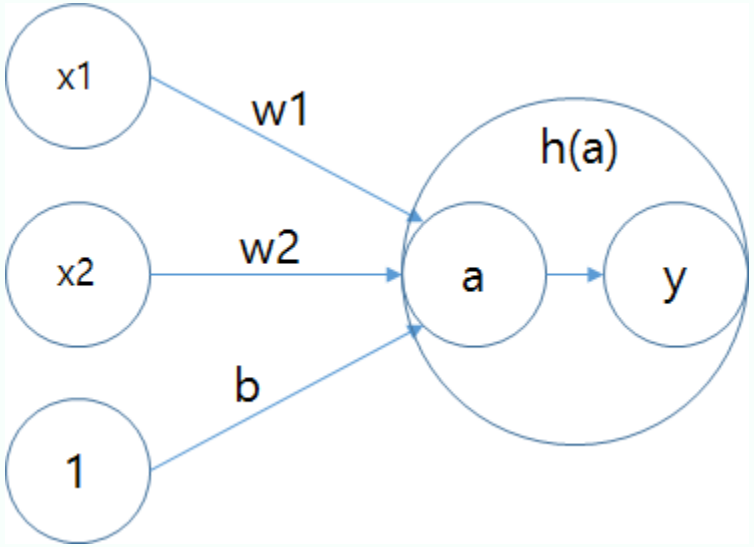


신경망이란

- 퍼셉트론은 복잡한 함수를 표현할 수 있지만, 가중치를 프로그래머가 수동으로 바꿔줘야 한다는 단점이 존재한다.
- 신경망은 적절한 값을 데이터로부터 자동으로 학습하는 능력을 가지고 있어서 이 단점을 해결해준다.
- 즉 신경망 = 복잡한 함수 구현 가능 + 데이터를 이용해서 자동으로 가중치를 구한다.
- 입력층, 은닉층, 출력층으로 구성

01 신경망이란?

퍼셉트론과의 차이를 통해 알아보는 신경망의 정의와 특성



신경망의 특징

- 퍼셉트론에는 다음과 같은 그림을
- $y = h(b + w_1x_1 + w_2x_2)$ 과 같은 수식을 이용해서 표시가 가능.

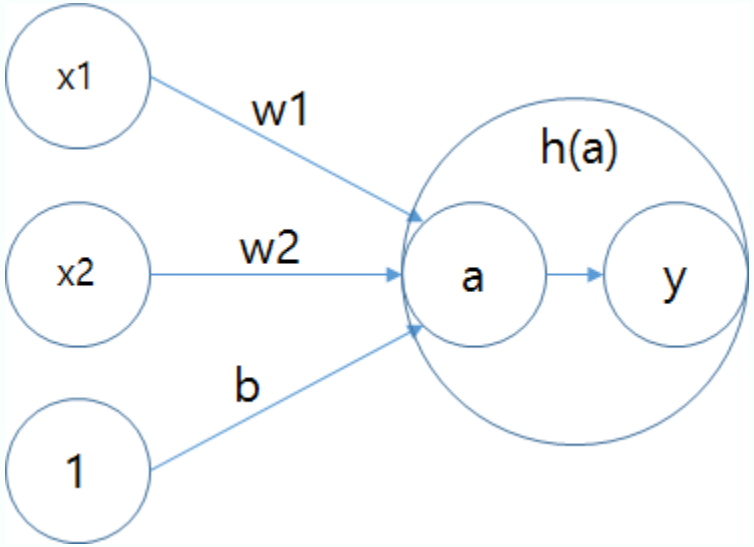
- 이 h(x) 함수를 입력 신호의 총합을 이용해서 출력 신호로 변환하는 이 함수를 **활성화 함수(activation function)**이라고 한다.
- 이는 $a = b + w_1x_1 + w_2x_2$, $y = h(a)$ 로 2단계로 나타낼 수 있으며, 입력에 가중치를 조합한 결과는 a라는 노드가 되며, 이를 이용해서 y 노드를 출력할 수 있습니다.

- 이러한 활성화 함수들이 n층으로 구성되어 다음층으로 계속 넘기는 방식을 신경망이라고 한다!

2 활성화 함수

02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!

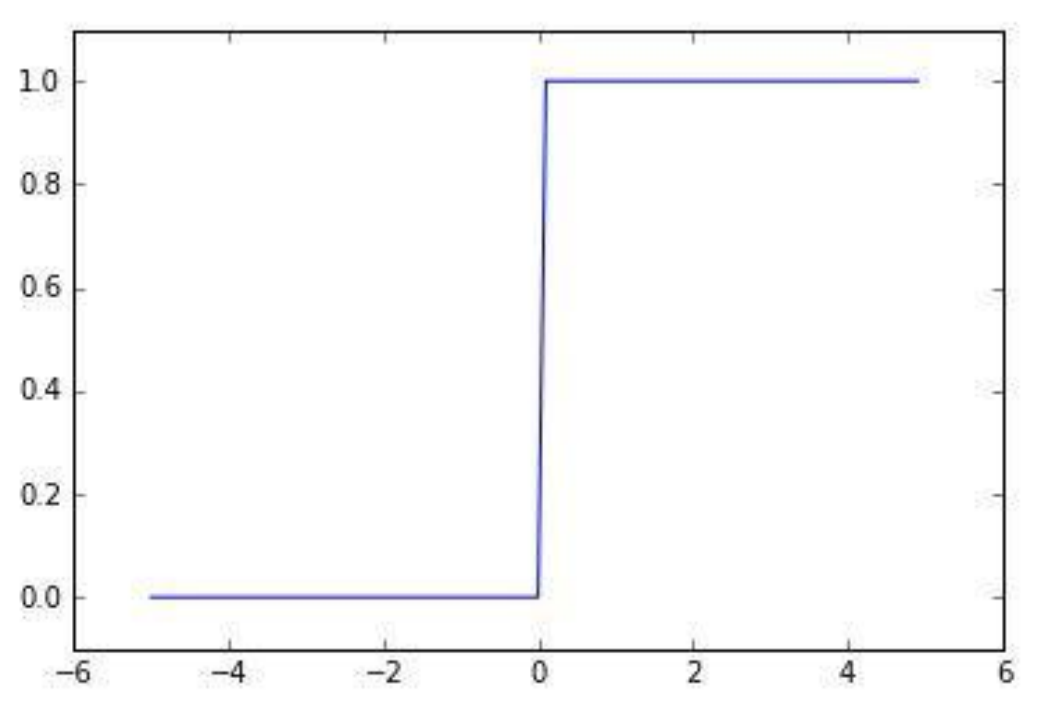


활성화 함수

- 다음 그리처럼 h(x) 함수를 입력 신호의 총합을 이용해서 출력 신호로 변환하는 이 함수를 **활성화 함수(activation function)**이라고 한다.
- 전 층에서 어떤 값을 입력받아 어떤 함수를 거친 뒤 어떤 조건에 맞는 출력을 통해 다음 층으로 값을 전달하는 용도.

02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!



계단 함수

- 정말 간단한 함수
- 만약 입력이 0을 넘어가면 1을 출력하고, 아니라면 0을 출력

02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!

- 구현은 쉬우나, 실수(float, double)형 만 받아들인다.
- 즉 numpy 배열을 인수로 받을 수 없다.

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

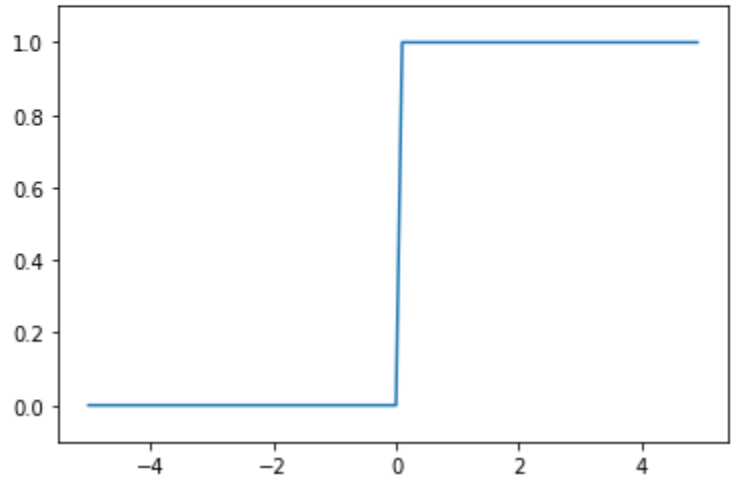
- 이러한 부분은 파이썬의 인터프리터와 넘파이의 트릭을 이용해서 이를 해결

```
# 계단 함수 - 퍼셉트론은 0, 1 중 하나만 흐른다.
def step_function(x) :
    y = x > 0
    return y.astype(np.int)
```

02 활성화 함수

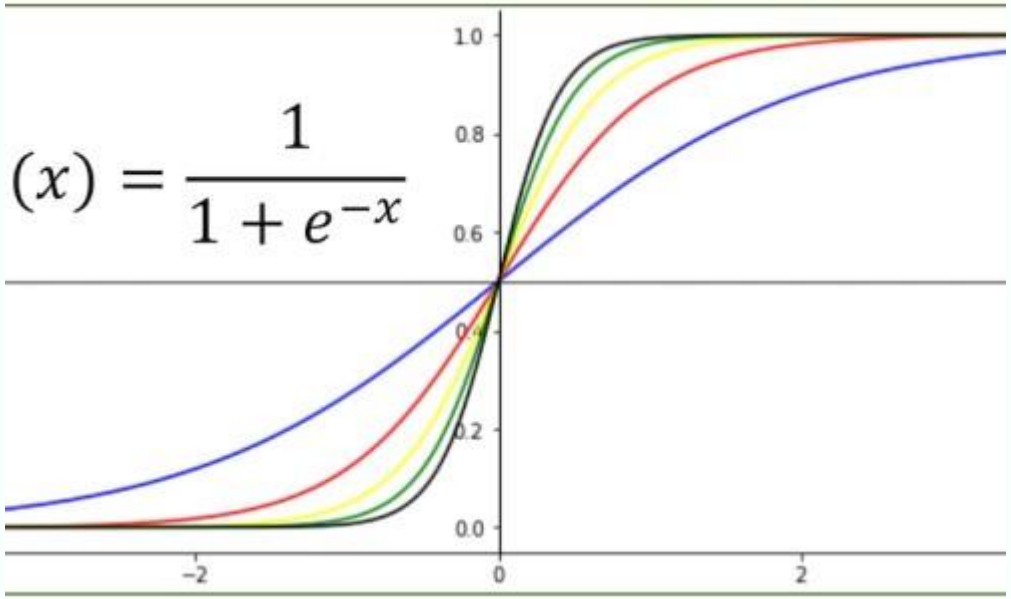
다양한 방식으로 다음 층에 값을 전달하자!

```
In [7]: 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()
```



02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!



◆ 시그모이드 함수

- 신경망에서 자주 이용되는 활성화 함수
- 어떤 입력이 들어오면 정규화처럼 0~1 사이의 실수로 만들어주는 함수
- $h(x) = \frac{1}{1+\exp(-x)}$

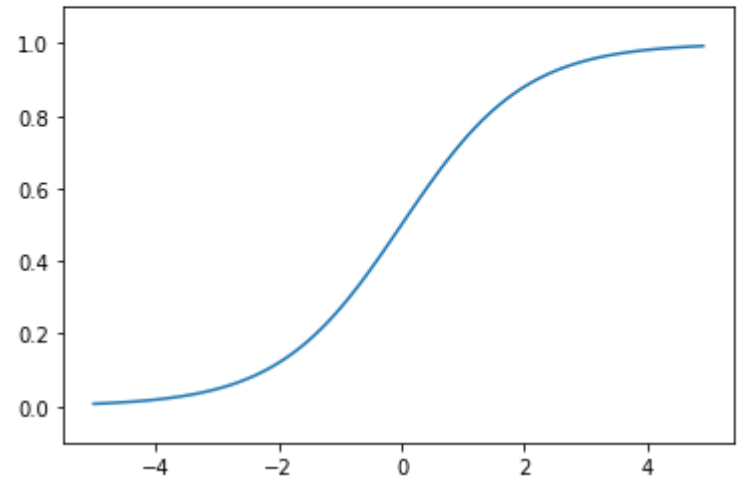
02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!

```python
# 시그모이드(S자 모양) 함수 - 연속적인 실수가 흐른다.
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]
```

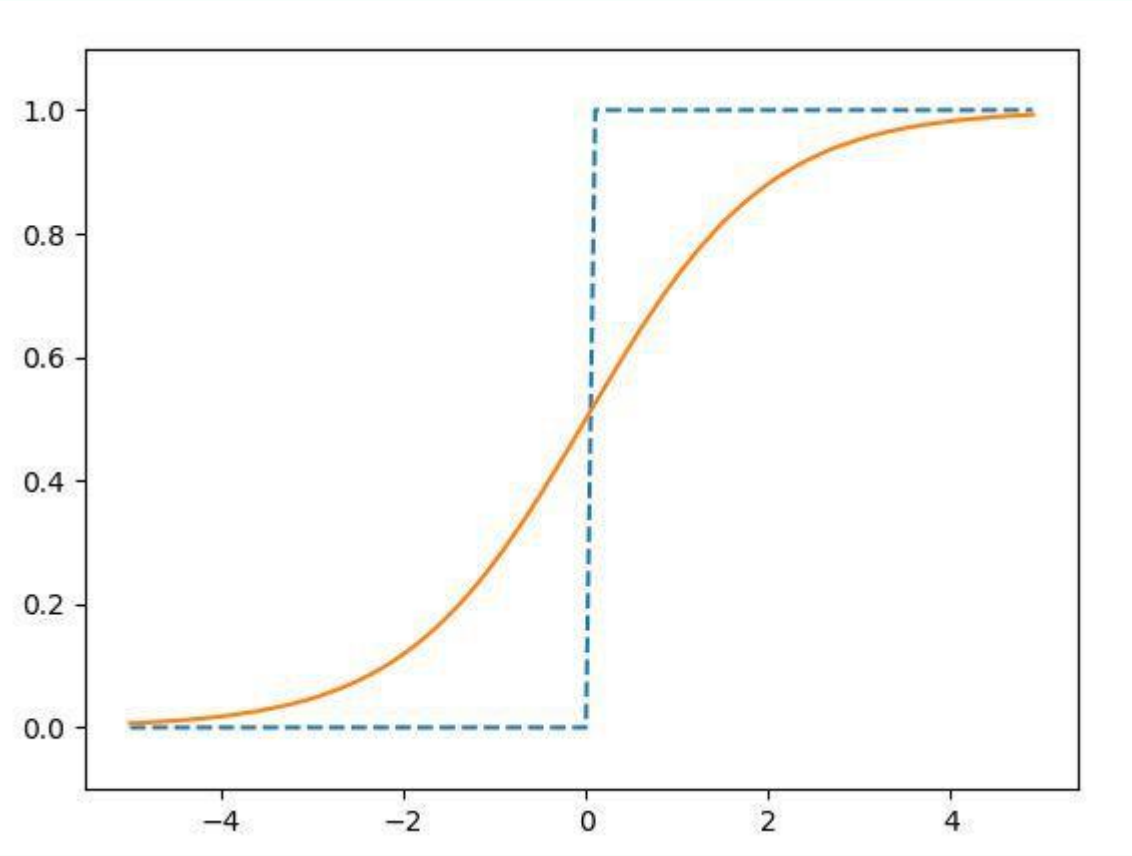
```python
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1)
plt.show()
```



- 넘파이의 exp 함수를 이용해서 구현
- 이를 이용해서 어느정도 정규화된 수를 출력으로 넘기는 것이 가능해졌다.
- 또한 비선형적으로 그리는 것 또한 가능하다.

02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!



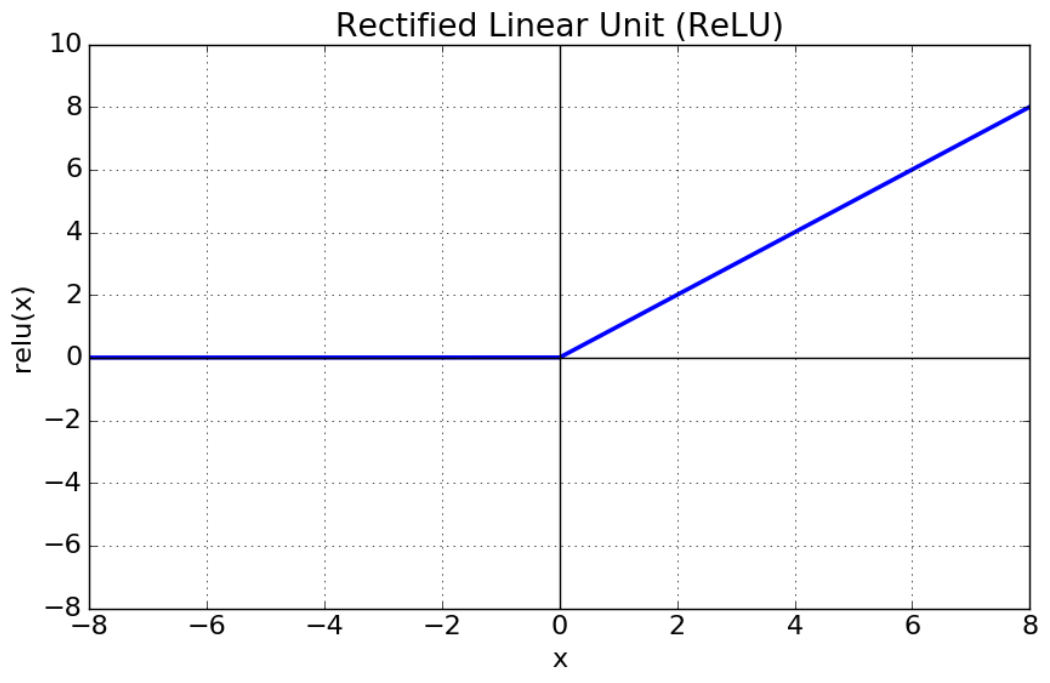
계단 함수와 시그모이드 함수

- 두 함수의 차이는 매끄러움.
- 계단 함수는 0, 1만 출력하나 시그모이드는 실수도 출력한다.

- 하지만 둘 다 동일하게 비선형 함수이다. (곡선과 계단)
- 선형은 은닉층이 없는 네트워크로도 같은 효과를 볼 수 있기 때문에 의미가 없기 때문에, 층을 쌓는 혜택을 보기 위해서는 이 비선형 함수들을 사용해야 한다.

02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!



ReLU 함수

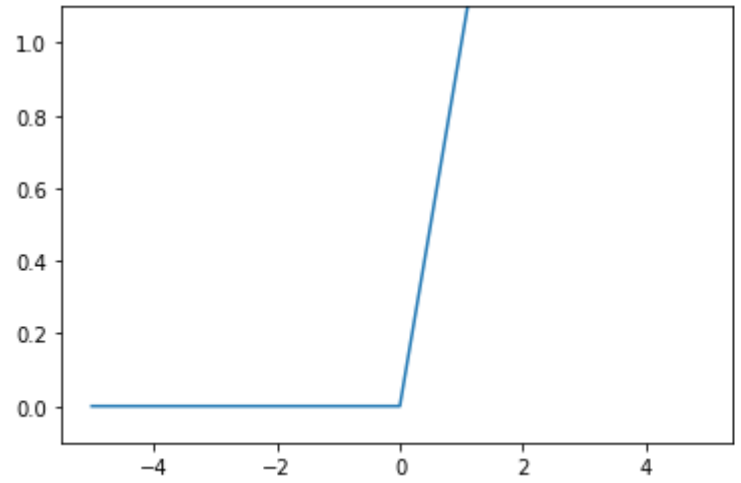
- 영어 명은 Rectified Linear Unit
- 최근 사용되기 시작한 함수
- 입력이 0이 넘어가면 그 입력 그대로 출력, 아니면 0 출력

- $h(x) = x\ (if\ x > 0), 0\ (x \leq 0)$

02 활성화 함수

다양한 방식으로 다음 층에 값을 전달하자!

```python
#ReLU 함수 - 0 이하면 0을 출력하고 그 위면 그 입력을 그대로 출력한다.
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(-0.1, 1.1)
plt.show()
```



3 다차원 배열의 계산

03 다차원 배열의 계산

N차원 배열을 사용할 때는 반드시 원소 수에 주의해야 한다!

```
A = np.array([1,2,3,4])
print(A)   # 배열 내용물 출력
np.ndim(A) # 현 넘파이 배열의 차원
```

```
[1 2 3 4]
```

```
(4,)
```

```
A.shape # n차원에 각각 몇 개의 원소가 들어갔나
```

```
(4,)
```

```
A.shape[0] # 1차원의 내용이 몇 개인가
```

```
4
```

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

```
[[1 2]
 [3 4]
 [5 6]]
```

```
np.ndim(B)
```

```
2
```

```
B.shape
```

```
(3, 2)
```

- ndim함수를 이용해서 배열의 차원 확인이 가능하다.
- Shape를 통해서는 몇 차원이며 몇 개의 원소가 들어갔나를 확인할 수 있다.

03 다차원 배열의 계산

N차원 배열을 사용할 때는 반드시 원소 수에 주의해야 한다!

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

```
array([[19, 22],
       [43, 50]])
```

행렬의 곱

- np.dot 함수를 이용해서 크기가 같은 정방행렬의 곱이 가능하다.

```
A = np.array([[1,2,3],[4,5,6]])
B = np.array([[1,2],[3,4],[5,6]])
np.dot(A,B)
```

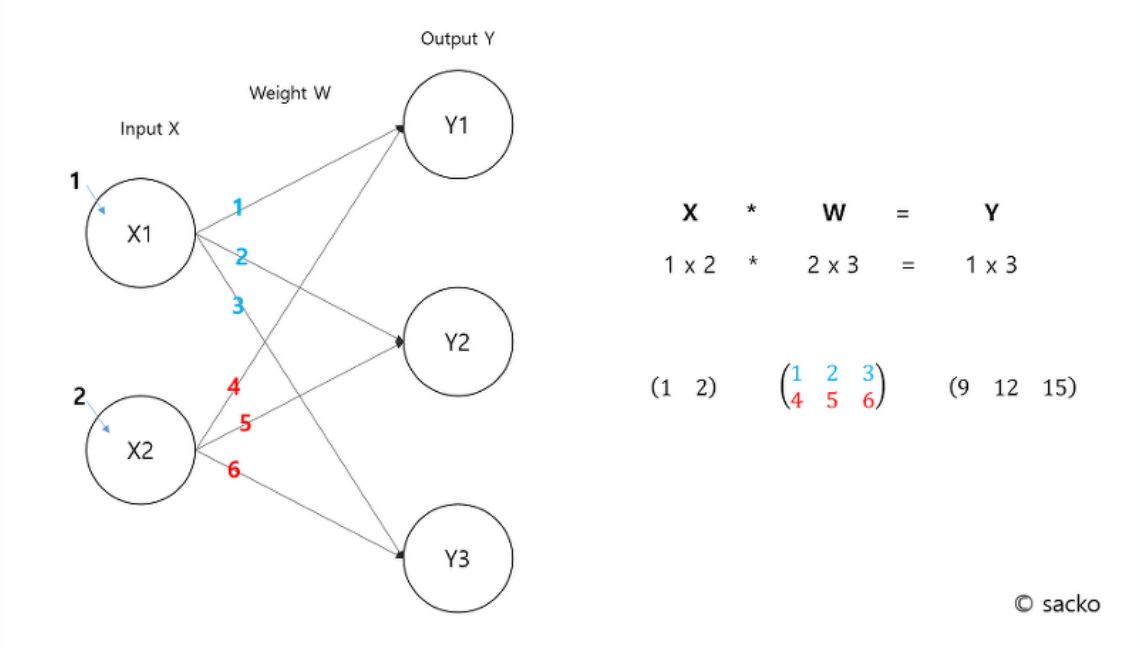
```
array([[22, 28],
       [49, 64]])
```

- n by m m by p 행렬 또한 곱하기가 가능하다.

- 단! 수학상 풀이가 안 되는 행렬은 당연히 연산이 불가능하다.

03 다차원 배열의 계산

N차원 배열을 사용할 때는 반드시 원소 수에 주의해야 한다!



X * W = Y

1 x 2 * 2 x 3 = 1 x 3

$(1 \quad 2) \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad (9 \quad 12 \quad 15)$

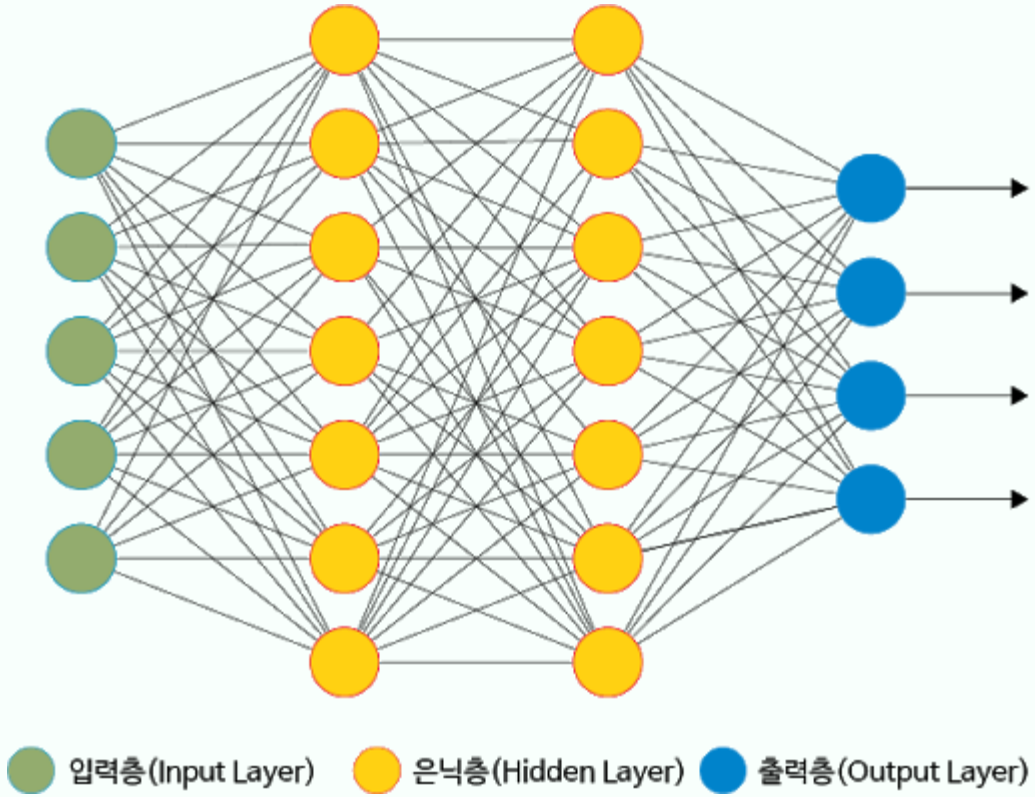
신경망에서 행렬의 곱

- 각 입력 행렬과 출력 행렬의 차원의 원소 수가 대응되야 한다는 점은 반드시 기억해야 한다!

4 3층 신경망 구현

04 3층 신경망 구현

좀 더 그럴싸한 신경망을 만들어보자

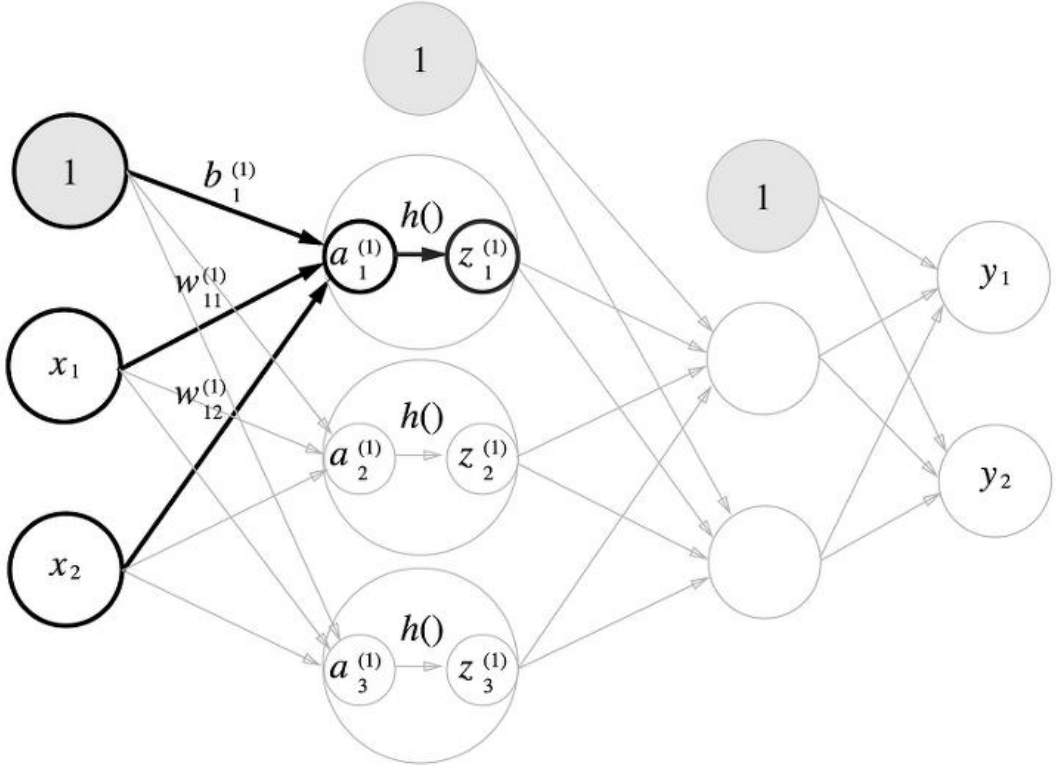


그럴싸한 신경망

- 왼쪽 그림처럼 입력부터 출력까지 순방향으로 처리되는 다차원 배열이 입력값으로 들어오는 신경망을 구축해보자!

04 3층 신경망 구현

좀 더 그럴싸한 신경망을 만들어보자



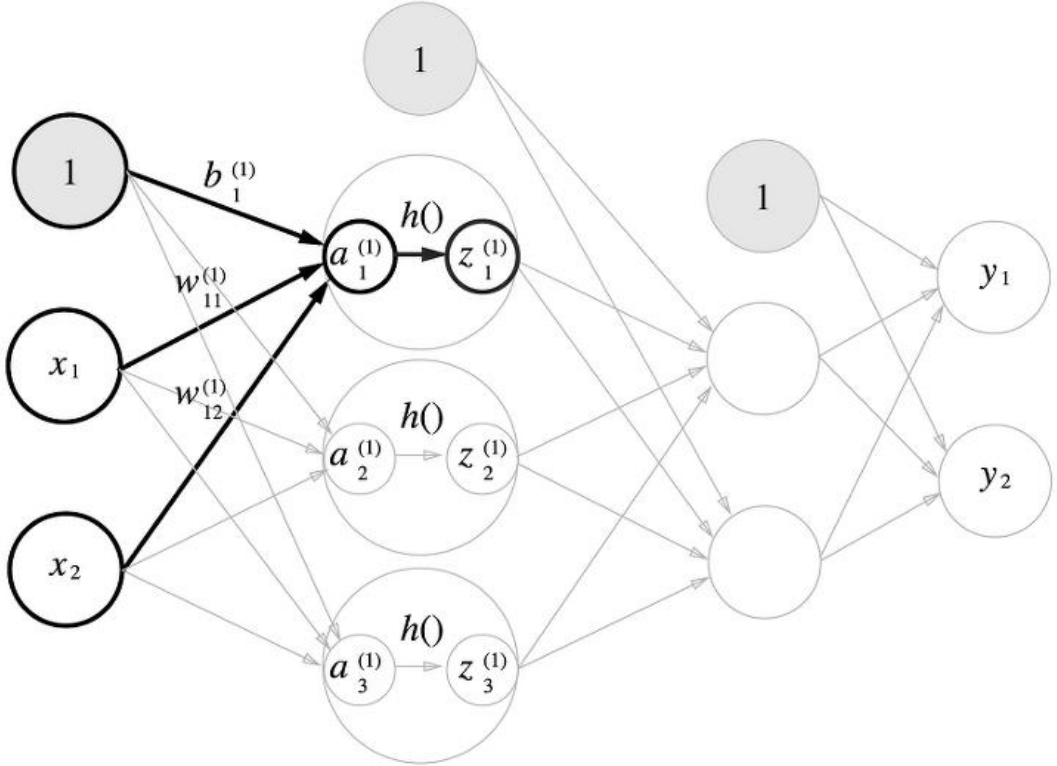

```python
def init_network() :
    network = {}
    network['W1'] = np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
    network['b1'] = np.array([0.1,0.2,0.3])
    network['W2'] = np.array([[0.1,0.4], [0.2,0.5], [0.3,0.6]])
    network['b2'] = np.array([0.1,0.2])
    network['W3'] = np.array([[0.1,0.3], [0.2,0.4]])
    network['b3'] = np.array([0.1,0.2])

    return network
```

은닉층의 역할!

- 다양한 입력과 정해진 편향치를 더한 뒤 이를 활성 함수를 이용해서 또 다른 출력을 만들어내고, 다시 이를 다음 층으로 전달
- 이 때 은닉 층 또한 어느 특정의 편향치를 가지고 있으며, 학습할 때 전 층에서 타고 온 입력 값에 연산된다.

04 3층 신경망 구현

좀 더 그럴싸한 신경망을 만들어보자




```python
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
```

◆ 은닉층의 역할!

- 이 때 연산으로는 전 입력과 가중치를 행렬 곱을 한 뒤 이에 편향치를 더한 뒤 이를 활성함수로 정규화를 한 뒤 출력으로 넘긴다.

04 3층 신경망 구현

좀 더 그럴싸한 신경망을 만들어보자

◆ 시그모이드 함수

```
network = init_network()
x = np.array([[1.0, 0.5],[0.75, 0.25]])
y = forward(network, x)
print(y)
```

```
[[0.31682708 0.69627909]
 [0.31404593 0.69009906]]
```

◆ ReLU 함수

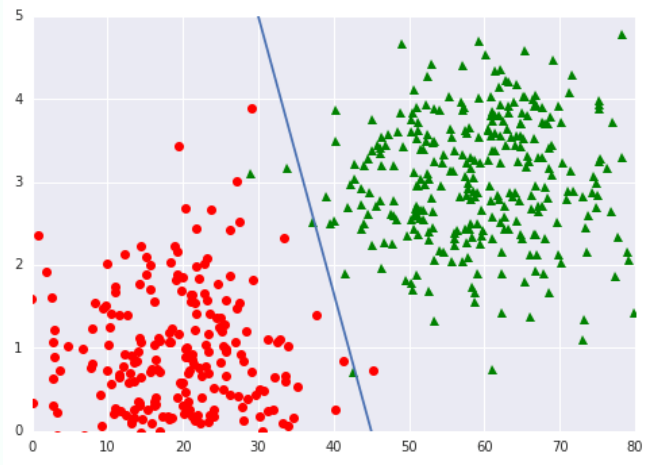
```
network = init_network()
x = np.array([[1.0, 0.5],[0.75, 0.25]])
y = forward_relu(network, x)
print(y)
```

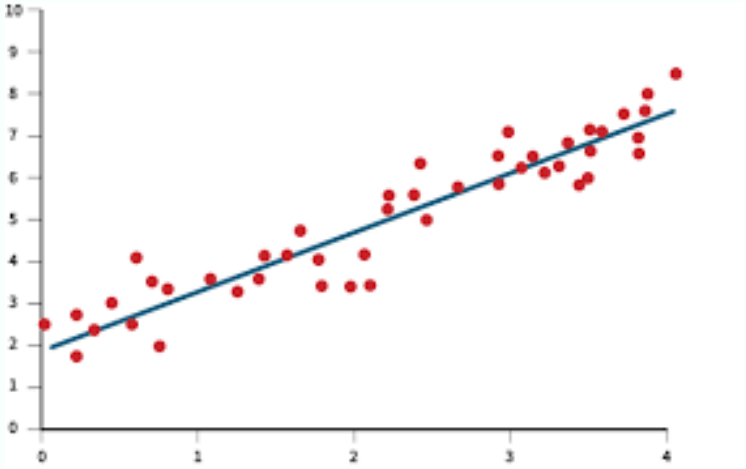
```
[[0.426  0.912 ]
 [0.357  0.7615]]
```

5 출력층 설계

05 출력층 설계

출력은 우리가 원하는 값을 얻을 수 있어야 한다!



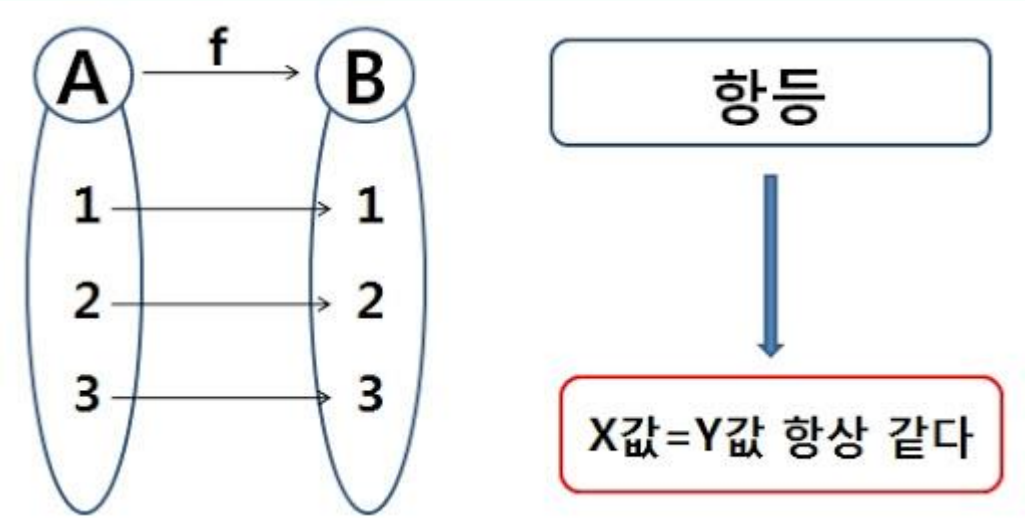


분류와 회귀

- 분류는 데이터가 어떤 클래스에 속하는가 구분하는 과정
- (ex. 사진 속 인물의 성별 가리기)
- 회귀는 입력 데이터에서 연속적인 수치를 예측
- (ex. 사진 속 인물의 몸무게 예측)

05 출력층 설계

출력은 우리가 원하는 값을 얻을 수 있어야 한다!



항등 함수

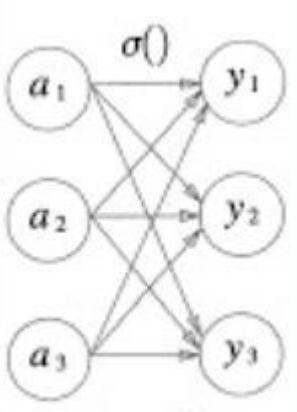
- 항등 함수는 입력 그대로를 출력한다.
- 즉 입력 신호 그대로 출력 신호가 된다는 뜻

- 이를 이용해서 주로 회귀 작업을 한다.

05 출력층 설계

출력은 우리가 원하는 값을 얻을 수 있어야 한다!

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



소프트맥스 함수

- N은 출력층의 뉴런 수
- Yk는 그 중 k번째 출력임을 나타냄

- 주로 분류 작업에서 사용한다.

05 출력층 설계

출력은 우리가 원하는 값을 얻을 수 있어야 한다!

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

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

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

◆ 소프트맥스 함수 구현

- 시그모이드 함수와 많은 차이는 없을 수 있으나, 만약 그 함수처럼 짠다면, 오버플로우 문제가 발생할 수 있다.
- 그렇기 때문에 임의의 정수 C를 정한 뒤 이를 분자 분모 양쪽에 곱한 뒤, 함수 exp 안에 옮겨 logC를 만든다.
- 마지막으로 logC를 C'라는 새로운 기호로 바꾼다.

- 이렇게 해도 결과는 바뀌지 않으면서, 오버플로우 방지가 가능하다!

05 출력층 설계

출력은 우리가 원하는 값을 얻을 수 있어야 한다!

```python
# 소프트맥스 함수 - n 개의 출력층의 뉴런 수 중 k번째 출력
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
```

05 출력층 설계

출력은 우리가 원하는 값을 얻을 수 있어야 한다!

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

print(y)

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

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

◆ 소프트맥스 함수의 특징

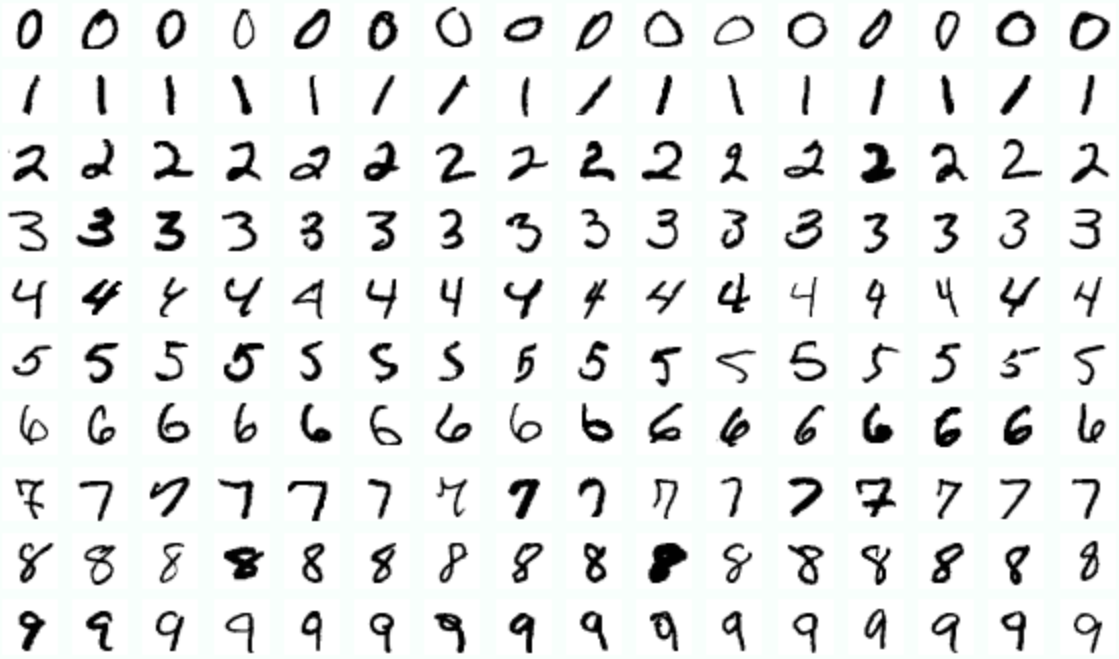
- 0~1.0 사이의 실수이다.
- 모든 출력의 총합은 1이다.
- 이를 통해서 소프트 맥스 함수의 출력은 확률임을 알 수 있다.

- 하지만 소프트맥스 함수를 적용해도 원소들의 대소 관계가 바뀌지 않고, 출력이 가장 큰 뉴런의 위치는 달라지지 않는다.
- 그렇기에 현업에서는 추론 단계에서는 생략하는 경우가 꽤 있다!

6 손글씨 숫자 인식

06 손글씨 숫자 인식

MNIST를 이용해서 실제 딥러닝을 느껴보자.



MNIST

- 손글씨 사진 데이터로써, 기계학습 분야의 Hello world라고 볼 수 있다.
- 실험용 데이터로 굉장히 자주 등장한다.
- 28x28 사이즈의 데이터이다.

06 손글씨 숫자 인식

MNIST를 이용해서 실제 딥러닝을 느껴보자.

```python
# 훈련 데이터 불러오기
def get_data() :
    (x_train, t_train), (x_test, t_test) = ₩
        load_mnist(flatten=True, normalize=True, one_hot_label=False)
    return x_test, t_test

# 시그모이드(S자 모양) 함수 - 연속적인 실수가 흐른다.
def sigmoid(x) :
    return 1 / (1 + np.exp(-x))

#ReLU 함수 - 0 이하면 0을 출력하고 그 위면 그 입력을 그대로 출력한다.
def ReLU(x) :
    return np.maximum(0, x)

# 항등 함수 - 입력을 그대로 출력
def identity_function(x) :
    return x

# 신경망 초기화
def init_network() :
    with open("sample_weight.pkl", 'rb') as f :
        network = pickle.load(f)

    return network
```

06 손글씨 숫자 인식

MNIST를 이용해서 실제 딥러닝을 느껴보자.

```python
# 추론 (전에 봤던 3층 신경망과 비슷하다.)
def predict_with_sigmoid(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

def predict_with_relu(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 = ReLU(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = ReLU(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)

    return y
```

06 손글씨 숫자 인식

MNIST를 이용해서 실제 딥러닝을 느껴보자.

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

MNIST

- (훈련 이미지, 훈련 테이블), (시험 이미지, 시험 테이블)
- 형식으로 반환한다.

- flatten: 입력 이미지를 평탄하게(1차원으로) 저장한다. 예로 들어 1x28x28 형식의 3차원 이미지가 있다면 784개의 원소를 가지는 1차원 배열로 변환한 뒤 저장.
- normalize: 입력 이미지의 픽셀을 0~1 사이의 값으로 정규화 할 것인가
- one_hot_label: 정답의 원소만 1이고 나머지는 모두 0인 방식으로 저장

06 손글씨 숫자 인식

MNIST를 이용해서 실제 딥러닝을 느껴보자.

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

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

print("accuracy : " + str(float(accurary_cnt) / len(x)))
```

```
accuracy : 0.9352
```

◆ **MNIST(시그모이드 함수를 이용한 결과)**

06 손글씨 숫자 인식

MNIST를 이용해서 실제 딥러닝을 느껴보자.

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

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

print("accuracy : " + str(float(accurary_cnt) / len(x)))
```

```
accuracy : 0.8415
```

◆ MNIST(ReLU 함수를 이용한 결과)

Thank you

End Of Document

