CUAI DLS스터디 2팀

2022.03.08

발표자 : 배준학, 임유민

스터디원 소개 및 만남 인증

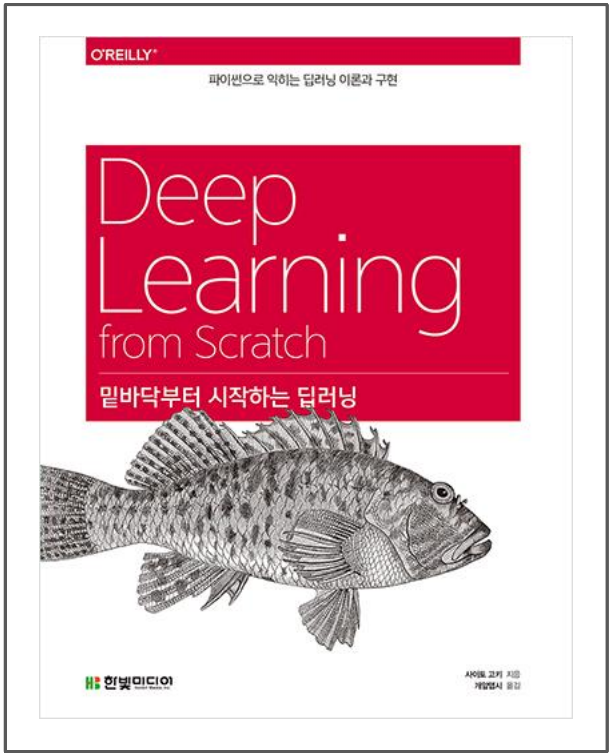
스터디원 1 : 박수빈

스터디원 2 : 배준학

스터디원 3 : 임유민

스터디원 4 : 최형용

스터디 계획



- ✓ **일시** : 매주 일요일 18시
- ✓ **방법** : 비대면 미팅(discord), 각자 공부한 내용 토대로 질문 형성 및 토의 진행
- ✓ **진도** : 한 주에 한 장 공부하기
- ✓ **목표** : 이번 학기까지 딥러닝 1권 끝내기

CHAPTER 3 신경망

-퍼셉트론과의 차이를 중심으로-

POINT

신경망의 중요한 성질

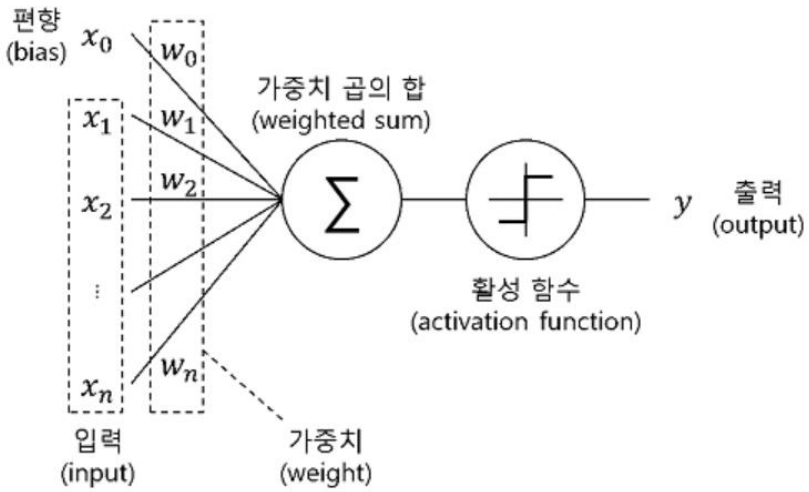
: 가중치 매개변수의 적절한 값을 데이터로부터 자동으로 학습하는 능력

(<->퍼셉트론: 사람이 수동으로 설정 했어야 했음)

1. 퍼셉트론에서 신경망으로

> 퍼셉트론 복습

[활성화 함수 처리과정 그림]



퍼셉트론

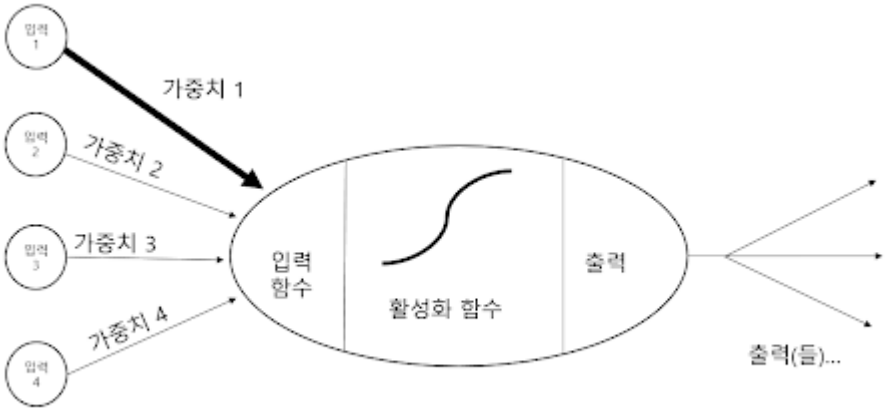
- 신호를 입력 받아 출력되는 알고리즘
- 편향 x0(b): 뉴런이 얼마나 쉽게 활성화 되는가
- w: 각 신호의 가중치

1. 퍼셉트론에서 신경망으로

> 활성화 함수의 등장

활성화 함수: 입력 신호의 총합을 출력 신호로 변환하는 함수

[활성화 함수 처리과정 그림]

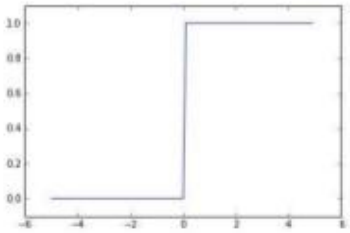


2. 활성화 함수

> 계단 함수

- 임계값을 기준으로 출력이 바뀌는 함수
- 퍼셉트론에선 계단함수를 이용
- (0 or 1)

[계단함수 그래프, 식]



계단 함수

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

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

2. 활성화 함수

> 시그모어드 함수

- 복잡해 보이지만 결국 입력에 따라 특정값을 출력하는 “함수”
- 신경망에선 활성화 함수로 시그모이드 함수를 이용하여

  신호를 변환하고, 그 변환된 신호를 다음 뉴런에 전달한다.

[시그모이드함수 그래프, 식]

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^{x}}{e^{x} + 1}.$$

- 사실 퍼셉트론과 신경망의 주된 차이는 이 활성화 함수 뿐

(뉴런이 여러층으로 이어지는 구조와 신호를 전달하는 방법은 기본적으로 같음)

2. 활성화 함수

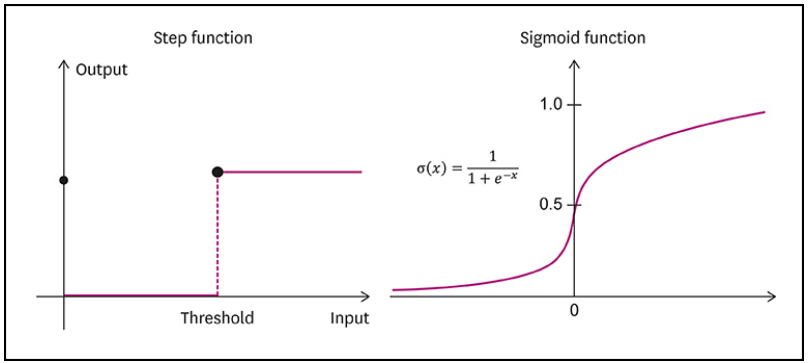
> 시그모어드 함수와 계단함수 비교

- 차이점: 보이는 것처럼 시그모이드 함수의 “연속적 변화”

  “연속적 변화”가 신경망 학습에 중요한 역할을 수행

- 공통점: 입력이 작을 때의 출력은 0, 커지면 1에 가까워지는 구조 즉, 입력이 중요하면 큰 값, 아니면 작은 값을 출력

[비교 그래프]



2. 활성화 함수

> 비선형 함수

- 그 밖에 중요한 공통점: 둘 모두 비선형 함수
- 신경망에선 활성화 함수로 비선형 함수를 사용해야 됨

➔ why?

: 선형 함수를 이용하면 신경망의 층을 깊게 하는 의미가 없어지기 때문

Eg) h(x)=cx를 활성화 함수로 사용한 3층 네트워크

Y(x) = h(h(h(x))) = c*c*c*x

= ax (a=c^3 일 때)

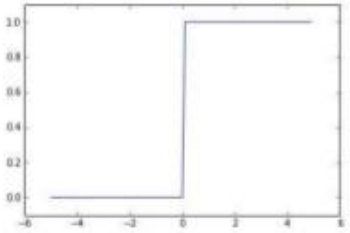
-> 결국 은닉층이 없는 네트워크와 표현이 가능해짐. (층을 쌓은 이유가 없어짐)

2. 활성화 함수

> ReLU 함수

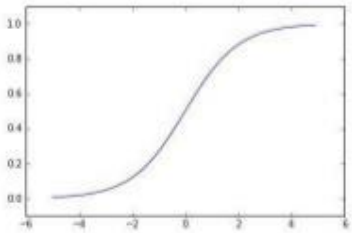
- 신경망 분야에서 최근 주로 사용
- 0을 넘으면 입력 그대로 출력, 0이하면 0을 출력

[계단함수, 시그모이드 함수, ReLU 함수 비교]



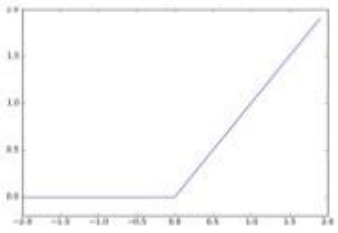
계단 함수

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



시그모이드 함수

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



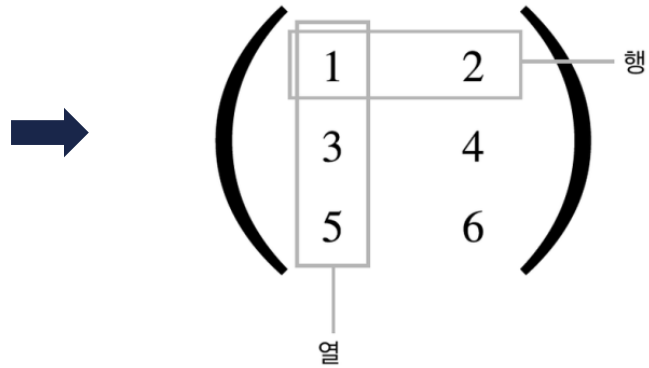
ReLU 함수

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

3. 다차원 배열의 계산

> 다차원 배열

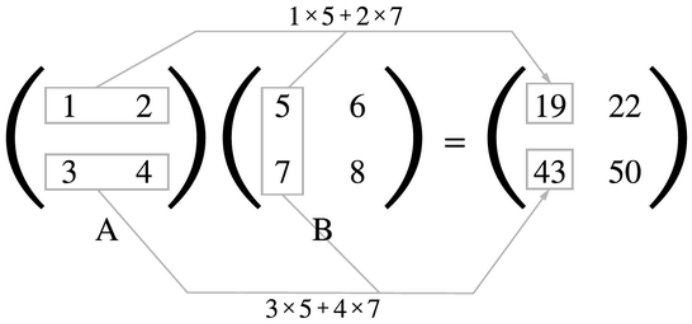
```
>>> B = np.array([[1, 2], [3, 4], [5, 6]])

>>> print(B)
[[1 2]
 [2 3]
 [4 5]]

>>> np.ndim(B)
2

>>> B.shape
(3, 2)
```



- 2차원 배열은 특히 행렬(matrix)이라고 부름
- 가로 방향은 행(row), 세로 방향은 열(column)

3. 다차원 배열의 계산

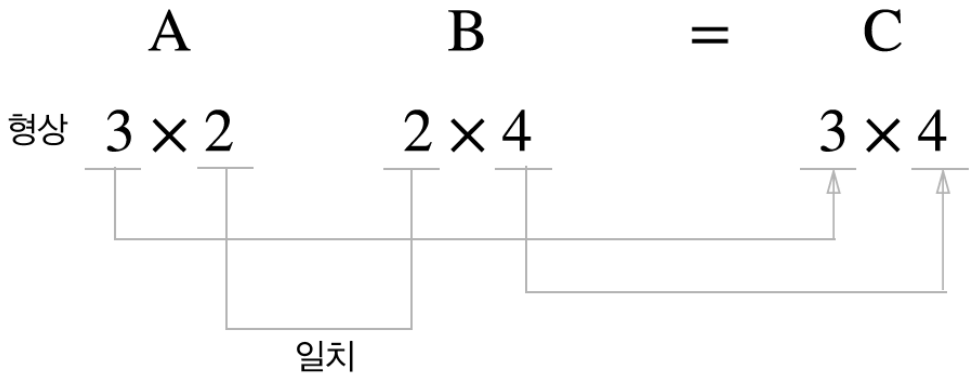
> 행렬의 곱




```
>>> A = np.array([[1, 2], [3, 4]])

>>> A.shape
(2, 2)

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

>>> B.shape
(2, 2)

>>> np.dot(A, B)
array([[19, 22],
       [43, 50])
```

3. 다차원 배열의 계산

> 행렬의 곱



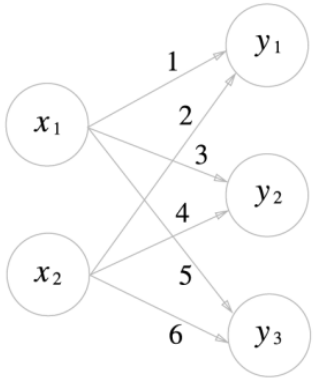
" 다차원 배열을 곱하려면 두 행렬의 대응하는 차원의 원소 수를 일치시켜야 한다. "

1) 첫 번째 행렬의 열 수와 두 번째 행렬의 행 수가 같아야 곱셈 가능
2) 결과로 도출되는 행렬은 (첫 번째 행렬의 행 수) x (두 번째 행렬의 열 수)

Ex)
A : 3 x 2
B : 2 x 4
C(결과) : 3 x 4

3. 다차원 배열의 계산

> 신경망에서 행렬의 곱



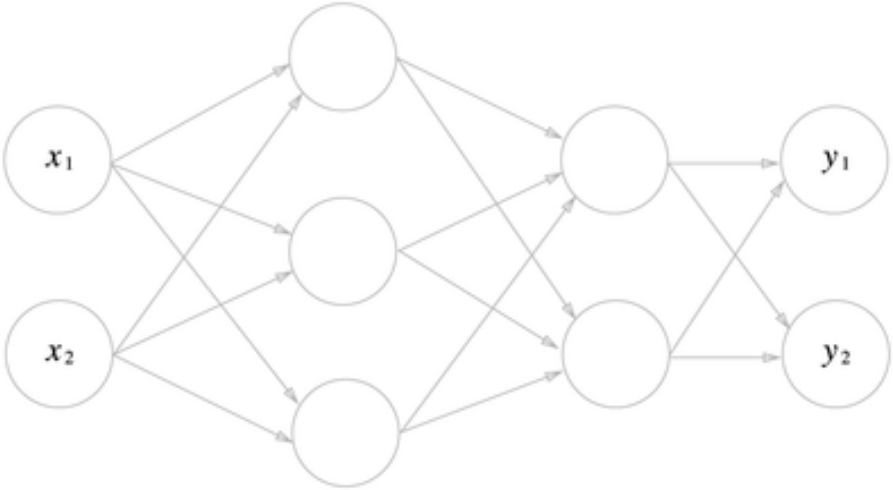
$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

$$\underset{2}{X} \quad \underset{2\times 3}{W} \quad = \quad \underset{3}{Y}$$

일치

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

>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17 ]
```

4. 3층 신경망 구현하기하기



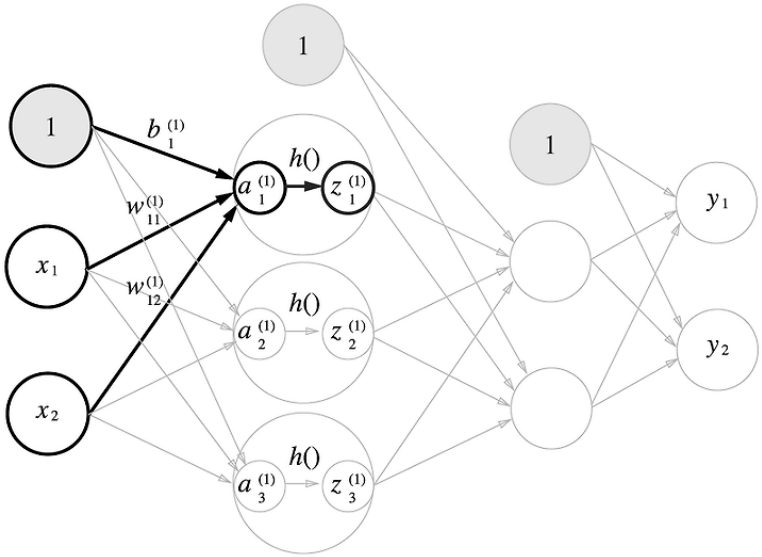
<<주어진 3층 신경망의 뉴런 개수>>

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

4. 3층 신경망 구현하기

> 각 층의 신호 전달 구현하기

(1) 입력층(0층) -> 1층



" X, W1, B1 → A1 → Z1 "

< 가중치 계산 >

```
# 입력값
X = np.array([1.0, 0.5])
# 가중치(weight)
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
# 편향(bias)
B1 = np.array([0.1, 0.2, 0.3])

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

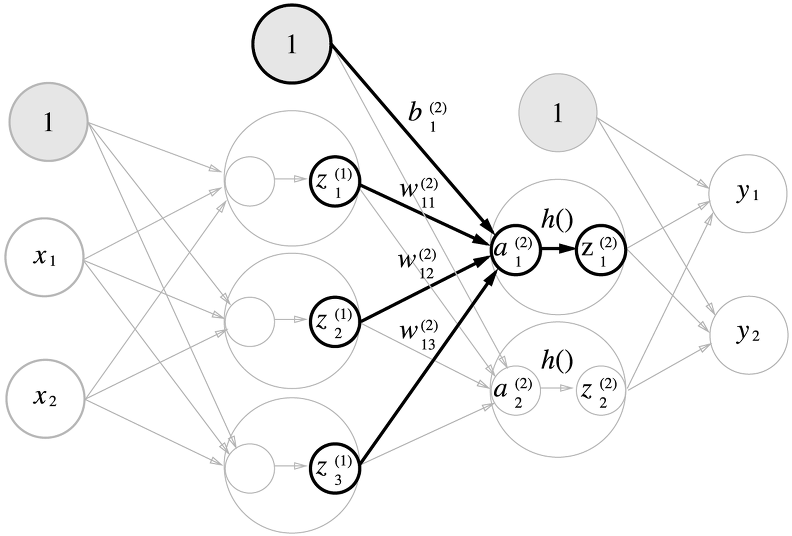
< 활성화 함수>

```
def sigmoid(x) : # 시그모이드 함수 구현
return 1/(1+np.exp(-x))

# 가중치 계산값을 활성화
Z1 = sigmoid(A1)
```

4. 3층 신경망 구현하기

> 각 층의 신호 전달 구현하기

(2) 1층 -> 2층

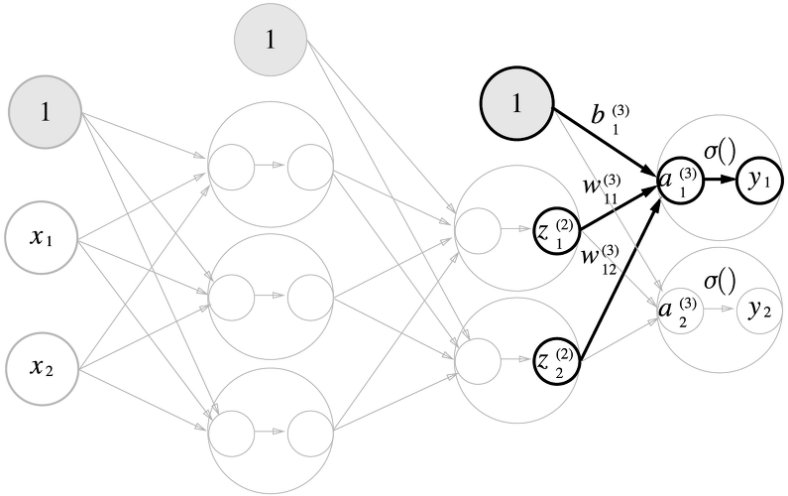



```
# 1->2 가중치(weight)
W2 = np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
# 1->2 편향(bias)
B2 = np.array([0.1,0.2])

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

# 2층의 가중치 계산값을 활성화
Z2 = sigmoid(A2)
```

4. 3층 신경망 구현하기

> 각 층의 신호 전달 구현하기

(3) 2층 -> 3층(출력층)




```
# 활성화 함수 : 항등함수 구현
def identity_function(x):
return x

# 2->3 가중치(weight)
W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
# 2->3 편향(bias)
B3 = np.array([0.1, 0.2])

A3 = np.dot(Z2, W3) + B3

# 출력층의 가중치 계산값을 활성화
Y = identity_function(A3)
```

4. 3층 신경망 구현하기 > 구현 정리

```
//////////////////구현부//////////////////////
def init_network() :
network = {} # dictionary 데이터 타입 선언
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

//////////////////실행부//////////////////////
network = init_network()
x = np.array([1.0, 0.5])

y = forward(network, x)
print(y)
```

✔ init_network()

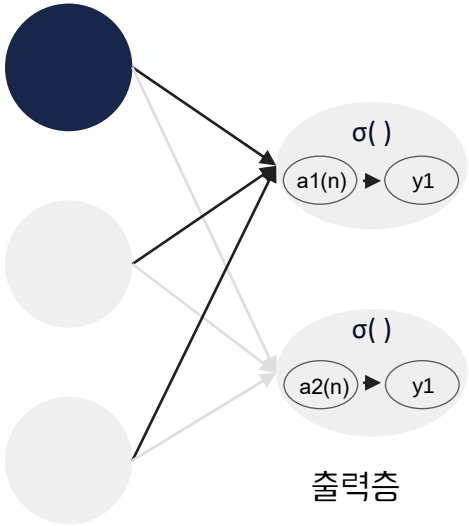
가중치와 편향을 초기화하여 딕셔너리 변수인 network에 저장

✔ forward()

입력 신호를 출력으로 변환(가중치/편향 연산 -> 활성화 함수)

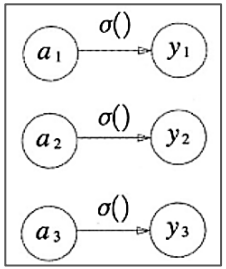
5. 출력층 설계하기

> 항등 함수와 소프트맥스 함수 구현하기



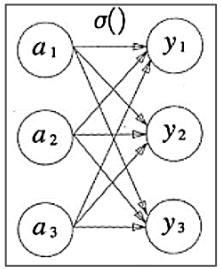
출력층

- 회귀 문제에서의 활성화 함수



항등함수

- 분류 문제에서의 활성화 함수



소프트맥스 함수

5. 출력층 설계하기

> 소프트맥스 함수 구현

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

- n: 출력층의 뉴런 수
- y_k: 출력층의 뉴런 중 k 번째 출력
- exp(): 지수함수

- 분자: 입력신호 a_k의 지수함수
- 분모: 모든 입력 신호의 지수함수의 합

5. 출력층 설계하기

> 소프트맥스 함수 구현

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

- 분모: sum(exp_a) -> array의 지수함수 값들의 합
- 분자: exp_a -> array의 지수함수 값

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

exp_a = np.exp(a) # 지수 함수
sum_exp_a = np.sum(exp_a) # 지수 함수의 합

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

```
[ 0.01821127 0.24519181 0.73659691 ]
```

→

```
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 구현 시 주의점

- 컴퓨터로 소프트맥스를 계산할 때 오버플로 문제 발생가능
- **오버플로(Overflow)** : 표현할 수 있는 수의 범위가 한정되어 너무 큰 값은 표현할 수 없음

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

→ exp(1010) → 2.718^{1010} = Infinity

array([nan, nan, nan])

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

- ✓ 첫 번째 변형에서 C라는 임의의 정수를 분자와 분보 양쪽에 곱함
- ✓ C를 지수 함수 exp() 안으로 옮겨 logC로 만듬
- ✓ logC를 C'라는 새로운 기호로 바꿈
- ✓ **소프트맥스의 지수 함수를 계산할 어떤 정수를 더해도 (혹은 빼도) 결과는 바뀌지 않음**
- ✓ 오버플로를 막을 목적으로는 입력 신호 중 최대값을 이용하는 것이 일반적

5. 출력층 설계하기

> 소프트맥스 구현 시 주의점

```
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
```

- C = 입력 신호 중 최대 값
- exp(a - c) = 각 입력 신호에서 최댓값을 뺀 후 지수 함수를 적용한 값
- sum(exp_a) = 최댓값을 뺀 입력 신호의 지수함수 값을 전부 합친 값

array([nan, nan, nan]) → array([9.99954600e-01, 4.53978686e-05, 2.06106005e-09])

5. 출력층 설계하기

> 소프트맥스 함수의 특징

- 문제를 통계적으로 대응할 수 있음

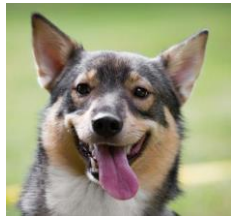
$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

결국 식의 전체적인 형태는 해당 값 / 전체 값 => 확률로 해석이 가능하다. 때문에 0과 1 사이의 값을 가진다.

- 상대적인 비교 가능

예를 들어 신경망을 통해 개와 고양이, 말을 분류한다고 하자. 이들의 결과 값을 통계적으로 대응할 수 있으며, 상대적인 비교가 가능하기에, 분류 모델에 적합하다.

dog



dog: 92%

cat: 6%

cow: 2%

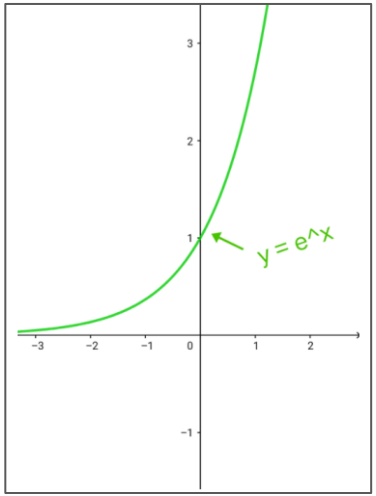
5. 출력층 설계하기

> 소프트맥스 함수의 특징

- **결과가 더욱 확실하게 나올 수 있다**

  자연상수의 지수를 사용하지 않고도 상대적인 비교를 통한 수치를 얻을 수는 있지만, 지수 함수를 적용함으로써, 분류 결과의 차이를 더욱 강조할 수 있다. 이는 특히 입력값들이 서로 비슷한 크기를 가질 때 유용하며, 소프트맥스 함수를 통해 계산된 확률 값은 분류에 대한 결정이 더욱 분명하게 이루어질 수 있도록 도와준다..

- **각 원소의 대소관계는 변하지 않는다**



- y = exp(x) 는 왼쪽처럼 단조증가 함수이다
- 소프트맥스 함수를 적용해도 출력이 가장 큰 뉴련의 위치는 달라지지 않는다 -> 신경망으로 분류시 적용 생략 가능
- 다중 클래스 분류 문제에서 확률적 해석이 중요할 때 필요함

5. 출력층 설계하기

> 소프트맥스 출력의 뉴런 수 정하기

>> 출력층의 뉴런 수 = 분류하고 싶은 클래스의 수

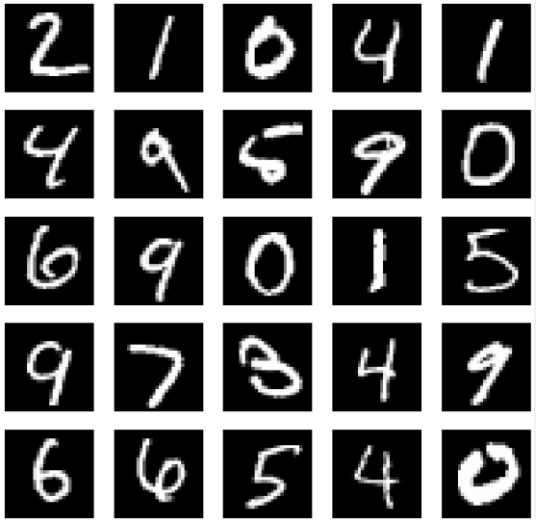
> 클래스 수: 10개 (0부터 9까지의 숫자)
> 출력층 뉴런 수: 10개
> 각 뉴런은 특정 숫자가 이미지에 표시된 확률을 나타내며, 가장 높은 확률을 출력하는 뉴런의 인덱스가 모델의 예측으로 사용된다.

> 클래스 수: 3개 (예: 고양이, 개, 소)
> 출력층 뉴런 수: 3개
> 각 뉴런은 이미지가 특정 클래스(고양이, 개, 소)에 속할 확률을 나타내며, 가장 높은 확률을 출력하는 뉴런이 모델의 예측을 결정한다.

> 클래스 수: 5개 (예: 매우 부정적, 부정적, 중립적, 긍정적, 매우 긍정적)
> 출력층 뉴런 수: 5개
> 출력층에는 5개의 뉴런을 사용하여 각 감정 상태에 대한 확률을 출력하고, 가장 높은 확률을 가진 뉴런이 해당 텍스트의 감정 상태를 나타낸다.

6. 손글씨 숫자 인식

> MNIST 데이터셋 ⟶ 손글씨 숫자 이미지 집합



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

# (훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True,
normalize=False)

# 각 데이터의 형상 출력
print(x_train.shape)
print(t_train.shape)
print(x_test.shape)
print(t_test.shape)
```

```
(60000, 784)
(60000,)
(10000, 784)
(10000,)
```

6. 손글씨 숫자 인식

> 신경망의 추론 처리

◎ 입력층

▣ 뉴런 784개

(28 x 28 사이즈 이미지 평활화 ⇒ 784 x 1 배열)

◎ 은닉층 2개 - (저자가 임의로 정함)

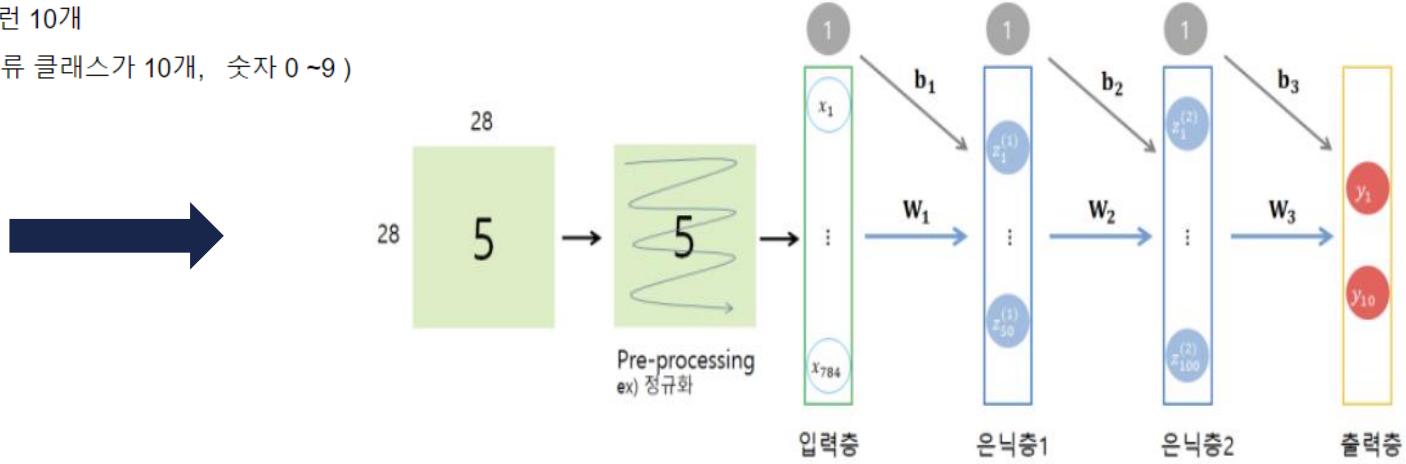
▣ layer1 : 뉴런 50개

▣ layer2 : 뉴런 100개

◎ 출력층

▣ 뉴런 10개

(분류 클래스가 10개, 숫자 0 ~9)



6. 손글씨 숫자 인식

> 신경망의 추론 처리

(1) MNIST 데이터를 np.ndarray 객체로 불러오기

```
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      # 시험 데이터셋
                                # 이번 단원의 학습 목표는 추론 단계를 시현하는 것
```

(2) 사전 학습된 가중치 파일 불로오기

```
def init_network():
    with open('sample_weight.pkl', mode='rb') as f:
        network    =  pickle.load(f)                    # 직렬화된 파일 복원

```

(3) 추론

```
def predict(network, x):
    # print(network)
    # print(type(network))
    W1, W2, W3    =  network['W1'], network['W2'], network['W3'] # weight
    b1, b2, b3    =  network['b1'], network['b2'], network['b3'] # bias

    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
```

6. 손글씨 숫자 인식

> 신경망의 추론 처리

```
x,  label   =  get_data()
network = init_network()    # 사전 학습된 가중치 불러오기
```

```
accuracy_cnt = 0            # 정확하게 예측한 것 카운트 (TP)

for i in range(len(x)):
    y = predict(network, x[i])  # x[i]: 평활화된 i번째 이미지
    print("확률 = ",y)   # 출력된 클래스별 확률
    print("₩n")
    max_idx = np.argmax(y)      # 값이 가장 큰 원소의 인덱스 반환

    if max_idx == label[i]:
        accuracy_cnt += 1
```

```
확률 =  [8.4412488e-05 2.6350631e-06 7.1549421e-04 1.2586262e-03 1.1727954e-06
 4.4990808e-05 1.6269318e-08 9.9706501e-01 9.3744793e-06 8.1831159e-04]
```

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

```
Accuracy: 0.9352
```

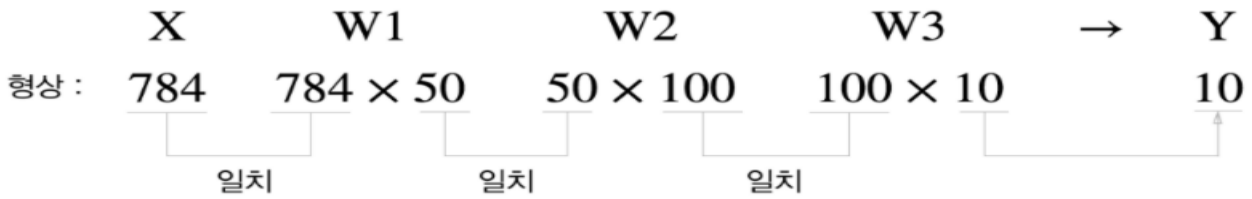
6. 손글씨 숫자 인식

> 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)
```

배치란, 곧 묶음이라는 의미.
이 결과에서 다차원 배열의 대응하는 차원의 원소 수가 일치함을 확인.

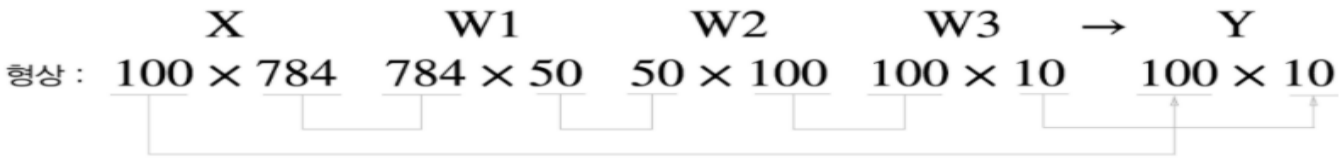
이는 이미지 **데이터를 1장**만 입력했을 때의 처리 흐름입니다. 그렇다면 다음 장에서 이미지 **여러 장을 한꺼번**에 입력하는 경우를 생각해봅시다.



신경망 각 층의 배열 형상의 추이

6. 손글씨 숫자 인식

> 배치 처리



배치 처리를 위한 배열들의 형상 추이

배치 처리의 이점

1. 수치 계산 라이브러리 대부분이 큰 배열을 효율적으로 처리할 수 있도록 고도로 최적화되어 있기 때문
입니다.

2. 커다란 신경망에서는 데이터 전송이 병목으로 작용하는 경우가 자주 있는데, 배치 처리를 함으로써
버스에 주는 부하를 줄입니다. (느린 I/O를 통해 데이터를 읽는 횟수가 줄어, 빠른 CPU나 GPU로 순수
계산을 수행하는 비율이 높아집니다.)

즉, 배치 처리를 수행함으로써 큰 배열로 이뤄진 계산을 하게 되는데, 컴퓨터에서는 큰 배열을 한꺼번에 계산하는 것이 분할된 작은 배열을 여러번 계산하는 것보다 빠릅니다.

감 사 합 니 다