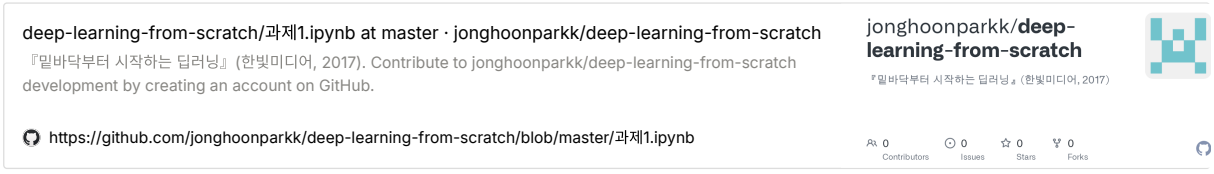
# 과제1

작성자 박종훈

날짜 240605

1차과제






# Deep Learning Assignment #1

High Performance Integrated Circuit Design Lab.

실행해보는 문제는 직접 실행하고, 그 캡쳐본 또는 코드 및 결과를 답변으로 제출하시면 됩니다.

**Deep Learning Assignment #1**

High Performance Integrated Circuit Design Lab.

- 실행해보는 문제는 직접 실행하고, 그 캡처본 또는 코드 및 결과를 답변으로 제출하시면 됩니다.

1. (Chapter 1)

1.1. (넘파이 배열)

1.1.1. 1 차원, 2 차원 배열의 산술연산에 대해서 설명하고, 그 예시를 실행해보시오. 또한, dimension, shape, dtype 에 대해서 설명하시오.

1.1.2. 브로드캐스트 기능에 대해서 설명하시오.

1.1.2.1. 2 차원 배열 * 스칼라 연산을 직접 실행해보고, 그 결과에 대해서 고찰하시오.

1.1.2.2. 2 차원 배열 * 1 차원 배열 연산을 직접 실행해보고, 그 결과에 대해서 고찰하시오.

1.1.3. flatten 기능을 이용하여 2 차원 배열을 1 차원 배열로 변환해보고, 1 차원 배열로 바꾸는 이유에 대해서 설명하시오.

2. (Chapter 2)

2.1. 퍼셉트론으로 XOR gate 를 구현하고, 단층 퍼셉트론으로는 XOR gate 를 구현할 수 없는 이유에 대해서 설명하시오.



# 1. (Chapter 1)

## 1.1. (넘파이 배열)

### 1.1.1. 1 차원, 2차원 배열의 산술연산에 대해서 설명하고, 그 예시를 실행해보시오. 또한, dimension, shape, dtype 에 대해서 설명하시오.

```
# 1.1. (넘파이 배열)
import numpy as np
x = np.array([1.0, 2.0, 3.0])
y = np.array([2.0, 4.0, 6.0])
```

```
x+y
>>> array([3., 6., 9.])
x-y
>>> array([-1., -2., -3.])
x/y
>>> array([0.5, 0.5, 0.5])
x*y
>>> array([ 2.,  8., 18.])
```

넘파이의 N차원 배열

넘파이는 1차원 배열(1줄로 늘어선 배열)뿐 아니라 다차원 배열도 작성할 수 있다

예를 들어 2차원 배열(행렬)은 다음처럼 작성한다

```
A = np.array([[1, 2], [3, 4]])
print(A)
print(A.shape) # 형상 shape
print(A.dtype) # 자료형 dtype
>>>
[[1 2]
 [3 4]]
(2, 2)
int64
```

방금 2×2의 A라는 행렬을 작성했다

행렬의 형상은 shape으로, 행렬에 담긴 원소의 자료형은 dtype으로 알 수 있다

### 1.1.2. 브로드캐스트 기능에 대해서 설명하시오.

넘파이 배열은 원소별 계산뿐 아니라 넘파이 배열과 수치 하나(스칼라값)의 조합으로 된 산술 연산도 수행할 수 있다

이 경우 스칼라값과의 계산이 넘파이 배열의 원소별로 한 번씩 수행된다

이 기능을 브로드캐스트라고 한다

```
x = np.array([1.0, 2.0, 3.0])
x / 2.0
>>> array([0.5, 1. , 1.5])
```

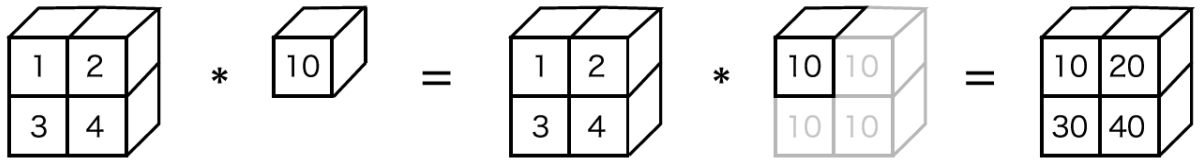
넘파이에서는 형상이 다른 배열끼리도 계산할 수 있다

앞의 예에서는 2×2 행렬 A에 스칼라값 10을 곱했다

이때 그림과 같이 10이라는 스칼라값이 2×2 행렬로 확대된 후 연산이 이뤄진다

이 똑똑한 기능을 브로드캐스트(broadcast)라고 한다

#### 1.1.2.1. 2 차원 배열 * 스칼라 연산을 직접 실행해보고, 그 결과에 대해서 고찰하시오.



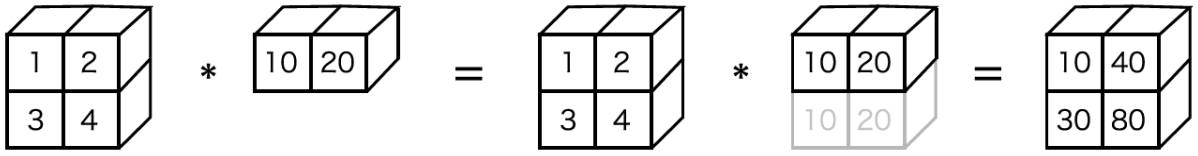
```
A = np.array([[1, 2], [3, 4]])
B = 10
A * B
>>>
array([[10, 20],
       [30, 40]])
```

형상이 같은 행렬끼리면 행렬의 산술 연산도 대응하는 원소별로 계산된다 배열과 마찬가지로

행렬과 스칼라값의 산술 연산도 가능하다

이때도 배열과 마찬가지로 브로드캐스트 기능이 작동한다

### 1.1.2.2. 2 차원 배열 * 1차원 배열 연산을 직접 실행해보고, 그 결과에 대해서 고찰하시오.




```
A = np.array([[1, 2], [3, 4]])
B = np.array([10, 20])
A * B
>>>
array([[10, 40],
       [30, 80]])
```

여기에서는 그림처럼 1차원 배열인 B가 '똑똑하게도' 2차원 배열 A와 똑같은 형상으로 변형된 후 원소별 연산이 이뤄진다

### 1.1.3. flatten 기능을 이용하여 2차원 배열을 1차원 배열로 변환해보고, 1차원 배열로 바꾸는 이유에 대해서 설명하시오.

```
import numpy as np

# 2차원 배열 생성
array_2d = np.array([[1, 2, 3], [4, 5, 6]])

# 1차원 배열로 변환
array_1d = array_2d.flatten()

print("2차원 배열:")
print(array_2d)
print("\n1차원 배열:")
print(array_1d)
```

```
2차원 배열:
[[1 2 3]
 [4 5 6]]

1차원 배열:
[1 2 3 4 5 6]
```

**1차원 배열로 변환하는 이유**

1. 데이터 처리 용이성: 데이터 분석이나 처리 과정에서 일부 알고리즘은 1차원 배열 형태의 입력을 요구한다. 예를 들어, 머신 러닝 모델이나 신경망에서 입력 데이터를 한 줄로 정렬된 벡터로 다루는 경우가 많다.
2. 호환성: 다양한 라이브러리나 함수들이 1차원 배열을 인자로 받는 경우가 있다. 예를 들어, 일부 시각화 도구나 통계 함수는 1차원 배열을 기대한다.

3. 메모리 접근 효율성: 연속적인 메모리 블록으로 데이터를 다루면 캐시 히트율이 높아져서 성능이 개선될 수 있다. 1차원 배열은 메모리에서 연속적으로 저장되기 때문에, 반복적인 접근 시 빠른 속도를 제공할 수 있다.
4. 단순화된 데이터 구조: 데이터를 직관적이고 간단하게 다룰 수 있다. 특히, 복잡한 데이터 구조를 단순화하고, 각 요소에 대한 접근이 쉬워진다.

# 2. (Chapter 2)

## 2.1. 퍼셉트론으로 XOR gate 를 구현하고, 단층 퍼셉트론으로는 XOR gate 를 구현할 수 없는 이유에 대해서 설명하시오.

→ AND, NAND, OR는 모두 같은 구조의 퍼셉트론이고, 차이는 가중치 매개변수의 값뿐이다

### and_gate.py

w = np.array([0.5, 0.5])
b = -0.7

```
# and_gate.py

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

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

$-\theta$가 편향 b로 치환된다

$w_1$와 $w_2$의 가중치는 입력 신호가 결과에 주는 영향력(중요도)를 조절하는 매개변수

편향b는 뉴런이 얼마나 쉽게 활성화(결과로 1을출력)하느냐를 조정하는 매개변수

문맥에 따라 w와 b모두를 가중치라고 할때도 있다

### nand_gate.py

w = np.array([-0.5, -0.5])
b = 0.7

```
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

if __name__ == '__main__':
    for xs in [(0, 0), (1, 0), (0, 1), (1, 1)]:
        y = NAND(xs[0], xs[1])
        print(str(xs) + " -> " + str(y))
```

### or_gate.py

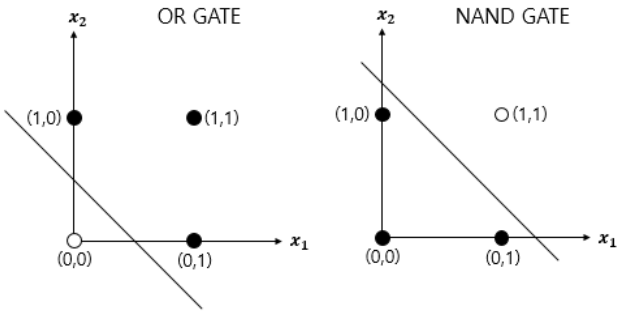
w = np.array([0.5, 0.5])
b = -0.2

```
import numpy as np


def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.2
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1

if __name__ == '__main__':
    for xs in [(0, 0), (1, 0), (0, 1), (1, 1)]:
        y = OR(xs[0], xs[1])
        print(str(xs) + " -> " + str(y))
```

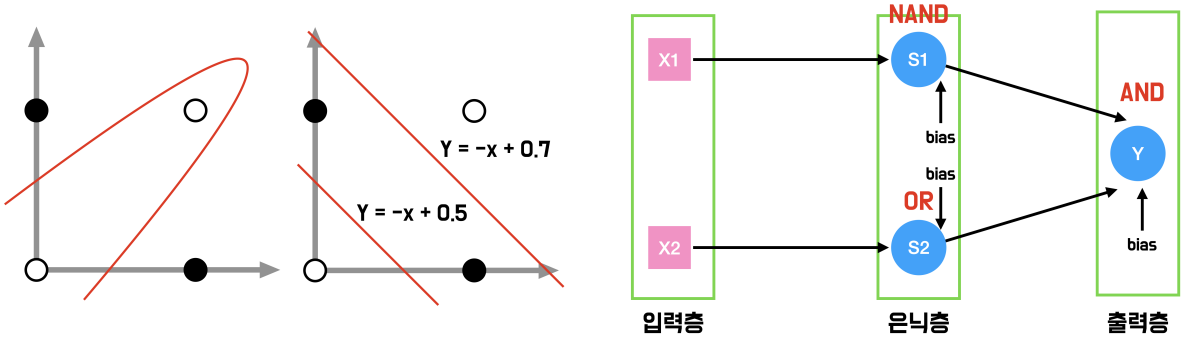
### XOR 게이트 구현




https://wikidocs.net/24958

퍼셉트론을 시각화 하면 다음과같다

빈 동그라미가 0이고 검은 동그라미가 1을 출력한다




https://velog.io/@skyepodium/퍼셉트론이-왜-XOR-문제를-못푸는지-알아보기

### 단층 퍼셉트론으로는 XOR gate 를 구현할 수 없는 이유

XOR 문제를 해결하려면 곡선을 그리거나, 2개의 직선을 그려야 한다

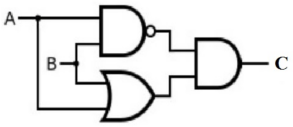
하지만, 기존의 or, and, nand는 1개의 직선을 그리는 방식이고 이를 해결하지 못했다

입력층과 출력층으로 구성된 단층 퍼셉트론에 은닉층이 추가되면 다층 퍼셉트론이라고 부른다

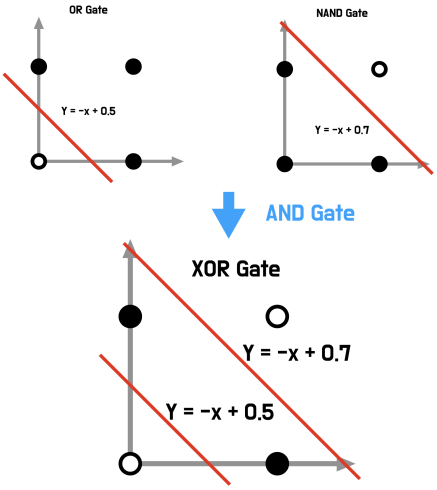
은닉층은 기존의 출력층과 같이 가중치합을 계산한다

위 그림처럼 3개의 논리 게이트 NAND, OR, AND 게이트를 조합하면 XOR 문제를 해결할 수 있다

회로 그림으로 보면 다음과 같다




https://asthtls.tistory.com/966




왼쪽 선과 오른쪽 선을 AND 게이트로 합쳐준다

최종 코드는 아래와 같다

### xor_gate.py

```
from and_gate import AND
from or_gate import OR
from nand_gate import NAND


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

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