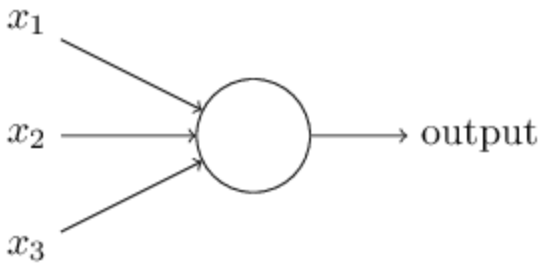
Artificial Intelligence

Perceptron

❖ Perceptron

- 로젠 블라트의 퍼셉트론.
  - 다수의 신호를 입력으로 받아 하나의 신호를 출력함.
  - 2진 값을 갖는 다중의 입력을 하나의 2진 값으로 출력.



❖ Perceptron

- 로젠 블라트의 퍼셉트론.
  - 다수의 신호를 입력으로 받아 하나의 신호를 출력함.
  - 2진 값을 갖는 다중의 입력을 하나의 2진 값으로 출력.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

❖ Perceptron

- And 게이트
  - 입력이 둘, 출력이 하나.
  - 입력이 모두 1일 때 1, 그 외의 경우는 0을 출력.
- Or 게이트
  - 입력이 둘, 출력이 하나.
  - 입력이 모두 0일 때 0, 그 외의 경우는 1을 출력.

AND		
Input		Output
A	B	C
0	0	0
1	0	0
0	1	0
1	1	1

OR		
Input		Output
A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

❖ Perceptron

- **퍼셉트론 구현.**
  - AND Gate

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

```
AND(0, 0) # 0을 출력
AND(1, 0) # 0을 출력
AND(0, 1) # 0을 출력
AND(1, 1) # 1을 출력
```

❖ Perceptron

- **퍼셉트론 구현(가중치와 편향 도입).**

$$output = \begin{cases} 0 & if \sum_j w_j x_j + b \leq 0 \\ 1 & if \sum_j w_j x_j + b > 0 \end{cases}$$

```
import numpy as np
x = np.array([0, 1])
w = np.array([0.5, 0.5])
b = -0.7
np.sum(w*x) + b
```

❖ Perceptron

- **퍼셉트론 구현(가중치와 편향 도입).**

```
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
        elif tmp > 0:
                return 1
```

❖ Perceptron

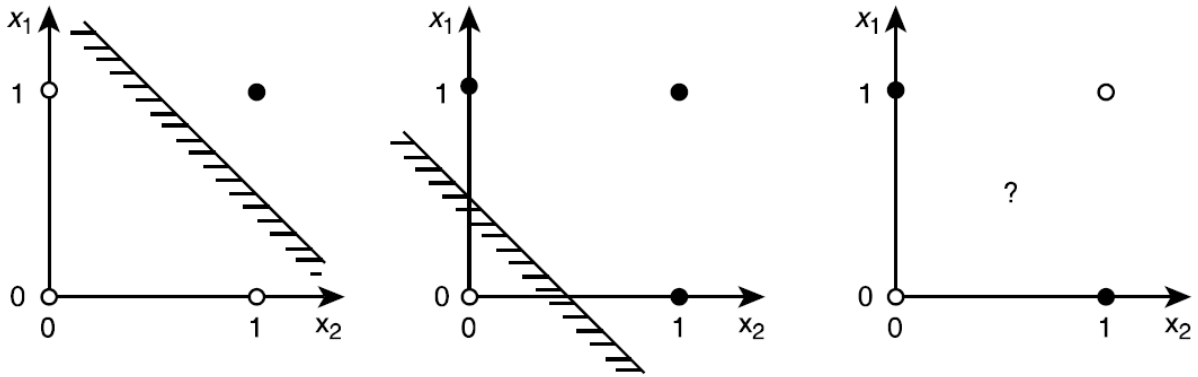
- **퍼셉트론 구현**(NAND, OR **게이트**).

```
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
        elif tmp > 0:
                return 1
```

```
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
        elif tmp > 0:
                return 1
```

❖ Perceptron

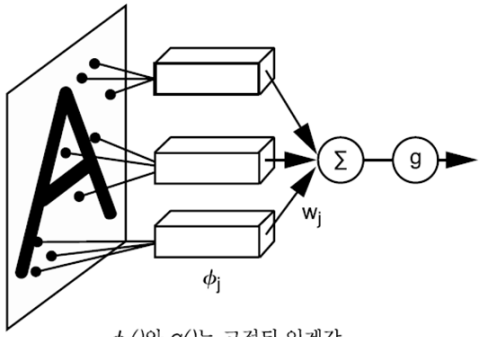
- 퍼셉트론의 한계.



(a) x_1 and x_2 (b) x_1 or x_2 (c) x_1 xor x_2

❖ Perceptron

- 퍼셉트론의 학습.
  - 델타 규칙이라는 학습 규칙을 사용.
  - 목적 패턴을 이용한 신경망의 학습을 인위로 제어.
  - 델타 규칙 : 만일 어떤 신경세포의 활성이 다른 신경세포가 잘못된 출력을 내는데 공헌을 했다면, 두 신경 세포간의 연결 가중치를 그것에 비례하게 조절.



$\phi_j()$와 $g()$는 고정된 임계값
w_j는 가변 가중치

[그림 13-10] 퍼셉트론

퍼셉트론 신경망에 적용된 학습규칙 → 델타 규칙(Delta Rule)

❖ Perceptron

▪ 델타 규칙

$$w_{ij}^{new} = w_{ij}^{old} + \alpha e_j a_i$$

$$e_j = t_j - b_j$$

※ w_{ij}^{new} : 신경세포 i, j 사이의 조절된 후 연결 가중치

w_{ij}^{old} : 신경세포 i, j 사이의 조절되기 전 연결 가중치

α : 학습률 ($0 < \alpha \leq 1$)

e_j : 신경세포 j의 오차

a_i : 입력층 신경세포 i의 활성값

t_j : 목적 패턴의 출력층 신경세포 j에 대응하는 성분값

b_j : 출력층 신경세포 j의 활성값

❖ Perceptron

▪ 델타 규칙

델타 규칙을 사용하여 신경망을 학습하는 과정을 요약하면 다음과 같다.

① 입력층에 입력 패턴을 제시
② 신경망을 동작
③ 델타 규칙에 의해 연결 가중치를 조절

$$w_{ij}^{new} = w_{ij}^{old} + \alpha e_j a_i$$

④ 신경망이 완전하게 학습될 때까지 과정 ①~③을 입력 패턴에 대해 반복

델타 규칙에 의한 학습 과정은 감독 학습 방법이 사용하는 일반적인 형태의 학습 과정과 동일하다. 단지 과정 ③에서 연결 가중치 조절식만 다를 뿐이다.

델타 규칙에서 학습 완료 정도를 나타내는 오차는 헤브의 규칙에서와 같다. 앞서 (13.4)에서와 같이 신경망의 실제 출력 패턴의 목적 패턴의 차이에 의해 계산된다. 이 부분에서 중요한 것은 "잘못된 출력을 낸 신경세포들의 연결 가중치를 그 정도에 비례하여 조절한다"이다.

❖ Perceptron

- **퍼셉트론의 학습 구현**

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

dataNum = 1000
trainData1 = np.random.randn(dataNum, 2)
trainData2 = np.random.randn(dataNum, 2) + 5
trainData = np.zeros((dataNum*2, 2))

trainData[0:dataNum, : ] = trainData1
trainData[dataNum:dataNum*2, : ] = trainData2

trainout = np.zeros((dataNum*2, 1))
trainout[dataNum: dataNum*2, :] = np.zeros((dataNum, 1)) + 1
```

❖ Perceptron

- **퍼셉트론의 학습 구현**

```
plt.plot(trainData1[:, 0], trainData1[:, 1], "*")
plt.plot(trainData2[:, 0], trainData2[:, 1], "*")

step = 100
input = 2; out = 1
w = np.random.rand(input, out); b = np.random.randn(1)
a = np.arange(-3, 6, 0.1)

plt.plot(a, (-w[0, 0]*a - b)/w[1, 0])

eta = 0.5
```

❖ Perceptron

- **퍼셉트론의 학습 구현**

```
for j in range(1, step, 1):
    for i in range(0, dataNum*2, 1):
        x = trainData[i]
        ry = trainout[i]
        if (np.dot(x, w) + b) > 0:
            y = 1
        else:
            y = 0
        e = ry - y

        dw = eta * e *np.transpose([np.array(x)])
        db = eta * e * 1

        w = w + dw
        b = b + db

plt.plot(a, (-w[0, 0]*a - b)/w[1, 0], 'r')

plt.show()
```