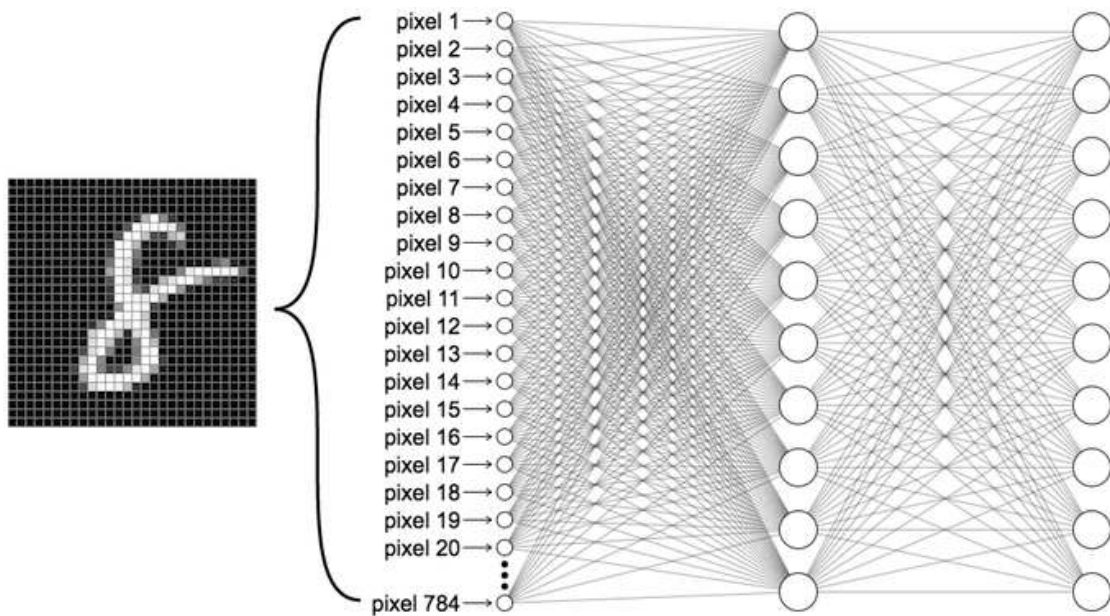


Chapter 04

NumPy DNN 구현과 활용

01 NumPy DNN 구현하기

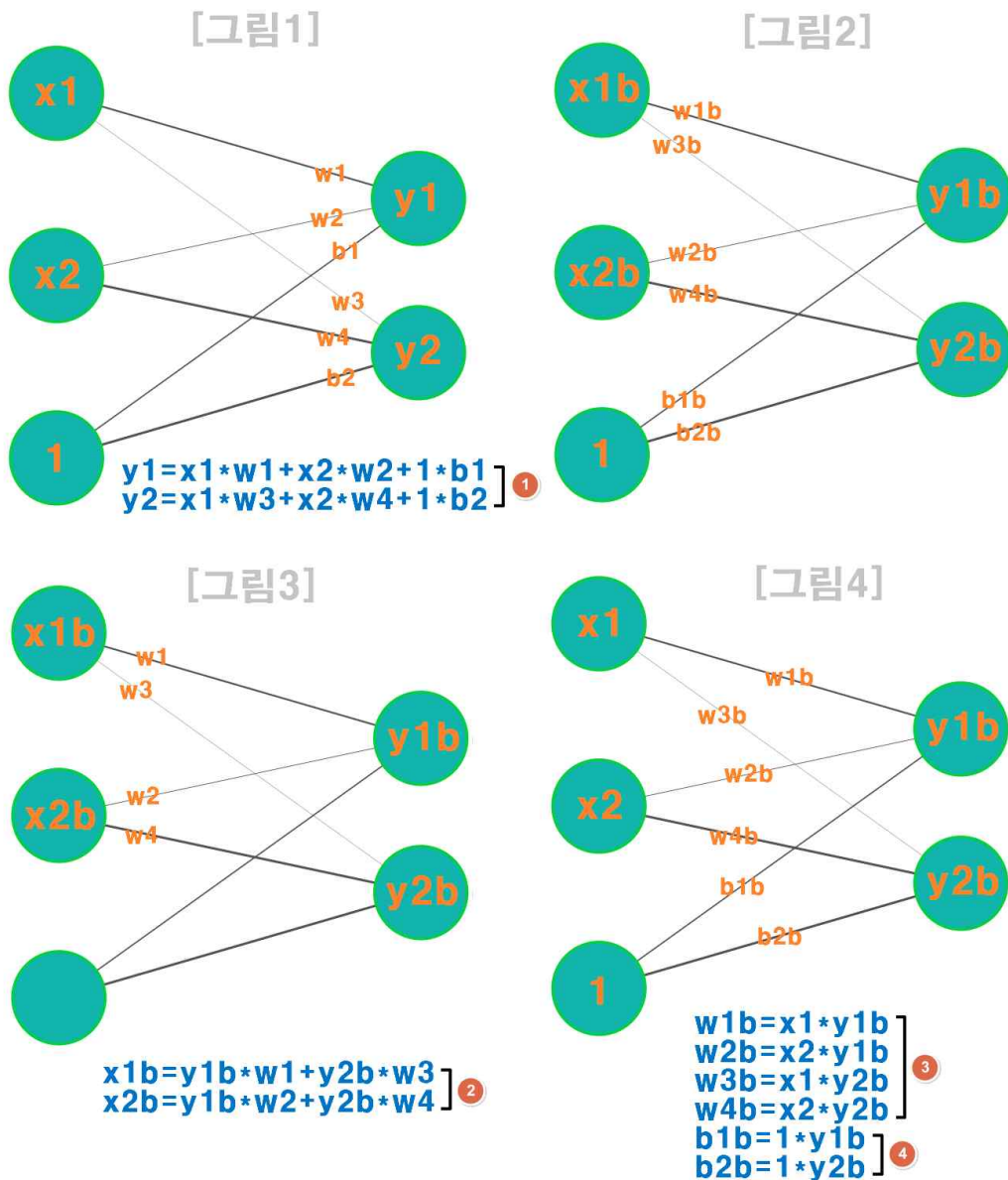
여기서는 인공 신경망을 확장할 수 있도록 NumPy 라이브러리를 활용하여 인공 신경망을 구현해 봅니다. NumPy 라이브러리를 이용하면, 커다란 인공 신경망을 자유롭게 구성하고 테스트해 볼 수 있습니다. 예를 들어, 1장에서 tensorflow 라이브러리를 이용하여 살펴보았던 다음과 같은 형태의 인공 신경망을 구성해서 테스트해 볼 수 있습니다.



<784개의 입력, 64개의 은닉 층, 10개의 출력 층>

01 2입력 2출력 인공 신경망 구현하기

다음 그림은 입력2 출력2로 구성된 인공 신경망과 순전파 역전파 수식을 나타냅니다. 우리는 다음 수식을 행렬 계산식으로 유도한 후, NumPy를 이용하여 인공 신경망을 구현해 봅니다.



*** ② x1b, x2b값은 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 y1b, y2b처럼 해당 인공 신경으로 역전파되는 값입니다. 역전파된 x1b, x2b값은 해당 인공 신경의 가중치와 편향 학습에 사용됩니다.

행렬 계산식 유도하기

이 그림을 통해 앞에서 우리는 다음 표의 왼쪽과 같은 수식을 유도했습니다. 이런 형태의 수식을 다원일차연립방정식이라고 합니다. 다원일차연립방정식은 행렬을 이용하면 깔끔하게 정리할 수 있습니다. 행렬 계산식으로 정리하면 다음 표의 오른쪽과 같습니다.

다원일차연립방정식	행렬 계산식

순 전 파	$\begin{bmatrix} x_1 w_1 + x_2 w_2 + 1b_1 = y_1 \\ x_1 w_3 + x_2 w_4 + 1b_2 = y_2 \end{bmatrix} \textcircled{1}$	$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \end{bmatrix} \textcircled{1}$
입 력 역 전 파	$\begin{bmatrix} y_{1b} w_1 + y_{2b} w_3 = x_{1b} \\ y_{1b} w_2 + y_{2b} w_4 = x_{2b} \end{bmatrix} \textcircled{2}$	$\begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} =$ $\begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix}^T = \begin{bmatrix} x_{1b} & x_{2b} \end{bmatrix} \textcircled{2}$
가 중 치 편 향 역 전 파	$\begin{bmatrix} x_1 y_{1b} = w_{1b} \\ x_2 y_{1b} = w_{2b} \\ x_1 y_{2b} = w_{3b} \\ x_2 y_{2b} = w_{4b} \end{bmatrix} \textcircled{3}$ $\begin{bmatrix} 1y_{1b} = b_{1b} \\ 1y_{2b} = b_{2b} \end{bmatrix} \textcircled{4}$	$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} =$ $\begin{bmatrix} x_1 & x_2 \end{bmatrix}^T \begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} \textcircled{3}$ $1 \begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} = \begin{bmatrix} b_{1b} & b_{2b} \end{bmatrix} \textcircled{4}$
인 공 신 경 망 학 습	$\begin{bmatrix} w_1 = w_1 - \alpha w_{1b} \\ w_2 = w_2 - \alpha w_{2b} \\ w_3 = w_3 - \alpha w_{3b} \\ w_4 = w_4 - \alpha w_{4b} \end{bmatrix} \textcircled{5}$ $\begin{bmatrix} b_1 = b_1 - \alpha b_{1b} \\ b_2 = b_2 - \alpha b_{2b} \end{bmatrix} \textcircled{6}$	$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \alpha \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} \textcircled{5}$ $\begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 \end{bmatrix} - \alpha \begin{bmatrix} b_{1b} & b_{2b} \end{bmatrix} \textcircled{6}$

이 표에서 몇 가지 계산에 주의할 행렬 계산식을 살펴봅니다.

순전파

행렬 계산식 ❶에서 다음은 순전파의 행렬 계산식이 일차연립방정식으로 해석되는 과정을 나타냅니다.

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \end{bmatrix}$$

$$X \quad W \quad B \quad Y$$

$$x_1 w_1 + x_2 w_2 + b_1 = y_1$$

$$x_1 w_3 + x_2 w_4 + b_2 = y_2$$

행렬의 곱 $X@W$ 는 앞에 오는 X 행렬의 가로줄 항목, 뒤에 오는 W 행렬의 세로줄 항목이 순서대로 곱해진 후, 모두 더해져서 임시 행렬(예를 들어, XW 행렬)의 항목 하나를 구성합니다. 그래서 X 행렬의 가로줄 항목 개수와 W 행렬의 세로줄 항목 개수는 같아야 합니다. 계속해서 XW 행렬의 각 항목은 B 행렬의 각 항목과 더해져 Y 행렬의 각 항목을 구성합니다.

*** 여기서 @ 문자는 행렬의 곱을 나타내기 위해 사용했습니다. 실제로 파이썬에서는 @문자를 이용하여 행렬의 곱을 수행합니다.

다음은 순전파의 행렬 계산식을 숫자로 표현한 구체적인 예입니다.

$$\begin{bmatrix} 2 & 3 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 19 & 30 \end{bmatrix}$$

$X \quad W \quad B \quad Y$

$$\begin{aligned} 2 \times 3 + 3 \times 4 + 1 &= 19 \\ 2 \times 5 + 3 \times 6 + 2 &= 30 \end{aligned}$$

입력 역전파

행렬 계산식 ②에서 다음은 순전파 때 사용된 가중치의 전치 행렬입니다.

$$\begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix}^T$$

전치행렬은 가로줄과 세로줄이 바뀐 행렬입니다.

다음은 입력 역전파의 행렬 계산식이 일차연립방정식으로 해석되는 과정을 나타냅니다.

$$\begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} = \begin{bmatrix} x_{1b} & x_{2b} \end{bmatrix}$$

$$Y_b \quad W^T \quad X_b$$

$$y_{1b}w_1 + y_{2b}w_3 = x_{1b}$$

$$y_{1b}w_2 + y_{2b}w_4 = x_{2b}$$

행렬의 곱 $Y_b @ W.T$ 는 앞에 오는 Y_b 행렬의 가로줄 항목, 뒤에 오는 $W.T$ 행렬의 세로줄 항목이 순서대로 곱해진 후, 모두 더해져서 X_b 행렬의 항목 하나를 구성합니다. 그래서 Y_b 행렬의 가로줄 항목 개수와 $W.T$ 행렬의 세로줄 항목 개수는 같아야 합니다. 또 $W.T$ 행렬의 가로줄 개수와 X_b 행렬의 가로줄 개수는 같아야 합니다.

*** 여기서 @ 문자는 행렬의 곱을 나타내기 위해 사용했습니다. 실제로 파이썬에서는 @문자를 이용하여 행렬의 곱을 수행합니다.

*** 여기서 W.T로 W 행렬의 전치행렬을 나타내기 위해 사용했습니다. 실제로 파이썬에서는 NumPy 행렬에 T문자를 점(.)으로 연결하여 전치 행렬을 나타냅니다.

다음은 입력 역전파의 행렬 계산식을 숫자로 표현한 구체적인 예입니다.

$$\begin{bmatrix} -8 & 60 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 276 & 328 \end{bmatrix}$$

$$Y_b \quad W^T \quad X_b$$

$$-8 \times 3 + 60 \times 5 = 276$$

$$-8 \times 4 + 60 \times 6 = 328$$

가중치 역전파

행렬 계산식 ⑥에서 다음은 순전파 때 사용된 입력의 전치 행렬입니다.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$$

다음은 가중치 역전파의 행렬 계산식이 일차연립방정식으로 해석되는 과정을 나타냅니다.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix}$$

$$X^T \quad Y_b \quad W_b$$

$$\begin{array}{ll} x_1 y_{1b} = w_{1b} & x_1 y_{2b} = w_{3b} \\ x_2 y_{1b} = w_{2b} & x_2 y_{2b} = w_{4b} \end{array}$$

행렬의 곱 $X.T@Y_b$ 는 앞에 오는 $X.T$ 행렬의 가로줄 항목 각각에 대해, 뒤에 오는 Y_b 행렬의 세로줄 항목 각각에 곱해진 후, W_b 행렬의 각각의 항목을 구성합니다.

*** 여기서 @ 문자는 행렬의 곱을 나타내기 위해 사용했습니다. 실제로 파이썬에서는 @문자를 이용하여 행렬의 곱을 수행합니다.

다음은 순전파의 행렬 계산식을 숫자로 표현한 구체적인 예입니다.

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} -8 & 60 \end{bmatrix} = \begin{bmatrix} -16 & 120 \\ -24 & 180 \end{bmatrix}$$

$$X^T \quad Y_b \quad W_b$$

$$\begin{array}{ll} 2 \times -8 = -16 & 2 \times 60 = 120 \\ 3 \times -8 = -24 & 3 \times 60 = 180 \end{array}$$

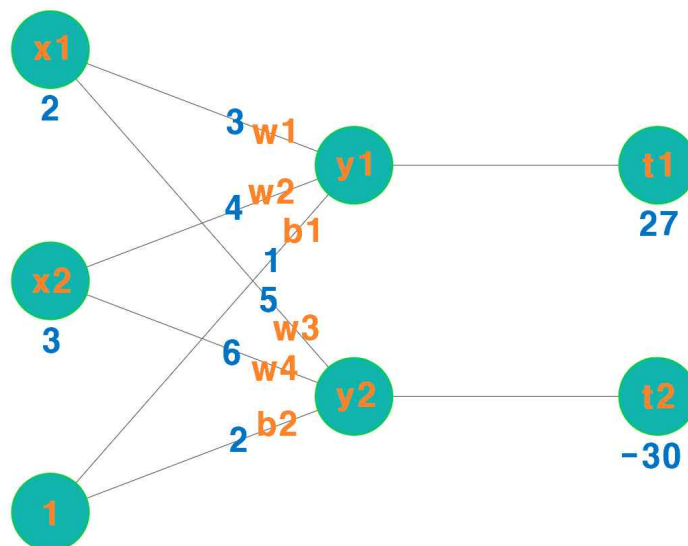
인공 신경망 행렬 계산식 정리하기

위 수식에서 표현된 행렬들에 다음 표의 왼쪽과 같이 이름을 붙여줍니다. 그러면 위의 행렬 계산식은 다음표의 오른쪽과 같이 정리할 수 있습니다. 오른쪽의 행렬 계산식은 행렬의 크기와 상관없이 성립합니다. 주의할 점은 행렬 곱은 순서를 변경하면 안 됩니다.

행렬 이름	인공 신경망 행렬 계산식
$\begin{bmatrix} x_1 & x_2 \end{bmatrix} = X$ $\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = W$ $\begin{bmatrix} b_1 & b_2 \end{bmatrix} = B$ $\begin{bmatrix} y_1 & y_2 \end{bmatrix} = Y$ $\begin{bmatrix} y_{1b} & y_{2b} \end{bmatrix} = Y_b$ $\begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix}^T = W^T$ $\begin{bmatrix} x_{1b} & x_{2b} \end{bmatrix} = X_b$ $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T = X^T$ $\begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} = W_b$ $\begin{bmatrix} b_{1b} & b_{2b} \end{bmatrix} = B_b$	<p>순전파</p> $Y = XW + B \quad \textcircled{1}$ <p>입력 역전파</p> $Y_b W^T = X_b \quad \textcircled{2}$ <p>가중치, 편향 역전파</p> $X^T Y_b = W_b \quad \textcircled{3}$ $1 Y_b = B_b \quad \textcircled{4}$ <p>인공 신경망 학습</p> $W = W - \alpha W_b \quad \textcircled{5}$ $B = B - \alpha B_b \quad \textcircled{6}$

NumPy로 인공 신경망 구현하기

지금까지 정리한 수식을 구현을 통해 살펴봅니다. 다음 그림을 살펴봅니다.



이 그림에서 입력 값 X, 가중치 W, 편향 B, 목표 값 T는 다음과 같습니다.

$$\begin{aligned} \begin{bmatrix} x_1 & x_2 \end{bmatrix} &= \begin{bmatrix} 2 & 3 \end{bmatrix} = X \\ \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} &= \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} = W \\ \begin{bmatrix} b_1 & b_2 \end{bmatrix} &= \begin{bmatrix} 1 & 2 \end{bmatrix} = B \\ \begin{bmatrix} t_1 & t_2 \end{bmatrix} &= \begin{bmatrix} 27 & -30 \end{bmatrix} = T \end{aligned}$$

X를 상수로 고정한 채 W, B에 대해 학습을 수행해 봅니다.

*** 이 값들은 임의의 값들입니다. 다른 값들을 사용하여 학습을 수행할 수도 있습니다.

1. 다음과 같이 예제를 작성합니다.

411_1.py

```
01 from ulab import numpy as np
02
03 X = np.array([[2, 3]])
04 T = np.array([[27, -30]])
05 W = np.array([[3, 5],
06               [4, 6]])
07 B = np.array([[1, 2]])
08
09 for epoch in range(1000):
10
11     print('epoch = %d' %epoch)
12
13     Y = np.dot(X, W) + B # ❶
14     print(' Y =', Y)
15
16     E = np.sum((Y - T) ** 2 / 2)
17     print(' E = %.7f' %E)
18     if E < 0.0000001:
19         break
20
21     Yb = Y - T
22     Xb = np.dot(Yb, W.T) # ❷
23     Wb = np.dot(X.T, Yb) # ❸
24     Bb = 1 * Yb # ❹
25     print(' Xb =\n', Xb)
26     print(' Wb =\n', Wb)
27     print(' Bb =\n', Bb)
```

```

28
29     lr = 0.01
30     W = W - lr * Wb # ⑤
31     B = B - lr * Bb # ⑥
32     print(' W  =\n', W)
33     print(' B  =\n', B)

```

01 : import문을 이용하여 ulab 모듈로부터 numpy 모듈을 np라는 이름으로 불러옵니다. numpy 모듈은 행렬 계산을 편하게 해주는 라이브러리입니다. 인공 신경망은 일반적으로 행렬 계산식으로 구성하게 됩니다. micropython에서는 ulab 모듈을 통해 numpy 모듈이 제한적으로 지원됩니다.

03 : np.array 함수를 호출하여 1x2 행렬을 생성하여 X 변수에 할당합니다.

04 : np.array 함수를 호출하여 1x2 행렬을 생성하여 T 변수에 할당합니다.

05, 06 : np.array 함수를 호출하여 2x2 행렬을 생성하여 W 변수에 할당합니다.

07 : np.array 함수를 호출하여 1x2 행렬을 생성하여 B 변수에 할당합니다.

09 : epoch값을 0에서 1000 미만까지 바꾸어가며 13~33줄을 1000회 수행합니다.

11 : print 함수를 호출하여 Y값을 출력합니다.

13 : np.dot 함수를 호출하여 입력 X와 가중치 W에 대해 행렬 곱을 수행한 후, 편향 B를 더해진 후, Y 변수에 할당합니다. np.dot 함수는 행렬 곱을 수행하는 함수입니다. 행렬 곱의 순서를 변경하지 않도록 주의합니다.

16 : 평균 제곱 오차를 구합니다.

17 : print 함수를 호출하여 E값을 출력합니다. 소수점 이하 7자리까지 출력합니다.

18, 19 : 평균 제곱 오차가 0.0000001(천만분의 1)보다 작으면 break문을 사용하여 11줄의 for 문을 빠져 나갑니다.

21 : 예측 값을 가진 Y 행렬에서 목표 값을 가진 T 행렬을 뺀 후, 결과 값을 Yb 변수에 할당합니다. Yb는 역전파 오차 값을 갖는 행렬입니다.

22 : Xb 변수를 선언한 후, 입력 값에 대한 역전파 값을 받아봅니다. 이 부분은 이 예제에서 필요한 부분은 아니며, 역전파 연습을 위해 추가하였습니다. W.T는 가중치 W의 전치 행렬을 내어줍니다. np.dot 함수를 호출하여 Yb와 W.T에 대해 행렬 곱을 수행한 후, 결과 값을 Xb 변수에 할당합니다. 행렬 곱의 순서를 변경하지 않도록 주의합니다.

23 : X.T는 입력 X의 전치 행렬을 내어줍니다. np.dot 함수를 호출하여 X.T와 Yb에 대해 행렬 곱을 수행한 후, 결과 값을 Wb 변수에 할당합니다. 행렬 곱의 순서를 변경하지 않도록 주의합니다.

24 : Yb 행렬에 1을 곱해주어 Bb에 할당합니다. 여기서 1은 수식을 강조하기 위해 생략하지 않았습니다.


25~27 : print 함수를 호출하여 Xb, Wb, Bb값을 출력합니다.

29 : lr 변수를 선언한 후, 0.01을 할당합니다. lr 변수는 학습률 변수입니다.

30 : 가중치를 갱신합니다.

31 : 편향을 갱신합니다.

32, 33 : print 함수를 호출하여 W, B값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 78
Y = array([[26.99994, -29.99953]], dtype=float32)
E = 0.0000001
Xb =
array([[ -0.001907044, -0.003529093]], dtype=float32)
Wb =
array([[ -0.0001182556, 0.0009307861],
       [ -0.0001773834, 0.001396179]], dtype=float32)
Bb =
array([[ -5.912781e-05, 0.0004653931]], dtype=float32)
W =
array([[4.142849, -3.57137],
       [5.714274, -6.857058]], dtype=float32)
B =
array([[1.571424, -2.285686]], dtype=float32)
epoch = 79
Y = array([[26.99995, -29.9996]], dtype=float32)
E = 0.0000001
```

(79+1)회 째 학습이 완료되는 것을 볼 수 있습니다.

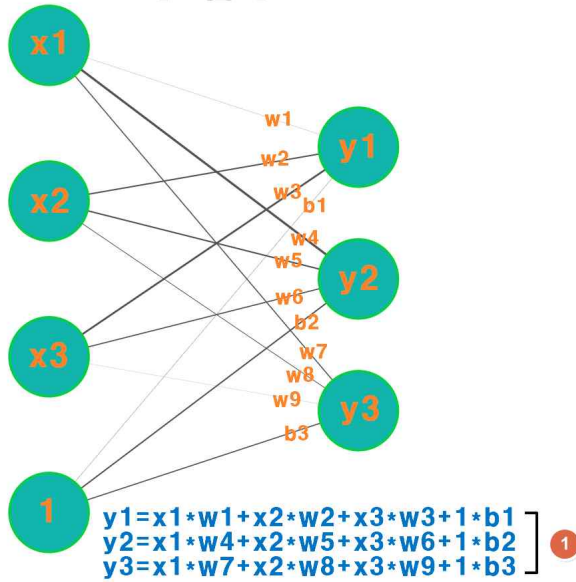
332_1.py 예제의 결과와 비교해 봅니다.

```
epoch = 78
y1, y2 = 27.000, -30.000
E = 0.0000001
x1b, x2b = -0.002, -0.004
w1b, w3b = -0.000, 0.001
w2b, w4b = -0.000, 0.001
b1b, b2b = -0.000, 0.000
w1, w3 = 4.143, -3.571
w2, w4 = 5.714, -6.857
b1, b2 = 1.571, -2.286
epoch = 79
y1, y2 = 27.000, -30.000
E = 0.0000001
```

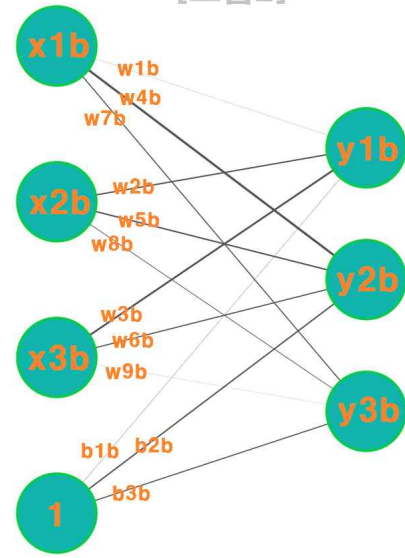
02 3입력 3출력 인공 신경망 구현하기

다음 그림은 입력3 출력3으로 구성된 인공 신경망과 순전파 역전파 수식을 나타냅니다. 우리는 다음 수식을 행렬 계산식으로 유도한 후, NumPy를 이용하여 인공 신경을 구현해 봅니다.

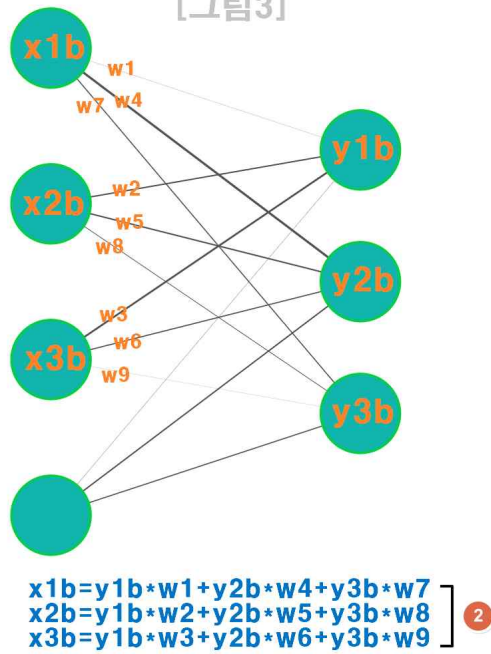
[그림1]



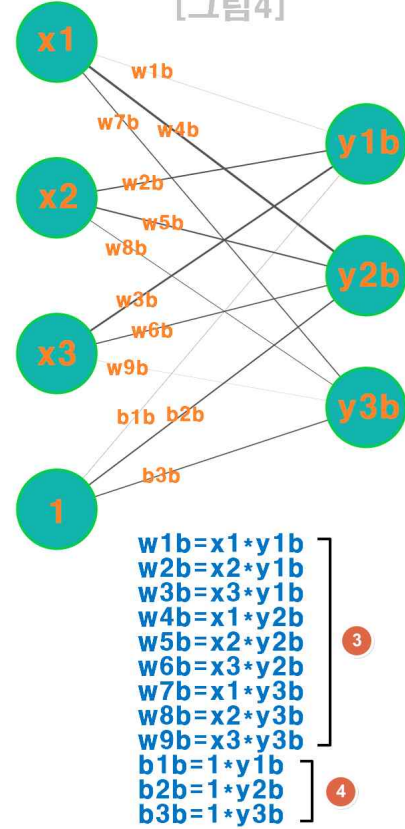
[그림2]



[그림3]



[그림4]



*** ② x1b, x2b, x3b값은 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 y1b, y2b, y3b처럼 해당 인공 신경으로 역전파되는 값입니다. 역전파된 x1b, x2b, x3b값은 해당 인공 신경의 가중치와 편향 학습에 사용됩니다.

행렬 계산식 유도하기

이 그림을 통해 앞에서 우리는 다음 표의 왼쪽과 같은 수식을 유도했습니다. 이런 형태의 수식을 다원일차연립방정식이라고 합니다. 다원일차연립방정식은 행렬을 이용하면 깔끔하게 정리할 수 있습니다. 행렬 계산식으로 정리하면 다음 표의 오른쪽과 같습니다.

	다원일차연립방정식	행렬 계산식
순 전 파	$\left. \begin{aligned} x_1w_1 + x_2w_2 + x_3w_3 + 1b_1 &= y_1 \\ x_1w_4 + x_2w_5 + x_3w_6 + 1b_2 &= y_2 \\ x_1w_7 + x_2w_8 + x_3w_9 + 1b_3 &= y_3 \end{aligned} \right\} \textcircled{1}$	$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_1 & w_4 & w_7 \\ w_2 & w_5 & w_8 \\ w_3 & w_6 & w_9 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \textcircled{1}$
입 력 역 전 파	$\left. \begin{aligned} y_{1b}w_1 + y_{2b}w_4 + y_{3b}w_7 &= x_{1b} \\ y_{1b}w_2 + y_{2b}w_5 + y_{3b}w_8 &= x_{2b} \\ y_{1b}w_3 + y_{2b}w_6 + y_{3b}w_9 &= x_{3b} \end{aligned} \right\} \textcircled{2}$	$\begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} = \begin{bmatrix} x_{1b} & x_{2b} & x_{3b} \end{bmatrix}$ $\begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} \begin{bmatrix} w_1 & w_4 & w_7 \\ w_2 & w_5 & w_8 \\ w_3 & w_6 & w_9 \end{bmatrix}^T = \begin{bmatrix} x_{1b} & x_{2b} & x_{3b} \end{bmatrix} \textcircled{2}$
가 중 치 편 향 역 전 파	$\left. \begin{aligned} x_1y_{1b} &= w_{1b} \\ x_2y_{1b} &= w_{2b} \\ x_3y_{1b} &= w_{3b} \\ x_1y_{2b} &= w_{4b} \\ x_2y_{2b} &= w_{5b} \\ x_3y_{2b} &= w_{6b} \\ x_1y_{3b} &= w_{7b} \\ x_2y_{3b} &= w_{8b} \\ x_3y_{3b} &= w_{9b} \end{aligned} \right\} \textcircled{3}$ $\left. \begin{aligned} 1y_{1b} &= b_{1b} \\ 1y_{2b} &= b_{2b} \\ 1y_{3b} &= b_{3b} \end{aligned} \right\} \textcircled{4}$	$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} =$ $\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{4b} & w_{7b} \\ w_{2b} & w_{5b} & w_{8b} \\ w_{3b} & w_{6b} & w_{9b} \end{bmatrix} \textcircled{3}$ $1 \begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} = \begin{bmatrix} b_{1b} & b_{2b} & b_{3b} \end{bmatrix} \textcircled{4}$

인 공 신 경 망 학 습	$\left[\begin{array}{l} w_1 = w_1 - \alpha w_{1b} \\ w_2 = w_2 - \alpha w_{2b} \\ w_3 = w_3 - \alpha w_{3b} \\ w_4 = w_4 - \alpha w_{4b} \\ w_5 = w_5 - \alpha w_{5b} \\ w_6 = w_6 - \alpha w_{6b} \\ w_7 = w_7 - \alpha w_{7b} \\ w_8 = w_8 - \alpha w_{8b} \\ w_9 = w_9 - \alpha w_{9b} \\ b_1 = b_1 - \alpha b_{1b} \\ b_2 = b_2 - \alpha b_{2b} \\ b_3 = b_3 - \alpha b_{3b} \end{array} \right]$	$\left[\begin{array}{l} w_1 \ w_4 \ w_7 \\ w_2 \ w_5 \ w_8 \\ w_3 \ w_6 \ w_9 \\ b_1 \ b_2 \ b_3 \end{array} \right] = \left[\begin{array}{l} w_1 \ w_4 \ w_7 \\ w_2 \ w_5 \ w_8 \\ w_3 \ w_6 \ w_9 \\ b_1 \ b_2 \ b_3 \end{array} \right] - \alpha \left[\begin{array}{l} w_{1b} \ w_{4b} \ w_{7b} \\ w_{2b} \ w_{5b} \ w_{8b} \\ w_{3b} \ w_{6b} \ w_{9b} \\ b_{1b} \ b_{2b} \ b_{3b} \end{array} \right]$
---------------------------------	--	---

행렬 계산식 ②에서 다음은 순전파 때 사용된 가중치의 전치 행렬입니다.

$$\left[\begin{array}{l} w_1 \ w_2 \ w_3 \\ w_4 \ w_5 \ w_6 \\ w_7 \ w_8 \ w_9 \end{array} \right] = \left[\begin{array}{l} w_1 \ w_4 \ w_7 \\ w_2 \ w_5 \ w_8 \\ w_3 \ w_6 \ w_9 \end{array} \right]^T$$

전치행렬은 가로줄과 세로줄이 바뀐 행렬입니다.

행렬 계산식 ③에서 다음은 순전파 때 사용된 입력의 전치 행렬입니다.

$$\left[\begin{array}{l} x_1 \\ x_2 \\ x_3 \end{array} \right] = \left[x_1 \ x_2 \ x_3 \right]^T$$

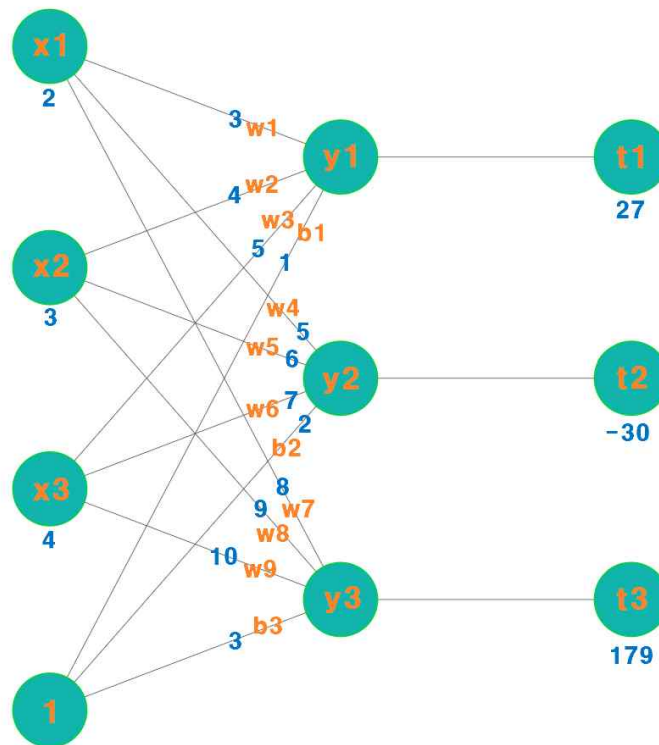
인공 신경망 행렬 계산식 정리하기

위 수식에서 표현된 행렬들에 다음 표의 왼쪽과 같이 이름을 붙여줍니다. 그러면 위의 행렬 계산식은 다음표의 오른쪽과 같이 정리할 수 있습니다. 오른쪽의 행렬 계산식은 행렬의 크기와 상관없이 성립합니다. 주의할 점은 행렬 곱은 순서를 변경하면 안 됩니다.

행렬 이름	인공 신경망 행렬 계산식
$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = X$ $\begin{bmatrix} w_1 & w_4 & w_7 \\ w_2 & w_5 & w_8 \\ w_3 & w_6 & w_9 \end{bmatrix} = W$ $\begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = B$ $\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = Y$ $\begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} = Y_b$ $\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} = \begin{bmatrix} w_1 & w_4 & w_7 \\ w_2 & w_5 & w_8 \\ w_3 & w_6 & w_9 \end{bmatrix}^T = W^T$ $\begin{bmatrix} x_{1b} & x_{2b} & x_{3b} \end{bmatrix} = X_b$ $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T = X^T$ $\begin{bmatrix} w_{1b} & w_{4b} & w_{7b} \\ w_{2b} & w_{5b} & w_{8b} \\ w_{3b} & w_{6b} & w_{9b} \end{bmatrix} = W_b$ $\begin{bmatrix} b_{1b} & b_{2b} & b_{3b} \end{bmatrix} = B_b$	<p>순전파</p> $Y = XW + B \quad \textcircled{1}$ <p>입력 역전파</p> $Y_b W^T = X_b \quad \textcircled{2}$ <p>가중치, 편향 역전파</p> $X^T Y_b = W_b \quad \textcircled{3}$ $1 Y_b = B_b \quad \textcircled{4}$ <p>인공 신경망 학습</p> $W = W - \alpha W_b \quad \textcircled{5}$ $B = B - \alpha B_b \quad \textcircled{6}$

NumPy로 인공 신경망 구현하기

지금까지 정리한 수식을 구현을 통해 살펴봅니다. 다음 그림을 살펴봅니다.



이 그림에서 입력 값 X , 가중치 W , 편향 B , 목표 값 T 는 다음과 같습니다.

$$\begin{aligned} [x_1 \ x_2 \ x_3] &= [2 \ 3 \ 4] = X \\ \begin{bmatrix} w_1 & w_4 & w_7 \\ w_2 & w_5 & w_8 \\ w_3 & w_6 & w_9 \end{bmatrix} &= \begin{bmatrix} 3 & 5 & 8 \\ 4 & 6 & 9 \\ 5 & 7 & 10 \end{bmatrix} = W \\ [b_1 \ b_2 \ b_3] &= [1 \ 2 \ 3] = B \\ [t_1 \ t_2 \ t_3] &= [27 \ -30 \ 179] = T \end{aligned}$$

X 를 상수로 고정한 채 W , B 에 대해 학습을 수행해 봅니다.

*** 이 값들은 임의의 값들입니다. 다른 값들을 사용하여 학습을 수행할 수도 있습니다.

1. 이전 예제를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

412_1.py

```
01 from ulab import numpy as np
02
03 X = np.array([[2, 3, 4]])
```




```

04 T = np.array([[27, -30, 179]])
05 W = np.array([[3, 5, 8],
06               [4, 6, 9],
07               [5, 7, 10]])
08 B = np.array([[1, 2, 3]])
09
10~끝 # 이전 예제와 같습니다.

```

03~08 : X, T, W, B를 변경해줍니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

epoch = 35
Y = array([[27.00005, -29.99967, 178.9997]], dtype=float32)
E = 0.0000001
Xb =
array([[[-0.005167882, -0.007223486, -0.009279093]], dtype=float32)
Wb =
array([[9.536743e-05, 0.000667572, -0.0007019043],
       [0.0001430511, 0.001001358, -0.001052856],
       [0.0001907349, 0.001335144, -0.001403809]], dtype=float32)
Bb =
array([[4.768372e-05, 0.000333786, -0.0003509521]], dtype=float32)
W =
array([[2.200002, -0.8666516, 14.19999],
       [2.800003, -2.799977, 18.29997],
       [3.400004, -4.733302, 22.39997]], dtype=float32)
B =
array([[0.6000011, -0.9333257, 6.099992]], dtype=float32)
epoch = 36
Y = array([[27.00003, -29.99977, 178.9998]], dtype=float32)
E = 0.0000001

```

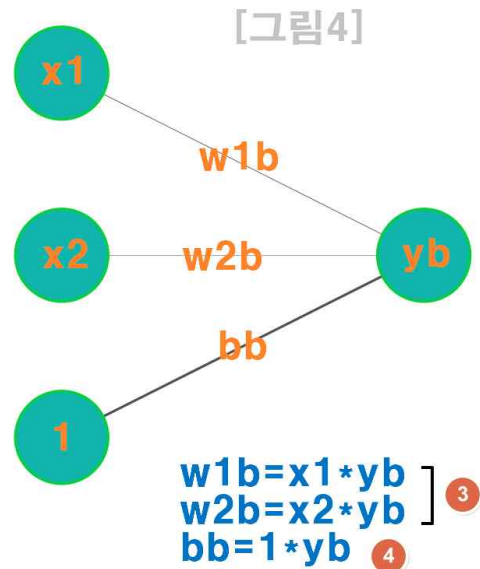
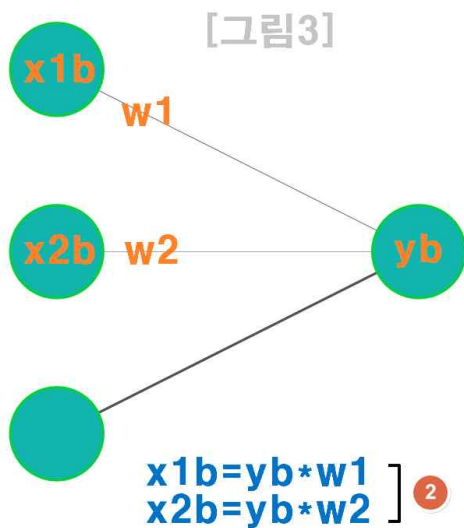
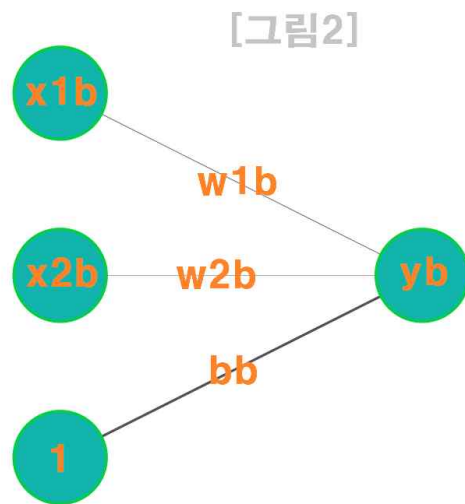
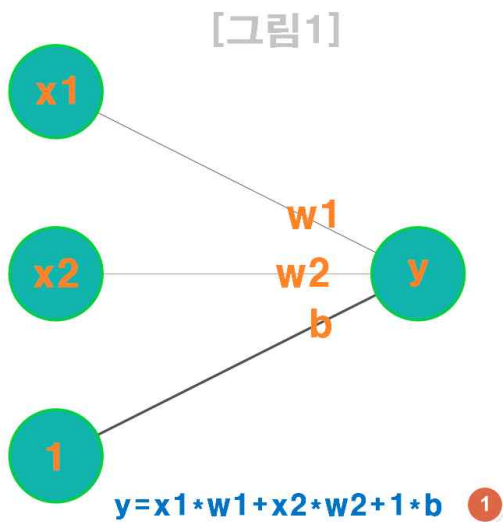
36회 째 학습이 완료되는 것을 볼 수 있습니다.

333_1 예제의 결과와 비교해 봅니다.

```
epoch = 35
y1, y2, y3 = 27.000, -30.000, 179.000
E = 0.00000001
x1b, x2b, x2b = -0.005, -0.007, -0.009
w1b, w4b, w7b = 0.000, 0.001, -0.001
w2b, w5b, w8b = 0.000, 0.001, -0.001
w3b, w6b, w9b = 0.000, 0.001, -0.001
b1b, b2b, b3b = 0.000, 0.000, -0.000
w1, w4, w7 = 2.200, -0.867, 14.200
w2, w5, w8 = 2.800, -2.800, 18.300
w3, w6, w9 = 3.400, -4.733, 22.400
b1, b2, b3 = 0.600, -0.933, 6.100
epoch = 36
y1, y2, y3 = 27.000, -30.000, 179.000
E = 0.00000001
```

03 2입력 1출력 인공 신경 구현하기

다음 그림은 입력2 출력1로 구성된 인공 신경과 순전파 역전파 수식을 나타냅니다. 우리는 다음 수식을 행렬 계산식으로 유도한 후, NumPy를 이용하여 인공 신경을 구현해 봅니다.



*** ② x1b, x2b값은 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 yb처럼 해당 인공 신경으로 역전파되는 값입니다. 역전파된 x1b, x2b값은 해당 인공 신경의 가중치와 편향 학습에 사용됩니다.

행렬 계산식 유도하기

이 그림을 통해 앞에서 우리는 다음 표의 왼쪽과 같은 수식을 유도했습니다. 이런 형태의 수식을 다원일차연립방정식이라고 합니다. 다원일차연립방정식은 행렬을 이용하면 깔끔하게 정리할 수 있습니다. 행렬 계산식으로 정리하면 다음 표의 오른쪽과 같습니다.

	다원일차연립방정식	행렬 계산식
순전파	$x_1w_1 + x_2w_2 + 1b = y$ ❶	$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + [b] = [y]$ ❶
입력 역전파	$\begin{bmatrix} y_bw_1 = x_{1b} \\ y_bw_2 = x_{2b} \end{bmatrix}$ ❷	$\begin{bmatrix} y_b \end{bmatrix} \begin{bmatrix} w_1 & w_2 \end{bmatrix} =$ $\begin{bmatrix} y_b \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}^T = [x_{1b} \ x_{2b}]$ ❷
가중치, 편향 역전파	$\begin{bmatrix} x_1y_b = w_{1b} \\ x_2y_b = w_{2b} \end{bmatrix}$ ❸ $1y_b = b_b$ ❹	$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} [y_b] =$ $\begin{bmatrix} x_1 & x_2 \end{bmatrix}^T [y_b] = \begin{bmatrix} w_{1b} \\ w_{2b} \end{bmatrix}$ ❸ $1[y_b] = [b_b]$ ❹
인공 신경망 학습	$\begin{bmatrix} w_1 = w_1 - \alpha w_{1b} \\ w_2 = w_2 - \alpha w_{2b} \end{bmatrix}$ ❺ $b = b - \alpha b_b$ ❻	$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \alpha \begin{bmatrix} w_{1b} \\ w_{2b} \end{bmatrix}$ ❺ $[b] = [b] - \alpha [b_b]$ ❻

행렬 계산식 ❷에서 다음은 순전파 때 사용된 가중치의 전치 행렬입니다.

$$\begin{bmatrix} w_1 & w_2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}^T$$

전치행렬은 가로줄과 세로줄이 바뀐 행렬입니다.

행렬 계산식 ❸에서 다음은 순전파 때 사용된 입력의 전치 행렬입니다.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$$

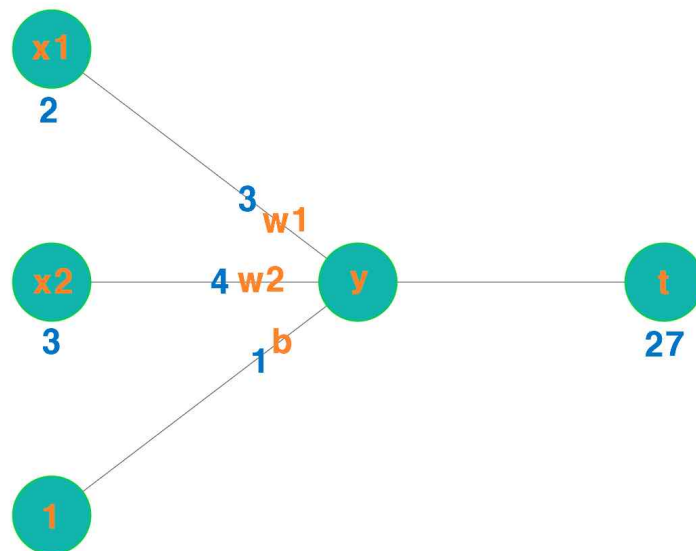
인공 신경망 행렬 계산식 정리하기

위 수식에서 표현된 행렬들에 다음 표의 왼쪽과 같이 이름을 붙여줍니다. 그러면 위의 행렬 계산식은 다음표의 오른쪽과 같이 정리할 수 있습니다. 오른쪽의 행렬 계산식은 행렬의 크기와 상관없이 성립합니다. 주의할 점은 행렬 곱은 순서를 변경하면 안 됩니다.

행렬 이름	인공 신경망 행렬 계산식
$[x_1 \ x_2] = X$ $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = W$ $[b] = B$ $[y] = Y$ $[y_b] = Y_b$ $[w_1 \ w_2] = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}^T = W^T$ $[x_{1b} \ x_{2b}] = X_b$ $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = [x_1 \ x_2]^T = X^T$ $\begin{bmatrix} w_{1b} \\ w_{2b} \end{bmatrix} = W_b$ $[b_b] = B_b$	<p>순전파</p> $Y = XW + B \quad \textcircled{1}$ <p>입력 역전파</p> $Y_b W^T = X_b \quad \textcircled{2}$ <p>가중치, 편향 역전파</p> $X^T Y_b = W_b \quad \textcircled{3}$ $1 Y_b = B_b \quad \textcircled{4}$ <p>인공 신경망 학습</p> $W = W - \alpha W_b \quad \textcircled{5}$ $B = B - \alpha B_b \quad \textcircled{6}$

NumPy로 인공 신경망 구현하기

지금까지 정리한 수식을 구현을 통해 살펴봅니다. 다음 그림을 살펴봅니다.



이 그림에서 입력 값 X , 가중치 W , 편향 B , 목표 값 T 는 다음과 같습니다.

$$\begin{aligned} \begin{bmatrix} x_1 & x_2 \end{bmatrix} &= \begin{bmatrix} 2 & 3 \end{bmatrix} = X \\ \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} &= \begin{bmatrix} 3 \\ 4 \end{bmatrix} = W \\ b &= 1 = B \\ t &= 27 = T \end{aligned}$$

X를 상수로 고정한 채 W, B에 대해 학습을 수행해 봅니다.


*** 이 값들은 임의의 값들입니다. 다른 값들을 사용하여 학습을 수행할 수도 있습니다.

1. 이전 예제를 복사합니다.
2. 다음과 같이 예제를 수정합니다.

413_1.py

```
01 from ulab import numpy as np
02
03 X = np.array([[2, 3]])
04 T = np.array([[27]])
05 W = np.array([[3],
06               [4]])
07 B = np.array([1])
08
09~끝 # 이전 예제와 같습니다.
```

03~07 : X, T, W, B를 변경해줍니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

epoch = 64
Y = array([[26.99949]], dtype=float32)
E = 0.0000001
Xb =
array([[-0.002125566, -0.002931811]], dtype=float32)
Wb =
array([[-0.001026154],
        [-0.00153923]], dtype=float32)
Bb =
array([[-0.0005130768]], dtype=float32)
W =
array([[4.142794],
        [5.714191]], dtype=float32)
B =
array([[1.571397]], dtype=float32)
epoch = 65
Y = array([[26.99956]], dtype=float32)
E = 0.0000001

```

65회 째 학습이 완료되는 것을 볼 수 있습니다.

331_1.py 예제의 결과와 비교해 봅니다.

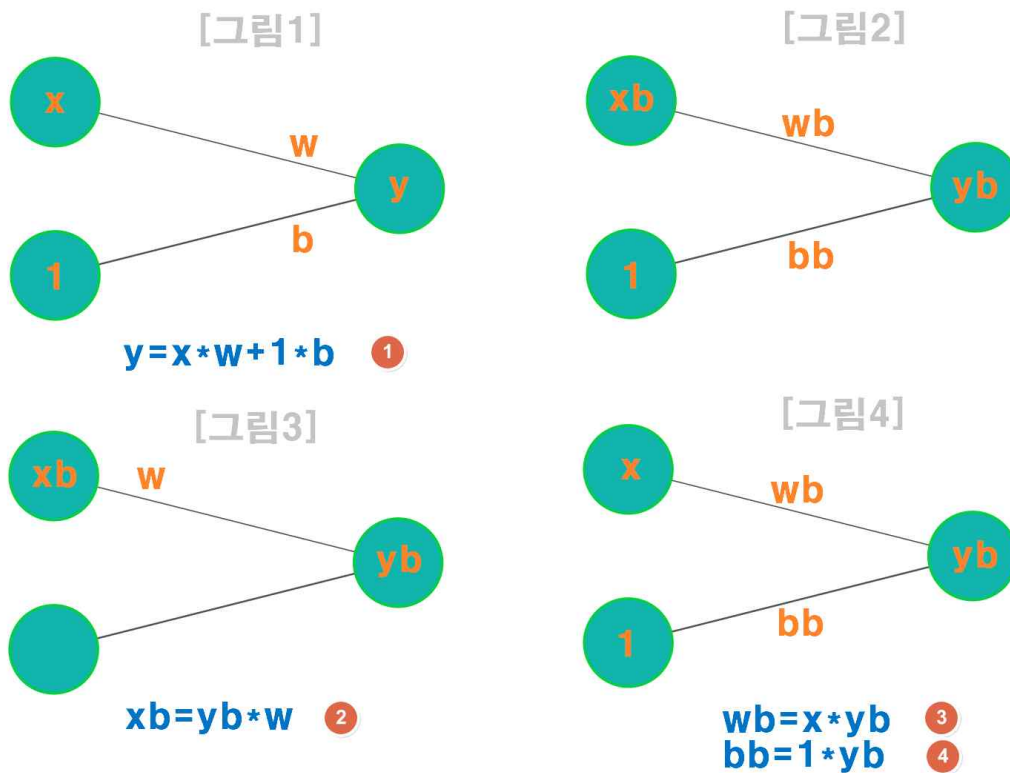
```

epoch = 64
y = 26.999
E = 0.0000001
x1b, x2b = -0.002, -0.003
w1b, w2b, bb = -0.001, -0.002, -0.001
w1, w2, b = 4.143, 5.714, 1.571
epoch = 65
y = 27.000
E = 0.0000001

```

04 1입력 1출력 인공 신경 구현하기

다음 그림은 입력1 출력1로 구성된 인공 신경과 순전파 역전파 수식을 나타냅니다. 우리는 다음 수식을 행렬 계산식으로 유도한 후, NumPy를 이용하여 인공 신경을 구현해 봅니다.



*** ② xb값은 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 yb처럼 해당 인공 신경으로 역전파되는 값입니다. 역전파된 xb값은 해당 인공 신경의 가중치와 편향 학습에 사용됩니다.

행렬 계산식 유도하기

이 그림을 통해 앞에서 우리는 다음 표의 왼쪽과 같은 수식을 유도했습니다. 이런 형태의 수식을 다원일차연립방정식이라고 합니다. 다원일차연립방정식은 행렬을 이용하면 깔끔하게 정리할 수 있습니다. 행렬 계산식으로 정리하면 다음 표의 오른쪽과 같습니다.

	다원일차연립방정식	행렬 계산식
순전파	$xw + 1b = y$ ①	$[x][w] + 1[b] = [y]$ ①
입력 역전파	$y_b w = x_b$ ②	$[y_b][w] =$ $[y_b][w]^T = [x_b]$ ②
가중치, 편향 역전파	$xy_b = w_b$ ③ $1y_b = b_b$ ④	$[x][y_b] =$ $[x]^T[y_b] = [w_b]$ ③ $1[y_b] = [b_b]$ ④

인공 신경망 학습	$w = w - \alpha w_b$ 5 $b = b - \alpha b_b$ 6	$[w] = [w] - \alpha [w_b]$ 5 $[b] = [b] - \alpha [b_b]$ 6
-----------	--	--

행렬 계산식 ②에서 다음은 순전파 때 사용된 가중치의 전치 행렬입니다.

$$[w] = [w]^T$$

전치행렬은 가로줄과 세로줄이 바뀐 행렬입니다. 여기서 가중치는 1x1 행렬이며 전치 행렬과 원래 행렬의 모양은 같습니다. 여기서는 수식을 일반화하기 위해 전치 행렬 형태로 표현하고 있습니다.

행렬 계산식 ③에서 다음은 순전파 때 사용된 입력의 전치 행렬입니다.

$$[x] = [x]^T$$

여기서 입력은 1x1 행렬이며 전치 행렬과 원래 행렬의 모양은 같습니다. 여기서는 수식을 일반화하기 위해 전치 행렬 형태로 표현하고 있습니다.

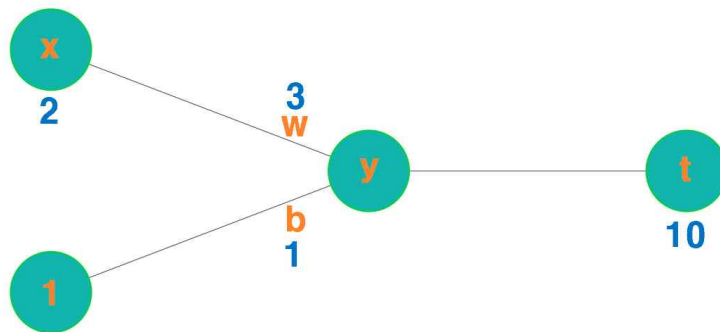
인공 신경망 행렬 계산식 정리하기

위 수식에서 표현된 행렬들에 다음 표의 왼쪽과 같이 이름을 붙여줍니다. 그러면 위의 행렬 계산식은 다음표의 오른쪽과 같이 정리할 수 있습니다. 오른쪽의 행렬 계산식은 행렬의 크기와 상관없이 성립합니다. 주의할 점은 행렬 곱은 순서를 변경하면 안 됩니다.

행렬 이름	인공 신경망 행렬 계산식
$[x] = X$ $[w] = W$ $[b] = B$ $[y] = Y$ $[y_b] = Y_b$ $[w] = [w]^T = W^T$ $[x_b] = X_b$ $[x] = [x]^T = X^T$ $[w_b] = W_b$ $[b_b] = B_b$	<p>순전파</p> $Y = XW + B \quad \textcircled{1}$ <p>입력 역전파</p> $Y_b W^T = X_b \quad \textcircled{2}$ <p>가중치, 편향 역전파</p> $X^T Y_b = W_b \quad \textcircled{3}$ $1 Y_b = B_b \quad \textcircled{4}$ <p>인공 신경망 학습</p> $W = W - \alpha W_b \quad \textcircled{5}$ $B = B - \alpha B_b \quad \textcircled{6}$

NumPy로 인공 신경망 구현하기

지금까지 정리한 수식을 구현을 통해 살펴봅니다. 다음 그림을 살펴봅니다.



이 그림에서 입력 값 X , 가중치 W , 편향 B , 목표 값 T 는 다음과 같습니다.

$$\begin{aligned}
 [x] &= [2] = X \\
 [w] &= [3] = W \\
 [b] &= [1] = B \\
 [t] &= [10] = T
 \end{aligned}$$

X 를 상수로 고정한 채 W , B 에 대해 학습을 수행해 봅니다.


*** 이 값들은 임의의 값들입니다. 다른 값들을 사용하여 학습을 수행할 수도 있습니다.

1. 이전 예제를 복사합니다.
2. 다음과 같이 예제를 수정합니다.

414_1.py

```
01 from ulab import numpy as np
02
03 X = np.array([[2]])
04 T = np.array([[10]])
05 W = np.array([[3]])
06 B = np.array([[1]])
07
08~끝 # 이전 예제와 같습니다.
```

03~06 : X, T, W, B를 변경해줍니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 171
Y = array([[9.999535]], dtype=float32)
E = 0.0000001
Xb =
array([[ -0.001954564]], dtype=float32)
Wb =
array([[ -0.0009307861]], dtype=float32)
Bb =
array([[ -0.0004653931]], dtype=float32)
W =
array([[4.199824]], dtype=float32)
B =
array([[1.599911]], dtype=float32)
epoch = 172
Y = array([[9.999558]], dtype=float32)
E = 0.0000001
```

172회 째 학습이 완료되는 것을 볼 수 있습니다.

322_1.py 예제의 결과와 비교해 봅니다.

```
epoch = 171
y = 10.000
E = 0.0000001
xb = -0.002, wb = -0.001, bb = -0.000
x = 2.000, w = 4.200, b = 1.600
epoch = 172
y = 10.000
E = 0.0000001
```

05 행렬 계산식과 1입력 1출력 수식 비교하기

지금까지의 내용을 정리하면 일반적인 인공 신경의 행렬 계산식은 다음과 같습니다. 그리고 입력1 출력1 인공 신경의 수식은 표의 오른쪽 기본 수식과 같습니다. 행렬 계산식의 구조가 기본 수식의 구조와 같은 것을 볼 수 있습니다.

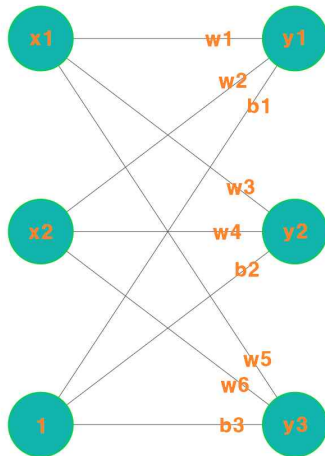
인공 신경망 동작	행렬 계산식	기본 수식
순전파	$Y = XW + B$	$y = xw + b$
입력 역전파	$X_b = Y_b W^T$	$x_b = y_b w$
가중치, 편향 역전파	$W_b = X^T Y_b$ $B_b = Y_b$	$w_b = x y_b$ $b_b = y_b$
인공 신경망 학습	$W = W - \alpha W_b$ $B = B - \alpha B_b$	$w = w - \alpha w_b$ $b = b - \alpha b_b$

*** 행렬 곱 연산은 순서를 지켜야 합니다.

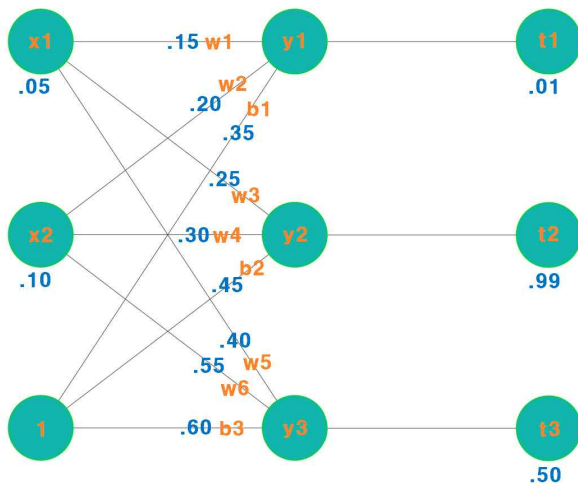
연습문제

❶ 2입력 3출력

1. 다음은 입력2 출력3의 인공 신경망입니다. 이 인공 신경망의 순전파, 역전파 행렬 계산식을 구합니다.

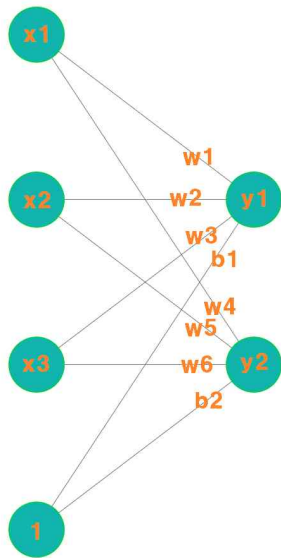


2. 앞에서 구한 행렬 계산식을 이용하여 다음과 같이 초기화된 인공 신경망을 NumPy를 이용하여 구현하고 학습시켜 봅니다. 입력 값 X 는 상수로 처리합니다.

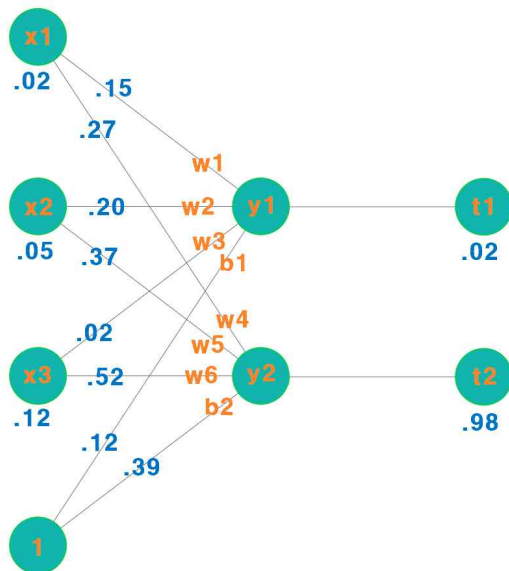


② 3입력 2출력

1. 다음은 입력3 출력2의 인공 신경망입니다. 이 인공 신경망의 순전파, 역전파 행렬 계산식을 구합니다.



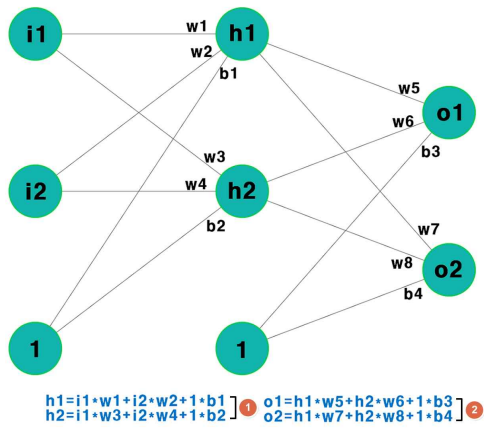
2. 앞에서 구한 행렬 계산식을 이용하여 다음과 같이 초기화된 인공 신경망을 NumPy를 이용하여 구현하고 학습시켜 봅니다. 입력 값 X는 상수로 처리합니다.



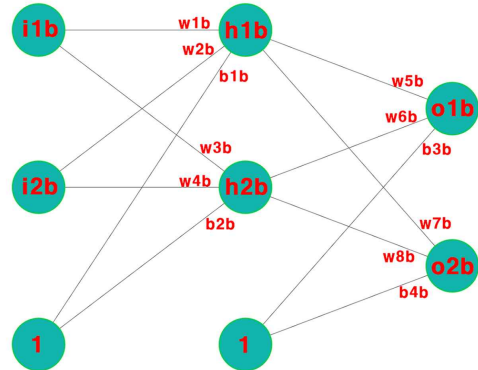
06 2입력 2은닉 2출력 인공 신경망 구현하기

다음 그림은 입력2 은닉2 출력2로 구성된 인공 신경망과 순전파 역전파 수식을 나타냅니다. 우리는 다음 수식을 행렬 계산식으로 유도한 후, NumPy를 이용하여 인공 신경망을 구현해 봅니다.

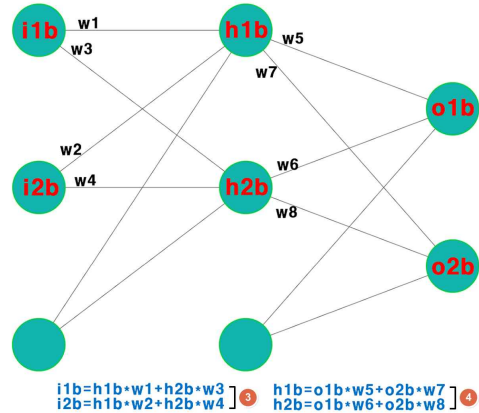
[그림1]



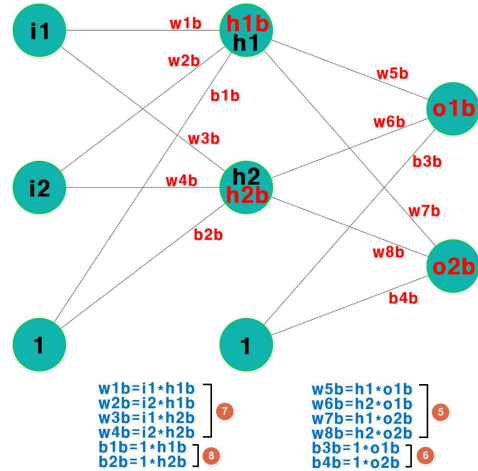
[그림2]



[그림3]



[그림4]



*** ㉓ i1b, i2b값은 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 h1b, h2b처럼 해당 인공 신경으로 역전파되는 값입니다. 역전파된 i1b, i2b값은 해당 인공 신경의 가중치와 편향 학습에 사용됩니다. 여기서 i1, i2는 은닉 층에 연결된 입력 층이므로 i1b, i2b의 수식은 필요치 않습니다.

행렬 계산식 유도하기

이 그림을 통해 앞에서 우리는 다음 표의 왼쪽과 같은 수식을 유도했습니다. 이런 형태의 수식을 다원일차연립방정식이라고 합니다. 다원일차연립방정식은 행렬을 이용하면 깔끔하게 정리할 수 있습니다. 행렬 계산식으로 정리하면 다음 표의 오른쪽과 같습니다.

다원일차연립방정식	행렬 계산식
-----------	--------

순전파	$\begin{bmatrix} i_1 w_1 + i_2 w_2 + 1b_1 = h_1 \\ i_1 w_3 + i_2 w_4 + 1b_2 = h_2 \\ h_1 w_5 + h_2 w_6 + 1b_3 = o_1 \\ h_1 w_7 + h_2 w_8 + 1b_4 = o_2 \end{bmatrix} \begin{matrix} \textcircled{1} \\ \textcircled{2} \end{matrix}$	$\begin{bmatrix} i_1 & i_2 \end{bmatrix} \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 \end{bmatrix} \textcircled{1}$ $\begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix} + \begin{bmatrix} b_3 & b_4 \end{bmatrix} = \begin{bmatrix} o_1 & o_2 \end{bmatrix} \textcircled{2}$
입력역전파	$\begin{bmatrix} o_{1b} w_5 + o_{2b} w_7 = h_{1b} \\ o_{1b} w_6 + o_{2b} w_8 = h_{2b} \end{bmatrix} \textcircled{4}$	$\begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} \begin{bmatrix} w_5 & w_6 \\ w_7 & w_8 \end{bmatrix} = \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix}$ $\begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix}^T = \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} \textcircled{4}$
가중치편향역전파	$\begin{bmatrix} i_1 h_{1b} = w_{1b} \\ i_2 h_{1b} = w_{2b} \\ i_1 h_{2b} = w_{3b} \\ i_2 h_{2b} = w_{4b} \end{bmatrix} \textcircled{7}$ $\begin{bmatrix} 1h_{1b} = b_{1b} \\ 1h_{2b} = b_{2b} \end{bmatrix} \textcircled{8}$ $\begin{bmatrix} h_1 o_{1b} = w_{5b} \\ h_2 o_{1b} = w_{6b} \\ h_1 o_{2b} = w_{7b} \\ h_2 o_{2b} = w_{8b} \end{bmatrix} \textcircled{5}$ $\begin{bmatrix} 1o_{1b} = b_{3b} \\ 1o_{2b} = b_{4b} \end{bmatrix} \textcircled{6}$	$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} \textcircled{7}$ $\begin{bmatrix} i_1 & i_2 \end{bmatrix}^T \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix}$ $1 \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} = \begin{bmatrix} b_{1b} & b_{2b} \end{bmatrix} \textcircled{8}$ $\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} = \begin{bmatrix} w_{5b} & w_{7b} \\ w_{6b} & w_{8b} \end{bmatrix} \textcircled{5}$ $\begin{bmatrix} h_1 & h_2 \end{bmatrix}^T \begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} = \begin{bmatrix} w_{5b} & w_{7b} \\ w_{6b} & w_{8b} \end{bmatrix}$ $1 \begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} = \begin{bmatrix} b_{3b} & b_{4b} \end{bmatrix} \textcircled{6}$
인공신경망학습	$\begin{bmatrix} w_1 = w_1 - \alpha w_{1b} \\ w_2 = w_2 - \alpha w_{2b} \\ w_3 = w_3 - \alpha w_{3b} \\ w_4 = w_4 - \alpha w_{4b} \\ b_1 = b_1 - \alpha b_{1b} \\ b_2 = b_2 - \alpha b_{2b} \end{bmatrix} \begin{matrix} \textcircled{11} \\ \textcircled{12} \end{matrix}$ $\begin{bmatrix} w_5 = w_5 - \alpha w_{5b} \\ w_6 = w_6 - \alpha w_{6b} \\ w_7 = w_7 - \alpha w_{7b} \\ w_8 = w_8 - \alpha w_{8b} \\ b_3 = b_3 - \alpha b_{3b} \\ b_4 = b_4 - \alpha b_{4b} \end{bmatrix} \begin{matrix} \textcircled{9} \\ \textcircled{10} \end{matrix}$	$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \alpha \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} \textcircled{11}$ $\begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 \end{bmatrix} - \alpha \begin{bmatrix} b_{1b} & b_{2b} \end{bmatrix} \textcircled{12}$ $\begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix} = \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix} - \alpha \begin{bmatrix} w_{5b} & w_{7b} \\ w_{6b} & w_{8b} \end{bmatrix} \textcircled{9}$ $\begin{bmatrix} b_3 & b_4 \end{bmatrix} = \begin{bmatrix} b_3 & b_4 \end{bmatrix} - \alpha \begin{bmatrix} b_{3b} & b_{4b} \end{bmatrix} \textcircled{10}$

행렬 계산식 ④에서 다음은 순전파 때 사용된 가중치의 전치 행렬입니다.

$$\begin{bmatrix} w_5 & w_6 \\ w_7 & w_8 \end{bmatrix} = \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix}^T$$

전치행렬은 가로줄과 세로줄이 바뀐 행렬입니다.

행렬 계산식 ⑦, ⑤에서 다음은 순전파 때 사용된 입력의 전치 행렬입니다.

$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = [i_1 \ i_2]^T$$

$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = [h_1 \ h_2]^T$$

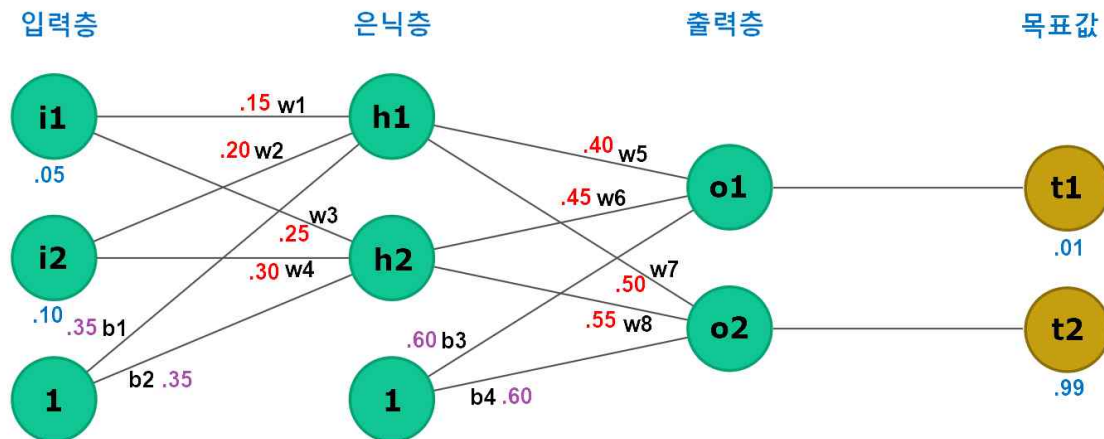
인공 신경망 행렬 계산식 정리하기

위 수식에서 표현된 행렬들에 다음 표의 왼쪽과 같이 이름을 붙여줍니다. 그러면 위의 행렬 계산식은 다음표의 오른쪽과 같이 정리할 수 있습니다. 오른쪽의 행렬 계산식은 행렬의 크기와 상관없이 성립합니다. 주의할 점은 행렬 곱은 순서를 변경하면 안 됩니다.

행렬 이름	인공 신경망 행렬 계산식
	순전파
	$H = IW_h + B_h$ ①
	$O = HW_o + B_o$ ②
	역전파
	$O_b W_o^T = H_b$ ④
	가중치, 편향 역전파
	$I^T H_b = W_{hb}$ ⑦
	$1 H_b = B_{hb}$ ⑧
	$H^T O_b = W_{ob}$ ⑤
	$1 O_b = B_{ob}$ ⑥
	인공 신경망 학습
	$W_h = W_h - \alpha W_{hb}$ ⑪
	$B_h = B_h - \alpha B_{hb}$ ⑫
	$W_o = W_o - \alpha W_{ob}$ ⑨
	$B_o = B_o - \alpha B_{ob}$ ⑩
$[i_1 \ i_2] = I$ $\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = W_h$ $[b_1 \ b_2] = B_h$ $[h_1 \ h_2] = H$ $\begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix} = W_o$ $[b_1 \ b_2] = B_o$ $[o_1 \ o_2] = O$ $[o_{1b} \ o_{2b}] = O_b$ $\begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix}^T = W_o^T$	$[h_{1b} \ h_{2b}] = H_b$ $[i_1 \ i_2]^T = I^T$ $\begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} = W_{hb}$ $[b_{1b} \ b_{2b}] = B_{hb}$ $[h_1 \ h_2]^T = H^T$ $\begin{bmatrix} w_{5b} & w_{7b} \\ w_{6b} & w_{8b} \end{bmatrix} = W_{ob}$ $[b_{3b} \ b_{4b}] = B_{ob}$

NumPy로 인공 신경망 구현하기

지금까지 정리한 수식을 구현을 통해 살펴봅니다. 다음 그림을 살펴봅니다.



이 그림에서 입력 값 I , 가중치 W_h , W_o , 편향 B_h , B_o , 목표 값 T 는 다음과 같습니다.

$$\begin{aligned} [i_1 \ i_2] &= [.05 \ .10] = I \\ \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} &= \begin{bmatrix} .15 & .25 \\ .20 & .30 \end{bmatrix} = W_h \\ [b_1 \ b_2] &= [.35 \ .35] = B_h \\ \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix} &= \begin{bmatrix} .40 & .50 \\ .45 & .55 \end{bmatrix} = W_o \\ [b_3 \ b_4] &= [.60 \ .60] = B_o \\ [t_1 \ t_2] &= [.01 \ .99] = T \end{aligned}$$

I 를 상수로 고정한 채 W_h , W_o , B_h , B_o 에 대해 학습을 수행해 봅니다.

*** 이 값들은 임의의 값들입니다. 다른 값들을 사용하여 학습을 수행할 수도 있습니다.

1. 다음과 같이 예제를 작성합니다.


416_1.py

```
01 from ulab import numpy as np
02
03 I = np.array([[.05, .10]])
04 T = np.array([[.01, .99]])
05 WH = np.array([[.15, .25],
```

```

06         [.20, .30]])
07 BH = np.array([[.35, .35]])
08 WO = np.array([[.40, .50],
09                 [.45, .55]])
10 BO = np.array([[.60, .60]])
11
12 for epoch in range(1000):
13
14     print('epoch = %d' %epoch)
15
16     H = np.dot(I, WH) + BH # ❶
17     O = np.dot(H, WO) + BO # ❷
18     print(' O  =\n', O)
19
20     E = np.sum((O - T) ** 2 / 2)
21     print(' E  = %.7f' %E)
22     if E < 0.0000001:
23         break
24
25     Ob = O - T
26     Hb = np.dot(Ob, WO.T) # ❸
27     WHb = np.dot(I.T, Hb) # ❹
28     BHb = 1 * Hb # ❺
29     WOb = np.dot(H.T, Ob) # ❻
30     BOb = 1 * Ob # ❼
31     print(' WHb =\n', WHb)
32     print(' BHb =\n', BHb)
33     print(' WOb =\n', WOb)
34     print(' BOb =\n', BOb)
35
36     lr = 0.01
37     WH = WH - lr * WHb # ❾
38     BH = BH - lr * BHb # ❿
39     WO = WO - lr * WOb # ⓫
40     BO = BO - lr * BOb # ⓬
41     print(' WH  =\n', WH)
42     print(' BH  =\n', BH)
43     print(' WO  =\n', WO)
44     print(' BO  =\n', BO)

```

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 664
O =
array([[0.01041593, 0.9898296]], dtype=float32)
E = 0.0000001
WHb =
array([[ -3.292622e-07,  2.892826e-07],
       [ -6.585243e-07,  5.785652e-07]], dtype=float32)
BHb =
array([[ -6.585243e-06,  5.785652e-06]], dtype=float32)
WOb =
array([[9.938012e-05, -4.071711e-05],
       [9.418975e-05, -3.859055e-05]], dtype=float32)
BOb =
array([[0.0004159268, -0.0001704097]], dtype=float32)
WH =
array([[0.1431573, 0.2418005],
       [0.1863148, 0.2836011]], dtype=float32)
BH =
array([[0.2131473, 0.1860073]], dtype=float32)
WO =
array([[0.2027305, 0.5334602],
       [0.2526767, 0.582771]], dtype=float32)
BO =
array([[ -0.09524895,  0.7303955]], dtype=float32)
epoch = 665
O =
array([[0.01041131, 0.9898315]], dtype=float32)
E = 0.0000001
```

(665+1)회 째 학습이 완료되는 것을 볼 수 있습니다.

334_1.py 예제의 결과와 비교해 봅니다.

```

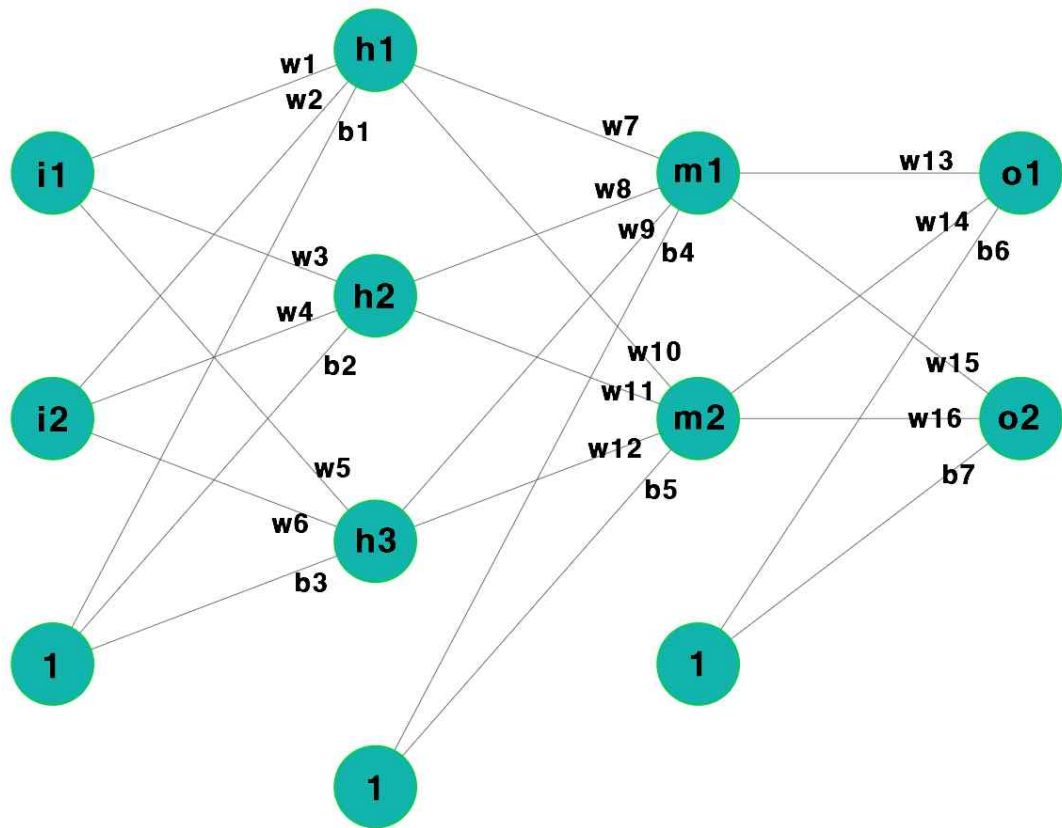
epoch = 664
h1, h2 = 0.239, 0.226
o1, o2 = 0.010, 0.990
E = 0.0000001
w1b, w3b = -0.000, 0.000
w2b, w4b = -0.000, 0.000
b1b, b2b = -0.000, 0.000
w5b, w7b = 0.000, -0.000
w6b, w8b = 0.000, -0.000
b3b, b4b = 0.000, -0.000
w1, w3 = 0.143, 0.242
w2, w4 = 0.186, 0.284
b1, b2 = 0.213, 0.186
w5, w7 = 0.203, 0.533
w6, w8 = 0.253, 0.583
b3, b4 = -0.095, 0.730
epoch = 665
h1, h2 = 0.239, 0.226
o1, o2 = 0.010, 0.990
E = 0.0000001

```

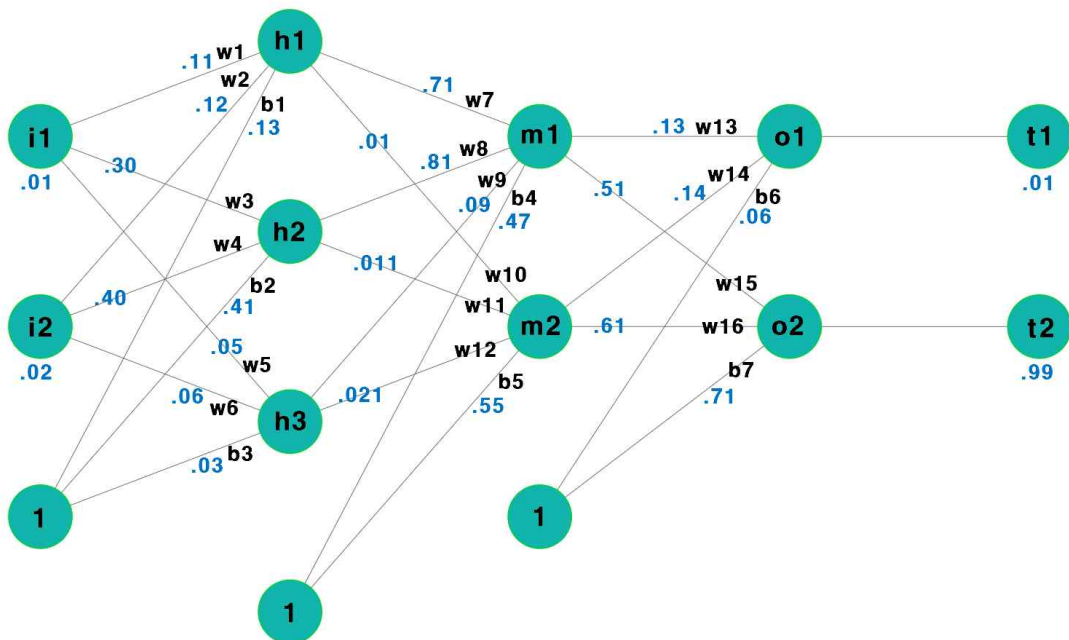
연습문제

2입력 2은닉 3은닉 2출력

1. 다음은 입력2 은닉3 은닉2 출력3의 심층 인공 신경망입니다. 이 신경망에는 2개의 은닉 층이 포함되어 있습니다. 일반적으로 은닉 층이 2층 이상일 경우 심층 인공 신경망이라고 합니다. 이 신경망의 순전파, 역전파 행렬 계산식을 구합니다.

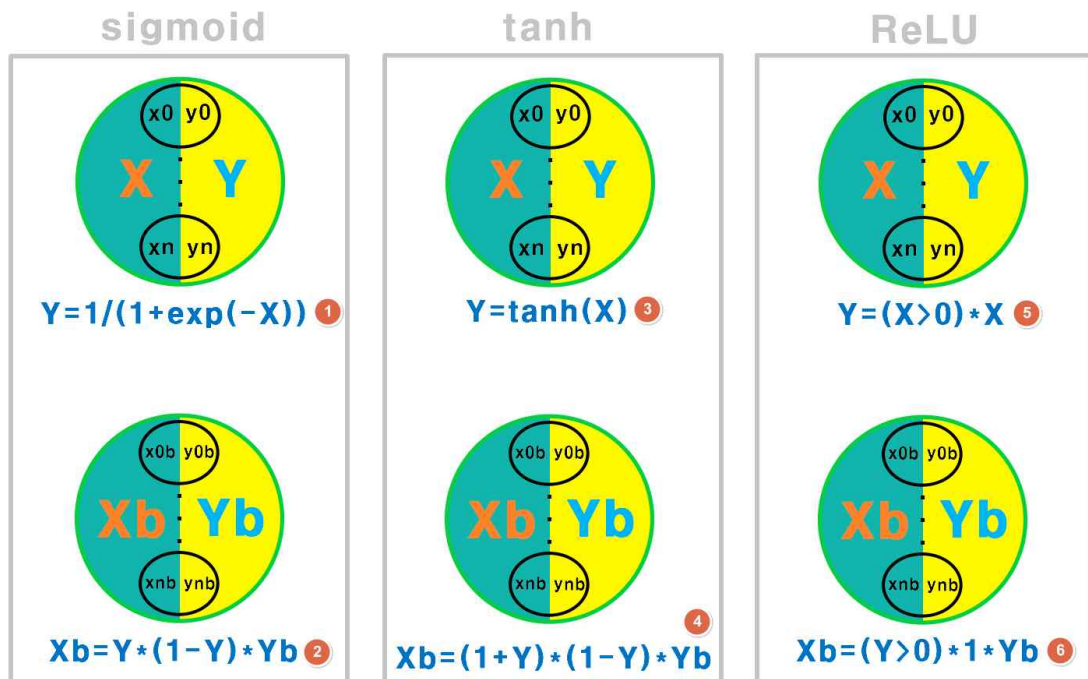


2. 앞에서 구한 행렬 계산식을 이용하여 다음과 같이 초기화된 인공 신경망을 구현하고 학습시켜 봅니다. 입력 값 i_1, i_2 는 상수로 처리합니다.



07 활성화 함수 적용하기

여기서는 sigmoid, tanh, ReLU 활성화 함수의 순전파와 역전파 수식을 살펴보고, 앞에서 NumPy를 이용해 구현한 인공 신경망에 활성화 함수를 적용하여 봅니다. 다음 그림은 활성화 함수의 순전파와 역전파 NumPy 수식을 나타냅니다.



이 그림에서 X, Y는 각각 $x_0 \sim x_n$, $y_0 \sim y_n$ (n 은 0보다 큰 정수)의 집합을 나타냅니다. 예를 들어, x_0 , y_0 는 하나의 노드 내에서 활성화 함수의 입력과 출력을 의미합니다. X, Y는 하나의 층 내에서 활성화 함수의 입력과 출력 행렬을 의미합니다.

이상에서 필요한 행렬 계산식을 정리하면 다음과 같습니다.

시그모이드 순전파와 역전파

$$Y = \frac{1}{1 + e^{-X}} \quad (1) \quad X_b = Y(1 - Y) Y_b \quad (2)$$

tanh 순전파와 역전파

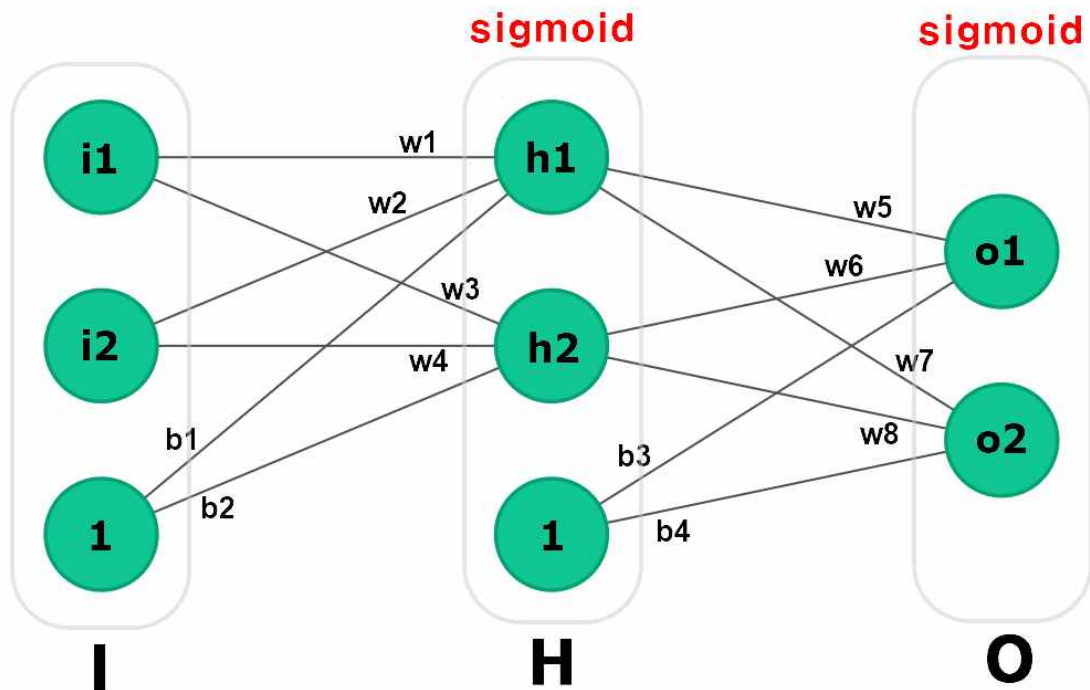
$$Y = \tanh(X) \quad (3) \quad X_b = (1 + Y)(1 - Y) Y_b \quad (4)$$

ReLU 순전파와 역전파

$$Y = (X > 0) X \quad (5) \quad X_b = (Y > 0) 1 Y_b \quad (6)$$

sigmoid 함수 적용해 보기

지금까지 정리한 행렬 계산식을 구현을 통해 살펴봅니다. 다음 그림을 살펴봅니다.



1. 이전 예제를 복사합니다.
2. 다음과 같이 예제를 수정합니다.
417_1.py

```
01 from ulab import numpy as np
02
03 I = np.array([[.05, .10]])
04 T = np.array([[.01, .99]])
05 WH = np.array([[.15, .25],
06               [.20, .30]])
07 BH = np.array([[.35, .35]])
08 WO = np.array([[.40, .50],
09               [.45, .55]])
10 BO = np.array([[.60, .60]])
11
12 for epoch in range(1000):
13
14     print('epoch = %d' %epoch)
15
16     H = np.dot(I, WH) + BH
```



```

17 H = 1/(1+np.exp(-H)) # ❶
18
19 O = np.dot(H, WO) + BO
20 O = 1/(1+np.exp(-O)) # ❶
21
22 print(' O =\n', O)
23
24 E = np.sum((O - T) ** 2 / 2)
25 if E < 0.0000001:
26     break
27
28 Ob = O - T
29 Ob = Ob*O*(1-O) # ❷
30
31 Hb = np.dot(Ob, WO.T)
32 Hb = Hb*H*(1-H) # ❷
33
34 WHb = np.dot(I.T, Hb)
35 BHb = 1 * Hb
36 WOb = np.dot(H.T, Ob)
37 BOb = 1 * Ob
38
39 lr = 0.01
40 WH = WH - lr * WHb
41 BH = BH - lr * BHb
42 WO = WO - lr * WOb
43 BO = BO - lr * BOb


```

17 : 은닉 층 H에 순전파 시그모이드 활성화 함수를 적용합니다.

20 : 출력 층 O에 순전파 시그모이드 활성화 함수를 적용합니다.

29 : 역 출력 층 Ob에 역전파 시그모이드 활성화 함수를 적용합니다.

32 : 역 은닉 층 Hb에 역전파 시그모이드 활성화 함수를 적용합니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```


epoch = 998
O =
array([[0.3059052, 0.8440722]], dtype=float32)
epoch = 999
O =
array([[0.3056745, 0.8441187]], dtype=float32)

```

(999+1)번째에 o1, o2가 각각 0.306, 0.844가 됩니다.

4. 다음과 같이 예제를 수정합니다.

```
14 for epoch in range(10000):
```

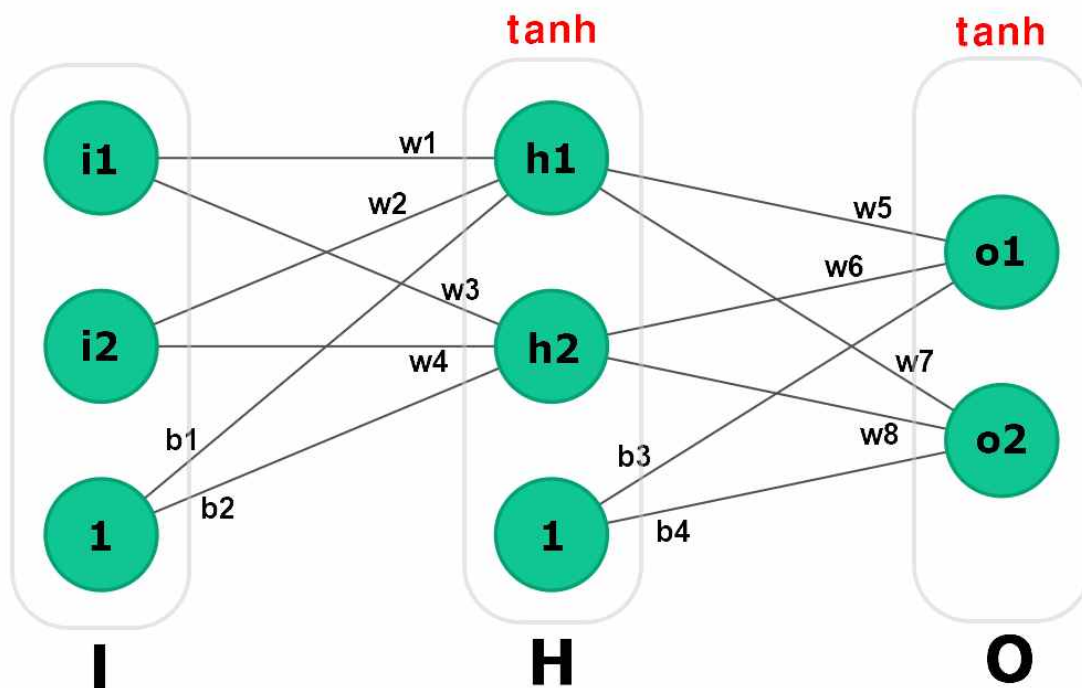
5.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 9998
O =
array([[0.0625474, 0.9427186]], dtype=float32)
epoch = 9999
O =
array([[0.06254376, 0.9427216]], dtype=float32)
```

(9999+1)번째에 o1, o2가 각각 0.063, 0.943이 됩니다.

tanh 함수 적용해 보기

이번에는 이전 예제에 적용했던 sigmoid 함수를 tanh 함수로 변경해 봅니다. 다음 그림을 살펴봅니다.



1. 이전 예제를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

417_2.py

```
16 H = np.dot(I, WH) + BH
17 H = np.tanh(H) # ③
```

```

18
19     O = np.dot(H, WO) + BO
20     O = np.tanh(O) # ③

```

17 : 은닉 층 H에 순전파 tanh 활성화 함수를 적용합니다.

20 : 출력 층 O에 순전파 tanh 활성화 함수를 적용합니다.


```

28     Ob = O - T
29     Ob = Ob*(1+O)*(1-O) # ④
30
31     Hb = np.dot(Ob, WO.T)
32     Hb = Hb*(1+H)*(1-H) # ④

```

29 : 역 출력 층 Ob에 역전파 tanh 활성화 함수를 적용합니다.

32 : 역 은닉 층 Hb에 역전파 tanh 활성화 함수를 적용합니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

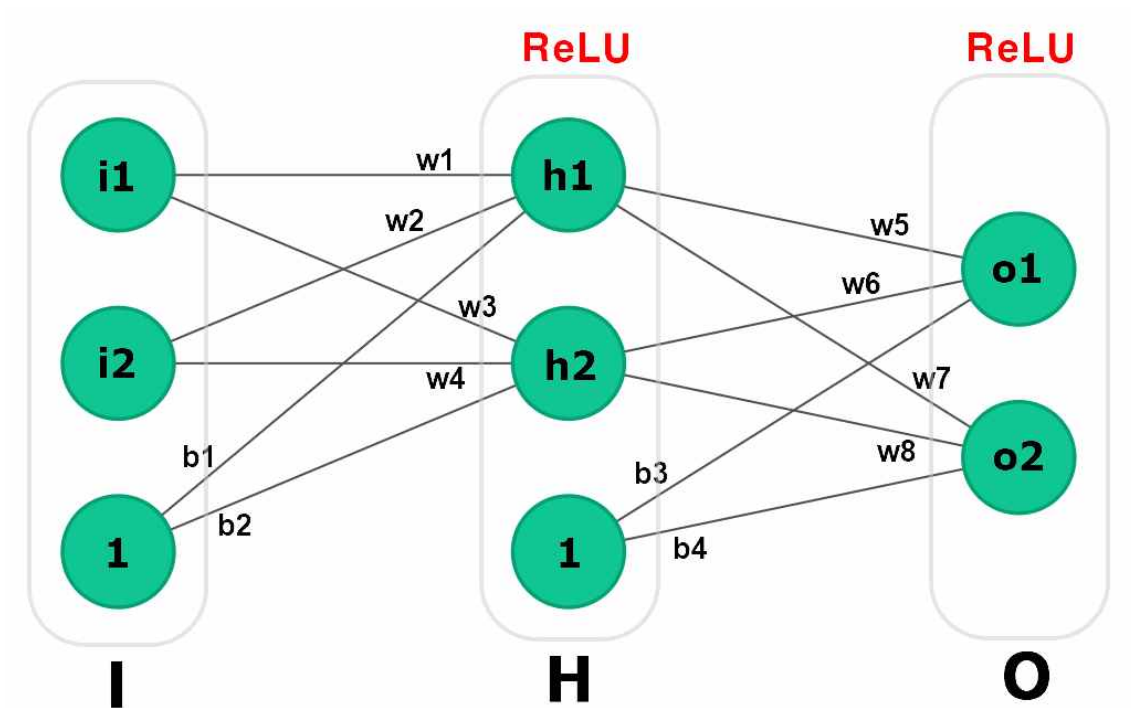
epoch = 9998
O =
array([[0.01010731, 0.9711369]], dtype=float32)
epoch = 9999
O =
array([[0.01010725, 0.9711382]], dtype=float32)

```

(9999+1)번째에 o1, o2가 각각 0.010, 0.971이 됩니다.

ReLU 함수 적용해 보기

이번에는 이전 예제에 적용했던 tanh 함수를 ReLU 함수로 변경해 봅니다. 다음 그림을 살펴 봅니다.



1. 이전 예제를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

417_3.py

```

16     H = np.dot(I, WH) + BH
17     H = (H>0)*H # ③
18
19     O = np.dot(H, WO) + BO
20     O = (O>0)*O # ③

```

17 : 은닉 층 H에 순전파 ReLU 활성화 함수를 적용합니다.

20 : 출력 층 O에 순전파 ReLU 활성화 함수를 적용합니다.


```

28     Ob = O - T
29     Ob = Ob*(O>0)*1 # ④
30
31     Hb = np.dot(Ob, WO.T)
32     Hb = Hb*(H>0)*1 # ④

```

29 : 역 출력 층 Ob에 역전파 ReLU 활성화 함수를 적용합니다.

32 : 역 은닉 층 Hb에 역전파 ReLU 활성화 함수를 적용합니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 664
O =
array([[0.01041593, 0.9898296]], dtype=float32)
epoch = 665
O =
array([[0.01041131, 0.9898315]], dtype=float32)
```

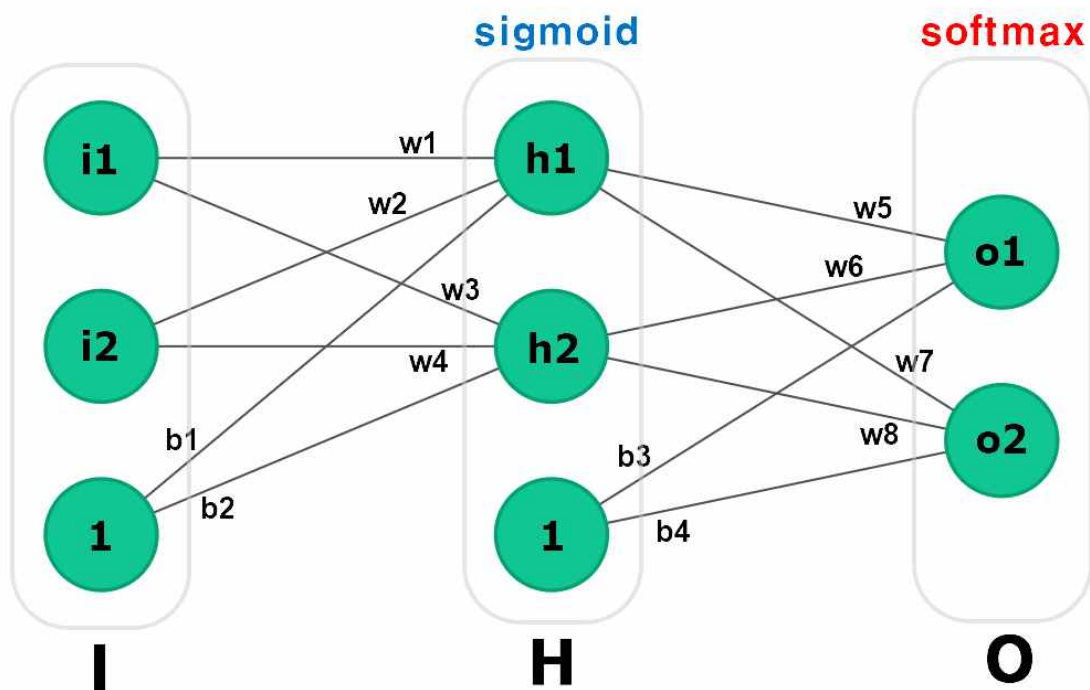
(665+1)번째에 오차가 0.0000001(천만분의 1)보다 작아집니다. o1, o2는 각각 0.010, 0.990이 된 상태입니다. sigmoid, tanh 함수보다 결과가 훨씬 더 빨리 나오는 것을 볼 수 있습니다.

08 출력 층에 softmax 함수 적용해 보기

여기서는 출력 층에 소프트맥스 함수를 적용해 봅니다.

sigmoid와 softmax

먼저 은닉 층은 sigmoid, 출력 층은 softmax 활성화 함수를 적용해 봅니다. 다음 그림을 살펴봅니다.



1. 이전 예제를 복사합니다.
2. 다음과 같이 예제를 수정합니다.
418_1.py

```

01 from ulab import numpy as np
02
03 I = np.array([[.05, .10]])
04 T = np.array([[ 0,  1]])
05 WH = np.array([[.15, .25],
06                [.20, .30]])
07 BH = np.array([[.35, .35]])
08 WO = np.array([[.40, .50],
09                [.45, .55]])
10 BO = np.array([[.60, .60]])
11
12 for epoch in range(10000):
13
14     print('epoch = %d' %epoch)
15
16     H = np.dot(I, WH) + BH
17     H = 1/(1+np.exp(-H))
18
19     O = np.dot(H, WO) + BO
20     OM = O - np.max(O)
21     O = np.exp(OM)/np.sum(np.exp(OM))
22
23     print(' O  =\n', O)
24
25     E = np.sum(-T*np.log(O))
26     if E < 0.0001:
27         break
28
29     Ob = O - T
30     # nothing for softmax + cross entropy error
31
32     Hb = np.dot(Ob, WO.T)
33     Hb = Hb*H*(1-H)
34
35     WHb = np.dot(I.T, Hb)
36     BHb = 1 * Hb
37     WOb = np.dot(H.T, Ob)
38     BOb = 1 * Ob
39
40     lr = 0.01

```

```

41 WH = WH - lr * WHb
42 BH = BH - lr * BHb
43 WO = WO - lr * WOb
44 BO = BO - lr * BOb

```

04 : 목표 값을 각각 0과 1로 변경합니다.

20, 21 : 출력 층의 활성화 함수를 소프트맥스로 변경합니다.

20 : O의 각 항목에서 O의 가장 큰 항목 값을 빼줍니다. 이렇게 하면 23 줄에서 오버플로우를 막을 수 있습니다. O에 대한 최종 결과는 같습니다. 자세한 내용은 [소프트맥스 오버플로우]를 검색해 봅니다.

25 : 오차 계산을 크로스 엔트로피 오차 형태의 수식으로 변경합니다. 소프트맥스 활성화 함수는 크로스 엔트로피 오차와 같이 사용합니다.


$$E = - \sum_k t_k \log o_k$$

26 : for 문을 빠져 나가는 오차 값을 0.0001로 변경합니다. 여기서 사용하는 값의 크기에 따라 학습의 정확도와 학습 시간이 결정됩니다.

29 : 소프트맥스 함수의 역전파 오차 계산 부분은 다음과 같습니다. 소프트맥스 함수는 크로스 엔트로피 함수와 같이 사용될 때 역전파 시 소프트맥스 함수를 역으로 거쳐 전파되는 오차가 다음과 같이 예측 값과 목표 값의 차가 됩니다.

$$o_{kb} = o_k - t_k$$

그래서 일반적으로 소프트맥스 함수를 활성화 함수로 사용할 경우 오차 함수는 크로스 엔트로피 오차 함수가 됩니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

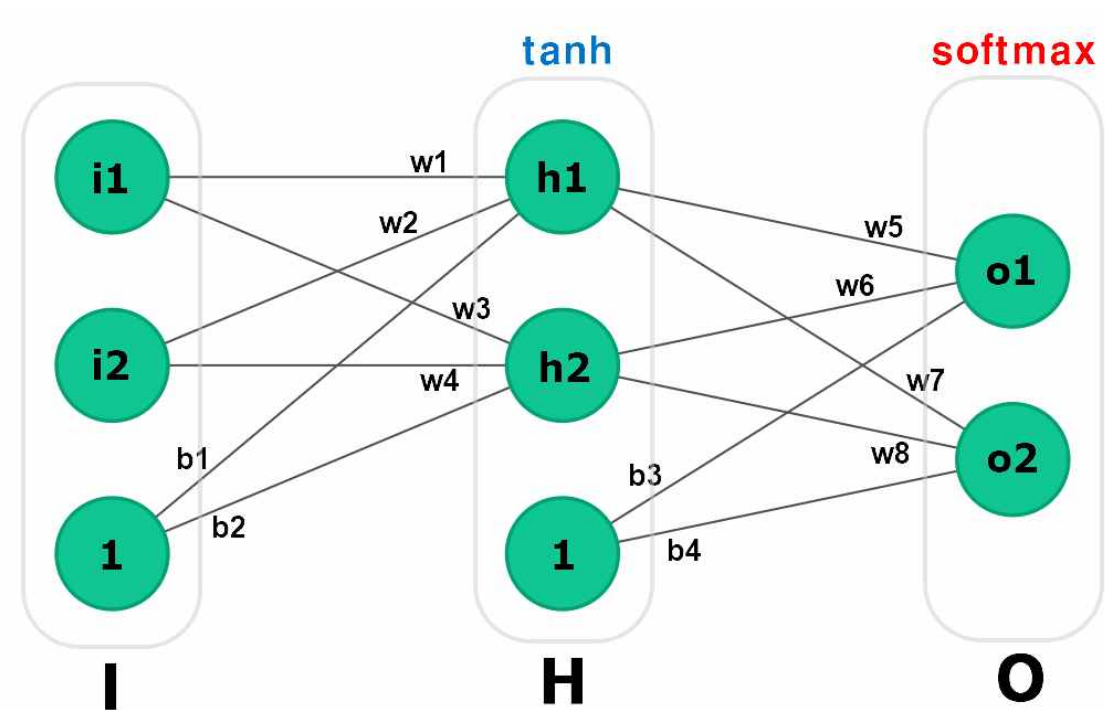
epoch = 9998
O =
  array([[0.002483801, 0.9975162]], dtype=float32)
epoch = 9999
O =
  array([[0.002483538, 0.9975165]], dtype=float32)

```

(9999+1)번째에 o1, o2가 각각 0.002, 0.998이 됩니다.

tanh와 softmax

여기서는 은닉 층 활성화 함수를 tanh로 변경해 봅니다. 다음 그림을 살펴봅니다.



1. 이전 예제를 복사합니다.
2. 다음과 같이 예제를 수정합니다.
418_2.py

```

16     H = np.dot(I, WH) + BH
17     H = np.tanh(H)
18
19     O = np.dot(H, WO) + BO
20     OM = O - np.max(O)
21     O = np.exp(OM)/np.sum(np.exp(OM))

```


- 17 : 은닉 층 H에 순전파 tanh 활성화 함수를 적용합니다.
20, 21 : 출력 층의 활성화 함수는 softmax입니다.

```

29     Ob = O - T
30     # nothing for softmax + cross entropy error
31
32     Hb = np.dot(Ob, WO.T)
33     Hb = Hb*(1+H)*(1-H)

```

- 29 : softmax 함수의 역전파 오차 계산 부분입니다.
33 : 역 은닉 층 Hb에 역전파 tanh 활성화 함수를 적용합니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.


```

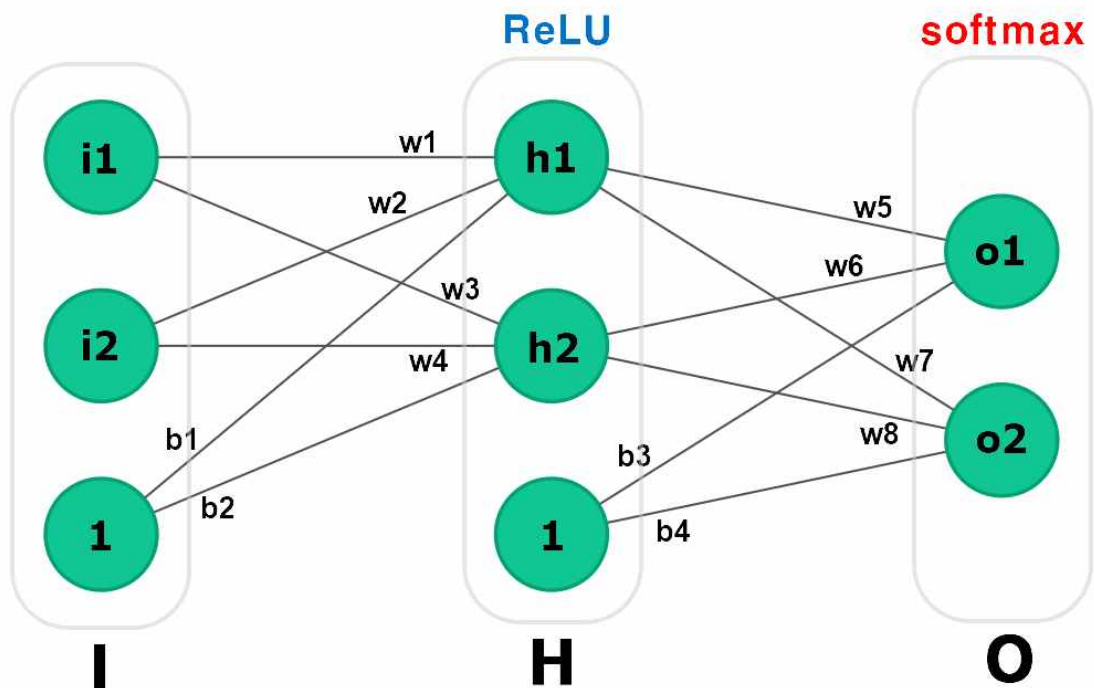
epoch = 9998
O =
array([[0.001968064, 0.9980319]], dtype=float32)
epoch = 9999
O =
array([[0.001967852, 0.9980322]], dtype=float32)

```

(9999+1)번째에 o1, o2가 각각 0.002, 0.998이 됩니다.

ReLU와 softmax

여기서는 은닉 층 활성화 함수를 ReLU로 변경해 봅니다. 다음 그림을 살펴봅니다.



1. 이전 예제를 복사합니다.
 2. 다음과 같이 예제를 수정합니다.
- 418_3.py

```

16 H = np.dot(I, WH) + BH
17 H = (H>0)*H
18
19 O = np.dot(H, WO) + BO
20 OM = O - np.max(O)
21 O = np.exp(OM)/np.sum(np.exp(OM))

```


17 : 은닉 층 H에 순전파 ReLU 활성화 함수를 적용합니다.

20, 21 : 출력 층의 활성화 함수는 softmax입니다.

```
29 Ob = O - T
30 # nothing for softmax + cross entropy error
31
32 Hb = np.dot(Ob, WO.T)
33 Hb = Hb*(H>0)*1
```

29 : softmax 함수의 역전파 오차 계산 부분입니다.

33 : 역 은닉 층 Hb에 역전파 ReLU 활성화 함수를 적용합니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

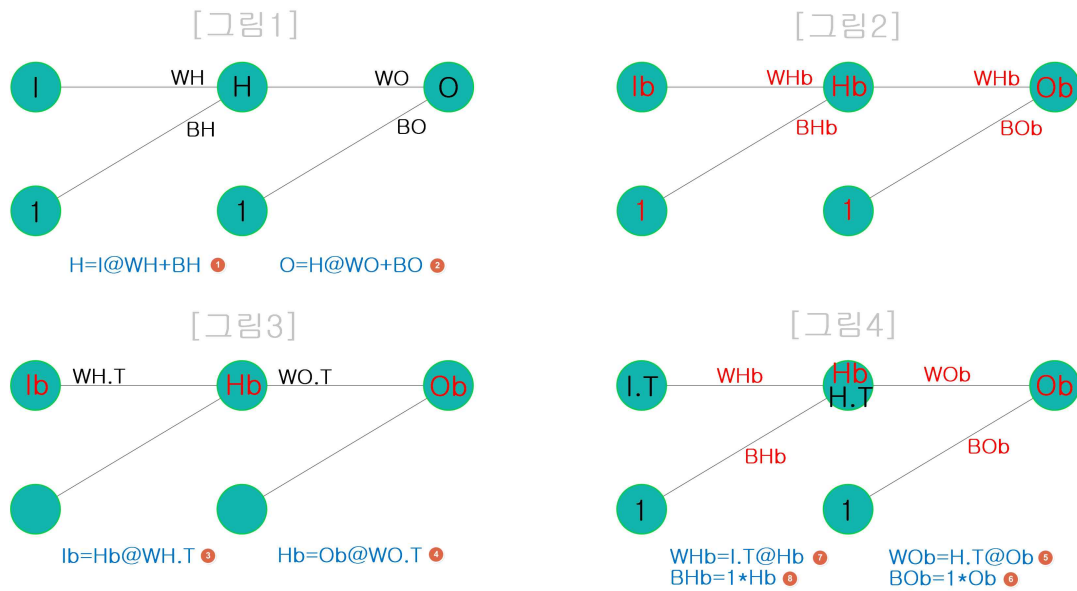
```
epoch = 9998
O =
array([[0.0008200751, 0.9991799]], dtype=float32)
epoch = 9999
O =
array([[0.0008199724, 0.99918]], dtype=float32)
```

(9999+1)번째에 o1, o2가 각각 0.010, 0.971이 됩니다.

이상에서 NumPy의 행렬 계산식을 이용하여 출력 층의 활성화 함수는 소프트맥스, 오차 계산 함수는 크로스 엔트로피 오차 함수인 인공 신경망을 구현해 보았습니다.

09 인공 신경망 행렬 계산식

여기서는 인공 신경망의 순전파 역전파를 행렬 계산식으로 정리해 봅니다. 인공 신경망을 행렬 계산식으로 정리하면 인공 신경망의 크기, 깊이와 상관없이 간결하게 정리할 수 있습니다. 다음 그림은 입력 층, 은닉 층, 출력 층으로 구성된 인공 신경을 나타냅니다.



[그림1]은 순전파 과정에 필요한 행렬과 행렬 계산식을 나타냅니다.

[그림2]는 역전파에 필요한 행렬입니다. 순전파에 대응되는 행렬이 모두 필요합니다.

[그림3]은 입력의 역전파에 필요한 행렬과 행렬 계산식을 나타냅니다.

[그림4]는 가중치와 편향의 역전파에 필요한 행렬과 행렬 계산식을 나타냅니다.

*** ❸ Ib는 I 층이 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 Hb처럼 해당 인공 신경으로 역전파되는 행렬 값입니다. 여기서 I는 은닉 층에 연결된 입력 층이므로 Ib의 수식은 필요치 않습니다.

*** @ 문자는 행렬 곱을 의미합니다.

이상에서 필요한 행렬 계산식을 정리하면 다음과 같습니다.

순전파	
$H = I@WH + BH$	❶
$O = H@WO + BO$	❷
역전파 오차	
$Ob = O - T$	
입력 역전파	
$Hb = Ob@WO.T$	❹
가중치, 편향 역전파	

$$WHb = I.T@Hb \quad 7$$

$$BHb = 1 * Hb \quad 8$$

$$WOb = H.T@Ob \quad 5$$

$$BOb = 1 * Ob \quad 6$$

가중치, 편향 학습

$$WH = WH - lr * WHb$$

$$BH = BH - lr * BHb$$

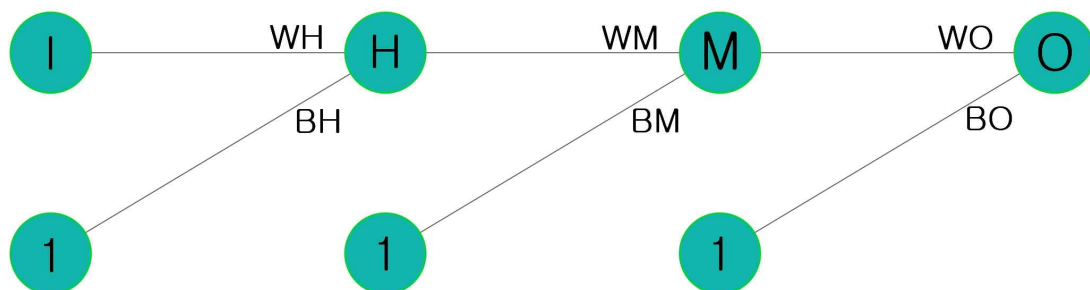
$$WO = WO - lr * WOb$$

$$BO = BO - lr * BOb$$

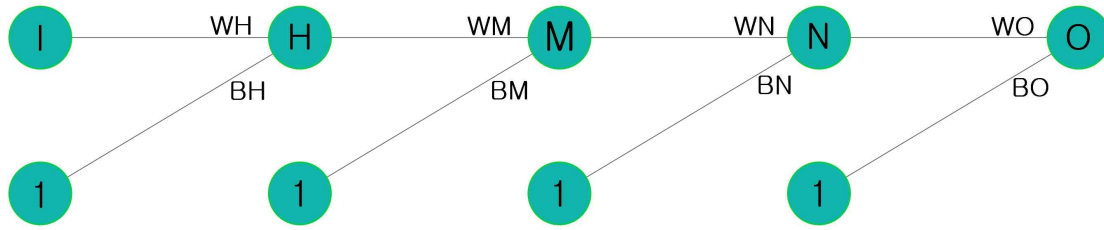
*** lr은 학습률을 나타냅니다.

연습문제

1. 다음은 입력I 은닉H 은닉M 출력O의 심층 인공 신경망입니다. 이 신경망에는 2개의 은닉 층이 포함되어 있습니다. 일반적으로 은닉 층이 2층 이상일 경우 심층 인공 신경망이라고 합니다. 이 신경망의 입력 역전파 그래프와 가중치, 편향 역전파 그래프를 그리고 순전파, 역전파 행렬 계산식을 구합니다.



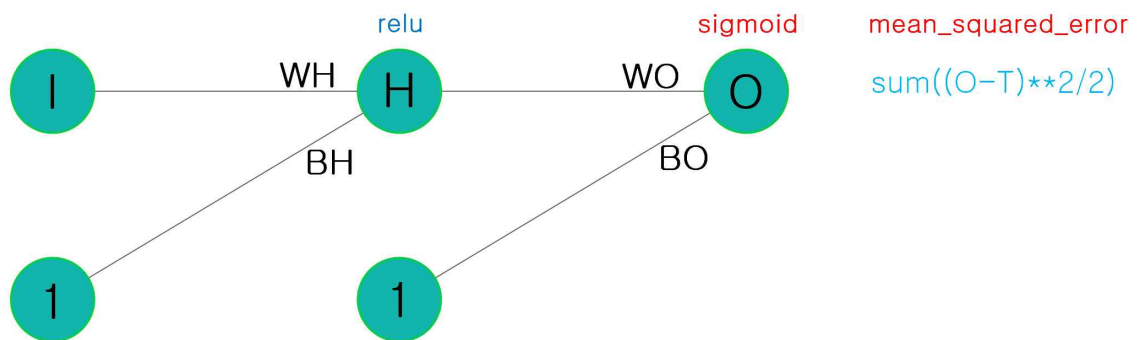
2. 다음은 입력I 은닉H 은닉M 은닉N 출력O의 심층 인공 신경망입니다. 이 신경망에는 3개의 은닉 층이 포함되어 있습니다. 일반적으로 은닉 층이 2층 이상일 경우 심층 인공 신경망이라고 합니다. 이 신경망의 입력 역전파 그래프와 가중치, 편향 역전파 그래프를 그리고 순전파, 역전파 행렬 계산식을 구합니다.



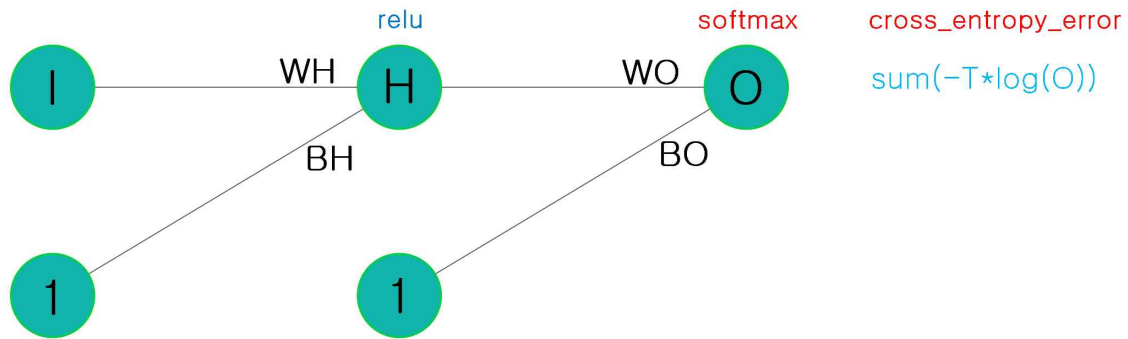
10 가중치 초기화하기

여기서는 활성화 함수에 따라 은닉 층과 출력 층의 가중치를 초기화하는 방법에 대해 살펴보고 해당 방법을 적용하여 은닉 층과 출력 층의 가중치를 초기화한 후 학습을 시켜봅니다. 미리 말씀드리면 활성화 함수에 따른 가중치와 편향의 적절한 초기화는 인공 신경망 학습에 아주 중요한 부분입니다. 우리는 앞으로 수행할 예제에서 은닉 층의 활성화 함수로 ReLU를 사용하고 출력 층의 활성화 함수로 sigmoid나 softmax를 사용합니다. 출력 층의 활성화 함수를 sigmoid로 사용할 경우 오차 계산 함수는 평균 제곱 오차 함수를 사용하고, 출력 층의 활성화 함수를 softmax로 사용할 경우 오차 계산 함수는 크로스 엔트로피 오차 함수를 사용합니다. 다음 그림을 참조합니다.

ReLU-sigmoid-mse 신경망



ReLU-softmax-cee 신경망



ReLU와 He 초기화

ReLU 활성화 함수를 사용할 경우엔 Kaming He가 2010년에 발표한 He 초기화 방법을 사용합니다. 수식은 다음과 같습니다.

$$normal(mean=0, stddev), stddev = \sqrt{\frac{2}{input}}$$

여기서 normal은 종모양의 정규 분포를 의미하며, mean은 평균값, stddev는 표준편차로 종모양이 퍼진 정도를 의미합니다. 이 수식을 적용하면 0에 가까운 값이 많도록 가중치가 초기화됩니다.

sigmoid, softmax와 Lecun 초기화

sigmoid와 softmax 활성화 함수를 사용할 경우엔 Yann Lecun 교수가 1998년에 발표한 Lecun 초기화 방법을 사용합니다. 수식은 다음과 같습니다.

$$normal(mean=0, stddev), stddev = \sqrt{\frac{1}{input}}$$

여기서 normal은 종모양의 표준 정규 분포를 의미하며, mean은 평균값, stddev는 표준편차로 종모양이 퍼진 정도를 의미합니다. 이 수식을 적용하면 0에 가까운 값이 많도록 가중치가 초기화됩니다.

He와 Lecun 가중치 초기화하기

이제 He와 Lecun으로 가중치를 초기화하여 학습시켜 봅니다.

1. 다음과 같이 예제를 작성합니다.

4110_1.py

```
01 from ulab import numpy as np
02 import urandom
03 import time
04 from math import sqrt, log
05
06 def randn() :
07
08     while True :
09         u = urandom.random() * 2 - 1 # -1.0 ~ 1.0 까지의 값
10         v = urandom.random() * 2 - 1 # -1.0 ~ 1.0 까지의 값
11
12         r = u * u + v * v;
13
14         if not(r == 0 or r >= 1) :
15             break
16
17     c = sqrt( (-2 * log(r)) / r )
18
19     return u * c
20
21 NUM_I = 2
22 NUM_H = 2
23 NUM_O = 2
24
25 I = np.array([[.05, .10]])
26 T = np.array([[.01, .99]])
27 WH = np.zeros((NUM_I, NUM_H))
28 BH = np.zeros((1, NUM_H))
29 WO = np.zeros((NUM_H, NUM_O))
30 BO = np.zeros((1, NUM_O))
31
32 urandom.seed(time.time())
33
34 for m in range(NUM_I):
35     for n in range(NUM_H):
36         WH[m, n] = randn()/sqrt(NUM_I/2) # He
37
38 for m in range(NUM_H):
```

```

39     for n in range(NUM_O):
40         WO[m, n] = randn()/sqrt(NUM_H) # Lecun
41
42     print("WH =\n", WH)
43     print("WO =\n", WO)
44     print()
45
46     for epoch in range(1, 100001):
47
48         H = np.dot(I, WH) + BH
49         H = (H>0)*H # ReLU
50
51         O = np.dot(H, WO) + BO
52         O = 1/(1+np.exp(-O)) #sigmoid
53
54         E = np.sum((O-T)**2/2) #mean squared error
55
56         if epoch==1 :
57             print("epoch  = %d" %epoch)
58             print("Error  = %.4f" %E)
59             print("output =", O)
60             print()
61
62         if E<0.001 or epoch == 100000:
63             print("epoch  = %d" %epoch)
64             print("Error  = %.4f" %E)
65             print("output =", O)
66             break
67
68         Ob = O - T
69         Ob = Ob*O*(1-O) #sigmoid
70
71         Hb = np.dot(Ob, WO.T)
72         Hb = Hb*(H>0)*1 # ReLU
73
74         WHb = np.dot(I.T, Hb)
75         BHb = 1 * Hb
76         WOb = np.dot(H.T, Ob)
77         BOb = 1 * Ob
78

```



```

79     lr = 0.01
80     WH = WH - lr * WHb
81     BH = BH - lr * BHb
82     WO = WO - lr * WOb
83     BO = BO - lr * BOb

```

01 : ulab 하위 모듈인 numpy 모듈을 np라는 이름으로 불러옵니다.

02 : urandom 모듈을 불러옵니다.

03 : time 모듈을 불러옵니다.

04 : math 모듈로부터 sqrt, log 함수를 불러옵니다.

06~19 : randn 함수를 정의합니다. randn 함수는 표준 정규 분포에 따른 난수를 생성하는 함수입니다. randn 함수에 대한 설명은 따로 하지 않습니다.

21 : NUM_I 변수를 선언한 후, 2로 초기화합니다. NUM_I 변수는 입력층 노드의 개수를 저장합니다.

22 : NUM_H 변수를 선언한 후, 2로 초기화합니다. NUM_H 변수는 은닉층 노드의 개수를 저장합니다.

23 : NUM_O 변수를 선언한 후, 2로 초기화합니다. NUM_O 변수는 출력층 노드의 개수를 저장합니다.

27 : 초기 값 0을 갖는 NUM_I x NUM_H 행렬을 생성한 후, 가중치 변수 WH에 할당합니다.

28 : 초기 값 0을 갖는 1 x NUM_H 행렬을 생성한 후, 편향 변수 BH에 할당합니다. 일반적으로 편향의 초기 값은 0으로 시작합니다.

29 : 초기 값 0을 갖는 NUM_H x NUM_O 행렬을 생성한 후, 가중치 변수 WO에 할당합니다.

30 : 초기 값 0을 갖는 1 x NUM_O 행렬을 생성한 후, 편향 변수 BO에 할당합니다. 일반적으로 편향의 초기 값은 0으로 시작합니다.

32 : urandom.seed 함수를 호출하여 난수 생성기를 초기화합니다. time.time 함수는 초단위의 현재 시간을 내어줍니다.

34~36 : WH 행렬의 각 항목에 대해 He 초기화를 수행합니다.

38~40 : WO 행렬의 각 항목에 대해 Lecun 초기화를 수행합니다.

42, 43 : print 함수를 호출하여 WH, WO 값을 출력해 봅니다.

46 : epoch 변수 1에서 100001(십만일) 미만에 대하여 48~83줄을 수행합니다.

49 : 은닉 층의 활성화 함수를 ReLU로 사용합니다.

52 : 출력 층의 활성화 함수를 sigmoid로 사용합니다.


54 : 오차 계산 함수는 평균 제곱 오차를 사용합니다.

56~60 : epoch값이 1일 때, 즉, 처음 시작할 때, 오차 값과 예측 값을 출력합니다.

62~66 : 오차 값이 0.001(천분의 일)보다 작을 때, 오차 값과 예측 값을 출력한 후, 46줄의 for문을 나옵니다.

69 : 출력 층의 역 활성화 함수를 sigmoid로 사용합니다.

72 : 은닉 층의 역 활성화 함수를 ReLU로 사용합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

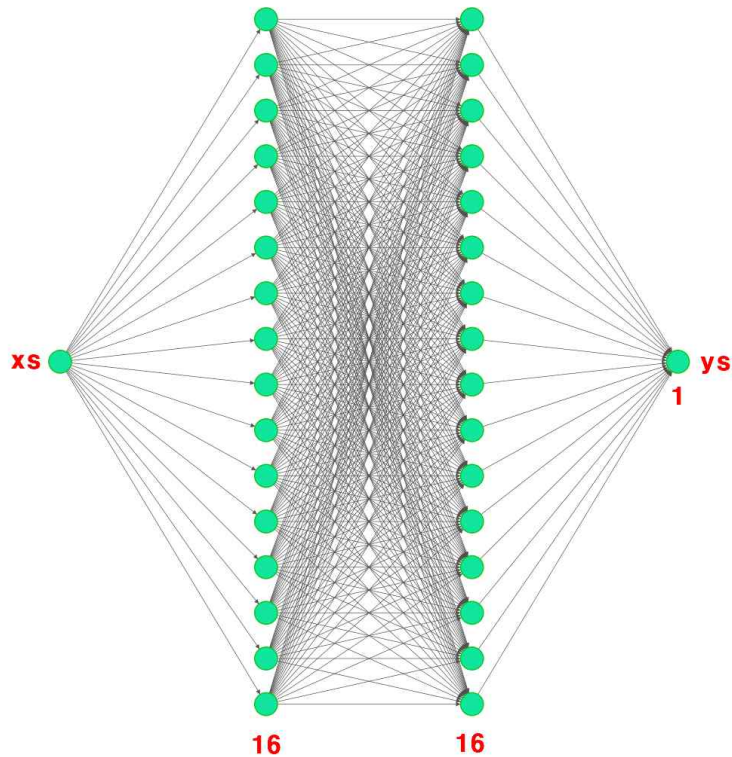
```
WH =  
    array([[ -0.5129153, -0.1467497],  
          [ 0.01298038, -0.4916947]], dtype=float32)  
WO =  
    array([[ -0.9509138,  1.293943],  
          [ 0.6510501, -0.2956532]], dtype=float32)  
  
epoch  = 1  
Error  = 0.2401  
output = array([[0.5, 0.5]], dtype=float32)  
  
epoch  = 40533  
Error  = 0.0010  
output = array([[0.04162265, 0.9583774]], dtype=float32)
```

필자의 경우 40533번 학습을 수행하였으며, 오차는 0.0010이고, 첫 번째 항목의 값은 0.04, 두 번째 항목은 0.96입니다. 가중치 초기 값에 따라 독자 여러분의 결과는 다를 수 있습니다.

02 NumPy DNN 활용하기

여기서는 지금까지 구현한 인공 신경망 라이브러리를 활용해 인공 신경망을 확장해 봅니다.

인공 신경망 라이브러리를 이용하면, 인공 신경망을 좀 더 자유롭게 구성하고 테스트해 볼 수 있습니다. 예를 들어, 다음과 같은 형태의 인공 신경망을 구성해서 테스트해 볼 수 있습니다.



01 7 세그먼트 입력 2 진수 출력 인공 신경망

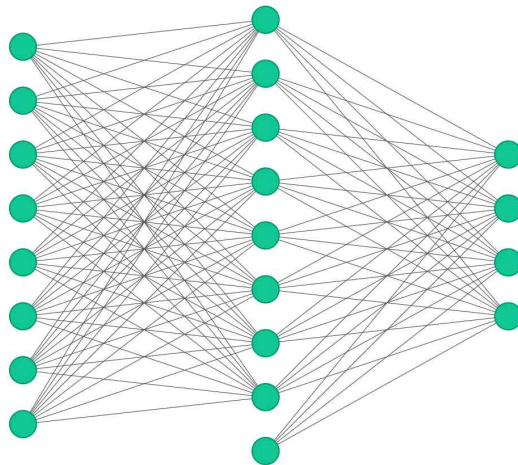
여기서는 7 세그먼트에 숫자 값에 따라 표시되는 LED의 ON, OFF 값을 입력으로 받아 2 진수로 출력하는 인공 신경망을 구성하고 학습시켜 봅니다. 다음은 7 세그먼트 디스플레이 2진수 연결 진리표입니다.

7 세그먼트 디스플레이
2 진수 연결 진리표

In	In	In	In	In	In	In	Out	Out	Out	Out
1	1	1	1	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
0	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	0	0	1	1	1	0	0	1

5 = 1011011 → 0101

그림에서 7 세그먼트에 5로 표시되기 위해 7개의 LED가 1011011(1-ON, 0-OFF)의 비트열에 맞춰 켜지거나 꺼져야 합니다. 해당 비트열에 대응하는 이진수는 0101입니다. 여기서는 다음 그림과 같이 7개의 입력, 8개의 은닉 층, 4개의 출력 층으로 구성된 인공 신경망을 학습시켜 봅니다.



1. 다음은 앞에서 라즈베리파이 피코에 저장한 파일입니다. 파일을 확인합니다.

myrandn.py

```

01 import urandom
02 from math import sqrt, log
03
04 def randn() :
05
06     while True :
07         u = urandom.random() * 2 - 1 # -1.0 ~ 1.0 까지의 값
08         v = urandom.random() * 2 - 1 # -1.0 ~ 1.0 까지의 값

```

```

09
10         r = u * u + v * v;
11
12         if not(r == 0 or r >= 1) :
13             break
14
15     c = sqrt( (-2 * log(r)) / r )
16
17     return u * c

```

2. 4110_1.py 예제를 421_1.py로 저장합니다.

3. 다음과 같이 예제를 수정합니다.

421_1.py

```

01 from ulab import numpy as np
02 import urandom
03 import time
04 from math import sqrt
05 from myrandn import *
06
07 NUM_PATTERN = 10
08 NUM_I = 7
09 NUM_H = 8
10 NUM_O = 4
11
12 I = [
13     np.array([[ 1, 1, 1, 1, 1, 1, 0 ]]), # 0
14     np.array([[ 0, 1, 1, 0, 0, 0, 0 ]]), # 1
15     np.array([[ 1, 1, 0, 1, 1, 0, 1 ]]), # 2
16     np.array([[ 1, 1, 1, 1, 0, 0, 1 ]]), # 3
17     np.array([[ 0, 1, 1, 0, 0, 1, 1 ]]), # 4
18     np.array([[ 1, 0, 1, 1, 0, 1, 1 ]]), # 5
19     np.array([[ 0, 0, 1, 1, 1, 1, 1 ]]), # 6
20     np.array([[ 1, 1, 1, 0, 0, 0, 0 ]]), # 7
21     np.array([[ 1, 1, 1, 1, 1, 1, 1 ]]), # 8
22     np.array([[ 1, 1, 1, 0, 0, 1, 1 ]]) # 9
23 ]
24 T = [
25     np.array([[ 0, 0, 0, 0 ]]),
26     np.array([[ 0, 0, 0, 1 ]]),

```

```

27     np.array([[ 0, 0, 1, 0 ]]),
28     np.array([[ 0, 0, 1, 1 ]]),
29     np.array([[ 0, 1, 0, 0 ]]),
30     np.array([[ 0, 1, 0, 1 ]]),
31     np.array([[ 0, 1, 1, 0 ]]),
32     np.array([[ 0, 1, 1, 1 ]]),
33     np.array([[ 1, 0, 0, 0 ]]),
34     np.array([[ 1, 0, 0, 1 ]])
35 ]
36 O = [np.zeros((1, NUM_O)) for no in range(NUM_PATTERN)]
37 WH = np.zeros((NUM_I, NUM_H))
38 BH = np.zeros((1, NUM_H))
39 WO = np.zeros((NUM_H, NUM_O))
40 BO = np.zeros((1, NUM_O))
41
42 urandom.seed(time.time())
43
44 for m in range(NUM_I):
45     for n in range(NUM_H):
46         WH[m, n] = randn()/sqrt(NUM_I/2) # He
47
48 for m in range(NUM_H):
49     for n in range(NUM_O):
50         WO[m, n] = randn()/sqrt(NUM_H) # Lecun
51
52 for epoch in range(1, 100001):
53
54     H = np.dot(I[2], WH) + BH
55     H = (H>0)*H # ReLU
56
57     O[2] = np.dot(H, WO) + BO
58     O[2] = 1/(1+np.exp(-O[2])) #sigmoid
59
60     E = np.sum((O[2]-T[2])**2/2) #mean squared error
61
62     if epoch==1 :
63         print("epoch  = %d" %epoch)
64         print("Error  = %.4f" %E)
65         print("output =", O[2])
66         print()

```

```

67
68     if E<0.001 or epoch == 100000:
69         print("epoch = %d" %epoch)
70         print("Error = %.4f" %E)
71         print("output =", O[2])
72         break
73
74     Ob = O[2] - T[2]
75     Ob = Ob*O[2]*(1-O[2]) #sigmoid
76
77     Hb = np.dot(Ob, WO.T)
78     Hb = Hb*(H>0)*1 # ReLU
79
80     WHb = np.dot(I[2].T, Hb)
81     BHb = 1 * Hb
82     WOb = np.dot(H.T, Ob)
83     BOb = 1 * Ob
84
85     lr = 0.01
86     WH = WH - lr * WHb
87     BH = BH - lr * BHb
88     WO = WO - lr * WOb
89     BO = BO - lr * BOb

```

05 : myrandn 모듈의 함수를 불러옵니다.

07 : NUM_PATTERN 변수를 선언한 후, 10으로 초기화합니다. NUM_PATTERN 변수는 다음 진리표의 가로줄의 개수입니다.

7 세그먼트 디스플레이 2 진수 연결 진리표

In	In	In	In	In	In	In	Out	Out	Out	Out
1	1	1	1	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
0	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	0	0	1	1	1	0	0	1



= 1011011 → 0101

08 : NUM_I 변수를 선언한 후, 7로 초기화합니다.

09 : NUM_H 변수를 선언한 후, 8로 초기화합니다.

10 : NUM_O 변수를 선언한 후, 4로 초기화합니다.

12~23 : (1 x 7) 크기의 2차 행렬을 항목으로 갖는 리스트 I를 선언한 후, 진리표의 입력 값에 맞게 2차 행렬 값을 초기화합니다. ulab의 numpy는 3차 이상의 행렬을 지원하지 않습니다. 그래서 2차 행렬을 항목으로 갖는 리스트를 이용합니다.

24~35 : (1 x 4) 크기의 2차 행렬을 항목으로 갖는 리스트 T를 선언한 후, 진리표의 입력 값에 맞게 2차 행렬 값을 초기화합니다. ulab의 numpy는 3차 이상의 행렬을 지원하지 않습니다. 그래서 2차 행렬을 항목으로 갖는 리스트를 이용합니다.


36 : (1 x NUM_O) 크기의 0으로 초기화된 2차 행렬을 항목으로 갖는 리스트 O를 선언합니다. 항목의 개수는 NUM_PATTERN입니다.

37~40 : 가중치와 편향 행렬의 모양을 위 그림에 맞게 변경합니다.

54, 80 : I을 I[2]로 변경합니다. I 행렬의 2번 항목을 입력 값으로 학습 테스트를 수행합니다.

60, 74 : T을 T[2]로 변경합니다. T 행렬의 2번 항목을 목표 값으로 학습 테스트를 수행합니다.

57, 58, 60, 65, 71, 74, 75 : O을 O[2]로 변경합니다. O 행렬의 2번 항목을 예측 값으로 학습 테스트를 수행합니다. 58, 75 줄의 경우 2 군데씩 수정합니다.

4.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.


```
epoch = 1
Error = 0.6434
output = array([[0.6071887, 0.5036945, 0.4328424, 0.5854818]], dtype=float32)

epoch = 4253
Error = 0.0010
output = array([[0.02248295, 0.02353702, 0.9773762, 0.02070251]], dtype=float32)
```

필자의 경우 4253번 학습을 수행하였으며, 오차는 0.0010이고, 첫 번째, 두 번째, 네 번째 항목의 값은 0에 가깝고, 세 번째 항목의 값은 1에 가깝습니다. 다음 그림에서 진리표의 2번 항목에 맞게 학습된 것을 볼 수 있습니다.

7 세그먼트 디스플레이
2 진수 연결 진리표

In	In	In	In	In	In	In	Out	Out	Out	Out
1	1	1	1	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
0	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	0	0	1	1	1	0	0	1


 = 1011011 → 0101

02 7 세그먼트 입력 2 진수 출력 인공 신경망 2

계속해서 7 세그먼트에 숫자 값에 따라 표시되는 LED의 ON, OFF 값을 입력으로 받아 2 진수로 출력하는 인공 신경망을 구성하고 학습시켜 봅니다. 여기서는 다음 진리표의 전체 입력 값에 대해 목표 값에 대응되도록 학습을 시켜봅니다.

7 세그먼트 디스플레이
2 진수 연결 진리표

In	In	In	In	In	In	In	Out	Out	Out	Out
1	1	1	1	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
0	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	0	0	1	1	1	0	0	1

 = 1011011 → 0101

0. 먼저 421_1.py 예제를 422_1.py로 저장합니다.

1. 다음과 같이 예제를 작성합니다.

422_1.py

```
01 from ulab import numpy as np
02 import urandom
03 import time
04 from math import sqrt
05 from myrandn import *
06
07 NUM_PATTERN = 10
08 NUM_I = 7
09 NUM_H = 8
10 NUM_O = 4
11
12 I = [
13     np.array([[ 1, 1, 1, 1, 1, 1, 0 ]]), # 0
14     np.array([[ 0, 1, 1, 0, 0, 0, 0 ]]), # 1
15     np.array([[ 1, 1, 0, 1, 1, 0, 1 ]]), # 2
16     np.array([[ 1, 1, 1, 1, 0, 0, 1 ]]), # 3
17     np.array([[ 0, 1, 1, 0, 0, 1, 1 ]]), # 4
```

```

18     np.array([[ 1, 0, 1, 1, 0, 1, 1 ]]), # 5
19     np.array([[ 0, 0, 1, 1, 1, 1, 1 ]]), # 6
20     np.array([[ 1, 1, 1, 0, 0, 0, 0 ]]), # 7
21     np.array([[ 1, 1, 1, 1, 1, 1, 1 ]]), # 8
22     np.array([[ 1, 1, 1, 0, 0, 1, 1 ]]) # 9
23 ]
24 T = [
25     np.array([[ 0, 0, 0, 0 ]]),
26     np.array([[ 0, 0, 0, 1 ]]),
27     np.array([[ 0, 0, 1, 0 ]]),
28     np.array([[ 0, 0, 1, 1 ]]),
29     np.array([[ 0, 1, 0, 0 ]]),
30     np.array([[ 0, 1, 0, 1 ]]),
31     np.array([[ 0, 1, 1, 0 ]]),
32     np.array([[ 0, 1, 1, 1 ]]),
33     np.array([[ 1, 0, 0, 0 ]]),
34     np.array([[ 1, 0, 0, 1 ]])
35 ]
36 O = [np.zeros((1, NUM_O)) for no in range(NUM_PATTERN)]
37 WH = np.zeros((NUM_I, NUM_H))
38 BH = np.zeros((1, NUM_H))
39 WO = np.zeros((NUM_H, NUM_O))
40 BO = np.zeros((1, NUM_O))
41
42 urandom.seed(time.time())
43
44 for m in range(NUM_I):
45     for n in range(NUM_H):
46         WH[m, n] = randn()/sqrt(NUM_I/2) # He
47
48 for m in range(NUM_H):
49     for n in range(NUM_O):
50         WO[m, n] = randn()/sqrt(NUM_H) # Lecun
51
52 for epoch in range(1, 10001):
53
54     for pc in range(NUM_PATTERN) :
55
56         H = np.dot(I[pc], WH) + BH
57         H = (H>0)*H # ReLU

```

```

58
59         O[pc] = np.dot(H, WO) + BO
60         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
61
62         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
63
64         Ob = O[pc] - T[pc]
65         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
66
67         Hb = np.dot(Ob, WO.T)
68         Hb = Hb*(H>0)*1 # ReLU
69
70         WHb = np.dot(I[pc].T, Hb)
71         BHb = 1 * Hb
72         WOb = np.dot(H.T, Ob)
73         BOb = 1 * Ob
74
75         lr = 0.01
76         WH = WH - lr * WHb
77         BH = BH - lr * BHb
78         WO = WO - lr * WOb
79         BO = BO - lr * BOb
80
81     if epoch%100==0 :
82         print("epoch : %5d" %(epoch))
83
84     print()
85
86     for pc in range(NUM_PATTERN) :
87         print("target %d : "%pc, end='')
88         for node in range(NUM_O) :
89             print("%.0f "%T[pc][0][node], end='')
90         print("pattern %d : "%pc, end='');
91         for node in range(NUM_O) :
92             print("%.2f "%O[pc][0][node], end='')
93         print()

```

52 : epoch 변수를 1000001(백만일) 미만에서 10001(일만일) 미만으로 변경합니다.

54 : pc 변수 0에서 NUM_PATTERN 미만에 대하여 56~79줄을 수행합니다.


56, 59, 60, 62, 64, 65, 70 : 숫자 2를 pc로 변경합니다. 60, 62, 64, 65줄은 두 군데 변경합니다.

62~64 : if 조건문 2개를 없앱니다.

81 : epoch값이 100의 배수가 될 때마다 현재 학습 회수를 출력합니다.

84 : 개 행 문자를 출력합니다.

86~93 : 학습이 끝난 후에 목표 값과 예측 값을 출력하여 비교합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch : 10000

target 0 : 0 0 0 0 pattern 0 : 0.03 0.02 0.02 0.03
target 1 : 0 0 0 1 pattern 1 : 0.01 0.07 0.06 0.96
target 2 : 0 0 1 0 pattern 2 : 0.03 0.00 0.99 0.00
target 3 : 0 0 1 1 pattern 3 : 0.00 0.03 0.99 1.00
target 4 : 0 1 0 0 pattern 4 : 0.03 0.94 0.00 0.02
target 5 : 0 1 0 1 pattern 5 : 0.03 0.97 0.03 1.00
target 6 : 0 1 1 0 pattern 6 : 0.00 0.98 0.97 0.00
target 7 : 0 1 1 1 pattern 7 : 0.00 0.93 0.94 1.00
target 8 : 1 0 0 0 pattern 8 : 0.95 0.00 0.00 0.01
target 9 : 1 0 0 1 pattern 9 : 0.97 0.05 0.00 1.00
```

학습을 1만 번 수행한 후에 목표 값의 0과 1에 예측 값이 가까운 값을 갖는지 확인합니다.

03 입력 데이터 임의로 섞기

여기서는 매 회기마다 입력 데이터를 임의로 섞어 인공 신경망을 학습 시켜봅니다. 입력 데이터를 임의로 섞으면 인공 신경망 학습에 도움이 됩니다.

0. 먼저 422_1.py 예제를 423_1.py로 저장합니다.

1. 다음과 같이 파일을 수정합니다.

423_1.py

```
001 from ulab import numpy as np
002 import urandom
003 import time
004 from math import sqrt
005 from myrandn import *
006
007 NUM_PATTERN = 10
008 NUM_I = 7
009 NUM_H = 8
010 NUM_O = 4
011
012 I = [
013     np.array([[ 1, 1, 1, 1, 1, 1, 0 ]]), # 0
```

```

014     np.array([[ 0, 1, 1, 0, 0, 0, 0 ]]), # 1
015     np.array([[ 1, 1, 0, 1, 1, 0, 1 ]]), # 2
016     np.array([[ 1, 1, 1, 1, 0, 0, 1 ]]), # 3
017     np.array([[ 0, 1, 1, 0, 0, 1, 1 ]]), # 4
018     np.array([[ 1, 0, 1, 1, 0, 1, 1 ]]), # 5
019     np.array([[ 0, 0, 1, 1, 1, 1, 1 ]]), # 6
020     np.array([[ 1, 1, 1, 0, 0, 0, 0 ]]), # 7
021     np.array([[ 1, 1, 1, 1, 1, 1, 1 ]]), # 8
022     np.array([[ 1, 1, 1, 0, 0, 1, 1 ]]) # 9
023 ]
024 T = [
025     np.array([[ 0, 0, 0, 0 ]]),
026     np.array([[ 0, 0, 0, 1 ]]),
027     np.array([[ 0, 0, 1, 0 ]]),
028     np.array([[ 0, 0, 1, 1 ]]),
029     np.array([[ 0, 1, 0, 0 ]]),
030     np.array([[ 0, 1, 0, 1 ]]),
031     np.array([[ 0, 1, 1, 0 ]]),
032     np.array([[ 0, 1, 1, 1 ]]),
033     np.array([[ 1, 0, 0, 0 ]]),
034     np.array([[ 1, 0, 0, 1 ]])
035 ]
036 O = [np.zeros((1, NUM_O)) for no in range(NUM_PATTERN)]
037 WH = np.zeros((NUM_I, NUM_H))
038 BH = np.zeros((1, NUM_H))
039 WO = np.zeros((NUM_H, NUM_O))
040 BO = np.zeros((1, NUM_O))
041
042 shuffled_pattern = [pc for pc in range(NUM_PATTERN)] #정수로!
043
044 urandom.seed(time.time())
045
046 for m in range(NUM_I):
047     for n in range(NUM_H):
048         WH[m, n] = randn()/sqrt(NUM_I/2) # He
049
050 for m in range(NUM_H):
051     for n in range(NUM_O):
052         WO[m, n] = randn()/sqrt(NUM_H) # Lecun
053

```

```

054 for epoch in range(1, 10001):
055
056     tmp_a = 0;
057     tmp_b = 0;
058     for pc in range(NUM_PATTERN) :
059         tmp_a = urandom.randrange(0, NUM_PATTERN)
060         tmp_b = shuffled_pattern[pc]
061         shuffled_pattern[pc] = shuffled_pattern[tmp_a]
062         shuffled_pattern[tmp_a] = tmp_b
063
064     sumError = 0.
065
066     for rc in range(NUM_PATTERN) :
067
068         pc = shuffled_pattern[rc]
069
070         H = np.dot(I[pc], WH) + BH
071         H = (H>0)*H # ReLU
072
073         O[pc] = np.dot(H, WO) + BO
074         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
075
076         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
077
078         sumError += E
079
080         Ob = O[pc] - T[pc]
081         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
082
083         Hb = np.dot(Ob, WO.T)
084         Hb = Hb*(H>0)*1 # ReLU
085
086         WHb = np.dot(I[pc].T, Hb)
087         BHb = 1 * Hb
088         WOb = np.dot(H.T, Ob)
089         BOb = 1 * Ob
090
091         lr = 0.01
092         WH = WH - lr * WHb
093         BH = BH - lr * BHb

```

```

094         WO = WO - lr * WOb
095         BO = BO - lr * BOb
096
097     if epoch%100==0 :
098         print("epoch : %5d, sum error : %f" %(epoch, sumError))
099         for i in range(NUM_I) :
100             for j in range(NUM_H) :
101                 print("%7.3f "%WH[i][j], end='')
102             print()
103
104     if sumError<0.0001 : break
105
106 print()
107
108 for pc in range(NUM_PATTERN) :
109     print("target %d : "%pc, end='')
110     for node in range(NUM_O) :
111         print("%.0f "%T[pc][0][node], end='')
112     print("pattern %d : "%pc, end='');
113     for node in range(NUM_O) :
114         print("%.2f "%O[pc][0][node], end='')
115     print()

```

042 : NUM_PATTERN 개수의 정수 배열 shuffled_pattern을 선언하고, 각 항목을 순서대로 초기화해 줍니다.

056, 057 : 입력 데이터의 순서를 변경하기 위해 사용할 정수 변수 2개를 선언합니다.

058 : pc 변수에 대해 0에서 NUM_PATTERN 미만에 대하여 059~062줄을 수행합니다.

059 : urandom.randrange 함수를 호출하여 0에서 NUM_PATTERN 미만 사이 값을 생성하여 tmp_a 변수에 할당합니다. 이 예제에서는 0에서 10 미만의 값이 생성됩니다.

060~062 : shuffled_pattern의 tmp_a 번째 항목과 pc 번째 항목을 서로 바꿔줍니다.

064 : sumError 변수를 선언한 후, 0.0으로 초기화해줍니다.

066 : 이전 예제에서 pc를 rc로 변경해 줍니다.

068 : shuffled_pattern의 rc 번째 항목을 pc로 가져옵니다.


070~076 : 이전 예제와 같습니다.

078 : 076줄에서 얻은 오차 값을 sumError에 더해줍니다.

99~102 : 현재까지 학습된 가중치 WH 값을 출력해 봅니다.

104 : sumError 값이 0.0001보다 작으면 066줄의 for 문을 빠져 나와 106줄로 이동합니다.

108~115 : 이전 예제와 같습니다. 목표 값과 예측 값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

epoch : 10000, sum error : 0.013842
-2.244   3.470  -0.302   1.025   3.256  -0.266  -0.402  -0.977
 1.102  -1.198   0.703   1.403   1.638  -0.757   2.133  -0.252
 0.417   0.956  -0.647  -0.339   1.375  -0.397  -0.573   1.137
 0.135   0.696  -0.280   0.250  -0.216  -0.294   1.790  -0.991
 0.138  -1.072   0.341  -1.226  -0.619  -0.059   1.115  -1.291
-1.149  -1.935  -0.421  -2.193   0.143  -0.456   0.614   1.299
-1.095  -2.089  -0.548   0.507  -0.002   0.664  -0.168   0.428

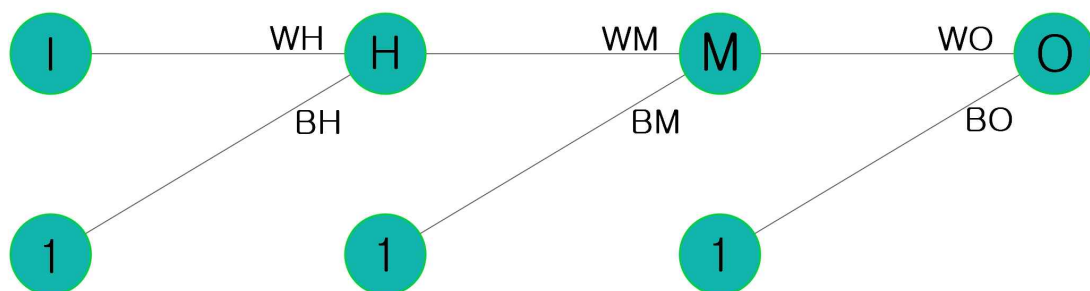
target 0 : 0 0 0 0 pattern 0 : 0.05 0.04 0.04 0.02
target 1 : 0 0 0 1 pattern 1 : 0.01 0.03 0.03 0.97
target 2 : 0 0 1 0 pattern 2 : 0.02 0.00 0.99 0.02
target 3 : 0 0 1 1 pattern 3 : 0.00 0.03 0.99 1.00
target 4 : 0 1 0 0 pattern 4 : 0.03 0.98 0.01 0.03
target 5 : 0 1 0 1 pattern 5 : 0.02 0.98 0.02 0.97
target 6 : 0 1 1 0 pattern 6 : 0.00 0.95 0.94 0.00
target 7 : 0 1 1 1 pattern 7 : 0.00 0.96 0.96 1.00
target 8 : 1 0 0 0 pattern 8 : 0.95 0.00 0.00 0.01
target 9 : 1 0 0 1 pattern 9 : 0.97 0.02 0.00 0.99

```

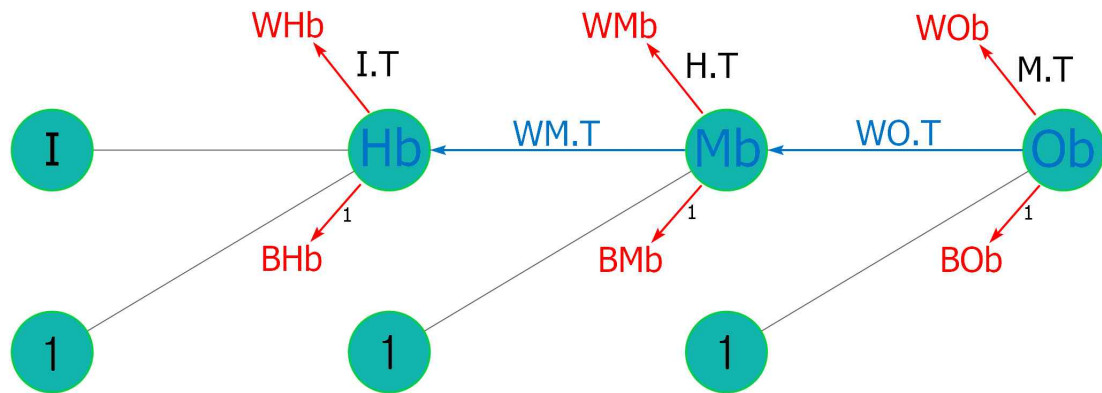
학습이 진행됨에 따라 가중치 값이 갱신되는 것을 볼 수 있습니다. 학습이 끝나기 전 마지막 1회 가중치 갱신 결과를 볼 수 있으며, 마지막에는 학습된 결과의 예측 값을 목표 값과 비교하여 보여줍니다. 예측 값이 목표 값이 적당히 가까운 것을 볼 수 있습니다. 예측 값을 목표 값에 더 가깝게 하려면 훈련의 횟수를 늘리면 됩니다.

04 은닉층 추가하기

여기서는 은닉층을 하나 더 추가해 봅니다. 일반적으로 은닉층의 개수가 2개 이상일 때 심층 인공 신경망이라고 합니다. 다음은 은닉층 M이 추가된 I-H-M-O 심층 인공 신경망입니다. 이 신경망은 순전파 과정을 나타냅니다.



다음은 역전파 과정을 나타냅니다.



0. 먼저 423_1.py 예제를 424_1.py로 저장합니다.

1. 다음과 같이 파일을 수정합니다.

424_1.py

```
001 from ulab import numpy as np
002 import urandom
003 import time
004 from math import sqrt
005 from myrandn import *
006
007 NUM_PATTERN = 10
008 NUM_I = 7
009 NUM_H = 16
010 NUM_M = 16
011 NUM_O = 4
012
013 I = [
014     np.array([[ 1, 1, 1, 1, 1, 1, 0 ]]), # 0
015     np.array([[ 0, 1, 1, 0, 0, 0, 0 ]]), # 1
016     np.array([[ 1, 1, 0, 1, 1, 0, 1 ]]), # 2
017     np.array([[ 1, 1, 1, 1, 0, 0, 1 ]]), # 3
018     np.array([[ 0, 1, 1, 0, 0, 1, 1 ]]), # 4
019     np.array([[ 1, 0, 1, 1, 0, 1, 1 ]]), # 5
020     np.array([[ 0, 0, 1, 1, 1, 1, 1 ]]), # 6
021     np.array([[ 1, 1, 1, 0, 0, 0, 0 ]]), # 7
022     np.array([[ 1, 1, 1, 1, 1, 1, 1 ]]), # 8
023     np.array([[ 1, 1, 1, 0, 0, 1, 1 ]]), # 9
024 ]
025 T = [
```

```

026     np.array([[ 0, 0, 0, 0 ]]),
027     np.array([[ 0, 0, 0, 1 ]]),
028     np.array([[ 0, 0, 1, 0 ]]),
029     np.array([[ 0, 0, 1, 1 ]]),
030     np.array([[ 0, 1, 0, 0 ]]),
031     np.array([[ 0, 1, 0, 1 ]]),
032     np.array([[ 0, 1, 1, 0 ]]),
033     np.array([[ 0, 1, 1, 1 ]]),
034     np.array([[ 1, 0, 0, 0 ]]),
035     np.array([[ 1, 0, 0, 1 ]])
036 ]
037 O = [np.zeros((1, NUM_O)) for no in range(NUM_PATTERN)]
038
039
040 WH = np.zeros((NUM_I, NUM_H))
041 BH = np.zeros((1, NUM_H))
042 WM = np.zeros((NUM_H, NUM_M))
043 BM = np.zeros((1, NUM_M))
044 WO = np.zeros((NUM_M, NUM_O))
045 BO = np.zeros((1, NUM_O))
046
047 shuffled_pattern = [pc for pc in range(NUM_PATTERN)] #정수로!
048
049 urandom.seed(time.time())
050
051 for m in range(NUM_I):
052     for n in range(NUM_H):
053         WH[m, n] = randn()/sqrt(NUM_I/2) # He
054
055 for m in range(NUM_H):
056     for n in range(NUM_M):
057         WM[m, n] = randn()/sqrt(NUM_H/2) # He
058
059 for m in range(NUM_M):
060     for n in range(NUM_O):
061         WO[m, n] = randn()/sqrt(NUM_M) # Lecun
062
063 begin = time.ticks_ms()
064 t_prev = time.ticks_ms()
065

```

```

066 for epoch in range(1, 10001):
067
068     tmp_a = 0;
069     tmp_b = 0;
070     for pc in range(NUM_PATTERN) :
071         tmp_a = urandom.randrange(0, NUM_PATTERN)
072         tmp_b = shuffled_pattern[pc]
073         shuffled_pattern[pc] = shuffled_pattern[tmp_a]
074         shuffled_pattern[tmp_a] = tmp_b
075
076     sumError = 0.
077
078     for rc in range(NUM_PATTERN) :
079
080         pc = shuffled_pattern[rc]
081
082         H = np.dot(I[pc], WH) + BH
083         H = (H>0)*H # ReLU
084
085         M = np.dot(H, WM) + BM
086         M = (M>0)*M # ReLU
087
088         O[pc] = np.dot(M, WO) + BO
089         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
090
091         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
092
093         sumError += E
094
095         Ob = O[pc] - T[pc]
096         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
097
098         Mb = np.dot(Ob, WO.T)
099         Mb = Mb*(M>0)*1 # ReLU
100
101         Hb = np.dot(Mb, WM.T)
102         Hb = Hb*(H>0)*1 # ReLU
103
104         WHb = np.dot(I[pc].T, Hb)
105         BHb = 1 * Hb

```

```

106      WMb = np.dot(H.T, Mb)
107      BMb = 1 * Mb
108      WOb = np.dot(M.T, Ob)
109      BOb = 1 * Ob
110
111      lr = 0.01
112      WH = WH - lr * WHb
113      BH = BH - lr * BHb
114      WM = WM - lr * WMb
115      BM = BM - lr * BMb
116      WO = WO - lr * WOb
117      BO = BO - lr * BOb
118
119      if epoch%100==0 :
120          t_now = time.ticks_ms()
121          time_taken = t_now - t_prev
122          t_prev = t_now
123          print("epoch : %5d, sum error : %f" %(epoch, sumError), end='')
124          print(", %.3f sec" %(time_taken/1000))
125
126          if sumError<0.0001 : break
127
128      print()
129
130      for pc in range(NUM_PATTERN) :
131          print("target %d : "%pc, end='')
132          for node in range(NUM_O) :
133              print("%.0f "%T[pc][0][node], end='')
134          print("pattern %d : "%pc, end='');
135          for node in range(NUM_O) :
136              print("%.2f "%O[pc][0][node], end='')
137          print()
138
139      end = time.ticks_ms()
140      time_taken = end - begin
141      print("\nTime taken (in seconds) = {}".format(time_taken/1000))

```

009 : NUM_H 값을 16으로 변경합니다.

010 : NUM_M 변수를 선언한 후, 16으로 초기화합니다. NUM_M 변수는 2차 은닉층 노드의 개수를 저장합니다.

042 : 초기 값 0을 갖는 NUM_H x NUM_M 행렬을 생성한 후, 가중치 변수 WM에 할당합니다.

043 : 초기 값 0을 갖는 1 x NUM_M 행렬을 생성한 후, 편향 변수 BM에 할당합니다. 일반적으로 편향의 초기 값은 0으로 시작합니다.

044 : NUM_H를 NUM_M으로 변경합니다.

055~057 : WM 행렬의 각 항목에 대해 He 초기화를 수행합니다.

059, 061 : NUM_H를 NUM_M으로 변경합니다.

063 : begin 변수를 선언하고, time.tick_ms 함수를 호출하여 밀리 초 단위의 현재 시간으로 초기화합니다. begin은 학습을 시작한 최초 시간을 나타냅니다. begin 변수는 140줄에서 사용되어 전체 학습 시간을 측정합니다.

064 : t_prev 변수를 선언하고, time.tick_ms 함수를 호출하여 밀리 초 단위의 현재 시간으로 초기화합니다. t_prev 변수는 121,122줄에서 사용되어 학습을 100회 수행할 때마다의 학습 시간을 측정합니다.

085,086 : 은닉 층 M의 순전파 과정을 추가합니다.

088 : H를 M으로 바꿔줍니다.

098,099 : 은닉 층 M의 입력 역전파 과정을 추가합니다.

101 : Ob를 Mb로 WO.T를 WM.T로 바꿔줍니다.

106,107 : 은닉 층 M의 가중치, 편향 역전파 과정을 추가합니다.

108 : H.T를 M.T로 바꿔줍니다.

114, 115 : 은닉 층 M의 가중치, 편향 갱신 과정을 추가합니다.

119 : epoch 값이 100의 배수이면 120~126줄을 수행합니다.

120 : t_now 변수를 선언하고, time.tick_ms 함수를 호출하여 밀리 초 단위의 현재 시간을 저장합니다.

121 : 학습 100회 수행에 대해 현재 측정 시간에서 이전 측정 시간을 빼서 time_taken 변수에 저장합니다. time_taken 변수는 학습을 100회 수행할 때마다의 학습 시간을 저장합니다.

122 : t_prev 변수값을 t_now 변수값으로 갱신합니다.

123 : end 매개 변수를 추가합니다.


124 : 학습을 100회 수행할 때마다의 학습 시간을 출력합니다.

123~124 : 가중치 WH 값을 출력부분을 지워줍니다.

139 : end 변수를 선언하고, time.tick_ms 함수를 호출하여 밀리 초 단위의 현재 시간을 저장합니다.

140 : end에서 begin을 빼서 time_taken 변수에 저장합니다.

141 : 전체 학습 시간을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch : 10000, sum error : 0.001731, 7.348 sec

target 0 : 0 0 0 0 pattern 0 : 0.02 0.01 0.01 0.01
target 1 : 0 0 0 1 pattern 1 : 0.01 0.02 0.01 1.00
target 2 : 0 0 1 0 pattern 2 : 0.00 0.00 1.00 0.00
target 3 : 0 0 1 1 pattern 3 : 0.00 0.01 0.99 0.99
target 4 : 0 1 0 0 pattern 4 : 0.01 0.99 0.00 0.01
target 5 : 0 1 0 1 pattern 5 : 0.01 0.98 0.01 0.99
target 6 : 0 1 1 0 pattern 6 : 0.00 1.00 0.99 0.00
target 7 : 0 1 1 1 pattern 7 : 0.00 0.98 0.99 0.99
target 8 : 1 0 0 0 pattern 8 : 0.98 0.00 0.00 0.01
target 9 : 1 0 0 1 pattern 9 : 0.99 0.00 0.00 0.99

Time taken (in seconds) = 735.213
```

05 MyAI 클래스 만들기

myai.py 살펴보기

myai.py

```
001 from ulab import numpy as np
002 import urandom
003 import time
004 from math import sqrt
005 from myrandn import *
006
007 class MyAI:
008     def __init__(self, NUM_I, NUM_H, NUM_M, NUM_O):
009
010         self.WH = np.zeros((NUM_I, NUM_H))
011         self.BH = np.zeros((1, NUM_H))
012         self.WM = np.zeros((NUM_H, NUM_M))
013         self.BM = np.zeros((1, NUM_M))
014         self.WO = np.zeros((NUM_M, NUM_O))
015         self.BO = np.zeros((1, NUM_O))
016
017     def learning(self, I, T, EPOCH, LR):
018
019         NUM_I = self.WH.shape[0]
020         NUM_H = self.WM.shape[0]
021         NUM_M = self.WO.shape[0]
022         NUM_O = self.WO.shape[1]
023
024         WH, BH = self.WH, self.BH
025         WM, BM = self.WM, self.BM
026         WO, BO = self.WO, self.BO
027
028         NUM_PATTERN = len(I)
029
030         shuffled_pattern = [0 for node in range(NUM_PATTERN)] #정수로!
031
```

```

032         for pc in range(NUM_PATTERN) :
033             shuffled_pattern[pc] = pc
034
035         urandom.seed(time.time())
036
037         for m in range(NUM_I):
038             for n in range(NUM_H):
039                 WH[m, n] = randn()/sqrt(NUM_I/2) # He
040
041         for m in range(NUM_H):
042             for n in range(NUM_M):
043                 WM[m, n] = randn()/sqrt(NUM_H/2) # He
044
045         for m in range(NUM_M):
046             for n in range(NUM_O):
047                 WO[m, n] = randn()/sqrt(NUM_M) # Lecun
048
049         begin = time.time()
050         t_prev = time.time()
051
052         for epoch in range(1, EPOCH+1):
053
054             tmp_a = 0;
055             tmp_b = 0;
056             for pc in range(NUM_PATTERN) :
057                 tmp_a = urandom.randrange(0, NUM_PATTERN)
058                 tmp_b = shuffled_pattern[pc]
059                 shuffled_pattern[pc] = shuffled_pattern[tmp_a]
060                 shuffled_pattern[tmp_a] = tmp_b
061
062             sumError = 0.
063
064             for rc in range(NUM_PATTERN) :
065
066                 pc = shuffled_pattern[rc]
067
068                 H = np.dot(I[pc], WH) + BH
069                 H = (H>0)*H # ReLU
070
071                 M = np.dot(H, WM) + BM
072                 M = (M>0)*M # ReLU
073
074                 O = np.dot(M, WO) + BO
075 #                 O = 1/(1+np.exp(-O)) #sigmoid
076
077                 E = np.sum((O-T[pc])**2/2) #mean squared error
078
079                 sumError += E
080
081                 Ob = O - T[pc]
082 #                 Ob = Ob*O*(1-O) #sigmoid

```

```

083
084         Mb = np.dot(Ob, WO.T)
085         Mb = Mb*(M>0)*1 # ReLU
086
087         Hb = np.dot(Mb, WM.T)
088         Hb = Hb*(H>0)*1 # ReLU
089
090         WHb = np.dot([I[pc].T, Hb)
091         BHb = 1 * Hb
092         WMb = np.dot(H.T, Mb)
093         BMb = 1 * Mb
094         WOb = np.dot(M.T, Ob)
095         BOb = 1 * Ob
096
097         lr = LR
098         WH = WH - lr * WHb
099         BH = BH - lr * BHb
100         WM = WM - lr * WMb
101         BM = BM - lr * BMb
102         WO = WO - lr * WOb
103         BO = BO - lr * BOb
104
105         if epoch%10==0:
106             t_now = time.time()
107             time_taken = t_now - t_prev
108             t_prev = t_now
109             print("epoch : %5d, sum error : %f, %.3f sec" % (epoch,
sumError, time_taken/1000))
110
111         self.WH, self.BH = WH, BH
112         self.WM, self.BM = WM, BM
113         self.WO, self.BO = WO, BO
114
115         end = time.time()
116         time_taken = end - begin
117         print("\nTime taken (in seconds) = {}".format(time_taken/1000))
118
119
120     def think(self, I):
121
122         NUM_PATTERN = len(I)
123         NUM_O = self.WO.shape[1]
124         O = [np.zeros((1, NUM_O)) for no in range(NUM_PATTERN)]
125
126         WH, BH = self.WH, self.BH
127         WM, BM = self.WM, self.BM
128         WO, BO = self.WO, self.BO
129
130         for pc in range(NUM_PATTERN) :
131
132             H = np.dot([I[pc], WH] + BH

```



```

133             H = (H>0)*H # ReLU
134
135             M = np.dot(H, WM) + BM
136             M = (M>0)*M # ReLU
137
138             O[pc] = np.dot(M, WO) + BO
139 #             O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
140
141         return O


```

05 입력층과 목표층 바꿔보기

먼저 이전 예제의 입력층과 목표층을 바꿔 인공 신경망을 학습 시켜봅니다. 다음과 같이 2진수가 입력되면 해당되는 7 세그먼트의 켜지고 꺼져야 할 LED의 비트열을 출력합니다.

2 진수 7 세그먼트 연결 진리표

In	In	In	In	Out	Out	Out	Out	Out	Out	Out
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1

0101 ➡ 1011011 = 

예를 들어, “숫자 5에 맞게 7 세그먼트 LED를 켜줘!” 하고 싶을 때, 사용할 수 있는 인공 신경망입니다.

0. 먼저 424_1.py 예제를 425_1.py로 저장합니다.

1. 다음과 같이 파일을 수정합니다.

425_1.py

```

007 NUM_PATTERN = 10

```


```

008 NUM_I = 4
009 NUM_H = 16
010 NUM_M = 16
011 NUM_O = 7
012
013 I = [
014     np.array([[ 0, 0, 0, 0 ]]), # 0
015     np.array([[ 0, 0, 0, 1 ]]), # 1
016     np.array([[ 0, 0, 1, 0 ]]), # 2
017     np.array([[ 0, 0, 1, 1 ]]), # 3
018     np.array([[ 0, 1, 0, 0 ]]), # 4
019     np.array([[ 0, 1, 0, 1 ]]), # 5
020     np.array([[ 0, 1, 1, 0 ]]), # 6
021     np.array([[ 0, 1, 1, 1 ]]), # 7
022     np.array([[ 1, 0, 0, 0 ]]), # 8
023     np.array([[ 1, 0, 0, 1 ]]), # 9
024 ]
025 T = [
026     np.array([[ 1, 1, 1, 1, 1, 1, 0 ]]),
027     np.array([[ 0, 1, 1, 0, 0, 0, 0 ]]),
028     np.array([[ 1, 1, 0, 1, 1, 0, 1 ]]),
029     np.array([[ 1, 1, 1, 1, 0, 0, 1 ]]),
030     np.array([[ 0, 1, 1, 0, 0, 1, 1 ]]),
031     np.array([[ 1, 0, 1, 1, 0, 1, 1 ]]),
032     np.array([[ 0, 0, 1, 1, 1, 1, 1 ]]),
033     np.array([[ 1, 1, 1, 0, 0, 0, 0 ]]),
034     np.array([[ 1, 1, 1, 1, 1, 1, 1 ]]),
035     np.array([[ 1, 1, 1, 0, 0, 1, 1 ]])
036 ]

```

013~024 : 입력층의 입력값을 출력층의 값으로 변경합니다.

025~036 : 목표층의 목표값을 입력층의 값으로 변경합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch : 10000, sum error : 0.002765, 7.561 sec

target 0 : 1 1 1 1 1 1 0 pattern 0 : 0.99 0.99 1.00 0.99 0.99 0.99 0.02
target 1 : 0 1 1 0 0 0 0 pattern 1 : 0.02 1.00 1.00 0.01 0.00 0.01 0.01
target 2 : 1 1 0 1 1 0 1 pattern 2 : 0.99 1.00 0.02 1.00 1.00 0.00 1.00
target 3 : 1 1 1 1 0 0 1 pattern 3 : 0.99 1.00 0.99 0.99 0.01 0.00 0.99
target 4 : 0 1 1 0 0 1 1 pattern 4 : 0.00 0.99 1.00 0.01 0.01 1.00 0.99
target 5 : 1 0 1 1 0 1 1 pattern 5 : 0.99 0.01 1.00 0.98 0.00 1.00 1.00
target 6 : 0 0 1 1 1 1 1 pattern 6 : 0.02 0.01 0.99 0.99 0.99 0.99 0.99
target 7 : 1 1 1 0 0 0 0 pattern 7 : 0.99 0.99 1.00 0.01 0.00 0.01 0.01
target 8 : 1 1 1 1 1 1 1 pattern 8 : 1.00 0.99 1.00 1.00 0.99 1.00 0.99
target 9 : 1 1 1 0 0 1 1 pattern 9 : 1.00 1.00 1.00 0.01 0.00 0.99 0.99

Time taken (in seconds) = 756.32
```

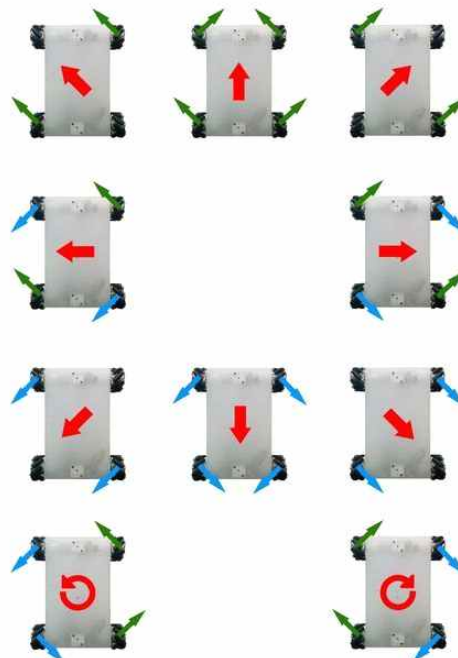
예측값이 목표값이 적당히 가까운 것을 볼 수 있습니다. 예측값을 목표값에 더 가깝게 하려면 훈련의 횟수를 늘리면 됩니다.

06 7 세그먼트 비트열로 매카넘 바퀴 제어하기

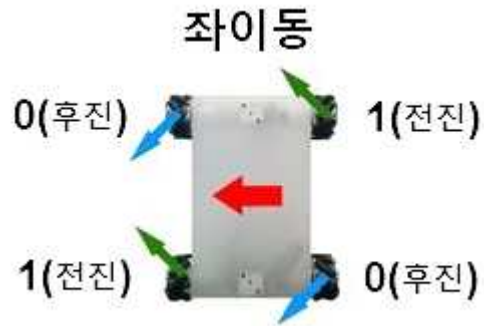
여기서는 7 세그먼트의 비트열을 입력으로 받아 매카넘 휠의 모터를 제어하는 출력을 내도록 인공 신경망을 구성하고, 학습시켜 봅니다.

	D7	D6	D5	D4	D3	D2	D1	D0	16진 코드 (C 언어)
a	b	c	d	e	f	g	dp		
1	1	1	1	1	1	1	0	0	0xFC
0	1	1	0	0	0	0	0	0	0x60
1	1	0	1	1	0	0	1	0	0xDA
1	1	1	1	0	0	0	1	0	0xF2
0	1	1	0	0	1	1	0	0	0x66
1	0	1	1	0	1	1	0	0	0xB6
1	0	1	1	1	1	1	1	0	0xBE
1	1	1	1	0	0	1	0	0	0xE4
1	1	1	1	1	1	1	1	0	0xFE
1	1	1	1	0	1	1	1	0	0xF6
1	1	1	0	1	1	1	1	0	0xEE
0	0	1	1	1	1	1	1	0	0x3E
1	0	0	1	1	1	1	0	0	0x9C
0	1	1	1	1	1	0	1	0	0x7A
1	0	0	1	1	1	1	1	0	0x9E
1	0	0	0	0	1	1	1	0	0x8E

7세그먼트 표준 디스플레이 모양



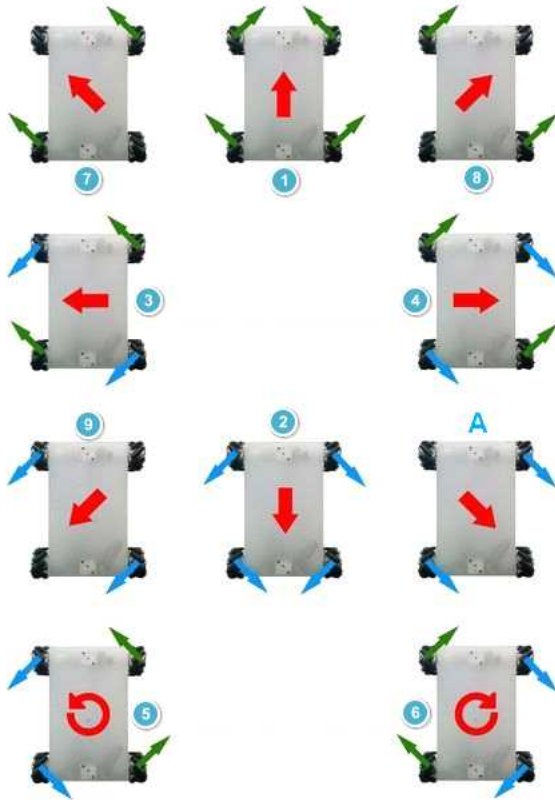
예를 들어, “7 세그먼트 숫자 3의 비트열에 맞게 4 바퀴의 매카넘 바퀴를 움직여줘!” 하고 싶을 때, 사용할 수 있는 인공 신경망입니다. 우리 예제에서 이 경우 매카넘 바퀴를 장착한 RC 카는 다음과 같이 왼쪽으로 수평 이동합니다.



다음과 같이 입력과 출력을 연결할 수 있도록 인공 신경망을 학습시킵니다.

- 0 : 멈춤
- 1 : 전진
- 2 : 후진
- 3 : 좌이동
- 4 : 우이동
- 5 : 좌회전
- 6 : 우회전
- 7 : 좌 대각선 전진
- 8 : 우 대각선 전진
- 9 : 좌 대각선 후진
- A : 우 대각선 후진

다음 그림을 참고합니다.



0. 먼저 425_1.py 예제를 426_1.py로 저장합니다.

1. 다음과 같이 파일을 수정합니다.

426_1.py

```

007 NUM_PATTERN = 11
008 NUM_I = 7
009 NUM_H = 16
010 NUM_M = 16
011 NUM_O = 4
012
013 I = [
014     np.array([[ 1, 1, 1, 1, 1, 1, 0 ]]), # 0
015     np.array([[ 0, 1, 1, 0, 0, 0, 0 ]]), # 1
016     np.array([[ 1, 1, 0, 1, 1, 0, 1 ]]), # 2
017     np.array([[ 1, 1, 1, 1, 0, 0, 1 ]]), # 3
018     np.array([[ 0, 1, 1, 0, 0, 1, 1 ]]), # 4
019     np.array([[ 1, 0, 1, 1, 0, 1, 1 ]]), # 5
020     np.array([[ 0, 0, 1, 1, 1, 1, 1 ]]), # 6
021     np.array([[ 1, 1, 1, 0, 0, 0, 0 ]]), # 7
022     np.array([[ 1, 1, 1, 1, 1, 1, 1 ]]), # 8

```

```

023 np.array([[ 1, 1, 1, 0, 0, 1, 1 ]]), # 9
024 np.array([[ 1, 1, 1, 0, 1, 1, 1 ]]) # A
025 ]
026 T = [
027 np.array([[ 0.5, 0.5, 0.5, 0.5 ]]),
028 np.array([[ 1, 1, 1, 1 ]]),
029 np.array([[ 0, 0, 0, 0 ]]),
030 np.array([[ 0, 1, 0, 1 ]]),
031 np.array([[ 1, 0, 1, 0 ]]),
032 np.array([[ 0, 1, 1, 0 ]]),
033 np.array([[ 1, 0, 0, 1 ]]),
034 np.array([[ 0.5, 1, 0.5, 1 ]]),
035 np.array([[ 1, 0.5, 1, 0.5 ]]),
036 np.array([[ 0.5, 0, 0.5, 0 ]]),
037 np.array([[ 0, 0.5, 0, 0.5 ]])
038 ]

```

007 : 패턴의 개수는 0~9, A까지 11개가 됩니다.

008 : 입력층의 노드 개수는 7개로 합니다. 7 세그먼트의 숫자 표시 LED의 개수가 7개이기 때문입니다.

011 : 출력층의 노드 개수는 4개로 합니다. 4바퀴에 각각에 대한 전진, 멈춤, 후진을 나타내는 값을 출력하게 됩니다.

013~025 : input 배열을 선언하고 초기화합니다. 입력값은 11가지로 7 세그먼트의 0~9, A에 대응되는 비트열입니다. 다음 그림의 D7~D1에 대응되는 비트열입니다.

	D7	D6	D5	D4	D3	D2	D1	D0	16진 코드 (C 언어)
	a	b	c	d	e	f	g	dp	
	1	1	1	1	1	1	0	0	0xFC
	0	1	1	0	0	0	0	0	0x60
	1	1	0	1	1	0	1	0	0xDA
	1	1	1	1	0	0	1	0	0xF2
	0	1	1	0	0	1	1	0	0x66
	1	0	1	1	0	1	1	0	0xB6
	1	0	1	1	1	1	1	0	0xBE
	1	1	1	0	0	1	0	0	0xE4
	1	1	1	1	1	1	1	0	0xFE
	1	1	1	1	0	1	1	0	0xF6
	1	1	1	0	1	1	1	0	0xEE
	0	0	1	1	1	1	1	0	0x3E
	1	0	0	1	1	1	0	0	0x9C
	0	1	1	1	1	0	1	0	0x7A
	1	0	0	1	1	1	1	0	0x9E
	1	0	0	0	1	1	1	0	0x8E

7세그먼트 표준 디스플레이 모양

026~038 : target 배열을 선언하고 초기화합니다. 0번 항목의 경우 메카넘 바퀴를 멈추기 위한 4 바퀴의 값입니다. 차례대로 왼쪽 앞바퀴, 오른쪽 앞바퀴, 오른쪽 뒷바퀴, 왼쪽 뒷바퀴에

대응되는 값입니다. 0.5의 경우 멈춤입니다. 1은 전진을 나타내고, 0은 후진을 나타냅니다. 그래서 1번 항목의 경우 전진을 위한 출력값이며 4 바퀴의 값이 모두 1입니다. 2번 항목의 경우 후진을 위한 출력값이며 4 바퀴의 값이 모두 0입니다. 3번 항목의 경우 좌이동을 위한 출력값이며 4 바퀴의 값이 각각 0(후진), 1(전진), 0(후진), 1(전진)이 됩니다. 다음 그림은 좌이동을 나타내는 그림입니다.




2. 계속해서 다음과 같이 파일을 수정합니다.

426_1.py

```
132 for pc in range(NUM_PATTERN) :
133     print("target %X : "%pc, end='')
134     for node in range(NUM_O) :
135         print("%.0f "%T[pc][0][node], end='')
136     print("pattern %X : "%pc, end='');
137     for node in range(NUM_O) :
138         print("%.2f "%O[pc][0][node], end='')
139     print()
```

133, 136 : %d를 %X로 변경하여 10진수를 16진수로 표시하게 합니다.

3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch : 10000, sum error : 0.002085, 8.098 sec

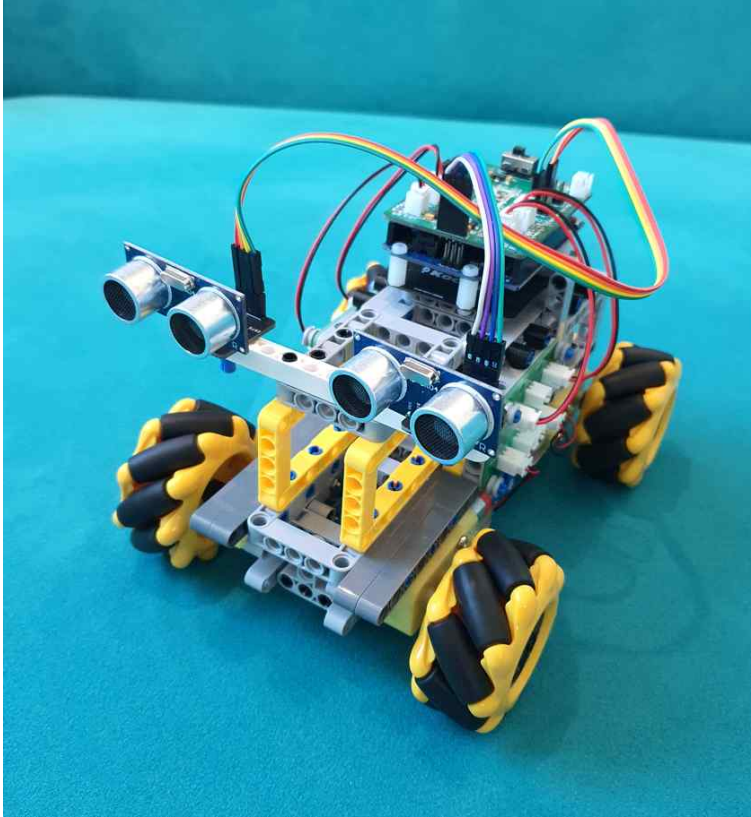
target 0 : 0 0 0 0 pattern 0 : 0.50 0.50 0.50 0.50
target 1 : 1 1 1 1 pattern 1 : 1.00 0.98 1.00 0.99
target 2 : 0 0 0 0 pattern 2 : 0.00 0.01 0.00 0.01
target 3 : 0 1 0 1 pattern 3 : 0.01 1.00 0.01 0.99
target 4 : 1 0 1 0 pattern 4 : 1.00 0.01 1.00 0.01
target 5 : 0 1 1 0 pattern 5 : 0.01 0.99 0.99 0.01
target 6 : 1 0 0 1 pattern 6 : 0.99 0.02 0.01 1.00
target 7 : 0 1 0 1 pattern 7 : 0.50 0.99 0.50 0.99
target 8 : 1 0 1 0 pattern 8 : 0.98 0.50 0.98 0.50
target 9 : 0 0 0 0 pattern 9 : 0.50 0.02 0.50 0.01
target A : 0 0 0 0 pattern A : 0.01 0.50 0.02 0.50

Time taken (in seconds) = 809.9299
```

예측값이 목표값이 적당히 가까운 것을 볼 수 있습니다. 예측값을 목표값에 더 가깝게 하려면 훈련의 횟수를 늘리면 됩니다.

07 초음파 센서 자율주행 인공 신경망

여기서는 RC카에 장착된 초음파 센서로부터 물체와의 거리를 입력받아 RC카의 모터를 제어하여 출력을 내도록 인공 신경망을 구성하고, 학습시켜 봅니다. 다음 그림은 초음파 센서가 장착된 라즈베리파이 피코 RC카입니다.



예를 들어, “왼쪽 25cm, 오른쪽 14cm에 물체가 있으면 왼쪽으로 움직여줘!” 하고 싶을 때, 사용할 수 있는 인공 신경망입니다.

0. 먼저 426_1.py 예제를 427_1.py로 저장합니다.

1. 다음과 같이 파일을 수정합니다.

427_1.py

```
007 NUM_PATTERN = 25
008 NUM_I = 2
009 NUM_H = 16
010 NUM_M = 16
011 NUM_O = 3
012
013 I = [
```



```

014 np.array([[ 25, 14 ]], # 0
015 np.array([[ 41, 33 ]], # 1
016 np.array([[ 44, 44 ]], # 2
017 np.array([[ 33, 41 ]], # 3
018 np.array([[ 14, 25 ]], # 4
019 np.array([[ 29, 22 ]], # 5
020 np.array([[ 43, 33 ]], # 6
021 np.array([[ 80, 90 ]], # 7
022 np.array([[ 33, 43 ]], # 8
023 np.array([[ 22, 29 ]], # 9
024 np.array([[ 35, 26 ]], # A
025 np.array([[ 55, 35 ]], # 0
026 np.array([[ 55, 55 ]], # 1
027 np.array([[ 35, 55 ]], # 2
028 np.array([[ 26, 35 ]], # 3
029 np.array([[ 33, 25 ]], # 4
030 np.array([[ 44, 32 ]], # 5
031 np.array([[ 150,150 ]], # 6
032 np.array([[ 32, 44 ]], # 7
033 np.array([[ 25, 33 ]], # 8
034 np.array([[ 38, 23 ]], # 9
035 np.array([[ 50, 36 ]], # A
036 np.array([[ 90,100 ]], # 8
037 np.array([[ 36, 50 ]], # 9
038 np.array([[ 23, 38 ]], # A
039 ]
040 T = [
041 np.array([[ 1,0,0 ]],
042 np.array([[ 0,1,0 ]],
043 np.array([[ 0,1,0 ]],
044 np.array([[ 0,1,0 ]],
045 np.array([[ 0,0,1 ]],
046 np.array([[ 1,0,0 ]],
047 np.array([[ 0,1,0 ]],
048 np.array([[ 0,1,0 ]],
049 np.array([[ 0,1,0 ]],
050 np.array([[ 0,0,1 ]],
051 np.array([[ 1,0,0 ]],
052 np.array([[ 0,1,0 ]],
053 np.array([[ 0,1,0 ]],

```

```

054 np.array([[ 0,1,0 ]]),
055 np.array([[ 0,0,1 ]]),
056 np.array([[ 1,0,0 ]]),
057 np.array([[ 0,1,0 ]]),
058 np.array([[ 0,1,0 ]]),
059 np.array([[ 0,1,0 ]]),
060 np.array([[ 0,0,1 ]]),
061 np.array([[ 1,0,0 ]]),
062 np.array([[ 0,1,0 ]]),
063 np.array([[ 0,1,0 ]]),
064 np.array([[ 0,1,0 ]]),
065 np.array([[ 0,0,1 ]])
066 ]

```

007 : 패턴의 개수는 25개로 합니다.

008 : 입력층의 노드 개수는 2개로 합니다. 오른쪽, 왼쪽 2 방향의 거리 값이 입력이 됩니다.

010 : 출력층의 노드 개수는 3개로 합니다. 오른쪽 전진, 왼쪽 전진, 양쪽 전진의 3가지 동작을 나타내는 값을 출력하게 됩니다.

13~39 : input 배열을 선언하고 초기화합니다. 입력값은 25가지로 오른쪽, 왼쪽 2 방향의 거리 값입니다. 예를 들어 0번 항목의 경우 왼쪽이 25cm, 오른쪽이 14cm일 경우를 나타냅니다.

40~66 : target 배열을 선언하고 초기화합니다. 0번 항목의 경우 왼쪽 전진을 의미합니다. 왼쪽이 25cm, 오른쪽이 14cm일 경우 물체가 더 먼 왼쪽 방향으로 이동해야 합니다. 2번 항목의 경우 양쪽 전진을 의미합니다. 물체가 조금 멀리 있는 경우로, 양쪽 전진을 하도록 합니다. 5번 항목의 경우 오른쪽 전진을 의미합니다. 왼쪽이 14cm, 오른쪽이 25cm일 경우 물체가 더 먼 오른쪽 방향으로 이동해야 합니다.

2. 계속해서 다음과 같이 파일을 수정합니다.

427_1.py

```

096 for pc in range(NUM_PATTERN):
097     I[pc] /= 250
098
099 for epoch in range(1, 10001):

```

096~097 : 초음파 센서의 입력값을 250으로 나누어 0.0~1.0 사이의 값이 되도록 합니다. 이 예제에서 사용하는 인공 신경망의 입력값이 0.0~1.0 사이가 되게 합니다.

3. 계속해서 다음과 같이 파일을 수정합니다.

427_1.py

```

163 for pc in range(NUM_PATTERN) :
164     print("target %2d : "%pc, end="")
165     for node in range(NUM_O) :


```

```

166         print("%.0f %T[pc][0][node], end='')
167     print("pattern %2d : %pc, end='');
168     for node in range(NUM_O) :
169         print("%.2f %O[pc][0][node], end='')
170     print()

```

164,167 : 입력 패턴의 종류가 25개이기 때문에 printf 함수의 출력 형식을 %2d로 하여 10진수 2자리로 출력하도록 합니다.

4.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch : 10000, sum error : 0.012392, 16.447 sec
```

```

target 0 : 1 0 0 pattern 0 : 0.99 0.00 0.01
target 1 : 0 1 0 pattern 1 : 0.02 0.98 0.00
target 2 : 0 1 0 pattern 2 : 0.00 1.00 0.00
target 3 : 0 1 0 pattern 3 : 0.00 0.98 0.02
target 4 : 0 0 1 pattern 4 : 0.01 0.01 0.99
target 5 : 1 0 0 pattern 5 : 0.96 0.00 0.05
target 6 : 0 1 0 pattern 6 : 0.02 0.99 0.00
target 7 : 0 1 0 pattern 7 : 0.00 1.00 0.00
target 8 : 0 1 0 pattern 8 : 0.00 1.00 0.02
target 9 : 0 0 1 pattern 9 : 0.04 0.01 0.95
target 10 : 1 0 0 pattern 10 : 0.95 0.02 0.01
target 11 : 0 1 0 pattern 11 : 0.00 1.00 0.00
target 12 : 0 1 0 pattern 12 : 0.00 1.00 0.00
target 13 : 0 1 0 pattern 13 : 0.00 1.00 0.00
target 14 : 0 0 1 pattern 14 : 0.01 0.02 0.96
target 15 : 1 0 0 pattern 15 : 0.97 0.01 0.03
target 16 : 0 1 0 pattern 16 : 0.04 0.99 0.00
target 17 : 0 1 0 pattern 17 : 0.00 1.00 0.00
target 18 : 0 1 0 pattern 18 : 0.00 1.00 0.04
target 19 : 0 0 1 pattern 19 : 0.03 0.01 0.96
target 20 : 1 0 0 pattern 20 : 0.99 0.02 0.00
target 21 : 0 1 0 pattern 21 : 0.00 1.00 0.00
target 22 : 0 1 0 pattern 22 : 0.00 1.00 0.00
target 23 : 0 1 0 pattern 23 : 0.00 1.00 0.00
target 24 : 0 0 1 pattern 24 : 0.00 0.02 1.00

```

```
Time taken (in seconds) = 1645.645
```

예측값이 목표값이 적당히 가까운 것을 볼 수 있습니다. 예측값을 목표값에 더 가깝게 하려면 훈련의 횟수를 늘리면 됩니다.