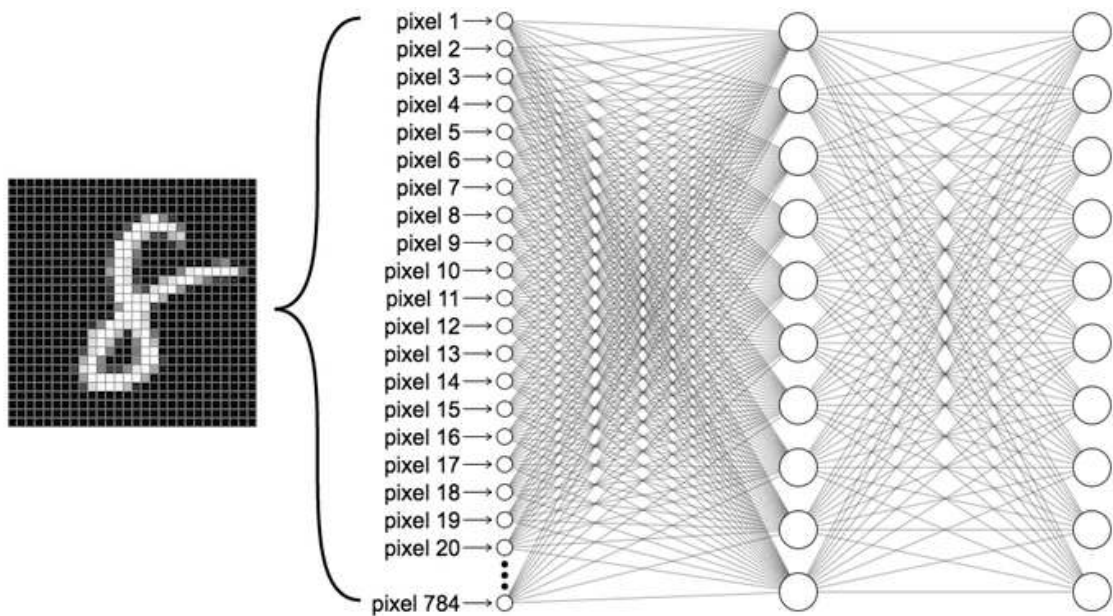


Chapter 03

NumPy DNN 구현과 활용

01 NumPy DNN 구현하기

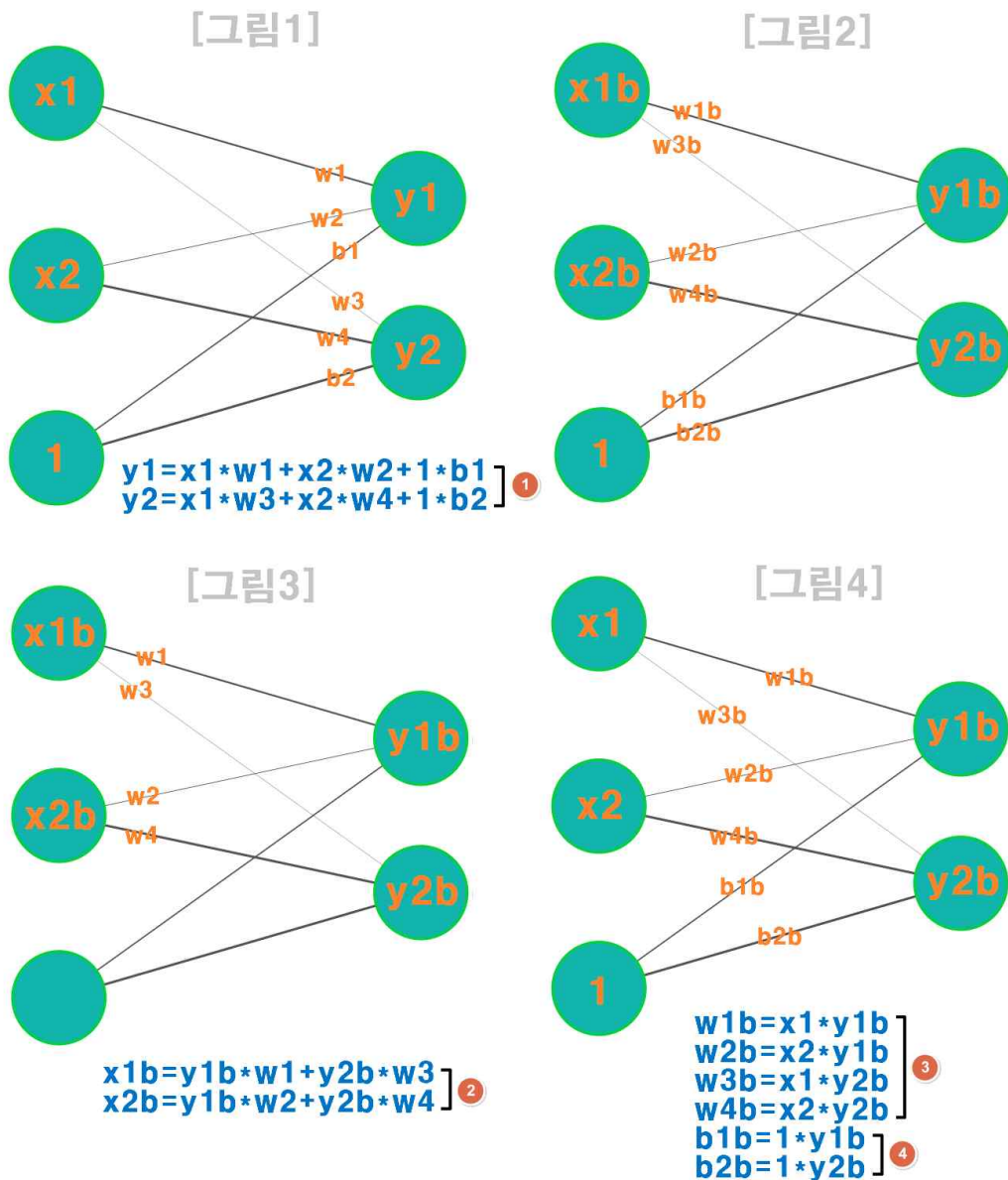
여기서는 인공 신경망을 확장할 수 있도록 NumPy 라이브러리를 활용하여 인공 신경망을 구현해 봅니다. NumPy 라이브러리를 이용하면, 커다란 인공 신경망을 자유롭게 구성하고 테스트해 볼 수 있습니다. 예를 들어, 1장에서 tensorflow 라이브러리를 이용하여 살펴보았던 다음과 같은 형태의 인공 신경망을 구성해서 테스트해 볼 수 있습니다.



<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.

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

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

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

행렬의 곱 XW 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 ❸에서 다음은 순전파 때 사용된 입력의 전치 행렬입니다.

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

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.

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

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

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

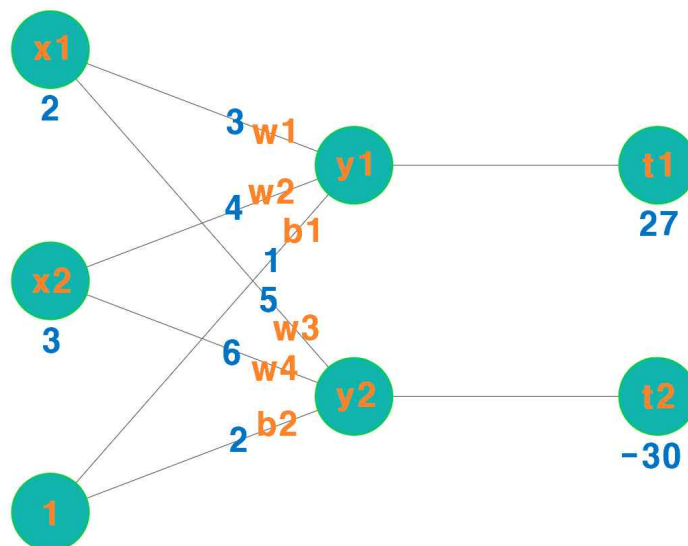
인공 신경망 행렬 계산식 정리하기

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 T 는 다음과 같습니다.

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

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.

311_1.py

```
01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
04
05 X = np.array([[2, 3]])
06 T = np.array([[27, -30]])
07 W = np.array([[3, 5],
08               [4, 6]])
09 B = np.array([[1, 2]])
10
11 for epoch in range(1000):
12
13     print('epoch = %d' %epoch)
14
15     Y = X @ W + B # ❶
16     print(' Y =', Y)
17
18     E = np.sum((Y - T) ** 2 / 2)
19     print(' E = %.7f' %E)
20     if E < 0.0000001:
21         break
22
23     Yb = Y - T
24     Xb = Yb @ W.T # ❷
25     Wb = X.T @ Yb # ❸
26     Bb = 1 * Yb # ❹
27     print(' Xb =\n', Xb)
```

```

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

```

01 : import문을 이용하여 numpy 모듈을 np라는 이름으로 불러옵니다. numpy 모듈은 행렬 계산을 편하게 해주는 라이브러리입니다. 인공 신경망은 일반적으로 행렬 계산식으로 구성하게 됩니다.

03 : np.set_printoptions 함수를 호출하여 numpy의 실수 출력 방법을 변경합니다. 이 예제에서는 소수점 이하 3자리까지 출력합니다.

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

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

07, 08 : np.array 함수를 호출하여 2x2 행렬을 생성하여 W 변수에 할당합니다.

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

11 : epoch값을 0에서 1000 미만까지 바꾸어가며 13~36줄을 1000회 수행합니다.

13 : print 함수를 호출하여 Y값을 출력합니다.

15 : 행렬 곱 연산자 @을 이용하여 입력 X와 가중치 W에 대해 행렬 곱을 수행한 후, 편향 B를 더해준 후, Y 변수에 할당합니다. 행렬 곱의 순서를 변경하지 않도록 주의합니다.

18 : 평균 제곱 오차를 구합니다.

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

20, 21 : 평균 제곱 오차가 0.0000001(천만분의 1)보다 작으면 break문을 사용하여 11줄의 for 문을 빠져 나갑니다.

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

24 : 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. 행렬 곱 연산자 @을 이용하여 Yb와 W.T에 대해 행렬 곱을 수행한 후, 결과 값을 Xb 변수에 할당합니다. 행렬 곱의 순서를 변경하지 않도록 주의합니다.

25 : X.T는 입력 X의 전치 행렬을 내어줍니다. 행렬 곱 연산자 @을 이용하여 X.T와 Yb에 대해 행렬 곱을 수행한 후, 결과 값을 Wb 변수에 할당합니다. 행렬 곱의 순서를 변경하지 않도록 주의합니다.

26 : Yb 행렬에 1을 곱해주어 Bb에 할당합니다. 여기서 1은 수식을 강조하기 위해 생략하지 않았습니니다.


27~29 : print 함수를 호출하여 Xb, Wb, Bb값을 출력합니다.

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

32 : 가중치를 갱신합니다.

33 : 편향을 갱신합니다.

34, 35 : print 함수를 호출하여 W, B값을 출력합니다.

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

```
epoch = 78
Y = [[27.000 -30.000]]
E = 0.0000001
Xb =
[[-0.002 -0.004]]
Wb =
[[-0.000 0.001]
 [-0.000 0.001]]
Bb =
[[-0.000 0.000]]
W =
[[ 4.143 -3.571]
 [ 5.714 -6.857]]
B =
[[ 1.571 -2.286]]
epoch = 79
Y = [[27.000 -30.000]]
E = 0.0000001
```

(79+1)회 째 학습이 완료되는 것을 볼 수 있습니다.

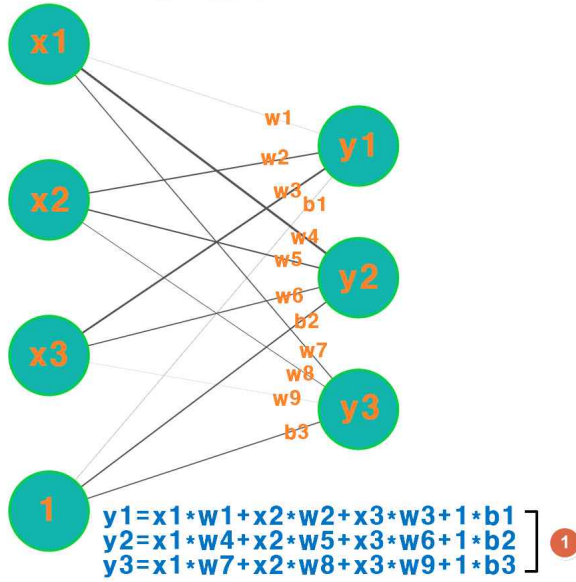
231_1.py 예제의 결과와 비교해 봅니다.

```
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
```

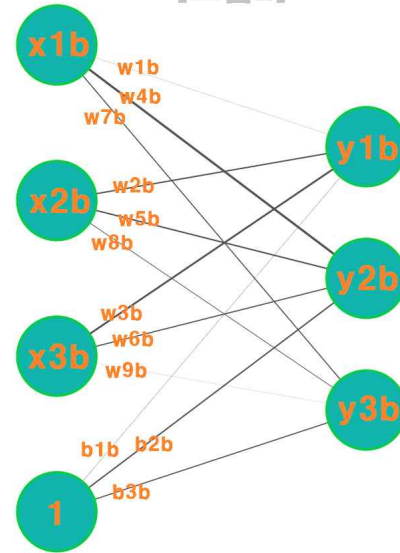
02 3입력 3출력 인공 신경망 구현하기

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

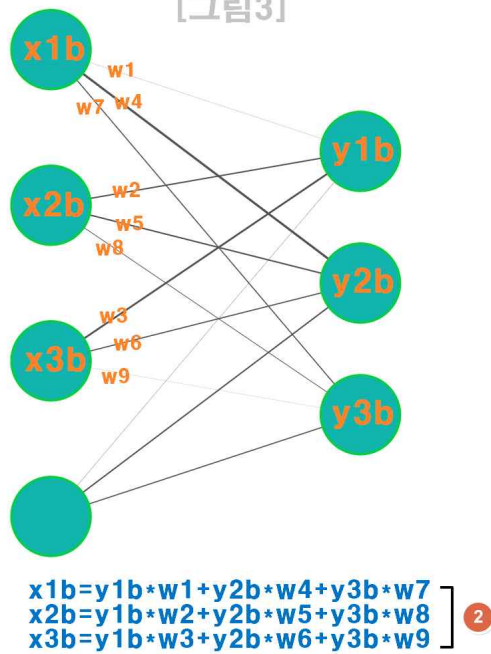
[그림1]



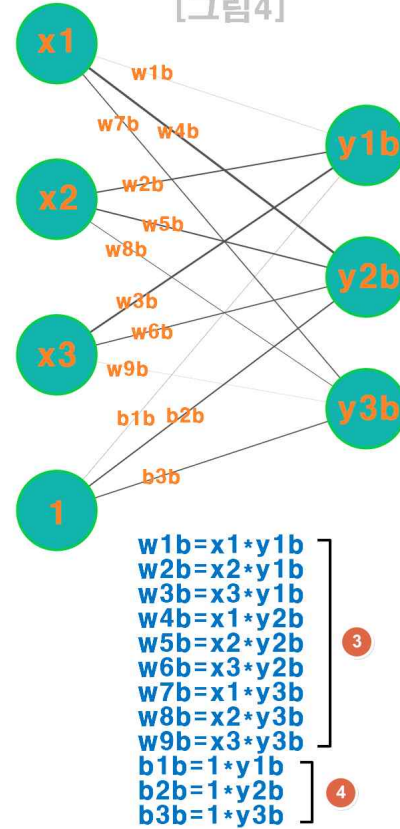
[그림2]



[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	$\left. \begin{aligned} y_{1b}w_1 + y_{2b}w_4 + y_{3b}w_7 &= x_{1b} \\ y_{1b}w_2 + y_{2b}w_5 + y_{3b}w_8 &= x_{2b} \\ y_{1b}w_3 + y_{2b}w_6 + y_{3b}w_9 &= x_{3b} \end{aligned} \right\} \textcircled{2}$	$\begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} =$ $\begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} \begin{bmatrix} w_1 & w_4 & w_7 \\ w_2 & w_5 & w_8 \\ w_3 & w_6 & w_9 \end{bmatrix}^T = \begin{bmatrix} x_{1b} & x_{2b} & x_{3b} \end{bmatrix} \textcircled{2}$
가 중 치 편 향 역 전 파	$\left. \begin{aligned} x_1y_{1b} &= w_{1b} \\ x_2y_{1b} &= w_{2b} \\ x_3y_{1b} &= w_{3b} \\ x_1y_{2b} &= w_{4b} \\ x_2y_{2b} &= w_{5b} \\ x_3y_{2b} &= w_{6b} \\ x_1y_{3b} &= w_{7b} \\ x_2y_{3b} &= w_{8b} \\ x_3y_{3b} &= w_{9b} \\ 1y_{1b} &= b_{1b} \\ 1y_{2b} &= b_{2b} \\ 1y_{3b} &= b_{3b} \end{aligned} \right\} \textcircled{3}$ $\left. \begin{aligned} 1y_{1b} &= b_{1b} \\ 1y_{2b} &= b_{2b} \\ 1y_{3b} &= b_{3b} \end{aligned} \right\} \textcircled{4}$	$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} =$ $\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{4b} & w_{7b} \\ w_{2b} & w_{5b} & w_{8b} \\ w_{3b} & w_{6b} & w_{9b} \end{bmatrix} \textcircled{3}$ $1 \begin{bmatrix} y_{1b} & y_{2b} & y_{3b} \end{bmatrix} = \begin{bmatrix} b_{1b} & b_{2b} & b_{3b} \end{bmatrix} \textcircled{4}$

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.

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

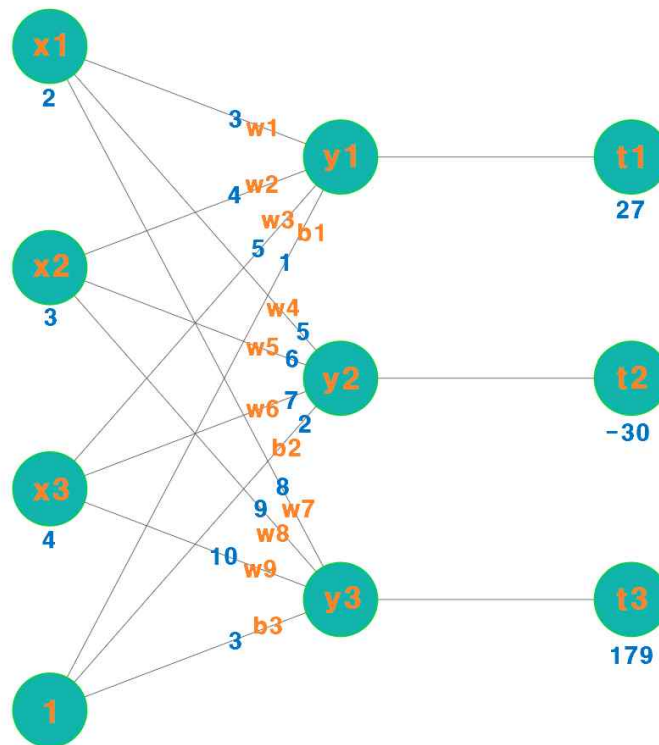
인공 신경망 행렬 계산식 정리하기

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.

312_1.py

```
01~03 # 이전 예제와 같습니다.
04
05 X = np.array([[2, 3, 4]])
```




```

06 T = np.array([[27, -30, 179]])
07 W = np.array([[3, 5, 8],
08               [4, 6, 9],
09               [5, 7, 10]])
10 B = np.array([1, 2, 3])
11
12~끝 # 이전 예제와 같습니다.

```

05~10 : X, T, W, B를 변경해줍니다.

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

```

epoch = 35
Y = [[27.000 -30.000 179.000]]
E = 0.0000001
Xb =
[[-0.005 -0.007 -0.009]]
Wb =
[[ 0.000  0.001 -0.001]
 [ 0.000  0.001 -0.001]
 [ 0.000  0.001 -0.001]]
Bb =
[[ 0.000  0.000 -0.000]]
W =
[[ 2.200 -0.867 14.200]
 [ 2.800 -2.800 18.300]
 [ 3.400 -4.733 22.400]]
B =
[[ 0.600 -0.933  6.100]]
epoch = 36
Y = [[27.000 -30.000 179.000]]
E = 0.0000001

```

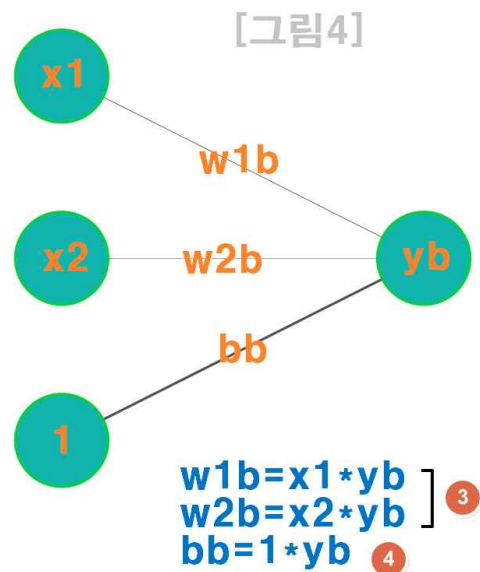
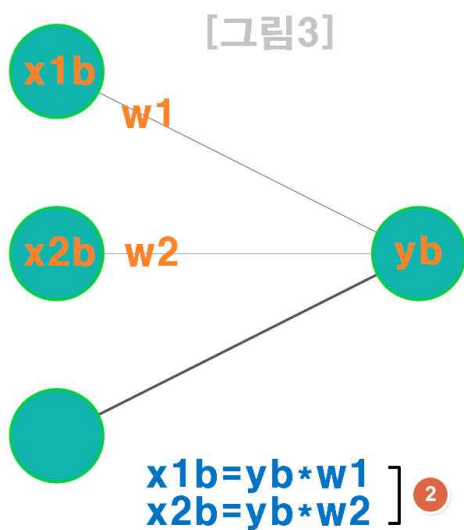
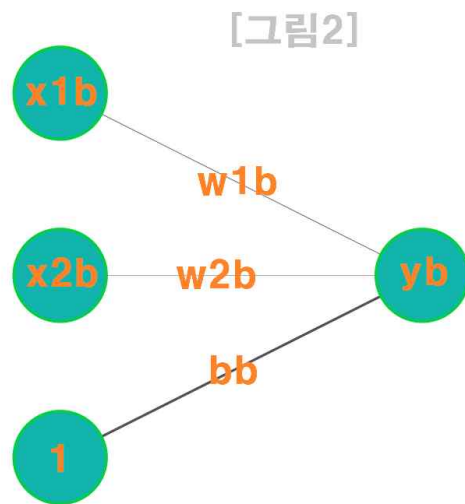
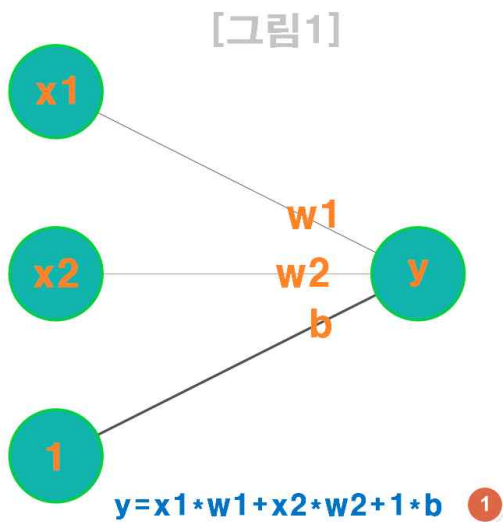
36회 째 학습이 완료되는 것을 볼 수 있습니다.

233_1 예제의 결과와 비교해 봅니다.

```
epoch = 35
y1, y2, y3 = 27.000, -30.000, 179.000
E = 0.0000001
x1b, x2b, x3b = -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 = 0.0000001
```

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.

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

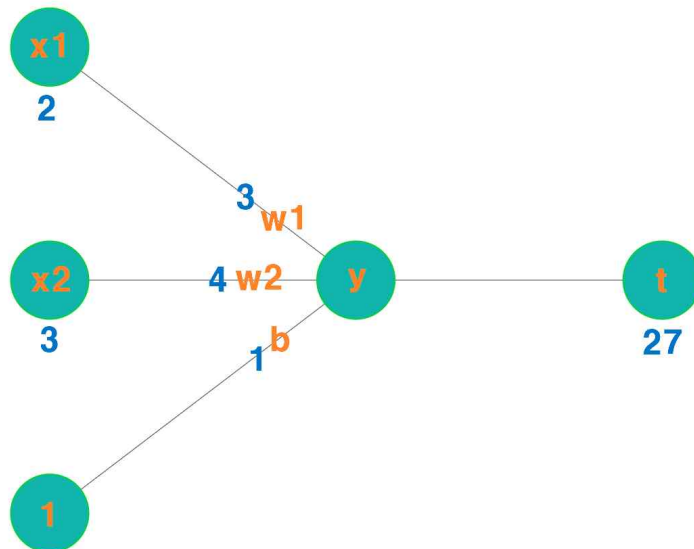
인공 신경망 행렬 계산식 정리하기

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.

313_1.py

```
01~03 # 이전 예제와 같습니다.
04
05 X = np.array([[2, 3]])
06 T = np.array([[27]])
07 W = np.array([[3],
08               [4]])
09 B = np.array([1])
10
11~끝 # 이전 예제와 같습니다.
```

05~09 : X, T, W, B를 변경해줍니다.

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

```

epoch = 64
Y = [[26.999]]
E = 0.00000001
Xb =
[[-0.002 -0.003]]
Wb =
[[-0.001]
 [-0.002]]
Bb =
[[-0.001]]
W =
[[ 4.143]
 [ 5.714]]
B =
[[ 1.571]]
epoch = 65
Y = [[27.000]]
E = 0.00000001

```

65회 째 학습이 완료되는 것을 볼 수 있습니다.

231_1.py 예제의 결과와 비교해 봅니다.

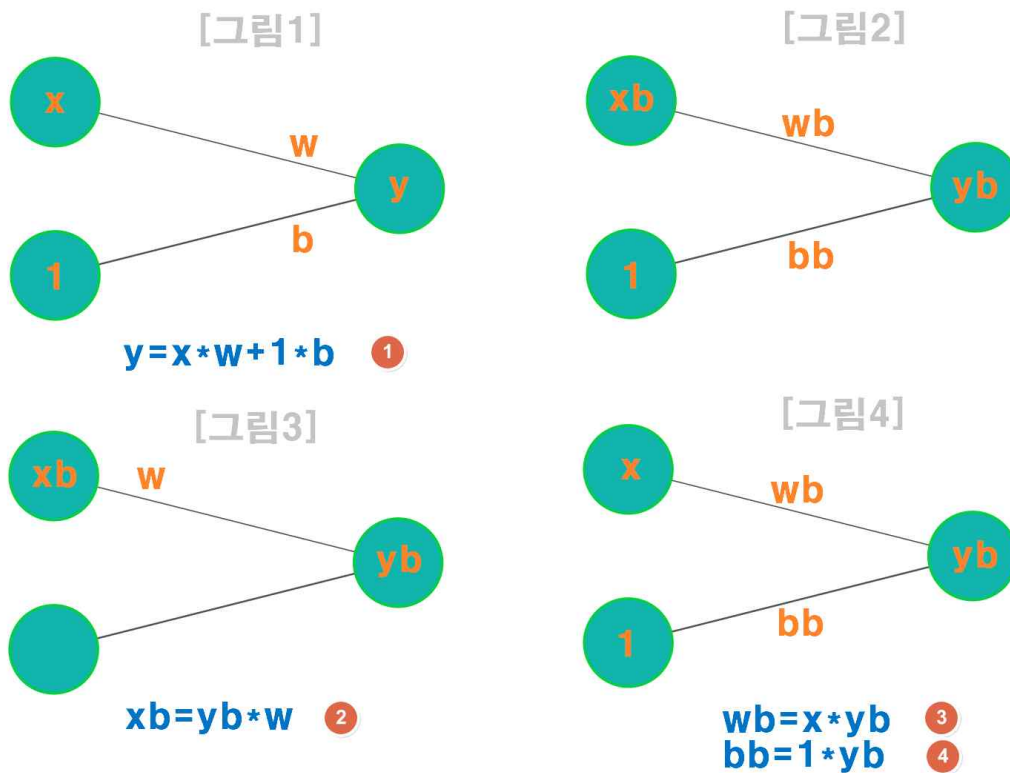
```

epoch = 64
y = 26.999
E = 0.00000001
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 = 0.00000001

```

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 x_b 값은 앞부분에 또 다른 인공 신경과 연결되어 있을 경우 y_b 처럼 해당 인공 신경으로 역전파되는 값입니다. 역전파된 x_b 값은 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	$[y_b][w] = [x_b]$ $[y_b][w]^T = [x_b]$ ②
가중치, 편향 역전파	$xy_b = w_b$ ③ $1y_b = b_b$ ④	$[x][y_b] = [w_b]$ $[x]^T[y_b] = [w_b]$ 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.

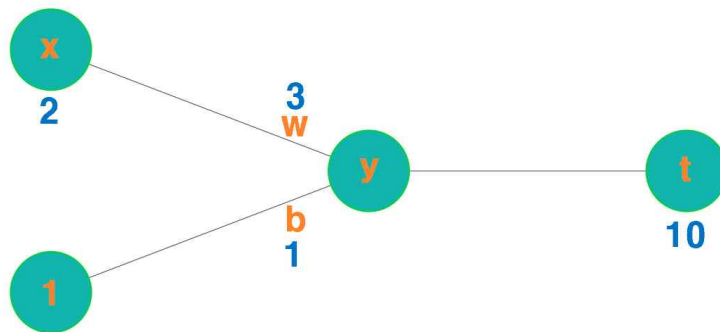
인공 신경망 행렬 계산식 정리하기

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.

314_1.py

```
01~03 # 이전 예제와 같습니다.  
04  
05 X = np.array([[2]])  
06 T = np.array([[10]])  
07 W = np.array([[3]])  
08 B = np.array([[1]])  
09  
10~끝 # 이전 예제와 같습니다.
```

05~08 : X, T, W, B를 변경해줍니다.

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

```
epoch = 171  
Y = [[10.000]]  
E = 0.0000001  
Xb =  
[[-0.002]]  
Wb =  
[[-0.001]]  
Bb =  
[[-0.000]]  
W =  
[[ 4.200]]  
B =  
[[ 1.600]]  
epoch = 172  
Y = [[10.000]]  
E = 0.0000001
```

172회 째 학습이 완료되는 것을 볼 수 있습니다.

222_1.py 예제의 결과와 비교해 봅니다.

```

epoch = 171
y = 10.000
E = 0.00000001
xb = -0.002, wb = -0.001, bb = -0.000
x = 2.000, w = 4.200, b = 1.600
epoch = 172
y = 10.000
E = 0.00000001

```

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.

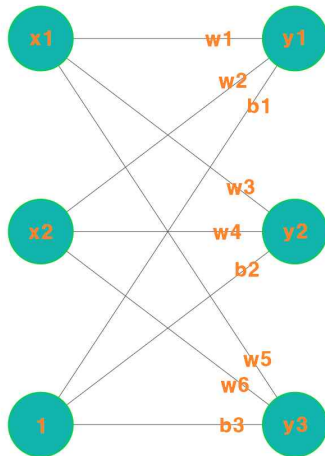
인공 신경망 동작	행렬 계산식	기본 수식
순전파	$Y = XW + B$	$y = xw + b$
입력 역전파	$X_b = Y_b W^T$	$x_b = y_b w$
가중치, 편향 역전파	$W_b = X^T Y_b$ $B_b = Y_b$	$w_b = x y_b$ $b_b = y_b$
인공 신경망 학습	$W = W - \alpha W_b$ $B = B - \alpha B_b$	$w = w - \alpha w_b$ $b = b - \alpha b_b$

*** 행렬 곱 연산은 순서를 지켜야 합니다.

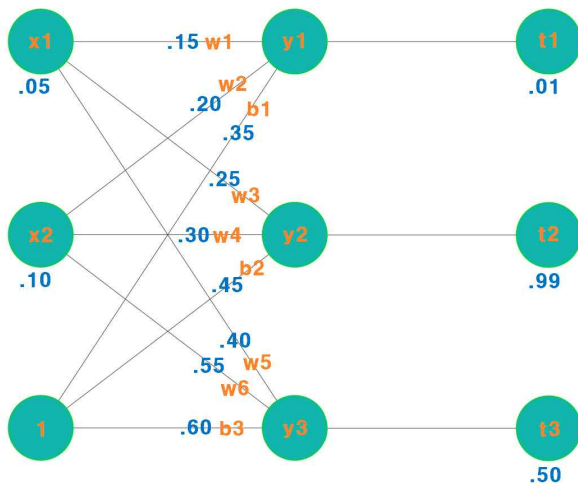
연습문제

❶ 2입력 3출력

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

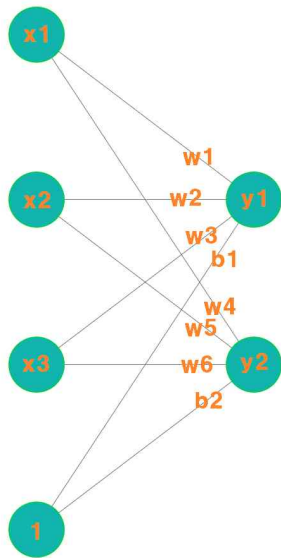


2. 앞에서 구한 행렬 계산식을 이용하여 다음과 같이 초기화된 인공 신경망을 NumPy를 이용하여 구현하고 학습시켜 봅니다. 입력 값 X 는 상수로 처리합니다.

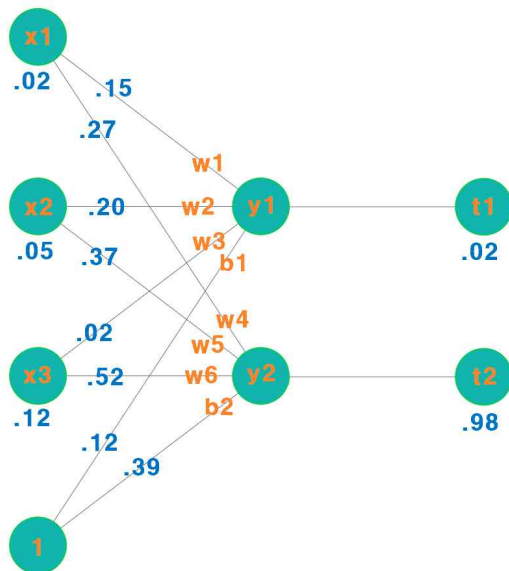


② 3입력 2출력

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



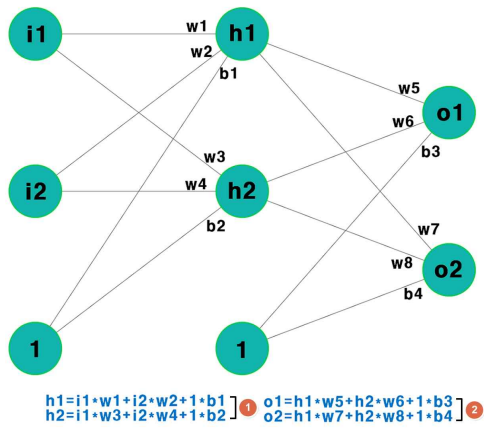
2. 앞에서 구한 행렬 계산식을 이용하여 다음과 같이 초기화된 인공 신경망을 NumPy를 이용하여 구현하고 학습시켜 봅니다. 입력 값 X는 상수로 처리합니다.



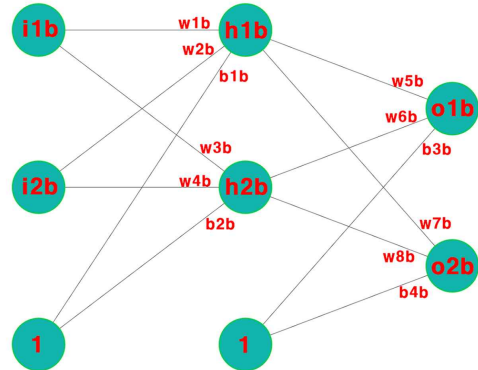
06 2입력 2은닉 2출력 인공 신경망 구현하기

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.

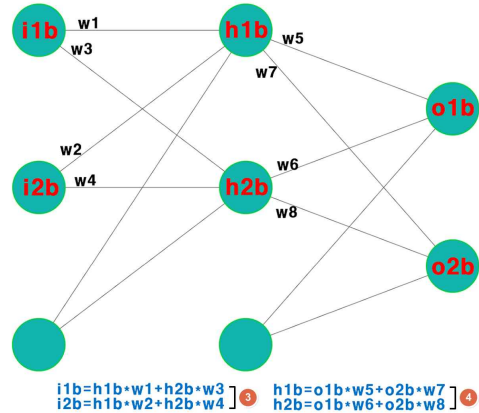
[그림1]



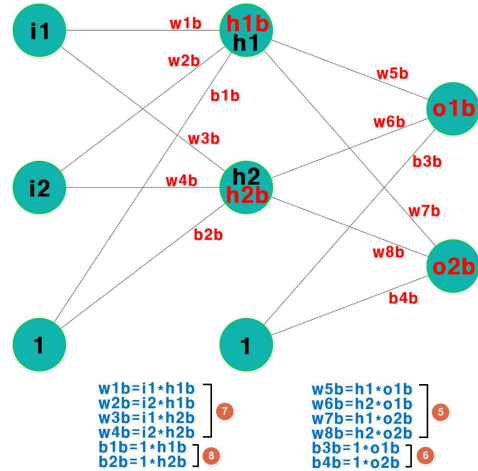
[그림2]



[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	$\begin{bmatrix} i_1 h_{1b} = w_{1b} \\ i_2 h_{1b} = w_{2b} \\ i_1 h_{2b} = w_{3b} \\ i_2 h_{2b} = w_{4b} \\ 1h_{1b} = b_{1b} \\ 1h_{2b} = b_{2b} \end{bmatrix} \begin{matrix} \textcircled{7} \\ \textcircled{8} \end{matrix}$ $\begin{bmatrix} h_1 o_{1b} = w_{5b} \\ h_2 o_{1b} = w_{6b} \\ h_1 o_{2b} = w_{7b} \\ h_2 o_{2b} = w_{8b} \\ 1o_{1b} = b_{3b} \\ 1o_{2b} = b_{4b} \end{bmatrix} \begin{matrix} \textcircled{5} \\ \textcircled{6} \end{matrix}$	$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix} \textcircled{7}$ $\begin{bmatrix} i_1 & i_2 \end{bmatrix}^T \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} = \begin{bmatrix} w_{1b} & w_{3b} \\ w_{2b} & w_{4b} \end{bmatrix}$ $1 \begin{bmatrix} h_{1b} & h_{2b} \end{bmatrix} = \begin{bmatrix} b_{1b} & b_{2b} \end{bmatrix} \textcircled{8}$ $\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} = \begin{bmatrix} w_{5b} & w_{7b} \\ w_{6b} & w_{8b} \end{bmatrix} \textcircled{5}$ $\begin{bmatrix} h_1 & h_2 \end{bmatrix}^T \begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} = \begin{bmatrix} w_{5b} & w_{7b} \\ w_{6b} & w_{8b} \end{bmatrix}$ $1 \begin{bmatrix} o_{1b} & o_{2b} \end{bmatrix} = \begin{bmatrix} b_{3b} & b_{4b} \end{bmatrix} \textcircled{6}$
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.

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

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

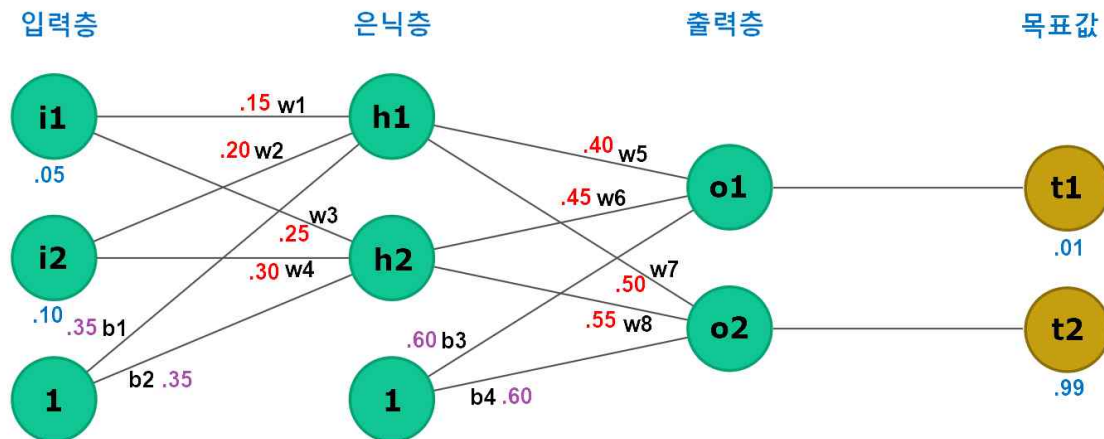
인공 신경망 행렬 계산식 정리하기

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.

316_1.py


```
01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
04
05 I = np.array([[.05, .10]])
```

```

06 T = np.array([[.01, .99]])
07 WH = np.array([[.15, .25],
08                [.20, .30]])
09 BH = np.array([[.35, .35]])
10 WO = np.array([[.40, .50],
11                [.45, .55]])
12 BO = np.array([[.60, .60]])
13
14 for epoch in range(1000):
15
16     print('epoch = %d' %epoch)
17
18     H = I @ WH + BH # ❶
19     O = H @ WO + BO # ❷
20     print(' O  =\n', O)
21
22     E = np.sum((O - T) ** 2 / 2)
23     print(' E  = %.7f' %E)
24     if E < 0.0000001:
25         break
26
27     Ob = O - T
28     Hb = Ob @ WO.T # ❸
29     WHb = I.T @ Hb # ❹
30     BHb = 1 * Hb # ❺
31     WOb = H.T @ Ob # ❻
32     BOb = 1 * Ob # ❼
33     print(' WHb =\n', WHb)
34     print(' BHb =\n', BHb)
35     print(' WOb =\n', WOb)
36     print(' BOb =\n', BOb)
37
38     lr = 0.01
39     WH = WH - lr * WHb # ❽
40     BH = BH - lr * BHb # ❾
41     WO = WO - lr * WOb # ❿
42     BO = BO - lr * BOb # ⓫
43     print(' WH  =\n', WH)
44     print(' BH  =\n', BH)
45     print(' WO  =\n', WO)

```

```
46 print(' BO =\n', BO)
```

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

```
epoch = 664
O =
[[ 0.010  0.990]]
E = 0.0000001
WHb =
[[-0.000  0.000]
 [-0.000  0.000]]
BHb =
[[-0.000  0.000]]
WOb =
[[ 0.000 -0.000]
 [ 0.000 -0.000]]
BOb =
[[ 0.000 -0.000]]
WH =
[[ 0.143  0.242]
 [ 0.186  0.284]]
BH =
[[ 0.213  0.186]]
WO =
[[ 0.203  0.533]
 [ 0.253  0.583]]
BO =
[[-0.095  0.730]]
epoch = 665
O =
[[ 0.010  0.990]]
E = 0.0000001
```

(665+1)회 째 학습이 완료되는 것을 볼 수 있습니다.

234_1.py 예제의 결과와 비교해 봅니다.

```

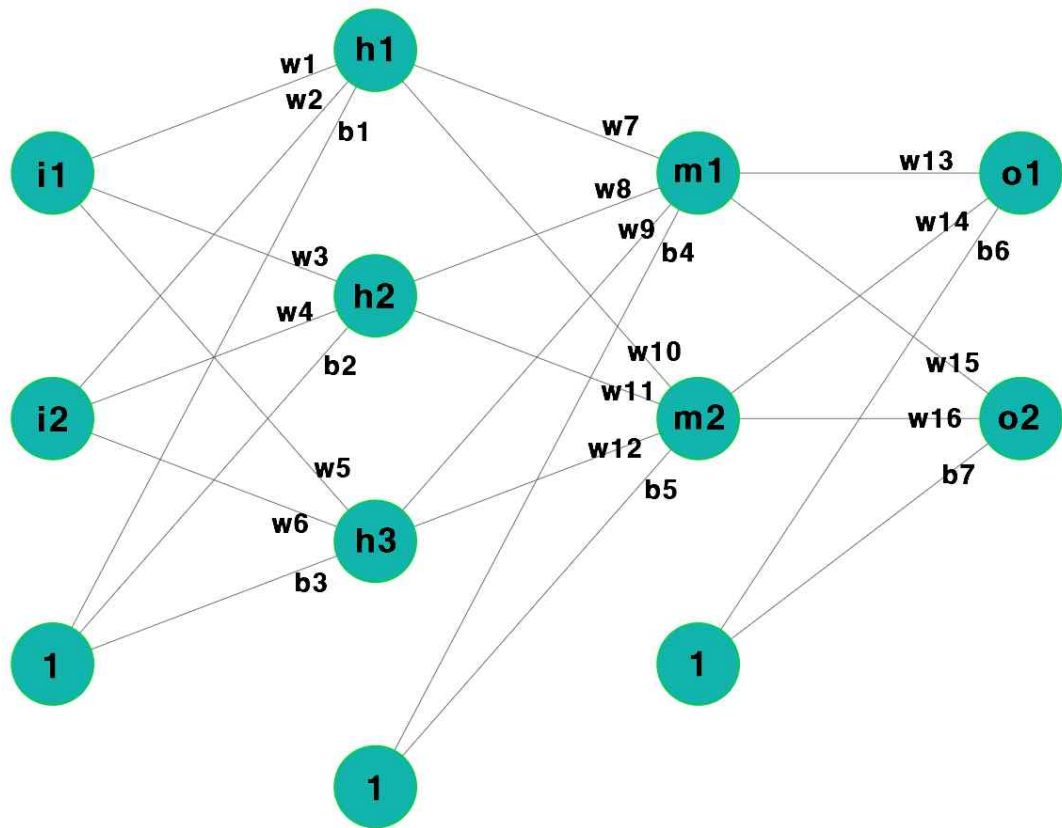
epoch = 664
h1, h2 = 0.239, 0.226
o1, o2 = 0.010, 0.990
E = 0.0000001
w1b, w3b = -0.000, 0.000
w2b, w4b = -0.000, 0.000
b1b, b2b = -0.000, 0.000
w5b, w7b = 0.000, -0.000
w6b, w8b = 0.000, -0.000
b3b, b4b = 0.000, -0.000
w1, w3 = 0.143, 0.242
w2, w4 = 0.186, 0.284
b1, b2 = 0.213, 0.186
w5, w7 = 0.203,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

```

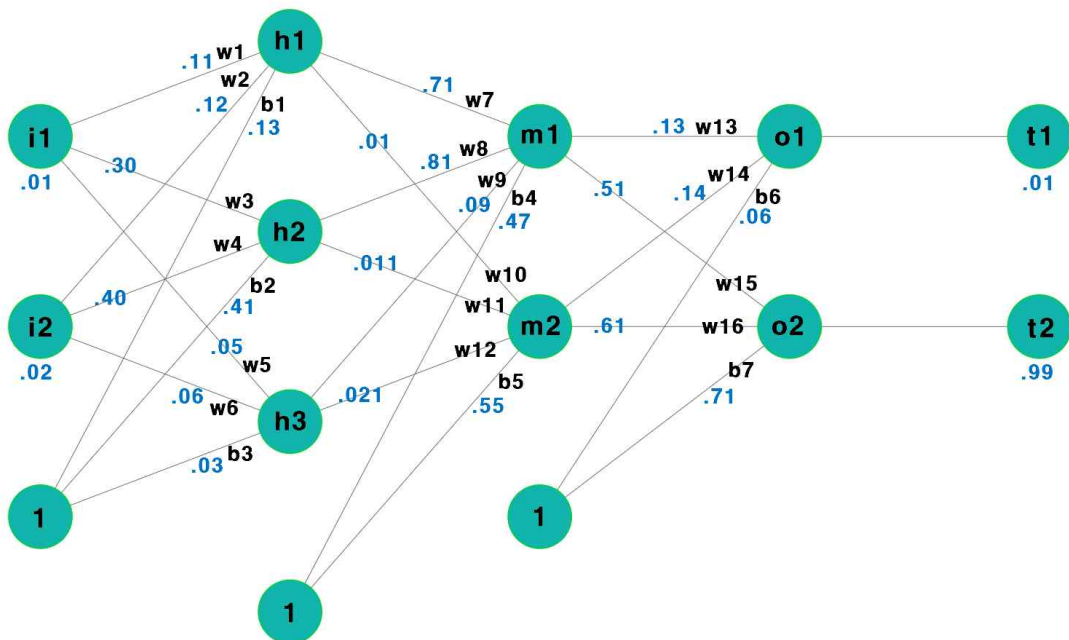
연습문제

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.

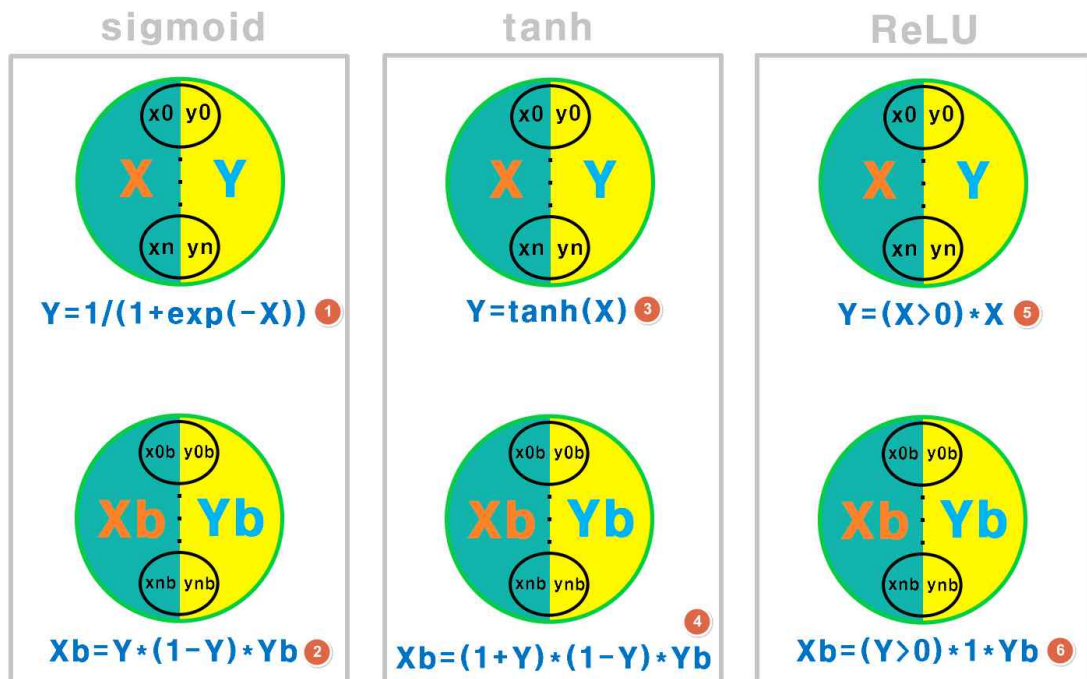


2. 앞에서 구한 행렬 계산식을 이용하여 다음과 같이 초기화된 인공 신경망을 구현하고 학습시켜 봅니다. 입력 값 i_1, i_2 는 상수로 처리합니다.



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

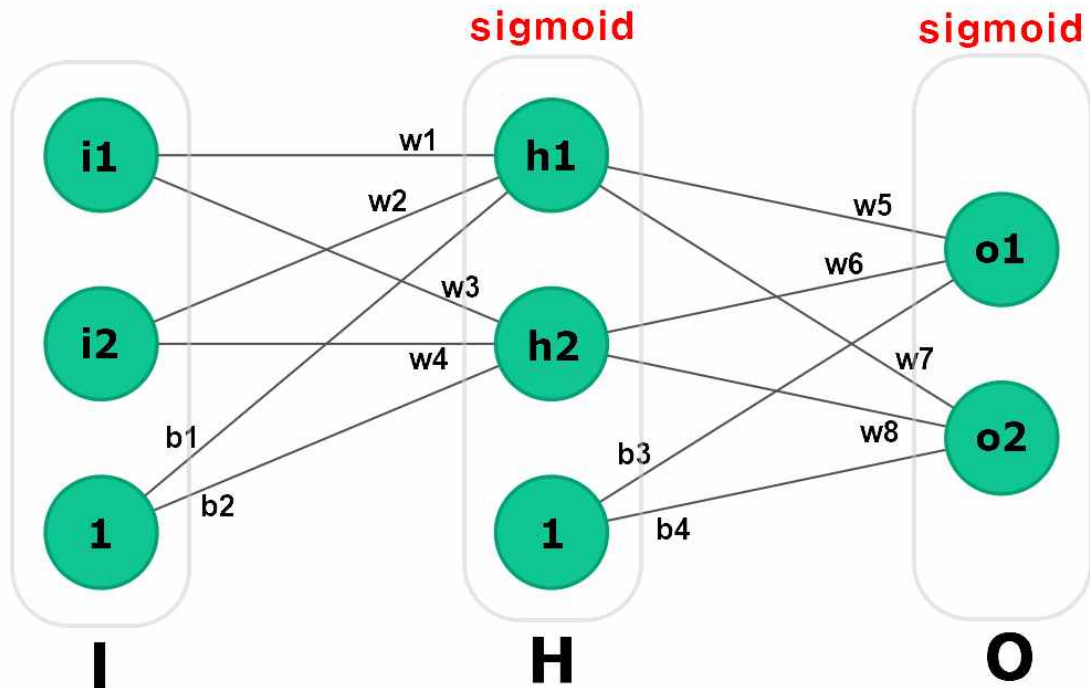
$$Y = \tanh(X) \quad ③ \quad X_b = (1 + Y)(1 - Y) Y_b \quad ④$$

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.
317_1.py

```
01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
04
05 I = np.array([[.05, .10]])
06 T = np.array([[.01, .99]])
07 WH = np.array([[.15, .25],
08               [.20, .30]])
09 BH = np.array([[.35, .35]])
10 WO = np.array([[.40, .50],
11               [.45, .55]])
12 BO = np.array([[.60, .60]])
13
14 for epoch in range(1000):
15
16     print('epoch = %d' %epoch)
```



```

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

```


19 : 은닉 층 H에 순전파 시그모이드 활성화 함수를 적용합니다.

22 : 출력 층 O에 순전파 시그모이드 활성화 함수를 적용합니다.

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

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

27, 41~44, 52~55 : 지우거나 주석 처리합니다.


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

```
epoch = 997
0 =
[[ 0.306  0.844]]
epoch = 998
0 =
[[ 0.306  0.844]]
epoch = 999
0 =
[[ 0.306  0.844]]
```

(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 = 9997
0 =
[[ 0.063  0.943]]
epoch = 9998
0 =
[[ 0.063  0.943]]
epoch = 9999
0 =
[[ 0.063  0.943]]
```

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

6. 다음과 같이 예제를 수정합니다.

```
14 for epoch in range(100000):
```


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

```
epoch = 99997
0 =
[[ 0.020  0.981]]
epoch = 99998
0 =
[[ 0.020  0.981]]
epoch = 99999
0 =
[[ 0.020  0.981]]
```

(99999+1)번째에 o1, o2가 각각 0.020, 0.981이 됩니다.

8. 다음과 같이 예제를 수정합니다.

```
14 for epoch in range(100000):
```

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


```
epoch = 999997
0 =
[[ 0.010  0.990]]
epoch = 999998
0 =
[[ 0.010  0.990]]
epoch = 999999
0 =
[[ 0.010  0.990]]
```

(999999+1)번째에 o1, o2가 각각 0.010, 0.990이 됩니다. 아직 오차는 0.0000001(천만분의 1)보다 큼니다.

10. 다음과 같이 예제를 수정합니다.

317_1.py

```
14 for epoch in range(1000000):
```

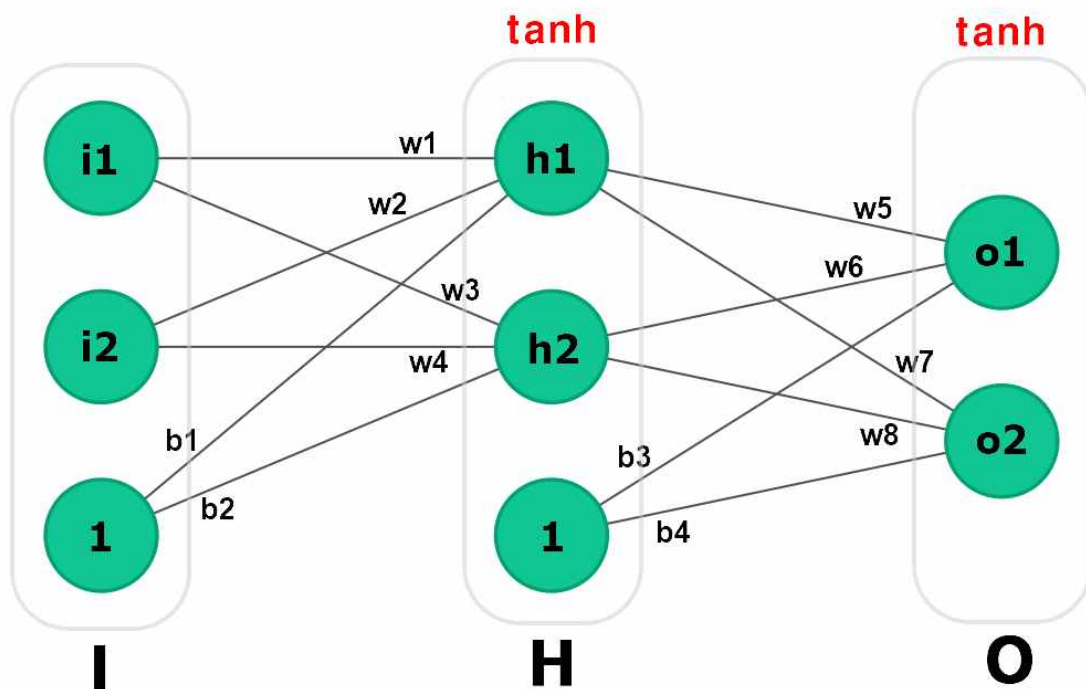
11.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 1078009
0 =
[[ 0.010  0.990]]
epoch = 1078010
0 =
[[ 0.010  0.990]]
epoch = 1078011
0 =
[[ 0.010  0.990]]
```

(1078011+1)번째에 오차가 0.0000001(천만분의 1)보다 작아집니다. o1, o2는 각각 0.010, 0.990이 된 상태입니다.

tanh 함수 적용해 보기

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



1. 이전 예제를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

317_2.py

```
18 H = I @ WH + BH
19 H = np.tanh(H) # ③
20
21 O = H @ WO + BO
22 O = np.tanh(O) # ③
```


19 : 은닉 층 H에 순전파 tanh 활성화 함수를 적용합니다.

22 : 출력 층 O에 순전파 tanh 활성화 함수를 적용합니다.

```
30 Ob = O - T
31 Ob = Ob*(1+O)*(1-O) # ④
32
33 Hb = Ob @ WO.T
34 Hb = Hb*(1+H)*(1-H) # ④
```

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

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

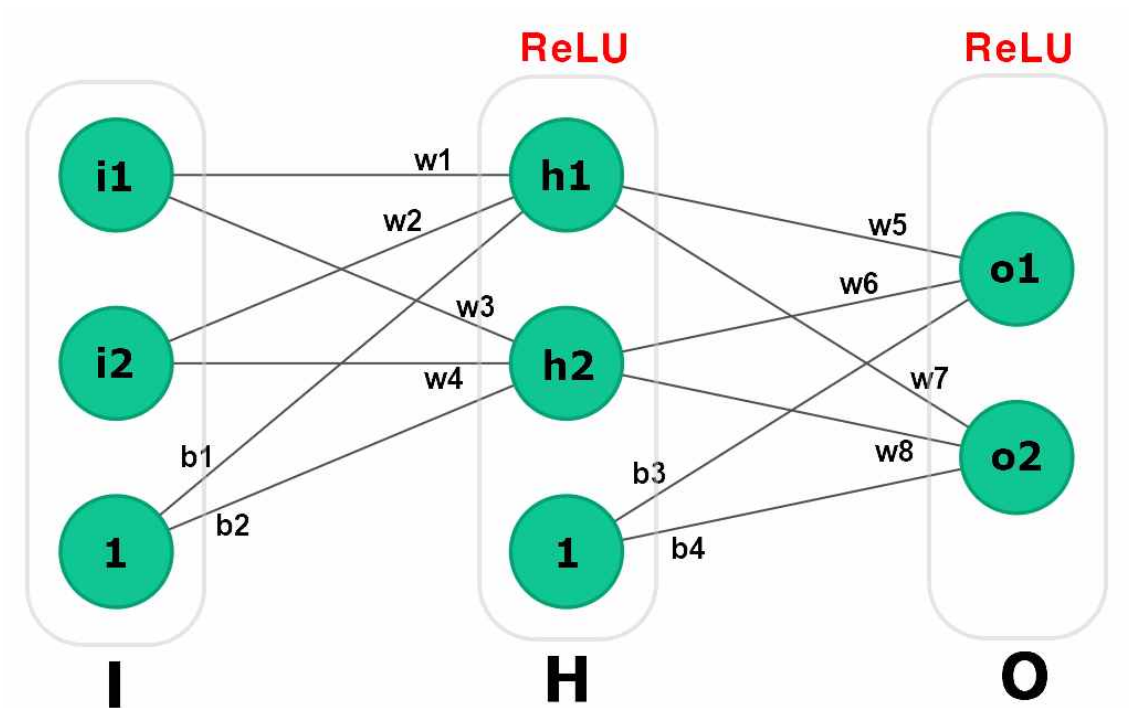
3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 236408
0 =
[[ 0.010  0.990]]
epoch = 236409
0 =
[[ 0.010  0.990]]
epoch = 236410
0 =
[[ 0.010  0.990]]
```

(236410+1)번째에 오차가 0.0000001(천만분의 1)보다 작아집니다. o1, o2는 각각 0.010, 0.990이 된 상태입니다. sigmoid 함수보다 결과가 더 빨리 나오는 것을 볼 수 있습니다.

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.

317_3.py

```

18     H = I @ WH + BH
19     H = (H>0)*H # ③
20
21     O = H @ WO + BO
22     O = (O>0)*O # ③

```

19 : 은닉 층 H에 순전파 ReLU 활성화 함수를 적용합니다.

22 : 출력 층 O에 순전파 ReLU 활성화 함수를 적용합니다.


```

30     Ob = O - T
31     Ob = Ob*(O>0)*1 # ④
32
33     Hb = Ob @ WO.T
34     Hb = Hb*(H>0)*1 # ④

```

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

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

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

```
epoch = 663
0 =
[[ 0.010  0.990]]
epoch = 664
0 =
[[ 0.010  0.990]]
epoch = 665
0 =
[[ 0.010  0.990]]
```

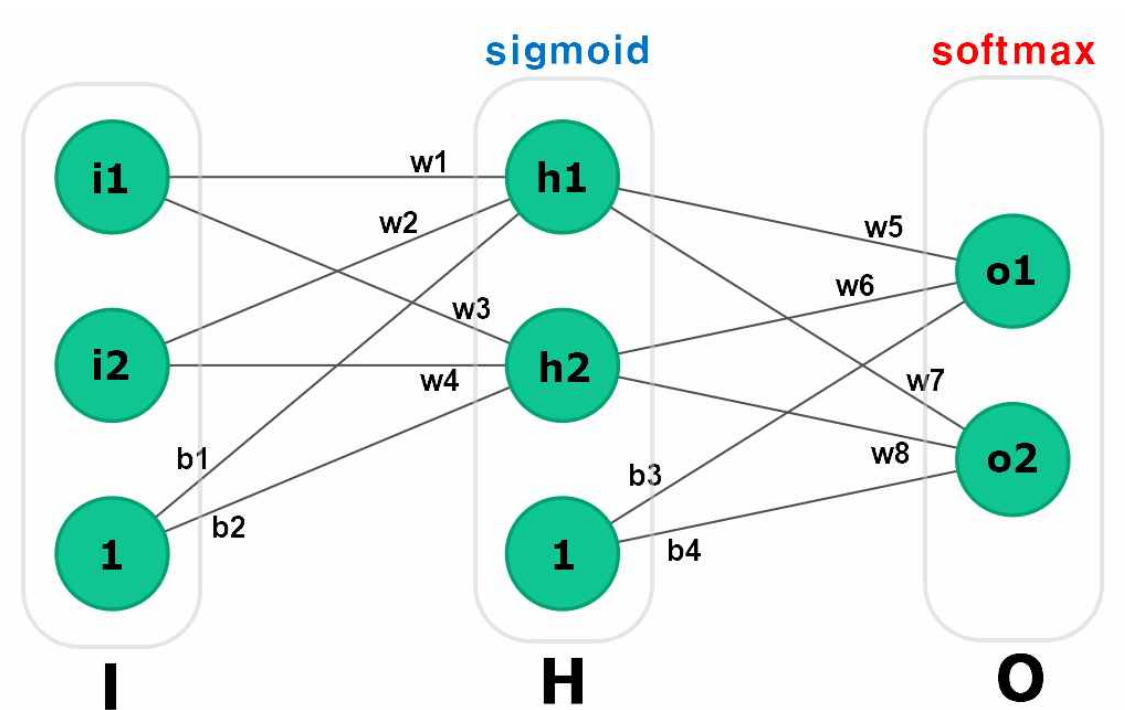
(665+1)번째에 오차가 0.0000001(천만분의 1)보다 작아집니다. o1, o2는 각각 0.010, 0.990이 된 상태입니다. sigmoid, tanh 함수보다 결과가 훨씬 더 빨리 나오는 것을 볼 수 있습니다.

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.

318_1.py

```

01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
04
05 I = np.array([[.05, .10]])
06 T = np.array([[ 0,  1]])
07 WH = np.array([[.15, .25],
08                [.20, .30]])
09 BH = np.array([[.35, .35]])
10 WO = np.array([[.40, .50],
11                [.45, .55]])
12 BO = np.array([[.60, .60]])
13
14 for epoch in range(10000000):
15
16     print('epoch = %d' %epoch)
17
18     H = I @ WH + BH

```



```

19 H = 1/(1+np.exp(-H))
20
21 O = H @ WO + BO
22 OM = O - np.max(O)
23 O = np.exp(OM)/np.sum(np.exp(OM))
24
25 print(' O =\n', O)
26
27 E = np.sum(-T*np.log(O))
28 if E < 0.0001:
29     break
30
31 Ob = O - T
32 # nothing for softmax + cross entropy error
33
34 Hb = Ob @ WO.T
35 Hb = Hb*H*(1-H)
36
37 WHb = I.T @ Hb
38 BHb = 1 * Hb
39 WOb = H.T @ Ob
40 BOb = 1 * Ob
41
42 lr = 0.01
43 WH = WH - lr * WHb
44 BH = BH - lr * BHb
45 WO = WO - lr * WOb
46 BO = BO - lr * BOb

```

06 : 목표 값을 각각 0과 1로 변경합니다.

22, 23 : 출력 층의 활성화 함수를 소프트맥스로 변경합니다.

22 : 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.

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

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


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

31 : 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.

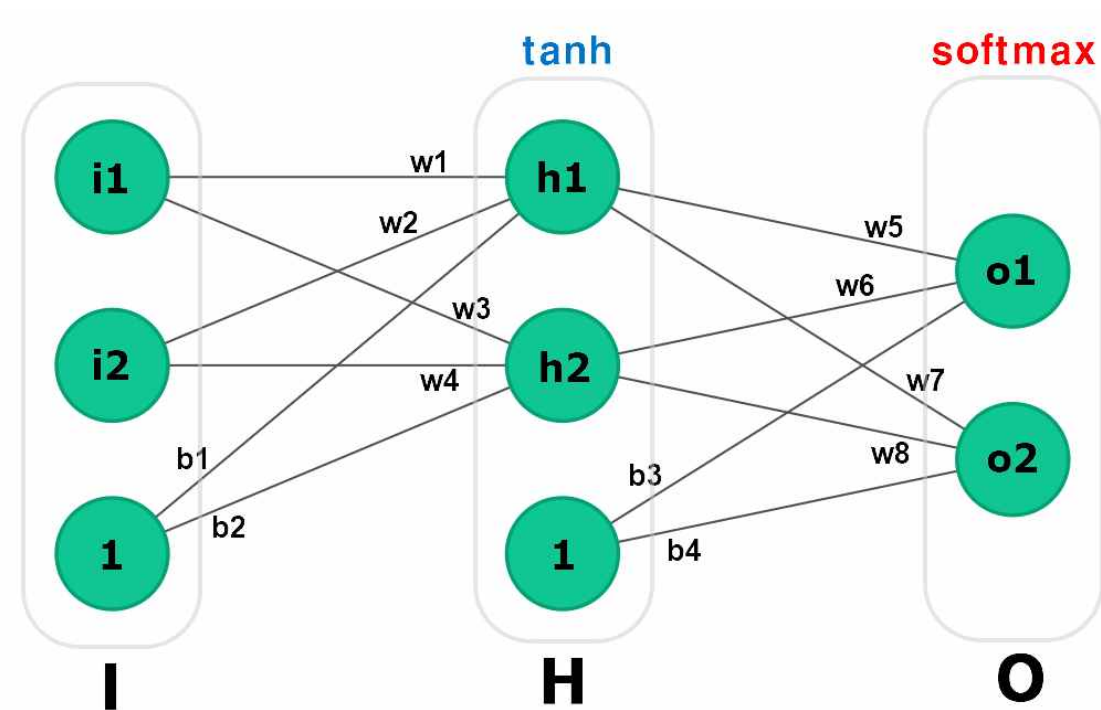
3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 211286
0 =
[[ 0.000  1.000]]
epoch = 211287
0 =
[[ 0.000  1.000]]
epoch = 211288
0 =
[[ 0.000  1.000]]
```

(211288+1)번째에 오차가 0.0001(만분의 1)보다 작아집니다. o1, o2는 각각 0.000, 1.000이 된 상태입니다.

[tanh와 softmax](#)

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



1. 이전 예제를 복사합니다.
2. 다음과 같이 예제를 수정합니다.
318_2.py

```

18     H = I @ WH + BH
19     H = np.tanh(H)
20
21     O = H @ WO + BO
22     OM = O - np.max(O)
23     O = np.exp(OM)/np.sum(np.exp(OM))

```


- 19 : 은닉 층 H에 순전파 tanh 활성화 함수를 적용합니다.
22, 23 : 출력 층의 활성화 함수는 softmax입니다.

```

31     Ob = O - T
32     # nothing for softmax + cross entropy error
33
34     Hb = Ob @ WO.T
35     Hb = Hb*(1+H)*(1-H)

```

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

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

```

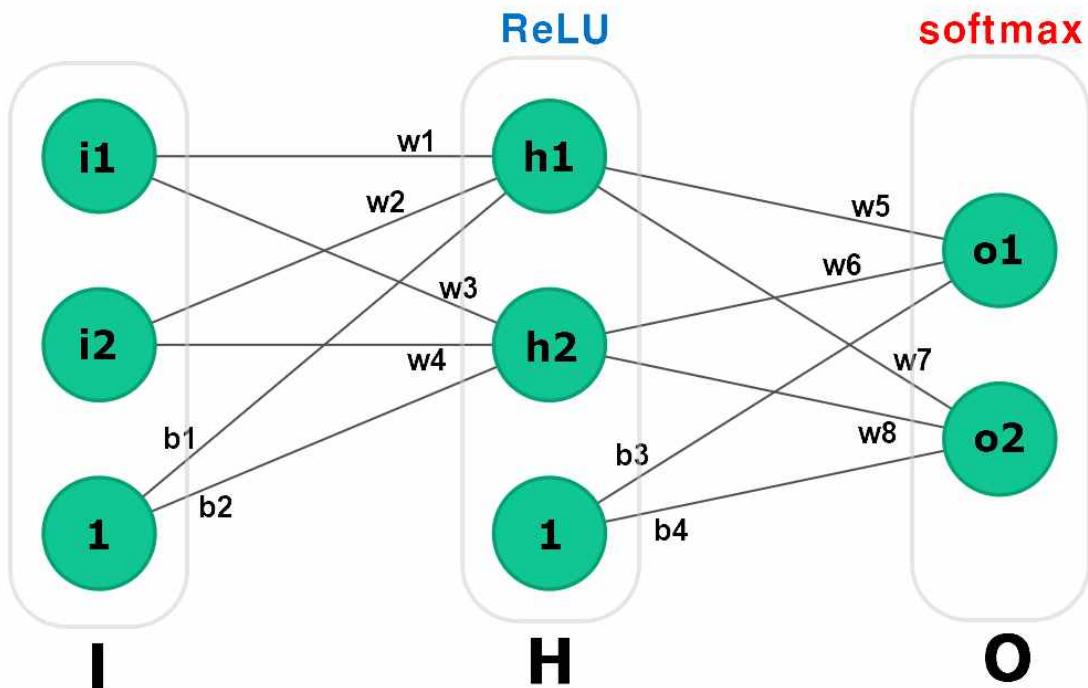
epoch = 174991
0 =
[[ 0.000  1.000]]
epoch = 174992
0 =
[[ 0.000  1.000]]
epoch = 174993
0 =
[[ 0.000  1.000]]

```

(174993+1)번째에 오차가 0.0001(만분의 1)보다 작아집니다. o1, o2는 각각 0.000, 1.000이 된 상태입니다.

ReLU와 softmax

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



1. 이전 예제를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

318_3.py

```
18     H = I @ WH + BH
19     H = (H>0)*H
20
21     O = H @ WO + BO
22     OM = O - np.max(O)
23     O = np.exp(OM)/np.sum(np.exp(OM))
```


19 : 은닉 층 H에 순전파 ReLU 활성화 함수를 적용합니다.

22, 23 : 출력 층의 활성화 함수는 softmax입니다.

```
31     Ob = O - T
32     # nothing for softmax + cross entropy error
33
34     Hb = Ob @ WO.T
35     Hb = Hb*(H>0)*1
```

31 : softmax 함수의 역전파 오차 계산 부분입니다.

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

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

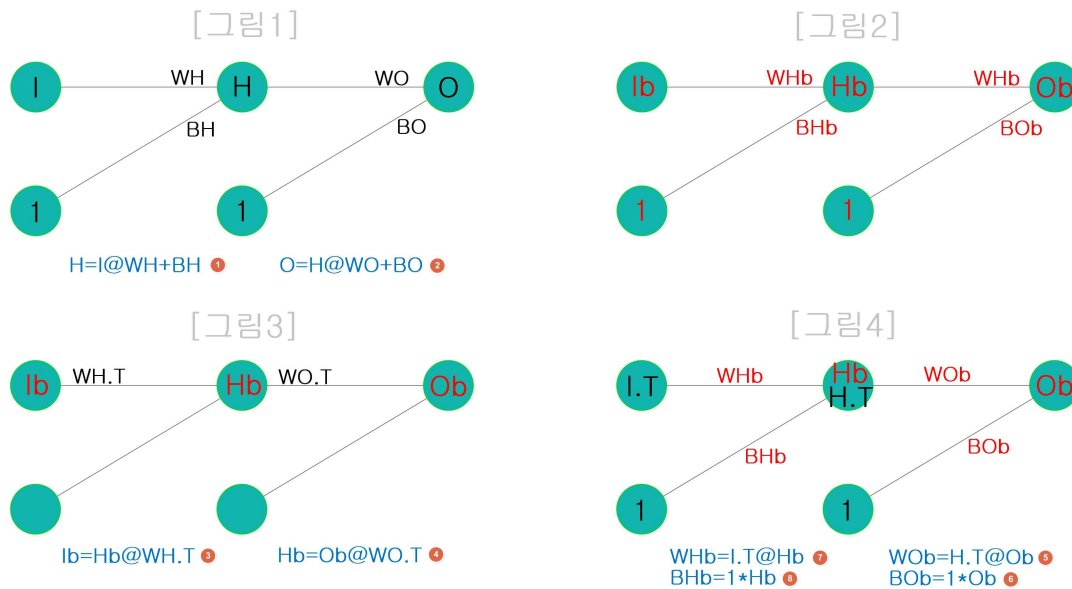
```
epoch = 56959
0 =
[[ 0.000  1.000]]
epoch = 56960
0 =
[[ 0.000  1.000]]
epoch = 56961
0 =
[[ 0.000  1.000]]
```

(56961+1)번째에 오차가 0.0001(만분의 1)보다 작아집니다. o1, o2는 각각 0.000, 1.000이 된 상태입니다.

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.

*** ③ 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

$$H = I @ WH + BH \quad ①$$

$$O = H @ WO + BO \quad ②$$

역전파 오차

$$Ob = O - T$$

입력 역전파

$$Hb = Ob @ WO.T \quad (4)$$

가중치, 편향 역전파

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

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

$$WO b = H.T @ Ob \quad (5)$$

$$BO b = 1 * Ob \quad (6)$$

가중치, 편향 학습

$$WH = WH - lr * WHb$$

$$BH = BH - lr * BHb$$

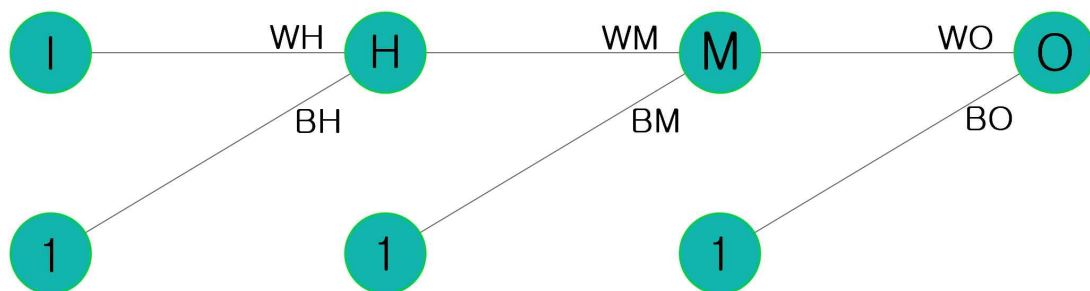
$$WO = WO - lr * WO b$$

$$BO = BO - lr * BO b$$

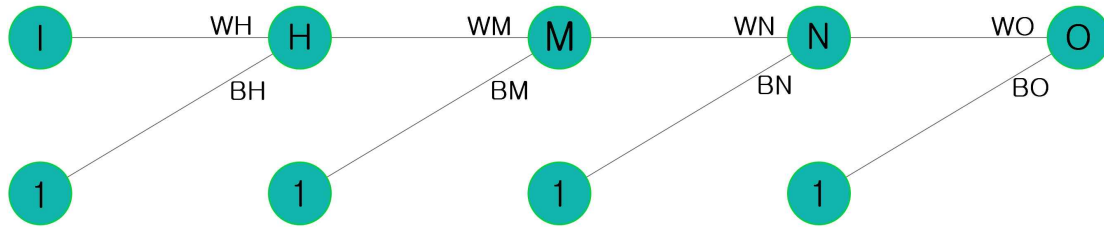
*** lr은 학습률을 나타냅니다.

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.



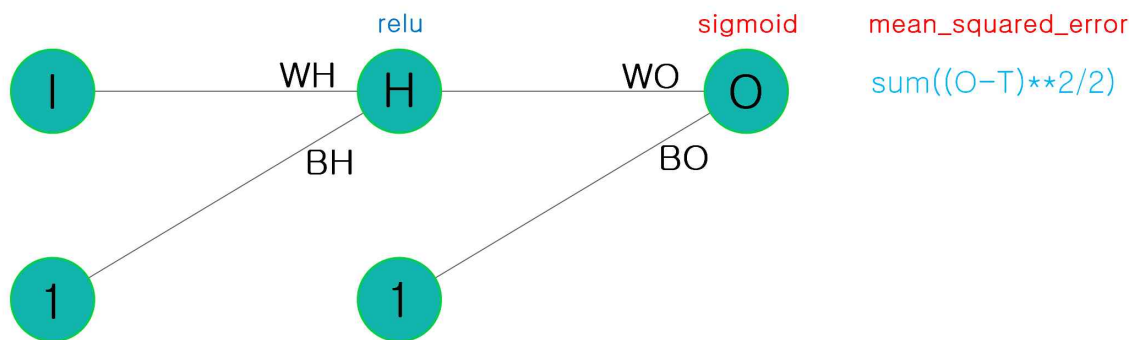
2. 다음은 입력I 은닉H 은닉M 은닉N 출력O의 심층 인공 신경망입니다. 이 신경망에는 3개의 은닉 층이 포함되어 있습니다. 일반적으로 은닉 층이 2층 이상일 경우 심층 인공 신경망이라고 합니다. 이 신경망의 입력 역전파 그래프와 가중치, 편향 역전파 그래프를 그리고 순전파, 역전파 행렬 계산식을 구합니다.



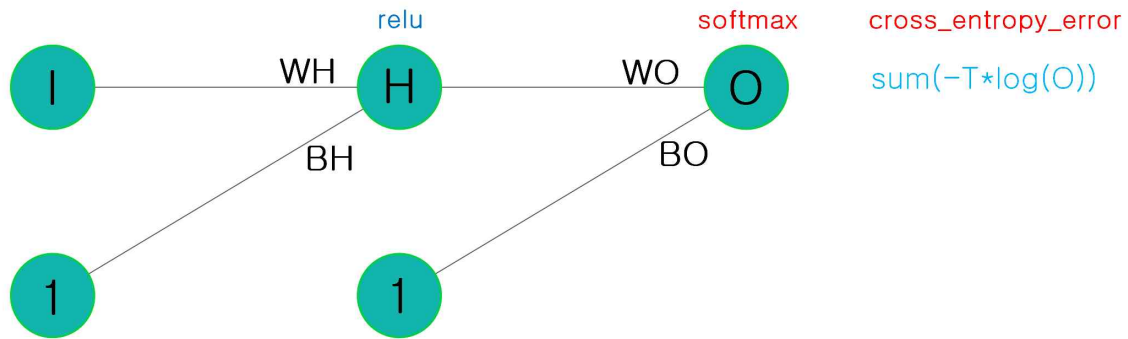
10 가중치 초기화하기

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.

위 수식은 numpy를 이용하면 다음과 같이 표현할 수 있습니다.

```
np.random.randn(INPUT, OUTPUT)/np.sqrt(INPUT/2)
```

np.random.randn(INPUT, OUTPUT)는 가중치로 들어오는 입력노드의 개수 INPUT, 가중치에서 나가는 출력노드의 개수 OUTPUT의 개수만큼 표준 정규 분포를 따르는 임의의 숫자를 생성합니다. np.sqrt(INPUT/2)는 표준편차 부분을 의미합니다.

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.

위 수식은 numpy를 이용하면 다음과 같이 표현할 수 있습니다.

`np.random.randn(INPUT, OUTPUT)/np.sqrt(INPUT)`

`np.random.randn(INPUT, OUTPUT)`는 가중치로 들어오는 입력노드의 개수 `INPUT`, 가중치에서 나가는 출력노드의 개수 `OUTPUT`의 개수만큼 표준 정규 분포를 따르는 임의의 숫자를 생성합니다. `np.sqrt(INPUT)`는 표준편차 부분을 의미합니다.

He와 Lecun 그려보기

여기서는 He와 Lecun 표준정규분포 곡선을 그려봅니다.

1. 다음과 같이 예제를 작성합니다.

3110_1.py


```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 He = np.random.randn(1000000)/np.sqrt(10000/2)
5 Le = np.random.randn(1000000)/np.sqrt(10000)
6
7 plt.hist(He, bins=100, density=True, alpha=0.7)
8 plt.hist(Le, bins=100, density=True, alpha=0.5)
9 plt.show()
```

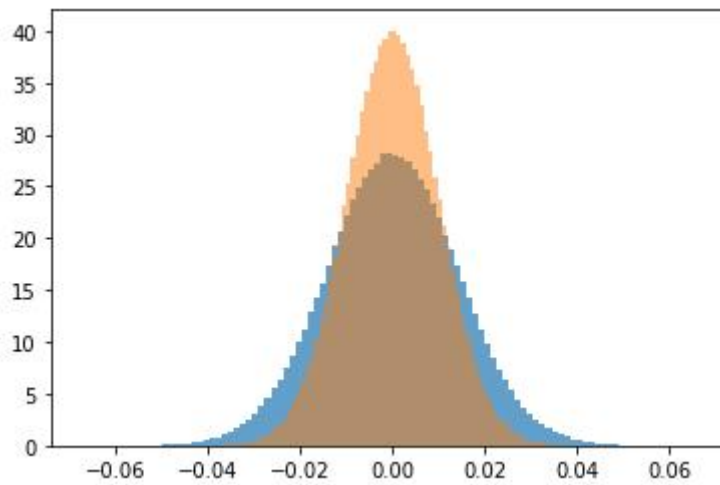
4 : 정규 분포를 갖는 임의의 실수 1000000(백만)개를 생성한 후, 입력의 개수 10000(만)에 대한 He 표준편차로 나누어 He 변수에 할당합니다.

5 : 정규 분포를 갖는 임의의 실수 1000000(백만)개를 생성한 후, 입력의 개수 10000(만)에 대한 Lecun 표준편차로 나누어 Le 변수에 할당합니다.

7 : `plt.hist` 함수를 호출하여 He 표준 정규 분포 곡선을 그립니다. 첫 번째 인자 He는 He 표준 정규 분포 값의 배열입니다. 두 번째 인자 `bins`는 가로축에 들어갈 막대의 개수로 100개로 채웁니다. 값을 10으로 줄여보면 이해하기 쉽습니다. 세 번째 인자 `density`는 `True`로 설정해주면, 밀도함수가 되어서 막대의 아래 면적이 1이 되도록 그립니다. 여기서는 He 분포와 Lecun 분포를 비교하기 쉽게 해 줍니다. 네 번째 인자 `alpha`는 그래프의 투명도를 의미합니다.

8 : `plt.hist` 함수를 호출하여 Lecun 표준 정규 분포 곡선을 그립니다.

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



0에 가까운 값들이 많은 것을 볼 수 있습니다.

*** 가중치의 초기 값은 0 근처에서 임의로 초기화되어야 인공 신경망을 학습하기에 좋습니다.

He와 Lecun 가중치 초기화하기

이제 He와 Lecun으로 가중치를 초기화하여 학습시켜 봅시다.

1. 다음과 같이 예제를 작성합니다.

3110_2.py

```
01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.4f}".format(x)})
04
05 I = np.array([[.05, .10]])
06 T = np.array([[.01, .99]])
07 WH = np.random.randn(2, 2)/np.sqrt(2/2) # He
08 BH = np.zeros((1, 2))
09 WO = np.random.randn(2, 2)/np.sqrt(2) # Lecun
10 BO = np.zeros((1, 2))
11
12 print("WH =\n", WH)
13 print("WO =\n", WO)
14 print()
15
16 for epoch in range(1, 1000001):
17
```

```

18 H = I @ WH + BH
19 H = (H>0)*H # ReLU
20
21 O = H @ WO + BO
22 O = 1/(1+np.exp(-O)) #sigmoid
23
24 E = np.sum((O-T)**2/2) #mean squared error
25
26 if epoch==1 :
27     print("epoch = %d" %epoch)
28     print("Error = %.4f" %E)
29     print("output =", O)
30     print()
31
32 if E<0.0001 :
33     print("epoch = %d" %epoch)
34     print("Error = %.4f" %E)
35     print("output =", O)
36     break
37
38 Ob = O - T
39 Ob = Ob*O*(1-O) #sigmoid
40
41 Hb = Ob @ WO.T
42 Hb = Hb*(H>0)*1 # ReLU
43
44 WHb = I.T @ Hb
45 BHb = 1 * Hb
46 WOb = H.T @ Ob
47 BOb = 1 * Ob
48
49 lr = 0.01
50 WH = WH - lr * WHb
51 BH = BH - lr * BHb
52 WO = WO - lr * WOb
53 BO = BO - lr * BOb

```

07 : He 초기 값을 갖는 2 x 2 행렬을 생성한 후, 가중치 변수 WH에 할당합니다.

08 : 초기 값 0을 갖는 1 x 2 행렬을 생성한 후, 편향 변수 BH에 할당합니다. 일반적으로 편향의 초기 값은 0으로 시작합니다.

09 : Lecun 초기 값을 갖는 2 x 2 행렬을 생성한 후, 가중치 변수 WO에 할당합니다.

10 : 초기 값 0을 갖는 1 x 2 행렬을 생성한 후, 편향 변수 BO에 할당합니다. 일반적으로 편

량의 초기 값은 0으로 시작합니다.

12, 13 : print 함수를 호출하여 WH, WO 값을 출력해 봅니다.

16 : epoch 변수 1에서 1000001(백만일) 미만에 대하여 18~53줄을 수행합니다.

19 : 은닉 층의 활성화 함수를 ReLU로 사용합니다.

22 : 출력 층의 활성화 함수를 sigmoid로 사용합니다.


24 : 오차 계산 함수는 평균 제곱 오차를 사용합니다.

26~30 : epoch값이 1일 때, 즉, 처음 시작할 때, 오차 값과 예측 값을 출력합니다.

32~36 : 오차 값이 0.0001(만분의 일)보다 작을 때, 오차 값과 예측 값을 출력한 후, 16줄의 for문을 나옵니다.

39 : 출력 층의 역 활성화 함수를 sigmoid로 사용합니다.

42 : 은닉 층의 역 활성화 함수를 ReLU로 사용합니다.

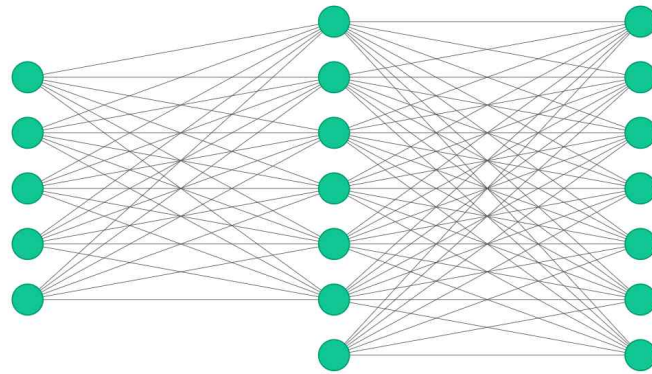
2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
WH =  
[[-0.0988 -1.5852]  
 [-0.3469 0.5997]]  
WO =  
[[-0.2536 -0.7170]  
 [-0.0301 0.3582]]  
  
epoch = 1  
Error = 0.2401  
output = [[0.5000 0.5000]]  
  
epoch = 207889  
Error = 0.0001  
output = [[0.0200 0.9800]]
```

필자의 경우 207889번 학습을 수행하였으며, 오차는 0.0001이고, 첫 번째 항목의 값은 0.02, 두 번째 항목은 0.98입니다. 가중치 초기 값에 따라 독자 여러분의 결과는 다를 수 있습니다.

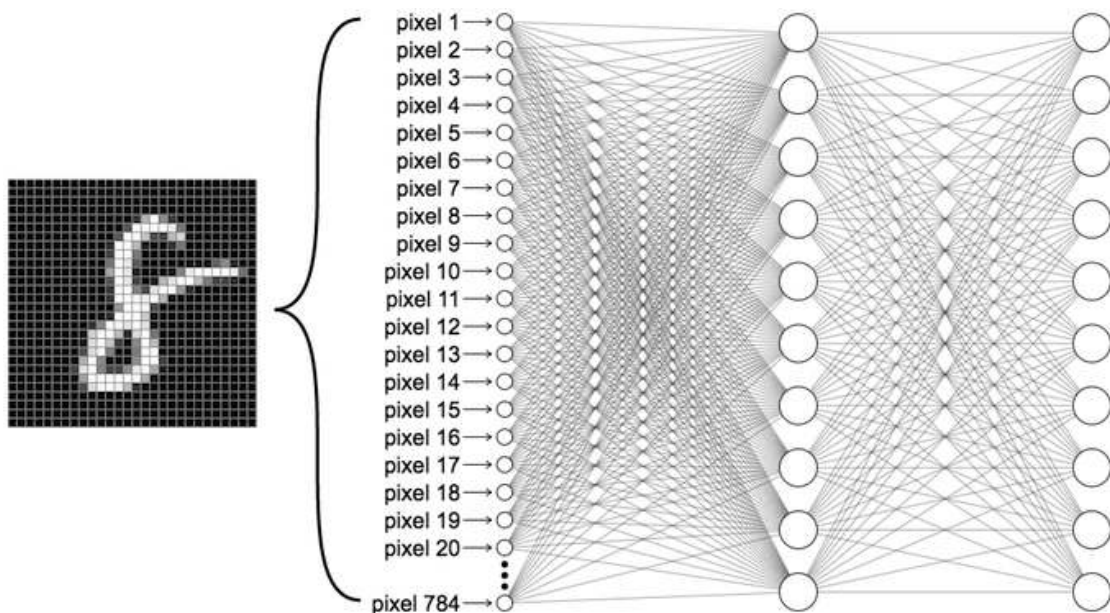
02 NumPy DNN 활용하기

여기서는 지금까지 구현한 NumPy 인공 신경망을 확장해 봅니다. 이 단위에서는 먼저 다음과 같은 형태의 인공 신경망을 구성해서 테스트해 봅니다.



<4개의 입력, 6개의 은닉 층, 7개의 출력 층, 편향 포함>

또, 1장에서 tensorflow 라이브러리를 이용하여 살펴보았던 다음과 같은 형태의 인공 신경망도 구성해서 테스트해 봅니다.



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 2진

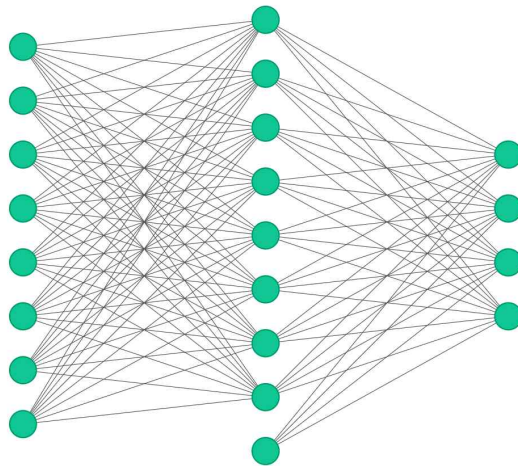
수 연결 진리표입니다.

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

 = 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. 다음과 같이 예제를 작성합니다.

321_1.py

```
01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.4f}".format(x)})
04
05 NUM_PATTERN = 10
06 NUM_IN = 7
07 NUM_HID = 8
```

```

08 NUM_OUT = 4
09
10 I = np.array([
11     [[ 1, 1, 1, 1, 1, 1, 0 ]], # 0
12     [[ 0, 1, 1, 0, 0, 0, 0 ]], # 1
13     [[ 1, 1, 0, 1, 1, 0, 1 ]], # 2
14     [[ 1, 1, 1, 1, 0, 0, 1 ]], # 3
15     [[ 0, 1, 1, 0, 0, 1, 1 ]], # 4
16     [[ 1, 0, 1, 1, 0, 1, 1 ]], # 5
17     [[ 0, 0, 1, 1, 1, 1, 1 ]], # 6
18     [[ 1, 1, 1, 0, 0, 0, 0 ]], # 7
19     [[ 1, 1, 1, 1, 1, 1, 1 ]], # 8
20     [[ 1, 1, 1, 0, 0, 1, 1 ]], # 9
21 ])
22 T = np.array([
23     [[ 0, 0, 0, 0 ]],
24     [[ 0, 0, 0, 1 ]],
25     [[ 0, 0, 1, 0 ]],
26     [[ 0, 0, 1, 1 ]],
27     [[ 0, 1, 0, 0 ]],
28     [[ 0, 1, 0, 1 ]],
29     [[ 0, 1, 1, 0 ]],
30     [[ 0, 1, 1, 1 ]],
31     [[ 1, 0, 0, 0 ]],
32     [[ 1, 0, 0, 1 ]],
33 ])
34 O = np.zeros((NUM_PATTERN, 1, NUM_OUT))
35 WH = np.random.randn(NUM_IN, NUM_HID)/np.sqrt(NUM_IN/2) # He
36 BH = np.zeros((1, NUM_HID))
37 WO = np.random.randn(NUM_HID, NUM_OUT)/np.sqrt(NUM_HID) # Lecun
38 BO = np.zeros((1, NUM_OUT))
39
40 for epoch in range(1, 1000001):
41
42     H = I[2] @ WH + BH
43     H = (H>0)*H # ReLU
44
45     O[2] = H @ WO + BO
46     O[2] = 1/(1+np.exp(-O[2])) #sigmoid
47

```



```

48     E = np.sum((O[2]-T[2])**2/2) #mean squared error
49
50     if epoch==1 :
51         print("epoch  = %d" %epoch)
52         print("Error  = %.4f" %E)
53         print("output =", O[2])
54         print()
55
56     if E<0.0001 :
57         print("epoch  = %d" %epoch)
58         print("Error  = %.4f" %E)
59         print("output =", O[2])
60         break
61
62     Ob = O[2] - T[2]
63     Ob = Ob*O[2]*(1-O[2]) #sigmoid
64
65     Hb = Ob @ WO.T
66     Hb = Hb*(H>0)*1 # ReLU
67
68     WHb = I[2].T @ Hb
69     BHb = 1 * Hb
70     WOb = H.T @ Ob
71     BOb = 1 * Ob
72
73     lr = 0.01
74     WH = WH - lr * WHb
75     BH = BH - lr * BHb
76     WO = WO - lr * WOb
77     BO = BO - lr * BOb

```

05 : 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

5 = 1011011 → 0101

06 : NUM_IN 변수를 선언한 후, 7로 초기화합니다.

07 : NUM_HID 변수를 선언한 후, 8로 초기화합니다.

08 : NUM_OUT 변수를 선언한 후, 4로 초기화합니다.

10~21 : 입력 I 행렬을 3차 행렬로 변경하고 진리표의 입력 값에 맞게 값을 초기화합니다. 입력 I 행렬의 모양은 (10, 1, 7)입니다.

22~33 : 목표 T 행렬을 3차 행렬로 변경하고 진리표의 출력 값에 맞게 값을 초기화합니다. 목표 T 행렬의 모양은 (10, 1, 4)입니다.


34 : 출력 O 행렬을 (NUM_PATTERN, 1, NUM_OUT) 모양의 행렬로 변경합니다. 출력 O 행렬의 모양은 (10, 1, 4)입니다.

35~38 : 가중치와 편향 행렬의 모양을 위 그림에 맞게 변경합니다.

42, 68 : I을 I[2]로 변경합니다. I 행렬의 2번 항목을 입력 값으로 학습 테스트를 수행합니다.

48, 62 : T을 T[2]로 변경합니다. T 행렬의 2번 항목을 목표 값으로 학습 테스트를 수행합니다.

45, 46, 48, 53, 59, 62, 63 : O을 O[2]로 변경합니다. O 행렬의 2번 항목을 예측 값으로 학습 테스트를 수행합니다. 46, 63 줄의 경우 2 군데씩 수정합니다.

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

epoch = 1

Error = 0.5003

output = [[0.4955 0.5135 0.4960 0.4873]]

epoch = 27818


Error = 0.0001

output = [[0.0070 0.0073 0.9927 0.0066]]

필자의 경우 27818번 학습을 수행하였으며, 오차는 0.0001이고, 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

NumPy 행렬 모양 살펴보기

우리는 앞에서 I, T, O 행렬을 변경하였습니다. 여기서는 변경된 행렬과 변경되기 전 행렬의 모양을 살펴봅니다.

1. 다음과 같이 예제를 작성합니다.


321_2.py

```
01 import numpy as np
02
03 I = np.array([
04     [[ 1, 1, 1, 1, 1, 1, 0 ]], # 0
05     [[ 0, 1, 1, 0, 0, 0, 0 ]], # 1
06     [[ 1, 1, 0, 1, 1, 0, 1 ]], # 2
07     [[ 1, 1, 1, 1, 0, 0, 1 ]], # 3
08     [[ 0, 1, 1, 0, 0, 1, 1 ]], # 4
09     [[ 1, 0, 1, 1, 0, 1, 1 ]], # 5
10     [[ 0, 0, 1, 1, 1, 1, 1 ]], # 6
11     [[ 1, 1, 1, 0, 0, 0, 0 ]], # 7
12     [[ 1, 1, 1, 1, 1, 1, 1 ]], # 8
13     [[ 1, 1, 1, 0, 0, 1, 1 ]], # 9
14 ])
15 T = np.array([
16     [[ 0, 0, 0, 0 ]],
```

```

17     [[ 0, 0, 0, 1 ]],
18     [[ 0, 0, 1, 0 ]],
19     [[ 0, 0, 1, 1 ]],
20     [[ 0, 1, 0, 0 ]],
21     [[ 0, 1, 0, 1 ]],
22     [[ 0, 1, 1, 0 ]],
23     [[ 0, 1, 1, 1 ]],
24     [[ 1, 0, 0, 0 ]],
25     [[ 1, 0, 0, 1 ]]
26 ])
27
28 print("I.shape={}".format(I.shape))
29 print("T.shape={}".format(T.shape))
30
31 print("I[2]: {}, shape={}".format(I[2], I[2].shape))
32 print("T[2]: {}, shape={}".format(T[2], T[2].shape))
33
34 I_0 = np.array([[.05, .10]])
35 T_0 = np.array([[0, 1]])
36
37 print("I_0: {}, shape={}".format(I_0, I_0.shape))
38 print("T_0: {}, shape={}".format(T_0, T_0.shape))

```

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

```

I.shape=(10, 1, 7)
T.shape=(10, 1, 4)
I[2]: [[1 1 0 1 1 0 1]], shape=(1, 7)
T[2]: [[0 0 1 0]], shape=(1, 4)
I_0: [[0.05 0.1 ]], shape=(1, 2)
T_0: [[0 1]], shape=(1, 2)

```

I와 T의 모양은 각각 (10, 1, 7), (10, 1, 4)입니다. I[2], T[2]의 모양은 각각 (1, 7), (1, 4)입니다. I_0, T_0의 모양은 각각 (1, 2), (1, 2)입니다.

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

1. 다음과 같이 예제를 작성합니다.

322_1.py

```
01 import numpy as np
02
03 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.4f}".format(x)})
04
05 NUM_PATTERN = 10
06 NUM_IN = 7
07 NUM_HID = 8
08 NUM_OUT = 4
09
10 I = np.array([
11     [[ 1, 1, 1, 1, 1, 1, 0 ]], # 0
12     [[ 0, 1, 1, 0, 0, 0, 0 ]], # 1
13     [[ 1, 1, 0, 1, 1, 0, 1 ]], # 2
14     [[ 1, 1, 1, 1, 0, 0, 1 ]], # 3
15     [[ 0, 1, 1, 0, 0, 1, 1 ]], # 4
16     [[ 1, 0, 1, 1, 0, 1, 1 ]], # 5
17     [[ 0, 0, 1, 1, 1, 1, 1 ]], # 6
18     [[ 1, 1, 1, 0, 0, 0, 0 ]], # 7
19     [[ 1, 1, 1, 1, 1, 1, 1 ]], # 8
20     [[ 1, 1, 1, 0, 0, 1, 1 ]], # 9
21 ])
22 T = np.array([
23     [[ 0, 0, 0, 0 ]],
```

```

24     [[ 0, 0, 0, 1 ]],
25     [[ 0, 0, 1, 0 ]],
26     [[ 0, 0, 1, 1 ]],
27     [[ 0, 1, 0, 0 ]],
28     [[ 0, 1, 0, 1 ]],
29     [[ 0, 1, 1, 0 ]],
30     [[ 0, 1, 1, 1 ]],
31     [[ 1, 0, 0, 0 ]],
32     [[ 1, 0, 0, 1 ]]
33 ])
34 O = np.zeros((NUM_PATTERN, 1, NUM_OUT))
35 WH = np.random.randn(NUM_IN, NUM_HID)/np.sqrt(NUM_IN/2) # He
36 BH = np.zeros((1, NUM_HID))
37 WO = np.random.randn(NUM_HID, NUM_OUT)/np.sqrt(NUM_HID) # Lecun
38 BO = np.zeros((1, NUM_OUT))
39
40 for epoch in range(1, 10001):
41
42     for pc in range(NUM_PATTERN) :
43
44         H = I[pc] @ WH + BH
45         H = (H>0)*H # ReLU
46
47         O[pc] = H @ WO + BO
48         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
49
50         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
51
52         Ob = O[pc] - T[pc]
53         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
54
55         Hb = Ob @ WO.T
56         Hb = Hb*(H>0)*1 # ReLU
57
58         WHb = I[pc].T @ Hb
59         BHb = 1 * Hb
60         WOb = H.T @ Ob
61         BOb = 1 * Ob
62
63         lr = 0.01

```

```

64         WH = WH - lr * WHb
65         BH = BH - lr * BHb
66         WO = WO - lr * WO b
67         BO = BO - lr * BO b
68
69     if epoch%100==0 :
70         print(".", end='', flush=True)
71
72     print()
73
74     for pc in range(NUM_PATTERN) :
75         print("target %d : "%pc, end='')
76         for node in range(NUM_OUT) :
77             print("%.0f "%T[pc][0][node], end='')
78         print("pattern %d : "%pc, end='');
79         for node in range(NUM_OUT) :
80             print("%.2f "%O[pc][0][node], end='')
81         print()

```

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

42 : pc 변수 0에서 NUM_PATTERN 미만에 대하여 44~67줄을 수행합니다.


44, 47, 48, 50, 52, 53, 58 : 숫자 2를 pc로 변경합니다. 48, 50, 52, 53줄은 두 군데 변경합니다.

50~52 : if 조건문 2개를 없앱니다.

69 : epoch값이 100의 배수가 될 때마다 인공 신경망에 대해 학습하고 있다는 표시를 위해 점 하나를 출력합니다.

72 : 개 행 문자를 출력합니다.

74~81 : 학습이 끝난 후에 목표 값과 예측 값을 출력하여 비교합니다.

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

```

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

```

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.

1. 이전 예제를 복사합니다.
2. 다음과 같이 파일을 수정합니다.

323_1.py

```

001 import numpy as np
002 import random
003 import time
004
005 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.4f}".format(x)})
006
007 NUM_PATTERN = 10
008 NUM_IN = 7
009 NUM_HID = 8
010 NUM_OUT = 4
011
012 I = np.array([
013     [[ 1, 1, 1, 1, 1, 1, 0 ]], # 0
014     [[ 0, 1, 1, 0, 0, 0, 0 ]], # 1

```



```

015     [[ 1, 1, 0, 1, 1, 0, 1 ]], # 2
016     [[ 1, 1, 1, 1, 0, 0, 1 ]], # 3
017     [[ 0, 1, 1, 0, 0, 1, 1 ]], # 4
018     [[ 1, 0, 1, 1, 0, 1, 1 ]], # 5
019     [[ 0, 0, 1, 1, 1, 1, 1 ]], # 6
020     [[ 1, 1, 1, 0, 0, 0, 0 ]], # 7
021     [[ 1, 1, 1, 1, 1, 1, 1 ]], # 8
022     [[ 1, 1, 1, 0, 0, 1, 1 ]], # 9
023 ])
024 T = np.array([
025     [[ 0, 0, 0, 0 ]],
026     [[ 0, 0, 0, 1 ]],
027     [[ 0, 0, 1, 0 ]],
028     [[ 0, 0, 1, 1 ]],
029     [[ 0, 1, 0, 0 ]],
030     [[ 0, 1, 0, 1 ]],
031     [[ 0, 1, 1, 0 ]],
032     [[ 0, 1, 1, 1 ]],
033     [[ 1, 0, 0, 0 ]],
034     [[ 1, 0, 0, 1 ]],
035 ])
036 O = np.zeros((NUM_PATTERN, 1, NUM_OUT))
037 WH = np.random.randn(NUM_IN, NUM_HID)/np.sqrt(NUM_IN/2) # He
038 BH = np.zeros((1, NUM_HID))
039 WO = np.random.randn(NUM_HID, NUM_OUT)/np.sqrt(NUM_HID) # Lecun
040 BO = np.zeros((1, NUM_OUT))
041
042 shuffled_pattern = [pc for pc in range(NUM_PATTERN)] #정수로!
043
044 random.seed(int(time.time()))
045
046 for epoch in range(1, 10001):
047
048     tmp_a = 0;
049     tmp_b = 0;
050     for pc in range(NUM_PATTERN) :
051         tmp_a = random.randrange(0, NUM_PATTERN)
052         tmp_b = shuffled_pattern[pc]
053         shuffled_pattern[pc] = shuffled_pattern[tmp_a]
054         shuffled_pattern[tmp_a] = tmp_b

```

```

055
056     sumError = 0.
057
058     for rc in range(NUM_PATTERN) :
059
060         pc = shuffled_pattern[rc]
061
062         H = I[pc] @ WH + BH
063         H = (H>0)*H # ReLU
064
065         O[pc] = H @ WO + BO
066         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
067
068         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
069
070         sumError += E
071
072         Ob = O[pc] - T[pc]
073         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
074
075         Hb = Ob @ WO.T
076         Hb = Hb*(H>0)*1 # ReLU
077
078         WHb = I[pc].T @ Hb
079         BHb = 1 * Hb
080         WOb = H.T @ Ob
081         BOb = 1 * Ob
082
083         lr = 0.01
084         WH = WH - lr * WHb
085         BH = BH - lr * BHb
086         WO = WO - lr * WOb
087         BO = BO - lr * BOb
088
089     if epoch%100==0 :
090         print("epoch : %5d, sum error : %f" %(epoch, sumError))
091         for i in range(NUM_IN) :
092             for j in range(NUM_HID) :
093                 print("%7.3f "%WH[i][j], end='')
094                 print(flush=True)

```

```

095
096     if sumError<0.0001 : break
097
098 print()
099
100 for pc in range(NUM_PATTERN) :
101     print("target %d : "%pc, end='')
102     for node in range(NUM_OUT) :
103         print("%.0f "%T[pc][0][node], end='')
104     print("pattern %d : "%pc, end='');
105     for node in range(NUM_OUT) :
106         print("%.2f "%O[pc][0][node], end='')
107     print()

```

002 : random 모듈을 불러옵니다. 047, 057 번째 줄에서 임의 숫자를 생성하기 위해 사용합니다.

003 : time 모듈을 불러옵니다. 047 번째 줄에서 임의 숫자 생성기 모듈을 초기화하기 위해 사용합니다.

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

044 : random.seed 함수를 호출하여 임의 숫자 생성기 모듈을 초기화합니다. time.time() 함수를 호출하여 현재 시간을 정수 값으로 변환하여 입력 값으로 줍니다.

048, 049 : 입력 데이터의 순서를 변경하기 위해 사용할 정수 변수 2개를 선언합니다.

050 : pc 변수에 대해 0에서 NUM_PATTERN 미만에 대하여 057~060줄을 수행합니다.

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

052~054 : shuffled_pattern의 tmp_a 번째 항목과 pc 번째 항목을 서로 바꿔줍니다.

056 : sumError 변수를 선언한 후, 0.0으로 초기화해줍니다.

058 : 이전 예제에서 pc를 rc로 변경해 줍니다.

060 : shuffled_pattern의 rc 번째 항목을 pc로 가져옵니다.


062~068 : 이전 예제와 같습니다.

070 : 068줄에서 얻은 오차 값을 sumError에 더해줍니다.

091~094 : 현재까지 학습된 가중치 WH 값을 출력해 봅니다.

096 : sumError 값이 0.0001보다 작으면 052줄의 for 문을 빠져 나와 98줄로 이동합니다.

100~107 : 이전 예제와 같습니다. 목표 값과 예측 값을 출력합니다.

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

epoch : 10000, sum error : 0.011086

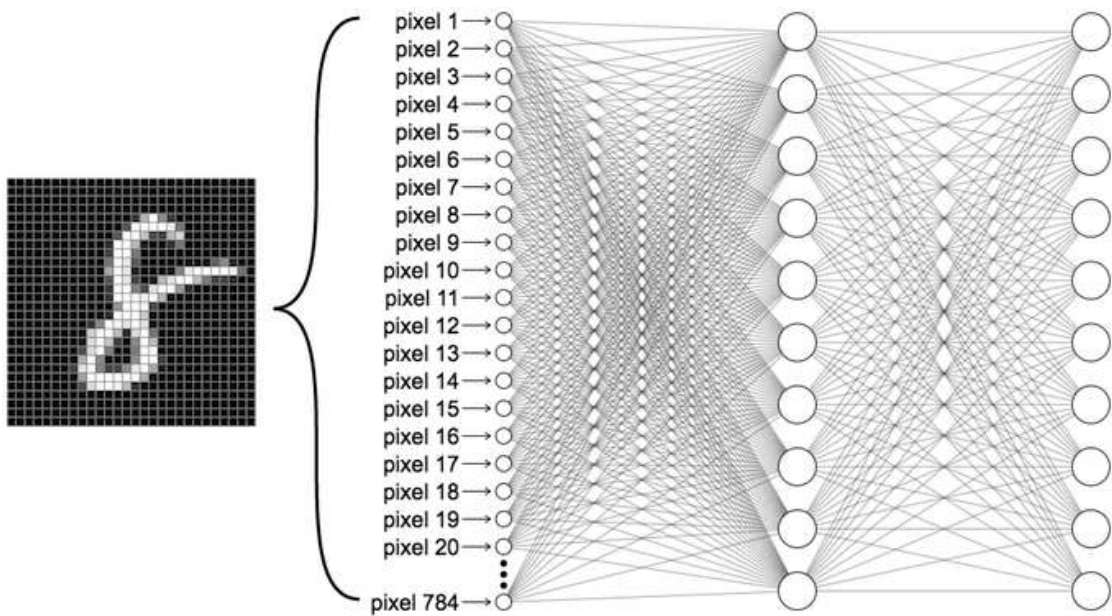
-2.986	-1.803	0.339	1.216	-0.381	2.319	-0.483	0.334
-2.181	0.706	0.455	-0.330	1.298	0.126	-0.575	0.397
1.470	0.659	-1.162	-0.030	0.817	0.770	-0.108	1.086
0.366	0.927	0.758	-0.424	0.659	0.420	-0.214	-1.159
-0.367	-0.790	1.629	-1.417	0.063	-2.181	-0.538	-0.484
0.753	-1.168	-1.357	1.844	0.955	-2.360	-0.044	-1.158
2.234	-0.635	1.053	2.195	0.140	0.329	-0.263	-0.896

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

여기서는 NumPy 인공 신경망으로 MNIST 데이터를 학습시키기 위해 입력 층, 은닉 층, 출력 층의 개수를 늘려봅니다. 다음 그림과 같이 학습할 데이터의 개수는 6만개, 입력 층의 개수는 784개, 은닉 층의 개수는 64개, 출력 층의 개수는 10개로 수정한 후, 임의의 입력 데이터를 생성한 후 1회 학습 시간을 측정해 봅니다.



1. 이전 예제를 복사합니다.
 2. 다음과 같이 파일을 수정합니다.
- 324_1.py

```

01 import numpy as np
02 import random
03 import time
04
05 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.4f}".format(x)})
06
07 NUM_PATTERN = 60000
08 NUM_IN = 784
09 NUM_HID = 64
10 NUM_OUT = 10
11
12 I = np.random.randn(NUM_PATTERN, 1, NUM_IN)
13 T = np.random.randn(NUM_PATTERN, 1, NUM_OUT)
14 O = np.zeros((NUM_PATTERN, 1, NUM_OUT))
15 WH = np.random.randn(NUM_IN, NUM_HID)/np.sqrt(NUM_IN/2) # He
16 BH = np.zeros((1, NUM_HID))
17 WO = np.random.randn(NUM_HID, NUM_OUT)/np.sqrt(NUM_HID) # Lecun
18 BO = np.zeros((1, NUM_OUT))
19
20 shuffled_pattern = [pc for pc in range(NUM_PATTERN)] #정수로!

```

```

21
22 random.seed(int(time.time()))
23
24 begin = time.time()
25
26 for epoch in range(1, 2):
27
28     tmp_a = 0;
29     tmp_b = 0;
30     for pc in range(NUM_PATTERN) :
31         tmp_a = random.randrange(0, NUM_PATTERN)
32         tmp_b = shuffled_pattern[pc]
33         shuffled_pattern[pc] = shuffled_pattern[tmp_a]
34         shuffled_pattern[tmp_a] = tmp_b
35
36     sumError = 0.
37
38     for rc in range(NUM_PATTERN) :
39
40         pc = shuffled_pattern[rc]
41
42         H = I[pc] @ WH + BH
43         H = (H>0)*H # ReLU
44
45         O[pc] = H @ WO + BO
46         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
47
48         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
49
50         sumError += E
51
52         Ob = O[pc] - T[pc]
53         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
54
55         Hb = Ob @ WO.T
56         Hb = Hb*(H>0)*1 # ReLU
57
58         WHb = I[pc].T @ Hb
59         BHb = 1 * Hb
60         WOb = H.T @ Ob

```

```

61         BOb = 1 * Ob
62
63         lr = 0.01
64         WH = WH - lr * WHb
65         BH = BH - lr * BHb
66         WO = WO - lr * WOb
67         BO = BO - lr * BOb
68
69         if rc%1000==999 :
70             print(".", end='', flush=True)
71
72 end = time.time()
73
74 time_taken = end - begin
75
76 print("\nTime taken (in seconds) = {}".format(time_taken))

```

07 : NUM_PATTERN 값을 60000으로 바꿔줍니다. 뒤에서 사용할 손글씨 MNIST 데이터 셋의 학습용 데이터의 개수가 60000입니다.

24 : time.time() 함수를 호출하여 begin 변수에 학습 시작 시간을 써줍니다.


26 : for 문을 1회만 수행하도록 range 값을 1에서 2 미만으로 변경합니다.

69, 70 : rc값이 1000으로 나눈 나머지가 999일 때, 즉, 매 1000번 마다 학습 수행중임을 표시하기 위해 print 함수를 호출하여 점을 출력해줍니다. flush 인자를 True로 설정하여 점을 바로 출력하도록 합니다.

72 : time.time() 함수를 호출하여 end 변수에 학습 종료 시간을 써줍니다.

74 : time_taken 변수에 학습 측정 시간을 계산해 써줍니다.

76 : print 함수를 호출하여 학습에 걸린 시간을 출력해 줍니다.

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

```

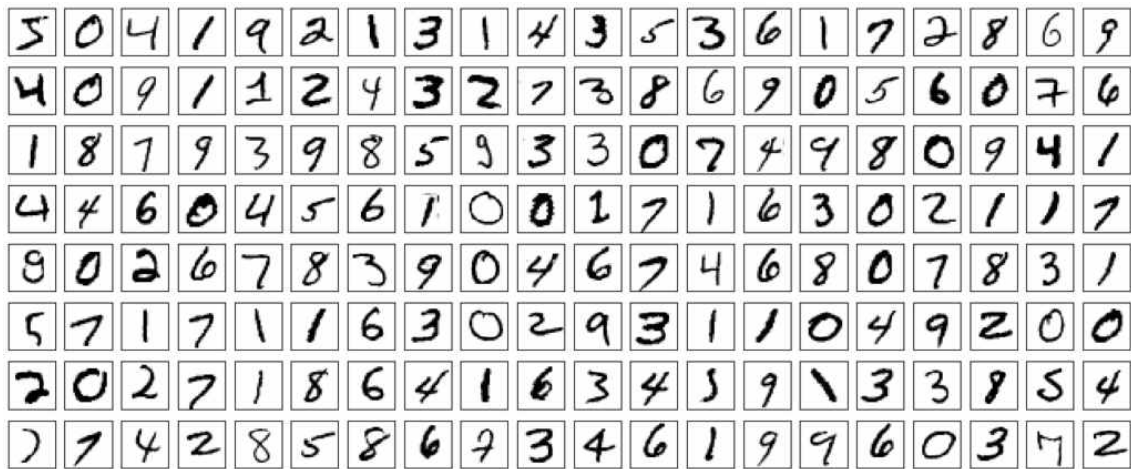
.....
Time taken (in seconds) = 22.50955057144165

```

필자의 경우, 22.5096 초가 걸립니다.

05 MNIST 파일 읽어서 학습해 보기

여기서는 NumPy 인공 신경망으로 다음과 같이 MNIST 데이터를 학습시켜 봅니다.



1. 이전 예제를 복사합니다.
 2. 다음과 같이 파일을 수정합니다.
- 325_1.py

```

01 import numpy as np
02 import random
03 import time
04 import tensorflow as tf
05
06 mnist = tf.keras.datasets.mnist
07
08 (x_train, y_train), (x_test, y_test) = mnist.load_data()
09 x_train, x_test = x_train / 255.0, x_test / 255.0
10 x_train = x_train.reshape((60000, 1, 784))
11 y_train = np.array(tf.one_hot(y_train, depth=10))
12 y_train = y_train.reshape((60000, 1, 10))
13
14 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.4f}".format(x)})
15
16 NUM_PATTERN = 60000
17 NUM_IN = 784
18 NUM_HID = 64
19 NUM_OUT = 10
20
21 I = x_train
22 T = y_train
23 O = np.zeros((NUM_PATTERN, 1, NUM_OUT))
24 WH = np.random.randn(NUM_IN, NUM_HID)/np.sqrt(NUM_IN/2) # He

```



```

25 BH = np.zeros((1, NUM_HID))
26 WO = np.random.randn(NUM_HID, NUM_OUT)/np.sqrt(NUM_HID) # Lecun
27 BO = np.zeros((1, NUM_OUT))
28
29 shuffled_pattern = [pc for pc in range(NUM_PATTERN)] #정수로!
30
31 random.seed(int(time.time()))
32
33 begin = time.time()
34
35 for epoch in range(1, 4):
36
37     tmp_a = 0;
38     tmp_b = 0;
39     for pc in range(NUM_PATTERN) :
40         tmp_a = random.randrange(0,NUM_PATTERN)
41         tmp_b = shuffled_pattern[pc]
42         shuffled_pattern[pc] = shuffled_pattern[tmp_a]
43         shuffled_pattern[tmp_a] = tmp_b
44
45     sumError = 0.
46
47     hit, miss = 0, 0
48
49     for rc in range(NUM_PATTERN) :
50
51         pc = shuffled_pattern[rc]
52
53         H = I[pc] @ WH + BH
54         H = (H>0)*H # ReLU
55
56         O[pc] = H @ WO + BO
57         O[pc] = 1/(1+np.exp(-O[pc])) #sigmoid
58         if np.argmax(O[pc][0])!=np.argmax(T[pc][0]) :
59             hit+=1
60         else :
61             miss+=1
62
63         E = np.sum((O[pc]-T[pc])**2/2) #mean squared error
64

```

```

65         sumError += E
66
67         Ob = O[pc] - T[pc]
68         Ob = Ob*O[pc]*(1-O[pc]) #sigmoid
69
70         Hb = Ob @ WO.T
71         Hb = Hb*(H>0)*1 # ReLU
72
73         WHb = I[pc].T @ Hb
74         BHb = 1 * Hb
75         WOb = H.T @ Ob
76         BOb = 1 * Ob
77
78         lr = 0.01
79         WH = WH - lr * WHb
80         BH = BH - lr * BHb
81         WO = WO - lr * WOb
82         BO = BO - lr * BOb
83
84         if rc%10000==9999 :
85             print("epoch: %2d rc: %6d " %(epoch, rc+1), end='')
86             print("hit: %6d miss: %6d " %(hit, miss), end='')
87             print("loss: %f accuracy: %f" \
88                   %(sumError/10000, hit/(hit+miss)))
89             sumError = 0
90
91     end = time.time()
92
93     time_taken = end - begin
94
95     print("\nTime taken (in seconds) = {}".format(time_taken))

```

04 : import문을 이용하여 tensorflow 모듈을 tf라는 이름으로 불러옵니다. tensorflow 모듈은 구글에서 제공하는 인공 신경망 라이브러리입니다. 여기서는 06 번째 줄에서 mnist 데이터 셋을 사용하기 위해 필요합니다.

06 : mnist 변수를 생성한 후, tf.keras.datasets.mnist 모듈을 가리키게 합니다. mnist 모듈은 손 글씨 숫자 데이터를 가진 모듈입니다. mnist 모듈에는 6만개의 학습용 손 글씨 숫자 데이터와 1만개의 시험용 손 글씨 숫자 데이터가 있습니다. 이 데이터들에 대해서는 1 장에서 자세히 살펴보았습니다.

08 : mnist.load_data 함수를 호출하여 손 글씨 숫자 데이터를 읽어와 x_train, y_train, x_test, y_test 변수가 가리키게 합니다. x_train, x_test 변수는 각각 6만개의 학습용 손 글씨 숫자 데이터와 1만개의 시험용 손 글씨 숫자 데이터를 가리킵니다. y_train, y_test 변수

는 각각 6만개의 학습용 손 글씨 숫자 라벨과 1만개의 시험용 손 글씨 숫자 라벨을 가리킵니다.

09 : `x_train`, `x_test` 변수가 가리키는 6만개, 1만개의 그림은 각각 28x28 픽셀로 구성된 그림이며, 1픽셀의 크기는 8비트로 0에서 255사이의 숫자를 가집니다. 모든 픽셀의 숫자를 255.0으로 나누어 각 픽셀을 0.0에서 1.0사이의 실수로 바꾸어 인공 신경망에 입력하게 됩니다.

10 : `x_train` 변수가 가리키는 6만개 그림은 28x28 픽셀로 구성되어 있습니다. NumPy 인공 신경망의 경우 그림 데이터를 입력할 때 28x28 픽셀을 784(=28x28) 픽셀로 일렬로 세워서 입력하게 됩니다. 그래서 `x_train`의 모양을 (60000, 1, 784)로 변경해 줍니다.

11 : `tf.one_hot` 함수를 호출하여 `y_train`의 값을 10개의 0또는 1로 구성된 형태의 배열로 변경합니다. 이런 형태의 라벨 변환을 one hot encoding이라고 합니다.

12 : `y_train`의 모양을 (60000, 1, 10)으로 변경해 줍니다.

21 : `I`을 `x_train`으로 할당합니다.

22 : `T`를 `y_train`으로 할당합니다.

35 : `for` 문을 3회만 수행하도록 `range` 값을 1에서 4 미만으로 변경합니다.


47 : `hit`, `miss` 변수를 생성한 후, 각각 0으로 초기화합니다. `hit` 변수는 59줄에서 예측이 맞을 때, `miss`는 61줄에서 예측이 틀릴 때, 하나씩 증가시킵니다. `hit`, `miss`는 매 회기마다 47줄에서 0으로 초기화되어 전체 입력 데이터에 대해 맞은 예측과 틀린 예측을 기록합니다.

58~61 : `np.argmax` 함수를 호출하여 `O[pc]` 행렬의 가장 큰 항목 색인값과 `T[pc]` 행렬의 가장 큰 항목 위치값을 비교하여 같으면 `hit` 값을 1 증가시키고, 그렇지 않으면 `miss` 값을 1 증가시킵니다.

84 : `rc` 값을 10000으로 나누어 나머지가 9999일 때마다, 즉 매 10000번 마다 85~89줄을 수행합니다.

85~88 : `epoch`, `rc+1`, `hit`, `miss`, `loss`, `accuracy` 값을 출력합니다.

89 : `sumError` 값을 0으로 초기화합니다. `sumError` 값은 매 10000번 마다 초기화되어 10000번 수행에 대한 오차 합을 기록합니다.

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

```

epoch: 1 rc: 10000 hit: 7629 miss: 2371 loss: 0.216441 accuracy: 0.762900
epoch: 1 rc: 20000 hit: 16540 miss: 3460 loss: 0.106507 accuracy: 0.827000
epoch: 1 rc: 30000 hit: 25539 miss: 4461 loss: 0.090568 accuracy: 0.851300
epoch: 1 rc: 40000 hit: 34678 miss: 5322 loss: 0.079028 accuracy: 0.866950
epoch: 1 rc: 50000 hit: 43856 miss: 6144 loss: 0.073834 accuracy: 0.877120
epoch: 1 rc: 60000 hit: 53100 miss: 6900 loss: 0.067624 accuracy: 0.885000
epoch: 2 rc: 10000 hit: 9342 miss: 658 loss: 0.062061 accuracy: 0.934200
epoch: 2 rc: 20000 hit: 18655 miss: 1345 loss: 0.061443 accuracy: 0.932750
epoch: 2 rc: 30000 hit: 28070 miss: 1930 loss: 0.053897 accuracy: 0.935667
epoch: 2 rc: 40000 hit: 37474 miss: 2526 loss: 0.053564 accuracy: 0.936850
epoch: 2 rc: 50000 hit: 46874 miss: 3126 loss: 0.052256 accuracy: 0.937480
epoch: 2 rc: 60000 hit: 56321 miss: 3679 loss: 0.049771 accuracy: 0.938683
epoch: 3 rc: 10000 hit: 9476 miss: 524 loss: 0.047631 accuracy: 0.947600
epoch: 3 rc: 20000 hit: 19011 miss: 989 loss: 0.043784 accuracy: 0.950550
epoch: 3 rc: 30000 hit: 28533 miss: 1467 loss: 0.043348 accuracy: 0.951100
epoch: 3 rc: 40000 hit: 38054 miss: 1946 loss: 0.044344 accuracy: 0.951350
epoch: 3 rc: 50000 hit: 47585 miss: 2415 loss: 0.043094 accuracy: 0.951700
epoch: 3 rc: 60000 hit: 57132 miss: 2868 loss: 0.041638 accuracy: 0.952200

```

Time taken (in seconds) = 69.69139266014099

3회 학습을 수행한 후, 95.22%의 정확도로 학습 데이터를 예측하고 있습니다. 맨 마지막의 학습 결과는 60000개의 입력 데이터에 대해 57132개를 올바르게 예측하고, 2868개를 틀리게 예측합니다. 학습 수행 시간은 약 70초 정도 걸립니다.

softmax, cross entropy error 함수 사용하기

이번에는 출력 층의 활성화 함수를 softmax 함수로, 오차 계산 함수를 cross entropy error 함수로 변경한 후, 학습을 수행해 봅니다.

1. 이전 예제를 복사합니다.
2. 다음과 같이 파일을 수정합니다.
325_2.py

```

01~52 #이전 예제와 같습니다.
53         H = I[pc] @ WH + BH
54         H = (H>0)*H # ReLU
55
56         O[pc] = H @ WO + BO
57         OM = O[pc] - np.max(O[pc])
58         O[pc] = np.exp(OM)/np.sum(np.exp(OM)) #softmax
59         if np.argmax(O[pc][0])!=np.argmax(T[pc][0]) :
60             hit+=1

```


```

61         else :
62             miss+=1
63
64         E = np.sum(-T[pc]*np.log(O[pc])) #cross entropy error
65
66         sumError += E
67
68         Ob = O[pc] - T[pc]
69         # nothing for softmax
70
71         Hb = Ob @ WO.T
72         Hb = Hb*(H>0)*1 # ReLU
73~끝 #이전 예제와 같습니다.

```

57, 58 : 출력 층의 함수를 softmax 함수로 변경합니다.

64 : 오차 계산 함수를 cross entropy error 함수로 변경합니다.

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

```

epoch: 1 rc: 10000 hit: 8597 miss: 1403 loss: 0.462304 accuracy: 0.859700
epoch: 1 rc: 20000 hit: 17862 miss: 2138 loss: 0.253578 accuracy: 0.893100
epoch: 1 rc: 30000 hit: 27197 miss: 2803 loss: 0.219167 accuracy: 0.906567
epoch: 1 rc: 40000 hit: 36674 miss: 3326 loss: 0.181170 accuracy: 0.916850
epoch: 1 rc: 50000 hit: 46146 miss: 3854 loss: 0.174503 accuracy: 0.922920
epoch: 1 rc: 60000 hit: 55660 miss: 4340 loss: 0.161298 accuracy: 0.927667
epoch: 2 rc: 10000 hit: 9606 miss: 394 loss: 0.133489 accuracy: 0.960600
epoch: 2 rc: 20000 hit: 19241 miss: 759 loss: 0.122667 accuracy: 0.962050
epoch: 2 rc: 30000 hit: 28866 miss: 1134 loss: 0.126253 accuracy: 0.962200
epoch: 2 rc: 40000 hit: 38492 miss: 1508 loss: 0.126328 accuracy: 0.962300
epoch: 2 rc: 50000 hit: 48144 miss: 1856 loss: 0.116273 accuracy: 0.962880
epoch: 2 rc: 60000 hit: 57802 miss: 2198 loss: 0.112065 accuracy: 0.963367
epoch: 3 rc: 10000 hit: 9698 miss: 302 loss: 0.098693 accuracy: 0.969800
epoch: 3 rc: 20000 hit: 19402 miss: 598 loss: 0.092623 accuracy: 0.970100
epoch: 3 rc: 30000 hit: 29130 miss: 870 loss: 0.089414 accuracy: 0.971000
epoch: 3 rc: 40000 hit: 38847 miss: 1153 loss: 0.089008 accuracy: 0.971175
epoch: 3 rc: 50000 hit: 48585 miss: 1415 loss: 0.093995 accuracy: 0.971700
epoch: 3 rc: 60000 hit: 58261 miss: 1739 loss: 0.100390 accuracy: 0.971017

```

Time taken (in seconds) = 70.44097638130188

3회 학습을 수행한 후, 97.10%의 정확도로 학습 데이터를 예측하고 있습니다. 맨 마지막의 학습 결과는 60000개의 입력 데이터에 대해 58261개를 올바르게 예측하고, 1739개를 틀리게 예측합니다. 학습 수행 시간은 약 70초 정도 걸립니다.

FASHION MNIST 데이터 테스트


이번엔 데이터 셋을 FASHION MNIST 데이터로 변경하여 테스트해 봅니다. 패션 MNIST 데이터 셋은 손 글씨 MNIST보다 좀 더 복잡한 형태의 이미지를 제공하기 위해 만들어졌습니다.



1. 이전 예제를 복사합니다.
2. 다음과 같이 파일을 수정합니다.
325_2.py

```
06 mnist = tf.keras.datasets.fashion_mnist
```

06 : minst 변수가 fashion_mnist 데이터 셋을 가리키도록 변경합니다.

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


```
epoch: 1 rc: 10000 hit: 7439 miss: 2561 loss: 0.703471 accuracy: 0.743900
epoch: 1 rc: 20000 hit: 15550 miss: 4450 loss: 0.528387 accuracy: 0.777500
epoch: 1 rc: 30000 hit: 23736 miss: 6264 loss: 0.494457 accuracy: 0.791200
epoch: 1 rc: 40000 hit: 32037 miss: 7963 loss: 0.464478 accuracy: 0.800925
epoch: 1 rc: 50000 hit: 40330 miss: 9670 loss: 0.464184 accuracy: 0.806600
epoch: 1 rc: 60000 hit: 48752 miss: 11248 loss: 0.445191 accuracy: 0.812533
epoch: 2 rc: 10000 hit: 8474 miss: 1526 loss: 0.420411 accuracy: 0.847400
epoch: 2 rc: 20000 hit: 16923 miss: 3077 loss: 0.421269 accuracy: 0.846150
epoch: 2 rc: 30000 hit: 25425 miss: 4575 loss: 0.411722 accuracy: 0.847500
epoch: 2 rc: 40000 hit: 34004 miss: 5996 loss: 0.390658 accuracy: 0.850100
epoch: 2 rc: 50000 hit: 42579 miss: 7421 loss: 0.398140 accuracy: 0.851580
epoch: 2 rc: 60000 hit: 51161 miss: 8839 loss: 0.401036 accuracy: 0.852683
epoch: 3 rc: 10000 hit: 8617 miss: 1383 loss: 0.374698 accuracy: 0.861700
epoch: 3 rc: 20000 hit: 17187 miss: 2813 loss: 0.385727 accuracy: 0.859350
epoch: 3 rc: 30000 hit: 25773 miss: 4227 loss: 0.382991 accuracy: 0.859100
epoch: 3 rc: 40000 hit: 34420 miss: 5580 loss: 0.371946 accuracy: 0.860500
epoch: 3 rc: 50000 hit: 43046 miss: 6954 loss: 0.371740 accuracy: 0.860920
epoch: 3 rc: 60000 hit: 51707 miss: 8293 loss: 0.363734 accuracy: 0.861783
```

Time taken (in seconds) = 69.41749691963196

3회 학습을 수행한 후, 86.18%의 정확도로 학습 데이터를 예측하고 있습니다. 맨 마지막의 학습 결과는 60000개의 입력 데이터에 대해 51707개를 올바르게 예측하고, 8293개를 틀리게 예측합니다. 학습 수행 시간은 약 69초 정도 걸립니다.

이상에서 NumPy 인공 신경망을 이용하여 MNIST 데이터 셋에 대해 학습과 테스트를 수행해 보았습니다.