

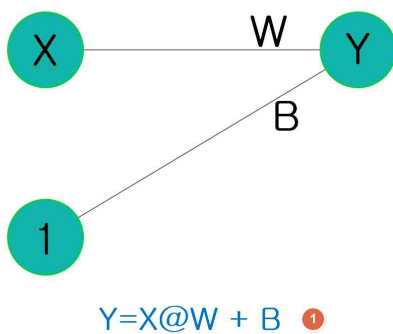
## Chapter 04

### Tensorflow 내부 동작 이해하기

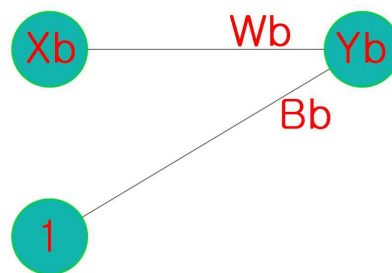
## 01 NumPy로 Tensorflow 내부 동작 이해하기

다음은 단일 인공 신경의 순전파, 역전파 행렬 계산식을 나타낸 그림입니다. 여기서는 다음 인공 신경을 NumPy와 Tensorflow로 구현해 보며 텐서플로우의 내부적인 동작을 이해해 봅니다. Tensorflow의 내부 동작을 잘 이해하여 Tensorflow의 활용도를 높일 수 있도록 합니다.

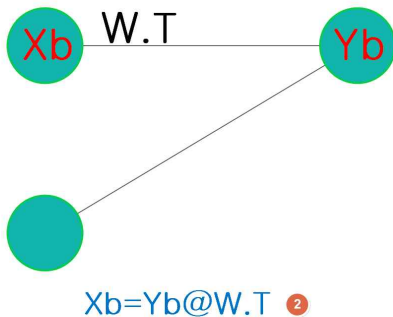
[그림1]



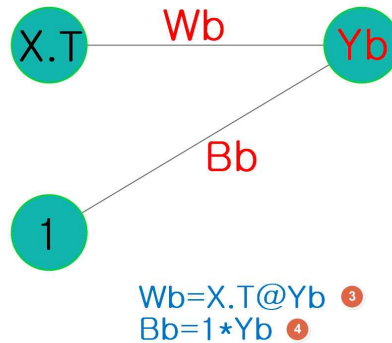
[그림2]



[그림3]



[그림4]



[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.

\*\*\* ② X 층이 입력 층일 경우엔 이 수식은 필요하지 않습니다.

\*\*\* @ 문자는 행렬 곱을 의미합니다.

이상에서 인공 신경 학습에 필요한 행렬 계산식을 정리하면 다음과 같습니다.

순전파

```
Y = X @ W + B ❶
```

오차

```
E = np.sum((Y - T) ** 2 / 2) ❺
```

역전파 오차

```
Yb = Y - T ❻
```

가중치, 편향 역전파

```
Wb = X.T @ Yb ❸
```

```
Bb = 1 * Yb ❹
```

가중치, 편향 학습

```
W = W - lr*Wb ❼
```

```
B = B - lr*Bb ❽
```

\*\*\* lr은 학습률을 나타냅니다.

❺ 지금까지 우리는 다음과 같이 오차 계산을 하였습니다.

```
E = np.sum((Y - T) ** 2 / 2)
```

이 코드는 다음 수식을 구현한 형태입니다.

$$MSE = \frac{1}{2} \sum_{i=1}^n (y_i - t_i)^2$$

그러나 Tensorflow와 비교하는 예제에서 우리는 다음과 같은 수식을 사용합니다.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - t_i)^2$$

이 수식은 Tensorflow에서 사용하는 평균 오차 계산식입니다. 여기서 n은 출력 층 노드의 개수를 의미합니다.

그래서 Tensorflow와 비교하는 NumPy예제에서 우리는 다음과 같은 형태로 오차를 계산하게

됩니다.

$$E = \text{np.sum}((Y - T) ** 2) / Y.\text{shape}[1]$$

여기서  $Y.\text{shape}[1]$ 은 출력 노드의 개수 값을 갖습니다.

결론적으로 오차 계산식에 차이가 있지만 학습의 결과는 크게 다르지 않습니다.

⑥ 지금까지 역전파 오차의 경우엔 다음과 같이 계산 하였습니다.

$$Yb = Y - T$$

그러나 Tensorflow에서는 바뀐 형태의 평균 제곱 오차를 기반으로 다음과 같은 형태로 역전파 오차를 계산해야 합니다.

$$Yb = 2 * (Y - T) / Y.\text{shape}[1]$$

2와  $Y.\text{shape}[1]$ 은 편미분 과정에서 추가된 부분이며, 여기서는 자세한 설명을 하지는 않습니다.

## 01 2입력 2출력 인공 신경망 구현하기

다음은 [입력2 출력2]의 인공 신경망 학습에 사용할 행렬을 나타냅니다.

$$\begin{aligned} X &= \begin{bmatrix} 2 & 3 \end{bmatrix} \textcircled{9} \\ T &= \begin{bmatrix} 27 & -30 \end{bmatrix} \textcircled{10} \\ W &= \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \textcircled{11} \\ B &= \begin{bmatrix} 1 & 2 \end{bmatrix} \textcircled{12} \end{aligned}$$

X를 입력 값으로, T를 목표 값으로 하여 가중치 W와 편향 B에 대해 NumPy 행렬 계산식과 Tensorflow로 구현해 봅니다.

### NumPy로 구현하기

먼저 NumPy로 구현해 봅니다.

1. 다음과 같이 예제를 작성합니다.


411\_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]]) # 9
06 T = np.array([[27, -30]]) # 10
07
08 W = np.array([[3, 5], # 11
09               [4, 6]])
10 B = np.array([[1, 2]]) # 12
11
12 for epoch in range(2):
13
14     print('epoch = %d' %epoch)
15
16     Y = X @ W + B # 1
17     print(' Y =', Y)
18
19     E = np.sum((Y - T) ** 2) / Y.shape[1] # 5
20
21     Yb = 2 * (Y - T) / Y.shape[1] # 6
22
23     Wb = X.T @ Yb # 3
24     Bb = 1 * Yb # 4
25
26     lr = 0.01
27
28     W = W - lr * Wb # 7
29     B = B - lr * Bb # 8
30     print(' W =\n', W)
31     print(' B =', B)
```

19 : Tensorflow에서 사용하는 평균 오차 계산식입니다.

21 : Tensorflow에서 사용하는 역전파 오차 계산식입니다.

21 : 예측 값을 가진 Y 행렬에서 목표 값을 가진 T 행렬을 뺀 후, 2를 곱해주고 Y의 노드 개수로 나눠준 후, 결과 값을 Yb 변수에 할당합니다. 2와 Y.shape[1]은 편미분 과정에서 추가된 부분이며, 여기서는 자세한 설명을 하지는 않습니다.

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

```

epoch = 0
Y = [[19 30]]
W =
[[ 3.160  3.800]
 [ 4.240  4.200]]
B = [[ 1.080  1.400]]
epoch = 1
Y = [[20.120 21.600]]
W =
[[ 3.298  2.768]
 [ 4.446  2.652]]
B = [[ 1.149  0.884]]

```

Y, W, B 값이 2회 출력됩니다.

## Tensorflow로 구현하기

다음은 텐서플로우로 구현해 봅니다.

1. 다음과 같이 예제를 작성합니다.

411\_2.py

```

01 import tensorflow as tf
02 import numpy as np
03
04 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
05
06 X = np.array([[2, 3]]) # 9
07 T = np.array([[27, -30]]) # 10
08
09 W = np.array([[3, 5], # 11
10               [4, 6]])
11 B = np.array([1, 2]) # 12
12
13 model = tf.keras.Sequential([
14     tf.keras.layers.Dense(2, input_shape=(2,)),
15 ])
16
17 model.layers[0].set_weights([W, B])
18
19 model.compile(
20     optimizer=tf.keras.optimizers.SGD(learning_rate=0.01), # 7 8

```

```

21         loss=tf.keras.losses.MeanSquaredError()) # ❸
22
23 for epoch in range(2):
24
25     print('epoch = %d' %epoch)
26
27     Y = model.predict(X) # ❶
28     print(' Y  =', Y)
29
30     model.fit(X, T, epochs=1)
31     print(' W =\n', model.layers[0].get_weights()[0])
32     print(' B =', model.layers[0].get_weights()[1])

```

01 : import문을 이용하여 tensorflow 모듈을 tf라는 이름으로 불러옵니다. tensorflow 모듈은 구글에서 제공하는 인공 신경망 라이브러리입니다.

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

04 : np.set\_printoptions 함수를 호출하여 numpy의 실수 출력 방법을 변경합니다. 이 예제에서는 소수점 이하 3자리까지 출력합니다.

6~10 : X, T, W 변수를 이차 배열의 numpy 행렬로 초기화합니다.

11 : B 변수를 일차 배열의 numpy 벡터로 초기화합니다.

13~15 : tf.keras.Sequential 클래스를 이용하여 인공 신경망을 생성합니다.


14 : tf.keras.layers.Dense 클래스를 이용하여 신경 망 층을 생성합니다. 입력 노드 2개와 연결된 출력 노드 2개로 구성된 인공 신경망 층을 생성합니다. 여기서 첫 번째 인자는 출력 노드의 개수를 나타냅니다. input\_shape 변수는 입력 노드의 개수를 나타냅니다.

17 : 인공 신경망에 임의로 설정된 가중치와 편향을 W, B로 변경합니다.

19~21 : model.compile 함수를 호출하여 내부적으로 인공 신경망을 구성합니다. 인공 신경망을 구성할 때에는 2개의 함수를 정해야 합니다. loss 함수와 optimizer 함수, 즉, 손실 함수와 최적화 함수를 정해야 합니다. 손실 함수로는 NumPy 예제의 ❸에 해당하는 tf.keras.losses.MeanSquaredError 함수를 사용하고 최적화 함수는 확률적 경사 하강(sgd : stochastic gradient descent) 함수인 NumPy 예제의 ❶, ❷에 해당하는 tf.keras.optimizers.SGD를 사용합니다.

27 : model.predict 함수를 호출하여 인공 신경망을 사용합니다. 이 과정은 순전파를 수행하는 과정입니다. NumPy 예제의 ❶에 해당하는 동작을 수행합니다.

30 : model.fit 함수를 호출하여 인공 신경망에 대한 학습을 시작합니다. fit 함수에는 X, T 데이터가 입력이 되는데 인공 신경망을 X, T 데이터에 맞도록 학습한다는 의미를 갖습니다. 즉, X, T 데이터에 맞도록 인공 신경망을 조물조물, 주물주물 학습한다는 의미입니다. fit 함수에는 학습을 몇 회 수행할지도 입력해 줍니다. epochs는 학습 횟수를 의미하며, 여기서는 1회 학습을 수행하도록 합니다. 일반적으로 학습 횟수에 따라 인공 신경망 근사 함수가 정확해 집니다.

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

```
epoch = 0
Y = [[19.000 30.000]]
1/1 [=====]
W =
[[ 3.160  3.800]
 [ 4.240  4.200]]
B = [ 1.080  1.400]
epoch = 1
Y = [[20.120 21.600]]
1/1 [=====]
W =
[[ 3.298  2.768]
 [ 4.446  2.652]]
B = [ 1.149  0.884]
```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 반복 학습 20회 수행하기

여기서는 반복 학습 20회를 수행해 봅니다.


1. 다음과 같이 앞에서 작성한 NumPy, 텐서플로우 예제를 수정합니다.

```
12 for epoch in range(20):
```

12 : epoch값을 0에서 20 미만까지로 바꾸어줍니다.

```
23 for epoch in range(20):
```

23 : epoch값을 0에서 20 미만까지로 바꾸어줍니다.

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

epoch = 19	epoch = 19
Y = [[26.544 -26.583]]	Y = [[26.544 -26.583]]
1/1 [=====]	1/1 [=====]
W =	W =
[[ 4.087 -3.152]	[[ 4.087 -3.152]
[ 5.630 -6.227]]	[ 5.630 -6.227]]
B = [[ 1.543 -2.076]]	B = [ 1.543 -2.076]

결과 값을 비교합니다. 결과 값이 같습니다.

## 반복 학습 200회 수행하기



여기서는 반복 학습 200회를 수행해 봅니다.


1. 다음과 같이 앞에서 작성한 NumPy, 텐서플로우 예제를 수정합니다.

```
12 for epoch in range(200):
```

12 : epoch값을 0에서 200 미만까지로 바꾸어줍니다.

```
23 for epoch in range(200):
```

23 : epoch값을 0에서 200 미만까지로 바꾸어줍니다.

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

epoch = 199	epoch = 199
Y = [[27.000 -30.000]]	Y = [[27.000 -30.000]]
W =	1/1 [=====]
[[ 4.143 -3.571]	W =
[ 5.714 -6.857]]	[[ 4.143 -3.571]
B = [[ 1.571 -2.286]]	[ 5.714 -6.857]]
	B = [ 1.571 -2.286]

결과 값을 비교합니다. 결과 값이 같습니다.

## 02 3입력 3출력 인공 신경망 구현하기

다음은 [입력3 출력3]의 인공 신경망 학습에 사용할 행렬을 나타냅니다.

$$\begin{aligned} X &= \begin{bmatrix} 2 & 3 & 4 \end{bmatrix} \\ T &= \begin{bmatrix} 27 & -30 & 179 \end{bmatrix} \\ W &= \begin{bmatrix} 3 & 5 & 8 \\ 4 & 6 & 9 \\ 5 & 7 & 10 \end{bmatrix} \\ B &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \end{aligned}$$

X를 입력 값으로, T를 목표 값으로 하여 가중치 W와 편향 B에 대해 NumPy 행렬 계산식과 Tensorflow로 구현해 봅니다.

### NumPy로 구현하기


먼저 NumPy로 구현해 봅니다.

1. 다음과 같이 예제를 복사하여 수정합니다.  
412\_1.py

```

01~04 # 이전 예제와 같습니다.
05 X = np.array([[2, 3, 4]])
06 T = np.array([[27, -30, 179]])
07
08 W = np.array([[3, 5, 8],
09               [4, 6, 9],
10               [5, 7, 10]])
11 B = np.array([1, 2, 3])
12~끝 # 이전 예제와 같습니다.

```

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

```

epoch = 199
Y = [[27.000 -30.000 179.000]]
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]]

```

## Tensorflow로 구현하기

다음은 텐서플로우로 구현해 봅니다.

1. 다음과 같이 예제를 복사하여 수정합니다.


412\_2.py

```

01~05 # 이전 예제와 같습니다.
06 X = np.array([[2, 3, 4]])
07 T = np.array([[27, -30, 179]])
08
09 W = np.array([[3, 5, 8],
10               [4, 6, 9],
11               [5, 7, 10]])
12 B = np.array([1, 2, 3])
13
14 model = tf.keras.Sequential([
15     tf.keras.layers.Dense(3, input_shape=(3,)),
16 ])
17~끝 : # 이전 예제와 같습니다.

```

15 : 출력 층 노드의 개수를 3으로, 입력 층 노드의 개수를 3으로 변경합니다.

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

```
epoch = 199
Y = [[27.000 -30.000 179.000]]
1/1 [=====]
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]
```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 03 2입력 1출력 인공 신경 구현하기

다음은 입력2 출력1의 인공 신경 학습에 사용할 행렬을 나타냅니다.

$$X = \begin{bmatrix} 2 & 3 \end{bmatrix}$$
$$T = \begin{bmatrix} 27 \end{bmatrix}$$
$$W = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$
$$B = \begin{bmatrix} 1 \end{bmatrix}$$

X를 입력 값으로, T를 목표 값으로 하여 가중치 W와 편향 B에 대해 NumPy 행렬 계산식과 Tensorflow로 구현해 봅니다.


### NumPy로 구현하기

먼저 NumPy로 구현해 봅니다.

1. 다음과 같이 예제를 복사하여 수정합니다.

413\_1.py

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

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

```
epoch = 199
Y = [[27.000]]
W =
[[ 4.143]
 [ 5.714]]
B = [[ 1.571]]
```

## Tensorflow로 구현하기


다음은 텐서플로우로 구현해 봅니다.

1. 다음과 같이 예제를 수정합니다.

413\_2.py

```
01~05 # 이전 예제와 같습니다.
06 X = np.array([[2, 3]])
07 T = np.array([[27]])
08
09 W = np.array([[3],
10               [4]])
11 B = np.array([1])
12
13 model = tf.keras.Sequential([
14     tf.keras.layers.Dense(1, input_shape=(2,)),
15 ])
16~끝 # 이전 예제와 같습니다.
```

14 : 출력 층 노드의 개수를 1로, 입력 층 노드의 개수를 2로 변경합니다.

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

```
epoch = 199
Y = [[27.000]]
1/1 [=====]
W =
[[ 4.143]
 [ 5.714]]
B = [ 1.571]
```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 04 1입력 1출력 인공 신경 구현하기

다음은 [입력1 출력1]의 인공 신경 학습에 사용할 행렬을 나타냅니다.

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

X를 입력 값으로, T를 목표 값으로 하여 가중치 W와 편향 B에 대해 NumPy 행렬 계산식과 Tensorflow로 구현해 봅니다.


### NumPy로 구현하기

먼저 NumPy로 구현해 봅니다.

1. 다음과 같이 예제를 복사하여 수정합니다.

414\_1.py

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

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

```
epoch = 199  
Y = [[10.000]]  
W =  
[[ 4.200]]  
B = [[ 1.600]]
```

### Tensorflow로 구현하기


다음은 텐서플로우로 구현해 봅니다.

1. 다음과 같이 예제를 수정합니다.

414\_2.py

```
01~05 # 이전 예제와 같습니다.
06 X = np.array([[2]])
07 T = np.array([[10]])
08
09 W = np.array([[3]])
10 B = np.array([1])
11
12 model = tf.keras.Sequential([
13     tf.keras.layers.Dense(1, input_shape=(1,)),
14 ])
15~끝 # 이전 예제와 같습니다.
```

13 : 출력 층 노드의 개수를 1로, 입력 층 노드의 개수를 1로 변경합니다.

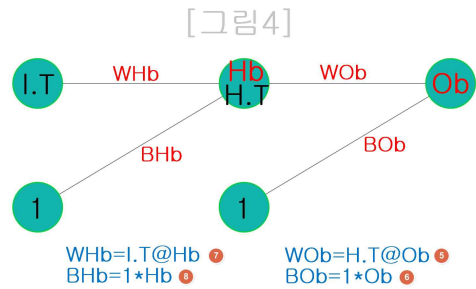
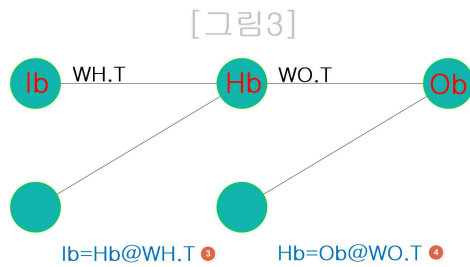
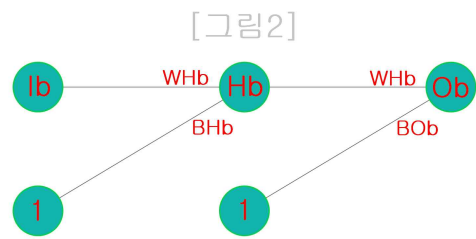
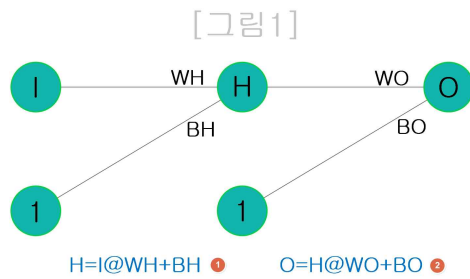
2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 199
Y = [[10.000]]
1/1 [=====]
W =
[[ 4.200]]
B = [ 1.600]
```

NumPy 예제와 결과가 같은지 비교해 봅시다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 05 2입력 2은닉 2출력 인공 신경망 구현하기

다음은 은닉 층이 포함된 인공 신경망의 순전파, 역전파 행렬 계산식을 나타낸 그림입니다. 이 그림은 앞에서 살펴본 그림입니다. 여기서는 다음 인공 신경망을 NumPy와 Tensorflow로 구현해 보며 텐서플로우의 내부적인 동작을 이해해 봅시다.



[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 I 층이 입력 층일 경우엔 이 수식은 필요하지 않습니다.

\*\*\* @ 문자는 행렬 곱을 의미합니다.

이상에서 필요한 행렬 계산식을 정리하면 다음과 같습니다.

순전파
$H = I@WH + BH$ ①
$O = H@WO + BO$ ②
역전파 오차
$Ob = O - T$ ⑩
입력 역전파
$Hb = Ob@WO.T$ ④
가중치, 편향 역전파

```

WHb = I.T@Hb 7
BHb = 1*Hb 8
WOb = H.T@Ob 5
BOb = 1*Ob 6

```

가중치, 편향 학습

```

WH = WH - lr*WHb 13
BH = BH - lr*BHb 14
WO = WO - lr*WOb 11
BO = BO - lr*BOb 12

```

\*\*\* lr은 학습률을 나타냅니다.

⑩ Tensorflow에서는 바뀐 형태의 평균 제곱 오차를 기반으로 다음과 같은 형태로 역전파 오차를 계산합니다.

$$Ob = 2 * (O - T) / O.shape[1]$$

2와 O.shape[1]은 편미분 과정에서 추가된 부분이며, 여기서는 자세한 설명을 하지는 않습니다.

다음은 [입력2 출력2]의 인공 신경 학습에 사용할 행렬을 나타냅니다.

```

I = [.05 .10]
T = [.01 .99]
WH = [.15 .25]
    [.20 .30]
BH = [.35 .35]
WO = [.40 .50]
    [.45 .66]
BO = [.60 .60]

```

I를 입력 값으로, T를 목표 값으로 하여 가중치와 편향 WH, WO, BH, BO에 대해 NumPy 행렬 계산식과 Tensorflow로 구현해 봅니다.

## NumPy로 구현하기

먼저 NumPy로 구현해 봅니다.

1. 다음과 같이 예제를 작성합니다.



415\_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(2):
15
16     print('epoch = %d' %epoch)
17
18     H = I @ WH + BH # ❶
19     O = H @ WO + BO # ❷
20     print(' O  =', O)
21
22     E = np.sum((O - T) ** 2) / O.shape[1] # ❸
23
24     Ob = 2 * (O - T) / O.shape[1]
25     Hb = Ob@WO.T # ❹
26
27     WOb = H.T @ Ob # ❺
28     BOb = 1 * Ob # ❻
29     WHb = I.T @ Hb # ❼
30     BHb = 1 * Hb # ❽
31
32     lr = 0.01
33
34     WO = WO - lr * WOb # ❾
35     BO = BO - lr * BOb # ❿
36     WH = WH - lr * WHb # ⓫
37     BH = BH - lr * BHb # ⓬
38     print(' WH =\n', WH)
39     print(' BH =\n', BH)
```

```
40     print(' WO =\n', WO)
41     print(' BO =\n', BO)
```

22 : Tensorflow에서 사용하는 평균 오차 계산식입니다.

24 : Tensorflow에서 사용하는 역전파 오차 계산식입니다.

38~41 : print 함수를 호출하여 학습이 수행된 WH, BH, WO, BO 행렬값을 출력합니다.

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

```
epoch = 0
O  = [[ 0.928  1.005]]
WH =
[[ 0.150  0.250]
 [ 0.200  0.300]]
BH =
[[ 0.346  0.346]]
WO =
[[ 0.397  0.500]
 [ 0.446  0.550]]
BO =
[[ 0.591  0.600]]
epoch = 1
O  = [[ 0.912  1.000]]
WH =
[[ 0.150  0.250]
 [ 0.199  0.299]]
BH =
[[ 0.343  0.342]]
WO =
[[ 0.393  0.500]
 [ 0.443  0.550]]
BO =
[[ 0.582  0.600]]
```

O, WH, BH, WO, BO 값이 2회 출력됩니다.

## Tensorflow로 구현하기

다음은 텐서플로우로 구현해 봅니다.

1. 다음과 같이 예제를 작성합니다.

415\_2.py

```
01 import tensorflow as tf
02 import numpy as np
03
04 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
05
06 I = np.array([[.05, .10]])
07 T = np.array([[.01, .99]])
08
```

```

09 WH = np.array([[.15, .25],
10                [.20, .30]])
11 BH = np.array([.35, .35])
12 WO = np.array([[.40, .50],
13                [.45, .55]])
14 BO = np.array([.60, .60])
15
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,)),
18     tf.keras.layers.Dense(2)
19 ])
20
21 model.layers[0].set_weights([WH, BH])
22 model.layers[1].set_weights([WO, BO])
23
24 model.compile(
25     optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
26     loss=tf.keras.losses.MeanSquaredError())
27
28 for epoch in range(2):
29
30     print('epoch = %d' %epoch)
31
32     O = model.predict(I)
33     print(' O  =', O)
34
35     model.fit(I, T, epochs=1)
36     print(' WH =\n', model.layers[0].get_weights()[0])
37     print(' BH =\n', model.layers[0].get_weights()[1])
38     print(' WO =\n', model.layers[1].get_weights()[0])
39     print(' BO =\n', model.layers[1].get_weights()[1])

```

11, 14 : BH, BO 변수를 일차 배열의 numpy 벡터로 초기화합니다.


17 : 은닉 층 노드의 개수를 2로, 입력 층 노드의 개수를 2로 설정합니다.

18 : 출력 층 노드의 개수를 2로 설정합니다.

21 : 인공 신경망 은닉 층에 임의로 설정된 가중치와 편향을 WH, BH로 변경합니다.

22 : 인공 신경망 출력 층에 임의로 설정된 가중치와 편향을 WO, BO로 변경합니다.

36~39: print 함수를 호출하여 학습이 수행된 WH, BH, WO, BO 행렬값을 출력합니다.

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

```

epoch = 0
0 = [[ 0.928  1.005]]
1/1 [=====]
WH =
[[ 0.150  0.250]
 [ 0.200  0.300]]
BH =
[ 0.346  0.346]
WO =
[[ 0.397  0.500]
 [ 0.446  0.550]]
BO =
[ 0.591  0.600]
epoch = 1
0 = [[ 0.912  1.000]]
1/1 [=====]
WH =
[[ 0.150  0.250]
 [ 0.199  0.299]]
BH =
[ 0.343  0.342]
WO =
[[ 0.393  0.500]
 [ 0.443  0.550]]
BO =
[ 0.582  0.600]

```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 반복 학습 20회 수행하기

여기서는 반복 학습 20회를 수행해 봅니다.


- 다음과 같이 앞에서 작성한 NumPy, 텐서플로우 예제를 수정합니다.

```
14 for epoch in range(20):
```

14 : epoch값을 0에서 20 미만까지로 바꾸어줍니다.

```
28 for epoch in range(20):
```

28 : epoch값을 0에서 20 미만까지로 바꾸어줍니다.

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

epoch = 19	epoch = 19
0 = [[ 0.690 0.946]]	0 = [[ 0.690 0.946]]
WH =	1/1 [=====]
[[ 0.147 0.247]	WH =
[ 0.194 0.294]]	[[ 0.147 0.247]
BH =	[ 0.194 0.294]]
[[ 0.293 0.285]]	BH =
W0 =	[ 0.293 0.285]
[[ 0.345 0.501]	W0 =
[ 0.393 0.551]]	[[ 0.345 0.501]
B0 =	[ 0.393 0.551]]
[[ 0.442 0.604]]	B0 =
	[ 0.442 0.604]

결과 값을 비교합니다. 결과 값이 같습니다.

## 반복 학습 200회 수행하기

여기서는 반복 학습 200회를 수행해 봅니다.


- 다음과 같이 앞에서 작성한 NumPy, 텐서플로우 예제를 수정합니다.

```
14 for epoch in range(200):
```

14 : epoch값을 0에서 200 미만까지로 바꾸어줍니다.

```
28 for epoch in range(200):
```

28 : epoch값을 0에서 200 미만까지로 바꾸어줍니다.

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

epoch = 199	epoch = 199
0 = [[ 0.085 0.962]]	0 = [[ 0.085 0.962]]
WH =	1/1 [=====]
[[ 0.143 0.242]	WH =
[ 0.186 0.284]]	[[ 0.143 0.242]
BH =	[ 0.186 0.284]]
[[ 0.213 0.188]]	BH =
W0 =	[ 0.213 0.188]
[[ 0.219 0.527]	W0 =
[ 0.268 0.577]]	[[ 0.219 0.527]
B0 =	[ 0.268 0.577]]
[[ -0.029 0.704]]	B0 =
	[[ -0.029 0.704]

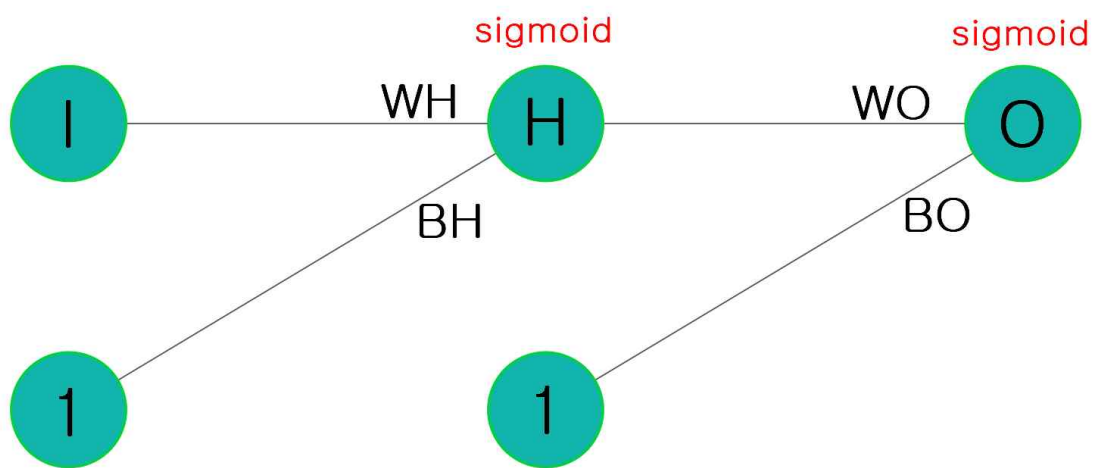
결과 값을 비교합니다. 결과 값이 같습니다.

## 06 활성화 함수 적용하기

여기서는 앞에서 NumPy와 Tensorflow를 이용해 구현한 인공 신경망에 sigmoid, tanh, ReLU 활성화 함수를 차례대로 적용해 보면서 텐서플로우의 내부적인 동작을 이해해 봅니다.

### sigmoid 함수 적용해 보기

먼저 은닉 층과 출력 층에 sigmoid 활성화 함수를 적용해 봅니다. 다음 그림을 살펴봅니다.



#### ❶ NumPy에 적용하기

먼저 NumPy 예제에 적용해 봅니다.

1. 이전 예제 415\_1.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다. # 표시된 부분은 추가되거나 수정된 부분을 나타냅니다.  
416\_1.py

```
01~17 # 이전 예제와 같습니다.
18     H = I @ WH + BH
19     H = 1/(1+np.exp(-H)) #
20
21     O = H @ WO + BO
22     O = 1/(1+np.exp(-O)) #
23
24     print(' O =', O)
25
```

```

26     E = np.sum((O - T) ** 2) / O.shape[1]
27
28     Ob = 2 * (O - T) / O.shape[1]
29     Ob = Ob * O * (1 - O) #
30
31     Hb = Ob @ WO.T
32     Hb = Hb * H * (1 - H) #
33~끝 # 이전 예제와 같습니다.


```

19 : 은닉 층 H에 순전파 시그모이드 활성화 함수를 적용합니다.

22 : 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 = 199
O = [[ 0.649  0.793]]
WH =
[[ 0.149  0.249]
 [ 0.199  0.298]]
BH =
[[ 0.337  0.334]]
WO =
[[ 0.230  0.542]
 [ 0.279  0.592]]
BO =
[[ 0.312  0.670]]

```

O, WH, BH, WO, BO 값을 확인합니다.

## ② Tensorflow에 적용하기

다음은 Tensorflow 예제에 적용해 봅니다.

1. 이전 예제 415\_2.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

416\_2.py

```

01~15 # 이전 예제와 같습니다.
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='sigmoid'),
18     tf.keras.layers.Dense(2, activation='sigmoid')


```

19 )]

20~끝 # 이전 예제와 같습니다.

17 : 은닉 층에 활성화 함수 sigmoid를 적용합니다.

18 : 출력 층에 활성화 함수 sigmoid를 적용합니다.

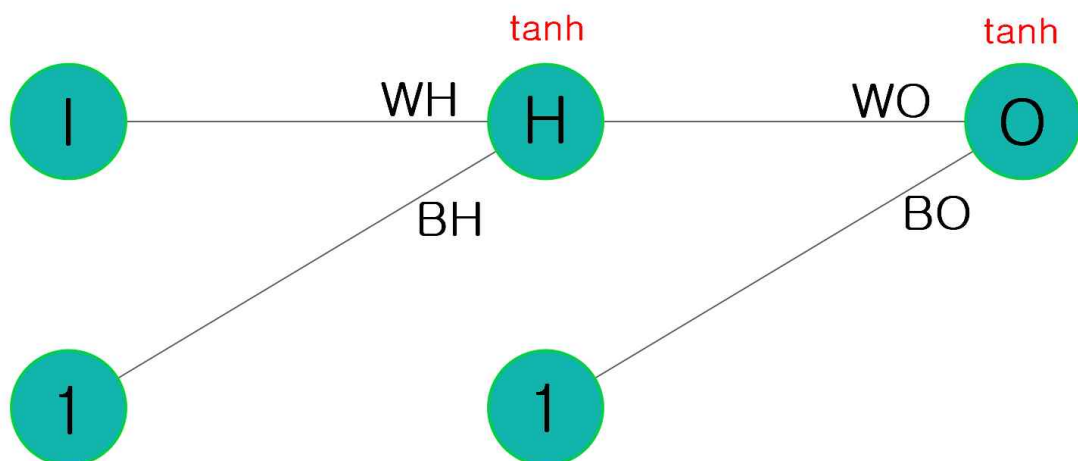
3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 199
0 = [[ 0.649  0.793]]
1/1 [=====]
WH =
[[ 0.149  0.249]
 [ 0.199  0.298]]
BH =
[ 0.337  0.334]
WO =
[[ 0.230  0.542]
 [ 0.279  0.592]]
BO =
[ 0.312  0.670]
```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## tanh 함수 적용해 보기

다음은 은닉 층과 출력 층에 tanh 활성화 함수를 적용해 봅니다. 다음 그림을 살펴봅니다.



❶ NumPy에 적용하기



먼저 NumPy 예제에 적용해 봅니다.

1. 이전 예제 416\_1.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다. # 표시된 부분은 추가되거나 수정된 부분을 나타냅니다.  
416\_3.py


```
01~17 # 이전 예제와 같습니다.
18     H = I @ WH + BH
19     H = np.tanh(H) #
20
21     O = H @ WO + BO
22     O = np.tanh(O) #
23
24     print(' O =', O)
25
26     E = np.sum((O - T) ** 2) / O.shape[1]
27
28     Ob = 2 * (O - T) / O.shape[1]
29     Ob = Ob*(1+O)*(1-O) #
30
31     Hb = Ob@WO.T
32     Hb = Hb*(1+H)*(1-H) #
33~끝 # 이전 예제와 같습니다.
```

19 : 은닉 층 H에 순전파 tanh 활성화 함수를 적용합니다.

22 : 출력 층 O에 순전파 tanh 활성화 함수를 적용합니다.

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 = 199
0 = [[ 0.149  0.802]]
WH =
[[ 0.146  0.245]
 [ 0.192  0.290]]
BH =
[[ 0.271  0.254]]
WO =
[[ 0.215  0.553]
 [ 0.262  0.604]]
BO =
[[ 0.011  0.771]]
```

O, WH, BH, WO, BO 값을 확인합니다.

## ② Tensorflow에 적용하기

다음은 Tensorflow 예제에 적용해 봅니다.

1. 이전 예제 416\_2.py를 복사합니다.


2. 다음과 같이 예제를 수정합니다.

416\_4.py

```
01~15 # 이전 예제와 같습니다.  
16 model = tf.keras.Sequential([  
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='tanh'),  
18     tf.keras.layers.Dense(2, activation='tanh')  
19 ])  
20~끝 # 이전 예제와 같습니다.
```

17 : 은닉 층에 활성화 함수 tanh를 적용합니다.

18 : 출력 층에 활성화 함수 tanh를 적용합니다.

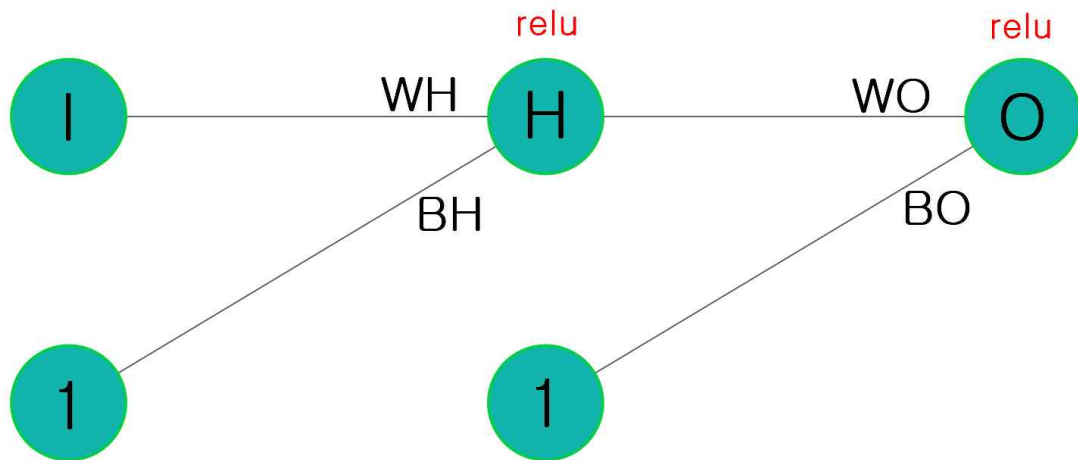
3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 199  
0 = [[ 0.149  0.802]]  
1/1 [=====]  
WH =  
[[ 0.146  0.245]  
 [ 0.192  0.290]]  
BH =  
[ 0.271  0.254]  
WO =  
[[ 0.215  0.553]  
 [ 0.262  0.604]]  
BO =  
[ 0.011  0.771]
```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## ReLU 함수 적용해 보기

다음은 은닉 층과 출력 층에 ReLU 활성화 함수를 적용해 봅니다. 다음 그림을 살펴봅니다.



## ❶ NumPy에 적용하기

먼저 NumPy 예제에 적용해 봅니다.

1. 이전 예제 416\_3.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다. # 표시된 부분은 추가되거나 수정된 부분을 나타냅니다.  
416\_5.py

```

01~17 # 이전 예제와 같습니다.
18     H = I @ WH + BH
19     H = (H>0)*H #
20
21     O = H @ WO + BO
22     O = (O>0)*O #
23
24     print(' O =', O)
25
26     E = np.sum((O - T) ** 2) / O.shape[1]
27
28     Ob = 2 * (O - T) / O.shape[1]
29     Ob = Ob*(O>0)*1 #
30
31     Hb = Ob@WO.T
32     Hb = Hb*(H>0)*1 #
33~끝 # 이전 예제와 같습니다.


```

19 : 은닉 층 H에 순전파 ReLU 활성화 함수를 적용합니다.

22 : 출력 층 O에 순전파 ReLU 활성화 함수를 적용합니다.

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 = 199
O = [[ 0.085  0.962]]
WH =
[[ 0.143  0.242]
 [ 0.186  0.284]]
BH =
[[ 0.213  0.188]]
WO =
[[ 0.219  0.527]
 [ 0.268  0.577]]
BO =
[[-0.029  0.704]]
```

O, WH, BH, WO, BO 값을 확인합니다.

## ❷ Tensorflow에 적용하기

다음은 Tensorflow 예제에 적용해 봅니다.

1. 이전 예제 416\_4.py를 복사합니다.


2. 다음과 같이 예제를 수정합니다.

416\_6.py

```
01~15 # 이전 예제와 같습니다.
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='relu'),
18     tf.keras.layers.Dense(2, activation='relu')
19 ])
20~끝 # 이전 예제와 같습니다.
```

17 : 은닉 층에 활성화 함수 relu를 적용합니다.

18 : 출력 층에 활성화 함수 relu를 적용합니다.

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

```

epoch = 199
0 = [[ 0.085  0.962]]
1/1 [=====]
WH =
[[ 0.143  0.242]
 [ 0.186  0.284]]
BH =
[ 0.213  0.188]
WO =
[[ 0.219  0.527]
 [ 0.268  0.577]]
BO =
[-0.029  0.704]

```

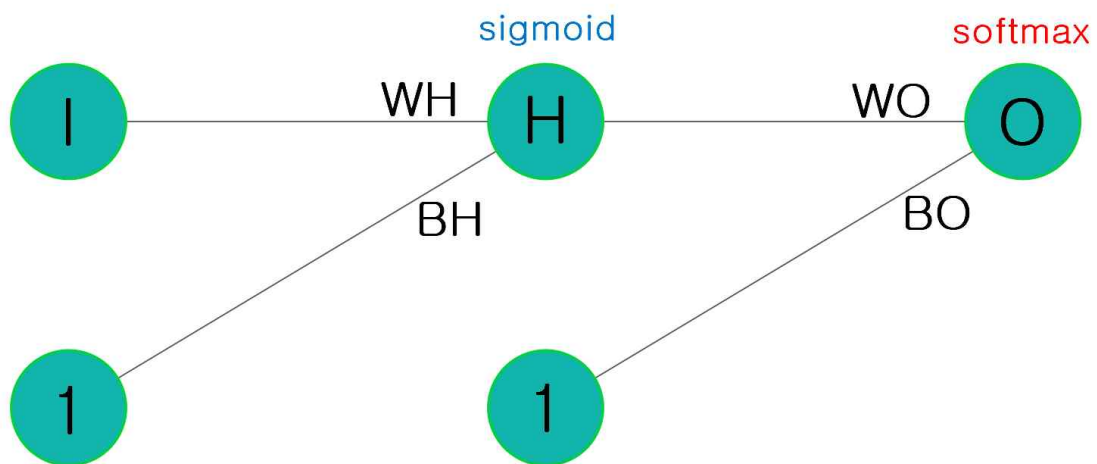
NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 07 출력 층에 softmax 함수 적용해 보기

여기서는 앞에서 NumPy와 Tensorflow를 이용해 구현한 인공 신경망의 출력 층에 소프트맥스 함수를 적용해 보면서 텐서플로우의 내부적인 동작을 이해해 봅니다.

### sigmoid와 softmax

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



소프트맥스 활성화 함수는 크로스 엔트로피 오차 함수와 같이 사용하며 일반적으로 목표 값은 0 또는 1의 값만 가지며, 총 합은 1이 됩니다. 그래서 목표 값을 다음과 같이 바꿔주도록 합니다. 다른 값들은 지금까지 사용한 값을 그대로 사용합니다.

$$\begin{aligned}
I &= \begin{bmatrix} .05 & .10 \end{bmatrix} \\
T &= \begin{bmatrix} 0 & 1 \end{bmatrix} \\
WH &= \begin{bmatrix} .15 & .25 \\ .20 & .30 \end{bmatrix} \\
BH &= \begin{bmatrix} .35 & .35 \end{bmatrix} \\
WO &= \begin{bmatrix} .40 & .50 \\ .45 & .66 \end{bmatrix} \\
BO &= \begin{bmatrix} .60 & .60 \end{bmatrix}
\end{aligned}$$

## ❶ NumPy에 적용하기

먼저 NumPy 예제에 적용해 봅니다.

1. 이전 예제 416\_5.py를 복사합니다.
2. 다음과 같이 예제를 수정합니다. # 표시된 부분은 추가되거나 수정된 부분을 나타냅니다.  
417\_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(200):
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  =', O)
26
27     E = np.sum(-T*np.log(O)) #
28
29     Ob = O - T #
30
31
32     Hb = Ob@WO.T
33     Hb = Hb*H*(1-H)
34~끝 # 이전 예제와 같습니다.

```

06 : 목표 값을 각각 0과 1로 변경합니다.

19 : 은닉 층 H에 순전파 sigmoid 활성화 함수를 적용합니다.

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.


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

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.

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

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

```

epoch = 199
O = [[ 0.136  0.864]]
WH =
[[ 0.152  0.252]
 [ 0.205  0.305]]
BH =
[[ 0.398  0.398]]
WO =
[[ 0.101  0.799]
 [ 0.149  0.851]]
BO =
[[ 0.100  1.100]]

```

O, WH, BH, WO, BO 값을 확인합니다.

## ② Tensorflow에 적용하기

다음은 Tensorflow 예제에 적용해 봅니다.

1. 이전 예제 416\_6.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

417\_2.py

```

01 import tensorflow as tf
02 import numpy as np
03
04 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
05
06 I = np.array([[.05, .10]])
07 T = np.array([[ 0, 1]])
08
09 WH = np.array([[.15, .25],
10                [.20, .30]])
11 BH = np.array([.35, .35])
12 WO = np.array([[.40, .50],
13                [.45, .55]])
14 BO = np.array([.60, .60])
15
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='sigmoid'),
18     tf.keras.layers.Dense(2, activation='softmax')
19 ])

```



```

20
21 model.layers[0].set_weights([WH, BH])
22 model.layers[1].set_weights([WO, BO])
23
24 model.compile(
25     optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
26     loss=tf.keras.losses.CategoricalCrossentropy())
27
28 for epoch in range(200):
29
30     print('epoch = %d' %epoch)
31
32     O = model.predict(I)
33     print(' O  =', O)
34
35     model.fit(I, T, epochs=1)
36     print(' WH =\n', model.layers[0].get_weights()[0])
37     print(' BH =\n', model.layers[0].get_weights()[1])
38     print(' WO =\n', model.layers[1].get_weights()[0])
39     print(' BO =\n', model.layers[1].get_weights()[1])


```

07 : 목표 값을 각각 0과 1로 변경합니다.

17 : 은닉 층에 활성화 함수 sigmoid를 적용합니다.

18 : 출력 층에 활성화 함수 softmax를 적용합니다.

26 : 손실 함수로는 크로스 엔트로피 오차 함수인 tf.keras.losses.CategoricalCrossentropy를 사용합니다.

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

```

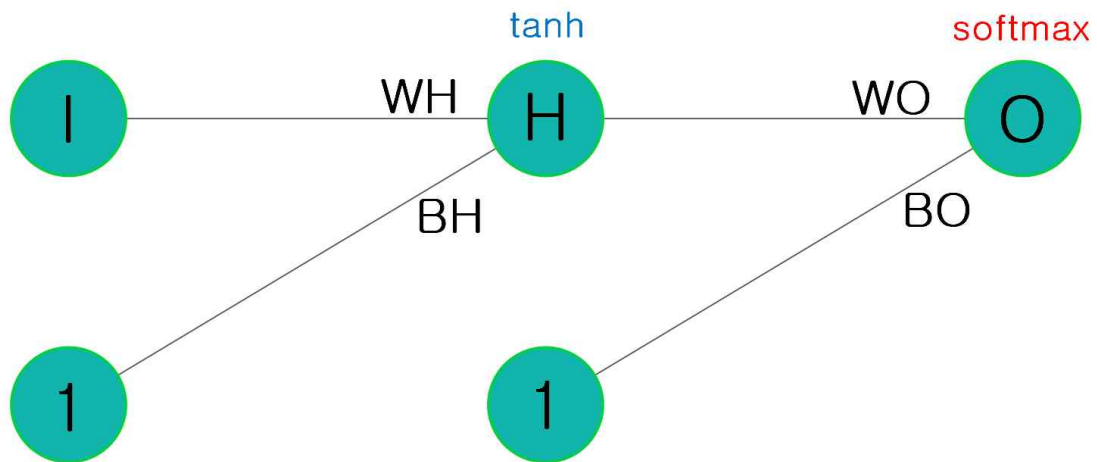
epoch = 199
O  = [[ 0.136  0.864]]
1/1 [=====]
WH =
[[ 0.152  0.252]
 [ 0.205  0.305]]
BH =
[ 0.398  0.398]
WO =
[[ 0.101  0.799]
 [ 0.149  0.851]]
BO =
[ 0.100  1.100]

```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## tanh와 softmax

여기서는 은닉 층은 tanh, 출력 층은 softmax 활성화 함수를 적용해 봅니다. 다음 그림을 살펴봅니다.



### ❶ NumPy에 적용하기

먼저 NumPy 예제에 적용해 봅니다.

1. 이전 예제 417\_1.py를 복사합니다.
2. 다음과 같이 예제를 수정합니다. # 표시된 부분은 추가되거나 수정된 부분을 나타냅니다. 417\_3.py

```
01~17 # 이전 예제와 같습니다.
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))
24
25     print(' O  =', O)
26
27     E = np.sum(-T*np.log(O))
28
29     Ob = O - T
```

```

30
31
32     Hb = Ob@WO.T
33     Hb = Hb*(1+H)*(1-H) #
34~끝 # 이전 예제와 같습니다.


```

19 : 은닉 층 H에 순전파 tanh 활성화 함수를 적용합니다.

22, 23 : 출력 층의 활성화 함수는 softmax입니다.

27 : 오차 계산은 크로스 엔트로피 오차 함수를 사용합니다.

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

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

```

epoch = 199
0  = [[ 0.158  0.842]]
WH =
[[ 0.157  0.257]
 [ 0.215  0.315]]
BH =
[[ 0.496  0.497]]
WO =
[[ 0.170  0.730]
 [ 0.213  0.787]]
BO =
[[ 0.038  1.162]]

```

O, WH, BH, WO, BO 값을 확인합니다.

## ② Tensorflow에 적용하기

다음은 Tensorflow 예제에 적용해 봅니다.

1. 이전 예제 417\_2.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

417\_4.py

```

01~15 # 이전 예제와 같습니다.
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='tanh'),
18     tf.keras.layers.Dense(2, activation='softmax')
19 ])
20
21 model.layers[0].set_weights([WH, BH])
22 model.layers[1].set_weights([WO, BO])

```


```

23
24 model.compile(
25     optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
26     loss=tf.keras.losses.CategoricalCrossentropy())
27~끝 # 이전 예제와 같습니다.

```

17 : 은닉 층에 활성화 함수 tanh를 적용합니다.

18 : 출력 층에 활성화 함수 softmax를 적용합니다.

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

```

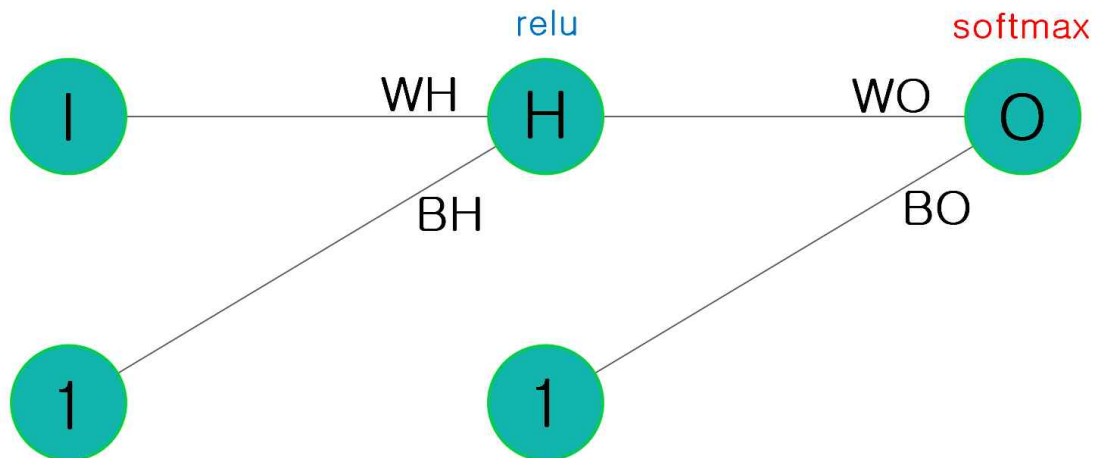
epoch = 199
0 = [[ 0.158  0.842]]
1/1 [=====]
WH =
[[ 0.157  0.257]
 [ 0.215  0.315]]
BH =
[ 0.496  0.497]
WO =
[[ 0.170  0.730]
 [ 0.213  0.787]]
BO =
[ 0.038  1.162]

```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## ReLU와 softmax

여기서는 은닉 층은 ReLU, 출력 층은 softmax 활성화 함수를 적용해 봅니다. 다음 그림을 살펴봅니다.




## ❶ NumPy에 적용하기

먼저 NumPy 예제에 적용해 봅니다.

1. 이전 예제 417\_3.py를 복사합니다.
2. 다음과 같이 예제를 수정합니다. # 표시된 부분은 추가되거나 수정된 부분을 나타냅니다.  
417\_5.py

```
01~17 # 이전 예제와 같습니다.
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))
24
25     print(' O  =', O)
26
27     E = np.sum(-T*np.log(O))
28
29     Ob = O - T
30
31
32     Hb = Ob@WO.T
33     Hb = Hb*(H>0)*1 #
34~끝 # 이전 예제와 같습니다.
```

- 19 : 은닉 층 H에 순전파 ReLU 활성화 함수를 적용합니다.  
22, 23 : 출력 층의 활성화 함수는 softmax입니다.  
27 : 오차 계산은 크로스 엔트로피 오차 함수를 사용합니다.  
33 : 역 은닉 층 Hb에 역전파 ReLU 활성화 함수를 적용합니다.

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

```

epoch = 199
O = [[ 0.146  0.854]]
WH =
[[ 0.159  0.259]
 [ 0.218  0.318]]
BH =
[[ 0.528  0.533]]
WO =
[[ 0.156  0.744]
 [ 0.197  0.803]]
BO =
[[ 0.053  1.147]]

```

O, WH, BH, WO, BO 값을 확인합니다.

## ② Tensorflow에 적용하기

다음은 Tensorflow 예제에 적용해 봅니다.

1. 이전 예제 417\_4.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

417\_6.py


```

01~15 # 이전 예제와 같습니다.
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='relu'),
18     tf.keras.layers.Dense(2, activation='softmax')
19 ])
20
21 model.layers[0].set_weights([WH, BH])
22 model.layers[1].set_weights([WO, BO])
23
24 model.compile(
25     optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
26     loss=tf.keras.losses.CategoricalCrossentropy())
27~끝 # 이전 예제와 같습니다.

```

17 : 은닉 층에 활성화 함수 ReLU를 적용합니다.

18 : 출력 층에 활성화 함수 softmax를 적용합니다.

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

```

epoch = 199
0 = [[ 0.146  0.854]]
1/1 [=====]
WH =
[[ 0.159  0.259]
 [ 0.218  0.318]]
BH =
[ 0.528  0.533]
WO =
[[ 0.156  0.744]
 [ 0.197  0.803]]
BO =
[ 0.053  1.147]

```

NumPy 예제와 결과가 같은지 비교해 봅니다. NumPy와 결과가 같은 것을 볼 수 있습니다.

## 08 GradientTape 사용해 보기

GradientTape은 tensorflow에서 제공하는 기능으로 인공 신경망 학습 과정을 좀 더 세세하게 제어하고자 할 경우에 사용합니다. 여기서는 GradientTape 기능을 이용하여 예제를 작성해 봅니다.

1. 이전 예제 417\_6.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

418\_1.py

```

01 import tensorflow as tf
02 import numpy as np
03
04 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
05
06 I = np.array([[.05, .10]])
07 T = np.array([[ 0, 1]])
08
09 WH = np.array([[.15, .25],
10                [.20, .30]])
11 BH = np.array([.35, .35])
12 WO = np.array([[.40, .50],
13                [.45, .55]])
14 BO = np.array([.60, .60])
15

```

```

16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='relu'),
18     tf.keras.layers.Dense(2, activation='softmax')
19 ])
20
21 model.layers[0].set_weights([WH, BH])
22 model.layers[1].set_weights([WO, BO])
23
24 optimizer=tf.keras.optimizers.SGD(learning_rate=0.01)
25 loss=tf.keras.losses.CategoricalCrossentropy()
26
27 for epoch in range(200):
28
29     print('epoch = %d' %epoch)
30
31     with tf.GradientTape() as tape:
32
33         O = model(I)
34         print(' O =', O.numpy())
35         E = loss(T, O)
36
37     gradients = tape.gradient(E, model.trainable_variables)
38
39     optimizer.apply_gradients(zip(gradients, model.trainable_variables))
40     print(' WH =\n', model.layers[0].get_weights()[0])
41     print(' BH =\n', model.layers[0].get_weights()[1])
42     print(' WO =\n', model.layers[1].get_weights()[0])
43     print(' BO =\n', model.layers[1].get_weights()[1])

```

01~22 : 이전 예제와 같습니다.

24 : 최적화 함수를 설정합니다.

25 : 손실 함수를 설정합니다.

31 : 미분을 위한 GradientTape를 적용합니다.


33 : 순전파를 수행합니다.

35 : 오차를 계산합니다.

37 : 가중치와 편향의 역전파 오차 값을 구합니다.

39 : 가중치와 편향의 역전파 오차 값을 적용합니다.

40~43 : 갱신된 가중치와 편향 값을 출력합니다.

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



```

epoch = 199
O = [[ 0.146  0.854]]
WH =
[[ 0.159  0.259]
 [ 0.218  0.318]]
BH =
[ 0.528  0.533]
WO =
[[ 0.156  0.744]
 [ 0.197  0.803]]
BO =
[ 0.053  1.147]

```

이전 예제와 결과를 비교합니다.

## 역전파 오차 살펴보기

GradientTape를 이용하면 가중치와 편향의 역전파 오차 값을 확인해 볼 수 있습니다. 여기서는 이전 예제를 수정하여 가중치와 편향의 역전파 오차 값을 확인해봅니다.

1. 이전 예제 418\_1.py를 복사합니다.

2. 다음과 같이 예제를 수정합니다.

418\_2.py

```

01 import tensorflow as tf
02 import numpy as np
03
04 np.set_printoptions(formatter={'float_kind':lambda x: "{0:6.3f}".format(x)})
05
06 I = np.array([[.05, .10]])
07 T = np.array([[ 0, 1]])
08
09 WH = np.array([[.15, .25],
10                [.20, .30]])
11 BH = np.array([.35, .35])
12 WO = np.array([[.40, .50],
13                [.45, .55]])
14 BO = np.array([.60, .60])
15
16 model = tf.keras.Sequential([
17     tf.keras.layers.Dense(2, input_shape=(2,), activation='relu'),
18     tf.keras.layers.Dense(2, activation='softmax')


```

```

19 ])
20
21 model.layers[0].set_weights([WH, BH])
22 model.layers[1].set_weights([WO, BO])
23
24 optimizer=tf.keras.optimizers.SGD(learning_rate=0.01)
25 loss=tf.keras.losses.CategoricalCrossentropy()
26
27 for epoch in range(200):
28
29     print('epoch = %d' %epoch)
30
31     with tf.GradientTape() as tape:
32
33         O = model(I)
34         print(' O  =', O.numpy())
35         E = loss(T, O)
36
37         gradients = tape.gradient(E, model.trainable_variables)
38         print(' WHb =\n', gradients[0].numpy())
39         print(' BHb =\n', gradients[1].numpy())
40         print(' WOb =\n', gradients[2].numpy())
41         print(' BOb =\n', gradients[3].numpy())
42
43         optimizer.apply_gradients(zip(gradients, model.trainable_variables))
44         print(' WH =\n', model.layers[0].get_weights()[0])
45         print(' BH =\n', model.layers[0].get_weights()[1])
46         print(' WO =\n', model.layers[1].get_weights()[0])
47         print(' BO =\n', model.layers[1].get_weights()[1])

```

38~41 : 가중치와 편향의 역전파 오차 값을 출력합니다.

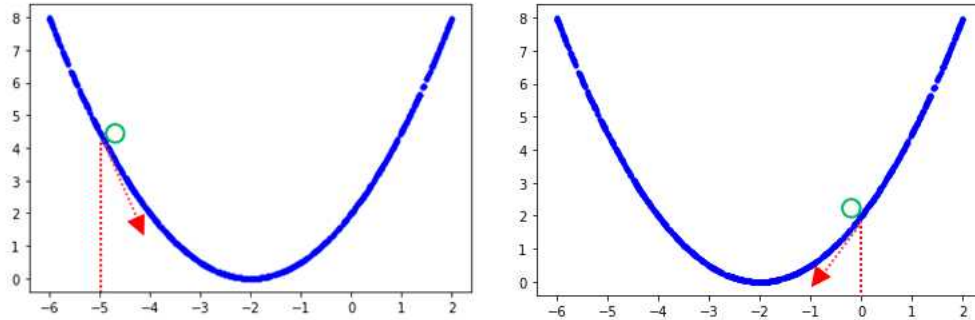
3.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
epoch = 199
O = [[ 0.146  0.854]]
WHb =
[[-0.004 -0.004]
 [-0.009 -0.009]]
BHb =
[-0.085 -0.088]
WOb =
[[ 0.081 -0.081]
 [ 0.084 -0.084]]
BOb =
[ 0.146 -0.146]
WH =
[[ 0.159  0.259]
 [ 0.218  0.318]]
BH =
[ 0.528  0.533]
WO =
[[ 0.156  0.744]
 [ 0.197  0.803]]
BO =
[ 0.053  1.147]
```

WHb, BHb, WOb, BOb 값을 확인합니다.

## 02 경사 하강 법 이해하기

여기서는 인공 신경망 학습에 아주 중요한 요소인 경사 하강 법에 대해 자세히 살펴봅니다. 경사 하강 법은 경사 방향으로 하강하는 방법이라는 의미입니다. 다음 그림과 같이 공(초록색 원)이 기운 방향으로 이동하여 단계적으로 최저점에 도달하는 원리를 수식으로 표현한 것을 경사 하강 법이라고 합니다. 경사 하강 법에 대해 예제를 통해 이해해 보도록 합니다.



### 01 오차 함수 그래프 그려보기

다음은 경사 하강 법을 나타내는 수식으로 p의 최적화 함수라고도 합니다.

$$p = p - \alpha \frac{dE}{dp}$$

이전 예제에서 우리는 다음 수식을 구현하였습니다.

$$E = \frac{1}{2}(p - t)^2, (p: \text{prediction}, t: \text{target})$$

이 식에서는 p의 값이 t의 값에 가까울수록 E의 값은 0에 가까워집니다. 이전 예제에서 t의 값은 -2.0이었으므로 이 식은 다음과 같이 변경할 수 있습니다.

$$E = \frac{1}{2}(p + 2)^2$$

이 수식을 p값이 -6에서 2사이 범위에서 그래프를 그려보도록 합니다.

1. 다음과 같이 예제를 작성합니다.

421\_1.py

```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
```

```

04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()

```

01 : import문을 이용하여 numpy 모듈을 np라는 이름으로 불러옵니다. 07, 09, 11 줄에서 사용합니다.

02 : import문을 이용하여 time 모듈을 불러옵니다. 07줄에서 임의 숫자(난수) 생성 초기화에 사용합니다.

03 : import문을 이용하여 matplotlib.pyplot 모듈을 plt라는 이름으로 불러옵니다. 여기서는 matplotlib.pyplot 모듈을 이용하여 13, 14줄에서 그래프를 그립니다.

05 : NUM\_SAMPLES 변수를 생성한 후, 1000으로 초기화합니다. NUM\_SAMPLES 변수는 생성할 데이터의 개수 값을 가지는 변수입니다.

07 : np.random.seed 함수를 호출하여 임의 숫자 생성을 초기화합니다. time.time 함수를 호출하여 현재 시간을 얻어낸 후, 정수 값으로 변환하여 np.random.seed 함수의 인자로 줍니다. 이렇게 하면 현재 시간에 맞춰 임의 숫자 생성이 초기화됩니다.

09 : np.random.uniform 함수를 호출하여 (-6, 2) 범위에서 NUM\_SAMPLES 만큼의 임의 값을 차례대로 고르게 추출하여 ps 변수에 저장합니다.


11 : 다음 식을 이용하여 추출된 ps 값에 해당하는 es 값을 계산합니다. es 값도 NUM\_SAMPLES 개수만큼 추출됩니다.

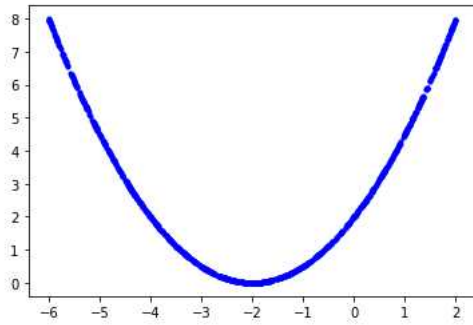
$$E = \frac{1}{2}(p+2)^2$$

파이썬에서 \*는 곱셈기호, \*\*는 거듭제곱기호를 나타냅니다.

13 : plt.plot 함수를 호출하여 ps, es 좌표 값에 맞추어 그래프를 내부적으로 그립니다. 그래프의 색깔은 파란색으로 그립니다. 'b.'은 파란색을 의미합니다.

14 : plt.show 함수를 호출하여 화면에 그래프를 표시합니다.

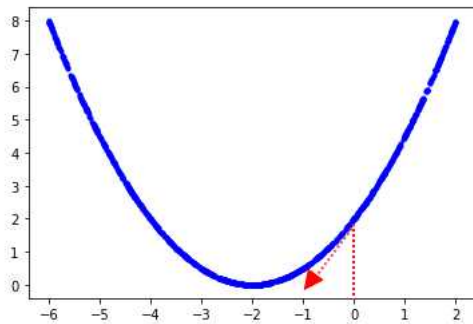
2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.



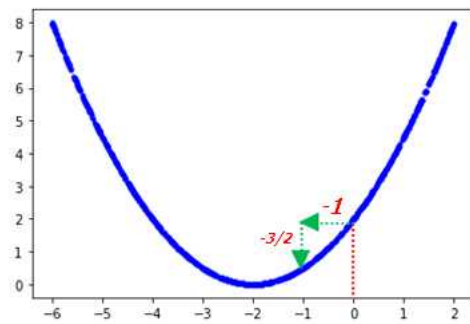
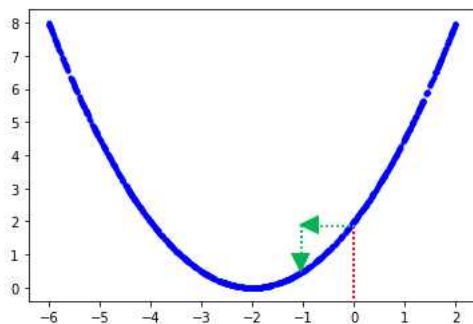
이 그래프에서  $p$ 값이  $-2$ 일 때  $E$ 는 최소값  $0$ 이 됩니다.

## 02 왼쪽 이동 경사 하강 법

다음 그림을 살펴봅시다. 이전 예제에서  $p$ 값은  $0$ 이었습니다.  $E$ 값이 최소가 되기 위해서  $p$ 값은  $0$ 에서  $-2$ 로 옮겨가야 합니다. 그러기 위해서  $p$ 값은 기울어진 방향으로 이동하여야 합니다. 이 그림에서  $p$ 값은 왼쪽으로 이동하여야 합니다.



다음 그림에서  $p$ 값이 왼쪽으로 이동하면  $E$ 값은 아래로 이동합니다.



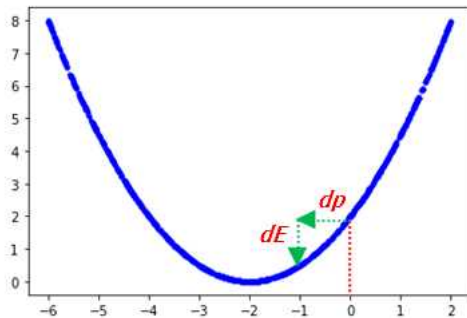
예를 들어,  $p$ 값이 현재 위치  $0$ 에서  $-1$ (왼쪽으로  $1$ )만큼 이동하면  $E$ 값은  $-3/2$ (아래로  $3/2$ )만큼 이동합니다. 다음은 이 과정에 대한 수식입니다.

$$E_0 = \frac{1}{2}(0+2)^2 = 2$$

$$E_{-1} = \frac{1}{2}(-1+2)^2 = \frac{1}{2}$$

$$E_{-1} - E_0 = \frac{1}{2} - 2 = -\frac{3}{2}$$

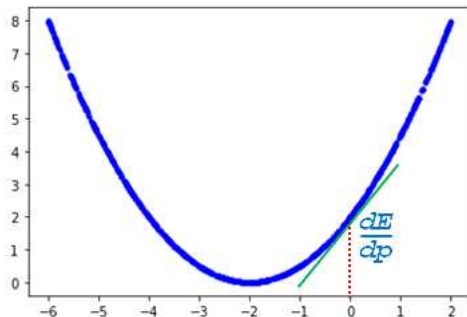
p값이 왼쪽으로 아주 조금 이동하면 E값은 아래로 아주 조금 이동합니다. 이것을 그림으로 나타내면 다음과 같습니다.



dp에 대한 dE의 비율을 다음과 같이 표현합니다.

$$\frac{dE}{dp}$$

이 표현법은 p값이 아주 작게 변할 때, E값이 아주 작게 변하는 비율을 나타냅니다. 이 표현법을 E의 p에 대한 기울기 또는 경사도라고 합니다. 다음 그림은 p값이 0일 때 해당 위치에서의 기울기를 나타냅니다.



이 그림의 경우 기울기

$$\frac{dE}{dp} > 0 \quad \frac{dE}{dp} > 0$$

입니다. 일 경우 p값은 현재 위치에서 왼쪽으로 이동해야 E값이 작아집니다. 이 경우 다음 수식에 의해 p 값은 기울어진 방향(왼쪽 방향)으로 이동합니다.

$$p = p - \frac{dE}{dp}$$

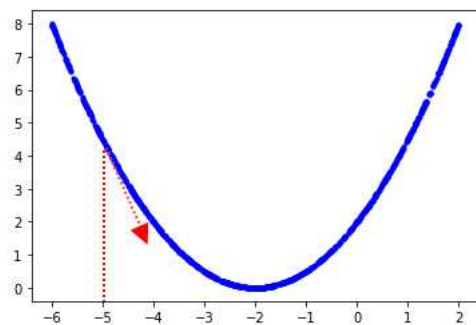
즉, 새로운 p값은 이전 p값보다 작아집니다. p의 이동정도를 조절하기 위해서 다음과 같이 수식을 변경할 수 있습니다.

$$p = p - \alpha \frac{dE}{dp}$$

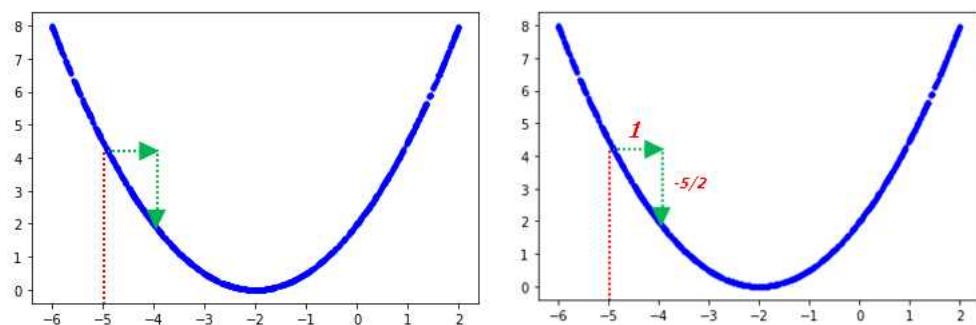
이 수식을 p의 최적화 함수라고 합니다.  $\alpha$  는 학습률(learning rate)라고 하며, p값의 이동정도를 조절합니다.  $\alpha$  는 보통 0.01~0.001 정도에서 실험적으로 적당한 값을 사용합니다.

### 03 오른쪽 이동 경사 하강 법

다음 그림을 살펴봅시다. 이번엔 p값이 -5라고 가정해 봅시다. E값이 최소가 되기 위해서 p값은 -5에서 -2로 옮겨가야 합니다. 그러기 위해서 p값은 기울어진 방향으로 이동하여야 합니다. 이 그림에서 p값은 오른쪽으로 이동하여야 합니다.



다음 그림에서 p값이 오른쪽으로 이동하면 E값은 아래로 이동합니다.



예를 들어, p값이 현재 위치 -5에서 +1(오른쪽으로 1)만큼 이동하면 E값은 -5/2(아래로 5/2)만큼 이동합니다. 다음은 이 과정에 대한 수식입니다.

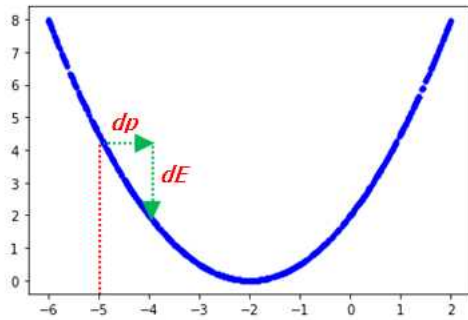


$$E_{-5} = \frac{1}{2}(-5+2)^2 = \frac{9}{2}$$

$$E_{-4} = \frac{1}{2}(-4+2)^2 = 2$$

$$E_{-4} - E_{-5} = 2 - \frac{9}{2} = -\frac{5}{2}$$

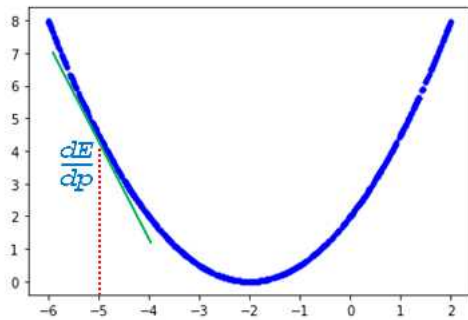
p값이 오른쪽으로 아주 조금 이동하면 E값은 아래로 아주 조금 이동합니다. 이것을 그림으로 나타내면 다음과 같습니다.



dp에 대한 dE의 비율을 다음과 같이 표현합니다.

$$\frac{dE}{dp}$$

이 표현법은 p값이 아주 작게 변할 때, E값이 아주 작게 변하는 비율을 나타냅니다. 이 표현법을 E의 p에 대한 기울기 또는 경사도라고 합니다. 다음 그림은 p값이 -5일 때 해당 위치에 서의 기울기를 나타냅니다.



이 그림의 경우 기울기

$\frac{dE}{dp} < 0$  입니다.  $\frac{dE}{dp} < 0$  일 경우 p값은 현재 위치에서 오른쪽으로 이동해야 E값이 작아집니다. 이 경우 다음 수식에 의해 p 값은 기울어진 방향(오른쪽 방향)으로 이동합니다.

$$p = p - \frac{dE}{dp}$$

즉, 새로운 p값은 이전 p값보다 커집니다. p의 이동정도를 조절하기 위해서 다음과 같이 수식을 변경할 수 있습니다.

$$p = p - \alpha \frac{dE}{dp}$$

이 수식을 p의 최적화 함수라고 합니다.  $\alpha$ 는 학습률(learning rate)라고 하며, p값의 이동정도를 조절합니다.

## 04 오차 함수 기울기 구하기

다음 수식에 대해

$$E = \frac{1}{2}(p+2)^2$$

$\frac{dE}{dp}$ 는 다음과 같습니다.

$$\frac{dE}{dp} = \frac{d(\frac{1}{2}(p+2)^2)}{dp} = \frac{1}{2}2(p+2) = p+2$$

그래서 p의 최적화 함수는 다음과 같습니다.

$$\begin{aligned} p &= p - \alpha \frac{dE}{dp} \\ &= p - \alpha(p+2) \end{aligned}$$

## 최적화 함수 왼쪽 이동하기

그러면 최적화 함수를 이용하여 p값이 0에서 -2로 가까워지는 예제를 수행해 봅니다.

1. 다음과 같이 이전 예제를 수정합니다.

424\_1.py

```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
16 p = 0
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 DpE = p+2
21 p = p - lr*DpE
22 print('p : ', p)
```

16 : p 변수를 선언한 후, 0으로 초기화합니다.

17 : 다음 수식을 구현합니다.

$$E = \frac{1}{2}(p+2)^2$$

18 : lr 변수를 선언한 후, 0.5로 초기화합니다. lr은 학습률을 나타냅니다. 이 값에 따라 학습의 정도가 빠르거나 늦어집니다.


20 : 다음 수식을 구현합니다.

$$\frac{dE}{dp} = p+2$$

21 : 다음 수식을 구현합니다.

$$p = p - \alpha \frac{dE}{dp}$$

22 : print 함수를 호출하여 왼쪽으로 이동한 p값을 출력해 봅니다.

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

```
p : -1.0
```

p값이 0에서 -1로 이동한 것을 확인합니다.

## 최적화 함수 반복 적용해 보기


여기서는 이 과정을 2회 수행해 보도록 합니다.

1. 다음과 같이 예제를 수정합니다.

424\_2.py

```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
16 p = 0
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 for i in range(2):
21     DpE = p+2
22     p = p - lr*DpE
23     print('p : ', p)
```

20 : for 문을 이용하여 21~23 과정을 2회 수행합니다.

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

```
p : -1.0
```


```
p : -1.5
```

p값이 -1.0에서 -1.5로 한 번 더 이동한 것을 확인합니다.

3. 다음과 같이 예제를 수정합니다.

```
20 for i in range(4):
```

20 : for 문을 이용하여 4회 수행합니다.

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


```
p : -1.0  
p : -1.5  
p : -1.75  
p : -1.875
```

p값이 -1.875까지 이동한 것을 확인합니다.

5. 다음과 같이 예제를 수정합니다.

```
20 for i in range(8):
```

20 : for 문을 이용하여 21~23 과정을 8회 수행합니다.

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


```
p : -1.0  
p : -1.5  
p : -1.75  
p : -1.875  
p : -1.9375  
p : -1.96875  
p : -1.984375  
p : -1.9921875
```

p값이 -1.9921875까지 이동한 것을 확인합니다.

7. 다음과 같이 예제를 수정합니다.

```
20 for i in range(16):
```

20 : for 문을 이용하여 21~23 과정을 16회 수행합니다.

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

```
p : -1.0
p : -1.5
p : -1.75
p : -1.875
p : -1.9375
p : -1.96875
p : -1.984375
p : -1.9921875
p : -1.99609375
p : -1.998046875
p : -1.9990234375
p : -1.99951171875
p : -1.999755859375
p : -1.9998779296875
p : -1.99993896484375
p : -1.999969482421875
```

p값이 -1.999969까지 이동한 것을 확인합니다.

## 최적화 함수 오른쪽 이동하기

이번엔 최적화 함수를 이용하여 p값이 -5에서 -2로 가까워지는 예제를 수행해 봅니다.

1. 다음과 같이 예제를 수정합니다.

424\_3.py


```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
```

```

16 p = -5
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 for i in range(16):
21     DpE = p+2
22     p = p - lr*DpE
23     print('p : ', p)

```

16 : p값을 -5로 변경합니다.

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

```

p : -3.5
p : -2.75
p : -2.375
p : -2.1875
p : -2.09375
p : -2.046875
p : -2.0234375
p : -2.01171875
p : -2.005859375
p : -2.0029296875
p : -2.00146484375
p : -2.000732421875
p : -2.0003662109375
p : -2.00018310546875
p : -2.000091552734375
p : -2.0000457763671875

```

p값이 -2.0000까지 이동한 것을 확인합니다.

\*\*\* 경사 하강 법은 기울기 하강 법, 미분 하강 법이라고도 할 수 있습니다.

## 03 확장 경사 하강 법

앞에서 우리는 다음과 같은 수식의 경사 하강 법을 살펴 보았습니다.

$$p = p - \alpha \frac{dE}{dp}$$

이 수식은 기울어진 방향으로 기울어진 정도에 따라 이동하겠다는 단순한 방법입니다. 이 식은 기울어진 정도가 클수록 더 많이 이동하게 됩니다. 최종적으로 기울기  $\frac{dE}{dp}$ 가 0이 되는 지점에서 멈추게 됩니다. 여기서는 이 수식을 확장한 형태의 수식을 살펴봅니다. 확장된 수식은 모멘텀, AdaGrad, RMSProp, Adam 등이 있습니다. 일반적으로 Adam이 가장 많이 사용되는 최적화 수식입니다. 지금부터 이 수식들을 살펴보고 구현해 봅니다.

### 01 Momentum

모멘텀의 수식은 다음과 같습니다.

$$v = mv - \alpha \frac{dE}{dp}$$
$$p = p + v$$

모멘텀은 오차의 최소 위치를 찾기 위해 운동량 개념을 적용한 최적화 방법입니다. 이 수식에서  $v$ 는 이동 속도를 의미합니다. 즉, 경사에 의해서 속도가 점점 더해지는 상황을 고려한 최적화 방법입니다. 갱신되는 위치 값은 경사에 의한 속도 값이 더해지는 형태로 갱신됩니다.  $m$ 은 momentum을 의미하며, 속도를 얼마나 고려할지를 정하게 됩니다.  $m$ 은 일반적으로 0.9 정도의 값을 사용합니다.  $m$ 이 0일 경우 앞에서 본 기본적인 경사하강 법과 같은 수식입니다. 모멘텀은 현재 속도에 기울기가 더해지기 때문에 최소 위치를 찾는 속도가 더 빠릅니다. 또 찾는 속도가 점점 더 빨라지게 됩니다.

1. 다음과 같이 예제를 수정합니다.

431\_1.py

```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
```



```

07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
16 p = -5
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 v = 0
21 m = 0.5
22 for i in range(16):
23     DpE = p+2
24     v = m*v - lr*DpE
25     p = p + v
26     print('p : ', p)

```


20 : v 변수를 선언한 후, 0으로 초기화합니다. 모멘텀의 초기 속도 값을 0으로 초기화합니다.

21 : m 변수를 선언한 후, 0.5로 초기화합니다. m 변수는 24줄에서 속도 v에 곱해져 속도를 얼마나 고려할지 결정하는 인자입니다.

24, 25 : 다음 모멘텀 수식을 구현합니다.

$$v = mv - \alpha \frac{dE}{dp}$$

$$p = p + v$$

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

```

p : -3.5
p : -2.0
p : -1.25
p : -1.25
p : -1.625
p : -2.0
p : -2.1875
p : -2.1875
p : -2.09375
p : -2.0
p : -1.953125
p : -1.953125
p : -1.9765625
p : -2.0
p : -2.01171875
p : -2.01171875

```

p값이 -2를 중심으로 좌우로 흔들리면서 -2에 가까워지는 것을 확인합니다.

## 02 AdaGrad

AdaGrad의 수식은 다음과 같습니다.

$$h = h + \left(\frac{dE}{dp}\right)^2$$

$$p = p - \frac{\alpha}{\sqrt{h}} \frac{dE}{dp}$$

$h$ 는 경사도를 제곱하여 계속 더해준 값입니다. 즉, 학습 과정에서 얻게 된 경사도 제곱  $\left(\frac{dE}{dp}\right)^2$ 의 합입니다.  $\sqrt{h}$ 는 현재 위치까지 누적된 경사도의 합을 반영합니다. 경사가 있는 동안에  $\sqrt{h}$  값은 계속해서 커지게 되며, 경사도가 0에 가까워질수록  $h$  값은 일정한 값으로 수렴하게 됩니다.  $\frac{\alpha}{\sqrt{h}}$ 는 학습률  $\alpha$ 를  $\sqrt{h}$ 로 나누어 학습이 많이 진행될수록 학습률이 작아지도록 조절하는 역할을 합니다. 그래서 학습이 진행될수록 최소 위치를 찾는 속도는 점점 느려지게 됩니다.

1. 다음과 같이 예제를 수정합니다.


432\_1.py

```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
16 p = -5
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 h = 0
21 for i in range(16):
22     DpE = p+2
23     h = h + DpE*DpE
24     p = p - lr*1/np.sqrt(h)*DpE
25     print('p :', p)
```

20 : h 변수를 선언한 후, 0으로 초기화합니다. AdaGrad의 기울기 제곱의 누적 값을 0으로 초기화합니다.

23, 24 : 다음 AdaGrad 수식을 구현합니다.

$$h = h + \left(\frac{dE}{dp}\right)^2$$
$$p = p - \frac{\alpha}{\sqrt{h}} \frac{dE}{dp}$$

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

```

p : -4.5
p : -4.17990780016776
p : -3.936198871936842
p : -3.73755267275177
p : -3.5696397289117936
p : -3.42448608972912
p : -3.297102434927234
p : -3.184110572800342
p : -3.083088830988888
p : -2.9922243398489448
p : -2.910112967319001
p : -2.835636906159039
p : -2.767886021231685
p : -2.706105258180596
p : -2.6496582908584156
p : -2.5980016754318234

```

p값이 -2로 가까워지는 것을 확인합니다. 또, -2에 가까워질수록 속도가 늦어지는 것을 확인합니다.

### 03 RMSProp

RMSProp의 수식은 다음과 같습니다.

$$h = \rho h + (1 - \rho) \left( \frac{dE}{dp} \right)^2$$

$$p = p - \frac{\alpha}{\sqrt{h}} \frac{dE}{dp}$$

RMSProp은 AdaGrad를 개선한 방법으로 경사도 제곱의 합  $h$ 와 현재 경사도 제곱  $\left( \frac{dE}{dp} \right)^2$ 을 비율적으로 선택할 수 있도록 합니다. 수식에서  $\rho$ 값을 조절하여 현재 경사도의 제곱의 비율을 많이 주거나 적게 줄 수 있습니다. 예를 들어,  $\rho$ 값이 작을수록  $h$ 에 현재 경사도를 크게 반영하게 됩니다.  $h$ 에 현재 경사도가 크게 반영될 경우  $\sqrt{h}$ 도 현재 경사도를 크게 반영하게 됩니다. 그래서 학습률  $\alpha$ 를  $\sqrt{h}$ 로 나누어 학습률이 작아지도록 조종하는 역할을 합니다.

1. 다음과 같이 예제를 수정합니다.

433\_1.py


```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
16 p = -5
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 h = 0
21 lo = 0.9
22 for i in range(16):
23     DpE = p+2
24     h = lo*h + (1-lo)*DpE*DpE
25     p = p - lr*1/np.sqrt(h)*DpE
26     print('p :', p)
```

20 : h 변수를 선언한 후, 0으로 초기화합니다.

21 : lo 변수를 선언한 후, 0.9로 초기화합니다.

24, 25 : 다음 RMSProp 수식을 구현합니다.

$$h = \rho h + (1 - \rho) \left( \frac{dE}{dp} \right)^2$$
$$p = p - \frac{\alpha}{\sqrt{h}} \frac{dE}{dp}$$

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

```

p : -3.41886116991581
p : -2.7134110396666205
p : -2.349554711258639
p : -2.1629433880058793
p : -2.0714037348256906
p : -2.029135223319893
p : -2.0109564566037306
p : -2.003750520229244
p : -2.0011504207124537
p : -2.000309734675076
p : -2.0000711482607567
p : -2.0000133787199332
p : -2.000001928128071
p : -2.0000001886138024
p : -2.0000000092460635
p : -1.99999999977627

```

p값이 -2로 가까워지는 것을 확인합니다.

## 04 Adam

Adam의 수식은 다음과 같습니다.

$$\begin{aligned}
 m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \frac{dE}{dp} \\
 v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \left( \frac{dE}{dp} \right)^2 \\
 M_t &= \frac{m_t}{1 - \beta_1^t} \\
 V_t &= \frac{v_t}{1 - \beta_2^t} \\
 p &= p - \frac{\alpha}{\sqrt{V_t} + \epsilon} M_t
 \end{aligned}$$

Adam은 모멘텀과 AdaGrad를 조합한 방법입니다. 앞에서 모멘텀은 새로운 인자로 속도 개념을 갖는  $v$ 를, AdaGrad는 가속도 제곱의 누적  $h$ 를 추가하여 최적화를 수행했는데, Adam은 두 방법을 조합하고,  $v$ ,  $h$ 가 각각 처음에 0으로 설정되어 학습 초반에 0으로 편향되는 문제를 해결하는 방법을 더하여 만들어진 방법입니다. 수식에서  $m_t$ 는 모멘텀의  $v$ 를,  $v_t$ 는 AdaGrad

의  $h$ 를 의미합니다.  $M_t$ ,  $V_t$ 는 각각 0에 편향되지 않게 하는 역할을 합니다. 보통  $\beta_1$ 로는 0.9,  $\beta_2$ 로는 0.999,  $\epsilon$ 으로는  $10^{-8}$  정도의 값을 사용합니다.

1. 다음과 같이 예제를 수정합니다.

434\_1.py

```
01 import numpy as np
02 import time
03 import matplotlib.pyplot as plt
04
05 NUM_SAMPLES = 1000
06
07 np.random.seed(int(time.time()))
08
09 ps = np.random.uniform(-6, 2, NUM_SAMPLES)
10
11 es = 0.5*(ps+2)**2
12
13 plt.plot(ps, es, 'b.')
14 plt.show()
15
16 p = -5
17 E = 0.5*(p+2)**2
18 lr = 0.5
19
20 m = 0
21 v = 0
22 t = 0
23 beta_1 = 0.9
24 beta_2 = 0.999
25 eps = 10**-8
26 for i in range(16):
27     t = t + 1
28     DpE = p+2
29     m = beta_1*m + (1-beta_1)*DpE
30     v = beta_2*v + (1-beta_2)*DpE*DpE
31     M = m/(1-beta_1**t)
32     V = v/(1-beta_2**t)
33     p = p - lr*M/(np.sqrt(V)+eps)
34     print('p :', p)
```

20, 21 : m, v 변수를 선언한 후, 0으로 초기화합니다.

22 : t 변수를 선언한 후, 0으로 초기화합니다. t 변수는 28줄에서 최적화 단계를 기록하기 위해 사용합니다.


23, 24 : beta\_1, beta\_2 변수를 선언한 후, 0으로 초기화합니다.

25 : eps 변수를 선언한 후,  $10^{-8}$ 으로 초기화합니다.

27 : t 변수를 1씩 증가시켜 단계를 기록합니다.

29~33 : 다음 Adam 수식을 구현합니다.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{dE}{dp}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left( \frac{dE}{dp} \right)^2$$
$$M_t = \frac{m_t}{1 - \beta_1^t}$$
$$V_t = \frac{v_t}{1 - \beta_2^t}$$
$$p = p - \frac{\alpha}{\sqrt{V_t} + \epsilon} M_t$$

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

```
p : -4.500000001666667
p : -4.0044135661447235
p : -3.51770813287177
p : -3.0459154724609996
p : -2.596907461354961
p : -2.180399226763144
p : -1.8073832706986472
p : -1.4887778921304025
p : -1.2334779898811434
p : -1.046552266834851
p : -0.9284485074849226
p : -0.875461954432766
p : -0.8809741868124046
p : -0.9367635792767339
p : -1.0339729342963133
p : -1.163663645331026
```

p값이 -2를 중심으로 좌우로 흔들리면서 -2에 가까워지는 것을 확인합니다.



지금까지 확장 경사 하강 법으로 모멘텀, AdaGrad, RMSProp, Adam을 살펴보았습니다. 일반적으로 Adam이 가장 많이 사용되는 최적화 함수입니다.