인공지능

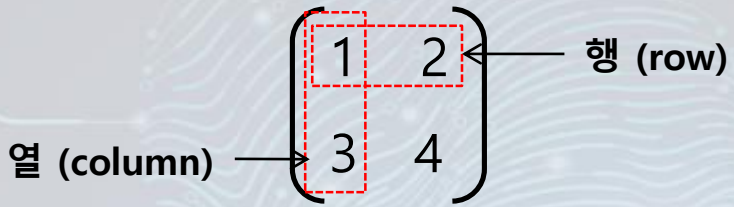
딥러닝 기초_손실함수_경사하강법

강성관 (silicon1@hanmail.net)

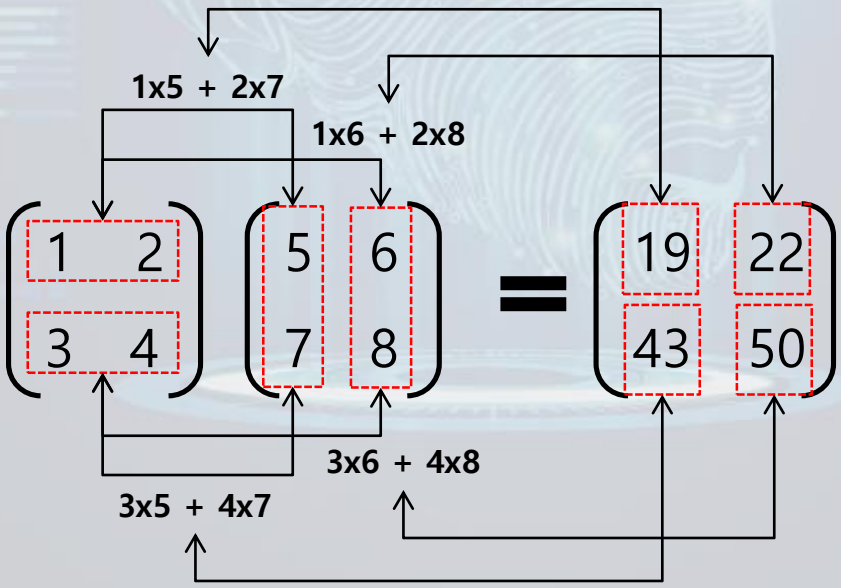
딥러닝 기초

다차원 배열의 계산

- 2차원 배열 → 행렬 (matrix)



- 행렬의 곱을 사용하여 입력과 가중치의 곱을 계산할 수 있음



다차원 배열의 계산

- **np.ndim()** : 배열의 차원 수를 반환
- **shape()** : 배열의 행과 열을 튜플로 반환

```
1  import numpy as np
2
3  b = np.array([[1, 2], [3, 4], [5, 6]])
4  print(b)
5  print(np.ndim(b))
6  print(b.shape)
```

```
[[1 2]
 [3 4]
 [5 6]]
2
(3, 2)
```

다차원 배열의 계산

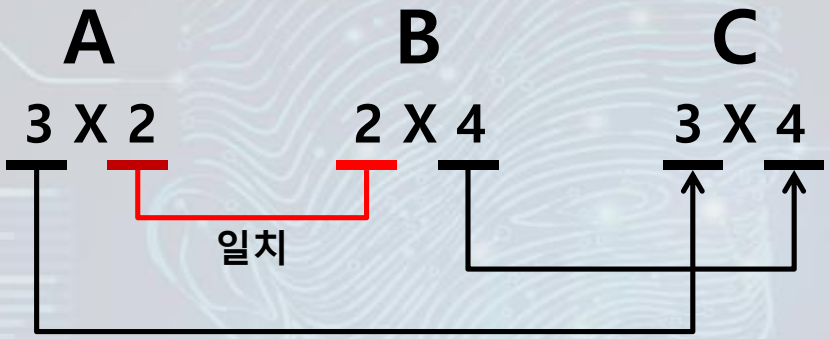
- **np.dot()** : 행렬의 곱

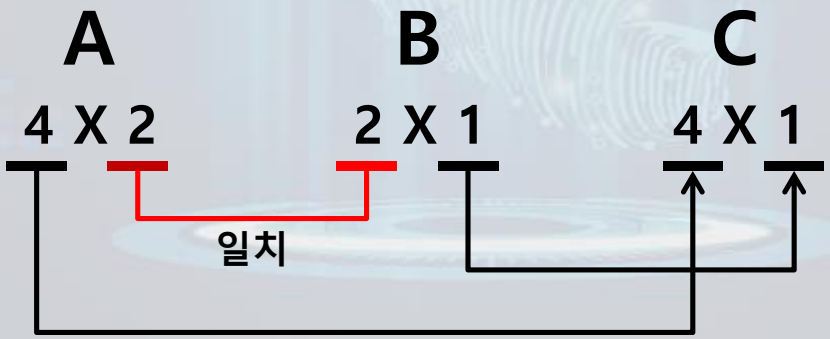
```
1  import numpy as np
2
3  a = np.array([[1, 2], [3, 4]])
4  b = np.array([[5, 6], [7, 8]])
5
6  print(np.dot(a, b))
```

```
[[19 22]
 [43 50]]
```

다차원 배열의 계산

- 행렬의 곱에서 행렬의 형상 (shape)의 조건
  - 첫 번째 차원의 열 수와 두 분째 차원의 행 수가 같아야 한다





다차원 배열의 계산

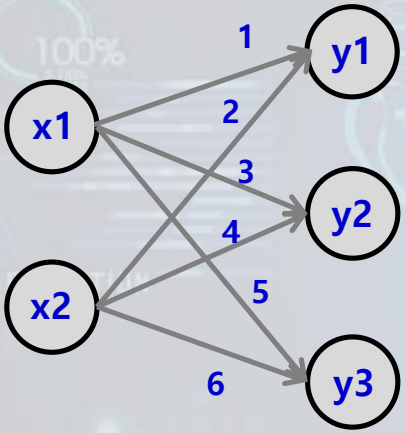
- 실습

```
1  import numpy as np
2
3  a = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
4  b = np.array([[5], [6]])
5
6  print(np.dot(a, b))
```

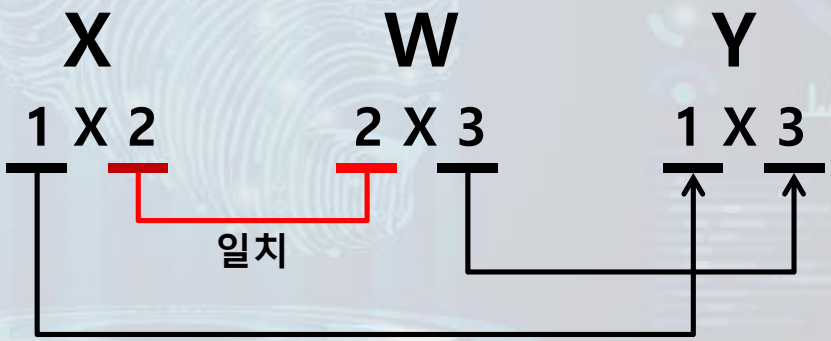
```
[[17]
 [39]
 [61]
 [83]]
```

다차원 배열의 계산

- 신경망에서 행렬의 곱



$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$



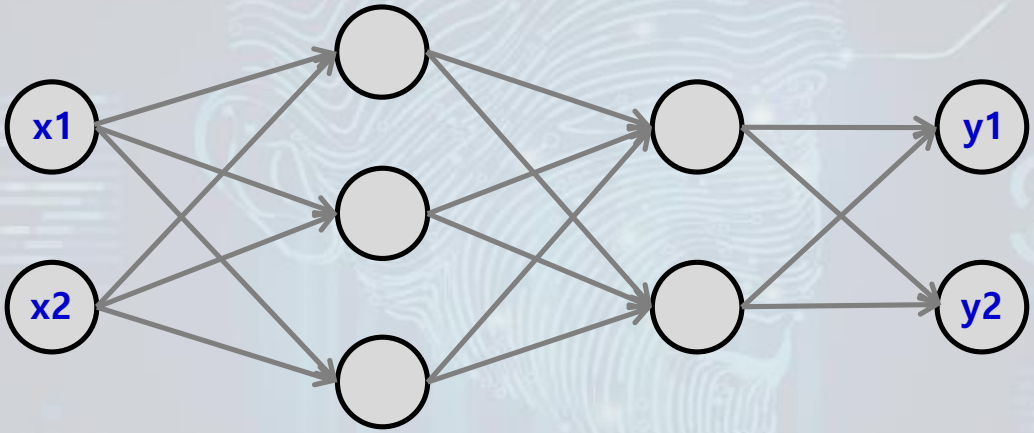
다차원 배열의 계산

- 실습

```
1  import numpy as np
2
3  x = np.array([1, 2])
4  w = np.array([[1, 3, 5], [2, 4, 6]])
5
6  print(np.dot(x, w))
```

```
[[ 5 11 17]]
```

3층 신경망 구현



3층 신경망 구현

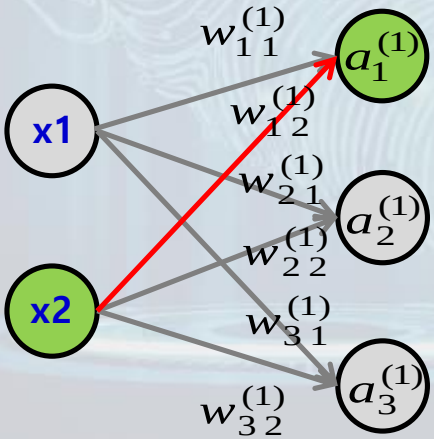
- 표기법

1층의 가중치

$$w_{12}^{(1)}$$

다음 층의 뉴런 인덱스

앞 층의 뉴런 인덱스

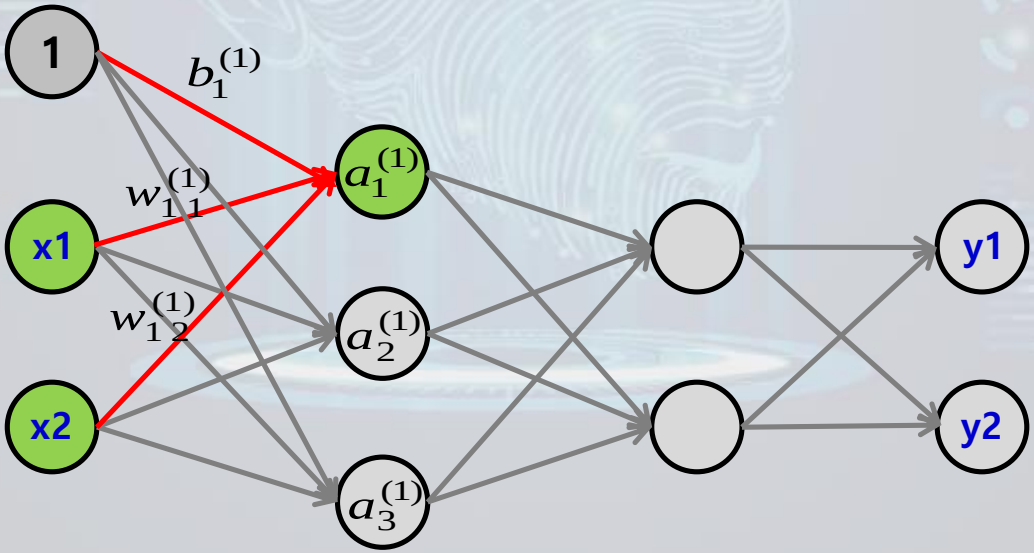


3층 신경망 구현

- 각 층의 신호 전달 구현

$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

$$A^{(1)} = XW^{(1)} + B^{(1)}$$



3층 신경망 구현

- 각 층의 신호 전달 구현

$$A^{(1)} = (a_1^{(1)} + a_2^{(1)} + a_3^{(1)})$$

$$X = (x_1, x_2)$$

$$B^{(1)} = (b_1^{(1)} + b_2^{(1)} + b_3^{(1)})$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

3층 신경망 구현

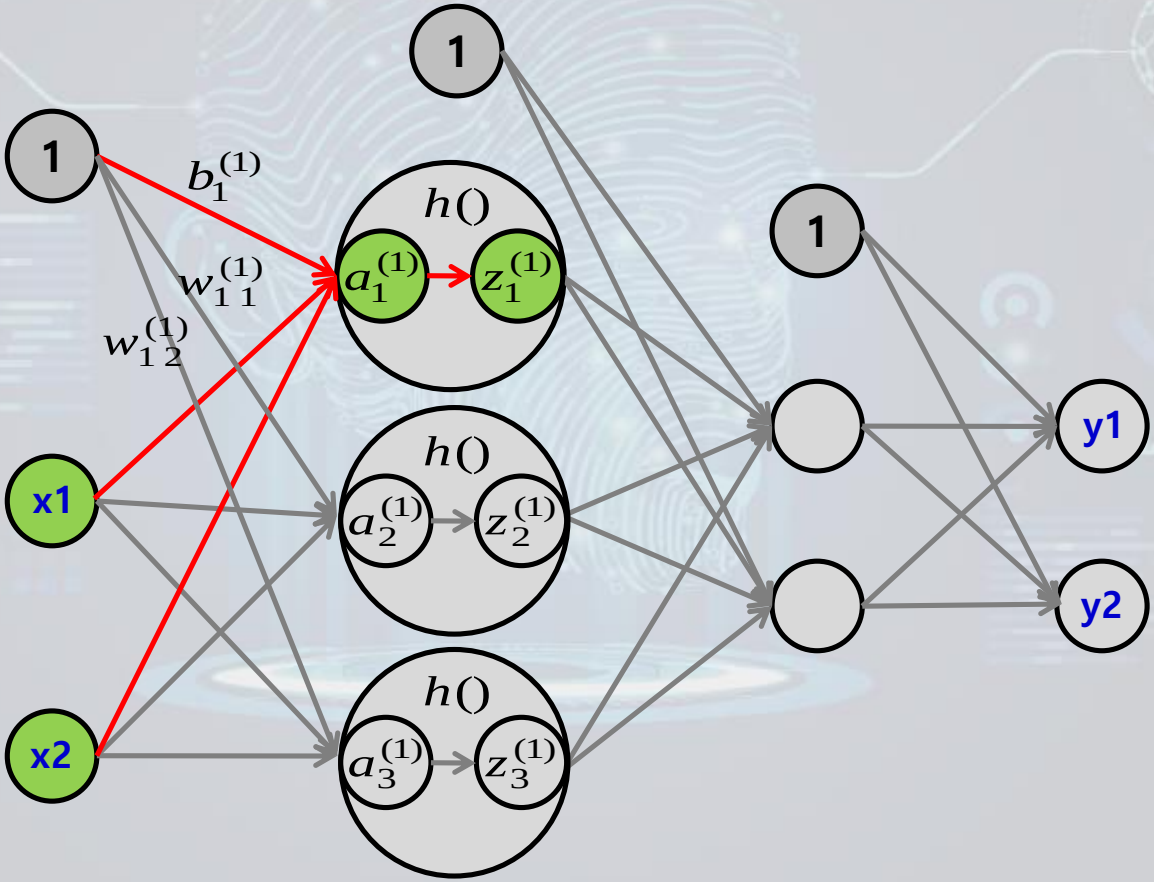
- 실습

```
1  import numpy as np
2
3  X = np.array([1.0, 0.5])
4  W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
5  B1 = np.array([0.1, 0.2, 0.3])
6
7  A1 = np.dot(X, W1) + B1
```

```
[[0.3 0.7 1.1]]
```

3층 신경망 구현

- 입력층에서 1층으로 신호 전달
  - 은닉층에서 가중치 합을 a로 표기, 활성화 함수 h()로 변환된 신호를 z로 표기



3층 신경망 구현

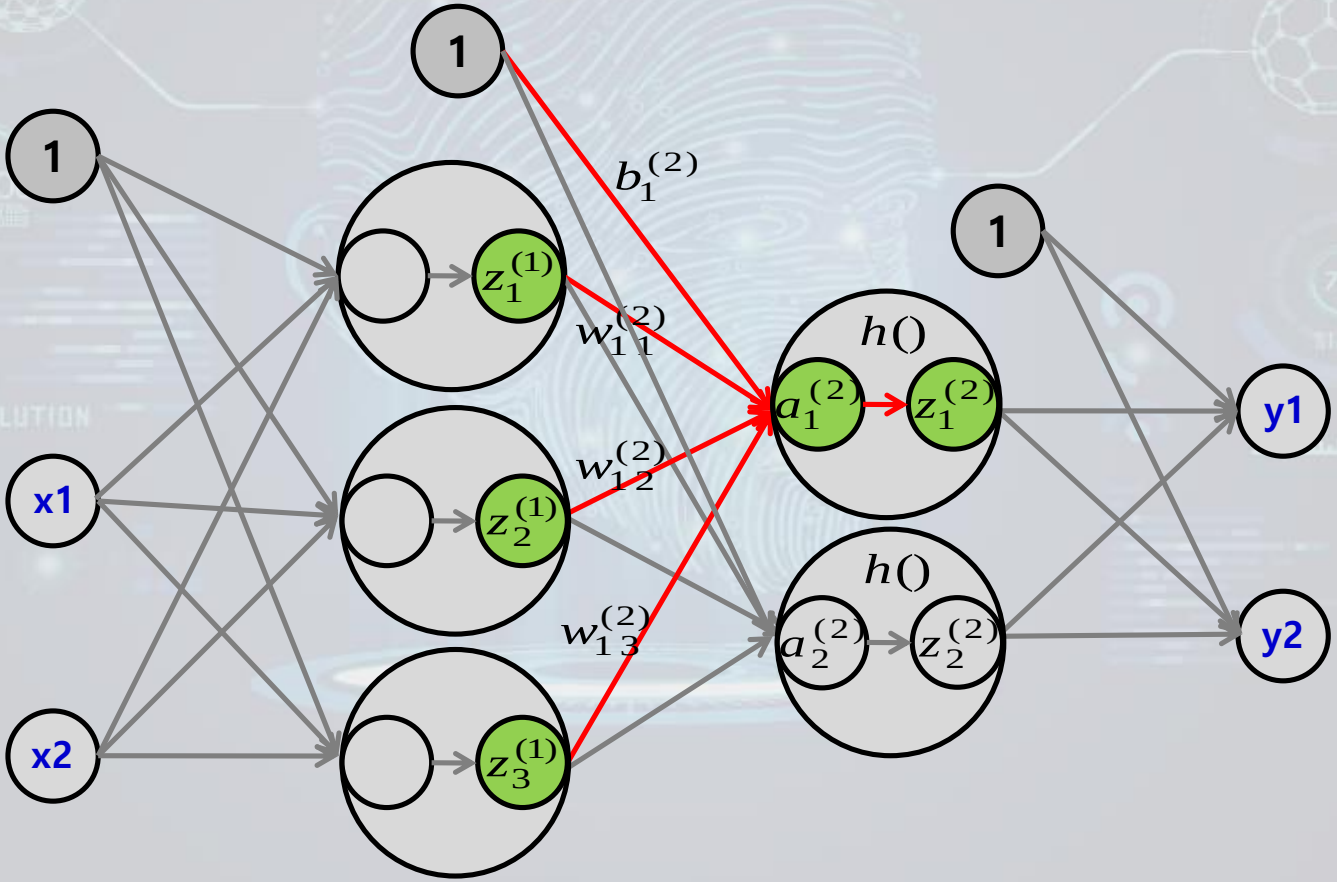
- 실습 – 활성화 함수를 sigmoid 함수 사용

```
1   import numpy as np
2
3   def sigmoid(x):
4       return 1/(1 + np.exp(-x))
5
6   X = np.array([1.0, 0.5])
7   W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
8   B1 = np.array([0.1, 0.2, 0.3])
9
10  A1 = np.dot(X, W1) + B1
11  print(A1)
12
13  Z1 = sigmoid(A1)
14  print(Z1)
```

```
[[0.3 0.7 1.1]]
[[0.57444252 0.66818777 0.75026011]]
```

3층 신경망 구현

- 1층에서 2층으로 신호 전달



3층 신경망 구현

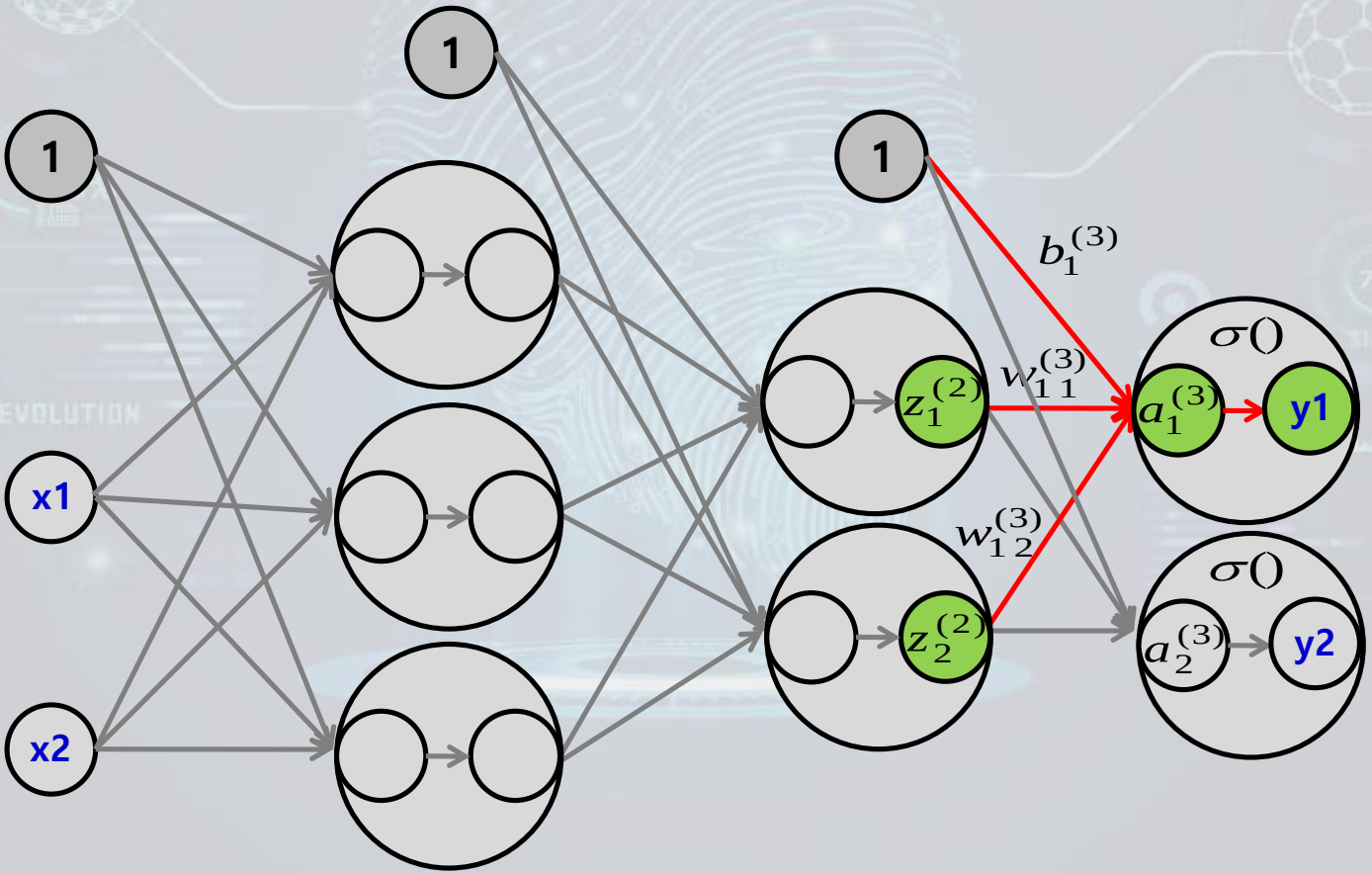
- 실습 – 활성화 함수를 sigmoid 함수 사용

```
16  W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
17  B2 = np.array([0.1, 0.2])
18
19  A2 = np.dot(Z1, W2) + B2
20  Z2 = sigmoid(A2)
21  print(Z2)
```

```
[0.62624937 0.7710107 ]
```

3층 신경망 구현

- 2층에서 출력층으로 신호 전달



3층 신경망 구현

- 실습 – 활성화 함수를 sigmoid 함수 사용

```
23  W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
24  B3 = np.array([0.1, 0.2])
25
26  A3 = np.dot(Z2, W3) + B3
27  Y1 = A3        #항등 함수
28
29  print(Y1)
```

```
[0.31682708 0.69627909]
```

3층 신경망 구현

- 통합

```
1   import numpy as np
2
3   def sigmoid(x):
4       return 1/(1 + np.exp(-x))
5
6   def init_network():
7       network = {}
8       network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
9       network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
10      network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
11      network['b1'] = np.array([0.1, 0.2, 0.3])
12      network['b2'] = np.array([0.1, 0.2])
13      network['b3'] = np.array([0.1, 0.2])
14
15      return network
16
```

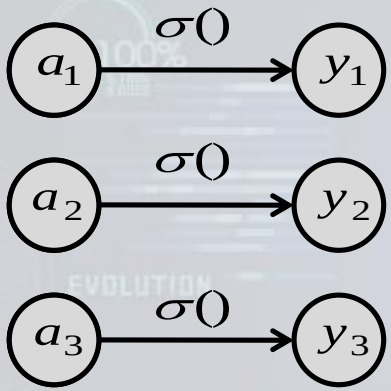
3층 신경망 구현

- 통합

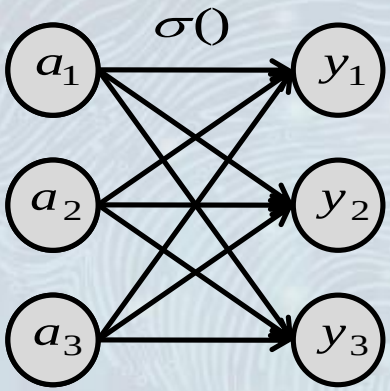
```
17  def forward(network, x):
18      W1, W2, W3 = network['W1'], network['W2'] , network['W3']
19      b1, b2, b3 = network['b1'], network['b2'], network['b3']
20
21      a1 = np.dot(x, W1) + b1
22      z1 = sigmoid(a1)
23      a2 = np.dot(z1, W2) + b2
24      z2 = sigmoid(a2)
25      A3 = np.dot(z2, W3) + b3
26      y = A3        #항등함수
27
28      return y
29
30  network = init_network()
31  x = np.array([1.0, 0.5])
32  y = forward(network, x)
33  print(y)
```

출력층 설계하기

- 일반적으로 출력층의 활성화 함수로는 **회귀에는 항등 함수**를 **분류에는 소프트맥스 함수**를 사용
  - **항등함수 (identify function)** : 입력을 그대로 출력하는 함수
  - **softmax function** : 출력층의 각 뉴런이 모든 입력 신호의 영향을 받는 함수



항등함수



소프트맥스 함수

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

출력층 설계하기

- 소프트맥스 함수 구현 → 지수는 너무 빨리 커지는 문제가 발생 → **overflow**

```
1  import numpy as np
2
3  def softmax(a):
4      exp_a = np.exp(a)
5      sum_exp_a = np.sum(exp_a)
6      y = exp_a / sum_exp_a
7
8      return y
9
10 a = np.array([1010, 1000, 900])
11 print(softmax(a))
```

```
[nan nan nan]
```

출력층 설계하기

- 소프트맥스 함수 구현 : 일반적으로 최대값을 빼주는 방법으로 **overflow** 문제 해결

```
1   import numpy as np
2
3   def softmax(a):
4       c = np.max(a)
5       exp_a = np.exp(a - c)
6       sum_exp_a = np.sum(exp_a)
7       y = exp_a / sum_exp_a
8
9       return y
10
11  a = np.array([1010, 1000, 900])
12  print(softmax(a))
```

```
[9.99954602e-01 4.53978687e-05 1.68883521e-48]
```

소프트맥스 함수의 특징

- **소프트맥스 함수의 출력 : 0에서 1.0 사이의 실수**
- **소프트맥스 함수 출력의 총합 : 1**
  - 소프트맥스 함수의 **출력을 확률로 해석 가능**
  - 각 원소의 대소 관계는 변하지 않음 → y = exp(x)가 단조 증가함수이기 때문
  - 신경망에서는 가장 큰 출력을 내는 뉴런에 해당하는 클래스만 인식하므로 **추론 단계에서는 소프트맥스 함수를 생략**해도 됨 → 지수함수로 인한 속도저하문제 해결

- **신경망을 학습시킬 때는 출력층에 소프트맥스 함수를 사용**

출력층 설계하기

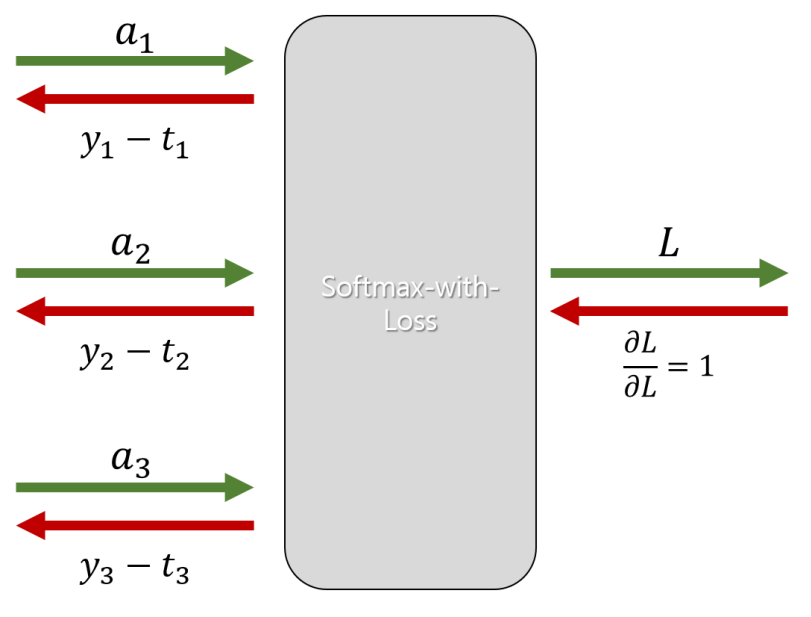
- 소프트맥스 함수 출력 총합과 원소의 대소 관계가 변하지 않는 것을 확인

```
1   import numpy as np
2
3   def softmax(a):
4       c = np.max(a)
5       exp_a = np.exp(a - c)
6       sum_exp_a = np.sum(exp_a)
7       y = exp_a / sum_exp_a
8
9       return y
10
11  a = np.array([0.3, 2.9, 4.0])
12  y = softmax(a)
13  print(y)
14  print(np.sum(y))
```

```
[0.01821127 0.24519181 0.73659691]
1.0
```

Softmax-with-Loss 노드의 역전파

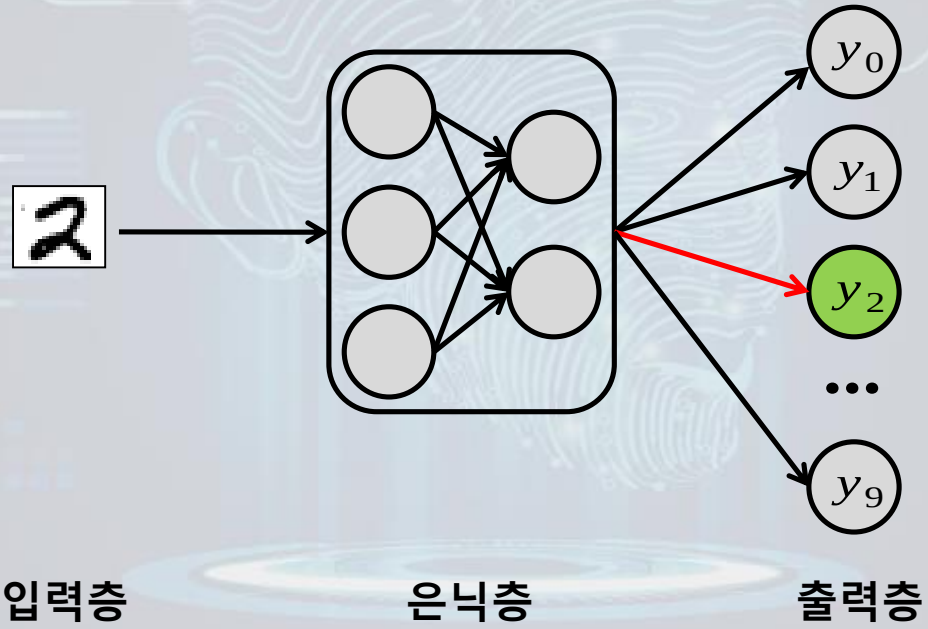
- Softmax-with-Loss 노드는 a을 입력받아 L을 출력하고 정답인 노드만 1을 빼줌
- 만약 정답이 t_3이라면 y_3 -1으로 역전파하고 나머지는 y_1, y_2로 역전파



https://ratsgo.github.io/deep learning/2017/05/14/backprop/

출력층의 뉴런 수 설정하기

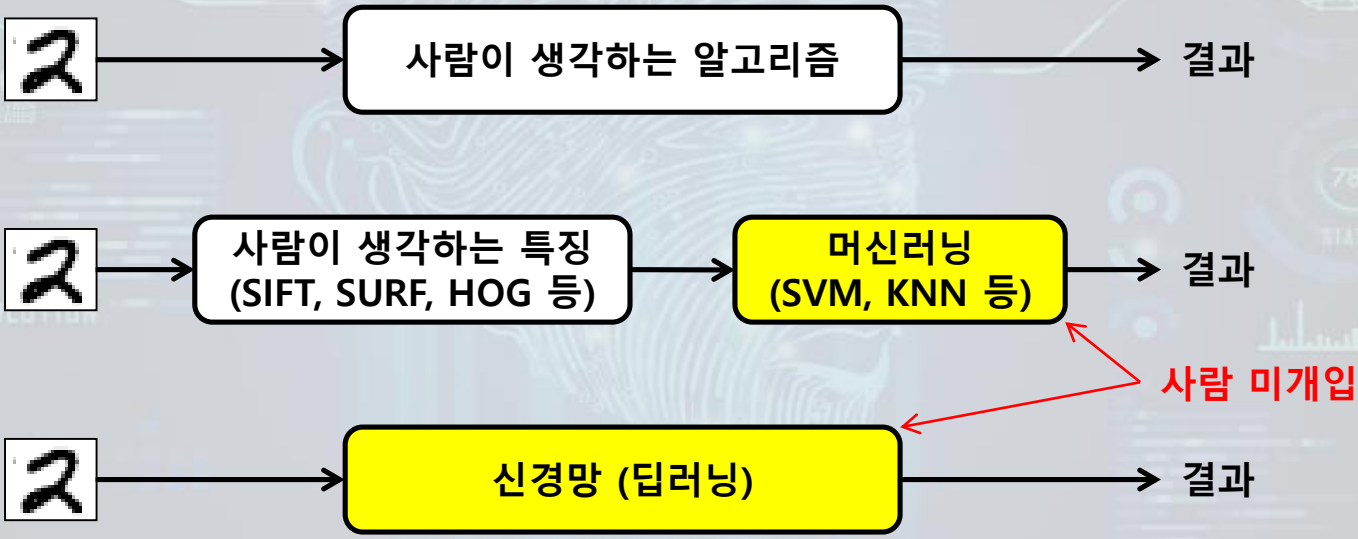
- 출력층의 뉴런 수는 **풀려는 문제에 맞게** 적절하게 설정해야 함
  - 숫자 0부터 9까지를 분류하고 싶다면 출력층의 개수는 10개로 설정



신경망 학습

개요

- **학습** : 훈련 데이터로부터 **가중치 매개변수의 최적값**을 자동으로 획득하는 것 → **손실함수의 결과값을 가장 작게** 만드는 가중치 매개변수를 찾는 것



평균 제곱 오차

- **손실함수 (Loss function)** : 최적의 매개변수 값을 탐색하기 위한 기준 지표 → **평균제곱오차 (MSE)**와 **교차 엔트로피 오차 (CEE)**를 주로 사용
- 정확도를 사용하지 않고 손실함수를 사용하는 이유 → **정확도의 미분값이 대부분 0이 되기때문**
  - 신경망 학습에서 가중치, 편향을 탐색 → 손실 함수를 최대한 작게 하는 매개변수를 찾음 → 매개변수의 기울기 (미분)을 계산하고 기울기를 이용하여 매개변수를 갱신
  (예) x가 가중치 매개변수, y가 손실함수라고 하면
  미분이 음수이면 기울기도 음수이므로 x를 h만큼 증가시키면 y는 감소 → 매개변수를 증가시켜 손실함수를 감소시킴

- **손실함수의 미분** : 가중치 매개변수의 값을 아주 조금 변화시켰을 때 손실 함수가 어떻게 변하는지 의미
  - 미분 값이 음수이면 가중치 매개변수를 양의 방향으로 변화시켜 손실 함수의 값을 감소
  - 미분 값이 양수이면 가중치 매개변수를 음의 방향으로 변화시켜 손실 함수의 값을 감소
  - 미분 값이 0이면 가중치 매개변수를 어느쪽으로 움직여도 손실 함수의 값은 변하지 않음
  → 계단함수를 손실함수로 사용하지 않는 이유

대표적인 손실함수

- 실제값 yt, 예측값 y0인 경우

평균 제곱 계열	mean_squared_error	평균제곱오차 mean(square(yt – y0))
	mean_absolute_error	평균절대 오차 mean(abs(yt – y0))
	mean_absolute_percentage_error	평균 절대 백분율 오차 mean(abs(yt-y0) / abs(yt))
	mean_squared_logarithmic_error	평균제곱 로그 오차 mean(square(log(y0)+1)-(log(yt)+1)))
교차 엔트로피 계열	categorical_crossentropy	범주형 교차 엔트로피
	binary_crossentropy	이항 교차 엔토로피

평균 제곱 오차

- **평균제곱오차 (Mean Squared Error)** → 회귀에서 주로 사용

$$E = \frac{1}{2}\sum_{k=1}^{n}(y_k - t_k)^2$$

y_k : 신경망의 출력 (신경망이 추정한 값)
t_k : 정답 레이블
k : 데이터의 차원 수

평균 제곱 오차

- 실습 – 정답 레이블과 추정 값이 일치하는 경우

```
1  import numpy as np
2
3  def mean_squard_error(y, t, n):
4      return 0.5 * np.sum((y - t)**2)
5
6  t = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0])      #one-hot encoding
7  y = np.array([0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0])
8
9  print(mean_squard_error(y, t, y.size))
```

```
0.09750000000000003
```

평균 제곱 오차

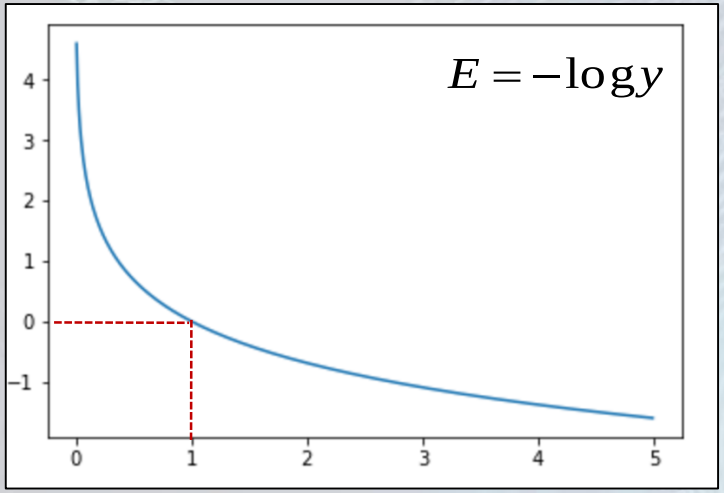
- 실습 – 정답 레이블과 추정 값이 일치하지 않는 경우

```
1  import numpy as np
2
3  def mean_squard_error(y, t, n):
4      return 0.5 * np.sum((y - t)**2)
5
6  t = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0])      #one-hot encoding
7  y = np.array([0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0])
8
9  print(mean_squard_error(y, t, y.size))
```

```
0.5975
```

교차 엔트로피 오차

- **교차 엔트로피 오차 (Cross Entropy Error : CEE)** → 분류에서 주로 사용
  - **레이블이 one-hot 인코딩인 경우에만 사용 가능**



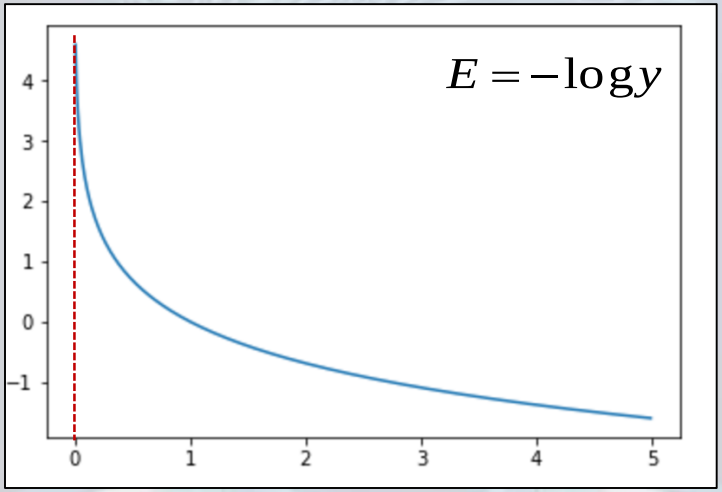
$$E = -\sum_{k=1}^{n} t_k \log y_k$$

y_k : 신경망의 출력 (신경망이 추정한 값)
t_k : 정답 레이블
k : 데이터의 차원 수

- t_k가 1일 때 (정답일 때)만 자연로그의 합을 계산하는 식이 됨.
- 정답에 해당하는 출력이 커질수록 0에 가까워지다가 출력이 1일 때 0이 되며 **정답일 때의 출력이 작아질수록 오차는 커짐**

교차 엔트로피 오차

- **교차 엔트로피 오차 (Cross Entropy Error : CEE)**
  - 출력이 0이 되면 CEE값이 inf가 되어 더이상 계산을 진행할 수 없게 됨 → **아주 작은 값을 더해서 절대 0이 되지 않게 해주어야 함**



평균 제곱 오차

- 실습 – 정답 레이블과 추정 값이 일치하는 경우

```
1   import numpy as np
2
3   def cross_entropy_error(y, t):
4       delta = 1e-7
5       return -np.sum(t * np.log(y + delta))
6
7   t = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0])
8   y = np.array([0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0])
9
10  print(cross_entropy_error(y, t))
```

```
0.510825457099338
```

평균 제곱 오차

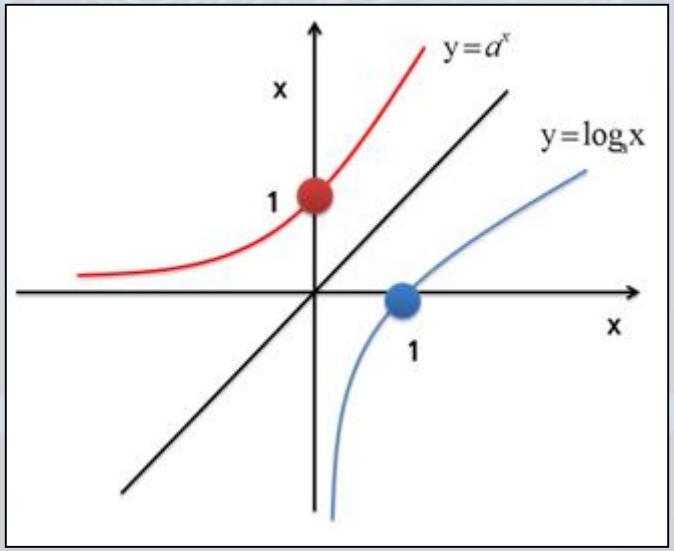
- 실습 – 정답 레이블과 추정 값이 일치하지 않는 경우

```
1   import numpy as np
2
3   def cross_entropy_error(y, t):
4       delta = 1e-7
5       return -np.sum(t * np.log(y + delta))
6
7   t = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0])
8   y = np.array([0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0])
9
10  print(cross_entropy_error(y, t))
```

```
2.302584092994546
```

교차 엔트로피 오차

- 손실함수 계산에 log를 사용하는 이유
  - 손실의 최소값 검색 → 미분 (기울기 계산) → 기울기가 작아지는 방향으로 매개변수를 갱신
  - **활성화 함수로 비선형 형태의 sigmoid 함수를 사용** → 지수의 성질 → 기울기가 급격히 변하므로 최소값을 찾기가 힘들어짐 → 반대되는 성질의 **log를 사용하여 선형 특성으로 변경하여 문제 해결**



교차 엔트로피 오차

- 분류 오차(Classification Error)가 33% (분류 정확도 67%)인 2가지 학습 결과
  - 왼쪽 결과 : 2개의 샘플은 겨우 맞추고 1개의 샘플은 완전히 틀림
  - 오른쪽 결과 : 2개의 샘플은 확실히 맞추고 1개의 샘플은 아깝게 틀림

계산결과			라벨 (A/B/C)				정답?
0.3	0.3	0.4	0	0	1	A	O
0.3	0.4	0.3	0	1	0	B	O
0.1	0.2	0.7	1	0	0	C	X

계산결과			라벨 (A/B/C)				정답?
0.1	0.2	0.7	0	0	1	A	O
0.1	0.7	0.2	0	1	0	B	O
0.3	0.4	0.3	1	0	0	C	X

- 단순 **분류 오차**는 **틀린 개수에 대한 결과만 보여주고** 라벨과 비교하여 얼마나 많이 틀렸는지, 얼마나 정확하게 맞았는지에 대한 값은 제공하지 않음.
- 훈련 후 신경회로망을 평가하는 경우는 분류 오차가 적합하지만 **역전파를 사용한 훈련에서는 MSE나 ACE (평균 교차 엔트로피)**를 사용

http://funmv2013.blogspot.com/2017/01/cross-entropy.html

교차 엔트로피 오차

- MSE를 사용 → **0인 경우도 고려함으로써 잘못된 출력을 너무 많이 강조**

- 왼쪽 결과

$$E_1 = (0.3-0)^2 + (0.3-0)^2 + (0.4-1)^2 = 0.09+0.09+0.36 = 0.54$$

$$E_2 = (0.3-0)^2 + (0.4-1)^2 + (0.3-0)^2 = 0.09+0.36+0.09 = 0.54$$

$$E_3 = (0.1-1)^2 + (0.2-0)^2 + (0.7-0)^2 = 0.81+0.04+0.49 = 1.34$$

$$E = (E_1 + E_2 + E_3)/3 = 0.806666$$

- 오른쪽 결과

$$E_1 = (0.1-0)^2 + (0.2-0)^2 + (0.7-1)^2 = 0.01+0.04+0.09 = 0.14$$

$$E_2 = (0.1-0)^2 + (0.7-1)^2 + (0.2-0)^2 = 0.01+0.09+0.04 = 0.14$$

$$E_3 = (0.3-1)^2 + (0.4-0)^2 + (0.3-0)^2 = 0.49+0.16+0.09 = 0.74$$

$$E = (E_1 + E_2 + E_3)/3 = 0.34$$

http://funmv2013.blogspot.com/2017/01/cross-entropy.html

교차 엔트로피 오차

- 교차 엔트로피 오차를 사용 → **1인 경우만 고려**

  - 왼쪽 결과

$$E_1 = -((\ln(0.3)*0)+(\ln(0.3)*0)+(\ln(0.4)*1) = -\ln(0.4)$$
$$E_2 = -((\ln(0.3)*0)+(\ln(0.4)*1)+(\ln(0.3)*0) = -\ln(0.4)$$
$$E_3 = -((\ln(0.1)*1)+(\ln(0.2)*0)+(\ln(0.7)*0) = -\ln(0.1)$$
$$E = (E_1+E_2+E_3)/3 = 1.38$$

- 오른쪽 결과

$$E_1 = -((\ln(0.1)*0)+(\ln(0.2)*0)+(\ln(0.7)*1) = -\ln(0.7)$$
$$E_2 = -((\ln(0.1)*0)+(\ln(0.7)*1)+(\ln(0.2)*0) = -\ln(0.7)$$
$$E_3 = -((\ln(0.3)*1)+(\ln(0.4)*0)+(\ln(0.3)*0) = -\ln(0.3)$$
$$E = (E_1+E_2+E_3)/3 = 0.64$$

http://funmv2013.blogspot.com/2017/01/cross-entropy.html

교차 엔트로피 오차

- 분류에서 교차 엔트로피 오차를 사용하는 이유

  - 역전파 학습 중에 목표값에 따라 출력 노드 값을 1.0 또는 0.0으로 설정하려고 한다면

  → MSE를 사용하면 조정 요소가 점점 작아지면서 가중치 변화도 작아지고 학습 진행이 멈춤

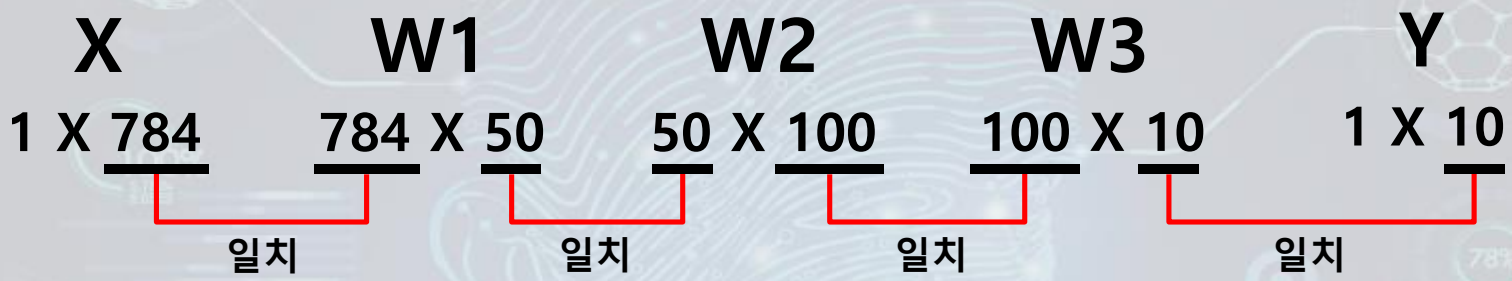
  → MSE가 출력의 크기에 의존하기 때문

출력노드 값 (y)	MSE $(y-1)^2$	교차 엔트로피 오차 -log(y)
0.95	0.0025	0.051
0.6	0.16	0.511
0.1	0.81	2.303

http://funmv2013.blogspot.com/2017/01/cross-entropy.html

배치 처리

- 신경망 각 층의 배열 형상 추이



- 배치 처리를 위한 배열들의 형상 추이 – 100장의 이미지를 한 번에 입력하는 경우 → batch 데이터

배치 처리

- 실습

```
40  x, t = get_data()
41  network = init_network()
42
43  batch_size = 100
44
45  accuracy_cnt = 0
46  for i in range(0, len(x), batch_size):
47      x_batch = x[i:i+batch_size]
48      y_batch = predict(network, x_batch)
49      p = np.argmax(y_batch, axis=1)
50      accuracy_cnt += np.sum(p == t[i:i+batch_size])
51
52  print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

```
Accuracy:0.9352
```

배치 처리

- np.argmax() 함수 분석
  - 최대값의 인덱스를 반환하는 함수
  - **axis = 1** : 여러 개의 배열 중 1번째 차원을 구성하는 원소

```
1  import numpy as np
2
3  x = np.array([[0.1, 0.8, 0.1],[0.3, 0.1, 0.6],[0.2, 0.5, 0.3],[0.8, 0.1, 0.1]])
4  y = np.argmax(x, axis=1)
5  print(y)
```

```
[1 2 1 0]
```

배치 처리

- np.sum() 함수 분석

  - 원소 중에 같은 값의 개수를 카운트

```
1  import numpy as np
2
3  y = np.array([1, 2, 1, 0])
4  t = np.array([1, 2, 0, 0])
5
6  print(np.sum(y ==  t))
```

```
3
```

미니배치 학습

- 훈련데이터 N개에 대해 손실함수를 계산 → **평균 교차 엔트로피 (ACE : Average Cross Entropy)**

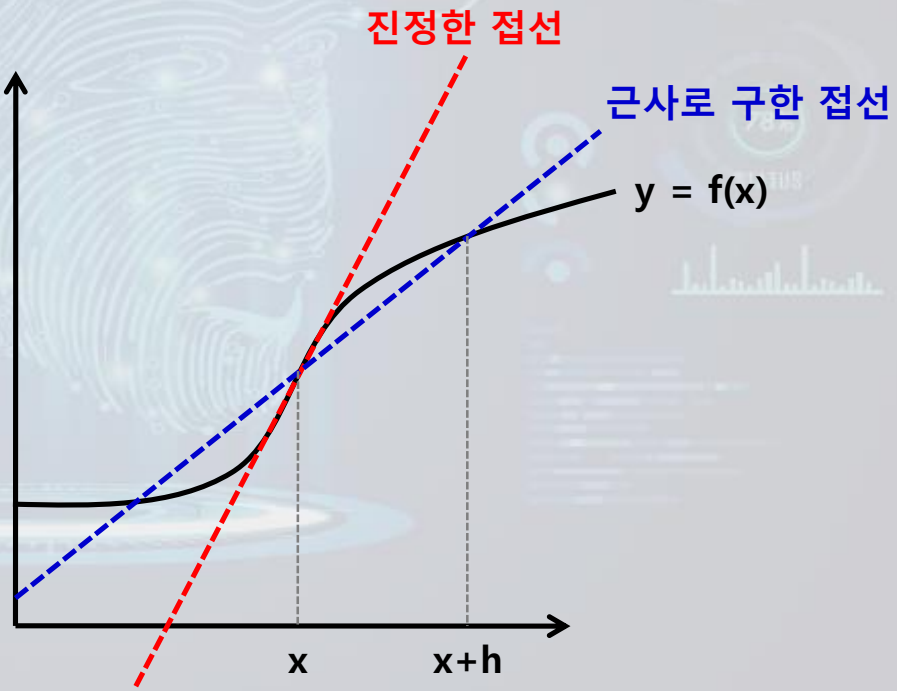
$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk} \log y_{nk}$$

- **미니배치 (mini batch) 학습** : 대용량 데이터를 대상으로 손실함수를 계산하려면 시간이 걸림 → 데이터들 중에서 일부만 골라서 학습

수치 미분

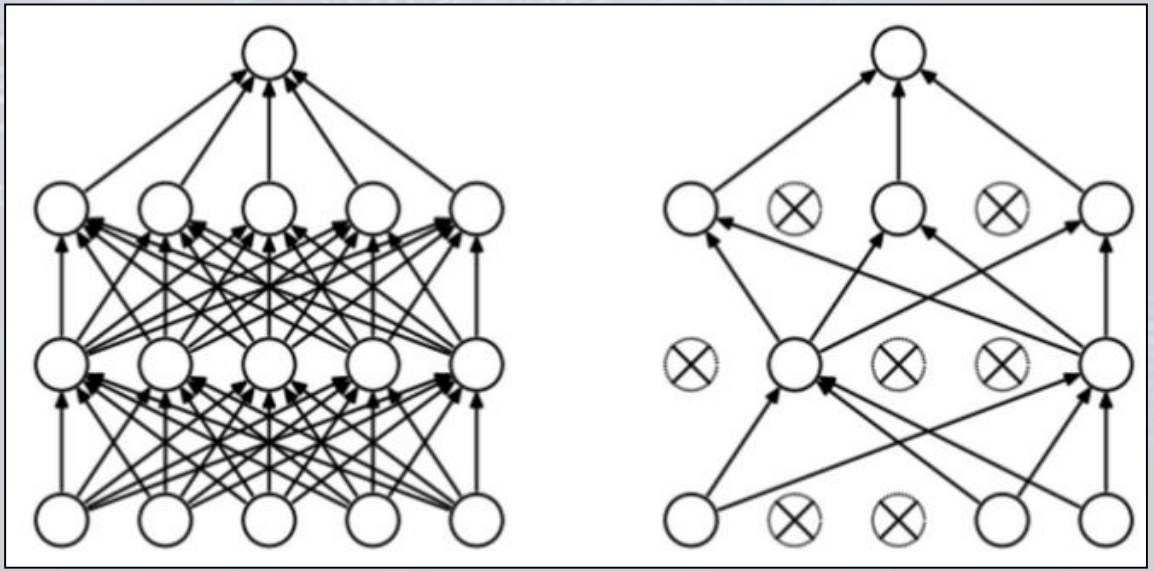
- **미분** : 한순간의 변화량

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



미니배치 학습

- **Dropout** : **매 샘플마다 다른 일부의 다른 노드들을 사용하는 것**으로 overfitting을 방지하기 위한 정규화의 일종, 모델을 평균화하여 훈련 데이터 내에서의 복잡한 서로간의 상호성을 막아주는 방법
  - **매 샘플마다 다른 노드들을 학습할 수 있지만 사용하는 가중치는 서로 같음**
  - **모델의 일부만 사용(sparse한 네트워크)하여 학습 → 앙상블 효과** (여러 개의 네트워크를 사용하는 것) → fully connected된 일반적인 신경망은 노드간 상호관계가 복잡함

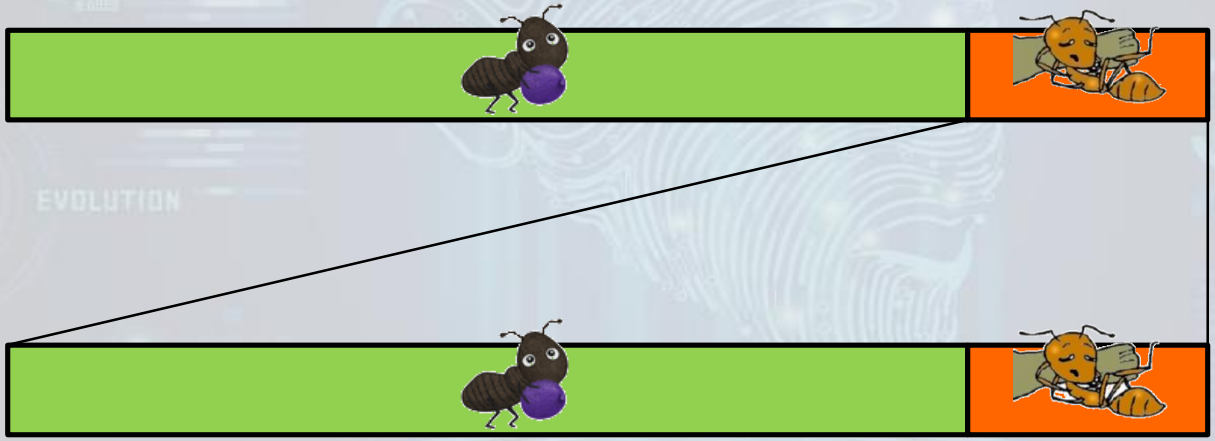


일반적인 신경망 **Dropout 적용 후**

미니배치 학습

- **Dropout**

  - 80%의 개미만 일하고 20%의 개미는 항상 논다고 할 때 → 노는 20%의 개미를 열심히 일하도록 만들기 위해서는 기존의 일하는 개미 80%는 제외하고 노는 20% 개미들만 일을 시킴 → 20%의 개미 중 80% 개미는 열심히 일하고 20% 개미는 놀게 됨



퍼셉트론 (Perceptron) 학습 과정

(1) 임의의 w와 b를 설정

(2) 주어진 훈련 데이터를 가지고 결과 값 y를 구함

(3) 결과값 y와 실제값 t 사이의 오차 계산

(4) 오차를 줄이는 방향으로 w와 b를 재설정 (학습)

(5) 만족할만한 결과가 나올 때까지 (2)-(3)과정을 반복

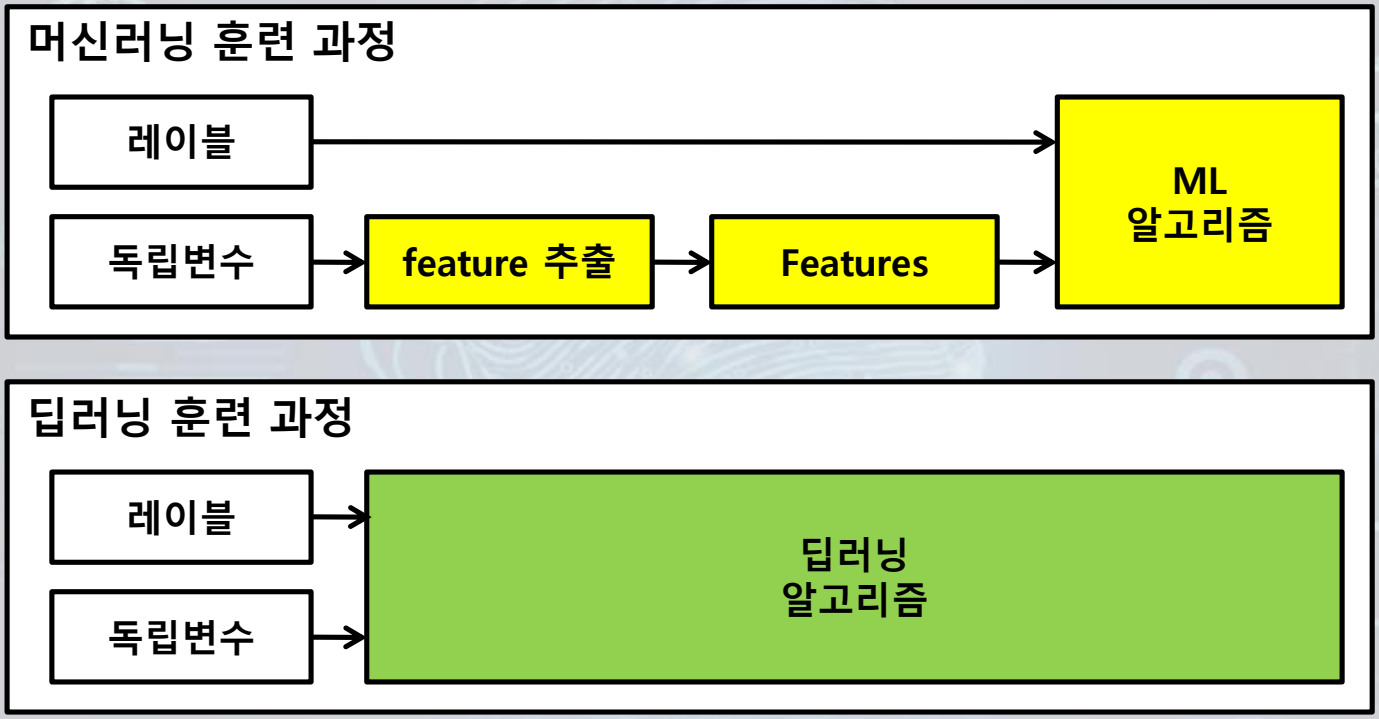
학습 규칙

- **경사하강법 (SGD : Stochastic gradient descent)** : 학습 데이터의 가중치를 하나씩 갱신하는 방법
  → 학습 성능이 무작위로 변하는 경향이 있음.

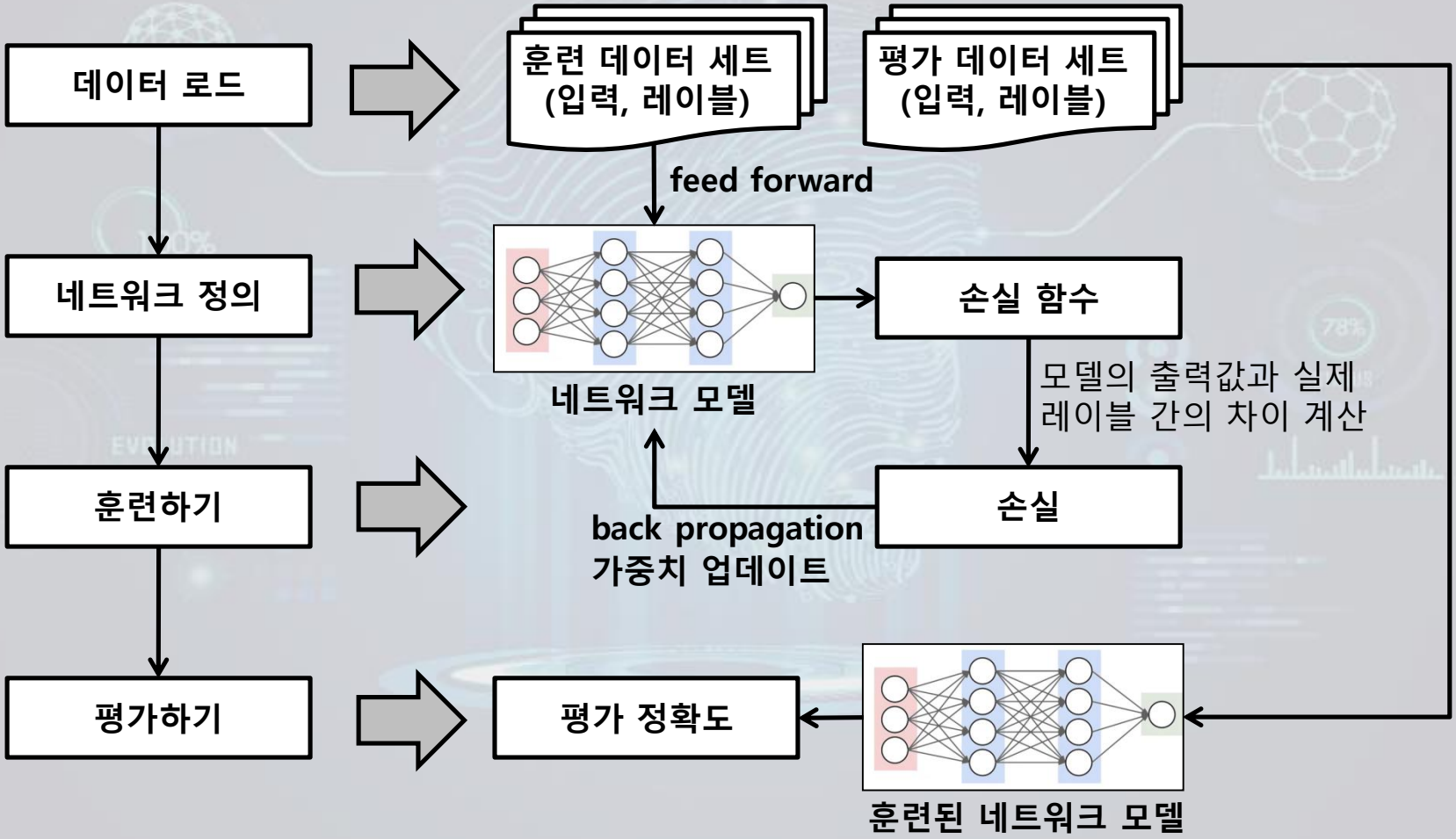
- **배치 방식 (batch)** : 모든 학습 데이터의 오차에 관한 가중치를 계산한 후 이들 평균값으로 가중치를 갱신하는 방법 (무작위 변화는 줄어드나 학습 시간이 오래 걸림)

- **미니 배치 방식 (mini batch)** : 전체 학습 데이터 중 일부 데이터만 골라 가중치를 갱신하는 방식

머신러닝 vs 딥러닝


머신러닝 훈련 과정
레이블
독립변수
feature 추출
Features
ML
알고리즘
딥러닝 훈련 과정
레이블
독립변수
딥러닝
알고리즘

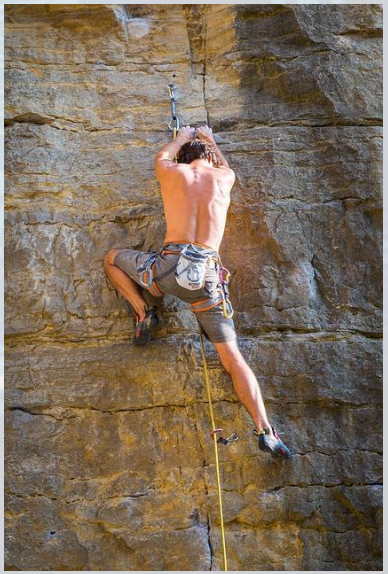
딥러닝 학습절차


데이터 로드
네트워크 정의
훈련하기
평가하기
훈련 데이터 세트
(입력, 레이블)
평가 데이터 세트
(입력, 레이블)
feed forward
네트워크 모델
손실 함수
모델의 출력값과 실제
레이블 간의 차이 계산
손실
back propagation
가중치 업데이트
평가 정확도
훈련된 네트워크 모델

경사하강법

- **feed forward** : 실행에 옮기기 전에 결함을 **미리 예측 (목표의 기대치)**해 행하는 피드백 제어 → **미래로부터**
- **feed back :** 현재의 수정해야 할 점에 **과거의 데이터에 근거**하여 행하는 피드백 제어로 **상황이 발생한 후**에 평가가 이루어짐 → **과거로부터**

- **경사 하강법 (Gradient Descent)** : 너무나 **많은 신경망** 안의 **가중치 조합**을 모두 계산하면 **시간이 오래** 걸리기 때문에 효율적으로 이를 하기 위해 고안된 방법



경사하강법

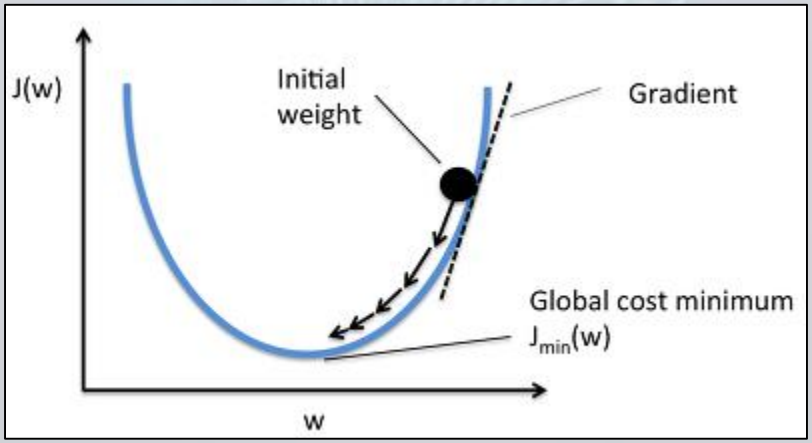
- 가중치 업데이터를 에러를 줄이는 방향으로 진행되며 방향은 학습률을 곱해서 가중치를 이동시키는 동작을 반복

에러를 줄이는 방향

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

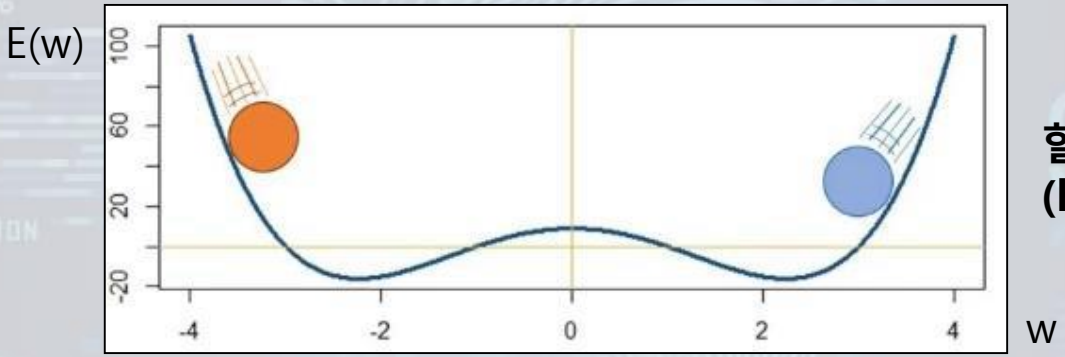
기울기 (에러)

학습률 (보폭)



경사하강법

- 경사 하강법은 주어진 데이터 세트에 대해 손실함수 E(w)가 최소가 되는 가중치 w를 찾는 작업
  - 현재 지점의 w에서 기울기가 가장 가파르게 하강하는 곳을 따라 조금씩 이동
  - 현재 지점의 w에서 E(w)의 w에 대한 음의 미분값이 가장 높은 방향(기울기가 작은, 더 낮은 방향)으로 w를 조금씩 이동



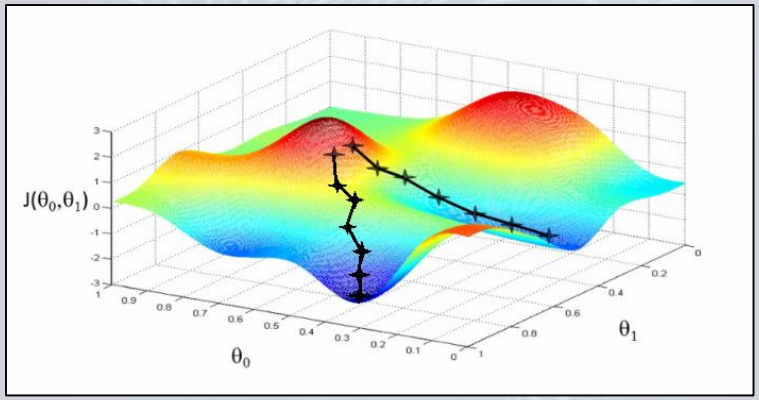
힐 클라이밍 (hill climbing)

- 문제는 이동하는 정도가 **크면 최저 지점을 지나갈 수도 있고**, 너무 **작으면 이동 횟수가 많아져서** 최저 지점을 찾지 못할 수도 있음.

http://ucanalytics.com/blogs/intuitive-machine-learning-gradient-descent-simplified/

경사하강법

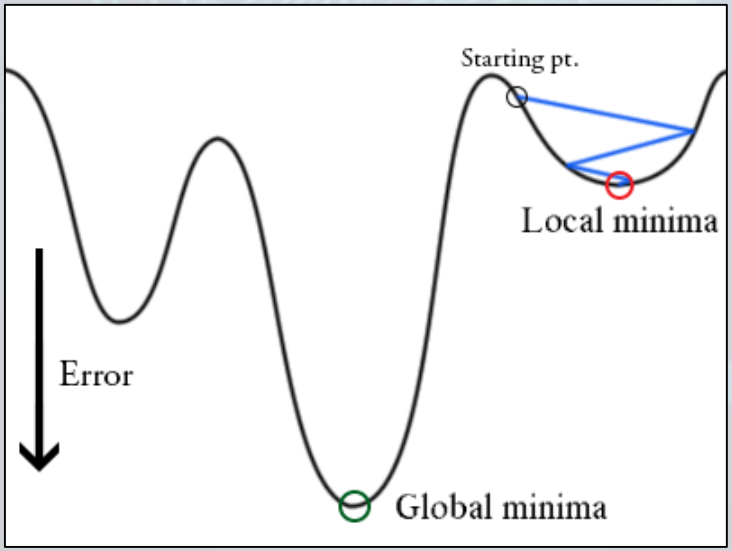
- 또 다른 문제는 엉뚱한 최저점을 찾을 수 있다는 것 → 차원이 여러 개라고 한다면 단순한 2차함수 형태가 되지는 않아서 **여러 개의 계곡이 있고 발을 잘못 딛으면 엉뚱한 계곡으로 갈 수도 있음**
- 이러한 문제를 피하기 위해서는 **서로 다른 초기값으로 주어** 산을 내려가게 하는 방법으로 신경망의 경우에서는 weight의 초기 값을 다르게 주면서 경사하강법을 활용



- 한 발 내려간 시점에서 등산객이 자기가 산을 제대로 내려가고 있는지 주변 경사(기울기)를 보면서 판단하듯이 기울기가 음수인지? 양수인지? 확인하면서 최저지점으로 내려감.

경사하강법

- **지역 최소값 (local minimum)** : 학습 중에 손실함수의 전역 최소값 (global minimum)을 찾지 못하고 **지역 최소값에 빠져 나오지 못하는 상황**
  - 파라미터의 초기값에서 경사를 따라 가중치를 업데이트할 때 손실함수의 지역 최소값에 빠지는 경우 에폭(epoch)이 진행되더라도 손실 함수의 값이 줄어들지 않고 모델이 학습되지 않을 수 있음.
- **에폭(epoch)** : 전체 데이터 세트를 사용한 회수 (세대)



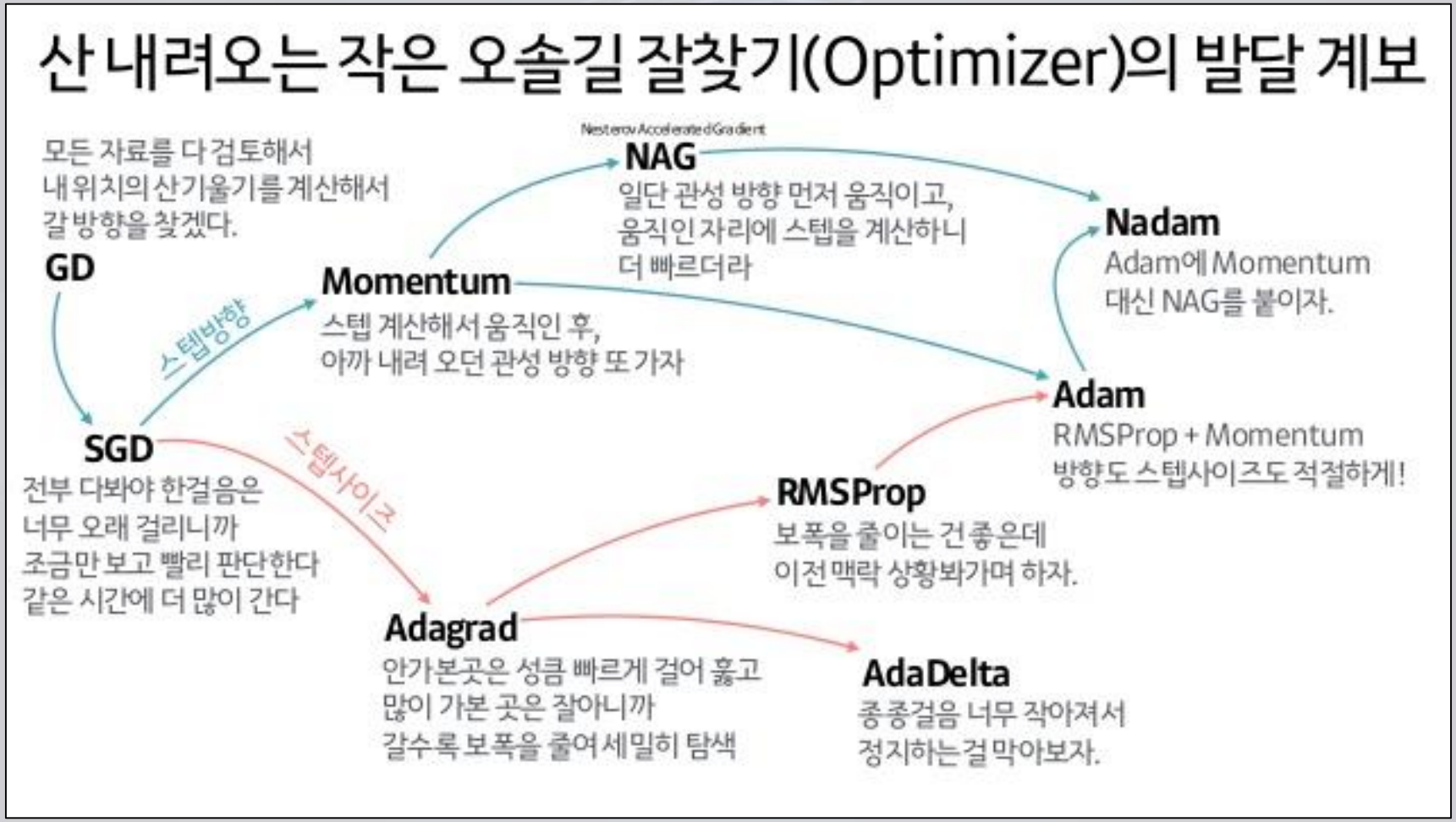
https://thinkingandcomputing.com/posts/genetic-algorithms-neural-networks.html

고급 경사하강법의 종류

- lr(learning rate) 값만 적절히 변경하고 epsilon, rho, decay는 그대로 사용하는 것을 권장

종류	개요	효과	케라스 사용법
SGD	랜덤하게 추출한 일부 데이터를 사용해 더 빨리, 자주 업데이트 하게 하는 것	속도개선	keras.optimizers.SGD(lr=0.1)
모멘텀	관성의 방향을 고려해 진동과 폭을 줄이는 효과	정확도개선	keras.optimizers.SGD(lr=0.1, momentum=0.9)
NAG	모멘텀이 이동시킬 방향으로 미리 이동해서 경사를 계산, 불필요한 이동을 줄이는 효과	정확도개선	keras.optimizers.SGD(lr=0.1, momentum=0.9, nesterov=true)
Adagrad	변수의 업데이트가 잦으면 학습률을 적게하여 이동 보폭을 조절하는 방법	보폭 크기 개선	keras.optimizers.Adagrad(lr=0.01, epsilon=1e-6)
RMSProp	Adagrid의 보폭 민감도를 보완한 방법	보폭 크기 개선	keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
Adam	모멘텀과 RMSProp를 합친 방법	정확도, 보폭 크기개선	keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)

고급 경사하강법의 종류

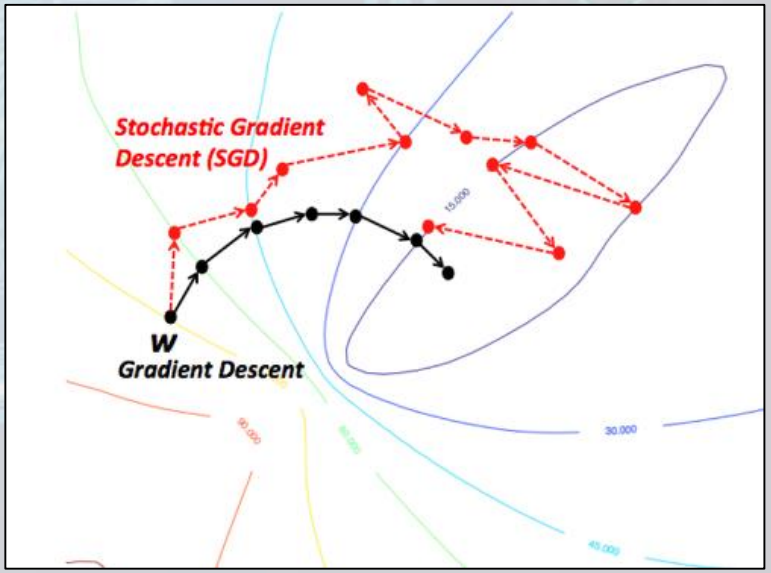

산 내려오는 작은 오솔길 잘찾기(Optimizer)의 발달 계보
모든 자료를 다 검토해서
내 위치의 산기울기를 계산해서
갈 방향을 찾겠다.
GD
스텝방향
SGD
전부 다봐야 한걸음은
너무 오래 걸리니까
조금만 보고 빨리 판단한다
같은 시간에 더 많이 간다
Momentum
스텝 계산해서 움직인 후,
아까 내려 오던 관성 방향 또 가자
Nesterov Accelerated Gradient
NAG
일단 관성 방향 먼저 움직이고,
움직인 자리에 스텝을 계산하니
더 빠르더라
Nadam
Adam에 Momentum
대신 NAG를 붙이자.
Adam
RMSProp + Momentum
방향도 스텝사이즈도 적절하게!
스텝사이즈
RMSProp
보폭을 줄이는 건 좋은데
이전 맥락 상황봐가며 하자.
Adagrad
안가본곳은 성큼 빠르게 걸어 훑고
많이 가본 곳은 잘아니까
갈수록 보폭을 줄여 세밀히 탐색
AdaDelta
종종걸음 너무 작아져서
정지하는걸 막아보자.

확률적 경사하강법 (SGD)

- 경사하강법의 많은 계산량(batch)은 속도를 느리게 하고 최적해를 찾기 전에 멈출 수도 있음.
- **SGD (Stochastic Gradient Descent)** : 전체 데이터가 아닌 랜덤하게 추출한 일부 데이터(Mini-batch)를 사용하는 방법 → 더 빠르고 자주 업데이터 가능
- 중간 결과의 진폭이 크고 불안정해 보일 수 있지만 빠르고 최적해에 근사한 값을 찾아내는 장점

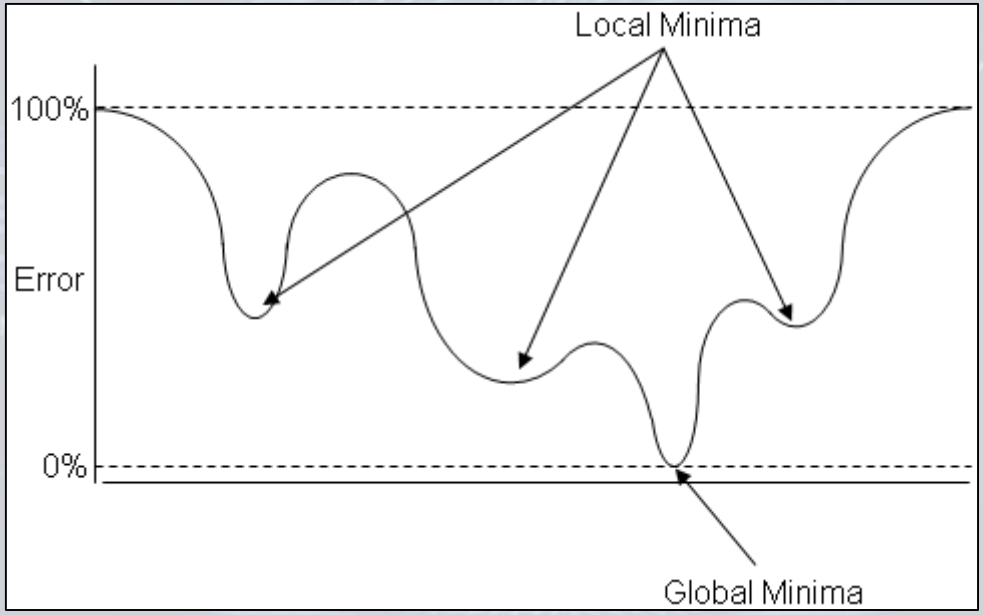
$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

- W : 업데이트할 가중치
- η : 학습률 (일반적으로 0.01이나 0.001을 사용)
- $\frac{\partial L}{\partial W}$: W에 대한 손실함수의 기울기



모멘텀

- 확률적 경사하강법은 방향에 따라서 기울기 값이 달라지는 경우에 적합하지 않음 → 최소 기울기 방향으로 움직이므로 (Local Minima에 빠질 수 있음)



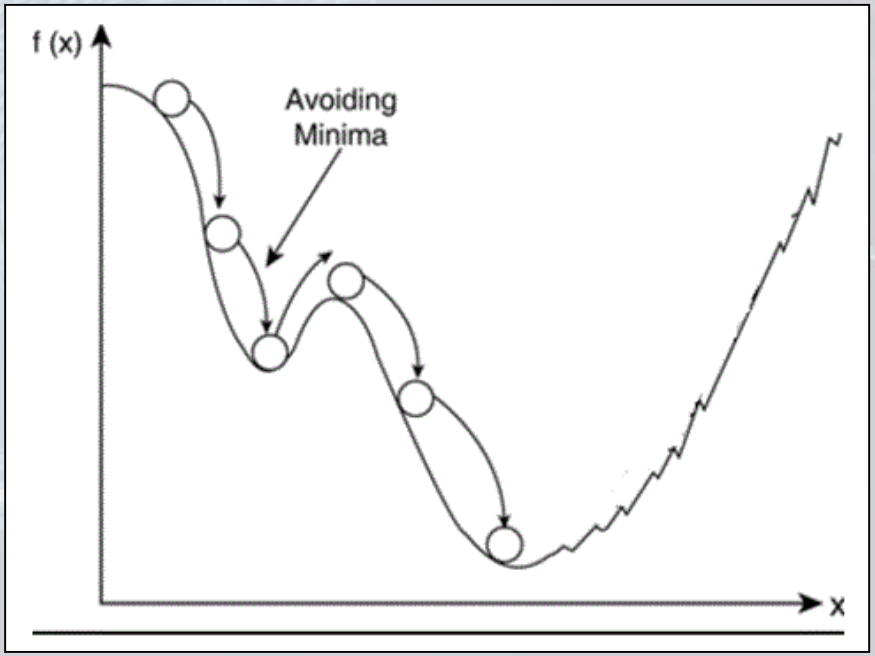
모멘텀

- 관성을 부여(업데이트에 사용했던 기울기의 일정 %를 남겨서 현재 기울기에 더하여 업데이트)하는 방법

$$v \leftarrow av - \eta \frac{\partial L}{\partial W}$$

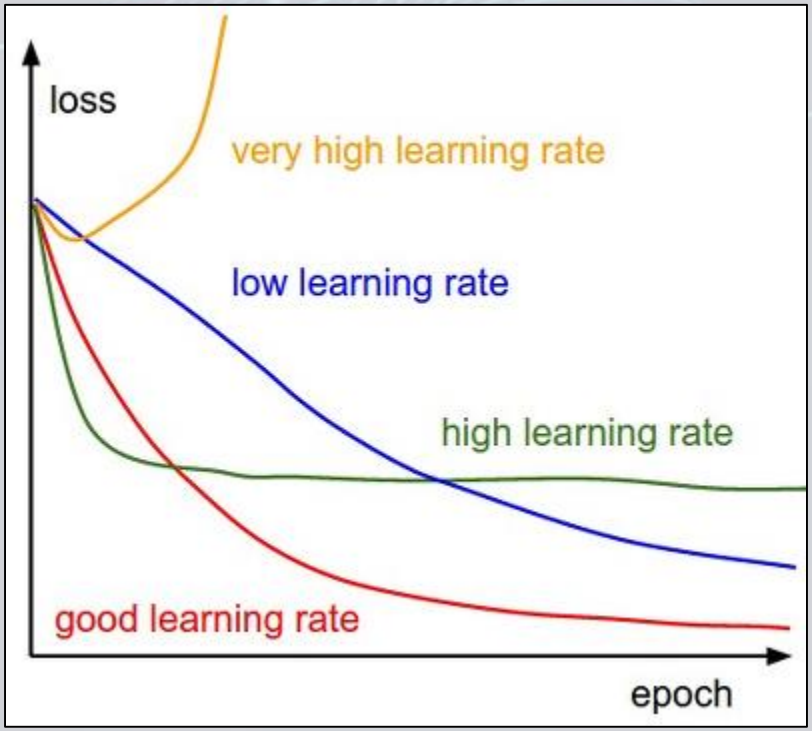
$$W \leftarrow W + v$$

- v : 속도
- a : 모멘텀 계수 (보통 0.9)



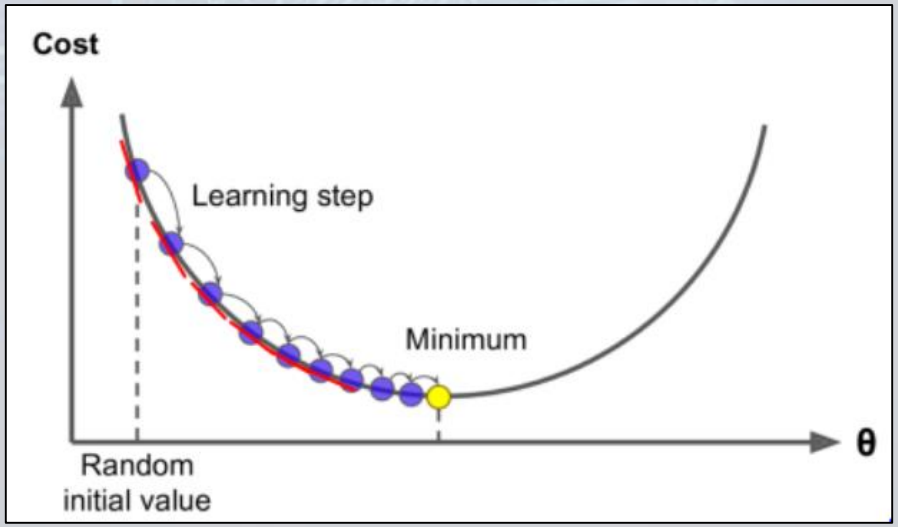
Adagrad

- 학습률 (Learning rate)는 스텝 사이즈로 작으면 학습이 오래 걸리고 크면 최적 값을 찾지 못하므로 적절한 학습률 적용이 필요



Adagrad

- 신경망에서 학습률은 이동 보폭으로 생각할 수 있는데 한번 갱신하는 가중치의 값의 크기를 결정
- 학습률을 작게하면 학습 시간이 느리고 크게하면 최적 점을 지나칠 수 있음.
- 학습 과정에서 점차 기울기가 감소하므로 학습률을 줄여가는데 (**Learning rate decay**) 보통 과거의 기울기 값을 제곱해서 더하는 하는 방식을 사용 → SD, SGD, 모멘텀처럼 동일하게 줄이지 않고 이전 기울기를 참조하는 적응형 방법



Adagrad

- 매개변수 원소 중에서 많이 움직인 (크게 갱신된) 원소는 학습률이 낮아지는 방식으로 학습률이 매개변수에 따라 변경되도록 하는 방식

$$h \leftarrow h + \frac{\partial L}{\partial W} \otimes \frac{\partial L}{\partial W}$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

- h : 기존 기울기들의 제곱의 합

RMSprop

- AdaGrad는 제곱을 하여 학습률을 줄이기 때문에 빠르게 0에 가까워져 학습이 멈추는 문제가 발생
- RMSprop는 과거의 모든 기울기를 균등하게 더하지 않고 새로운 기울기의 정보만 반영하도록 해서 학습률이 크게 떨어져 0에 가까워지는 것을 방지하는 방법

Adam

- 모멘텀과 AdaGrad 방식을 섞은 방식으로 관성 계수를 사용하여 Local Minima에 빠지지 않고 학습률에 대한 계수를 적응형을 변경하는 방식

고급 경사하강법의 동작 비교