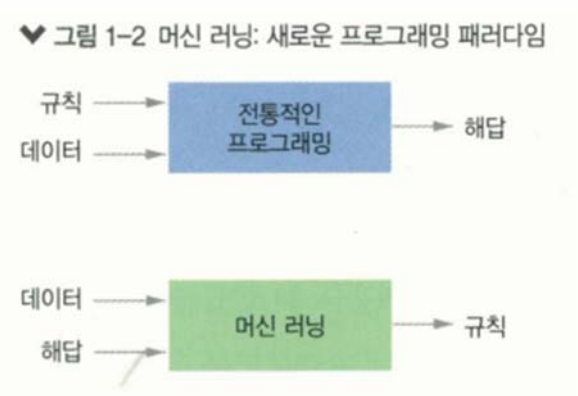
텐서플로우 빅데이터 분석

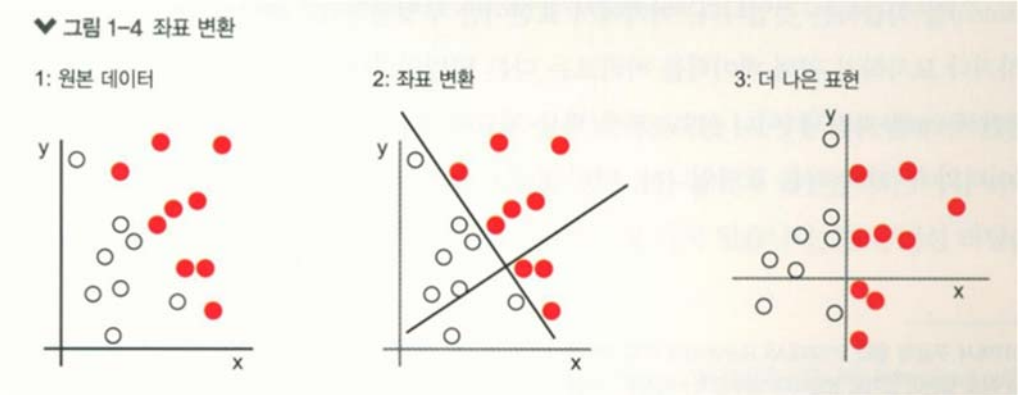
1. 참고도서

1.1 텐서플로 딥러닝 원리 공부

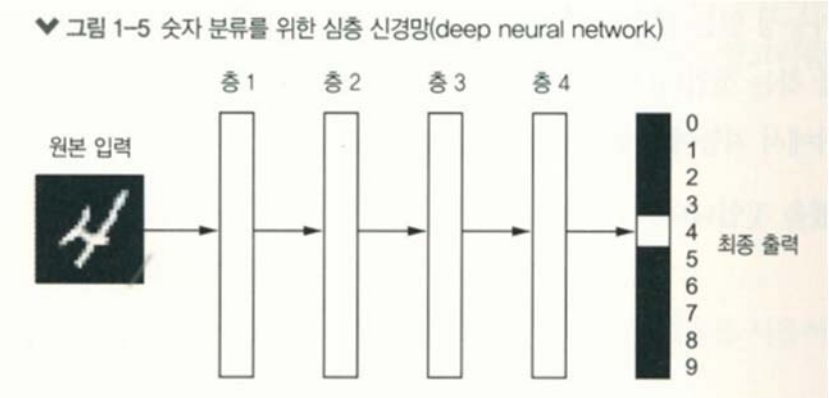
- 머신러닝은 데이터에서 규칙을 찾는 것이다



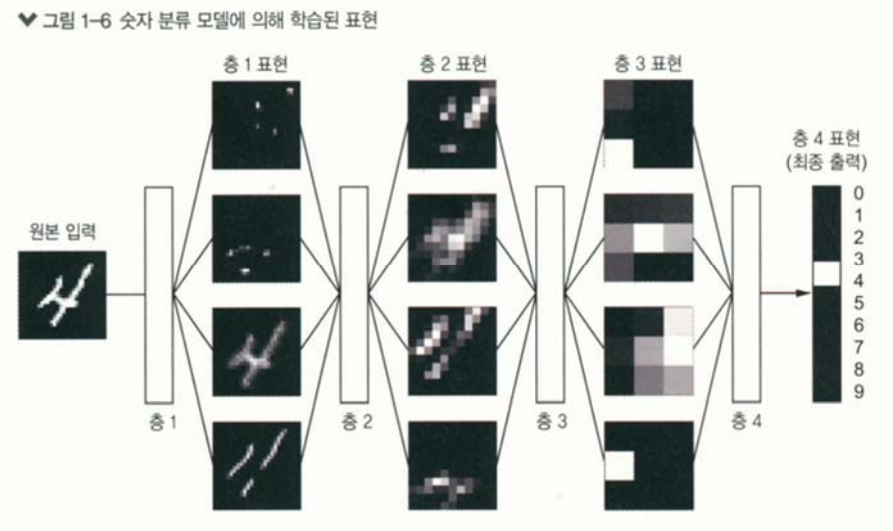
- 데이터 변환 - 데이터 분류 작업을 더 쉽게 해결



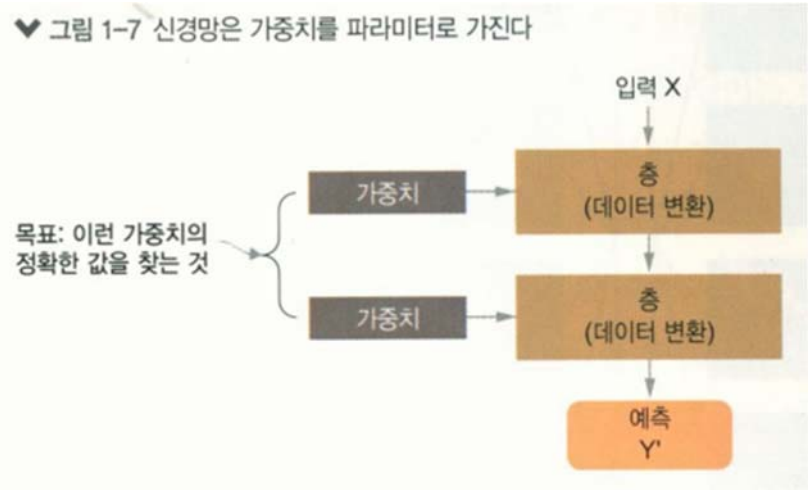
- 딥러닝 알고리즘으로 학습된 표현 > 각 층이 filter



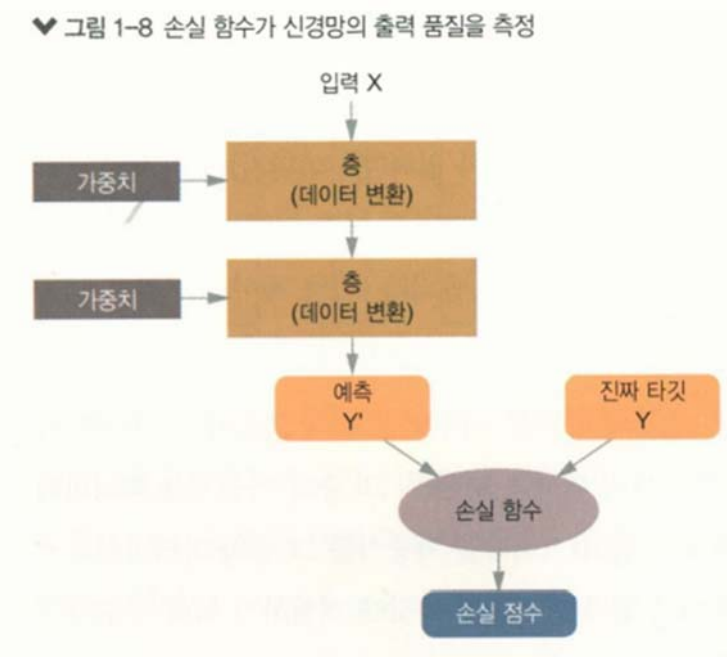
- 다층 구조에 의한 학습



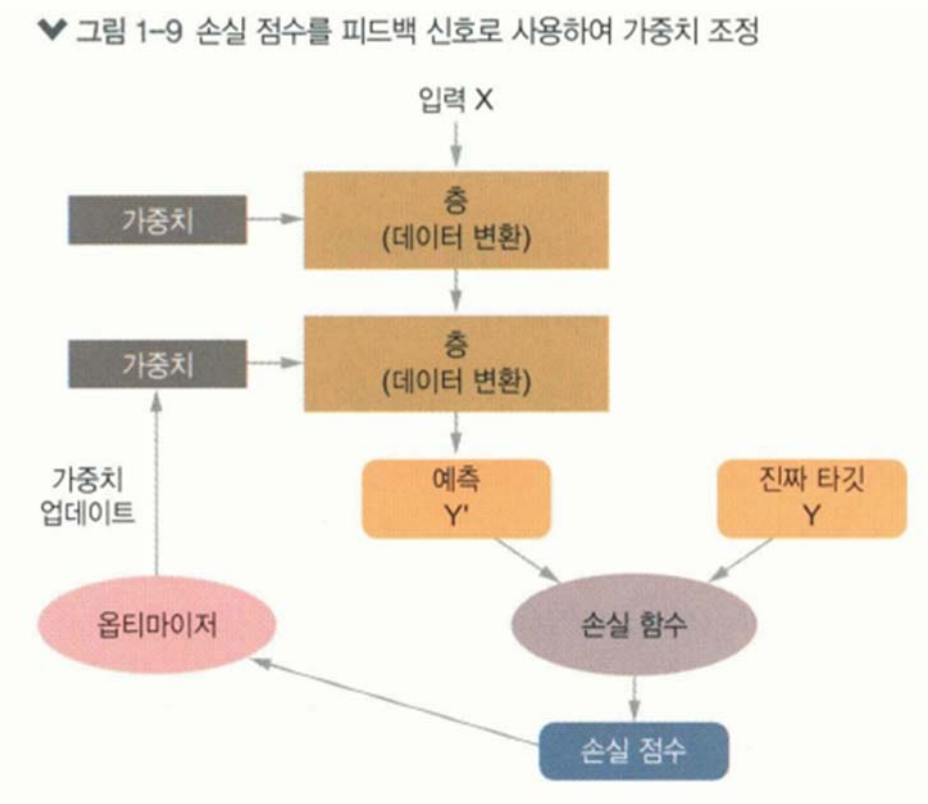
- 학습이란? 각 층에서 가중치(weight)를 찾는 것 = 입력 X에 대하여 예측 Y'를 만들어 주는 2개 층의 가중치 값을 찾는 것이 학습이다



- 예측 값이 기대 값에 대하여 얼마나 벗어났는가? loss function

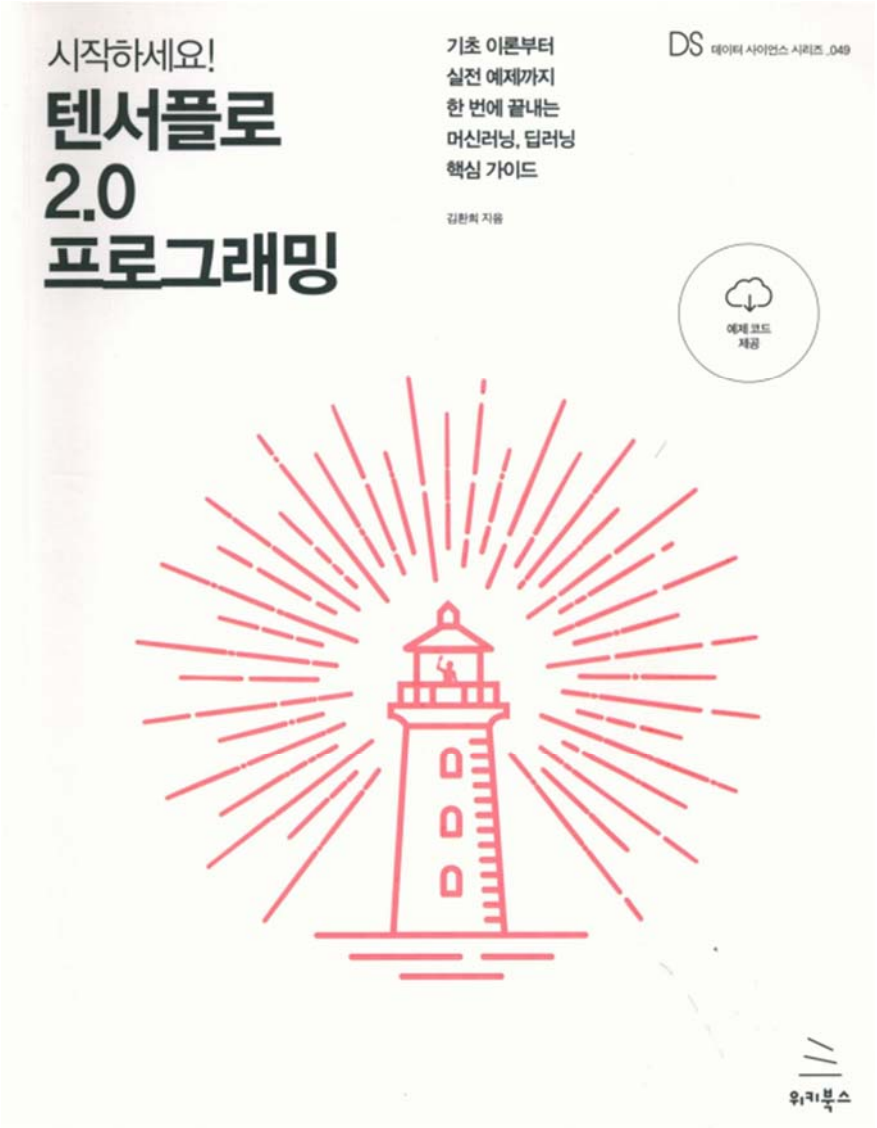


- 손실 점수를 feedback 신호로 사용하여 현재 sample(입력)의 손실 점수가 감소되는 방향으로 가중치 값을 조금씩 수정: 역전파(backtracking) 알고리즘을 구현한 optimizer가 수행



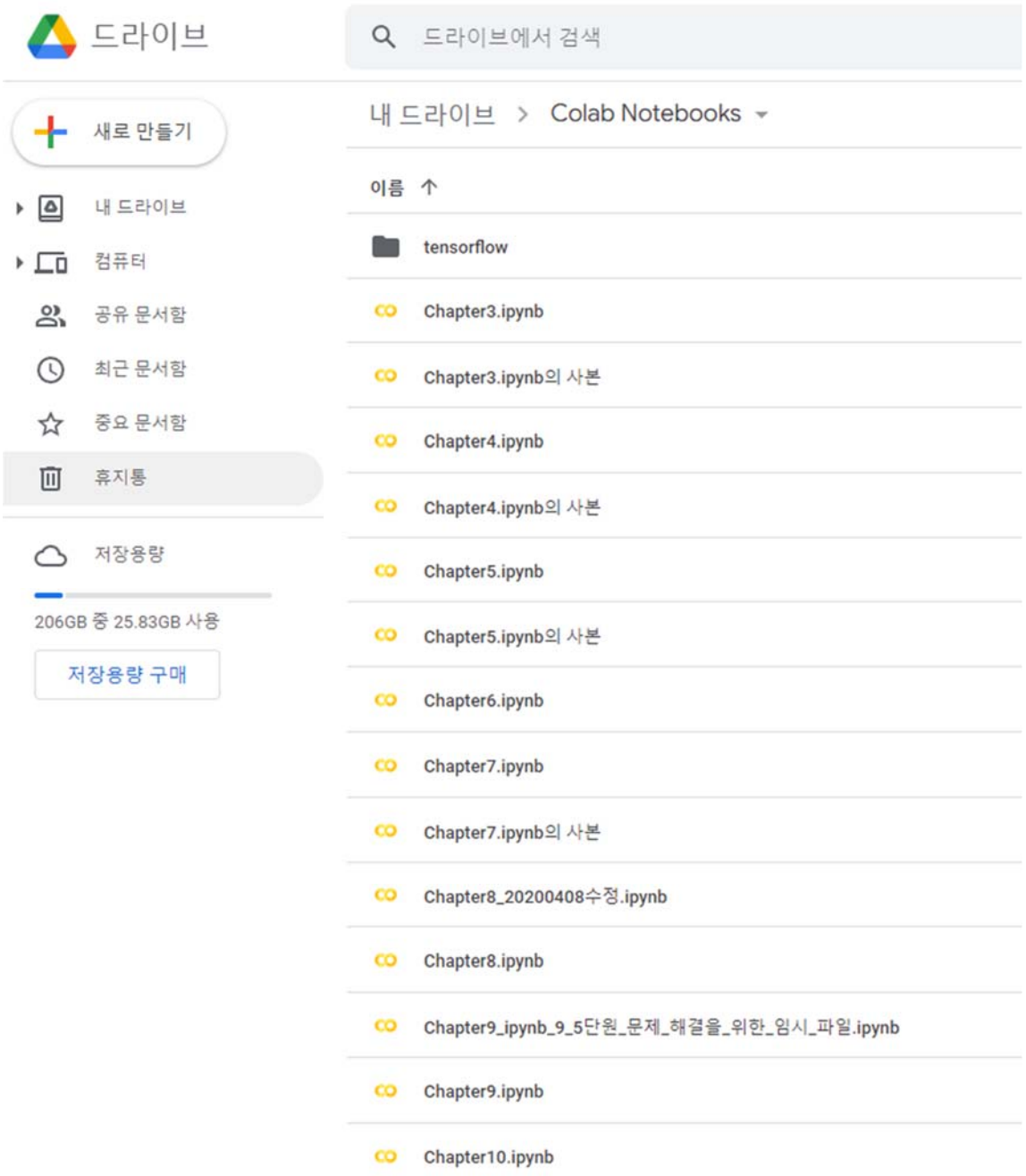
- 훈련 반복(training loop)을 통해 가중치가 조금씩 올바른 방향으로 조정 > 손실 점수가 감소

2. 텐서플로우 프로그래밍 참고도서:



- 구글 코랩 실습 코드:

- 구글드라이브에 upload한 후에 해당 파일을 클릭하면 구글 코랩 작동


드라이브
드라이브에서 검색
새로 만들기
내 드라이브
컴퓨터
공유 문서함
최근 문서함
중요 문서함
휴지통
저장용량
206GB 중 25.83GB 사용
저장용량 구매
내 드라이브 > Colab Notebooks
이름
tensorflow
Chapter3.ipynb
Chapter3.ipynb의 사본
Chapter4.ipynb
Chapter4.ipynb의 사본
Chapter5.ipynb
Chapter5.ipynb의 사본
Chapter6.ipynb
Chapter7.ipynb
Chapter7.ipynb의 사본
Chapter8_20200408수정.ipynb
Chapter8.ipynb
Chapter9_ipynb_9_5단원_문제_해결을_위한_임시_파일.ipynb
Chapter9.ipynb
Chapter10.ipynb

- 데이터 변환 층 구조

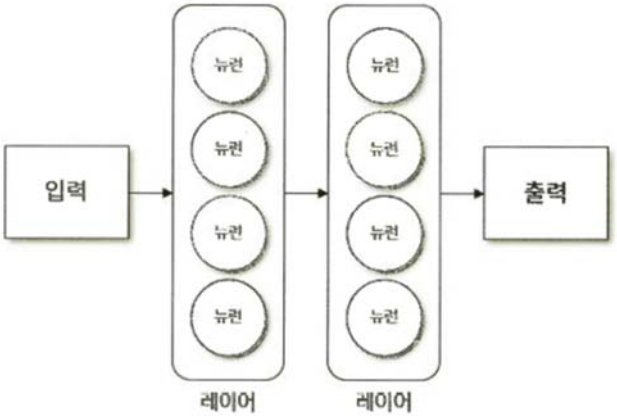
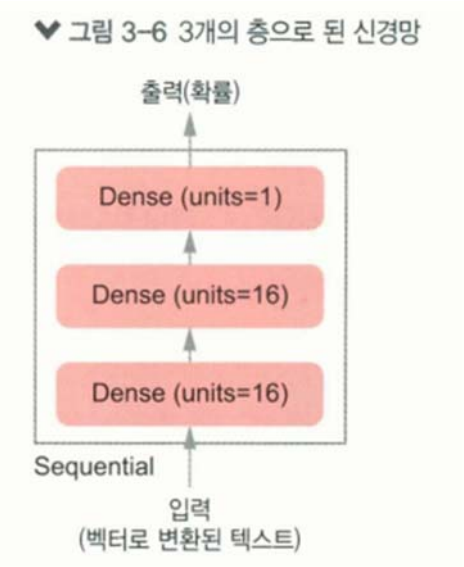


그림 3.7 뉴런과 레이어로 구체화한 신경망의 구조

- 딥러닝의 구성 단위: 층(layer)



- 3개 층으로 구성된 모델 구성

코드 3-3 모델 정의하기

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- 뉴런의 가중치 사용한 활성화 함수

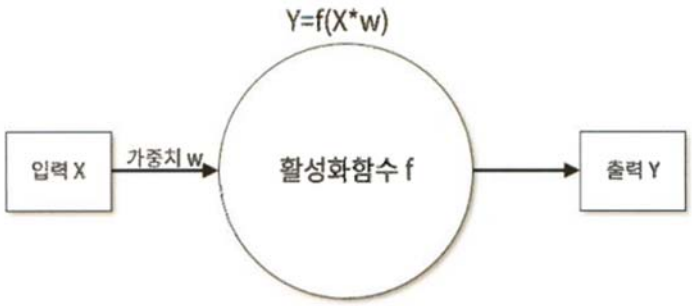


그림 3.9 뉴런의 출력 계산식

- 뉴론 학습 사례: 초기 가중치는 random 값을 사용, 학습 과정에서 일정한 값으로 수렴

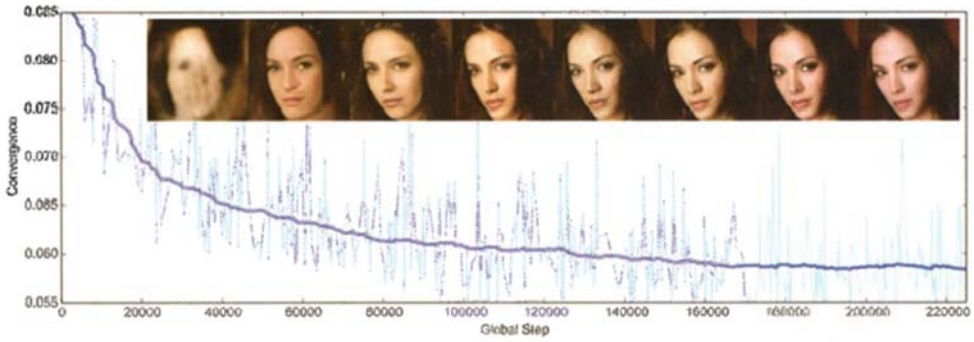


그림 3.10 딥러닝 학습을 이용한 가상 인물의 생성 사례[8]

- 활성화 함수: sigmoid, ReLU(Reectified Linear Unit)

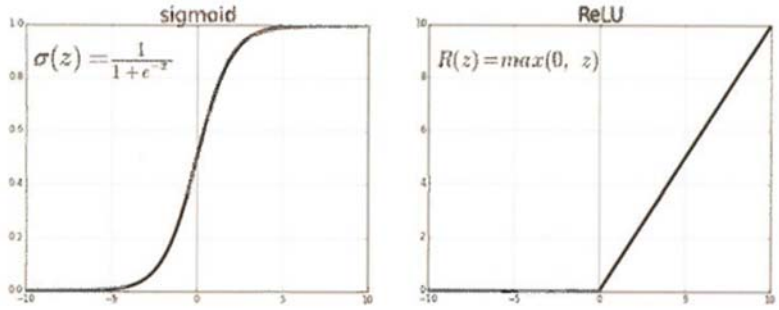


그림 3.11 시그모이드와 ReLU 함수의 그래프

+ 시그모이드: 0 ~ 1, 역전파시에 시그모이드 함수가 값을 작아지게 하는 문제점

+ ReLU: 음수는 0으로, 양수는 그대로 > 왜곡이 적어진다

- 시그모이드 함수

3.3.2 뉴런 만들기

```
# 3.10 sigmoid 함수
import math
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

- 간단한 뉴런 예

```
# 3.11 뉴런의 입력과 출력 정의
x = 1
y = 0
w = tf.random.normal([1],0,1)
output = sigmoid(x * w)
print(output)
```

```
0.6151933278951046
```

+ 실제 출력: 0.43, 기대값 :0, 에러: 0 - 0.43 = -0.43

+ 뉴론의 학습: 에러가 0이 되게하는 가중치 구하는 것

+ 뉴론: w 값

- 경사 하강법을 사용한 활성화 함수

```
# 3.12 경사 하강법을 이용한 뉴런의 학습
for i in range(1000):
    output = sigmoid(x * w)
    error = y - output
    w = w + x * 0.1 * error

    if i % 100 == 99:
        print(i, error, output)
```

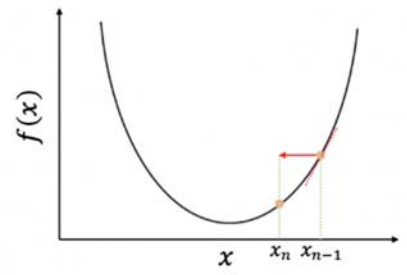
```
99 -0.11042229100036205 0.11042229100036205
199 -0.05456814895844875 0.05456814895844875
299 -0.03583382872044337 0.03583382872044337
399 -0.026586214708450608 0.026586214708450608
499 -0.021102184880016577 0.021102184880016577
599 -0.017480714292535105 0.017480714292535105
699 -0.014913710486740 43 0.014913710486740 43
799 -0.013000528555884554 0.013000528555884554
899 -0.011520273396928904 0.011520273396928904
999 -0.010341291290596 18 0.010341291290596 18
```

> x = 1, y = 0,에 대하여 적용되는 활성화 함수 y = f(x*w) = sigmoid(x*w)

+ 뉴런: w 값 - w 값을 변화하여 학습

+ 경사 하강법: w = w + x* 0.1 * error(0.1이 학습률(α)이고 step size이다 미분값은 error 값이다)

경사 하강법(Gradient Descent)



1-D의 경우

$$x_n = x_{n-1} - \alpha \frac{df(x_{n-1})}{dx}$$

N-D 의 경우

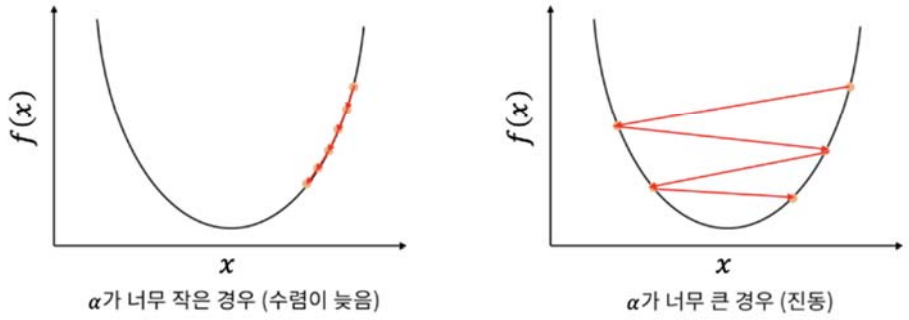
$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} - \alpha \nabla f(\boldsymbol{x}_{n-1})$$

α: 학습률 (Learning rate)

경사 하강법의 Step

경사 하강법(Gradient Descent)의 학습률(Learning rate)

$$x_n = x_{n-1} - \alpha \nabla f(x_{n-1})$$



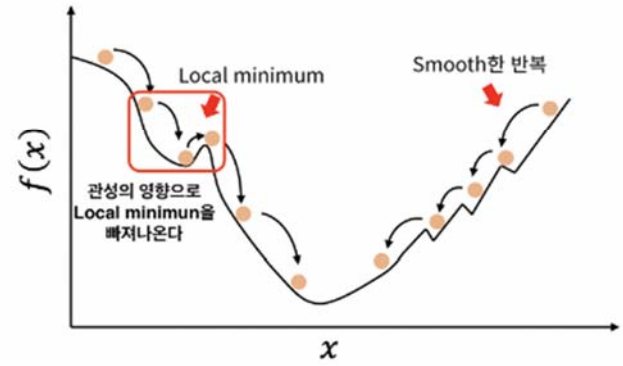
α가 너무 작은 경우 (수렴이 늦음)

α가 너무 큰 경우 (진동)

+ 적절한 학습률(Learning rate)을 선택하지 못하는 경우

- Stochastic Gradient Descent(SGD): Mini-batch로 분할해 빠르게 전진

Momentum



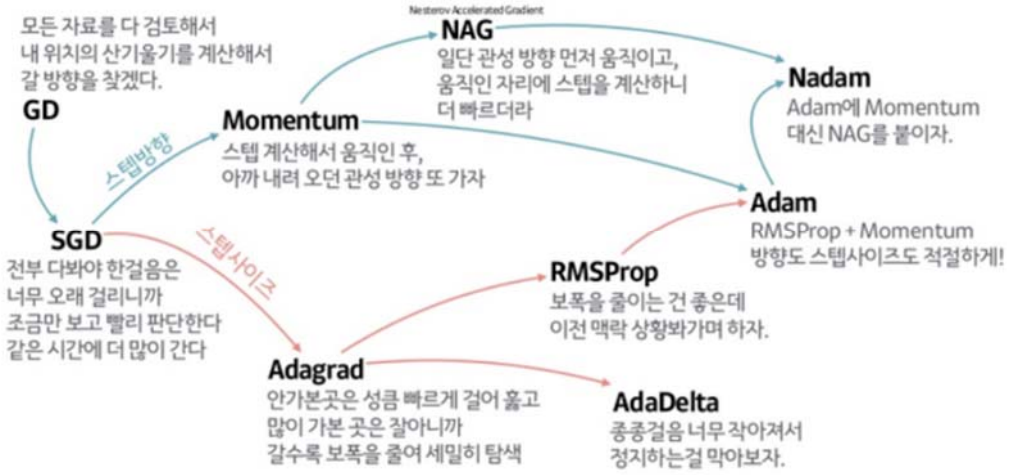
$$v_t = \gamma v_{t-1} + \eta \nabla f(x_{t-1})$$
$$x_t = x_{t-1} - v_t$$

γ : 관성 계수 (momentum term) ≈ 0.9

η : 학습율 (Learning rate)

v_t : t번째 step에서의 x의 이동 벡터

- optimization 계보



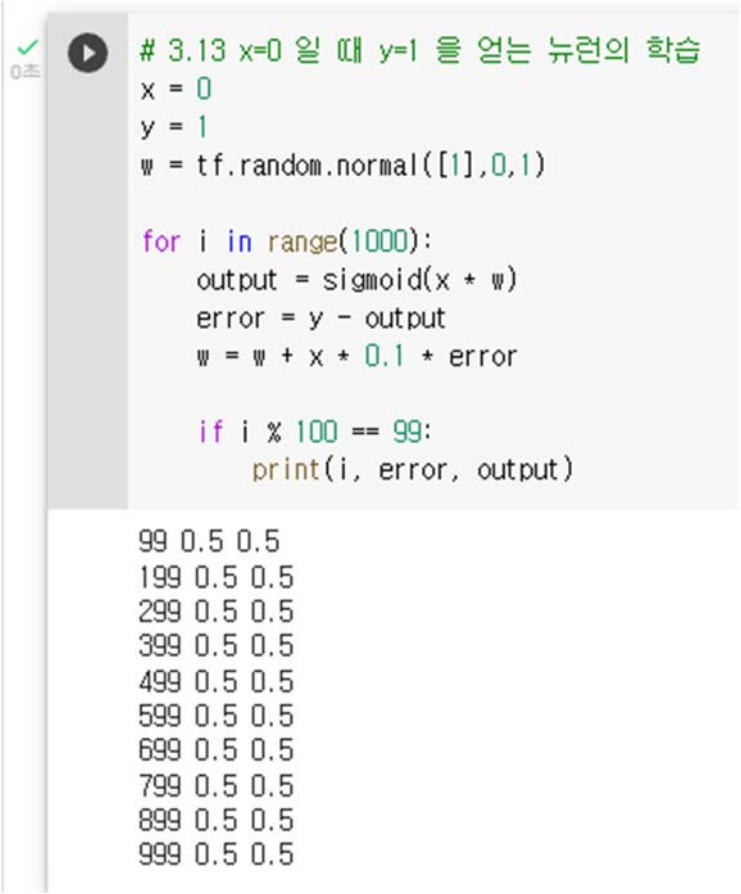
+ stochastic gradient descent: 전체 데이터 중 단 하나의 데이터를 이용하여 경사 하강법을 1회

진행(배치 크기 = 1) > 전체 학습 데이터 중에서 random하게 하나의 데이터를 선택 => stochastic

+ min-batch stochastic gradient descent > 전체 데이터를 batch size로 나누어 배치로 학습

- 뉴런에 편향(bias)를 사용해야 하는 경우

+ 경사하강법 계산식: w = w + x * a * error (error = y - sigmoid(x*w))



```
# 3.13 x=0 일 때 y=1 을 얻는 뉴런의 학습
x = 0
y = 1
w = tf.random.normal([1],0,1)

for i in range(1000):
    output = sigmoid(x * w)
    error = y - output
    w = w + x * 0.1 * error

    if i % 100 == 99:
        print(i, error, output)
```

```
99 0.5 0.5
199 0.5 0.5
299 0.5 0.5
399 0.5 0.5
499 0.5 0.5
599 0.5 0.5
699 0.5 0.5
799 0.5 0.5
899 0.5 0.5
999 0.5 0.5
```

> output = sigmoid(x*w + b)

+ x = 0, y = 1일때 학습일 때 output = 0.5, error = 0.5 변동 없음

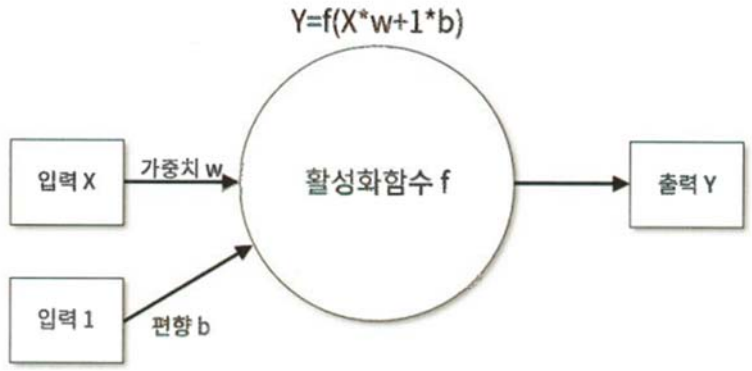


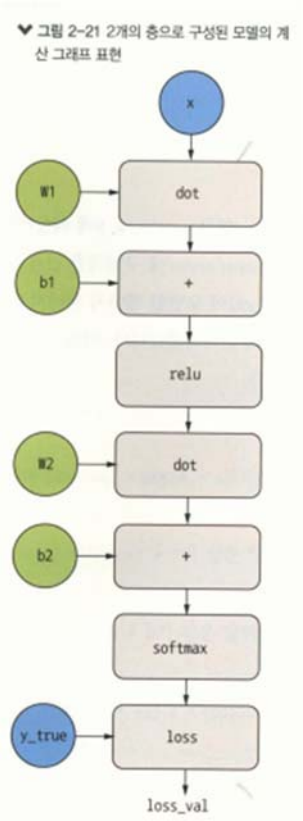
그림 3.12 편향이 더해진 뉴런의 출력 계산식

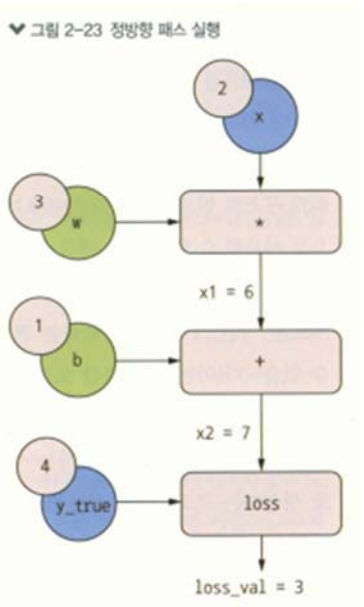
- bias를 사용한 학습 뉴런 모델

```
# 3.14 x=0 일 때 y=1 을 얻는 뉴런의 학습에 편향을 더함
x = 0
y = 1
w = tf.random.normal([1],0,1)
b = tf.random.normal([1],0,1)

for i in range(1000):
    output = sigmoid(x * w + 1 * b)
    error = y - output
    w = w + x * 0.1 * error
    b = b + 1 * 0.1 * error

    if i % 100 == 99:
        print(i, error, output)
```

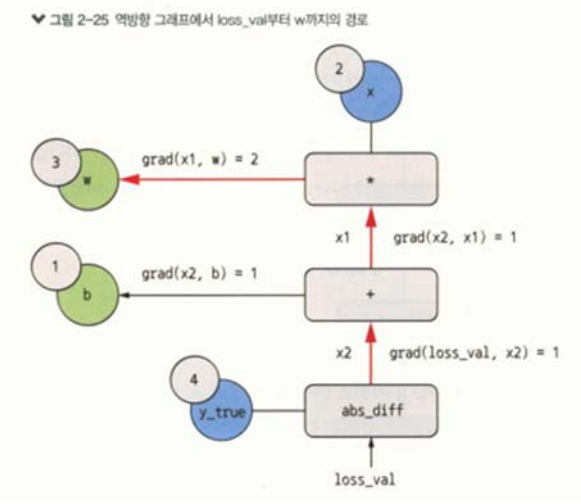
```
99 0.05613929410015883 0.9438607058998412
199 0.036517984833149386 0.9634820151668506
299 0.026964494466793054 0.973035505533207
399 0.021341149037506058 0.9786588509624939
499 0.01764497585304714 0.9823550241469529
599 0.015033419259517289 0.9849665807404827
699 0.013091573572549864 0.986908426427451
799 0.011591766057125419 0.9884082339428746
899 0.010398927668489 6 0.9896010723315104
999 0.009427771543452201 0.9905722284565478
```

- 계산 그래프를 사용한 자동 미분





- 역전파 : 역방향 그래프



3. 신경망 네트워크 학습: AND 모델

- x1 & x2 모델

표 3.2 2개의 정수 입력을 받을 때 AND 연산의 진리표

입력 1	입력 2	AND 연산
1	1	1
1	0	0
0	1	0
0	0	0

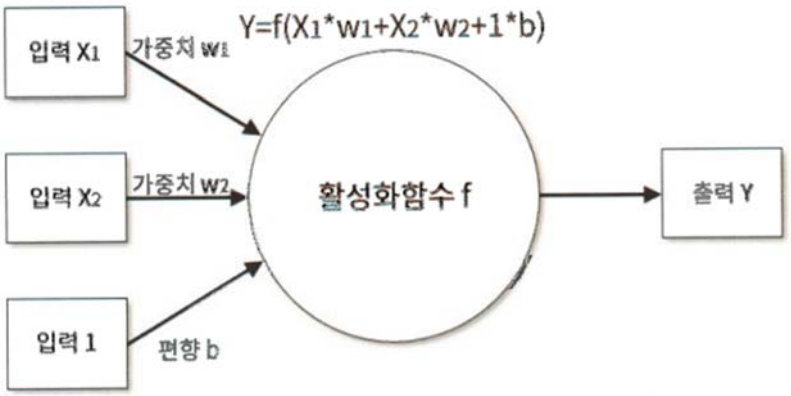


그림 3.13 입력이 2개, 편향이 1개인 뉴런의 출력 계산식

```
# 3.16 첫번째 신경망 네트워크 : AND
import numpy as np
x = np.array([[1,1], [1,0], [0,1], [0,0]])
y = np.array([[1], [0], [0], [0]])
w = tf.random.normal([2],0,1)
b = tf.random.normal([1],0,1)
b_x = 1

for i in range(2000):
    error_sum = 0
    for j in range(4):
        output = sigmoid(np.sum(x[j]*w)+b_x*b)
        error = y[j][0] - output
        w = w + x[j] * 0.1 * error
        b = b + b_x * 0.1 * error
        error_sum += error

    if i % 200 == 199:
        print(i, error_sum)
```

```
199 -0.11085527470487731
399 -0.06589773131529673
599 -0.04677397040203471
799 -0.03614882251299572
999 -0.02940277980158417
1199 -0.024749751696087993
1399 -0.021352154675056505
1599 -0.018765476620651037
1799 -0.016730131667345667
1999 -0.015089296105792674
```

- 1차 AI 겨울: XOR 네트워크

표 3.6 2개의 정수 입력을 받을 때 XOR 연산의 진리표

입력 1	입력 2	XOR 연산
1	1	0
1	0	1
0	1	1
0	0	0

3.3.5 세번째 신경망 네트워크 : XOR

```
# 3.23 세번째 신경망 네트워크 : XOR
import numpy as np
x = np.array([[1,1], [1,0], [0,1], [0,0]])
y = np.array([[0], [1], [1], [0]])
w = tf.random.normal([2],0,1)
b = tf.random.normal([1],0,1)
b_x = 1

for i in range(2000):
    error_sum = 0
    for j in range(4):
        output = sigmoid(np.sum(x[j]*w)+b_x*b)
        error = y[j][0] - output
        w = w + x[j] * 0.1 * error
        b = b + b_x * 0.1 * error
        error_sum += error

    if i % 200 == 199:
        print(i, error_sum)
```

```
199 -0.008864795391776759
399 -0.0003605671627422 4243
599 -1.4665165693705795e-05
799 -6.031003636497445e-07
999 3.7228424787372205e-09
1199 3.722842145670313e-09
1399 3.722842145670313e-09
1599 3.722842145670313e-09
1799 3.722842145670313e-09
1999 3.722842145670313e-09
```

+ output이 0.5에 수렴

```
# 3.24 XOR 네트워크의 평가
for i in range(4):
    print('X:', x[i], 'Y:', y[i], 'Output:', sigmoid(np.sum(x[i]*w)+b))
```

```
X: [1 1] Y: [0] Output: 0.5128176286712095
X: [1 0] Y: [1] Output: 0.5128176305326305
X: [0 1] Y: [1] Output: 0.4999999990686774
X: [0 0] Y: [0] Output: 0.5000000009313226
```

```
# 3.25 XOR 네트워크의 w, b 값 확인
print('w:', w)
print('b:', b)
```

```
w: tf.Tensor([ 5.1281754e-02 -7.4505806e-09], shape=(2,), dtype=float32)
b: tf.Tensor([3.7252903e-09], shape=(1,), dtype=float32)
```

```
# 3.26 AND 네트워크의 w, b 값 확인
# w: tf.Tensor([6.9484286 6.951607 ], shape=(2,), dtype=float32)
# b: tf.Tensor([-10.601849], shape=(1,), dtype=float32)
```

+ 활성화 함수 계산 값

표 3.8 AND 네트워크의 입력과 중간 계산, 출력

X1	X2	중간 계산 np.sum(x[j]*w)+b	출력 sigmoid(np.sum(x[j]*w)+b)
1	1	3.2981866	0.9643665480247O8
1	0	-3.6534204	0.025248387249846 36
0	1	-3.650242	0.025326728701507022
0	0	-10.601849	2.4869364094058595e-05

+ XOR, AND 뉴론의 가중치 차이 비교

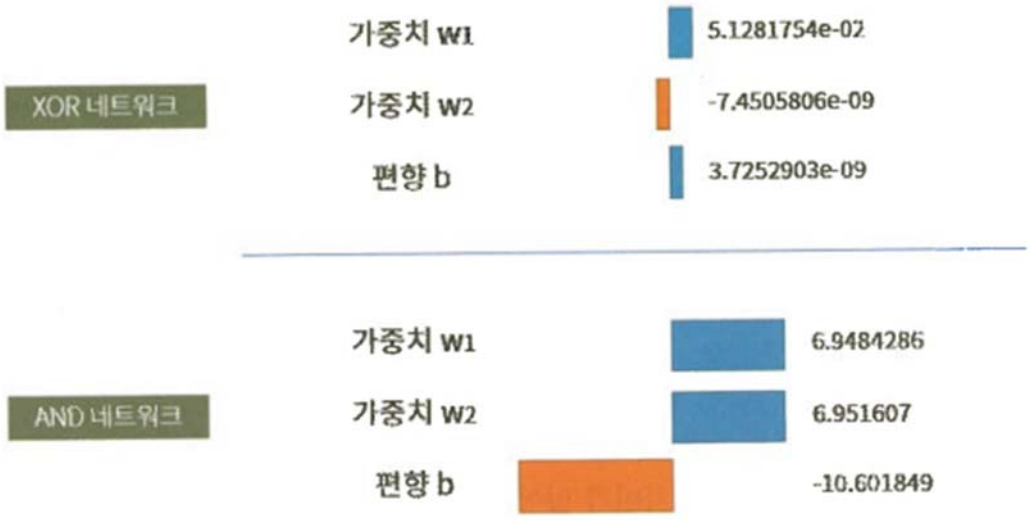


그림 3.14 XOR과 AND 네트워크의 가중치 그래프화

+ XOR 문제: 하나의 뉴론으로 XOR 구현 불가능

> 해결: 여러개의 뉴론을 사용

```
# 3.27 tf.keras 를 이용한 XOR 네트워크 계산
import numpy as np
x = np.array([[1,1], [1,0], [0,1], [0,0]])
y = np.array([[0], [1], [1], [0]])

model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=2, activation='sigmoid', input_shape=(2,)),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.3), loss='mse')

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 6

 dense_1 (Dense)             (None, 1)                 3

=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

- sequential 모델

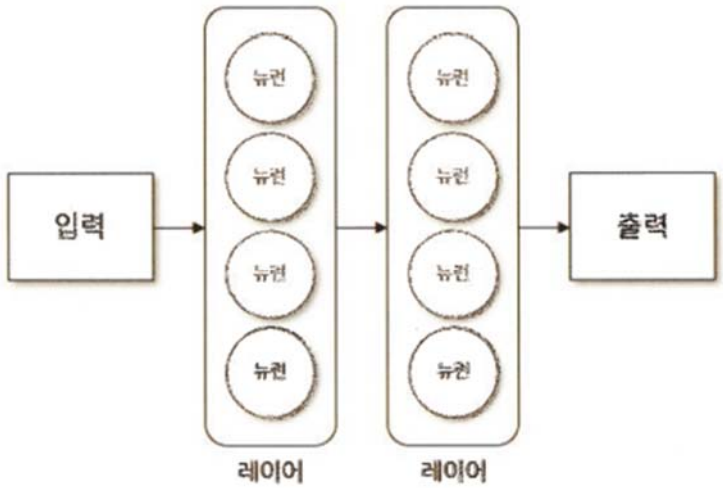


그림 3.15 tf.keras.sequential의 일직선 구조

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=2, activation='sigmoid', input_shape=(2,)),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])
```

-

- XOR 뉴론 다층 구조

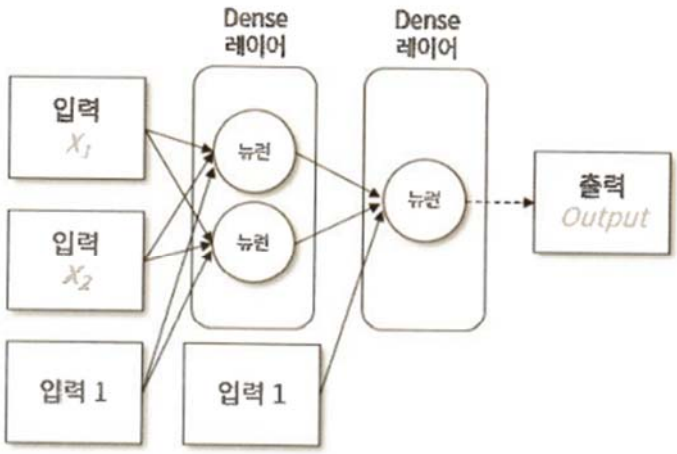


그림 3.17 편향을 포함한 2-레이어 XOR 네트워크의 구조

- XOR 학습

```
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.1), loss='mse')
```

+ optimizer: 딥러닝 학습 식

> SGD(Stochastic Gradient Descent): w = w + x[j] * 0.1 * error의 계산시에 경사 하강법으로 미분 기울기가 0이 되는 쪽으로 하강하도록 update

> stochastic은 확률적으로 일부 sample을 구해서 계산

> loss는 error와 유사한 개념

> mse = Mean Squared Error

$$\text{Mean Squared Error} = \frac{1}{n}\sum_{k=1}^{n}(y_k - output_k)^2$$

> error = y - output 계산식과 유사

- 네트워크 학습

```
# 3.28 tf.keras 를 이용한 XOR 네트워크 학습
history = model.fit(x, y, epochs=2000, batch_size=1)
```

```
4/4 [==============================] - 0s 2ms/step - loss: 0.2631
Epoch 263/2000
4/4 [==============================] - 0s 2ms/step - loss: 0.2631
Epoch 264/2000
4/4 [==============================] - 0s 2ms/step - loss: 0.2622
Epoch 265/2000
4/4 [==============================] - 0s 2ms/step - loss: 0.2621
Epoch 266/2000
4/4 [==============================] - 0s 2ms/step - loss: 0.2629
Epoch 267/2000
4/4 [==============================] - 0s 3ms/step - loss: 0.2630
Epoch 268/2000
4/4 [==============================] - 0s 3ms/step - loss: 0.2629
Epoch 269/2000
```

- 네트워크 평가

```
# 3.29 tf.keras 를 이용한 XOR 네트워크 평가
model.predict(x)
```

```
1/1 [==============================] - 0s 153ms/step
array([[0.09728624],
       [0.92214876],
       [0.919878  ],
       [0.08459373]], dtype=float32)
```

- 학습 결과로 얻어진 가중치와 bias 계산 확인

```
# 3.30 XOR 네트워크의 가중치와 편향 확인
for weight in model.weights:
    print(weight)
```

```
<tf.Variable 'dense/kernel:0' shape=(2, 2) dtype=float32, numpy=
array([[ 7.102673 , -3.8543596],
       [ 6.3426666, -3.775786 ]], dtype=float32)>
<tf.Variable 'dense/bias:0' shape=(2,) dtype=float32, numpy=array([-2.7972329,  5.666881 ], dtype=float32)>
<tf.Variable 'dense_1/kernel:0' shape=(2, 1) dtype=float32, numpy=
array([[6.1795464],
       [6.492127 ]], dtype=float32)>
<tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32, numpy=array([-9.206421], dtype=float32)>
```

- 학습에 의한 가중치 계산 값

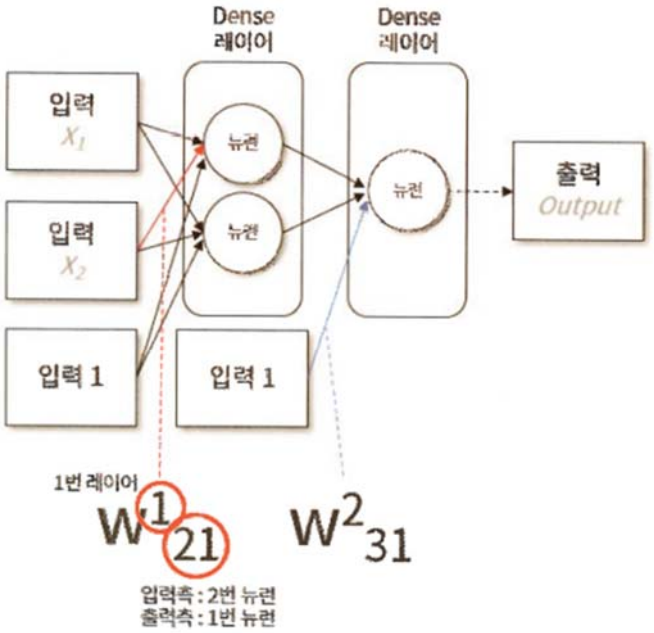


그림 3.18 가중치에 이름을 붙이는 규칙

- XOR 네트워크 가중치 값 수치

그림 3.19 2-레이어 XOR 네트워크의 가중치 그래프화

- 단순한 뉴론의 간단한 구현

output = activation(dot(w, input) + b)

```
class NaiveDense:
    def __init__(self, input_size, output_size, activation):
```

```
        self.activation = activation

        w_shape = (input_size, output_size) ····· 랜덤한 값으로 초기화된 (input_size, output_size) 크기의 행렬 W를 만듭니다.
        w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
        self.W = tf.Variable(w_initial_value)

        b_shape = (output_size,) ····· 0으로 초기화된 (output_size,) 크기의 벡터 b를 만듭니다.
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)

    def __call__(self, inputs): ····· 정방향 패스를 수행합니다.
        return self.activation(tf.matmul(inputs, self.W) + self.b)

    @property
    def weights(self): ····· 층의 가중치를 추출하기 위한 메서드
        return [self.W, self.b]
```

- 단순한 모델의 층 구현

```
model = NaiveSequential([
    NaiveDense(input_size=28 * 28, output_size=512, activation=tf.nn.relu),
    NaiveDense(input_size=512, output_size=10, activation=tf.nn.softmax)
])
assert len(model.weights) == 4
```

```
class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
            x = layer(x)
        return x

    @property
    def weights(self):
        weights = []
        for layer in self.layers:
            weights += layer.weights
        return weights
```

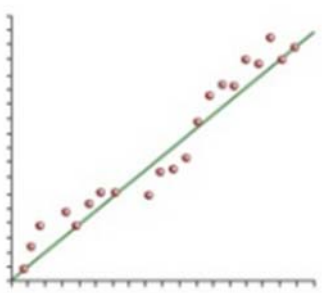
```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"에포크 {epoch_counter}")
        batch_generator = BatchGenerator(images, labels)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            if batch_counter % 100 == 0:
                print(f"{batch_counter}번째 배치 손실: {loss:.2f}")
```

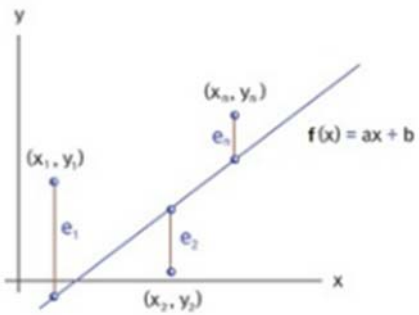
4. 딥러닝에 의한 선형 회귀 분석

- regression: an act of going or coming back

  + 평균으로 회귀(regression to the mean)

- 손실함수가 최소가 되는 가중치 a,b를 찾는 것: y = ax + b(활성화 함수)





- 손실 함수를 최소 제곱법(Method of Least Squares)으로 계산

  + 잔차의 제곱을 최소화하는 계산

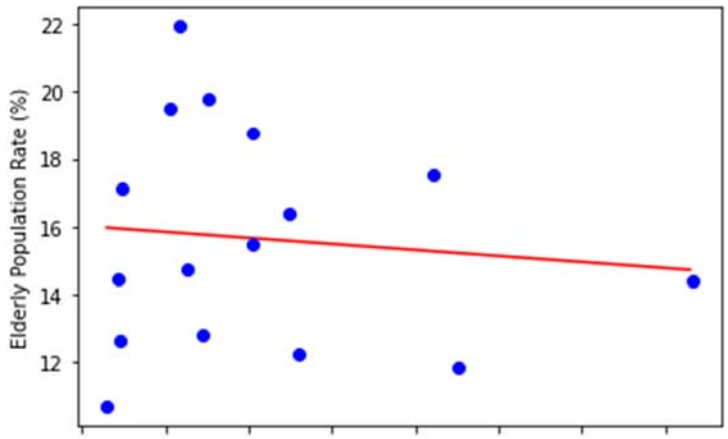
$$\sum_{i=1}^{n}(y_i - f(x_i))^2$$

  + 가중치 a,b의 계산

$$a = \frac{\sum_{i=1}^{n}(y_i - \bar{y})(x_i - \bar{x})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}, \qquad b = \bar{y} - a\bar{x}$$

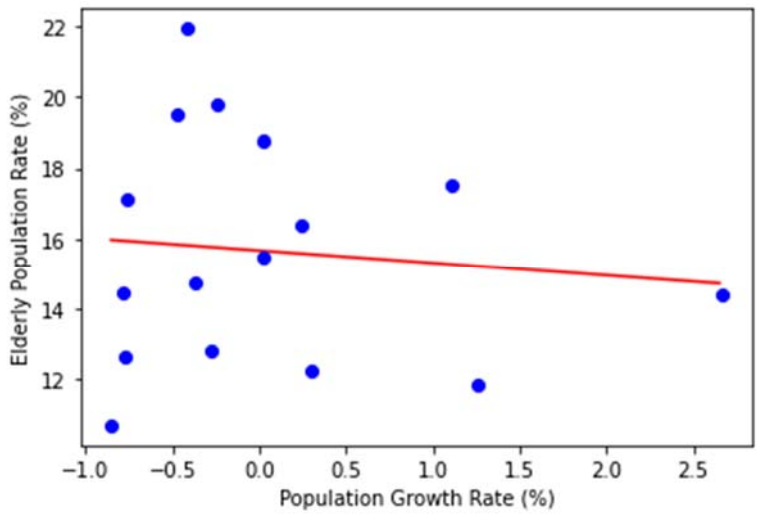
```
[ ]  # 4.3 최소제곱법으로 회귀선 구하기
     import numpy as np
     import matplotlib.pyplot as plt
     X = [0.3, -0.78, 1.26, 0.03, 1.11, 0.24, -0.24, -0.47, -0.77, -0.37, -0.85, -0.41,
     Y = [12.27, 14.44, 11.87, 18.75, 17.52, 16.37, 19.78, 19.51, 12.65, 14.74, 10.72, 2

     # X, Y의 평균을 구합니다.
     x_bar = sum(X) / len(X)
     y_bar = sum(Y) / len(Y)

     # 최소제곱법으로 a, b를 구합니다.
     a = sum([(y - y_bar) * (x - x_bar) for y, x in list(zip(Y, X))])
     a /= sum([(x - x_bar) ** 2 for x in X])
     b = y_bar - a * x_bar
     print('a:', a, 'b:', b)

     # 그래프를 그리기 위해 회귀선의 x, y 데이터를 구합니다.
     line_x = np.arange(min(X), max(X), 0.01)
     line_y = a * line_x + b

     # 붉은색 실선으로 회귀선을 그립니다.
     plt.plot(line_x,line_y,'r-')

     plt.plot(X,Y,'bo')
     plt.xlabel('Population Growth Rate (%)')
     plt.ylabel('Elderly Population Rate (%)')
     plt.show()
```

```
a: -0.355834147915461 b: 15.669317743971302
```



- 텐서플로우를 사용한 회귀선 구하기

```python
# 4.4 텐서플로를 이용해서 회귀선 구하기
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import random
X = [0.3, -0.78, 1.26, 0.03, 1.11, 0.24, -0.24, -0.47, -0.77, -0.37, -0.85, -0.41, -0.27, 0.02, -0.76, 2.66]
Y = [12.27, 14.44, 11.87, 18.75, 17.52, 16.37, 19.78, 19.51, 12.65, 14.74, 10.72, 21.94, 12.83, 15.51, 17.14, 14.42]

# a와 b를 랜덤한 값으로 초기화합니다.
a = tf.Variable(random.random())
b = tf.Variable(random.random())

# 잔차의 제곱의 평균을 반환하는 함수입니다.
def compute_loss():
    y_pred = a * X + b
    loss = tf.reduce_mean((Y - y_pred) ** 2)
    return loss

optimizer = tf.keras.optimizers.Adam(lr=0.07)
for i in range(1000):
    # 잔차의 제곱의 평균을 최소화(minimize)합니다.
    optimizer.minimize(compute_loss, var_list=[a,b])

    if i % 100 == 99:
        print(i, 'a:', a.numpy(), 'b:', b.numpy(), 'loss:', compute_loss().numpy())

line_x = np.arange(min(X), max(X), 0.01)
line_y = a * line_x + b

# 그래프를 그립니다.
plt.plot(line_x,line_y,'r-')
plt.plot(X,Y,'bo')
plt.xlabel('Population Growth Rate (%)')
plt.ylabel('Elderly Population Rate (%)')
plt.show()
```



- 최적화 함수(optimizer) 손실을 최소화하는 딥러닝 알고리즘 > 복잡한 미분 계산 및 가중치 update를 자동으로 진행하는 도구

5. 딥러닝 네트워크를 사용한 회귀분석

4.3 딥러닝 네트워크를 이용한 회귀

+ 코드 + 텍스트

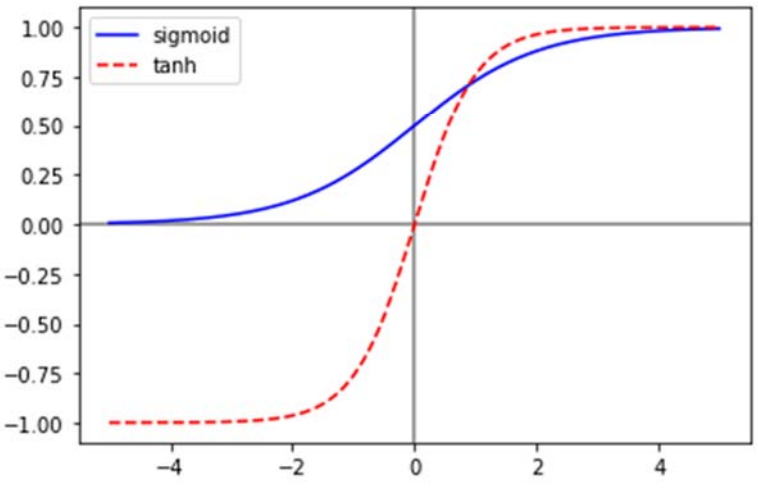
```
# 4.7 딥러닝 네트워크를 이용한 회귀
import tensorflow as tf
import numpy as np

X = [0.3, -0.78, 1.26, 0.03, 1.11, 0.24, -0.24, -0.47, -0.77, -0.37, -0.85, -0.41, -0.27, 0.02, -0.76, 2.66]
Y = [12.27, 14.44, 11.87, 18.75, 17.52, 16.37, 19.78, 19.51, 12.65, 14.74, 10.72, 21.94, 12.83, 15.51, 17.14, 14.42]

model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=6, activation='tanh', input_shape=(1,)),
    tf.keras.layers.Dense(units=1)
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.1), loss='mse')

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 6)                 12

 dense_1 (Dense)             (None, 1)                 7

=================================================================
Total params: 19
Trainable params: 19
Non-trainable params: 0
_________________________________________________________________
```

+ 활성화 함수: tanh(x)

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

```python
# 그림 4.2 출력 코드
import math
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

x = np.arange(-5, 5, 0.01)
sigmoid_x = [sigmoid(z) for z in x]
tanh_x = [math.tanh(z) for z in x]

plt.axhline(0, color='gray')
plt.axvline(0, color='gray')
plt.plot(x, sigmoid_x, 'b-', label='sigmoid')
plt.plot(x, tanh_x, 'r--', label='tanh')
plt.legend()
plt.show()
```



6. 분류 분석

- 정답이 있는 supervised learning로서 binary classification

```
# 5.1 와인 데이터셋 불러오기
import pandas as pd
red = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv', sep=';')
white = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv', sep=';')
print(red.head())
print(white.head())
```

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  ₩
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  ₩
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
```

- 데이터 전처리 작업

```
# 5.2 와인 데이터셋 합치기
red['type'] = 0
white['type'] = 1
print(red.head(2))
print(white.head(2))

wine = pd.concat([red, white])
print(wine.describe())
```

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  ₩
0            7.4              0.70          0.0             1.9      0.076
1            7.8              0.88          0.0             2.6      0.098

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  ₩
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68

   alcohol  quality  type
0      9.4        5     0
1      9.8        5     0
```

- 데이터 정규화

```
# 5.5 데이터 정규화
wine_norm = (wine - wine.min()) / (wine.max() - wine.min())
print(wine_norm.head())
print(wine_norm.describe())
```

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  ₩
0       0.297521          0.413333     0.000000        0.019939   0.111296
1       0.330579          0.533333     0.000000        0.030675   0.147841
2       0.330579          0.453333     0.024096        0.026074   0.137874
3       0.611570          0.133333     0.337349        0.019939   0.109635
4       0.297521          0.413333     0.000000        0.019939   0.111296

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  ₩
0             0.034722              0.064516  0.206092  0.612403   0.191011
1             0.083333              0.140553  0.186813  0.372093   0.258427
2             0.048611              0.110599  0.190669  0.418605   0.241573
3             0.055556              0.124424  0.209948  0.341085   0.202247
4             0.034722              0.064516  0.206092  0.612403   0.191011

    alcohol   quality  type
0  0.202899  0.333333   0.0
1  0.260870  0.333333   0.0
2  0.260870  0.333333   0.0
3  0.260870  0.500000   0.0
4  0.202899  0.333333   0.0
```

- 학습 적용을 위한 numpy array로 전환

```
# 5.6 데이터 섞은 후 numpy array로 변환
import numpy as np
wine_shuffle = wine_norm.sample(frac=1)
print(wine_shuffle.head())
wine_np = wine_shuffle.to_numpy()
print(wine_np[:5])
```

```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  ₩
536        0.504132          0.366667     0.144578        0.027607   0.112957
2715       0.330579          0.113333     0.168675        0.101227   0.051495
4837       0.223140          0.133333     0.228916        0.110429   0.036545
2045       0.404959          0.106667     0.210843        0.000000   0.054817
1569       0.198347          0.286667     0.084337        0.019939   0.078073

      free sulfur dioxide  total sulfur dioxide   density        pH  ₩
536              0.017361              0.062212  0.198381  0.286822
2715             0.156250              0.398618  0.160015  0.325581
4837             0.184028              0.483871  0.085406  0.240310
2045             0.034722              0.149770  0.105842  0.279070
1569             0.048611              0.064516  0.132061  0.589147

      sulphates   alcohol   quality  type
536    0.196629  0.202899  0.333333   0.0
2715   0.213483  0.304348  0.500000   1.0
4837   0.112360  0.739130  0.500000   1.0
2045   0.089888  0.376812  0.333333   1.0
1569   0.196629  0.507246  0.500000   0.0
```

- train data set와 test data set의 분리

```python
# 5.7 train 데이터와 test 데이터로 분리
import tensorflow as tf
train_idx = int(len(wine_np) * 0.8)
train_X, train_Y = wine_np[:train_idx, :-1], wine_np[:train_idx, -1]
test_X, test_Y = wine_np[train_idx:, :-1], wine_np[train_idx:, -1]
print(train_X[0])
print(train_Y[0])
print(test_X[0])
print(test_Y[0])
train_Y = tf.keras.utils.to_categorical(train_Y, num_classes=2)
test_Y = tf.keras.utils.to_categorical(test_Y, num_classes=2)
print(train_Y[0])
print(test_Y[0])
```

```
[0.50413223 0.36666667 0.14457831 0.02760736 0.11295681 0.01736111
 0.06221198 0.19838057 0.28682171 0.19662921 0.20289855 0.33333333]
0.0
[0.3553719  0.11333333 0.20481928 0.14570552 0.06810631 0.10069444
 0.26497696 0.19529593 0.34883721 0.15168539 0.30434783 0.5       ]
1.0
[1. 0.]
[0. 1.]
```

- one-hot encoding: 분류 문제에서 해당 정답은 1, 오답은 0으로

  + 이항분류는 [0,1] 또는 [1,0]으로 분류

  + to_categorical() 함수는 one-hot encoding으로 변환

- 분류 모델

```python
# 5.8 와인 데이터셋 분류 모델 생성
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=48, activation='relu', input_shape=(12,)),
    tf.keras.layers.Dense(units=24, activation='relu'),
    tf.keras.layers.Dense(units=12, activation='relu'),
    tf.keras.layers.Dense(units=2, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.07), loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 48)                624

 dense_1 (Dense)             (None, 24)                1176

 dense_2 (Dense)             (None, 12)                300

 dense_3 (Dense)             (None, 2)                 26

=================================================================
Total params: 2,126
Trainable params: 2,126
Non-trainable params: 0
```

- 활성화 함수: softmax 사용

  + 출력 값들을 자연로그 e를 취하여 모두 더한 값으로 나눈 값

$$\mathrm{P}(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} (for\ j = 1, 2, \cdots, \mathrm{K})$$

  + 예: 출력값: [2,1,0]인 경우

$$\mathrm{sum} = \sum_{k=1}^{K} e^{z_k} = e^2 + e^1 + e^0 = 11.1073$$

$$\mathrm{softmax} = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} = \left[\frac{e^2}{sum}, \frac{e^1}{sum}, \frac{e^0}{sum}\right] = [0.67, 0.24, 0.09]$$

  + max 함수를 약화시키는 활성화 함수

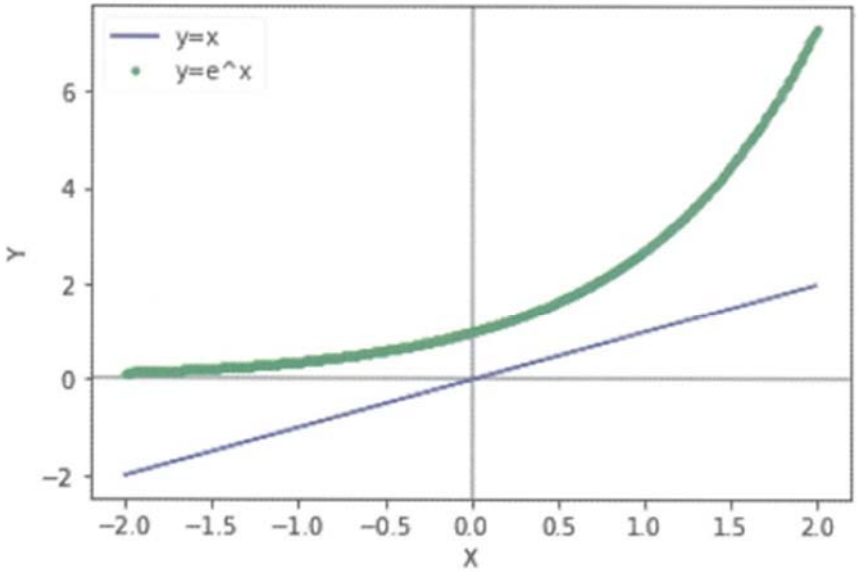
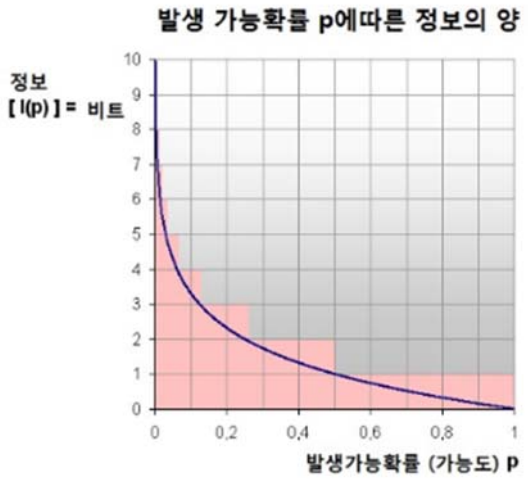


그림 5.5 지수 함수는 큰 값을 강조하고 작은 값은 약화하는 효과가 있습니다.

- optimizer에서 손실 함수로 mse 대신에 categorical_crossentropy 사용

+ 정보이론



+ 확률이 낮으면 발생하는 정보량이 많다 > 이미 정보를 많이 알수록 확률은 높고 새롭게 알 수 있는 정보양은 감소한다

+ entropy: 어떤 상태에서의 불확실성 > 예측하기가 어려울수록 정보량이 많아지고, 엔트로피도 커진다

$$h(x) = \log\frac{1}{p(x)} = -\log p(x)$$

+ 확률의 역수에 로그 값 >확률이 크면 정보양(놀라움)이 적어진다

+ 비가 올 확률 1%, 오지 않을 확률 99%

$$h(\text{비}) = -\log 0.01 = 4.605$$

$$h(\text{비가 오지 않음}) = -\log 0.99 = 0.010$$

> 비가 오면 460배 정도 더 놀라운 정보량이 발생한다

+ entropy 기대 값: 각 엔트로피에 확률을 곱하기

$$\mathrm{E}(\mathrm{X}) = -p(x)\log p(x)$$

>> 엔트로피 기대값을 낮추려고 피드백

- categorical_crossentropy

+ categorical: 범주 분류

+ crossentropy: 엔트로피 기대값과 유사하나 다음과 같이 차이

+ p(x)는 정답에 대한 확률, q(x)는 분류 네트워크가 계산한 정답 확률

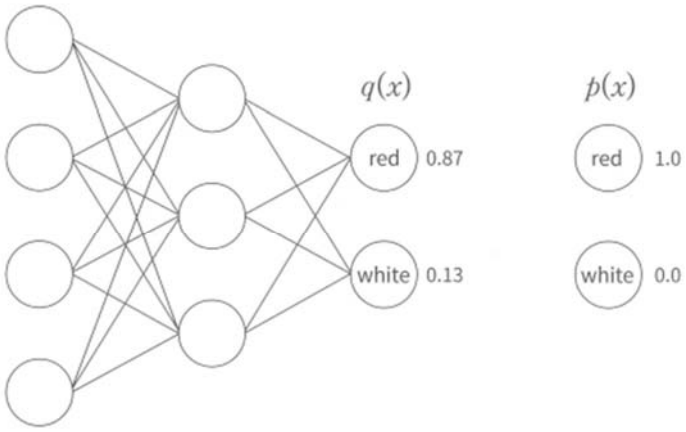


그림 5.6 $p(x)$는 정답 라벨, $q(x)$는 분류 네트워크의 계산 결과인 예측 라벨입니다.

>

$$CCE = -\frac{1}{n}\sum_{j=1}^{n} p(x)\log q(x)$$

>

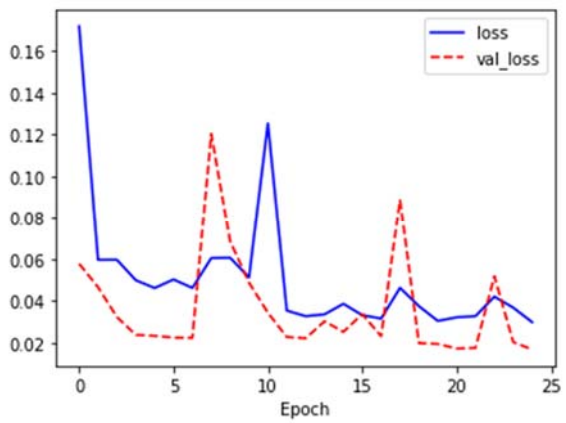
> 정답이 red인데 red일 확률 0.87, 화이트일 확률 0.13이면 CCE1이 0.06으로 엔트로피가 낮다

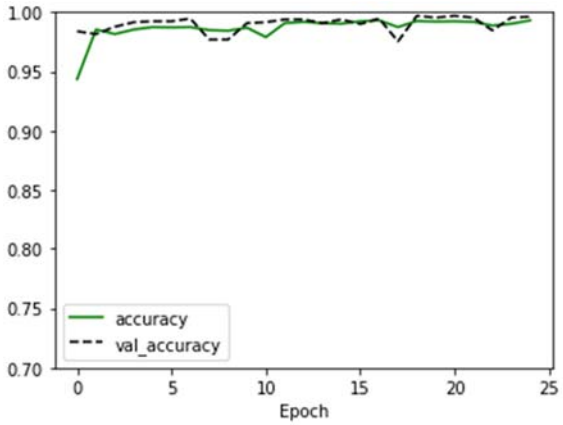
$$CCE_1 = -\frac{1}{2}(1.0 \times \log 0.87 + 0.0 \times \log 0.13) = 0.0696$$

> 정답이 white이고 레드일 확률이 0.6, 화이트일 확률이 0.4이면 CCE2는 0.458로 엔트로피가 높다

$$CCE_2 = -\frac{1}{2}(0.0 \times \log 0.6 + 1.0 \times \log 0.4) = 0.4581$$

```
# 5.9 와인 데이터셋 분류 모델 학습
history = model.fit(train_X, train_Y, epochs=25, batch_size=32, validation_split=0.25)

Epoch 1/25
122/122 [==============================] - 1s 4ms/step - loss: 0.1082 - accuracy: 0.9633 - val_loss: 0.0464 - val_accuracy: 0.9846
Epoch 2/25
122/122 [==============================] - 0s 3ms/step - loss: 0.0666 - accuracy: 0.9805 - val_loss: 0.0301 - val_accuracy: 0.9908
Epoch 3/25
122/122 [==============================] - 0s 3ms/step - loss: 0.0559 - accuracy: 0.9861 - val_loss: 0.0335 - val_accuracy: 0.9900
Epoch 4/25
122/122 [==============================] - 0s 3ms/step - loss: 0.0524 - accuracy: 0.9861 - val_loss: 0.0649 - val_accuracy: 0.9800
Epoch 5/25
122/122 [==============================] - 0s 3ms/step - loss: 0.1125 - accuracy: 0.9707 - val_loss: 0.0378 - val_accuracy: 0.9923
Epoch 6/25
122/122 [==============================] - 0s 3ms/step - loss: 0.0563 - accuracy: 0.9838 - val_loss: 0.0269 - val_accuracy: 0.9931
Epoch 7/25
122/122 [==============================] - 0s 2ms/step - loss: 0.0453 - accuracy: 0.9861 - val_loss: 0.0349 - val_accuracy: 0.9900
```





0초

```
# 5.11 분류 모델 평가
model.evaluate(test_X, test_Y)
```

```
41/41 [==============================] - 0s 1ms/step - loss: 0.0248 - accuracy: 0.9954
[0.024795057252049446, 0.9953846335411072]
```