# 딥 러닝을 이용한 자연어 처리 입문

6장 머신 러닝

발표자 : 김성윤

# 목차

# 1. 머신 러닝 – 개념

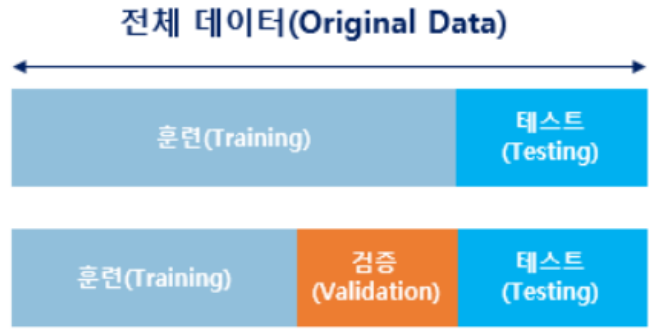
- 기존 방식의 한계

- 다양한 사진에서 공통된 명확한 특징을 잡아내는 것이 쉽지 않음 (=절대적 기준이 존재하지 않음)


⇒ 기존 프로그래밍의 접근 방식
⇒ 직접 규칙 정의


⇒ 머신러닝의 접근 방식
⇒ 기계가 데이터로부터 스스로 규칙을 찾아냄

# 1. 머신 러닝 – 용어 정리




- 검증데이터의 용도

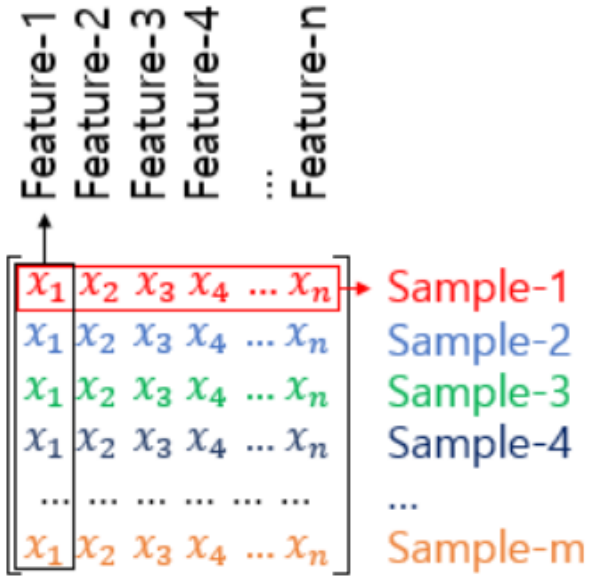
- 모델 성능 평가(X), **모델 성능 조정**(O)
- 과적합 판단 & *하이퍼파라미터 조정(=하이퍼파라미터 튜닝)
- 수능에서 모의고사 역할

- 샘플

- 행렬 관점에서는 하나의 행
- 데이터베이스 과점에서 '레코드'

- 특성

- 종속 변수 y를 예측하기 위한 각각의 독립 변수 x




*하이퍼파라미터 : 모델의 성능에 영향을 주는 사람이 값을 지정하는 변수.
**매개변수 : 가중치와 편향. 학습을 하는 동안 값이 계속해서 변하는 수.

# 1. 머신 러닝 – 학습 방법

- 지도 학습
  - 레이블이라는 **정답과 함께 학습**
  - 대부분의 자연어처리 학습 방식

- 비지도 학습
  - 데이터에 별도의 **레이블이 없이 학습**
  - LSA, LDA

- 자기지도 학습
  - 데이터가 주어지면, 모델이 학습을 위해서 **스스로 데이터로부터 레이블을 만들어서 학습**
  - Word2Bec, BERT

# 1. 머신 러닝 – 성능 평가 기준

- 혼동 행렬

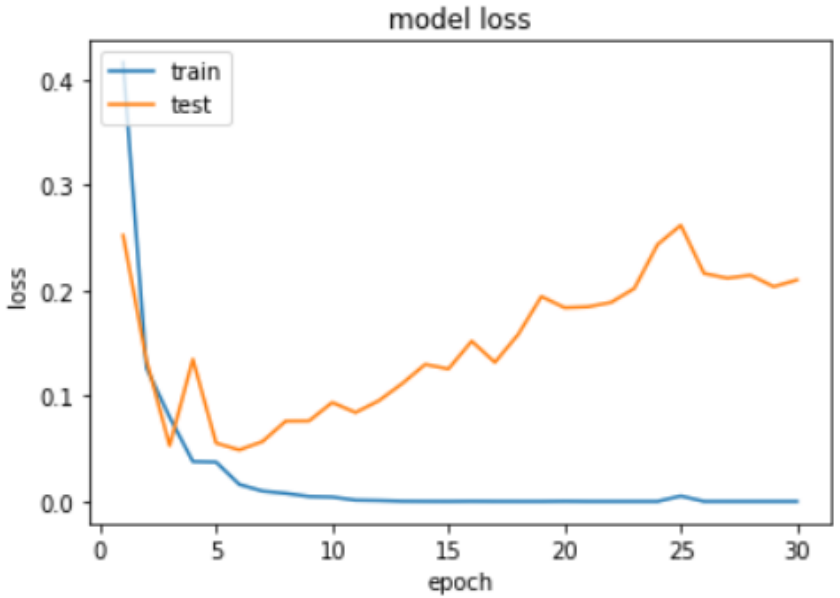
$$\text{정밀도} = \frac{TP}{TP + FP}$$

$$\text{재현율} = \frac{TP}{TP + FN}$$

$$\text{정확도} = \frac{TP + TN}{TP + FN + FP + TN}$$

| | **예측 참** | **예측 거짓** |
|---|---|---|
| 실제 참 | TP | FN |
| 실제 거짓 | FP | TN |

# 1. 머신 러닝 – 학습 상태




- 과적합(Overfitting)
  - 훈련 데이터를 **과하게 학습**한 경우
  - 훈련 데이터 오차 감소, 테스트 데이터 오차 증가

- 과소 적합(Underfitting)
  - 테스트 데이터의 성능이 올라갈 여지가 있음에도 **훈련을 덜 한 상태**
  - 훈련 횟수인 에포크가 지나치게 적으면 발생
  - 훈련 데이터, 테스트 데이터 모두 오차 높음.

# 1. 머신 러닝 – 문제 유형

- 회귀

- 회귀 문제 : 어떠한 연속적인 값의 범위 내에서 예측값이 나오는 경우

- 분류

- 이진 분류 문제 : 두 개의 선택지 중 하나의 답을 선택
- 다중 클래스 분류 문제 : 세 개 이상의 선택지 중에서 답을 선택

# 2. 회귀 – 회귀 문제

- 선형 회귀

- 한 개 이상의 독립 변수 x와 종속변수 y의 선형 관계

1) 단순 선형 회귀 : 독립 변수 x가 1개

$$y = wx + b$$

2) 다중 선형 회귀 : 독립 변수 x가 1개 이상

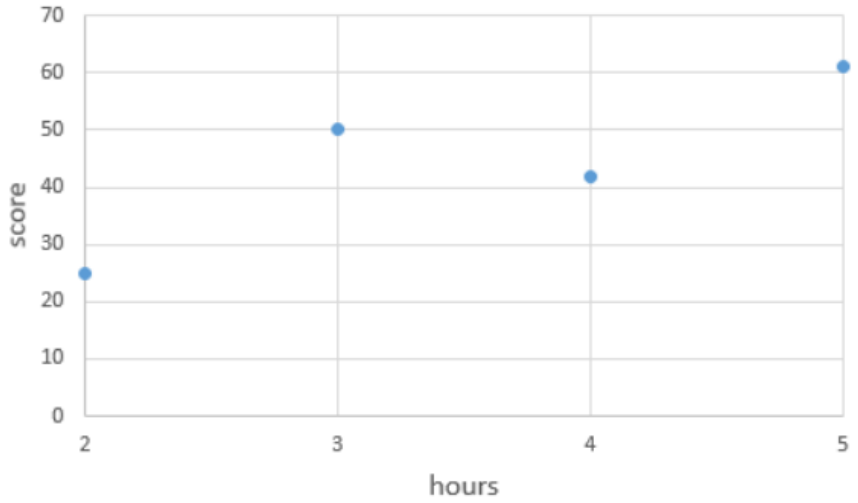
$$y = w_1x_1 + w_2x_2 + \ldots w_nx_n + b$$

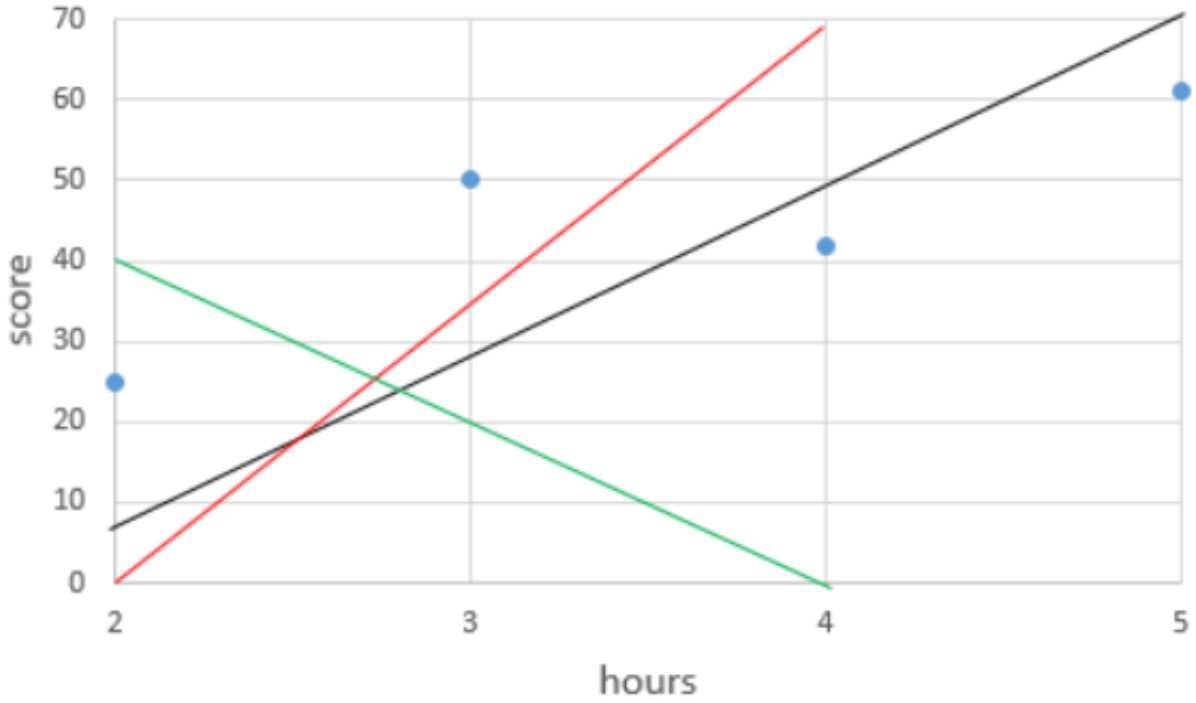
*ω : 가중치
**b : 편향

# 2. 회귀 – 회귀 문제

$$H(x) = wx + b$$

| hours($x$) | score($y$) |
|---|---|
| 2 | 25 |
| 3 | 50 |
| 4 | 42 |
| 5 | 61 |




ω와 b에 따라 수많은 직선을 그릴 수 있음
⇒ 문제에 대한 규칙을 가장 잘 표현하는 ω와 b를 찾아야함

*H(x) : 가설
**ω : 가중치
***b : 편향

# 2. 회귀 – 목적 함수, 비용 함수, 손실 함수

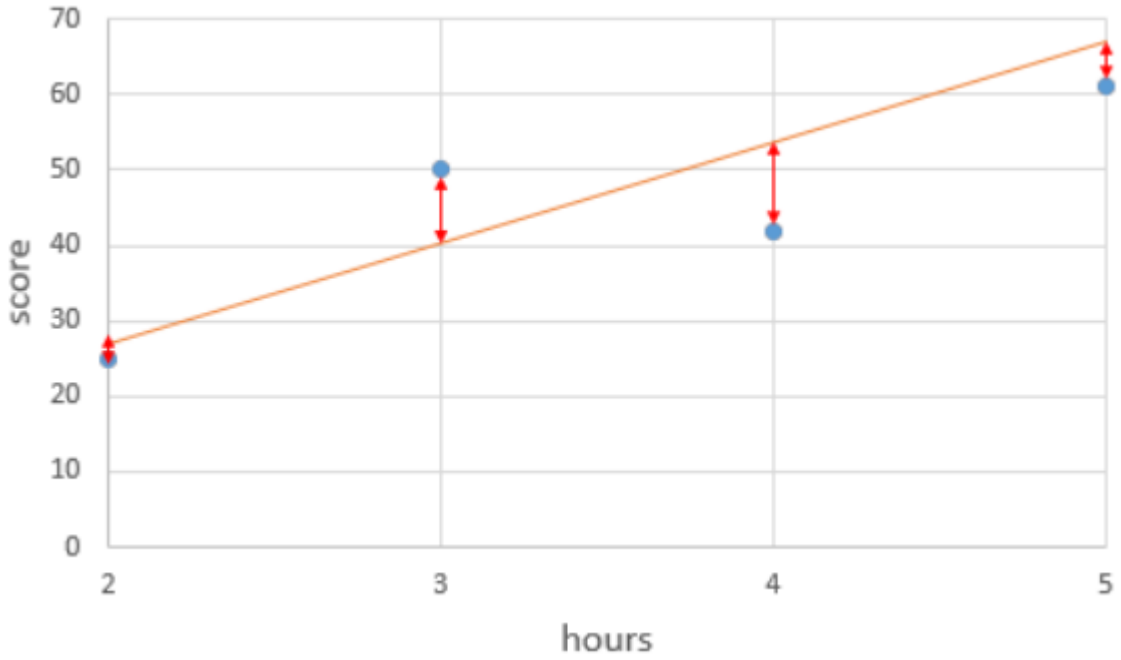
- 실제값과 예측값에 대한 **오차**에 대한 식

- 목적 함수
  - 함수의 값을 최소화하거나, 최대화하거나 하는 목적을 가진 함수

- 비용 함수 & 손실 함수
  - 함수의 값을 최소화하려는 함수

# 2. 회귀 – 비용 함수

- 실제값과 예측값에 대한 오차를 표현 + 예측값의 오차를 줄이는 일에 최적화 된 식
- 각 문제들에는 적합한 비용 함수들이 존재
- 회귀의 대표적 비용 함수 예) 평균 제곱 오차(Mean Squard Error, MSE)
- ω와 b를 바꿔가며 전체 오차가 가장 작아지도록 함.



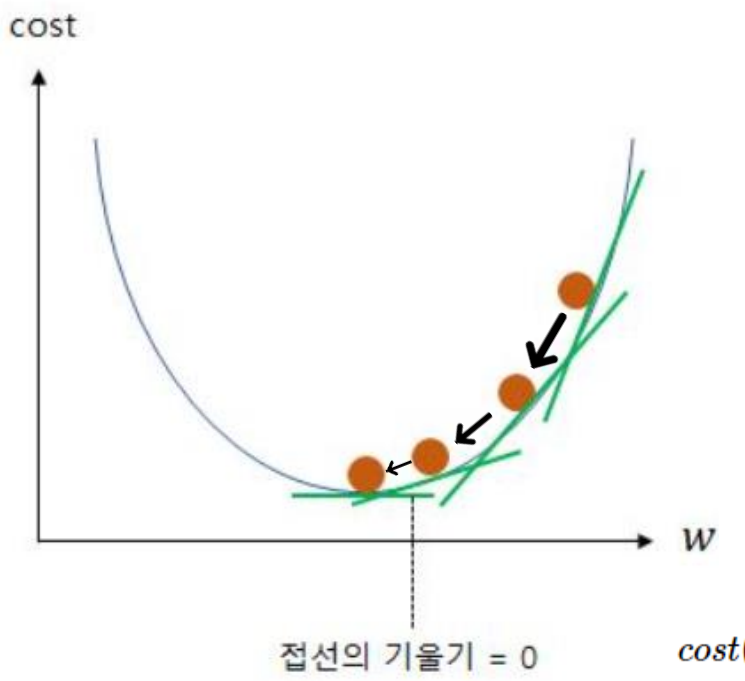

전체 오차를 구하는 방법 : $\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)} - H(x^{(i)})\right]^2$

(절대적 크기를 구하기 위해 제곱)

↕ : 각 점에서의 오차의 크기

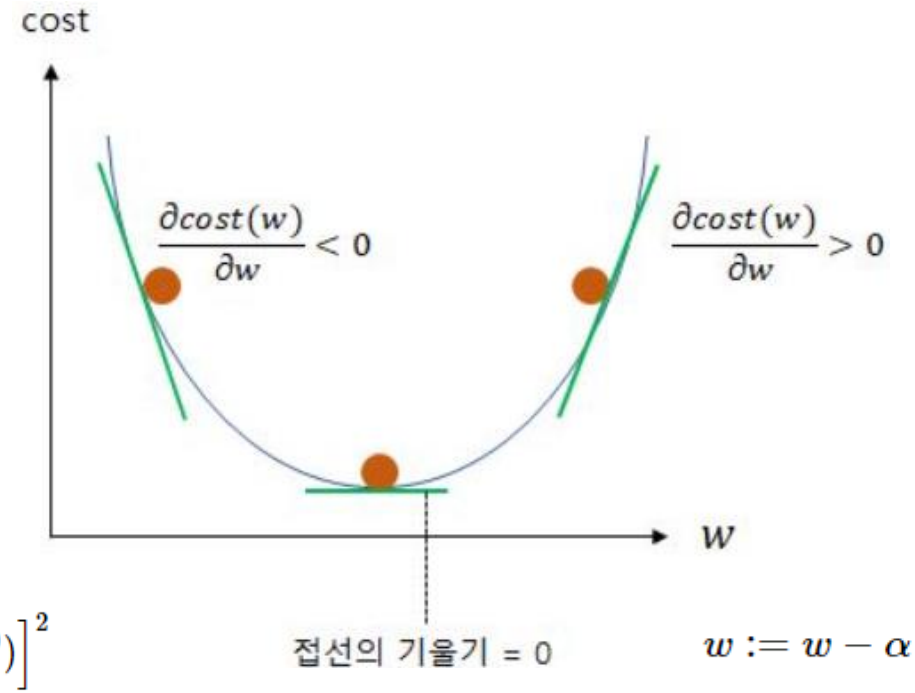
# 2. 회귀 – 옵티마이저

- 최적의 ω와 b를 구하기 위한 알고리즘
- 예) 경사 하강법




$$cost(w,b) = \frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)} - H(x^{(i)})\right]^2$$

W : 가중치




$$w := w - \alpha\frac{\partial}{\partial w}cost(w)$$

α : 학습률 (ω의 변경 정도)

# 2. 회귀 – 다중 선형 회귀

- y를 결정하는 독립변수(x)의 개수가 2개 이상인 회귀 문제

$$H(X) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

```
# 중간 고사, 기말 고사, 가산점 점수
X = np.array([[70,85,11], [71,89,18], [50,80,20], [99,20,10], [50,10,10]])
y = np.array([73, 82 ,72, 57, 34]) # 최종 성적

model = Sequential()
model.add(Dense(1, input_dim=3, activation='linear'))

sgd = optimizers.SGD(lr=0.0001)
model.compile(optimizer=sgd, loss='mse', metrics=['mse'])
model.fit(X, y, epochs=2000)
```

```
print(model.predict(X))
```

```
[[73.15294 ]
 [81.98001 ]
 [71.93192 ]
 [57.161617]
 [33.669353]]
```

입력 데이터에 대한 예측

# 2. 회귀 – 자동 미분 활용

- 텐서플로우 (Tensorflow) 이용

```
@tf.function
def hypothesis(x):
  return w*x + b
```

```
@tf.function
def mse_loss(y_pred, y):
  # 두 개의 차이값을 제곱을 해서 평균을 취한다.
  return tf.reduce_mean(tf.square(y_pred - y))
```

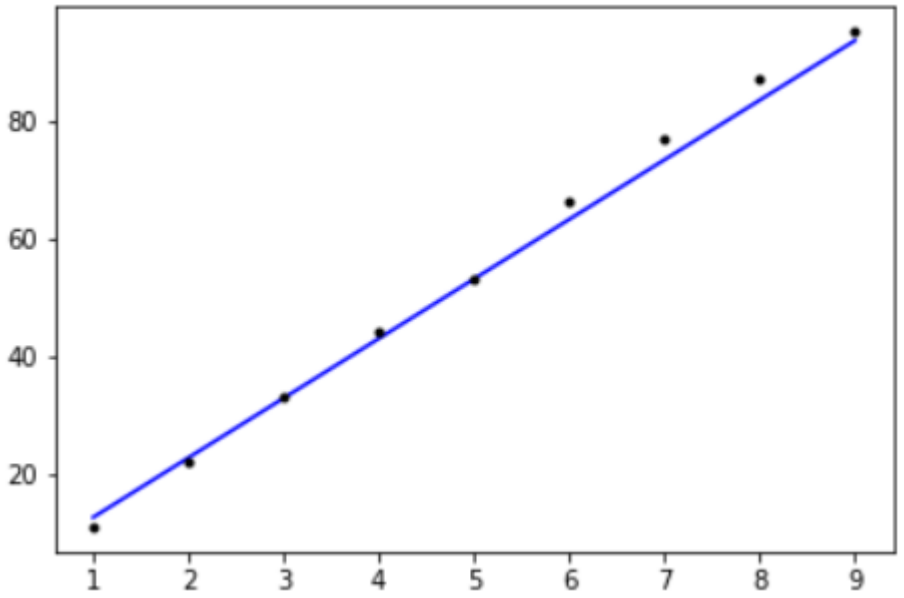
```
optimizer = tf.optimizers.SGD(0.01)
```

```
for i in range(301):
  with tf.GradientTape() as tape:
    # 현재 파라미터에 기반한 입력 x에 대한 예측값을 y_pred
    y_pred = hypothesis(x)

    # 평균 제곱 오차를 계산
    cost = mse_loss(y_pred, y)

  # 손실 함수에 대한 파라미터의 미분값 계산
  gradients = tape.gradient(cost, [w, b])

  # 파라미터 업데이트
  optimizer.apply_gradients(zip(gradients, [w, b]))

  if i % 10 == 0:
    print("epoch : {:3} | w의 값 : {:5.4f} | b의 값 : {:5.4} | cost : {:5.6f}".format(i, w.numpy(), b.numpy(), cost))
```

```
epoch :   0 | w의 값 : 8.2133 | b의 값 : 1.664 | cost : 1402.555542
... 중략 ...
epoch : 280 | w의 값 : 10.6221 | b의 값 : 1.191 | cost : 1.091434
epoch : 290 | w의 값 : 10.6245 | b의 값 : 1.176 | cost : 1.088940
epoch : 300 | w의 값 : 10.6269 | b의 값 : 1.161 | cost : 1.086645
```

# 2. 회귀 – 자동 미분 활용

- 케라스 (Keras) 이용

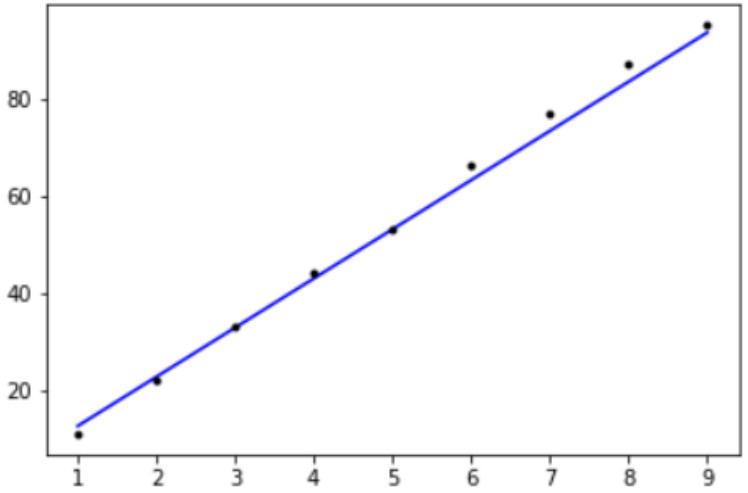



```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import optimizers

x = [1, 2, 3, 4, 5, 6, 7, 8, 9] # 공부하는 시간
y = [11, 22, 33, 44, 53, 66, 77, 87, 95] # 각 공부하는 시간에 맵핑되는 성적

model = Sequential()

# 출력 y의 차원은 1. 입력 x의 차원(input_dim)은 1
# 선형 회귀이므로 activation은 'linear'
model.add(Dense(1, input_dim=1, activation='linear'))

# sgd는 경사 하강법을 의미. 학습률(learning rate, lr)은 0.01.
sgd = optimizers.SGD(lr=0.01)

# 손실 함수(Loss function)은 평균제곱오차 mse를 사용합니다.
model.compile(optimizer=sgd, loss='mse', metrics=['mse'])

# 주어진 x와 y데이터에 대해서 오차를 최소화하는 작업을 300번 시도합니다.
model.fit(x, y, epochs=300)
```

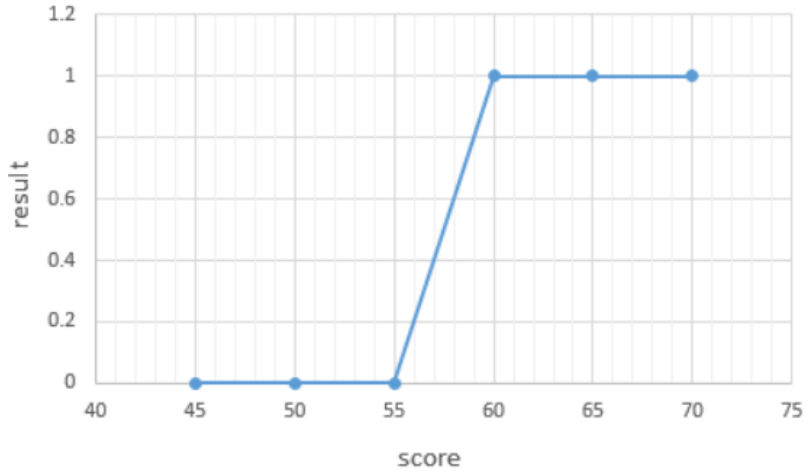
# 3. 분류 – 이진 분류

- 직선 함수를 이용했을 때의 문제점

1) 0과 1 사이의 값만을 가지지 않음
2) y=ωx+b의 직선을 사용할 경우, y값이 (-∞,∞)의 범위를 가짐



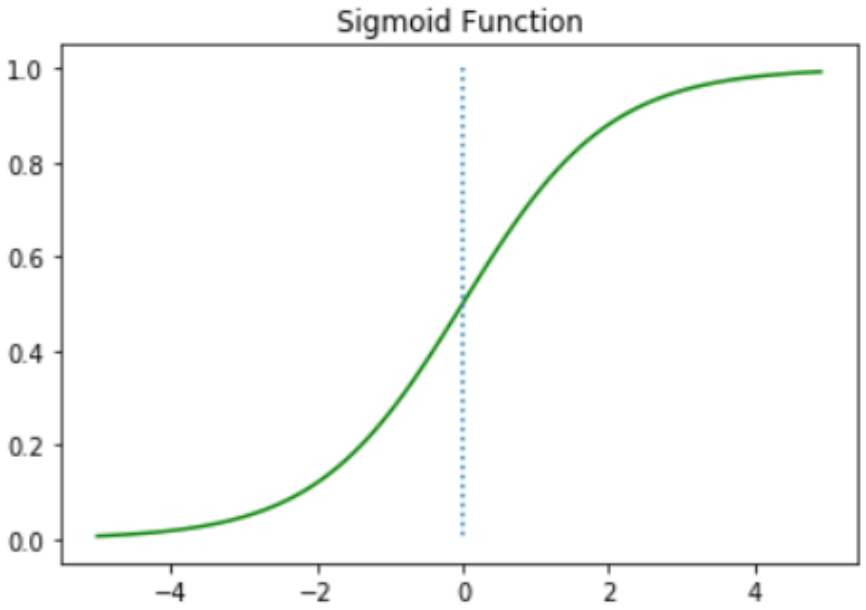

직선 함수




분류 데이터의 좌표끼리 연결한 함수

# 3. 분류 – 이진 분류

- S자 형태의 함수가 필요
- 예) 시그모이드 함수



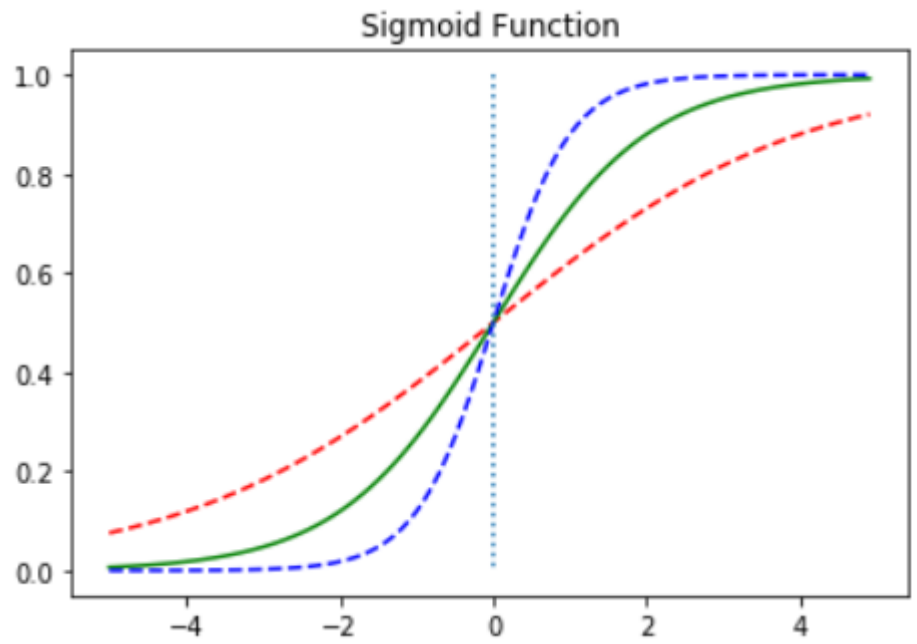

$$H(x) = \frac{1}{1+e^{-(wx+b)}} = sigmoid(wx+b) = \sigma(wx+b)$$

```
def sigmoid(x):
    return 1/(1+np.exp(-x))

x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)

plt.plot(x, y, 'g')
plt.plot([0,0],[1.0,0.0], ':') # 가운데 점선 추가
plt.title('Sigmoid Function')
plt.show()
```

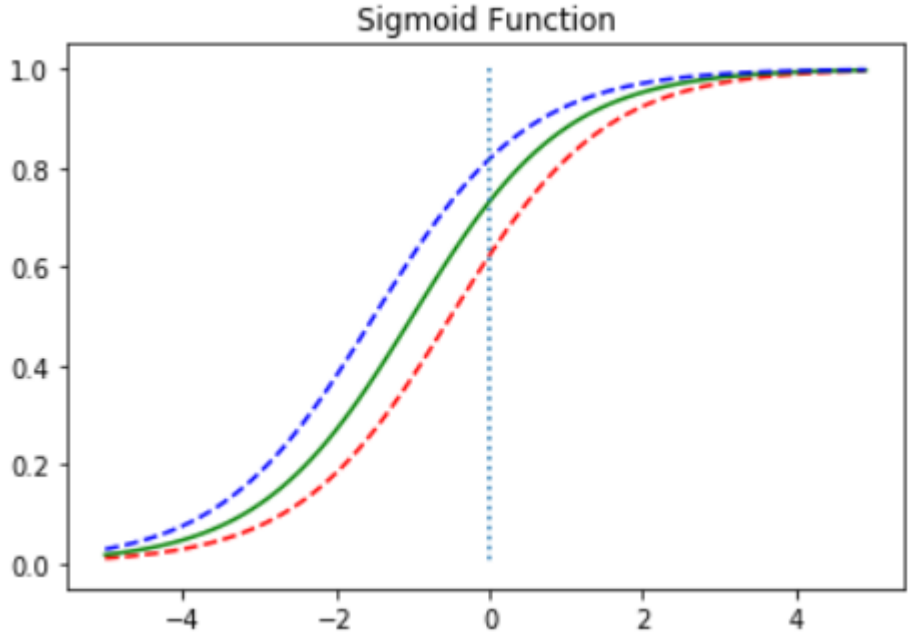
# 3. 분류 – 시그모이드 함수

- ω와 b값에 따라 그래프 개형 결정

$$H(x) = \frac{1}{1 + e^{-(wx+b)}} = sigmoid(wx + b) = \sigma(wx + b)$$

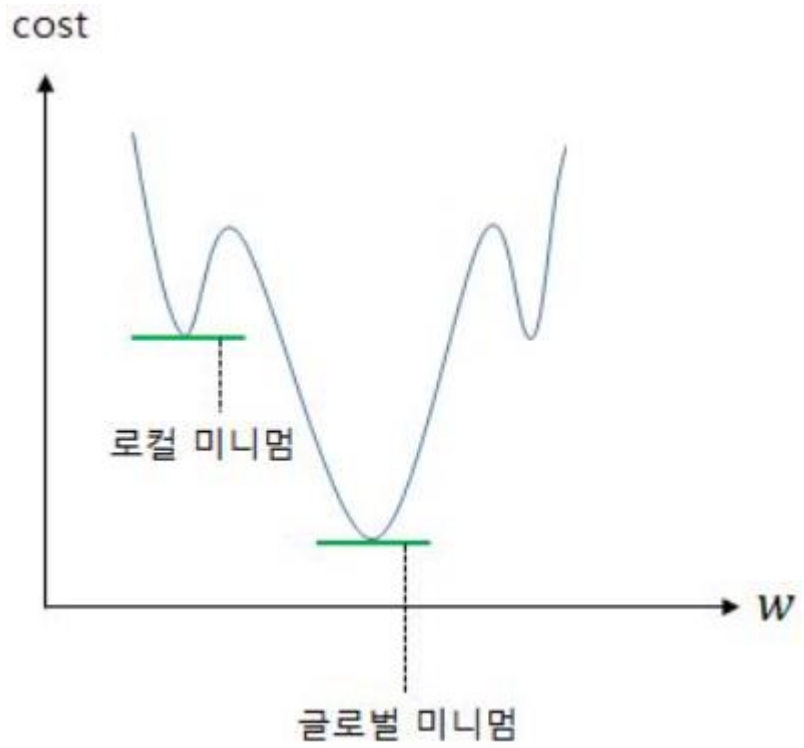



Blue : ω값을 증가, Green : 기본, Red : ω값을 감소

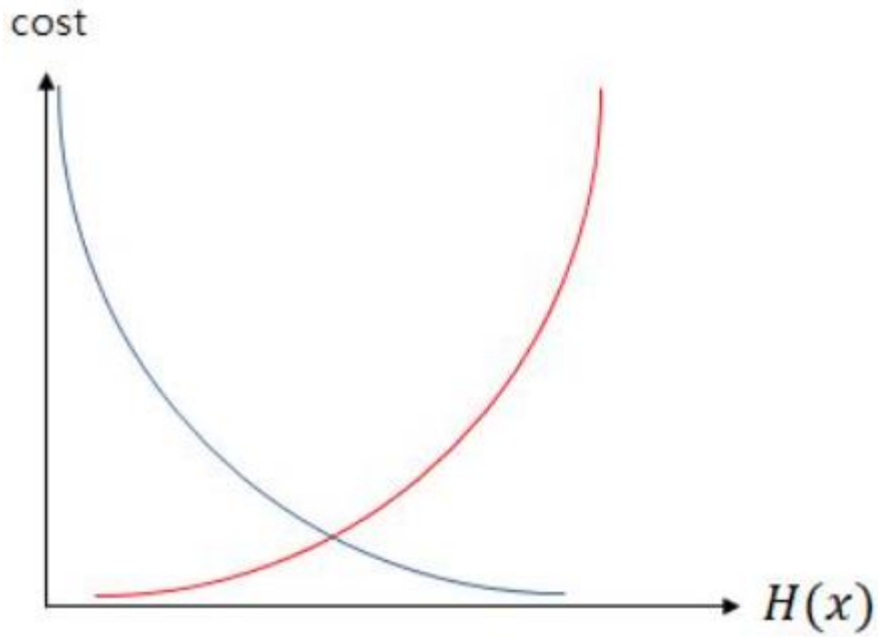



Blue : b값을 증가, Green : 기본, Red : b값을 감소

# 3. 분류 – 비용 함수




경사 하강법 이용시 로컬 미니엄에 빠질 가능성이 존재



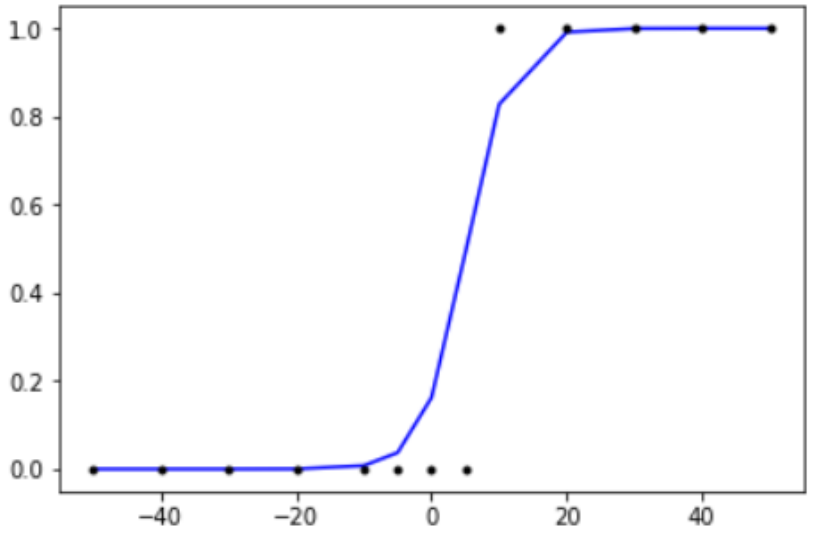

$$J(w) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}logH(x^{(i)}) + (1-y^{(i)})log(1-H(x^{(i)}))]$$

실제값(y)이 0일 때, H(x)의 값이 1에 가까워지면 오차 증가

실제값(y)이 1일 때, H(x)의 값이 0에 가까워지면 오차 증가

# 3. 분류 – 이진 분류 코드 확인

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import optimizers

x = np.array([-50, -40, -30, -20, -10, -5, 0, 5, 10, 20, 30, 40, 50])
y = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]) # 숫자 10부터 1

model = Sequential()
model.add(Dense(1, input_dim=1, activation='sigmoid'))

sgd = optimizers.SGD(lr=0.01)
model.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['binary_accuracy'])

model.fit(x, y, epochs=200)
```




epoch가 190일 때부터 정확도가 100%인 그래프

# 3. 분류 – 다중 로지스틱 회귀

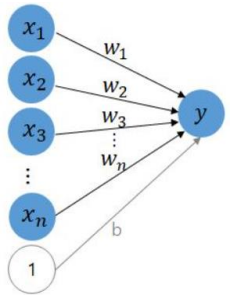
- y를 결정하는 독립변수(x)가 2개 이상인 분류 문제

$$y = sigmoid(w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n + b) = \sigma(w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n + b)$$

```
X = np.array([[0, 0], [0, 1], [1, 0], [0, 2], [1, 1], [2, 0]])
y = np.array([0, 0, 0, 1, 1, 1])

model = Sequential()
model.add(Dense(1, input_dim=2, activation='sigmoid'))
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['binary_accuracy'])

model.fit(X, y, epochs=2000)
```

```
print(model.predict(X))
```

```
[[0.23379876]
 [0.48773268]
 [0.4808667 ]
 [0.7481605 ]
 [0.74294543]
 [0.7376603 ]]
```



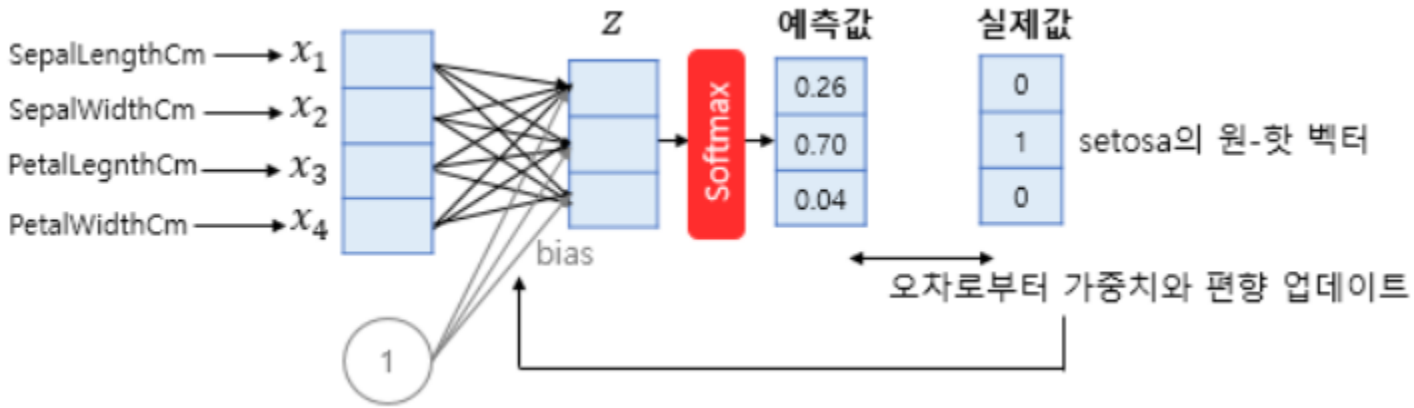

: 인공신경망으로 해석해도 됨

# 3. 분류 – 다중 클래스 분류

- 3개 이상의 선택지 중 하나를 고르는 문제
- 예) 소프트 맥스 함수 : 합이 1인 모든 선택지에 대한 확률을 구하고 가장 높은 확률을 선택

$$softmax(z) = [\frac{e^{z_1}}{\sum_{j=1}^{3} e^{z_j}} \quad \frac{e^{z_2}}{\sum_{j=1}^{3} e^{z_j}} \quad \frac{e^{z_3}}{\sum_{j=1}^{3} e^{z_j}}] = [p_1, p_2, p_3] = \hat{y} = \text{예측값}$$

(입력값 z = [z1, z2, z3])



# 3. 분류 – 다중 클래스 분류

- 실제값을 원-핫 인코딩해야 하는 이유

```
{Banana :1, Tomato :2, Apple :3, Strawberry :4, ... Watermelon :10}
```

$(2-1)^2 = 1$

$(3-1)^2 = 4$

⇒ 원-핫 인코딩이 아닌 경우, 클래스 간의 제곱 오차가 불균등

$((1,0,0)-(0,1,0))^2 = (1-0)^2 + (0-1)^2 + (0-0)^2 = 2$

$((1,0,0)-(0,0,1))^2 = (1-0)^2 + (0-0)^2 + (0-1)^2 = 2$

⇒ 원-핫 인코딩을 할 경우, 클래스 간의 제곱 오차가 균등

# 3. 분류 – 다중 클래스 분류

- 비용 함수 : 크로스 엔트로피 함수

$$cost = -\sum_{j=1}^{k} y_j \, log(p_j)$$ ⇒ 정확히 예측해서 p=1이 된다면 비용함수는 0이 되어 최고가 됨

일반화된 비용 함수 식

$$cost = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_j^{(i)} \, log(p_j^{(i)})$$

y : 실제값
k : 클래스의 개수
p : 해당 클래스일 확률
n : 전체 데이터 개수

<참고사항>
If k=2,
then 로지스틱 회귀의 비용 함수와 일치

$$cost = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_j^{(i)} \, log(p_j^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}log(p^{(i)}) + (1-y^{(i)})log(1-p^{(i)})]$$

# 3. 분류 – 다중 클래스 분류

- 아이리스(iris) 데이터를 활용한 예시

1) 전처리 단계 - Target 확인

```
# 중복을 허용하지 않고, 있는 데이터의 모든 종류를 출력
print("품종 종류:", data["Species"].unique(), sep="\n")
```

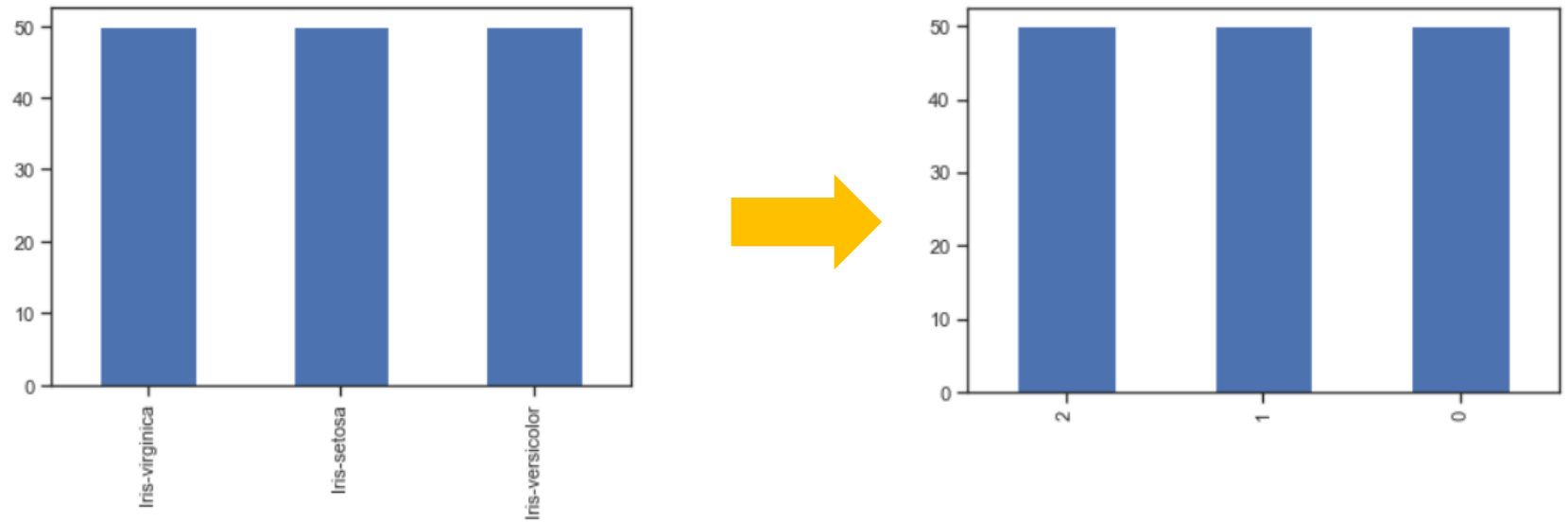
```
품종 종류:
['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
```

# 3. 분류 – 다중 클래스 분류

- 아이리스(iris) 데이터를 활용한 예시

1) 전처리 단계 – 정수 인코딩

```
# Iris-virginica는 0, Iris-setosa는 1, Iris-versicolor는 2가 됨.
data['Species'] = data['Species'].replace(['Iris-virginica','Iris-setosa','Iris-versicolor'],[0,1,2])
data['Species'].value_counts().plot(kind='bar')
```



# 3. 분류 – 다중 클래스 분류

- 아이리스(iris) 데이터를 활용한 예시

1) 전처리 단계 – 학습 데이터 확인 및 분리

```
# X 데이터. 특성은 총 4개.
data_X = data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values

# Y 데이터. 예측 대상.
data_y = data['Species'].values
```

```
print(data_X[:5])
print(data_y[:5])
```

```
# 훈련 데이터와 테스트 데이터를 8:2로 나눈다.
(X_train, X_test, y_train, y_test) = train_test_split(data_X, data_y, train_size=0.8, random_state=1)

# 원-핫 인코딩
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
[1 1 1 1 1]
```

# 3. 분류 – 다중 클래스 분류

- 아이리스(iris) 데이터를 활용한 예시

2) 소프트맥스 회귀 – 모델 제작 및 학습

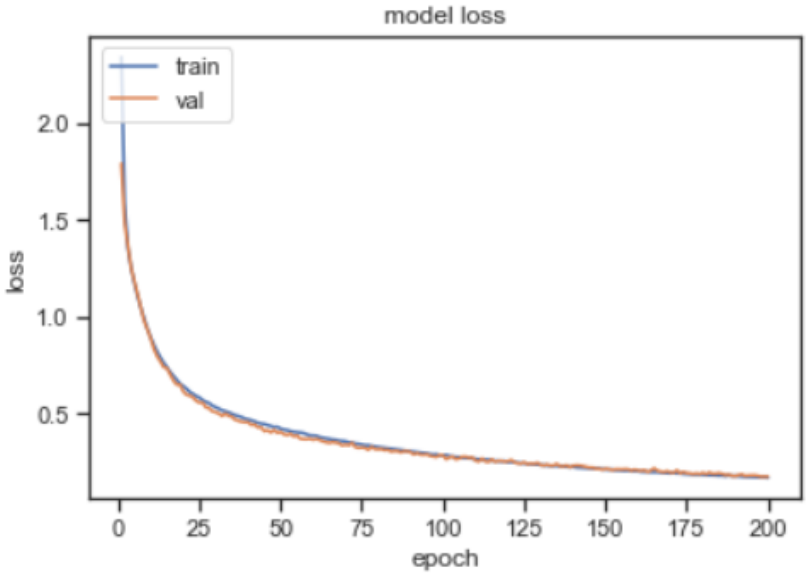
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(3, input_dim=4, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=200, batch_size=1, validation_data=(X_test, y_test))
```

# 3. 분류 – 다중 클래스 분류

- 아이리스(iris) 데이터를 활용한 예시

2) 소프트맥스 회귀 – 시각화 및 정확도

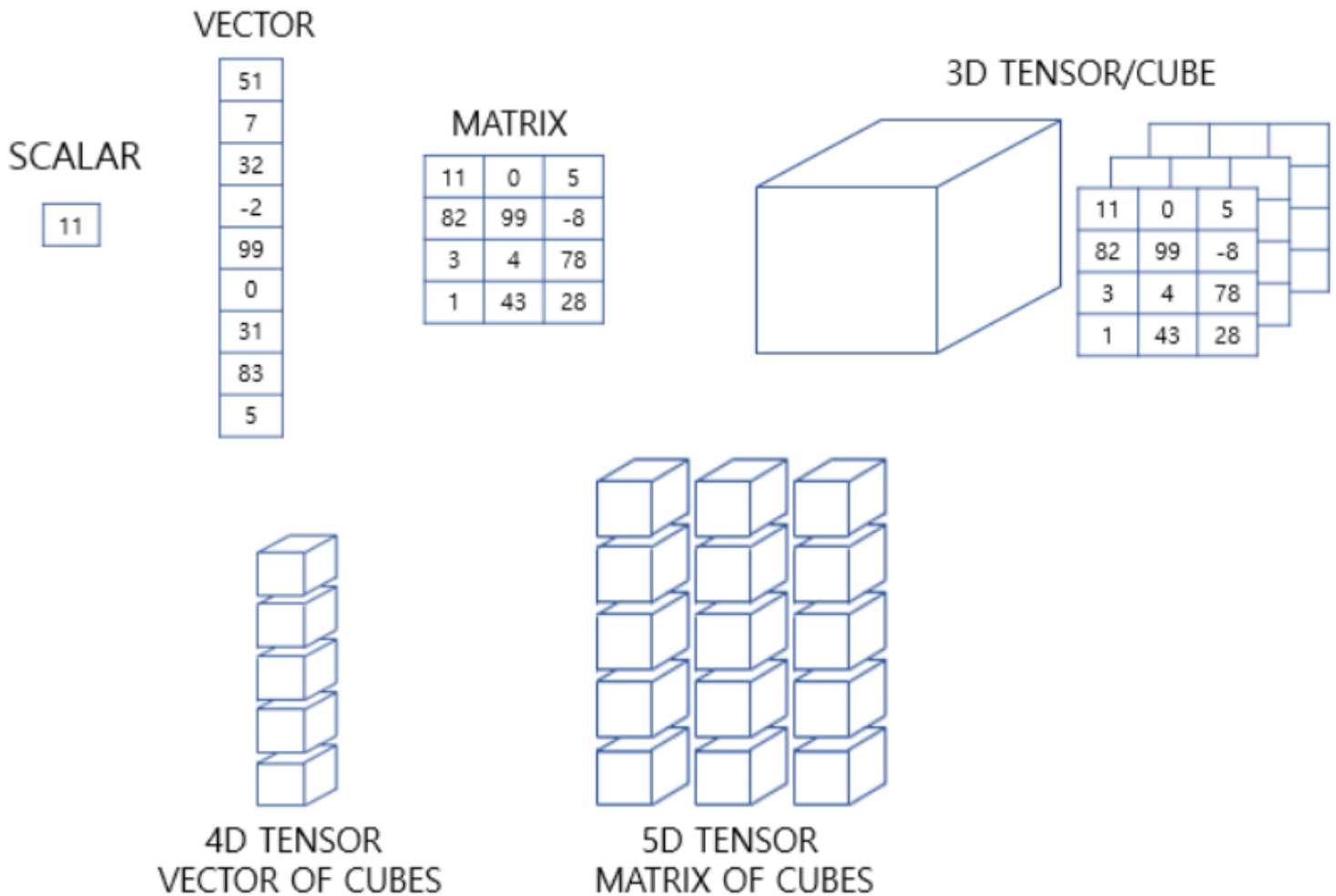
```
epochs = range(1, len(history.history['accuracy']) + 1)
plt.plot(epochs, history.history['loss'])
plt.plot(epochs, history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```




```
print("\n 테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
```

```
테스트 정확도: 0.9667
```

# 4. 벡터와 행렬 연산 – 텐서


SCALAR
11
VECTOR
51
7
32
-2
99
0
31
83
5
MATRIX
11 0 5
82 99 -8
3 4 78
1 43 28
3D TENSOR/CUBE
11 0 5
82 99 -8
3 4 78
1 43 28
4D TENSOR
VECTOR OF CUBES
5D TENSOR
MATRIX OF CUBES

# 4. 벡터와 행렬 연산 – 케라스에서 텐서

- 신경망의 층에 입력의 크기(shape)를 인자로 줄 때 input_shape라는 인자를 사용
  예1) input_shape(6, 5)라는 인자값을 사용한다면 이 텐서의 크기는 (?, 6, 5)을 의미
  예2) 배치 크기까지 지정하고 싶다면, batch_input_shape=(8, 2, 10)와 같이 인자를 주면 이 텐서의 크기는 (8, 2, 10)을 의미

*배치(batch) : 훈련 데이터를 다수 묶어 입력으로 사용하는 것

# 4. 벡터와 행렬 연산 – 행렬의 덧셈과 뺄셈

$$A+B=\begin{bmatrix}8\\4\\5\end{bmatrix}+\begin{bmatrix}1\\2\\3\end{bmatrix}=\begin{bmatrix}9\\6\\8\end{bmatrix}$$

$$A-B=\begin{bmatrix}8\\4\\5\end{bmatrix}-\begin{bmatrix}1\\2\\3\end{bmatrix}=\begin{bmatrix}7\\2\\2\end{bmatrix}$$

```
A = np.array([8, 4, 5])
B = np.array([1, 2, 3])
print('두 벡터의 합 :',A+B)
print('두 벡터의 차 :',A-B)
```

```
두 행렬의 합 : [9 6 8]
두 행렬의 차 : [7 2 2]
```

$$A+B=\begin{bmatrix}10&20&30&40\\50&60&70&80\end{bmatrix}+\begin{bmatrix}5&6&7&8\\1&2&3&4\end{bmatrix}=\begin{bmatrix}15&26&37&48\\51&62&73&84\end{bmatrix}$$

$$A-B=\begin{bmatrix}10&20&30&40\\50&60&70&80\end{bmatrix}-\begin{bmatrix}5&6&7&8\\1&2&3&4\end{bmatrix}=\begin{bmatrix}5&14&23&32\\49&58&67&76\end{bmatrix}$$

```
A = np.array([[10, 20, 30, 40], [50, 60, 70, 80]])
B = np.array([[5, 6, 7, 8],[1, 2, 3, 4]])
print('두 행렬의 합 :')
print(A + B)
print('두 행렬의 차 :')
print(A - B)
```

```
두 행렬의 합 :
[[15 26 37 48]
 [51 62 73 84]]
두 행렬의 차 :
[[ 5 14 23 32]
 [49 58 67 76]]
```

# 4. 벡터와 행렬 연산 – 내적과 행렬 곱

$$A \cdot B = [1\ 2\ 3] \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1\times4 + 2\times5 + 3\times6 = 32(\text{스칼라})$$

$$AB = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1\times5 + 3\times6 & 1\times7 + 3\times8 \\ 2\times5 + 4\times6 & 2\times7 + 4\times8 \end{bmatrix} = \begin{bmatrix} 23 & 31 \\ 34 & 46 \end{bmatrix}$$

```
A = np.array([1, 2, 3])
B = np.array([4, 5, 6])
print('두 벡터의 내적 :',np.dot(A, B))
```

```
두 벡터의 내적 : 32
```

```
A = np.array([[1, 3],[2, 4]])
B = np.array([[5, 7],[6, 8]])
print('두 행렬의 행렬곱 :')
print(np.matmul(A, B))
```

```
두 행렬의 행렬곱 :
[[23 31]
 [34 46]]
```

# 4. 벡터와 행렬 연산 – 다중 선형 회귀

- 독립변수가 2개 이상일 경우, 행렬 연산을 통해 H(x)표현

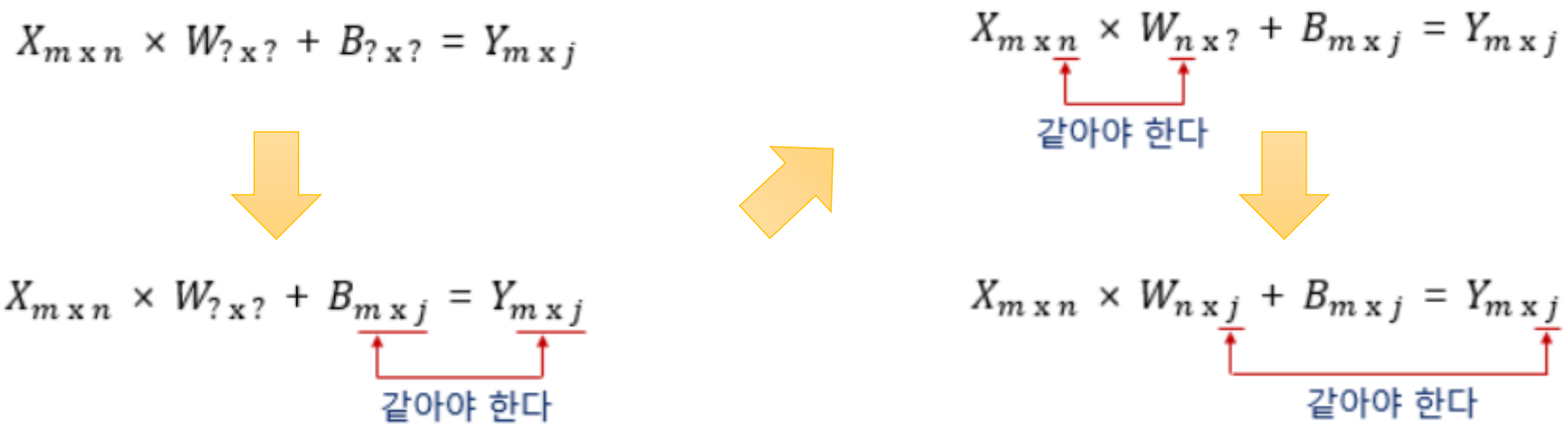
$$H(X) = WX + B$$

$$y = \begin{bmatrix} w_1 & w_2 & w_3 & \cdots & w_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_n \end{bmatrix} + b = x_1w_1 + x_2w_2 + x_3w_3 + \ldots + x_nw_n + b$$

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_{11} & x_{21} & x_{31} & x_{41} & x_{51} \\ x_{12} & x_{22} & x_{32} & x_{42} & x_{52} \\ x_{13} & x_{23} & x_{33} & x_{43} & x_{53} \\ x_{14} & x_{24} & x_{34} & x_{44} & x_{54} \end{bmatrix} + \begin{bmatrix} b & b & b & b & b \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 & y_5 \end{bmatrix}$$

# 4. 벡터와 행렬 연산 – 가중치와 편향 크기

- 두 행렬의 곱 J × K이 성립되기 위해서는 행렬 J의 열의 개수와 행렬 K의 행의 개수는 같아야 한다.
- 두 행렬의 곱 J × K의 결과로 나온 행렬 JK의 크기는 J의 행의 개수와 K의 열의 개수를 가진다.



# 감사합니다.