

3장. 기본기 다지기

기본 연산 해보기

- 텐서플로우가 다루는 자료형: **텐서(Tensor)**



- C, JAVA 언어에서 볼 수 있는 int, float, string 등과 같은 자료형에 해당
- 여러 형태를 가질 수 있는 넘파이 배열(NumPy Array)
- 배열 차원을 **랭크(Rank)**로 표현
- 아래 표의 '주로 사용하는 표현'으로 소통하지만, 코드에 Rank로 표현된 부분이 있어 알아두면 좋음

텐서가 필요한 이유
행렬로 인풋/w값 저장 가능
node값 계산식 쉬워짐



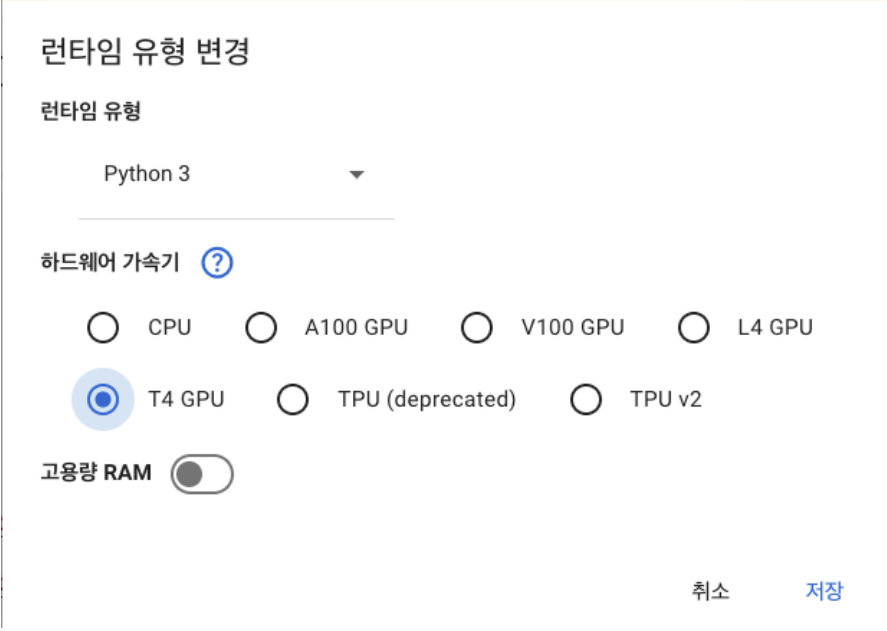
예	주로 사용하는 표현	텐서 표현	랭크
0, 1, 2, ...	스칼라(Scalar)	0-D Tensor	0
[1, 2, 3, 4, 5]	벡터(Vector)	1-D Tensor	1
[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]	행렬(Matrix)	2-D Tensor	2
[...[...]]	n차원 배열	n-D Tensor	n

[표 3-1] 우리가 사용하는 표현과 텐서의 표현 비교

Google Colab 사용

1. Google Colabe 접속 후, Google 드라이브 탭에서 “새 노트” 생성
<https://colab.research.google.com/>

2. 런타임 - 런타임 유형변경 에서 런타임 T4 GPU 유형으로 변경



런타임 유형 변경

런타임 유형

Python 3 ▼

하드웨어 가속기 ?

☐ CPU ☐ A100 GPU ☐ V100 GPU ☐ L4 GPU

☒ T4 GPU ☐ TPU (deprecated) ☐ TPU v2

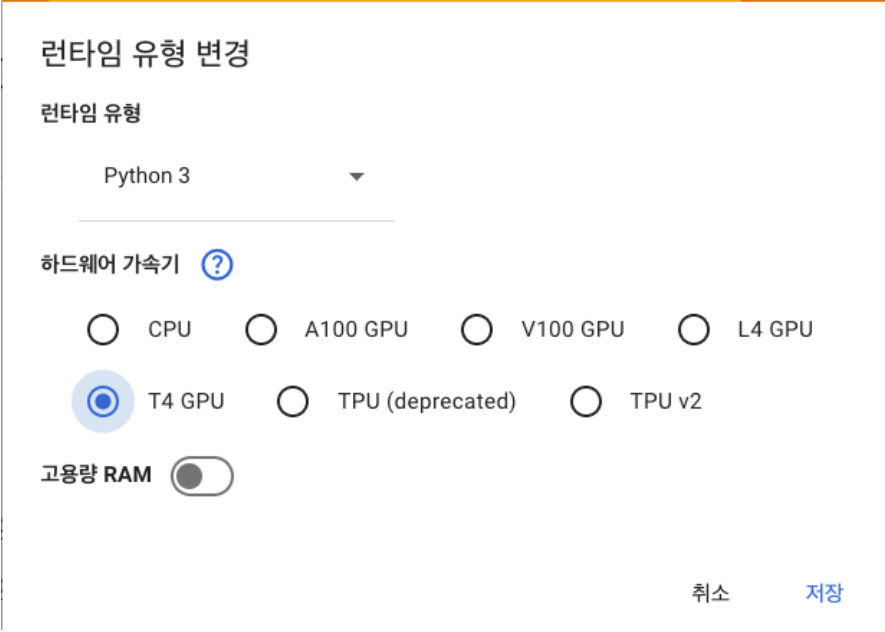
고용량 RAM ☐

취소 저장

Google Colab 사용

1. Google Colabe 접속 후, Google 드라이브 탭에서 “새 노트” 생성
<https://colab.research.google.com/>

2. 런타임 - 런타임 유형변경 에서 런타임 T4 GPU 유형으로 변경



런타임 유형 변경

런타임 유형

Python 3 ▼

하드웨어 가속기 ?

☐ CPU ☐ A100 GPU ☐ V100 GPU ☐ L4 GPU

☒ T4 GPU ☐ TPU (deprecated) ☐ TPU v2

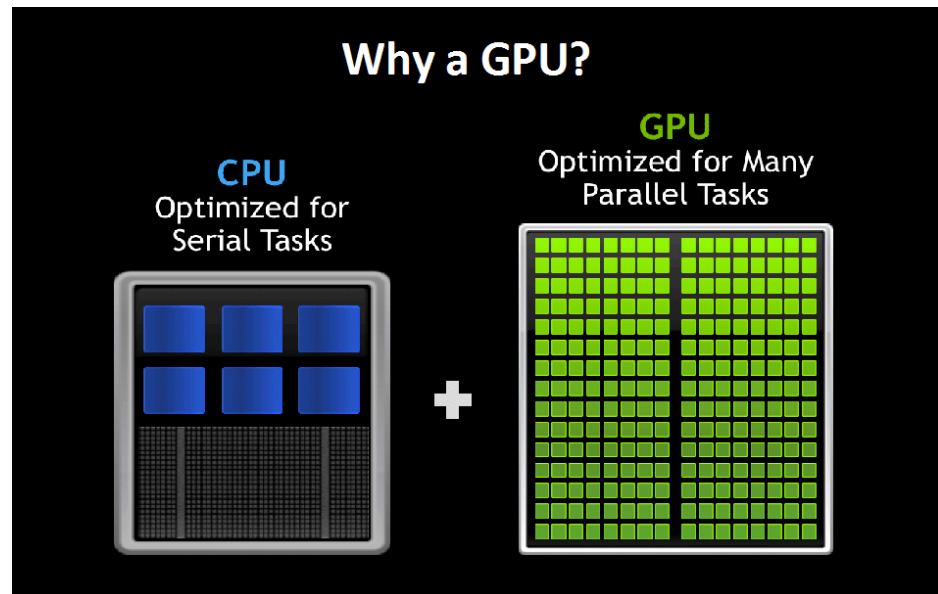
고용량 RAM ☐

취소 저장

Google Colab 사용

What are CPU and GPU?

- Numpy와 같은 연산을 GPU 가속 연산으로 대체
- 심층 신경망 구축
- **사용 편리성 & 기술 단순성**



처리 방식에서의 차이

- CPU: 직렬 처리에 최적화된 몇 개의 코어
- GPU: 병렬 처리 용으로 설계된 수천 개의 소형 코어

기본 연산 해보기 - colab

텐서 랭크 확인해 보기 - tf.rank() 함수

```
▶ import tensorflow as tf

a = tf.constant(2)
print(tf.rank(a))

b = tf.constant([1,2])
print(tf.rank(b))

c = tf.constant([[1,2], [3,4]])
print(tf.rank(c))
```

```
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
```

tensor의 datatype

기본 연산 해보기 - colab

```
import tensorflow as tf
import numpy as np

a = tf.constant(3)
b = tf.constant(2)

# 기본연산
# 텐서 형태로 출력해 보기
print(tf.add(a,b))      # 더하기
print(tf.subtract(a,b)) # 빼기

# 넘파이 배열 형태로 출력해 보기
print(tf.multiply(a,b).numpy()) # 곱하기
print(tf.divide(a,b).numpy())   # 나누기
```

```
tf.Tensor(5, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
6
1.5
```

기본 연산 해보기 - 행렬곱

Matrix Multiplication

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 5 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 3 + 12 & 15 + 28 \\ 2 + 3 & 10 + 7 \end{bmatrix}$$

Matrix 1

Matrix 2

$$= \begin{bmatrix} 15 & 43 \\ 5 & 17 \end{bmatrix}$$

```
import tensorflow as tf

a = tf.constant([[3,4],[2,1]])
b = tf.constant([[1,5],[3,7]])

print(tf.matmul(a, b))
```

```
tf.Tensor(
[[15 43]
 [ 5 17]], shape=(2, 2), dtype=int32)
```

tf.add()
tf.subtract()
tf.divide()
tf.multiply()
tf.matmul()



A와 B행렬의 곱 : AB
일명 dot product

기본 연산 해보기 - colab

```
c = tf.constant([1.9, 2.1], dtype=tf.float32) # 상수
```

```
d = tf.cast(c, tf.int32)
```

```
print(c, d)
```

```
tf.Tensor([1.9 2.1], shape=(2,), dtype=float32) tf.Tensor([1 2], shape=(2,), dtype=int32)
```

Variable

```
a = tf.Variable([2.0, 3.0])
```

```
a.assign([1, 2]) # assign으로 변경함
```

```
print(a)
```

```
<tf.Variable 'Variable:0' shape=(2,) dtype=float32, numpy=array([1., 2.], dtype=float32)>
```

기본 연산 해보기 - colab

tf.zeros()

```
b = tf.zeros([3, 4], tf.int32)
print(b)
print(b.shape)
print(b.dtype)
```

3행 4열

3 Row 4 Column

0만 담긴 텐서 만들어줌

```
tf.Tensor(
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]], shape=(3, 4), dtype=int32)
(3, 4)
<dtype: 'int32'>
```

기본 연산 해보기 - colab

```
import tensorflow as tf
tf.ones([3,4], tf.int32)
```

```
<tf.Tensor: shape=(3, 4), dtype=int32, numpy=
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]], dtype=int32)>
```

```
tf.eye(4)
```

```
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]], dtype=float32)>
```

기본 연산 해보기 - colab

tf.reshape()

```
import tensorflow as tf

t1 = [[1,2,3], [4,5,6]]
print(t1)

t2 = tf.reshape(t1, [6])
print(t2)

t3 = tf.reshape(t2, [3,2])
print(t3)
```

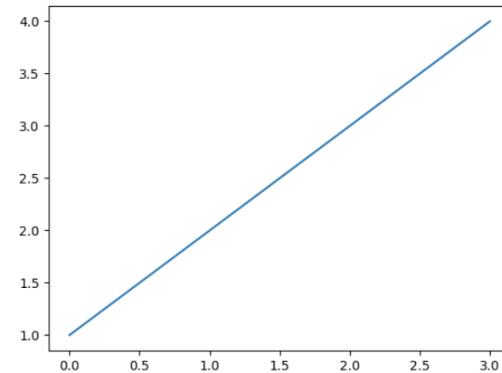
```
[[1, 2, 3], [4, 5, 6]]
tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)
tf.Tensor(
[[1 2]
 [3 4]
 [5 6]], shape=(3, 2), dtype=int32)
```

기본 연산 해보기 - matplotlib.pyplot 라이브러리 사용

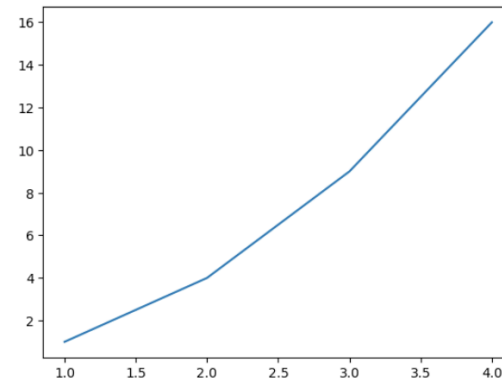
matplotlib.pyplot 모듈의 각각의 함수를 사용해서 간편하게 그래프를 만들고
변화를 줄 수 있습니다.

```
import matplotlib.pyplot as plt
```

```
plt.plot([1, 2, 3, 4])  
plt.show()
```



```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])  
plt.show()
```

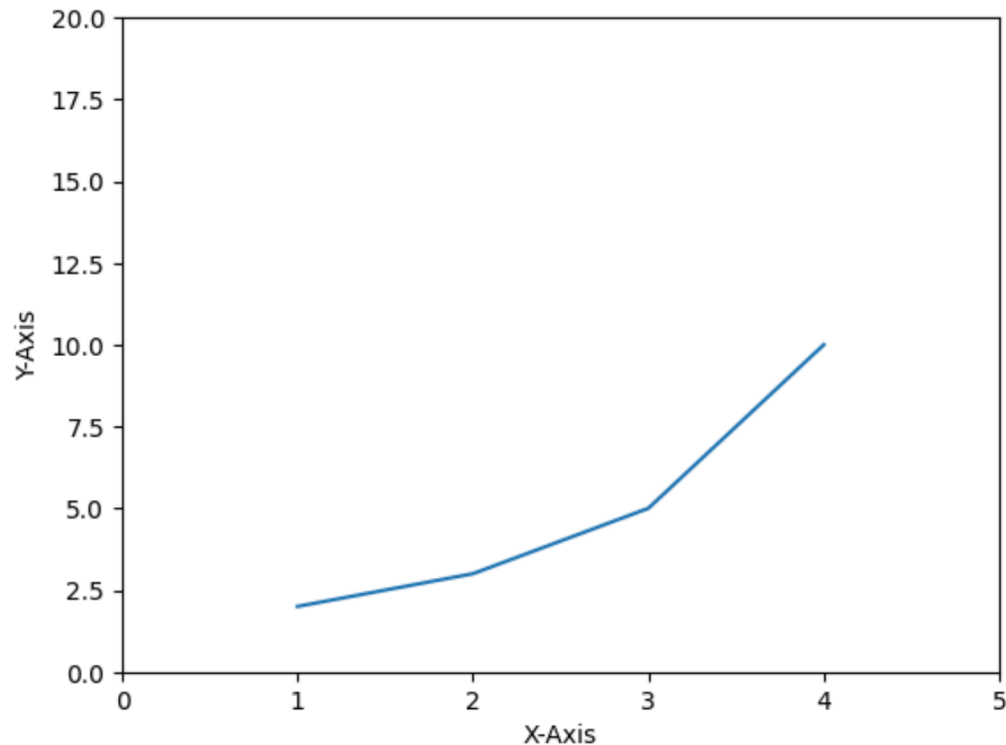


기본 연산 해보기 - matplotlib.pyplot 라이브러리 사용

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], [2, 3, 5, 10])
plt.xlabel('X-Axis')
plt.ylabel('Y-Axis')
plt.xlim([0, 5])      # x축의 범위: [xmin, xmax]
plt.ylim([0, 20])     # y축의 범위: [ymin, ymax]

plt.show()
```



기본 연산 해보기 - GPU 체크

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()

if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')

print('Found GPU at : {}'.format(device_name))
```

Found GPU at : /device:GPU:0

텐서에서 넘파이, 넘파이에서 텐서

넘파이 형태 배열로 변환하여 사용해보기

- numpy()와 convert_to_tensor() 함수
- tensor와 numpy array간 변환이 매우 유연

Numpy는 행렬이나 다차원 배열을 쉽게 처리할 수 있도록 지원하는 라이브러리

[함께 해봐요] 텐서에서 넘파이로, 넘파이에서 텐서로

basic_calc.ipynb

```
01 import tensorflow as tf
02 import numpy as np
03
04 c = tf.add(a, b).numpy() # a와 b를 더한 후 NumPy 배열 형태로 변환합니다.
05 c_square = np.square(c, dtype = np.float32)
    # NumPy 모듈에 존재하는 square 함수를 적용합니다.
06 c_tensor = tf.convert_to_tensor(c_square) # 다시 텐서로 변환해줍니다.
07
08 # 넘파이 배열과 텐서 각각을 확인하기 위해 출력합니다.
09 print('numpy array : %0.1f, applying square with numpy : %0.1f,
    convert_to_tensor : %0.1f' % (c, c_square, c_tensor))
```

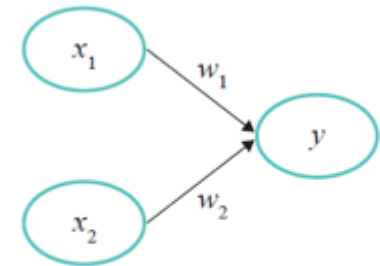
```
numpy array : 5.0, applying square with numpy : 25.0, convert_to_tensor : 25.0,
```


퍼셉트론 (Perceptron)

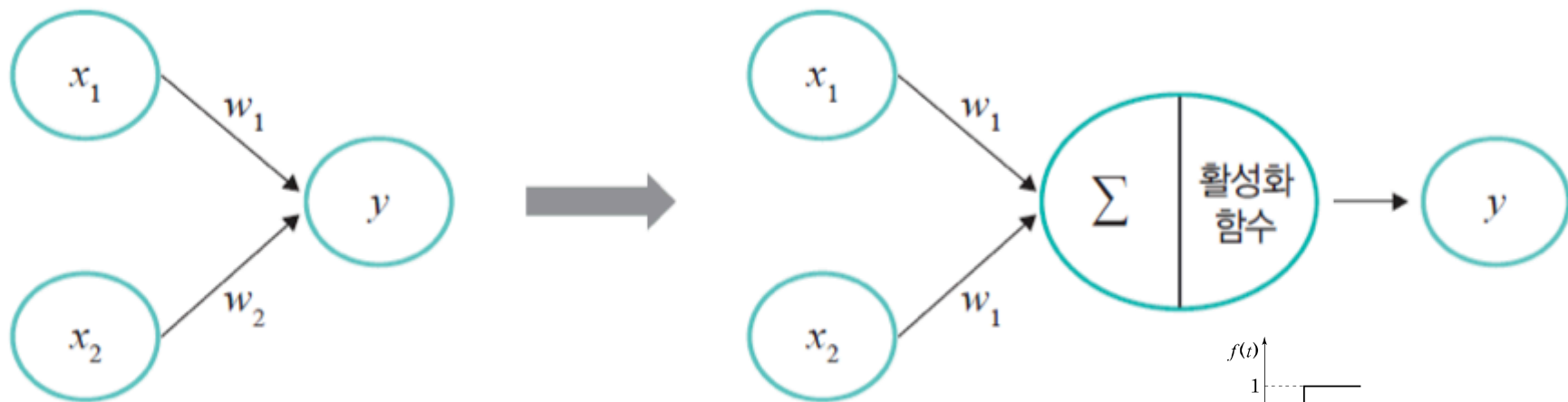
- 여러 개의 신호를 입력으로 받아 하나의 값을 출력
- x 는 입력, y 는 출력, w 는 가중치
- x 와 가중치 w 를 곱한 값을 모두 더하여 하나의 값(y)로 만들어냄
- 이때, 임계값(threshold)과 비교하여 크면 1, 그렇지 않으면 0을 출력

→ 활성화 함수(Activation Function)

→ 위에서 사용한 것은 계단 함수(Step Function)

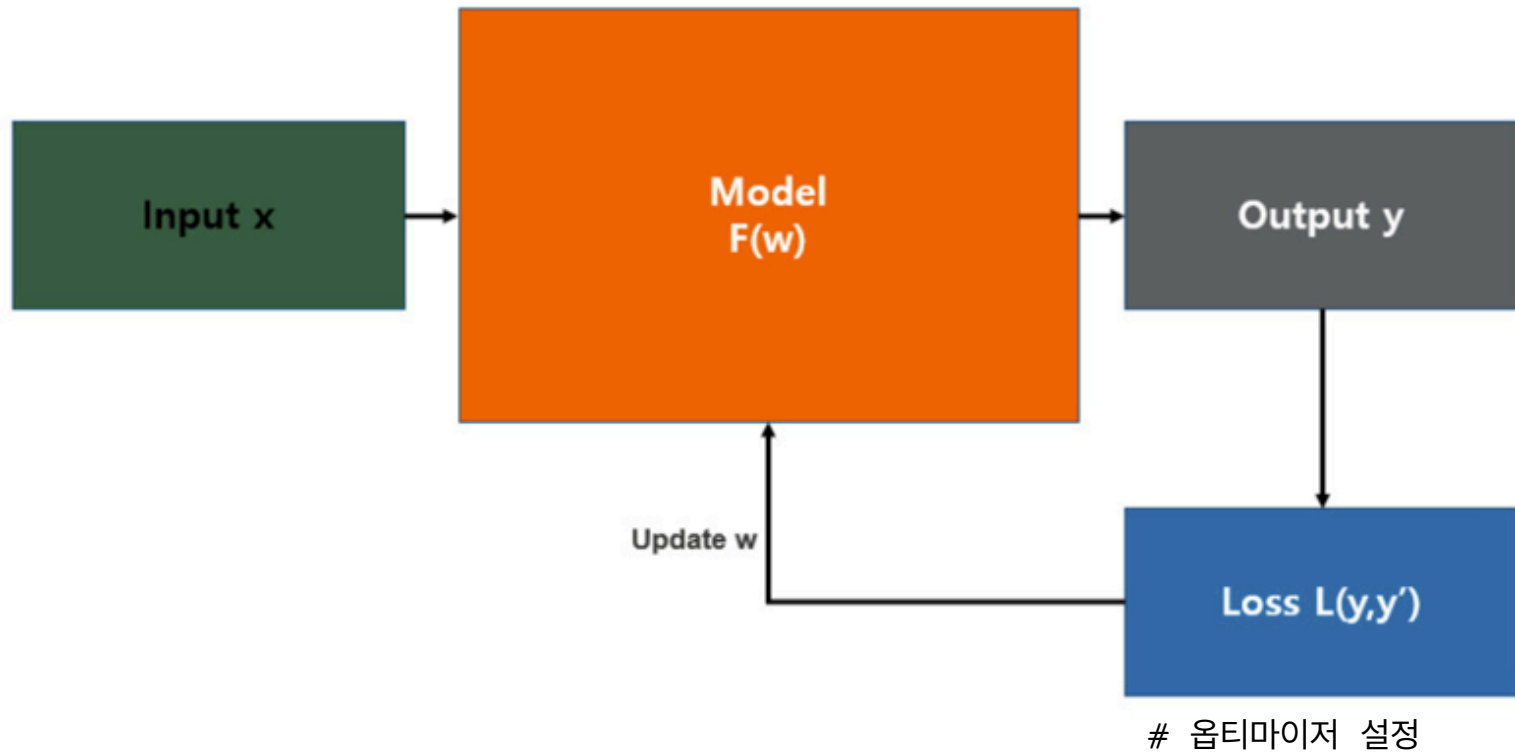


[그림 3-1] 퍼셉트론



[그림 3-2] 퍼셉트론과 퍼셉트론의 기본 단위

신경망 - 키와 몸무게를 예측하려면 (Linear Regression)



신경망 - 키와 몸무게를 예측하려면 (Linear Regression)

```
import tensorflow as tf

# 변수 초기화
height = 170
shoe_size = 260

# 회귀 계수 (a)와 절편 (b)를 텐서플로 변수로 선언
a = tf.Variable(0.1)
b = tf.Variable(0.2)

# 손실 함수 정의
def loss_function():
    predicted_value = height * a + b
    return tf.square(shoe_size - predicted_value) # 실제 값과 예측값의 차이의 제곱을 반환

# 옵티마이저 설정
opt = tf.keras.optimizers.Adam(learning_rate=0.1)

# 최적화 과정 수행
for i in range(300):
    # 손실함수를 최소화하는 방향으로 a와b를 업데이트
    opt.minimize(loss_function, var_list=[a, b])
    if i % 20 == 0: # 20번의 반복마다 한 번씩 a,b 값 출력
        print(f"Step {i}: a = {a.numpy()}, b = {b.numpy()}")
```

신경망 - 리스트 훈련데이터 (Linear Regression)

```
import tensorflow as tf

# 리스트 훈련데이터
train_x = [1, 2, 3, 4, 5, 6, 7]
train_y = [3, 5, 7, 9, 11, 13, 15]

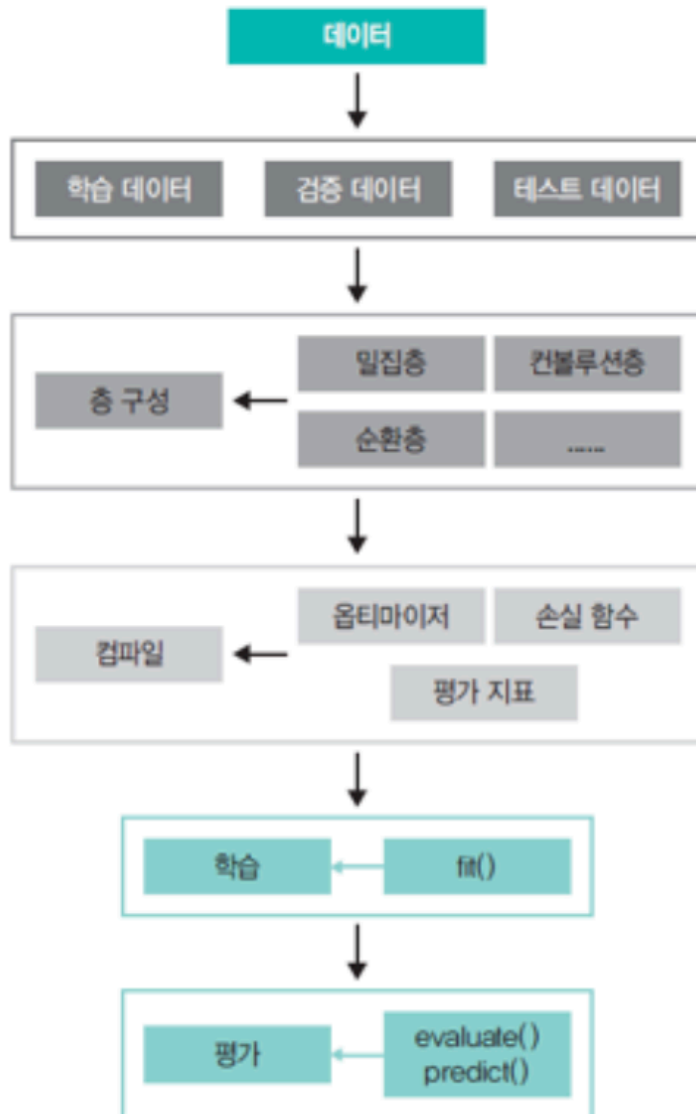
# 모델 파라미터 초기화 (기울기 a와 절편 b)
a = tf.Variable(0.1)
b = tf.Variable(0.1)

# 평균 제곱 오차를 계산하는 손실 함수 정의
def loss_function():
    predicted_y = train_x * a + b
    return tf.keras.losses.mse(train_y, predicted_y)

# 최적화 설정
opt = tf.keras.optimizers.Adam(learning_rate=0.001)

# 최적화 과정 수행
for i in range(300):
    opt.minimize(loss_function, var_list=[a, b])
    print(a.numpy(), b.numpy())
```

케라스에서의 개발 과정



1. 학습 데이터를 정의합니다.
2. 데이터에 적합한 모델을 정의합니다.
3. 손실 함수, 옵티마이저, 평가지표를 선택하여 학습 과정을 설정합니다.
4. 모델을 학습시킵니다.
5. 모델을 평가합니다.

[그림 3-12] 케라스에서의 개발 과정

신경망 - 퍼셉트론 OR 게이트 문제

```
import tensorflow as tf
tf.random.set_seed(777) # 시드를 설정합니다 (실험의 재생산성)
```

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.losses import mse
```

```
# 데이터 준비하기
x = np.array([[0,0], [1,0], [0,1], [1,1]])
y = np.array([[0], [1], [1], [1]])
```



모델 구성하기

```
model = Sequential()
# 단층 퍼셉트론을 구성합니다.
model.add(Dense(1, input_shape=(2,), activation='linear'))
```



모델 준비하기

```
model.compile(optimizer=SGD(), loss=mse, metrics=['acc']) # list 형태로 평가지표를 전달합니다.
```



학습시키기

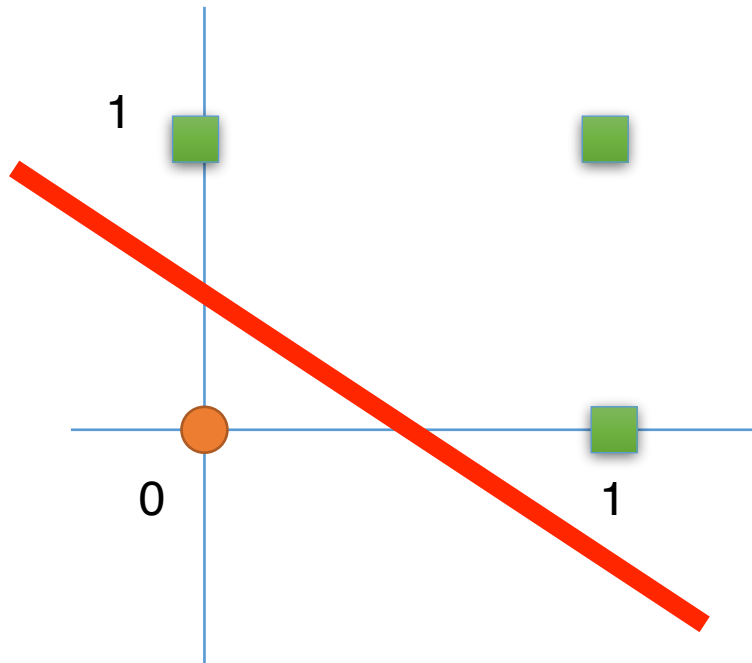
```
model.fit(x, y, epochs=500)
```



```
Epoch 194/500
4/4 [=====] - 0s 2ms/sample - loss: 0.1164 - acc: 1.0000
... 생략 ...
```

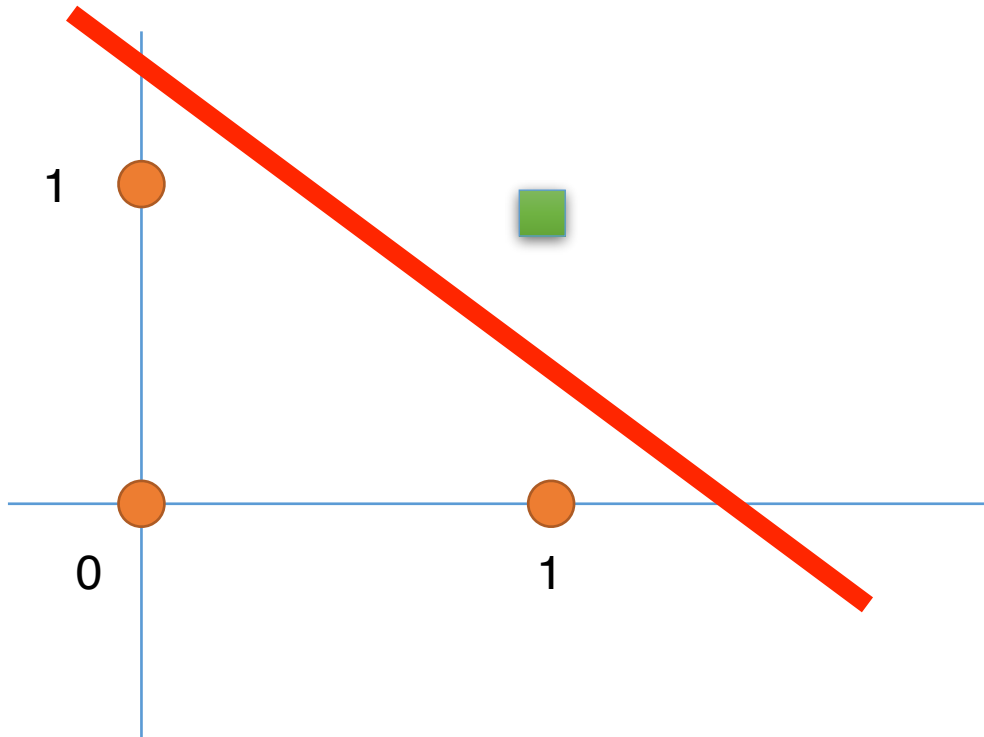
퍼셉트론 - OR 게이트

```
data = np.array([[0,0], [1,0],[0,1], [1,1]])  
label = np.array([[0], [1], [1], [1]])
```



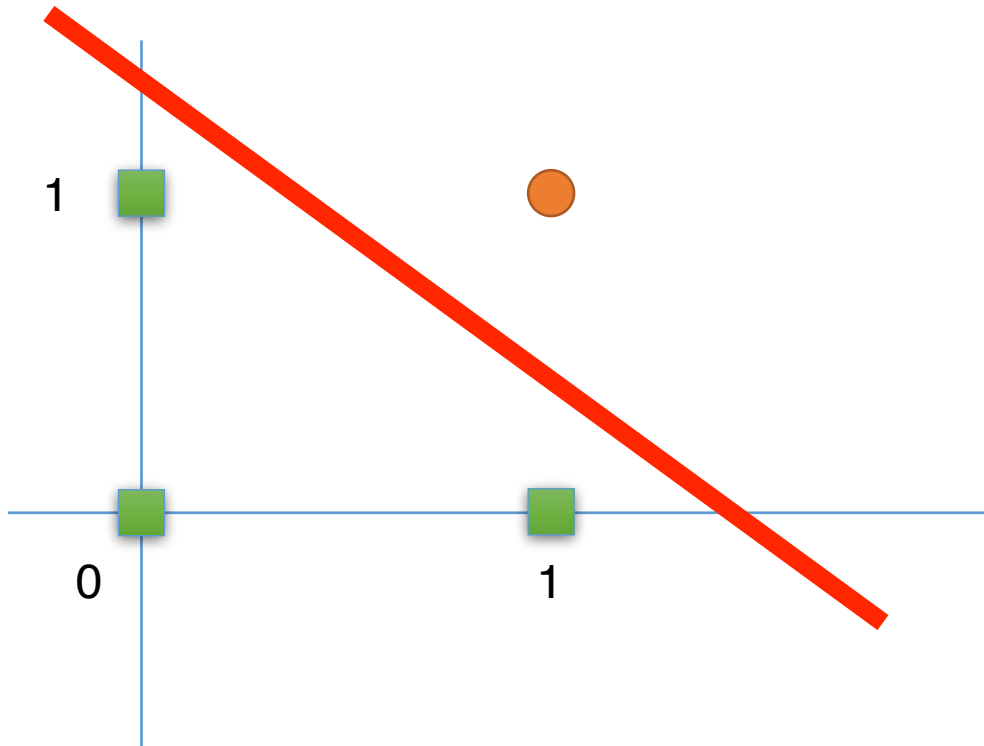
퍼셉트론 - AND 게이트

```
data = np.array([[0,0], [1,0],[0,1], [1,1]])  
label = np.array([[0], [0], [0], [1]])
```



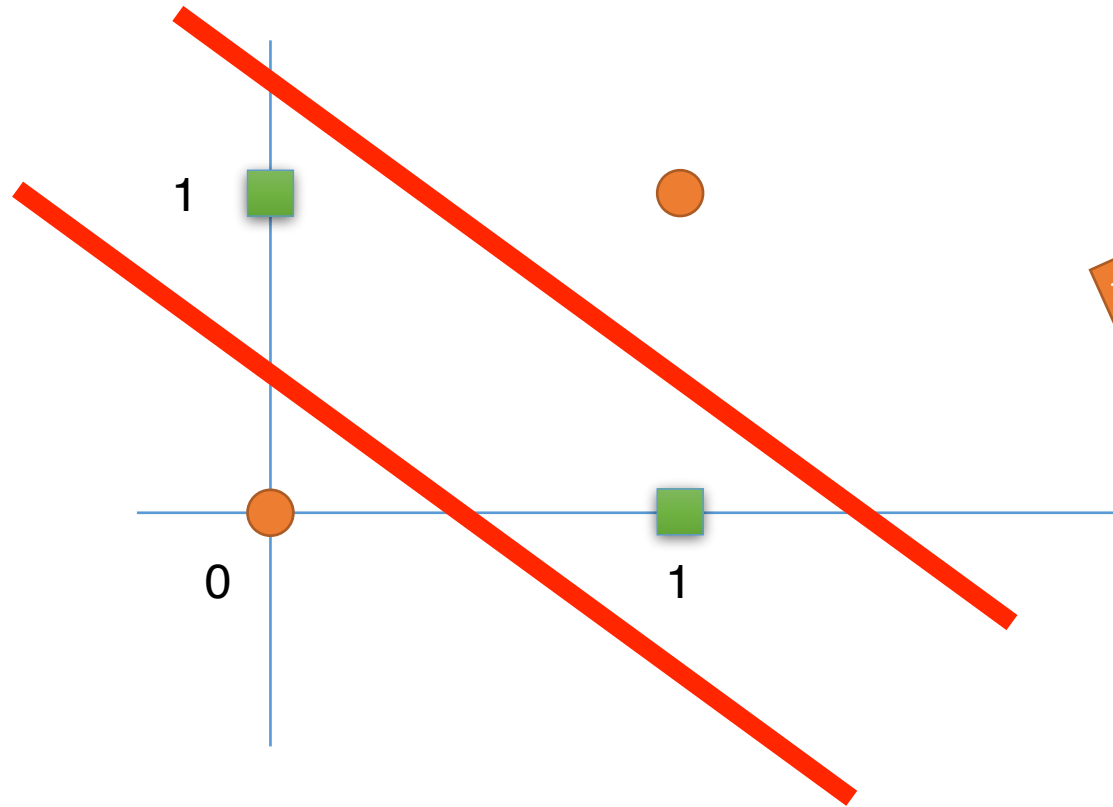
퍼셉트론 - NAND 게이트

```
data = np.array([[0,0], [1,0],[0,1], [1,1]])  
label = np.array([[1], [1], [1], [0]])
```



XOR 게이트 - 네모와 동그라미를 구분할 수 있는가?

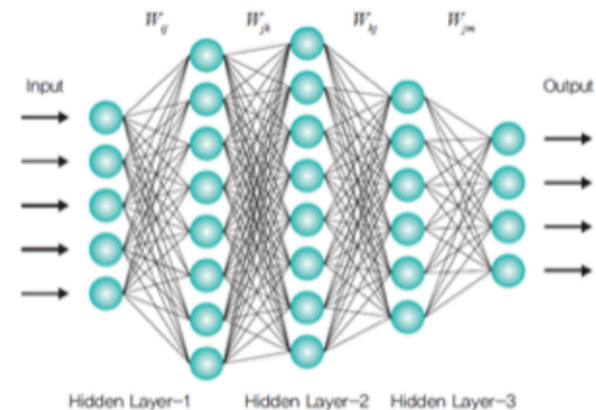
```
data = np.array([[0,0], [1,0],[0,1], [1,1]])  
label = np.array([[0], [1], [1], [0]])
```



퍼셉트론으로는
해결 할 수 없음

전과 같은 문제를 해결한 것이 **다층 퍼셉트론(Multi-Layer Perceptron)**

- 그림의 선이 전부 가중치에 해당함
- 실제로 사용한 퍼셉트론은 굉장히 많음
 - 연산 비용이 큼
 - **벡터화(Vectorization)**을 이용



[그림 3-4] 다층 퍼셉트론

m : 데이터의 개수
n : 데이터 특성의 개수
k : 층의 유닛 수

$$\begin{bmatrix} X_{11} & \cdots & X_{1n} \\ \vdots & \ddots & \vdots \\ X_{m1} & \cdots & X_{mn} \end{bmatrix} \cdot \begin{bmatrix} W_{11} & \cdots & W_{1k} \\ \vdots & \ddots & \vdots \\ W_{n1} & \cdots & W_{nk} \end{bmatrix} = \begin{bmatrix} Z_{11} & \cdots & Z_{1k} \\ \vdots & \ddots & \vdots \\ Z_{m1} & \cdots & Z_{mk} \end{bmatrix}$$

(m, n)
(n, k)
(m, k)

벡터의 내적

```
x = tf.random.uniform((10,5))  
w = tf.random.uniform((5,3))  
  
d = tf.matmul(x,w)  
  
print(f'x와 w의 벡터 내적의 결과 크기 {d.shape}')
```

x와 w의 벡터 내적의 결과 크기 (10, 3)

다층 퍼셉트론 - XOR

```
import tensorflow as tf
tf.random.set_seed(777)
```

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop
from keras.losses import mse
```

```
data = np.array([[0,0], [1,0],[0,1], [1,1]])
label = np.array([[0], [1], [1], [0]])
```

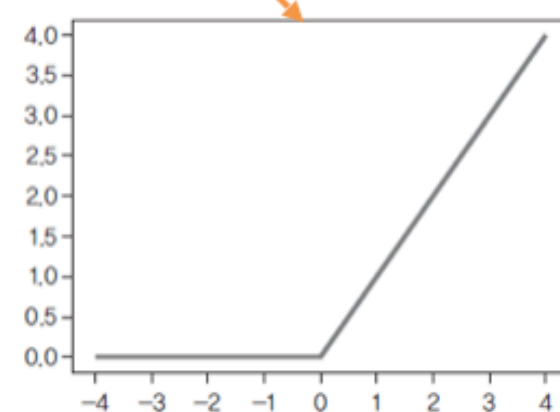
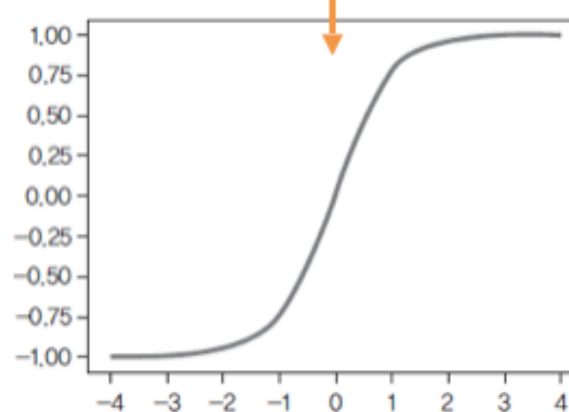
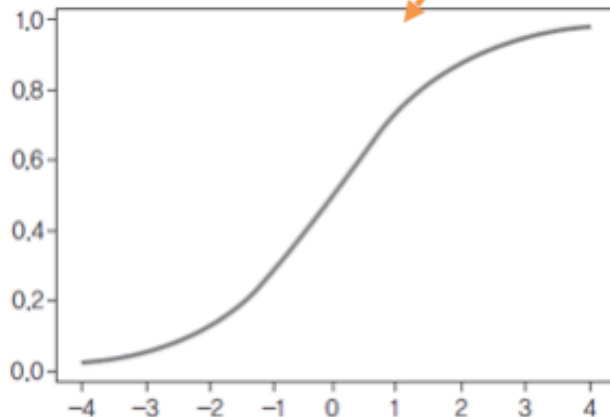
```
model = Sequential()
model.add(Dense(32, input_shape=(2,), activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
model.compile(optimizer=RMSprop(), loss=mse, metrics = ['acc'])

model.fit(data, label, epochs=100)
```

활성화 함수

- XOR 게이트 문제에서 ReLU 활성화 함수를 사용
 - 비선형 활성화 함수
 - 선형 활성화 함수를 쓰면, $f(f(f(x))) \rightarrow f(x)$ 와 동일
 - 층을 쌓는 의미가 없어져... \rightarrow 비선형 활성화 함수 사용으로 해결
- 대표적으로 사용되는 시그모이드(Sigmoid), 하이퍼볼릭 탄젠트(tanh), ReLU 활성화 함수
 - 그 외에 PReLU, ELU, Leaky-ReLU, swish 등이 존재



활성화 함수 - 직접 구현해 보기

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

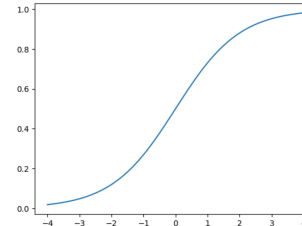
```
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

```
def tanh(x):
    return list(map(lambda x : math.tanh(x),x))
```

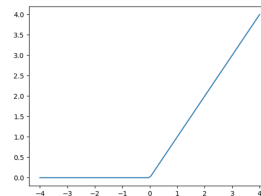
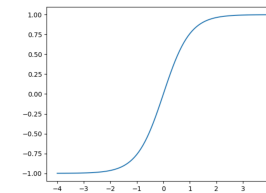
```
def relu(x):
    result = []
    for ele in x:
        if(ele <= 0):
            result.append(0)
        else:
            result.append(ele)
```

```
    return result
```

```
x = np.linspace(-4,4, 100)
print(x)
sig =sigmoid(x)
plt.plot(x,sig)
plt.show()
```



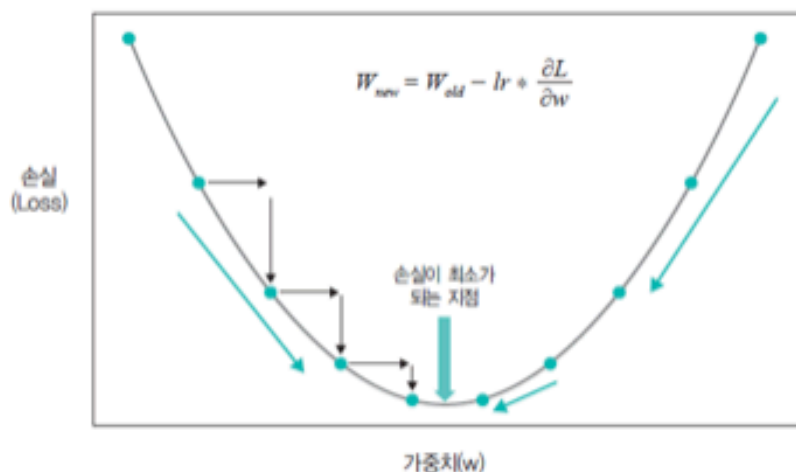
0~1 사이의 확률



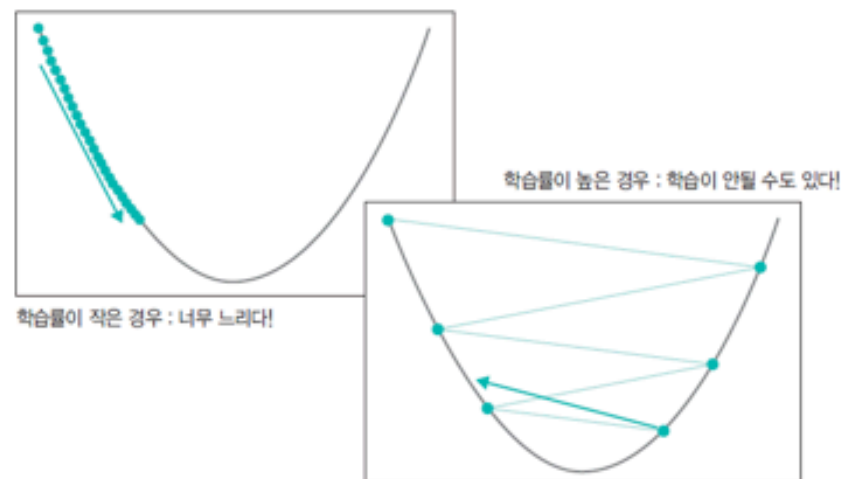
경사하강법

Gradient Descent Algorithm

- 특정 함수에서의 미분을 통해 얻은 기울기를 활용하여 최적의 값을 찾아가는 방법



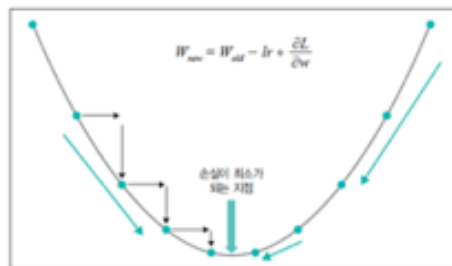
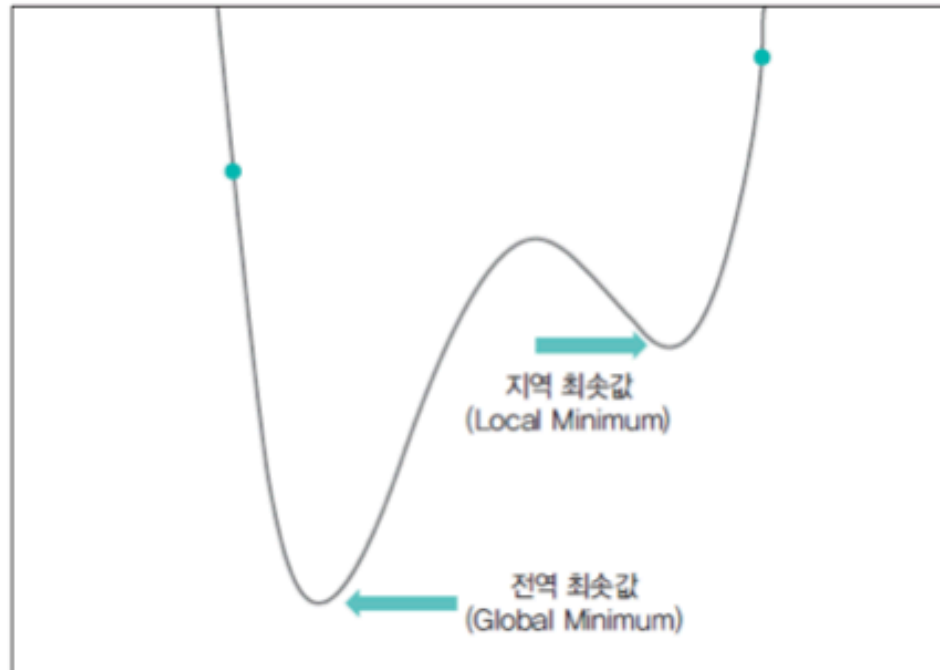
[그림 3-8] 경사하강법($y=x^2$)



[그림 3-10] 학습에 영향을 미치는 학습률의 크기

- 그림의 **학습률(lr, learning rate)**은 성능, 학습 속도에 중요한 영향을 끼치는 하이퍼파라미터
 - 학습률이 너무 높으면 학습이 되지 않을 수 있음
 - 그렇다고 너무 낮으면, 최적값에 도달하기 전에 학습이 종료
 - 주로 **1e-3(0.001, 또는 1e-4)**을 기본값으로 사용
- 위 함수는 어느 지점에서 출발해도 경사를 따라가다 보면 최적값(손실이 최소가 되는 지점)에 도달함
- 하지만 우리가 만날 함수 공간(space)은?

경사하강법



[그림 3-4] 경사하강법(Gradient Descent)

- 경사하강법은 항상 최적값을 반환한다는 보장을 할 수 없음

- 왼쪽 점에서 시작할 경우
 - 전역 최솟값 ← Good!!
- 오른쪽 점에서 시작할 경우
 - 지역 최솟값 ← Bad!!

- 가중치 초기화 문제

- weight initialization
- 왼쪽 점? 오른쪽 점?
- 특별한 경우가 아닌 이상, 케라스가 제공하는 기본 값(default)을 사용해도 무방
- Glorot(Xavier), he, Lecun 초기화

- 배치 단위를 사용하여 진행

- 확률적 경사 하강법
- SGD; Stochastic Gradient Descent

경사하강법 - 실습

```
import numpy as np
import matplotlib.pyplot as plt

lr_list = [0.001, 0.1, 0.5, 0.9]

def get_derivative(lr):
    w_old = 2
    derivative = [w_old]

    y = [w_old ** 2]

    for i in range(1,10):
        dev_value = w_old * 2

        w_new = w_old - lr * dev_value
        w_old = w_new

        derivative.append(w_old)
        y.append(w_old ** 2)

    return derivative, y
```

경사하강법 - 실습

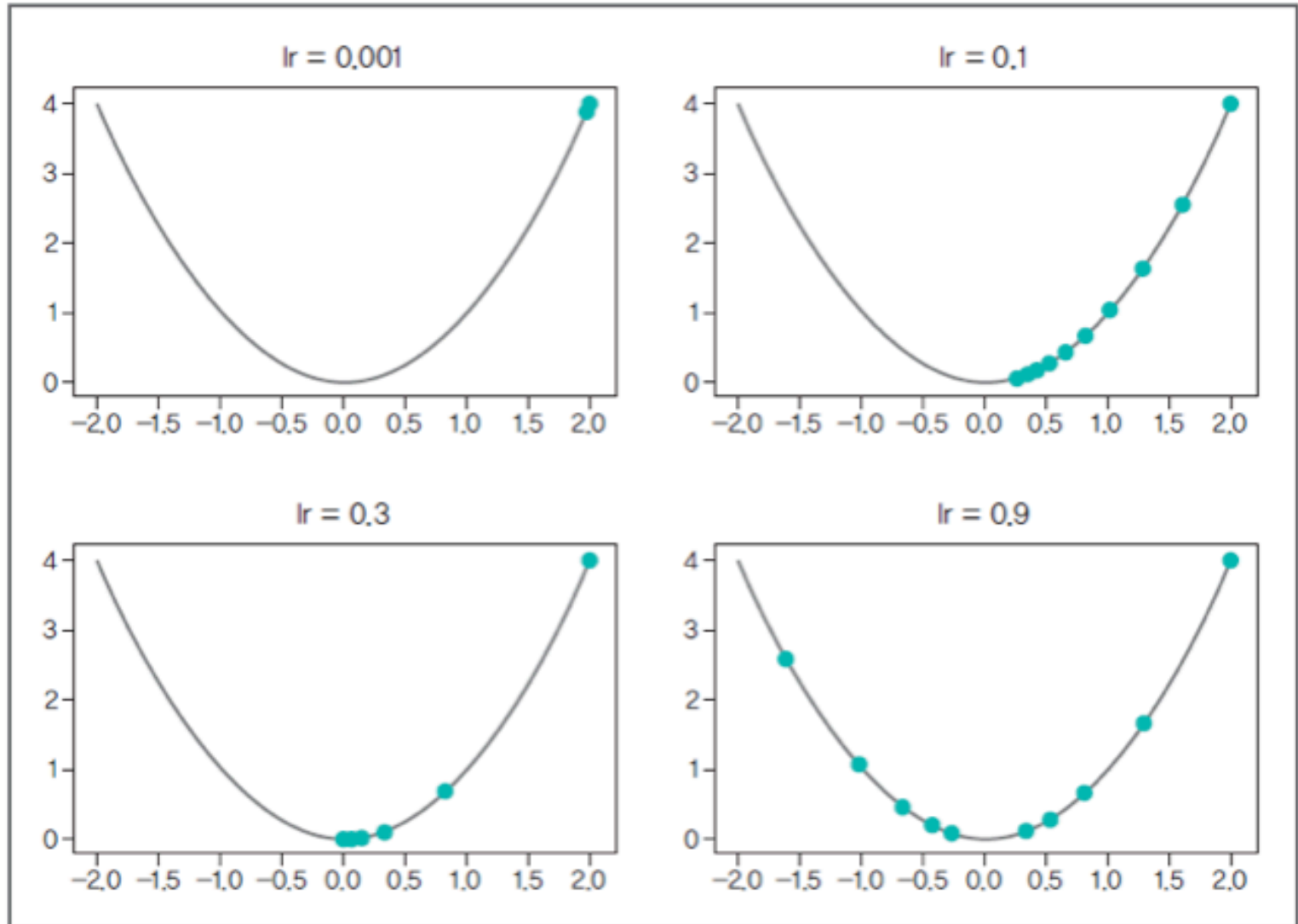
```
x = np.linspace(-2, 2, 50)
x_square = [i**2 for i in x]

fig = plt.figure(figsize=(12,7))

for i, lr in enumerate(lr_list):
    derivative, y = get_derivative(lr)
    ax = fig.add_subplot(2,2,i+1)
    ax.scatter(derivative, y, color='red')
    ax.plot(x,x_square)
    ax.title.set_text('lr =' + str(lr))

plt.show()
```

경사하강법 - 실습



경사하강법 - 역전파

신경망을 학습시킬 멋진 방법

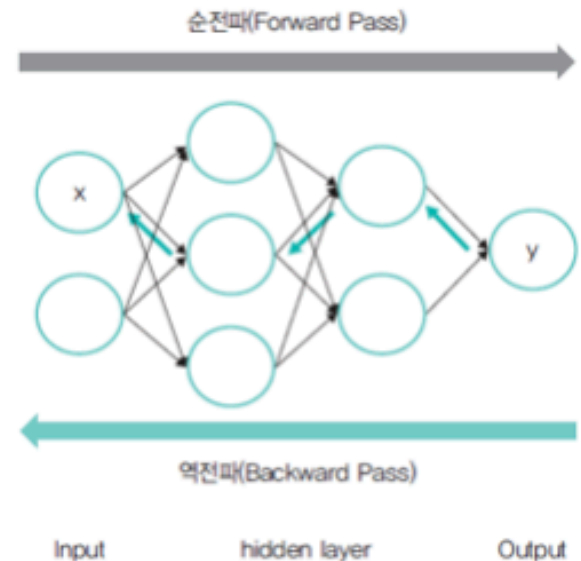
- 역전파 알고리즘, Backpropagation Algorithm
- 가중치를 무작위로 설정한 모델에서 결괏값을 도출하고, 이를 정답과 비교하여 가중치를 재조정하는 과정에서 사용
- ex) 햄버거 만드는 과정 → 순전파(Forward-Pass),
햄버거에 고객의 피드백을 반영 → 역전파(Backward-Pass)

효율적인 계산을 위한 체인룰(Chain-Rule)을 사용

- 미분 개념이 필요함

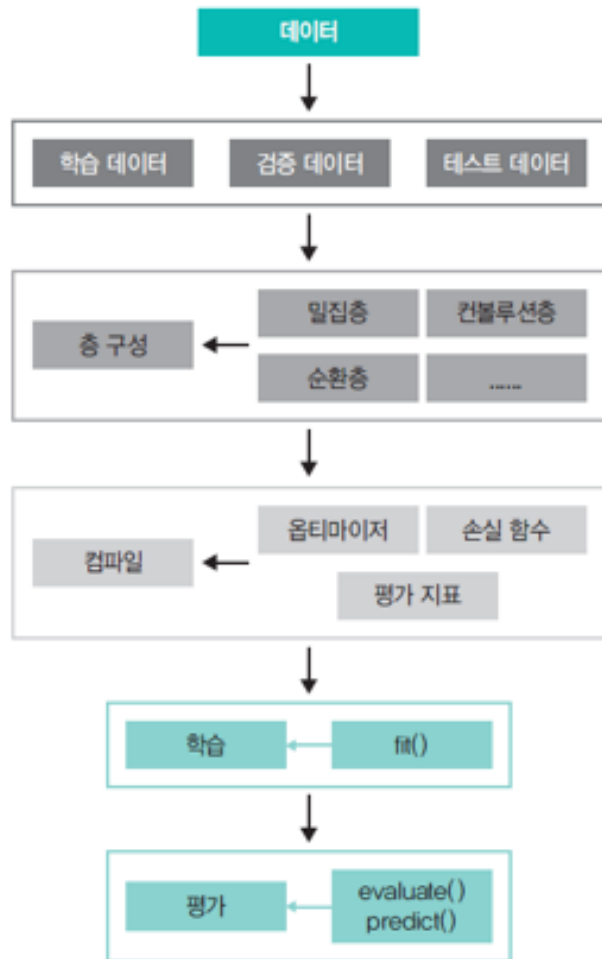
$$\frac{\partial f}{\partial x} = \frac{\partial f(g)}{\partial g} \cdot \frac{\partial g(x)}{\partial x}$$

- 하지만 딥러닝 라이브러리가 다 해주므로,
'신경망의 가중치가 역전파를 통해 업데이트되는구나!'를 파악!



[그림 3-11] 순전파와 역전파

케라스에서의 개발 과정



[그림 3-12] 케라스에서의 개발 과정

1. 학습 데이터를 정의합니다.
2. 데이터에 적합한 모델을 정의합니다.
3. 손실 함수, 옵티마이저, 평가지표를 선택하여 학습 과정을 설정합니다.
4. 모델을 학습시킵니다.
5. 모델을 평가합니다.

케라스에서의 개발 과정

compile() 함수를 통한 학습 과정 설정

예시: model.compile()

```
01 # 평균 제곱 오차 회귀 문제
02 model.compile(optimizer = RMSprop(),
03               loss = 'mse',
04               metrics = [ ])
05
06 # 이항 분류 문제
07 model.compile(optimizer = RMSprop(),
08               loss = 'binary_crossentropy',
09               metrics = ['acc'])
10
11 # 다항 분류 문제
12 model.compile(optimizer = RMSprop(),
13               loss = 'categorical_crossentropy',
14               metrics = ['acc'])
```

- **옵티마이저(optimizer)** : 최적화 방법을 설정, SGD(), RMSProp(), Adam(), NAdam() 등
 - 'sgd', 'rmsprop', 'adam'과 같이 문자열로 지정하여 사용 가능
 - tf.keras.optimizers
- **손실 함수(loss function)** : 학습 과정에서 최적화시켜야 할 손실 함수를 설정
 - mse(mean_squared_error), binary_crossentropy, categorical_crossentropy
 - tf.keras.losses
- **평가 지표(metrics)**: 학습 과정을 모니터링하기 위해 설정
 - tf.keras.metrics

케라스에서의 개발 과정

fit() 함수를 통한 모델 학습

예시: model.fit()

```
01 model.fit(data, label, epochs = 100)
02
03 model.fit(data, label, epochs = 100, validation_data = (val_data, val_label))
```

- 에폭(epochs): 전체 학습 데이터를 몇 회 반복할지 결정
- 배치 크기(batch_size): 전달한 배치 크기만큼 학습 데이터를 나누어 학습을 진행
- 검증 데이터(validation_data): 모델 성능을 모니터링하기 위해 사용

평가 진행

- evaluate(), predict()

예시: model.evaluate(), model.predict()

```
01 model.evaluate(data, label)
```

```
[0.21061016619205475, 1.0] 손실과 평가지표
```

```
01 result = model.predict(data)
02 print(result)
```

```
array([[0.48656905],
       [0.5464304 ],
       [0.552116  ],
       [0.4465039  ]], dtype=float32)
```


RECAP

1. 신경망은 **퍼셉트론 알고리즘**에서 부터 출발합니다.
다중퍼셉트론을 사용하면 XOR 게이트 문제를 해결 할 수 있습니다.
2. **경사하강법**과 **역전파**는 학습을 위해 사용되는 주요 개념입니다.
경사하강법에서는 **학습률**, **가중치 초기화**에 대해 알아보았으며,
역전파에서는 **체인 룰**을 사용하는 것을 배웠습니다.
3. 케라스에서의 개발 과정은
[데이터 정의 -> 모델 정의 -> 손실함수, 옵티마이저, 평가지표 선택 -> 모델 학습]
으로 이루어집니다.
4. 케라스 모델의 **첫 번째 층**은 항상 입력 데이터의 형태를 전달해 주어야 합니다.
5. 대표적으로 손실 함수에는 ['mse', 'binary_crossentropy', 'categorical_crossentropy']
옵티마이저에는 ['sgd', 'rmsprop', 'Adam'] 이 있으며, 문자열로 지정하여
사용할 수 있습니다.