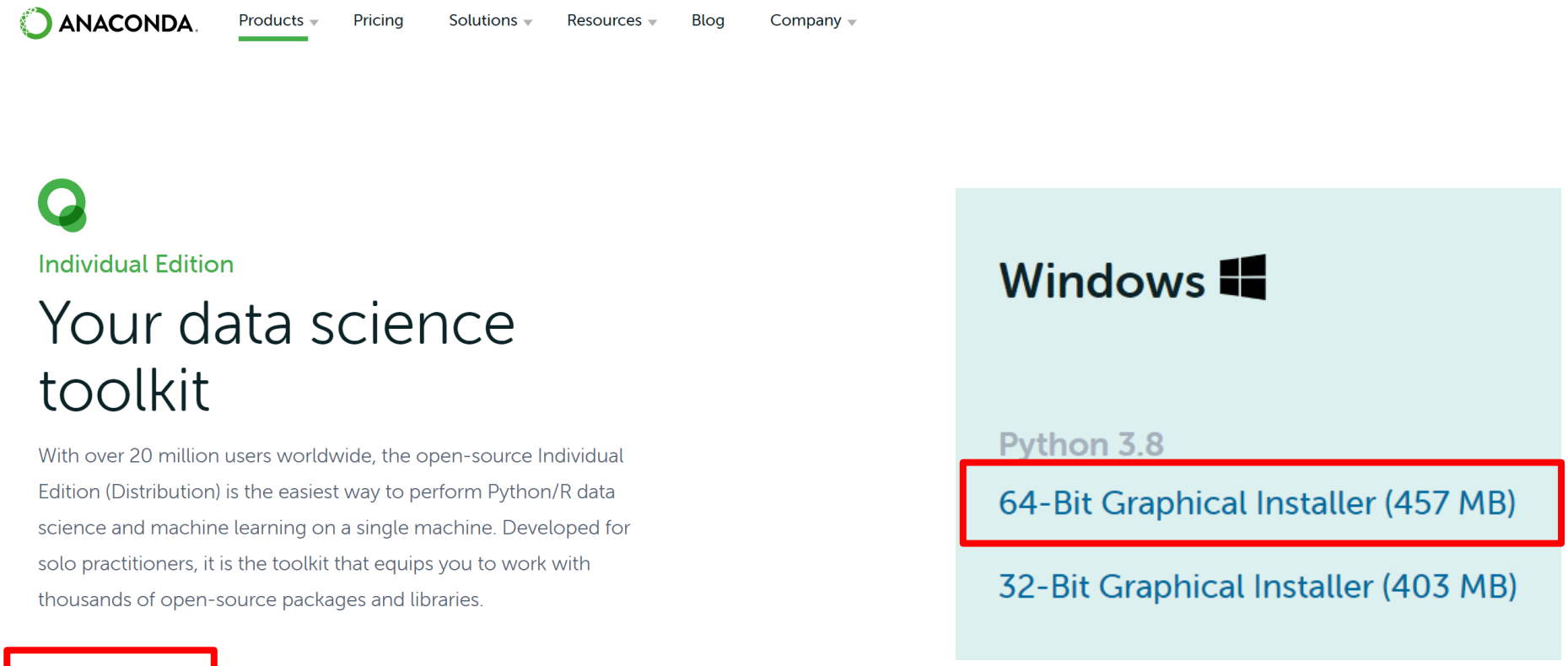
환경 설정

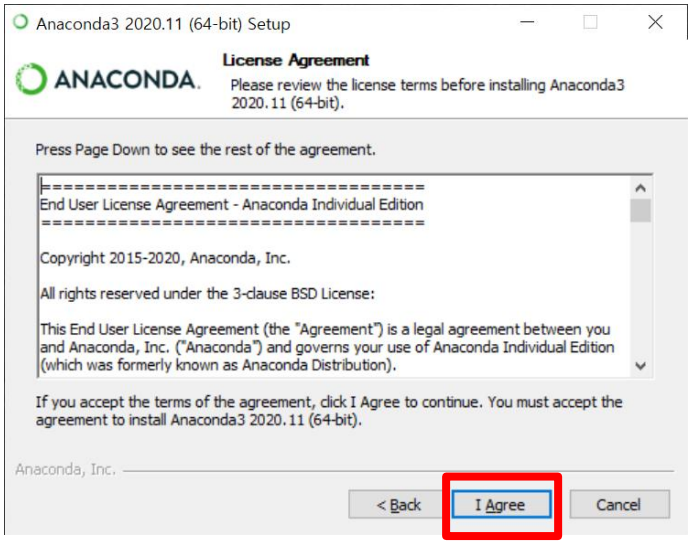
아나 콘다 설치

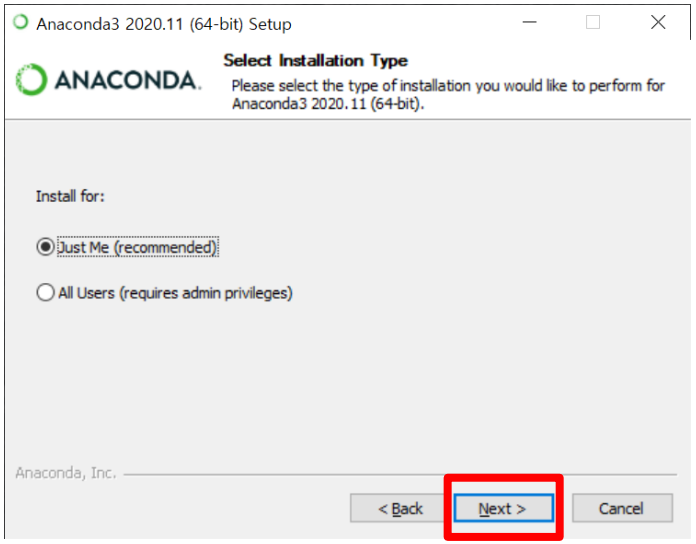
https://www.anaconda.com/products/individual

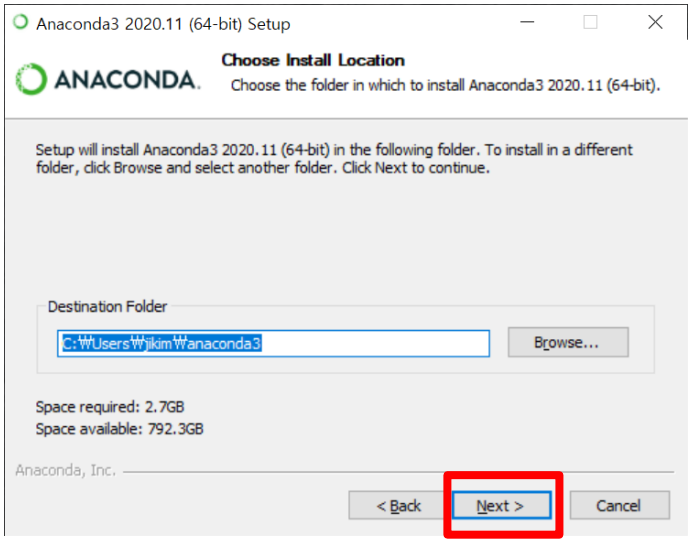


환경 설정

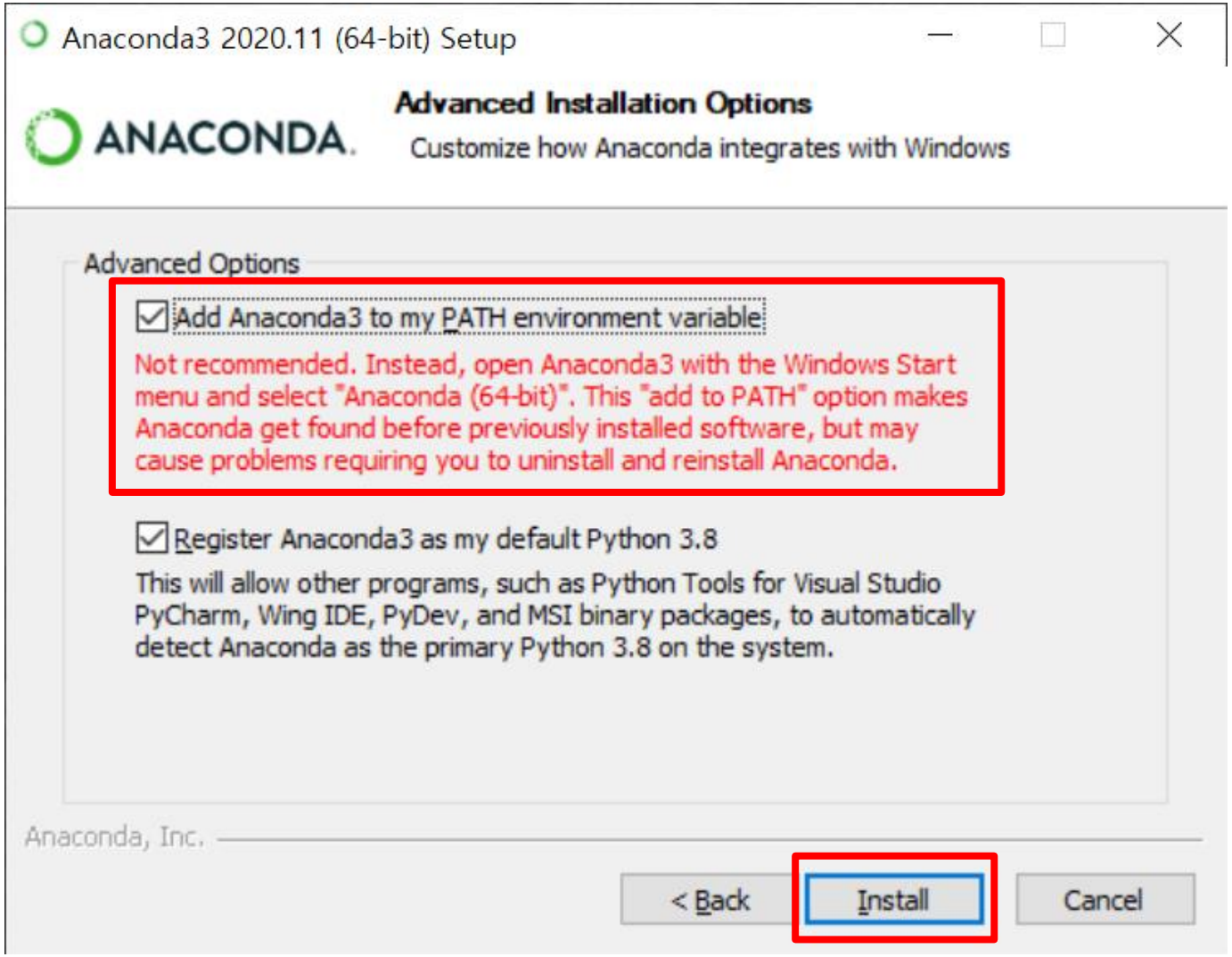

Anaconda3 2020.11 (64-bit) Setup
ANACONDA.
Welcome to Anaconda3 2020.11 (64-bit) Setup
Setup will guide you through the installation of Anaconda3 2020.11 (64-bit).
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel


Anaconda3 2020.11 (64-bit) Setup
ANACONDA.
License Agreement
Please review the license terms before installing Anaconda3 2020.11 (64-bit).
Press Page Down to see the rest of the agreement.
End User License Agreement - Anaconda Individual Edition
Copyright 2015-2020, Anaconda, Inc.
All rights reserved under the 3-clause BSD License:
This End User License Agreement (the "Agreement") is a legal agreement between you and Anaconda, Inc. ("Anaconda") and governs your use of Anaconda Individual Edition (which was formerly known as Anaconda Distribution).
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 2020.11 (64-bit).
Anaconda, Inc.
< Back
I Agree
Cancel


Anaconda3 2020.11 (64-bit) Setup
ANACONDA.
Select Installation Type
Please select the type of installation you would like to perform for Anaconda3 2020.11 (64-bit).
Install for:
Just Me (recommended)
All Users (requires admin privileges)
Anaconda, Inc.
< Back
Next >
Cancel

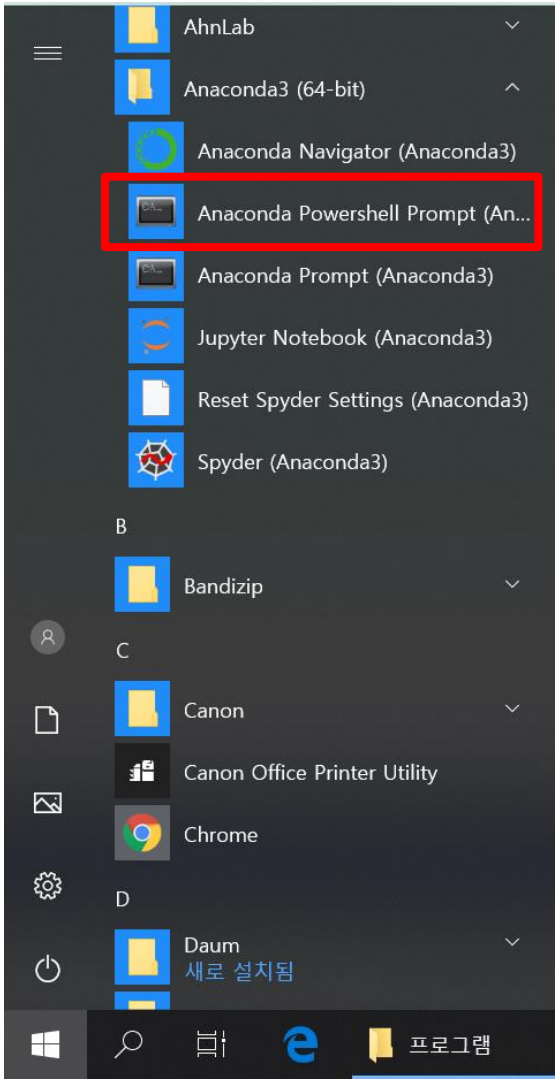

Anaconda3 2020.11 (64-bit) Setup
ANACONDA.
Choose Install Location
Choose the folder in which to install Anaconda3 2020.11 (64-bit).
Setup will install Anaconda3 2020.11 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:₩Users₩jikim₩anaconda3
Browse...
Space required: 2.7GB
Space available: 792.3GB
Anaconda, Inc.
< Back
Next >
Cancel

환경 설정


Anaconda3 2020.11 (64-bit) Setup
ANACONDA.
Advanced Installation Options
Customize how Anaconda integrates with Windows
Advanced Options
Add Anaconda3 to my PATH environment variable
Not recommended. Instead, open Anaconda3 with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.
Register Anaconda3 as my default Python 3.8
This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.8 on the system.
Anaconda, Inc.
< Back
Install
Cancel

환경 설정

아나콘다 파워 쉘 실행



환경 설정

텐서 플로 설치

```
(base) PS C:\Users\jikim> pip install tensorflow
```

파이썬 실행

```
(base) PS C:\Users\jikim> python
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

텐서플로 임포트 확인

```
>>> import tensorflow as tf
>>> print(tf.__version__)
>>> exit()
```

소스가 있는 디렉토리로 이동 후

```
(base) C:\Users\jikim> cd "F:\텐서플로우_수업자료\동영상_컨텐츠\TF2-AI-main"
```

주피터 노트북 실행

```
(base) F:\텐서플로우_수업자료\동영상_컨텐츠\TF2-AI-main> jupyter notebook
```

환경 설정

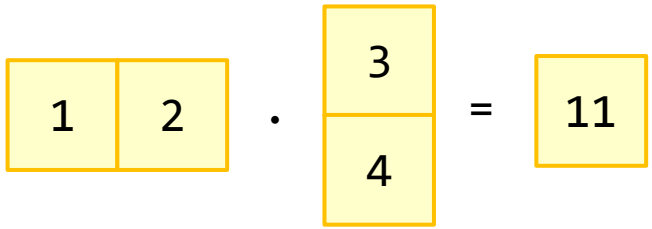
tensorflow import시 dll 에러 나시는 분들은
아래 사이트 에서

https://support.microsoft.com/en-my/help/2977003/the-latest-supported-visual-c-downloads

x86: vc_redist.x86.exe
x64: vc_redist.x64.exe
ARM64: vc_redist.arm64.exe

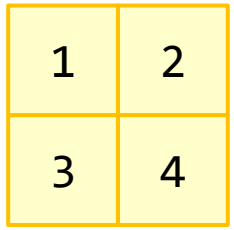
배재포 실행 파일을 다운 받아 설치 한후
다시 import 해보세요.
텐서 플로 2.1부터는 visual c++ 재배포 가능 dll을 사용해서 그렇다고 합니다.

```
x = [[1,2]]
y = [[3],[4]]
m = tf.matmul(x, y)
print("hello, {}".format(m))
```

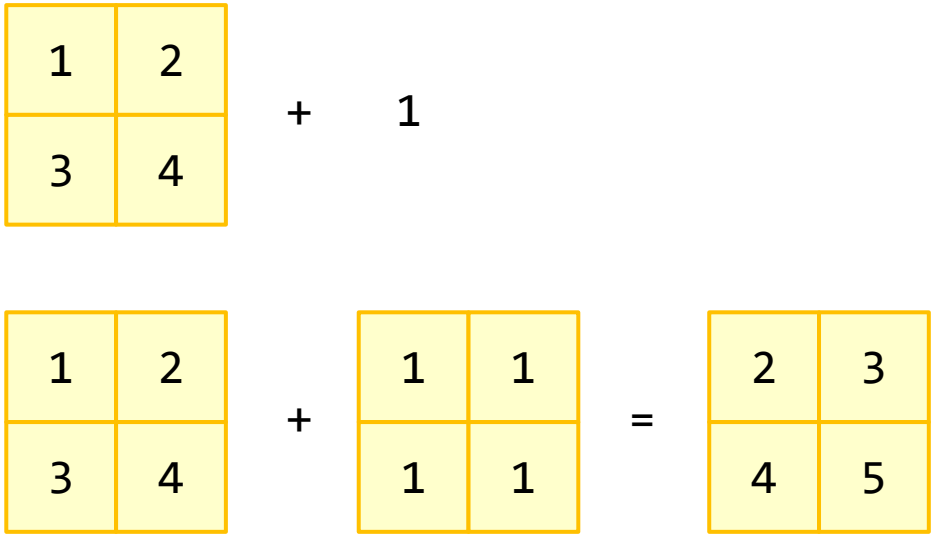


(1,2)(2,1) => (1,1)

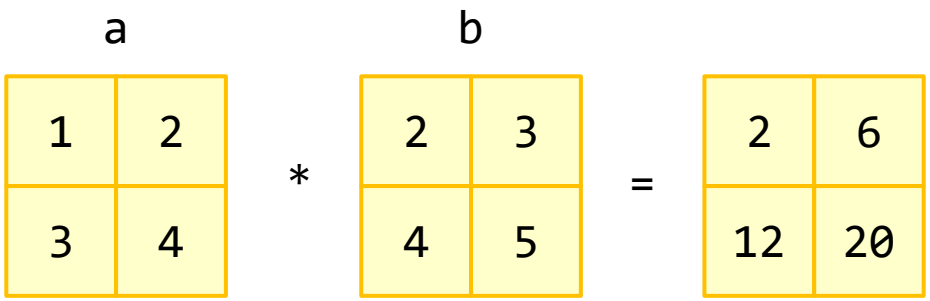
```
a = tf.constant([[1, 2],
                 [3, 4]])
print(a)
```



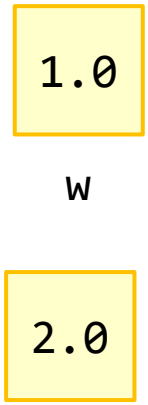
```
b = tf.add(a, 1)
print(b)
```



```
# 연산자 오버로딩 지원
print(a * b)
```



```
w = tf.Variable([[1.0]])
with tf.GradientTape() as tape:
  loss = w * w

grad = tape.gradient(loss, w)
print(grad)
```

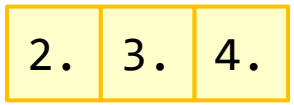


$$loss = w^2 \qquad \frac{\partial loss}{\partial w} = 2w$$

```
rank_0_tensor = tf.constant(4)
print(rank_0_tensor)

# tf.Tensor(4, shape=(), dtype=int32)
```

4 스칼라

```
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])
print(rank_1_tensor)

# tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```



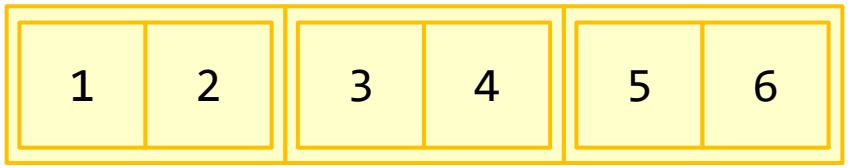
벡터

```
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=tf.float16)
print(rank_2_tensor)

# tf.Tensor(
[[1. 2.]
 [3. 4.]
 [5. 6.]], shape=(3, 2), dtype=float16)
```

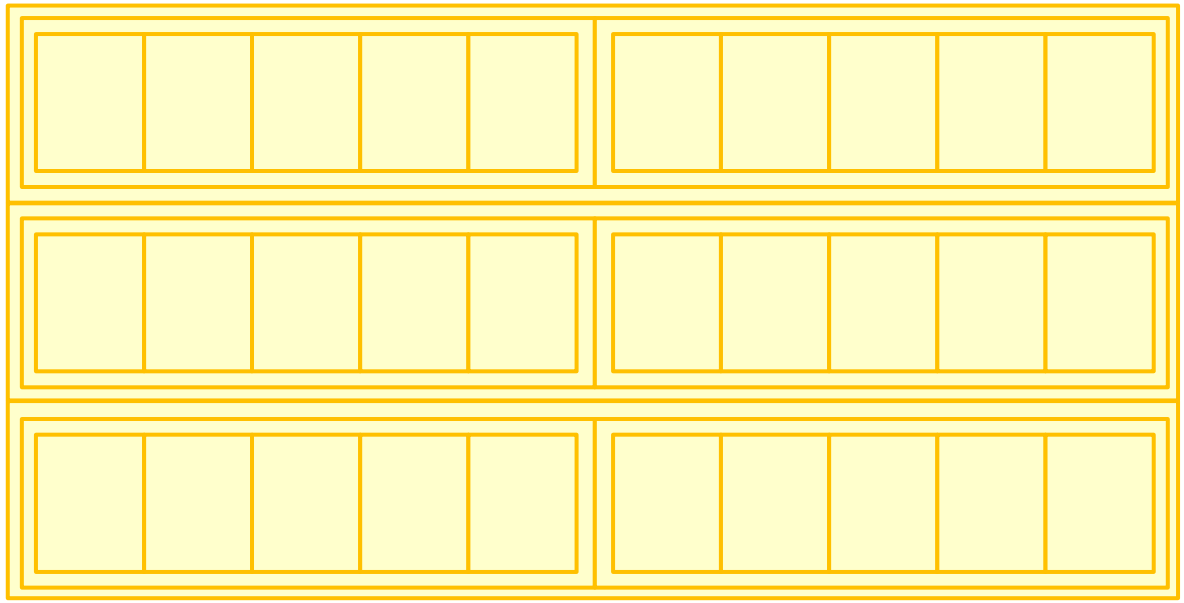
1.	2.
3.	4.
5.	6.

행렬

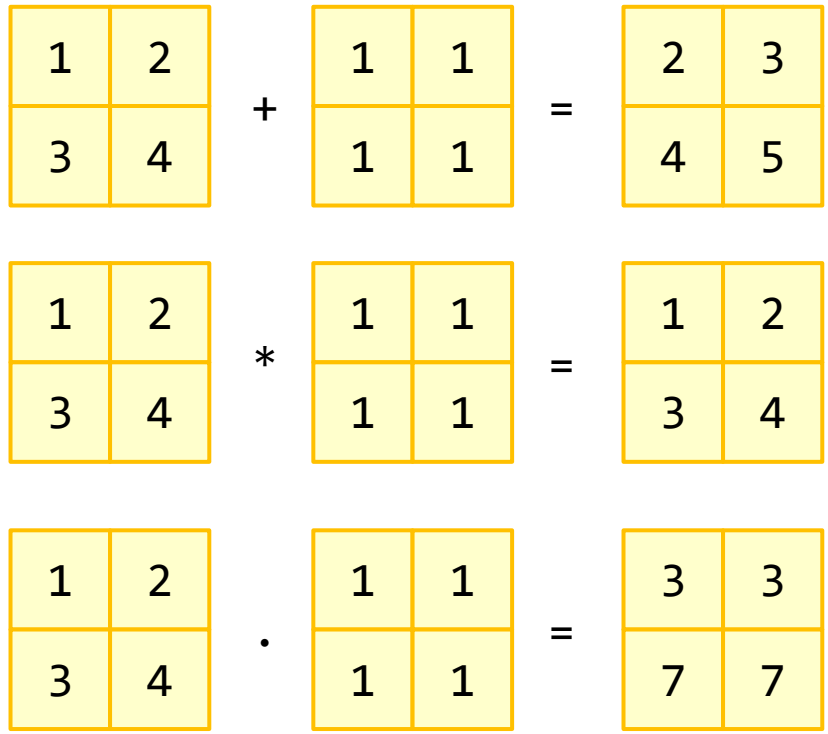
(3, 2)



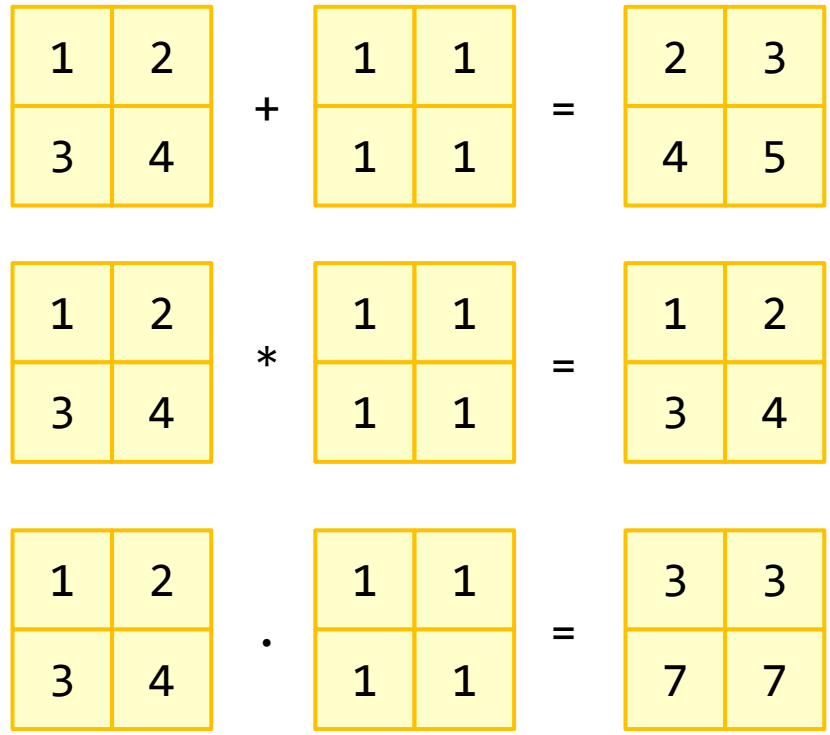
```
rank_3_tensor = tf.constant([
  [[0, 1, 2, 3, 4],
   [5, 6, 7, 8, 9]],
  [[10, 11, 12, 13, 14],
   [15, 16, 17, 18, 19]],
  [[20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29]],])

print(rank_3_tensor)

# shape=(3, 2, 5)
```



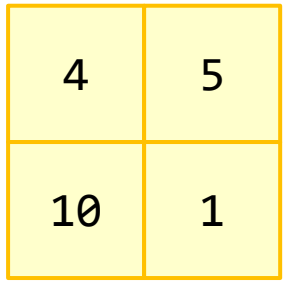
```
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[1, 1],
                 [1, 1]])

print(tf.add(a, b), "\n")
print(tf.multiply(a, b), "\n")
print(tf.matmul(a, b), "\n")
```



```
print(a + b, "\n")
print(a * b, "\n")
print(a @ b, "\n")
```

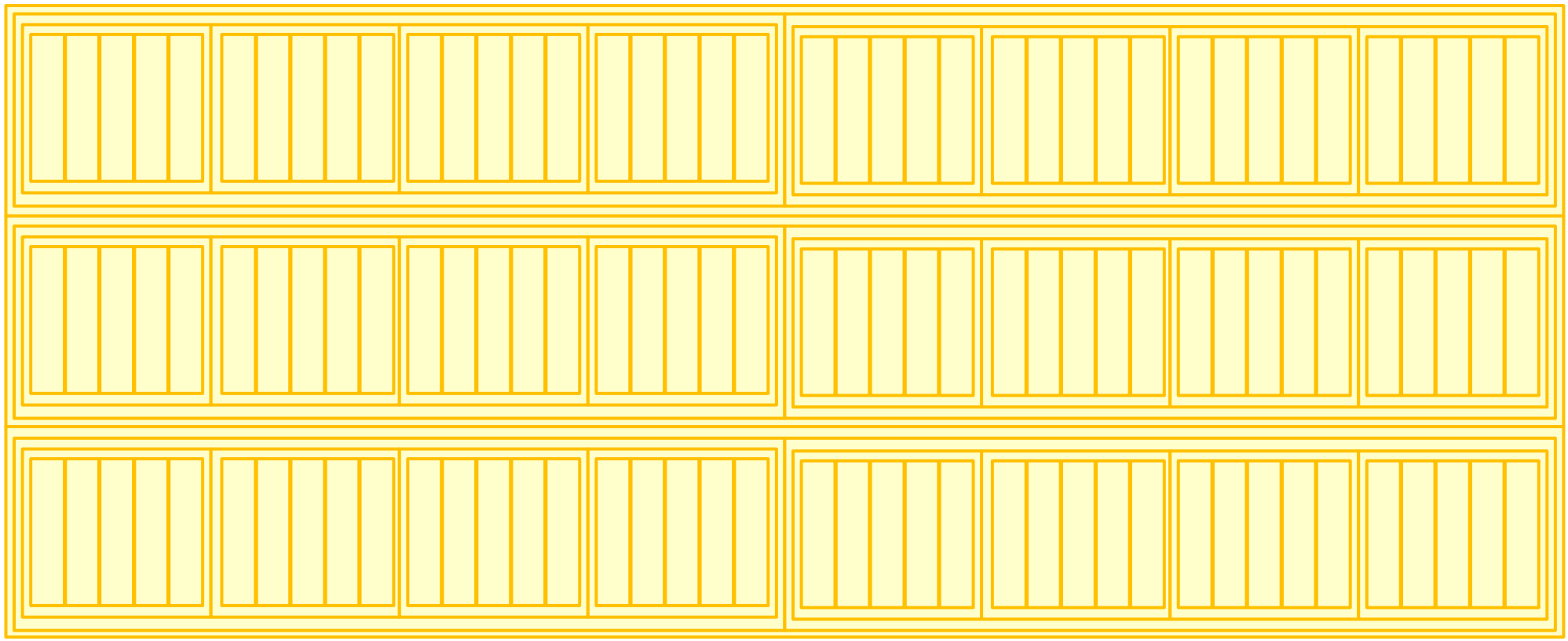


```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])

print(tf.reduce_max(c))  # 10
print(tf.argmax(c))      # [1 0]
print(tf.nn.softmax(c))
```



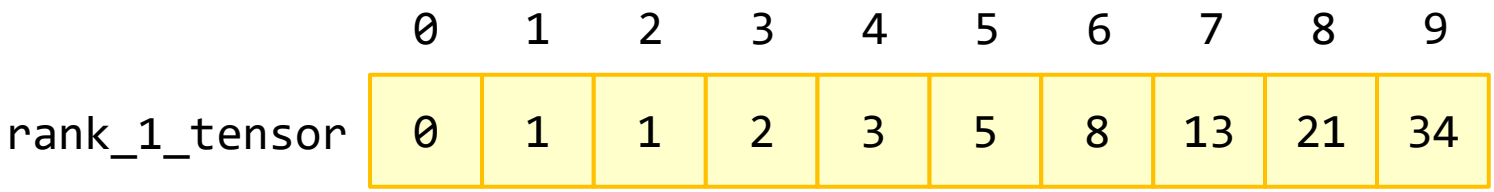

```
tf.Tensor(10.0, shape=(), dtype=float32)
tf.Tensor([1 0], shape=(2,), dtype=int64)
tf.Tensor(
[[2.6894143e-01 7.3105854e-01]
 [9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```

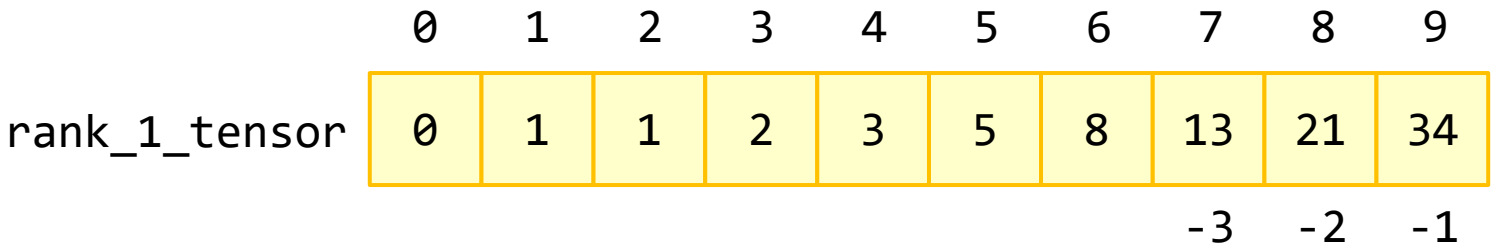


```
rank_4_tensor = tf.zeros([3, 2, 4, 5])

print("Type of every element:", rank_4_tensor.dtype)  # float32
print("Number of dimensions:", rank_4_tensor.ndim)    # 4
print("Shape of tensor:", rank_4_tensor.shape)        # (3, 2, 4, 5)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0]) # 3
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1]) # 5
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy()) # 12
```

```
rank_1_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
print(rank_1_tensor.numpy())
```



```
print("Everything:", rank_1_tensor[:].numpy())
print("Before 4:", rank_1_tensor[:4].numpy())  # 0 1 1 2
print("From 4 to the end:", rank_1_tensor[4:].numpy()) # 3 5 8 13 21 34
print("From 2, before 7:", rank_1_tensor[2:7].numpy()) # 1 2 3 5 8
print("Every other item:", rank_1_tensor[::2].numpy()) # 0 1 3 8 21
print("Reversed:", rank_1_tensor[::-1].numpy()) #
```

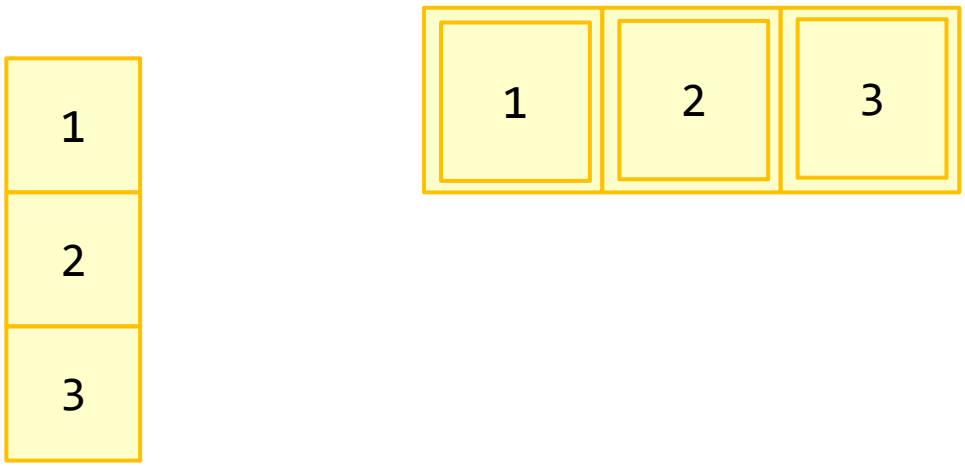


```
print(rank_2_tensor.numpy())  # 4.


[[1. 2.]
 [3. 4.]
 [5. 6.]]

print(rank_2_tensor[1, 1].numpy())
```

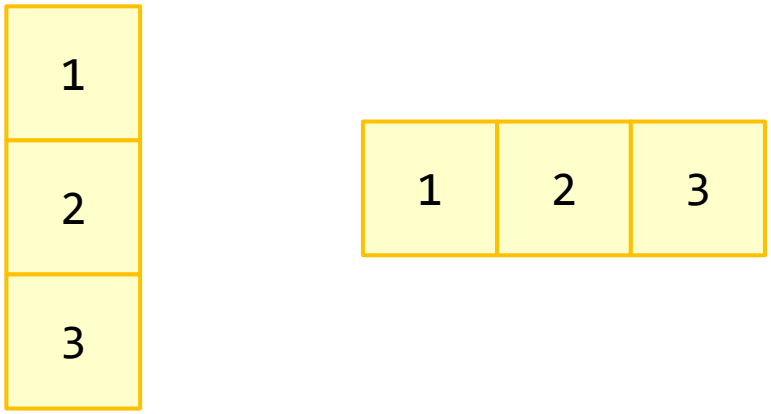
```
print(rank_2_tensor.numpy())  # 4.

[[1. 2.]
 [3. 4.]
 [5. 6.]]

print("Second row:", rank_2_tensor[1, :].numpy())  # [3. 4.]
print("Second column:", rank_2_tensor[:, 1].numpy()) # [2 4 6]
print("Last row:", rank_2_tensor[-1, :].numpy())  # [5. 6.]
print("First item in last column:", rank_2_tensor[0, -1].numpy()) # 2
print("Skip the first row:")
print(rank_2_tensor[1:, :].numpy(), "\n")
# [[3. 4.]
   [5. 6.]]
```

```
print(rank_3_tensor)

tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)

print(rank_3_tensor[:, :, 4])

# [[ 4  9]
   [14 19]
   [24 29]]
```

```
var_x = tf.Variable(tf.constant([[1], [2], [3]]))
print(var_x.shape)  # ( 3, 1 )
```



```
reshaped = tf.reshape(var_x, [1, 3])
print(var_x.shape)
print(reshaped.shape)
```



```
print(rank_3_tensor)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]]
```

(3, 2, 5)

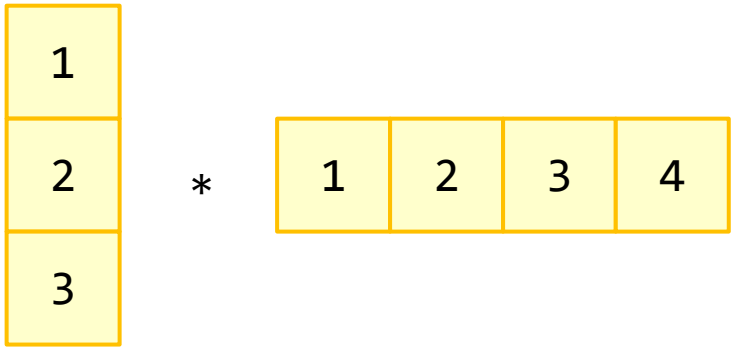
```
print(rank_3_tensor)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]]
```

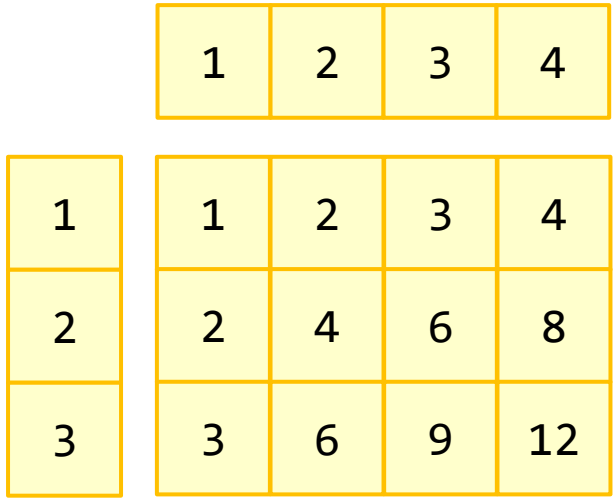
(3, 2, 5) => (30,)

```
print(tf.reshape(rank_3_tensor, [-1]))
```

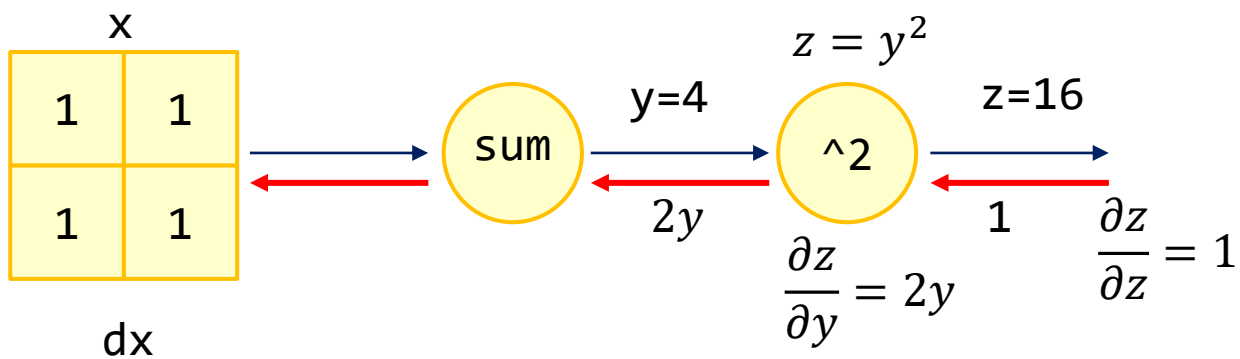
```
print(tf.reshape(rank_3_tensor, [2, 3, 5]), "\n")
                                (3, 2, 5) => (2, 3, 5)
print(tf.reshape(rank_3_tensor, [5, 6]), "\n")

                                (3, 2, 5) => (5, 6)
try:
  tf.reshape(rank_3_tensor, [7, -1])
except Exception as e:
  print(f"{type(e).__name__}: {e}")
```

```
x = tf.reshape(x,[3,1])
y = tf.range(1, 5)
print(x, "\n")
print(y, "\n")
print(tf.multiply(x, y))
```





```
x = tf.ones((2, 2))
with tf.GradientTape() as t:
  t.watch(x)
  y = tf.reduce_sum(x)
  z = tf.multiply(y, y)
```



dx

8	8
8	8

$\frac{\partial z}{\partial x}$

sum 의 미분은 repeat 이다.

```
x = tf.constant(3.0)
with tf.GradientTape(persistent=True) as t:
  t.watch(x)
  y = x * x
  z = y * y
dz_dx = t.gradient(z, x)  # 108.0 (4*x^3 at x = 3)
dy_dx = t.gradient(y, x)  # 6.0
del t
```

$$y = x^2 \qquad \frac{\partial y}{\partial x} = 2x^1$$

$$z = y^2 \qquad z = x^4 \qquad \frac{\partial z}{\partial x} = 4x^3$$

```
def f(x, y):
  output = 1.0
  for i in range(y):
    if i > 1 and i < 5:
      output = tf.multiply(output, x)
  return output

def grad(x, y):
  with tf.GradientTape() as t:
    t.watch(x)
    out = f(x, y)
  return t.gradient(out, x)

x = tf.convert_to_tensor(2.0)

assert grad(x, 6).numpy() == 12.0
assert grad(x, 5).numpy() == 12.0
assert grad(x, 4).numpy() == 4.0
```

```
output = 1*x

output = x*x

output = x*x*x
```

$$y = x^3$$

$$\frac{\partial y}{\partial x} = 3x^2$$

$$y = x^2$$

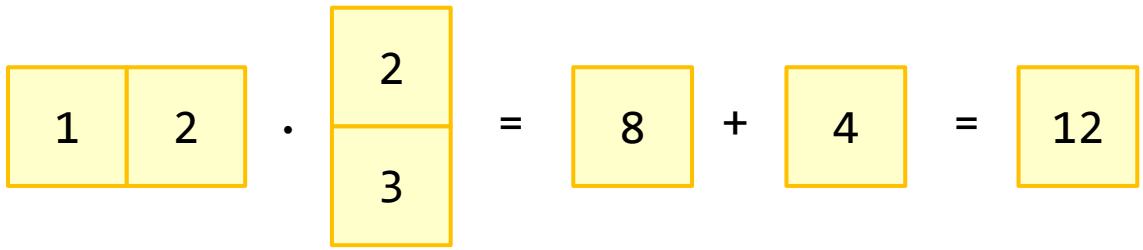
$$\frac{\partial y}{\partial x} = 2x^1$$

```
x = tf.Variable(1.0)  # 1.0으로 초기화된 텐서플로 변수를 생성합니다.

with tf.GradientTape() as t:
  with tf.GradientTape() as t2:
    y = x * x * x
  # 't' 컨텍스트 매니저 안의 그래디언트를 계산합니다.
  # 이것은 또한 그래디언트 연산 자체도 미분가능하다는 것을 의미합니다.
  dy_dx = t2.gradient(y, x)
d2y_dx2 = t.gradient(dy_dx, x)

assert dy_dx.numpy() == 3.0
assert d2y_dx2.numpy() == 6.0
```

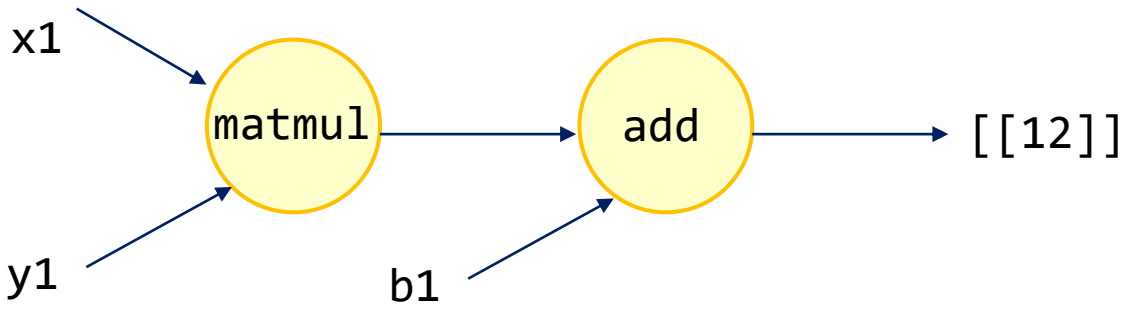
$$\text{dy\_dx} = 3x^2$$

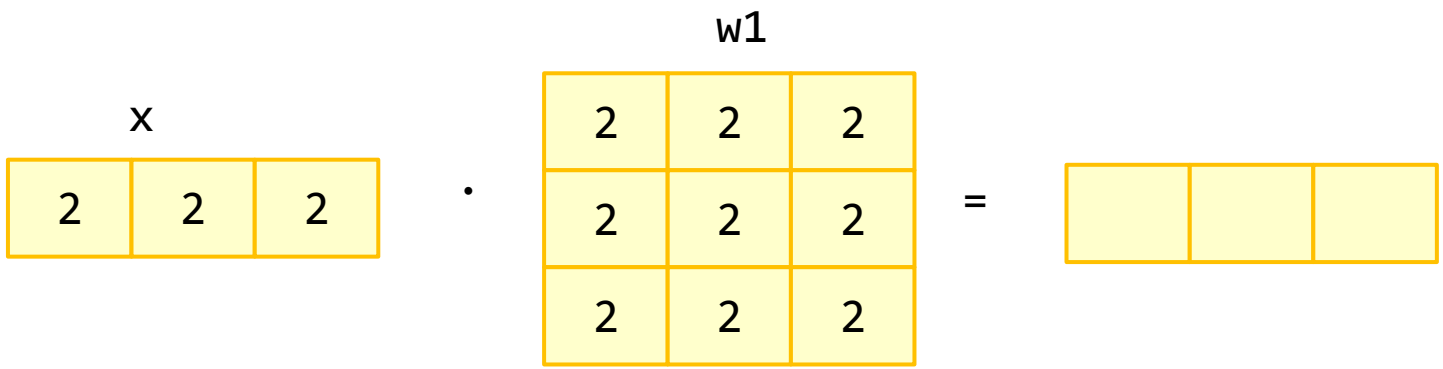
$$y = x^3 \qquad \frac{\partial y}{\partial x} = 3x^2 \qquad \frac{\partial dy\_dx}{\partial x} = 6x^1$$

```
def function_to_get_faster(x, y, b):
  x = tf.matmul(x, y)
  x = x + b
  return x

a_function_that_uses_a_graph = tf.function(function_to_get_faster)

x1 = tf.constant([[1.0, 2.0]])
y1 = tf.constant([[2.0], [3.0]])
b1 = tf.constant(4.0)

print(function_to_get_faster(x1, y1, b1).numpy())
```

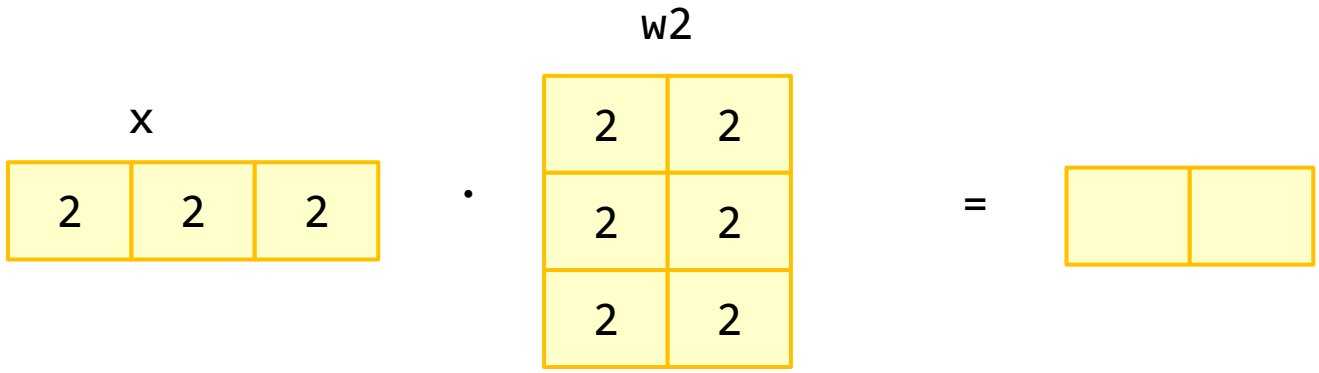


```
def function_to_get_faster(x, y, b):
  x = tf.matmul(x, y)
  x = x + b
  return x

a_function_that_uses_a_graph = tf.function(function_to_get_faster)

x1 = tf.constant([[1.0, 2.0]])
y1 = tf.constant([[2.0], [3.0]])
b1 = tf.constant(4.0)

a_function_that_uses_a_graph(x1, y1, b1).numpy()
```





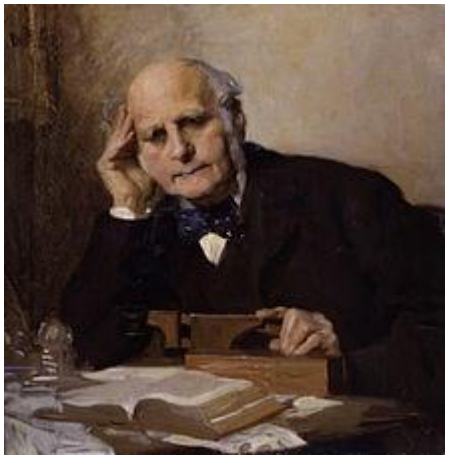
(1,3)(3,3)=>(1,3)



(1,3)(3,2)=>(1,2)

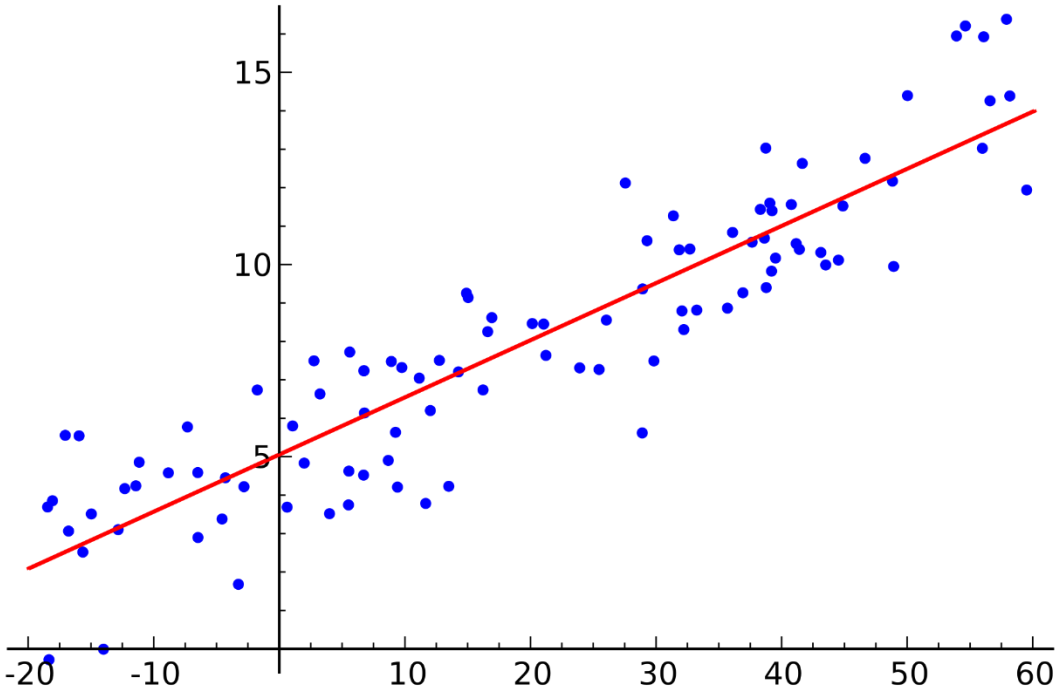
Regression - 회귀

"Regression toward the mean"



Sir Francis Galton
(1822 ~ 1911)

Linear Regression - 선형 회귀



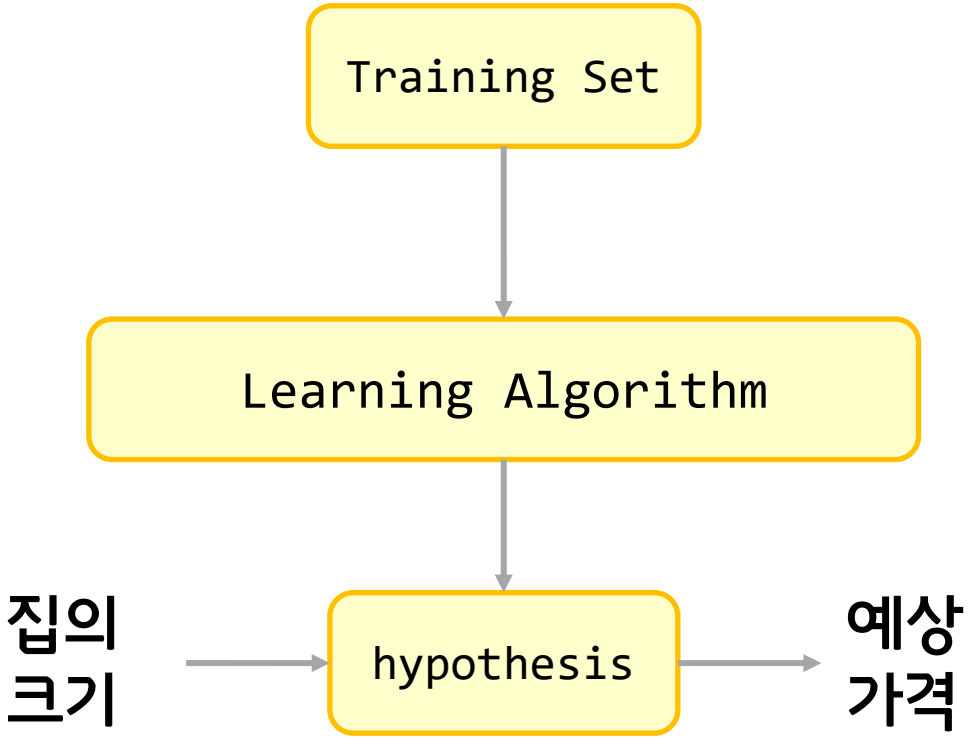
$$y = 2x + 1$$

$$y = ax + b$$

$$\downarrow$$

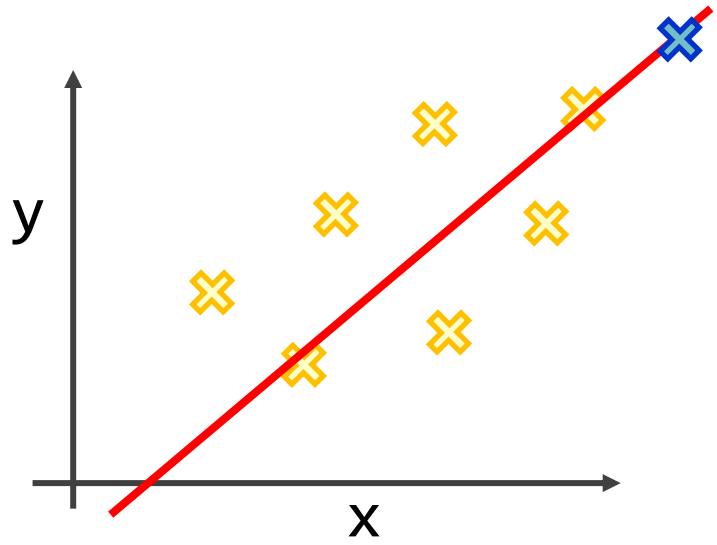
$$y = wx + b$$

https://en.wikipedia.org/wiki/Linear_regression



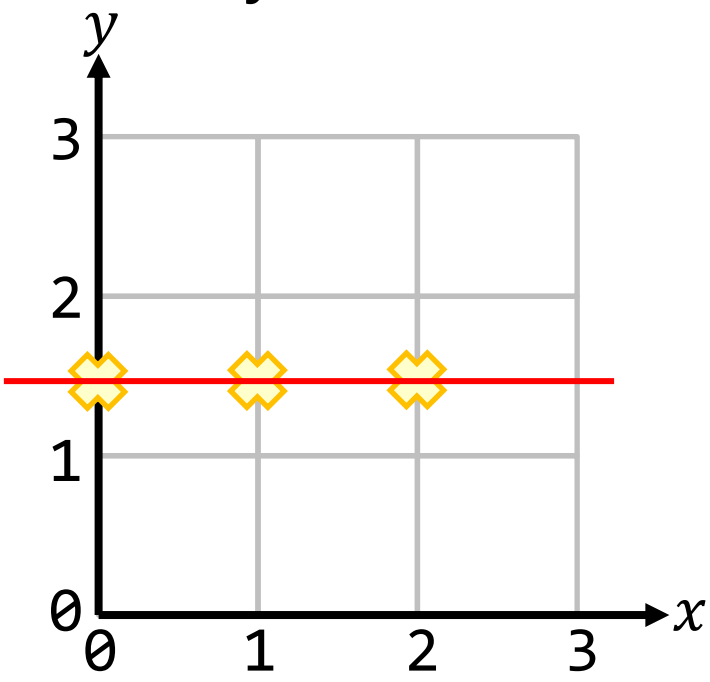
hypothesis ?

$$\hat{y} = wx + b$$

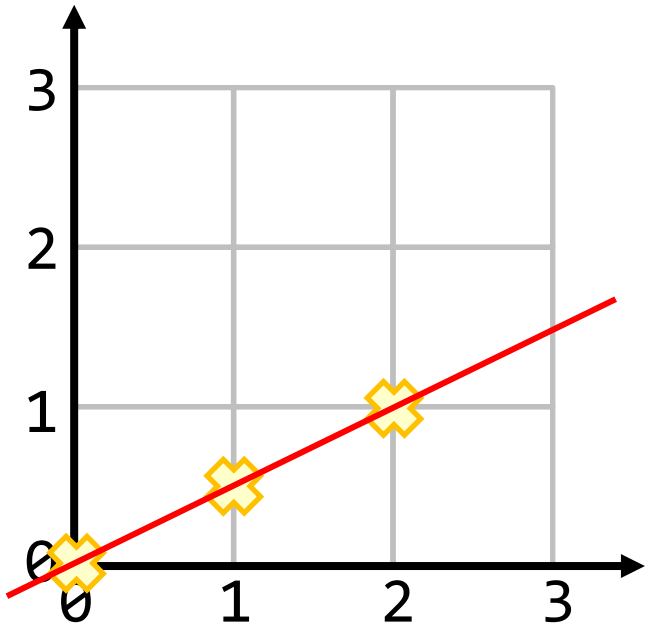


$$\hat{y} = wx + b$$

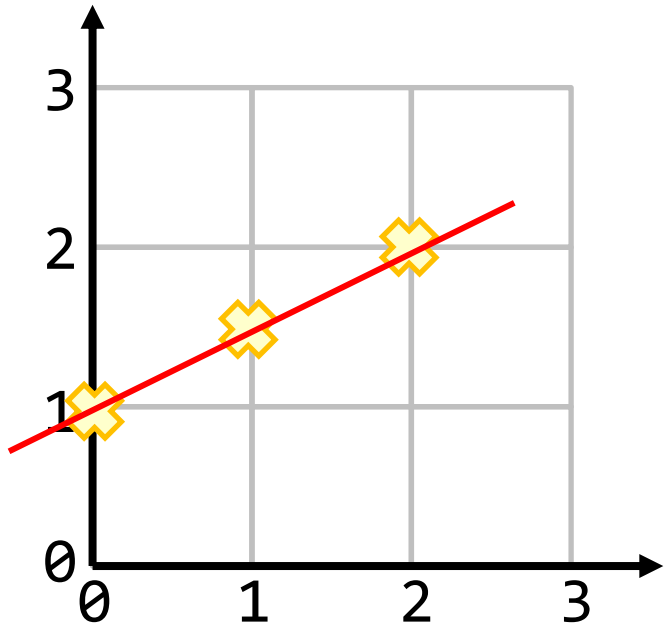
$$\frac{dy}{dx}$$



w = 0
b = 1.5



w = 0.5
b = 0



w = 0.5
b = 1

Hypothesis :

$$\hat{y} = wx + b$$

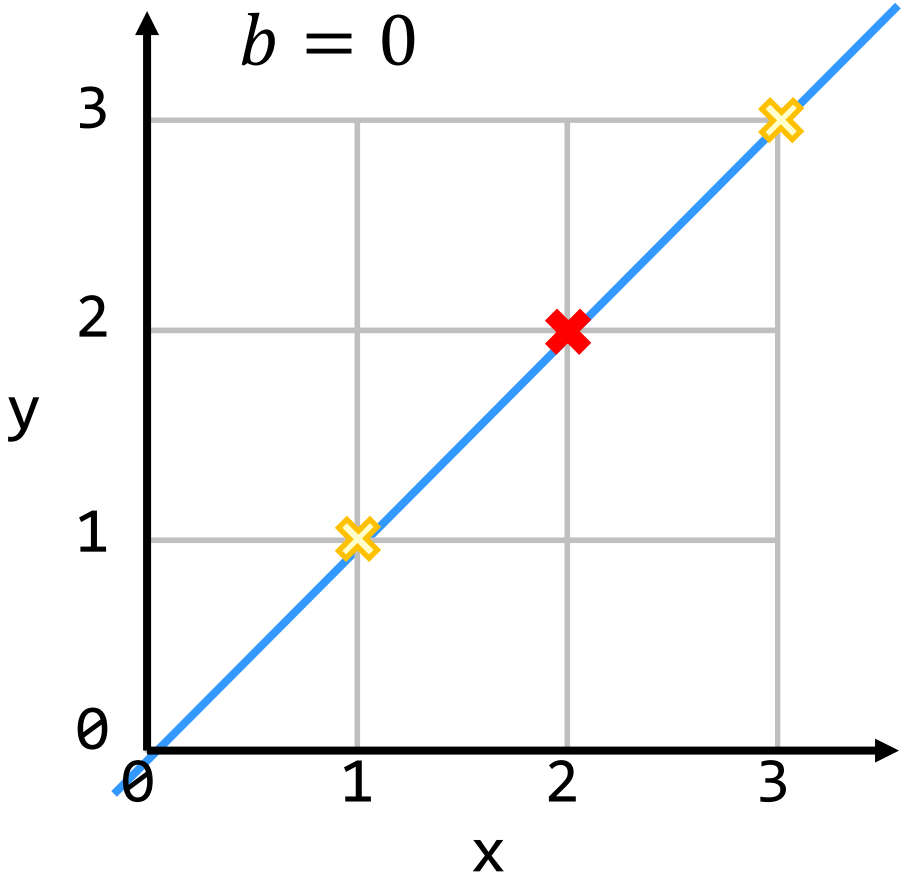
Parameters:

$$w, b$$

Cost Function

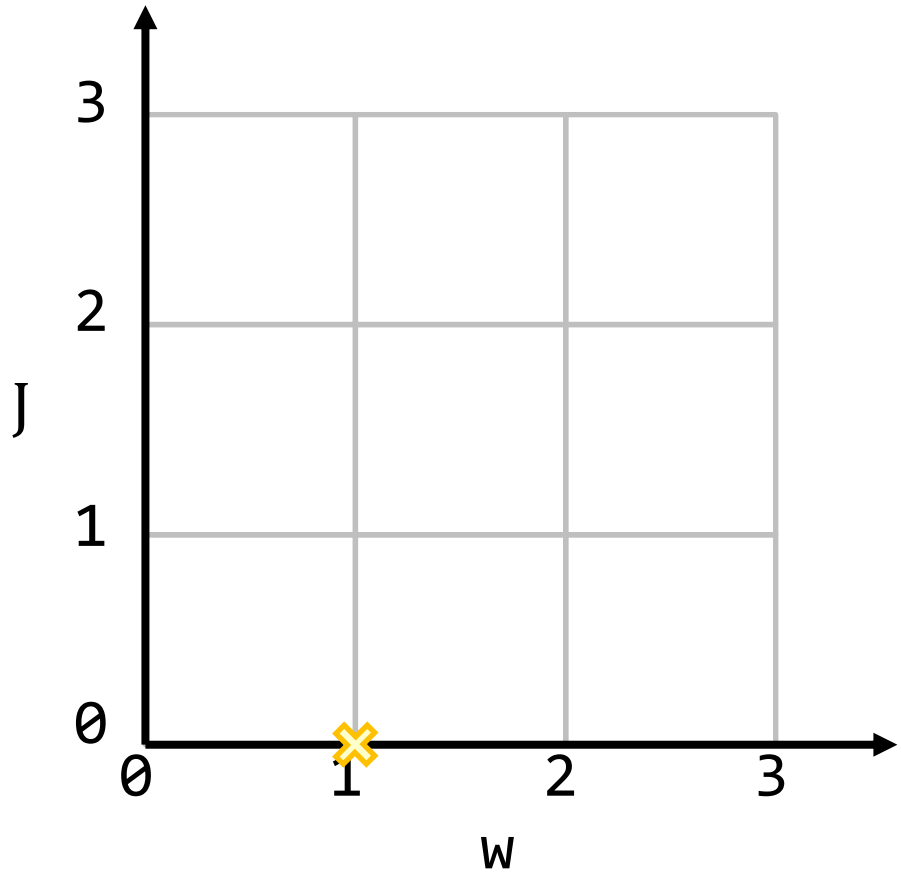
$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

$$\hat{y} = wx + b$$
$$w = 1$$
$$b = 0$$



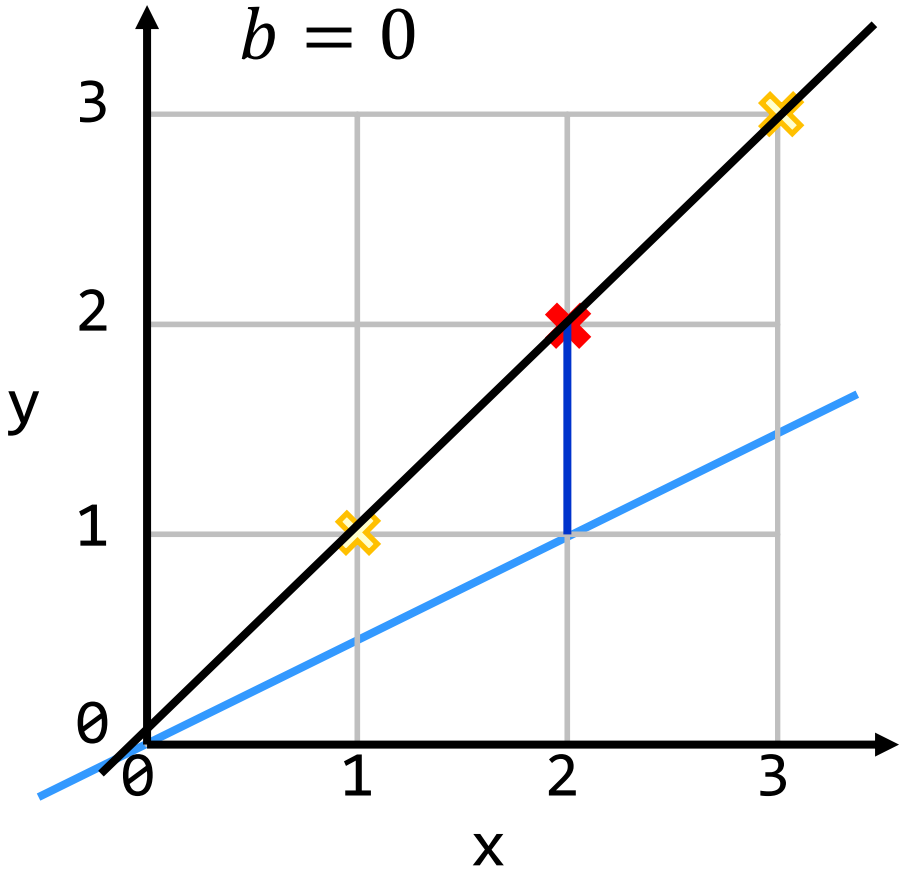

```
x = [1,2,3]
y = [1,2,3]
```

$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

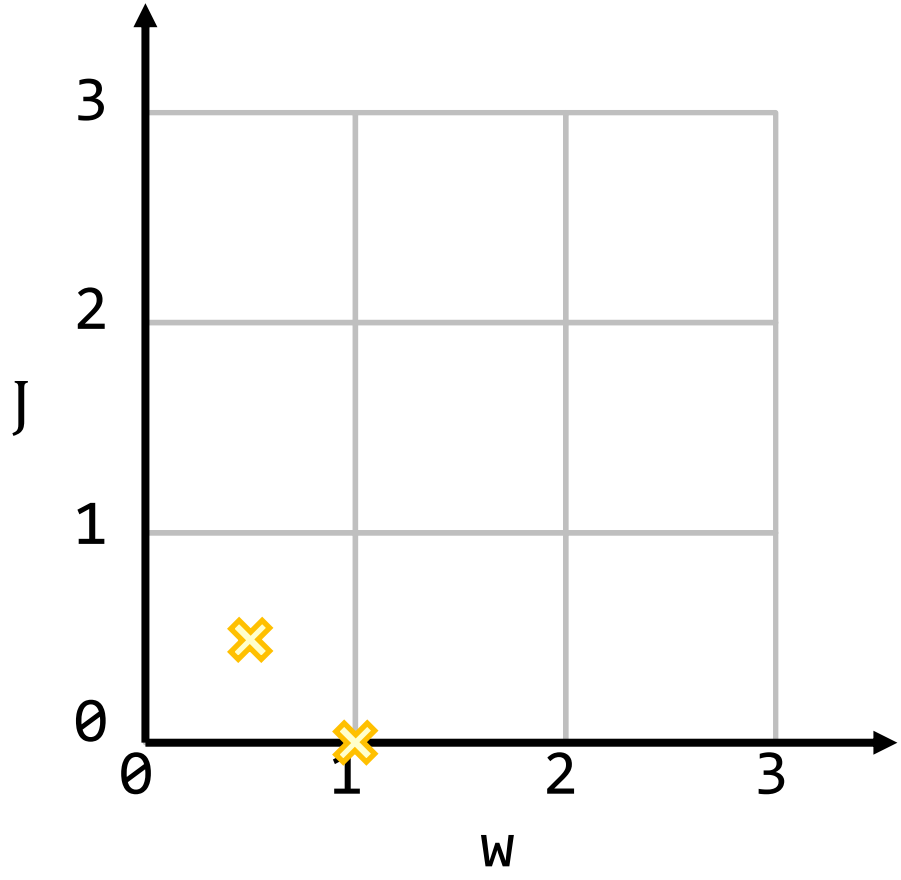



```
x      = [1,2,3]
y_hat  = [1,2,3]
```

$$\hat{y} = wx + b$$
$$w = 0.5$$
$$b = 0$$

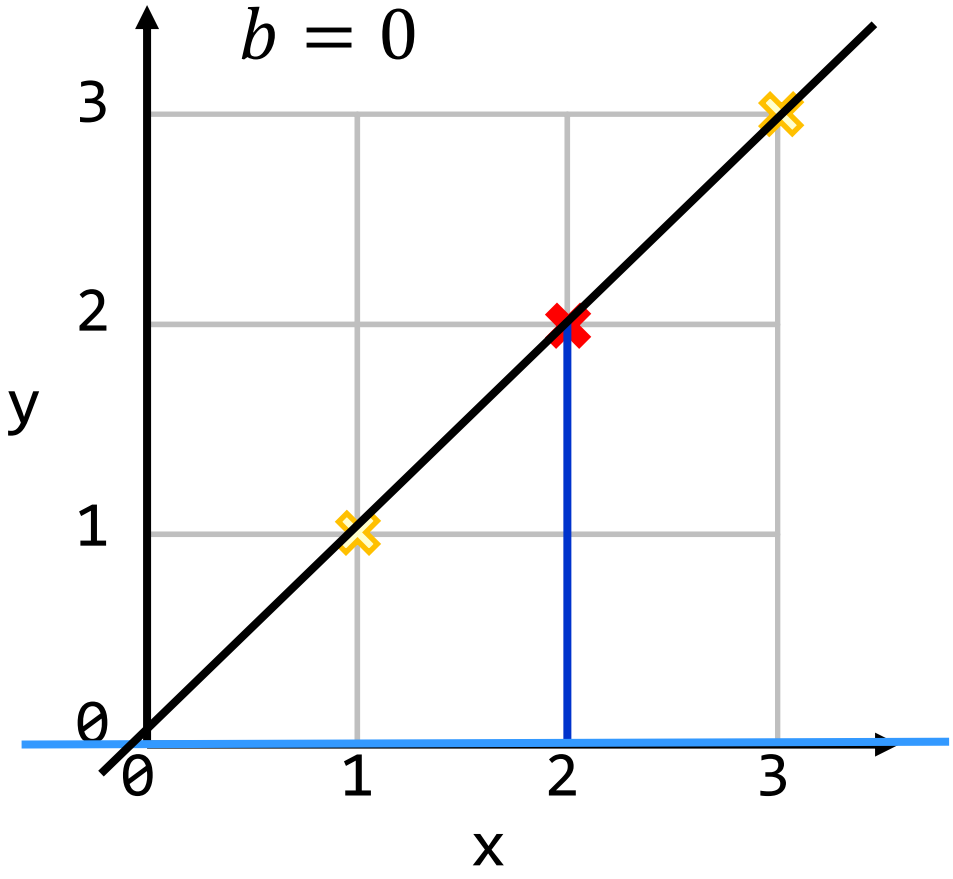


$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$

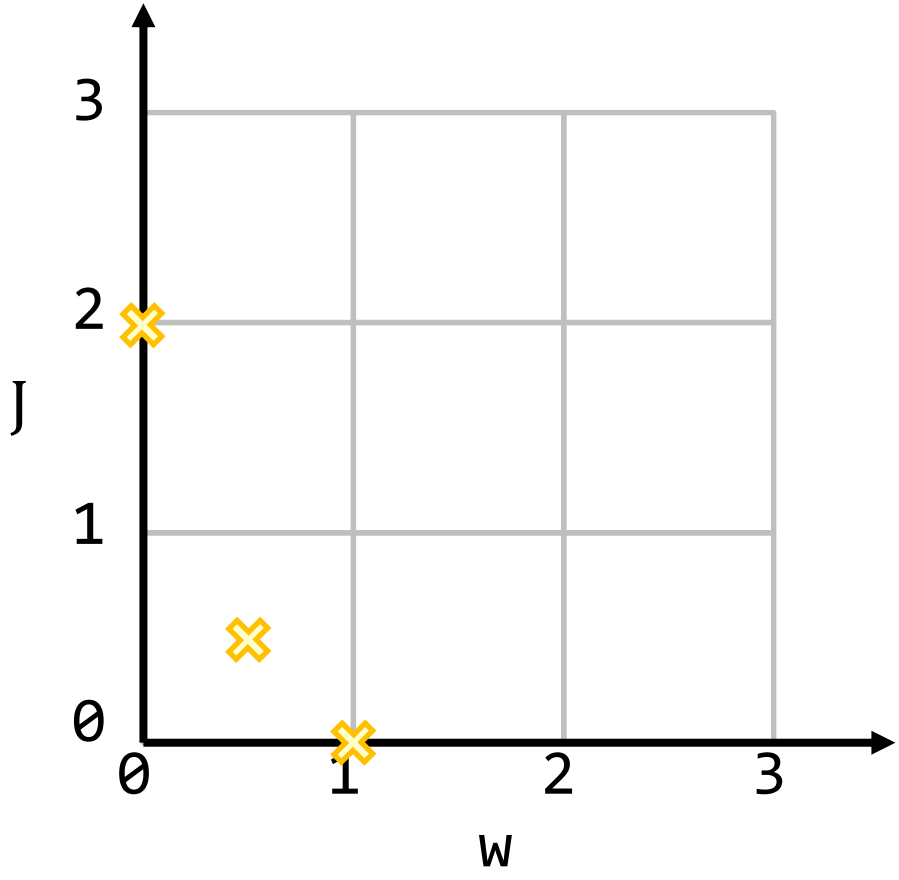



```
x = [1,2,3]
y = [1,2,3]
```

$$\hat{y} = wx + b$$
$$w = 0$$
$$b = 0$$



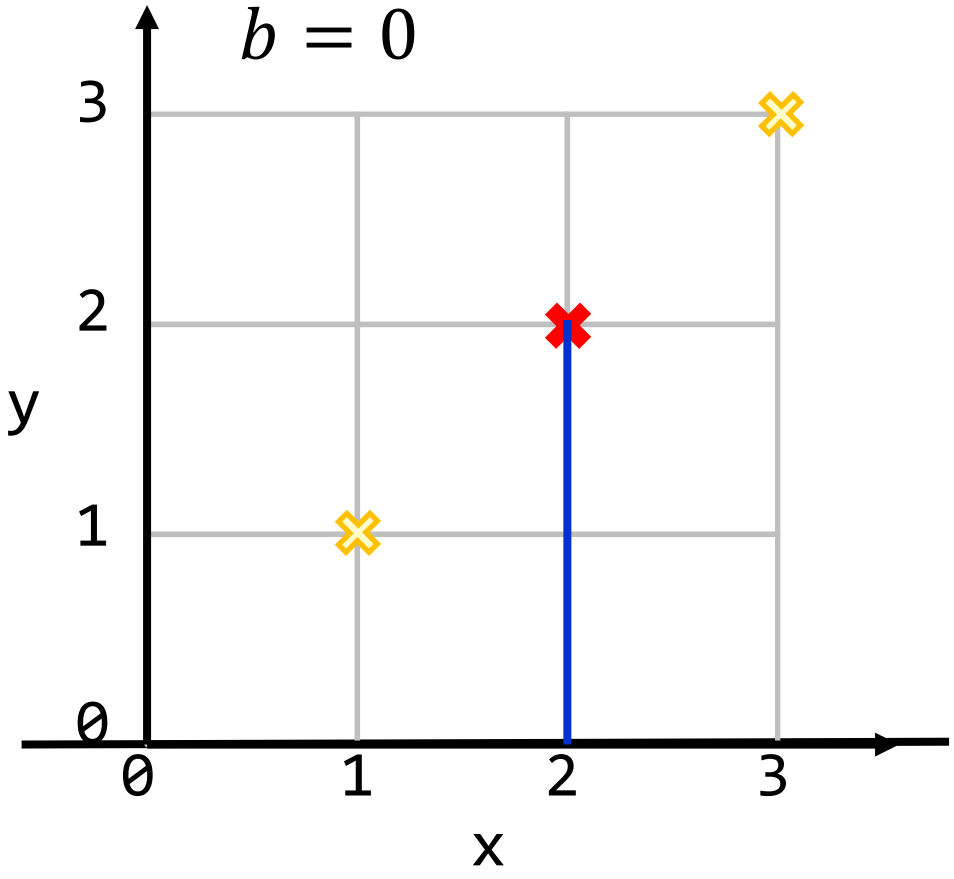
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



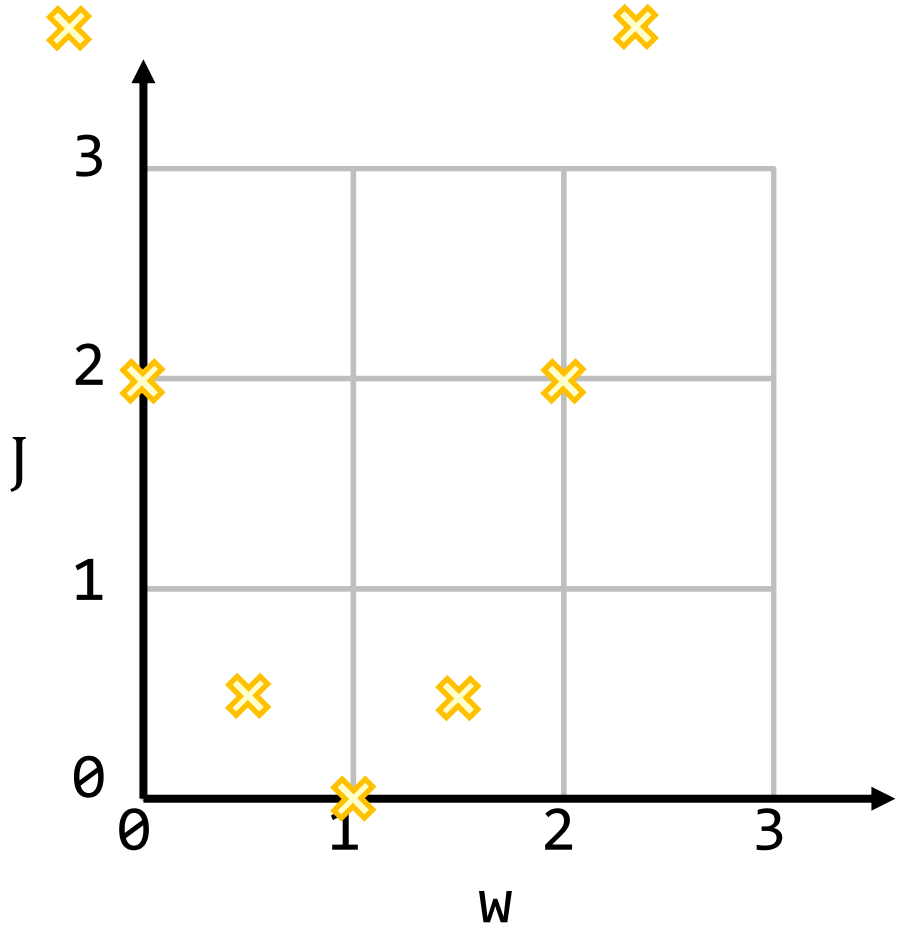
$$\hat{y} = wx + b$$

$$w = 0$$

$$b = 0$$



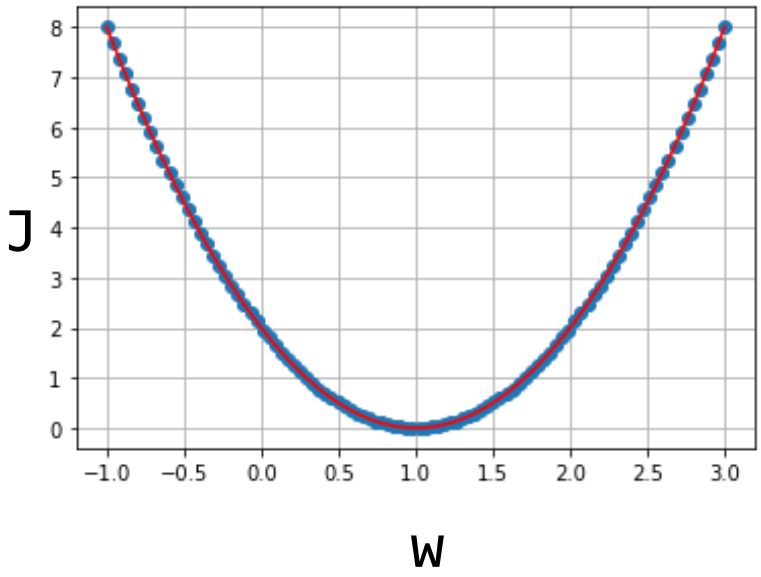
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



Cost function

2.1 MSE(Mean Squared Error)

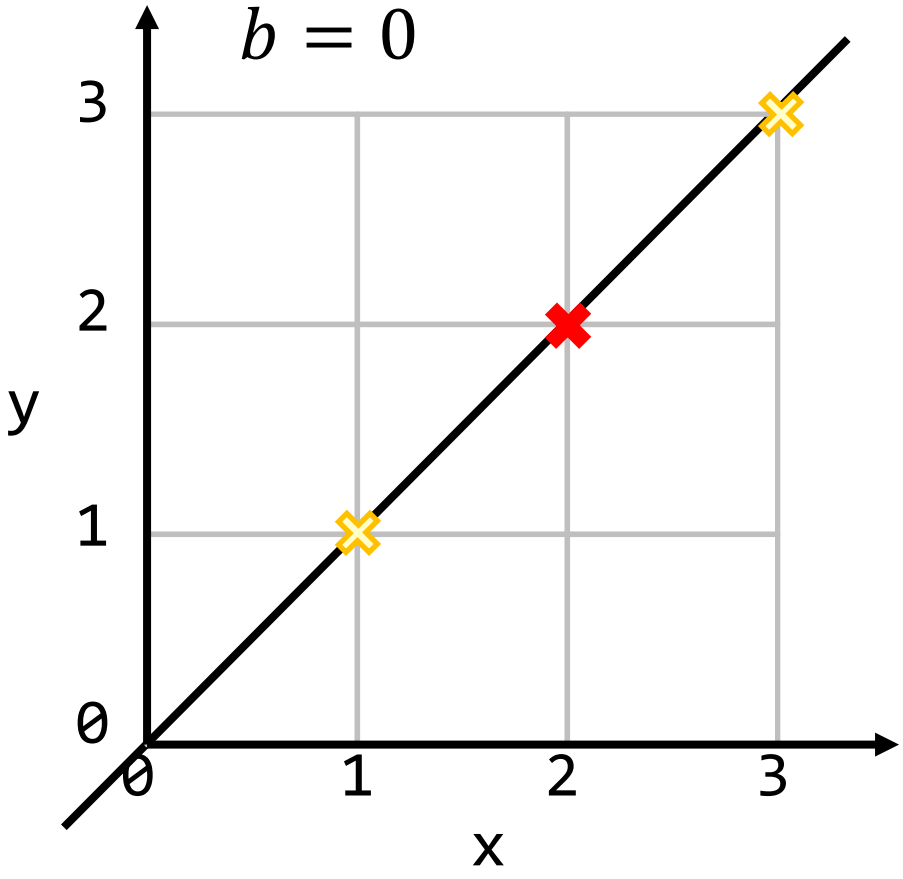
```
import numpy as np   # numeriacal computing
import matplotlib.pyplot as plt  # plotting core

w = np.linspace(-1,3,100)
print(w)
print(len(w))
b = 0

j = np.zeros(100)

for i in range(len(w)):
    y_hat = w[i]*2 + b;
    j[i] = 0.5 * (y_hat - 2)**2

plt.plot(w,j, 'o' )
plt.plot(w,j, 'r-' )

plt.grid(True)
plt.show()
```



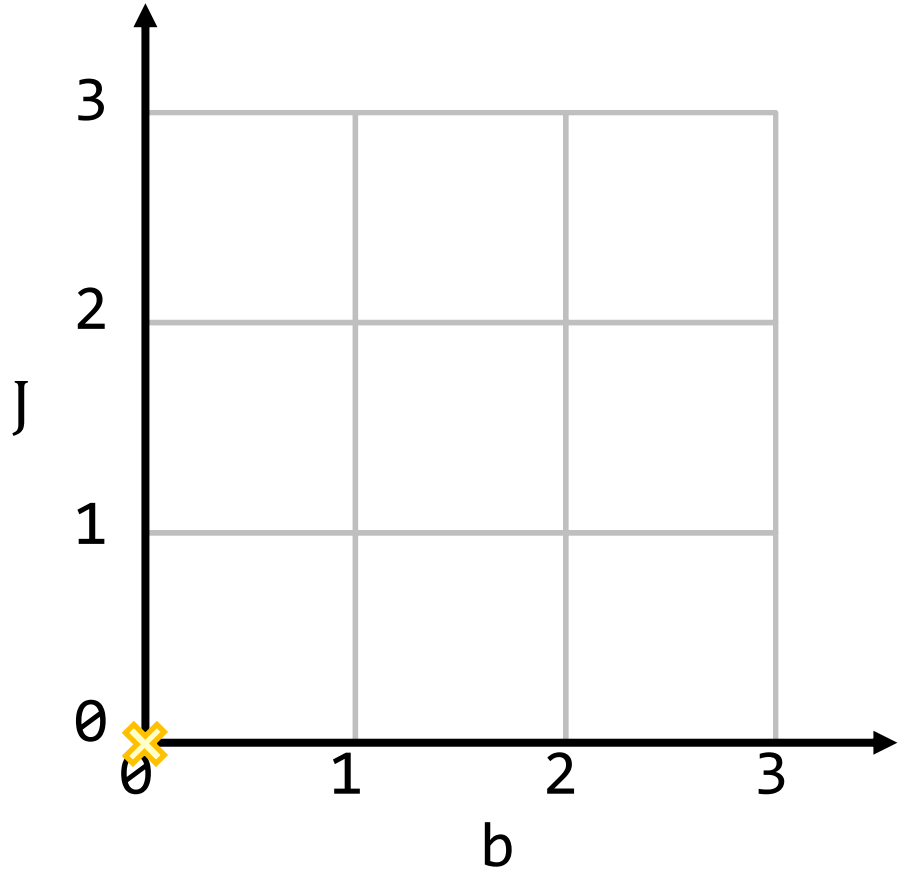
$$\hat{y} = wx + b$$

$$w = 1$$

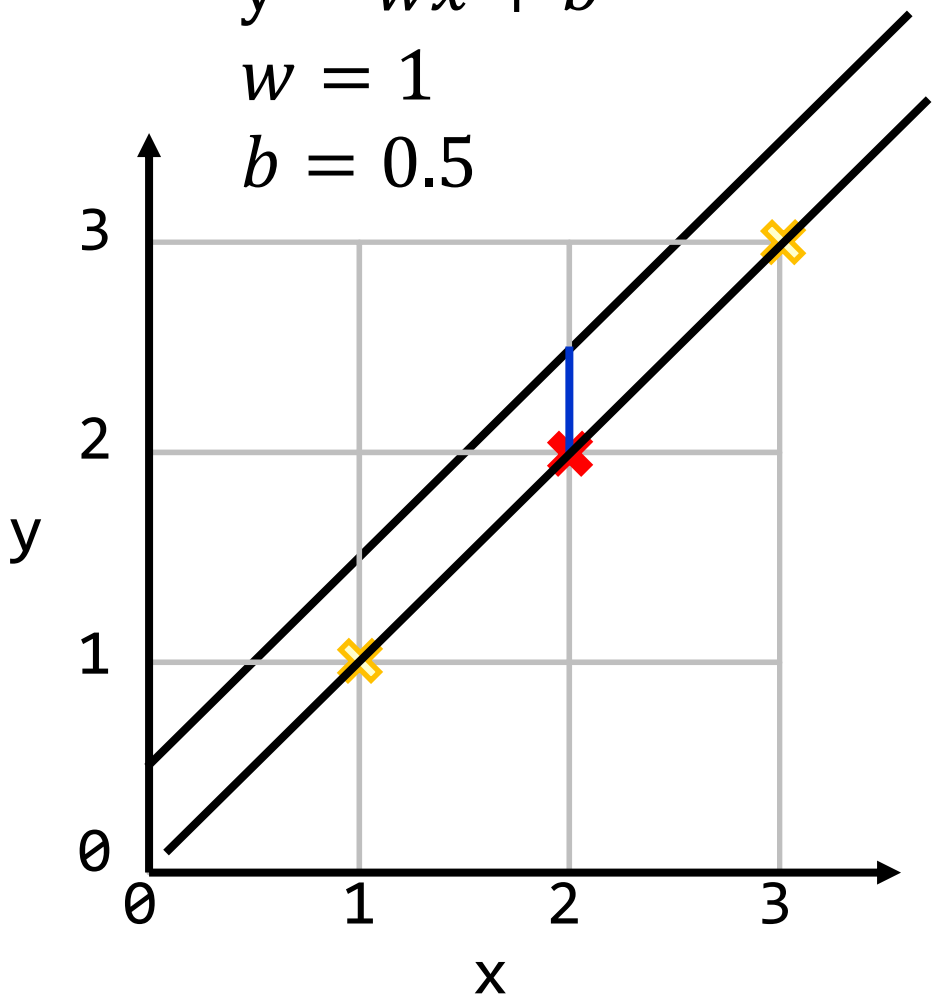
$$b = 0$$



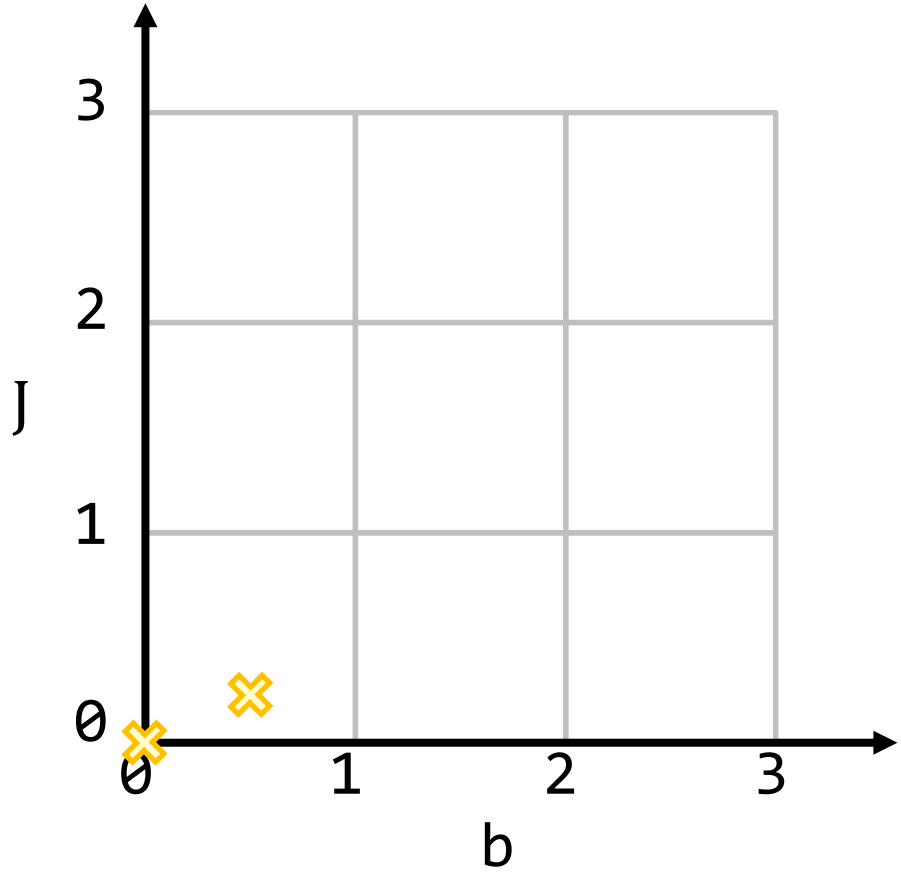
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



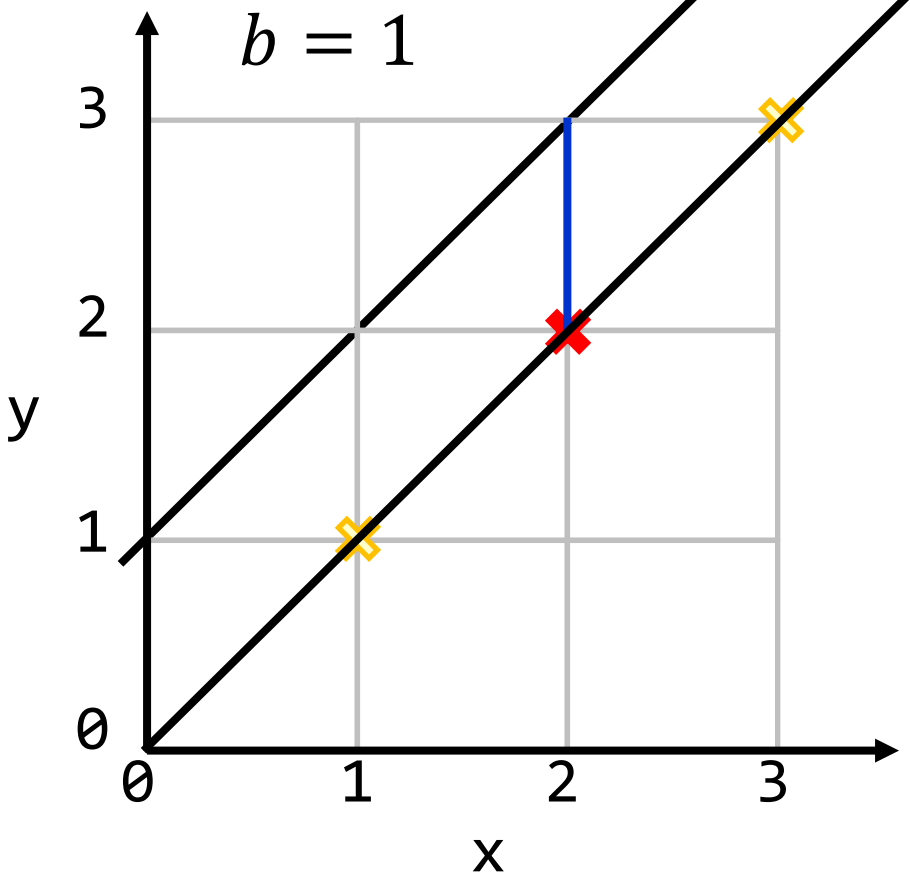
$$\hat{y} = wx + b$$
$$w = 1$$
$$b = 0.5$$



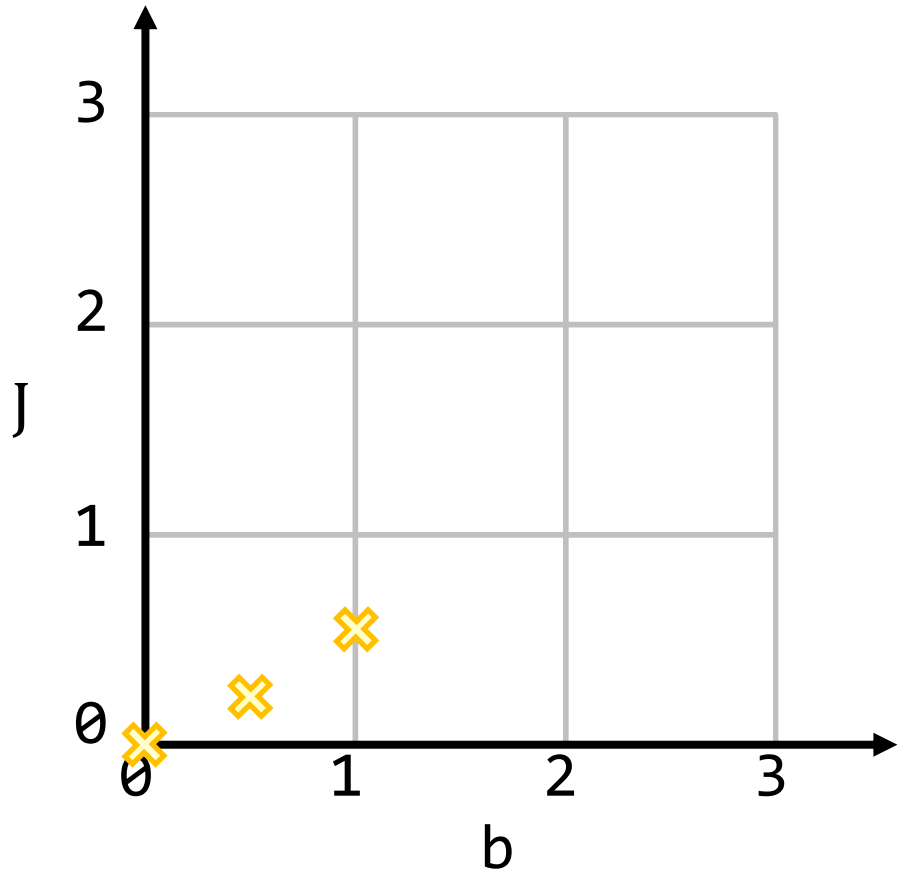
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$

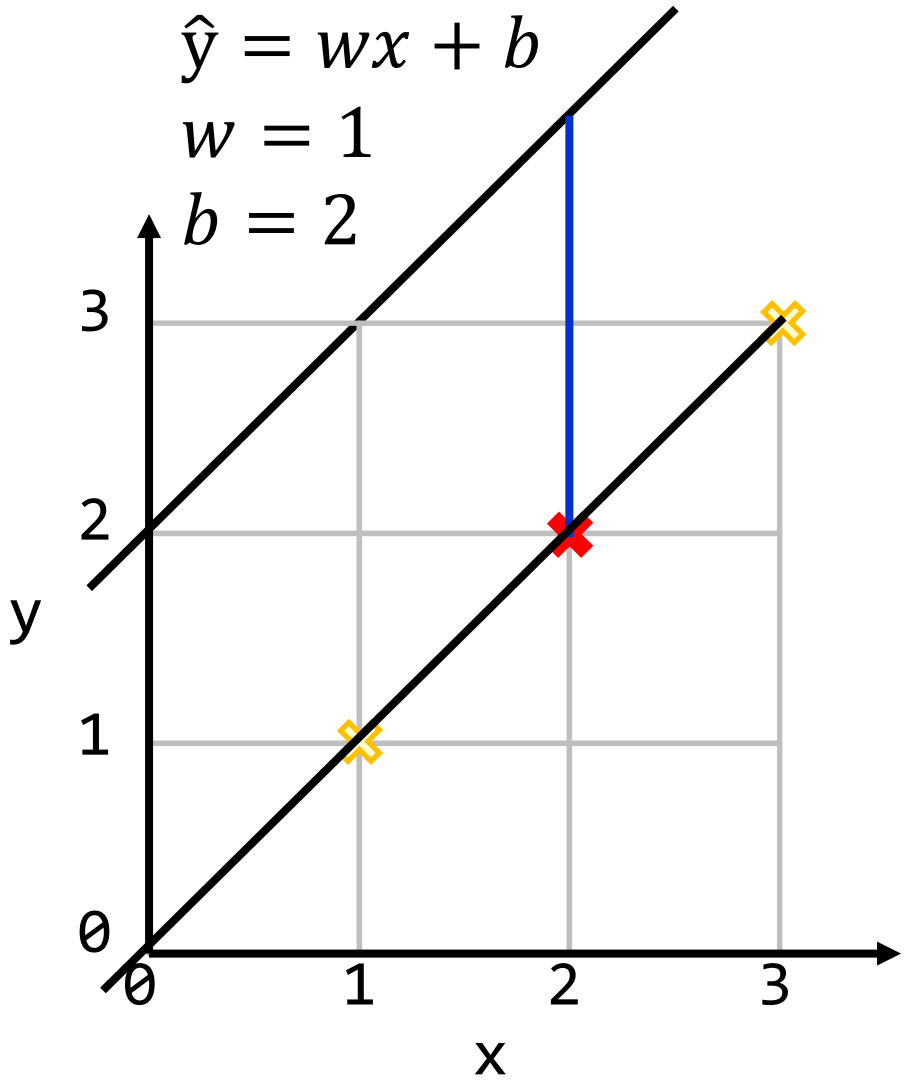


$$\hat{y} = wx + b$$
$$w = 1$$
$$b = 1$$

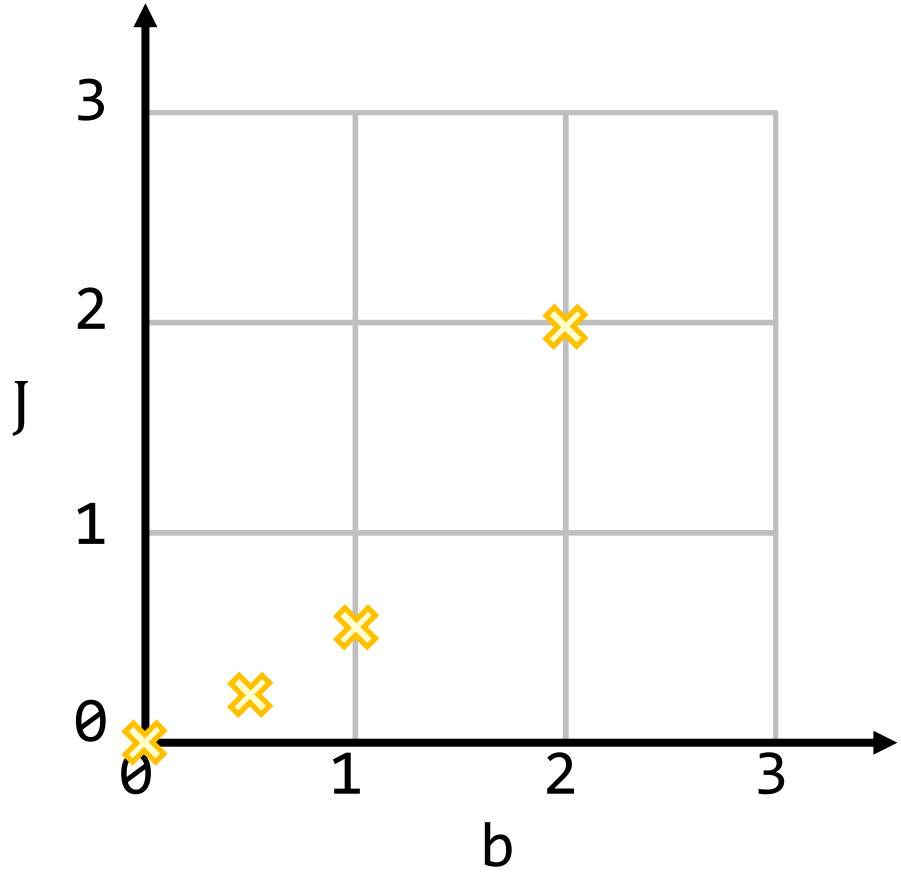


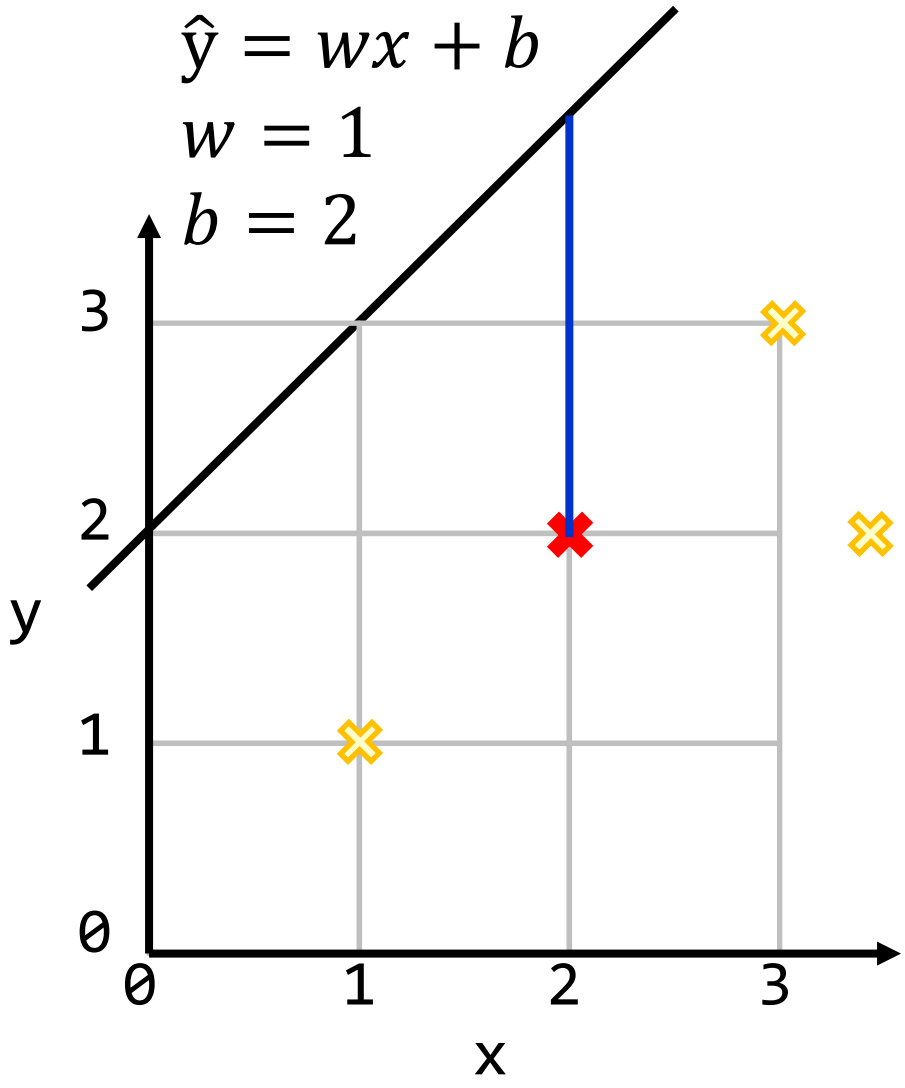
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



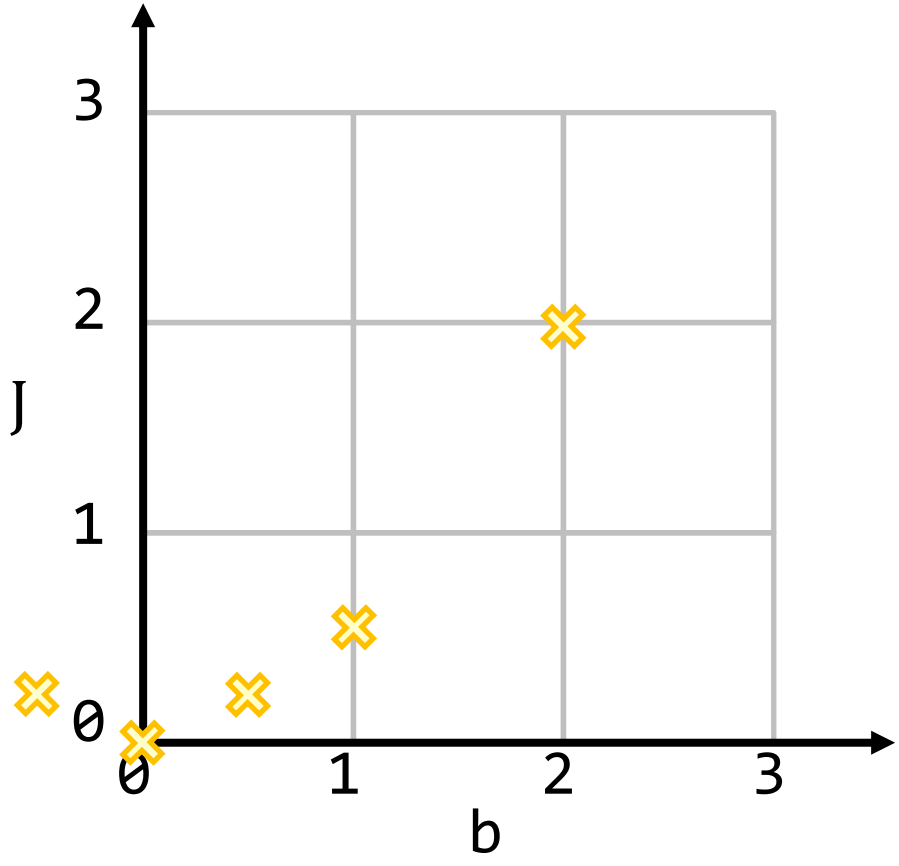


$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$





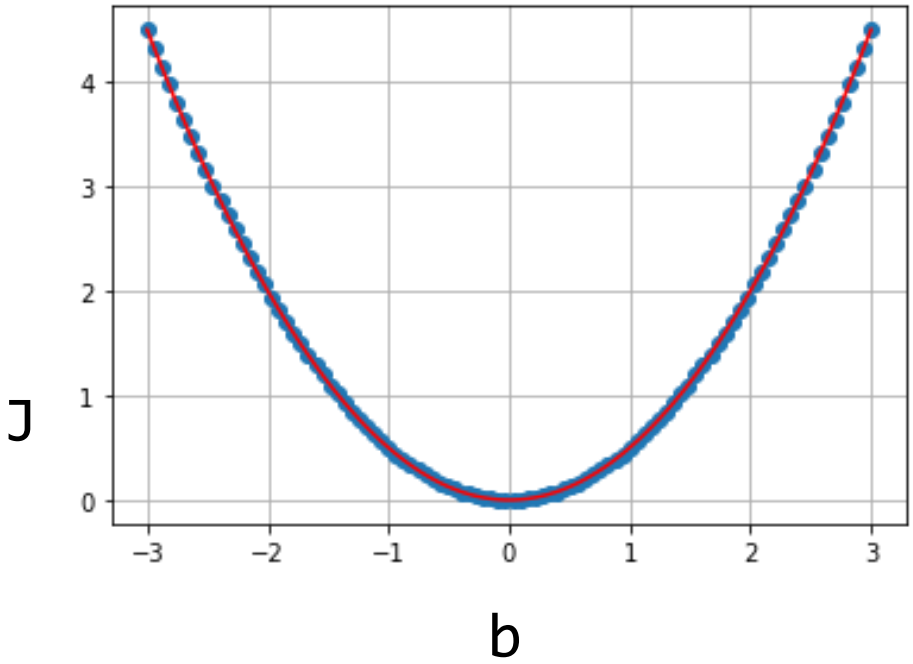
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$

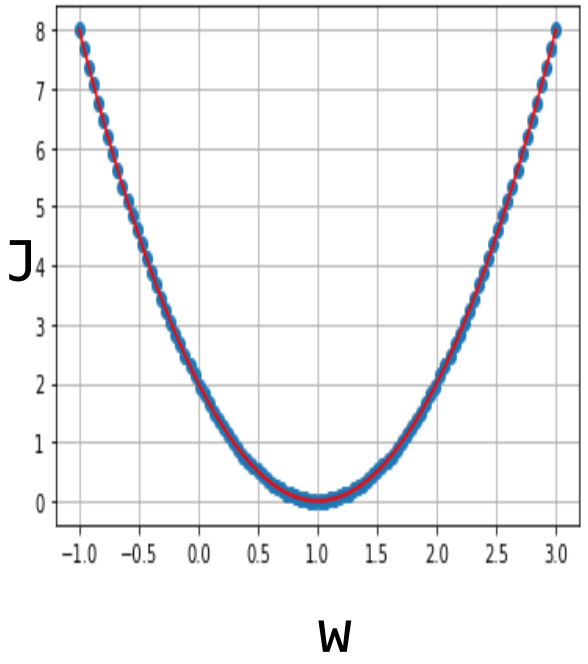


Cost function

2.1 MSE(Mean Squared Error)

$$\hat{y} = wx + b$$





Gradient descent algorithm

- 비용 함수 최소화
- 경사 하강은 많은 최소화 문제에 사용된다.
- 주어진 비용 함수, 비용 (W, b)에 대해 비용을 최소화하기 위해 W, b를 찾는다.
- 일반적인 함수 : 비용 (w1, w2,...)에 적용 가능

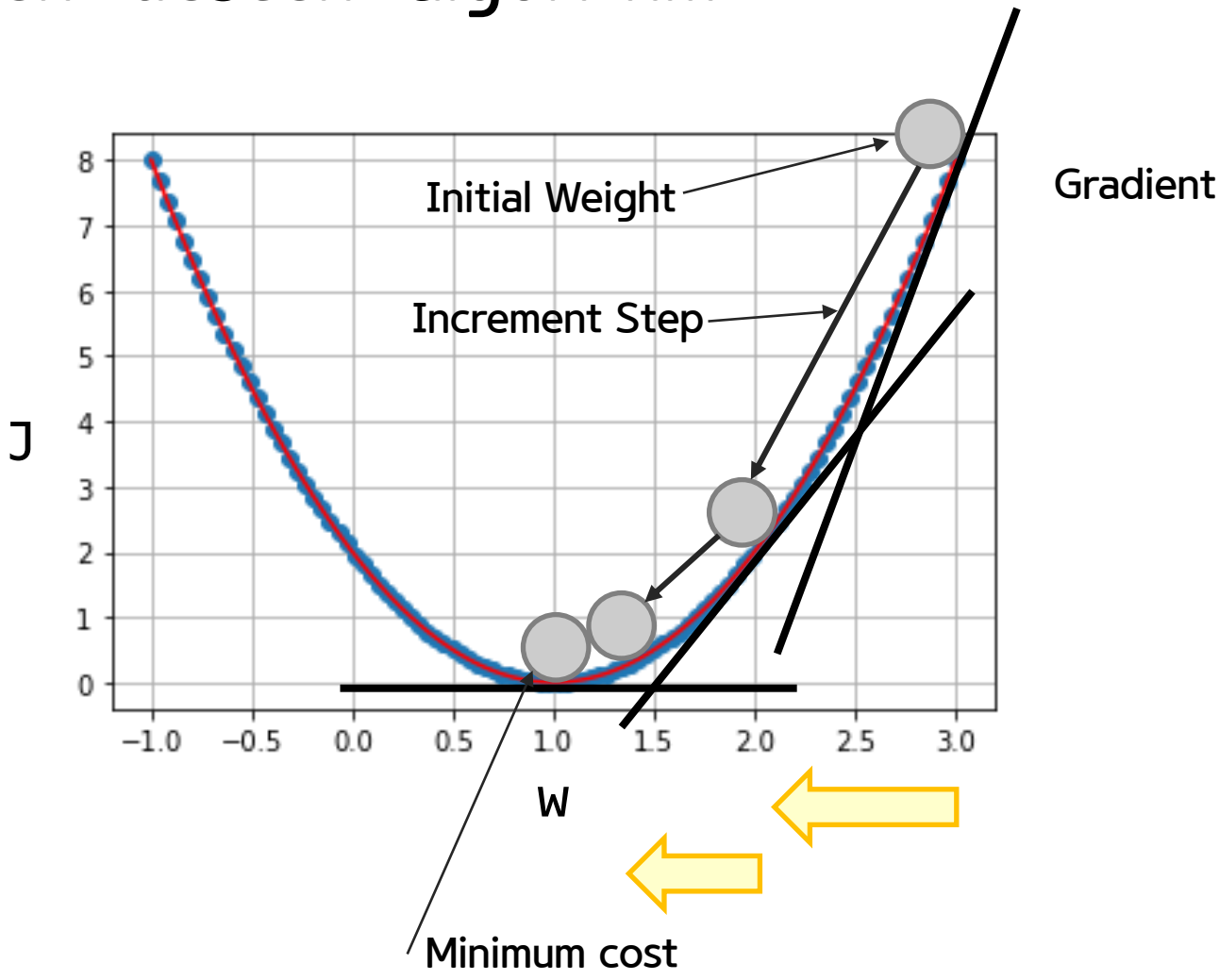
작동 방식

- 초기 추측으로 시작
  - 0,0 (또는 다른 값)에서 시작
  - W와 b를 약간 변경하여 cost(W, b)의 비용을 줄이려고 노력
- 매개 변수를 변경할 때마다 가능한 가장 낮은 cost(W, b)을 감소시키는 기울기를 선택
- 반복
- 최소한의 지역으로 수렴 할 때까지 수행

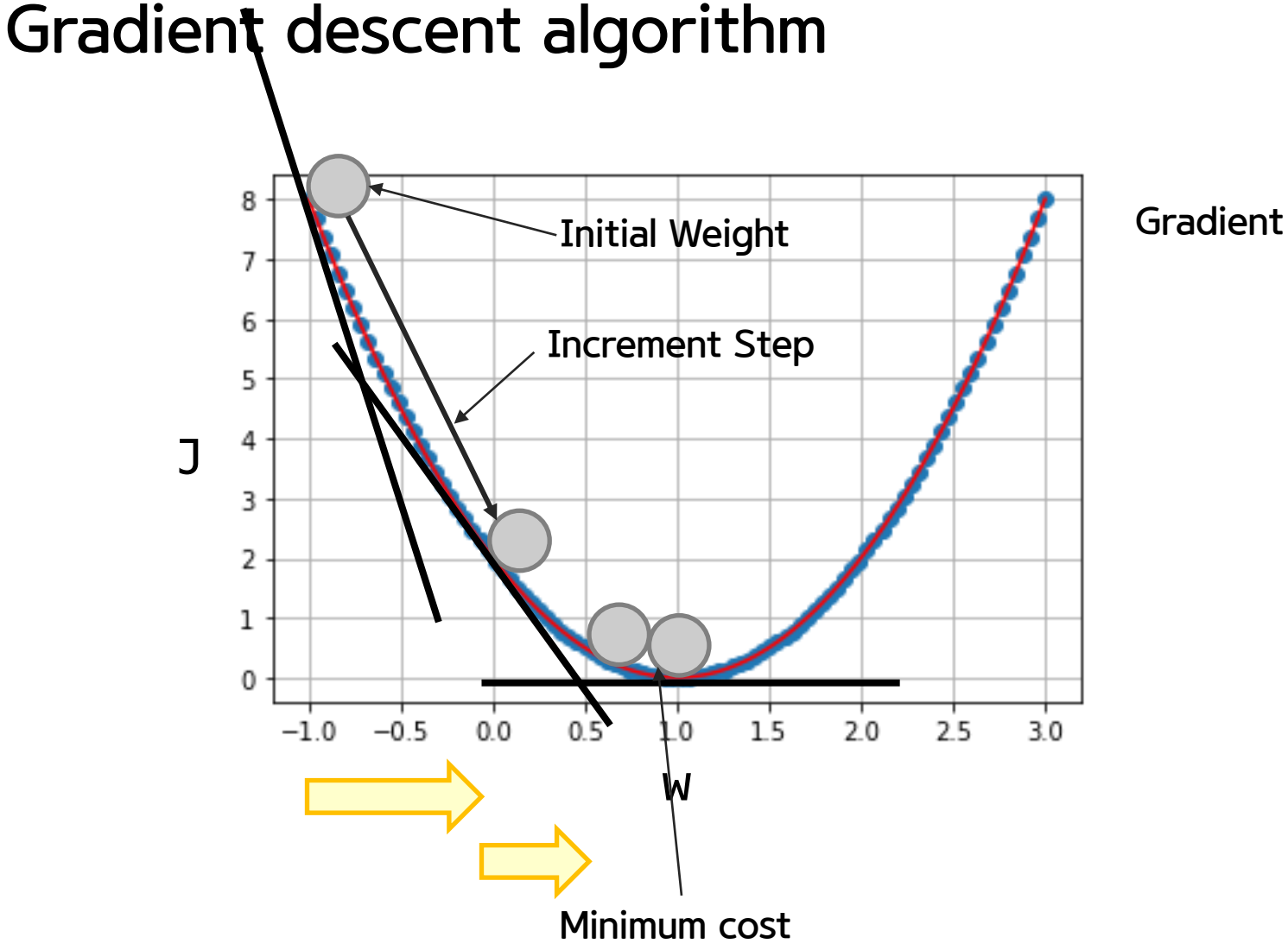
Gradient descent algorithm

$$\hat{y} = 2 * w + b$$

$$J(w, b) = \frac{1}{2}(\hat{y} -$$




```
w = w - dw

w = 3 - 1
w = 2 - 0.7
..
w = 1 - 0.0
```



$$\hat{y} = 2 * w + b$$

$$J(w, b) = \frac{1}{2}(\hat{y} -$$

Gradient

```
w = w - dw

w = -1 + 3
w = 0 + 0.7
..
w = 1 - 0.0
```

수렴까지 반복

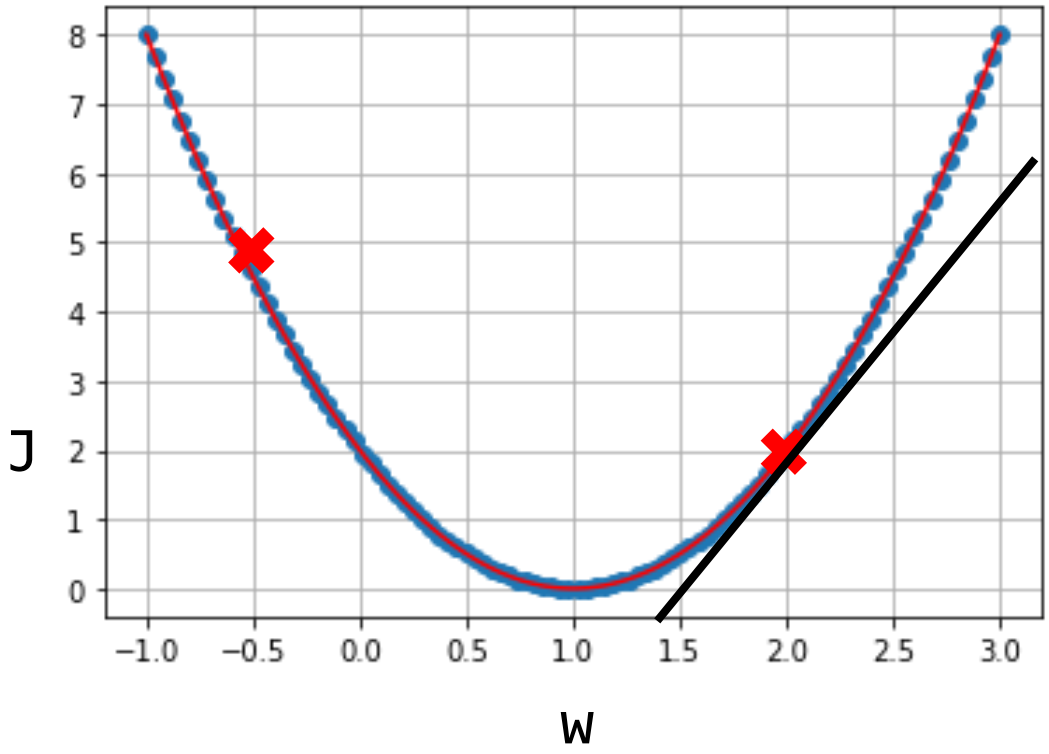
{

$$w = w - \alpha \frac{\partial}{\partial w} J(w, b)$$

}

learning rate

derivative



$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

$\frac{\partial}{\partial w} J(w, b)$**는** $\frac{1}{2}(\hat{y} - y)^2$ **을 w에 대해서 편미분 하면 된다.**

$$= \frac{1}{2}((wx + b) - y)^2$$

$$= \frac{1}{2}((wx + b)^2 - 2(wx + b)y + y^2)$$

$$= \frac{1}{2}((wx + b)^2 - 2ywx - 2by + y^2)$$

$$= \frac{1}{2}(w^2x^2 + 2wxb + b^2 - 2ywx - 2by + y^2)$$

$$= \frac{1}{2}(2wx^2 + 2xb - 2yx)$$

$$= x(wx + b - y)$$

$$= x(\hat{y} - y)$$

$$\hat{y} = wx + b$$

$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$

$$w = w - \propto * x(\hat{y} - y)$$

$$\frac{\partial J}{\partial \hat{y}} = \frac{1}{2}(\hat{y} - y)^2$$

$$= \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial w} = wx + b$$

$$= x$$

$$\frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w} = x(\hat{y} - y)$$

Chain Rule 사용

$$\hat{y} = wx + b$$

$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

$$\frac{\partial}{\partial w} J(w,b) = \frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$$

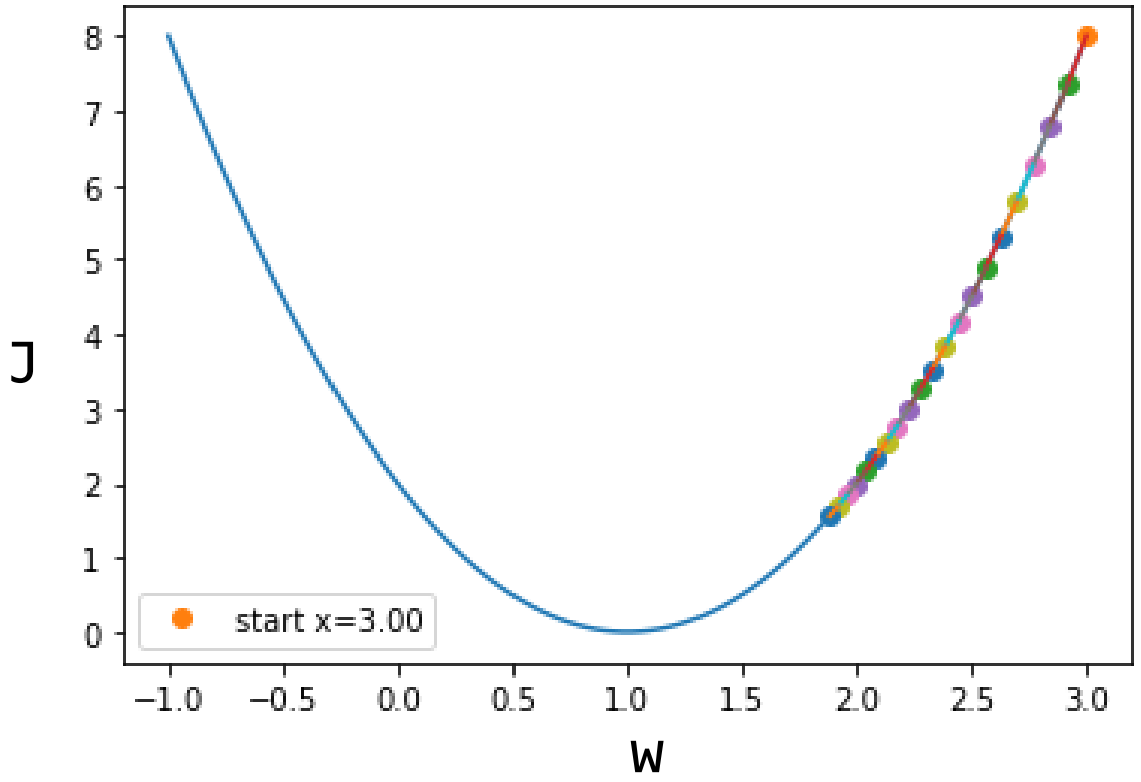
수렴까지 반복

{

$$w = w - \alpha * (\hat{y} - y) * x$$

}

α 가 0.01인 경우



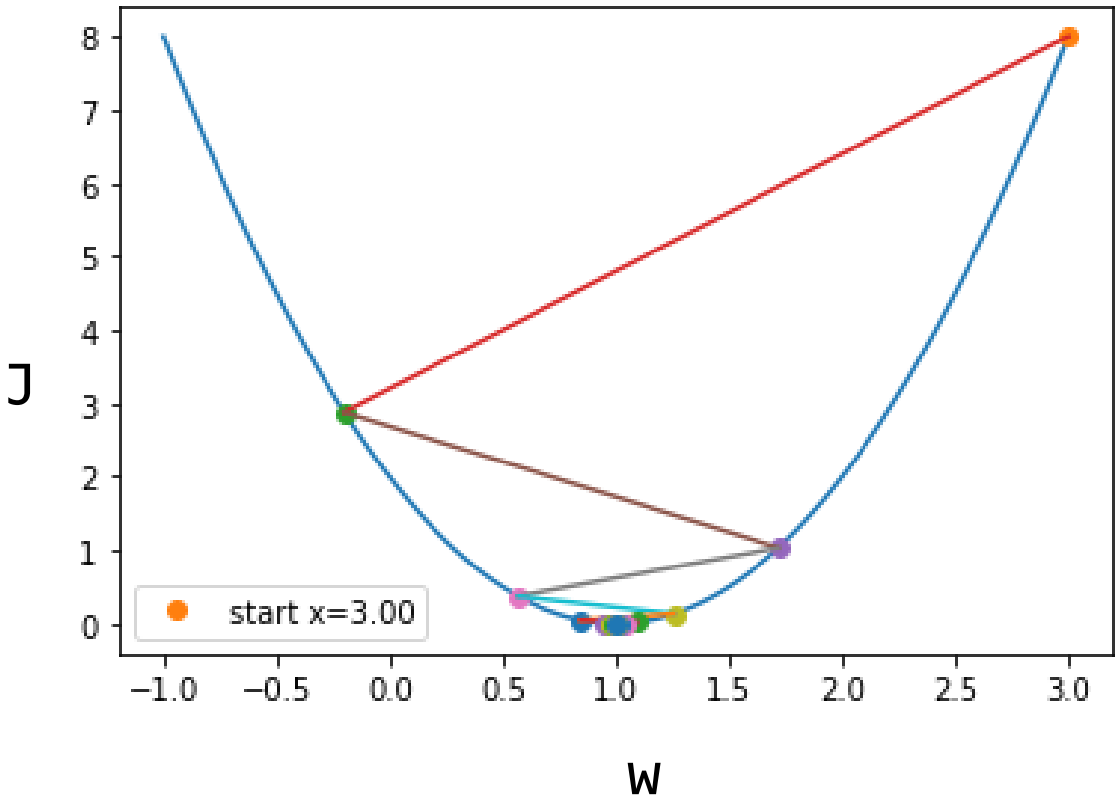
수렴까지 반복

{

$$w = w - \alpha * (\hat{y} - y) * x$$

}

α 가 **0.4**인 경우



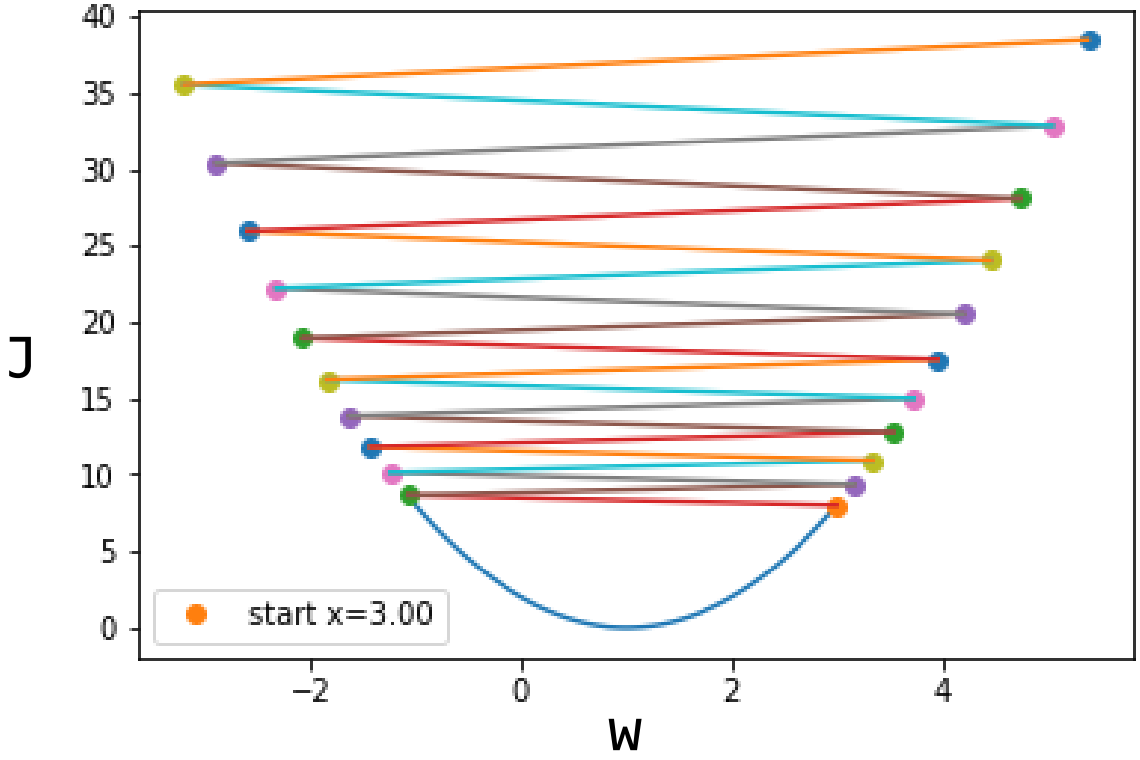
수렴까지 반복

{

$$w = w - \alpha * (\hat{y} - y) * x$$

}

α 가 0.51인 경우



$$\frac{\partial J}{\partial \hat{y}} = \frac{1}{2}(\hat{y} - y)^2$$

$$= \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial b} = wx + b$$

$$= 1$$

$$\frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b} = (\hat{y} - y)$$

Chain Rule 사용

$$\hat{y} = wx + b$$

$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

$$\frac{\partial}{\partial b} J(w,b) = \frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b}$$

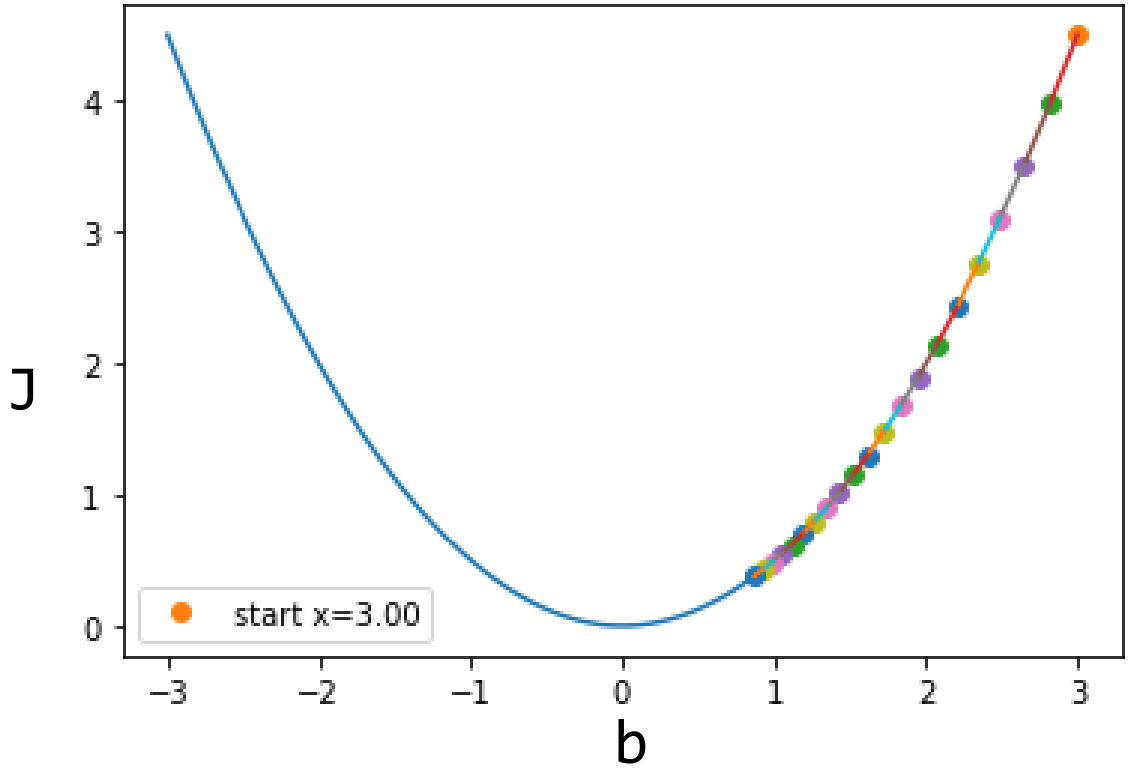
수렴까지 반복

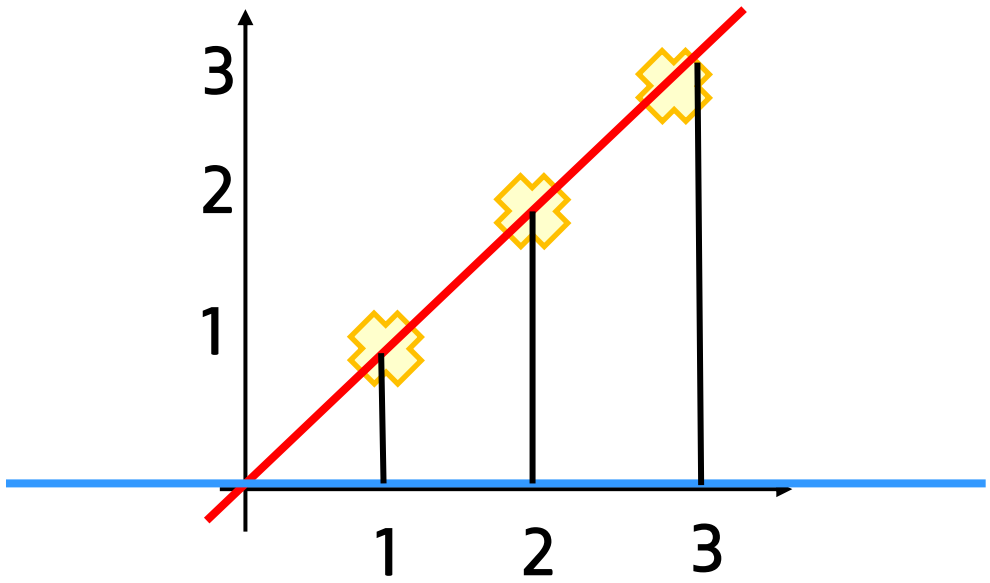
{

$b = b - \alpha * (\hat{y} - y)$

}

α 가 **0.03**인 경우






```
w = 1
b = 0
y = wx + b
```

```
x = 1,2,3
y = 1,2,3
```

```
y_hat=0,0,0
```

```
w = 0
b = 0
```

$$\frac{1}{2}(\hat{y} - y)^2$$ SE

```
(1 + 4 + 9) / 6
2.3333
```

$$\frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$ MSE => 배치경사하강

```
def train(model, x, y, learning_rate):
  for x_i, y_i in zip(x,y):
      with tf.GradientTape() as t:
        current_loss = loss(y_i, model(x_i))

      dw, db = t.gradient(current_loss, [model.w, model.b])
      print(dw.shape)

      model.w.assign_sub(learning_rate * dw)
      model.b.assign_sub(learning_rate * db)
```

w=5, b=0

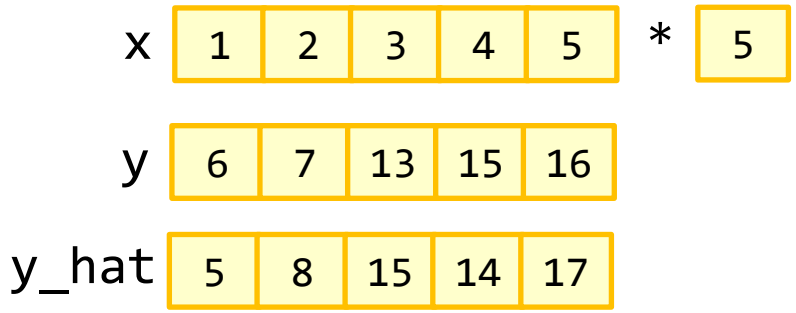
x	1	2	3	4	5
y	6	7	13	15	16

x_i=1, y_i=6

y_hat=5

```
def train(model, x, y, learning_rate):
      with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

      dw, db = t.gradient(current_loss, [model.w, model.b])
      print(dw.shape)

      model.w.assign_sub(learning_rate * dw)
      model.b.assign_sub(learning_rate * db)
```

x*w + b w=5, b=0



y_hat=5

```
x = tf.Variable([[-1,2],[-3,4],[-5,6],[-7,8],[-9,10]], dtype=tf.float32)
y = tf.Variable([[8],[10],[12],[14],[16]], dtype=tf.float32)
```

-1*1+2*2

X

-1	2
-3	4
-5	6
-7	8
-9	10

·

W

2.33
3.33

+

3.66

y

8
10
12
14
16

(5,2)(2,1)=>(5,1)

g W1=2.33 W2=3.33 b=3.66, loss=0.00002

$$y = xw + b$$

$$y = x_1 w_1 + x_2 w_2 + b$$

$$y = X \cdot W + b$$

$$y = XW + b$$

$$\frac{\partial y}{\partial w} = ERR(x)$$

$$\frac{\partial y}{\partial w_i} = ERR(x_i)$$

Logistic Regression

- 로지스틱 회귀 란?
  - Classification(분류)
  - Logistic vs Linear

- 동작 방식
  - 가설 표현
  - Sigmoid 함수
  - Decision Boundary(결정경계)
  - Cost Function
  - Optimizer (Gradient Descent)

Classification

Binary Classification(이진 분류) 란?

: 값은 0 또는 1

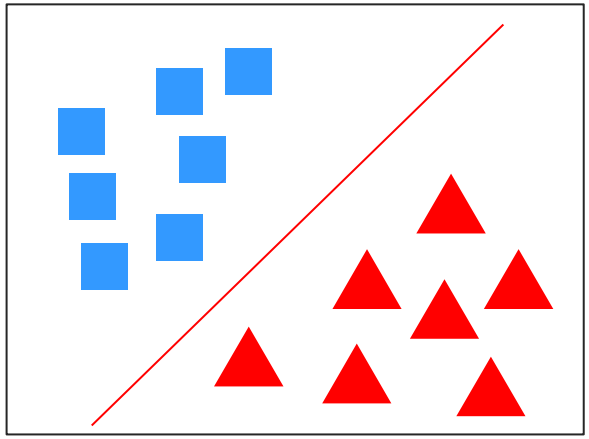
- Exam : Pass or Fail
- Spam : Not Spam or Spam
- Face : Real or Fake
- Tumor : Not Malignant or Malignant

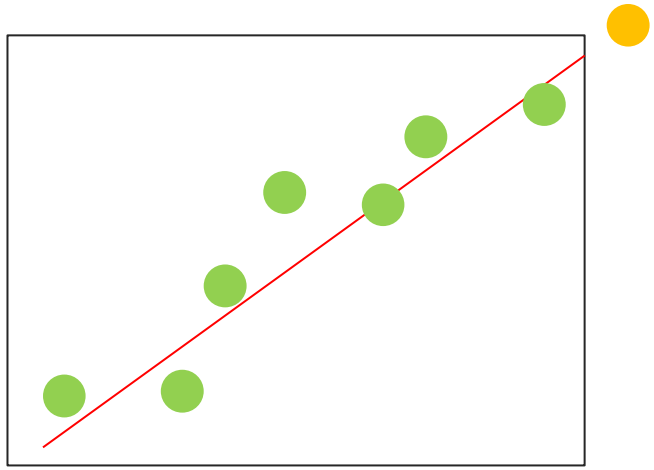
머신 러닝으로 시작하려면 변수를 인코딩해야 한다.

```
x_train = [[1,2], [2,3], [3,1], [4,3], [5,3], [6,2]]
y_train = [[0], [0], [0], [1], [1], [1]]
```

Logistic vs Linear

로지스틱 회귀와 선형 회귀의 차이점은?





Discrete(분리) : 분류 목적

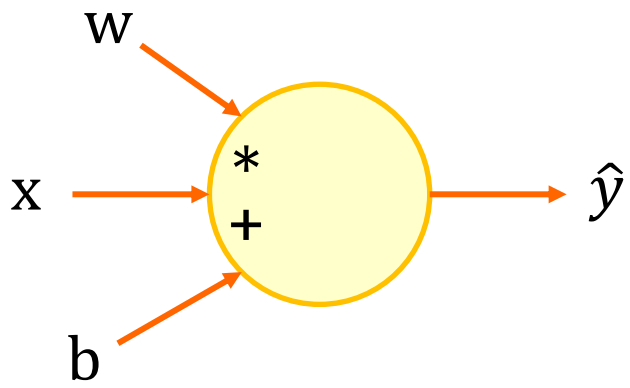
신발 사이즈 / 회사의 근로자수

Continuous (지속적인) : 값의 예측

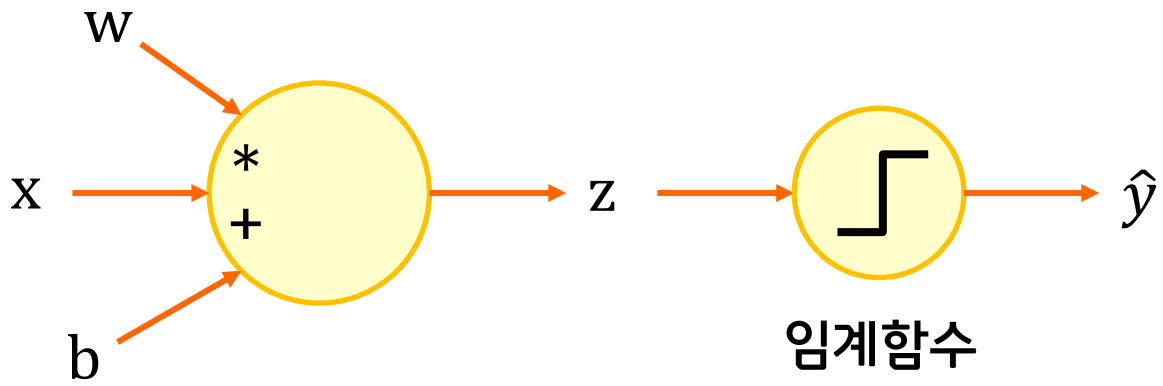
시간/ 무게 / 높이

머신 러닝으로 시작하려면 변수를 인코딩해야 한다.

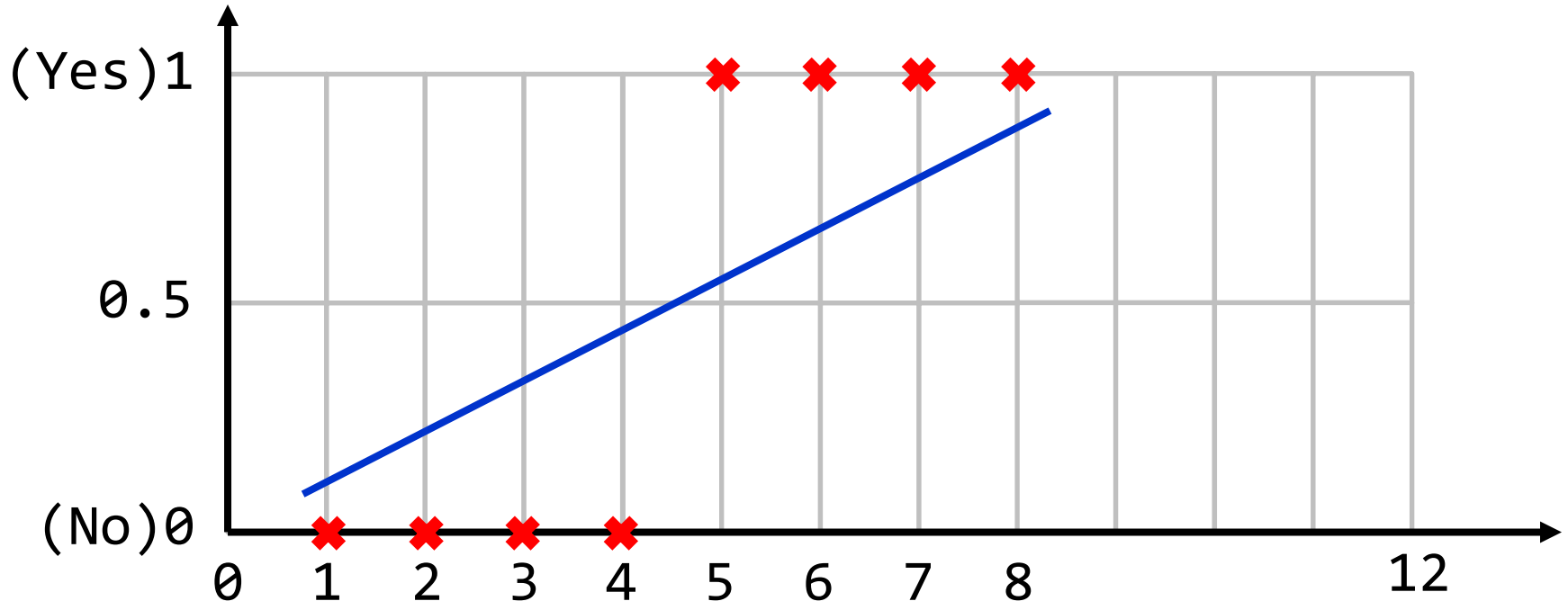
```
Logistic_Y = [[0], [0], [0], [1], [1], [1]]
Linear_Y = [[828.2], [764.2], [123. 5], [342.5], [987.3], [234.1]]
```



퍼셉트론



Sigmoid 함수 사용 이유

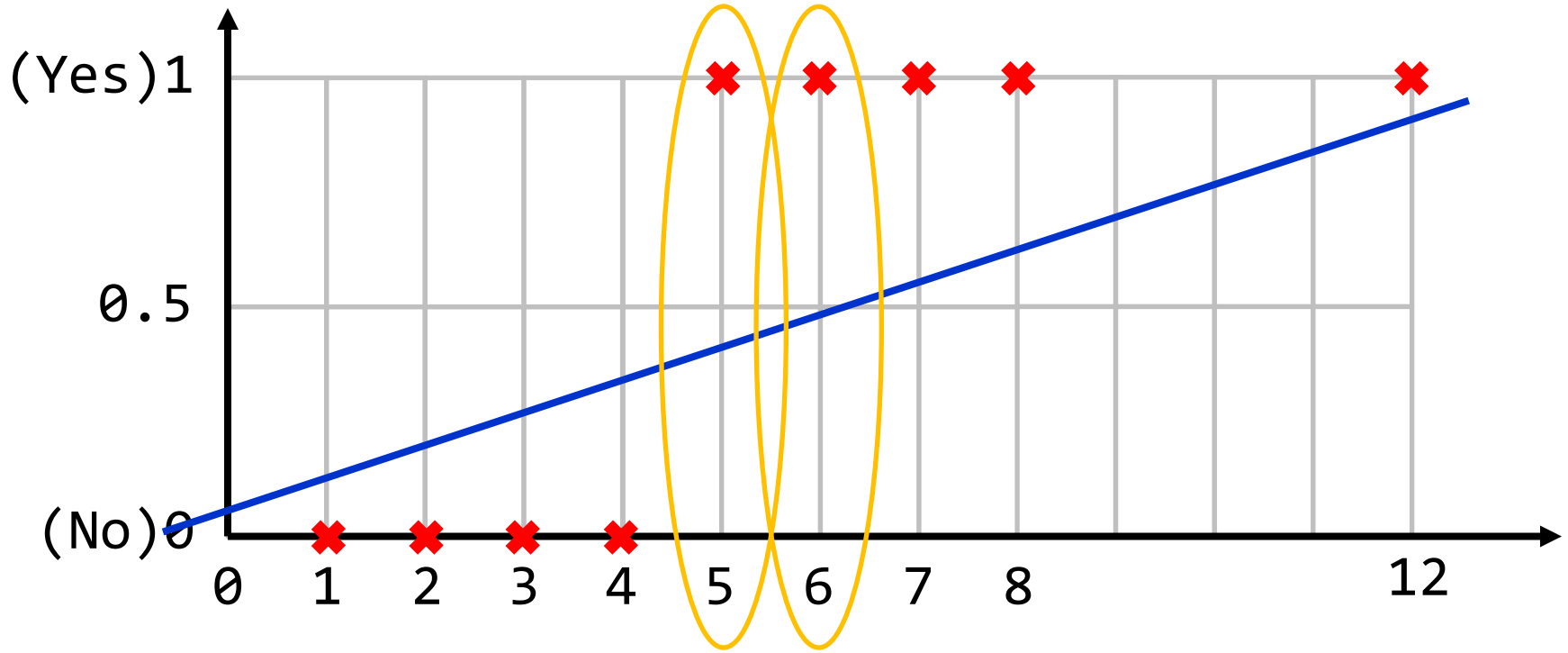


Threshold classifier output at 0.5:

If $\hat{y} \geq 0.5$, predict "y=1"
If $\hat{y} < 0.5$, predict "y=0"

1 => 양성
2 => 양성
3 => 양성
4 => 양성
5 => 악성
6 => 악성
7 => 악성
8 => 악성

Sigmoid 함수 사용 이유

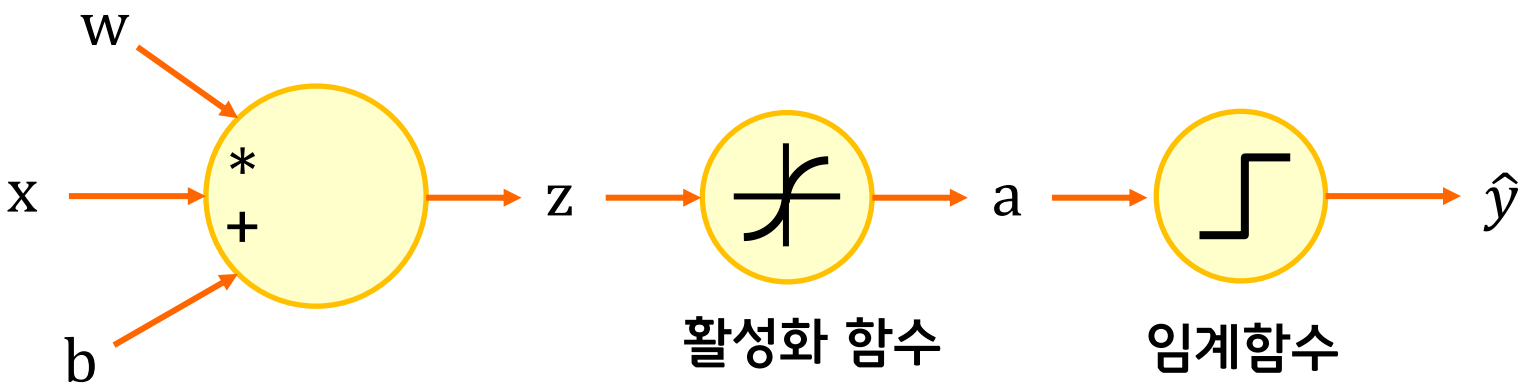


Threshold classifier output at 0.5:

If $\hat{y} \geq 0.5$, predict "y=1"
If $\hat{y} < 0.5$, predict "y=0"

1 => 양성
2 => 양성
3 => 양성
4 => 양성
5 => 양성
6 => 양성
7 => 악성
8 => 악성

Sigmoid 함수 사용 이유



Sigmoid 함수 사용 이유

Classification: y=0 or 1

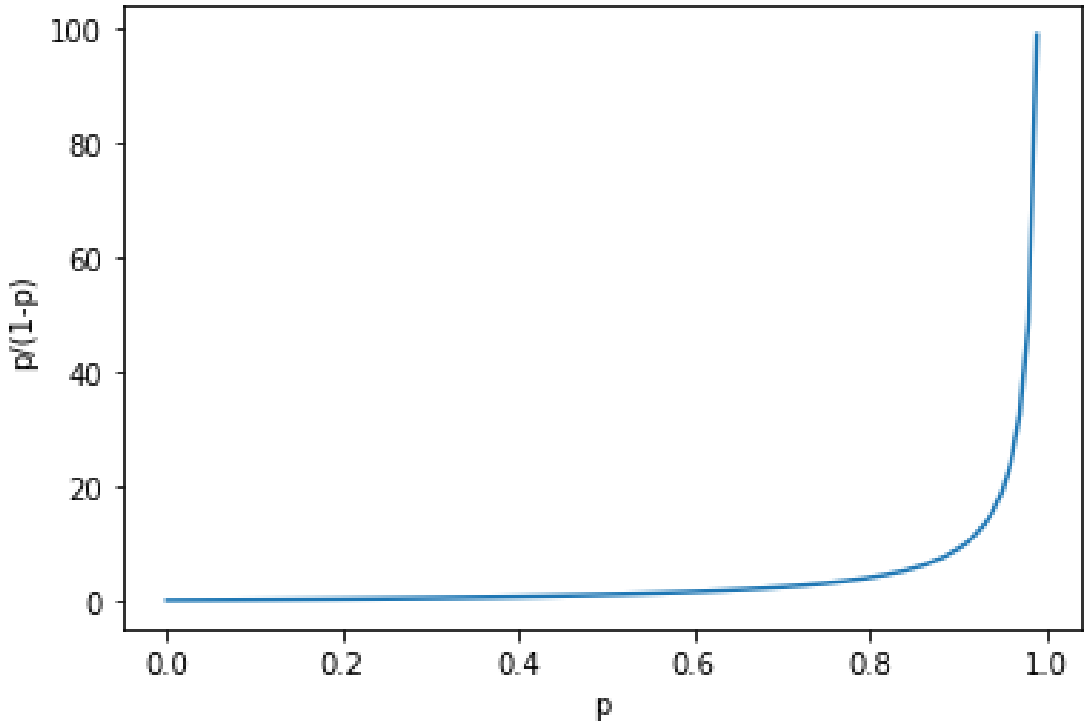
$\hat{y}$ can be > 1 or < 0

Logistic Regression: $0 < \hat{y} < 1$

분류 문제 사용

$$\text{OR(odds ratio)} = \frac{p}{1-p} \quad (\text{p=성공확률})$$

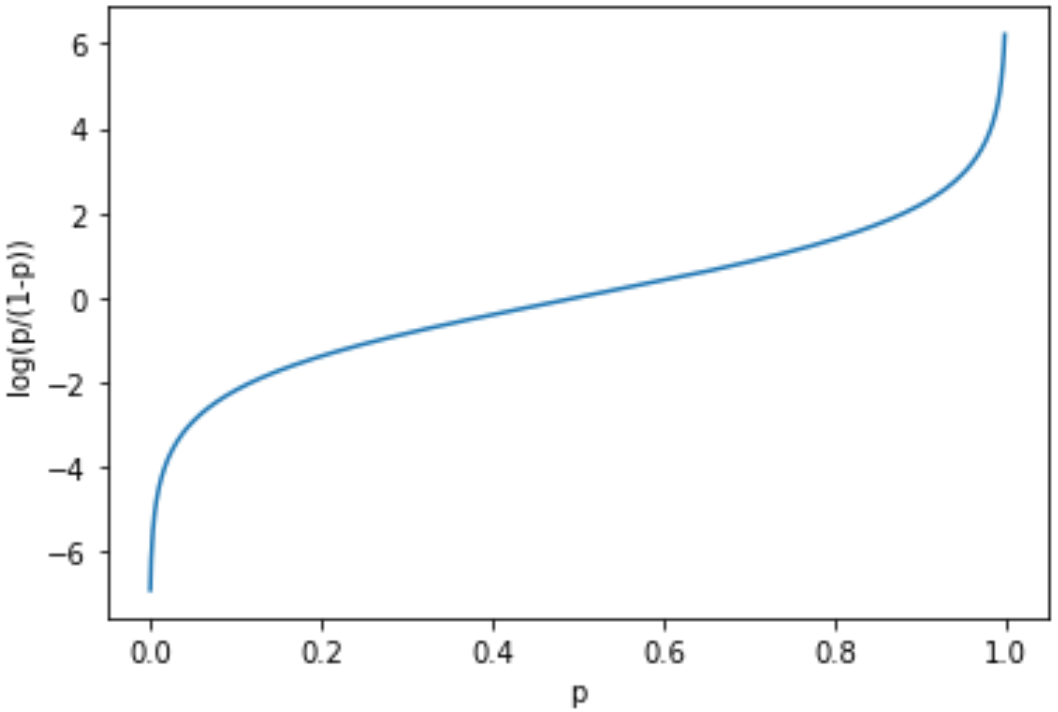
오즈 비를 그래프로 그리면 다음과 같다.
p가 0부터 1까지 증가할 때 오즈 비의 값은 처음에는
천천히 증가하지만 p가 1에 가까워지면 급격히 증가한다.



로짓 함수(Logit function)

$$\text{logit(p)} = \log\left(\frac{p}{1-p}\right)$$ (p=**성공확률**)

로짓 함수는 p가 0.5일 때 0이 되고 p가 0과 1일 때 각각 무한대로 음수와 양수가 되는 특징을 가진다.



시그모이드 함수(Sigmoid function)

$$\text{logit(p)} = \log(\frac{p}{1-p}) = z$$

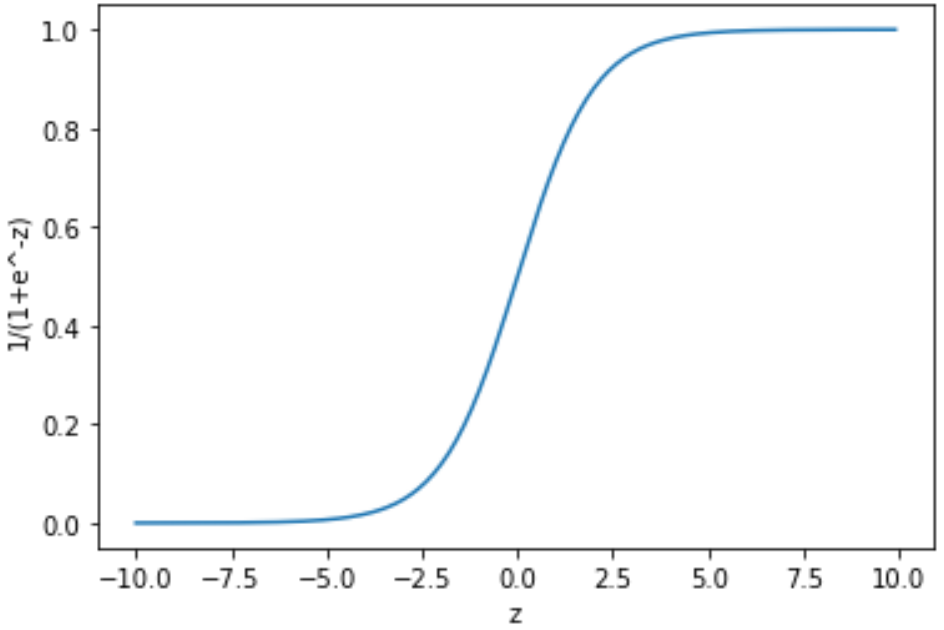
로지틱 함수의 유도 : p에대해 정리

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

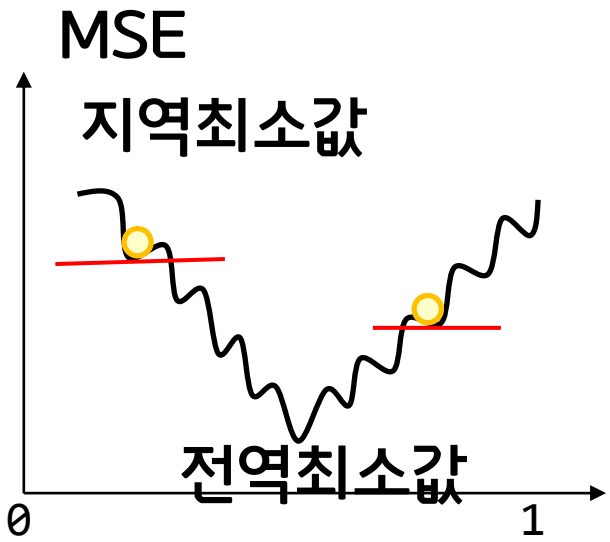
$$e^{\log(\frac{p}{1-p})} = e^{z}$$

$$\frac{p}{1-p} = e^{z}$$

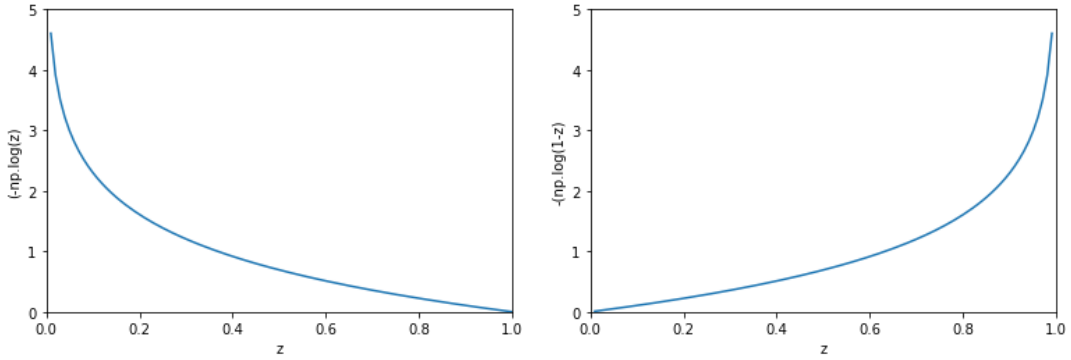
$$\text{p} = (1-p) * e^{z}$$
$$\text{p} = e^{z} - p * e^{z}$$
$$\text{p} + \text{p}* e^{z} = e^{z}$$
$$\text{p} = \frac{e^{z}}{1+e^{z}} = \frac{1}{1+e^{-z}}$$



A convex logistic regression cost function



binary cross entropy



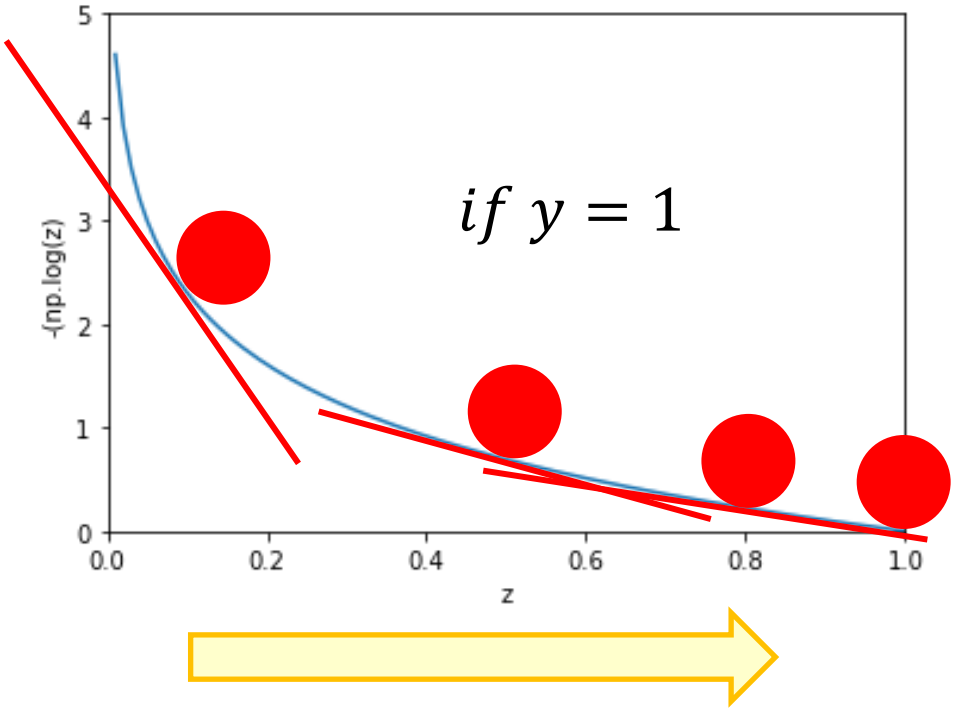
$$\text{Cost}(a, y) = \begin{cases} -\log(a) & if\ y = 1 \\ -\log(1-a) & if\ y = 0 \end{cases}$$

$$J(w) = \frac{1}{2}(sigmoid(\hat{y}) - y)^2$$

$$\text{cost}(a, y) = -(\text{y} * \log(a) + (1-y)\log(1-\text{a}))$$

Logistic regression cost function

$$\text{Cost}(a, y) = \begin{cases} -\log(a) & if\ y = 1 \\ -\log(1-a) & if\ y = 0 \end{cases}$$



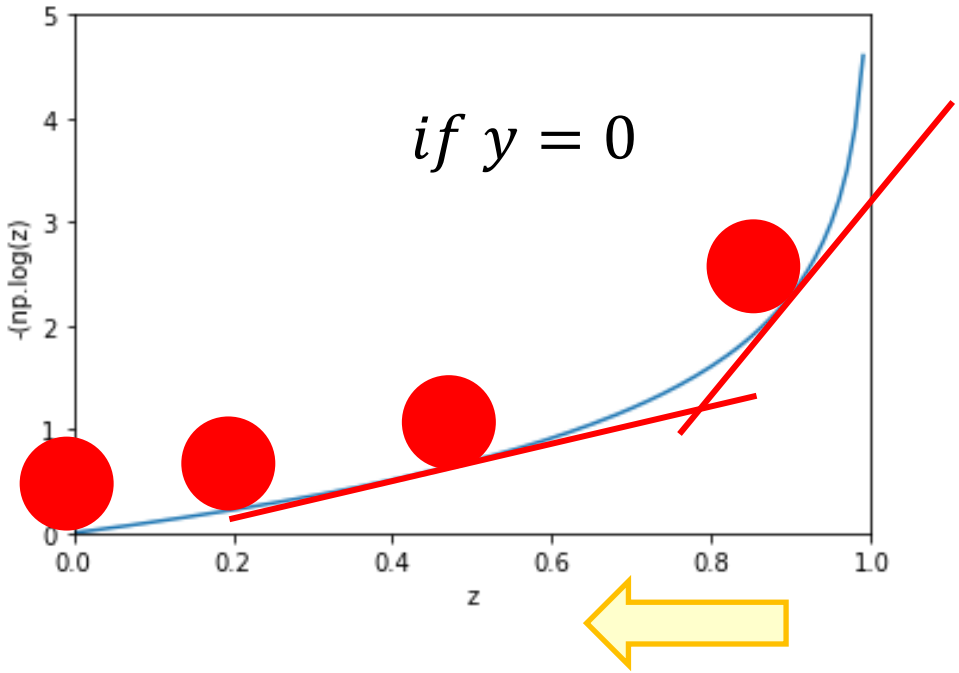
$\text{Cost} = 0 \ \text{if}\ y = 1,\ a = 1$

But as $a \rightarrow 0$

$\text{Cost} \rightarrow \infty$

Logistic regression cost function

$$\text{Cost}(a, y) = \begin{cases} -\log(a) & if\ y = 1 \\ -\log(1-a) & if\ y = 0 \end{cases}$$



$\text{Cost} = 0$ if $y = 0$, $a = 0$

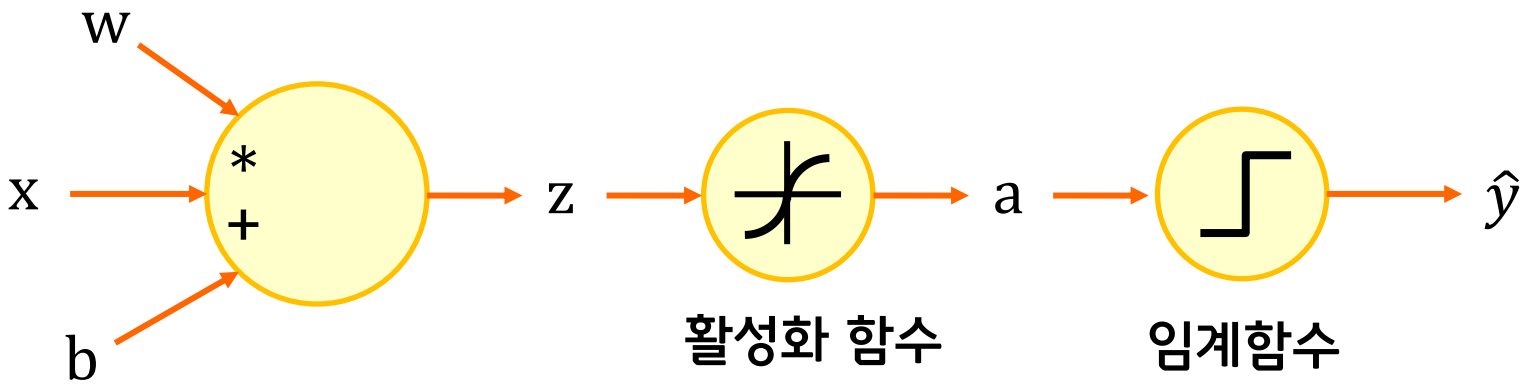
But as $a \rightarrow 1$

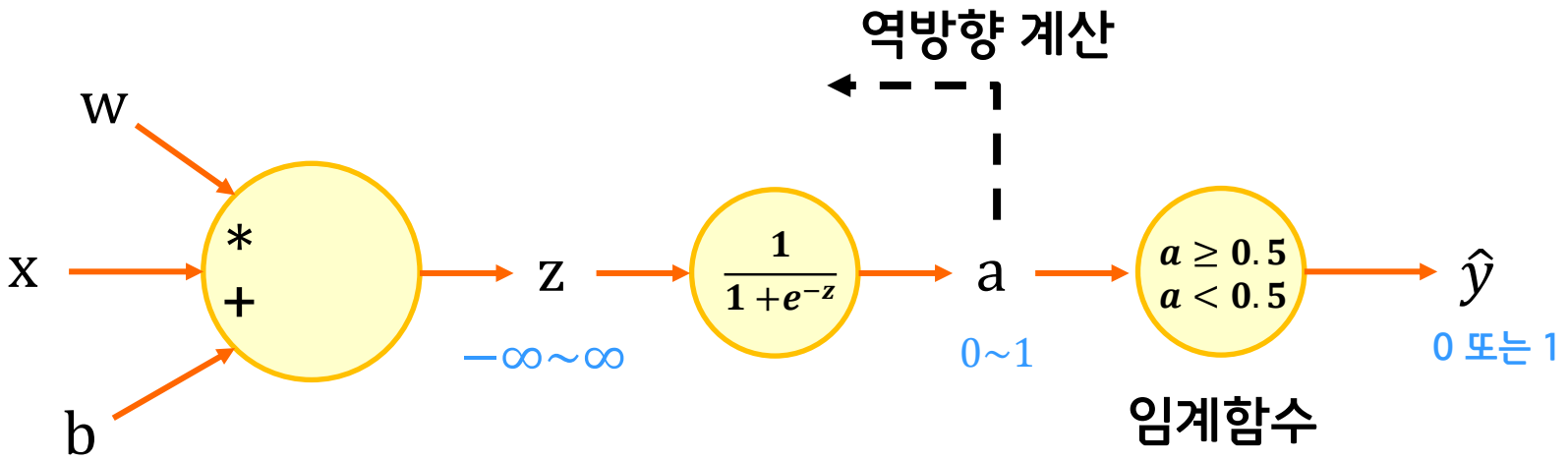
$\text{Cost} \rightarrow \infty$

Logistic regression cost function

$$\text{Cost}(a,\ y) = \begin{cases} -\log(a) & if\ y = 1 \\ -\log(1-a) & if\ y = 0 \end{cases}$$

$$\text{L} = -(\text{y} * \log(a) + (1-y)\log(1-\text{a}))$$

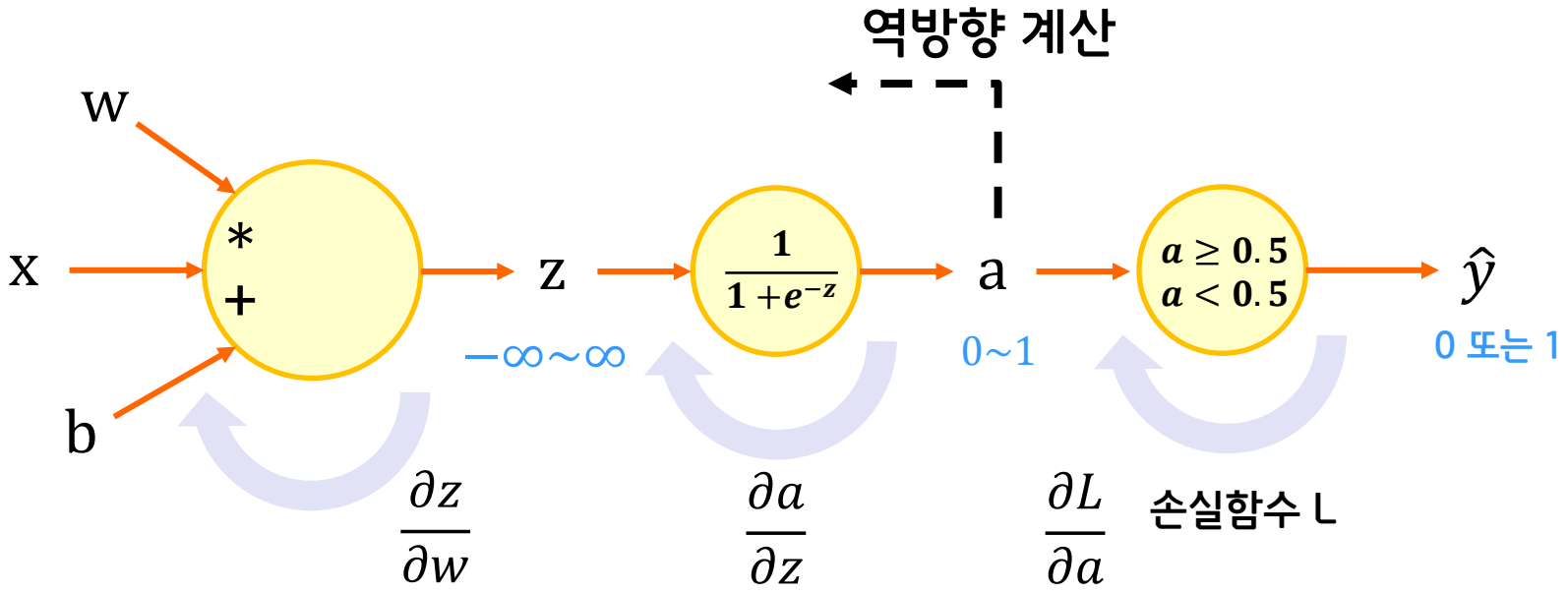

w
x
b
*
+
z
a
$\hat{y}$
활성화 함수
임계함수


역방향 계산
w
x
b
*
+
z
$\frac{1}{1+e^{-z}}$
a
$a \geq 0.5$
$a < 0.5$
$\hat{y}$
$-\infty \sim \infty$
0~1
0 또는 1
임계함수

로지스틱 회귀 손실 함수

특성이 하나인 경우 => x 1개

z = wx + b



chain rule을 이용하여 각 단계의 미분 결과를 곱한다.

w에 대하여 미분

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w} = (a-y)x$$

$$\frac{\partial L}{\partial a} = -(y\frac{1}{a} - (1-y)\frac{1}{1-a})$$

$$\frac{\partial a}{\partial z} = a(1-a)$$

$$\frac{\partial z}{\partial w} = x$$

b에 대하여 미분

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b} = (a-y)1$$

$$\frac{\partial L}{\partial a} = -(y\frac{1}{a} - (1-y)\frac{1}{1-a})$$

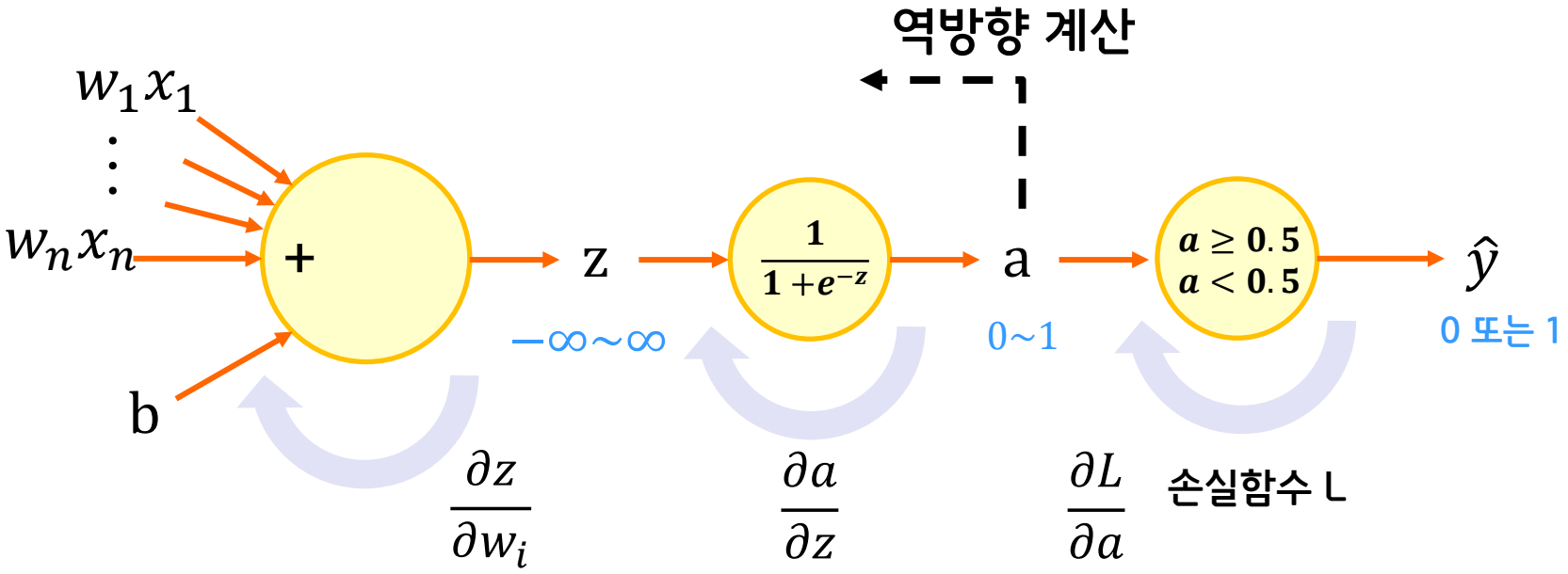
$$\frac{\partial a}{\partial z} = a(1-a)$$

$$\frac{\partial z}{\partial b} = 1$$

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b$$

특성이 여러개인 경우 => x n개

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



chain rule을 이용하여 각 단계의 미분 결과를 곱한다.

w에 대하여 미분

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i} = (a-y)x_i$$

$$\frac{\partial L}{\partial a} = -(y\frac{1}{a} - (1-y)\frac{1}{1-a})$$

$$\frac{\partial a}{\partial z} = a(1-a)$$

$$\frac{\partial z}{\partial w_i} = x_i$$

b에 대하여 미분

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b} = (a-y)1$$

$$\frac{\partial L}{\partial a} = -(y\frac{1}{a} - (1-y)\frac{1}{1-a})$$

$$\frac{\partial a}{\partial z} = a(1-a)$$

$$\frac{\partial z}{\partial b} = 1$$

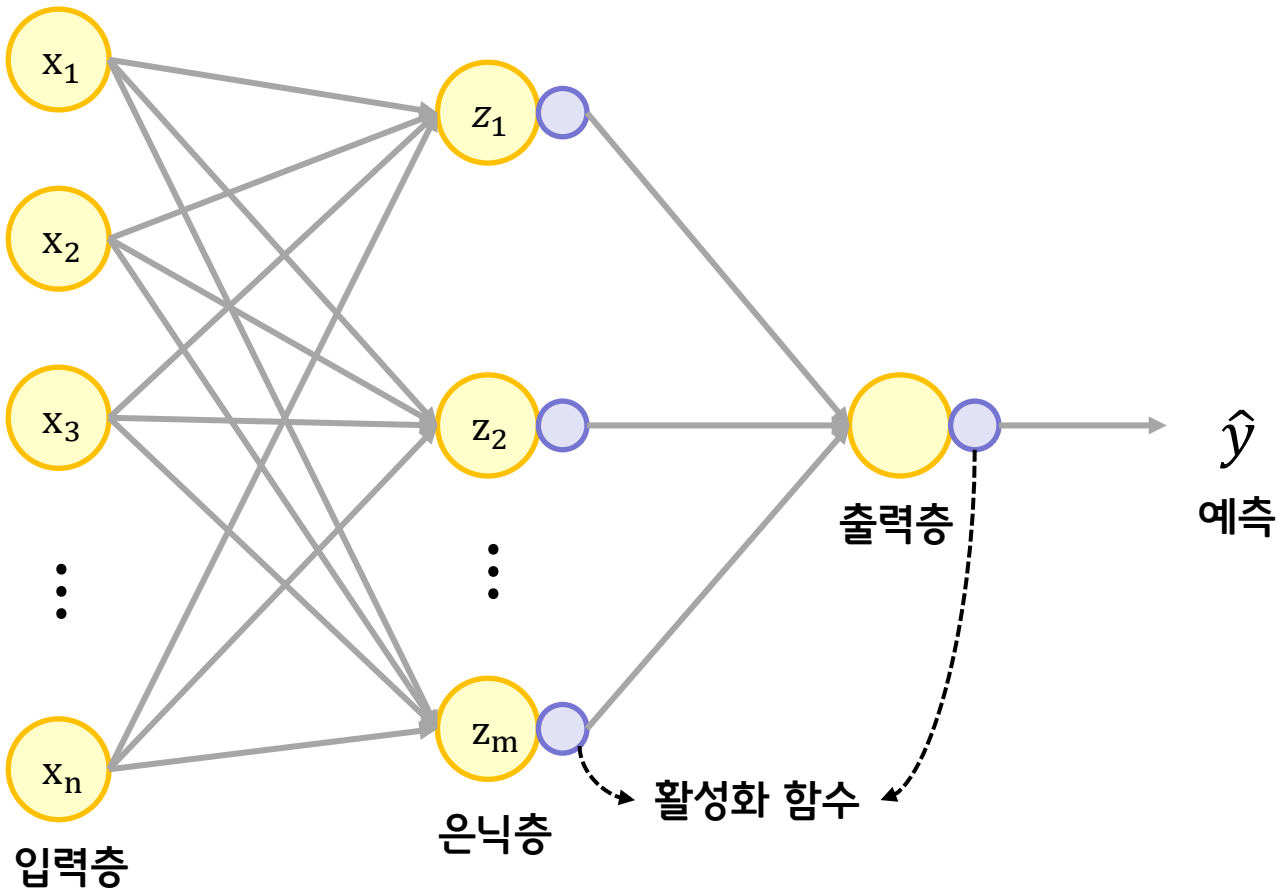
```
w = w - (예측값 - 실제값)*x
b = b - (예측값 - 실제값)
```

```
w0 -= (예측값 - 실제값)*x0
w1 -= (예측값 - 실제값)*x1
```

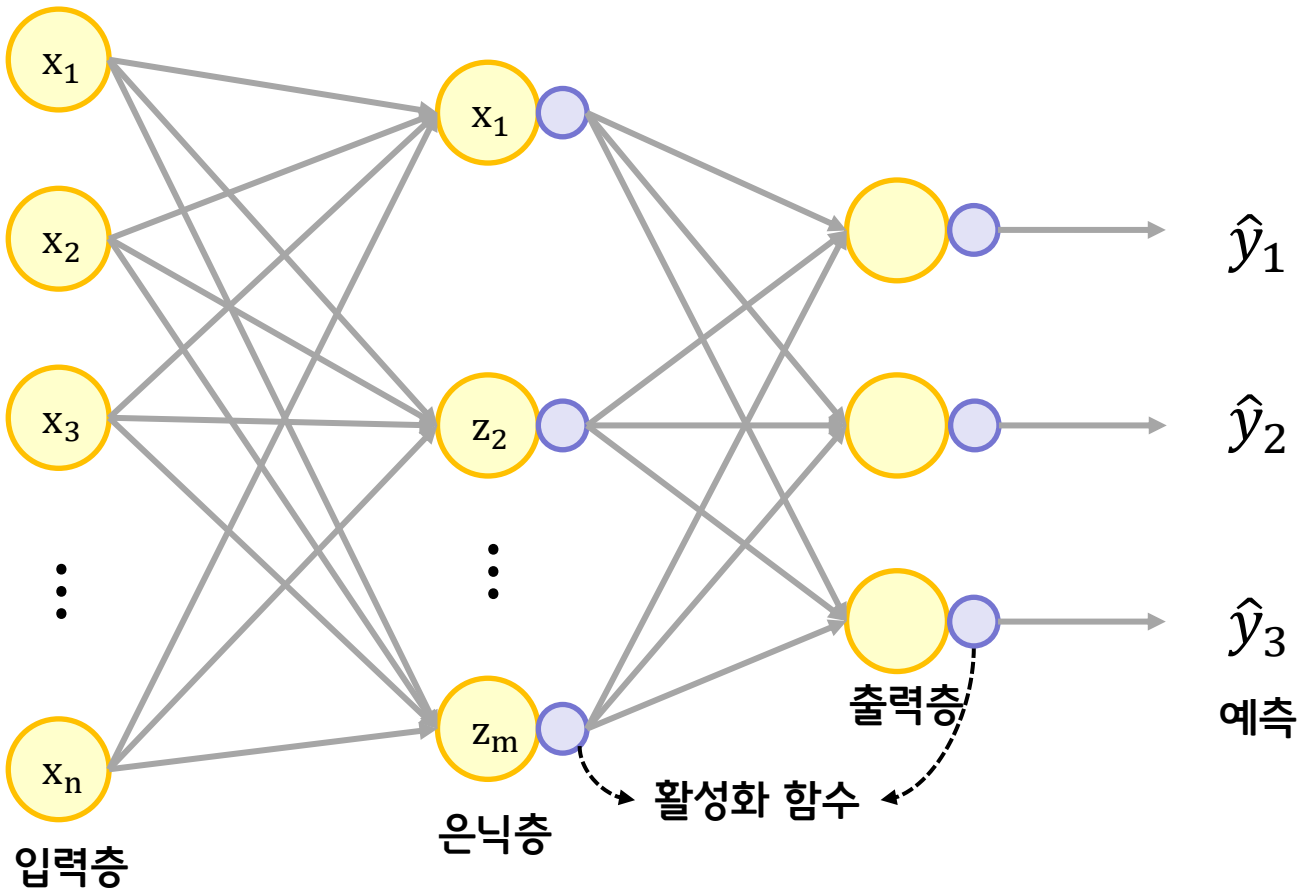
제곱 오차의 미분과 로지스틱 손실 함수의 미분은 $\hat{y}$ 이 a로 바뀌었을 뿐 동일 하다.
따라서 선형함수의 결과 값을 activation 함수인 시그모이드를 적용한 값이 a 이다.

	제곱 오차의 미분	로지스틱 손실 함수의 미분
가중치에 대한 미분	$\frac{\partial SE}{\partial w_i} = (\hat{y} - y)x_i$	$\frac{\partial L}{\partial w_i} = (a - y)x_i$
절편에 대한 미분	$\frac{\partial SE}{\partial b} = (\hat{y} - y)1$	$\frac{\partial L}{\partial b} = (a - y)1$

다중 분류 신경망



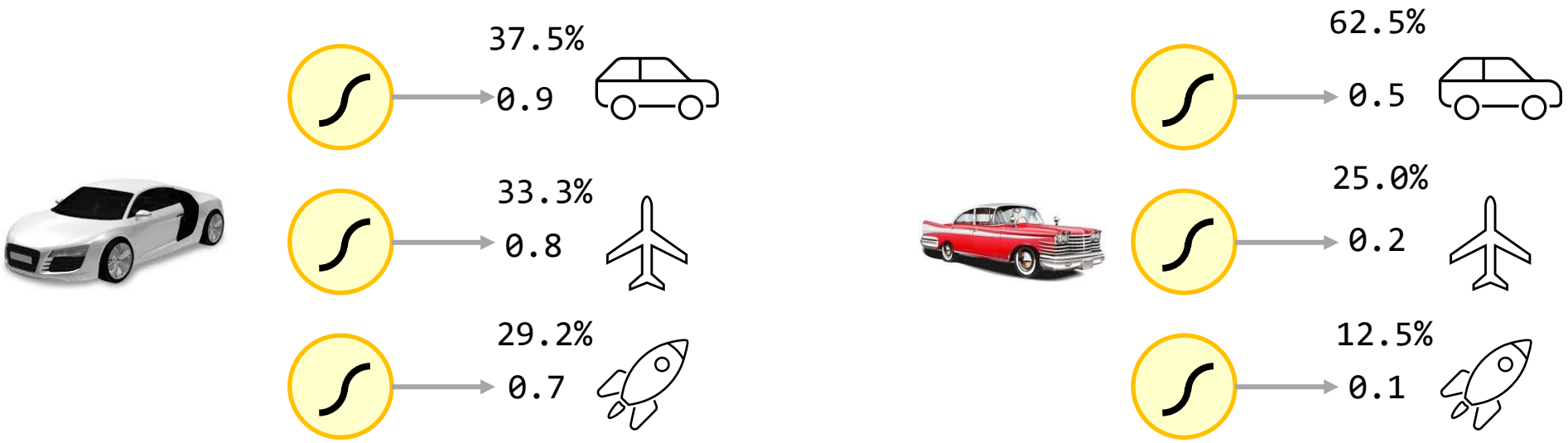
다중 분류 신경망



다중 분류 신경망

활성화 출력의 합이 1이 아니면 비교하기 어렵다.

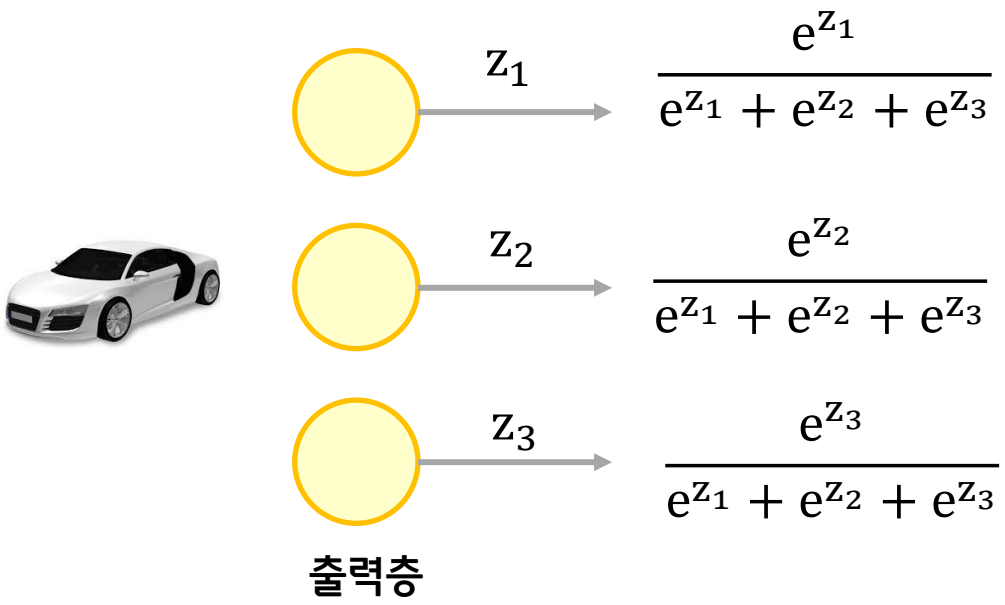
소프트맥스 함수를 적용해 출력 강도를 정규화 한다.



다중 분류 신경망

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

$$\frac{e^{z_i}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

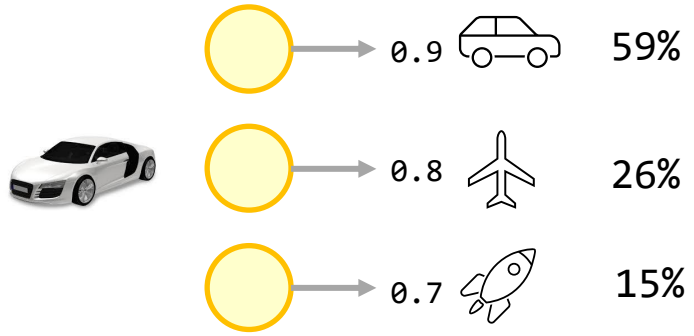


다중 분류 신경망

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z에 대해 정리하면

$$\hat{y} = \frac{1}{1 + e^{-z}} \Rightarrow z = -\log\left(\frac{1}{\hat{y}} - 1\right)$$



$$z_1 = -\log\left(\frac{1}{0.9} - 1\right) = 2.20 \qquad z_2 = -\log\left(\frac{1}{0.8} - 1\right) = 1.39 \qquad z_3 = -\log\left(\frac{1}{0.7} - 1\right) = 0.85$$

$$\hat{y} = \frac{e^{2.20}}{e^{2.20}+e^{1.39}+e^{0.85}} = 0.59 \qquad \hat{y} = \frac{e^{1.39}}{e^{2.20}+e^{1.39}+e^{0.85}} = 0.26 \qquad \hat{y} = \frac{e^{0.85}}{e^{2.20}+e^{1.39}+e^{0.85}} = 0.15$$

49.5%

31.3%

19.1%

자동차 59%

비행기 26%

로켓 15%

[1,0,0]

크로스 엔트로피 손실 함수

$$-(y\log(a)+(1-y)\log(1-a))$$

$$L=-\sum_{c=1}^{c} y_c\log(a_c)=-(y_1\log(a_1)+y_2\log(a_2)+\cdots+y_c\log(a_c))$$

크로스 엔트로피 손실 함수 미분

$$\frac{\partial L}{\partial z_1}=\frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial z_1}+\frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_1}+\frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial z_1}$$

크로스 엔트로피 손실 함수 미분

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial z_1} + \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_1} + \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial z_1}$$

$$\frac{\partial L}{\partial a_1} = -\frac{y_1}{a_1} \qquad \frac{\partial L}{\partial a_2} = -\frac{y_2}{a_2} \qquad \frac{\partial L}{\partial a_3} = -\frac{y_3}{a_3}$$

$$\frac{\partial L}{\partial z_1} = \left(-\frac{y_1}{a_1}\right)\frac{\partial a_1}{\partial z_1} + \left(-\frac{y_2}{a_2}\right)\frac{\partial a_2}{\partial z_1} + \left(-\frac{y_3}{a_3}\right)\frac{\partial a_3}{\partial z_1}$$

크로스 엔트로피 손실 함수 미분

$$\frac{\partial \mathrm{L}}{\partial \mathrm{z}_1} = \left(-\frac{\mathrm{y}_1}{\mathrm{a}_1}\right)\frac{\partial \mathrm{a}_1}{\partial \mathrm{z}_1} + \left(-\frac{\mathrm{y}_2}{\mathrm{a}_2}\right)\frac{\partial \mathrm{a}_2}{\partial \mathrm{z}_1} + \left(-\frac{\mathrm{y}_3}{\mathrm{a}_3}\right)\frac{\partial \mathrm{a}_3}{\partial \mathrm{z}_1}$$

$$\frac{\partial \mathrm{a}_1}{\partial \mathrm{z}_1} = \mathrm{a}_1(1-\mathrm{a}_1) \qquad \frac{\partial \mathrm{a}_2}{\partial \mathrm{z}_1} = -\mathrm{a}_2\mathrm{a}_1 \qquad \frac{\partial \mathrm{a}_3}{\partial \mathrm{z}_1} = -\mathrm{a}_3\mathrm{a}_1$$

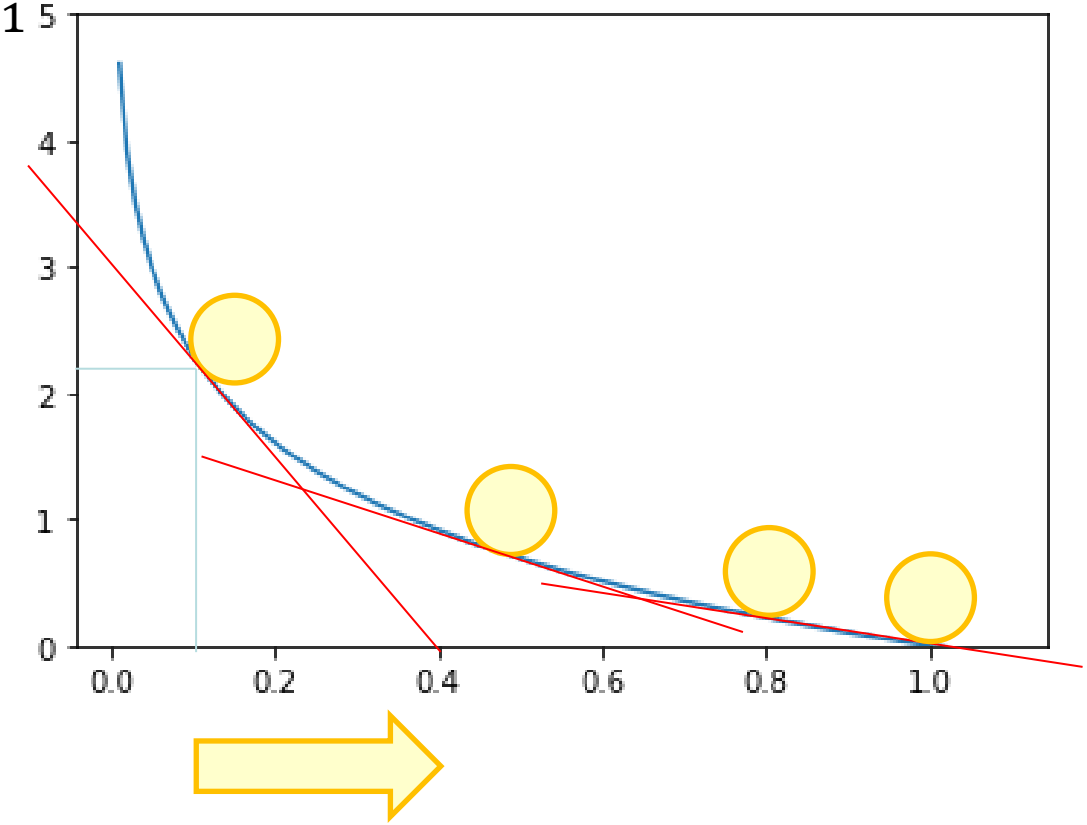
$$\begin{aligned}\frac{\partial \mathrm{L}}{\partial \mathrm{z}_1} &= \left(-\frac{\mathrm{y}_1}{\mathrm{a}_1}\right)\mathrm{a}_1(1-\mathrm{a}_1) + \left(-\frac{\mathrm{y}_2}{\mathrm{a}_2}\right)(-\mathrm{a}_2\mathrm{a}_1) + \left(-\frac{\mathrm{y}_3}{\mathrm{a}_3}\right)(-\mathrm{a}_3\mathrm{a}_1) \\ &= (\mathrm{a}_1 - y_1)\end{aligned}$$

$$\boldsymbol{\frac{\partial L}{\partial z} = (a - y)}$$

(a-y)
(0.1-1) => -0.9

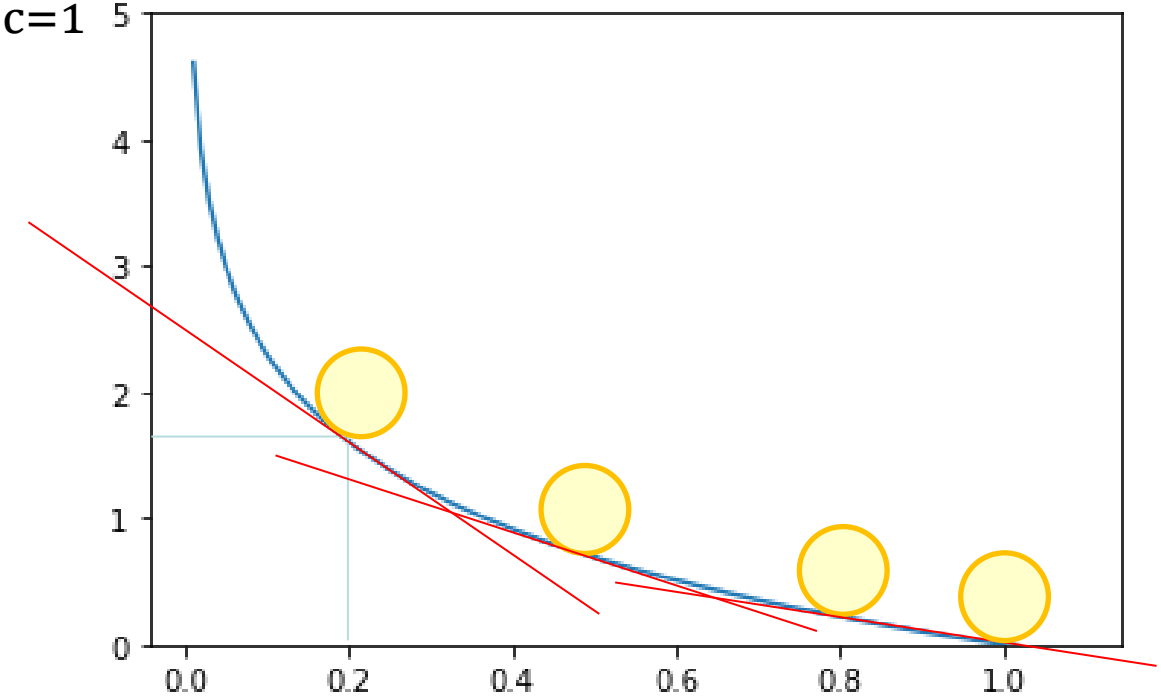
[0.1, 0.2, 0.7] 예측값
[1, 0, 0] 정답
[-0.9, 0.2, 0.7]

$$L = -\sum_{c=1}^{c} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + \cdots + y_c \log(a_c))$$



2 [0.5, 0.3, 0.2]
[1, 0, 0]
[-0.5, 0.3, 0.2]

$$L = -\sum_{c=1}^{c} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + \cdots + y_c \log(a_c))$$



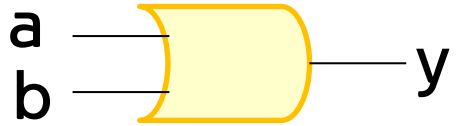
하나의 층에 여러개의 뉴런 사용

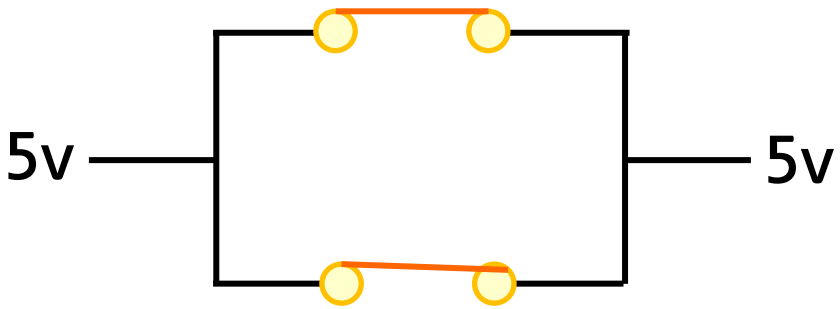
OR Gate

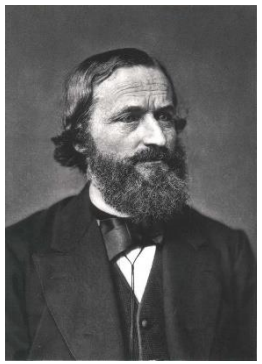
a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

0	1	0	1
| 0	0	1	1
0	1	1	1

a | b





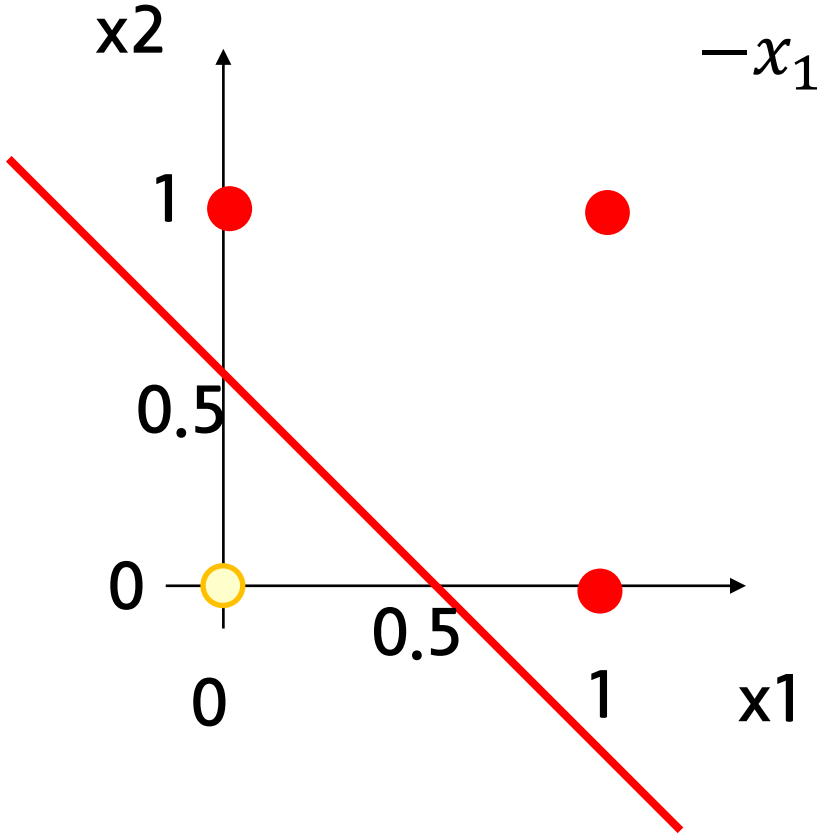


키르히 호프의 전류의 법칙

하나의 층에 여러개의 뉴런 사용

$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

$$-x_1 + 0.5 = x_2$$



$$w_1 = 1$$

$$w_2 = 1$$

$$b = -0.5$$

$$y: 0 \; w_1 x_1 + w_2 x_2 + b \leq 0$$

$$y: 1 \; w_1 x_1 + w_2 x_2 + b > 0$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

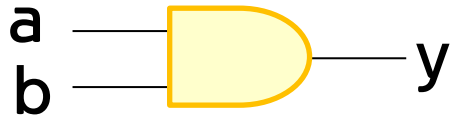
하나의 층에 여러개의 뉴런 사용

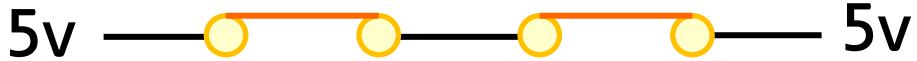
And Gate

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

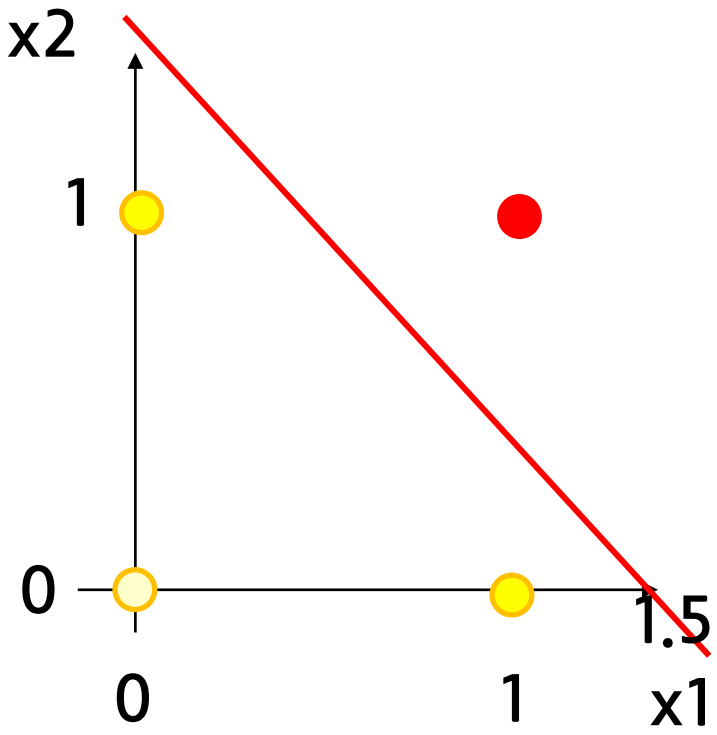
	0	1	0	1
&	0	0	1	1
	0	0	0	1

a & b





하나의 층에 여러개의 뉴런 사용



$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

$$-x_1 + 1.5 = x_2$$

$$w_1 = 1$$

$$w_2 = 1$$

$$b = -1.5$$

$$y{:}\,0 \;\; w_1 x_1 + w_2 x_2 + b \leq 0$$

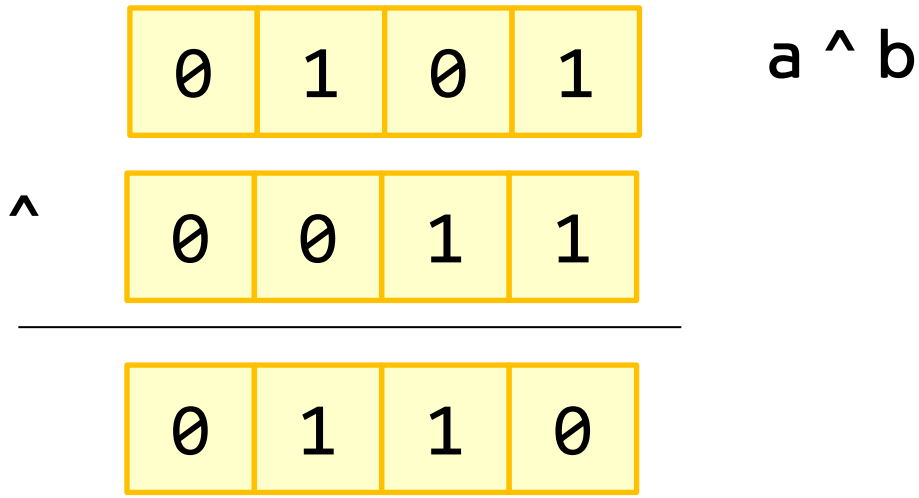
$$y{:}\,1 \;\; w_1 x_1 + w_2 x_2 + b > 0$$

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

하나의 층에 여러개의 뉴런 사용

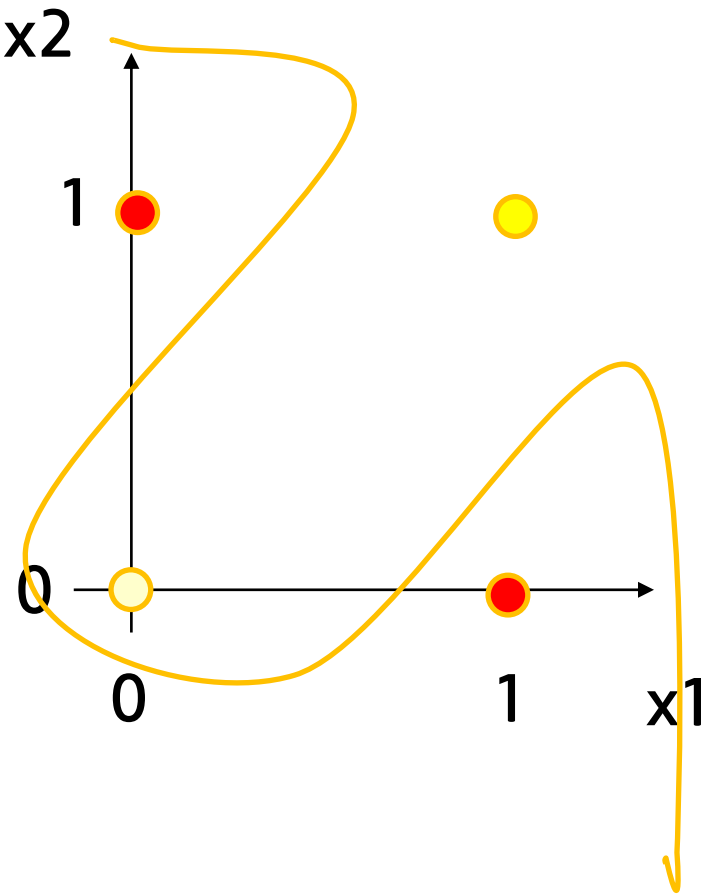
XOR : 상호배타 논리합

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0



a ^ b

하나의 층에 여러개의 뉴런 사용



$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

$$w_1 = 0.7$$

$$w_2 = 0.7$$

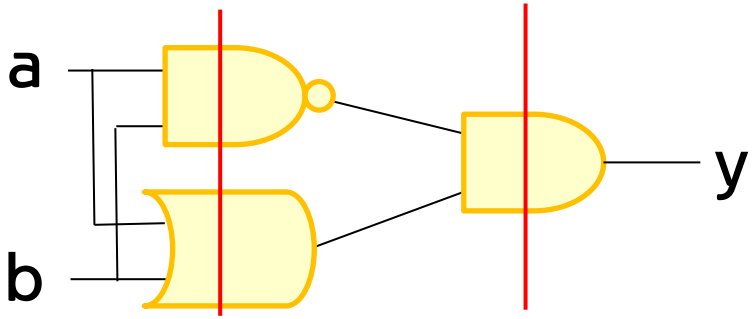
$$b = -0.3$$

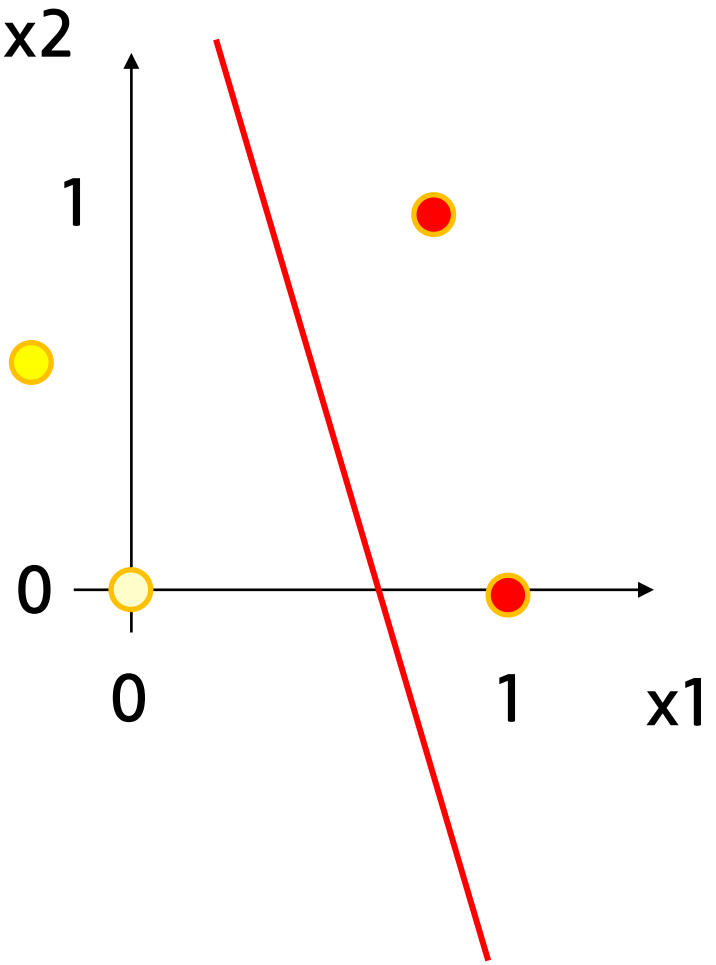
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

https://playground.tensorflow.org

NAND

a	b	~(a&b)	(a|b)	~(a&b) & (a|b)
0	0	1	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	1	0





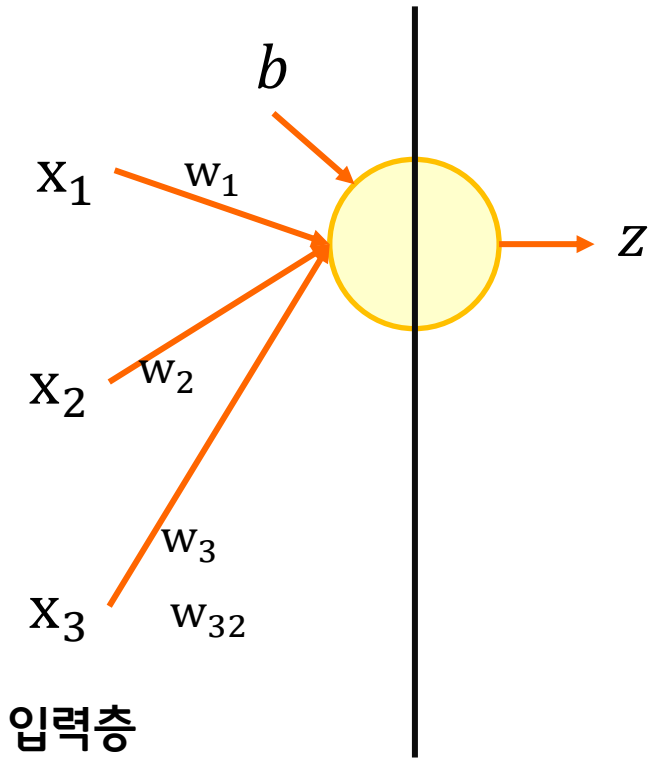
$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

$$w_1 = 0.7$$

$$w_2 = 0.7$$

$$b = -0.3$$

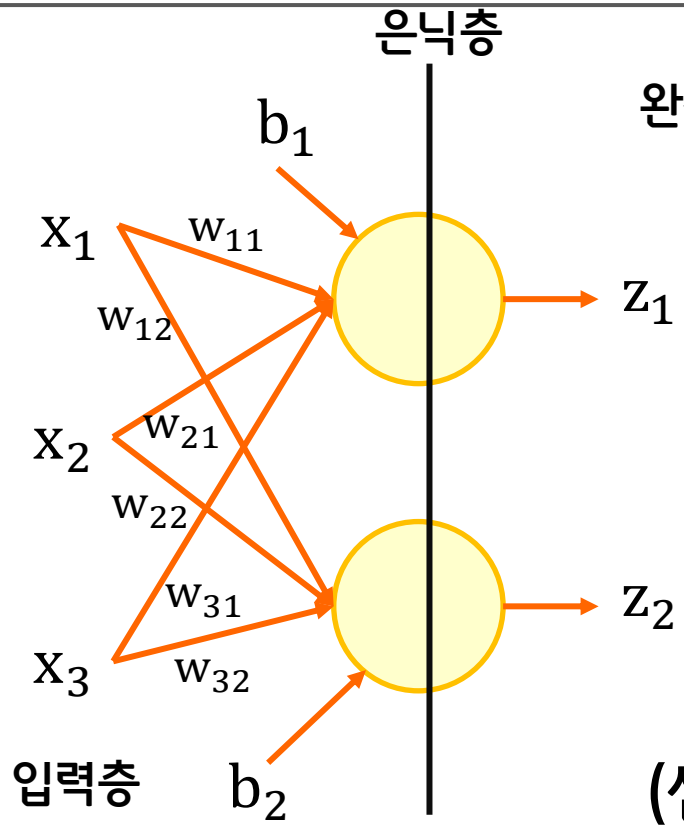
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



$$Xw + b = z$$

$$x_1w_1 + x_2w_2 + x_3w_3 + b = z$$

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} + b = z$$



완전 연결 신경망

$$XW + b = Z$$

(입력수,출력수)

affine연산:

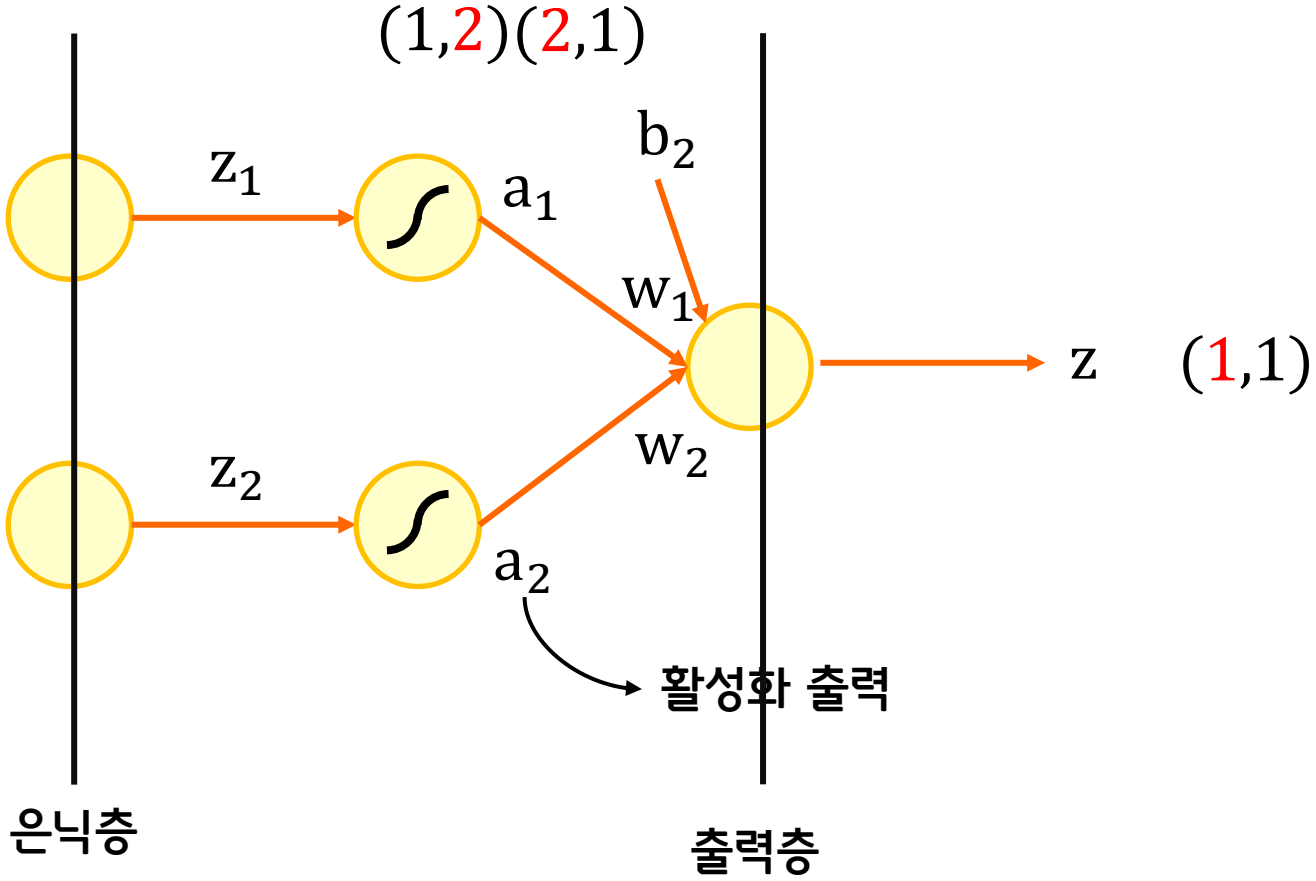
(샘플수,특성수)(특성수,뉴런수) => (샘플수,뉴런수)

(1,3) (3,2) (1,2)

$$x_1w_{11} + x_2w_{21} + x_3w_{31} + b_1 = z_1$$

$$x_1w_{12} + x_2w_{22} + x_3w_{32} + b_2 = z_2$$

$$[x_1 \quad x_2 \quad x_3]\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} + [b_1 \quad b_2] = [z_1 \quad z_2]$$

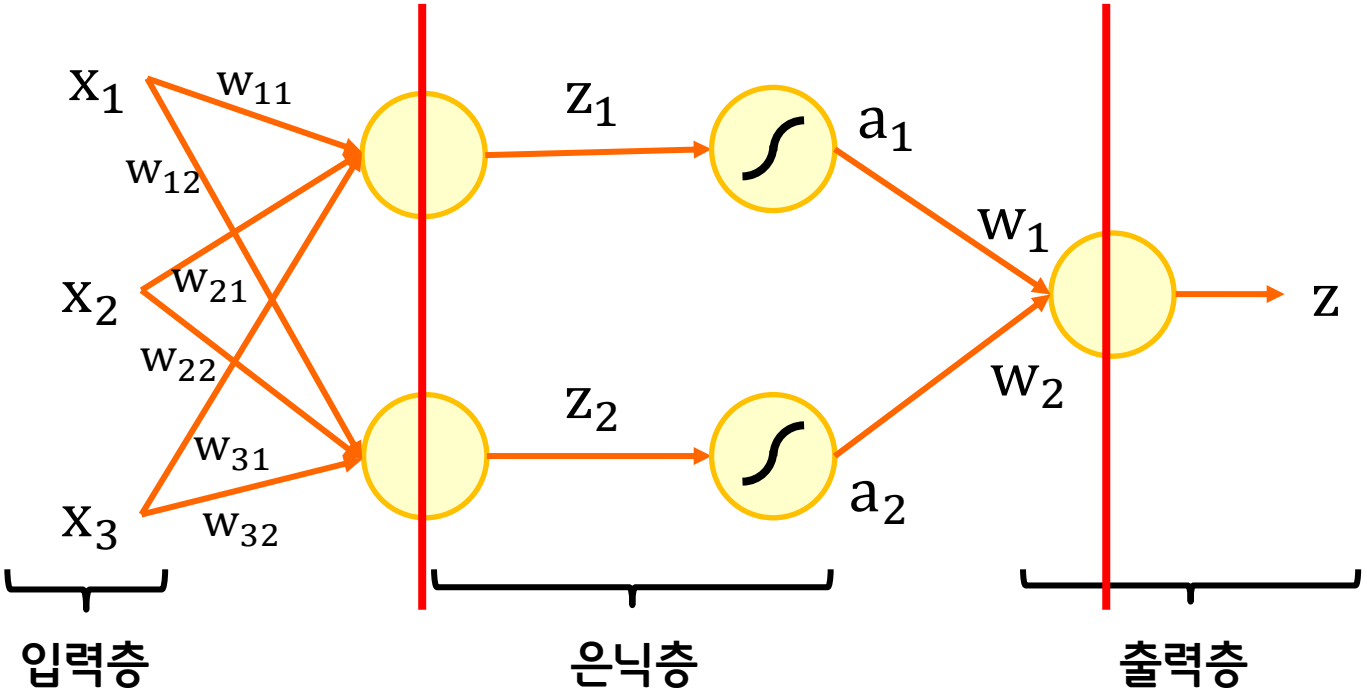


$$a_1 w_1 + a_2 w_2 + b_2 = z$$

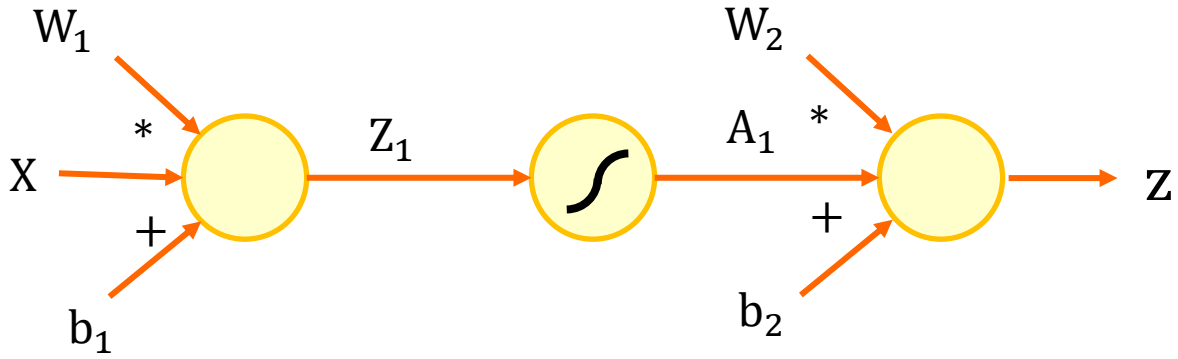
$$\begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + b_2 = z$$

$$A_1 W_2 + b_2 = z_2$$

하나의 층에 여러개의 뉴런 사용

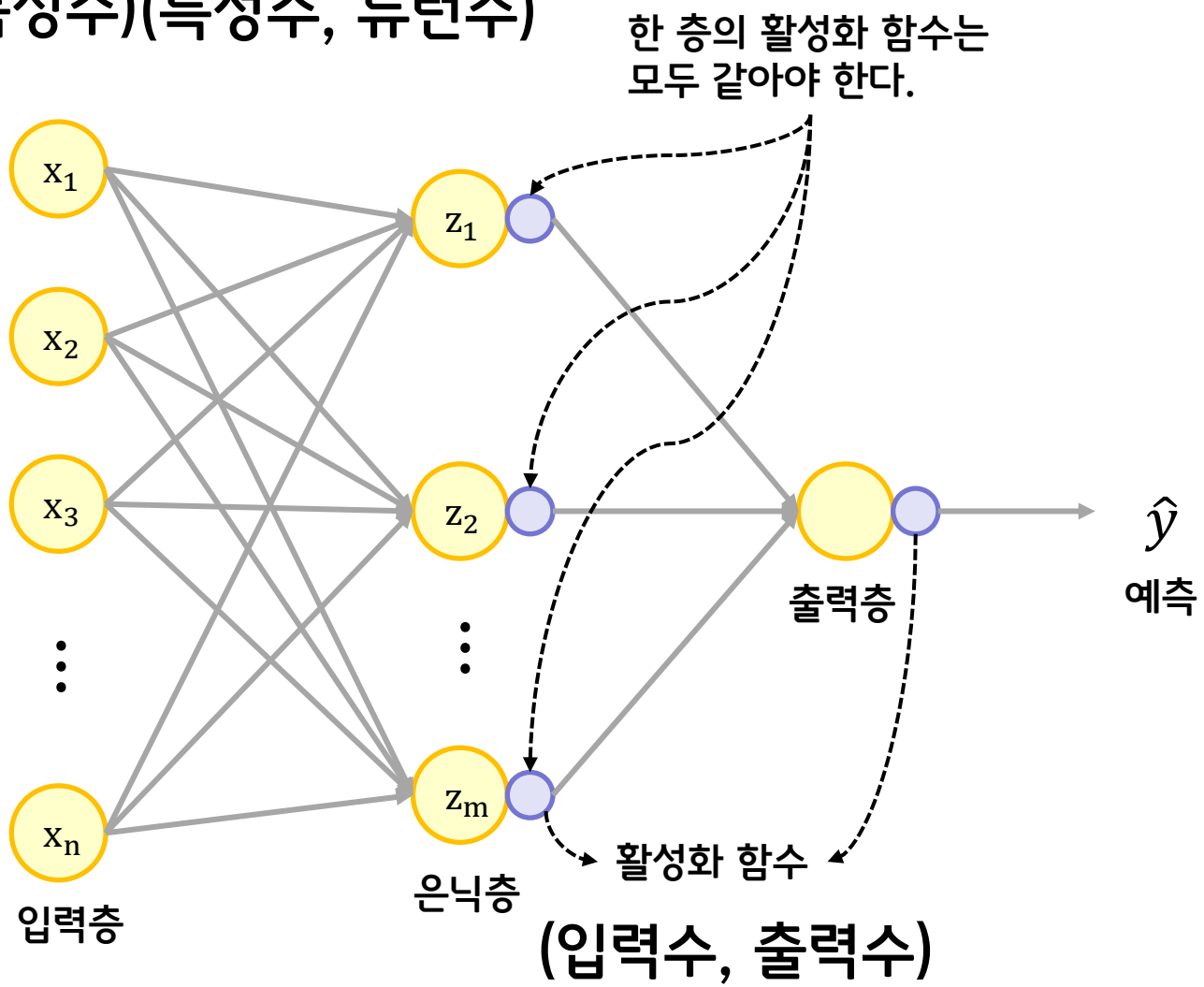


행렬로 표기

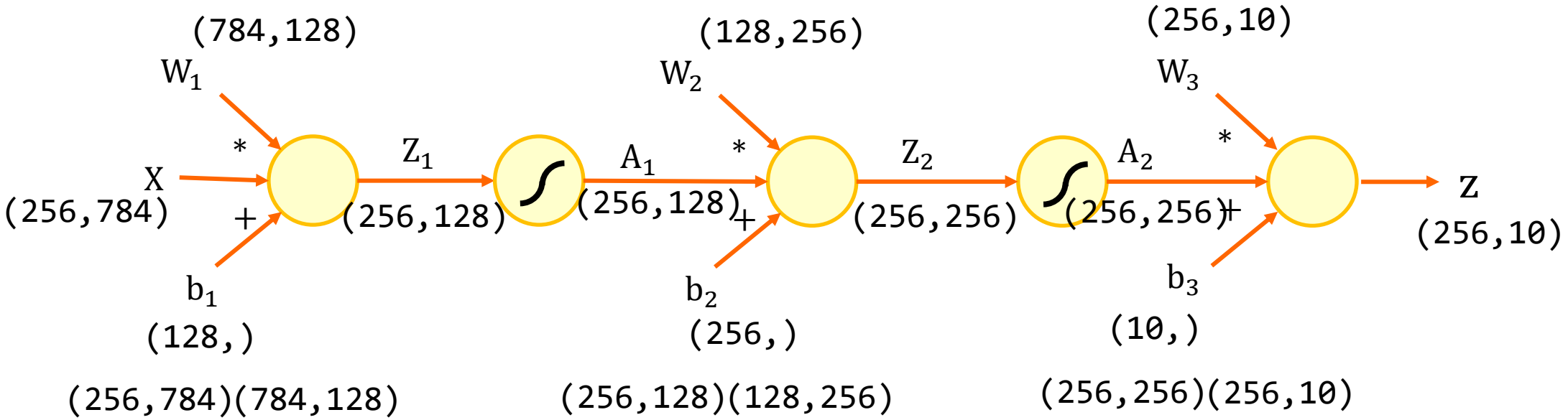


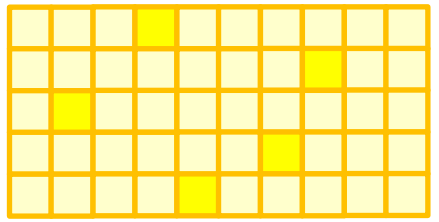
2층의 완전연결 신경망

(샘플수, 특성수)(특성수, 뉴런수)

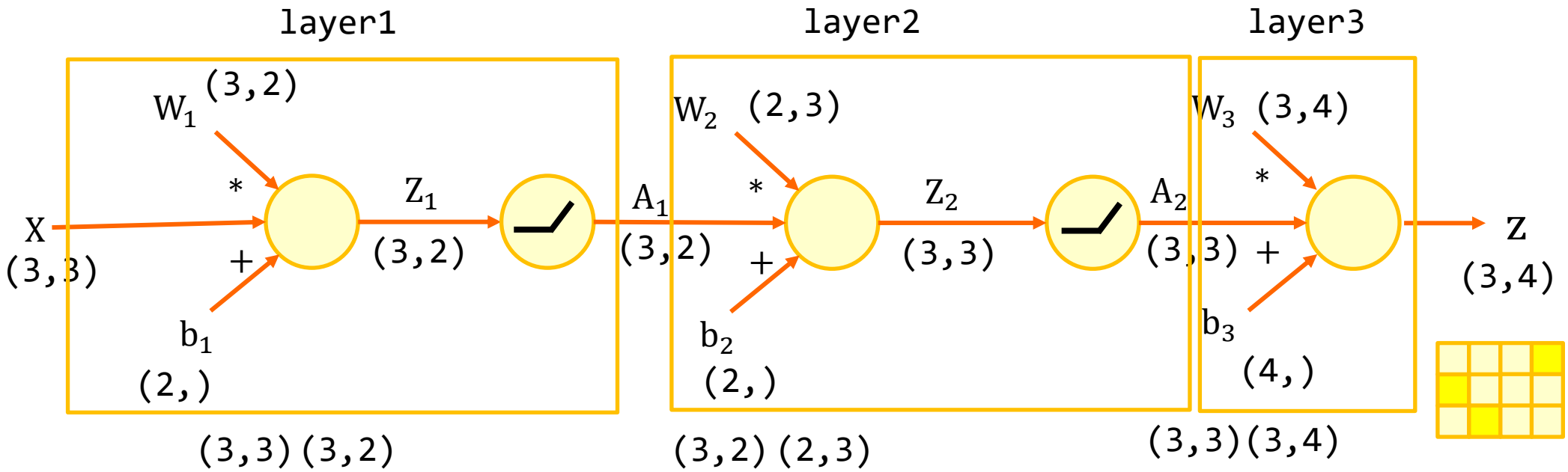


행렬로 표기


(784,128)
W1
*
X
(256,784)
+
b1
(128,)
Z1
(256,128)
A1
(256,128)
(128,256)
W2
*
+
b2
(256,)
Z2
(256,256)
A2
(256,256)
(256,10)
W3
*
+
b3
(10,)
Z
(256,10)
(256,784)(784,128)
(256,128)(128,256)
(256,256)(256,10)



행렬로 표기


layer1
layer2
layer3
X
(3,3)
W1
(3,2)
*
+
b1
(2,)
Z1
(3,2)
A1
(3,2)
W2
(2,3)
*
+
b2
(2,)
Z2
(3,3)
A2
(3,3)
W3
(3,4)
*
+
b3
(4,)
Z
(3,4)
(3,3)(3,2)
(3,2)(2,3)
(3,3)(3,4)

sepal_length	sepal_width	petal_length	petal_width	species
6.4	2.8	5.6	2.2	2
5.0	2.3	3.3	1.0	1
4.9	2.5	4.5	1.7	2
4.9	3.1	1.5	0.1	0
5.7	3.8	1.7	0.3	0

```
x_train.shape => (32,4)        y_train.shape => (32,)

[[6.4 2.8 5.6 2.2]
 [5.  2.3 3.3 1. ]
 [4.9 2.5 4.5 1.7]
 [4.9 3.1 1.5 0.1]
 [5.7 3.8 1.7 0.3]]
```

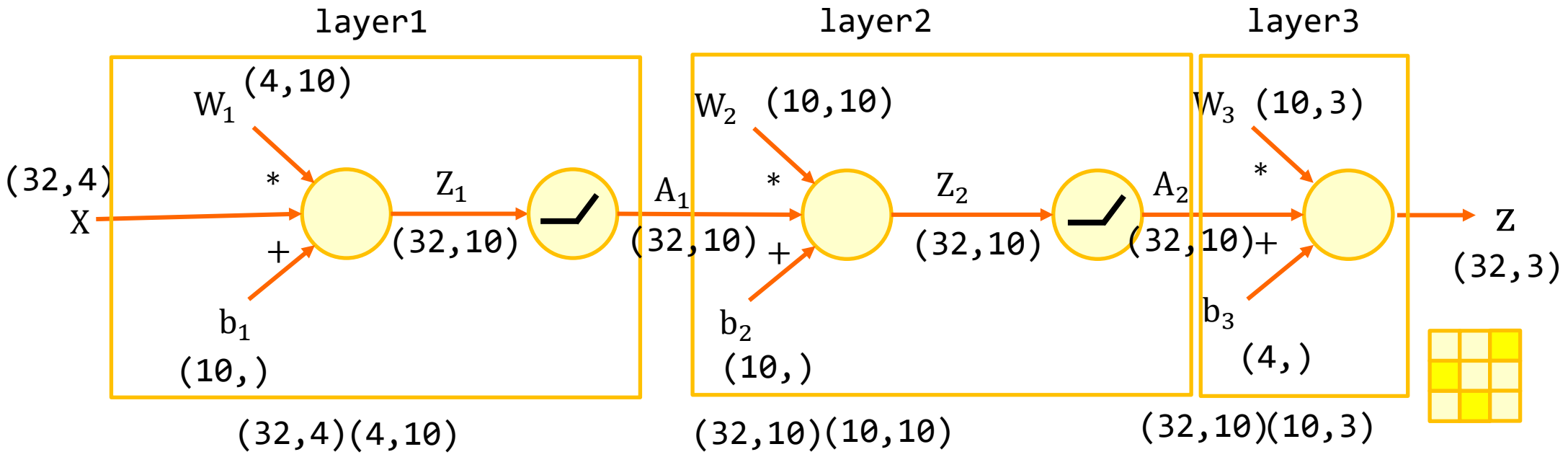
pydot 설치

아나콘다 파워 쉘 실행

```
(base) PS F:\텐서플로우_수업자료\동영상_컨텐츠\TF2-AI-main> conda install pydot
(base) PS F:\텐서플로우_수업자료\동영상_컨텐츠\TF2-AI-main> jupyter notebook
```

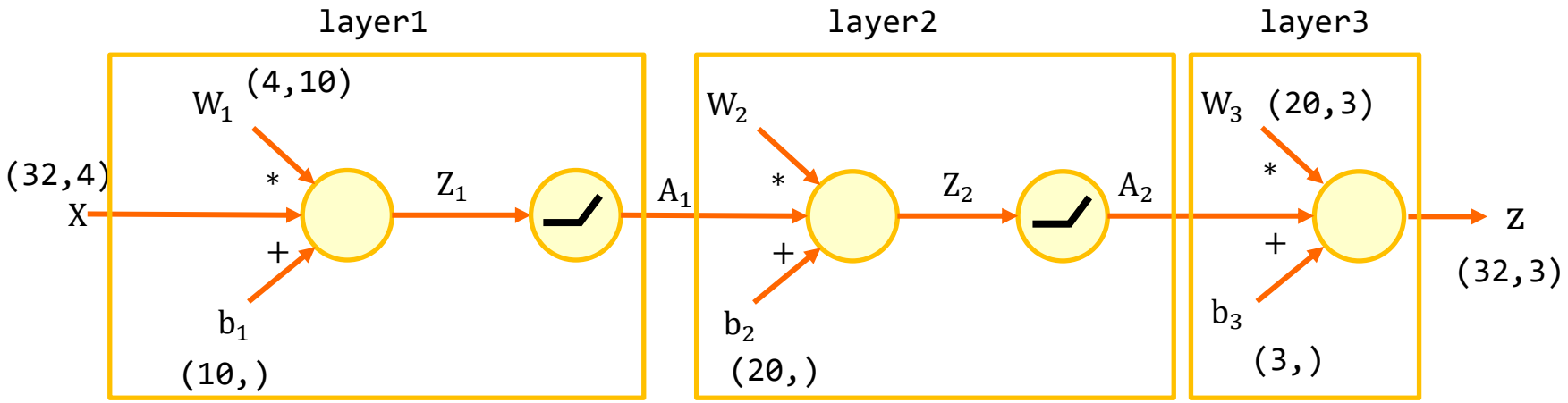
행렬로 표기

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,)),
  tf.keras.layers.Dense(10, activation=tf.nn.relu),
  tf.keras.layers.Dense(3)
])
```



행렬로 표기

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,)),
  tf.keras.layers.Dense(20, activation=tf.nn.relu),
  tf.keras.layers.Dense(3)
])
```



합성곱 연산

277

183

flatten

(1,183,277,3) => (1,152073)

(1,152073)(152073,256)

277

183

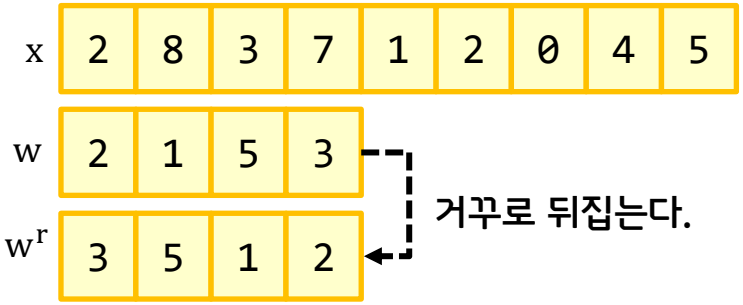
filter(weight) : (27,18,3,1)

(1,183,277,3)

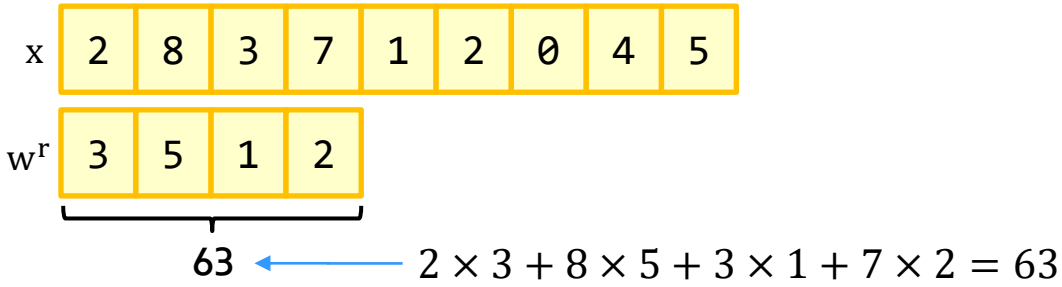
고양이:0

합성곱 연산

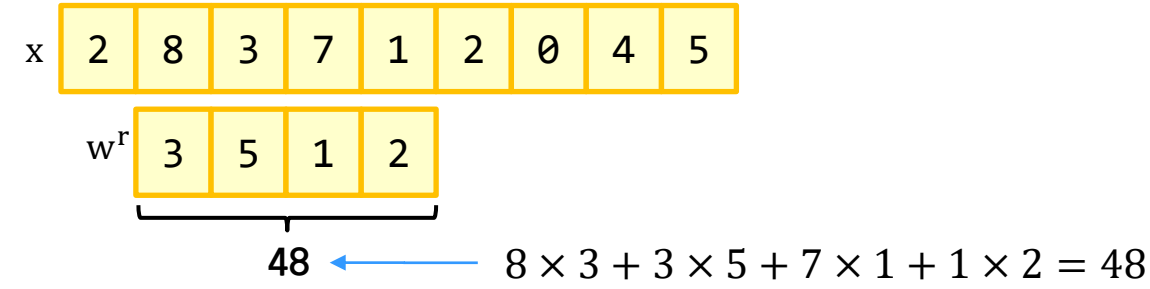
배열 하나 선택해 뒤집기



첫 번째 합성곱

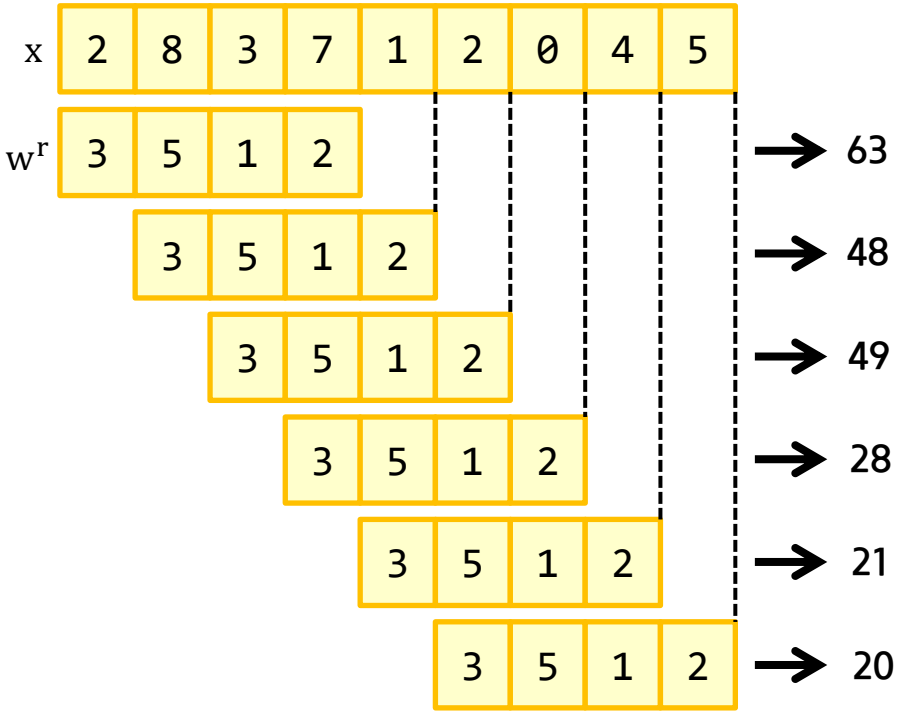


두 번째 합성곱



합성곱 연산

전체 합성곱



[63]
[48]
[49]
[28]
[21]
[20]

합성곱 연산

합성곱 구현

```
import numpy as np
x = np.array([2, 8, 3, 7, 1, 2, 0, 4, 5])
w = np.array([2, 1, 5, 3])
```

flip() 함수를 이용한 배열 뒤집기

```
w_r = np.flip(w)
print(w_r)
```

[3 5 1 2]

넘파이의 점 곱으로 합성곱 연산

```
for i in range(6):
    print(np.dot(x[i:i+4], w_r.reshape(-1,1)))
```

[63]
[48]
[49]
[28]
[21]
[20]

$$\begin{bmatrix} 2 & 8 & 3 & 7 \\ 8 & 3 & 7 & 1 \\ 3 & 7 & 1 & 2 \\ 7 & 1 & 2 & 0 \\ 1 & 2 & 0 & 4 \\ 2 & 0 & 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 63 \\ 48 \\ 49 \\ 28 \\ 21 \\ 20 \end{bmatrix}$$

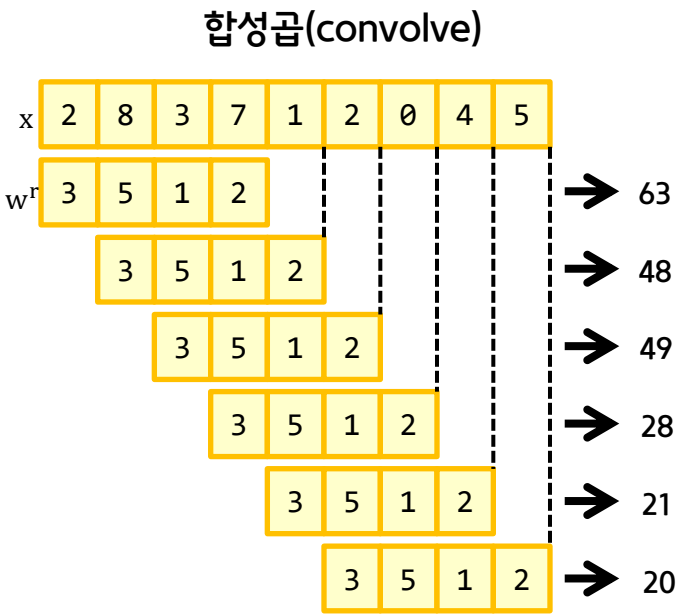
합성곱 연산

싸이파이로 합성곱 수행

```
from scipy.signal import convolve
convolve(x, w, mode='valid')
```

array([63, 48, 49, 28, 21, 20])

합성곱 신경망은 진짜 합성곱을 사용하지 않는다.
합성곱 대신 교차상관을 사용한다.



합성곱 연산

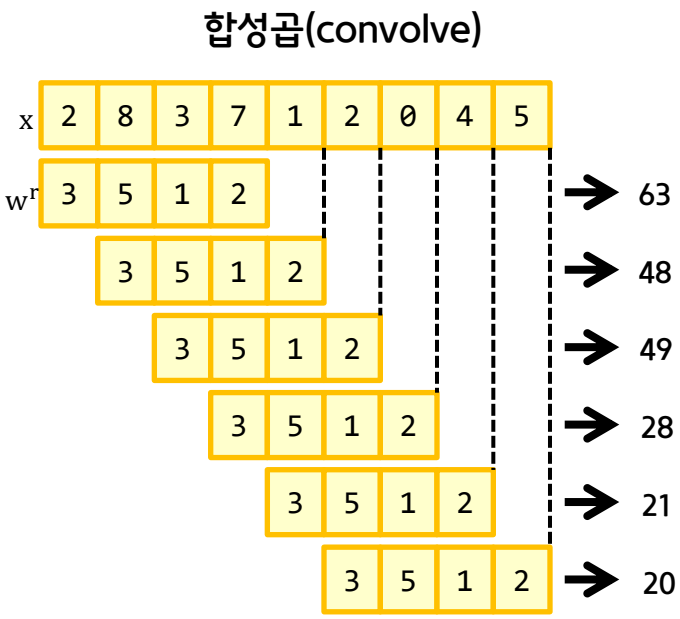
싸이파이로 교차상관 수행

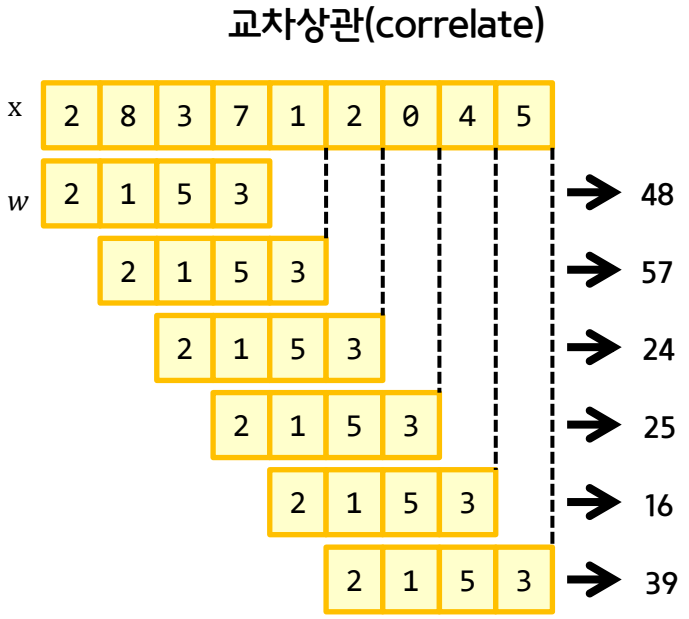
```
from scipy.signal import correlate
correlate(x, w, mode='valid')
```

array([48, 57, 24, 25, 16, 39])

합성곱 신경망은 진짜 합성곱을 사용하지 않는다.
합성곱 대신 교차상관을 사용한다.

(N-F+1) (9-4+1)

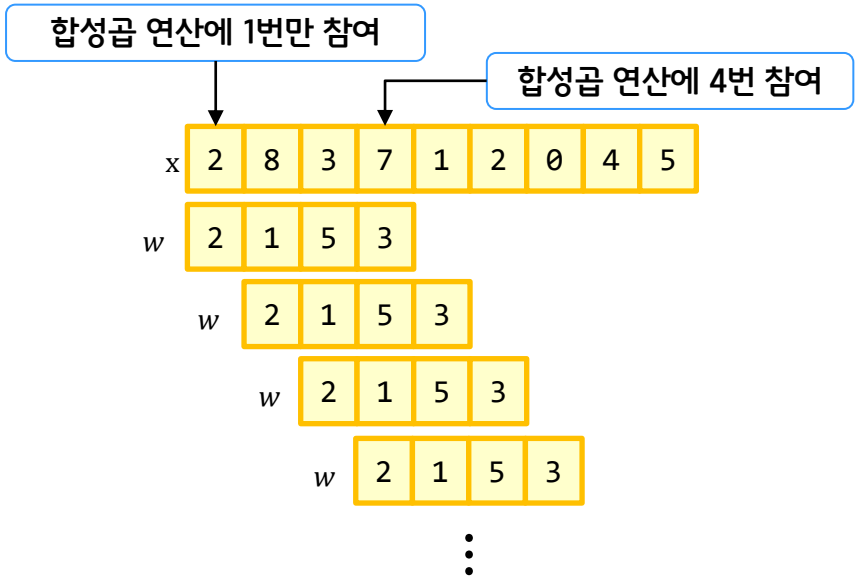




합성곱 연산

패딩과 스트라이드 이해

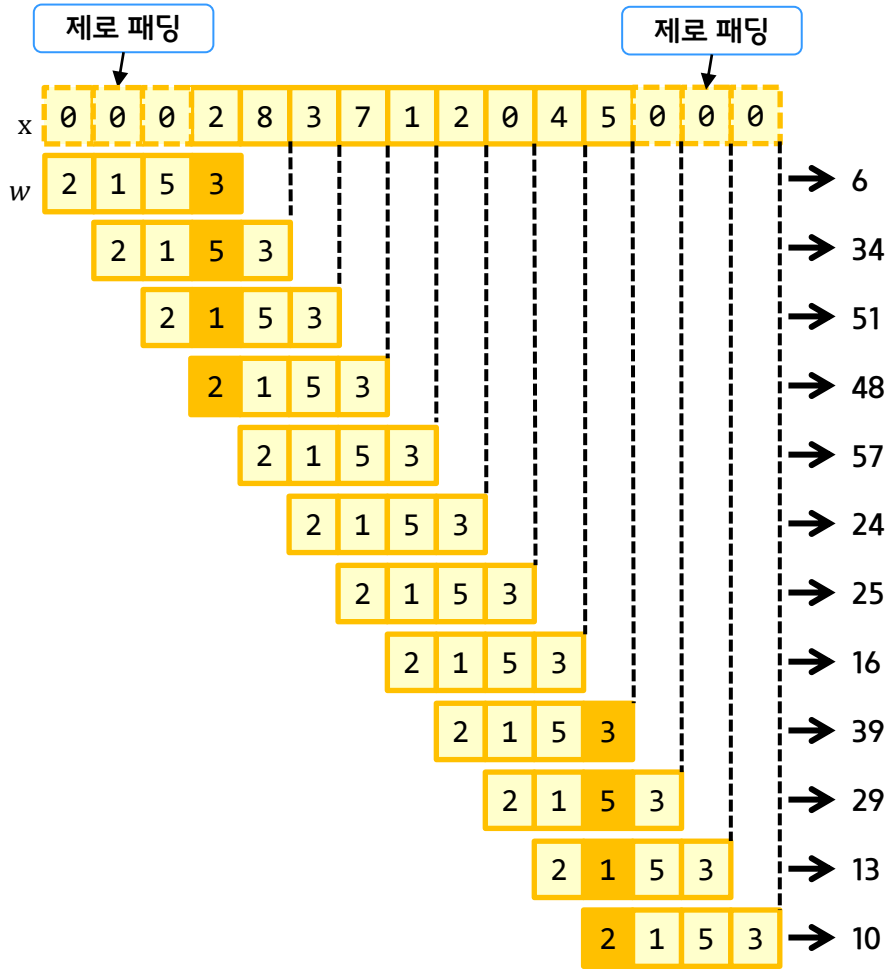
밸리드 패딩은 원본 배열의 원소가 합성곱 연산에 참여하는 정도가 다르다.



합성곱 연산

패딩과 스트라이드 이해

풀 패딩은 원본 배열의 원소의 연산 참여도를 동일하게 만든다.




```
correlate(x, w, mode='full')
```

array([6, 34, 51, 48, 57, 24, 25, 16, 39, 29, 13, 10])

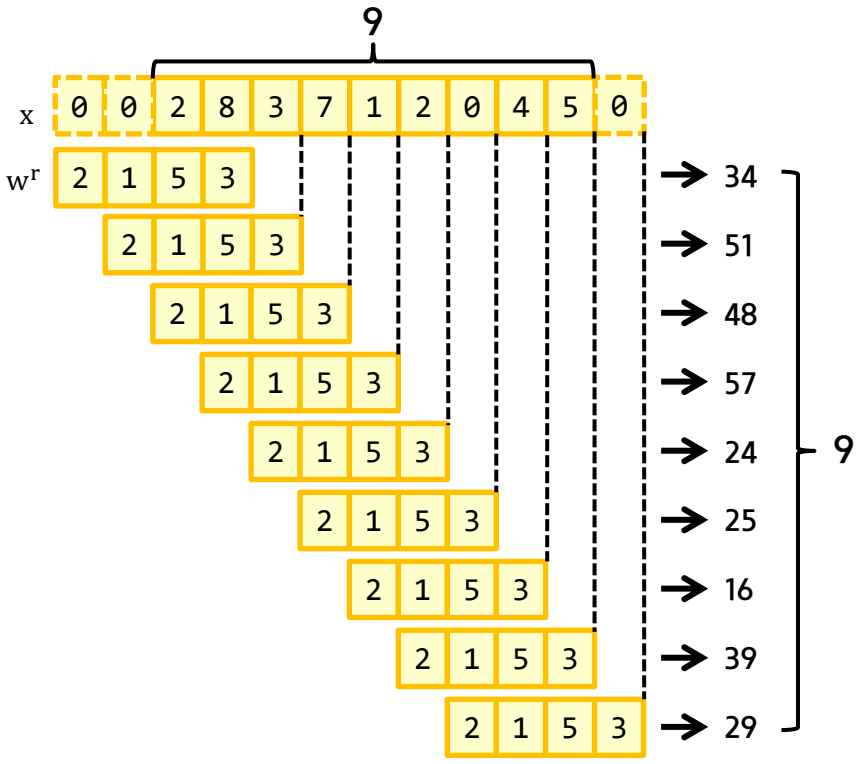
N-F+1+2P

9-4+1+2*3 => 12

합성곱 연산

패딩과 스트라이드 이해

세임 패딩은 출력 배열의 길이를 원본 배열의 원소의 길이와 동일하게 만든다.




```
correlate(x, w, mode='same')
```

```
array([34, 51, 48, 57, 24, 25, 16, 39, 29])
```

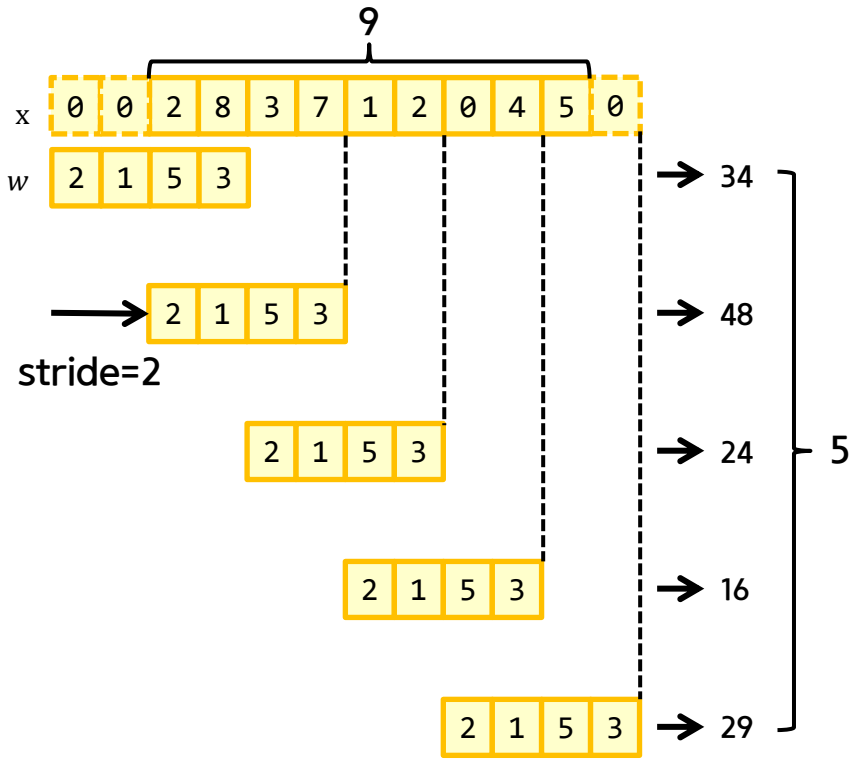
N-F+1+P

9-4+1+3 => 9

합성곱 연산

패딩과 스트라이드 이해

스트라이드는 미끄러지는 간격을 조정한다.




```
correlate(x, w, mode='same')
```

array([34, 51, 48, 57, 24, 25, 16, 39, 29])

O=N-F+P+1

9=9-4+3+1

O=(N-F+P)/S+1

5=(9-4+3)/2+1

합성곱 연산

2차원 배열에서 합성곱 수행 (mode='valid')

x

1	2	3
4	5	6
7	8	9

w

2	0
0	0

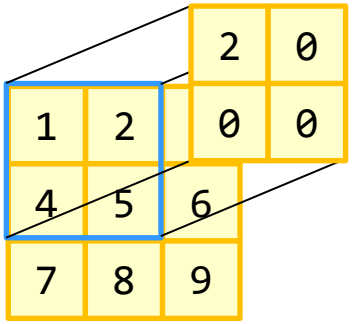
O

2	4
8	10

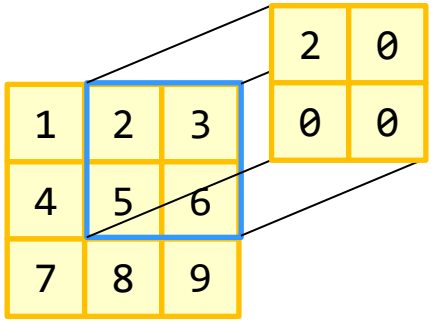
```
x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
w = np.array([[2, 0], [0, 0]])
from scipy.signal import correlate2d
correlate2d(x, w, mode='valid')
```

```
array([[ 2,  4],
       [ 8, 10]])
```

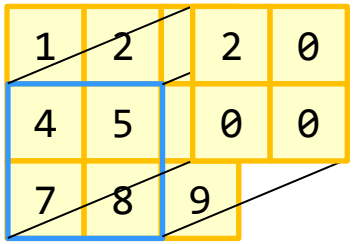
N-F+1 2=3-2+1



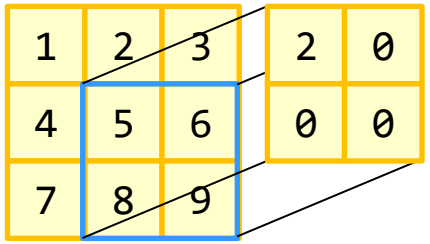
=> 2



=> 4



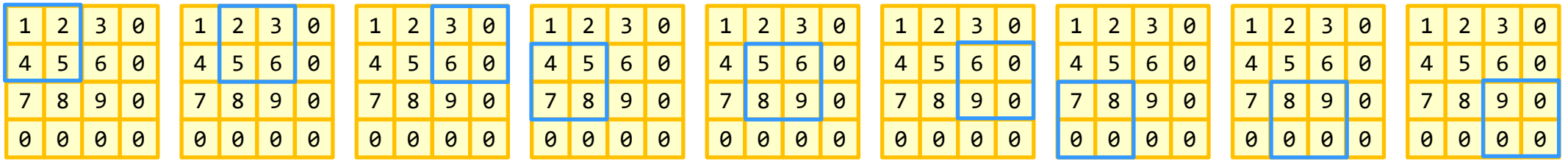
=> 8



=> 10

합성곱 연산

2차원 배열에서 same padding

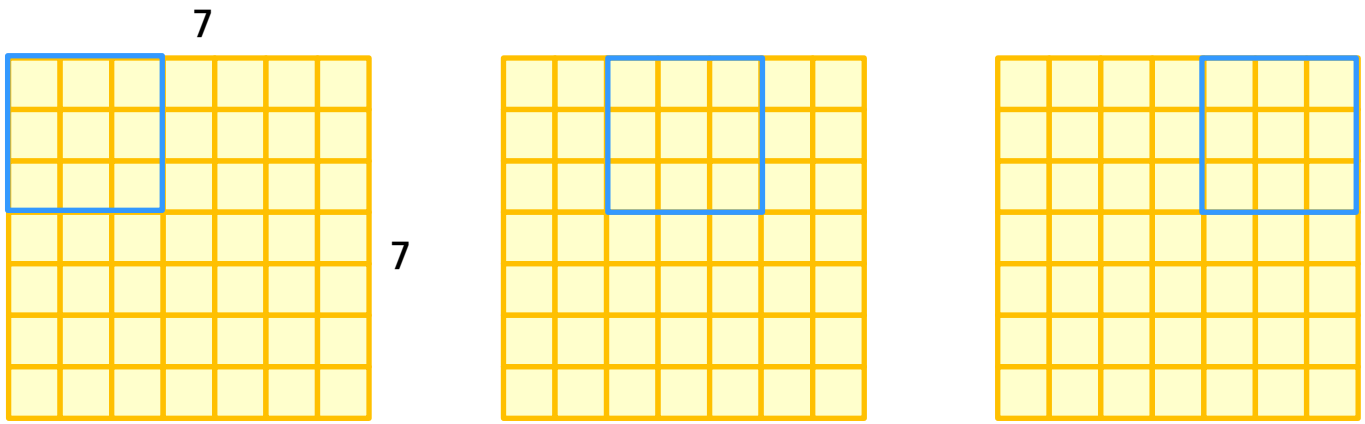



```
correlate2d(x, w, mode='same')
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

합성곱 연산

2차원 배열에서 스트라이드 이해



3 = (7-3)/2+1

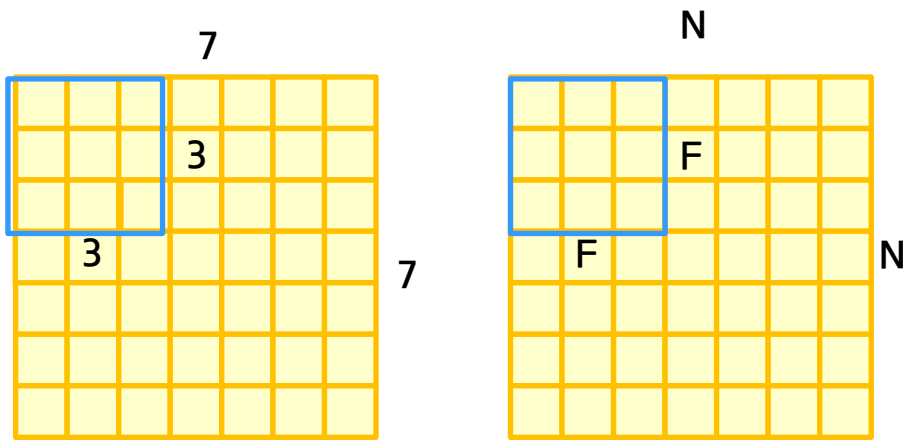
7x7 input
3x3 filter

stride = 2
mode = valid

=> 3x3 output

합성곱 연산

2차원 배열에서 스트라이드 이해



Output size :
(N - F) / stride + 1

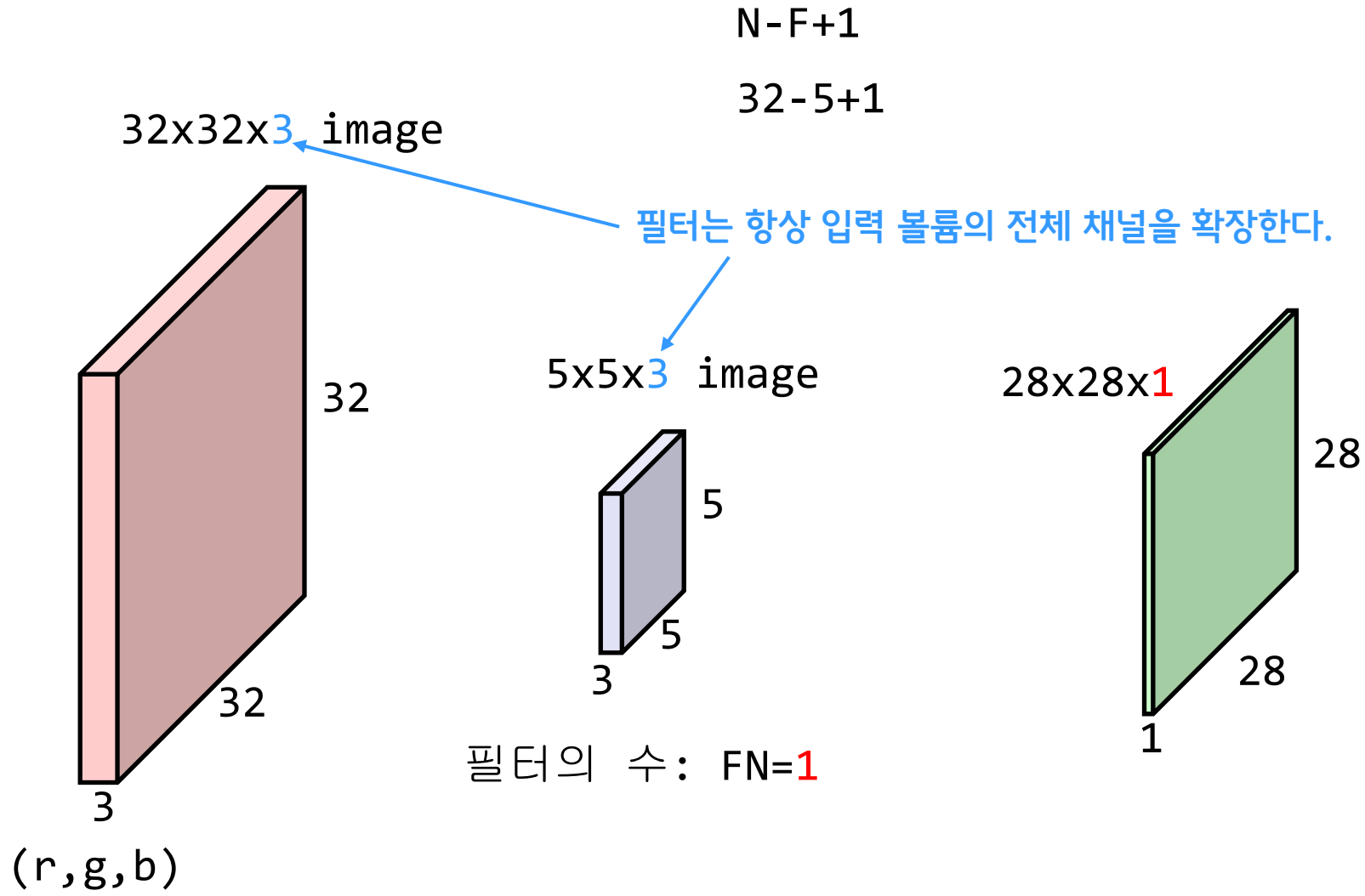
예) N = 7, F = 3
stride 1 => (7-3)/1+1 = 5
stride 2 => (7-3)/2+1 = 3
stride 3 => (7-3)/3+1 = 2.33

합성곱 연산

Convolution Layer

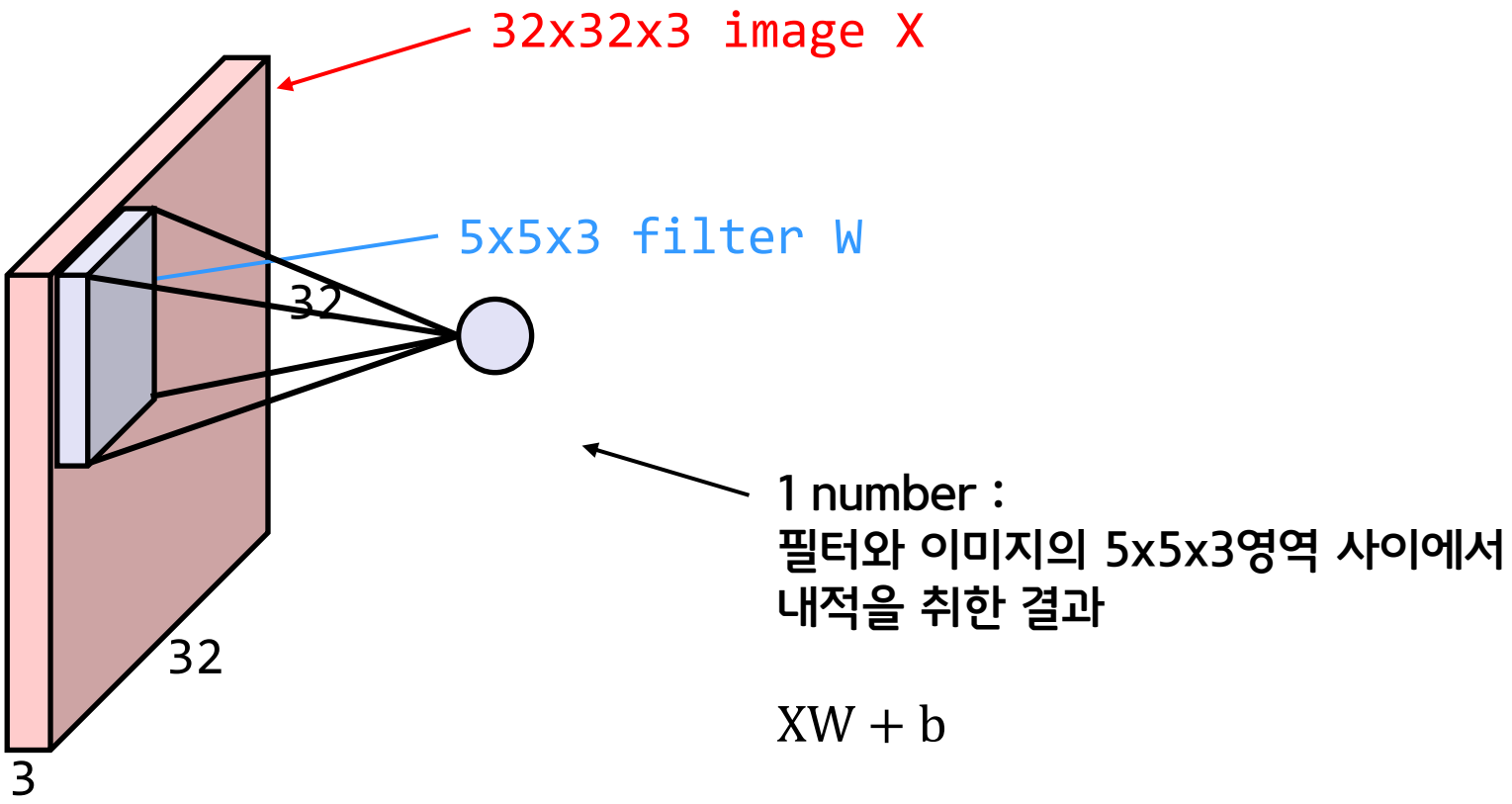
N-F+1

32-5+1



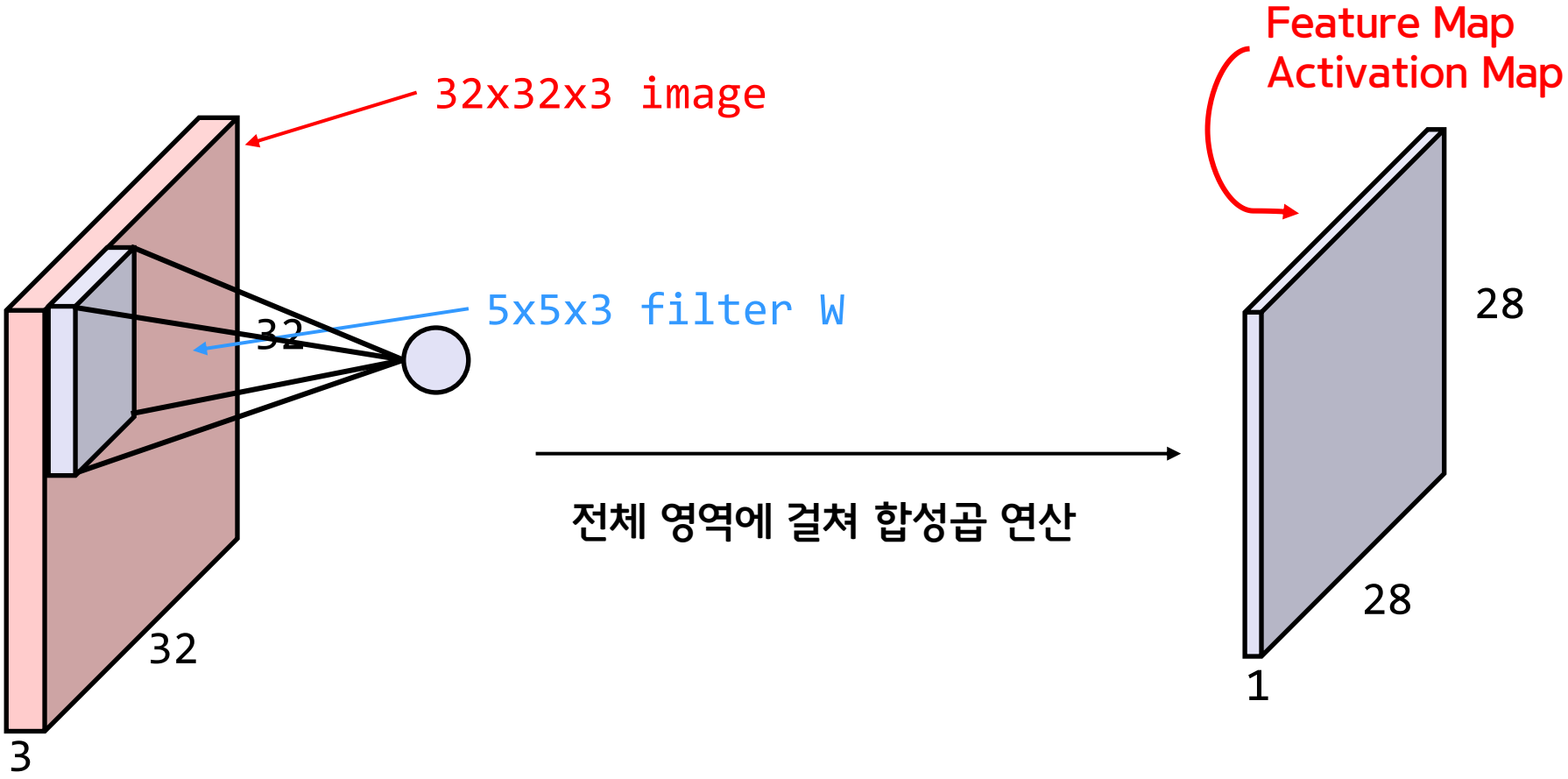
합성곱 연산

Convolution Layer

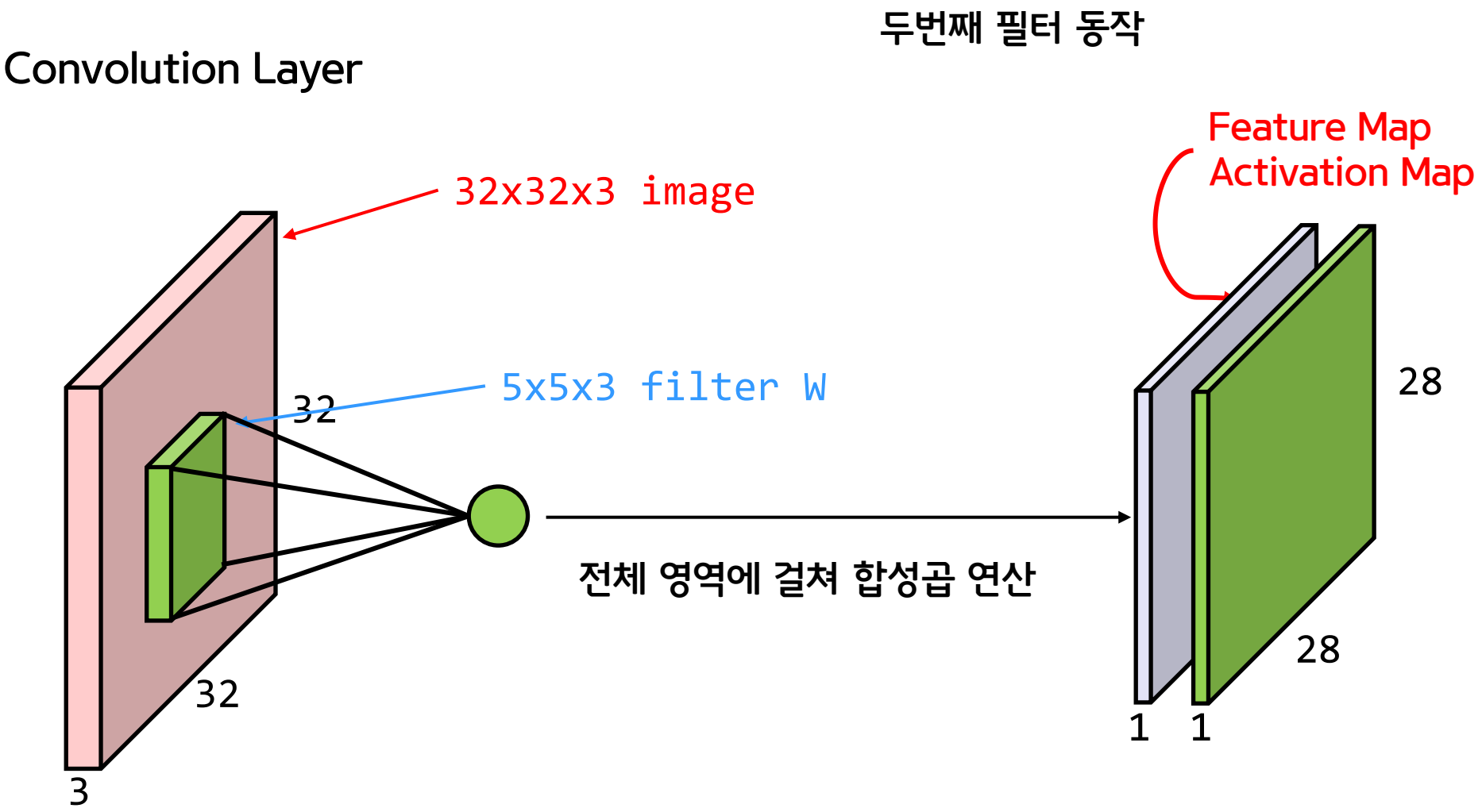


합성곱 연산

Convolution Layer



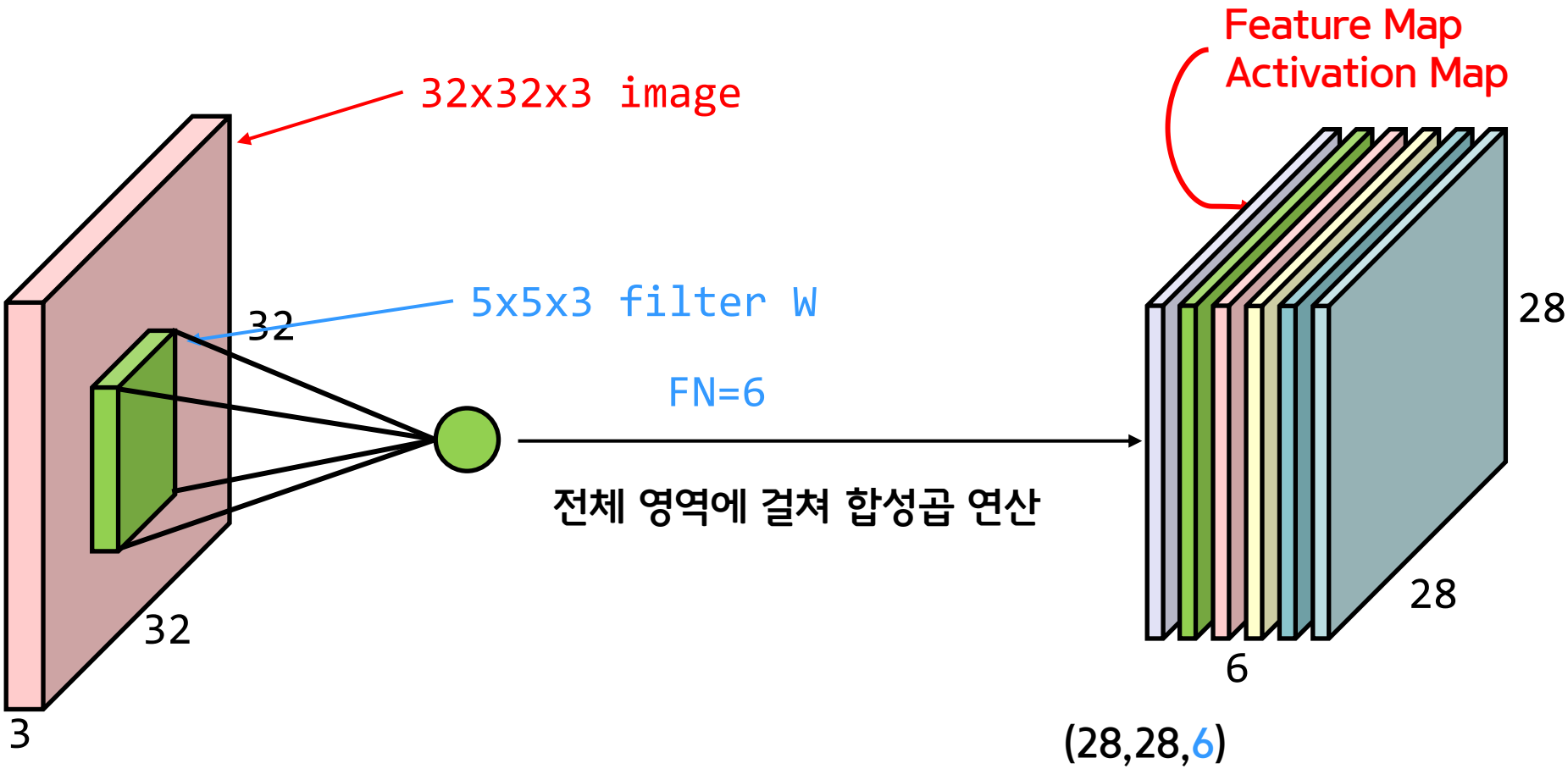
합성곱 연산


두번째 필터 동작
Convolution Layer
Feature Map
Activation Map
32x32x3 image
5x5x3 filter W
32
32
3
전체 영역에 걸쳐 합성곱 연산
28
28
1
1

합성곱 연산

Convolution Layer

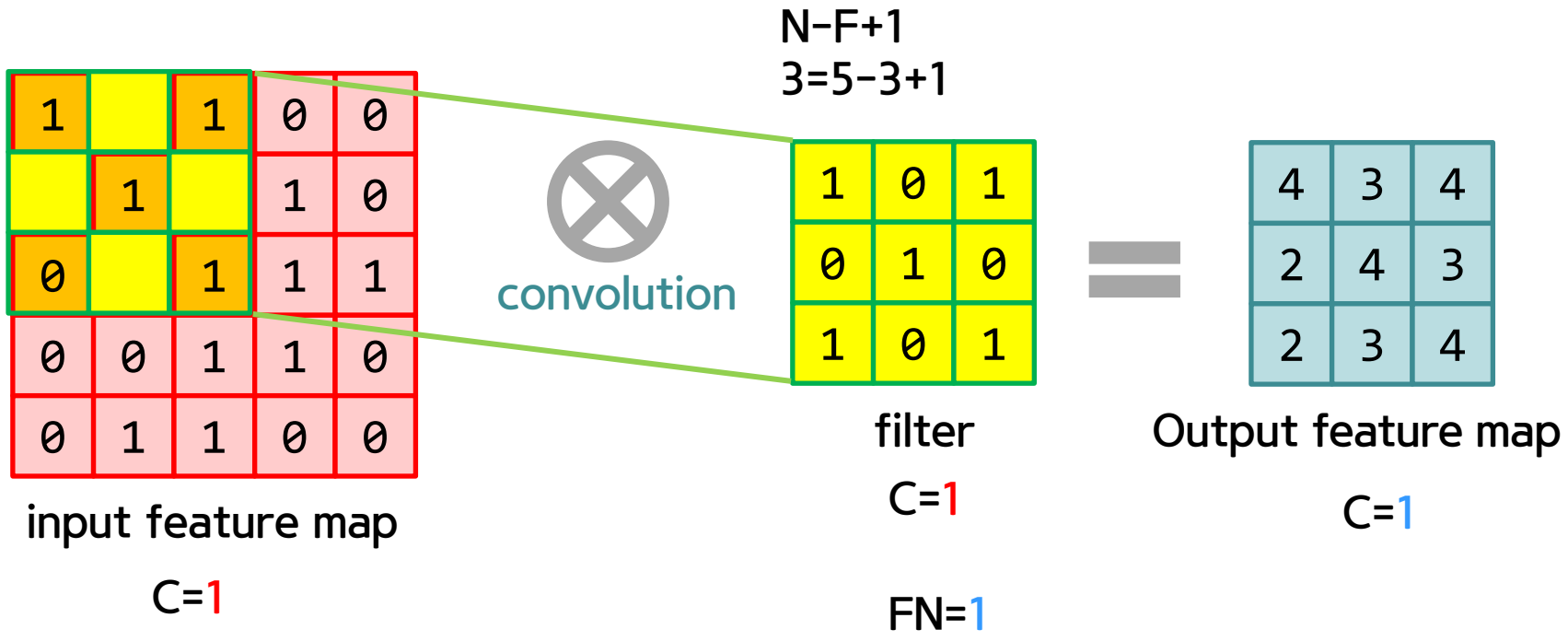
6개의 5x5필터가 있다면, 6개의 개별 feature map이 생성됨



전체 영역에 걸쳐 합성곱 연산

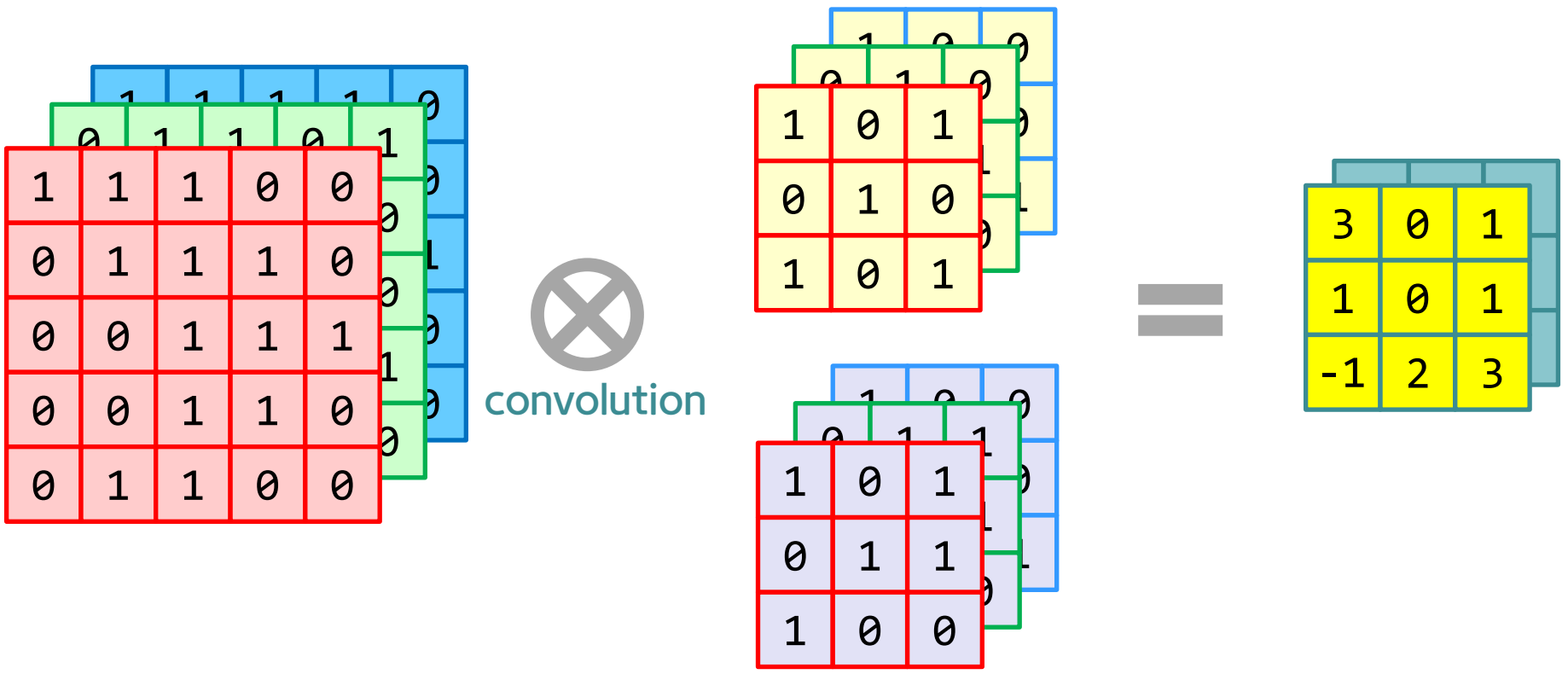
Convolution Layer - 계산

1x1 + 1x0 + 1x1 + 0x0 + 1x1 + 1x0 + 0x1 + 0x0 +1x1 = 4



합성곱 연산

Convolution Layer - 계산

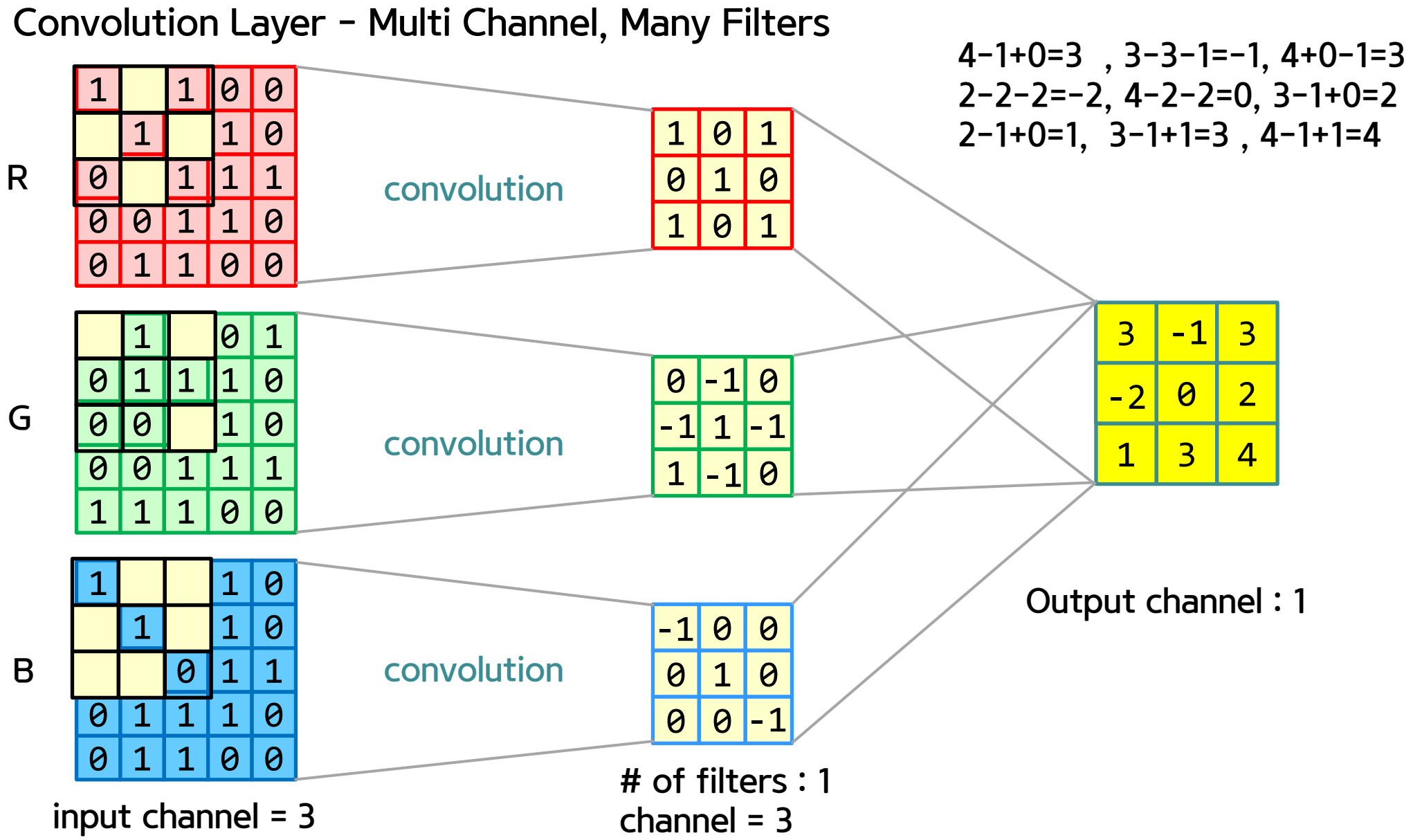


input channel = 3

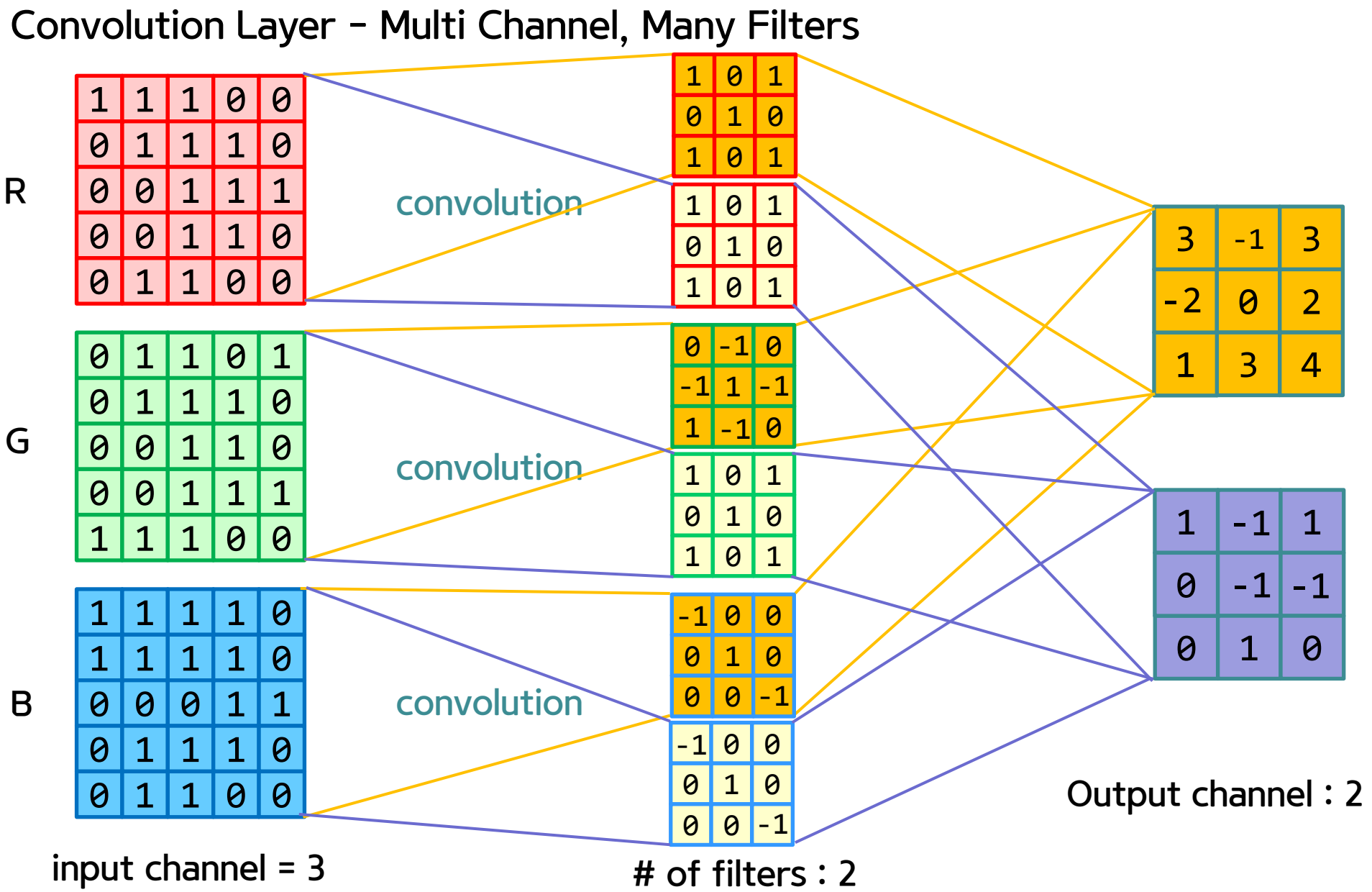
# of filters : 2
channel=3

Output channel : 2

합성곱 연산

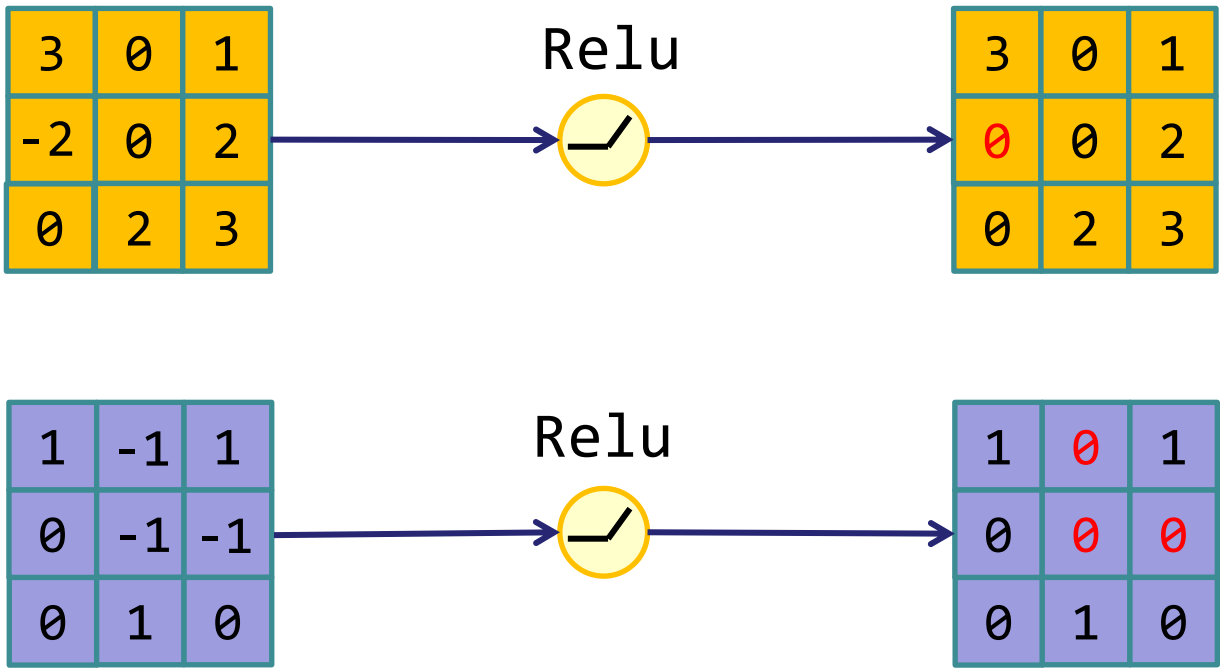

Convolution Layer - Multi Channel, Many Filters
R
G
B
convolution
convolution
convolution
4-1+0=3 , 3-3-1=-1, 4+0-1=3
2-2-2=-2, 4-2-2=0, 3-1+0=2
2-1+0=1, 3-1+1=3 , 4-1+1=4
Output channel : 1
input channel = 3
# of filters : 1
channel = 3

합성곱 연산


Convolution Layer - Multi Channel, Many Filters
R
G
B
convolution
convolution
convolution
input channel = 3
# of filters : 2
Output channel : 2

합성곱 연산

Activation Function



합성곱 연산

tf.keras.layers.Conv2D

Arguments:

- **`filters`**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

- **`kernel_size`**: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

- **`strides`**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.

- **`padding`**: one of `"valid"` or `"same"` (case-insensitive).

- **`data_format`**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch_size, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch_size, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

합성곱 연산

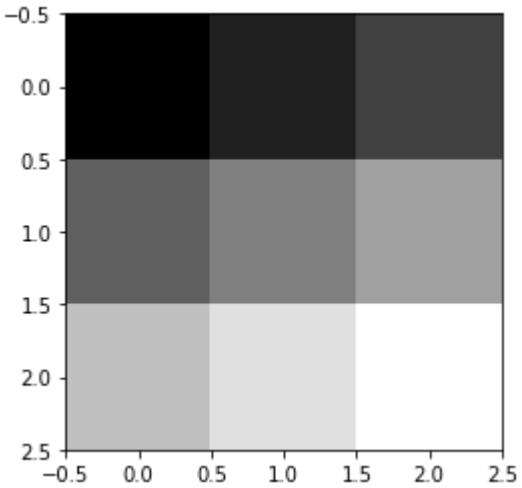
tf.keras.layers.Conv2D

- **activation**: Activation function to use. If you don't specify anything, no activation is applied (see keras.activations).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix (see keras.initializers).
- **bias_initializer**: Initializer for the bias vector (see keras.initializers).
- **kernel_regularizer**: Regularizer function applied to the kernel weights matrix (see keras.regularizers).
- **bias_regularizer**: Regularizer function applied to the bias vector (see keras.regularizers).

합성곱 연산

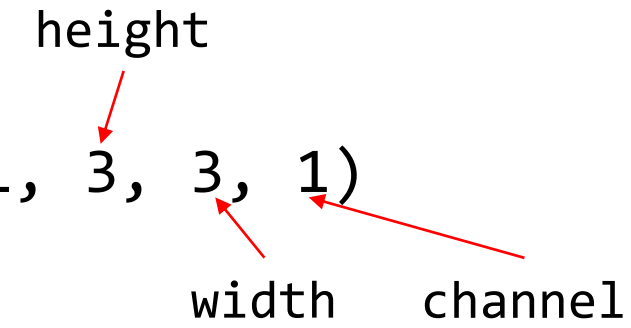
```
import tensorflow as tf
import numpy as np
import keras
from keras.layers import *
import matplotlib.pyplot as plt
image = tf.constant([[[[1],[2],[3]],
                      [[4],[5],[6]],
                      [[7],[8],[9]]]], dtype=np.float32)
print(image.shape)
plt.imshow(image.numpy().reshape(3,3), cmap='gray')
```

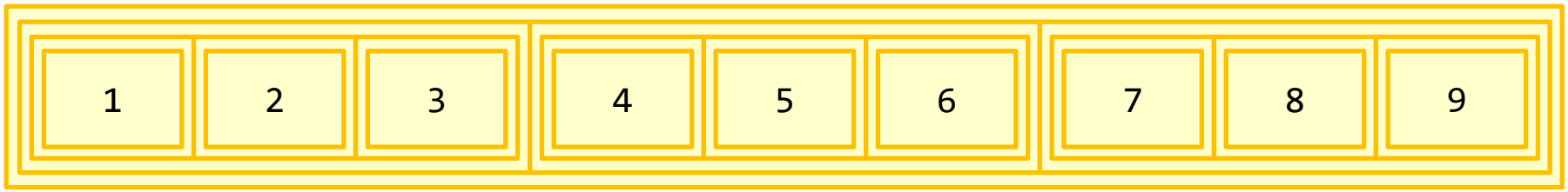
```
(1, 3, 3, 1)
```

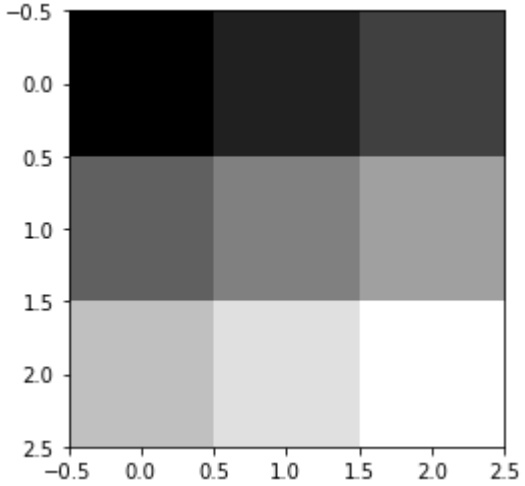


합성곱 연산

```
[[[[1],[2],[3]],
  [[4],[5],[6]],
  [[7],[8],[9]]]]
```







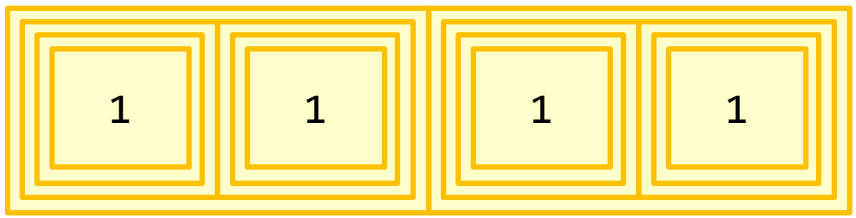
1	2	3
4	5	6
7	8	9

합성곱 연산

```
print("image.shape", image.shape)
weight = np.array([[[[1.]],[[1.]]],[[[1.]],[[1.]]]])
print("weight.shape", weight.shape)
weight_init = tf.constant_initializer(weight)
```

(2, 2, 1, 1)

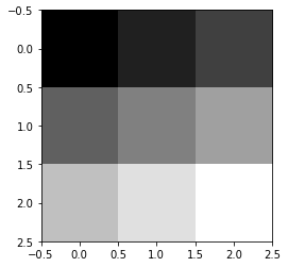
height width



image

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

(1,3,3,1)



weight

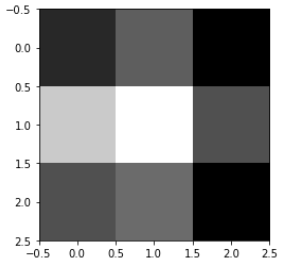
1	1
1	1

(2,2,1,1)

conv2d

12	16	9
24	28	15
15	17	9

(1,3,3,1)



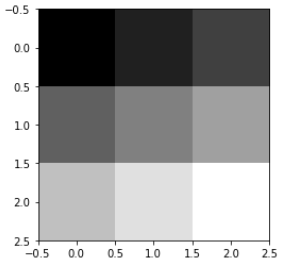
합성곱 연산

```
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2,
padding='same', kernel_initializer=weight_init)(image)
```

image

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

(1,3,3,1)



weight

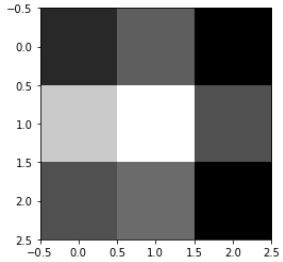
1	1
1	1

(2,2,1,1)

conv2d

12	16	9
24	28	15
15	17	9

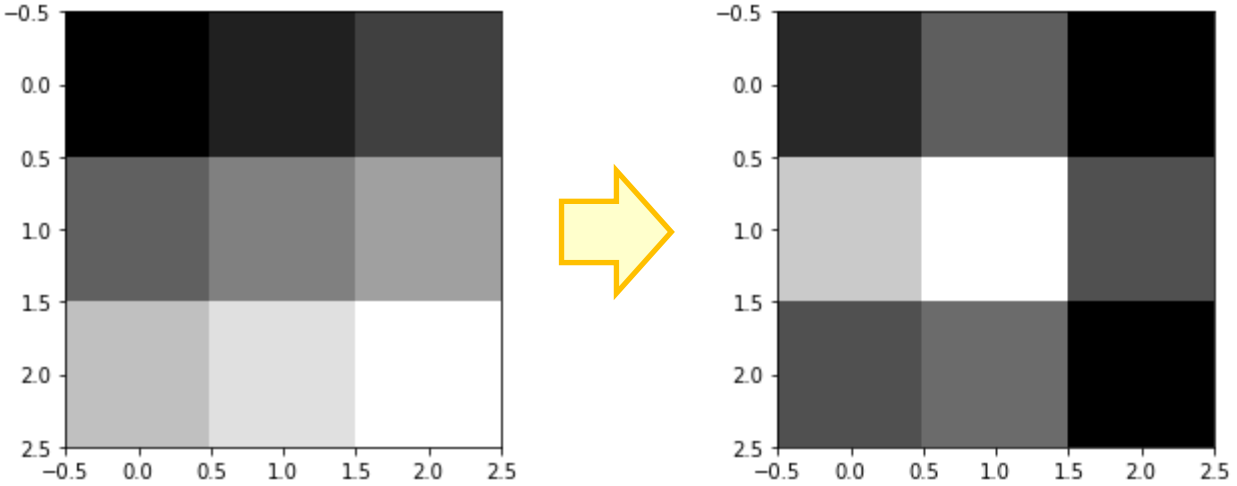
(1,3,3,1)



합성곱 연산

```
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2,
padding='same', kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
print(conv2d.numpy().reshape(3,3))
plt.imshow(conv2d.numpy().reshape(3,3), cmap='gray')
plt.show()
```

```
conv2d.shape (1, 3, 3, 1)
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]
```



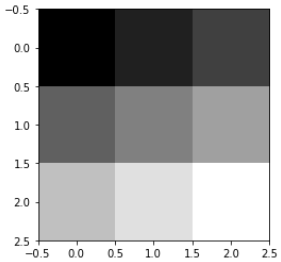
합성곱 연산

```
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2,
padding='valid', kernel_initializer=weight_init)(image)
```

image

1	2	3
4	5	6
7	8	9

(1,3,3,1)



weight

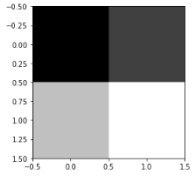
1	1
1	1

(2,2,1,1)

conv2d

12	16
24	28

(1,2,2,1)

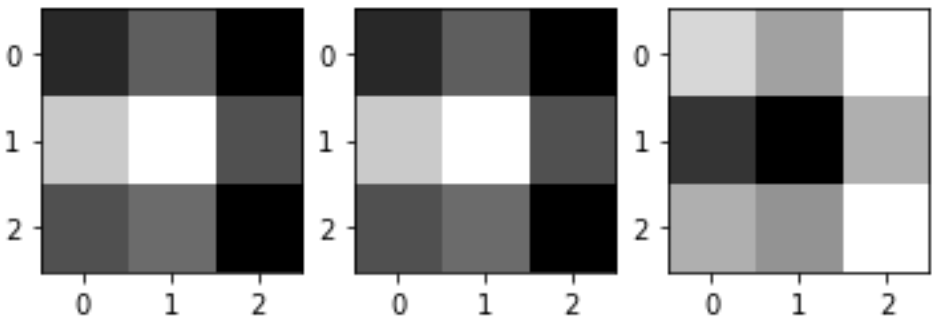


N-F+1

합성곱 연산

```
print("image.shape", image.shape)
weight = np.array([[[[1.,10.,-1.]],[[1.,10.,-1.]]],[[[1.,10.,-1.]],[[1.,10.,-1.]]]])
print("weight.shpe", weight.shape)
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=3, kernel_size=2, padding='same',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
feature_maps = np.swapaxes(conv2d, 0, 3)
for i, feature_map in enumerate(feature_maps):
    print(feature_map.reshape(3,3))
    plt.subplot(1,3,i+1), plt.imshow(feature_map.reshape(3,3), cmap='gray')
plt.show()
```

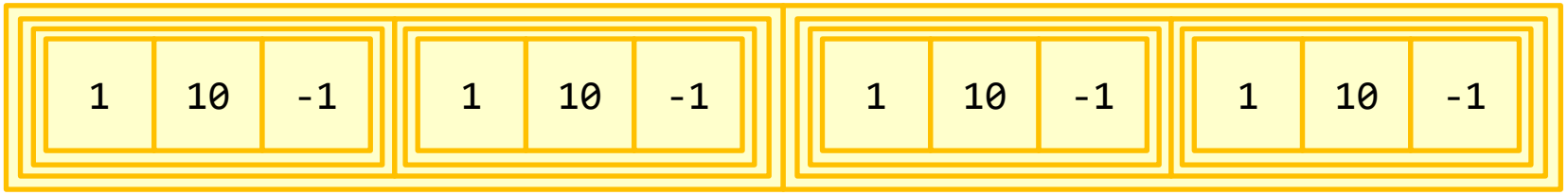
```
image.shape (1, 3, 3, 1)
weight.shape (2, 2, 1, 3)
conv2d.shape (1, 3, 3, 3)
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]
[[120. 160.  90.]
 [240. 280. 150.]
 [150. 170.  90.]]
[[-12. -16.  -9.]
 [-24. -28. -15.]
 [-15. -17.  -9.]]
```



```
weight = np.array([[[[1.,10.,-1.]],[[1.,10.,-1.]]],[[[1.,10.,-1.]],[[1.,10.,-1.]]]])
```

(2,2,1,3)

channel FN



(1,3,3,1)

1	1
1	1

10	10
10	10

-1	-1
-1	-1

```
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]
[[120. 160.  90.]
 [240. 280. 150.]
 [150. 170.  90.]]
[[-12. -16.  -9.]
 [-24. -28. -15.]
 [-15. -17.  -9.]]
```

np.swapaxes(conv2d, 0, 3)

(1,3,3,3)

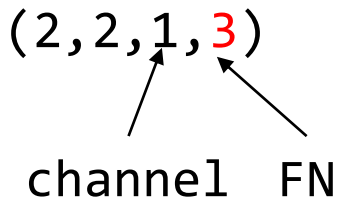
(3,3,3,1)

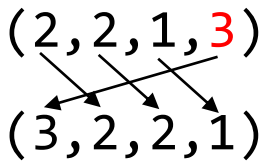
12	16	9
24	28	15
15	17	9

120	160	90
240	280	150
150	170	90

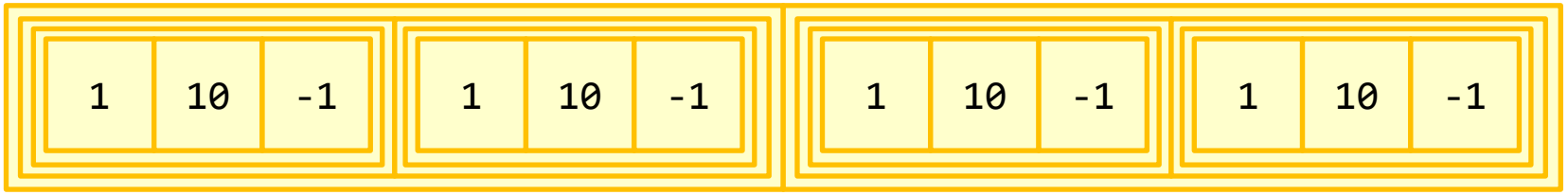
-12	-16	-9
-24	-28	-15
-15	-17	-9

```
np.transpose(weight, (3,0,1,2))
```





transpose(3,0,1,2)



1	1
1	1

10	10
10	10

-1	-1
-1	-1

```
[[1. 1.]
 [1. 1.]]
[[10. 10.]
 [10. 10.]]
[[-1. -1.]
 [-1. -1.]]
```

1	2	3
4	5	6
7	8	9

1	2	3
4	5	6
7	8	9

1	2	3
4	5	6
7	8	9

1	1
1	1

2	2
2	2

3	3
3	3

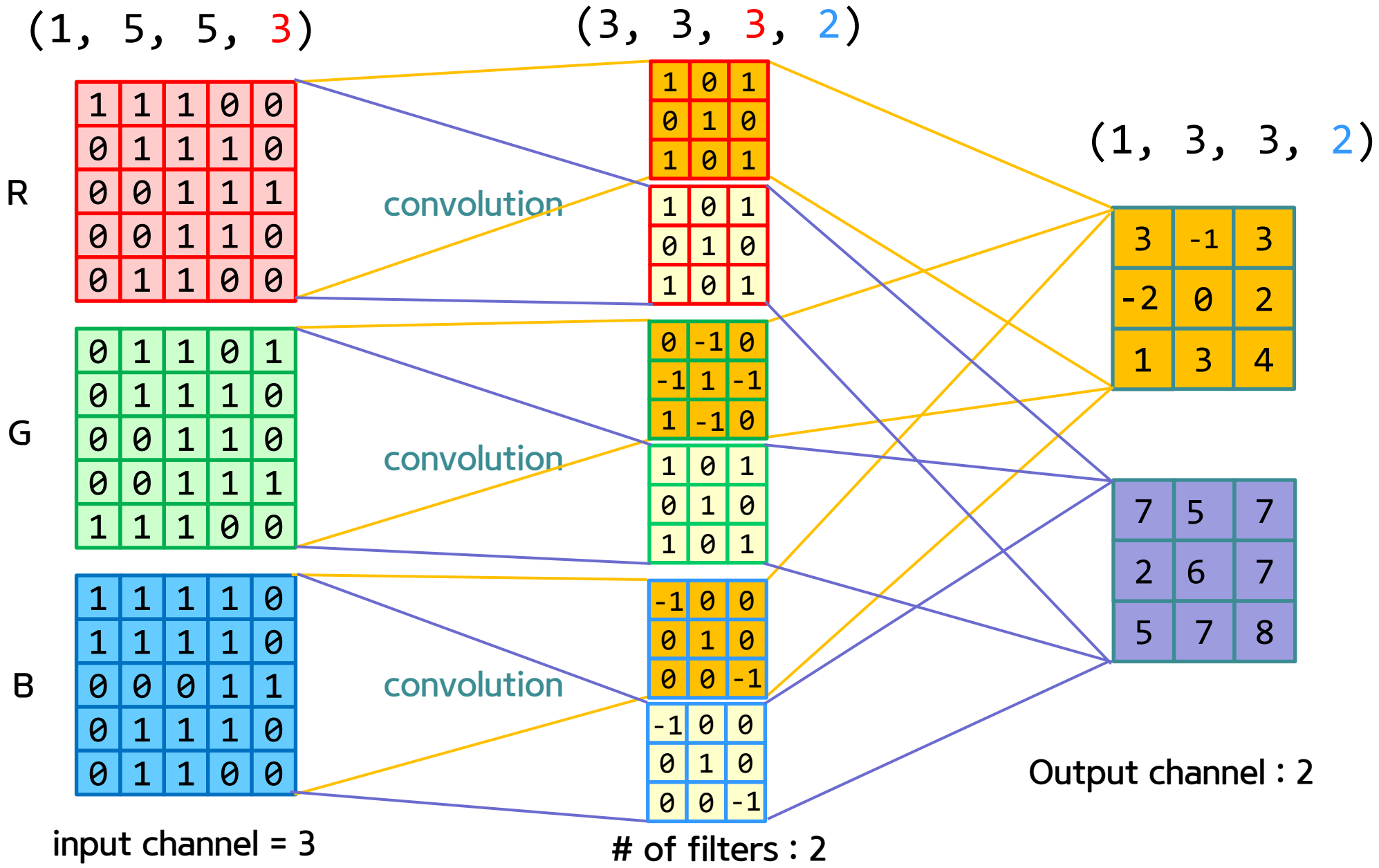
12	16	9
24	28	15
15	17	9

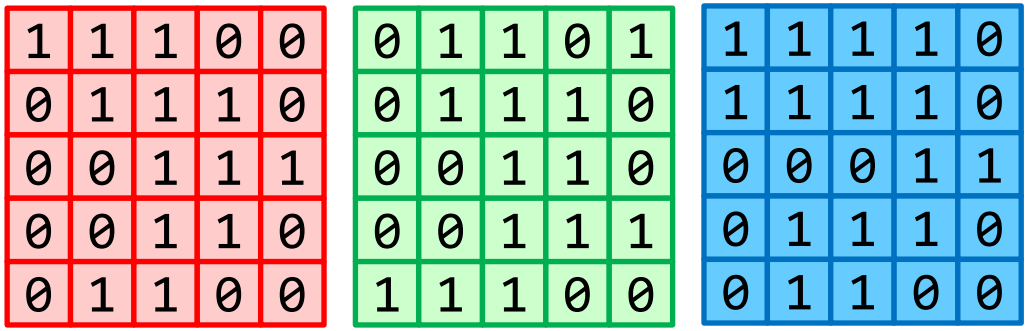
24	32	18
48	56	30
30	34	18

36	48	27
72	84	45
45	51	27

(1,3,3,1)

72	96	54
144	168	90
90	102	54


(1, 5, 5, 3)
(3, 3, 3, 2)
(1, 3, 3, 2)
R
G
B
convolution
convolution
convolution
input channel = 3
# of filters : 2
Output channel : 2



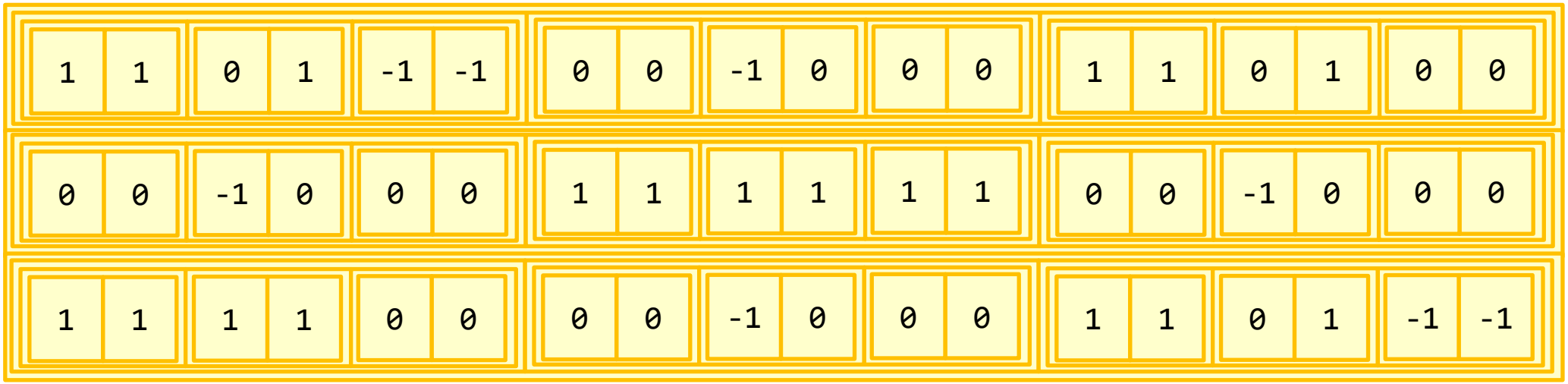
(1,5,5,3)

```
[[[[1,0,1], [1,1,1], [1,1,1], [0,0,1], [0,1,0]],
  [[0,0,1], [1,1,1], [1,1,1], [1,1,1], [0,0,0]],
  [[0,0,0], [0,0,0], [1,1,0], [1,1,1], [1,0,1]],
  [[0,0,0], [0,0,1], [1,1,1], [1,1,1], [0,1,0]],
  [[0,1,0], [1,1,1], [1,1,1], [0,0,0], [0,0,0]]]]
```

(1,5,5,3)

(3,3,3,2)

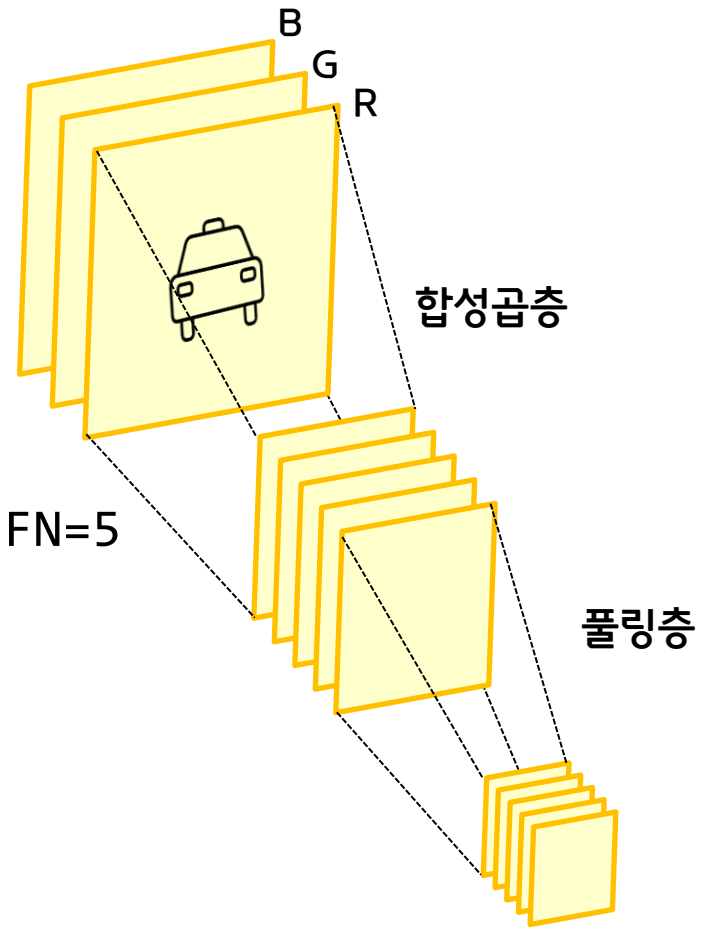
```
[[[[1,1], [0,1], [-1,-1]], [[0,0], [-1,0], [0,0]], [[1,1], [0,1], [0,0]]],
 [[[0,0], [-1,0], [0,0]], [[1,1], [1,1], [1,1]], [[0,0], [-1,0], [0,0]]],
 [[[1,1], [1,1], [0,0]], [[0,0], [-1,0], [0,0]], [[1,1], [0,1], [-1,-1]]]]
```



풀링 연산

풀링 연산

합성곱층과 풀링층을 거치면서 변환되는 과정

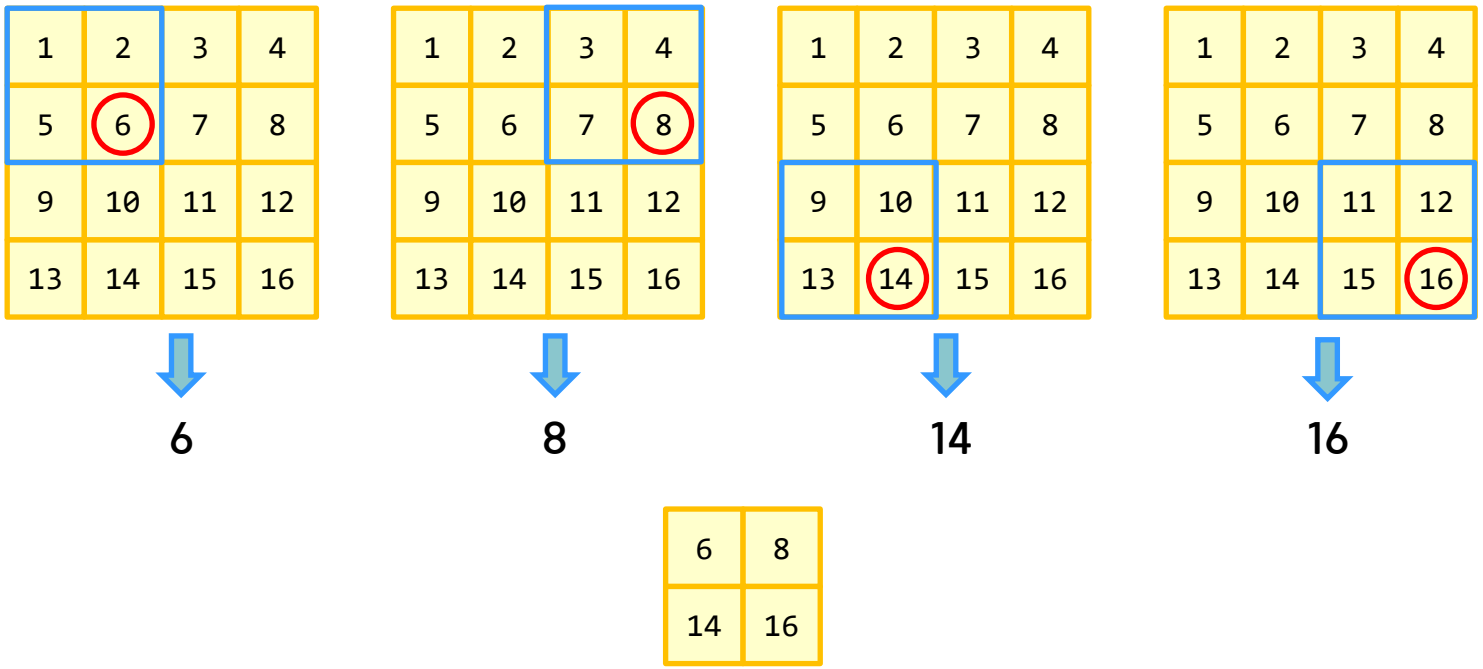


풀링 연산

풀링 연산

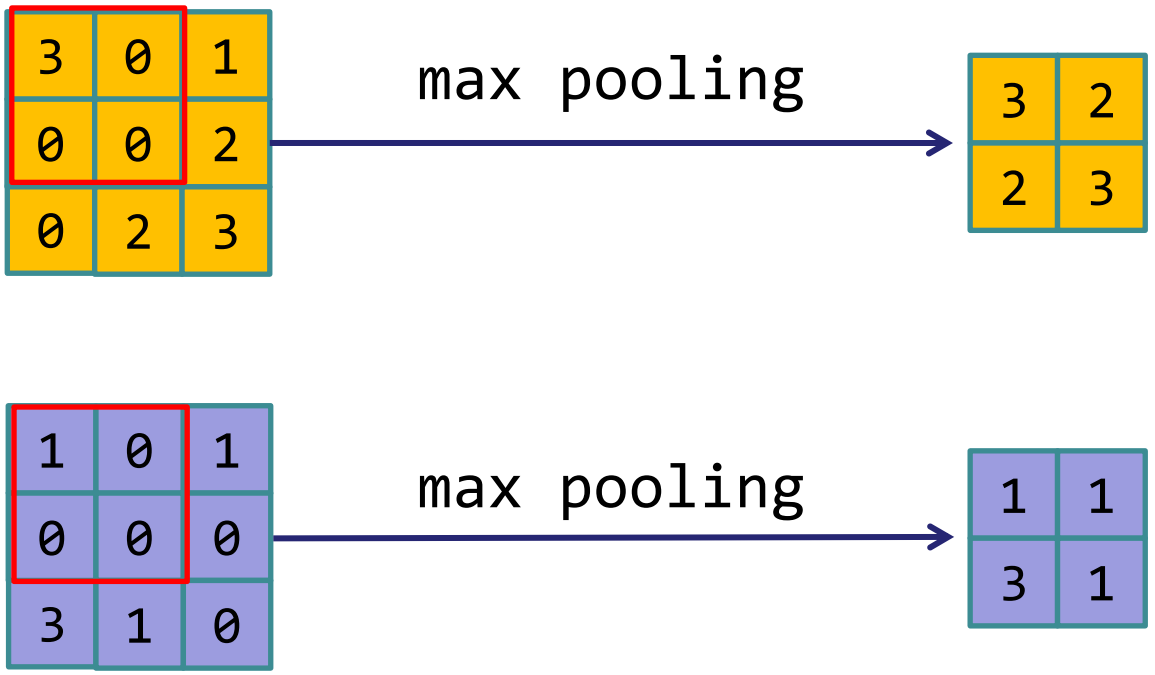
풀링이란? 특성 맵을 스캔하며 최대값을 고르거나 평균값을 계산하는 것을 말함

최대 풀링



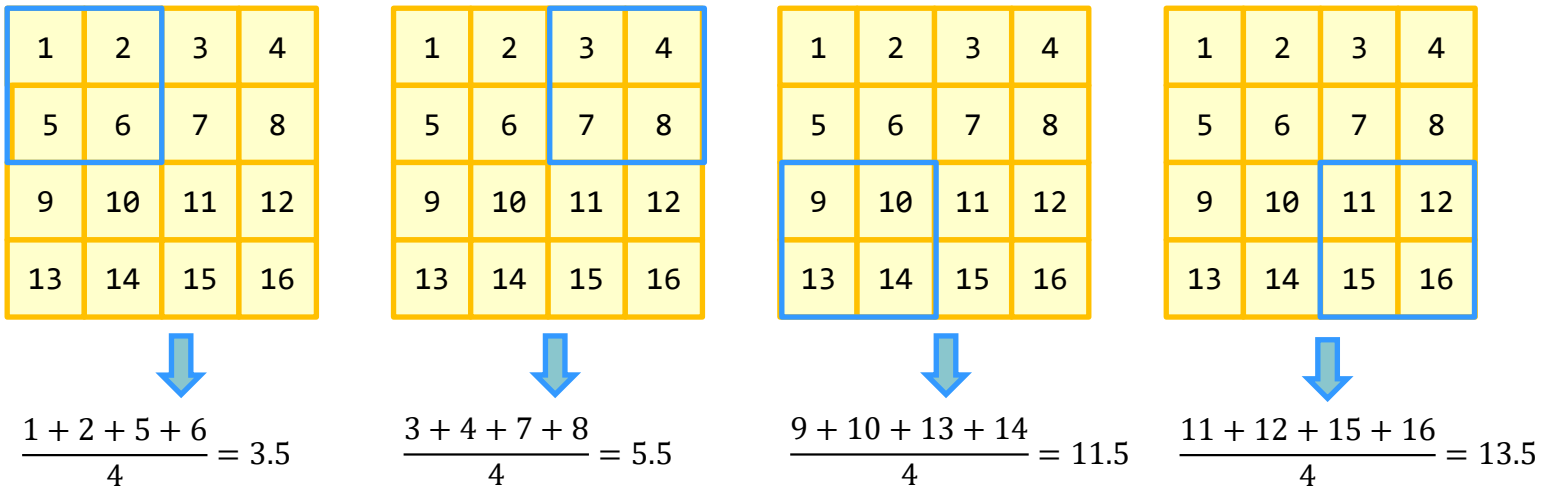
풀링 연산

Pooling(max pooling, 2x2 filter, stride 1)



풀링 연산

Pooling(average pooling, 2x2 filter, stride 2)



풀링 연산

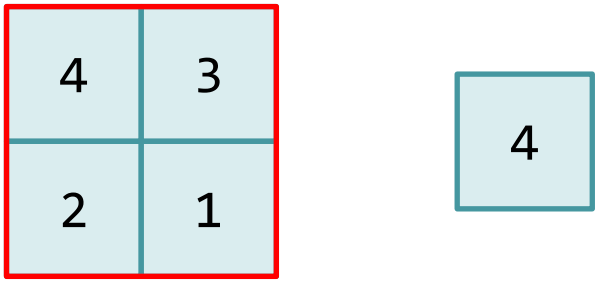
tf.keras.layers.MAXPool2D

- **`pool_size`**: integer or tuple of 2 integers, window size over which to take the maximum. `(2, 2)` will take the max value over a 2x2 pooling window. If only one integer is specified, the same window length will be used for both dimensions.
- **`strides`**: Integer, tuple of 2 integers, or None. Strides values. Specifies how far the pooling window moves for each pooling step. If None, it will default to `pool_size`.
- **`padding`**: One of `"valid"` or `"same"` (case-insensitive). "valid" adds no zero padding. "same" adds padding such that if the stride is 1, the output shape is the same as input shape.
- **`data_format`**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

풀링 연산

```
image = tf.constant([[[[4],[3]],[[2],[1]]]], dtype=np.float32)
pool = tf.keras.layers.MaxPool2D(pool_size=(2,2), strides=1, padding='valid')(image)
print(pool.shape)
print(pool.numpy())
```

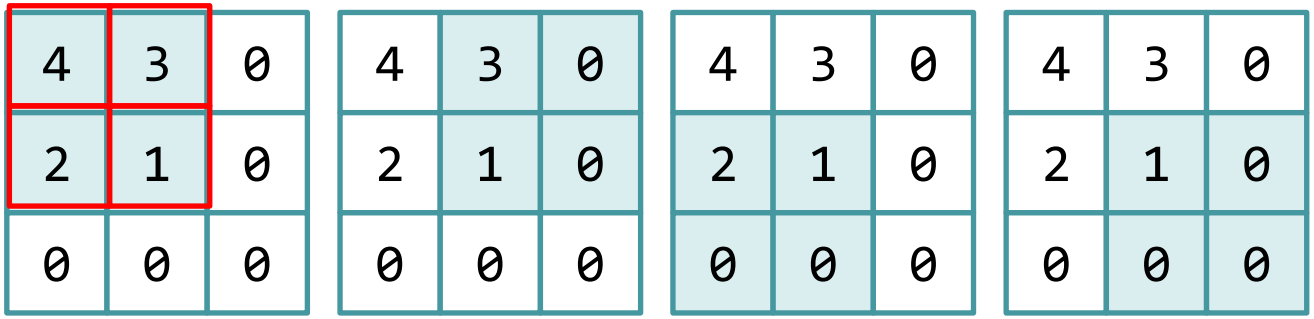
```
image.shape=(1, 2, 2, 1)
pool.shape=(1, 1, 1, 1)
[[[[4.]]]]
```



풀링 연산

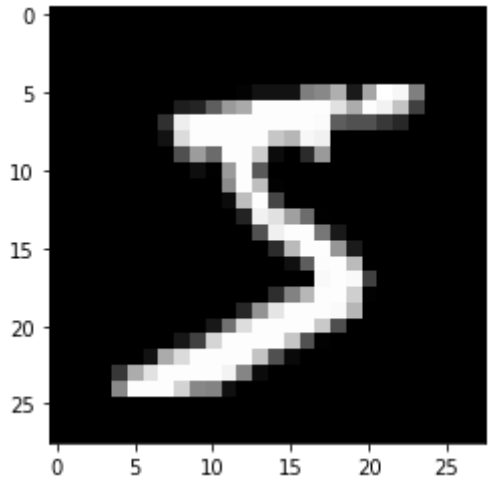
```
image = tf.constant([[[[4],[3]],[[2],[1]]]], dtype=np.float32)
pool = keras.layers.MaxPool2D(pool_size=(2,2), strides=1, padding='same')(image)
print(pool.shape)
print(pool.numpy())
```

```
(1, 2, 2, 1)
[[[[4.]
   [3.]]

  [[2.]
   [1.]]]]
```



Loading MNIST Data

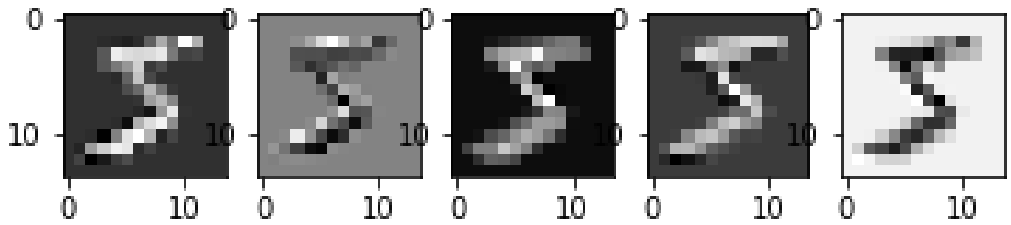
```
mnist = keras.datasets.mnist
class_names = ['0','1','2','3','4','5','6','7','8','9']

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.astype(np.float32) / 255.
test_images = test_images.astype(np.float32) / 255.

img = train_images[0]
plt.imshow( img, cmap='gray')
plt.show()
```



Loading MNIST Data

```
img = img.reshape(-1,28,28,1)
img = tf.convert_to_tensor(img)

print("weight.shpe", weight.shape)
weight_init = keras.initializers.RandomNormal(stddev=0.01)
conv2d = keras.layers.Conv2D(filters=5, kernel_size=3, padding='same',
                             strides=(2,2), kernel_initializer=weight_init)(img)
print("conv2d.shape", conv2d.shape)
feature_maps = np.swapaxes(conv2d, 0, 3)
for i, feature_map in enumerate(feature_maps):
    plt.subplot(1,5,i+1), plt.imshow(feature_map.reshape(14,14), cmap='gray')
plt.show()
```

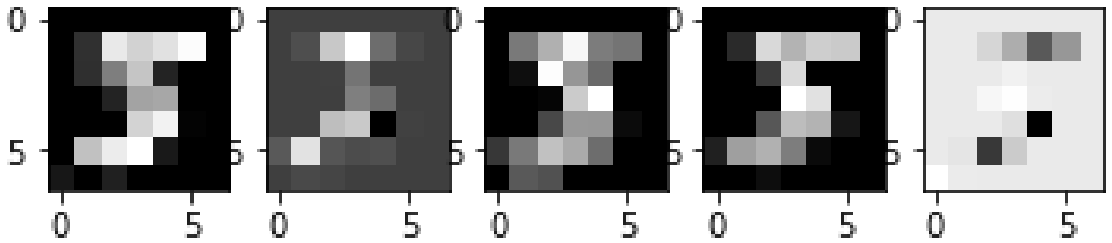
```
weight.shape (2, 2, 1, 5)
conv2d.shape (1, 14, 14, 5)
```



Loading MNIST Data

```
pool = keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding='same')(conv2d)
print(pool.shape)
feature_maps = np.swapaxes(pool, 0, 3)
for i, feature_map in enumerate(feature_maps):
    plt.subplot(1,5,i+1), plt.imshow(feature_map.reshape(7,7), cmap='gray')
plt.show()
```

```
(1, 7, 7, 5)
```



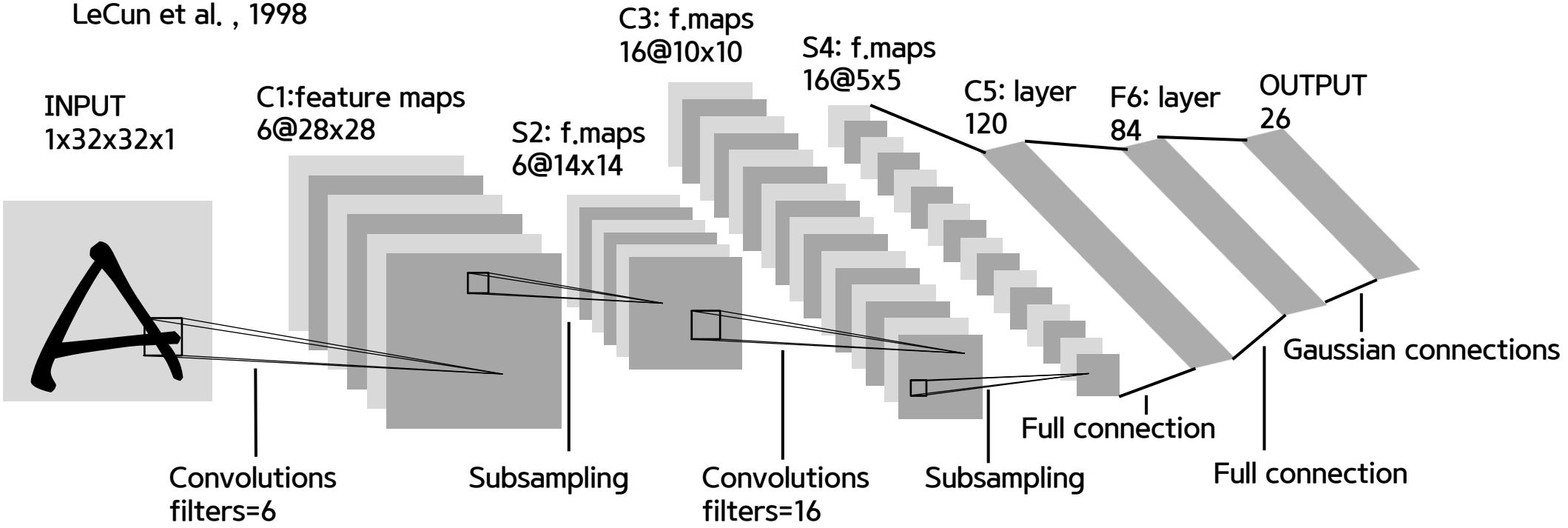
전체 합성곱 구조

LeNet-5

https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

(1,5,5,16)=>(1,400)(400,120)=>(1,120)(120,80)=>(1,80)(80,26)=>(1,26)

LeCun et al. , 1998



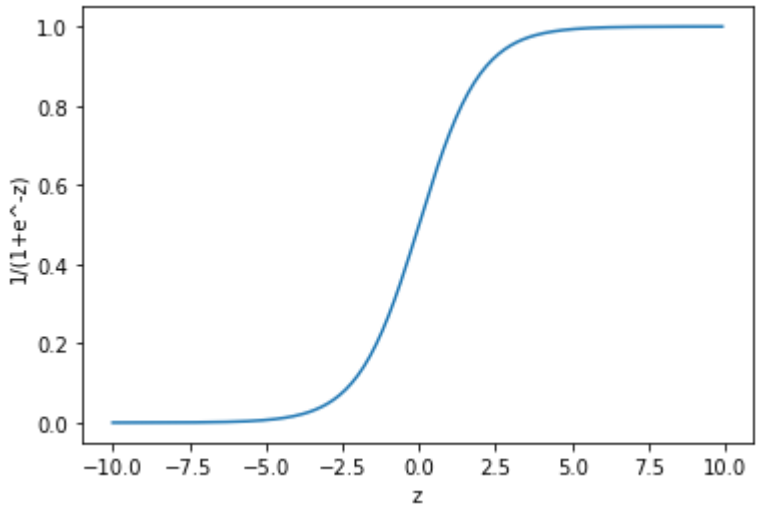
Conv filters 5x5, stride 1, padding=valid
Subsampling 2x2, stride 2, padding=same

활성화 함수

활성함수는 네트워크에 비선형성(nonlinearity)을 추가하기 위해 사용됨
- 활성함수 없이 layer를 쌓은 네트워크는 1-layer 네트워크와 동일하기 때문에 활성 함수는 비성형함수로 불리기도 한다.
- 멀티레이어 퍼셉트론을 만들 때 활성함수를 사용하지 않으면 쓰나마나이다.

1. 시그모이드 함수 (Sigmoid Function)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



- 결과값이 [0,1] 사이로 제한됨
- 뇌의 뉴런과 유사하여 많이 쓰였음

활성화 함수

- 문제점

1) 그레디언트가 죽는 현상이 발생한다 (Gradient vanishing 문제)

gradient 0이 곱해 지니까 그 다음 layer로 전파되지 않는다. 즉, 학습이 되지 않는다.

2) 활성함수의 결과 값의 중심이 0이 아닌 0.5이다.

3) 계산이 복잡하다 (지수함수 계산)

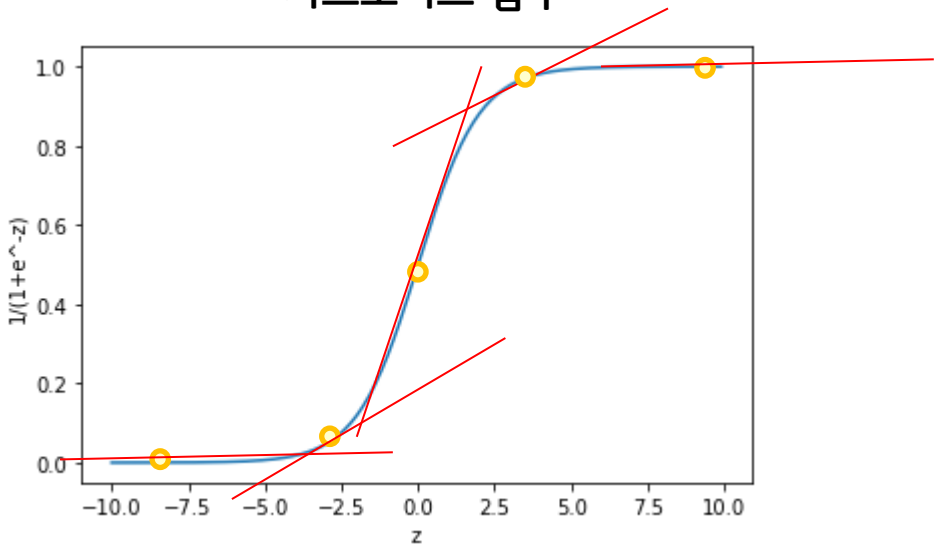
!! Gradient Vanishing

- 시그모이드와 같이 결과값이 포화(saturated)되는 함수는 gradient vanishing 현상을 야기
- 이전 레이어로 전파되는 그라디언트가 0에 가까워 지는 현상
- 레이어를 깊게 쌓으면 파라미터의 업데이트가 제대로 이루어지지 않음
- 양 극단의 미분값이 0에 가깝기 때문에 발생하는 문제

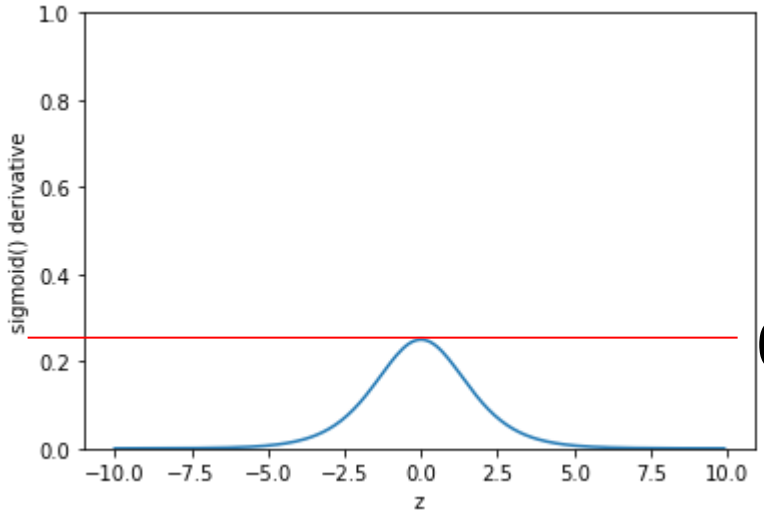
활성화 함수

a(1-a)

시그모이드 함수



시그모이드 미분값



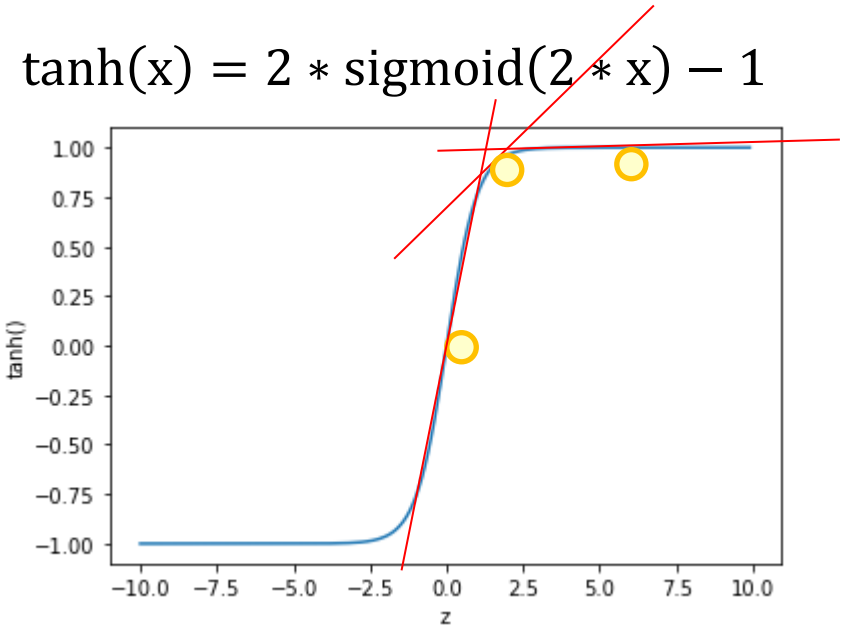
0.25

양쪽 꼬리가 0에 수렴하며 최대값이 0.25를 넘지 않는다.

활성화 함수

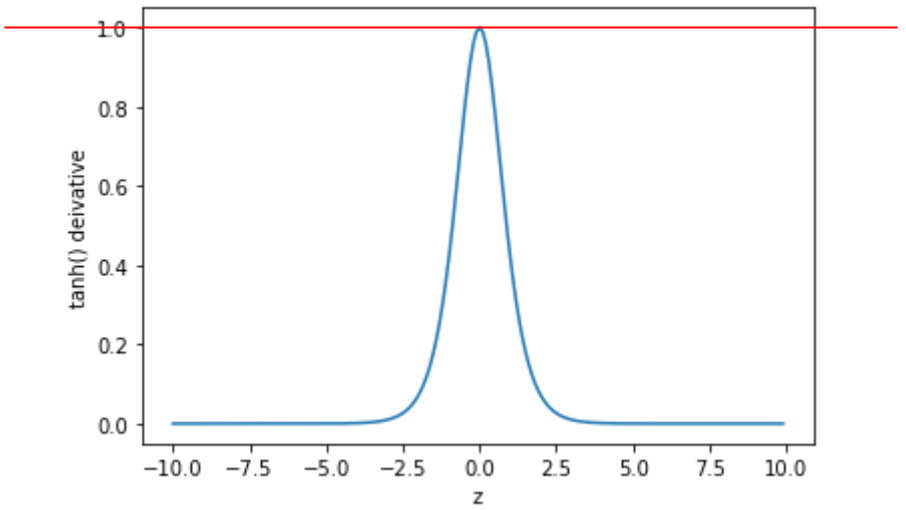
2. 하이퍼 볼릭 탄젠트(tanh)

$$\tanh(x) = 2 * \text{sigmoid}(2 * x) - 1$$



$$\tanh(x)' = (1 - \tanh(x))(1 + \tanh(x))$$

$$= (1 - H^2)$$



- 결과값이 [-1, 1] 사이로 제한됨. 결과값 중심이 0이다.
- 나머지 특성은 시그모이드와 비슷함. 시그모이드 함수를 이용하여 유도 가능
- 그러나, 여전히 gradient vanishing 문제가 발생

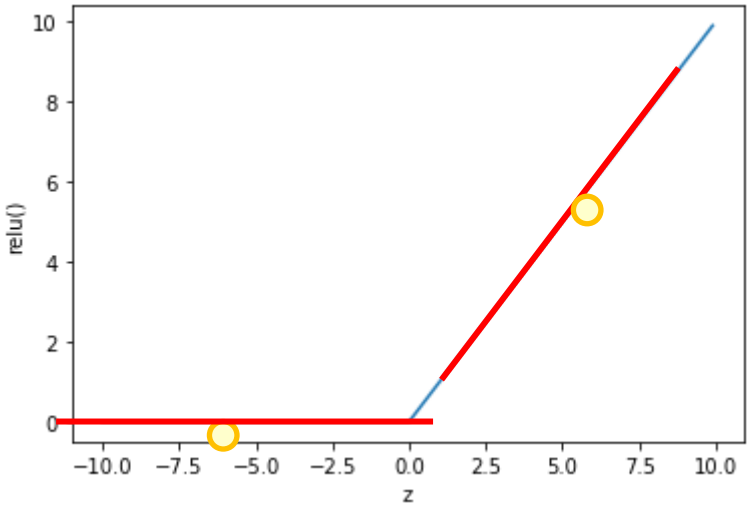
활성화 함수

3. 렐루(ReLU, Rectified Linear Unit)

$$f(x) = \max(0, x)$$

$$f(x) = x$$

$$f(x) = 0$$



최근 뉴럴 네트워크에서 가장 많이 쓰이는 활성 함수
선형아니야? NO! 0에서 확 꺾이기 때문에 비선형이라고 본다.

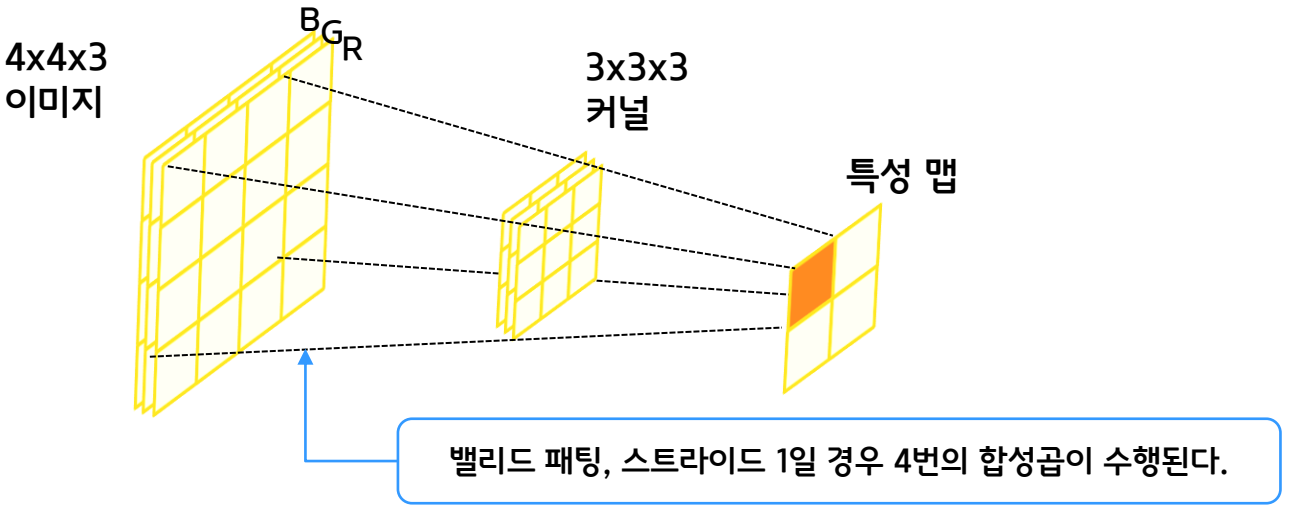
$$\frac{dy}{dx}$$

- 장점
(1) 양 극단값이 포화되지 않는다. (양수 지역은 선형적)
(2) 계산이 매우 효율적이다 (최대값 연산 1개)
(3) 수렴속도가 시그모이드류 함수대비 6배 정도 빠르다.

- 단점
(1) 중심값이 0이 아님 (마이너한 문제)
(2) 입력값이 음수인 경우 항상 0을 출력함 (마찬가지로 파라미터 업데이트가 안됨)

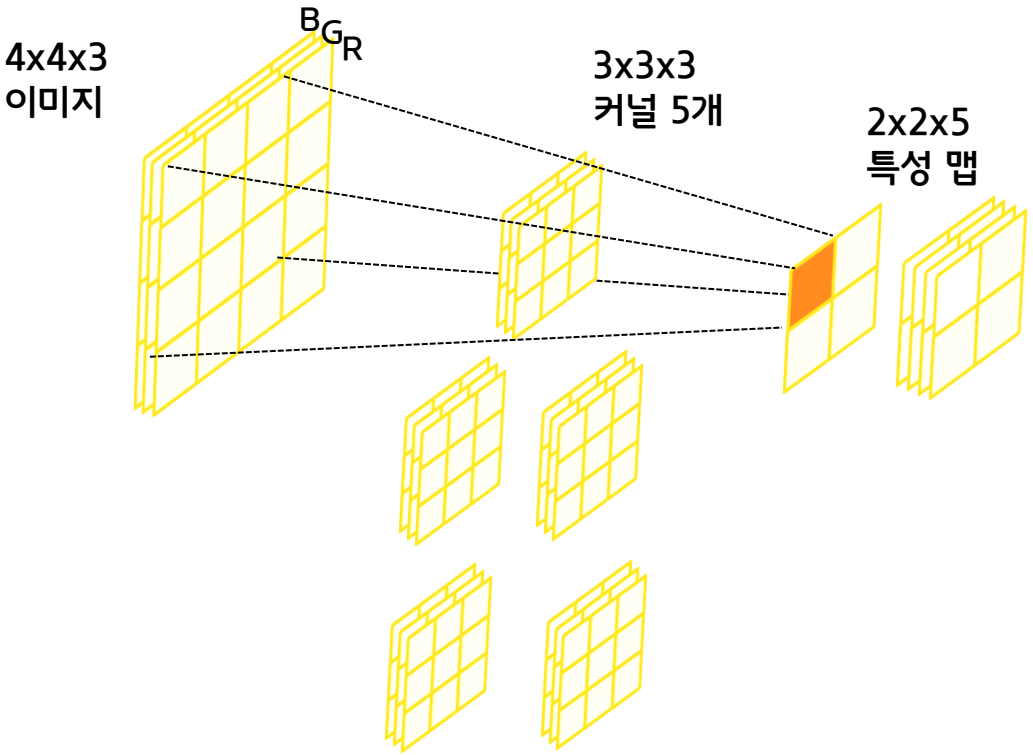
활성화 함수

합성곱 층에서 일어나는 일



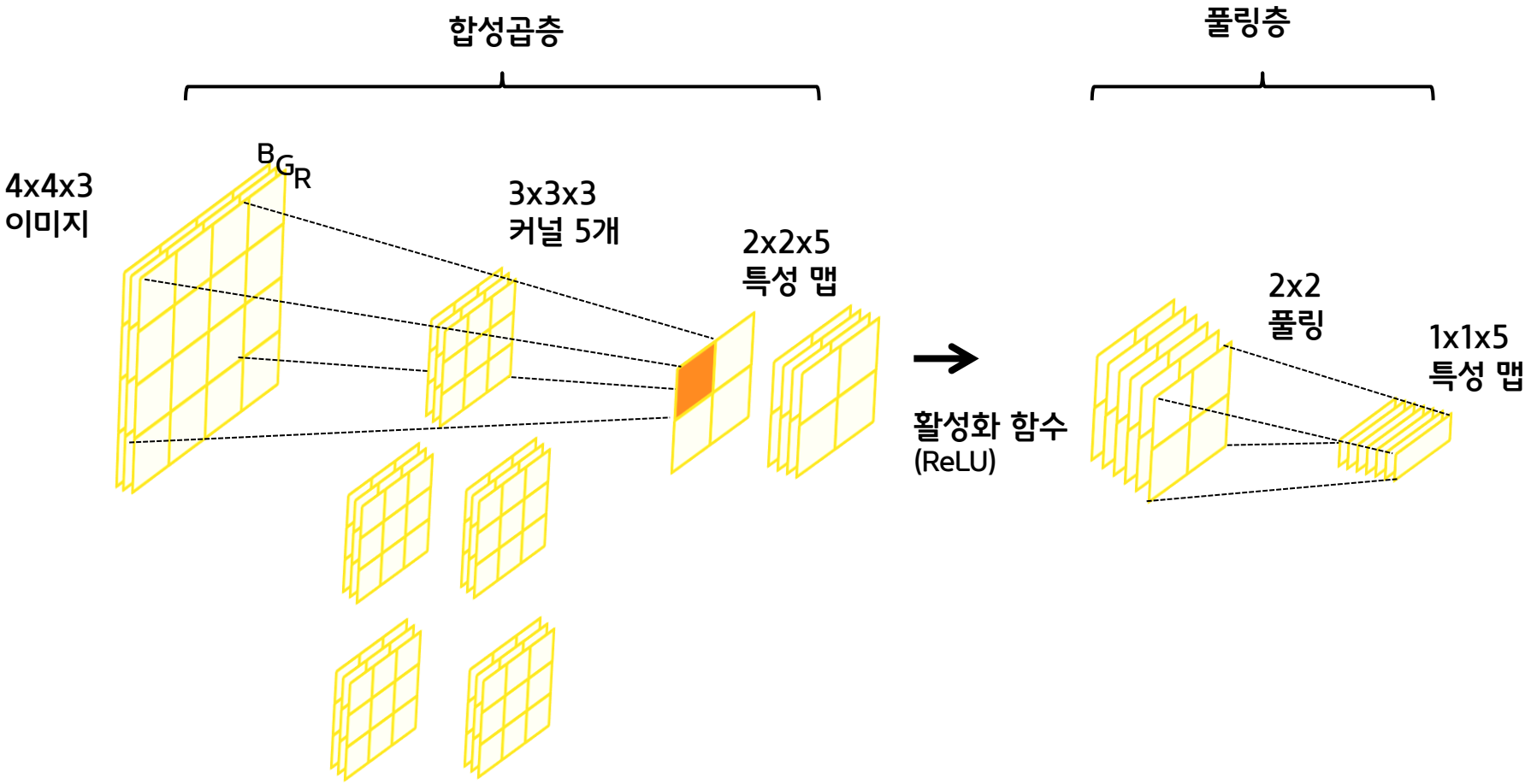
활성화 함수

합성곱 층에서 일어나는 일



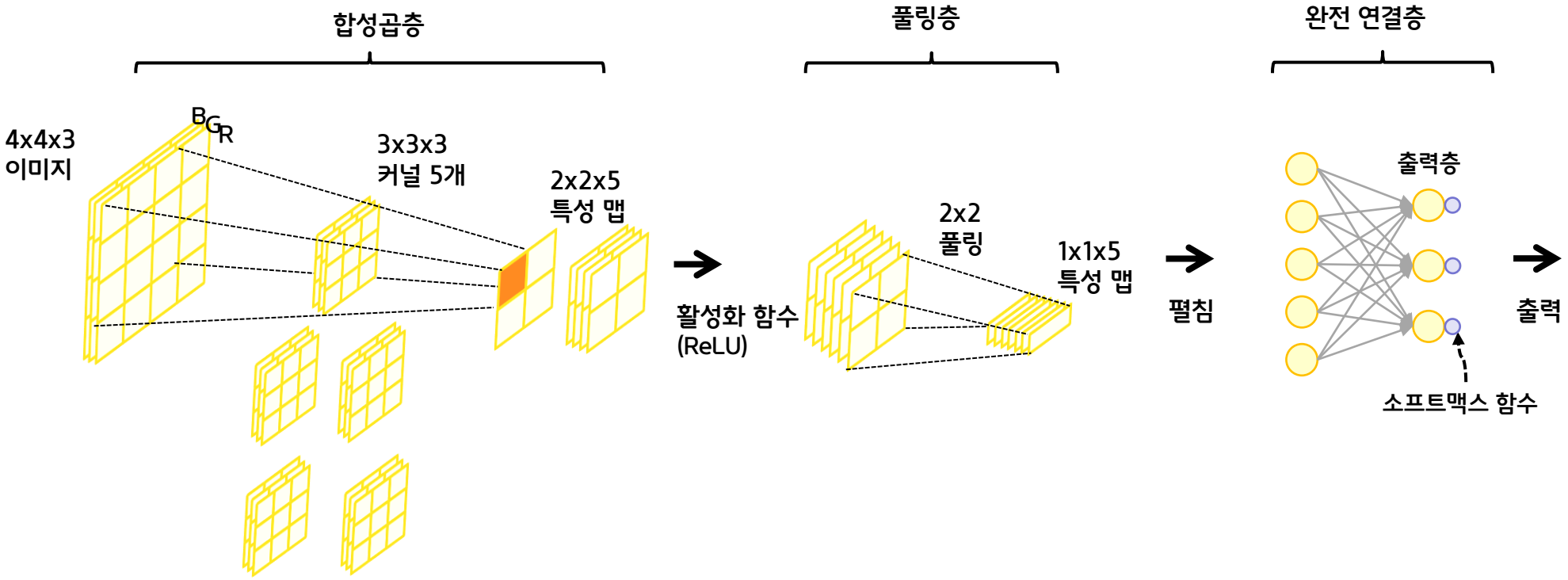
활성화 함수

풀링 층에서 일어나는 일

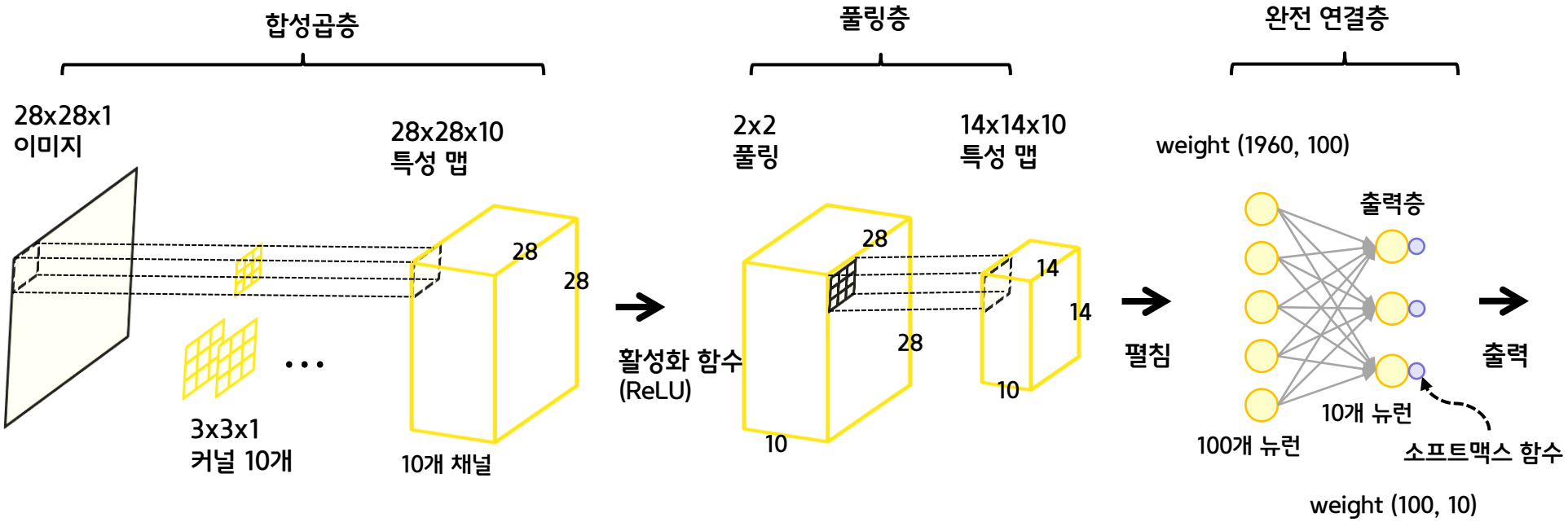


활성화 함수

특성 맵을 펼쳐 완전 연결 신경망에 주입한다.



합성곱 신경망의 전체 구조



- 28x28 크기의 흑백 이미지와 3x3 크기의 커널 10개로 합성곱 수행
- 2x2 크기의 최대 풀링을 수행하여 14x14x10로 특성 맵의 크기를 줄인다.
- 특성 맵을 일렬로 펼쳐서 100개의 뉴런을 가진 완전 연결층과 연결 시킨다.
- 10개의 클래스를 구분하기 위한 소프트맥스 함수에 연결한다.

텐서플로를 이용한 구현

합성곱 신경망의 정방향 구현

합성곱 적용

```
def forpass(self, x):
        # 3x3 합성곱 연산을 수행합니다.
        c_out = tf.nn.conv2d(x, self.conv_w, strides=1, padding='SAME') + self.conv_b
```

- self.conv_w
  self.conv_w는 합성곱에 사용할 가중치이다. 3x3x1 크기의 커널을 10개 사용하므로 가중치의 전체 크기는 3x3x1x10 이다.

- strides, padding
  특성 맵의 가로와 세로 크기를 일정하게 만들기 위하여 strides는 1, padding은 'SAME'으로 지정한다.

렐루 함수 적용

```
def forpass(self, x):
        ...
        r_out = tf.nn.relu(c_out)
```

합성곱 신경망의 정방향 구현

풀링 적용하고 완전 연결층 수정

```
def forpass(self, x):
      ...
      p_out = tf.nn.max_pool2d(r_out, ksize=2, strides=2, padding='VALID')
      # 첫 번째 배치 차원을 제외하고 출력을 일렬로 펼칩니다. (64, 14, 14, 10)
      f_out = tf.reshape(p_out, [x.shape[0], -1])  # (64, 1960)(1960, 100)+(100,)
      z1 = tf.matmul(f_out, self.w1) + self.b1     # 첫 번째 층의 선형 식을 계산합니다
      a1 = tf.nn.relu(z1)                          # 활성화 함수를 적용합니다
      z2 = tf.matmul(a1, self.w2) + self.b2        # 두 번째 층의 선형 식을 계산합니다.
      return z2  # (64, 10)
```

- max_pool2d() 함수를 사용하여 2x2 크기의 풀링을 적용 한다.
  이 단계에서 만들어진 특성 맵의 크기는 14x14x10 이다.

- tf.reshape() 함수를 사용해 일렬로 펼친다.
  이때 배치 차원을 제외한 나머지 차원만 펼쳐야 한다.

- np.dot() 함수를 텐서플로의 tf.matmul() 함수로 바꿔서 구현한다.
  이는 conv2d()와 max_pool2d() 등이 Tensor 객체를 반환하기 때문임

- 완전 연결층의 활성화 함수도 시그모이드 대신 렐루 함수를 사용한다.

텐서플로를 이용한 구현

합성곱 신경망의 역방향 계산 구현

자동 미분의 사용 방법

```
x = tf.Variable(np.array([1.0, 2.0, 3.0]))
with tf.GradientTape() as tape:
    y = x ** 3 + 2 * x + 5
# 그래디언트를 계산합니다.
print(tape.gradient(y, x))
```

tf.Tensor([5. 14. 29.], shape=(3,), dtype=float64)

- 텐서플로와 같은 딥러닝 패키지들은 사용자가 작성한 연산을 계산 그래프로 만들어 자동 미분 기능을 구현한다.
- 자동 미분기능을 사용하면 임의의 파이썬 코드나 함수에 대한 미분값을 계산할 수 있다.
- 텐서플로의 자동 미분 기능을 사용하려면 with블럭으로 tf.GradientTape() 객체가 감시할 코드를 감싸야 한다.
- tape객체는 with블럭 안에서 일어나는 모든 연산을 기록하고 텐서플로 변수인 tf.Variable객체를 자동으로 추적한다.
- 그레이디언트를 계산하려면 미분 대상 객체와 변수를 tape객체의 gradient() 함수에 전달해야 한다.

$x^3 + 2x + 5$ 를 미분하면 $3x^2 + 2$ 가 되므로

1.0, 2.0, 3.0을 미분 방정식에 대입하면 5.0, 14.0, 29.0 을 얻는다.

합성곱 신경망의 역방향 계산 구현

1. 역방향 계산 구현

```
def training(self, x, y):
        m = len(x)                          # 샘플 개수를 저장합니다.
        with tf.GradientTape() as tape:
            z = self.forpass(x)             # 정방향 계산을 수행합니다.
            # 손실을 계산합니다.
            loss = tf.nn.softmax_cross_entropy_with_logits(y, z)
            loss = tf.reduce_mean(loss)
```

- 자동 미분 기능을 사용하면 ConvolutionNetwork의 backprop() 함수를 구현할 필요가 없다.

- training() 함수에서 forpass() 함수를 호출하여 정방향 계산을 수행한 다음

- tf.nn.softmax_cross_entropy_with_logits() 함수를 호출하여 정방향 계산의 결과(z)와 타깃(y)을 기반으로 손실값을 계산한다.

- 이렇게 하면 크로스 엔트로피 손실과 그레이디언트 계산을 올바르게 처리해 주므로 편하다.

- softmax_cross_entropy_with_logits() 함수는 배치의 각 샘플에 대한 손실을 반환하므로 reduce_mean() 함수로 평균을 계산한다.

합성곱 신경망의 역방향 계산 구현

2. 그레이디언트 계산

```
def training(self, x, y):
        ...
        weights_list = [self.conv_w, self.conv_b,
                        self.w1, self.b1, self.w2, self.b2]
        # 가중치에 대한 그래디언트를 계산합니다.
        grads = tape.gradient(loss, weights_list)
        # 가중치를 업데이트합니다.
        self.optimizer.apply_gradients(zip(grads, weights_list))
```

- tape.gradient() 를 이용하면 그레이디언트를 자동으로 계산할 수 있다.
- 합성곱층의 가중치와 절편인 con_w와 con_b를 포함하여 그레이디언트가 필요한 weights_list로 나열한다.
- 텐서플로의 옵티마이저를 사용하면 간단하게 알고리즘을 바꾸어 테스트할 수 있다.
- self.optimizer.apply_gradients() 함수에는 그레이디언트와 가중치를 튜플로 묶은 리스트를 전달해야 한다.

옵티마이저 객체를 만들어 가중치 초기화

1. fit() 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
        self.init_weights(x.shape, y.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
        self.optimizer = tf.optimizers.SGD(learning_rate=self.lr)
        # epochs만큼 반복합니다.
        for i in range(epochs):
            print('에포크', i, end=' ')
            # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
            batch_losses = []
            for x_batch, y_batch in self.gen_batch(x, y):
                print('.', end='')
                self.training(x_batch, y_batch)
                # 배치 손실을 기록합니다.
                batch_losses.append(self.get_loss(x_batch, y_batch))
            print()
            # 배치 손실 평균내어 훈련 손실 값으로 저장합니다.
            self.losses.append(np.mean(batch_losses))
            # 검증 세트에 대한 손실을 계산합니다.
            self.val_losses.append(self.get_loss(x_val, y_val))
```

- 텐서플로는 tf.optimizers 모듈 아래에 여러 종류의 경사 하강법을 구현해 놓았다.

- SGD옵티마이저(tf.optimizers.SGD)객체는 기본 경사 하강법이다.

옵티마이저 객체를 만들어 가중치 초기화

2. init_weights() 함수 수정

```
def init_weights(self, input_shape, n_classes):
        g = tf.initializers.glorot_uniform()
        self.conv_w = tf.Variable(g((3, 3, 1, self.n_kernels)))
        self.conv_b = tf.Variable(np.zeros(self.n_kernels), dtype=float)
        n_features = 14 * 14 * self.n_kernels
        self.w1 = tf.Variable(g((n_features, self.units)))          # (특성 개수, 은닉층의 크기)
        self.b1 = tf.Variable(np.zeros(self.units), dtype=float)    # 은닉층의 크기
        self.w2 = tf.Variable(g((self.units, n_classes)))           # (은닉층의 크기, 클래스 개수)
        self.b2 = tf.Variable(np.zeros(n_classes), dtype=float)     # 클래스 개수
```

- 가중치를 glorot_uniform() 함수로 초기화 한다.

- 가중치를 tf.Variable() 함수로 만들어야 한다.

- np.zeros는 64bit로 초기화 되므로 dtype=float으로 32bit 바꿔준다.

텐서플로를 이용한 구현

합성곱 신경망 훈련

1. 데이터 세트 불러오기

```
(x_train_all, y_train_all), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

2. 훈련 세트와 검증 세트로 나누기

```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train_all, y_train_all, stratify=y_train_all,
                                                  test_size=0.2, random_state=42)
```

3. 타깃을 원-핫 인코딩으로 변환하기

```
y_train_encoded = tf.keras.utils.to_categorical(y_train)
y_val_encoded = tf.keras.utils.to_categorical(y_val)
```

4. 입력 데이터 준비하기

```
x_train = x_train.reshape(-1, 28, 28, 1)
x_val = x_val.reshape(-1, 28, 28, 1)
```

```
x_train.shape
```

(48000, 28, 28, 1)

합성곱 신경망 훈련

5. 입력 데이터 표준화 전처리하기

```
x_train = x_train / 255
x_val = x_val / 255
```

6. 모델 훈련하기

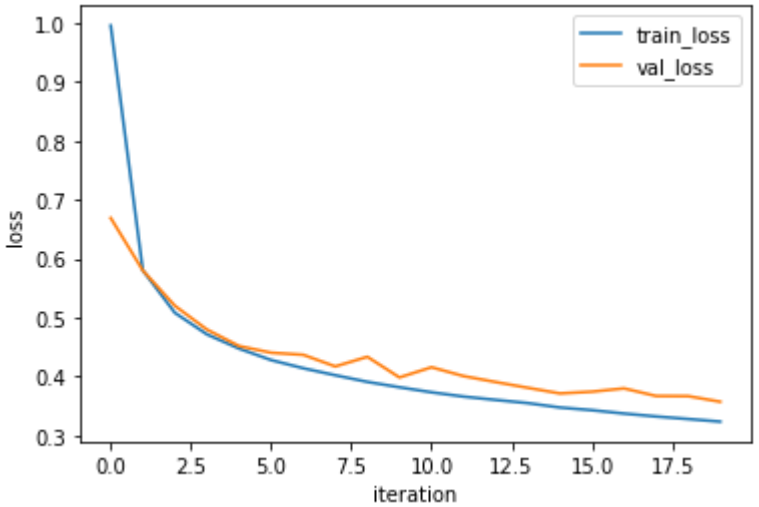
```
cn = ConvolutionNetwork(n_kernels=10, units=100, batch_size=128, learning_rate=0.01)
cn.fit(x_train, y_train_encoded,
       x_val=x_val, y_val=y_val_encoded, epochs=20)
```

```
에포크 0 ..........
에포크 1 ..........
.
.
.
에포크 19 ..........
```

합성곱 신경망 훈련

7. 훈련, 검증 손실 그래프 그리고 검증 세트의 정확도 확인

```
import matplotlib.pyplot as plt
plt.plot(cn.losses)
plt.plot(cn.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```




```
cn.score(x_val, y_val_encoded)
```

0.87725

케라스를 이용한 구현

케라스로 합성곱 신경망 만들기

1. 필요한 클래스들을 임포트하기

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

2. 합성곱층 쌓기

```
conv1 = tf.keras.Sequential()
conv1.add(Conv2D(10, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
```

- Conv2D클래스의 첫 번째 매개변수는 합성곱 커널의 개수이다.
- 두번째 매개변수는 합성곱 커널의 크기로 높이와 너비를 튜플로 전달한다.
- activation 매개변수에 렐루 활성화 함수를 지정한다.
- padding은 same을 지정한다. 이때 대소문자를 구분하지 않는다.
- Sequential 클래스에 층을 처음 추가할 때는 배치 차원을 제외한 입력의 크기를 지정한다.

3. 풀링층 쌓기

```
conv1.add(MaxPooling2D((2, 2)))
```

- MaxPooling2D 클래스의 첫 번째 매개변수는 풀링의 높이와 너비를 나타내는 튜플이며, 스트라이드는 strides에 지정할 수 있음
- 패딩은 padding에 지정하며 기본값은 'valid'이다.

4. 완전 연결층에 주입할 수 있도록 특성 맵 펼치기

```
conv1.add(Flatten())
```

- 풀링 다음에는 완전 연결층에 연결하기 위해 배치 차원을 제외하고 일렬로 펼쳐야 한다. 이 일은 Flatten 클래스가 수행함

케라스를 이용한 구현

케라스로 합성곱 신경망 만들기

5. 완전 연결층 쌓기

```
conv1.add(Dense(100, activation='relu'))
conv1.add(Dense(10, activation='softmax'))
```

- 첫 번째 완전 연결층에는 100개의 뉴런을 사용하고 렐루 활성화 함수를 적용한다.
- 마지막 출력층에는 10개의 클래스에 대응하는 10개의 뉴런을 사용하고 소프트맥스 활성화 함수를 적용한다.

6. 모델 구조 살펴보기

```
conv1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 10)        100       ( 3x3x1x10+10 )
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 10)        0
_________________________________________________________________
flatten (Flatten)            (None, 1960)              0         ( 14x14x10 )
_________________________________________________________________
dense (Dense)                (None, 100)               196100    ( 1960x100+100 )
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1010      ( 100x10+10 )
=================================================================
Total params: 197,210
Trainable params: 197,210
Non-trainable params: 0
```

케라스를 이용한 구현

합성곱 신경망 모델 훈련하기

1. 모델 컴파일

```
conv1.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- 다중 분류를 위한 크로스 엔트로피 손실 함수를 사용한다.
- 정확도를 관찰하기 위해 metrics 매개변수에 'accuracy'를 리스트로 전달한다.
- 아담(adam) 옵티마이저를 사용한다.
  아담은 Adaptive Moment Estimation을 줄여 만든 이름이다.
  아담은 손실 함수의 값이 최적값에 가까워질수록 학습률을 낮춰 손실 함수의 값이 안정적으로 수렴될 수 있게 해준다.

2. 모델 훈련

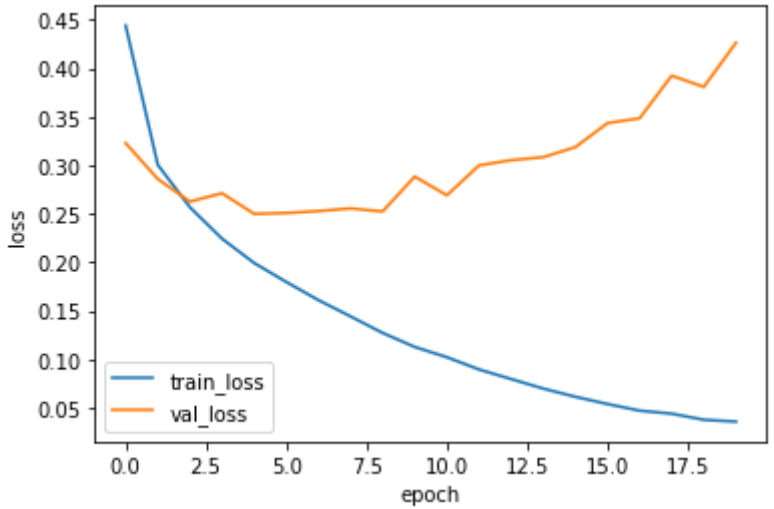
```
history = conv1.fit(x_train, y_train_encoded, epochs=20,
                    validation_data=(x_val, y_val_encoded))
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 8s 171us/sample - loss: 0.4442 - accuracy: 0.8434 - val_loss: 0.3229 - val_accuracy: 0.8862
Epoch 2/20
48000/48000 [==============================] - 8s 166us/sample - loss: 0.3005 - accuracy: 0.8923 - val_loss: 0.2860 - val_accuracy: 0.8968
Epoch 3/20
48000/48000 [==============================] - 8s 160us/sample - loss: 0.2568 - accuracy: 0.9062 - val_loss: 0.2626 - val_accuracy: 0.9043
...
Epoch 20/20
48000/48000 [==============================] - 8s 172us/sample - loss: 0.0356 - accuracy: 0.9875 - val_loss: 0.4264 - val_accuracy: 0.9135
```

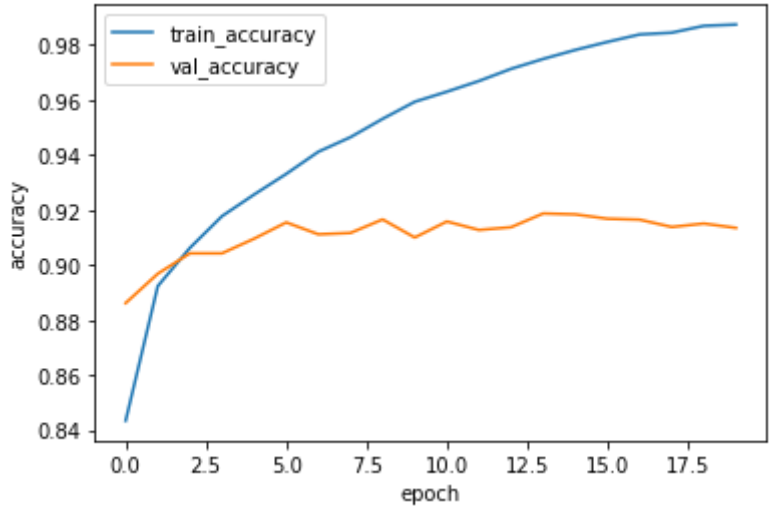
합성곱 신경망 모델 훈련하기

3. 손실 그래프와 정확도 그래프

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



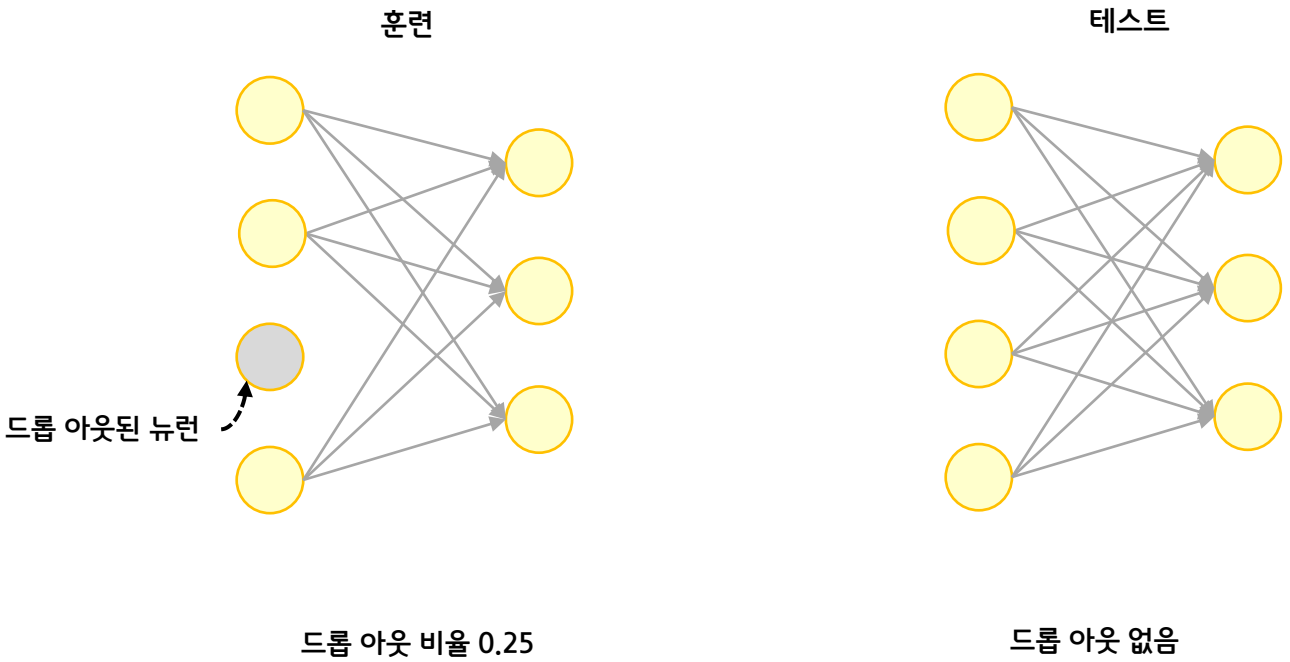

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



케라스를 이용한 구현

드롭아웃

드롭아웃에 대하여



드롭아웃 적용해 합성곱 신경망 구현

1. 케라스로 만든 합성곱 신경망에 드롭아웃 적용하기

```
from tensorflow.keras.layers import Dropout

conv2 = tf.keras.Sequential()
conv2.add(Conv2D(10, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
conv2.add(MaxPooling2D((2, 2)))
conv2.add(Flatten())
conv2.add(Dropout(0.5))
conv2.add(Dense(100, activation='relu'))
conv2.add(Dense(10, activation='softmax'))
```

2. 드롭아웃층 확인하기

```
conv2.summary()
```

3. 훈련하기

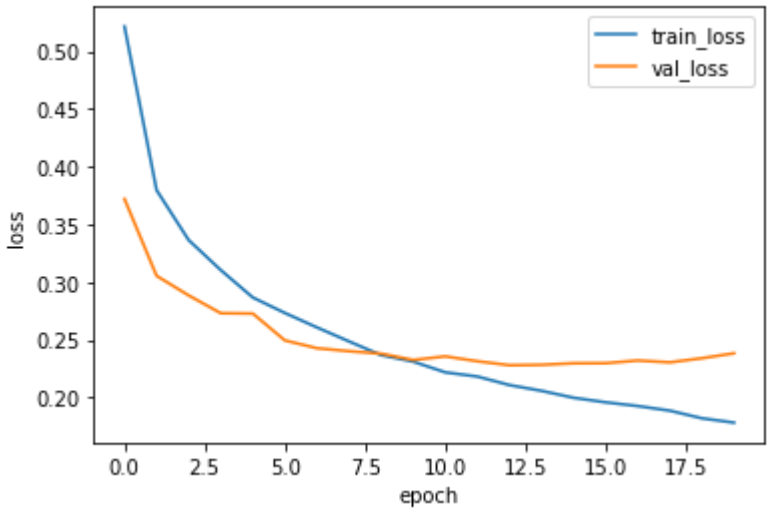
```
conv2.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
history = conv2.fit(x_train, y_train_encoded, epochs=20,
                    validation_data=(x_val, y_val_encoded))
```

케라스를 이용한 구현

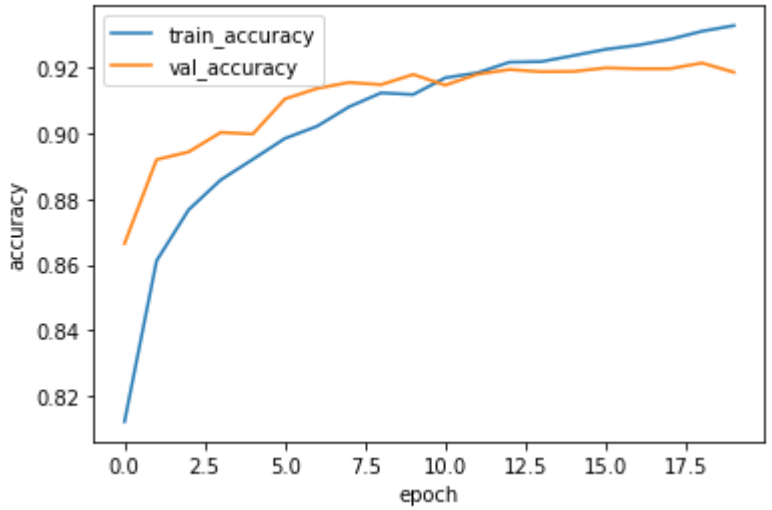
드롭아웃 적용해 합성곱 신경망 구현

4. 손실 그래프와 정확도 그래프 그리기

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```




```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```

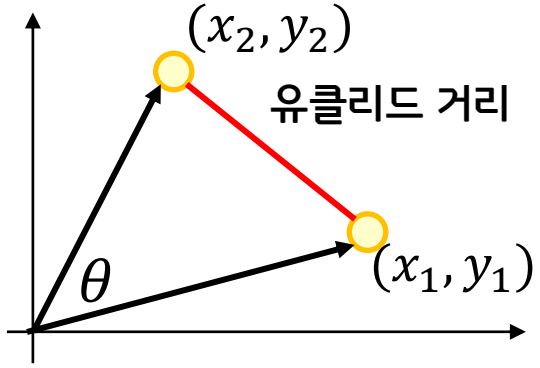


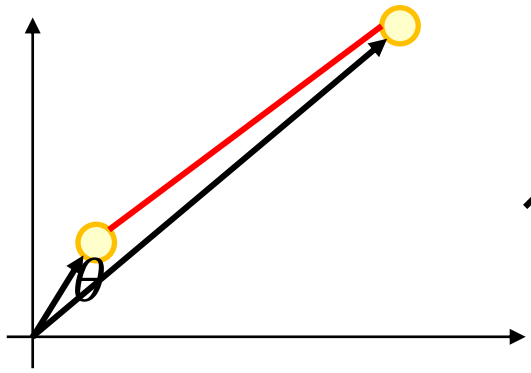
벡터간 유사도

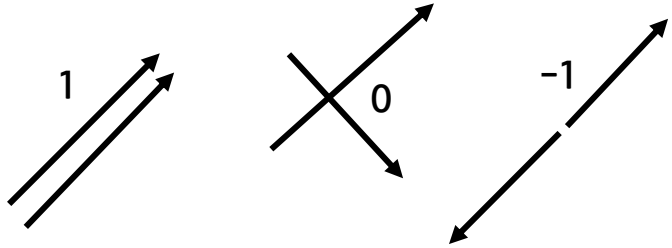
단어 벡터의 유사도를 나타낼 때는 코사인 유사도를 자주 이용한다.

$$similarity(x, y) = \frac{x \cdot y}{\|x\|\|y\|} = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{x_1^2 + \cdots + x_n^2}\sqrt{y_1^2 + \cdots + y_n^2}}$$

위의 식처럼 정의되며 분자에는 벡터의 내적이 분모에는 벡터의 노름(크기)이 등장한다.
위 식의 핵심은 벡터를 정규화하고 내적을 구하는 것이다.

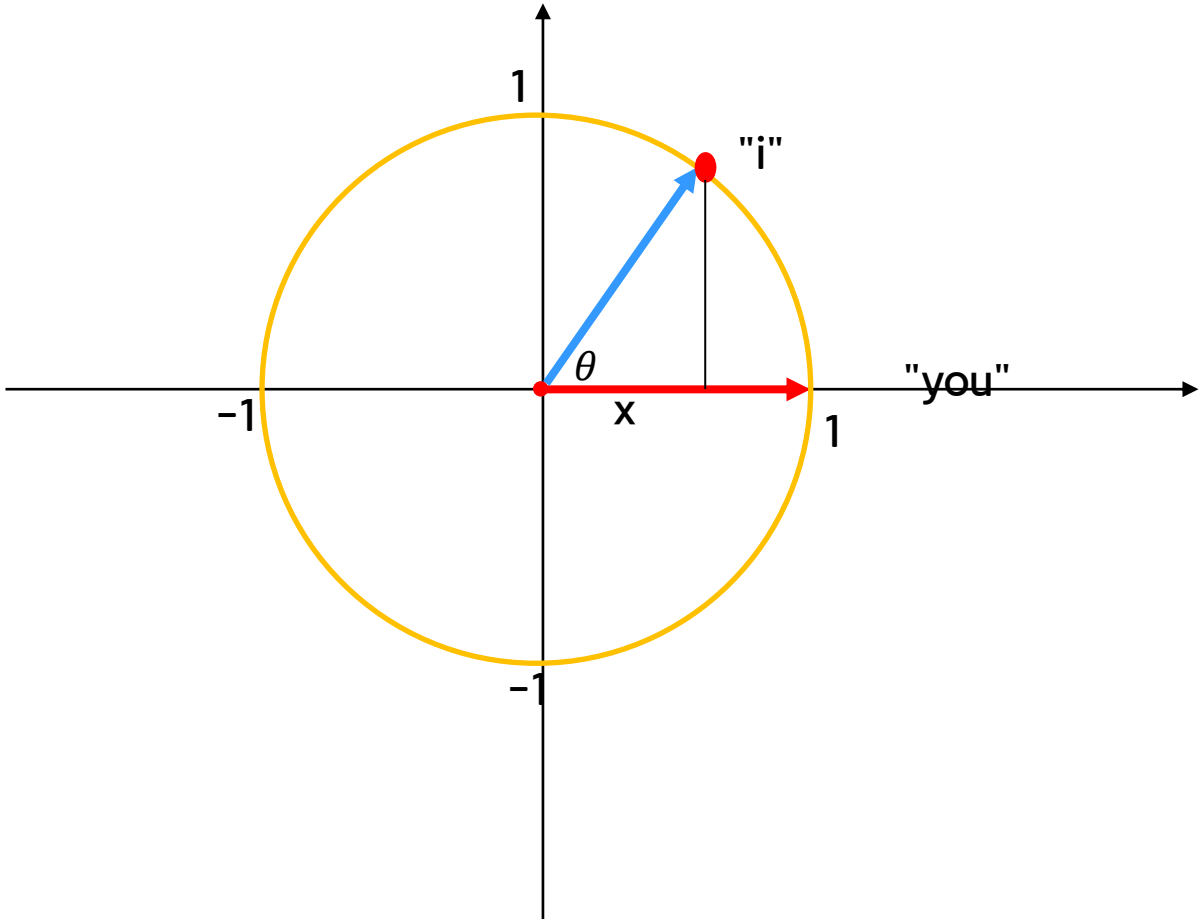






코사인 유사도

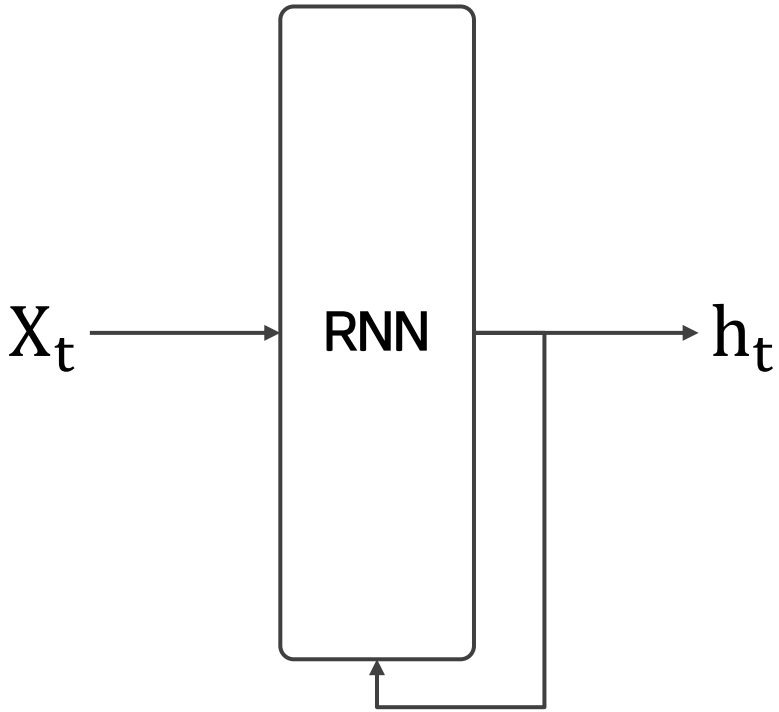
$$\cos(\theta) = \frac{x}{1}$$



순환하는 신경망

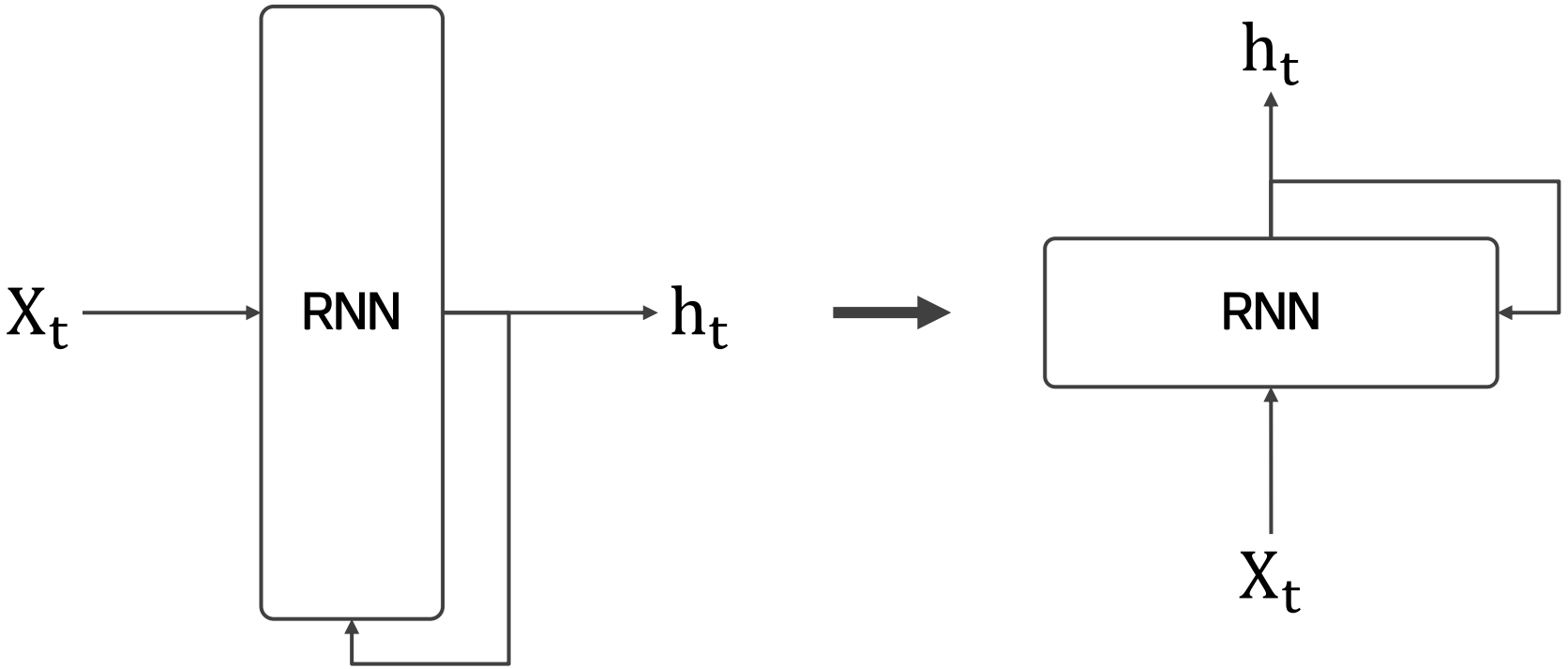
순환하기 위해서는 닫힌 경로가 필요하다.
닫힌 경로 혹은 순환하는 경로가 존재해야 데이터가 같은 장소를 반복해 왕래할 수 있고
데이터가 순환하면서 과거의 정보를 기억하는 동시에 최신 데이터로 갱신 될 수 있다.

순환 경로를 포함하는 RNN 계층



순환하는 신경망

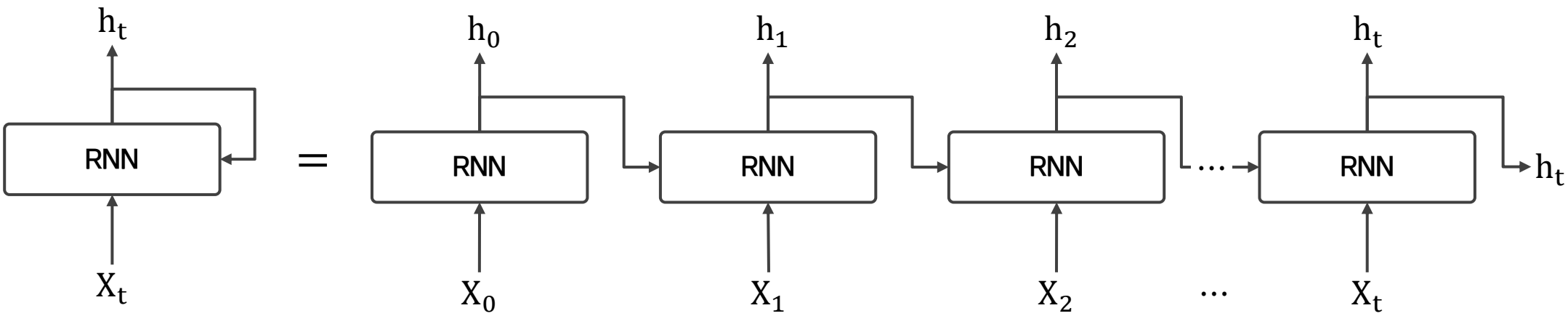
계층을 90도 회전시켜 그린다.



t: 시각
시계열 데이터(x0, x1,...xt, ...)가 RNN계층에 입력되고 이에 대응해 (h0, h1, ..., ht, ...)가 출력된다.
각 시각에 입력되는 xt를 벡터라고 가정했을 때
문장(단어 순서)을 다루는 경우를 예로 든다면 각 단어의 분산 표현(단어 벡터)이 xt가 되며 이 분산 표현이 순서대로 하나씩 RNN계층에 입력된다.

순환 구조 펼치기

RNN 계층의 순환 구조 펼치기



RNN계층의 순환 구조를 펼침으로써 오른쪽으로 성장하는 긴 신경망으로 변신
피드포워드 신경망(데이터가 한 방향으로만 흐른다)과 같은 구조이지만 위 그림에서는
다수의 RNN계층 모두가 실제로는 '같은 계층'인 것이 지금까지의 신경망과는 다른 점이다.

각 시각의 RNN계층은 그 계층으로의 입력과 1개 전의 RNN계층으로부터의 출력을 받는데
이 두 정보를 바탕으로 현 시각의 출력을 계산한다.

순환 구조 펼치기

$$h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

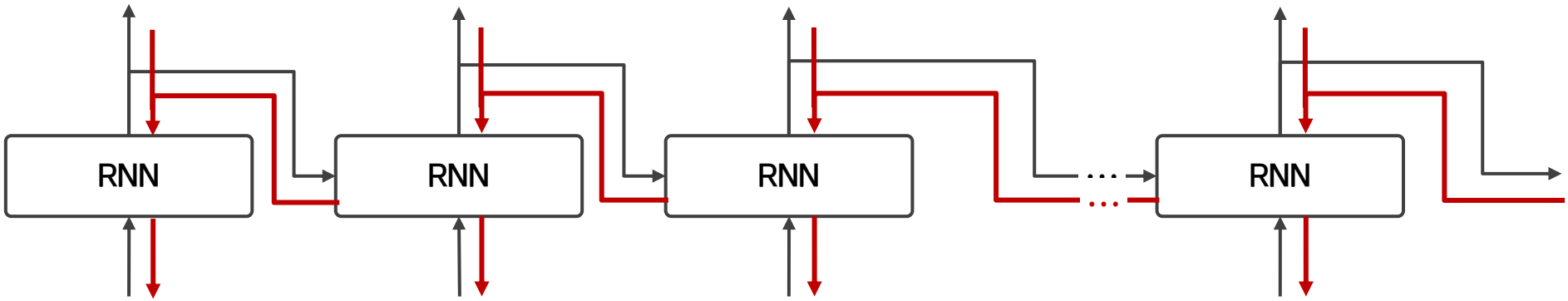
Wx: 입력 x를 출력 h로 변환하기 위한 가중치
Wh: 1개의 RNN출력을 다음 시각의 출력으로 변환하기 위한 가중치
b: 편향
h(t-1), xt: 행벡터

ht는 다른 계층을 향해 위쪽으로 출력되는 동시에 다음 시각의 RNN계층(자기 자신)을 향해 오른쪽으로도 출력된다.

RNN의 출력 ht는 은닉상태(hidden state) 혹은 은닉 상태 벡터(hidden state vector)라고 한다.

RNN은 h라는 '상태'를 가지고 있으며 위의 식의 형태로 갱신된다고 해석할 수 있다.
RNN계층을 '상태를 가지는 계층' 혹은 '메모리(기억력)가 있는 계층'이라고 한다.

순환 구조를 펼친 RNN 계층에서의 오차역전파



순환 구조를 펼친 후의 RNN에는 (일반적인) 오차역전파법을 적용할 수 있다.
먼저 순전파를 수행하고 이어서 역전파를 수행하여 원하는 기울기를 구할 수 있다.
여기서의 오차역전파법은 '시간 방향으로 펼친 신경망의 오차역전파법'이란 뜻으로
BPTT(Backpropagation Through Time)라고 한다.

문제점

- 시계열 데이터의 시간 크기가 커지는 것에 비례하여 BPTT가 소비하는 컴퓨팅 자원도 증가
- 시간 크기가 커지면 역전파 시의 기울기가 불안정해짐

Truncated BPTT : 시간축 방향으로 너무 길어진 신경망을 적당한 지점에서 잘라내어
작은 신경망 여러 개로 만들어 잘라낸 작은 신경망에서 오차역전파법을 수행한다.

- 계층이 너무 길면 계산량과 메모리 사용량 등이 문제가 되고 계층이 길어짐에 따라 신경망을 하나 통과할 때마다 기울기 값이 조금씩 작아져서 이전 시각 t까지 역전파되기 전에 0이 되어 소멸할 수도 있다.

- 순전파의 연결을 그대로 유지하면서(데이터를 순서대로 입력해야 한다) 역전파의 연결은 적당한 길이로 잘라내 잘라낸 신경망 단위로 학습을 수행한다.

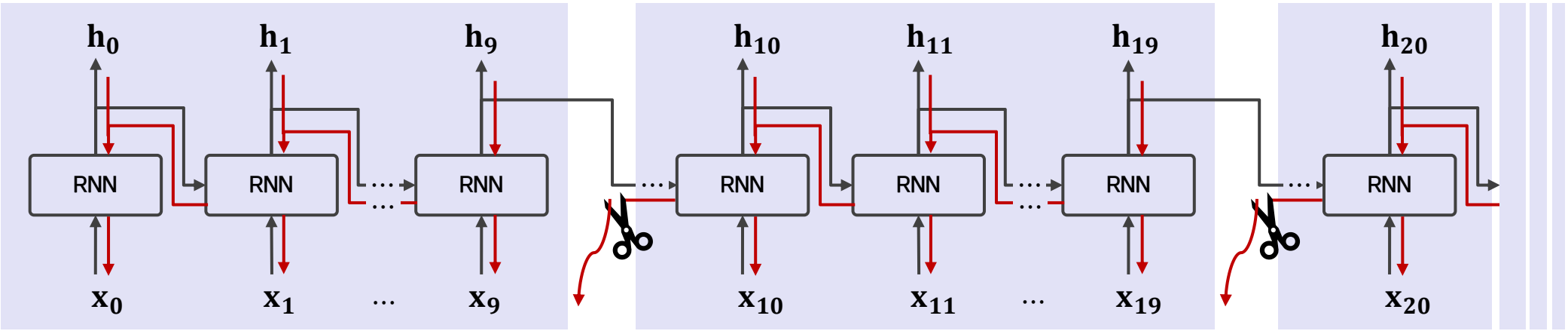
- 역전파의 연결을 잘라버리면 그보다 미래의 데이터에 대해서는 생각할 필요가 없어지기 때문에 각각의 블록 단위로 미래의 블록과는 독립적으로 오차역전파법을 완결시킨다.
  - 블록: 역전파가 연결되는 일련의 RNN계층

- 순전파를 수행하고 그 다음 역전파를 수행하여 원하는 기울기를 구한다.

- 다음 역전파를 수행할 때 앞 블록의 마지막 은닉 상태인 ht가 필요하다.
  ht로 순전파가 계속 연결될 수 있다.

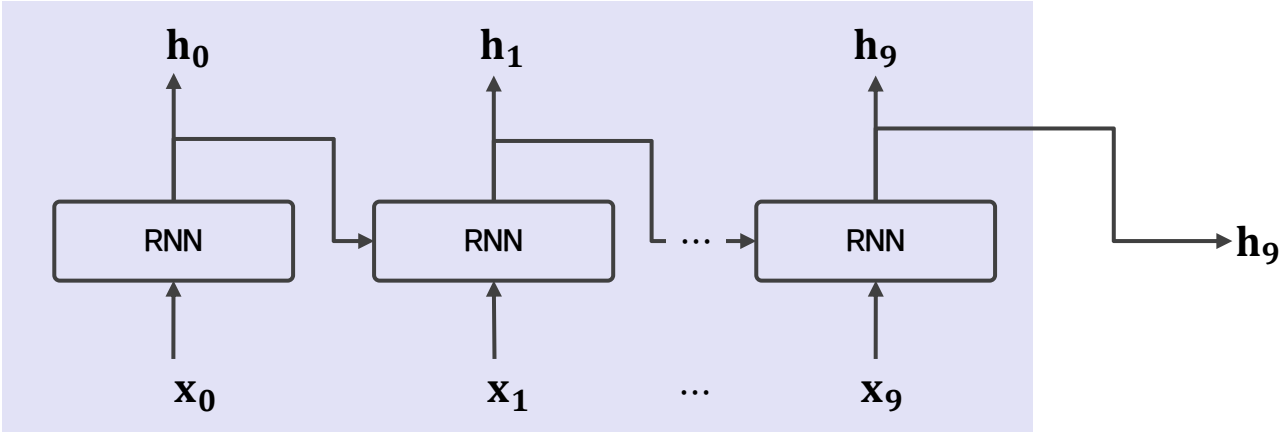
Truncated BPTT

역전파의 연결을 적당한 지점에서 끊는다.

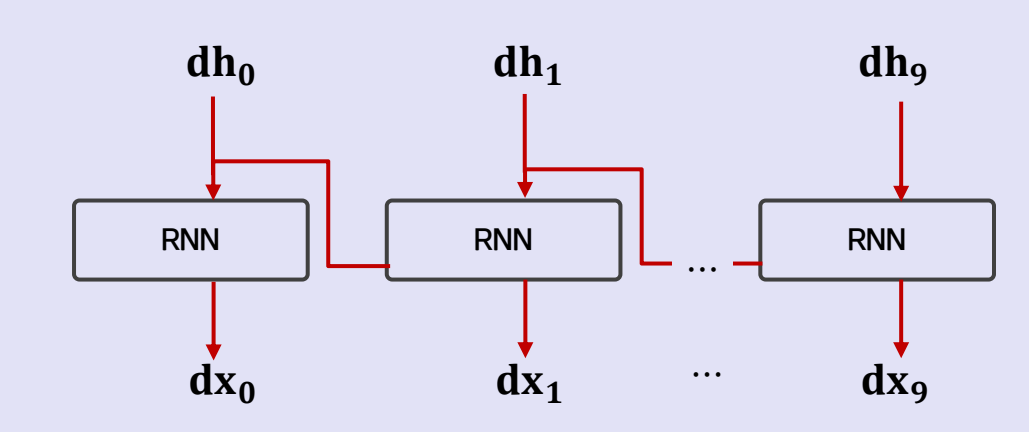


첫 번째 블록의 순전파와 순전파 : 이보다 앞선 시각으로부터의 기울기는 끊겼기 때문에 이 블록 내에서만 오차역전파법이 완결된다.

순전파

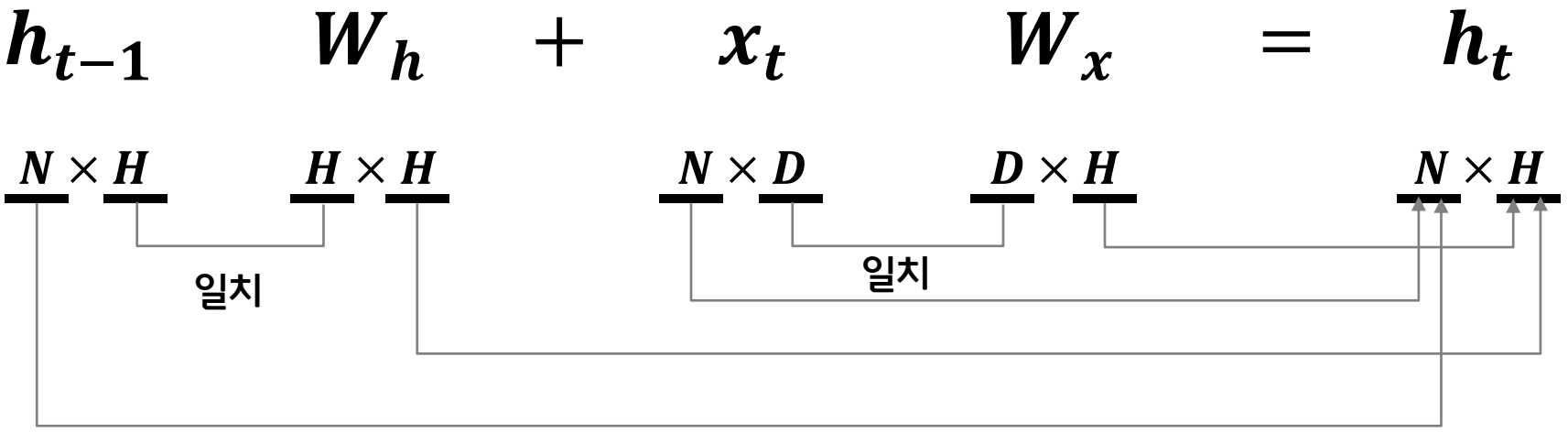


역전파



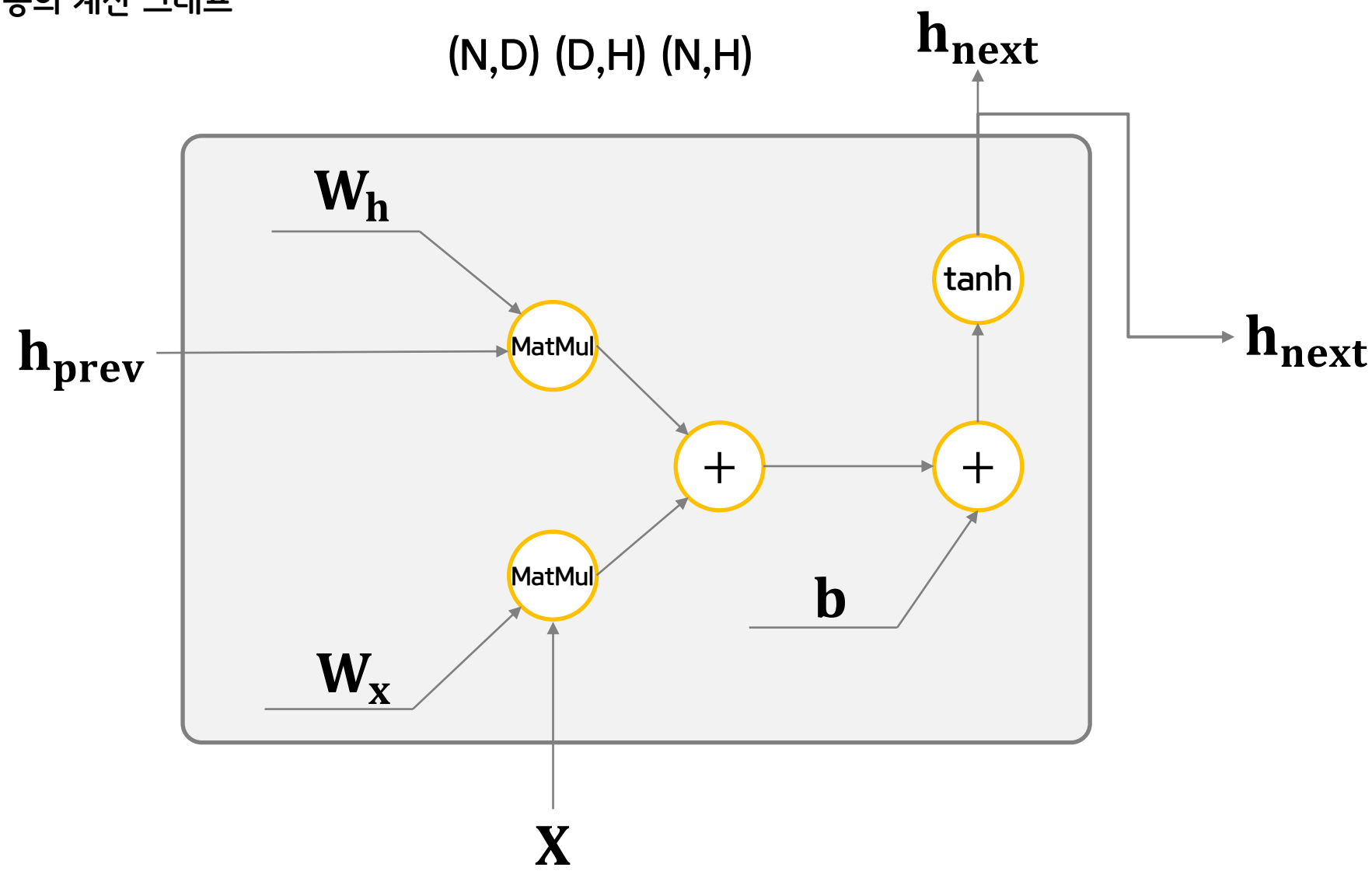
$$h_t = tanh(h_{t-1}W_h + x_tW_x + b)$$

형상 확인 : 형렬 곱에서는 대응하는 차원의 원소 수를 일치시킨다.

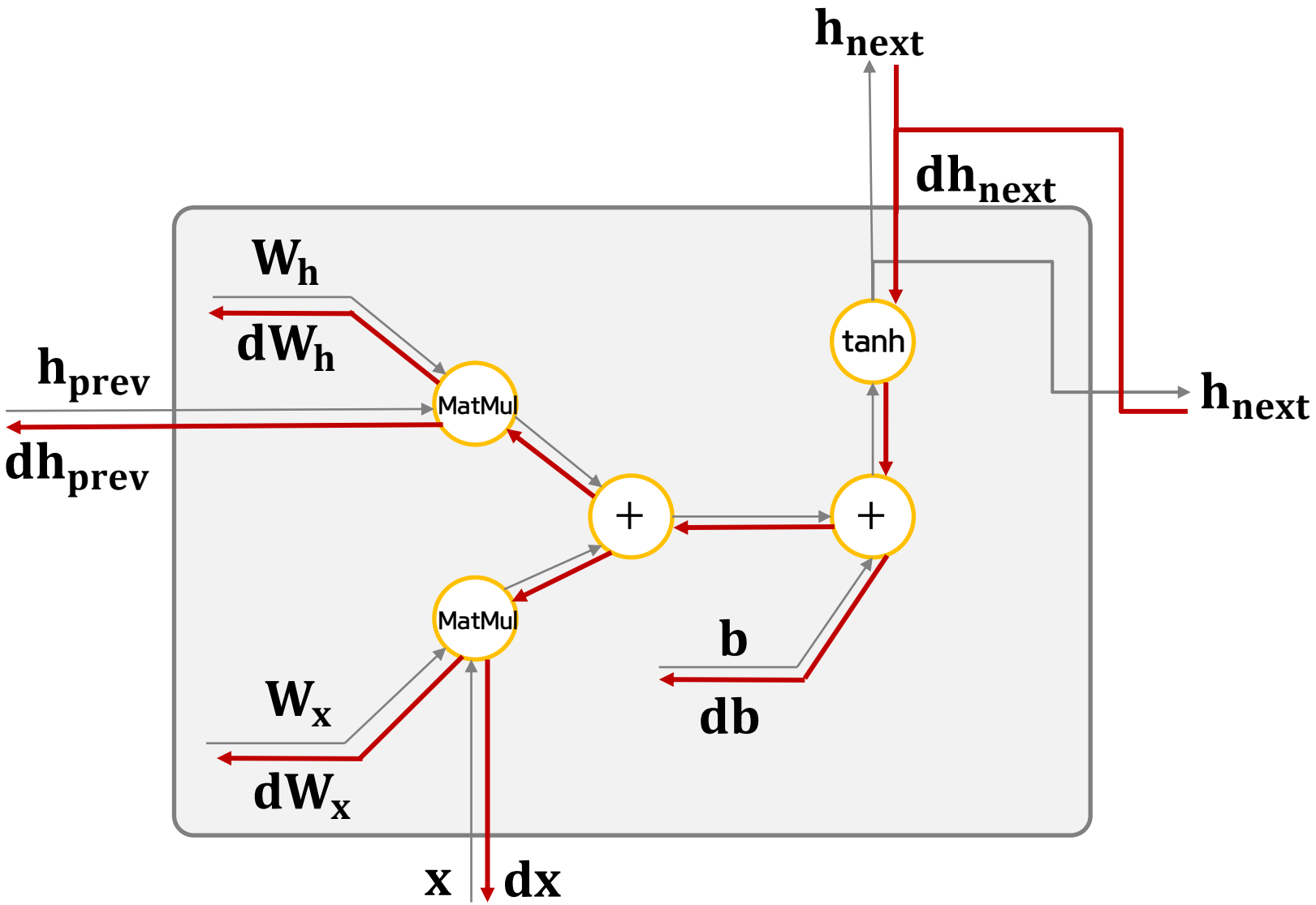


N: 미니배치 크기 D: 입력 벡터의 차원 수 H: 은닉 상태 벡터의 차원 수

RNN 계층의 계산 그래프

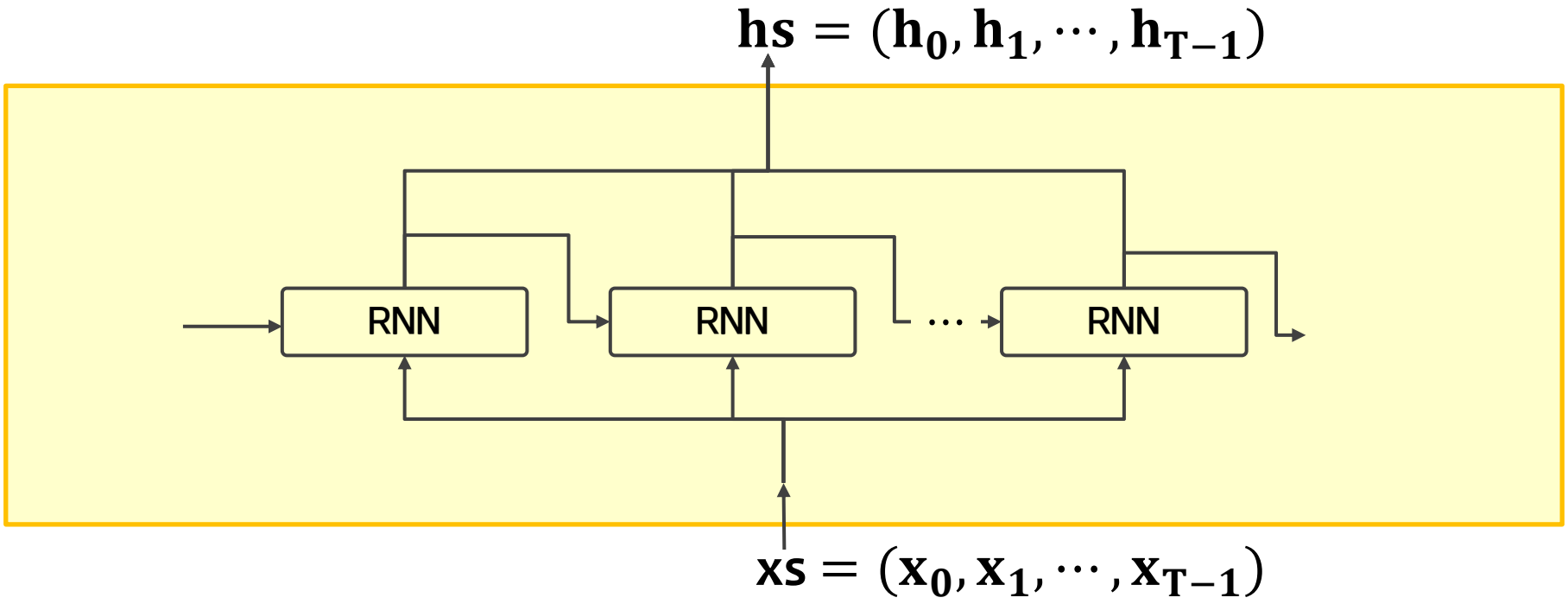


RNN 계층의 계산 그래프(역전파 포함)

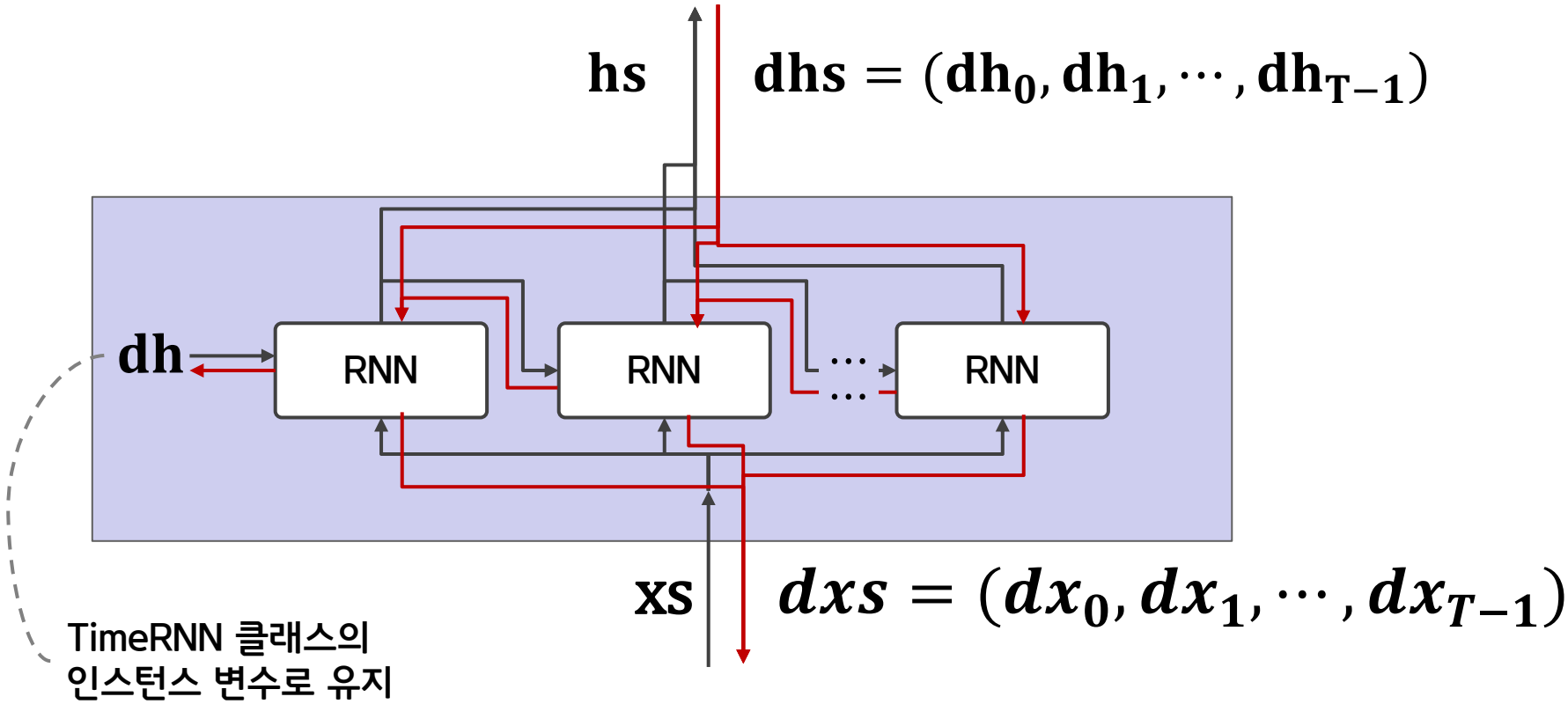


Time RNN 계층의 계산 그래프

Time RNN

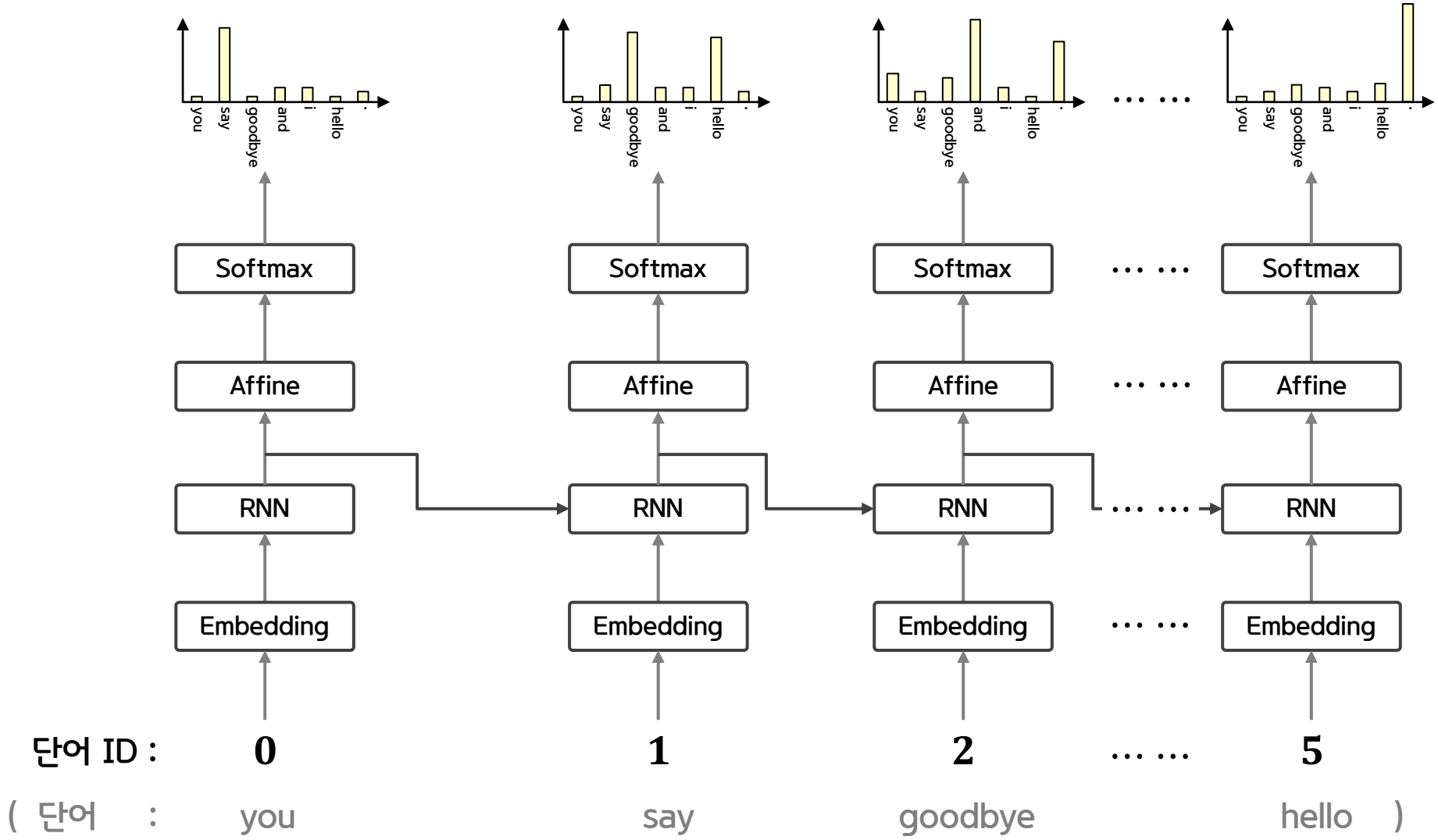


Time RNN 계층의 역전파



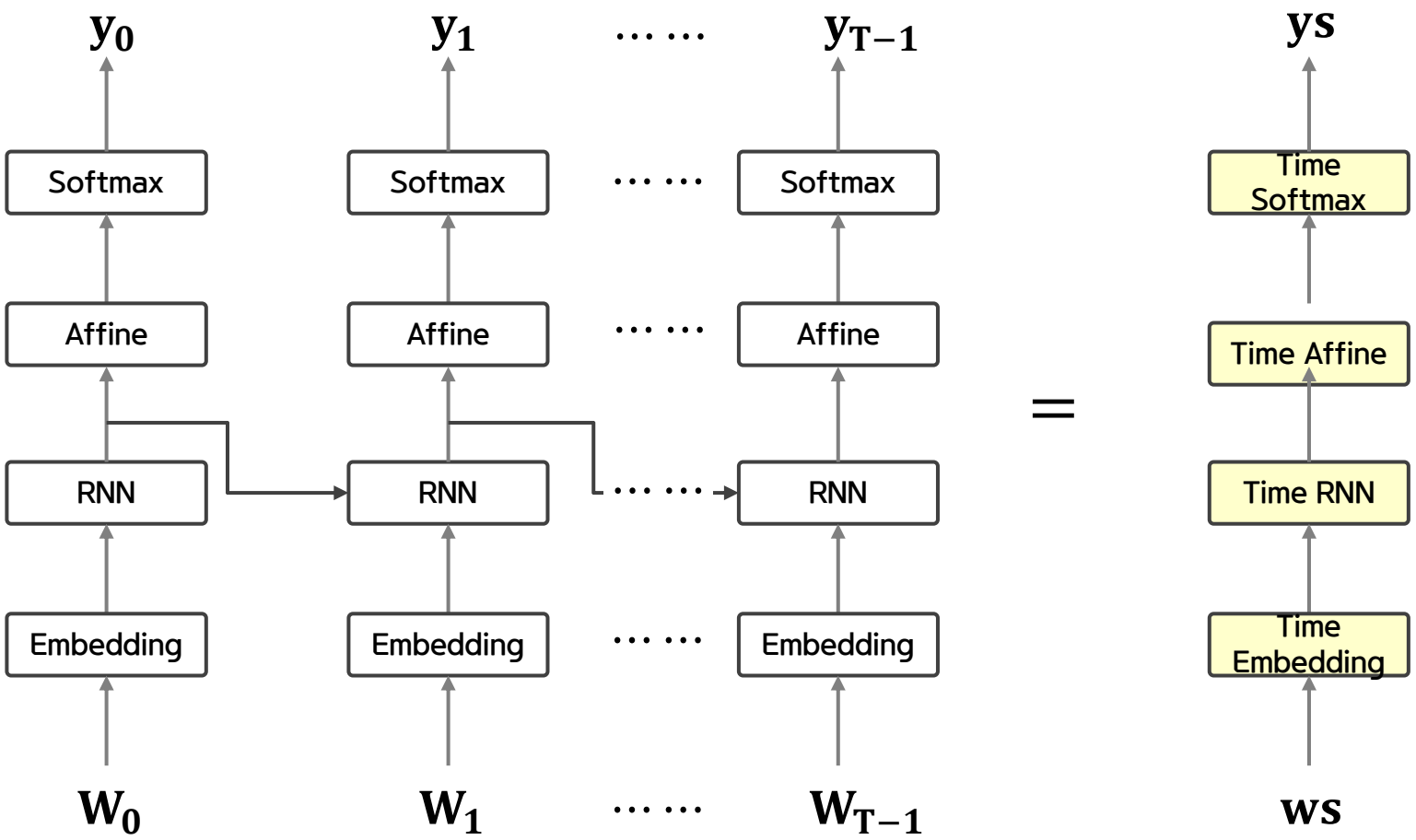
RNNLM의 전체 그림

샘플 말뭉치로 "you say goodbye and i say hello."를 처리하는 RNNLM의 예



Time 계층 구현

시계열 데이터를 한꺼번에 처리하는 계층을 Time XX 계층으로 구현



기울기 소실 또는 기울기 폭발

"?"에 들어갈 단어는 ? : (어느 정도의)장기 기억이 필요한 문제의 예

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

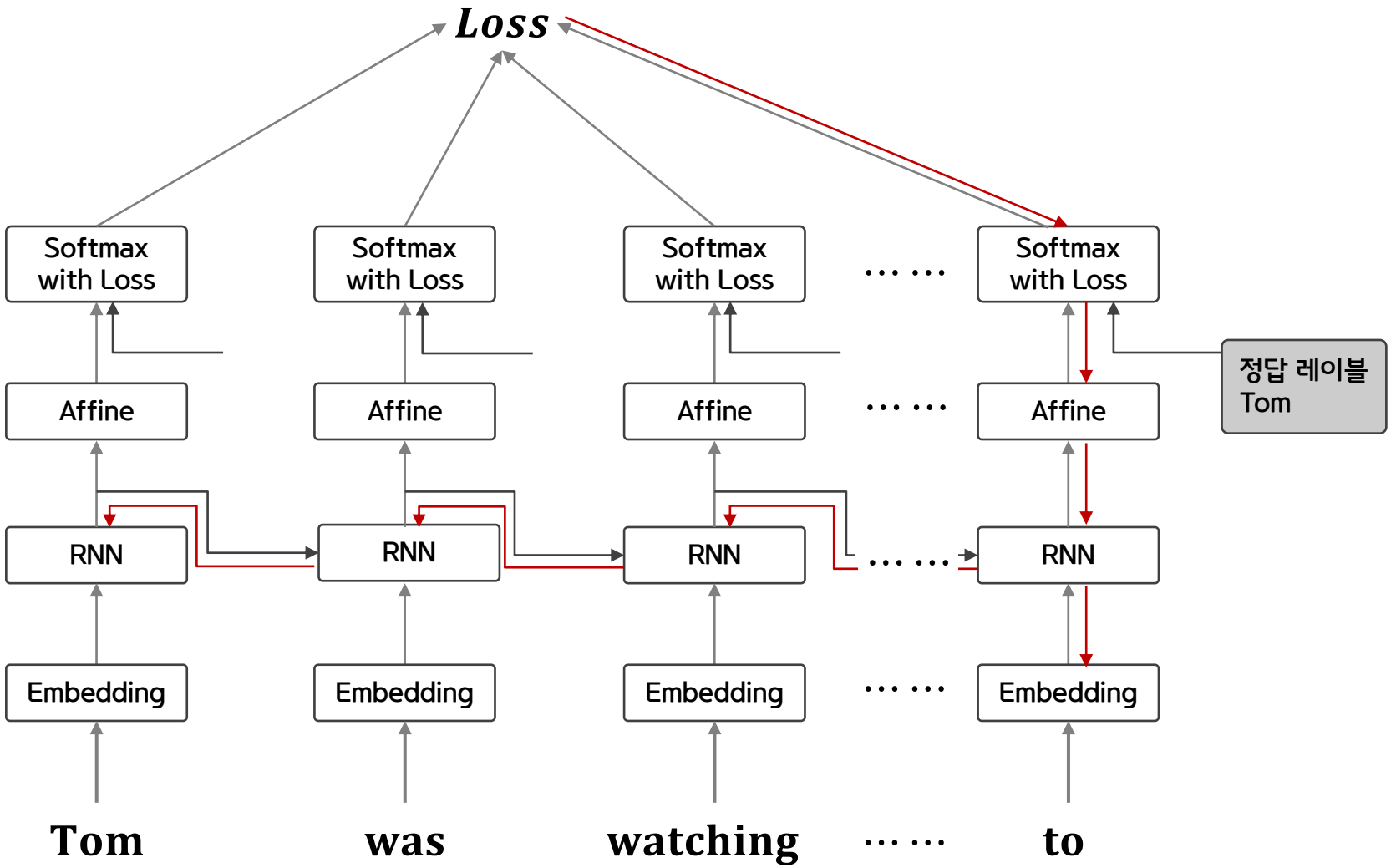
?

언어 모델은 주어진 단어들을 기초로 다음에 출현할 단어를 예측하는 일을 한다.
이번 절에서는 RNNLM의 단점을 확인하는 차원에서 다음의 문제를 다시 생각해보았다.

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?
여기서 ?에 들어갈 단어는 Tom이다. RNNLM이 이 문제에 올바르게 답하라면,
현재 맥락에서 'Tom이 방에서 TV를 보고 있음'과 '그 방에 Mary가 들어옴'이란 정보를 기억해야 한다.
즉, 이런 정보를 RNN 계층의 은닉 상태에 인코딩해 보관해야 한다.

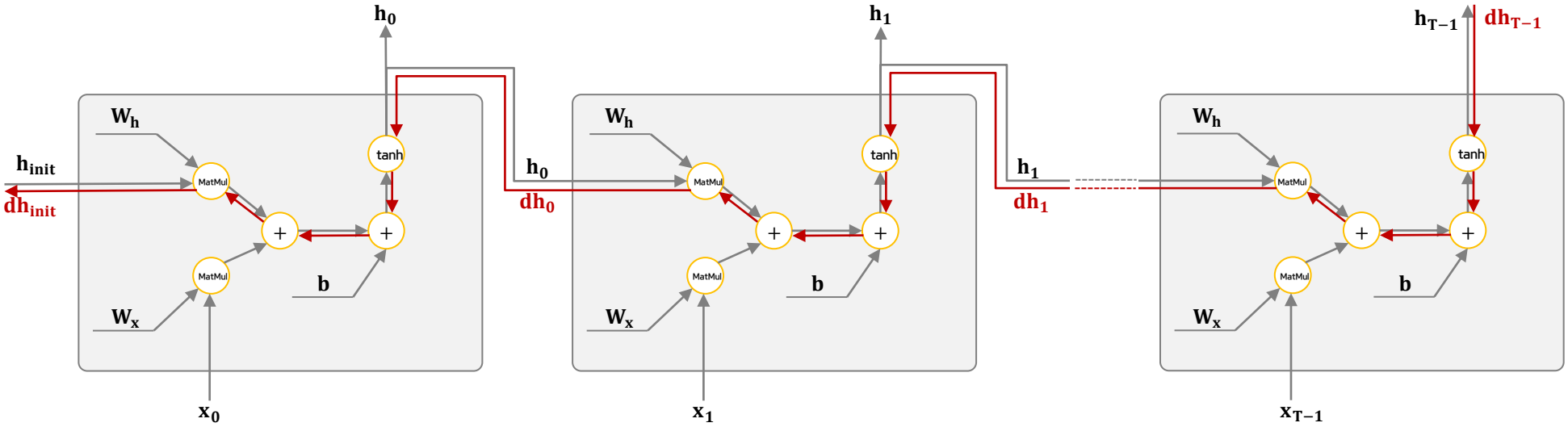
기울기 소실 또는 기울기 폭발

정답 레이블이 "Tom"임을 학습할 때의 기울기 흐름



기울기 소실 또는 기울기 폭발의 원인

RNN 계층에서 시간 방향으로의 기울기 전파

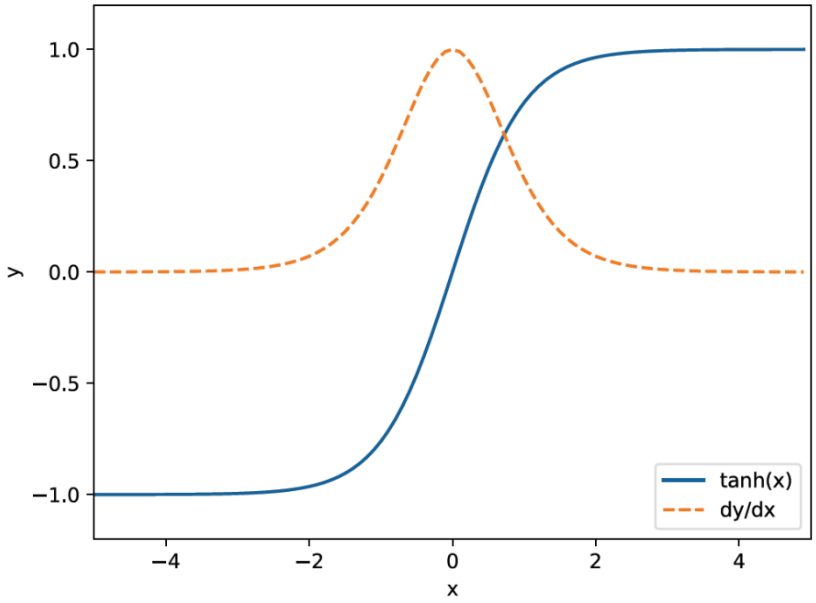


위의 RNN 계층의 그림에서 시간 방향 기울기 전파에만 주목해보자.
길이가 T인 시계열 데이터를 가정하여 T번째 정답 레이블로부터 전해지는 기울기가 어떻게 변하는지 보자.
앞의 문제에 대입하면 T번째 정답 레이블이 'Tom'인 경우에 해당한다.
이때 시간 방향 기울기에 주목하면 역전파로 전해지는 기울기는 차례로
'tanh','+','MatMul(행렬 곱)' 연산을 통과한다는 것을 알 수 있다.

'+'의 역전파는 상류에서 전해지는 기울기를 그대로 하류로 흘려보내 기울기는 변하지 않는다.

기울기 소실 또는 기울기 폭발의 원인

y=tanh(x)의 그래프(점선은 미분)

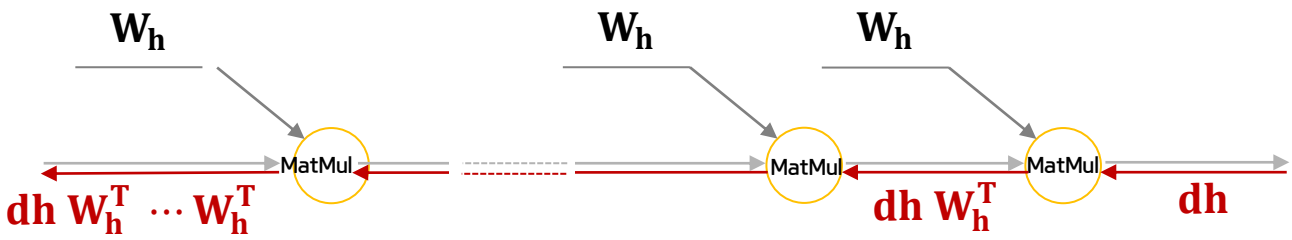


그림에서 점선이 y=tanh(x)의 미분이고 값은 1.0 이하이며, x가 0으로부터 멀어질수록 작아진다.
즉, 역전파에서 기울기가 tanh 노드를 지날 때마다 값은 계속 작아진다는 의미이다.
그리고 tanh 함수를 T번 통과하면 기울기도 T번 반복해서 작아진다.

'MatMul(행렬 곱)' 노드의 경우 tanh 노드를 무시하기로 한다.
그러면 RNN 계층의 역전파 시 기울기는 'MatMul' 연산에 의해서만 변화하게 된다.

기울기 소실 또는 기울기 폭발의 원인

RNN 계층의 행렬 곱에만 주목했을 때의 역전파의 기울기



상류로부터 dh라는 기울기가 흘러온다고 가정하고 이때 MatMul 노드에서의 역전파는
dh(W_h^T)라는 행렬 곱으로 기울기를 계산한다.

그리고 같은 계산을 시계열 데이터의 시간 크기만큼 반복한다.
주의할 점은 행렬 곱셈에서 매번 똑같은 W_h 가중치를 쓴다는 것이다.

즉, 행렬 곱의 기울기는 시간에 비례해 지수적으로 증가/감소함을 알 수 있으며
증가할 경우 기울기 폭발이라고 한다. 기울기 폭발이 일어나면 오버플로를 일으켜 NaN 같은 값을 발생시킨다.
반대로 기울기가 감소하면 기울기 소실이 일어나고 이는 일정 수준 이하로 작아지면 가중치 매개변수가
더 이상 갱신되지 않으므로 장기 의존 관계를 학습할 수 없게 된다.

기울기 폭발 대책

기울기 클리핑(gradients clipping)

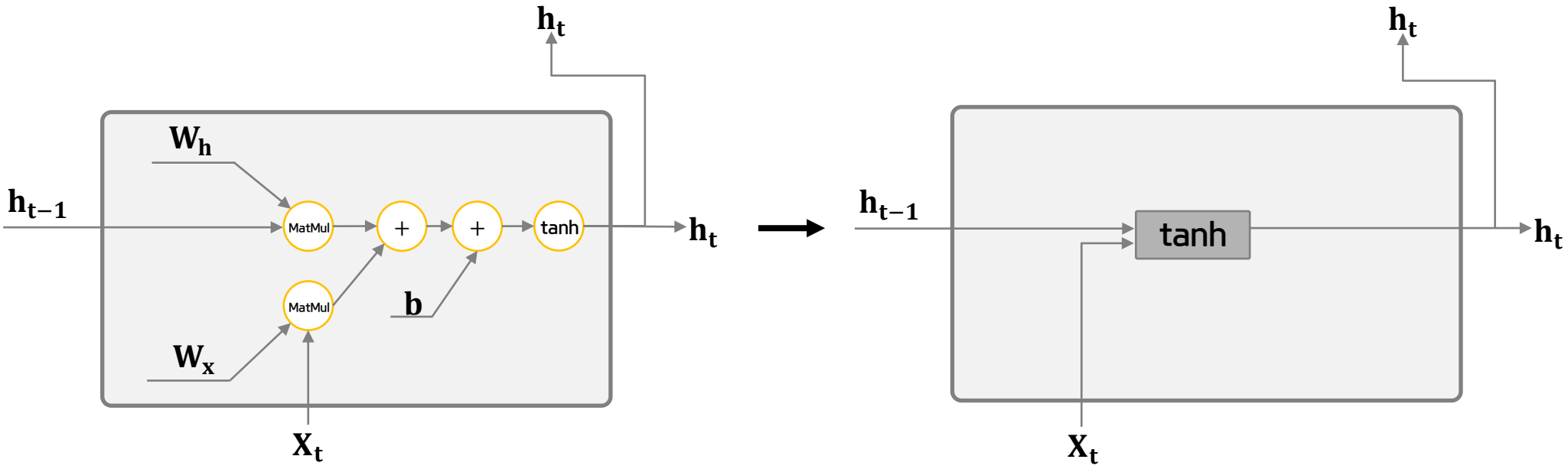
$$if\ \|\hat{g}\| \geq threshold:$$

$$\hat{g} = \frac{threshold}{\|\hat{g}\|}\hat{g}$$

LSTM의 인터페이스

RNN 학습에서 기울기 소실도 큰 문제이다.
이 문제를 해결하려면 RNN 계층의 아키텍처를 근본부터 뜯어고쳐야 한다.

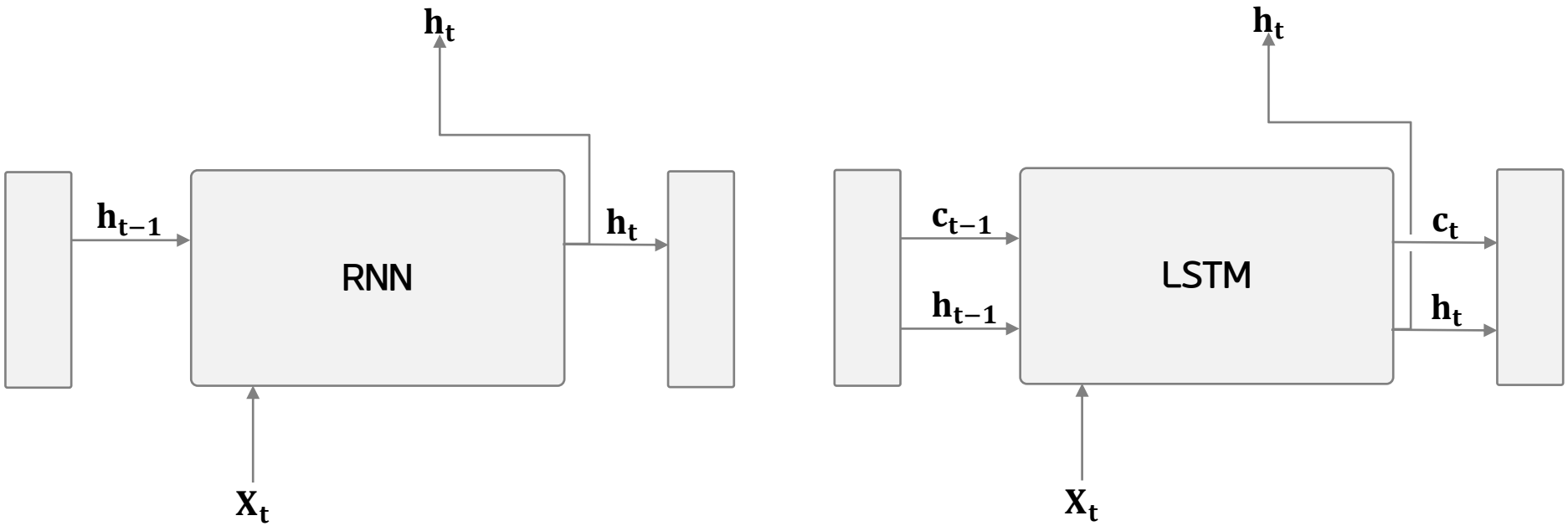
단순화한 도법을 적용한 RNN 계층



여기서는 $\tanh(h_{t-1}W_h + x_tW_x + b)$ 계산을 tanh라는 직사각형 노드 하나로
그리고 직사각형 노드 안에 행렬 곱과 편향의 합, 그리고 tanh 함수에 의한 변환이 모두 포함된 것이다.

LSTM의 인터페이스

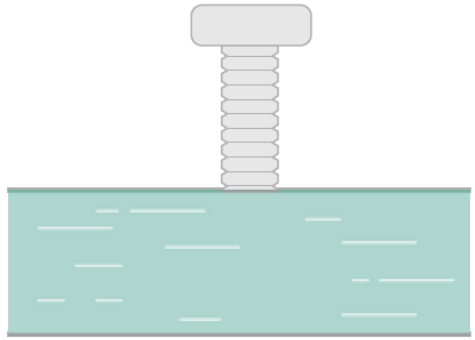
RNN 계층과 LSTM 계층 비교

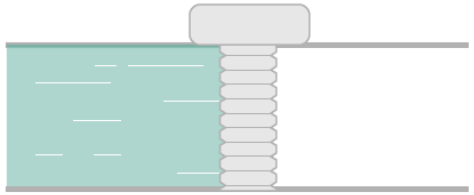


이 그림에서는 LSTM 계층의 인터페이스에는 c라는 경로가 있다는 차이가 있다.
여기서 c를 기억 셀이라 하며 LSTM 전용의 기억 메커니즘이다.

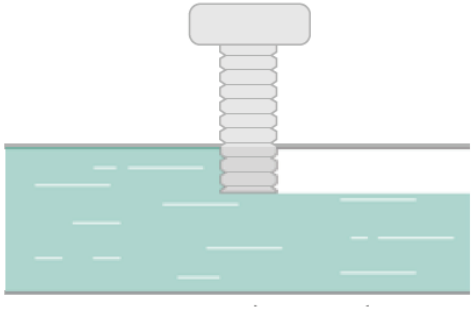
기억 셀의 특징은 데이터를 LSTM 계층 내에서만 주고받는다는 것이다.
다른 계층으로는 출력하지 않는다는 것이다. 반면, LSTM의 은닉 상태 h는
RNN 계층과 마찬가지로 다른 계층, 위쪽으로 출력된다.

비유하자면 게이트는 물의 흐름을 제어한다.

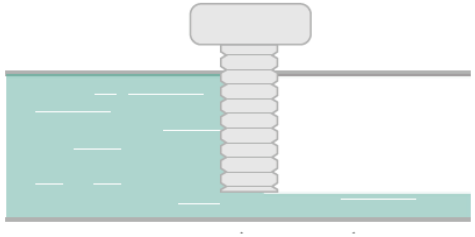




물이 흐르는 양을 0.0~1.0 범위에서 제어한다.



0.7(70%)



0.2(20%)

$\tanh(c_t)$의 각 원소에 대해 '그것이 다음 시각의 은닉 상태에 얼마나 중요한가'를 조정한다.
한편, 이 게이트는 다음 은닉 상태 h_t의 출력을 담당하는 게이트이므로 output 게이트라고 한다.

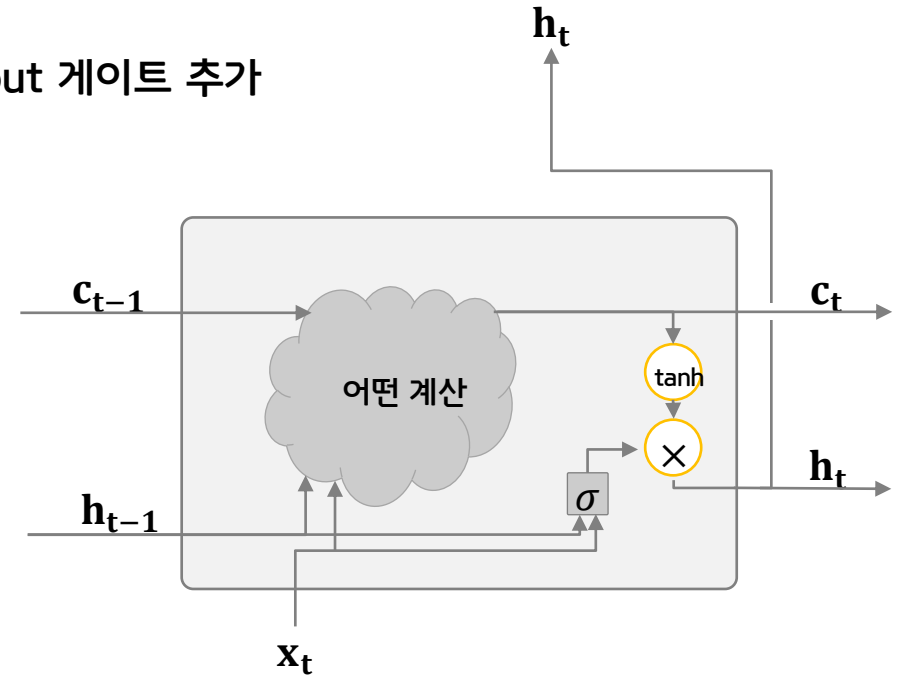
output 게이트의 열림 상태는 입력x_t와 이전 상태 h_{t-1}로부터 구한다.

이때의 식은 밑에 식과 같다.

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)})$$

밑에 그림에서 output 게이트에서 수행하는 식의 계산을 sigma로 표기했다.
sigma의 출력을 o라고 하면 $\mathbf{h_t}$ 는 o와 $\mathbf{tanh(c_t)}$의 곱으로 계산된다.

output 게이트 추가



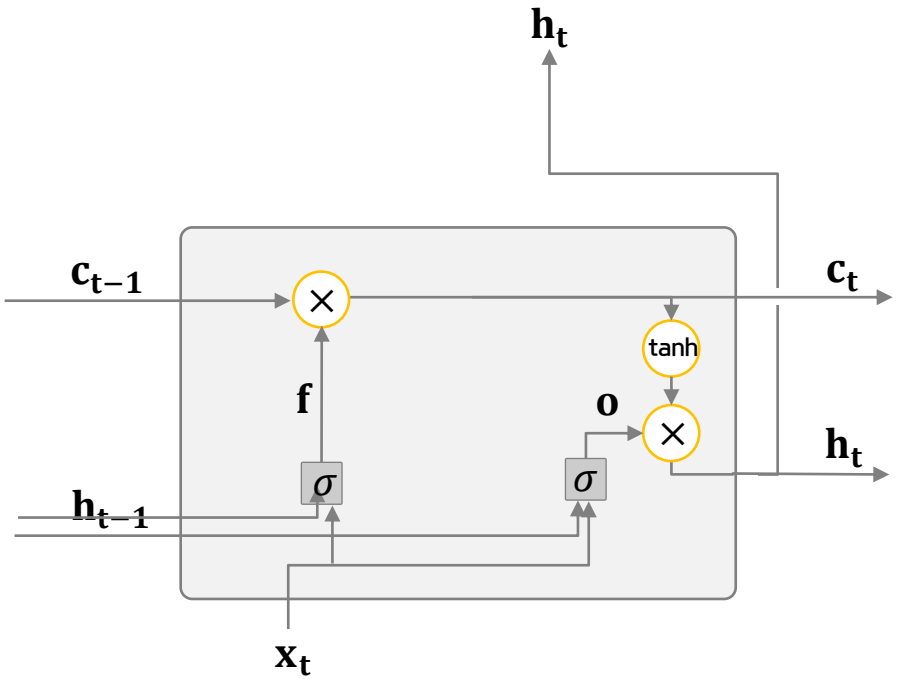
아다마르 곱(Hadamard product)

$$h_t = o \odot \tanh(c_t)$$

forget 게이트

다음에 해야 할 일은 기억 셀에 '무엇을 잊을까'를 명확하게 지시하는 것이다.

forget 게이트 추가

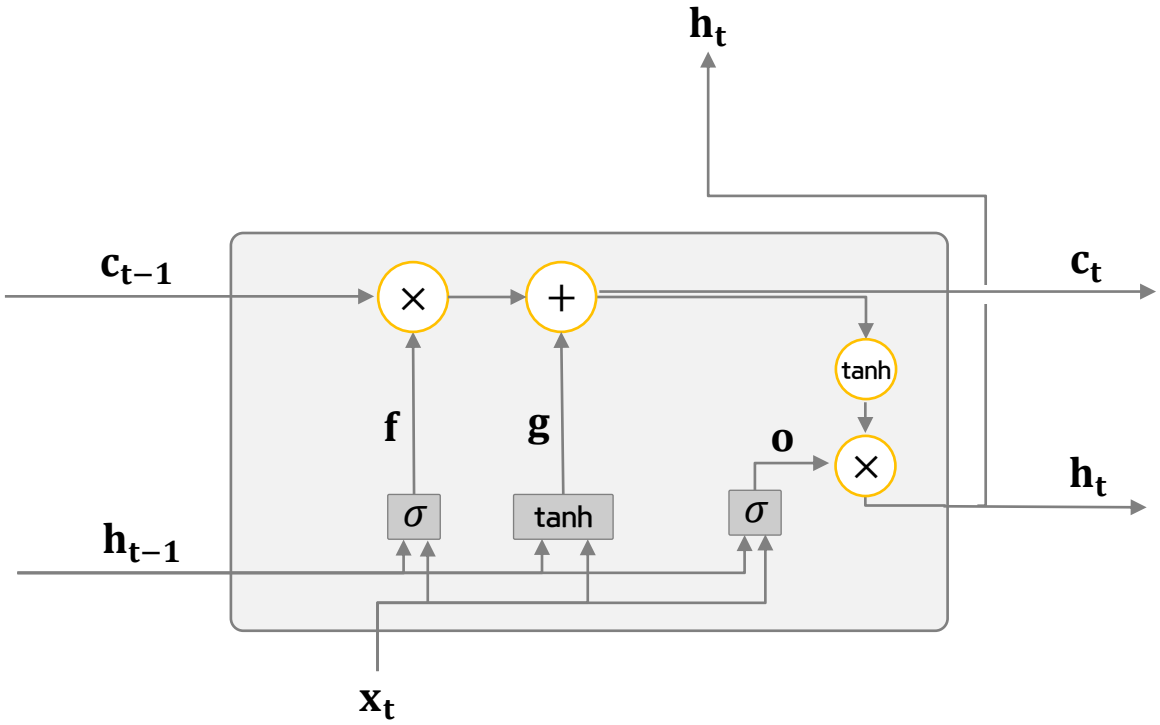


$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

새로운 기억 셀

forget 게이트를 거치면서 이전 시각의 기억 셀로부터 잊어야 할 기억이 삭제되었다.

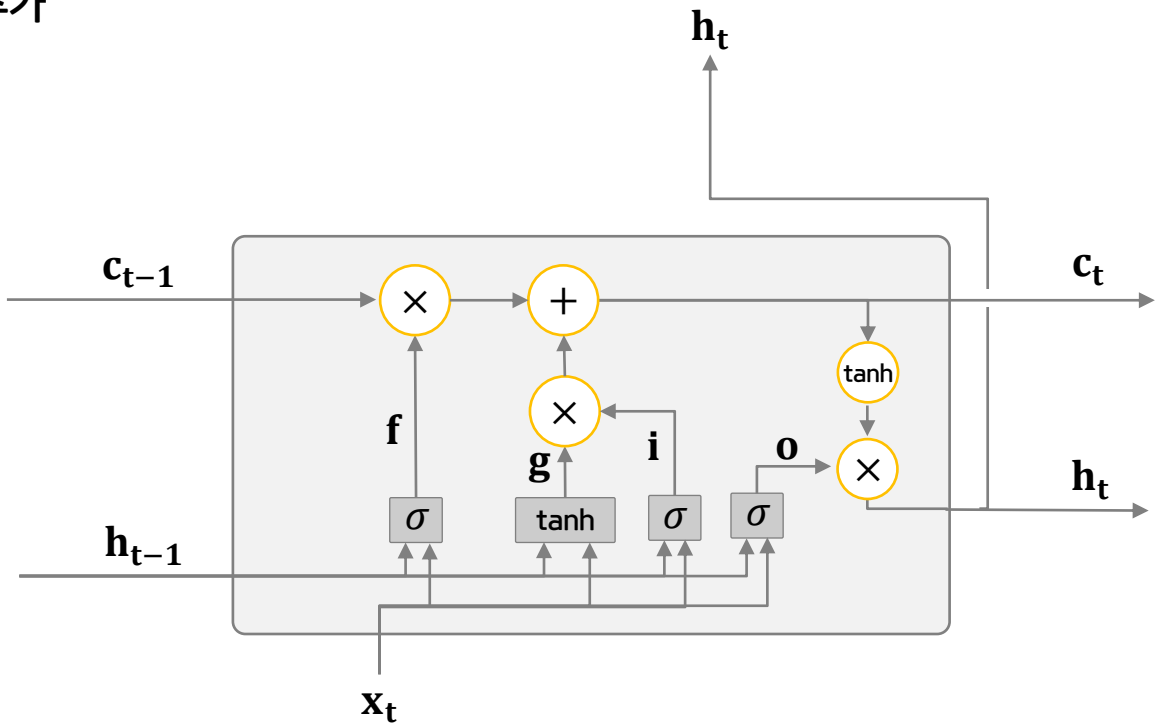
새로운 기억 셀



$$g = tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

input 게이트

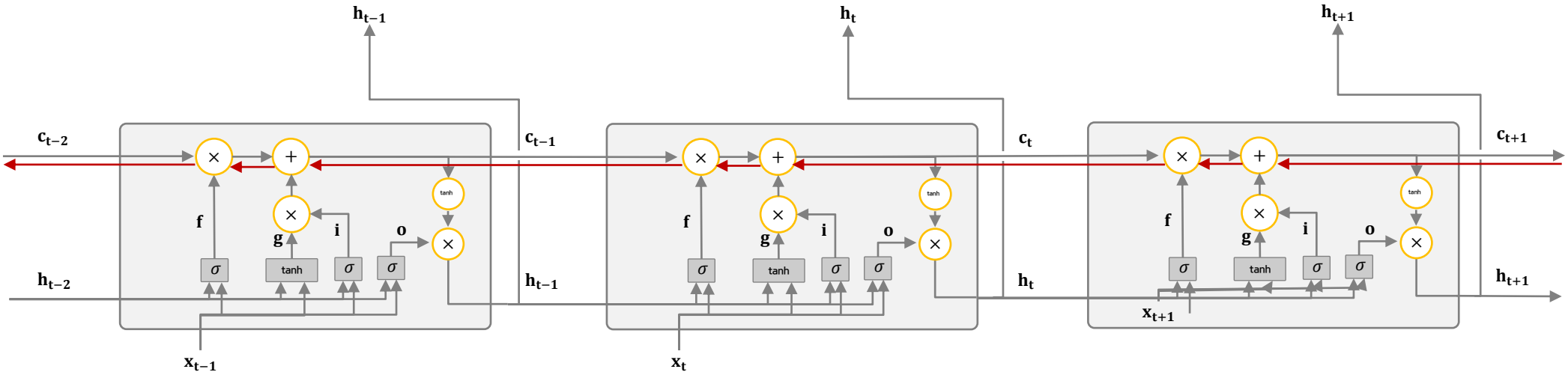
input 게이트 추가



$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$

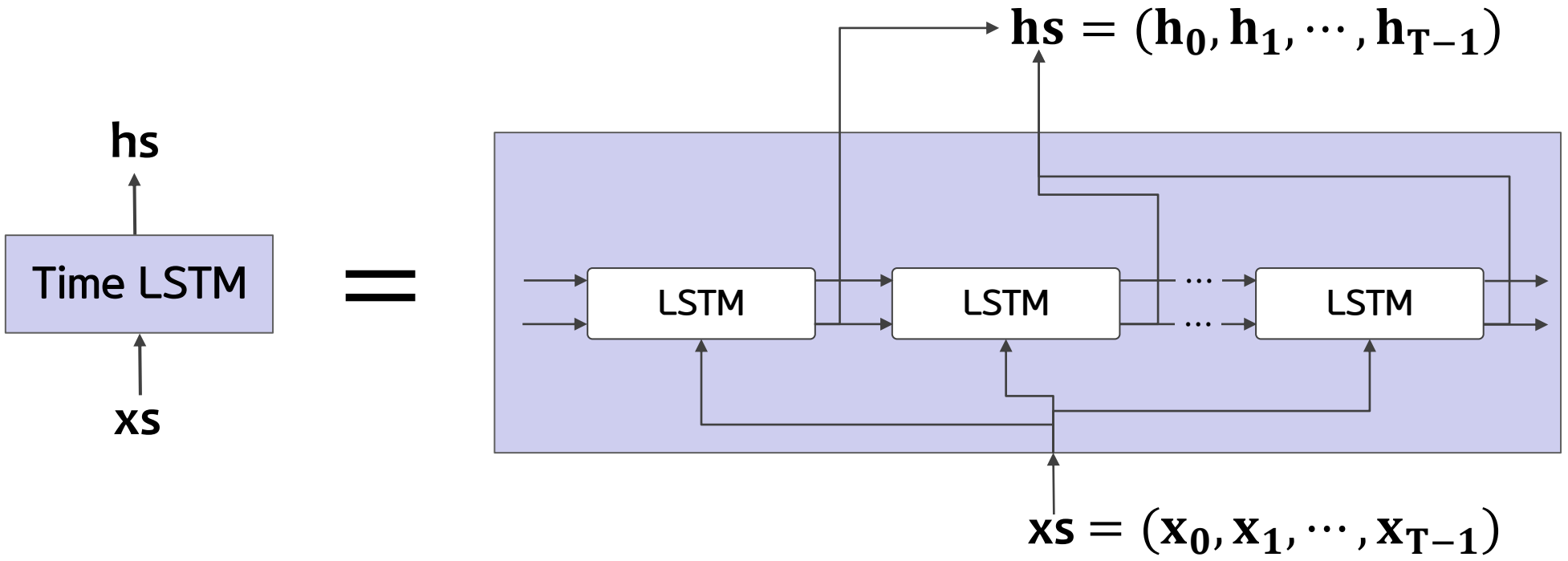
LSTM의 기울기 흐름

기억 셀의 역전파

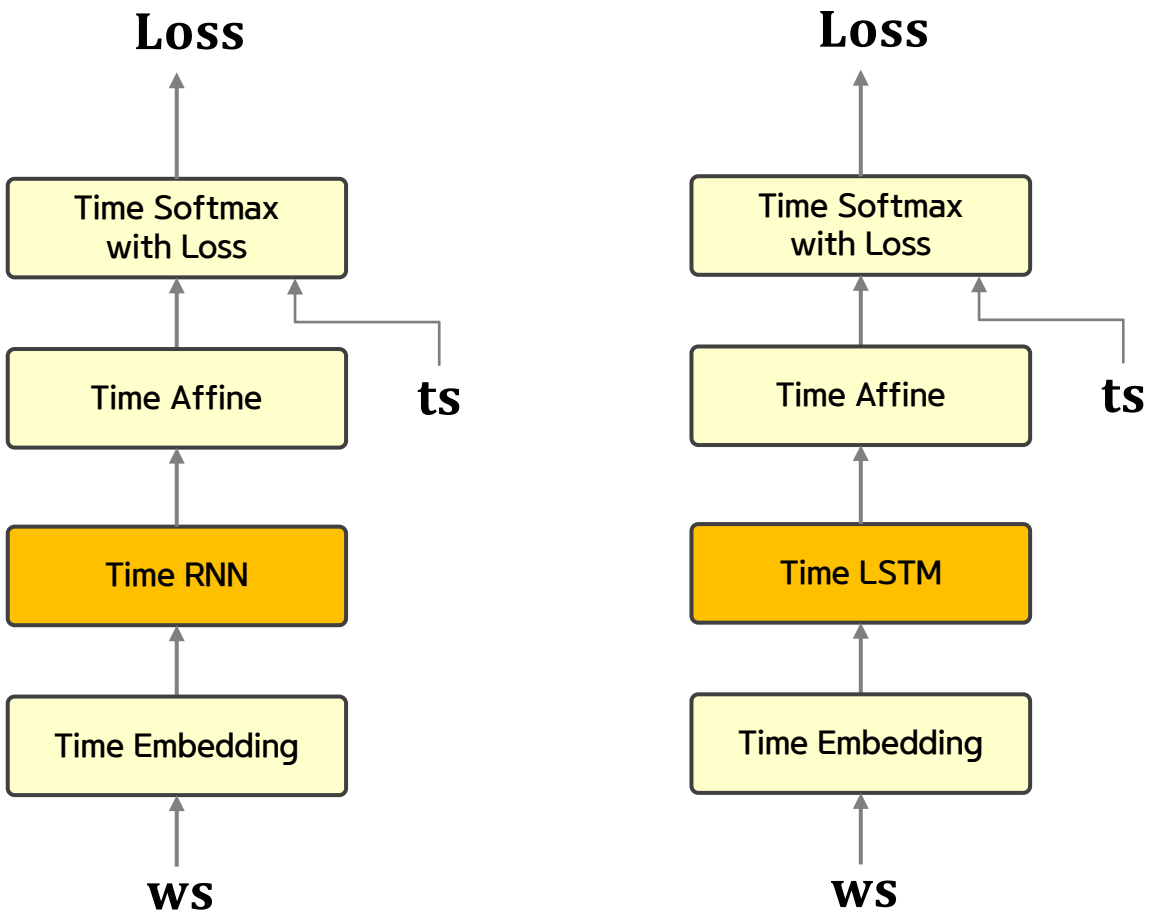


Time LSTM의 구현

Time LSTM의 입출력

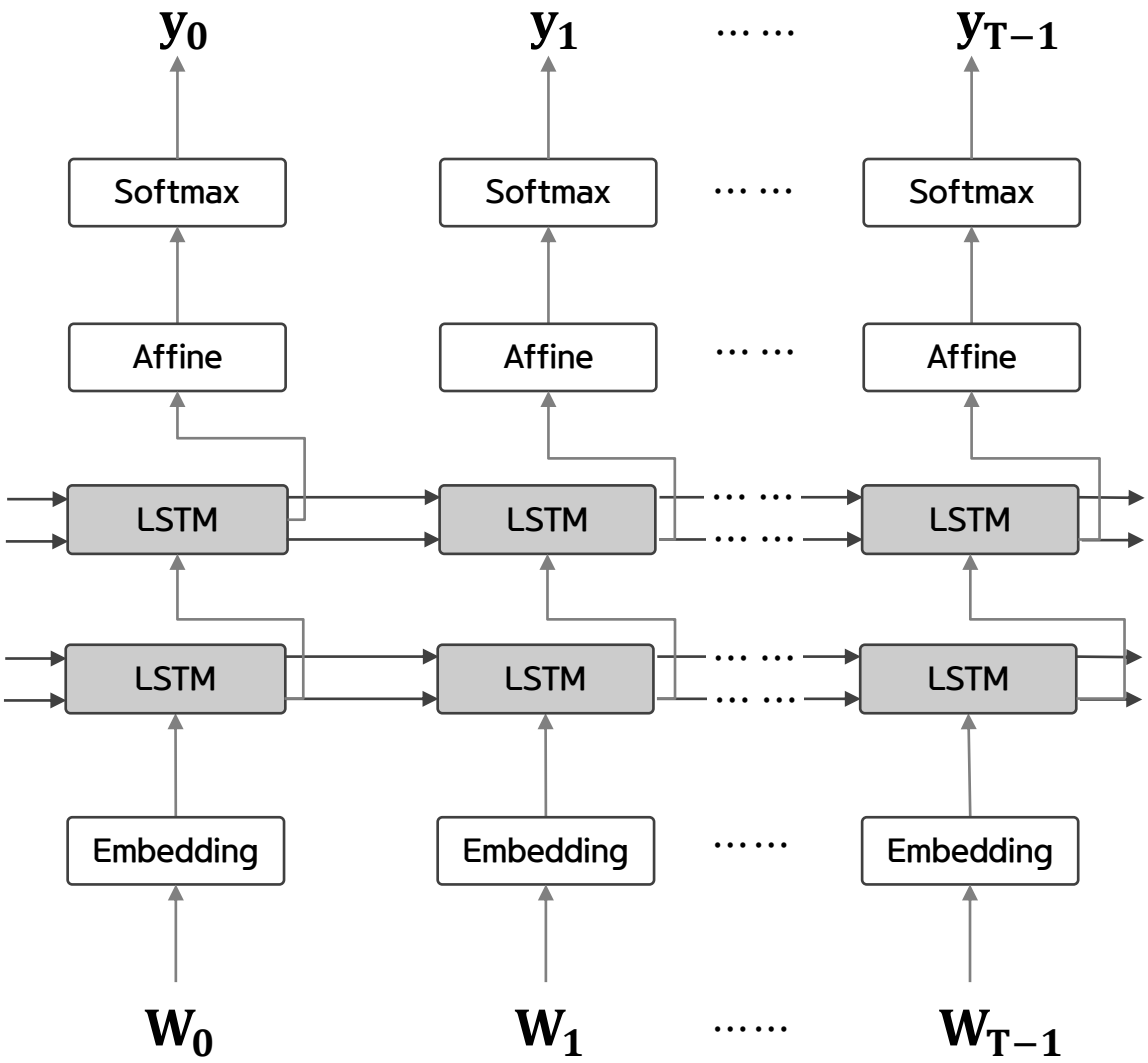


언어 모델의 신경망 구성


Loss
Time Softmax with Loss
Time Affine
ts
Time RNN
Time Embedding
ws
Loss
Time Softmax with Loss
Time Affine
ts
Time LSTM
Time Embedding
ws

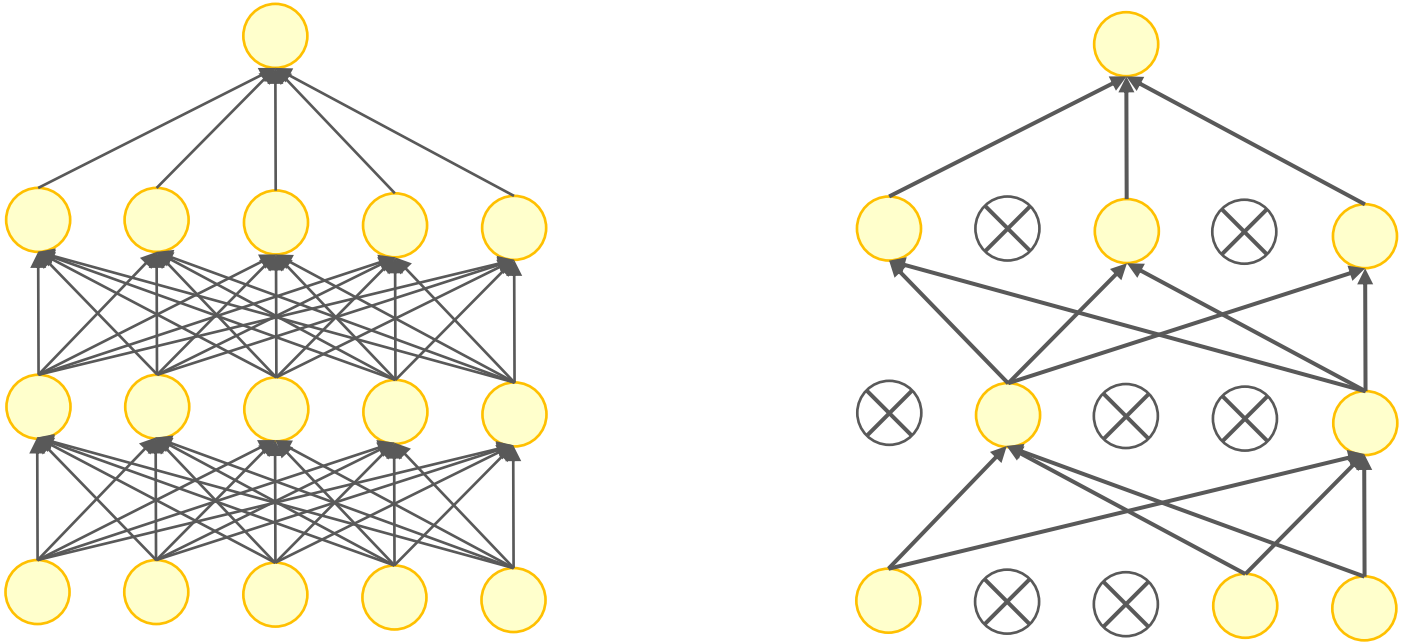
LSTM 계층 다층화

LSTM 계층을 2층으로 쌓은 RNNLM



드롭아웃에 의한 과적합 억제

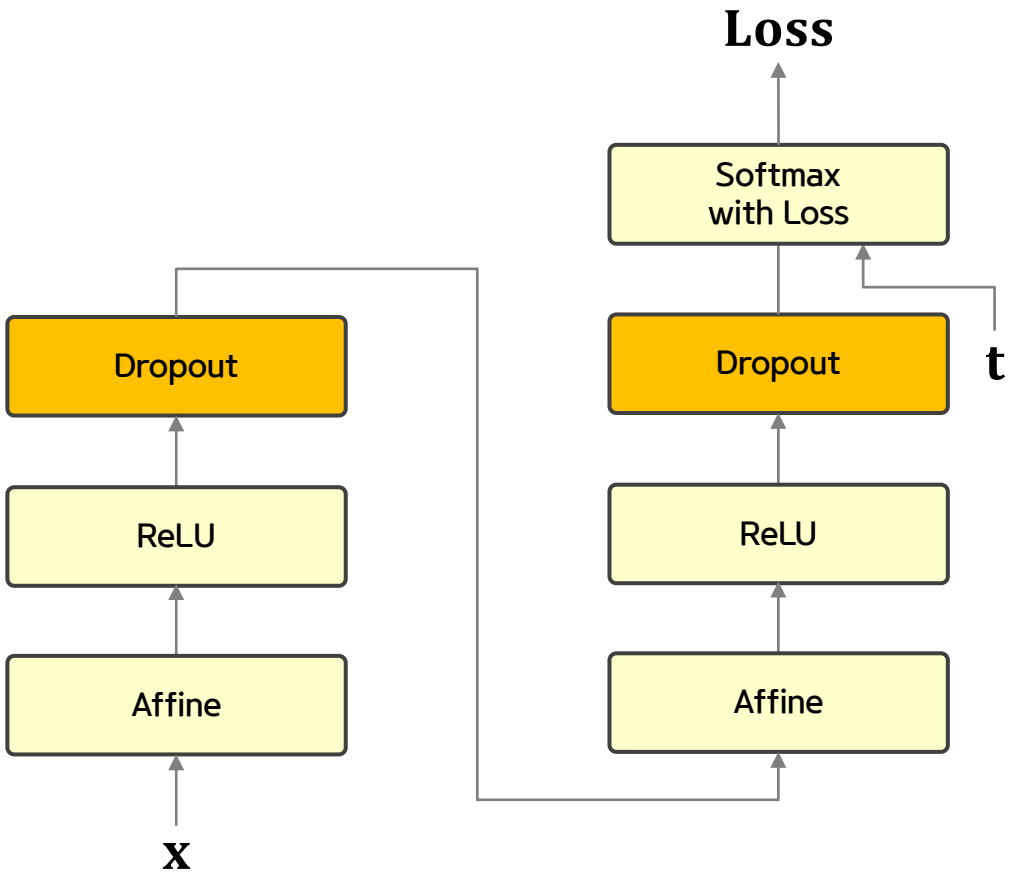
드롭아웃 개념도 : 왼쪽이 일반적인 신경망, 오른쪽이 드롭아웃을 적용한 신경망



드롭아웃은 무작위로 뉴런을 선택하여 선택한 뉴런을 무시한다.
무시한다는 말은 그 앞 계층으로부터의 신호 전달을 막는다는 뜻이다.
이 '무작위한 무시'가 제약이 되어 신경망의 일반화 성능을 개선하는 것이다.

드롭아웃에 의한 과적합 억제

피드포워드 신경망에 드롭아웃 계층을 적용하는 예



이 그림은 드롭아웃 계층을 활성화 함수 뒤에 삽입하는 방법으로 과적합 억제에 기여하는 모습이다.

RNN을 사용한 모델에서 드롭아웃 계층을 LSTM 계층의 시계열 방향으로 삽입하면 좋은 방법이 아니다.

드롭아웃에 의한 과적합 억제

좋은 예: 드롭아웃 계층을 깊이 방향(상하 방향)으로 삽입