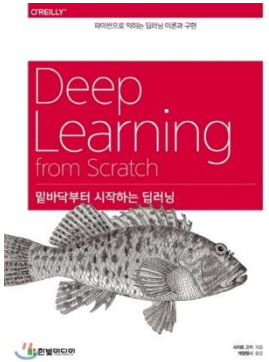
딥러닝 음성 처리 I

목차 1

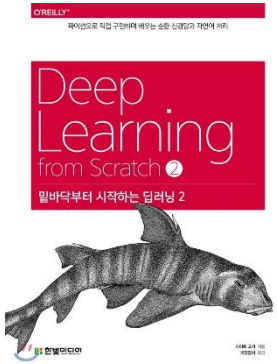
참고 자료

서적:

Do it! 딥러닝 입문
박해선 저 | 이지스퍼블리싱 | 2019년 09월 20일



밑바닥부터 시작하는 딥러닝
사이토 고키 저/개앞맵시 역 | 한빛미디어 | 2017년 01월 02일

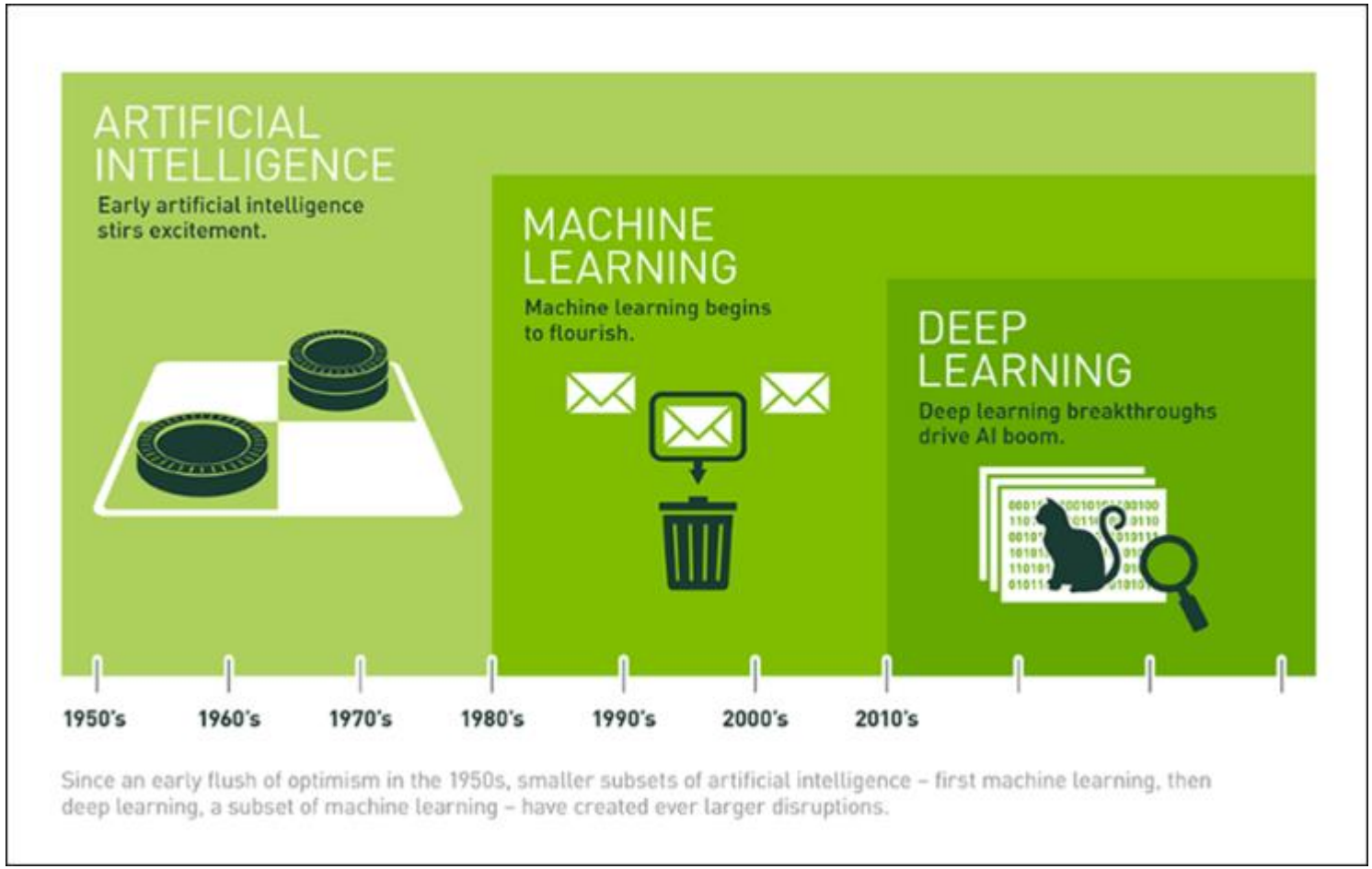


밑바닥부터 시작하는 딥러닝 2
사이토 고키 저/개앞맵시 역 | 한빛미디어 | 2019년 05월 01일

온라인 강의 자료:

모두를 위한 머신러닝/딥러닝 강의 : https://hunkim.github.io/ml/
코세라 머신 러닝 강의 (Andrew Ng) : https://ko.coursera.org/learn/machine-learning

1. 딥러닝이란 무엇인가?


ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
1950's
1960's
1970's
1980's
1990's
2000's
2010's
Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

"보통의 사람이 수행하는 지능적인 작업을 자동화하기 위한 연구 활동 - 본문"

인공지능(人工知能, 영어: artificial intelligence, AI)은 기계로부터 만들어진 지능을 말한다. - wikipedia

명시적인 규칙을 충분하게 만들어 기계의 수준을 높이는 접근 방법을 symbolic AI 라고 하며, 1950년대부터 1980년대까지 AI의 지배적인 패러다임이었습니다.

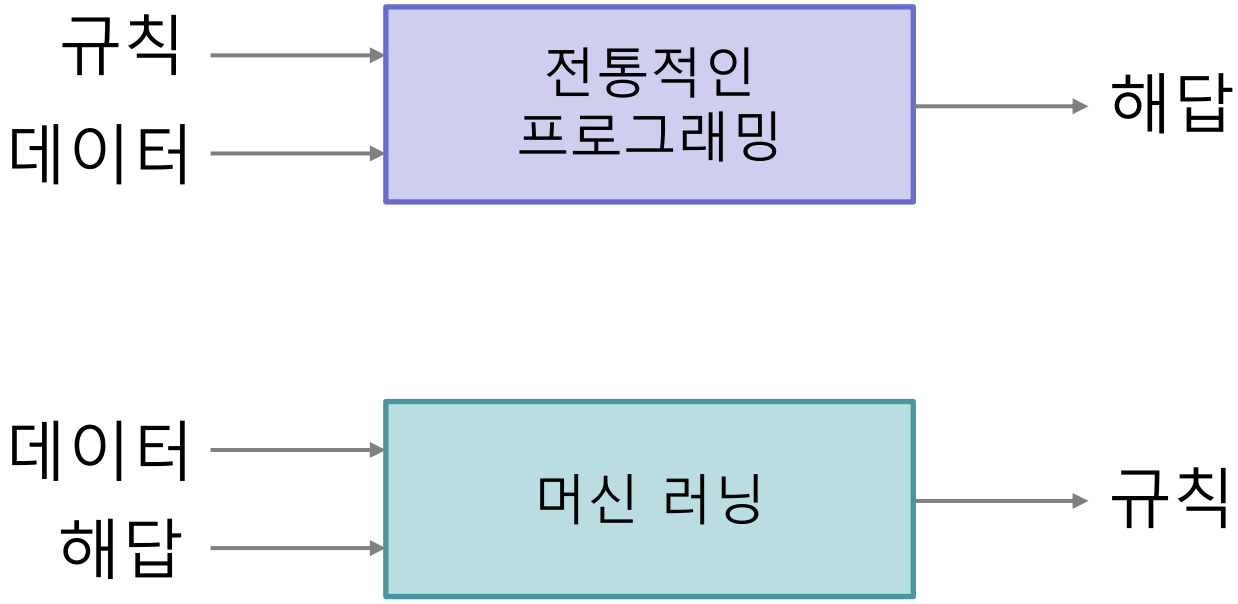
Basic concepts

- What is ML?
- What is learning?
  - supervised
  - unsupervised
- What is regression?
- What is classification?

Machine Learning

- 명시적 프로그램의 한계
  - 스팸 메일 필터 : 많은 규칙
  - 자율 주행 : 더 많은 규칙

- Machine learning :
  "기계가 일일이 코드로 명시하지 않은 동작을 데이터로부터 학습하여 실행할 수 있도록 하는 알고리즘을 개발하는 연구 분야 " Arthur Samuel(1959)


규칙
데이터
전통적인
프로그래밍
해답
데이터
해답
머신 러닝
규칙

Supervised/Unsupervised learning

- Supervised learning:
  - 레이블이 있는 예제로 학습

Supervised/Unsupervised learning

- Unsupervised learning:
  - 특정 입력(Input)에 대하여 올바른 정답(Right Answer)이 없는 데이터 집합이 주어지는 경우의 학습
  - 잘못된 예측에 대해 Feedback을 받고 교정할 수 없음

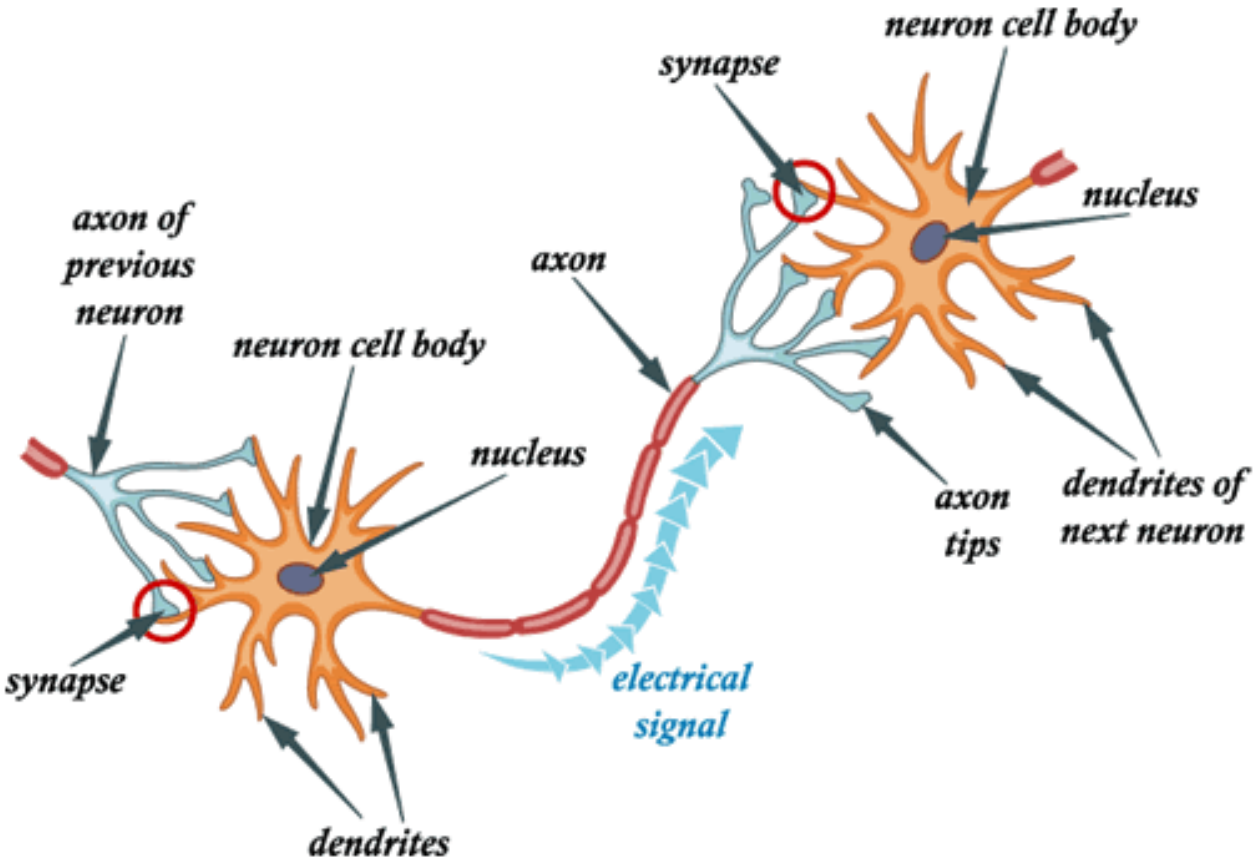
예) Google news grouping
페이스북 에서 특정 집단의 사람들을 그룹화
천체의 별 모양으로 분류

Supervised learning

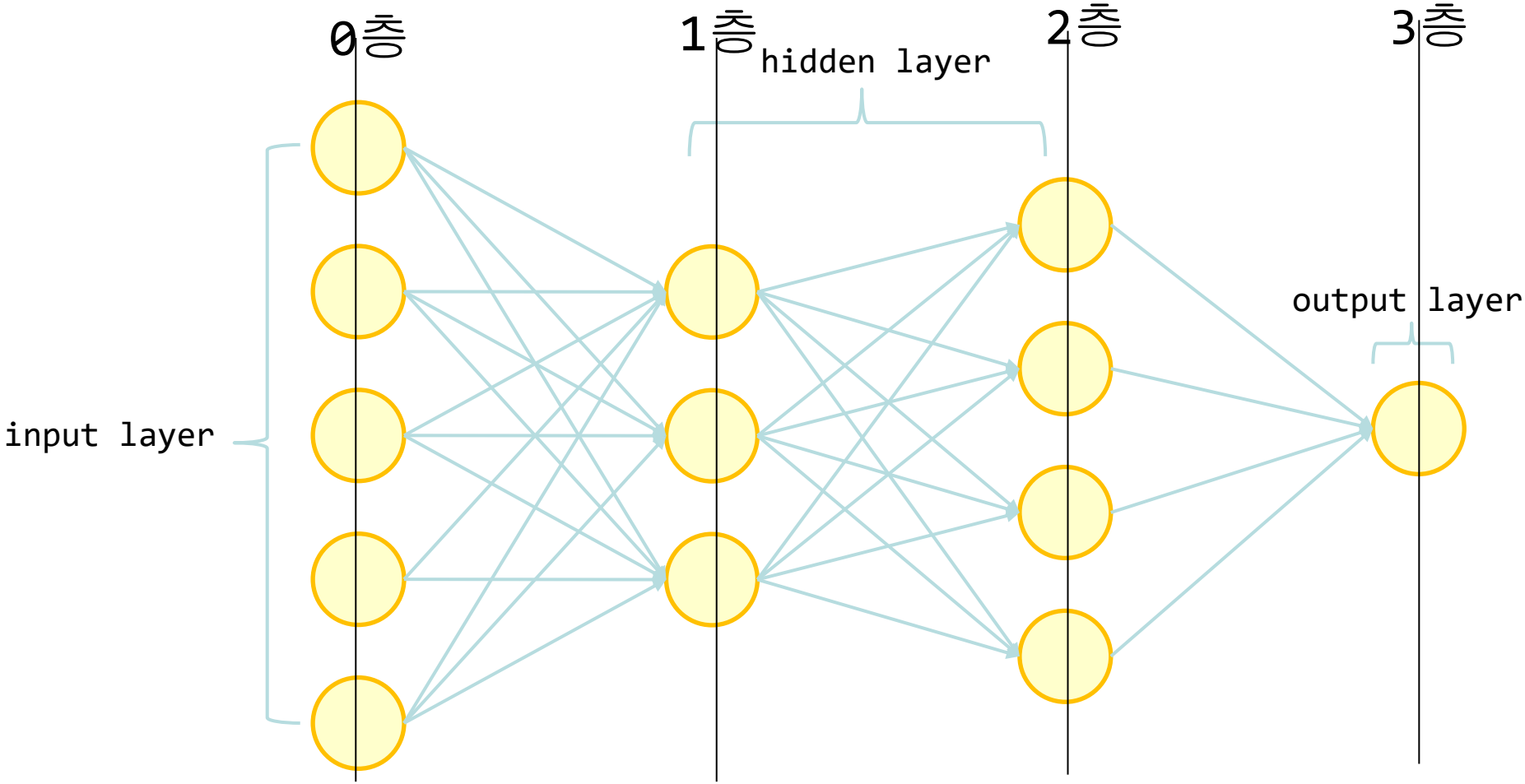
- ML에서 가장 일반적인 문제 유형
  - 시험 점수 예측 : 이전 시험에서 점수와 공부 시간
  - 이메일 스팸 필터 : 라벨이 있는 학습(스팸 또는 햄)
  - 이미지 라벨링 : 태그가 있는 이미지로 부터 학습

Types of supervised learning

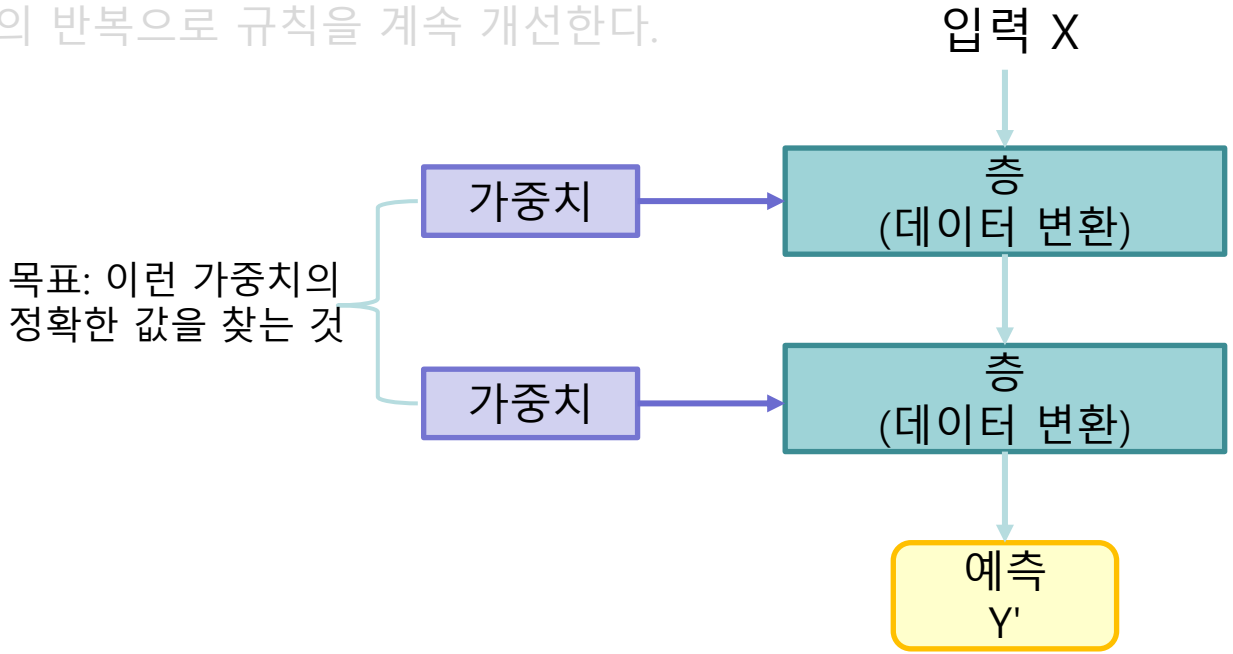
- 시험 점수 예측 : 이전 시험에서 점수와 공부 시간
  - regression
- 이메일 스팸 필터 : 라벨이 있는 학습(스팸 또는 햄)
  - binary classification
- 이미지 라벨링 : 태그가 있는 이미지로 부터 학습
  - multi-label classification


neuron cell body
synapse
nucleus
axon of previous neuron
axon
neuron cell body
nucleus
axon tips
dendrites of next neuron
synapse
electrical signal
dendrites

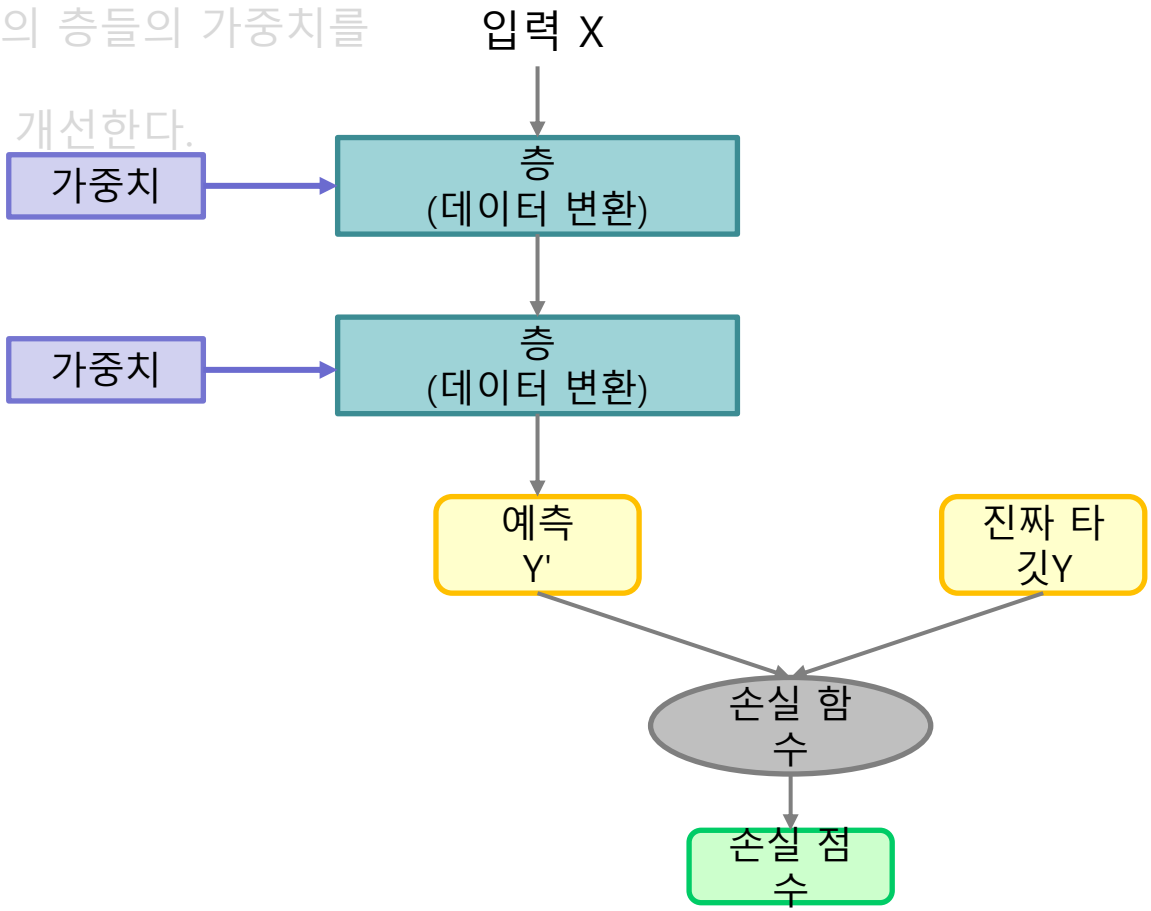
Multi-layer perceptron (MLP)는 Hidden layer라는 layer를 도입해 인풋을 한 차원 높은 단계의 특징, 즉 representation으로 나타낸다.



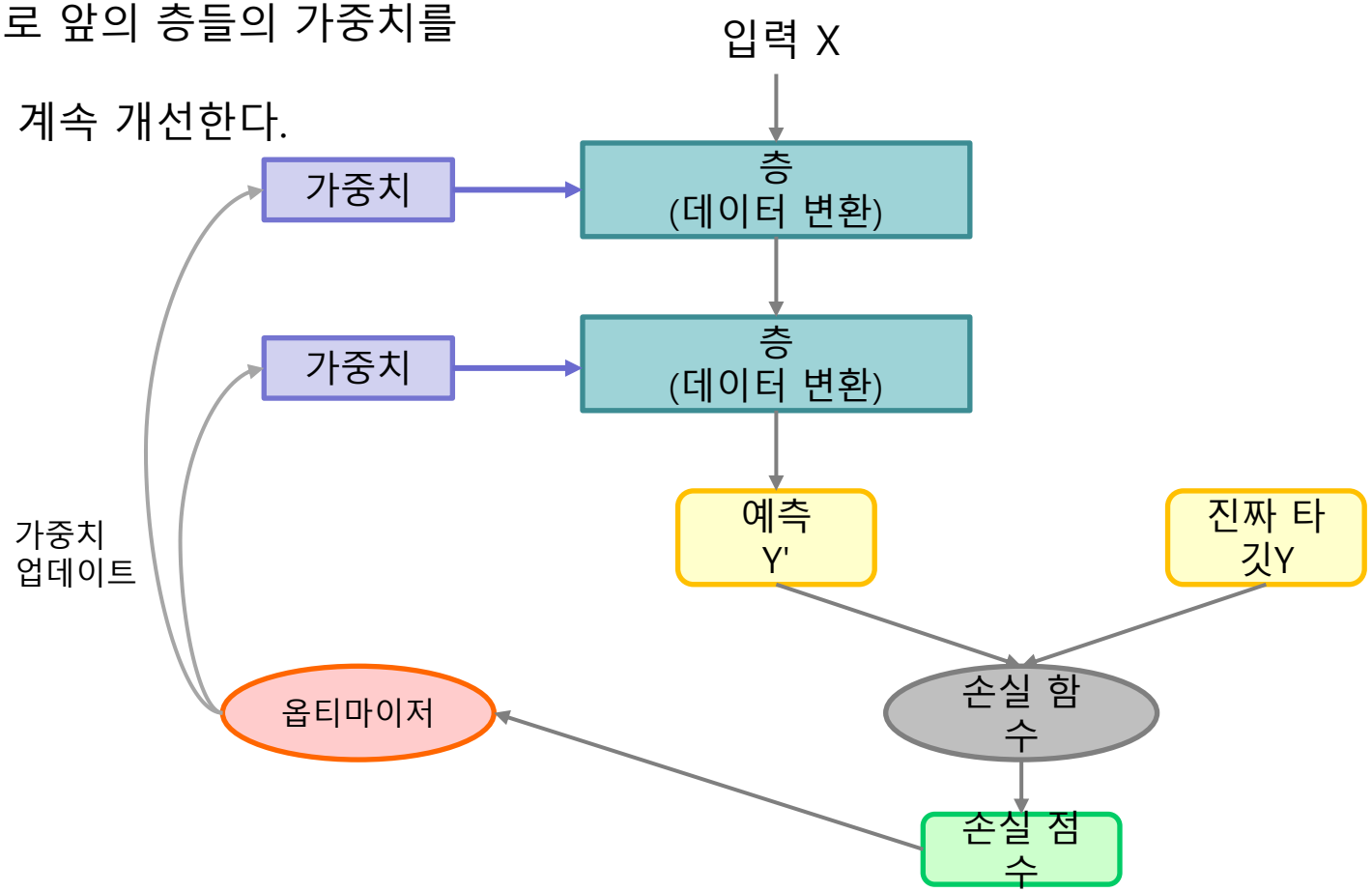
1.데이터를 입력한다.
2.여러 층을 통해 예상 결과값을 만든다. (매핑)
3.실제 값과 비교해서 그 차이를 구한다. (타깃과 손실함수)
4.차이를 줄이기 위한 방법으로 앞의 층들의 가중치를 수정해준다. (역전파)
5.이 방법의 반복으로 규칙을 계속 개선한다.



1.데이터를 입력한다.
2.여러 층을 통해 예상 결과값을 만든다. (매핑)
3.실제 값과 비교해서 그 차이를 구한다. (타깃과 손실함수)
4.차이를 줄이기 위한 방법으로 앞의 층들의 가중치를 수정해준다. (역전파)
5.이 방법의 반복으로 규칙을 계속 개선한다.



1.데이터를 입력한다.
2.여러 층을 통해 예상 결과값을 만든다. (매핑)
3.실제 값과 비교해서 그 차이를 구한다. (타깃과 손실함수)
4.차이를 줄이기 위한 방법으로 앞의 층들의 가중치를 수정해준다. (역전파)
5.이 방법의 반복으로 규칙을 계속 개선한다.



딥러닝의 성과

- 사람 수준의 이미지 분류, 음성 인식, 필기 인식
- 향상된 번역
- 향상된 TTS 변환
- 디지털 비서
- 자율 주행 능력
- 광고 타게팅
- 웹 엔진 결과
- 자연어 질의 대답 능력
- 바둑

- ◆ 지나친 기대는 큰 실망을 가져온다.
- ◆ 실망은 투자 감소로 이어진다.
- ◆ 투자 감소는 AI 겨울로 이어진다.

기술에 대한 거품이 증가하여, 갑작스럽게 지원이 많아지다 단기간내 성과가 없으면 훅 모두 투자를 안 하는 상황이 올 수 있다는 것이다.

이미 2번의 AI 겨울을 겪었고, 현재 3번째 겨울이 진행이 되고 있을지도 모른다는 점이다.

단기간의 기대는 비현실적이지만 장기적인 전망은 매우 밝다.

1. 딥러닝이란 무엇인가?

1.1 인공지능과 머신러닝, 딥러닝

1.2 딥러닝 이전 : 머신 러닝의 간략한 역사

1.3 왜 딥러닝일까? 왜 지금일까?

◆ **확률적 모델링(probability modeling)**
- 통계학 이론을 데이터 분석에 응용한 것
- 초창기 머신 러닝 형태 중 하나이고, 현재에도 많이 사용됨
- 가장 잘 알려진 알고리즘은 나이브 베이즈(Naive Bayes) 알고리즘

◆ **로지스틱 회귀(logistic regression)**
- 현대 머신 러닝의 "hello world"
- 이름은 회귀인데 회귀(regression) 알고리즘이 아닌 분류(classification) 알고리즘임
- 데이터 과학자가 분류 작업에 대한 초기 감을 위해 첫 번째로 선택되는 알고리즘임

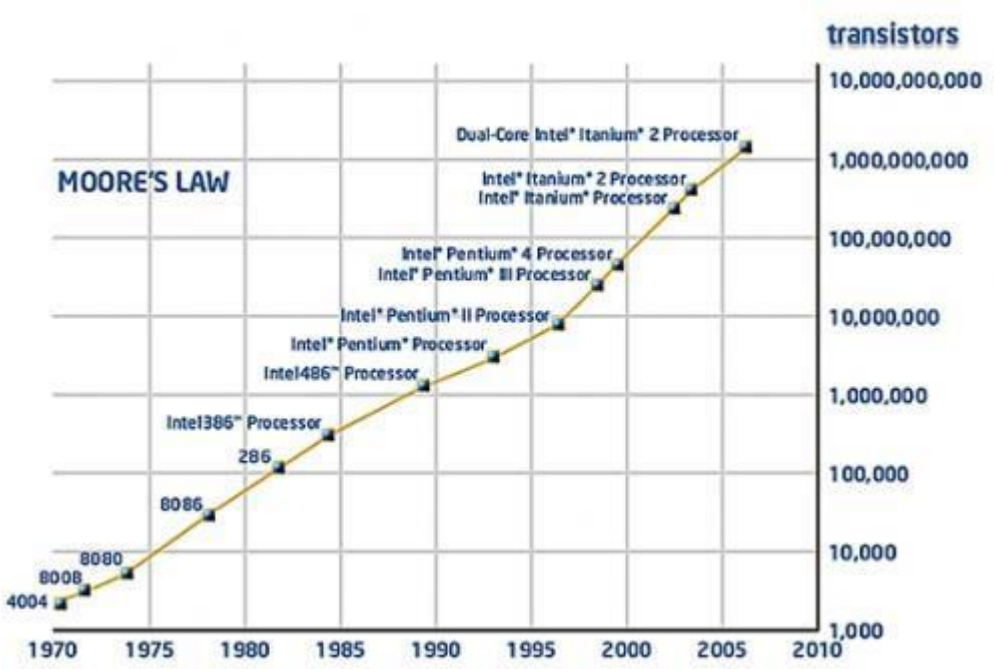
나이브 베이즈 알고리즘이란 입력 데이터의 특성이 모두 독립적이라고 가정하고 베이즈 정리(Bayes' theorem)를 적용하는 머신 러닝 분류 알고리즘이다.

◆ **1950년대**
- 신경망의 핵심 아이디어 등장
- 본격적으로 시작되지 못함

◆ **1980년대**
- 역전파 알고리즘 재발견
- 신경망에 역전파 알고리즘 적용 시작

◆ **1990년대**
- 초창기 합성곱 신경망과 역전파을 연결
- 미국 우편 서비스에 이용

2010년경부터 일부 사람들이 중요한 성과를 내며 신경망은 다시 주목을 받았다.



인텔의 고든 무어가 1965년에 주장한 법칙.
"반도체의 집적회로 성능은 18개월마다 2배로 증가한다"는 법칙이다.
이는 컴퓨팅 환경의 발전이 신경망의 이론을 뒷받침 하면서 이론으로만 존재 했던 여러 가지를 가능하게 만들었다.

딥러닝은 머신 러닝에서 가장 중요한 단계인 특성 공학을 자동화 한다는 점에서 매우 큰 장점을 가지고 있다.

특성공학(feature engineering) 이란 초기 학습을 위한 데이터의 변환을 의미한다.

딥러닝에서 데이터를 학습하는 방법에는 두가지 중요한 특징이 있다.

- **층을 거치며, 점진적으로 복잡한 표현이 만들어짐**
- **점진적인 중간 표현이 공동으로 학습**

1. 딥러닝이란 무엇인가?

1.1 인공지능과 머신러닝, 딥러닝

1.2 딥러닝 이전 : 머신 러닝의 간략한 역사

1.3 왜 딥러닝일까? 왜 지금일까?

CPU는 1990년부터 2010년 사이에 약 5000배 정도 빨라졌다.

2000년대 게임 그래픽 성능 개발을 위한 대용량 고속 병렬 칩(그래픽 처리장치 GPU)가 발전하였다.

GPU 제품을 위한 프로그래밍 인터페이서 CUDA를 출시하였다.

물리 모델링을 시작으로 신경망까지 병렬화가 가능해 졌다.

GPU인 NVIDIA TITAN X는 6.6 테라플롭의 단정도 연산 성능을 제공한다.

구글은 2016년에 텐서 처리 장치 프로젝트를 공개했다. 이 칩은 심층 신경망을 실행하기 위해 완전히 새롭게 설계한 것으로 최고 성능을 가진 GPU보다 10배 이상 빠르고 에너지 소비도 더 효율적이다.

2017에 발표한 TPU는 180 테라플롭이다.

'데이터의 바다'라는 용어가 있듯이 현재는 데이터가 매우 많다.

저장 장치의 발전, 데이터 셋을 수집하고 배포할 수 있는 인터넷의 성장은 머신 러닝에 필요한 데이터들을 마련할 수 있는 환경을 만들어주었다.

플리커에서 사용자가 붙인 이미지 태그

유튜브의 비디오

위키피디아는 자연어 처리 분야에 필요한 핵심 데이터셋

1400만개 이미지를 1000개의 범주로 구분해 놓은 ImageNet 데이터셋

신경망의 층에 더 잘 맞는 활성화 함수(activation function)

층별 사전 훈련(pretraining)을 불필요하게 만든 가중치 초기화

RMSProp과 Adam 같은 더 좋은 최적화 방법 개발

배치 정규화

잔차 연결

깊이별 분리 합성곱

같은 고급 기술들이 개발 됨

초창기에 딥러닝을 하려면 흔치 않은 C++와 CUDA의 전문가가 되어야 했음

씨아노와 텐서 플로가 개발되어 JAVA나 Python으로 쉽게 개발할 수 있게 됨

케라스의 도구로 레고 블럭을 만들 듯 쉽게 새로운 모델을 개발할 수 있게 됨

케라스 등장 스타트업과 활용함

단순함 : 딥러닝은 특성 공학이 필요하지 않아 복잡하고 불안정한 많은 엔지니어링 과정을 엔드-투-엔드로 훈련시킬 수 있는 모델로 바꾸어 준다.

확장성 : 딥러닝은 GPU 또는 TPU 에서 쉽게 병렬화할 수 있기 때문에 무어의 법칙 혜택을 크게 볼 수 이다. 또한 딥러닝 모델은 작은 배치 데이터에서 반복적으로 훈련되기 때문에 어떤 크기의 데이터셋에서도 훈련될 수 있다.

다용도와 재사용성 : 이전에 많은 머신 러닝 방법과는 다르게 딥러닝 모델은 처음부터 다시 시작하지 않고 추가되는 데이터로도 훈련할 수 있다.

2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 역전파 구현

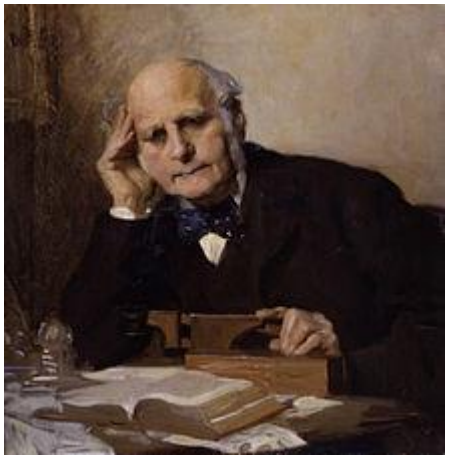
2.4 선형회귀 구현

2.5 시그모이드 함수

2.6 로지스틱 회귀 구현

Regression - 회귀

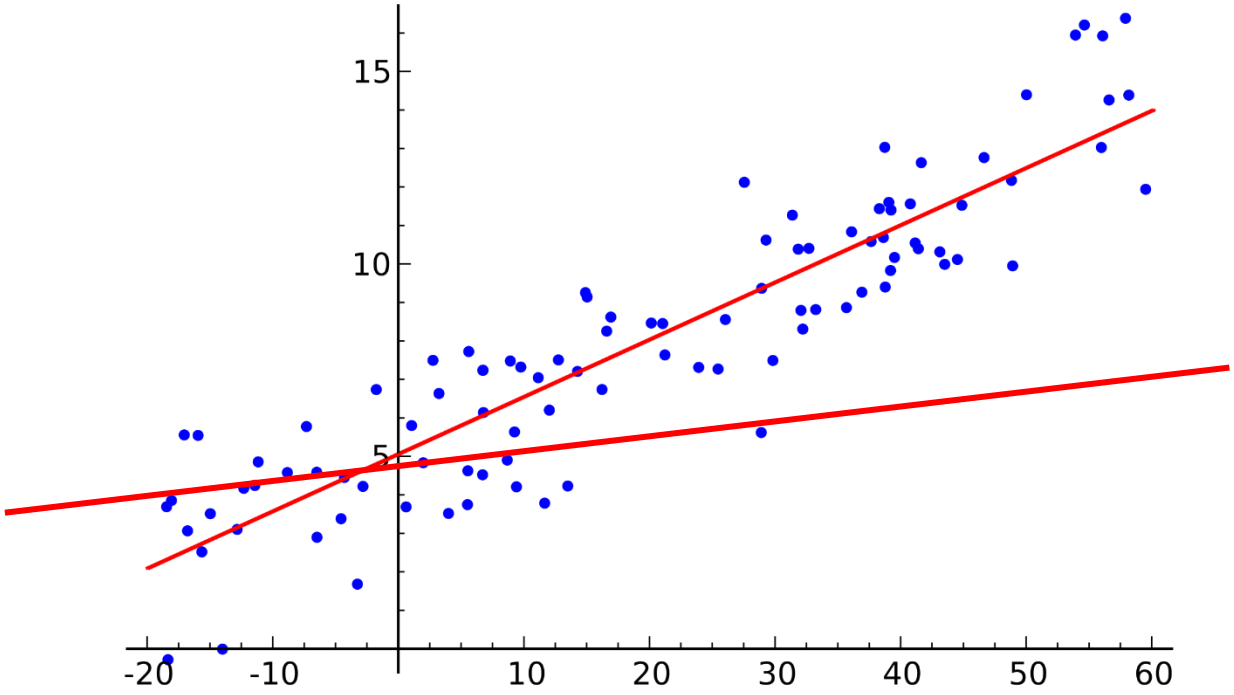
"Regression toward the mean"



Sir Francis Galton
(1822 ~ 1911)

Linear Regression - 선형 회귀

$$\frac{dy}{dx} = a$$

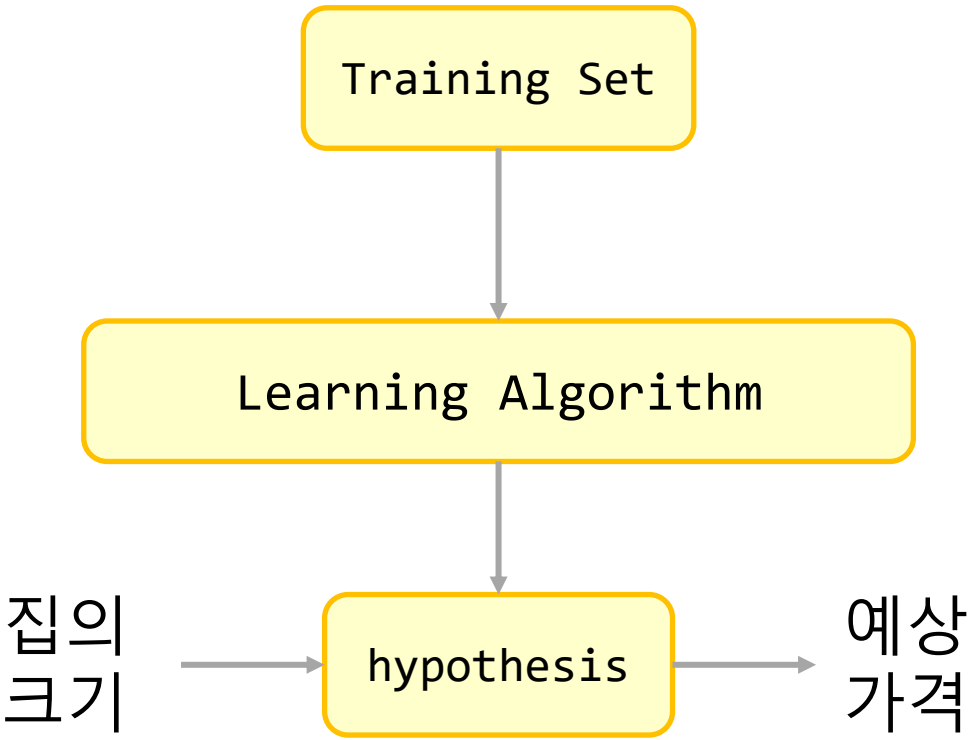


$$y = ax + b$$

$$\downarrow$$

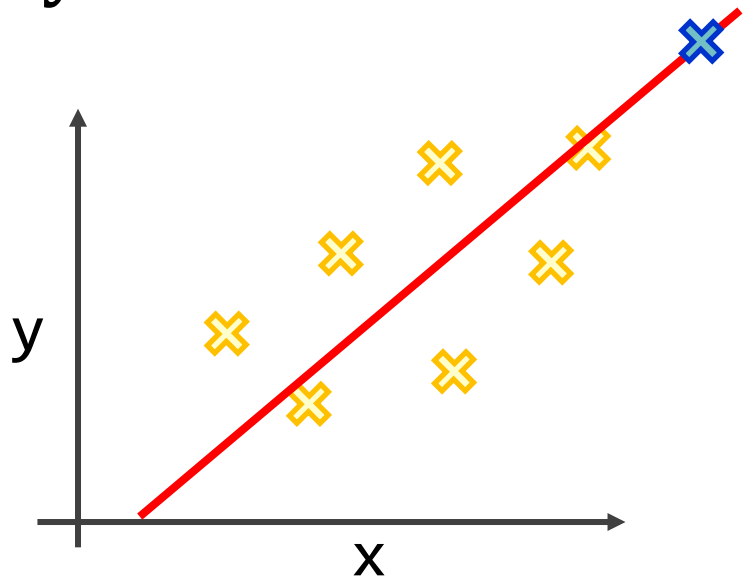
$$y = wx + b$$

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

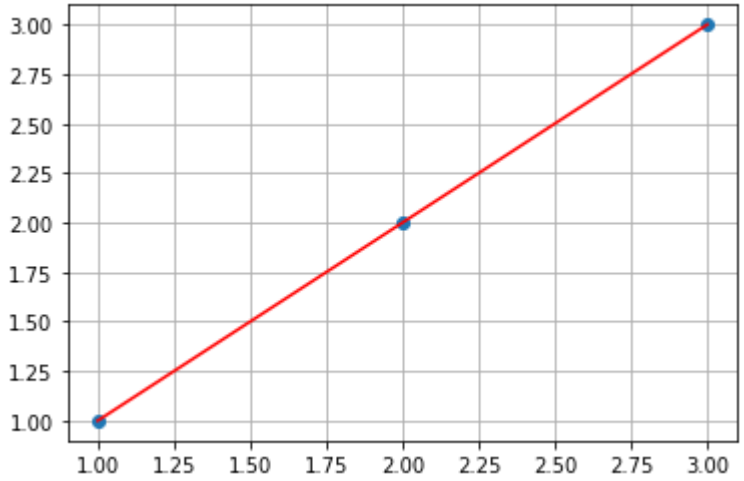


hypothesis ?

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

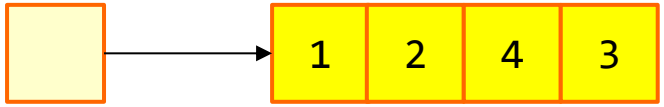


```
import numpy as np                  # numerical computing
import matplotlib.pyplot as plt  # plotting core

x = np.array([1,2,3])
y = np.array([1,2,3])

plt.plot(x,y, 'o' )
plt.plot(x,y, 'r-' )

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



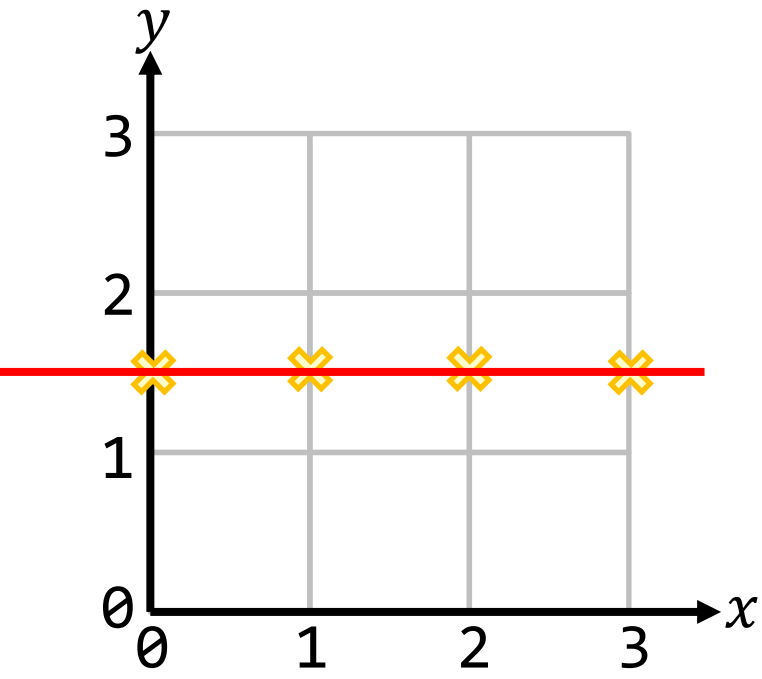
```
a = [1,2,3]
```


```
a = np.array([1,2,3])
```

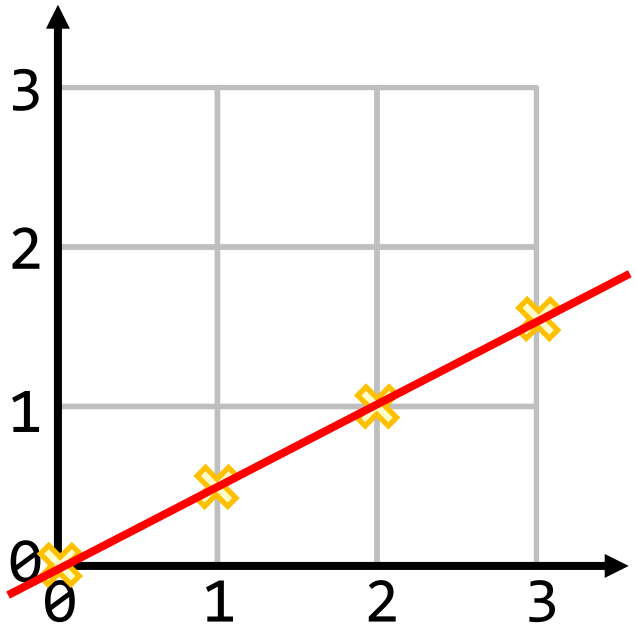


$$\hat{y} = wx + b$$

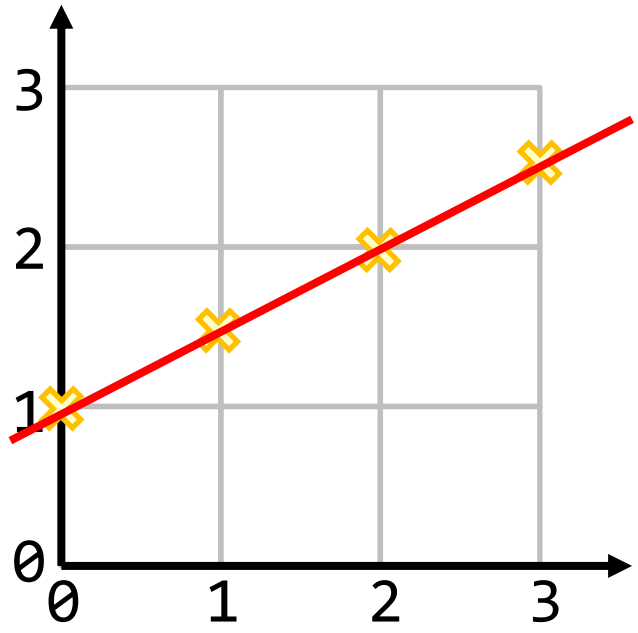
$$\frac{dy}{dx} = \frac{1}{2} = \text{기울기}$$



w = 0
b = 1.5



w = 0.5
b = 0



w = 0.5
b = 1

Hypothesis :

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

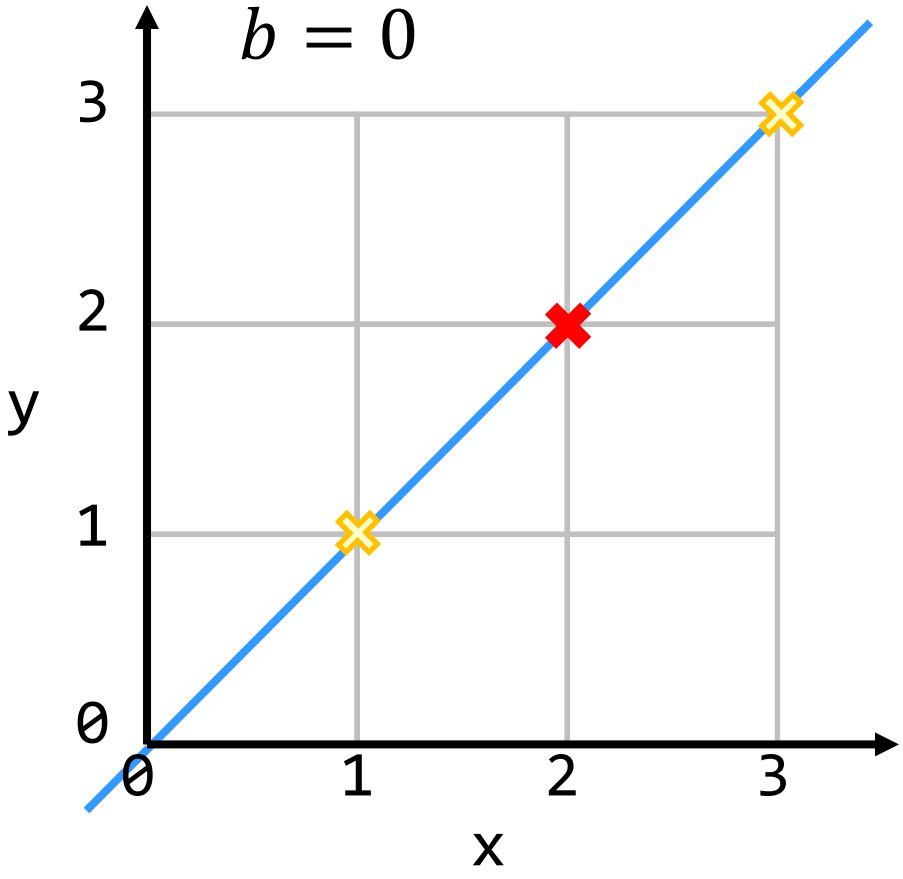
Parameters:

$$w, b$$

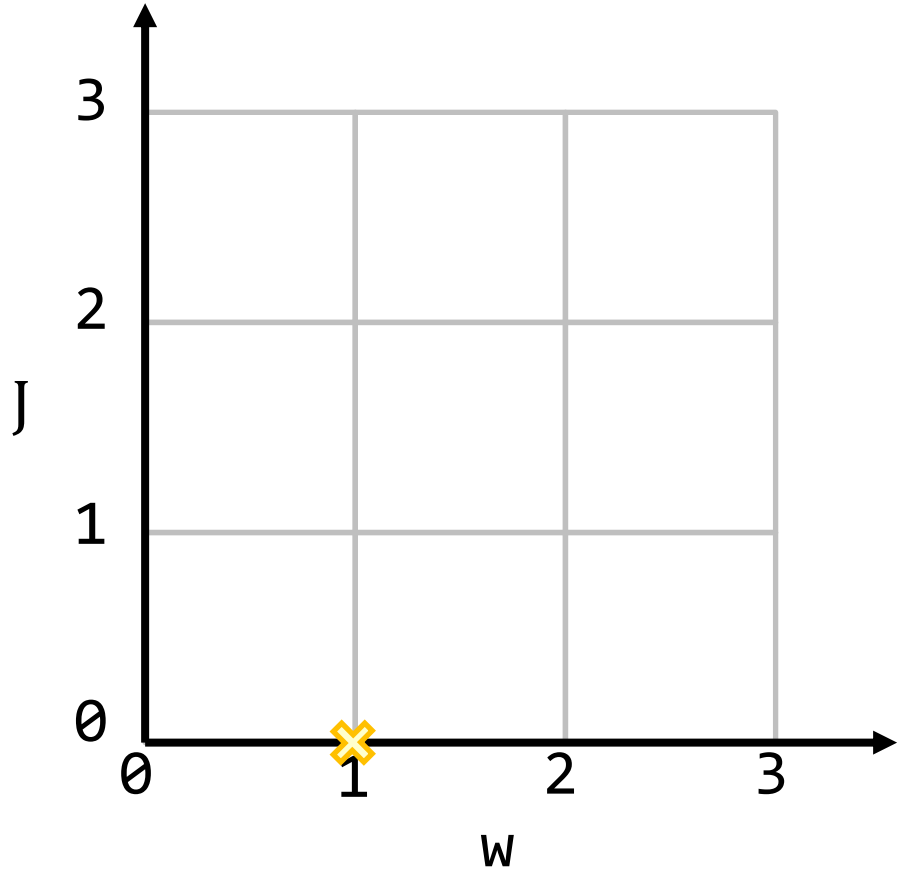
Cost(Loss) Function

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

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



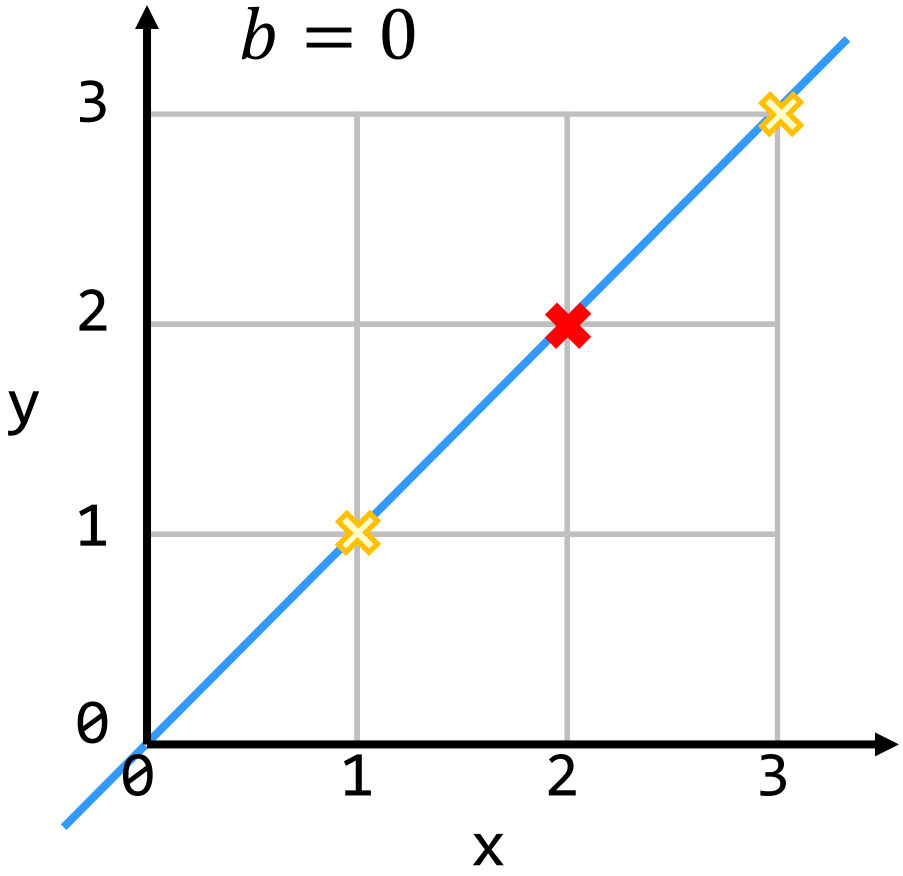
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



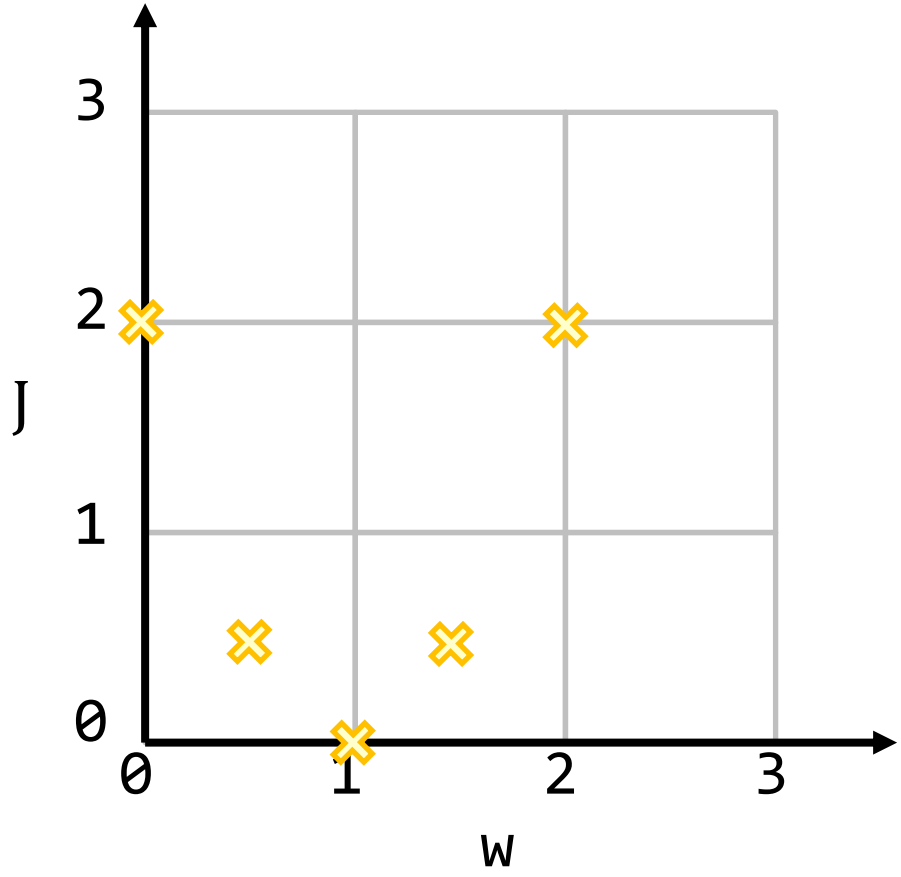
$$\hat{y} = wx + b$$

$$w = 1$$

$$b = 0$$



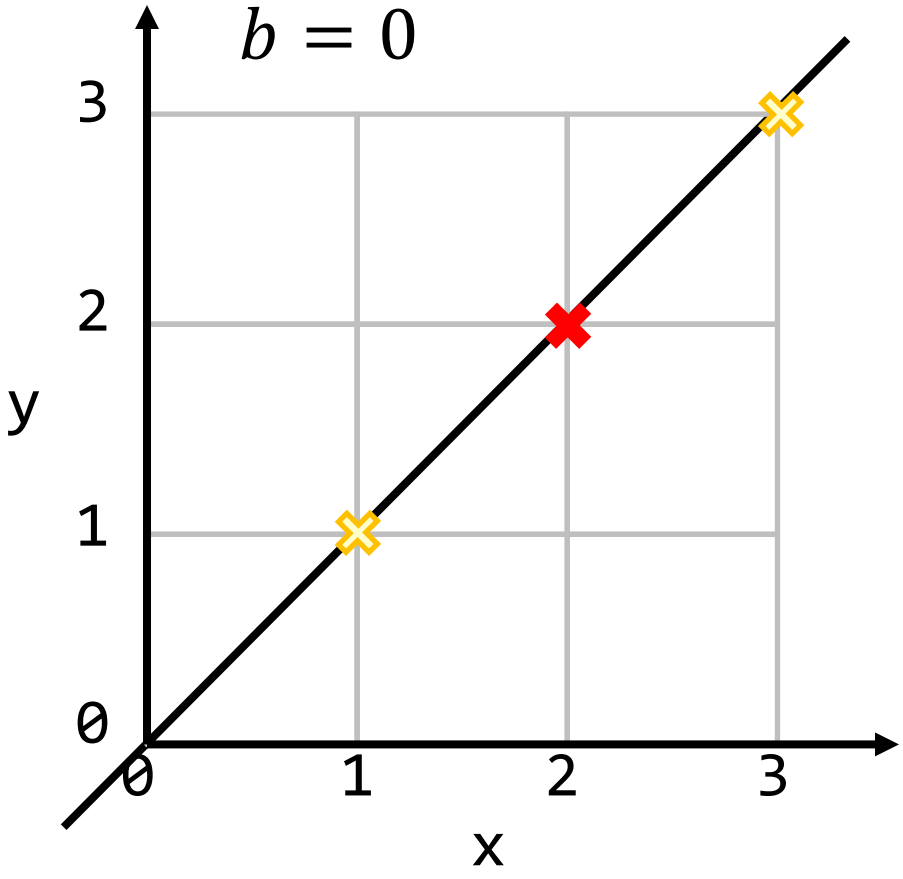
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



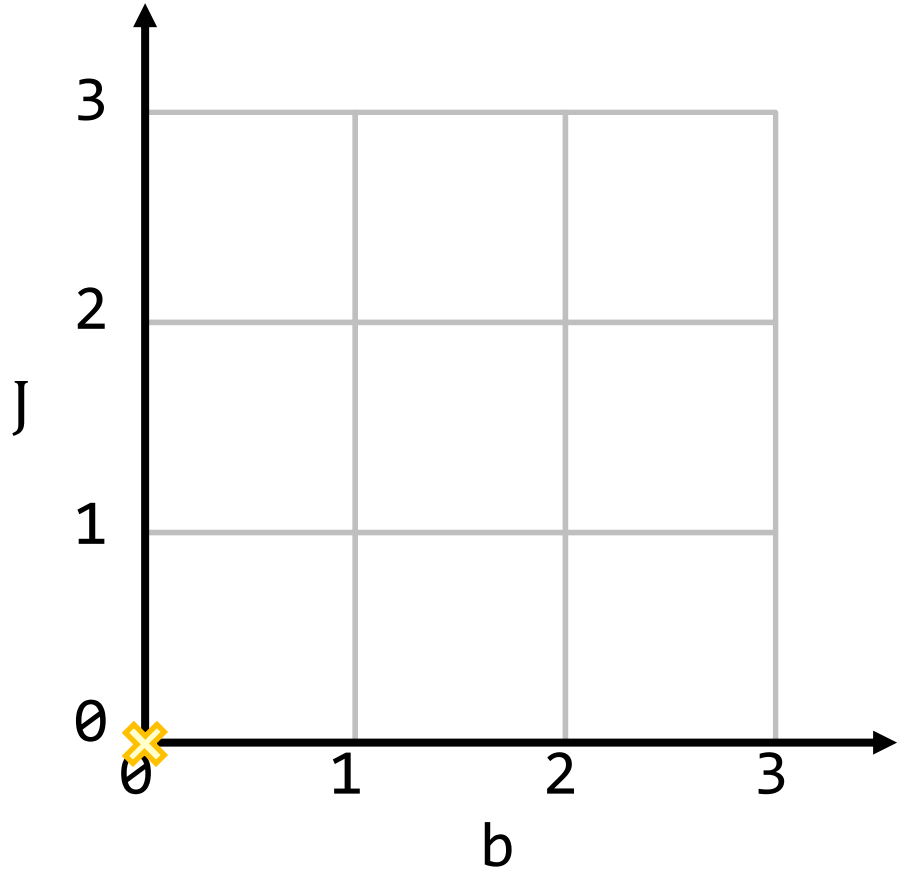
$$\hat{y} = wx + b$$

$$w = 1$$

$$b = 0$$



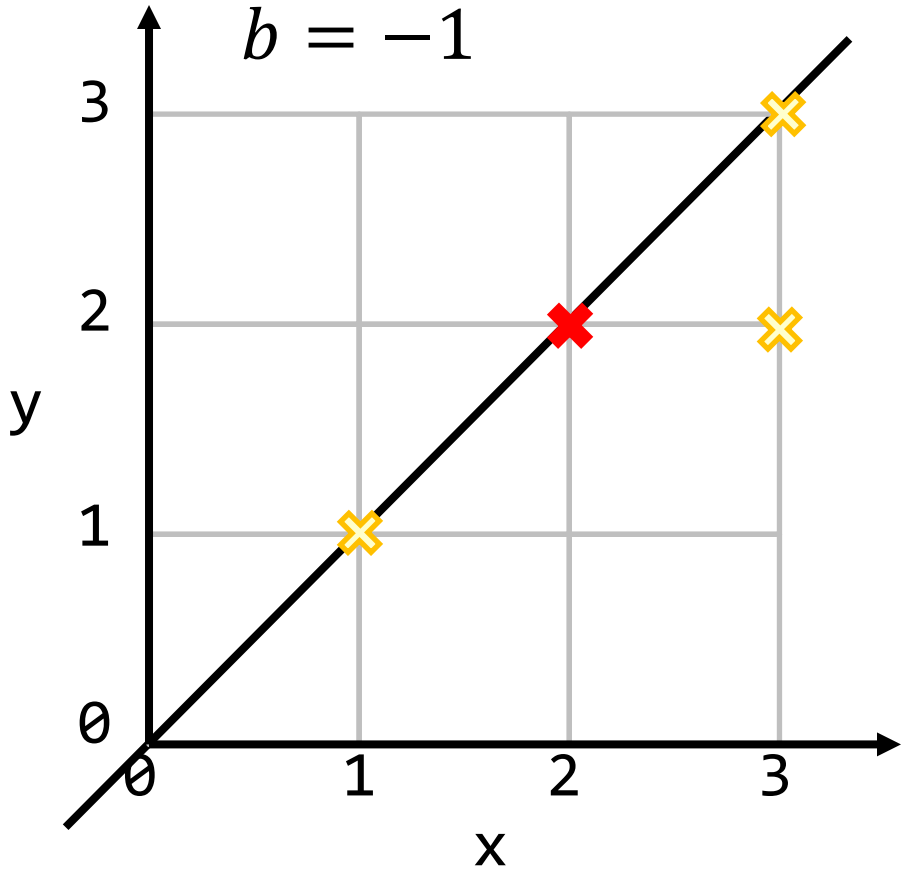
$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



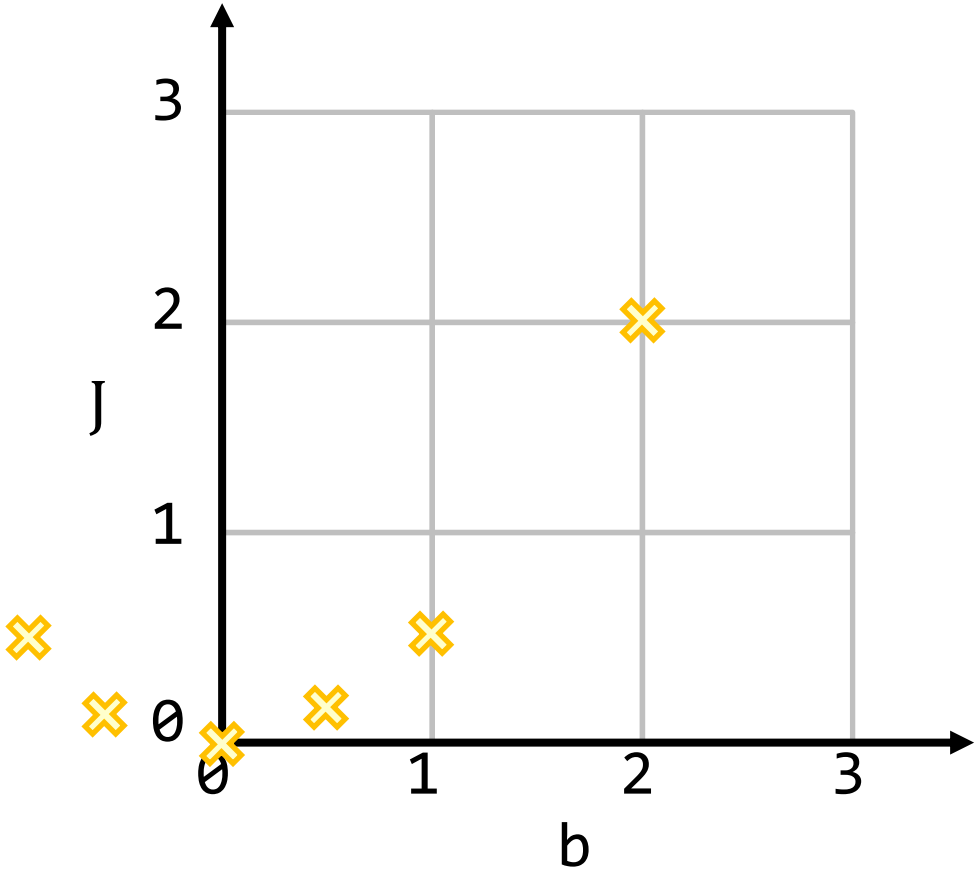
$$\hat{y} = wx + b$$

$$w = 1$$

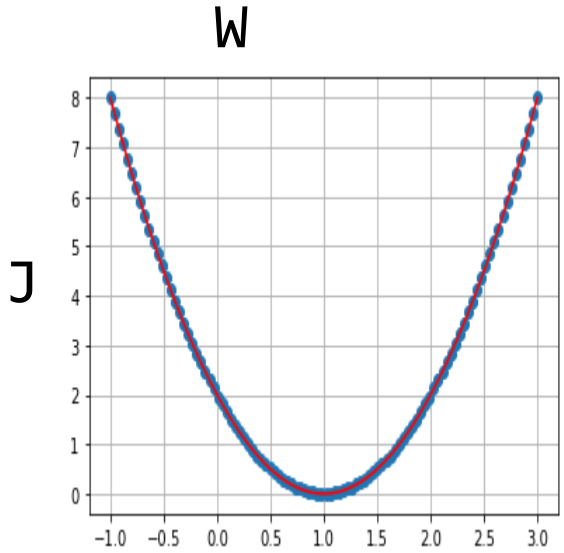
$$b = -1$$

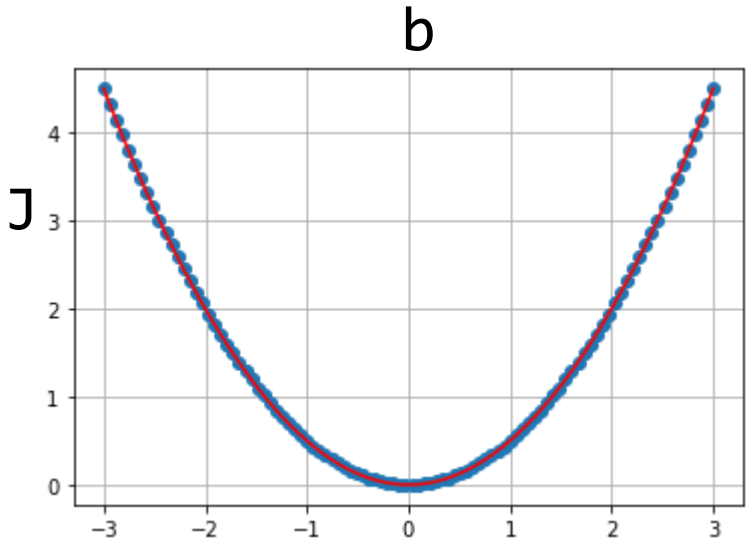


$$J(w, b) = \frac{1}{2}(\hat{y} - y)^2$$



```
y_hat = W*x+b
```





2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 역전파 구현

2.4 선형회귀 구현

2.5 시그모이드 함수

2.6 로지스틱 회귀 구현

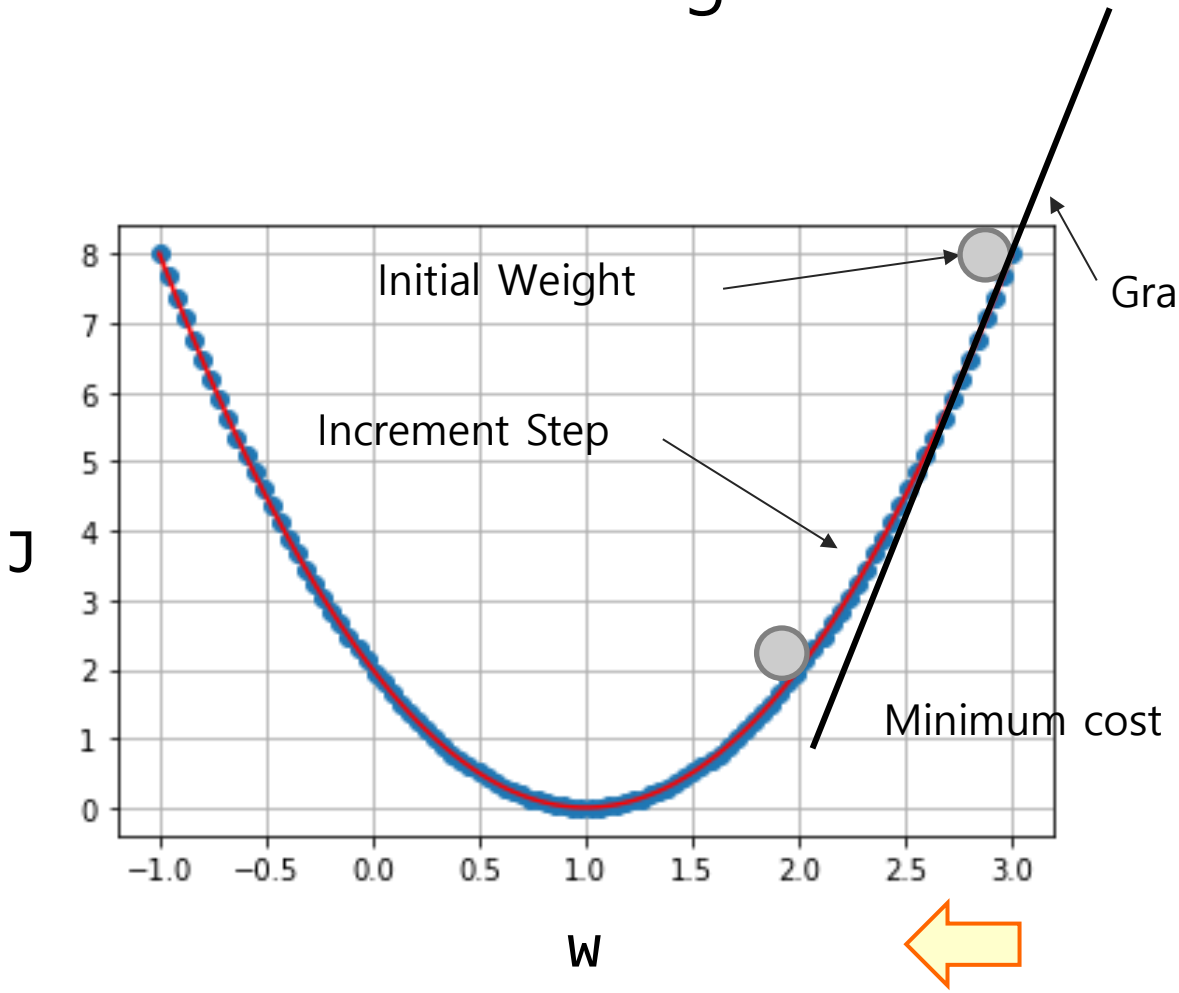
Gradient descent algorithm

- 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




```
x=2, y=2
w=3
b=0
```

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

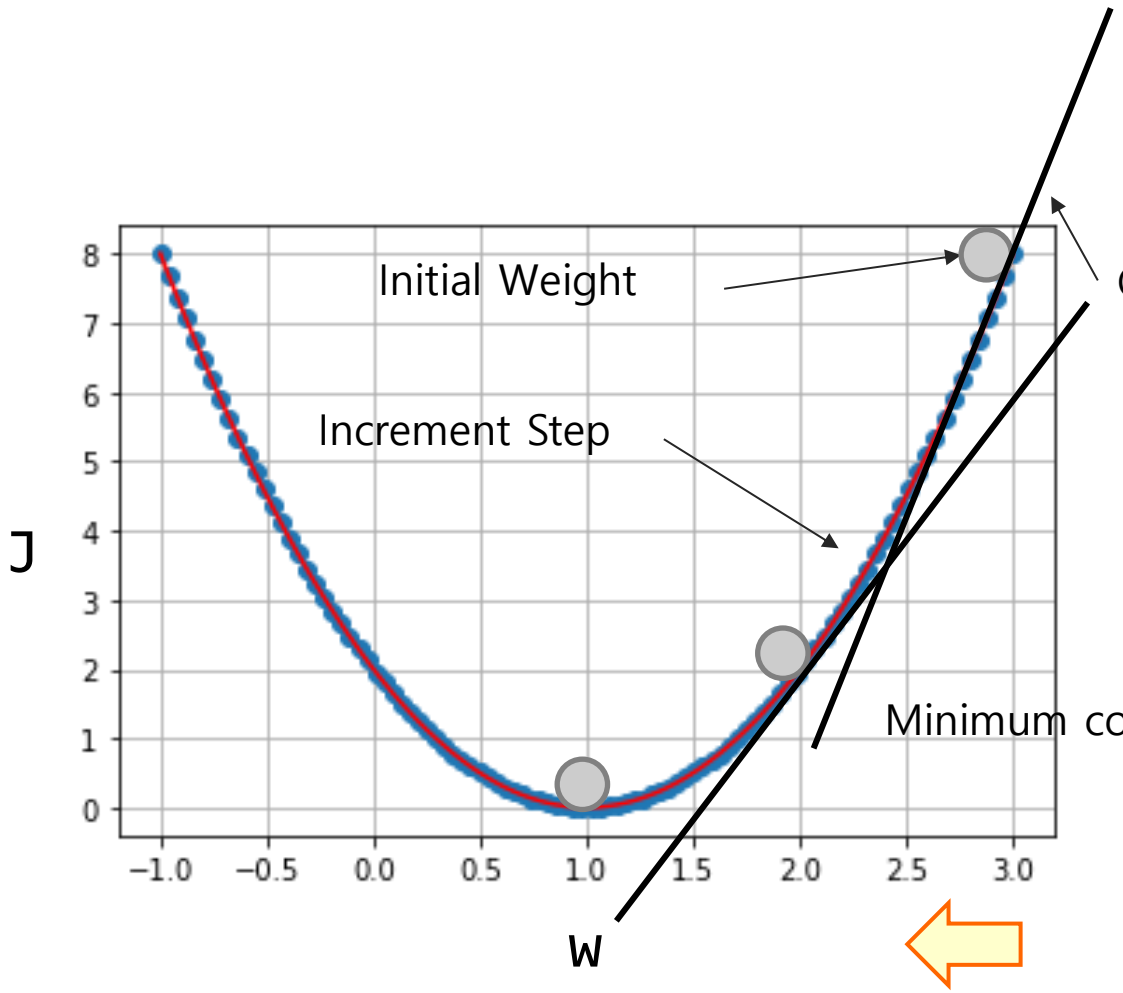
$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

$$w = w - \alpha * dw \qquad \begin{matrix} 0.01 \\ (\hat{y} - y)x \end{matrix}$$

$$w = 3 - 0.08$$

$$w = 2.92$$

Gradient descent algorithm



x=2, y=2
w=2
b=0

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

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

0.01

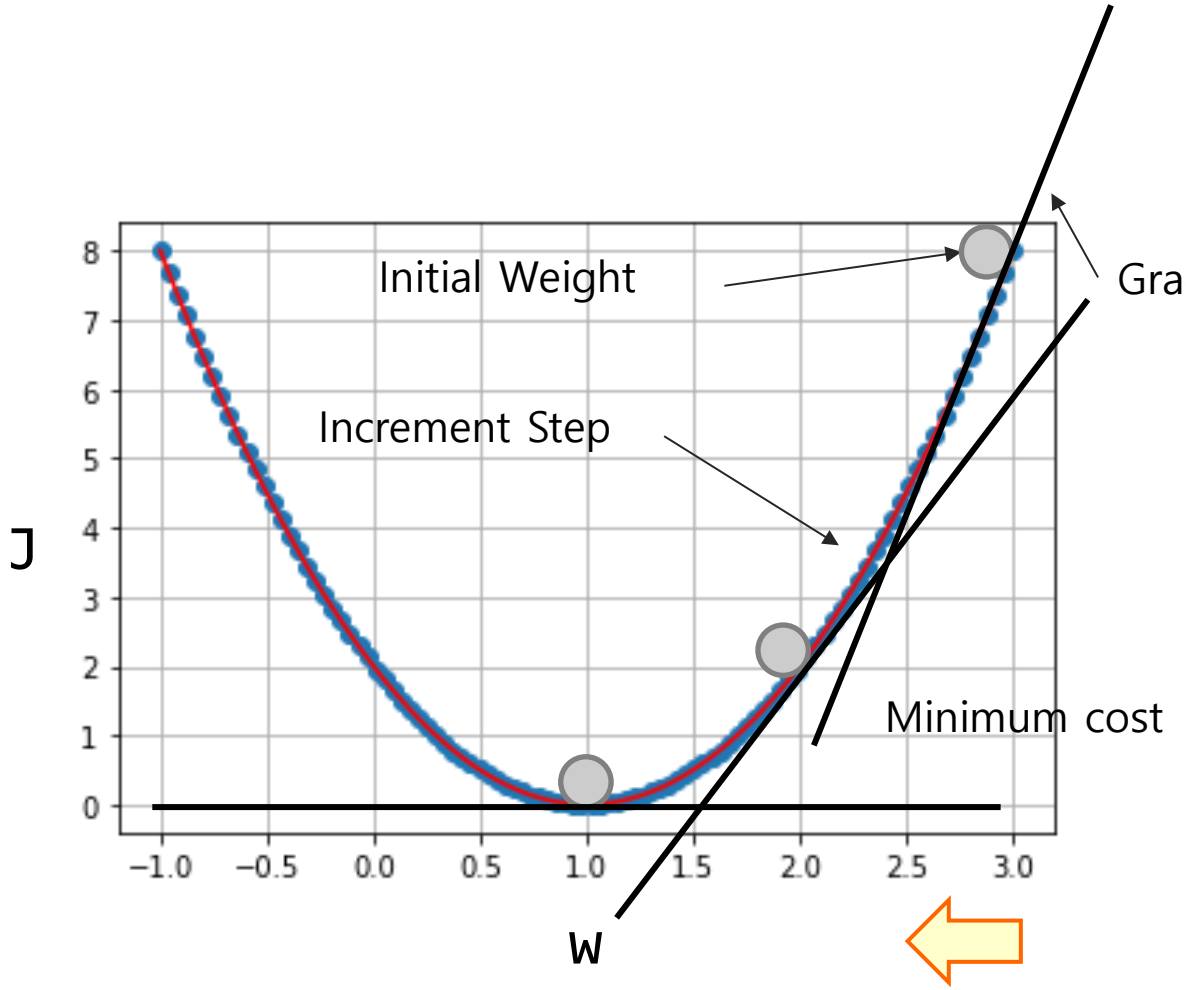
$$w = w - \alpha * dw$$

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

$$w = 2 - 0.04$$

$$w = 1.94$$

Gradient descent algorithm



x=2, y=2
w=1
b=0

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

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

$$0.01$$

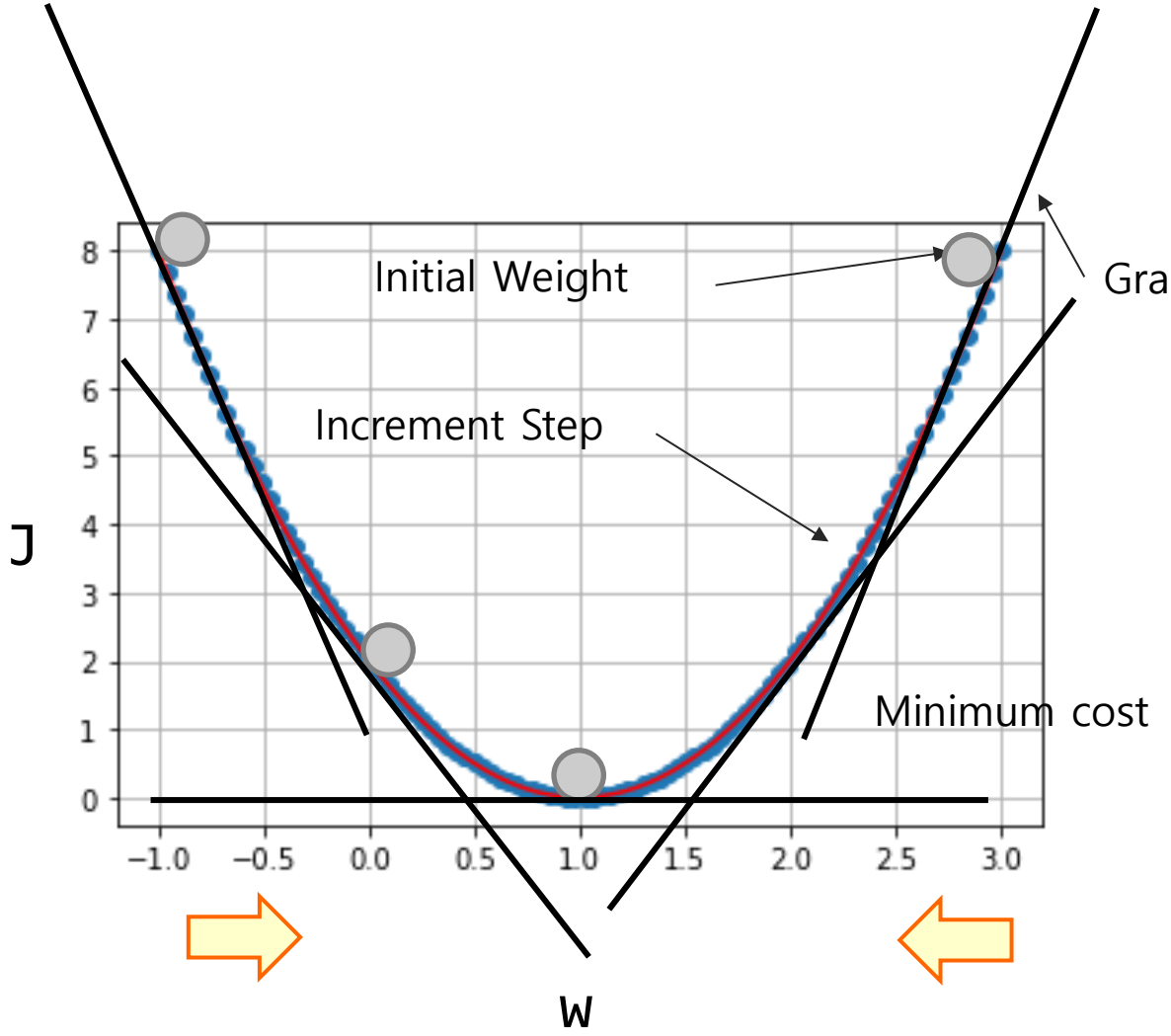
$$w = w - \alpha * dw$$

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

$$w = 1 - 0.00$$

$$w = 1$$

Gradient descent algorithm




```
x=2, y=2
w=-1
b=0
```

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

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

$$0.01$$

$$w = w - \alpha * dw$$

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

$$w = 1 + 0.00$$

$$w = 0.00$$

수렴까지 반복

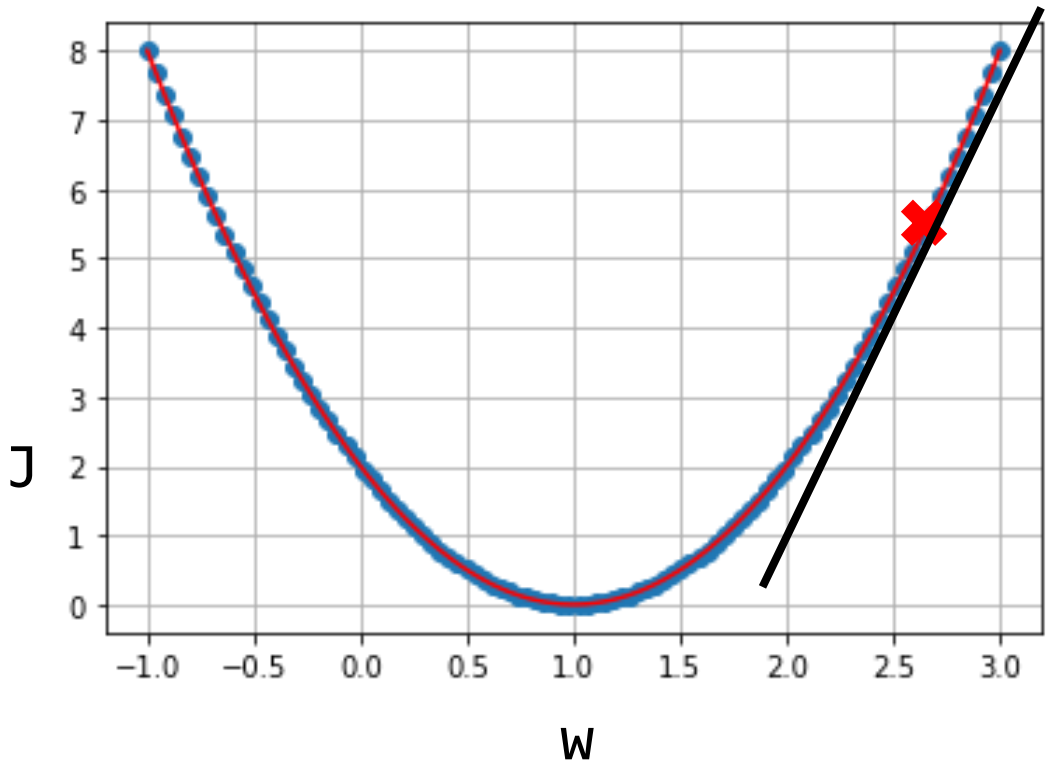
{

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

}

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.

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\partial}{\partial w} wx + b$$

$$= x$$

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

Chain Rule 사용

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

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

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

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

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

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

$$= 1$$

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

Chain Rule 사용

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

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

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

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

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

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

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

Chain Rule 사용

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

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

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

$$\frac{1}{2}(\hat{y} - y)^2 \qquad \frac{1}{2}(4-2)^2 = 2$$

$$\frac{1}{2}(y - \hat{y})^2 \qquad \frac{1}{2}(2-4)^2 = 2$$

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

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

$$= \frac{1}{2}(-2y + 2\hat{y})$$

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

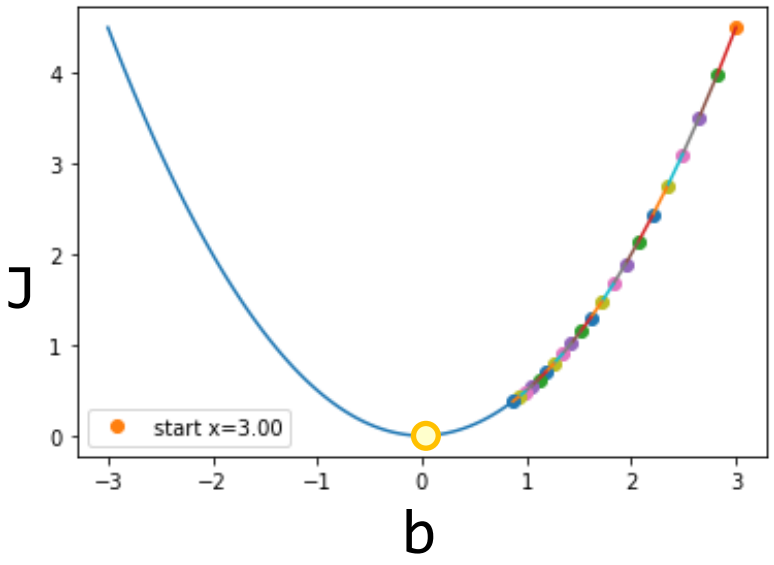
Chain Rule 사용

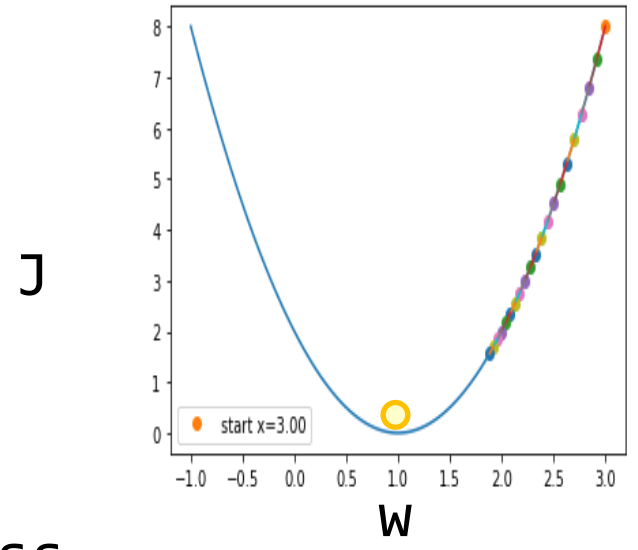
$$\hat{y} = wx + b$$

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

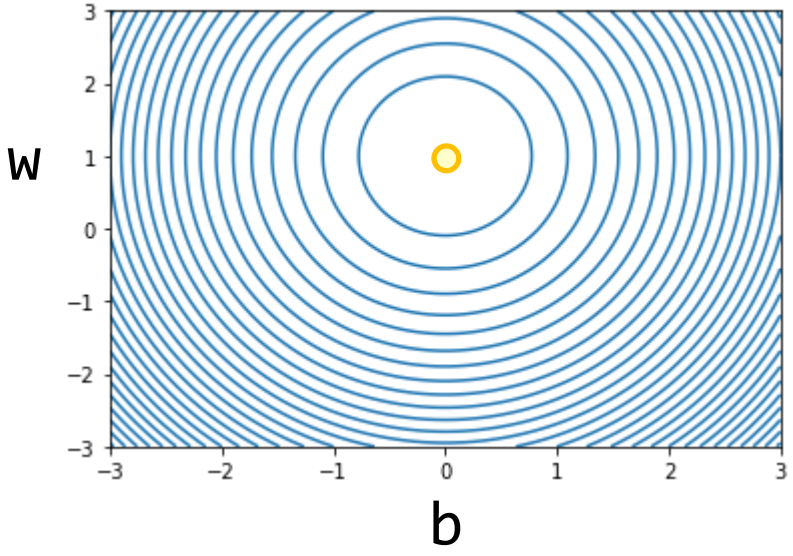
$$\frac{\partial}{\partial b} J(w,b) = \frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b}$$

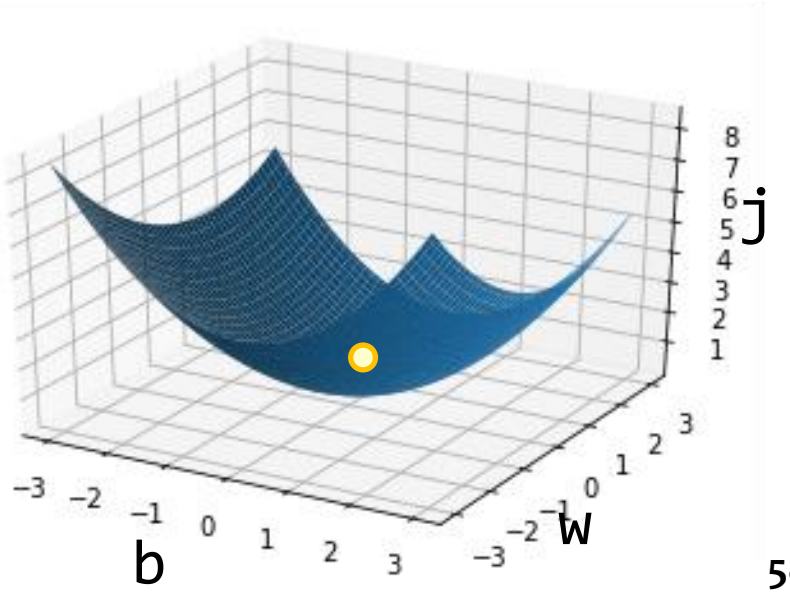
Gradient descent algorithm





J(w,b)=Loss





2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 역전파 구현

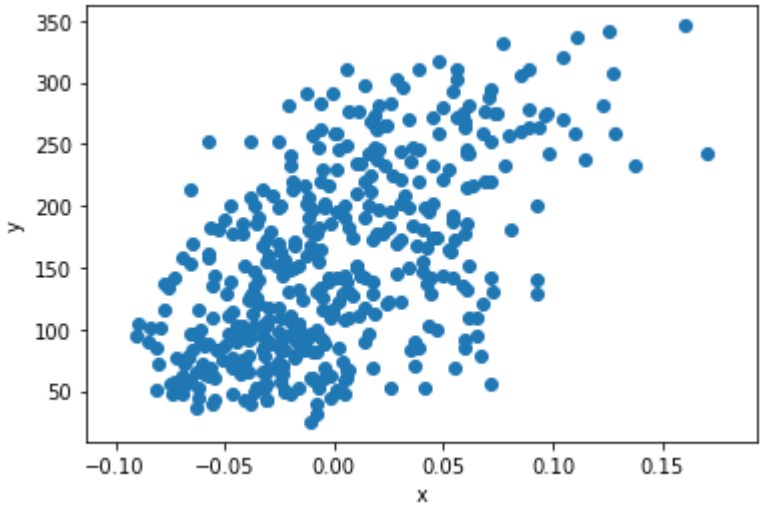
2.4 선형회귀 구현

2.5 시그모이드 함수

2.6 로지스틱 회귀 구현

data set 준비(sklearn 이용)

```
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
import matplotlib.pyplot as plt

x = diabetes.data[:, 2]
y = diabetes.target

plt.scatter(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

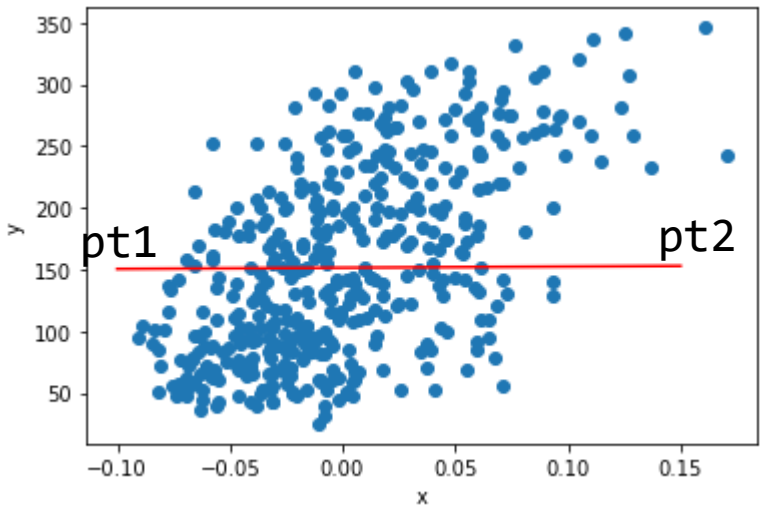


역전파 구현

1. batch=1
2. 점의 전체 개수만큼 1번 순회 (epoch=1)
   : SGD=>원소 하나마다 w,b 갱신

```
w = 1.0
b = 1.0
rate = 0.01
for x_i, y_i in zip(x, y):
    y_hat = x_i * w + b
    err = y_hat - y_i
    w = w - rate * err * x_i
    b = b - rate * err

plt.scatter(x, y)
pt1 = (-0.1, -0.1 * w + b)
pt2 = (0.15, 0.15 * w + b)
plt.plot([pt1[0], pt2[0]], [pt1[1], pt2[1]], 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
w= 9.819118656206367
b= 151.35590713560302
```

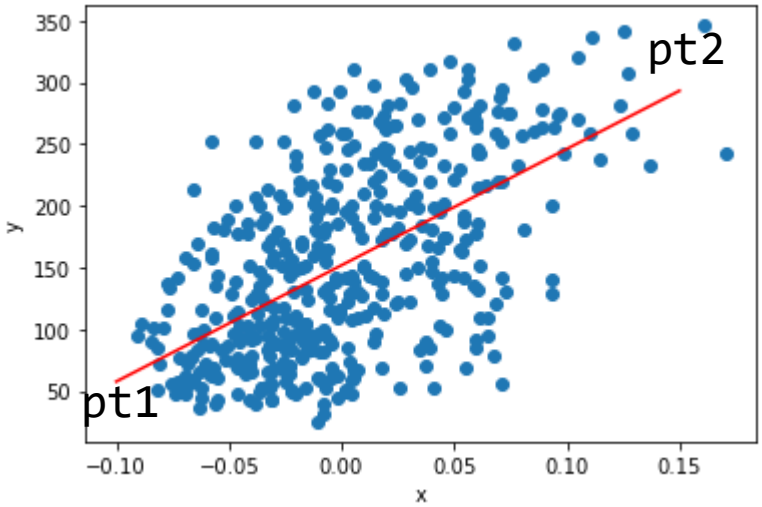


역전파 구현

1. batch=1
2. epoch=1000번 만큼 외부 루프에서 순회
3. 점의 전체 개수만큼 1번 순회
   : SGD=>원소 하나마다 w,b 갱신

```
w = 1.0
b = 1.0
rate = 0.01
for i in range(1000):
    for x_i, y_i in zip(x, y):
        y_hat = x_i * w + b
        err = y_hat - y_i
        w = w - rate * err * x_i
        b = b - rate * err

plt.scatter(x, y)
pt1 = (-0.1, -0.1 * w + b)
pt2 = (0.15, 0.15 * w + b)
plt.plot([pt1[0], pt2[0]], [pt1[1], pt2[1]], 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
w= 942.7013592248312
b= 151.74608061033615
```



2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 역전파 구현

2.4 선형회귀 구현

2.5 시그모이드 함수

2.6 로지스틱 회귀 구현

Neuron class 만들기

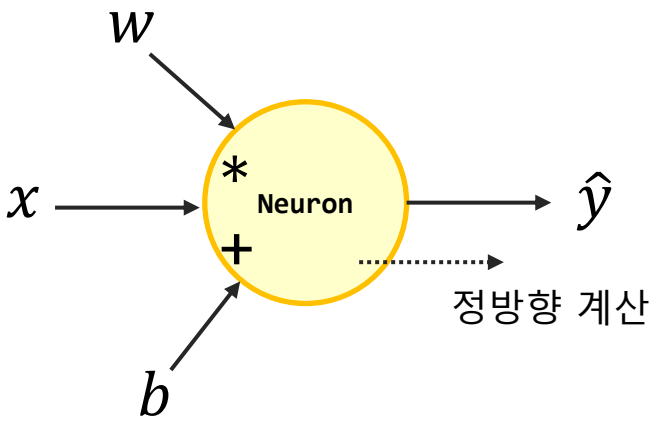
```
class Neuron:
    def __init__(self):
    # 초기화 작업을 수행합니다.
    ...

    # 필요한 함수를 추가합니다.
    ...
```

1. __init__() 함수 만들기

```
    def __init__(self):
        self.w = 1.0
        self.b = 1.0
```

2. 정방향 계산 만들기

```
    def forpass(self, x):
        y_hat = x * self.w + self.b        # 직선 방정식을 계산합니다
        return y_hat
```



$\hat{y} = wx + b$ # 아핀연산(affine)

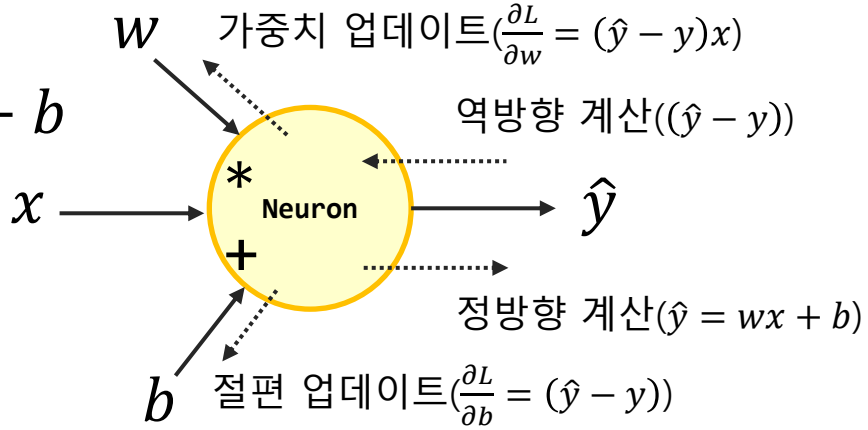
3. 역방향 계산 만들기

```
    def backprop(self, x, err):
        w_grad = x * err    # 가중치에 대한 그래디언트를 계산합니다
        b_grad = 1 * err    # 절편에 대한 그래디언트를 계산합니다
        return w_grad, b_grad
```

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

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

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



4. 훈련을 위한 fit() 함수 구현

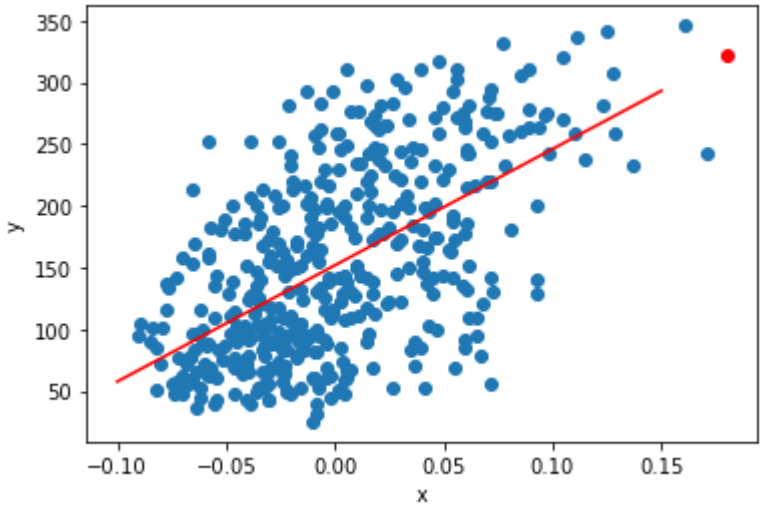
```
    def fit(self, x, y, epochs=1000, rate=0.01):
        for i in range(epochs):              # 에포크만큼 반복합니다
            for x_i, y_i in zip(x, y):       # 모든 샘플에 대해 반복합니다
                y_hat = self.forpass(x_i) # 정방향 계산
                err = y_hat - y_i            # 오차 계산
                w_grad, b_grad = self.backprop(x_i, err)  # 역방향 계산
                self.w -= rate*w_grad            # 가중치 업데이트
                self.b -= rate*b_grad            # 절편 업데이트
```

5. 모델 훈련하기

```
neuron = Neuron()
neuron.fit(x, y)
```

6. 학습이 완료된 모델의 가중치와 절편 확인

```
plt.scatter(x, y)
pt1 = (-0.1, -0.1 * neuron.w + neuron.b)
pt2 = (0.15, 0.15 * neuron.w + neuron.b)
plt.plot([pt1[0], pt2[0]], [pt1[1], pt2[1]],'r')
plt.plot(0.18, 0.18 * neuron.w + neuron.b, 'ro')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 역전파 구현

2.4 선형회귀 구현

2.5 시그모이드 함수

2.6 로지스틱 회귀 구현

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

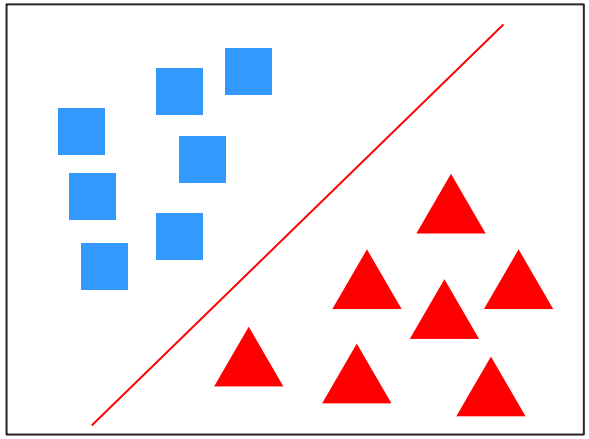
Binary Classification(이진 분류) 란?

: 값은 0 또는 1

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

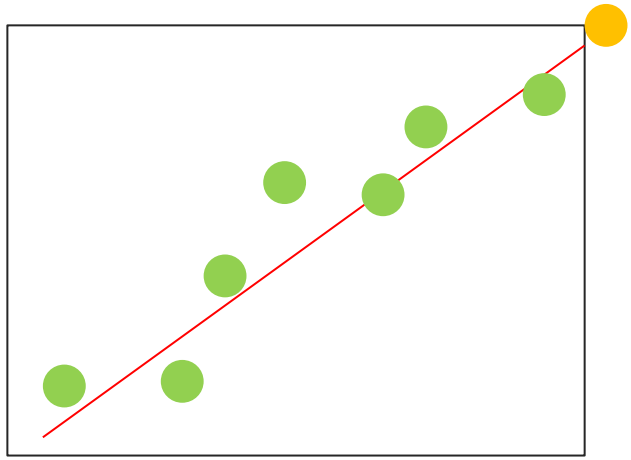
Logistic vs Linear

로지스틱 회귀와 선형 회귀의 차이점은?



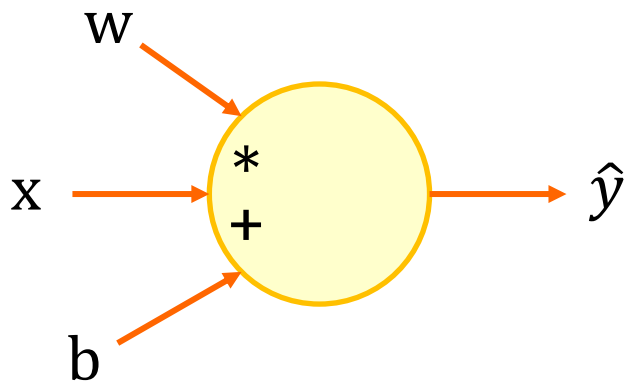
Discrete(분리) : 분류 목적

신발 사이즈 / 회사의 근로자수

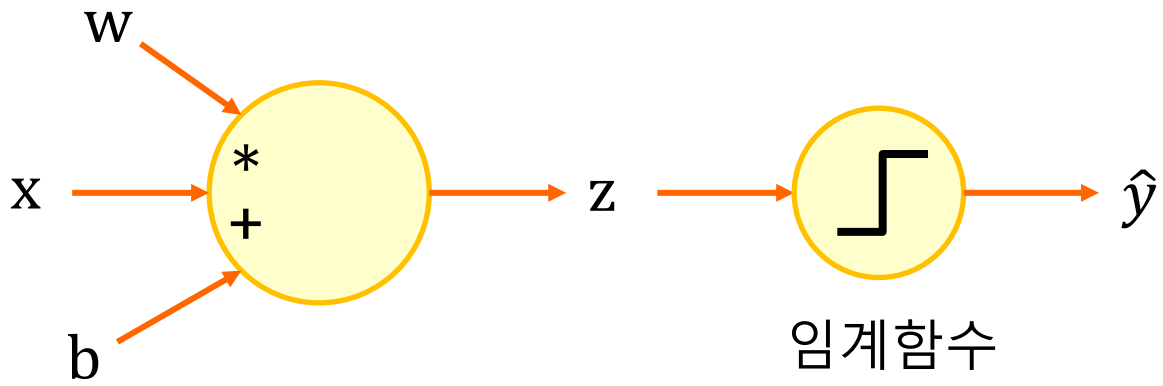


Continuous (지속적인) : 값의 예측

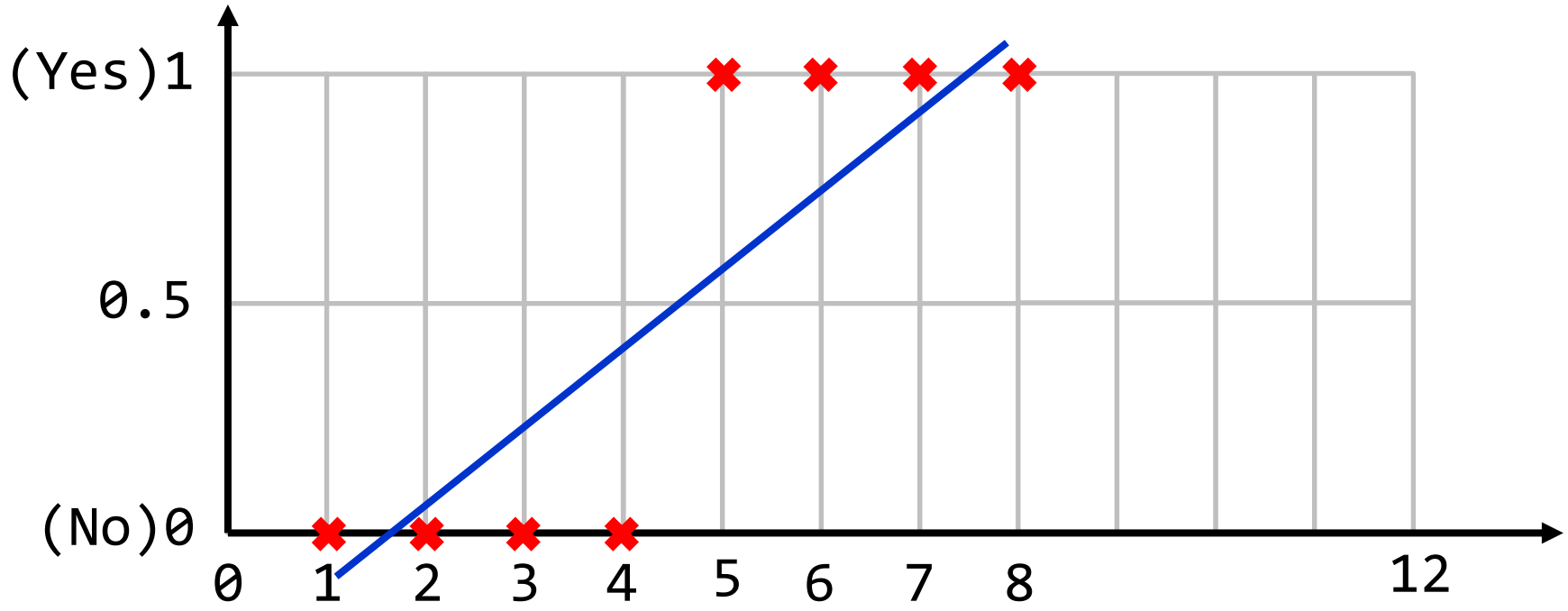
시간/ 무게 / 높이



퍼셉트론



임계함수



Threshold classifier output at 0.5:

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

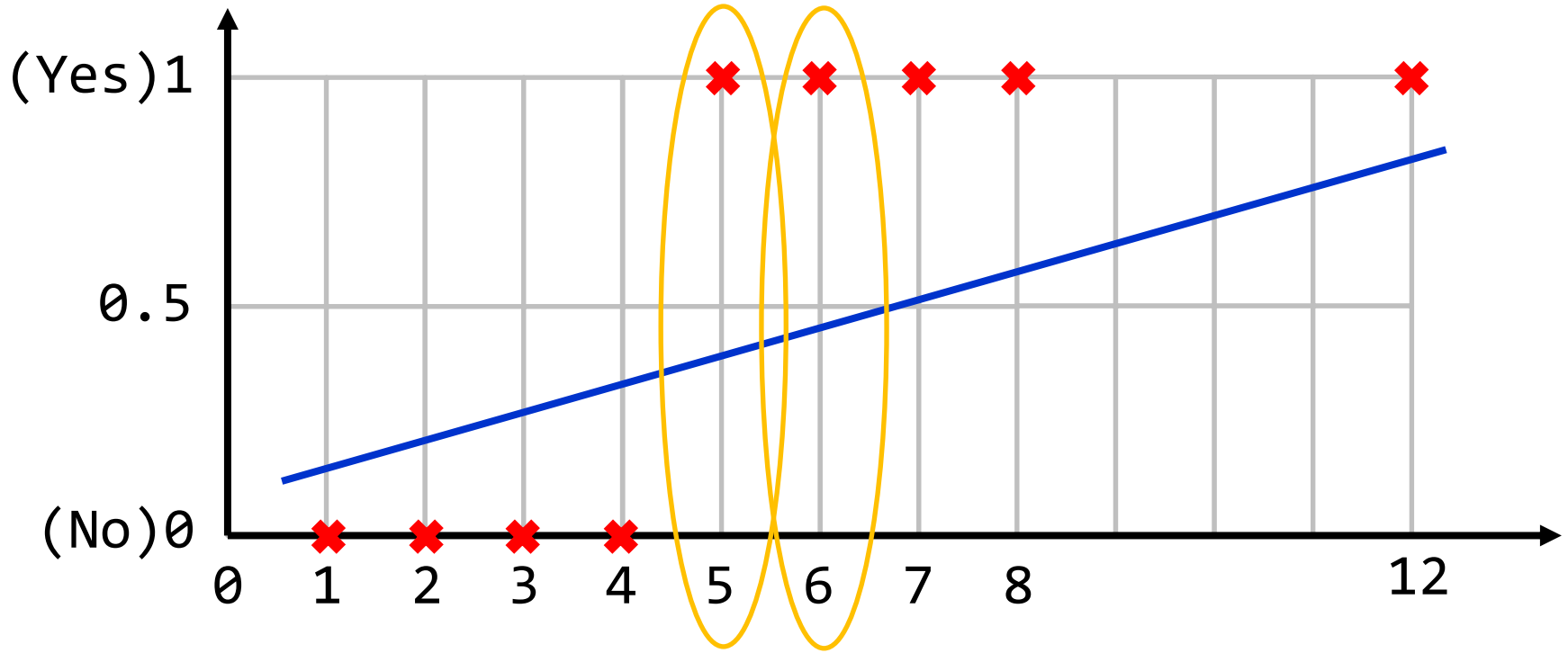
If $\hat{y} < 0.5$, predict "y=0"

1 => 양성 5 => 악성

2 => 양성 6 => 악성

3 => 양성 7 => 악성

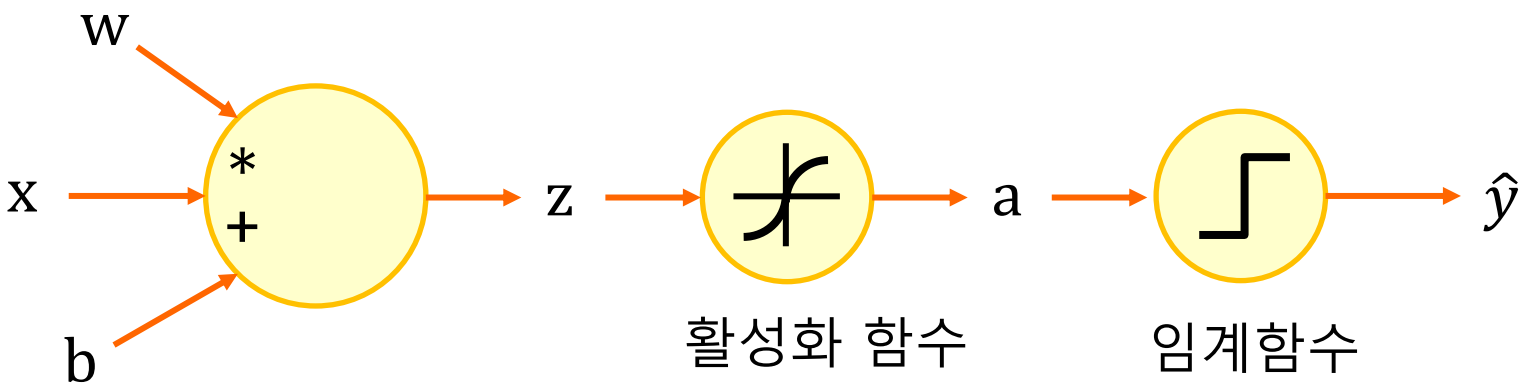
4 => 양성 8 => 악성



Threshold classifier output at 0.5:

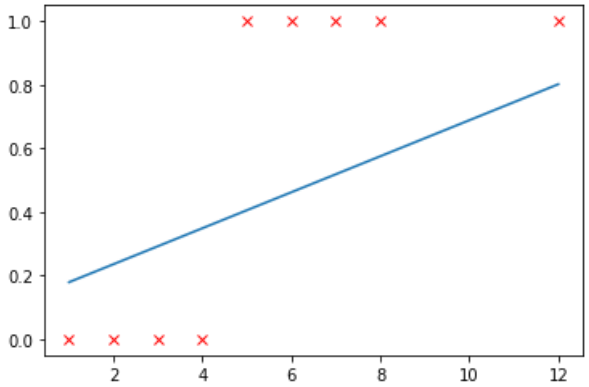
If $\hat{y} \geq 0.5$, predict "y=1"
If $\hat{y} < 0.5$, predict "y=0"

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


w
x
b
*
+
z
a
$\hat{y}$
활성화 함수
임계함수

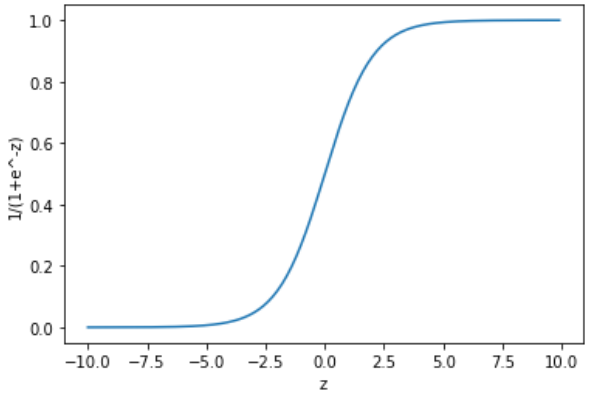
Classification: y=0 or 1

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



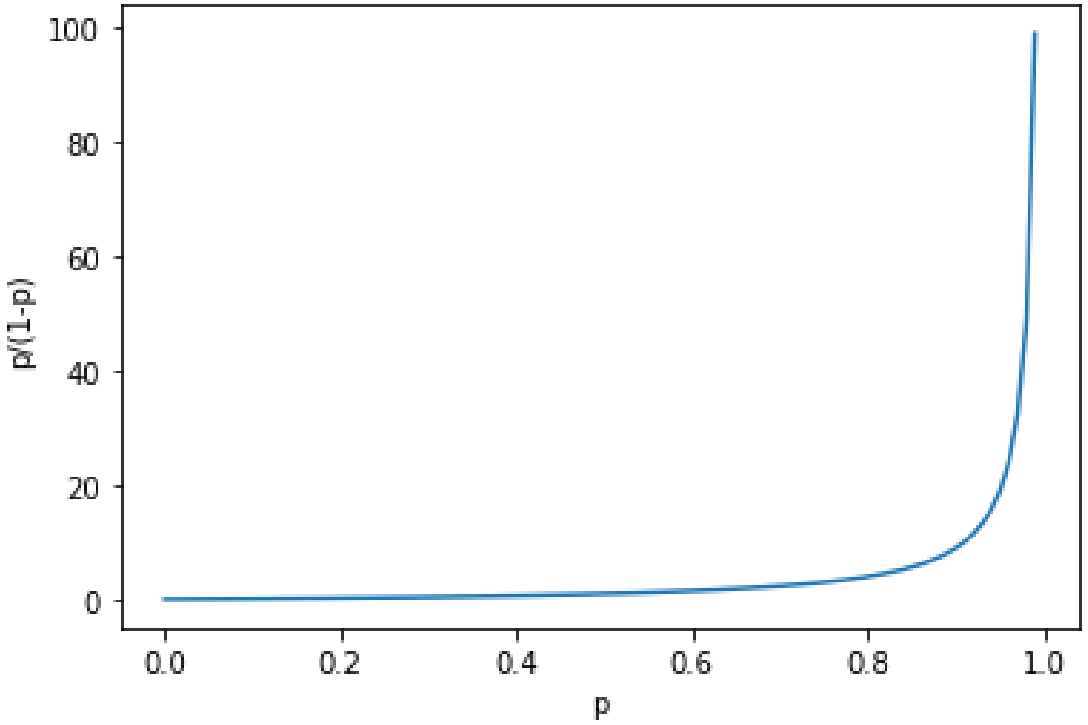
Logistic Regression: $0 < \hat{y} < 1$

분류 문제 사용



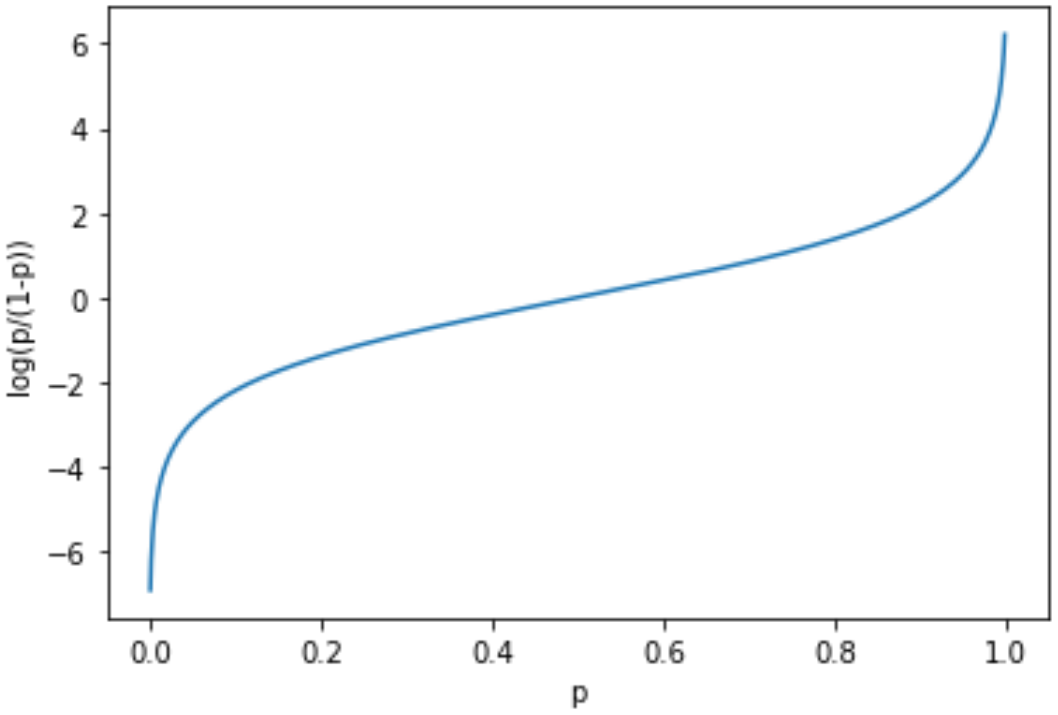
$$\text{OR(odds ratio)} = \frac{p}{1-p} \text{ (p=성공확률)}$$

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



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

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



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

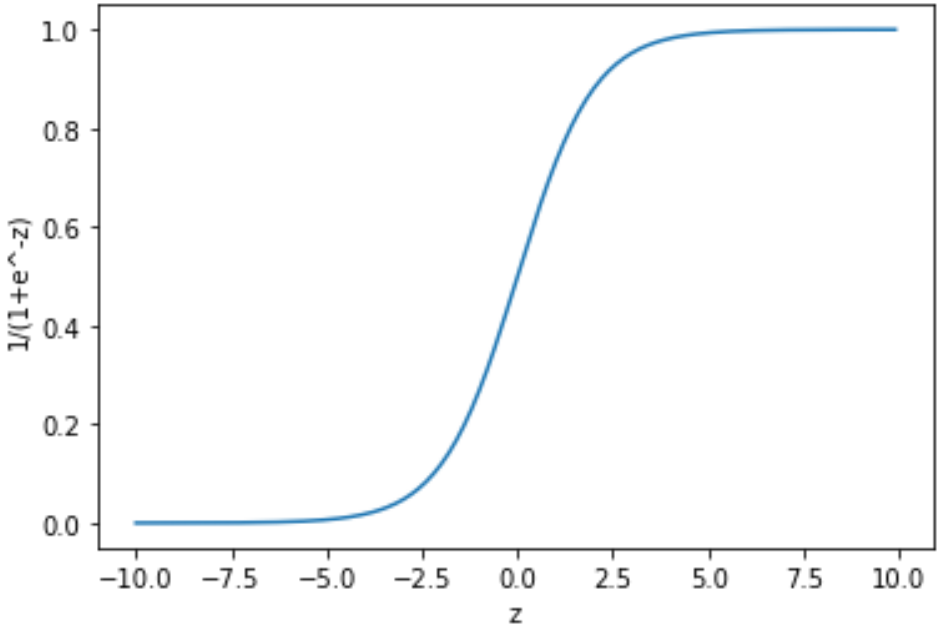
로짓 함수의 유도 : p에대해 정리

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

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

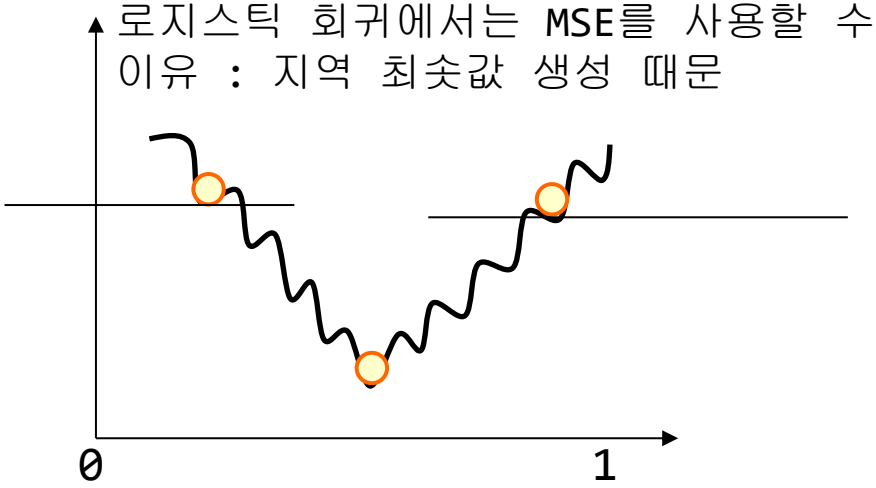
$$\frac{p}{1-p} = e^{z}$$

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

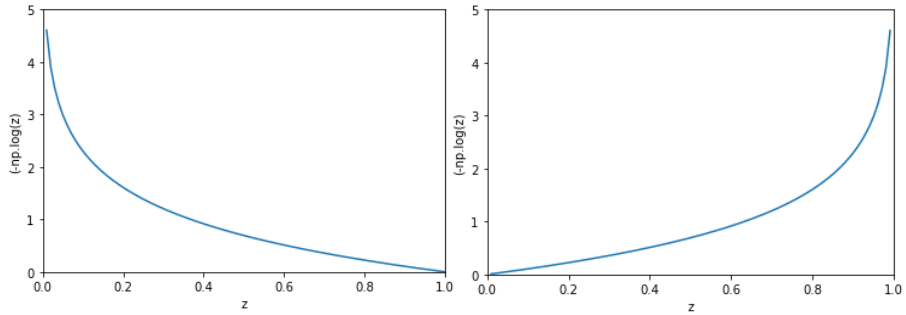


A convex logistic regression cost function

로지스틱 회귀에서는 MSE를 사용할 수 없다.
이유 : 지역 최솟값 생성 때문



Binary Cross Entropy 함수



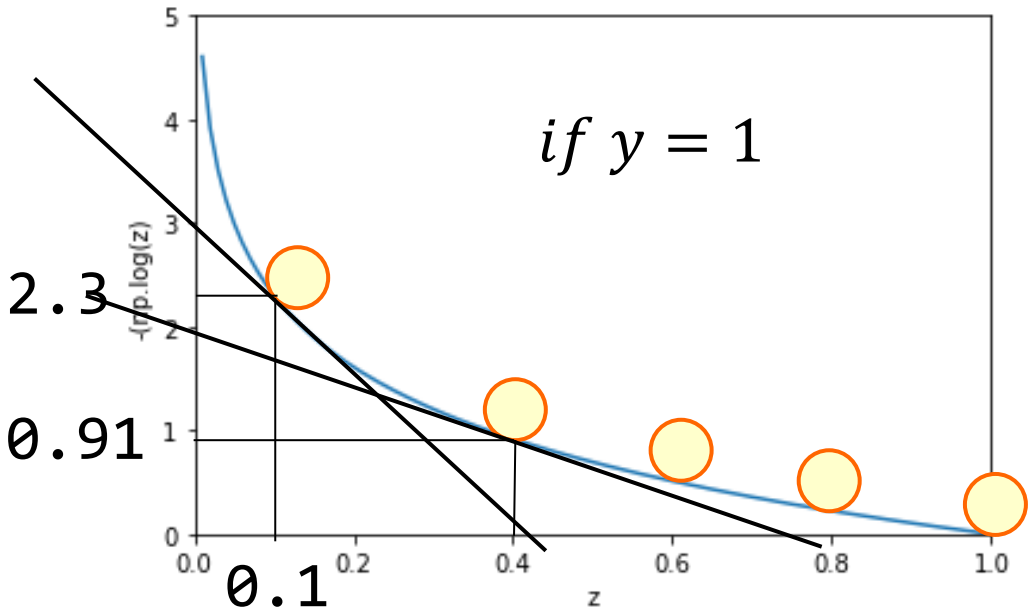
$$\text{Cost}(\hat{y}, y) = \begin{cases} -\log(\hat{y}) & if\ y = 1 \\ -\log(1-\hat{y}) & if\ y = 0 \end{cases}$$

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

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

Logistic regression cost function

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



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

But as $\hat{y} \rightarrow 0$

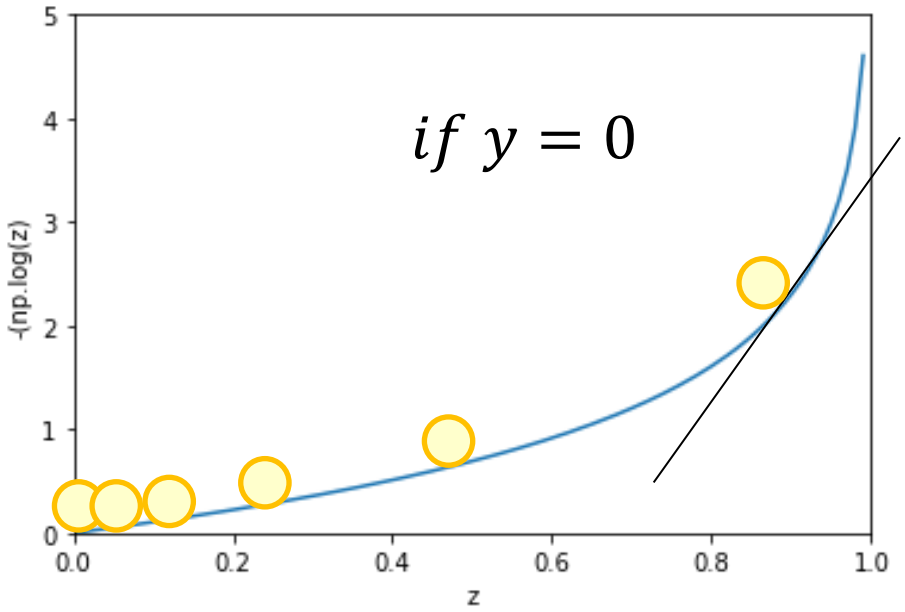
$\text{Cost} \rightarrow \infty$

$\hat{y} \rightarrow 0.6$

$\text{Cost} \rightarrow 2.3 \rightarrow 0.91 \rightarrow 0.51$

Logistic regression cost function

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



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

But as $\hat{y} \rightarrow 1$

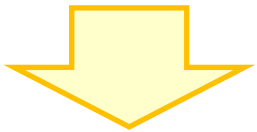
$\text{Cost} \rightarrow \infty$

$\hat{y} \rightarrow 0.9 \rightarrow 0.5$

$\text{Cost} \rightarrow 2.3 \rightarrow 0.69$

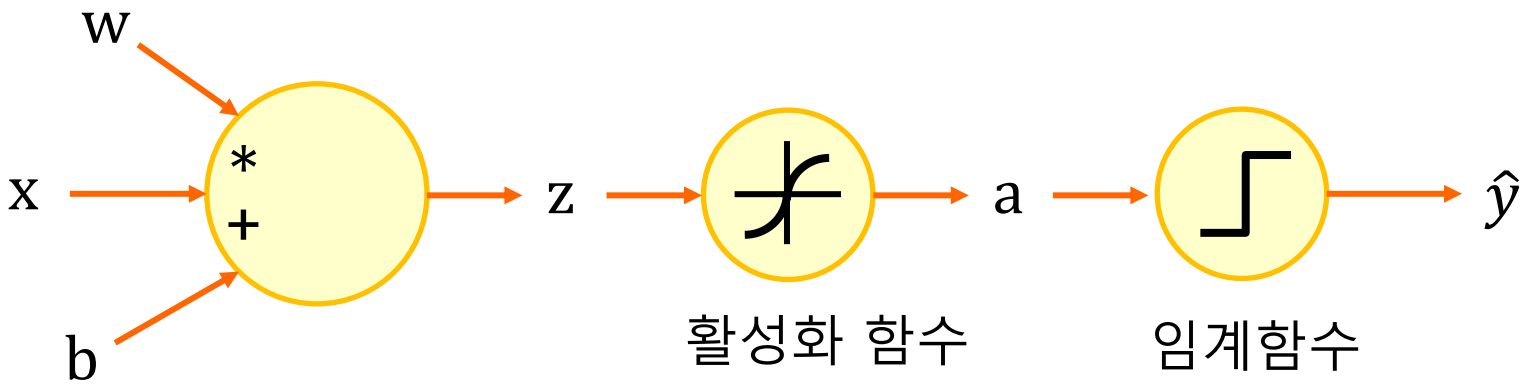
Logistic regression cost function

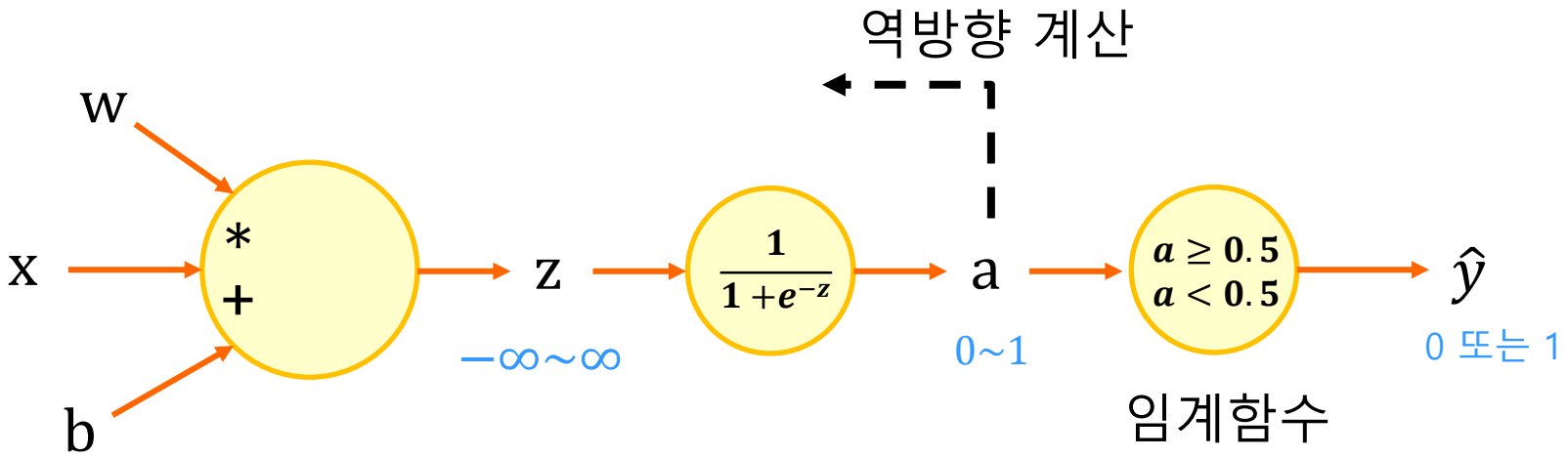
$$\text{Cost}(\hat{y}, y) = \begin{cases} -\log(\hat{y}) & if\ y = 1 \\ -\log(1-\hat{y}) & if\ y = 0 \end{cases}$$



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

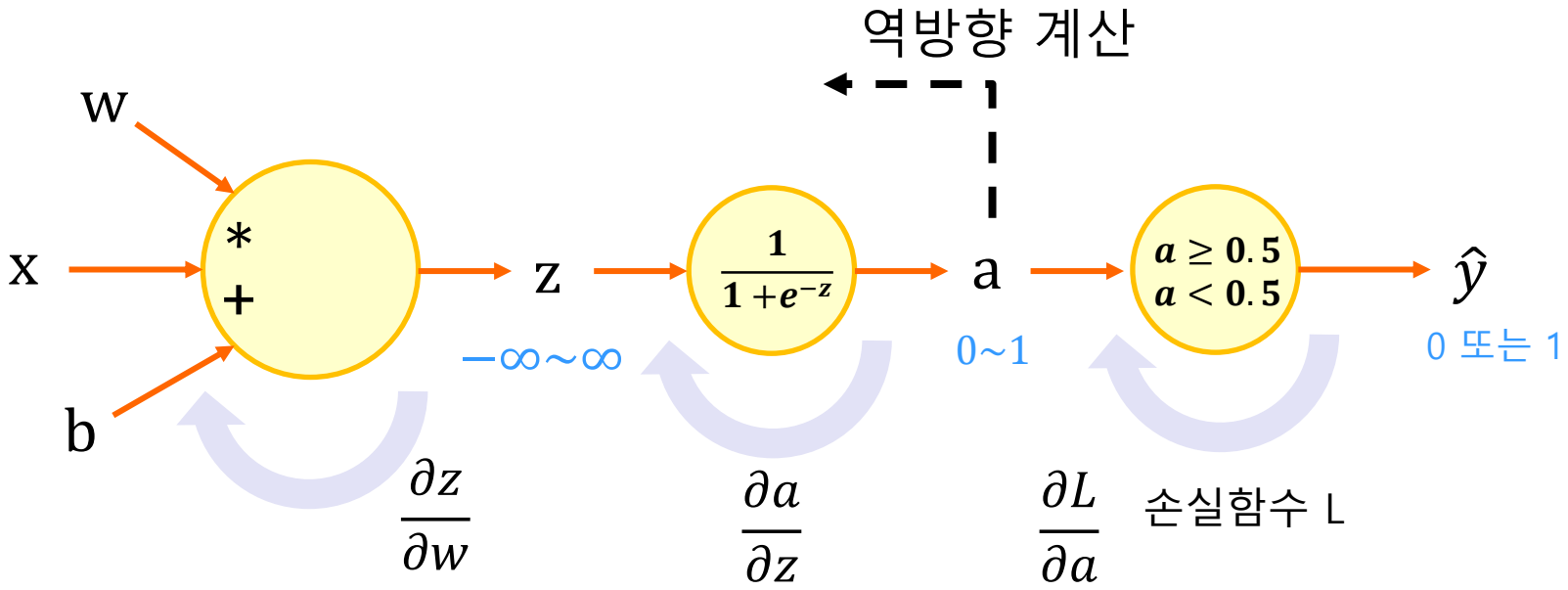
$$L = \frac{1}{2}((\hat{y}) - y)^2 \quad \text{MSE}$$


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

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

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

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

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

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

chain rule을 이용하여 각 단계의 미분 결과를 곱한다.

w에 대하여 미분

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

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

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

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

$$-(y\frac{1}{a} - (1-y)\frac{1}{1-a})\ a(1-a)$$

$$-(y(1-a) - (1-y)a)$$

$$-(y - ya - a + ya)$$

$$(a-y)$$

특성이 두개인 경우 => x 2개

$y = w_1x_1 + w_2x_2 + b$

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

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

$$\frac{\partial}{\partial w_2} w_1x_1 + w_2x_2 + b$$

$$= x_2$$

$$\frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2} = x_2(\hat{y} - y)$$

특성이 두개인 경우 => x 1개

$y = wx+b$

Chain Rule 사용

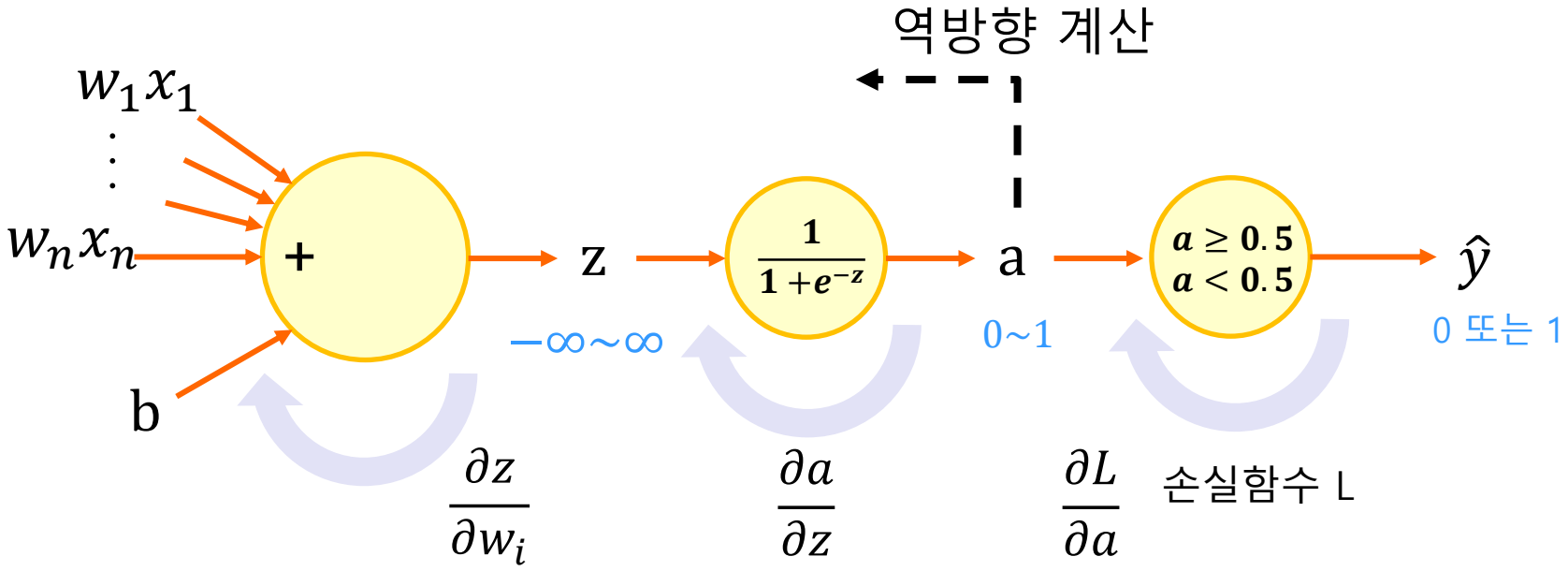
$$\hat{y} = w_1x_1 + w_2x_2 + b$$

$$J(w_1, w_2, b) = \frac{1}{2}(\hat{y} - y)^2$$

$$\frac{\partial}{\partial w_2} J(w_1, w_2, b) = \frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

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

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

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

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

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

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

2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 역전파 구현

2.4 선형회귀 구현

2.5 시그모이드 함수

2.6 로지스틱 회귀 구현

분류용 데이터셋을 준비(위스콘신 유방암 데이터 세트)

	의학	이진 분류
좋음	양성 종양	0
나쁨	악성 종양	1

사이킷런의 datasets 에서 불러온다.

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

data와 target의 크기를 알아본다.

```
print(cancer.data.shape, cancer.target.shape)
```

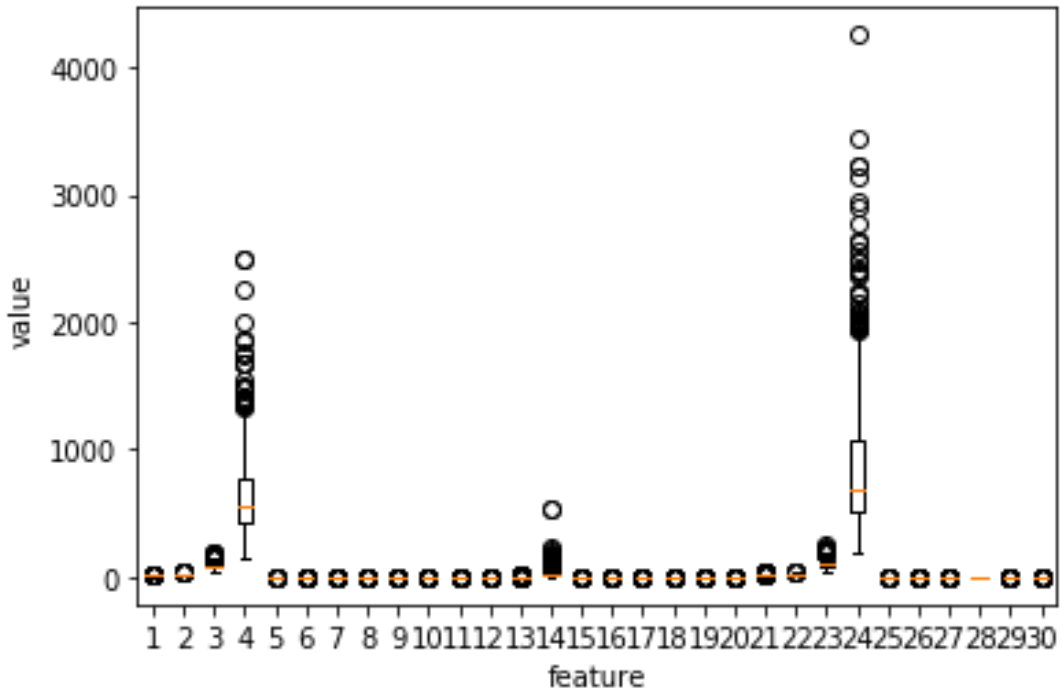
```
(569, 30)        (569,)
```

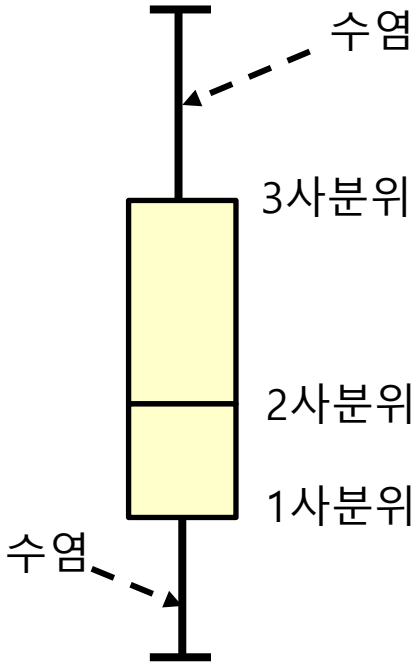
cance에는 569개의 샘플과 30개의 특성이 있다.

```
cancer.data[:3]
array([[1.799e+01, 1.038e+01, 1.228e+02, 1.001e+03, 1.184e-01, 2.776e-01,
        3.001e-01, 1.471e-01, 2.419e-01, 7.871e-02, 1.095e+00, 9.053e-01,
        8.589e+00, 1.534e+02, 6.399e-03, 4.904e-02, 5.373e-02, 1.587e-02,
        3.003e-02, 6.193e-03, 2.538e+01, 1.733e+01, 1.846e+02, 2.019e+03,
        1.622e-01, 6.656e-01, 7.119e-01, 2.654e-01, 4.601e-01, 1.189e-01],
       [2.057e+01, 1.777e+01, 1.329e+02, 1.326e+03, 8.474e-02, 7.864e-02,
        8.690e-02, 7.017e-02, 1.812e-01, 5.667e-02, 5.435e-01, 7.339e-01,
        3.398e+00, 7.408e+01, 5.225e-03, 1.308e-02, 1.860e-02, 1.340e-02,
        1.389e-02, 3.532e-03, 2.499e+01, 2.341e+01, 1.588e+02, 1.956e+03,
        1.238e-01, 1.866e-01, 2.416e-01, 1.860e-01, 2.750e-01, 8.902e-02],
       [1.969e+01, 2.125e+01, 1.300e+02, 1.203e+03, 1.096e-01, 1.599e-01,
        1.974e-01, 1.279e-01, 2.069e-01, 5.999e-02, 7.456e-01, 7.869e-01,
        4.585e+00, 9.403e+01, 6.150e-03, 4.006e-02, 3.832e-02, 2.058e-02,
        2.250e-02, 4.571e-03, 2.357e+01, 2.553e+01, 1.525e+02, 1.709e+03,
        1.444e-01, 4.245e-01, 4.504e-01, 2.430e-01, 3.613e-01, 8.758e-02]])
```

박스 플롯을 이용하면 각 특성의 사분위 값을 볼수 있다.

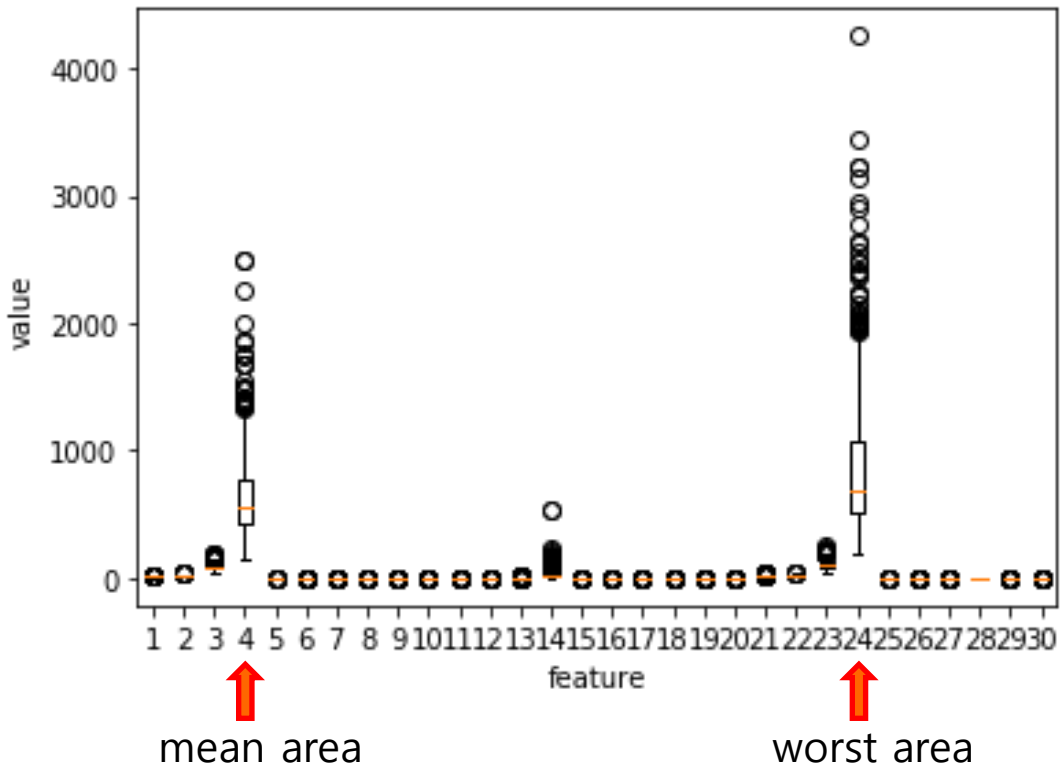
```
plt.boxplot(cancer.data)
plt.xlabel('feature')
plt.ylabel('value')
plt.show()
```





눈에 띄는 특성 보기 : 모두 넓이에 관한 특성이다.

```
cancer.feature_names[[3,23]]

array(['mean area', 'worst area'], dtype='<U23')
```



타겟 데이터 확인

```
np.unique(cancer.target, return_counts=True)

(array([0, 1]), array([212, 357], dtype=int64))
```

넘파이의 unique 함수는 중복 제거후 고유한 값을 찾아 반환한다.

cancer.target에는 0=>양성, 1=>악성 의 값이 있다.

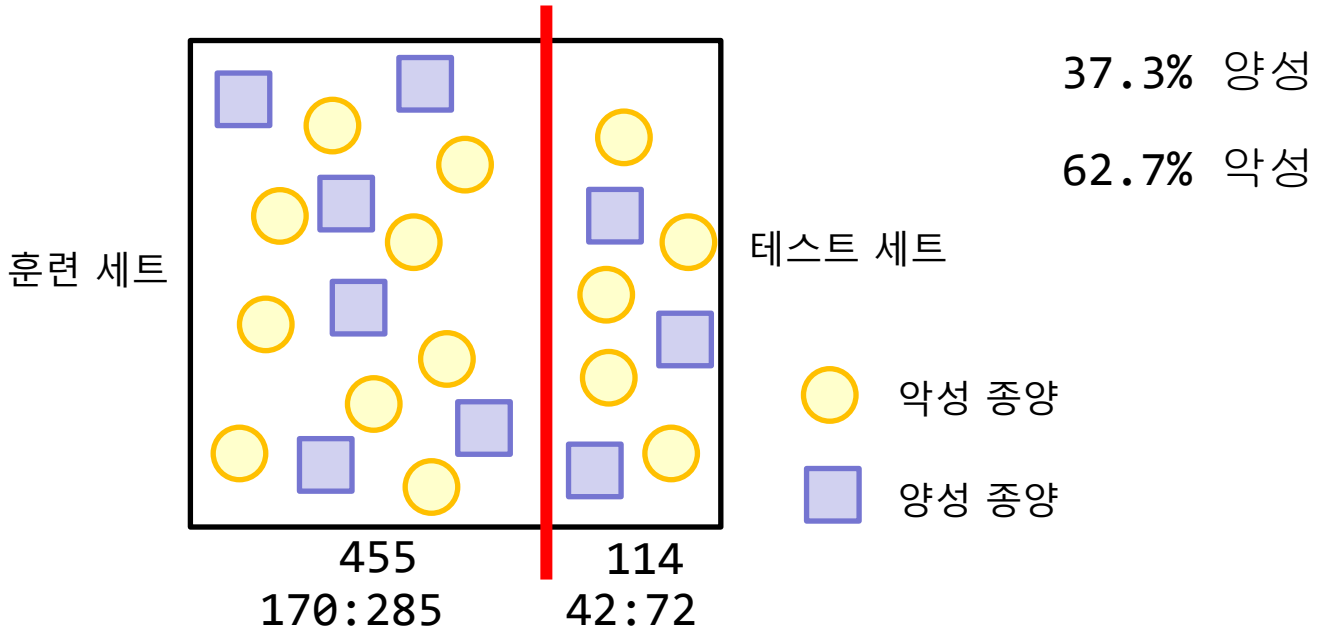
return_counts에는 고유한 값인 0의 개수와 1의 개수가 리턴 되므로

양성종양 212개와 357개의 악성종양 값이 있다.

모델의 성능 평가를 위한 훈련 세트와 테스트 세트

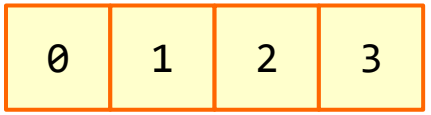
: 모델을 학습 시킨 훈련 데이터 세트로 성능을 평가 하면 상당이 좋은 성능이 나온다.
하지만 이렇게 하면 '과도하게 낙관적으로 일반화 성능을 추정한다' 라고 한다.
따라서 전체 데이터를 나누어 훈련 데이터 세트로 학습시킨 후 테스트 데이터로 성능을 평가 해야 한다.

주의 사항
- 보통 훈련 데이터가 더 많아야 하므로 8:2 정도로 나누어 훈련과 테스트를 진행한다.
- 데이터가 어느 한쪽에 몰리지 않도록 비율이 일정하게 골고루 섞어야 한다.

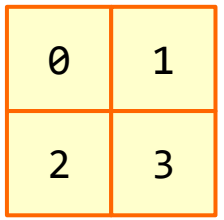


ctrl+d : 복사+붙혀넣기

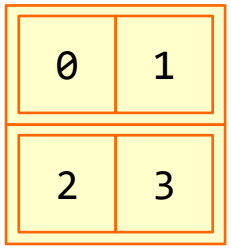
```
a = np.arange(4)
```



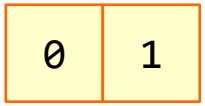

```
a = np.arange(4).reshape(2,2)
len(a) # a.shape[0] (2,2)
print(a[0].shape)  # (2,)
print(a[0])
```



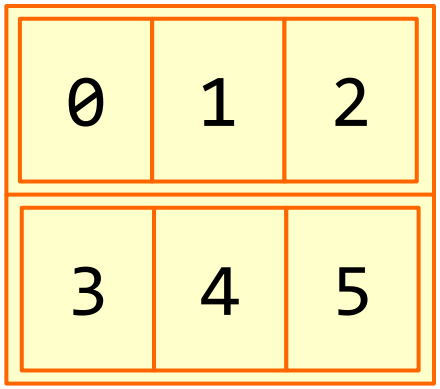
(2,2)

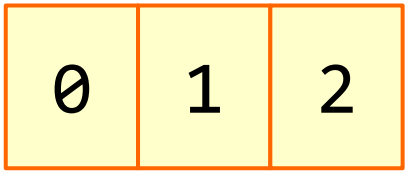


(2,)



```
a = np.arrange(6).reshape(2,3)
len(a)    # 2
len(a[0]) # 3
```





타겟 데이터 확인

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, stratify=y,
test_size=0.2, random_state=42)
```

훈련 데이터와 테스트 데이터의 비율을
8:2로 설정한다.

난수 초깃값 42로 지정하여 무작위로 데이터 세트를 섞은 다음 나눈다.

훈련 데이터를 나눌때 class의 비율을 동일하게 만든다.

8:2 나눈 비율 확인

```
print(x_train.shape, x_test.shape)
(455, 30) (114, 30)
```

data 비율 유지 확인

```
np.unique(y_train, return_counts=True)
(array([0, 1]), array([170, 285], dtype=int64))
```

이전 코드에서 바뀐 코드 설명

$$x_1 w_1 + x_2 w_2 + \cdots$$

```
def __init__(self):
        self.w = None
        self.b = None
```

특성이 하나가 아니라 여러개가 될수 있으므로 __init__함수에서 초기화 하지 않고
실행중 x에 입력되는 특성의 개수에 따라 초기화 한다.

```
np.sum(x * self.w)
```

forpass함수의 해당 코드는 x가 더이상 하나의 값이 아니라 30개의 특성을 가지고 있는 넘파이 배열이므로
np.sum(x * self.w) => x[0]*w[0] + x[1]*w[1] + ... + x[29]*w[29] 로 해석 된다.

```
self.w = np.ones(x.shape[1])
```

fit 함수의 w 초기화는 x.shape[1]의 값이 30 이므로 넘파이 배열의 개수를 30개로 할당 하고
ones 함수에 의해 각 원소는 1로 초기화 된다.

```
def activation(self, z):
        a = 1 / (1 + np.exp(-z))  # 시그모이드 계산
        return a
```

activation 함수를 구현하고 시그모이드 함수의 결과 값을 리턴한다.

이전 코드에서 바뀐 코드 설명

```
def predict(self, x):
        z = [self.forpass(x_i) for x_i in x]    # 정방향 계산
        a = self.activation(np.array(z))        # 활성화 함수 적용
        return a >= 0.5
```

예측 함수를 구현한다.

z = [self.forpass(x_i) for x_i in x] 코드는 파이썬에서 지원하는 코드로 대괄호([]) 안에 for in과 함수의 호출부를 넣을수 있다. 이때 동작은 x의 각 원소를 x_i에 뽑아서 x_i를 forpass의 인자로 전달하여 forpass의 리턴값을 list로 append 하여 x의 가공된 값을 z에 대입한다.

np.array(z) 함수는 list를 넘파이 배열로 바꿔준다.

a에는 최종적으로 시그모이드 함수의 결과가 대입되므로

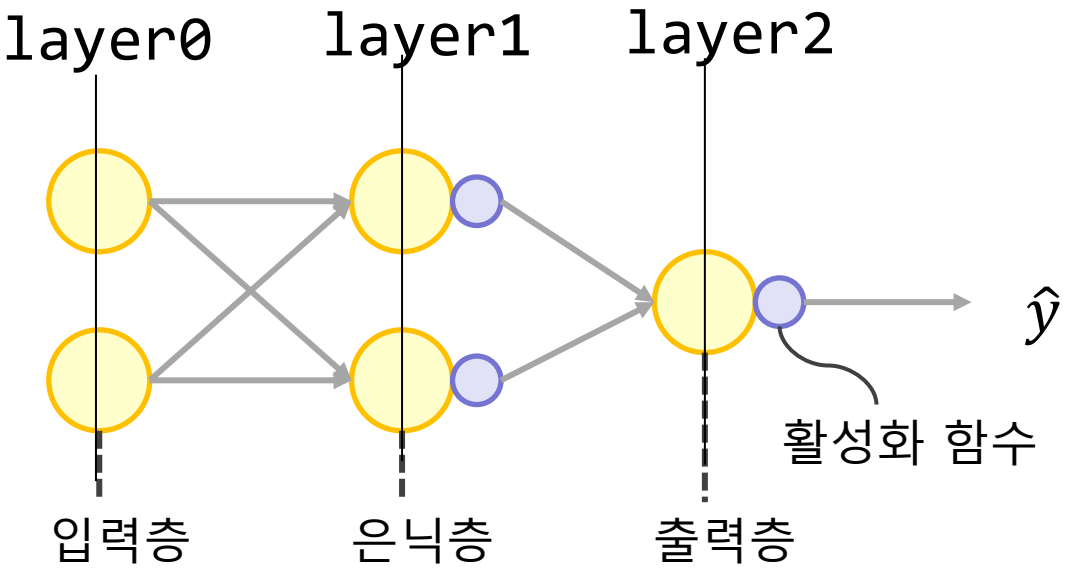
결과값이 0.5 보다 크다면 악성종양으로 판단하여 1을 리턴한다.

```
np.mean(neuron.predict(x_test) == y_test)
0.8245614035087719
```

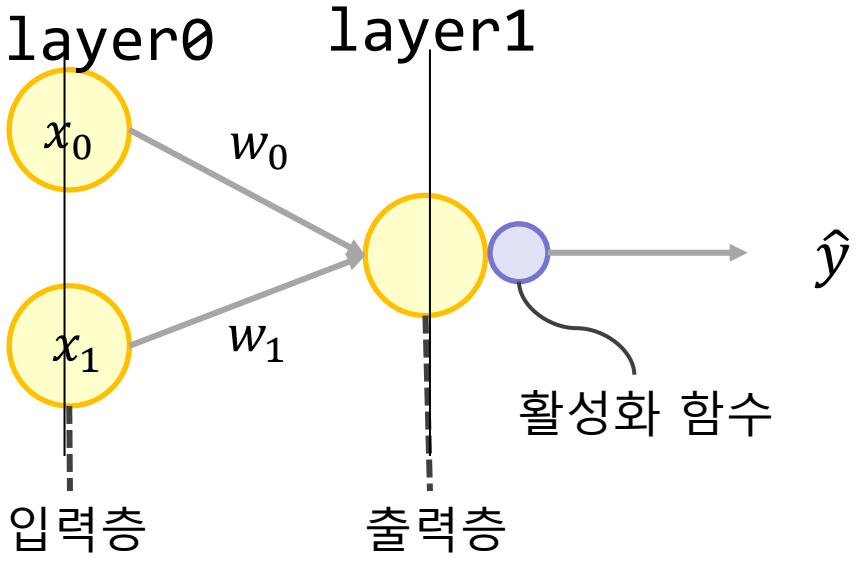
np.mean 함수는 평균을 계산하는 함수로 예측값과 실제값의 비교 값을 평균을 계산하여 정확도를 측정한다.

3. 신경망 시작하기

일반적인 신경망 모습



단일층 신경망



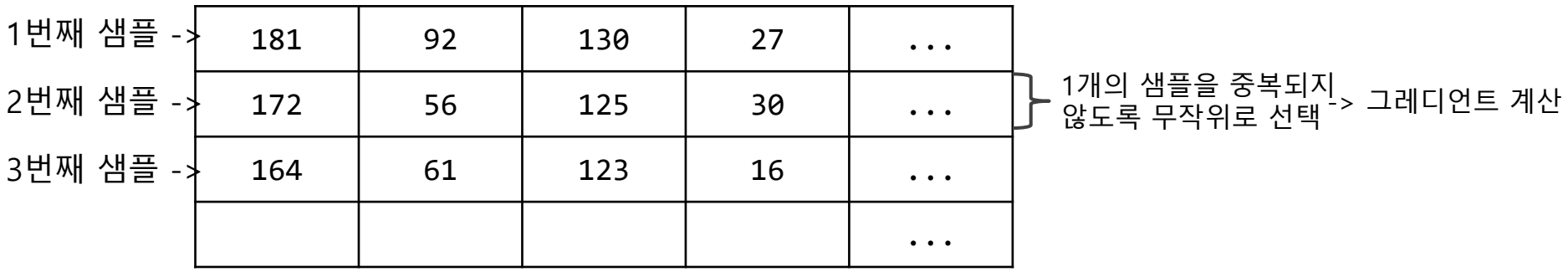
손실 함수의 결과값 조정해 저장 기능 추가하기

```
def __init__(self):
    self.losses = []   # empty list 손실 함수의 결과값을 저장할 리스트

def fit(self, x, y, epochs=100):
        ...
          for i in indexes:                          # 모든 샘플에 대해 반복합니다
              z = self.forpass(x[i])                 # 정방향 계산
              a = self.activation(z)                 # 활성화 함수 적용
              err = -(y[i] - a)                      # 오차 계산
              w_grad, b_grad = self.backprop(x[i], err) # 역방향 계산
              self.w -= w_grad                       # 가중치 업데이트
              self.b -= b_grad                       # 절편 업데이트
              # 안전한 로그 계산을 위해 클리핑한 후 손실을 누적합니다
              a = np.clip(a, 1e-10, 1-1e-10)
              loss += -(y[i]*np.log(a)+(1-y[i])*np.log(1-a))
          # 에포크마다 평균 손실을 저장합니다
          self.losses.append(loss/len(y))
```

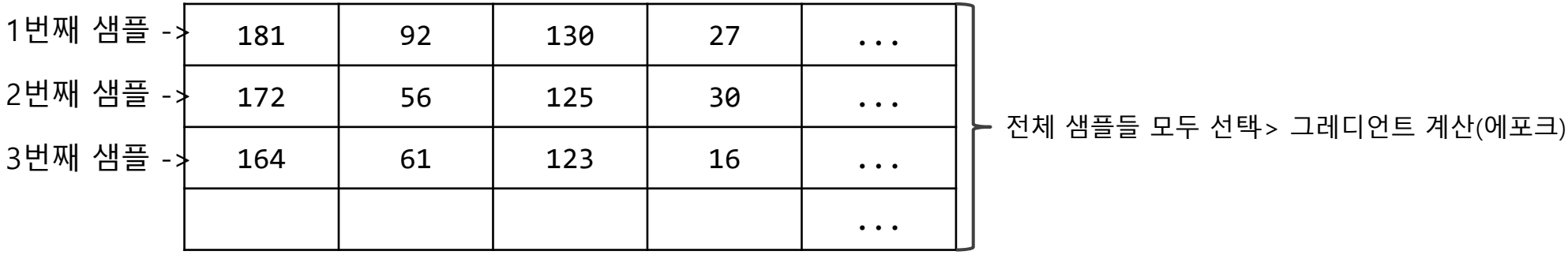
a가 0에 가까워지면 np.log() 함수의 값은 음의 무한대가 되고 a가 1에 가까워지면 np.log() 함수의 값은 0이 된다. 손실값이 무한해 지면 정확한 값을 계산할 수 없으므로 $-1 * 10^{-10} \sim 1 - 1 * 10^{-10}$ 사이가 되록 np.clip() 함수로 조정한다.

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

여러 가지 경사 하강법



확률적 경사 하강법 (SGD)



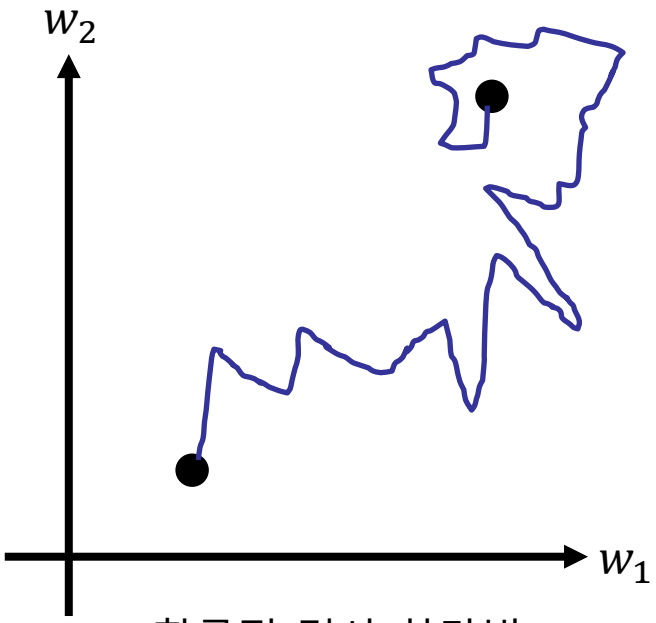
배치 경사 하강법

여러 가지 경사 하강법

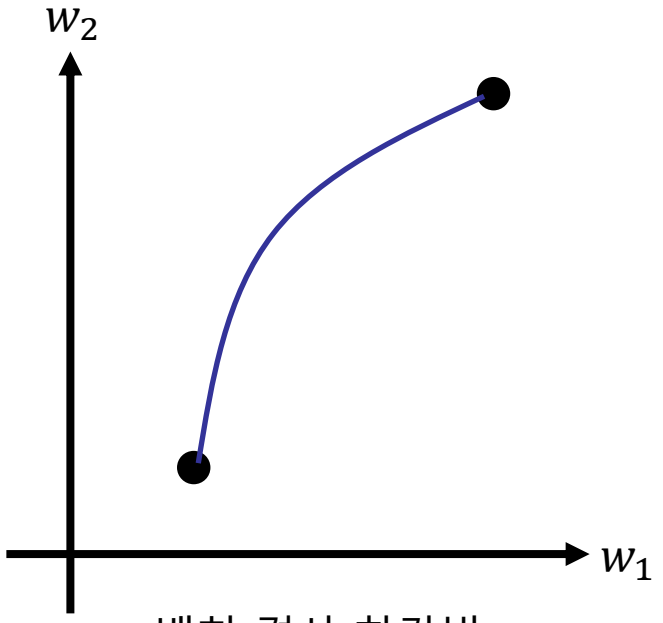
1번째 샘플 ->	181	92	130	27	...
2번째 샘플 ->	172	56	125	30	...
3번째 샘플 ->	164	61	123	16	...
					...

전체 샘플 중 몇 개의
샘플을 중복되지 않도록> 그레디언트 계산
무작위로 선택

미니 배치 경사 하강법



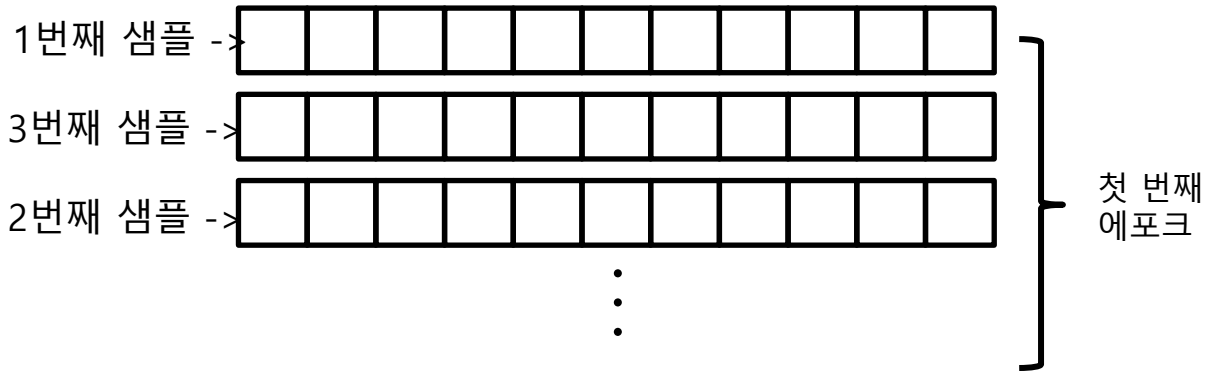
확률적 경사 하강법

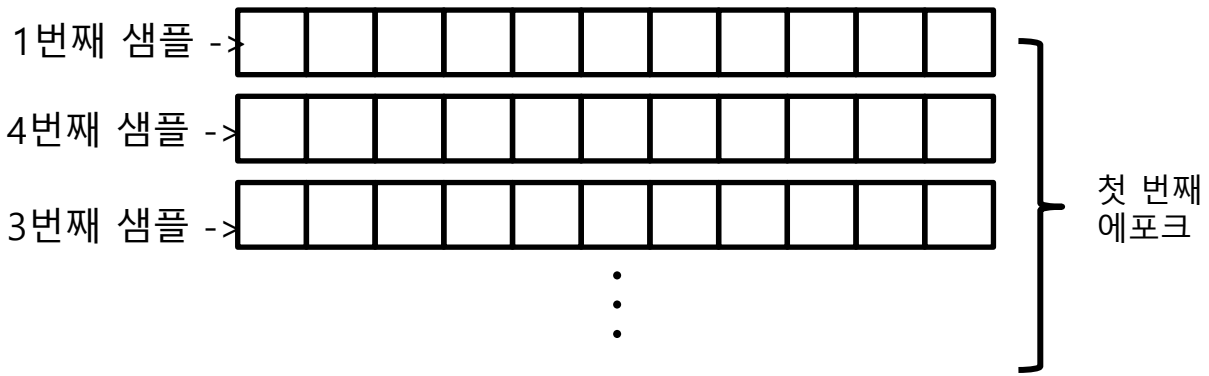


배치 경사 하강법

매 에포크 마다 훈련 세트의 샘플 순서 섞기

SGD : 확률적 경사하강





`np.random.permutation()` 함수를 사용하여 인덱스를 섞을수 있다.

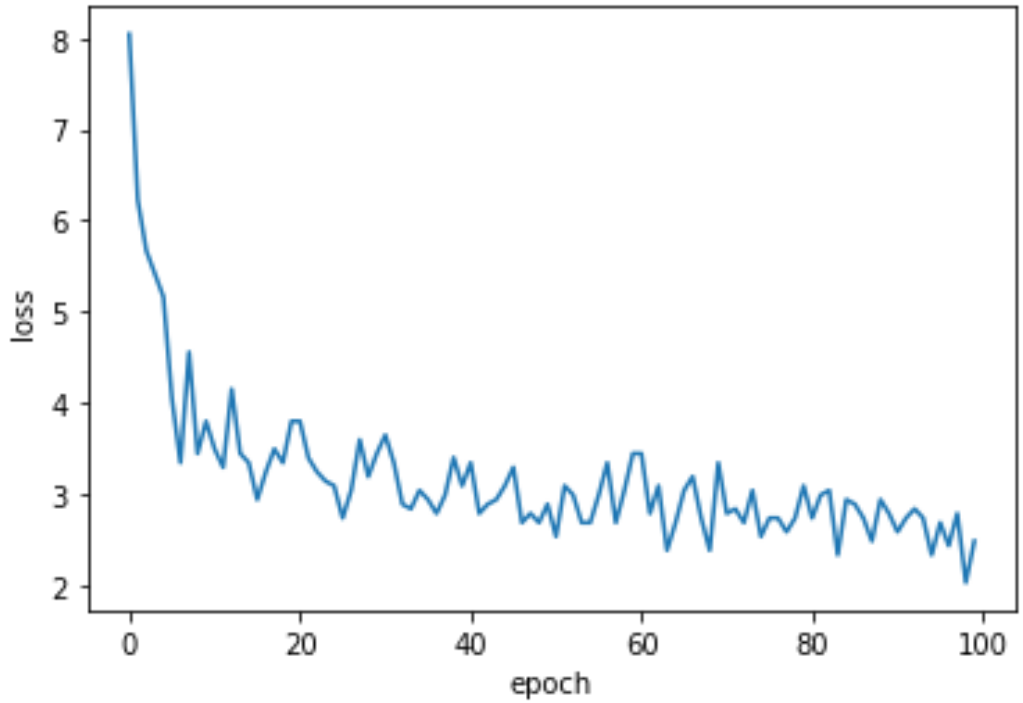
평균 손실 저장 후 시각화

```
def fit(self, x, y, epochs=100):
    self.w = np.ones(x.shape[1])           # 가중치를 초기화합니다.
    self.b = 0                             # 절편을 초기화합니다.
    for i in range(epochs):                # epochs만큼 반복합니다
        loss = 0
        indexes = np.random.permutation(np.arange(len(x)))
        for i in indexes:
            ...
            loss += -(y[i]*np.log(a)+(1-y[i])*np.log(1-a))
            # 에포크마다 평균 손실을 저장합니다
            self.losses.append(loss/len(y))
    ...

layer = SingleLayer()
layer.fit(x_train, y_train)
layer.score(x_test, y_test)
```

loss += -(y[i]*np.log(a)+(1-y[i])*np.log(1-a)) 로 손실 함수의 결과 값을 저장후
self.losses.append(loss/len(y)) 로 평균 손실을 저장한 후
layer.score(x_test, y_test) 로 시각화 하여 출력한다.

평균 손실 저장 후 시각화

```
plt.plot(layer.losses)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```



3. 신경망 시작하기

3.1 단일층 신경망 구현

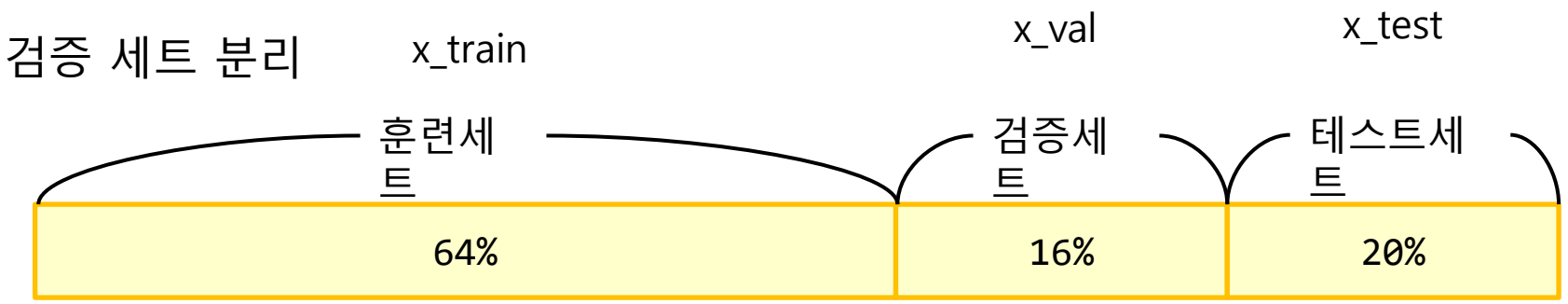
3.2 검증 세트 분리와 스케일링

3.3 과대적합과 과소적합

3.4 규제방법 구현

3.5 교차 검증 구현

"테스트 세트로 모델를 튜닝하면 실전에서 좋은 성능을 기대하기 어렵다"




```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()
x = cancer.data
y = cancer.target
x_train_all, x_test, y_train_all, y_test = train_test_split(x, y, stratify=y,
test_size=0.2, random_state=42)

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)
print(len(x_train), len(x_val))
```

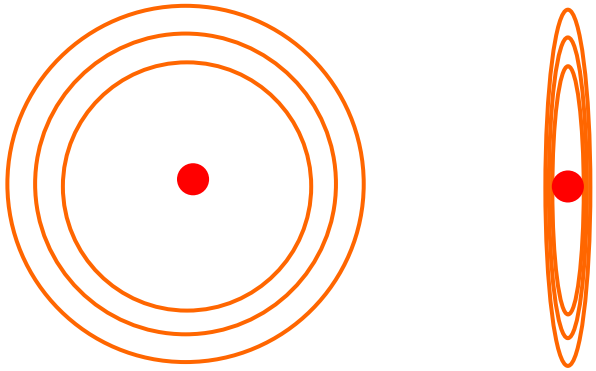
```
364 91
```

데이터 전처리와 특성의 스케일

$$w1 = w1-rate*(y\_hat-y)x1$$
$$w2 = w2-rate*(y\_hat-y)x2$$

	당도	무게	...
사과1	4	540	...
사과2	8	700	...
사과3	2	480	...

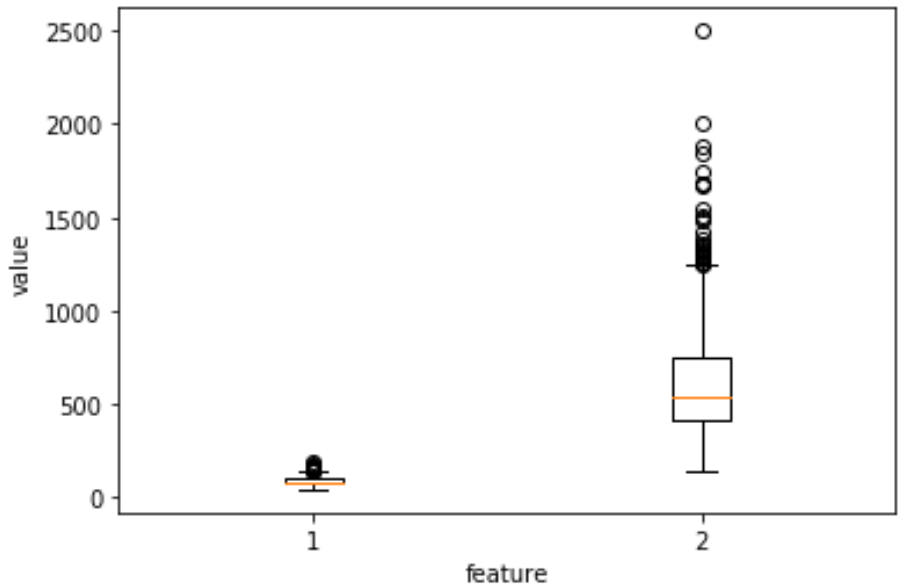
사과의 당도는 1~10이고 사과의 무게의 범위는 500~1000이다.
이런 경우 '두 특성의 스케일 차이가 크다'라고 말한다.



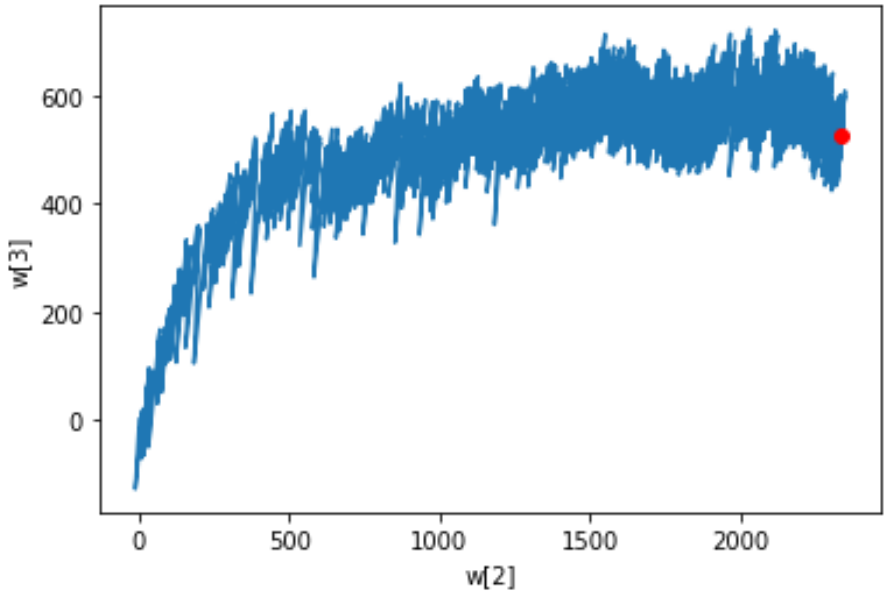
데이터 전처리와 특성의 스케일

```
print(cancer.feature_names[[2,3]])
plt.boxplot(x_train[:, 2:4])
plt.xlabel('feature')
plt.ylabel('value')
plt.show()
```

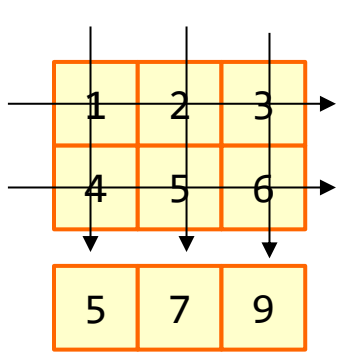
['mean perimeter' 'mean area']



```
w2 = []
w3 = []
for w in layer1.w_history:
    w2.append(w[2])
    w3.append(w[3])
plt.plot(w2, w3)
plt.plot(w2[-1], w3[-1], 'ro')
plt.xlabel('w[2]')
plt.ylabel('w[3]')
plt.show()
```

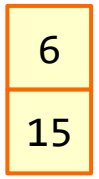


```
a = np.array([[1,2,3],
              [4,5,6]]

print(np.sum(a))         # 21
print(np.sum(a,axis=1)) # [6 15]
print(np.sum(a,axis=0)) # [5 7 9]
```

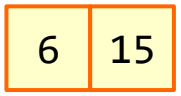


axis=0

(2, 3)



axis=1



(2,) => [6 15]

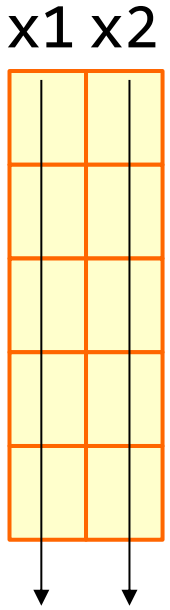
스케일 조정 : 평균 0, 표준편차 1 값이 된다.

$$z = \frac{x - \mu}{s}$$

표준화는 특성 값에서 평균을 빼고 표준 편차로 나누면 된다.

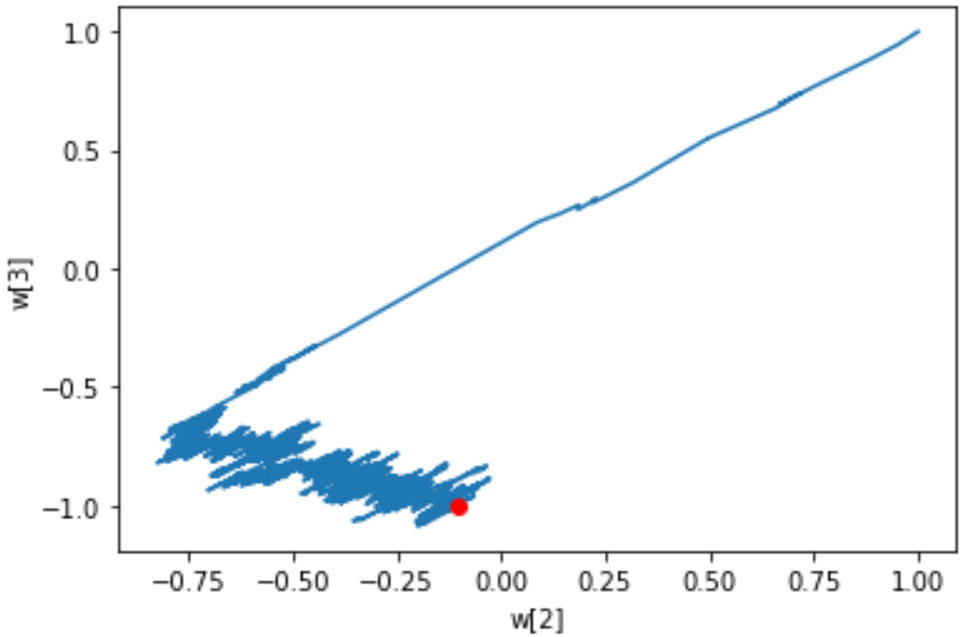
$$s = \sqrt{\frac{1}{m}\sum_{i=0}^{m}(x_i - \mu)^2}$$

표준 편차 공식




```
train_mean = np.mean(x_train, axis=0)
train_std = np.std(x_train, axis=0)
x_train_scaled = (x_train - train_mean) / train_std
```

```
w2 = []
w3 = []
for w in layer2.w_history:
    w2.append(w[2])
    w3.append(w[3])
plt.plot(w2, w3)
plt.plot(w2[-1], w3[-1], 'ro')
plt.xlabel('w[2]')
plt.ylabel('w[3]')
plt.show()
```



```
layer2.score(x_val, y_val)
```

0.37362637362637363

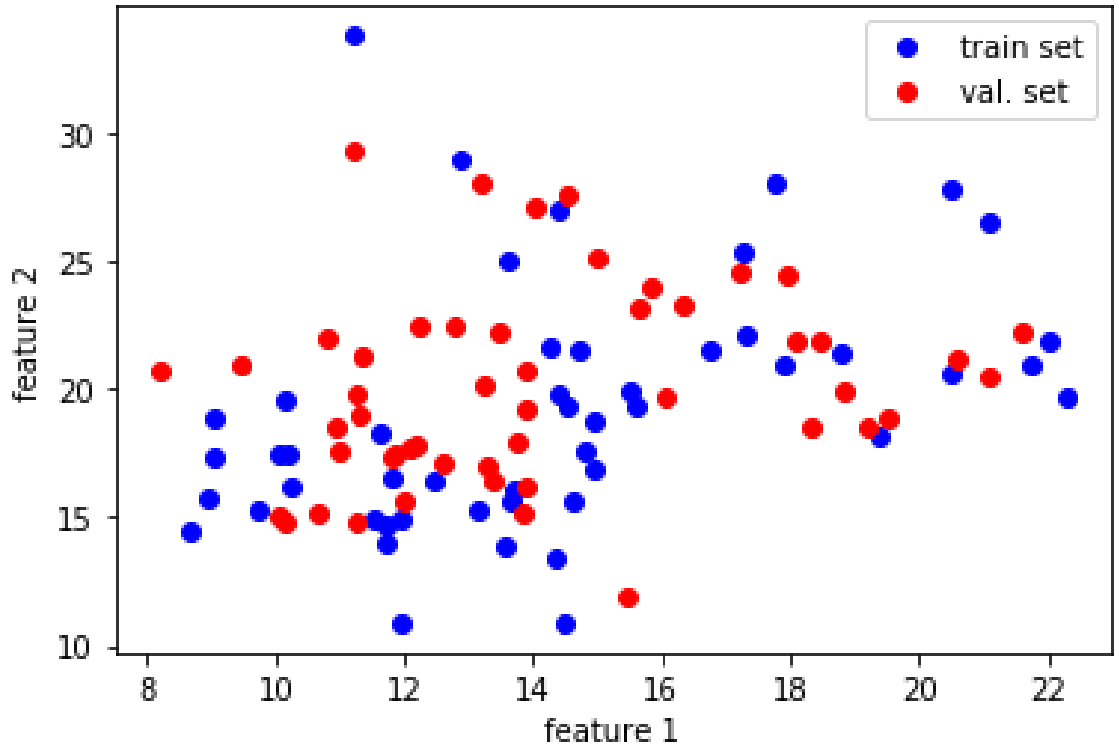
훈련 세트만 스케일을 조정하면 성능이 좋지않다.

검증세트도 표준화 전처리를 적용해야 한다.

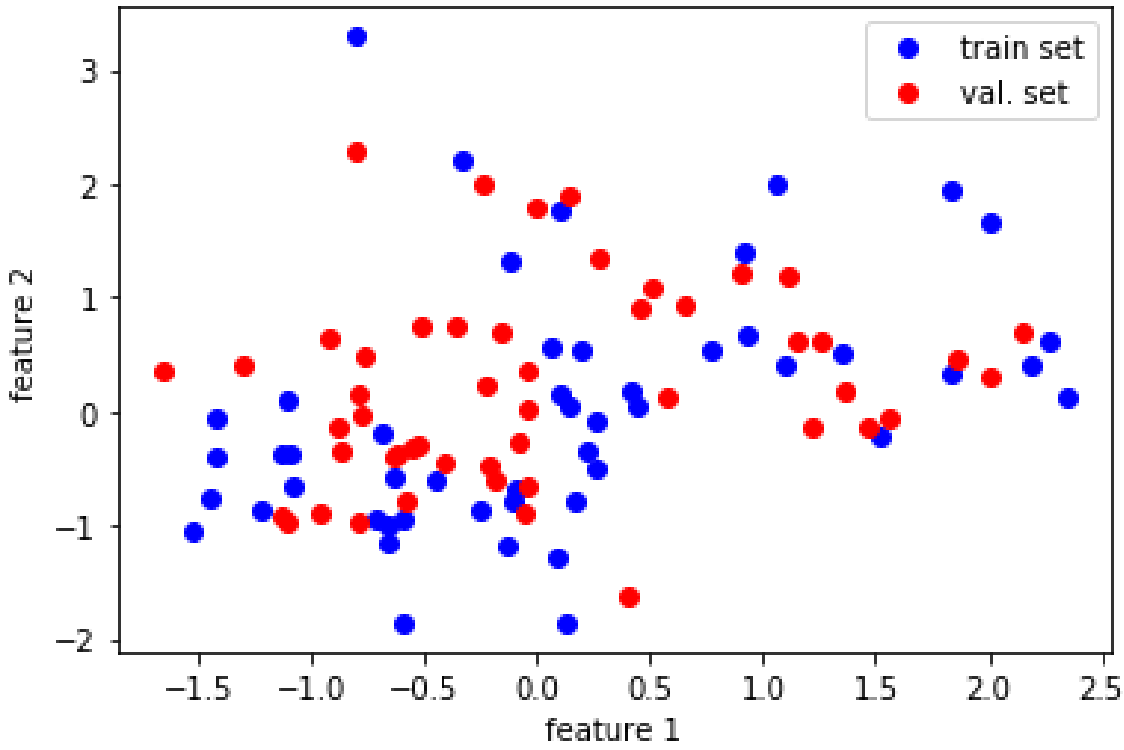
```
val_mean = np.mean(x_val, axis=0)
val_std = np.std(x_val, axis=0)
x_val_scaled = (x_val - val_mean) / val_std
layer2.score(x_val_scaled, y_val)
```

0.967032967032967

```
plt.plot(x_train[:50, 0], x_train[:50, 1], 'bo')
plt.plot(x_val[:50, 0], x_val[:50, 1], 'ro')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.legend(['train set', 'val. set'])
plt.show()
```

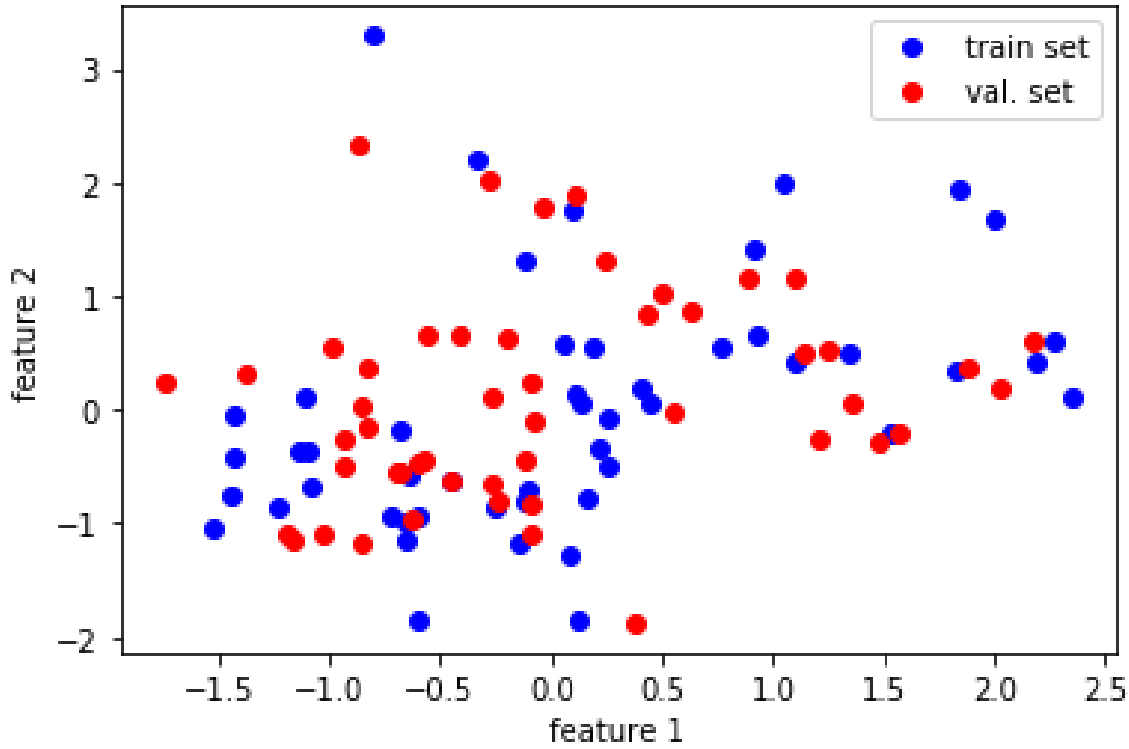


```
x_val_scaled = (x_val - train_mean) / train_std
plt.plot(x_train_scaled[:50, 0], x_train_scaled[:50, 1], 'bo')
plt.plot(x_val_scaled[:50, 0], x_val_scaled[:50, 1], 'ro')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.legend(['train set', 'val. set'])
plt.show()
```



훈련 세트의 평균,
표준 편차를 이용하여
검증 세트를 변환
하면
문제가 해결된다.

```
plt.plot(x_train_scaled[:50, 0], x_train_scaled[:50, 1], 'bo')
plt.plot(x_val_scaled[:50, 0], x_val_scaled[:50, 1], 'ro')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.legend(['train set', 'val. set'])
plt.show()
```



훈련 세트와 검증 세트의 거리가 그대로 유지 되어야 한다.

하지만 지금은 거리가 달라졌다.

이유는

훈련 세트과 검증 세트가 각각 다른 비율로 전처리 되었기 때문이다.

3. 신경망 시작하기

3.1 단일층 신경망 구현

3.2 검증 세트 분리와 스케일링

3.3 과대적합과 과소적합

3.4 규제방법 구현

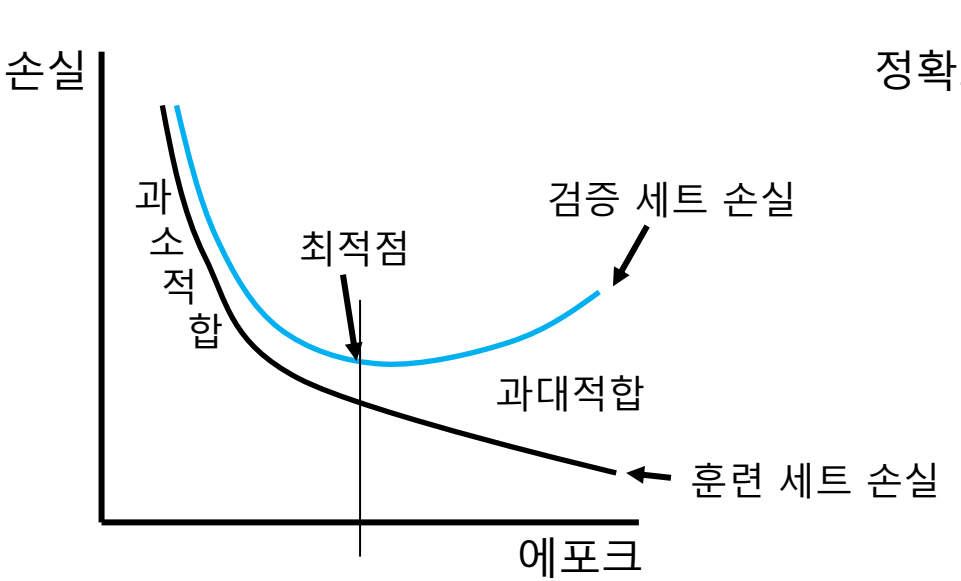
3.5 교차 검증 구현

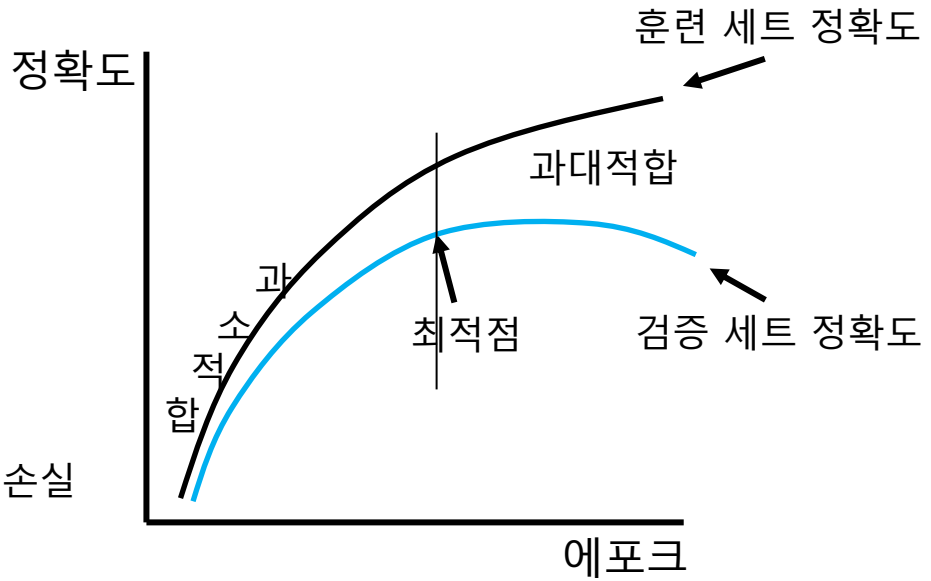
과대적합 :

훈련 세트에서는 좋은 성능을 내지만 검증 세트에서는 낮은 성능을 내는 경우

과소적합 :

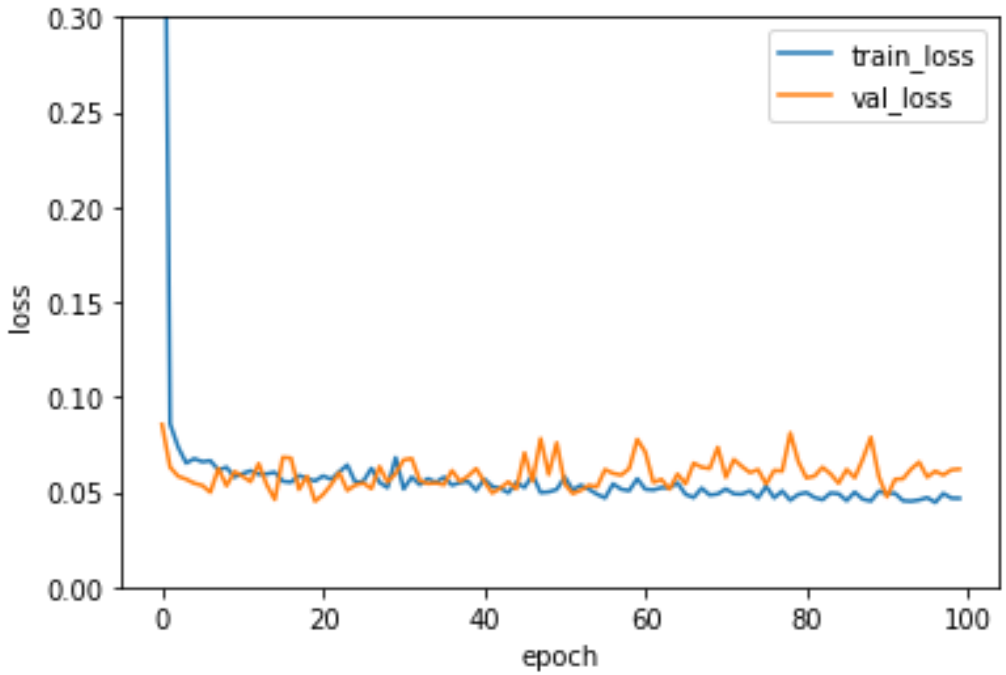
훈련 세트와 검증세트의 성능에는 차이가 크지 않지만 모두 낮은 성능을 내는 경우





과대적합 구간

```
layer3 = SingleLayer()
layer3.fit(x_train_scaled, y_train, x_val=x_val_scaled, y_val=y_val)
plt.ylim(0, 0.3)
plt.plot(layer3.losses)
plt.plot(layer3.val_losses)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



훈련 조기 종료

```
layer4 = SingleLayer()
layer4.fit(x_train_scaled, y_train, epochs=20)
layer4.score(x_val_scaled, y_val)
```

0.978021978021978

3. 신경망 시작하기

3.1 단일층 신경망 구현

3.2 검증 세트 분리와 스케일링

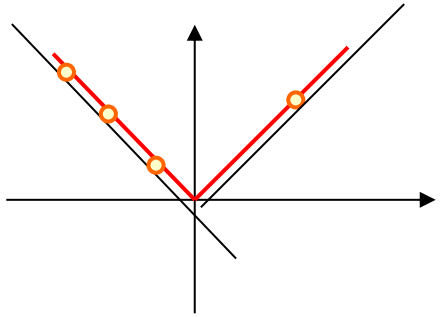
3.3 과대적합과 과소적합

3.4 규제방법 구현

3.5 교차 검증 구현

L1 규제는 손실 함수에 가중치의 절대값인 L1 노름(norm)을 추가한다.

$$||w||_1 = \sum_{i=1}^{n} |w_i|$$



$$w_1 x_1 + w_2 x_2 + \cdots + b$$

$$\alpha(|w_1| + |w_2| \ldots)$$

절대값을 미분하면 부호가 남는다.

$$L = -(ylog(a) + (1-y)\log(1-a))$$

L1 노름을 그냥 더하지 않고 규제의 양을 조절하는 파라미터 α 를 곱한 후 더한다.

$$L = -(ylog(a) + (1-y)\log(1-a)) + \alpha \sum_{i=1}^{n} |w_i|$$

L1 규제는 손실 함수에 가중치의 절대값인 L1 노름(norm)을 추가한다.

$$\frac{\partial}{\partial w_1} L = (a - y)x_1 + \alpha * sign(w_1)$$

$$w = w - \eta \frac{\partial L}{\partial w} = w - \eta((a - y)x + \alpha * sign(w))$$

파이썬으로 작성된 L1 규제 적용된 오차 역전파 구현

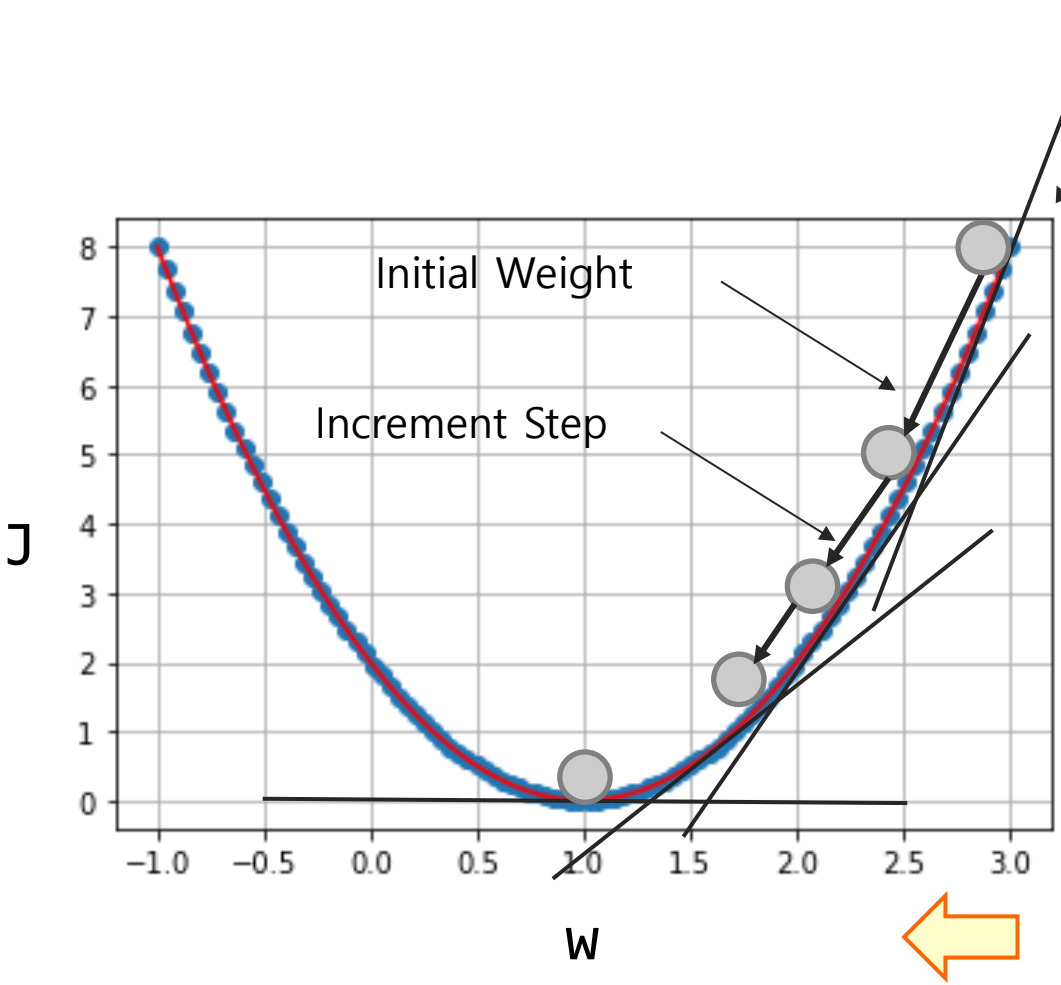
```
w_grad += alpha * np.sign(w)
```

회귀 모델에 L1 규제를 추가한 것을 라쏘 모델이라 한다.

Gradient descent algorithm

$x = 2, y = 2, b = 0$

$y\_hat = xw + b$



$$w = w - \eta((y_{hat} - y)x + \alpha * sign(w)$$

$0.01 \quad 8 \quad 0.0$

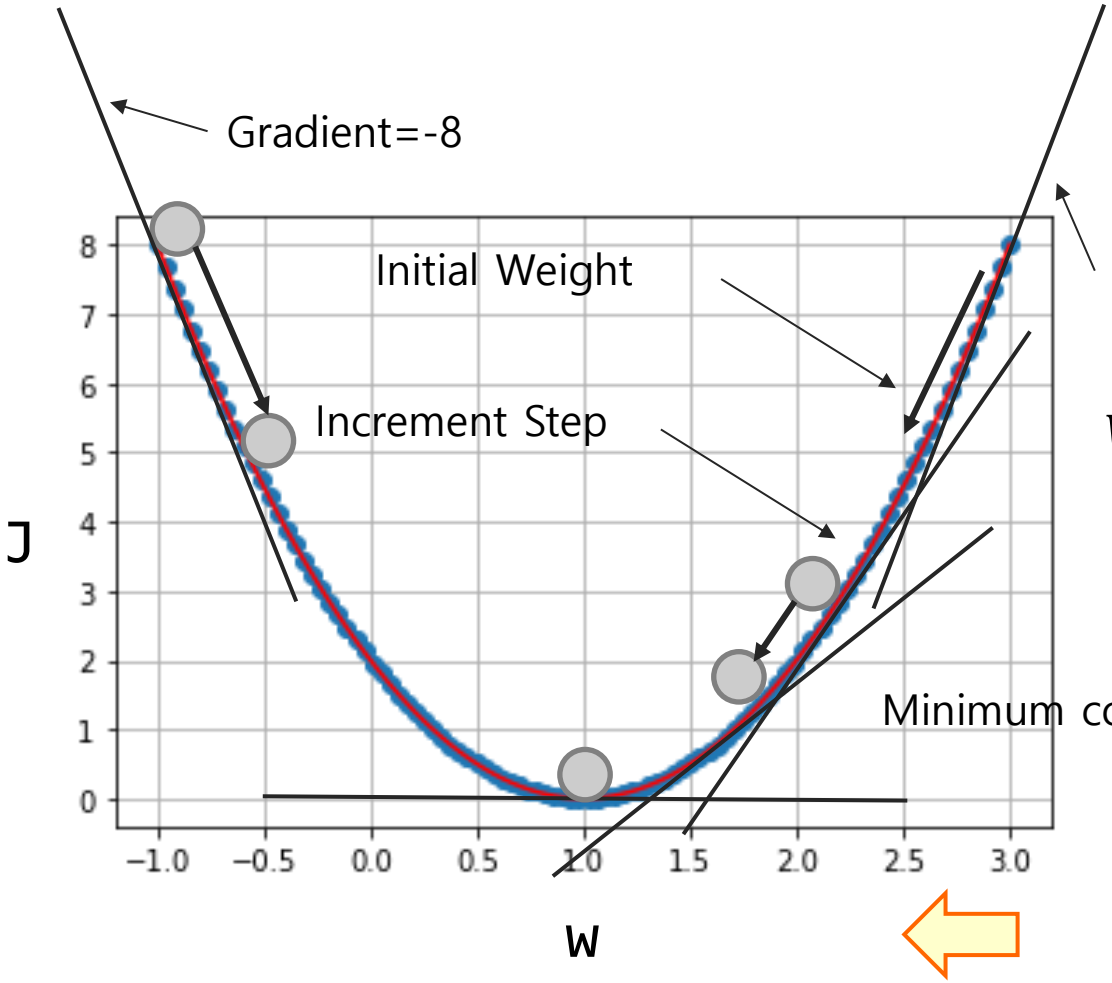
$w = 3 - 0.08$

$w = 2.92$

$w = 3 - 0.081$

$w = 2.919$

Gradient descent algorithm



$$x = 2, \qquad y = 2$$

$$a = xw + b$$

$$w = w - \eta((a - y)x + \alpha * sign(w))$$

(0.01 over η, 0.1 over α)

$$w = -1 - (-0.08)$$
$$w = -0.92$$

$$w = -1 - 0.01 * (-8 - 0.1)$$
$$w = -0.919$$

L2 규제는 손실 함수에 가중치에 대한 L2 노름(norm)의 제곱을 더한다.

$$\left|\left|w\right|\right|_2 = \sum_{i=1}^{n} |w_i|^2$$

$$\frac{1}{2}\alpha(w_1^2 + w_2^2 + \cdots)$$

$$L = -(ylog(a) + (1-y)\log(1-a))$$

L1 노름을 그냥 더하지 않고 규제의 양을 조절하는 파라미터 α 를 곱한 후 더한다.

$$L = -(ylog(a) + (1-y)\log(1-a)) + \frac{1}{2}\alpha \sum_{i=1}^{n} |w_i|^2$$

L2 규제는 손실 함수에 가중치에 대한 L2 노름(norm)의 제곱을 더한다.

$$\frac{\partial}{\partial w}L = (a - y)x + \alpha * w$$

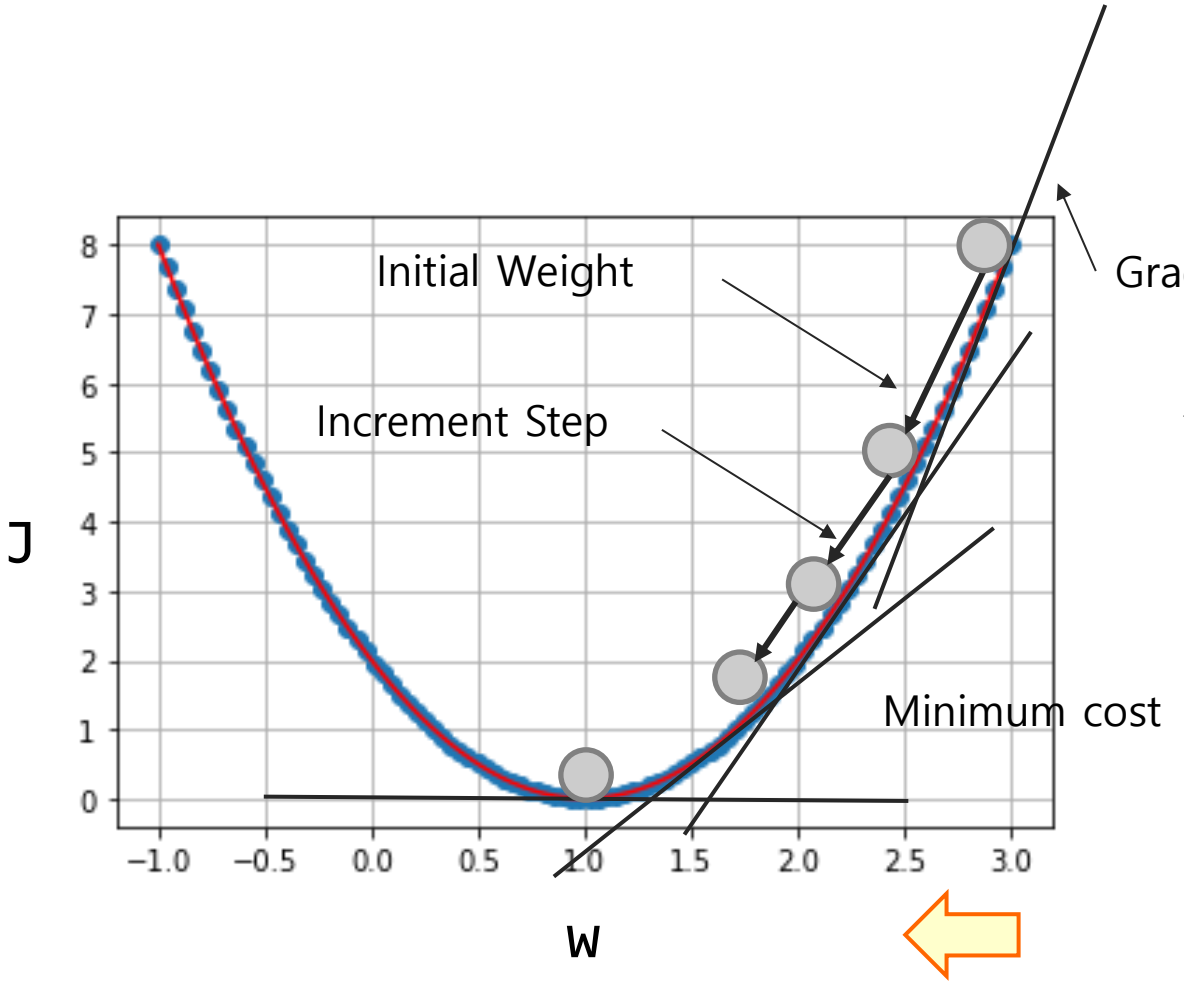
$$w = w - \eta\frac{\partial L}{\partial w} = w - \eta((a - y)x + \alpha * w)$$

파이썬으로 작성된 L2 규제 적용된 오차 역전파 구현

```
w_grad += alpha * w
```

회귀 모델에 L2 규제를 추가한 것을 릿지 모델이라 한다.

Gradient descent algorithm



$$w = w - \eta((a - y)x + \alpha * w)$$

(η: 0.01, α: 0.1)

$$w = 3 - 0.08$$

$$w = 2.92$$

$$w = 3 - 0.083$$

$$w = 2.917$$

3. 신경망 시작하기

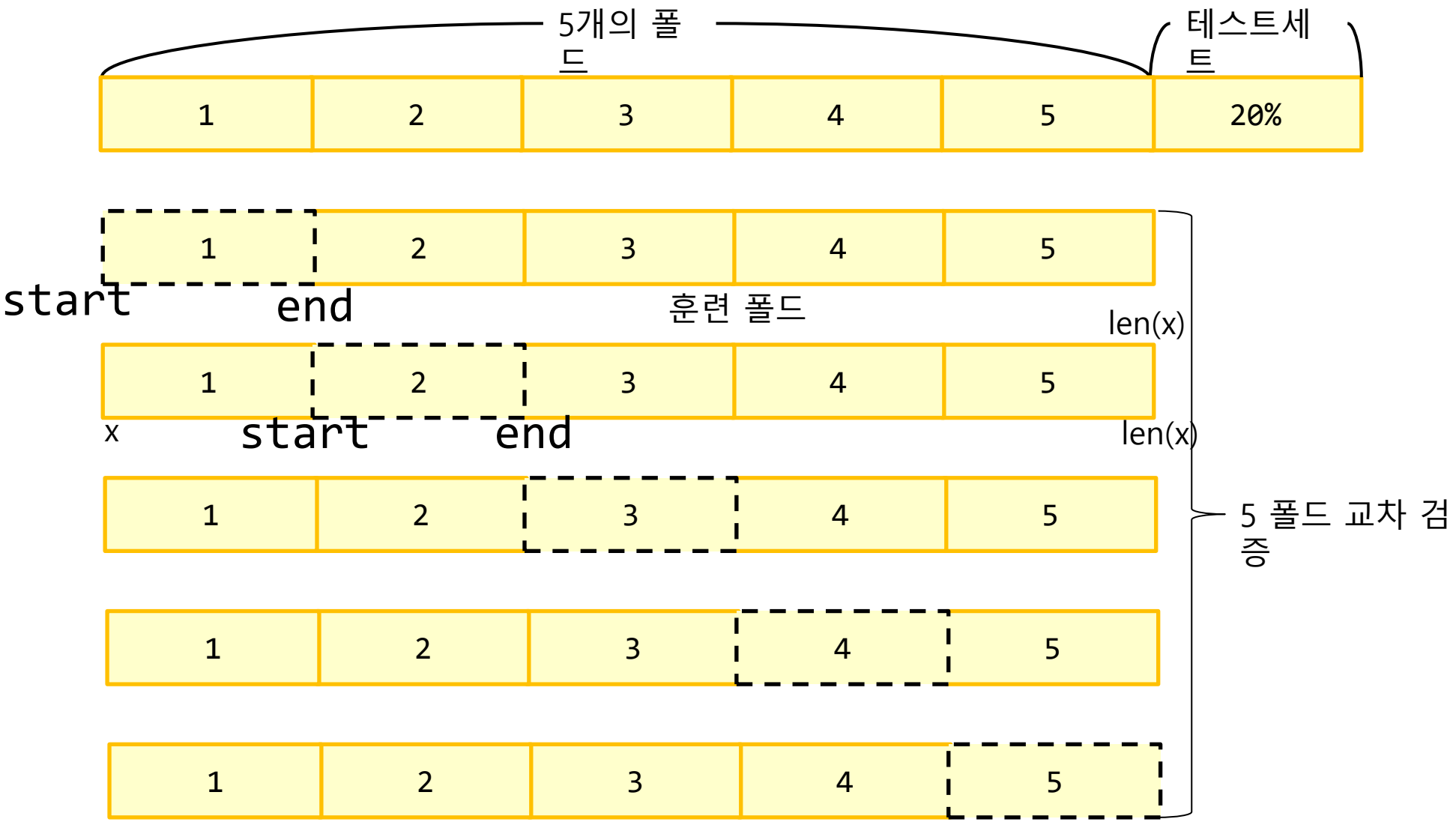
3.1 단일층 신경망 구현

3.2 검증 세트 분리와 스케일링

3.3 과대적합과 과소적합

3.4 규제방법 구현

3.5 교차 검증 구현

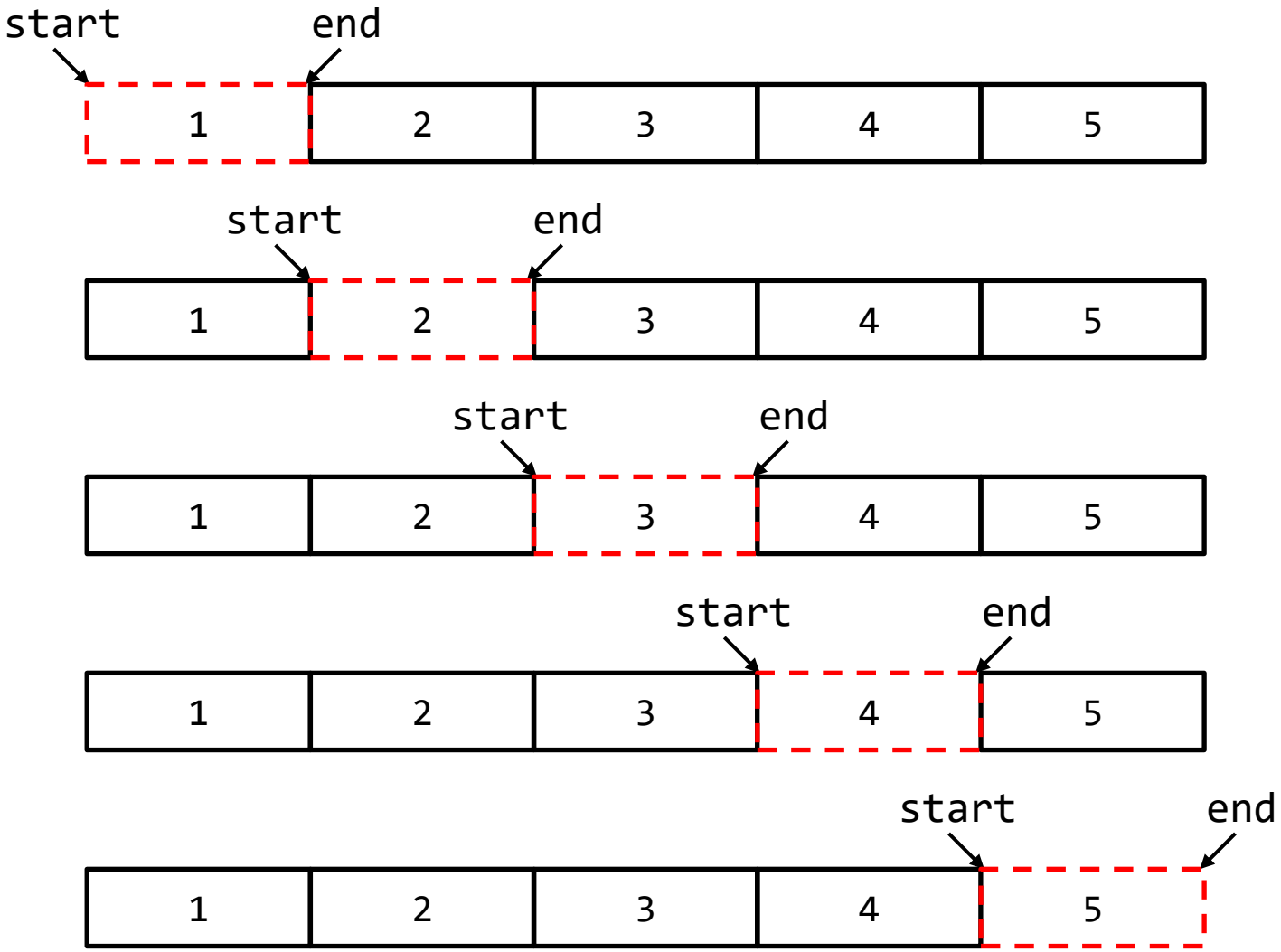

5개의 폴드
테스트세트
1
2
3
4
5
20%
start
end
훈련 폴드
len(x)
x
start
end
len(x)
5 폴드 교차 검증

교차 검증 과정

1. 훈련 세트를 k개의 폴드(fold)로 나눈다.
2. 첫 번째 폴드를 검증 세트로 사용하고 나머지 폴드(k-1개)를 훈련 세트로 사용 한다.
3. 모델을 훈련한 다음에 검증 세트로 평가 한다.
4. 차례대로 다음 폴드를 검증 세트로 사용하여 반복한다.
5. k개의 검증 세트로 k번 성능을 평가한 후 계산된 성능의 평균을 내어 최종 성능을 계산한다.

```
validation_scores = []
k = 10
bins = len(x_train_all) // k

for i in range(k):
    start = i*bins
    end = (i+1)*bins
    val_fold = x_train_all[start:end]
    val_target = y_train_all[start:end]

    train_index = list(range(0, start))+list(range(end, len(x_train)))
    train_fold = x_train_all[train_index]
    train_target = y_train_all[train_index]
    train_mean = np.mean(train_fold, axis=0)
    train_std = np.std(train_fold, axis=0)
    train_fold_scaled = (train_fold - train_mean) / train_std
    val_fold_scaled = (val_fold - train_mean) / train_std

    lyr = SingleLayer(l2=0.01)
    lyr.fit(train_fold_scaled, train_target, epochs=50)
    score = lyr.score(val_fold_scaled, val_target)
    validation_scores.append(score)

print(np.mean(validation_scores))
```

0.9711111111111113


start
end
1
2
3
4
5
start
end
1
2
3
4
5
start
end
1
2
3
4
5
start
end
1
2
3
4
5
start
end
1
2
3
4
5

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

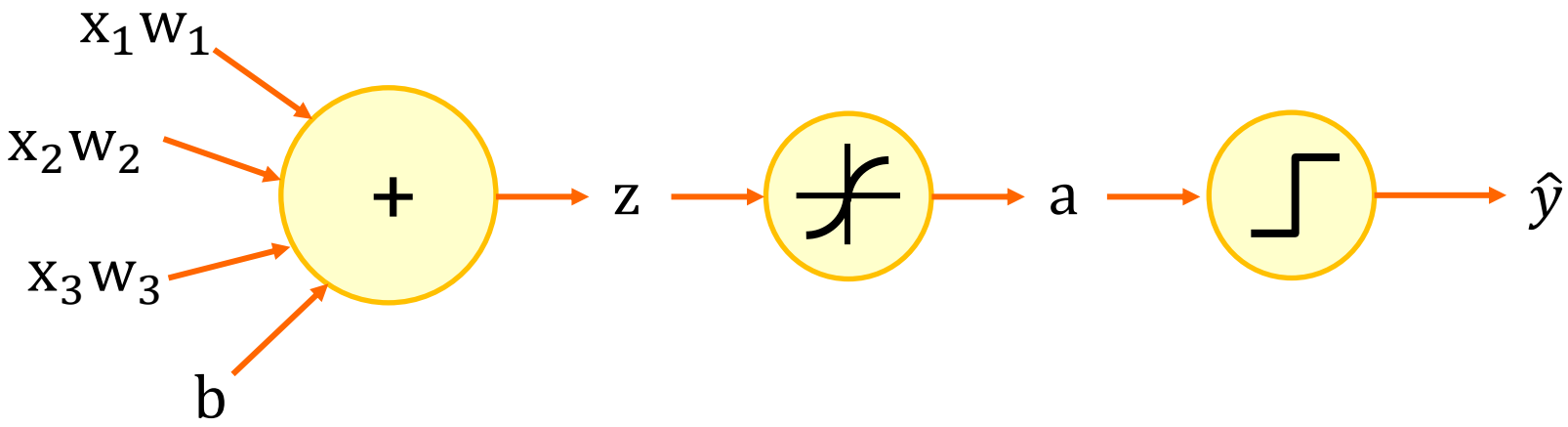
4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

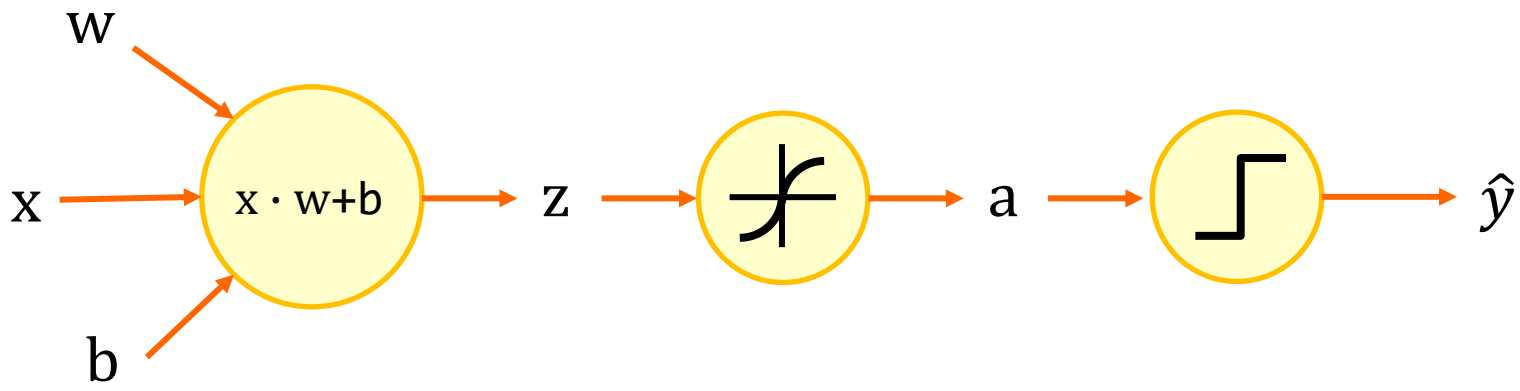
4.7 케라스를 이용한 다층신경망의 다양한 구현




```
def forpass(self, x):
    z = np.sum(x * self.w) + self.b
    return z
```

넘파이의 원소별 곱셈

$$x = [x_1, x_1, \cdots, x_n]$$
$$w = [w_1, w, \cdots, w_n]$$
$$x * w = [x_1 * w_1, x_2 * w_2, \cdots, x_n * w_1]$$

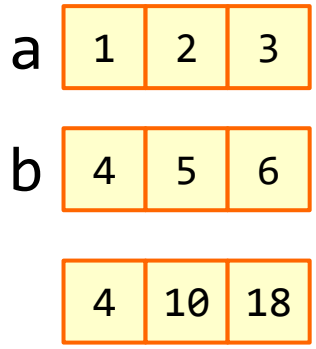


점 곱을 행렬 곱셈으로 표현

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

```
z = np.dot(x , self.w) + self.b
```

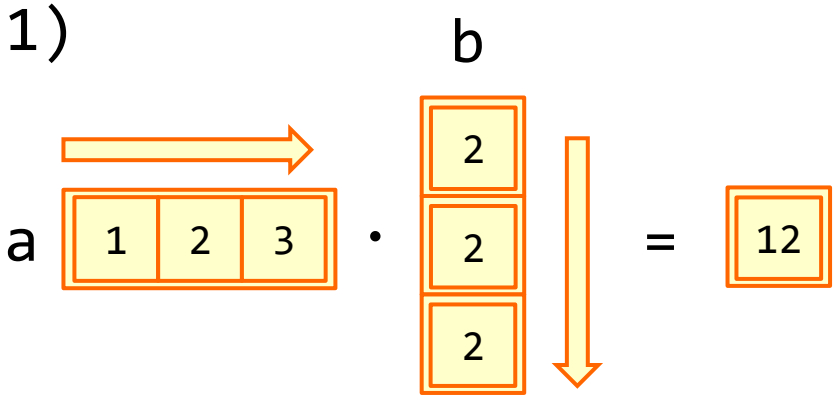
```
import numpy as np
a = np.array([1,2,3])
b = np.array([4,5,6])
# c = np.sum(a*b)
c = np.dot(a,b)
print(c)
```



32

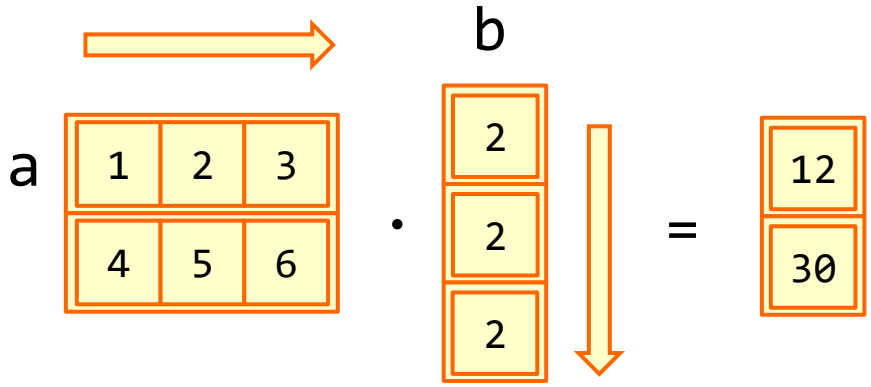
(3,)(3,)=>()

```
a = np.array([[1,2,3]]) # (1,3)
print(len(a))
print(a.shape)
b = np.array([[2],[2],[2]]) # (3,1)
print(len(b))
print(b.shape)
c = np.dot(a,b)
print(c.ndim)  # 2
print(c.shape) # (1,1)
print(c)       # [[12]]
```



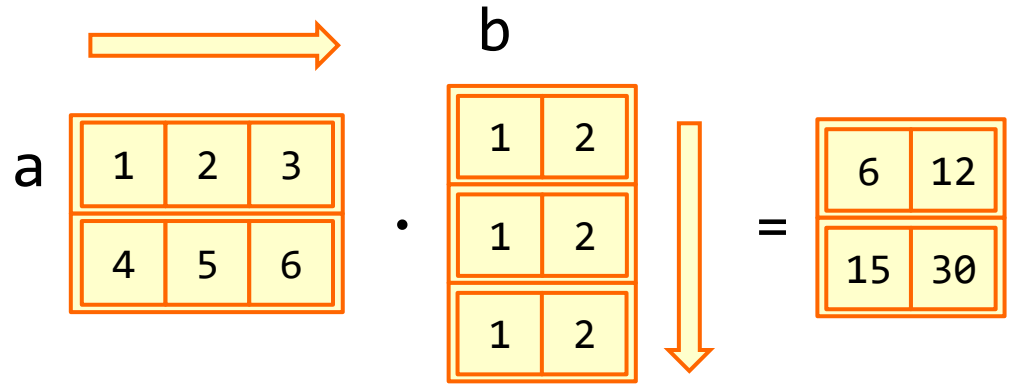
(1,3)(3,1)=>(1,1)

```
import numpy as np

a = np.array([[1,2,3],
              [4,5,6]]) # (2,3)
print(a.shape)
b = np.array([[2],[2],[2]]) # (3,1)
print(b.shape)
c = np.dot(a,b)    # (2,3)(3,1)
print(c.shape)
print(c.ndim)
print(c)
```



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

```
import numpy as np

a = np.array([[1,2,3],
              [4,5,6]]) # (2,3)
print(a.shape)
b = np.array([[1,2],
              [1,2],
              [1,2]]) # (3,2)
print(b.shape)
c = np.dot(a,b)    # (2,3)(3,2)
print(c.shape)
print(c.ndim)
print(c)
```



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

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

$$z = x_1 w_1 + x_2 w_2 + x_3 w_3 \ldots x_{30} w_{30} + b$$

$$XW = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ & \vdots & \\ x_1^{(m)} & x_2^{(m)} & x_3^{(m)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_1^{(1)} w_1 + x_2^{(1)} w_2 + x_3^{(1)} w_3 \\ x_1^{(2)} w_1 + x_2^{(2)} w_2 + x_3^{(2)} w_3 \\ \vdots \\ x_1^{(m)} w_1 + x_2^{(m)} w_2 + x_3^{(m)} w_3 \end{bmatrix}$$

(364, 30)(30, 1) => (364, 1)

행렬곱 가능과 곱 결과 크기

$$(m, n) \cdot (n, k) = (m, k)$$

첫 번째 행렬의 열(n)과 두 번째 행렬의 행(n)의 크기는 반드시 같아야 한다.

곱 결과의 크기는 첫 번째 행렬의 행(m)과 두 번째 행렬의 열(k)의 크기가 된다.

정방향 계산을 행렬 곱셉으로 표현

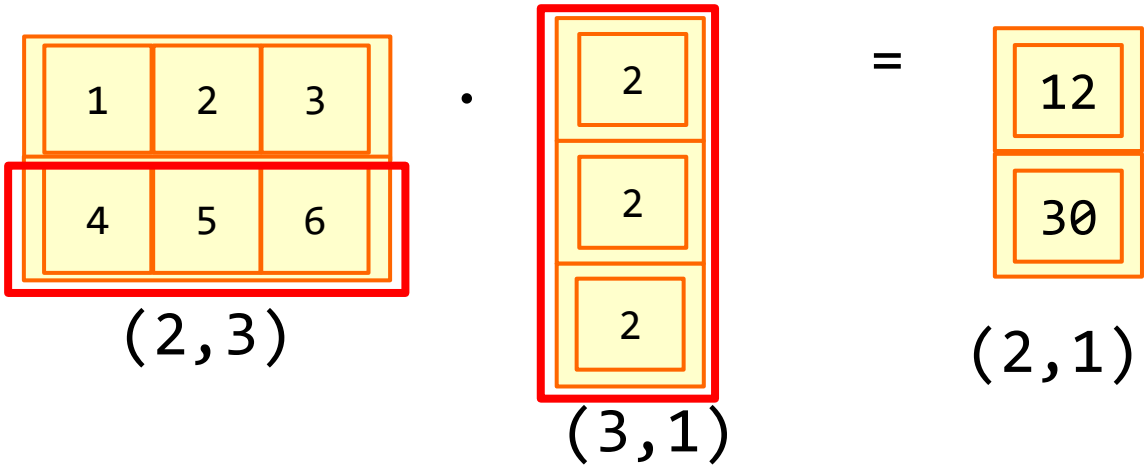
$$XW = \begin{bmatrix} x_1^{(1)} & \cdots & x_{30}^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(364)} & \cdots & x_{30}^{(364)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{30} \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(364)} \end{bmatrix}$$

$$= \begin{bmatrix} x_1^{(1)} & \cdots & x_{30}^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(364)} & \cdots & x_{30}^{(364)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{30} \end{bmatrix} + b = \begin{bmatrix} x_1^{(1)}w_1 + x_2^{(1)}w_2 + \cdots + x_{30}^{(1)}w_{30} + b \\ \vdots \\ \vdots \\ x_1^{(364)}w_1 + x_2^{(364)}w_2 + \cdots + x_{30}^{(364)}w_{30} + b \end{bmatrix}$$

```
Z = (364, 1)
A = sigmoid(Z)
L = Loss(A)/364
```

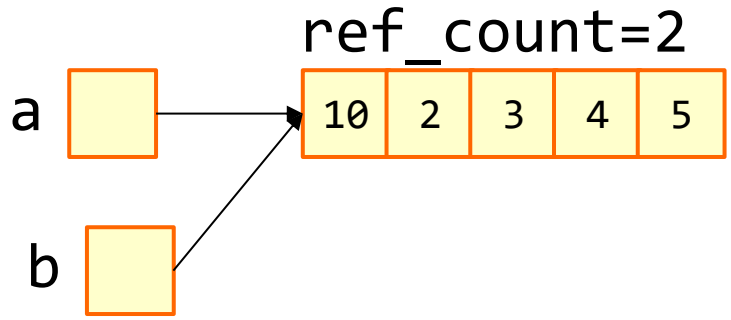
np.dot(a,b)

2차원 배열에서의 행렬의 곱

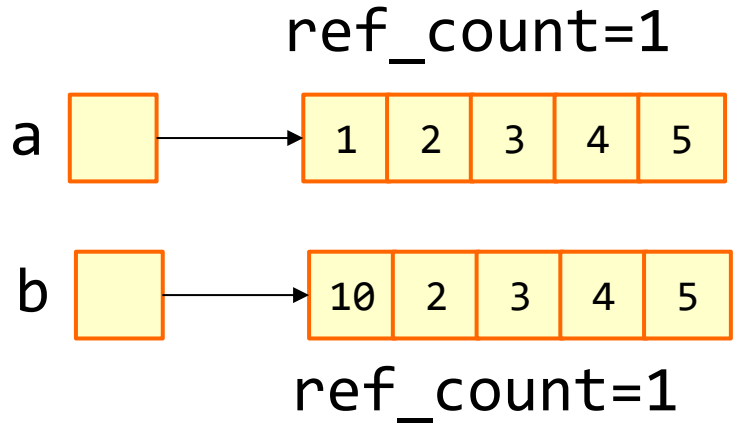


(2,3)(3,1)

```
a = np.array([1,2,3,4,5])
b = a
b[0] = 10
a
```




```
a = np.array([1,2,3,4,5])
b = a.copy()
```



(30,364)(364,1)

그레디언트 계산

$$X^T E = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(364)} \\ x_2^{(1)} & \cdots & x_2^{(364)} \\ \vdots & \cdots & \vdots \\ x_{30}^{(1)} & \cdots & x_{30}^{(364)} \end{bmatrix} \begin{bmatrix} e^{(1)} \\ e^{(2)} \\ \vdots \\ e^{(364)} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_{30} \end{bmatrix}$$

그레디언트는 X와 E의 행렬곱이다.
그러나 벡터 연산에서 X의 크기는 (364, 30) 이고
E는 (364, 1) 이므로 행렬의 곱을 할 수 없다.

(364,30)(364,1)

이때 X를 전치하면 행과 열이 바뀌므로
X^T (30, 364) 가 되므로 (30,364)(364,1)

$X^T E$ 는 (30, 364) •(364, 1) 이므로 곱이 가능하고
결과는 (30, 1) 이 되므로 그레디언트와 같은 행렬을 구할 수 있다.

$$W = W - \eta * X^T E$$

$(30, 364)$ $(364, 1)$

$(30,1)$

(364,30)(30,1)

$Z = XW + b$

$X \Rightarrow (364, 30)$

$W \Rightarrow (30, 1)$

$Z \Rightarrow (364, 1)$

$a \Rightarrow (364, 1)$

$err \Rightarrow (a - y) \Rightarrow (364, 1)$

$(30,364)(364,1) \Rightarrow (30,1)$

행렬곱의 미분(MatMul)

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

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

$$\frac{\partial}{\partial w} XW + b$$

$$= X$$

$$\frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w} = X^T E$$

Chain Rule 사용

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

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

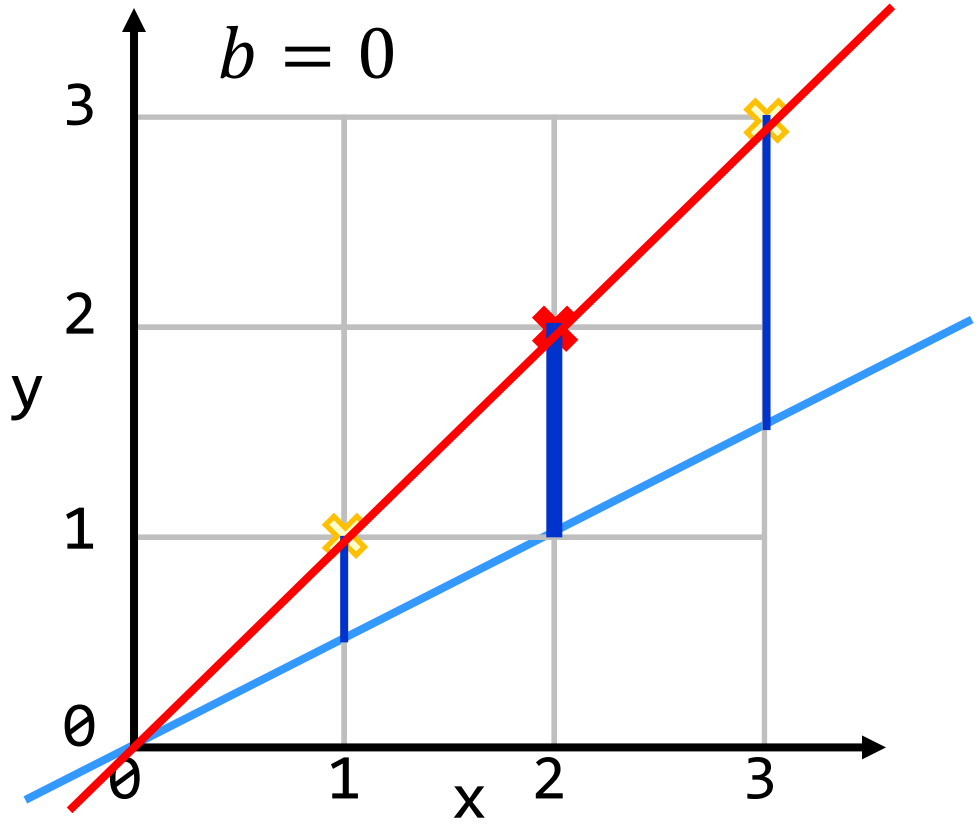
$$\frac{\partial}{\partial w} J(w,b) = \frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$$

전체식을 W로 편미분한 경우
사라진 W의 자리에는 뒤에서 날라온
미분값인 E를 쓰고
남아 있는 X는 전치하여
행렬 곱을 한다.

$\hat{y} = wx + b$

$w = 0.5$

$b = 0$



(0.5+1+1.5)^2

(0.25+1+2.25)/6 => 0.58

se

$\frac{1}{2}(\hat{y} - y)^2$ 확률적 경사하강

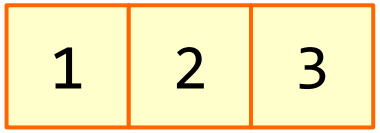
0.5

mse

$\frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$ 배치 경사하강

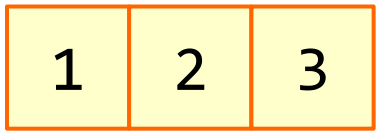
0.58

전치

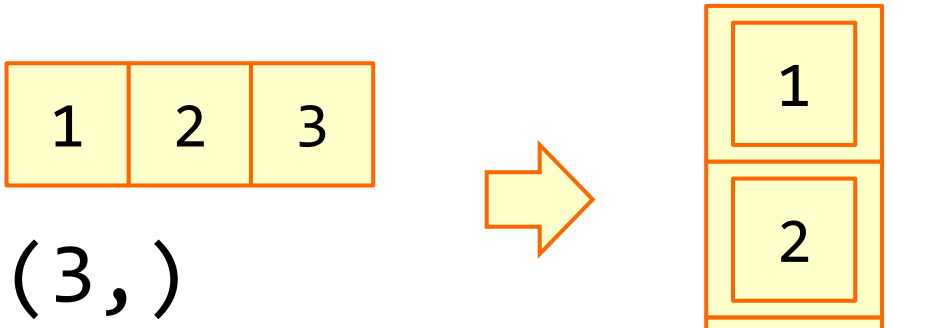


(3,)

```
a = np.array([1,2,3])
a.shape
a.T
```



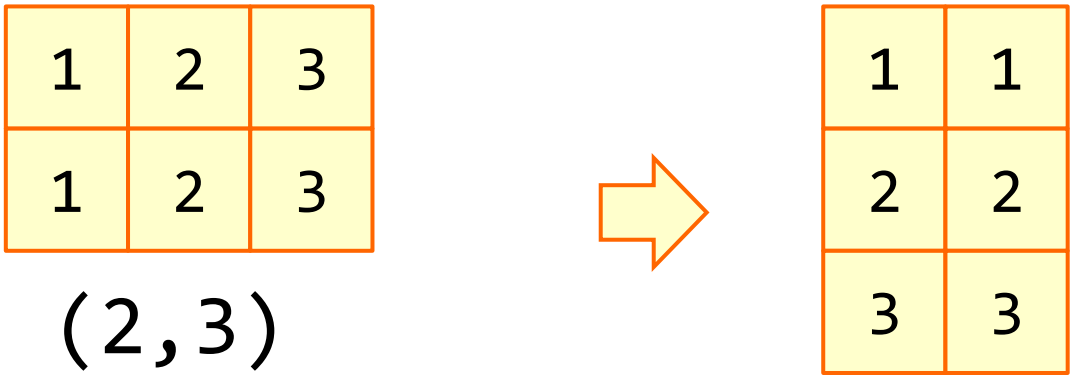
(3,)




```
a = np.array([1,2,3])
a.shape
a.T
```

```
b = np.reshape(a, (3,1))
b.shape

b = np.reshape(a, (-1,1))
b.shape
```

전치



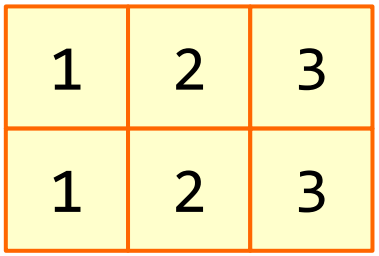

```
a = np.array([[1,2,3],
              [1,2,3]])
a.shape
a = a.T
```

(2,3)

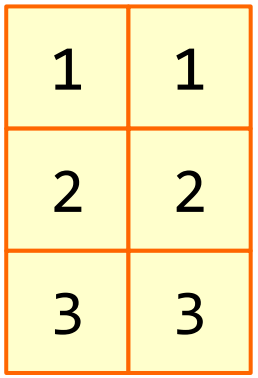
(3,2)

```
b = np.transpose( a, (1,0))
print(b)
```

전치



(2,3)

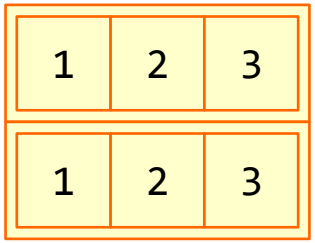


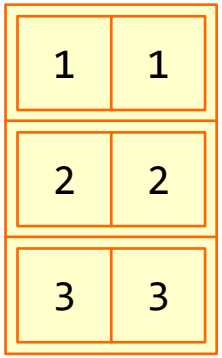
(3,2)

```
a = np.array([[1,2,3],
              [1,2,3]])
a.shape
a = a.T
```

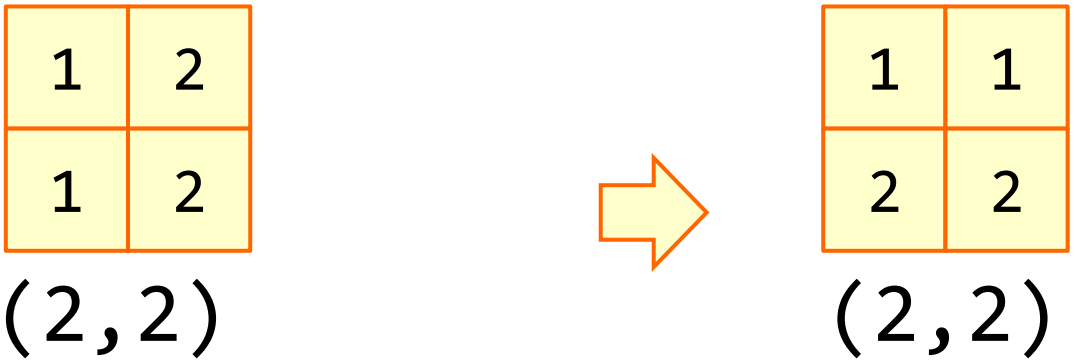
(2,3)

(3,2)

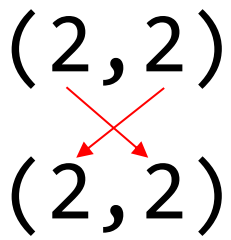


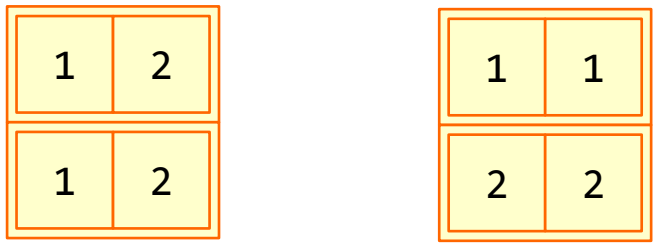


전치




```
a = np.array([[1,2],[1,2]])
a.shape
a.T
```

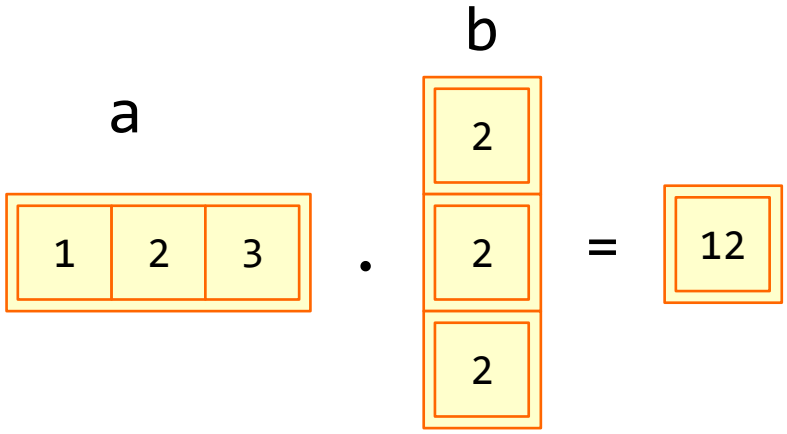




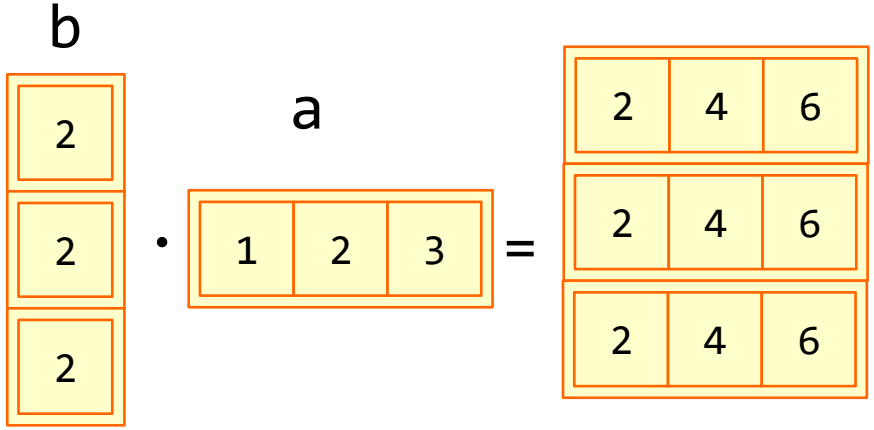
행렬의 교환 법칙

```
import numpy as np

a = np.array([[1,2,3]]) # (1,3)
b = np.array([[2],[2],[2]]) # (3,1)

c = np.dot(a,b)   # (1,3)(3,1) => (1,1)
print(c.shape)
print(c)

c = np.dot(b,a)  # (3,1)(1,3) => (3,3)
print(c.shape)
print(c)
```

3*4==4*3



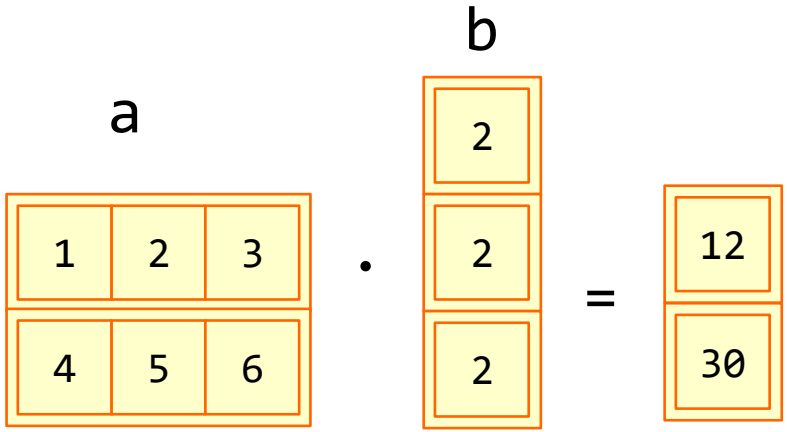
(1,3)(3,1)=>(1,1)



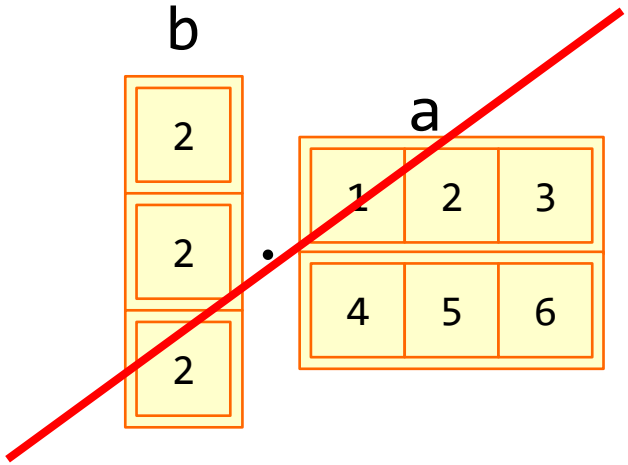
(3,1)(1,3)=>(3,3)

행렬의 교환 법칙

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]]) # (2,3)
b = np.array([[2],[2],[2]]) # (3,1)

c = np.dot(a,b)   # (2,3)(3,1) => (2,1)
print(c.shape)

c = np.dot(b,a)  # (3,1)(2,3)
print(c.shape)
```



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



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

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

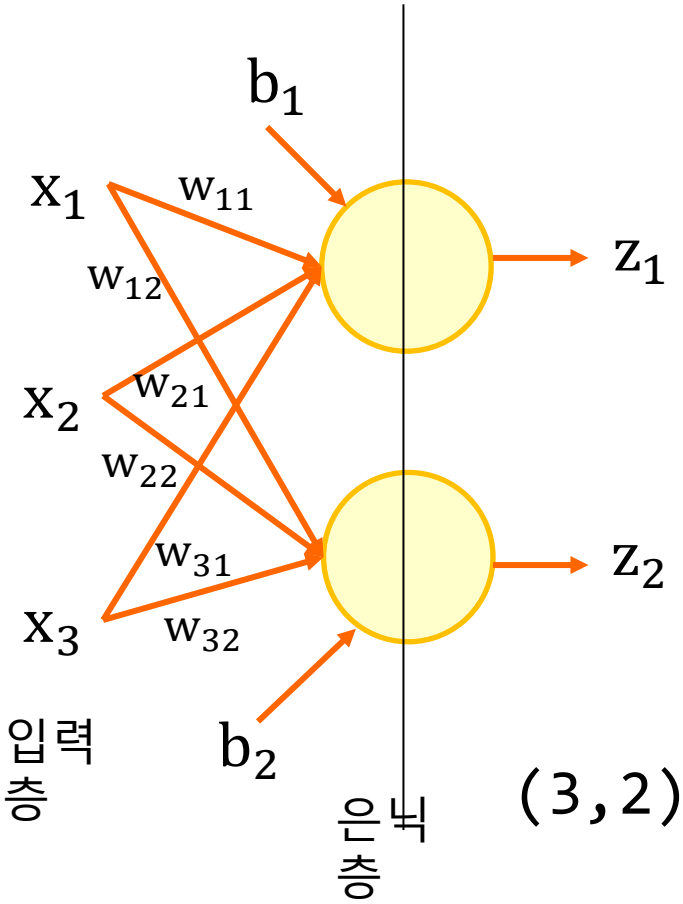
4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

하나의 층에 여러개의 뉴런 사용



playground.tensorflow.org

$$XW_1 + b_1 = z_1$$
$$XW_2 + b_2 = z_2$$

$$XW + b = Z$$

(3,2) => (특성수,뉴런수) => (입력수,출력수)

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

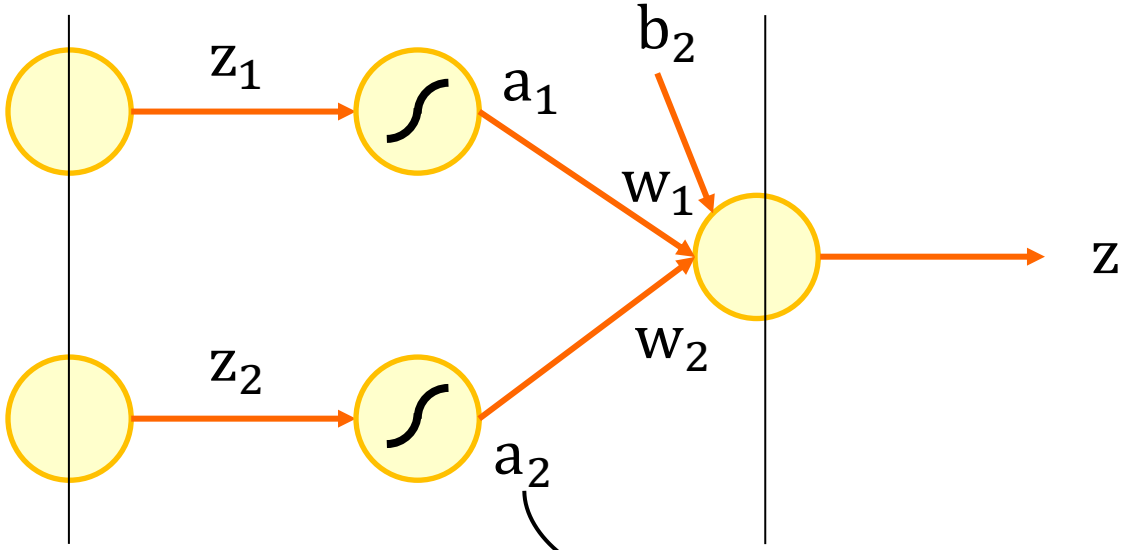
$$x_1w_{12} + x_2w_{22} + x_3w_{32} + b_2 = z_2$$

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

(m,n)(n,2) => (m,2)

하나의 층에 여러개의 뉴런 사용

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



활성화 출력 : 비선형 함수를 사용해야 한다.

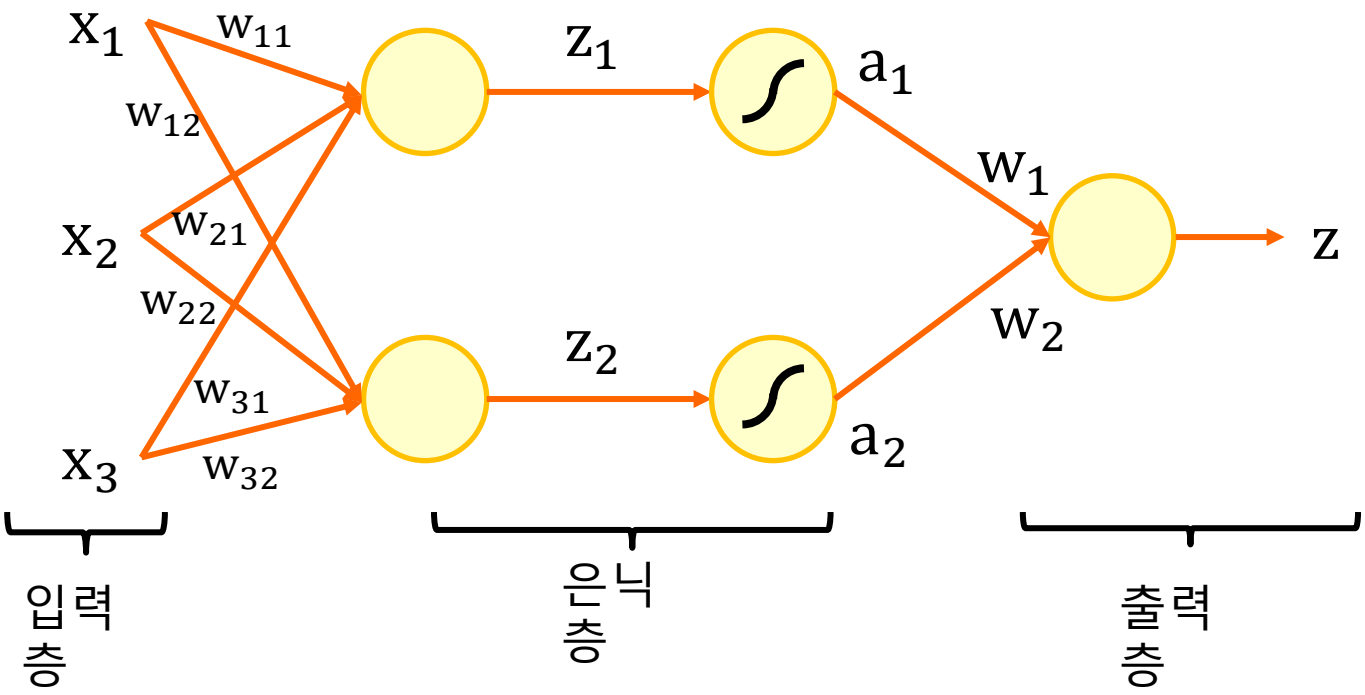
$$a_1 w_1 + a_2 w_2 + b_2 = z$$

$$[a_1 \quad a_2] \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + b_2 = z$$

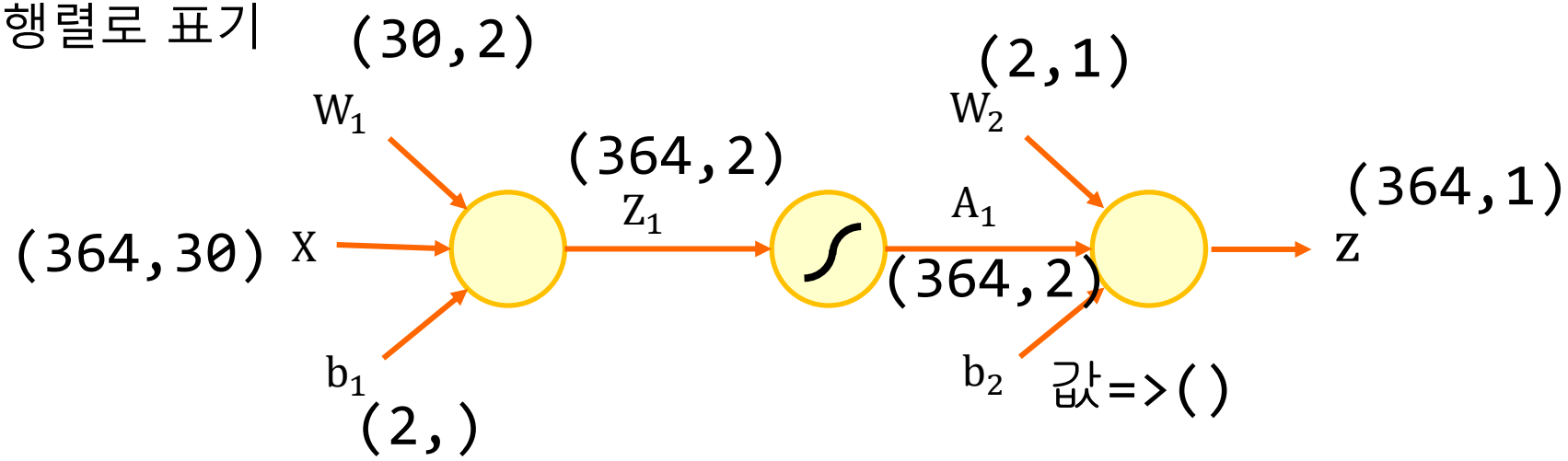
(m,2)(2,1) => (m,1)

$$A_1 W_2 + b_2 = z_2$$

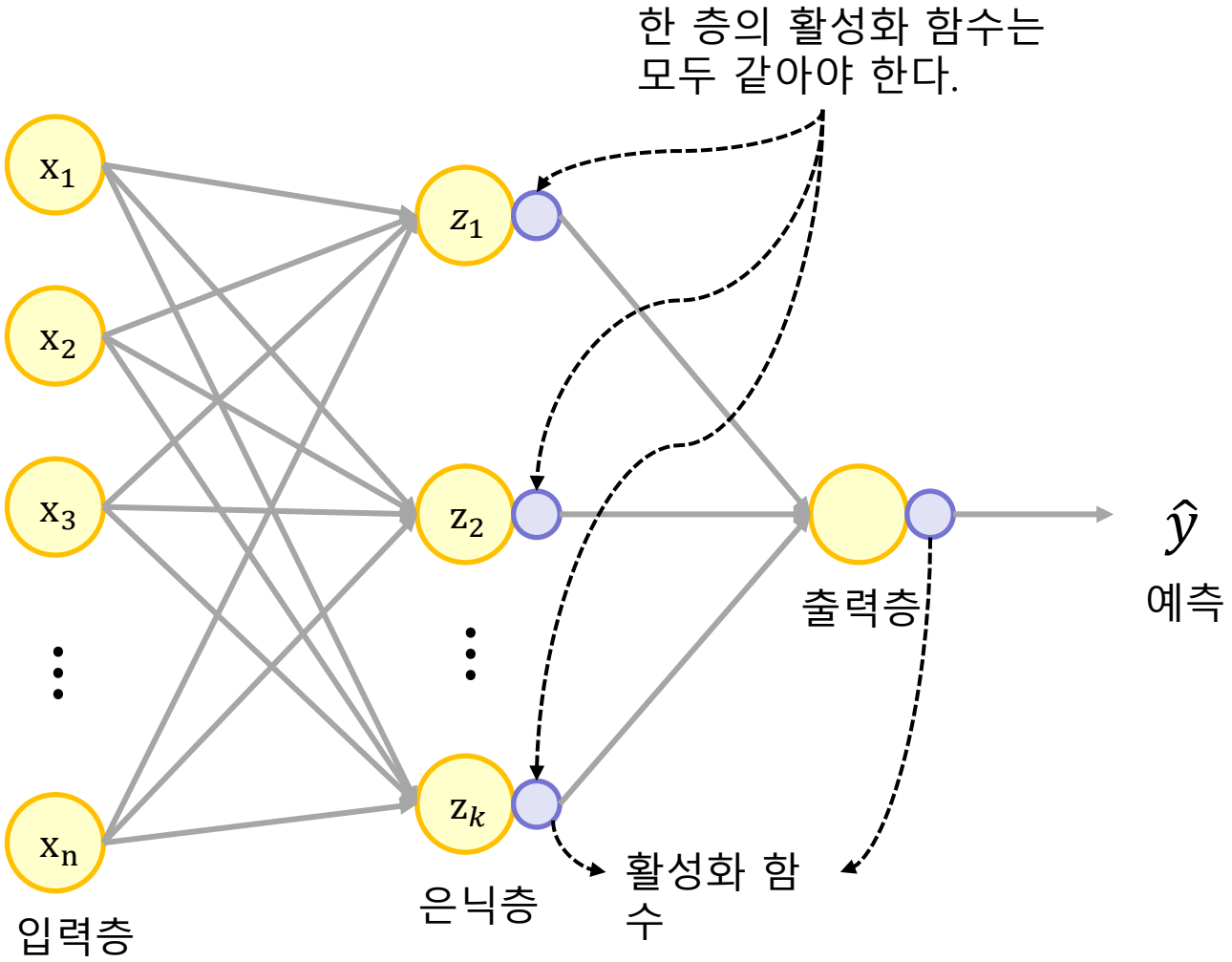
하나의 층에 여러개의 뉴런 사용



행렬로 표기



2층의 완전연결 신경망


한 층의 활성화 함수는
모두 같아야 한다.
x_1
x_2
x_3
x_n
z_1
z_2
z_k
$\hat{y}$
예측
출력층
은닉층
활성화 함수
입력층

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

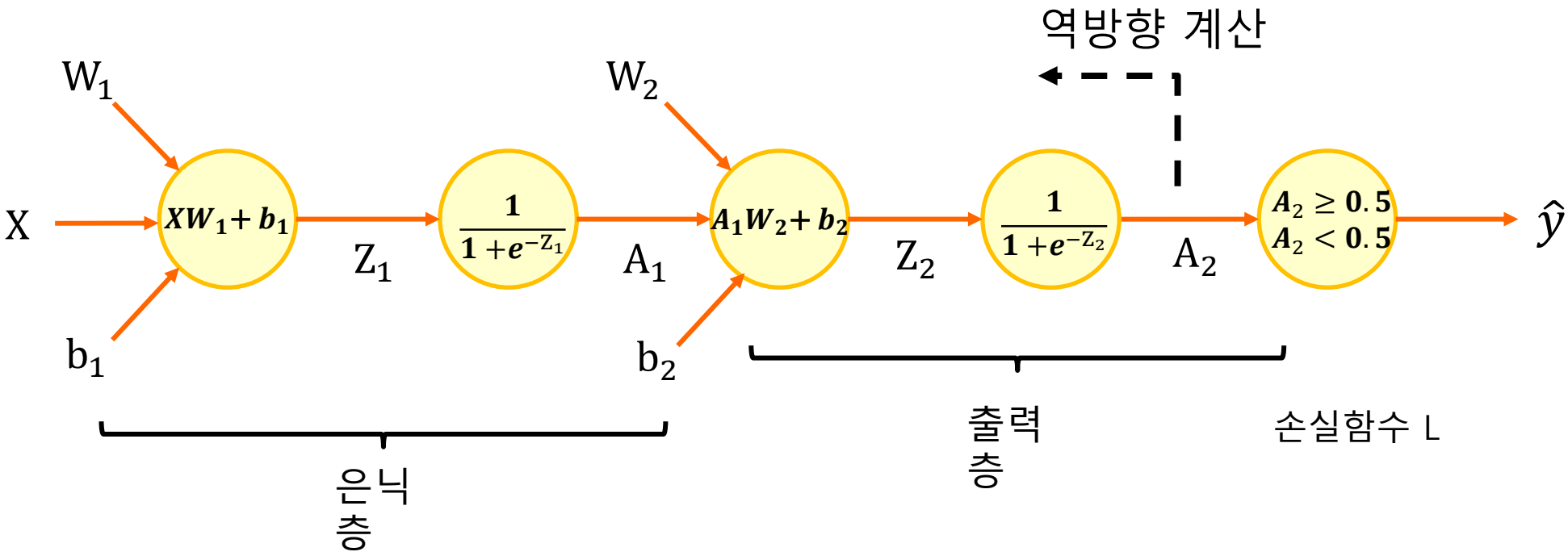
4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

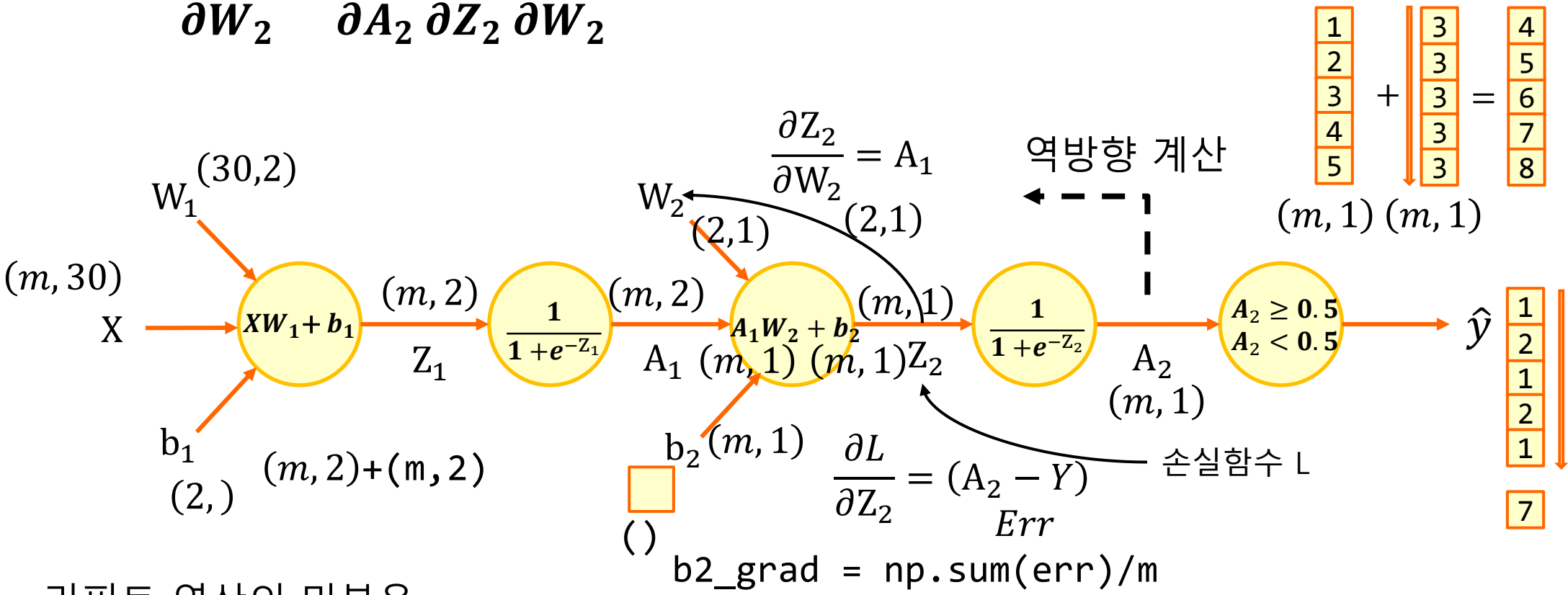

역방향 계산
W1
W2
X
XW1+b1
Z1
1/(1+e^-Z1)
A1
A1W2+b2
Z2
1/(1+e^-Z2)
A2
A2 ≥ 0.5
A2 < 0.5
ŷ
b1
b2
은닉층
출력층
손실함수 L

가중치 W_2 대하여 손실 함수를 미분한다. $\frac{\partial L}{\partial W_2} = (\mathrm{A}_1^T \cdot Err)/\mathrm{m}$

$(2, m)(m, 1) \Rightarrow (2,1)$ $\quad \boldsymbol{A_1 W_2 + b_2}$

$(\boldsymbol{m}, \boldsymbol{2})(\boldsymbol{2}, \boldsymbol{1}) \rightarrow (\boldsymbol{m}, \boldsymbol{1})$

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial W_2}} = \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial A_2}} \frac{\boldsymbol{\partial A_2}}{\boldsymbol{\partial Z_2}} \frac{\boldsymbol{\partial Z_2}}{\boldsymbol{\partial W_2}}$$



리피트 연산의 미분은
합연산 이다.(리피트의 방향과 합의 방향은 같아야 한다.)

가중치 W_2 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial Z_2} = (A_2 - Y) = \begin{bmatrix} 0.7 \\ 0.3 \\ \vdots \\ 0.6 \end{bmatrix} \} \text{ m개}$$

첫 번째 뉴런의 활성화 출력

$$\frac{\partial Z_2}{\partial W_2} = A_1 = \begin{bmatrix} -1.37 & 0.96 \\ & \vdots \\ 2.10 & -0.17 \end{bmatrix} \} \text{ m개}$$

첫 번째 뉴런의 활성화 출력

가중치 W_2 대하여 손실 함수를 미분한다.

행렬의 구성을 보면 A_1의 크기는 (m,2)이고 $(A_2 - Y)$ 의 크기는 (m,1)이므로 A_1을 전치하여 $(A_2 - Y)$ 와 곱해야 한다.

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial W_2} = A_1^T(A_2 - Y) = \begin{bmatrix} -1.37 & \cdots & 2.10 \\ 0.96 & \cdots & -0.17 \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \\ \cdots \\ 0.6 \end{bmatrix} = \begin{bmatrix} -0.12 \\ 0.36 \end{bmatrix}$$

그레디언트의 총합

(m, k)(m,1)

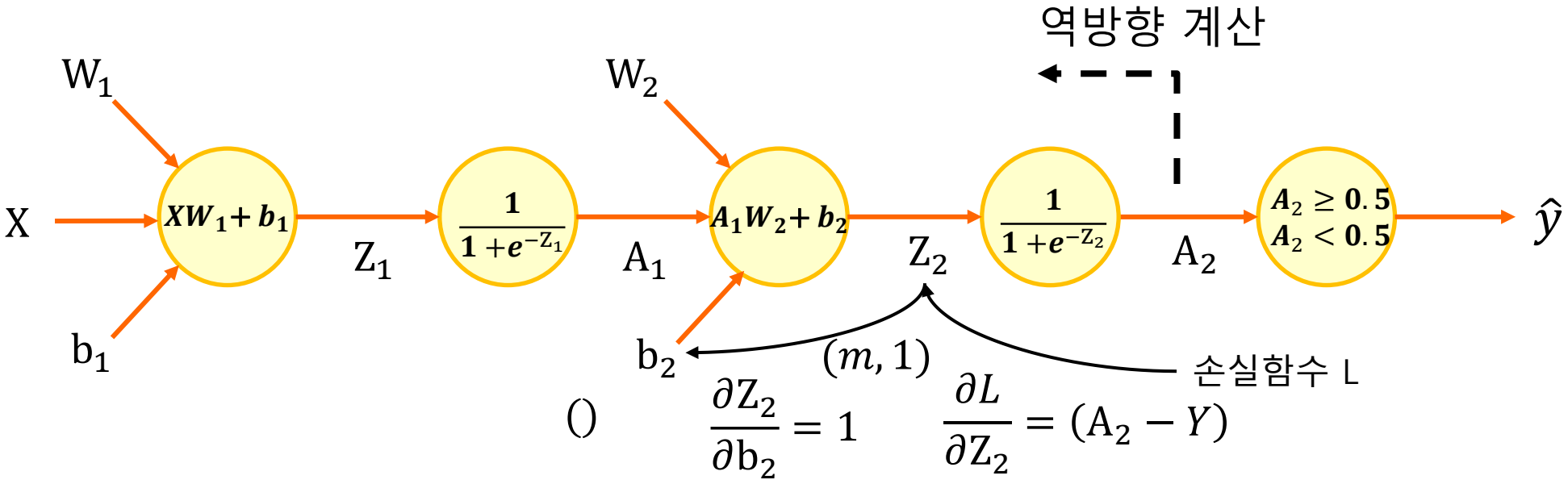
(k, m)(m,1) => (k, 1)

m개

타깃과 예측의 차이

절편 b_2 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial b_2}$$



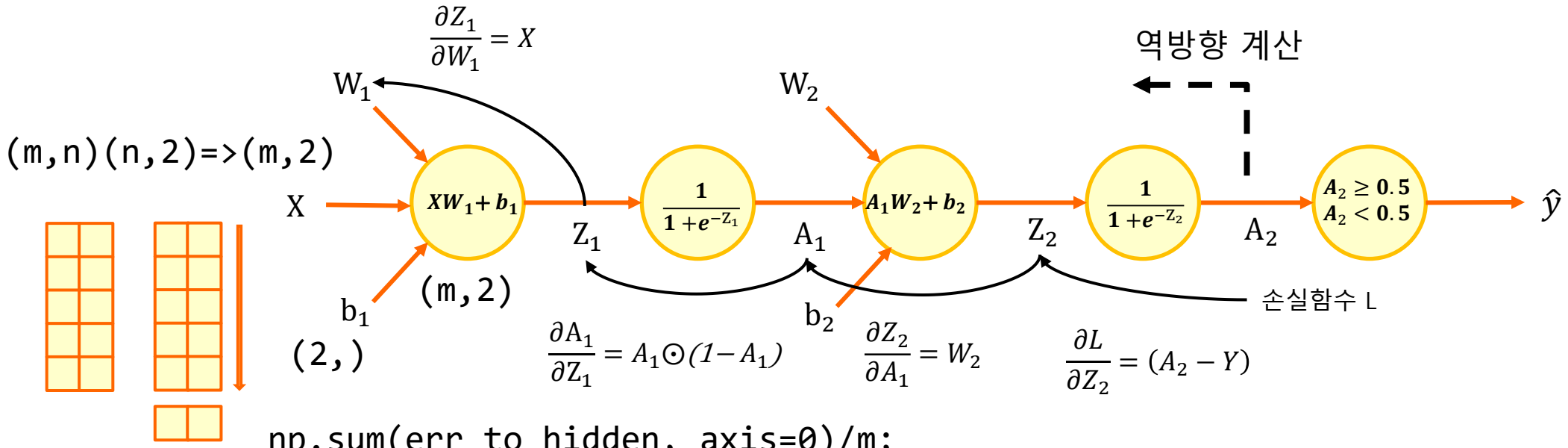
가중치 W_1 대하여 손실 함수를 미분한다. $Err \cdot W_2^T$

err_to_hidden

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial A_1}\frac{\partial A_1}{\partial Z_1}\frac{\partial Z_1}{\partial W_1} = X^T((A_2 - Y)\,W_2^T \odot A_1 \odot (1 - A_1))/m$$

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

(n,m)(m,2)=>(n,2)



np.sum(err_to_hidden, axis=0)/m;

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

미니배치 경사 하강법 이란?

배치 경사 하강법과 비슷하지만 에포크마다 전체 데이터를 사용하는 것이 아니라 조금씩 나누어 계산을 수행하고 그레디언트를 구하여 가중치를 업데이트 한다.

가중치 업데이트 방법

- 작게 나눈 미니 배치만큼 가중치를 업데이트 한다.
- 미니 배치의 크기는 보통 16,32,64등 2의 배수를 사용한다.
- 미니 배치의 크기가 1이면 확률적 경사 하강법이 된다.
- 미니 배치의 크기가 작으면 확률적 경사 하강법 처럼 작동하고 크면 배치 경사 하강법 처럼 작동한다.
- 미니 배치의 크기도 하이퍼파라미터이고 튜닝의 대상이다.

미니배치 경사 하강법 구현

__init__ () 함수에서 batch_size를 인자로 받아서 멤버 함수에 저장한다.

```
def __init__(self, units=10, batch_size=32, learning_rate=0.1,
l1=0, l2=0):
        super().__init__(units, learning_rate, l1, l2)
        self.batch_size = batch_size     # 배치 크기
```

fit() 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
        y_val = y_val.reshape(-1, 1)     # 타깃을 열 벡터로 바꿉니다.
        self.init_weights(x.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
        np.random.seed(42)
        # epochs만큼 반복합니다.
        for i in range(epochs):
            loss = 0
            # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
            for x_batch, y_batch in self.gen_batch(x, y):
                y_batch = y_batch.reshape(-1, 1) # 타깃을 열 벡터로 바꿉니다.
                m = len(x_batch)                 # 샘플 개수를 저장합니다.
                a = self.training(x_batch, y_batch, m)
                # 안전한 로그 계산을 위해 클리핑합니다.
                a = np.clip(a, 1e-10, 1-1e-10)
                # 로그 손실과 규제 손실을 더하여 리스트에 추가합니다.
                loss += np.sum(-(y_batch*np.log(a) + (1-y_batch)*np.log(1-a)))
            self.losses.append((loss + self.reg_loss()) / len(x))
            # 검증 세트에 대한 손실을 계산합니다.
            self.update_val_loss(x_val, y_val)
```

self.gen_batch(x, y) 함수를 이용하여 특정한 range를 추출 하여
training 하여 w, b를 미니배치 마다 업데이트 시킨다.

gen_batch() 함수 구현

```
# 미니배치 제너레이터 함수
    def gen_batch(self, x, y):  # x.shape = (364,30)
        length = len(x)  # length = 364
        bins = length // self.batch_size # 미니배치 횟수  364 // 32 == 11
        if length % self.batch_size:     # 32로 나누어 떨어지지 않는 나머지가 있는가?
            bins += 1                    # 나누어 떨어지지 않을 때
        indexes = np.random.permutation(np.arange(len(x))) # 인덱스를 섞습니다.
        x = x[indexes]
        y = y[indexes]
        for i in range(bins):
            start = self.batch_size * i
            end = self.batch_size * (i + 1)
            yield x[start:end], y[start:end]   # batch_size만큼 슬라이싱하여 반환합니다.
```

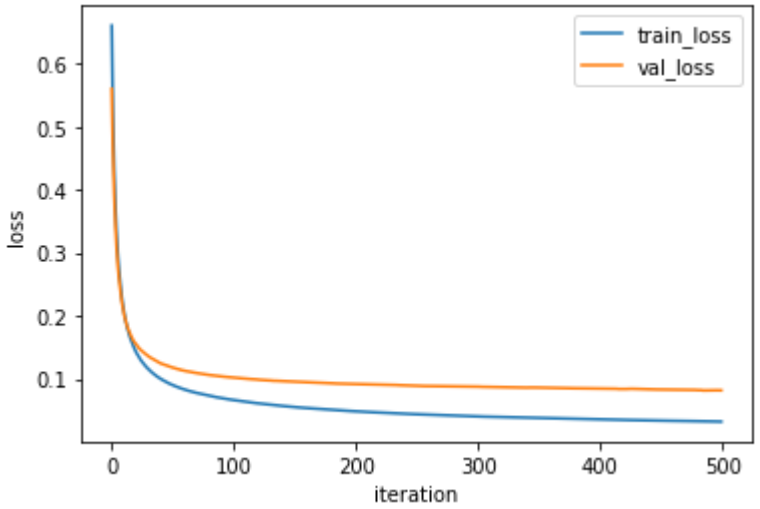
gen_batch() 함수는 파이썬 제너레이터로 구현한다.
제너레이터를 사용하면 명시적으로 리스트를 만들지 않으면서 필요한 만큼 데이터를 추출할 수 있으므로 효율적이다.
제너레이터 함수를 사용하려면 yield문을 사용하면 된다.
gen_batch() 함수는 batch_size 만큼씩 x,y 배열을 건너뛰며 미니 배치를 반환한다.
get_batch() 함수가 호출될 때마다 훈련 데이터 배열의 인덱스를 섞는다.

미니배치 하강법 적용 (batch_size = 32)

```
minibatch_net = MinibatchNetwork(l2=0.01, batch_size=32)
minibatch_net.fit(x_train_scaled, y_train,
                  x_val=x_val_scaled, y_val=y_val, epochs=500)
minibatch_net.score(x_val_scaled, y_val)
```

0.978021978021978

```
plt.plot(minibatch_net.losses)
plt.plot(minibatch_net.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```

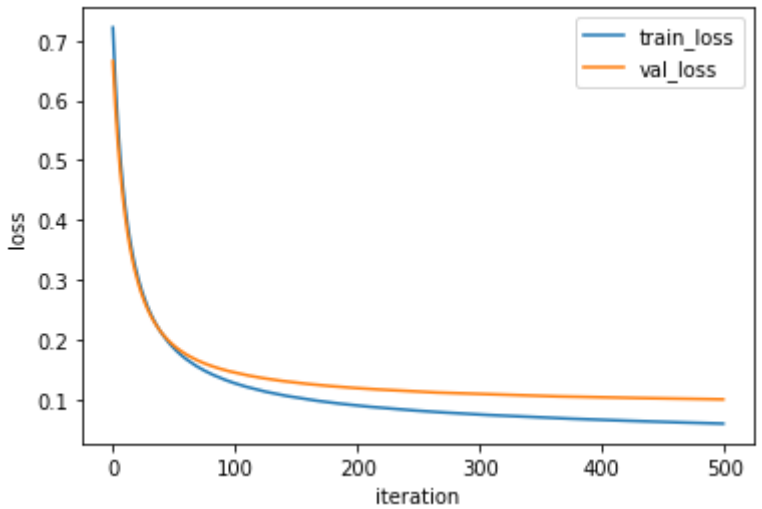


미니배치 하강법 적용 (batch_size = 128)

```
minibatch_net = MinibatchNetwork(l2=0.01, batch_size=128)
minibatch_net.fit(x_train_scaled, y_train,
                  x_val=x_val_scaled, y_val=y_val, epochs=500)
minibatch_net.score(x_val_scaled, y_val)
```

0.978021978021978

```
plt.plot(minibatch_net.losses)
plt.plot(minibatch_net.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

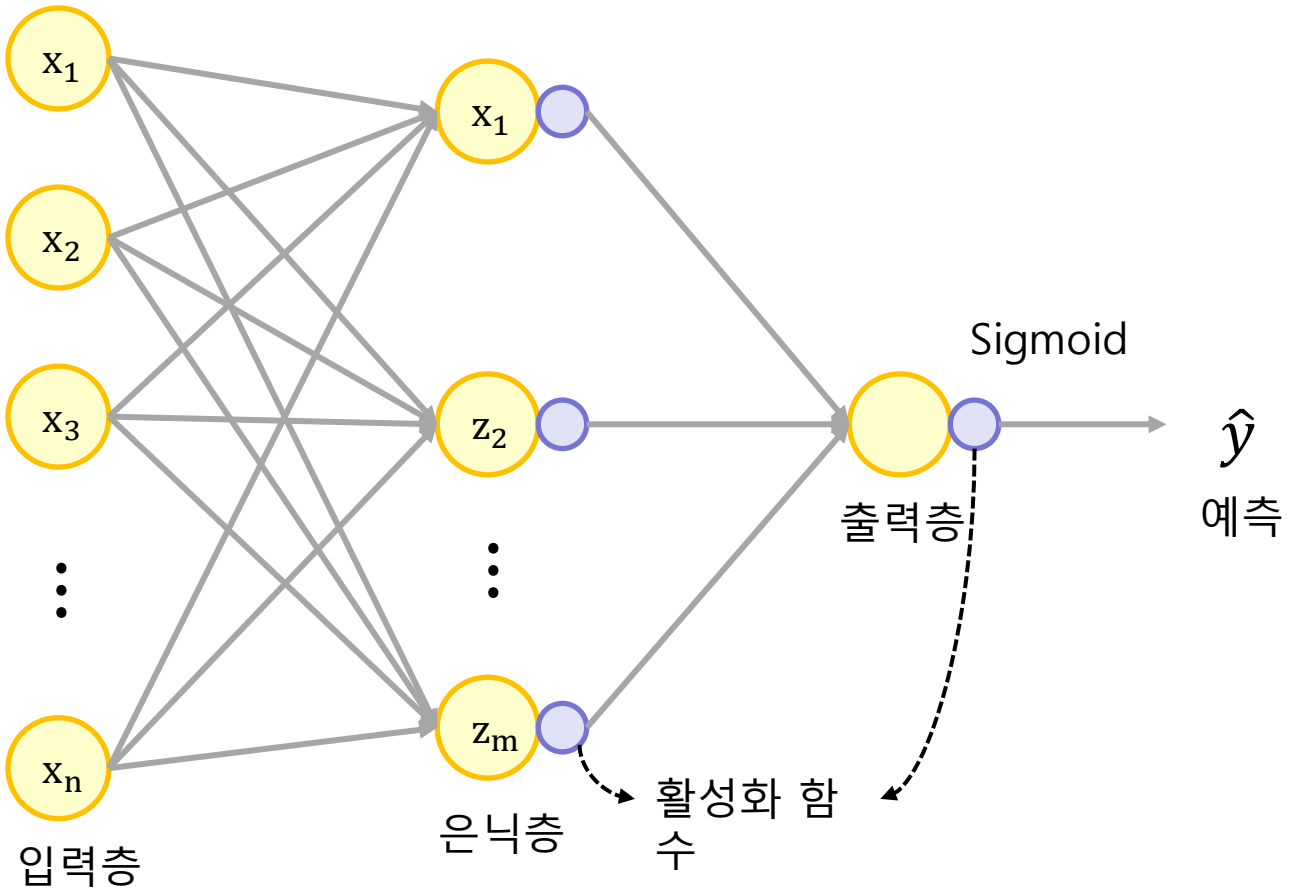
4.3 2개의 층을 가진 신경망 구현

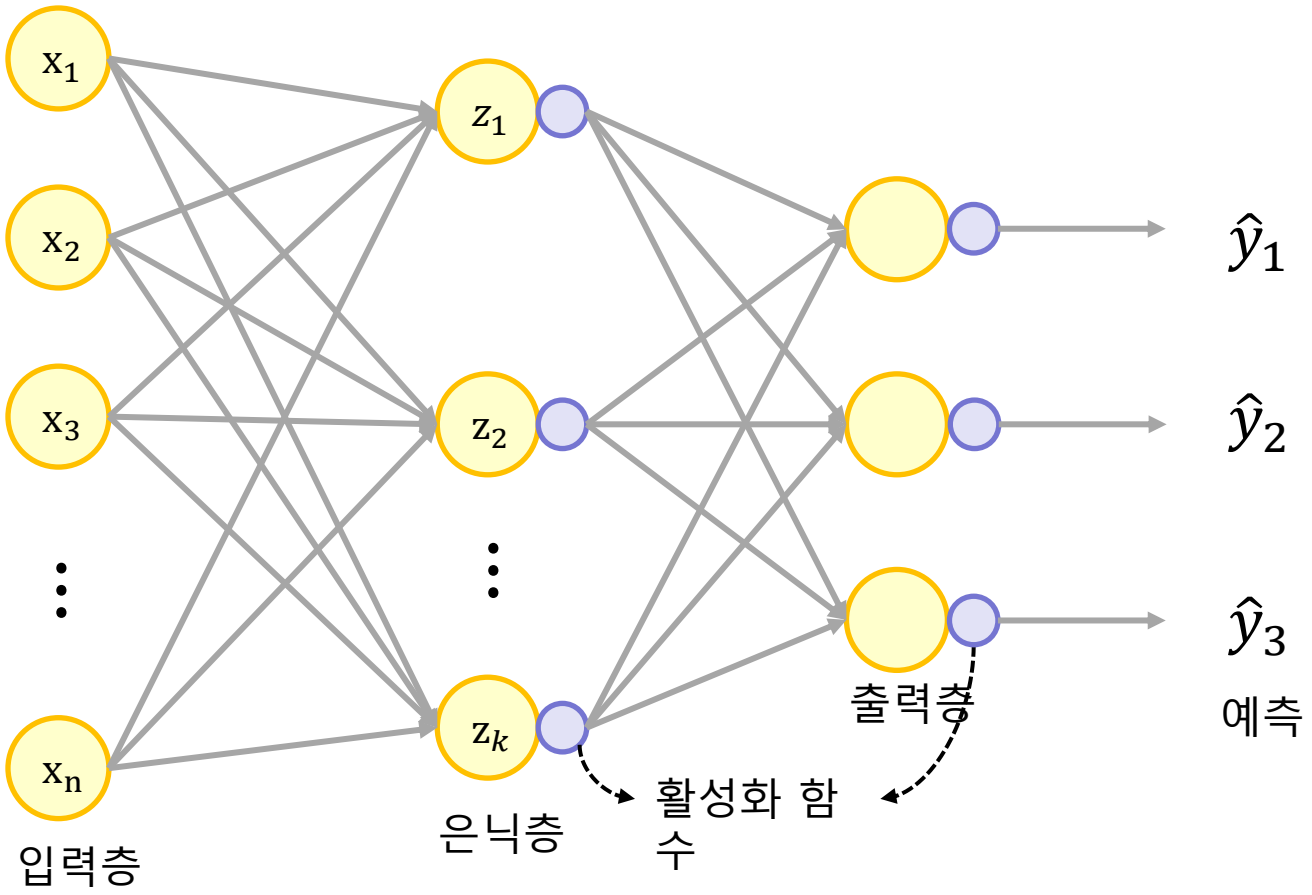
4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

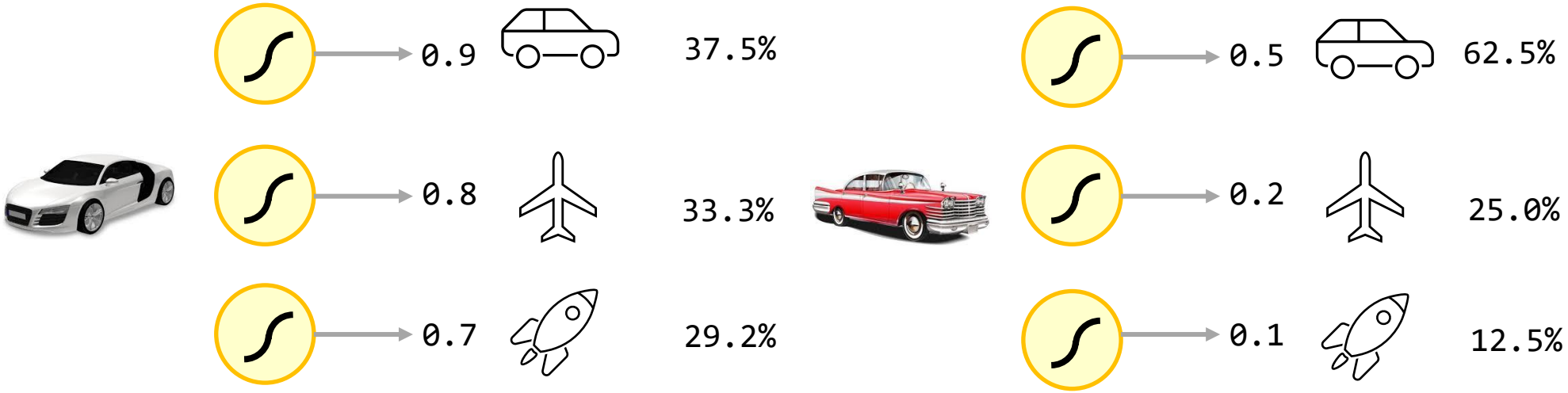
4.7 케라스를 이용한 다층신경망의 다양한 구현


x_1
x_2
x_3
x_n
입력층
x_1
z_2
z_m
은닉층
활성화 함수
Sigmoid
출력층
$\hat{y}$
예측


x_1
x_2
x_3
x_n
z_1
z_2
z_k
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
입력층
은닉층
활성화 함수
출력층
예측

활성화 출력의 합이 1이 아니면 비교하기 어렵다.

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.



소프트맥스 함수를 적용해 출력 강도를 정규화 한다.(1등을 더 1등 답게..)

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

2/(1+2+3)

$$\frac{e^{1}}{e^{1} + e^{2} + e^{3}}$$

$$\frac{e^{1}}{30.19}$$

2.718/(2.718+7.389+20.085)

z_1 1 $\frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$ 16.7% 9.0% $\frac{9.0}{9.0 + 4.0 + 2.3}$ 37.5%

z_2 2 $\frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$ 33.3% 24.5% $\frac{4.0}{9.0 + 4.0 + 2.3}$ 33.3%

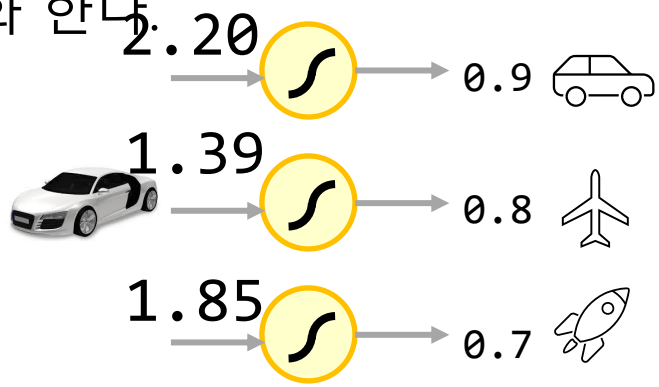
z_3 3 $\frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$ 50.0% 66.5% $\frac{2.3}{9.0 + 4.0 + 2.3}$ 29.2%

출력층

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z에 대해 정리하면

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



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

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

자동차 59%

37.5%

15.380

비행기 26%

33.3%

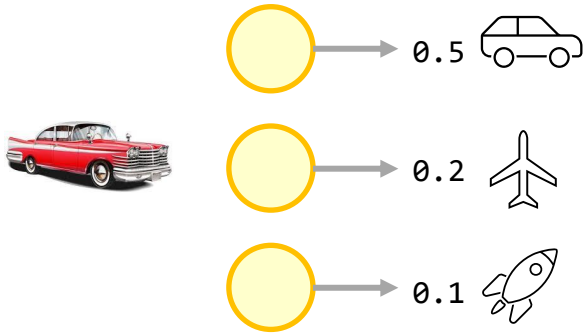
로켓 15%

29.2%

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z에 대해 정리하면

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



$$z_1 = -\log\left(\frac{1}{0.5} - 1\right) = 0.00 \qquad z_2 = -\log\left(\frac{1}{0.2} - 1\right) = -1.39 \qquad z_2 = -\log\left(\frac{1}{0.1} - 1\right) = -2.20$$

$$\hat{y} = \frac{e^{0.00}}{e^{0.00}+e^{-1.39}+e^{-2.20}} = 0.74 \qquad \hat{y} = \frac{e^{-1.39}}{e^{0.00}+e^{-1.39}+e^{-2.20}} = 0.18 \qquad \hat{y} = \frac{e^{-2.20}}{e^{0.00}+e^{-1.39}+e^{-2.20}} = 0.08$$

자동차 74%

62.5%

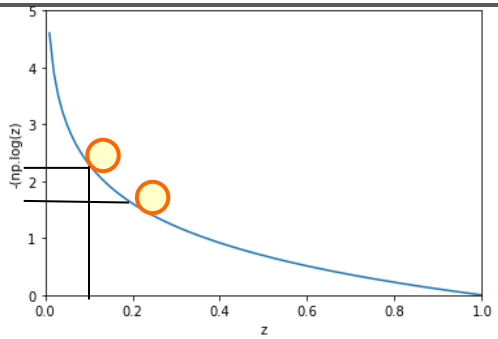
비행기 18%

25.0%

로켓 8%

12.5%

바이너리 크로스 엔트로피 손실 함수
카테고리컬 크로스 엔트로피 손실 함수



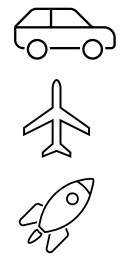
크로스 엔트로피 손실 함수

$$L = -\sum_{c=1}^{3} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + \cdots + y_c \log(a_c))$$

$$-\log(0.2)$$

y=1

a	y
0.7	0
0.1	0
0.2	1



크로스 엔트로피 손실 함수 미분

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

```
0 => 1000000000
1 => 0100000000
2 => 0010000000
```

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

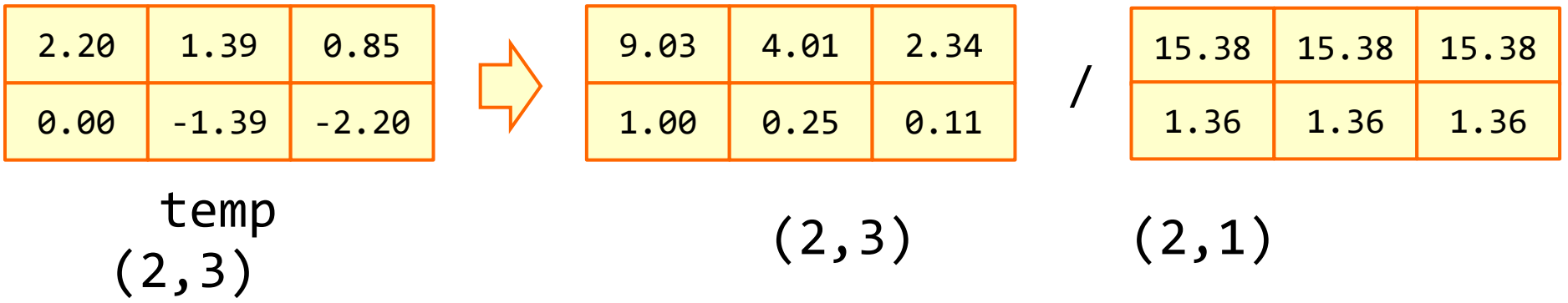
$$\frac{\partial \mathrm{L}}{\partial \mathrm{z}_1} = \left(-\frac{\mathrm{y}_1}{\mathrm{a}_1}\right)\frac{\partial \mathrm{a}_1}{\partial \mathrm{z}_1} + \left(-\frac{\mathrm{y}_2}{\mathrm{a}_2}\right)\frac{\partial \mathrm{a}_2}{\partial \mathrm{z}_1} + \left(-\frac{\mathrm{y}_3}{\mathrm{a}_3}\right)\frac{\partial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

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

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

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

```
import numpy as np
temp = np.array([[2.20,1.39,0.85],
                 [0.00,-1.39,-2.20]])
exp_temp = np.exp(temp)
print(exp_temp)
print(np.sum(exp_temp, axis=1))
ret = exp_temp / np.sum(exp_temp, axis=1).reshape(-1, 1)
print( ret )
```



다중 분류 신경망 구현

미니배치 하강법 적용 (batch_size = 128)

```
def sigmoid(self, z):
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    a = 1 / (1 + np.exp(-z))              # 시그모이드 계산
    return a

def softmax(self, z):
    # 소프트맥스 함수
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1).reshape(-1, 1)
```

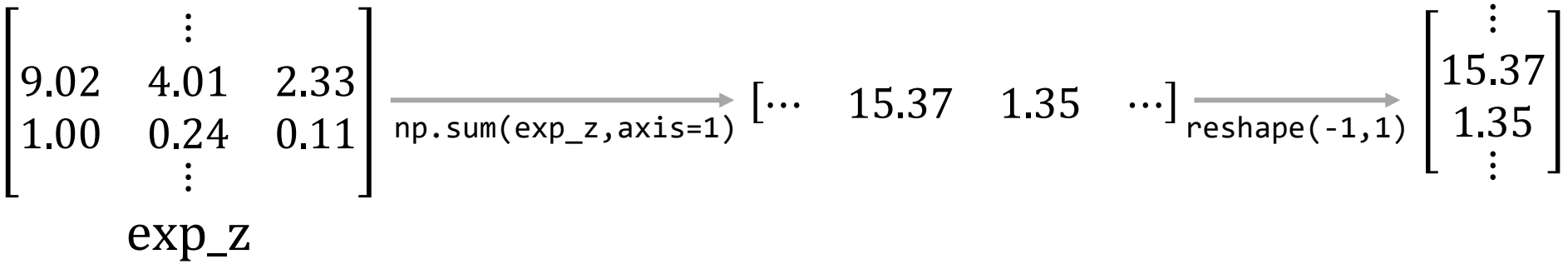
$$\begin{bmatrix} \vdots & \vdots & \vdots \\ 2.20 & 1.39 & 0.85 \\ 0.00 & -1.39 & -2.20 \\ \vdots & \vdots & \vdots \end{bmatrix} \xrightarrow{\texttt{np.exp(z)}} \begin{bmatrix} \vdots & \vdots & \vdots \\ e^{2.20} & e^{1.39} & e^{0.85} \\ e^{0.00} & e^{-1.39} & e^{-2.20} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

z $\qquad$ e^z

다중 분류 신경망 구현

미니배치 하강법 적용 (batch_size = 128)

```
def sigmoid(self, z):
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    a = 1 / (1 + np.exp(-z))              # 시그모이드 계산
    return a

def softmax(self, z):
    # 소프트맥스 함수
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1).reshape(-1, 1)
```



다중 분류 신경망 구현

미니배치 하강법 적용 (batch_size = 128)

```
def sigmoid(self, z):
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    a = 1 / (1 + np.exp(-z))              # 시그모이드 계산
    return a

def softmax(self, z):
    # 소프트맥스 함수
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1).reshape(-1, 1)
```

$$\begin{bmatrix} \vdots & & \\ 9.02 & 4.01 & 2.33 \\ 1.00 & 0.24 & 0.11 \\ & \vdots & \end{bmatrix} \div \begin{bmatrix} \vdots \\ 15.37 \\ 1.35 \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots & & \\ 0.59 & 0.26 & 0.15 \\ 0.74 & 0.18 & 0.08 \\ & \vdots & \end{bmatrix}$$

exp_z

np.sum(exp_z,axis=1).reshape(-1,1)

다중 분류 신경망 구현

가중치 초기화

```
def init_weights(self, n_features, n_classes):
    self.w1 = np.random.normal(0, 1,
                               (n_features, self.units))  # (특성 개수, 은닉층의 크기)
    self.b1 = np.zeros(self.units)                        # 은닉층의 크기
    self.w2 = np.random.normal(0, 1,
                               (self.units, n_classes))   # (은닉층의 크기, 클래스 개수)
    self.b2 = np.zeros(n_classes)
```

다중 분류 신경망 구현

fit() 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
    np.random.seed(42)
    self.init_weights(x.shape[1], y.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
    # epochs만큼 반복합니다.
    for i in range(epochs):
        loss = 0
        print('.', end='')
        # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
        for x_batch, y_batch in self.gen_batch(x, y):
            a = self.training(x_batch, y_batch)
            # 안전한 로그 계산을 위해 클리핑합니다.
            a = np.clip(a, 1e-10, 1-1e-10)
            # 로그 손실과 규제 손실을 더하여 리스트에 추가합니다.
            loss += np.sum(-y_batch*np.log(a))
        self.losses.append((loss + self.reg_loss()) / len(x))
        # 검증 세트에 대한 손실을 계산합니다.
        self.update_val_loss(x_val, y_val)
```

다중 분류 신경망 구현

training() 함수 수정

```
def training(self, x, y):
    m = len(x)                # 샘플 개수를 저장합니다.
    z = self.forpass(x)       # 정방향 계산을 수행합니다.
    a = self.softmax(z)       # 활성화 함수를 적용합니다.
```

predict() 함수 수정

```
def predict(self, x):
    z = self.forpass(x)           # 정방향 계산을 수행합니다.
    return np.argmax(z, axis=1)  # 가장 큰 값의 인덱스를 반환합니다.
```

score() 함수 수정

```
def score(self, x, y):
    # 예측과 타깃 열 벡터를 비교하여 True의 비율을 반환합니다.
    return np.mean(self.predict(x) == np.argmax(y, axis=1))
```

다중 분류 신경망 구현

검증 손실 계산

```
def update_val_loss(self, x_val, y_val):
    z = self.forpass(x_val)             # 정방향 계산을 수행합니다.
    a = self.softmax(z)                 # 활성화 함수를 적용합니다.
    a = np.clip(a, 1e-10, 1-1e-10)      # 출력 값을 클리핑합니다.
    # 크로스 엔트로피 손실과 규제 손실을 더하여 리스트에 추가합니다.
    val_loss = np.sum(-y_val*np.log(a))
    self.val_losses.append((val_loss + self.reg_loss()) / len(y_val))
```

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

텐서플로 임포트

```
import tensorflow as tf
```

텐서플로 버전 확인

```
tf.__version__
```

```
'2.1.0'
```

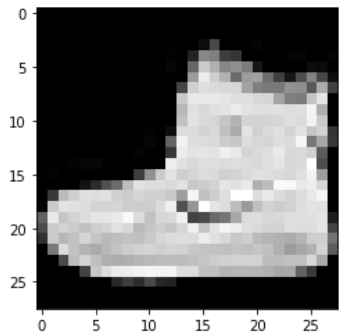
패션 MNIST 데이터 세트 불러오기

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

```
(60000, 28, 28) (60000,)
```

패션 MNIST 데이터 세트 불러오기

```
import matplotlib.pyplot as plt

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



타깃의 내용과 의미 확인

```
print(y_train_all[:10])
```

```
 [9 0 0 3 0 2 7 2 5 5]
```

```
class_names = ['티셔츠/윗도리', '바지', '스웨터', '드레스', '코트',
               '샌들', '셔츠', '스니커즈', '가방', '앵클부츠']
print(class_names[y_train_all[0]])
```

```
'앵클부츠'
```

타깃 분포 확인

```
np.bincount(y_train_all)
```

```
array([6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000], dtype=int64)
```

훈련 세트와 검증 세트 고르게 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)
```

```
np.bincount(y_train)
```

```
array([4800, 4800, 4800, 4800, 4800, 4800, 4800, 4800, 4800, 4800], dtype=int64)
```

```
np.bincount(y_val)
```

```
array([1200, 1200, 1200, 1200, 1200, 1200, 1200, 1200, 1200, 1200], dtype=int64)
```

입력 데이터 정규화

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

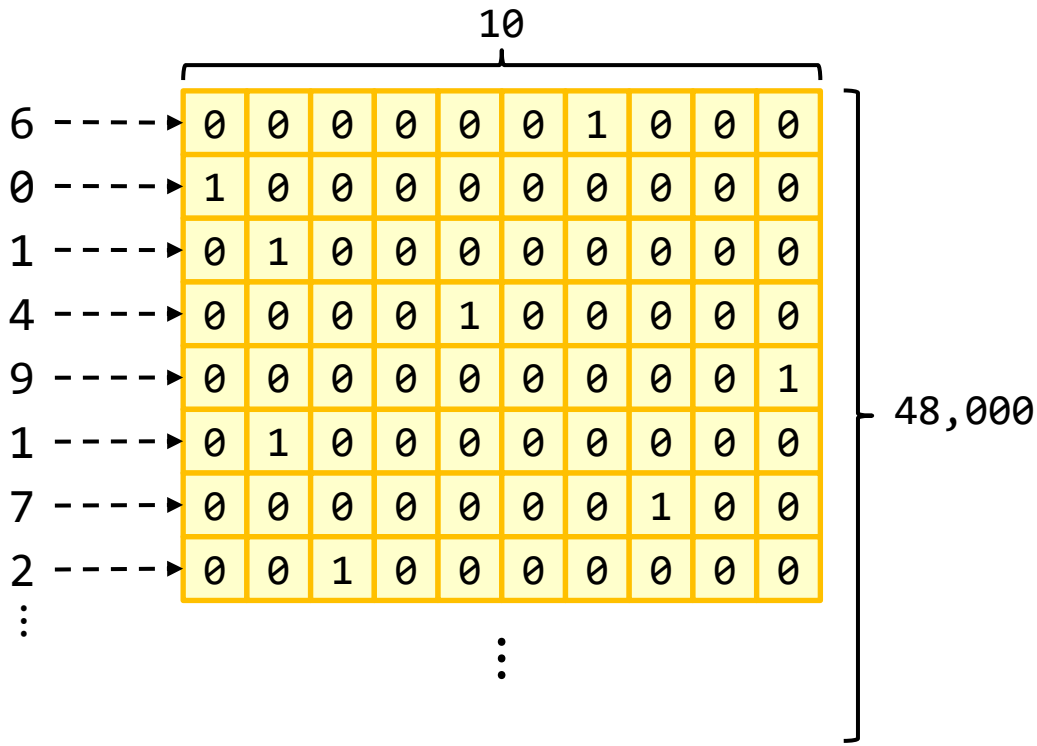
훈련 세트와 검증 세트의 차원 변경

```
x_train = x_train.reshape(-1, 784)
x_val = x_val.reshape(-1, 784)
print(x_train.shape, x_val.shape)
```

```
(48000, 784) (12000, 784)
```

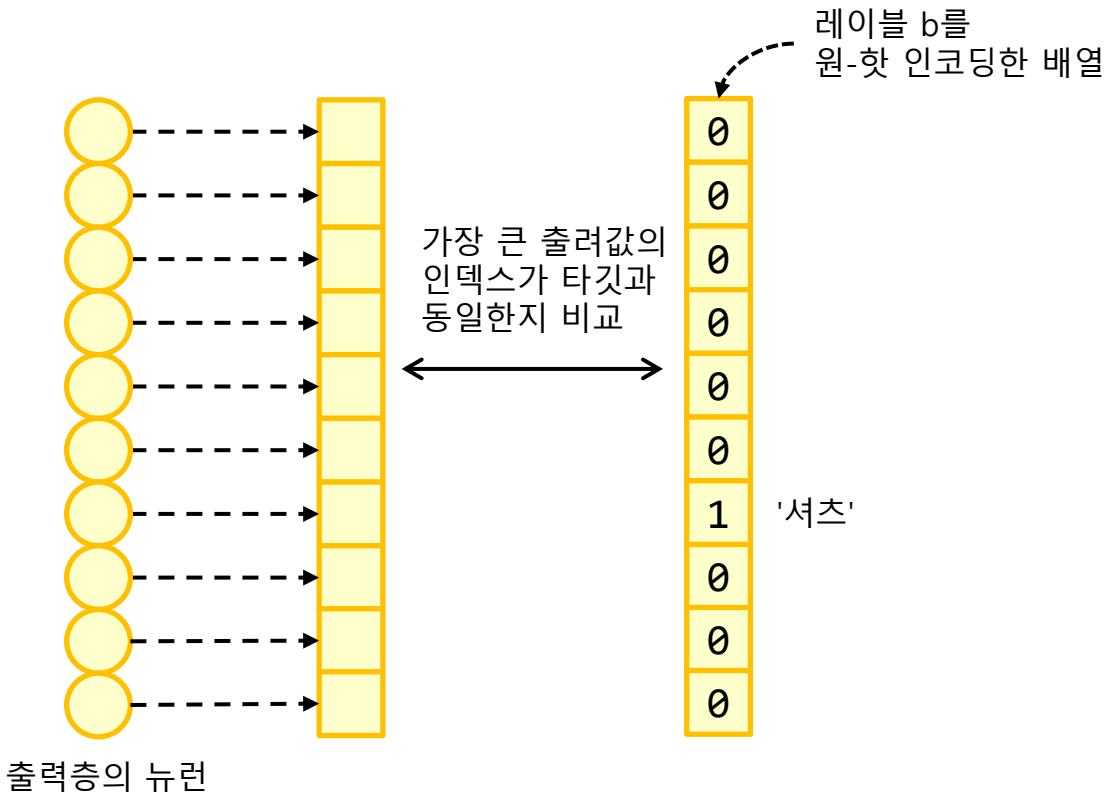
타깃 데이터를 준비하고 다중 분류 신경망을 훈련한다.

타깃을 원-핫 인코딩으로 변환



타깃 데이터를 준비하고 다중 분류 신경망을 훈련한다.

타깃을 원-핫 인코딩으로 변환



to_categorical() 함수 사용해 원-핫 인코딩하기

```
tf.keras.utils.to_categorical([0, 1, 3])
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 1.]], dtype=float32)
```

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

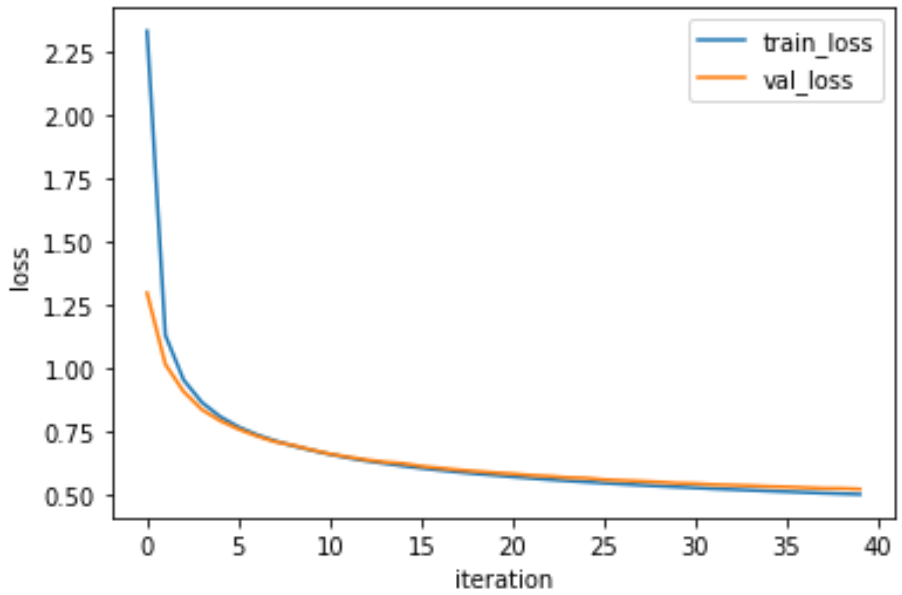
```
(48000, 10) (12000, 10)
```

```
print(y_train[0], y_train_encoded[0])
```

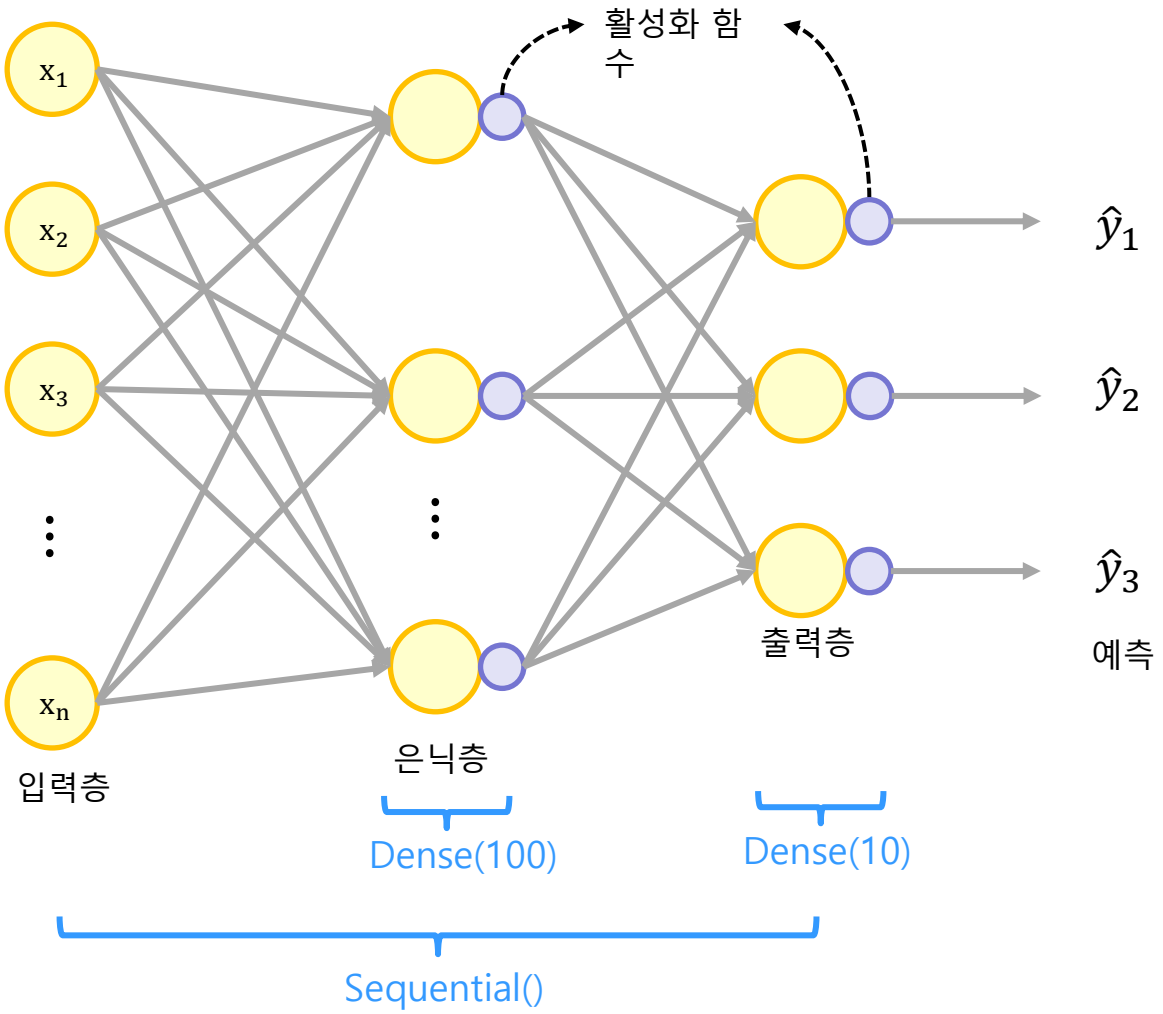
```
6 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

MultiClassNetwork 클래스로 다중 분류 신경망 훈련하기

```
fc = MultiClassNetwork(units=100, batch_size=256)
fc.fit(x_train, y_train_encoded,
       x_val=x_val, y_val=y_val_encoded, epochs=40)
plt.plot(fc.losses)
plt.plot(fc.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



케라스로 다중 분류 신경망 훈련하기



케라스로 다중 분류 신경망 훈련하기

모델 생성

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
```

은닉층과 출력층을 모델에 추가

```
model.add(Dense(100, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

최적화 알고리즘과 손실 함수 지정

```
model.compile(optimizer='sgd', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

모델 훈련하기

```
history = model.fit(x_train, y_train_encoded, epochs=40,
                    validation_data=(x_val, y_val_encoded))
```

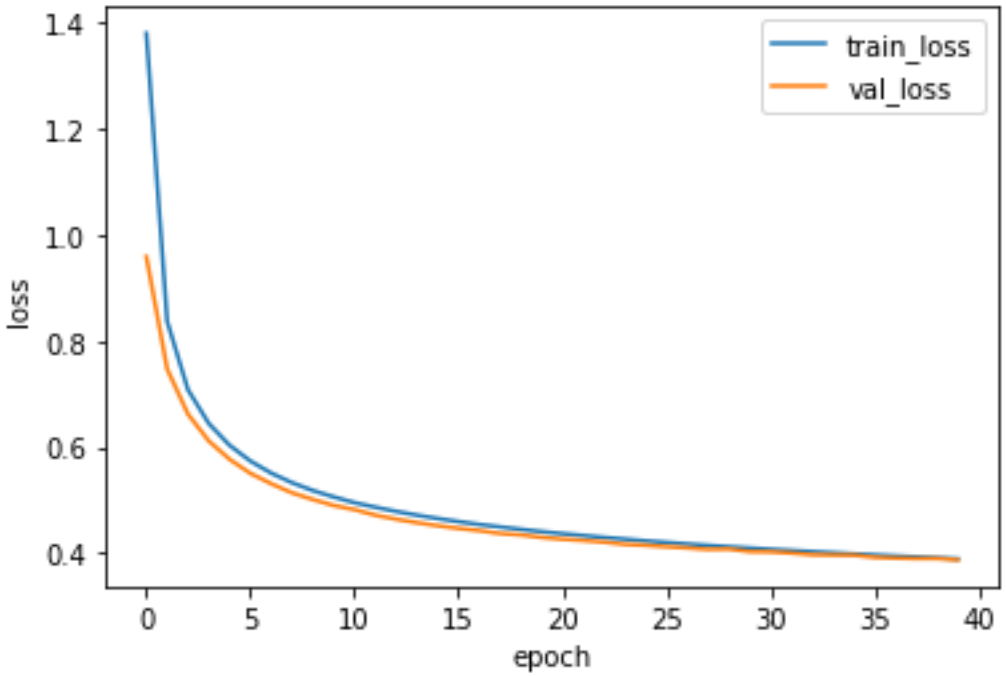
history객체의 history 딕셔너리

```
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

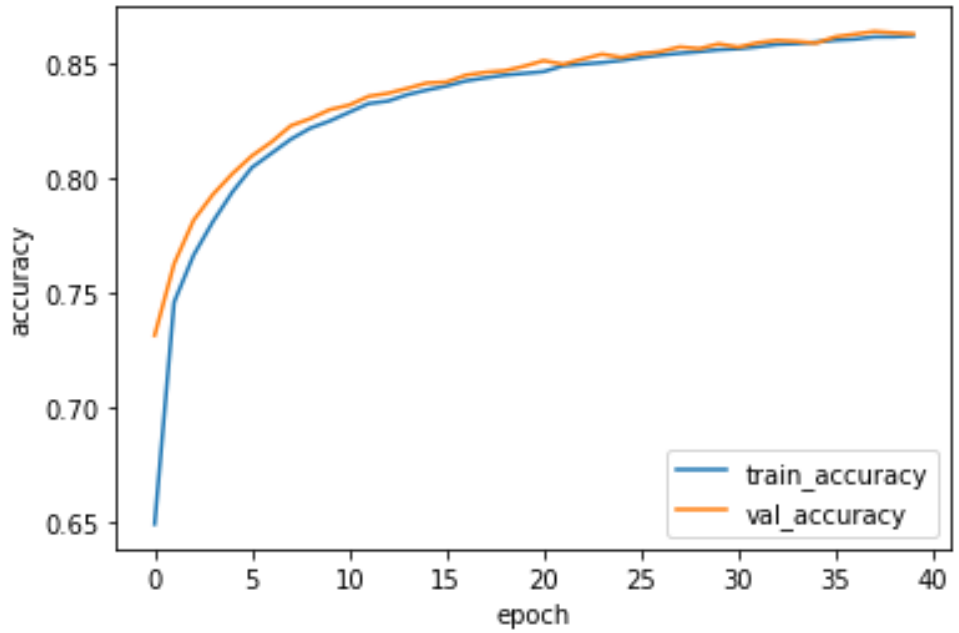
손실 그래프

```
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()
```



정확도 그래프

```
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()
```



검증 세트 정확도 계산

```
loss, accuracy = model.evaluate(x_val, y_val_encoded, verbose=0)
print(accuracy)
```

```
0.86291665
```

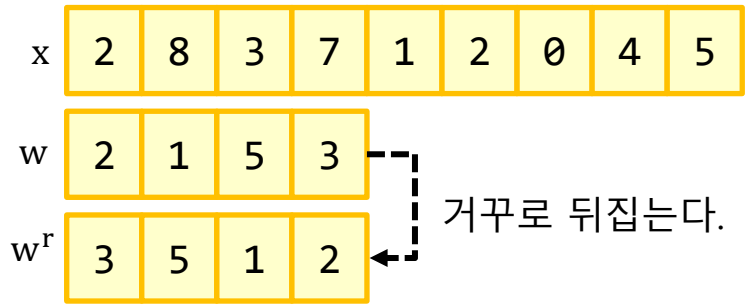
5. 합성곱 신경망 (CNN) 이해

5.1 합성곱 연산

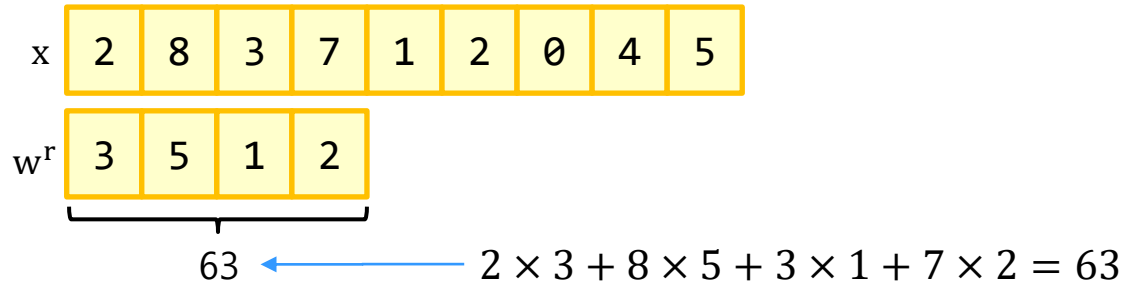
5.2 합성곱 신경망 구현

5.3 케라스로 합성곱 신경망 구현

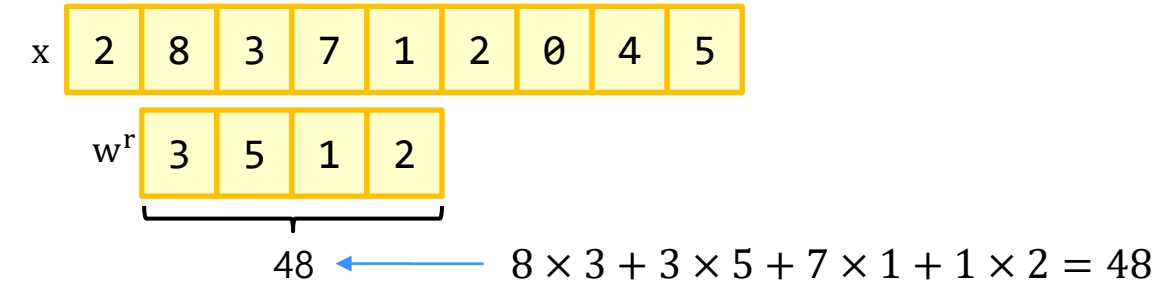
배열 하나 선택해 뒤집기



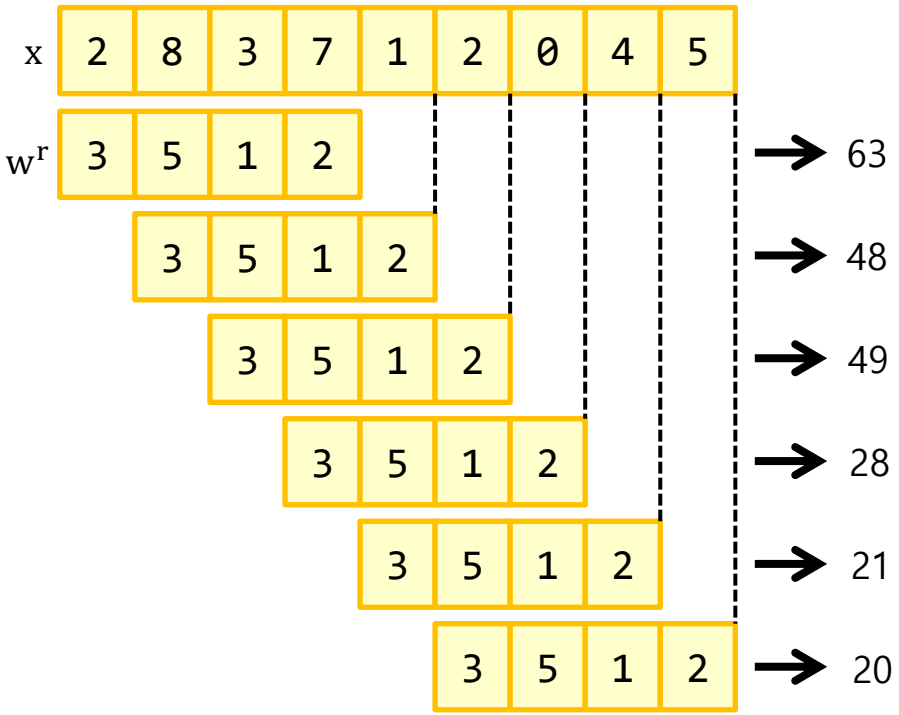
첫 번째 합성곱



두 번째 합성곱



전체 합성곱



N-F+1 = Out

9-4+1 = 6

63 48 49 28 21 20

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]

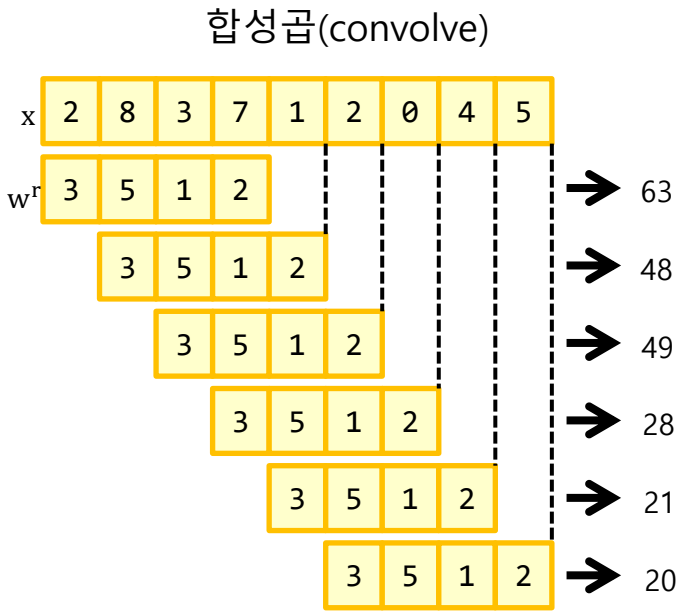
$$\begin{bmatrix} 2 & 8 & 3 & 7 \\ 8 & 3 & 7 & 1 \\ 3 & 7 & 1 & 2 \\ 7 & 1 & 2 & 0 \\ 1 & 2 & 0 & 4 \\ 2 & 0 & 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 63 \\ 48 \\ 49 \\ 28 \\ 21 \\ 20 \end{bmatrix}$$

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.

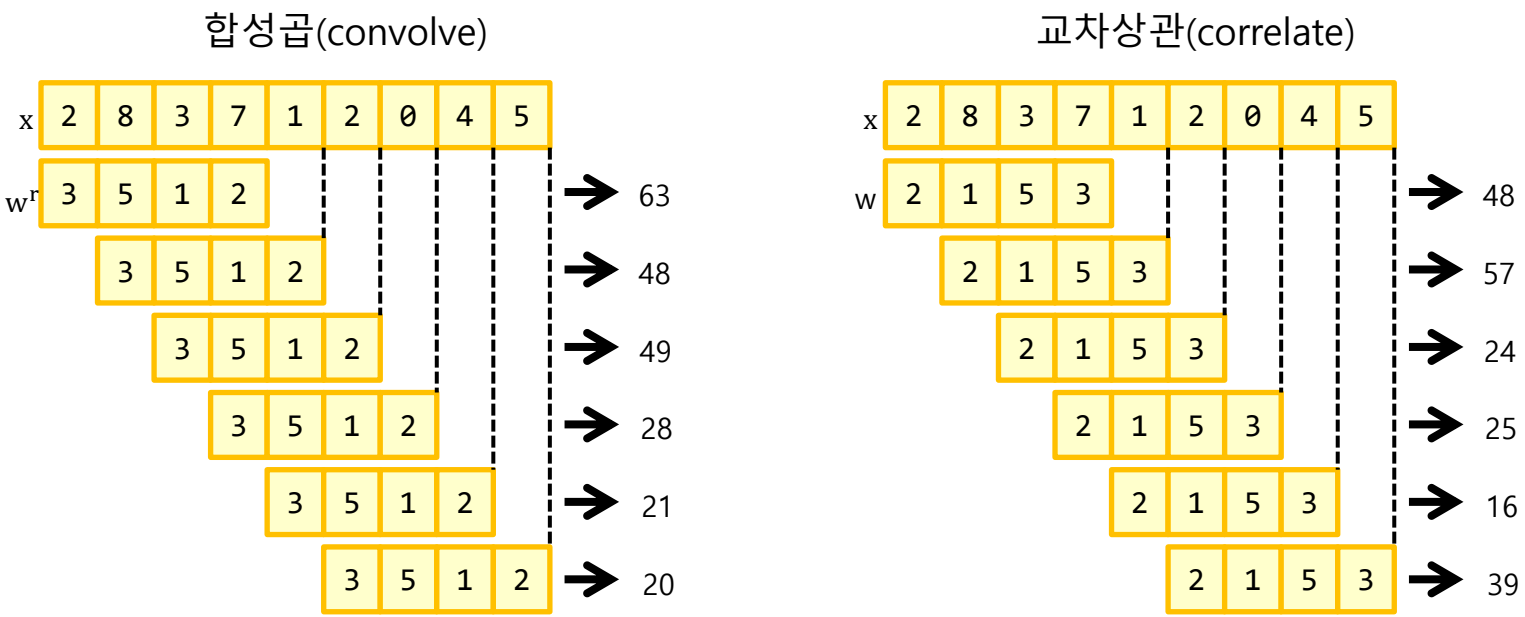


싸이파이로 교차상관 수행

```
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.

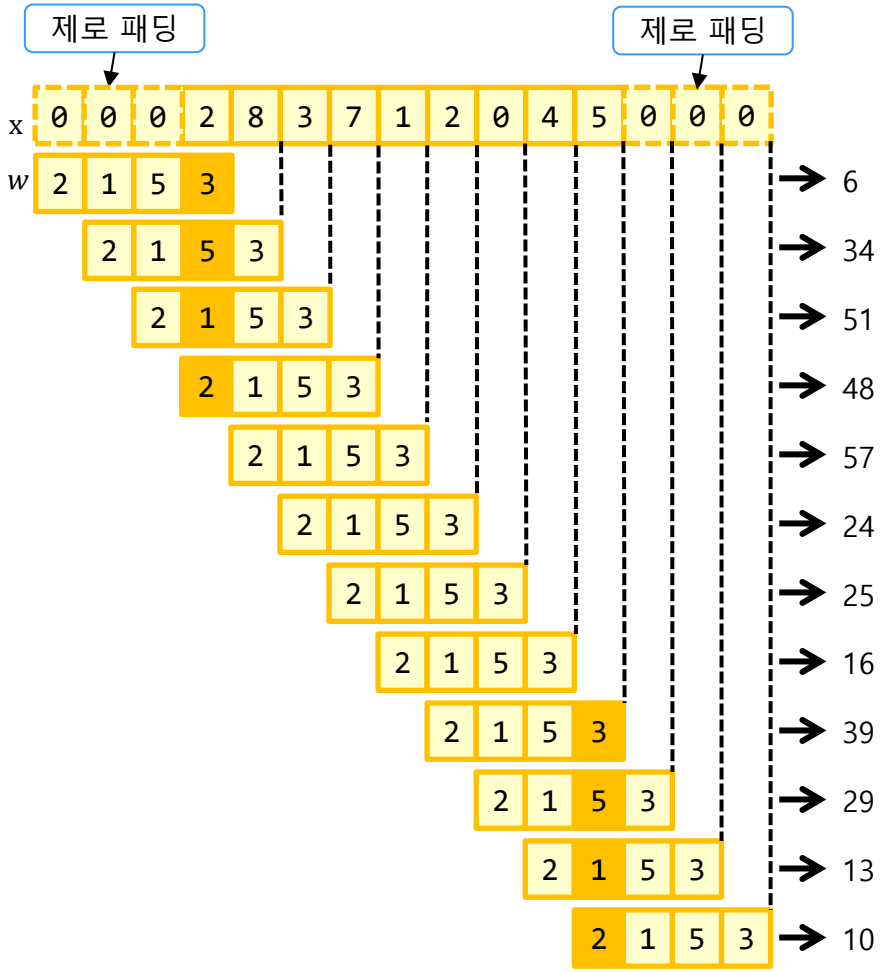


패딩과 스트라이드 이해

밸리드 패딩은 원본 배열의 원소가 합성곱 연산에 참여하는 정도가 다르다.

패딩과 스트라이드 이해

풀 패딩은 원본 배열의 원소의 연산 참여도를 동일하게 만든다.




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

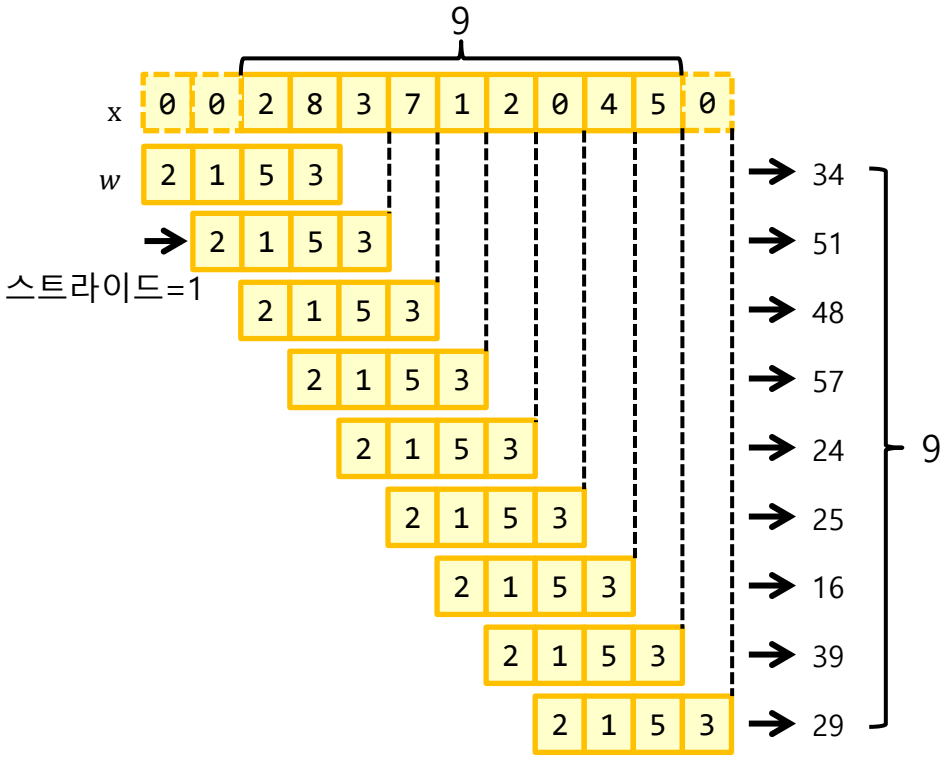
```
array([ 6, 34, 51, 48, 57, 24, 25, 16, 39, 29, 13, 10])
```

패딩과 스트라이드 이해

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

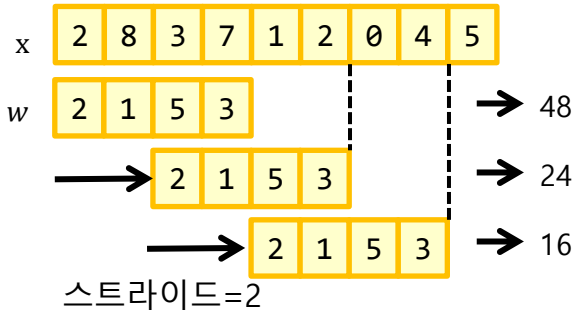
```
correlate(x, w, mode='same')
```

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



패딩과 스트라이드 이해

스트라이드는 미끄러지는 간격을 조정한다.



N-F+1

(N-F)//stride+1

(9-4)//1+1 = 6

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

2차원 배열에서 합성곱 수행 (mode='valid')

x

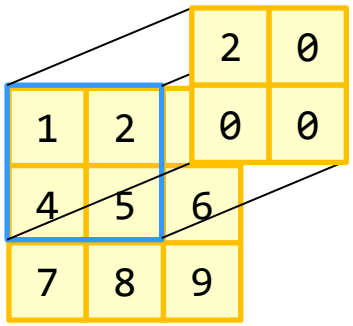
1	2	3
4	5	6
7	8	9

w

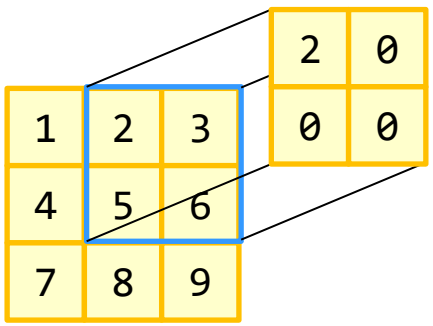
2	0
0	0

```
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')
```

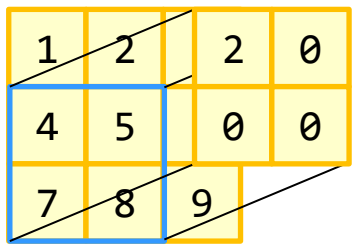
```
array([[ 2,  4],
       [ 8, 10]])
```



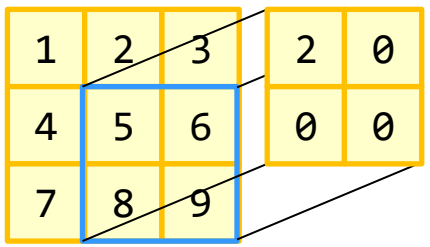
=> 2



=> 4



=> 8



=> 10

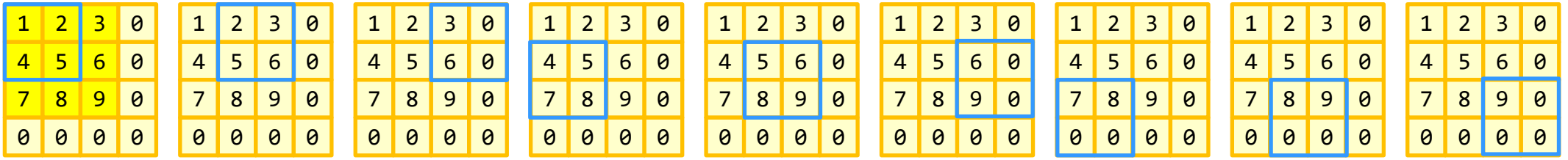
2	4
8	10

2차원 배열에서 same padding

$$N+P-F+1 = O$$

$$3+1-2+1 = 3$$

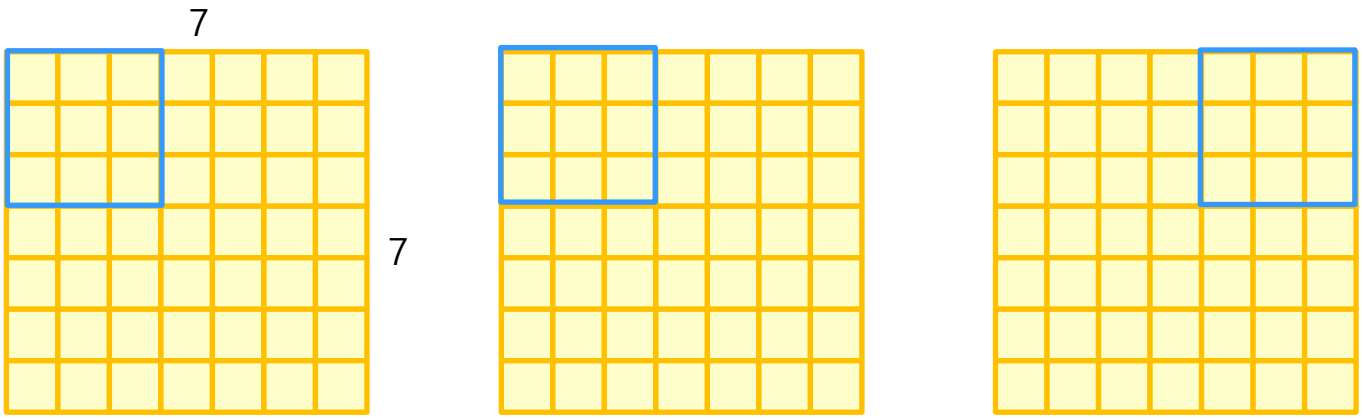
$$3+1-2+1 = 3$$




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

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

2차원 배열에서 스트라이드 이해



7x7 input
3x3 filter
mode = 'valid'
stride = 1

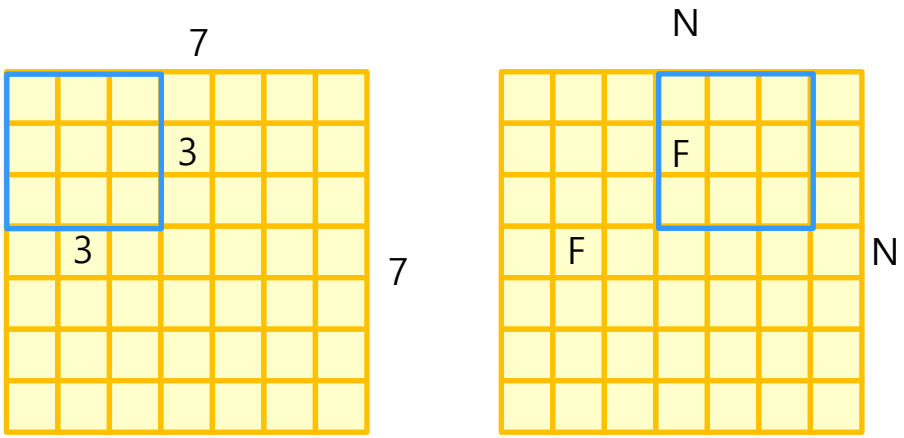
=> 5x5 output

7x7 input
3x3 filter
mode = 'valid'
stride = 2

=> 3x3 output

```
(N-F)//S+1

(7-3)//2+1
```

2차원 배열에서 스트라이드 이해

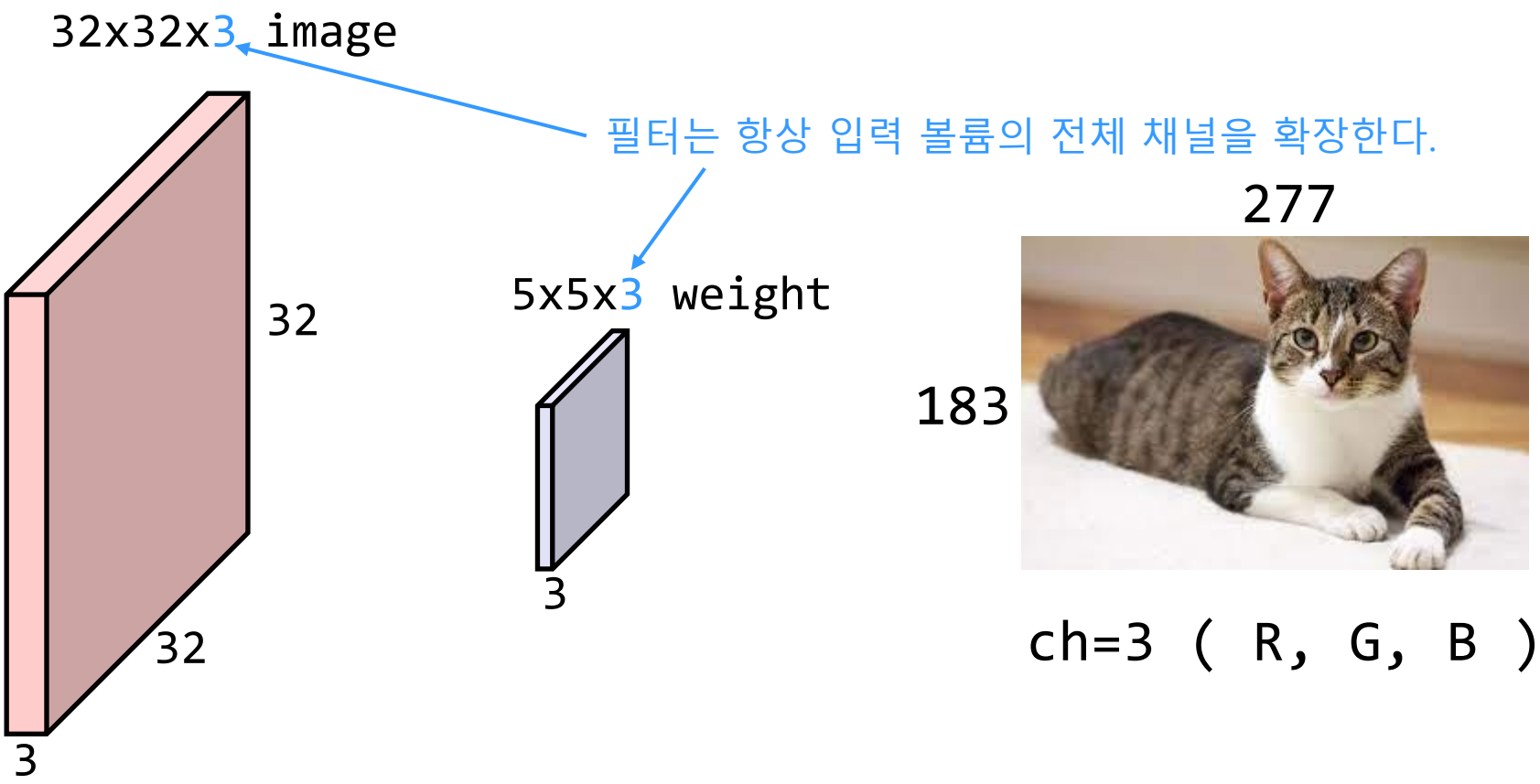


Output size :
(N - F) // stride + 1

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

Convolution Layer

image.shape => (32,32,3)

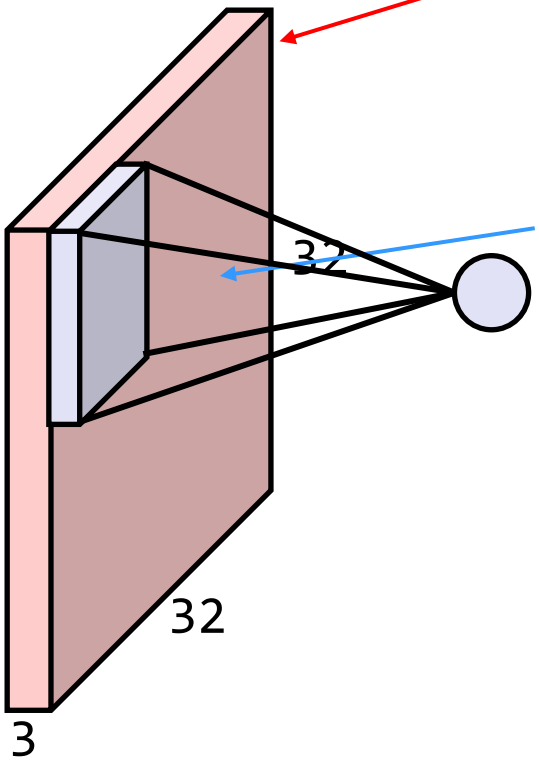


padding = valid

Convolution Layer

(32,32,3)

32x32x3 image



(5,5,3)

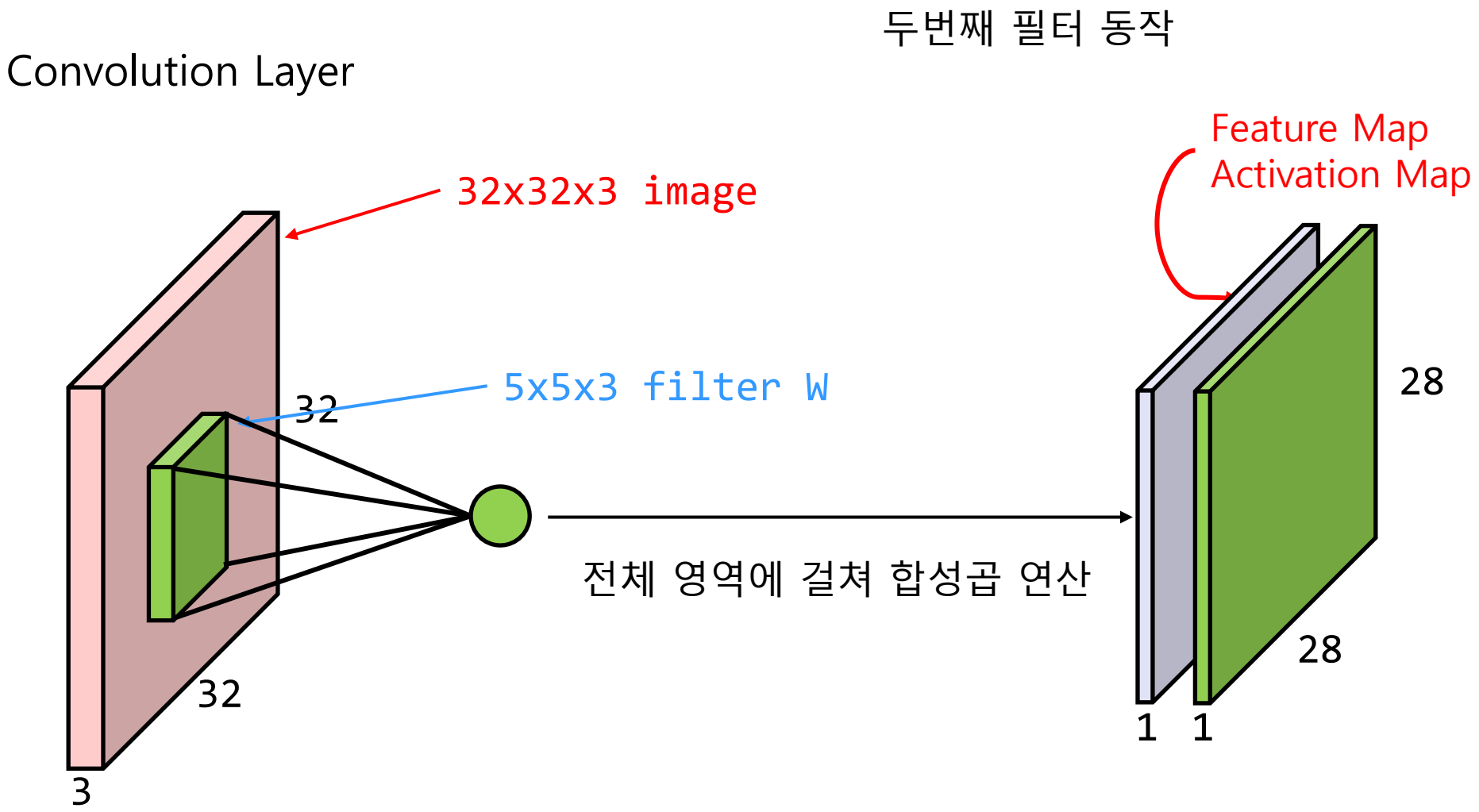
5x5x3 filter W

output.shape
(28,28,1)

1 number :
필터와 이미지의 5x5x3영역 사이에서
내적을 취한 결과

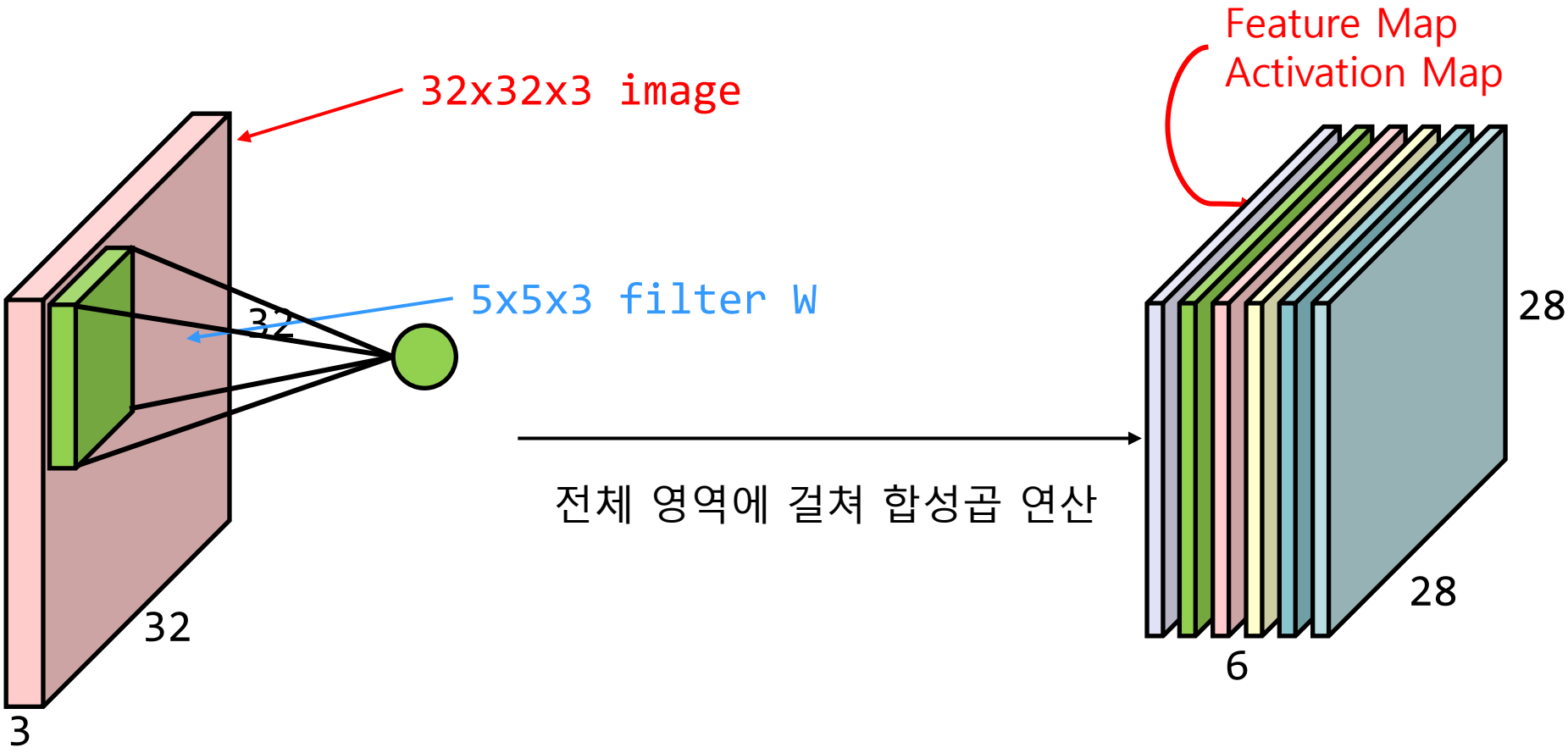
$XW + b$

Convolution Layer


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

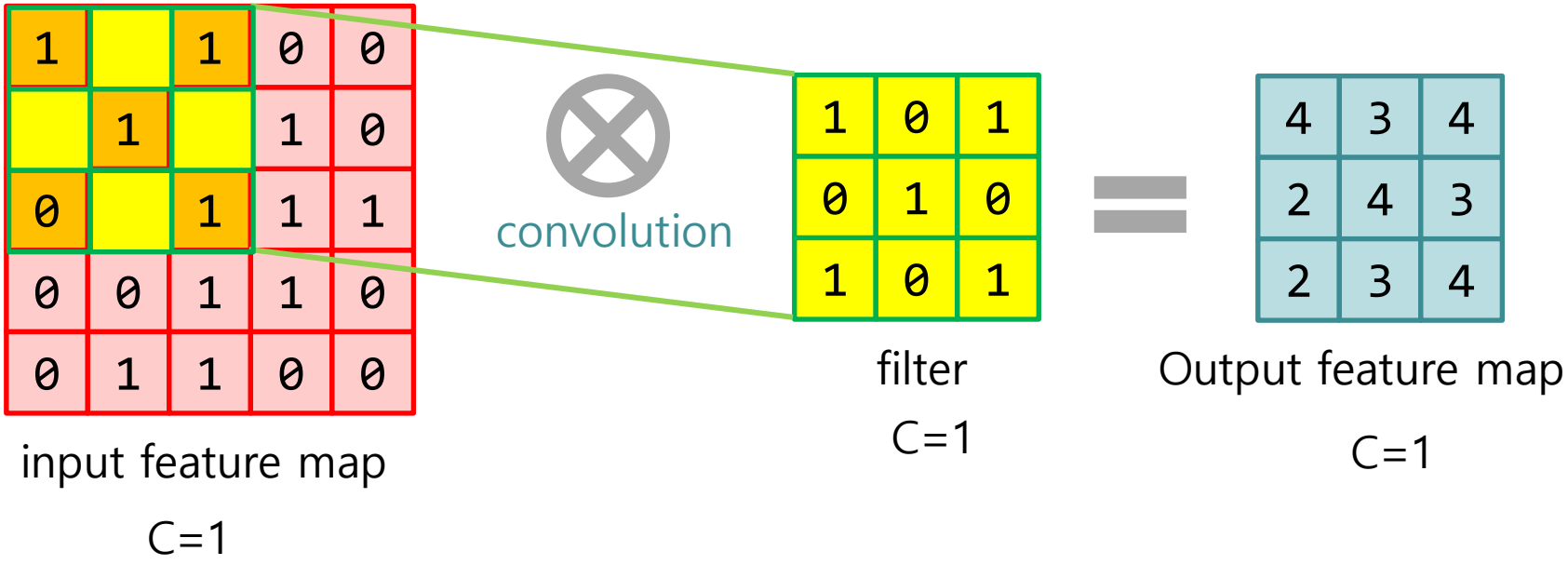
Convolution Layer

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

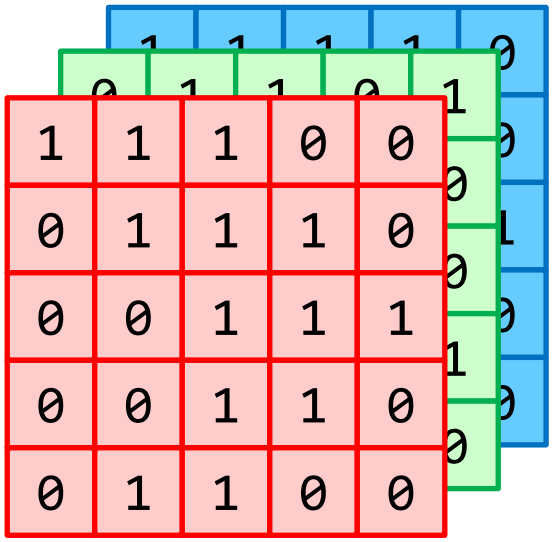


Convolution Layer - 계산

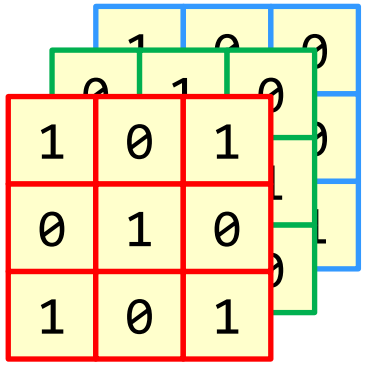
1x1 + 1x0 + 1x1 + 0x0 + 1x1 + 1x0 + 0x1 + 0x0 +1x1 = 4



Convolution Layer - 계산



convolution




```
[[ 3. -1.  3.]
 [-2.  0.  2.]
 [ 1.  3.  4.]]
```

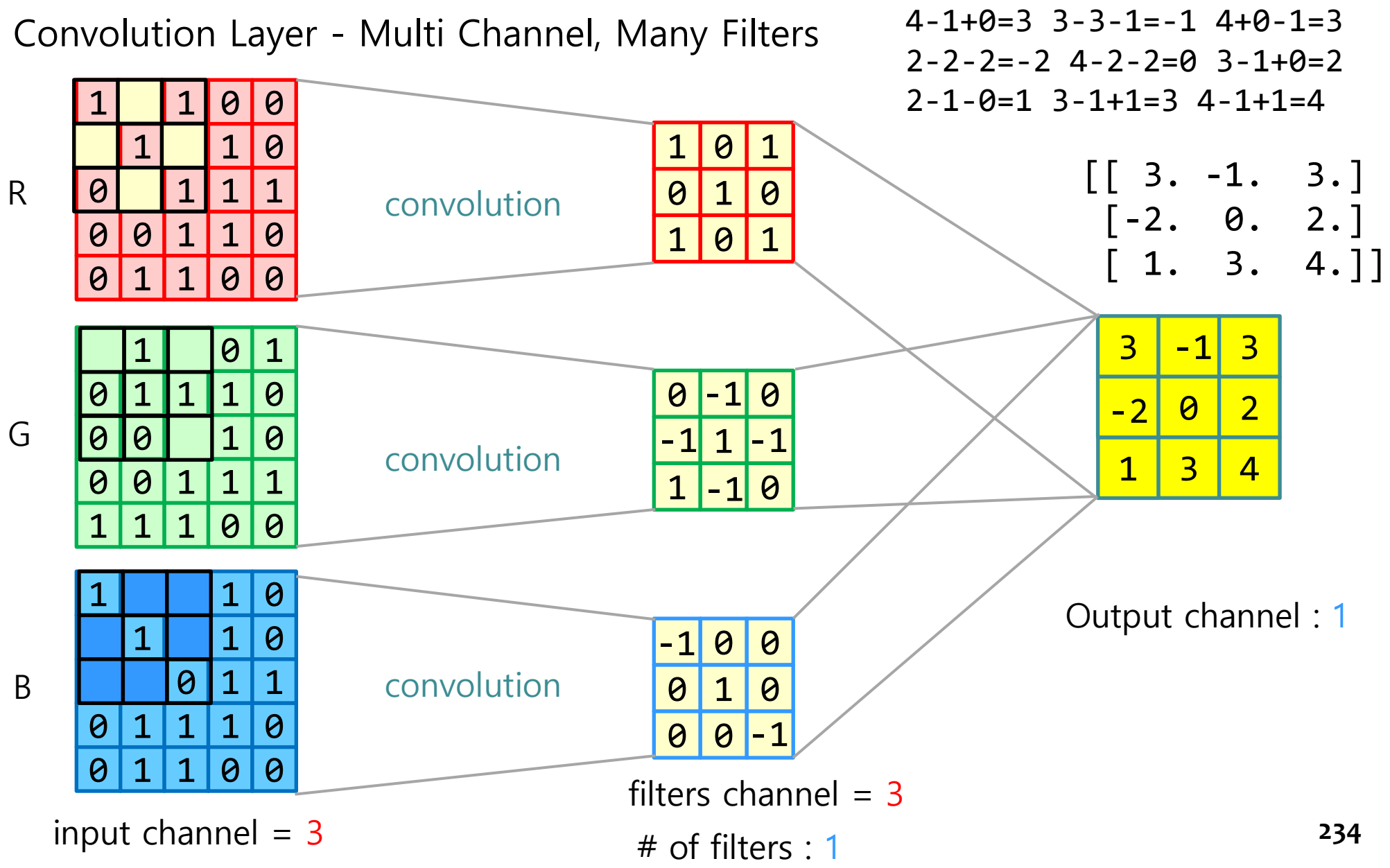
3	-1	3
-2	0	2
1	3	4

input channel = 3

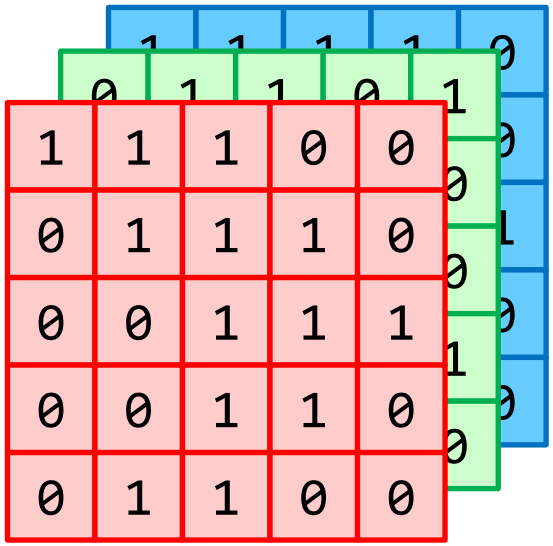
filter channel = 3

# of filters : 1

Output channel : 1

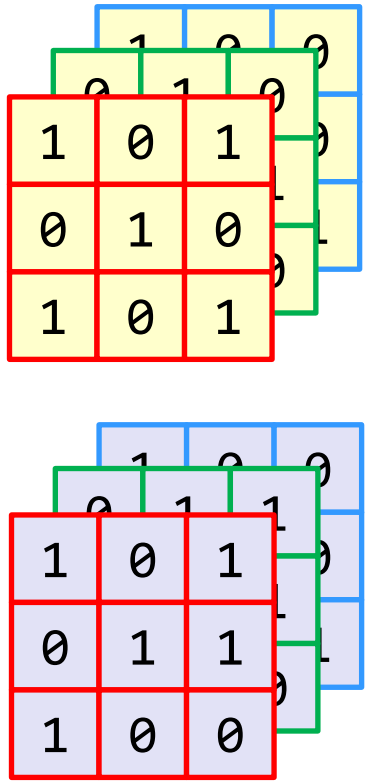

Convolution Layer - Multi Channel, Many Filters
4-1+0=3 3-3-1=-1 4+0-1=3
2-2-2=-2 4-2-2=0 3-1+0=2
2-1-0=1 3-1+1=3 4-1+1=4
[[3. -1. 3.]
[-2. 0. 2.]
[1. 3. 4.]]
R
G
B
convolution
convolution
convolution
Output channel : 1
input channel = 3
filters channel = 3
# of filters : 1

Convolution Layer - 계산



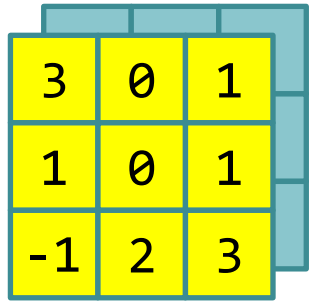
input channel = 3

convolution

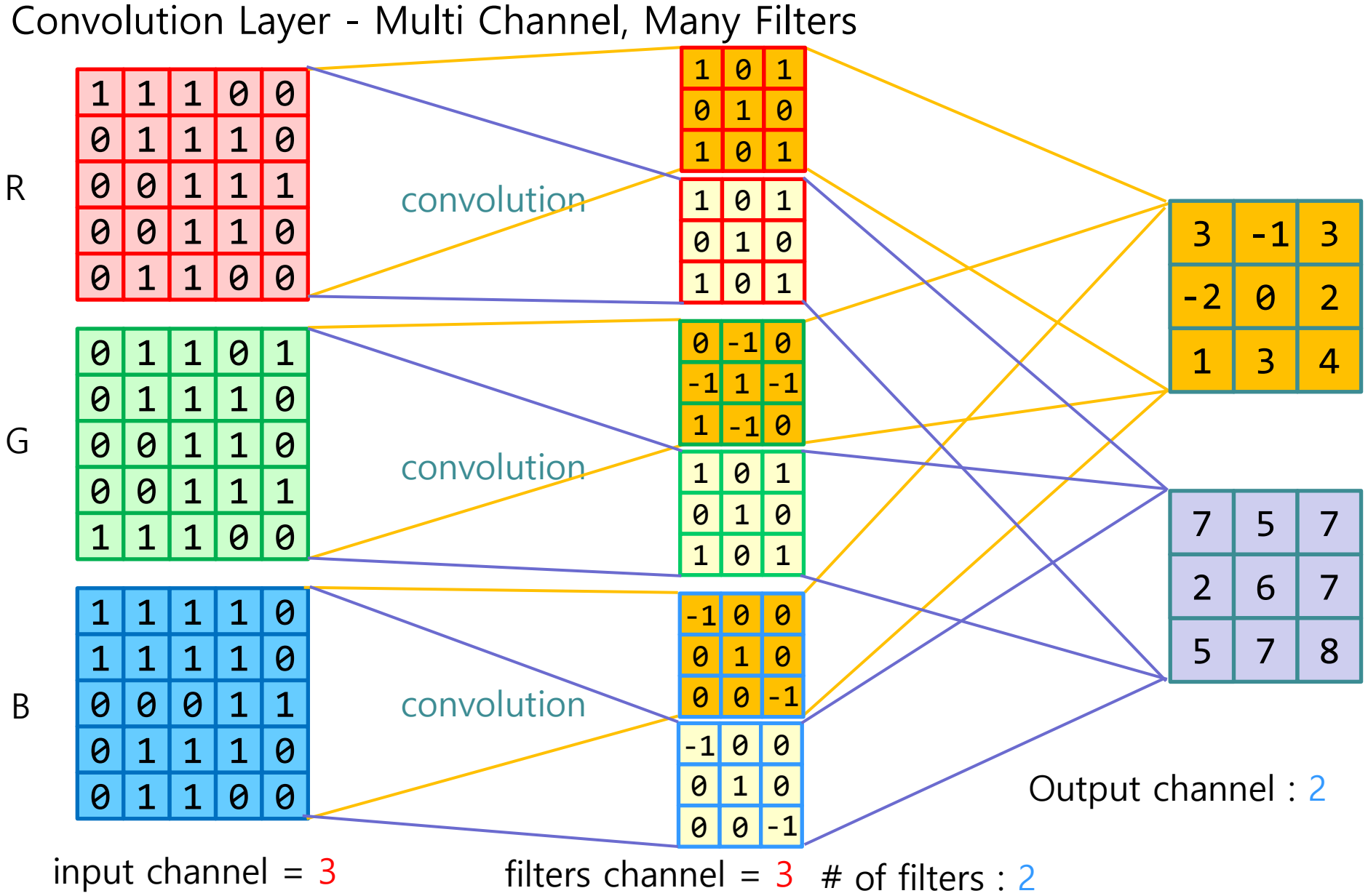


filter channel = 3
# of filters : 2

=



Output channel : 2

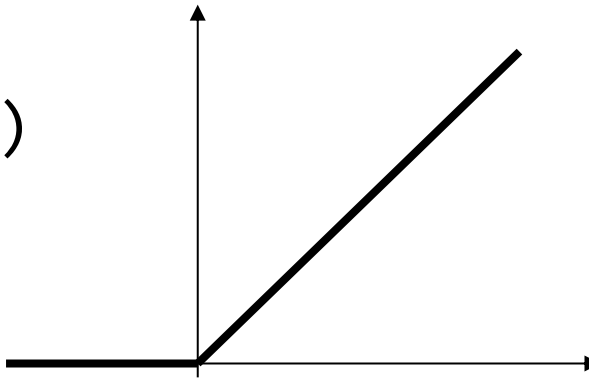

Convolution Layer - Multi Channel, Many Filters
R
G
B
convolution
convolution
convolution
Output channel : 2
input channel = 3
filters channel = 3
# of filters : 2

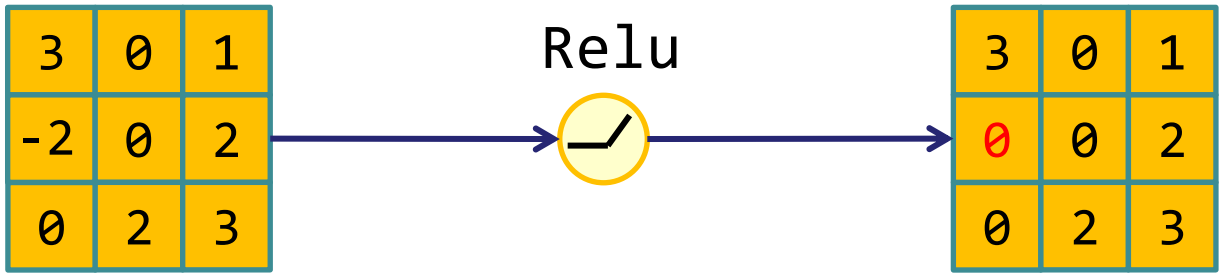
Activation Function

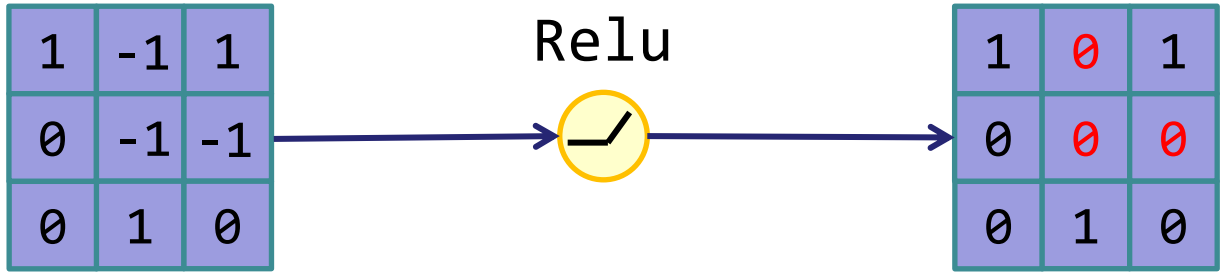
y = x

y = 0

y = max(0,x)







00000010 +2

11111110 -2 => 254

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).

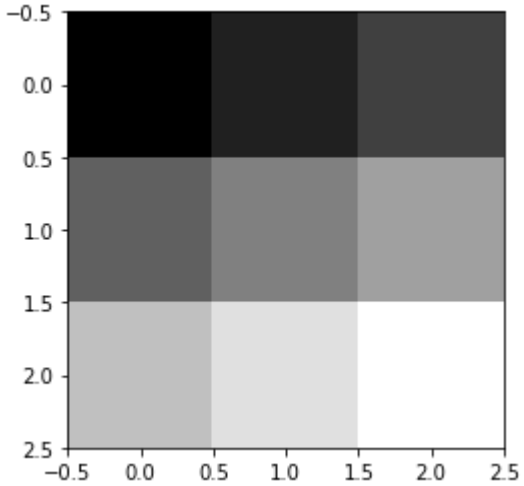
- **`data_format`**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch_size, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch_size, channels, height, 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".

tf.keras.layers.Conv2D

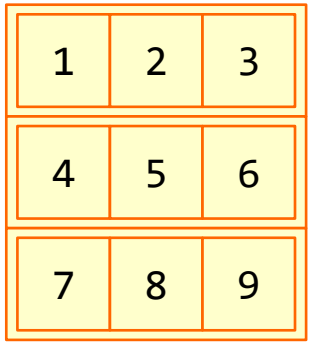
- **activation**: Activation function to use. If you don't specify anything, no activation is applied (see keras.activations).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix (see keras.initializers).
- **bias_initializer**: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

```
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)
```

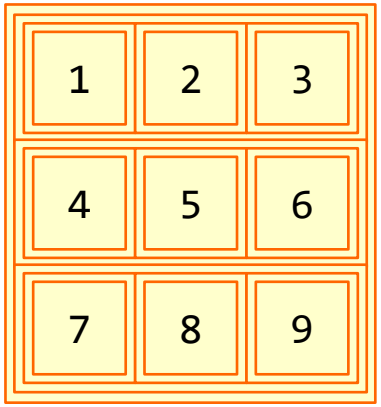


```
(3,3)

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

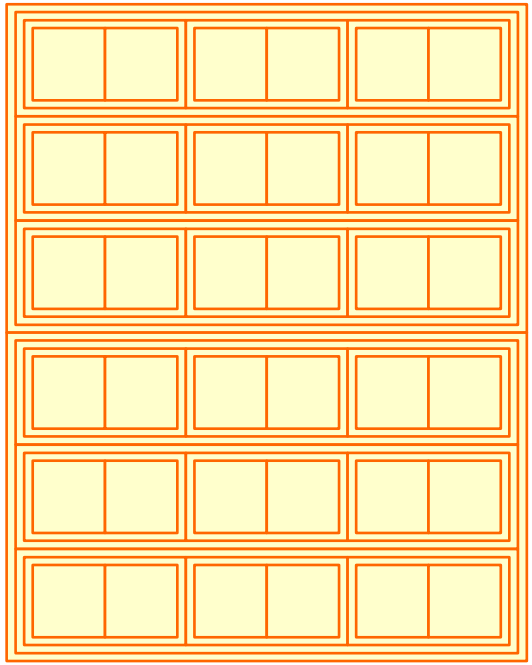


```
[[[[1],[2],[3]],
  [[4],[5],[6]],
  [[7],[8],[9]]]]
```

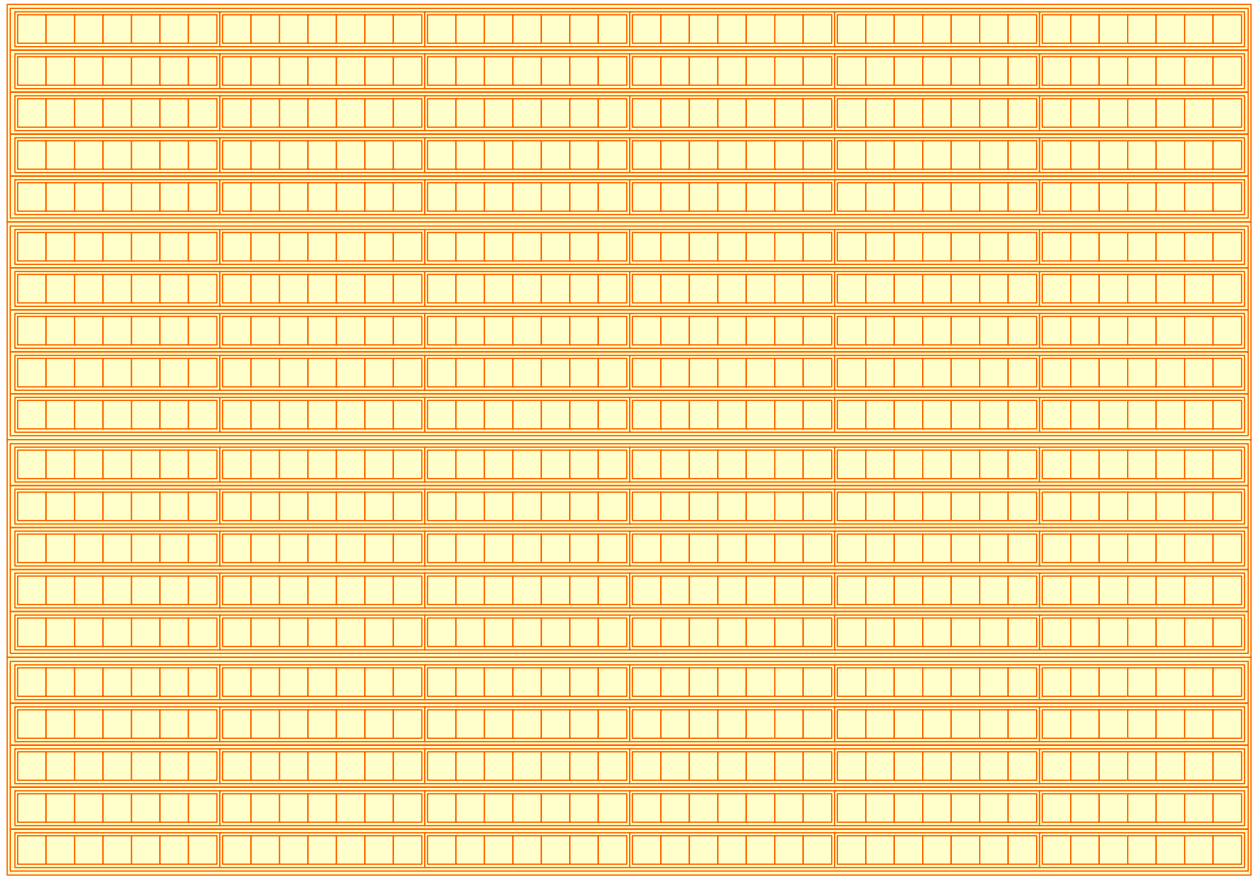
(1,3,3,1)=>
(batch_size,height,width,channel)



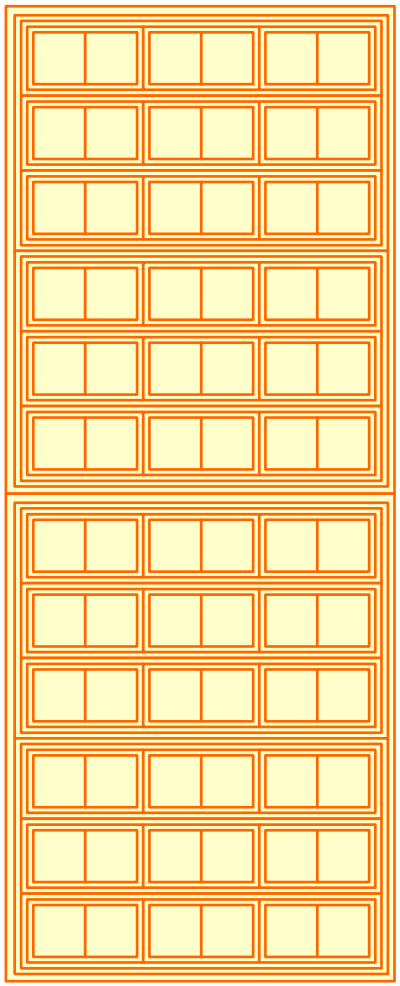
(2,3,3,2)=>



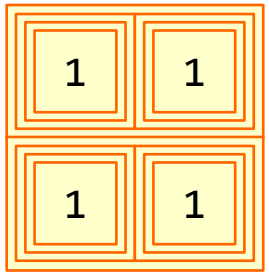
(4,5,6,7)=>



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



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



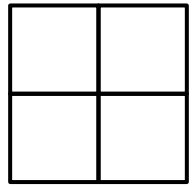
axis=0 axis=3

(2,2,1,1)=>

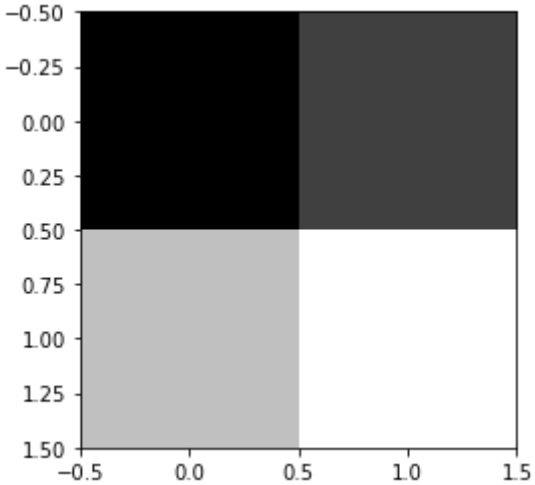
(height,width,channel,filters)

```
weight = np.array([  [ [ [ 1. ] ], [[1.]] ]  ,  [[[1.]],[[1.]]]   ])
print("weight.shape=", weight.shape)
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2, padding='valid',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
print(conv2d.numpy().reshape(2,2))
plt.imshow(conv2d.numpy().reshape(2,2), cmap='gray')
plt.show()
```

1	2	3
4	5	6
7	8	9

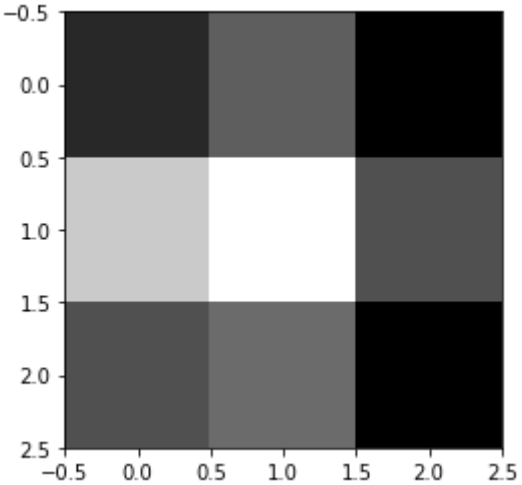


```
conv2d.shape(1, 2, 2, 1)
[[12. 16.]
 [24. 28.]]
```

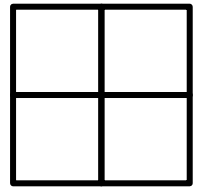


```
weight = np.array([  [ [ [ 1. ] ], [[1.]] ]  ,  [[[1.]],[[1.]]]   ])
print("weight.shape=", weight.shape)
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2, padding='same',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
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.]]
```

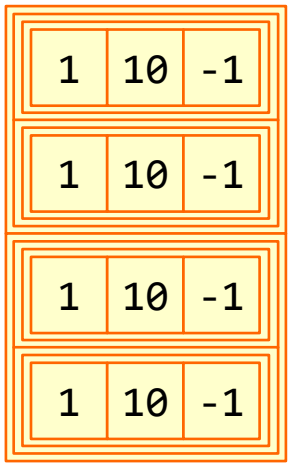


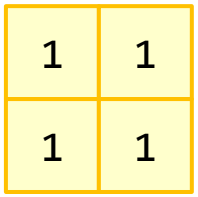
1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

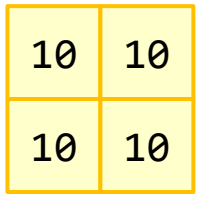


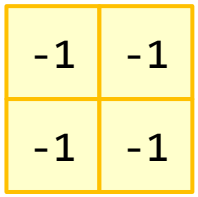
```
weight = np.array([[[[1.,10.,-1.]],[[1.,10.,-1.]]],[[[1.,10.,-1.]],[[1.,10.,-1.]]]])
print(weight.shape)
```

weight
(height,width,channel,nums)
(2,2,1,1) => (2,2,1,3)









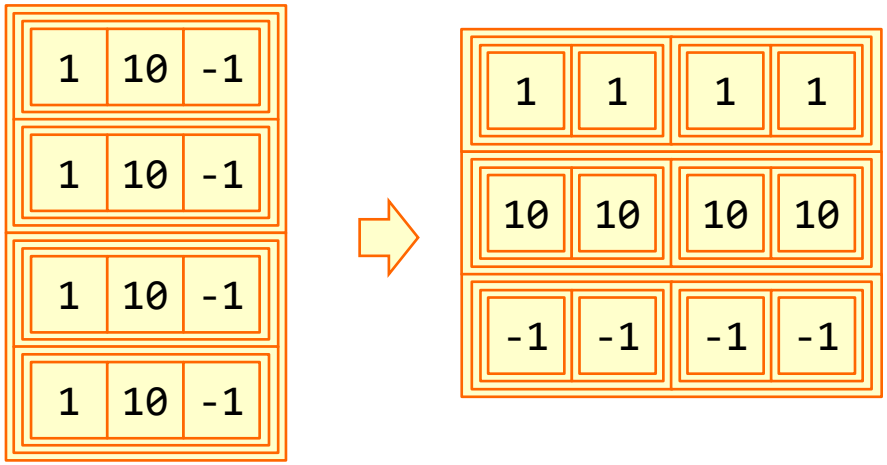
```
temp=np.transpose(weight,(3,0,1,2))
```

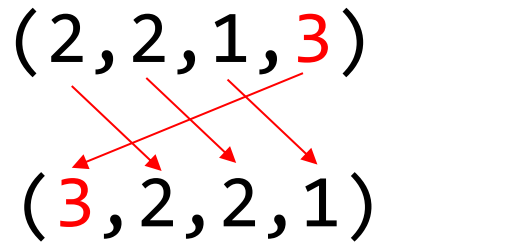
weight

(height,width,channel,nums)

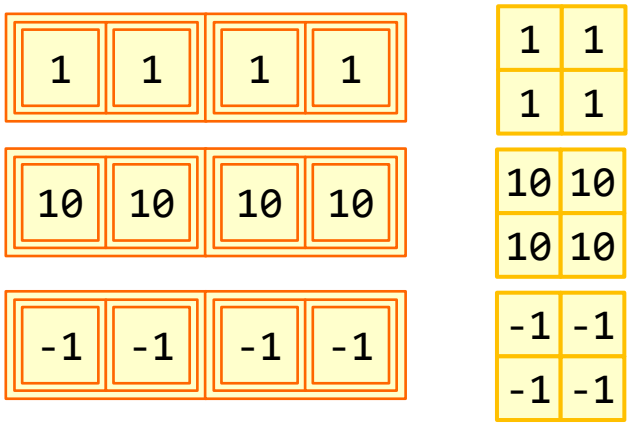
temp

weight





temp=np.transpose(weight,(3,0,1,2))



(1,3,3,1)

1	2	3
4	5	6
7	8	9

filters=3

1	1
1	1

10	10
10	10

-1	-1
-1	-1

12	16
24	28

120	160
240	280

-12	-16
-24	-28

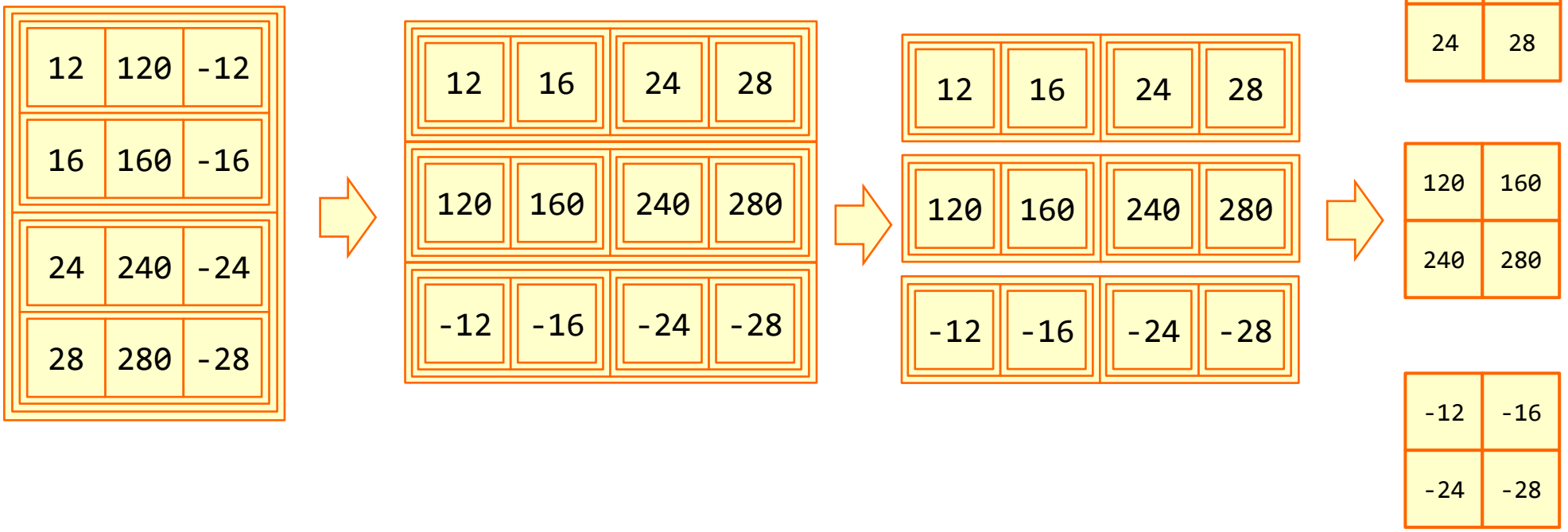
(1,2,2,3)

```
conv2d (1,2,2,3)
```

```
temp = np.swapaxes(conv2d, 0, 3)
```

(1,2,2,3)

(3,2,2,1)



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

image
(batch,height,width,channel)

weight
(height,width,channel,nums)

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

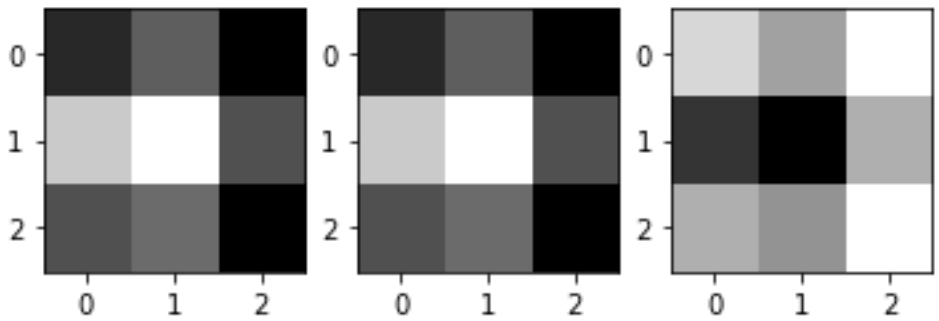
```
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]

[[120. 160.  90.]
 [240. 280. 150.]
 [150. 170.  90.]]

[[-12. -16.  -9.]
 [-24. -28. -15.]
 [-15. -17.  -9.]]
```

(1, 3, 3, 3)

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

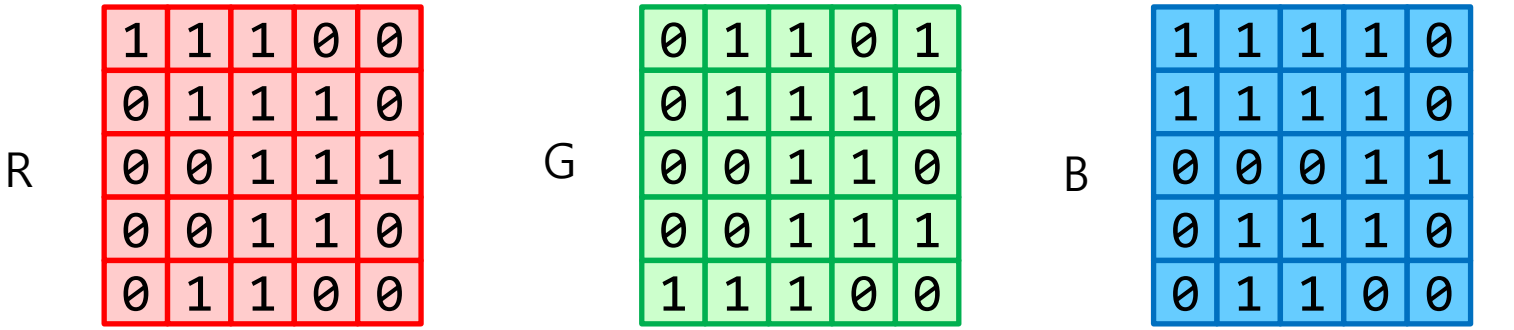
```
print("image.shpe", image.shape)
weight = np.array([[[[1.,10.,-1.]],[[1.,10.,-1.]]],[[[1.,10.,-1.]],[[1.,10.,-1.]]]])
print("weight.shpe", weight.shape)
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=3, kernel_size=2, padding='same',
kernel_initializer=weight_init)(image)
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.]]
```



```
image = tf.constant(  [[
                          [[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]]
                       ]],     dtype=np.float32)
```

(1,5,5,3)



input channel = 3

```
image = tf.constant(  [[
                         [[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]]
                       ]],     dtype=np.float32)

maps = np.swapaxes(image, 0, 3)
for i, map in enumerate(maps):
    print(map.reshape(5,5))
```

(1,5,5,3)

R

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

G

0	1	1	0	1
0	1	1	1	0
0	0	1	1	0
0	0	1	1	1
1	1	1	0	0

B

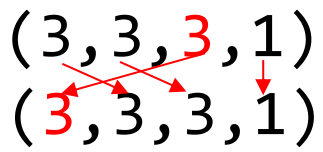
1	1	1	1	0
1	1	1	1	0
0	0	0	1	1
0	1	1	1	0
0	1	1	0	0

```
weight = np.array( [
                     [[[1],[0],[-1]], [[0],[-1],[0]], [[1],[0],[0]]],
                     [[[0],[-1],[0]], [[1],[1],[1]], [[0],[-1],[0]]],
                     [[[1],[1],[0]], [[0],[-1],[0]], [[1],[0],[-1]]]
                   ] )
# maps = np.swapaxes(weight, 1, 2)
# maps = np.swapaxes(maps, 0, 1)

maps = np.transpose(weight,(2,0,1,3))

for i, map in enumerate(maps):
    print(map.reshape(3,3))
```

(3,3,3,1)

(3,3,3,1)

(3,3,3,1)



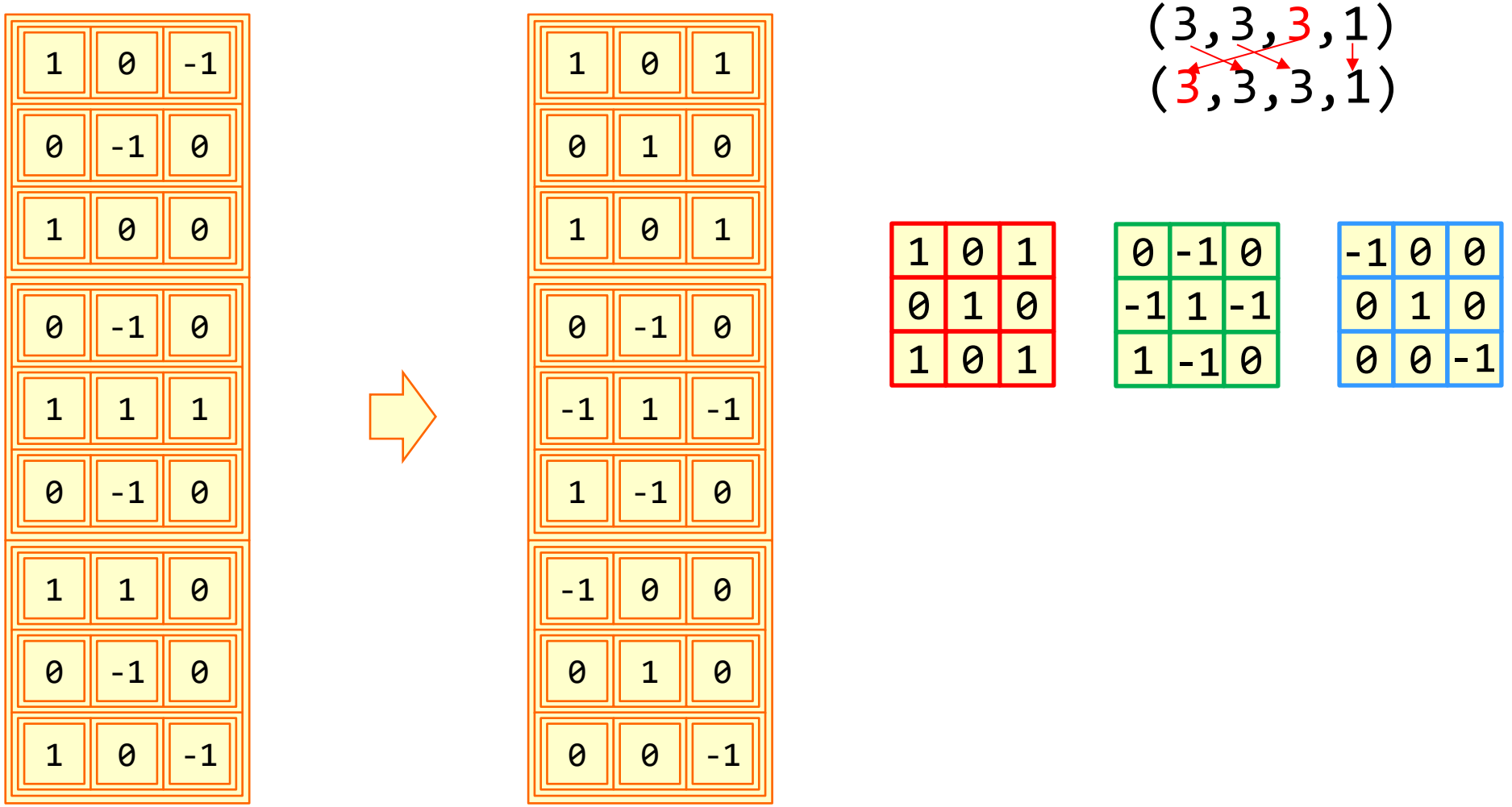

```
[[1 0 1]
 [0 1 0]
 [1 0 1]]
[[ 0 -1  0]
 [-1  1 -1]
 [ 1 -1  0]]
[[-1  0  0]
 [ 0  1  0]
 [ 0  0 -1]]
```

1	0	1
0	1	0
1	0	1

0	-1	0
-1	1	-1
1	-1	0

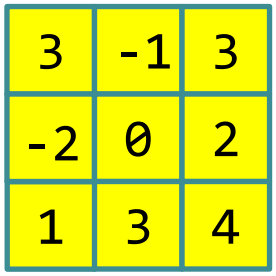
-1	0	0
0	1	0
0	0	-1

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



```
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=3, padding='valid',
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))
```

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


[[ 3. -1.  3.]
 [-2.  0.  2.]
 [ 1.  3.  4.]]
```




```
image  (1,5,5,3)
weight (3,3,3,1)
conv2d (1,3,3,1)
```

```
weight = np.array( [
                     [[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]]]
                   ] )
maps = np.swapaxes(weight, 1, 2)
maps = np.swapaxes(maps, 0, 1)

for map in maps:
    map = np.swapaxes(map, 1, 2)
    map = np.swapaxes(map, 0, 1)
    for filter in map:
       print(filter)
```

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

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

(3,3,3,2) => transpose(2,3,0,1)

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

1	0	1
0	1	0
1	0	1

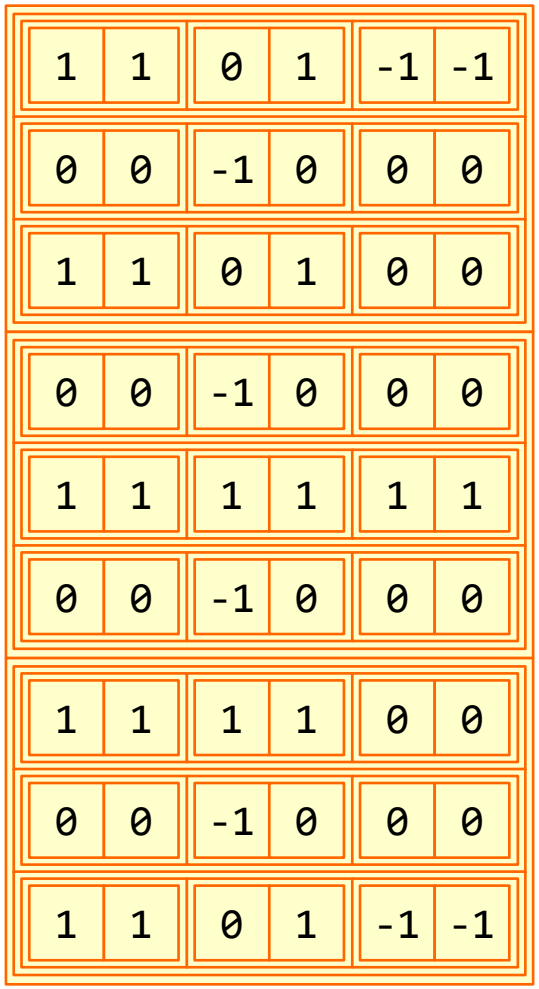
0	-1	0
-1	1	-1
1	-1	0

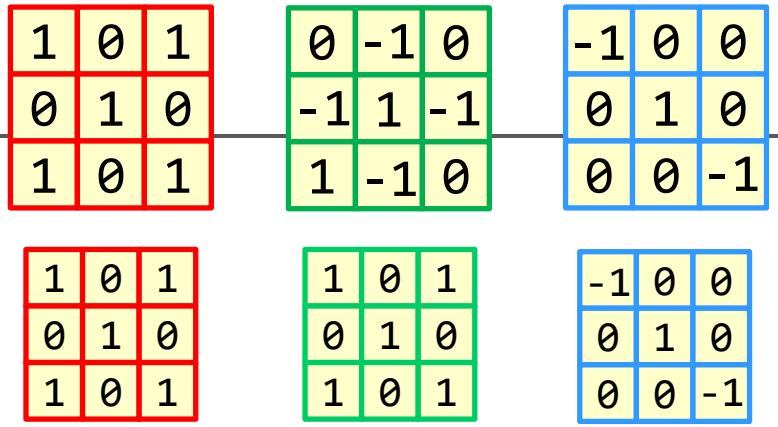
-1	0	0
0	1	0
0	0	-1

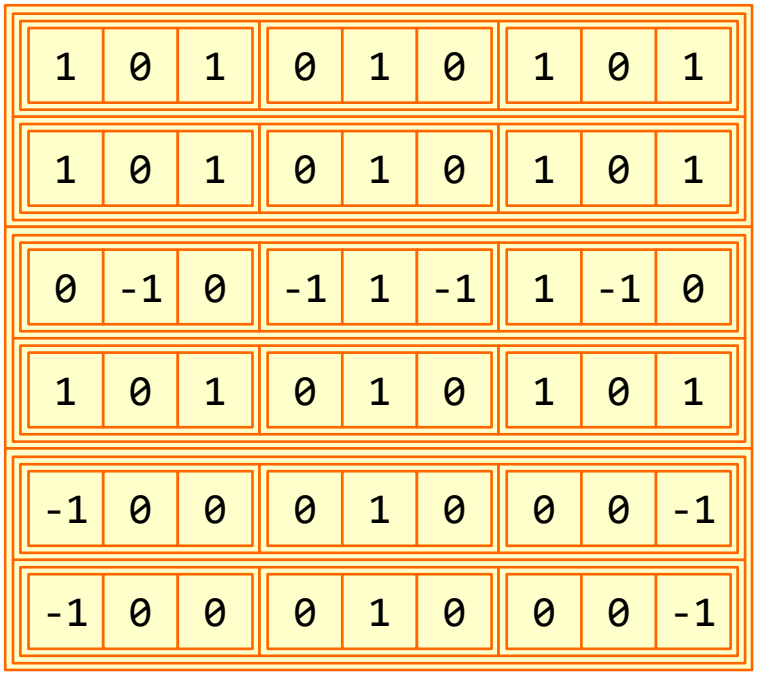
1	0	1
0	1	0
1	0	1

1	0	1
0	1	0
1	0	1

-1	0	0
0	1	0
0	0	-1


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


1 0 1
0 1 0
1 0 1
0 -1 0
-1 1 -1
1 -1 0
-1 0 0
0 1 0
0 0 -1
(3,3,3,2)
(3,2,3,3)


1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1
0 -1 0 -1 1 -1 1 -1 0
1 0 1 0 1 0 1 0 1
-1 0 0 0 1 0 0 0 -1
-1 0 0 0 1 0 0 0 -1

```
# (3,3,3,2) => (3,2,3,3)
weight = np.array( [
                     [[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]]
                   ] )

maps = np.transpose(weight, (2,3,0,1) )

for map in maps:
    for filter in map:
       print(filter)
```

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

1	0	1
0	1	0
1	0	1

0	-1	0
-1	1	-1
1	-1	0

-1	0	0
0	1	0
0	0	-1

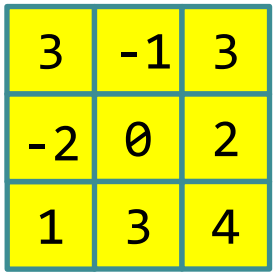
1	0	1
0	1	0
1	0	1

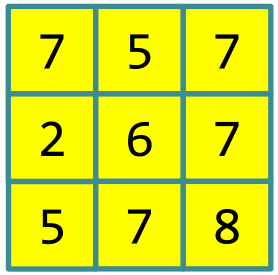
1	0	1
0	1	0
1	0	1

-1	0	0
0	1	0
0	0	-1

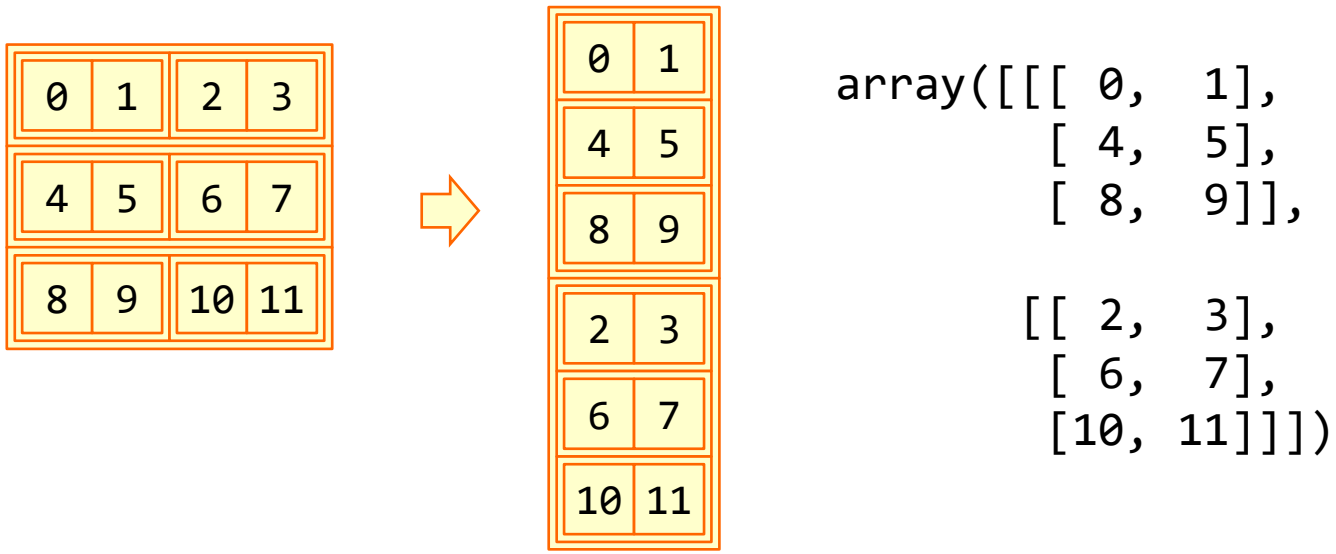
```
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=2, kernel_size=3, padding='valid',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)  # ( 1,3,3,2)
feature_maps = np.swapaxes(conv2d, 0, 3)
for feature_map in feature_maps:
    print(feature_map.reshape(3,3))
```

```
[[ 3. -1.  3.]
 [-2.  0.  2.]
 [ 1.  3.  4.]]
[[7. 5. 7.]
 [2. 6. 7.]
 [5. 7. 8.]]
```



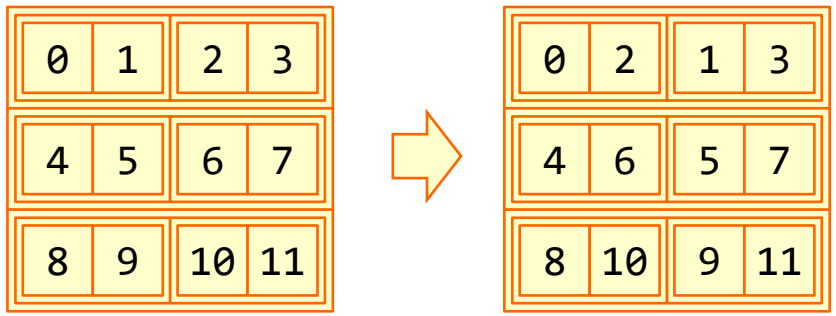


```
a = np.arange(12).reshape(3,2,2)
b = np.swapaxes(a, 0, 1)#(2,3,2)
```



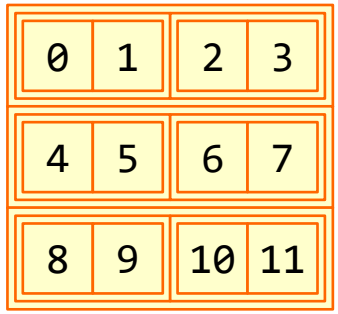

```
array([[[ 0,  1],
        [ 4,  5],
        [ 8,  9]],

       [[ 2,  3],
        [ 6,  7],
        [10, 11]]])
```

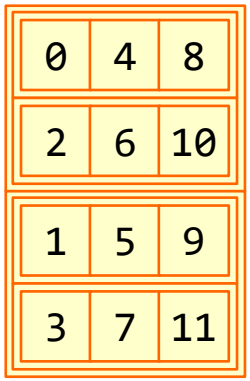
```
a = np.arange(12).reshape(3,2,2)
b = np.swapaxes(a, 1, 2)#(3,2,2)
```




```
array([[[ 0,  2],
        [ 1,  3]],

       [[ 4,  6],
        [ 5,  7]],

       [[ 8, 10],
        [ 9, 11]]])
```

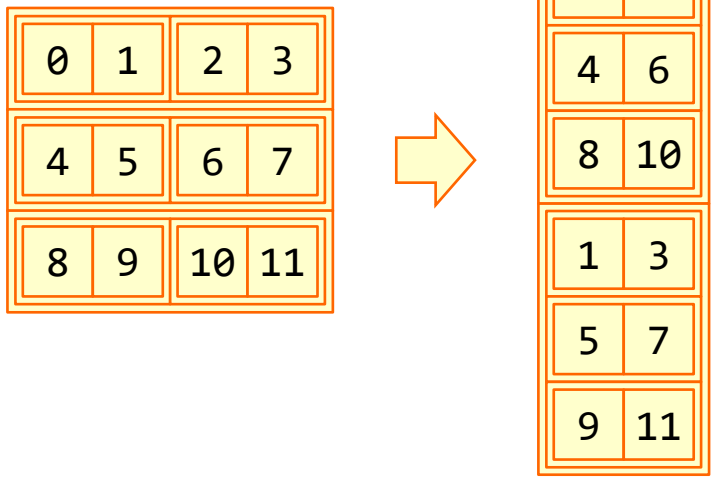
```
a = np.arange(12).reshape  (3,2,2)
b = np.swapaxes(a, 0, 2) # (2,2,3)
```






```
array([[[ 0,  4,  8],
        [ 2,  6, 10]],

       [[ 1,  5,  9],
        [ 3,  7, 11]]])
```

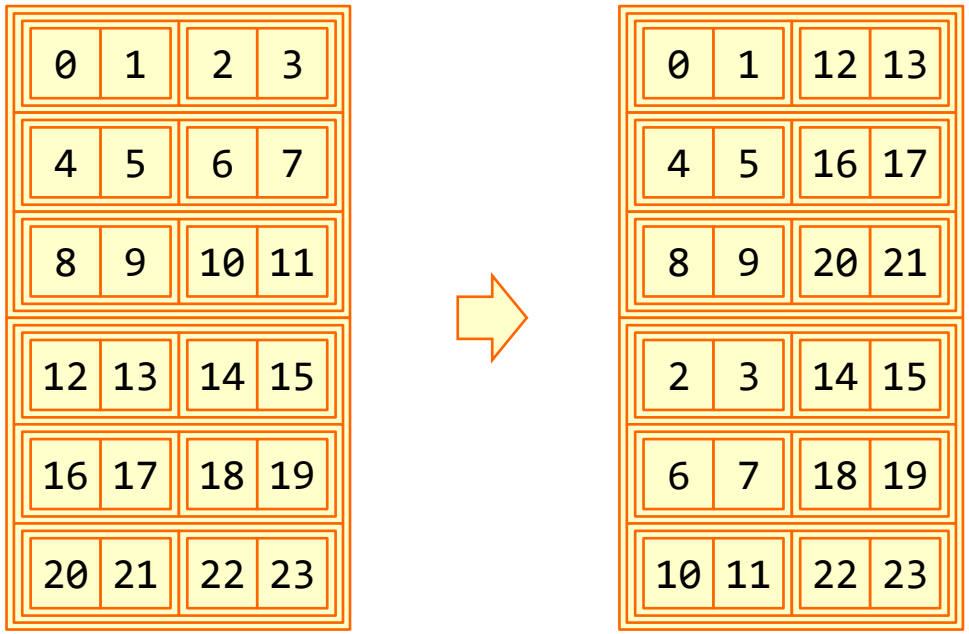
```
a = np.arange(12).reshape(3,2,2)
b = np.transpose(a, (2,0,1))
```



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

```
array([[[ 0,  2],
        [ 4,  6],
        [ 8, 10]],

       [[ 1,  3],
        [ 5,  7],
        [ 9, 11]]])
```

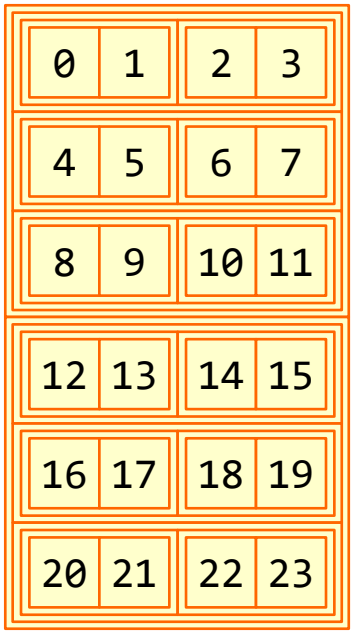
```
a = np.arange(24).reshape  (2,3,2,2)
b = np.swapaxes(a, 0, 2) # (2,3,2,2)
```

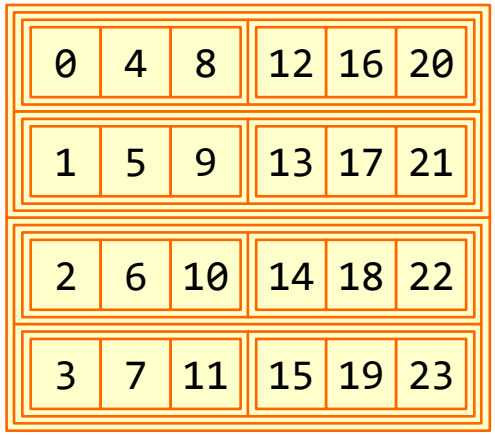



```
array([[[[ 0,  1],
         [12, 13]],

        [[ 4,  5],
         [16, 17]],

        [[ 8,  9],
         [20, 21]]],


       [[[ 2,  3],
         [14, 15]],

        [[ 6,  7],
         [18, 19]],

        [[10, 11],
         [22, 23]]]])
```

```
a = np.arange(24).reshape(2,3,2,2)
b = np.transpose(a, (2,3,0,1))
```

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

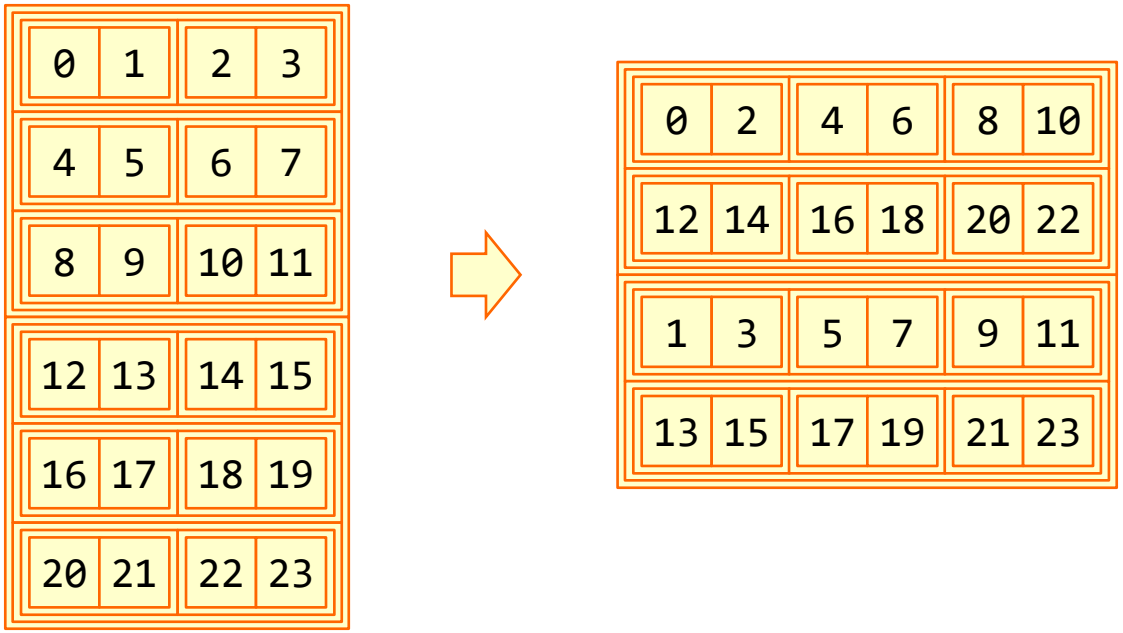





```
array([[[[ 0,  4,  8],
         [12, 16, 20]],

        [[ 1,  5,  9],
         [13, 17, 21]]],


       [[[ 2,  6, 10],
         [14, 18, 22]],

        [[ 3,  7, 11],
         [15, 19, 23]]]])
```

```
a = np.arange(24).reshape(2,3,2,2)
b = np.transpose(a, (3,0,1,2))
```

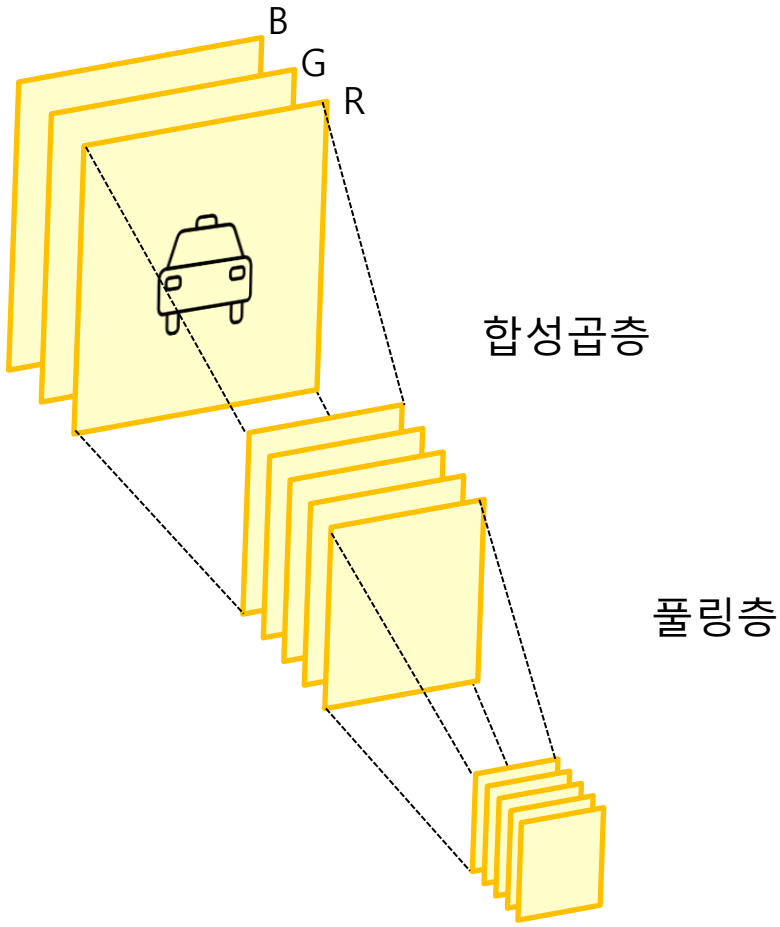
(2,3,2,2)
(2,2,3,2)




```
array([[[[ 0,  2],
         [ 4,  6],
         [ 8, 10]],

        [[12, 14],
         [16, 18],
         [20, 22]]],


       [[[ 1,  3],
         [ 5,  7],
         [ 9, 11]],

        [[13, 15],
         [17, 19],
         [21, 23]]]])
```

풀링 연산

합성곱층과 풀링층을 거치면서 변환되는 과정



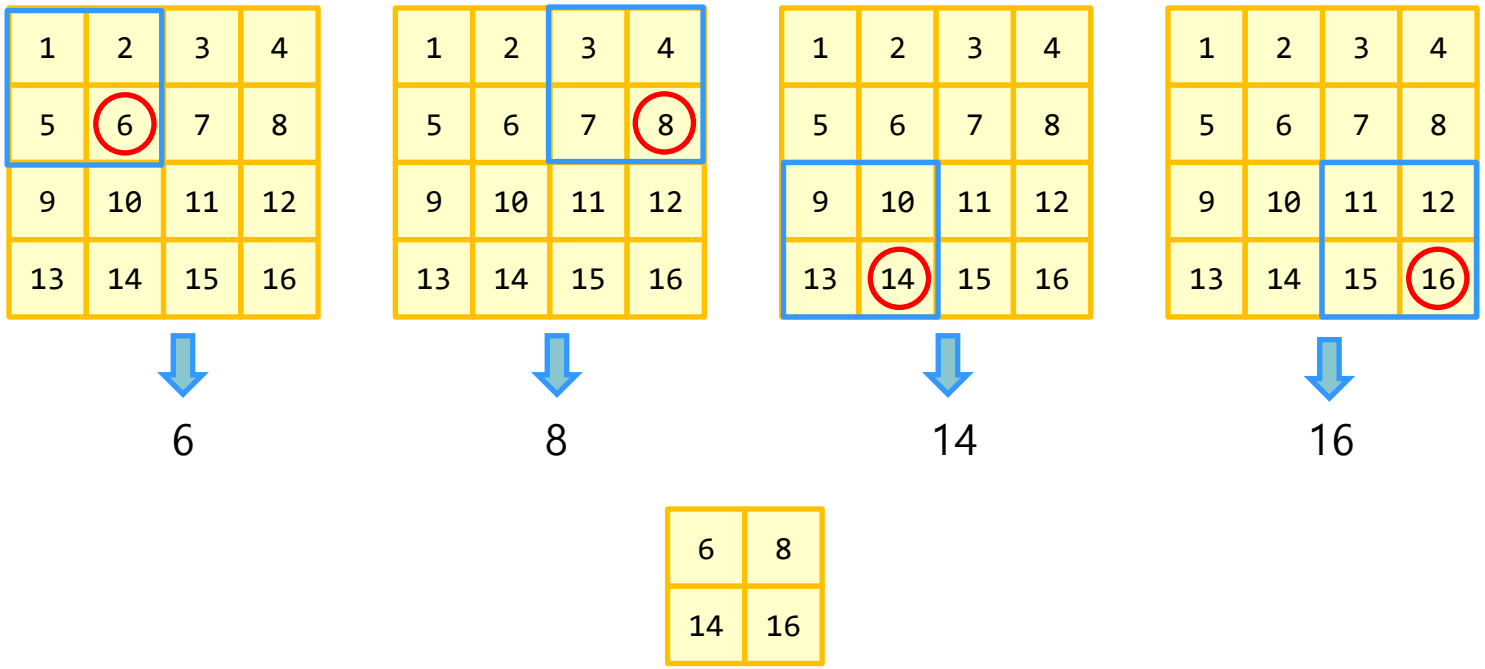
풀링 연산

$$(N-F)/S+1=Out$$

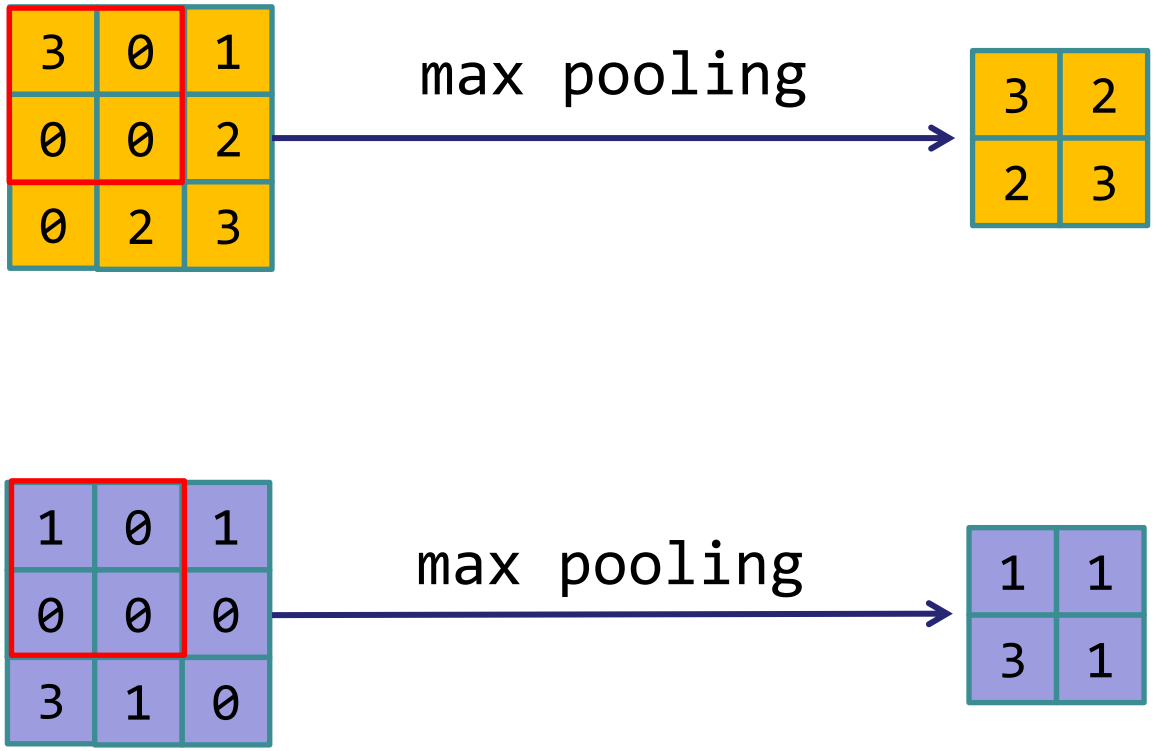
$$(4-2)/2+1=2$$

풀링이란? 특성 맵을 스캔하며 최대값을 고르거나 평균값을 계산하는 것을 말함

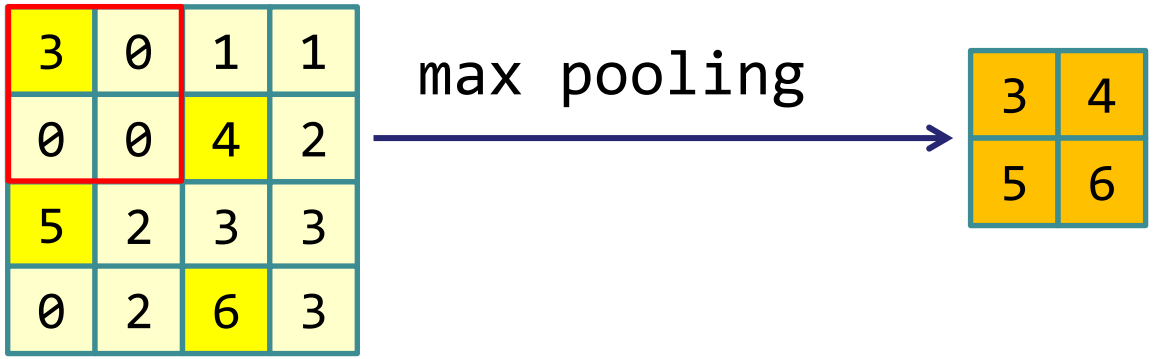
최대 풀링



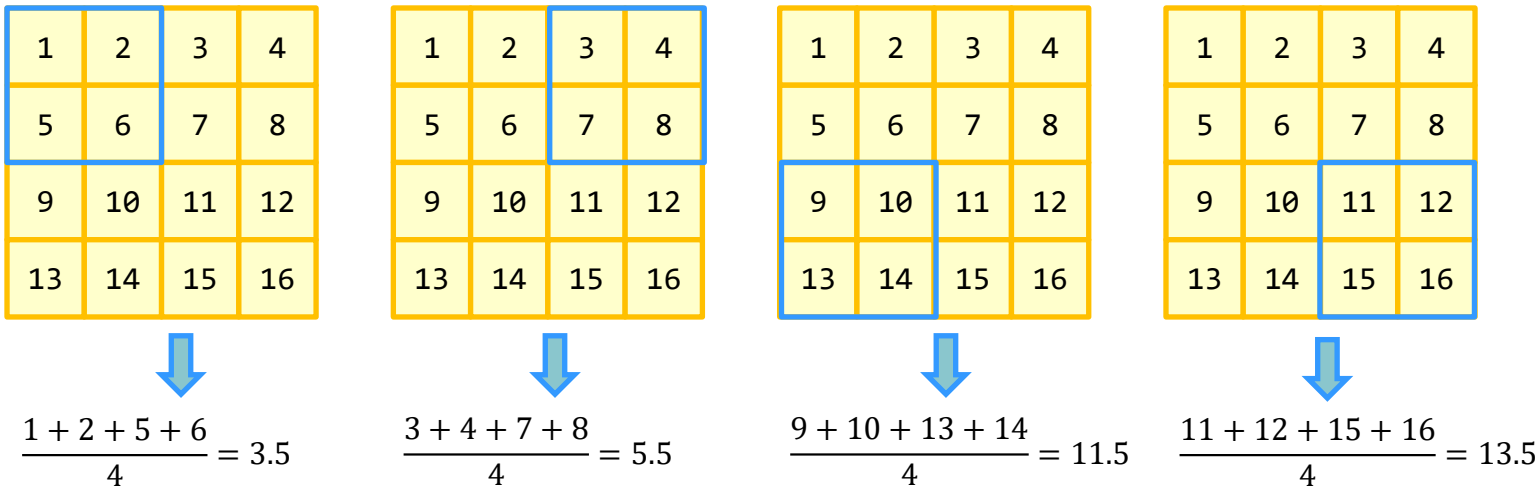
Pooling(max pooling, 2x2 filter, stride 1)



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



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



tf.keras.layers.MAXPool2D

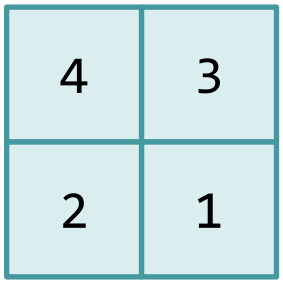
- **`pool_size`**: integer or tuple of 2 integers, window size over which to take the maximum. `(2, 2)` will take the max value over a 2x2 pooling window. If only one integer is specified, the same window length will be used for both dimensions.
- **`strides`**: Integer, tuple of 2 integers, or None. 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".

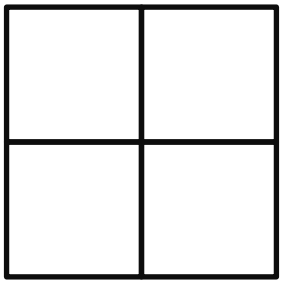
풀링 연산

```
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())
```

(1, 2, 2, 1)

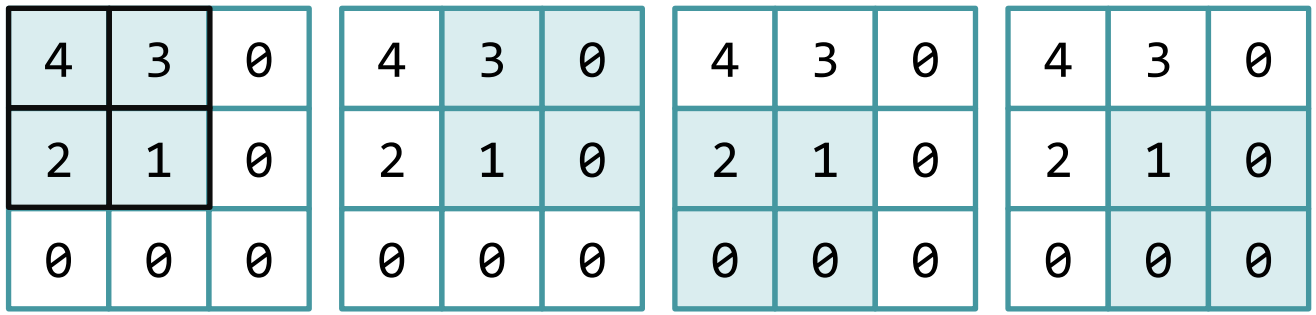
(1,1,1,1)
[[[[4.]]]]





```
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.]]]]
```

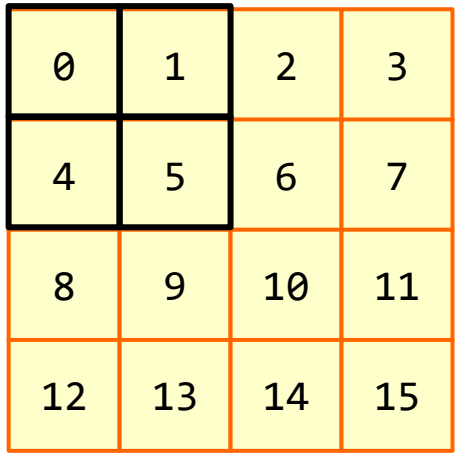


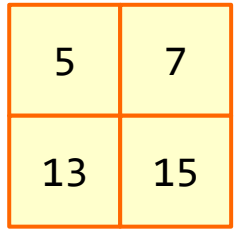
풀링 연산

```
image = tf.constant([[[[0],[1],[2],[3]],
                      [[4],[5],[6],[7]],
                      [[8],[9],[10],[11]],
                      [[12],[13],[14],[15]]]], dtype=np.float32)
pool = keras.layers.MaxPool2D(pool_size=(2,2), strides=2, padding='valid')(image)
print(pool.shape)
print(pool.numpy())
```

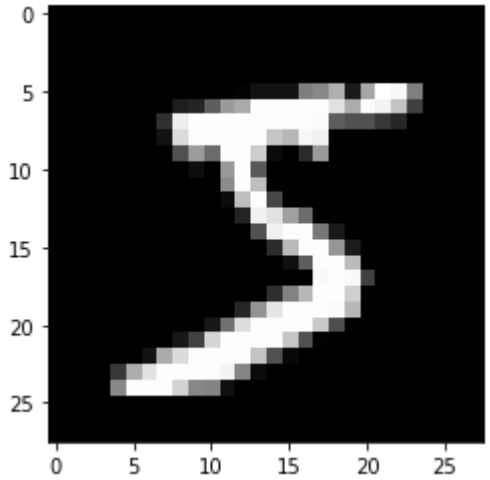
(1, 4, 4, 1)

(1, 2, 2, 1)



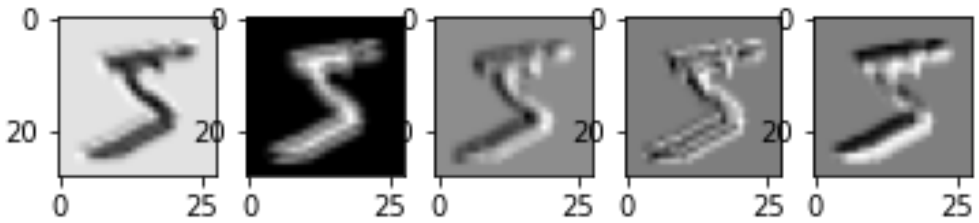


```
mnist = tf.keras.datasets.mnist
class_names = ['0','1','2','3','4','5','6','7','8','9']

(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()
```



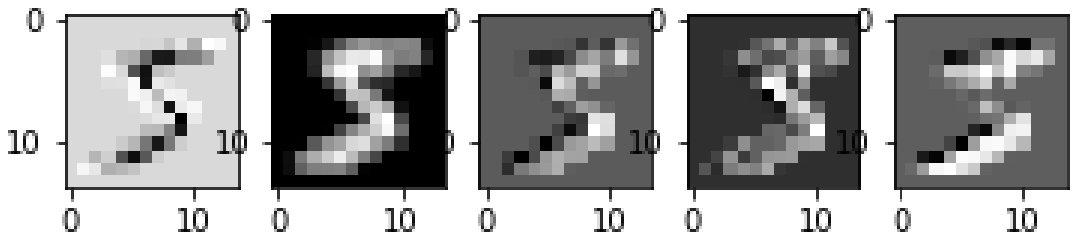
```
img = img.reshape(-1,28,28,1)
img = tf.convert_to_tensor(img)

print("weight.shape", weight.shape)
weight_init = keras.initializers.RandomNormal(stddev=0.01)
conv2d = keras.layers.Conv2D(filters=5, kernel_size=3,
                             padding='same',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(28,28), cmap='gray')
plt.show()
```

```
weight.shape (2, 2, 1, 5)
conv2d.shape (1, 28, 28, 5)
```



```
pool = keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding='valid')(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(14,14), cmap='gray')
plt.show()
```

```
(1, 28, 28, 5)
(1, 14, 14, 5)
```

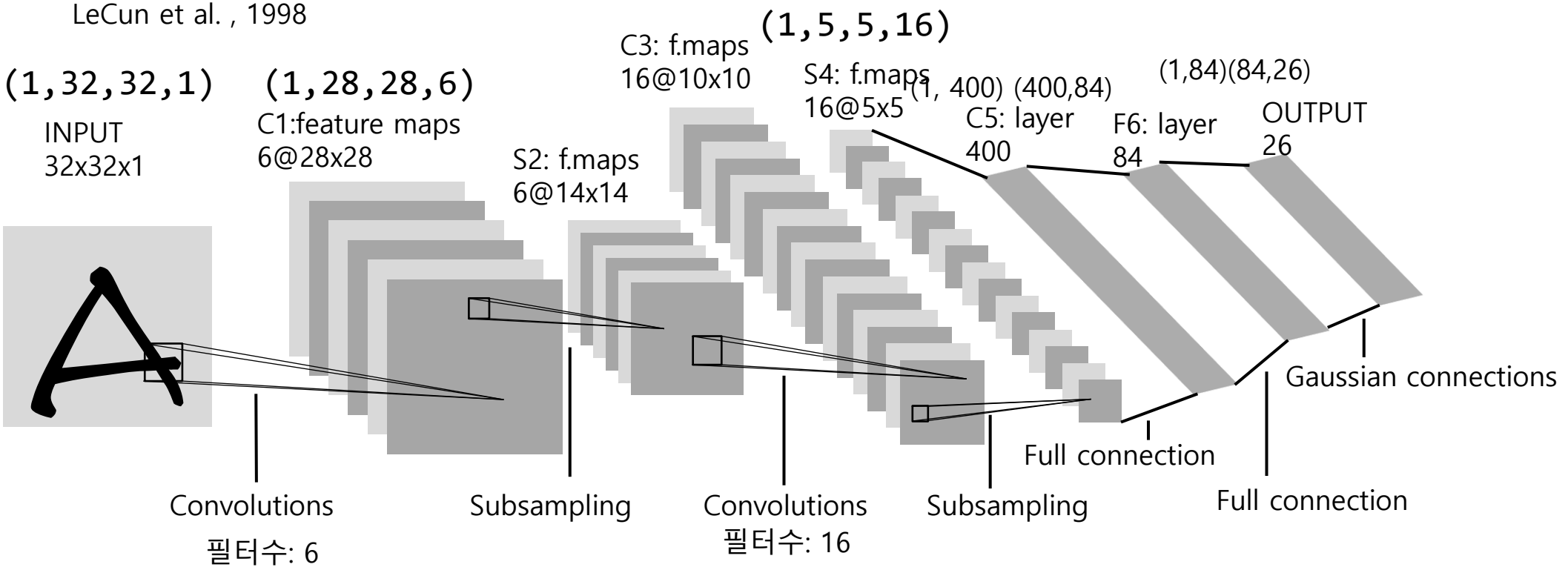


전체 합성곱 구조

LeNet-5

(1,5,5,16)=>(1,-1)=>(1,400)(400,84)=>(1,84)(84,26)=>(1,26)

LeCun et al. , 1998



Conv filters 5x5, stride 1
Subsampling 2x2, stride 2

활성화 함수

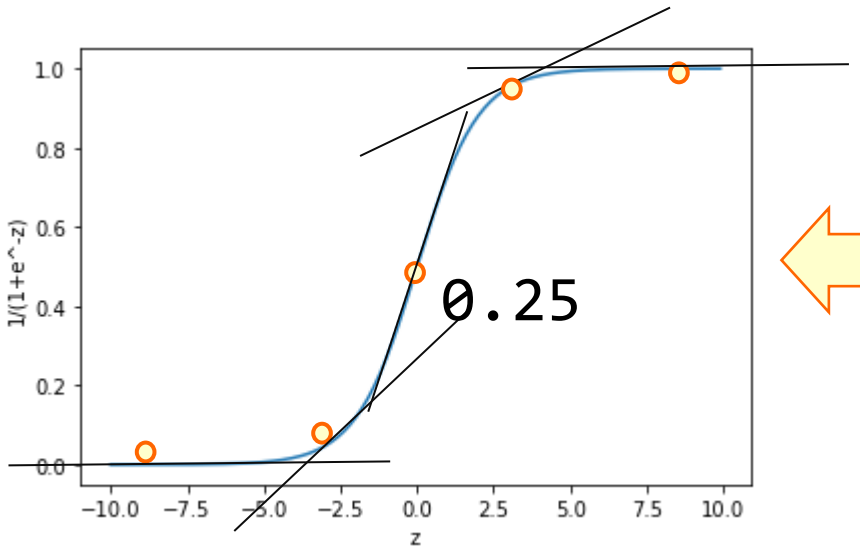
활성함수는 네트워크에 비선형성(nonlinearity)을 추가하기 위해 사용됨
- 활성함수 없이 layer를 쌓은 네트워크는 1-layer 네트워크와 동일하기 때문에 활성 함수는 비성형함수로 불리기도 한다.
- 멀티레이어 퍼셉트론을 만들 때 활성함수를 사용하지 않으면 쓰나마나이다.

1000*0.25=250*0.25=

1. 시그모이드 함수 (Sigmoid Function)

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

$$a(1-a)$$



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

- 문제점

1) 그레디언트가 죽는 현상이 발생한다 (Gradient vanishing 문제)

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

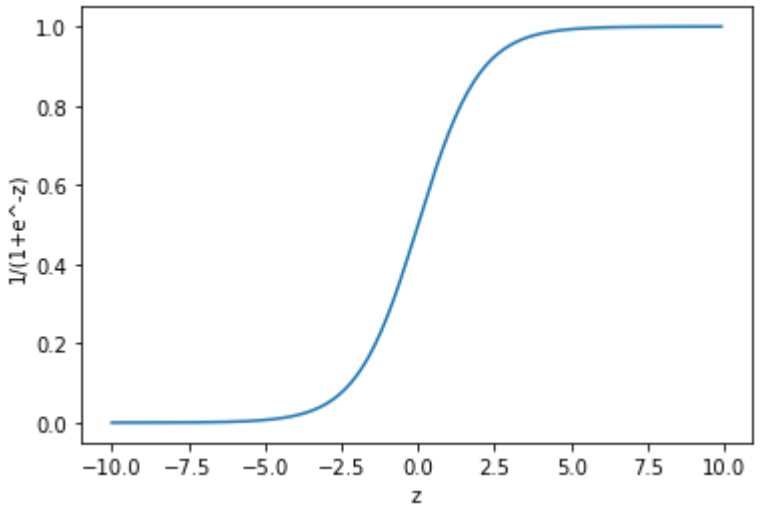
2) 활성함수의 결과 값의 중심이 0이 아닌 0.5이다.

3) 계산이 복잡하다 (지수함수 계산)

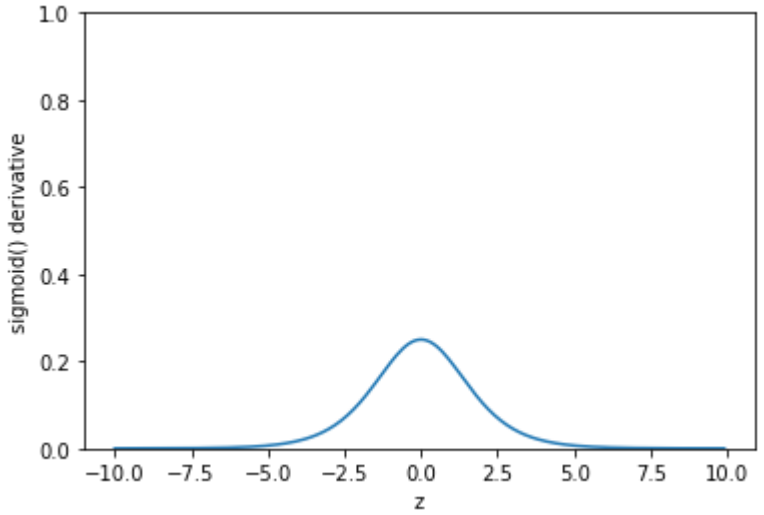
!! Gradient Vanishing

- 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

시그모이드 함수



시그모이드 미분값

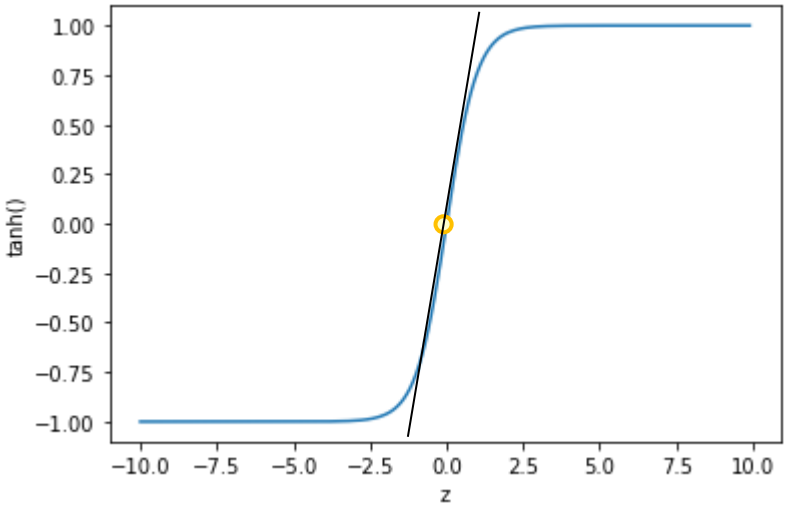


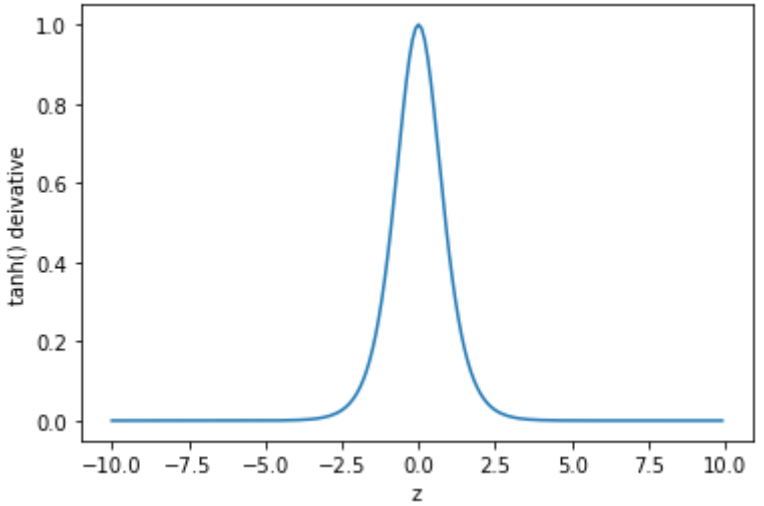
양쪽 꼬리가 0에 수렴하며 최대값이 0.25를 넘지 않는다.

2. 하이퍼 볼릭 탄젠트(tanh)

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

$$\tanh(x)' = \big(1 - \tanh(x)\big)\big(1 + \tanh(x)\big) = 1 - H^2$$



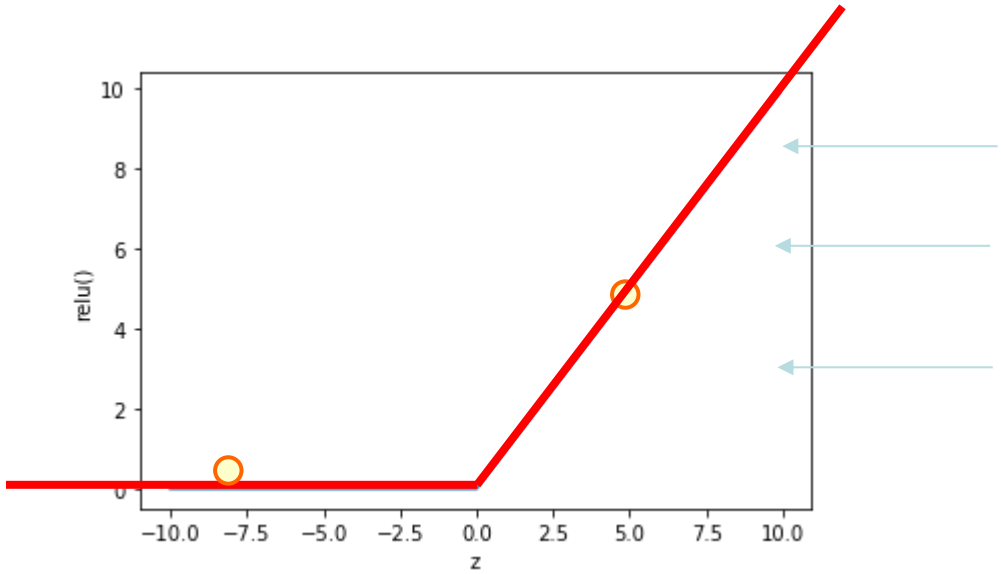


- 결과값이 [-1, 1] 사이로 제한됨. 결과값 중심이 0이다.
- 나머지 특성은 시그모이드와 비슷함. 시그모이드 함수를 이용하여 유도 가능
- 그러나, 여전히 gradient vanishing 문제가 발생

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

3. 렐루(ReLU, Rectified Linear Unit)

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

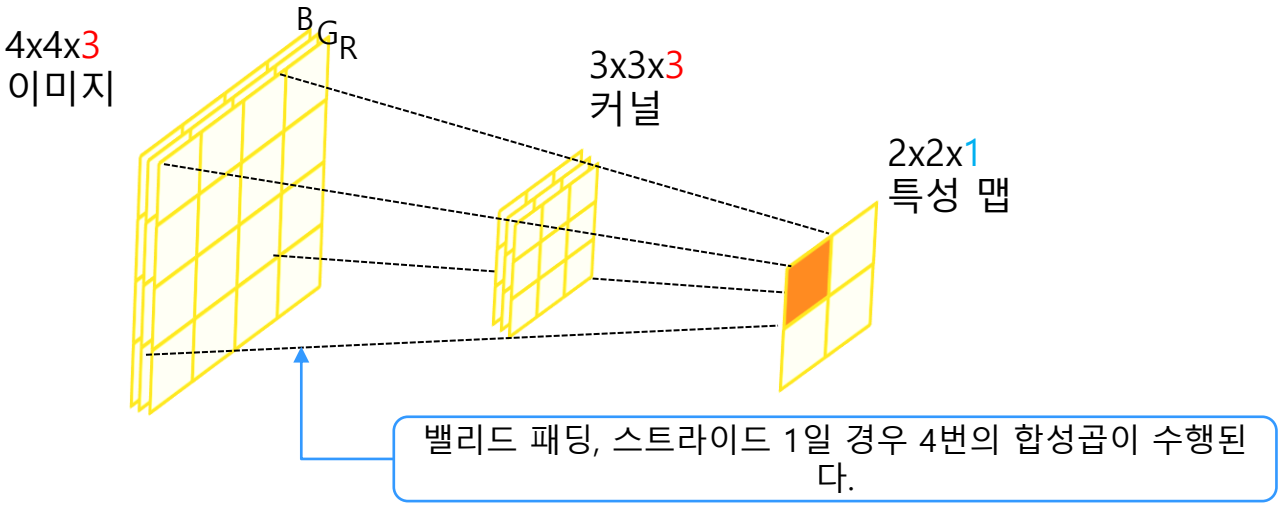


최근 뉴럴 네트워크에서 가장 많이 쓰이는 활성 함수
선형아니야? NO! 0에서 확 꺾이기 때문에 비선형이라고 본다.

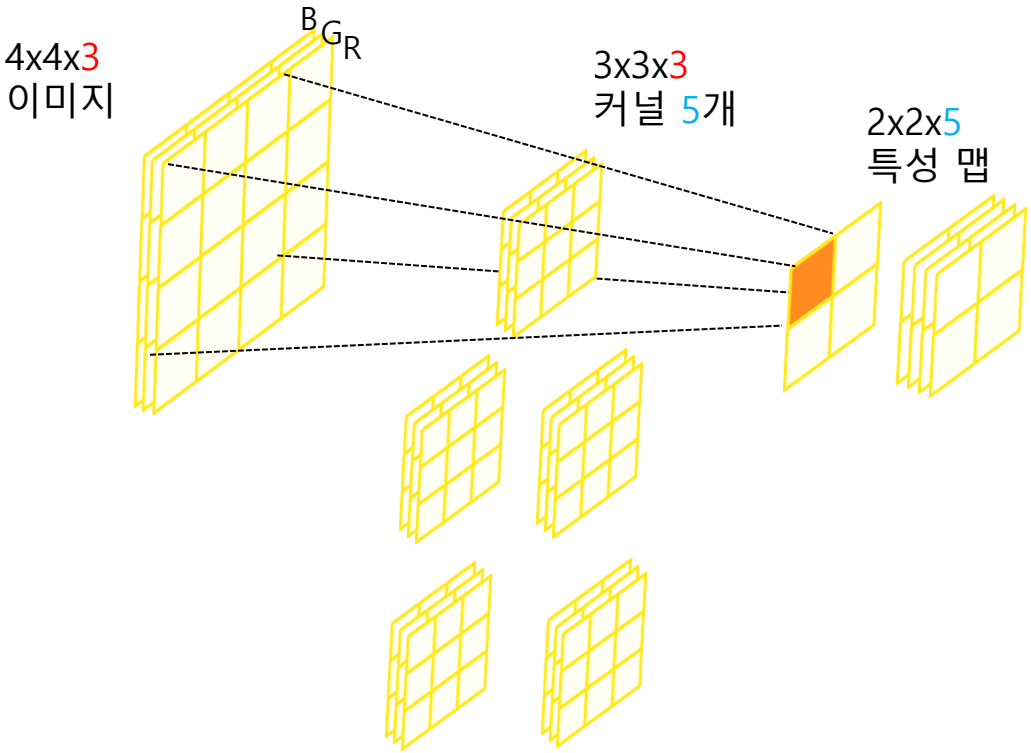
- 장점
(1) 양 극단값이 포화되지 않는다. (양수 지역은 선형적)
(2) 계산이 매우 효율적이다 (최대값 연산 1개)
(3) 수렴속도가 시그모이드류 함수대비 6배 정도 빠르다.

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

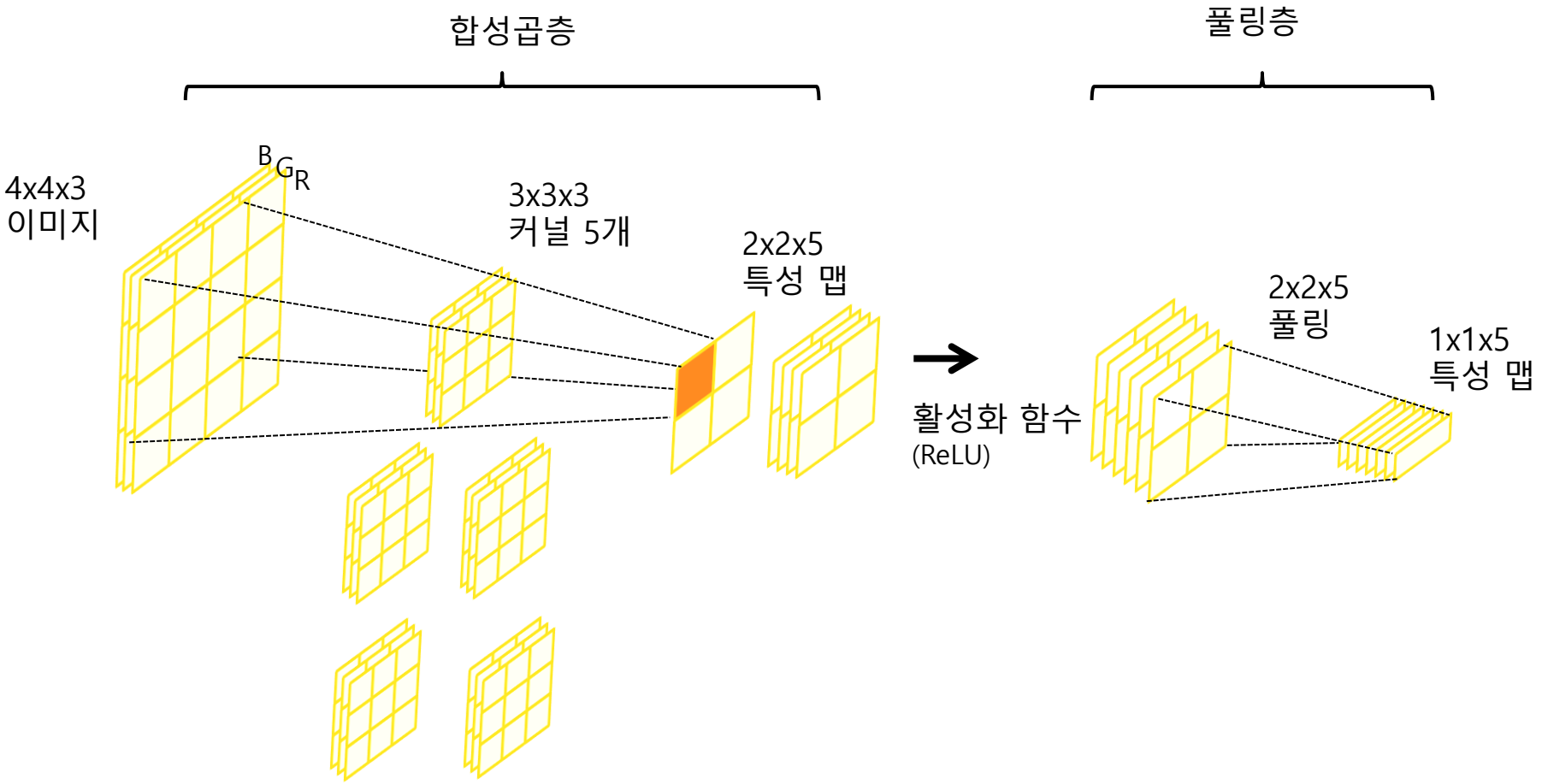
합성곱 층에서 일어나는 일



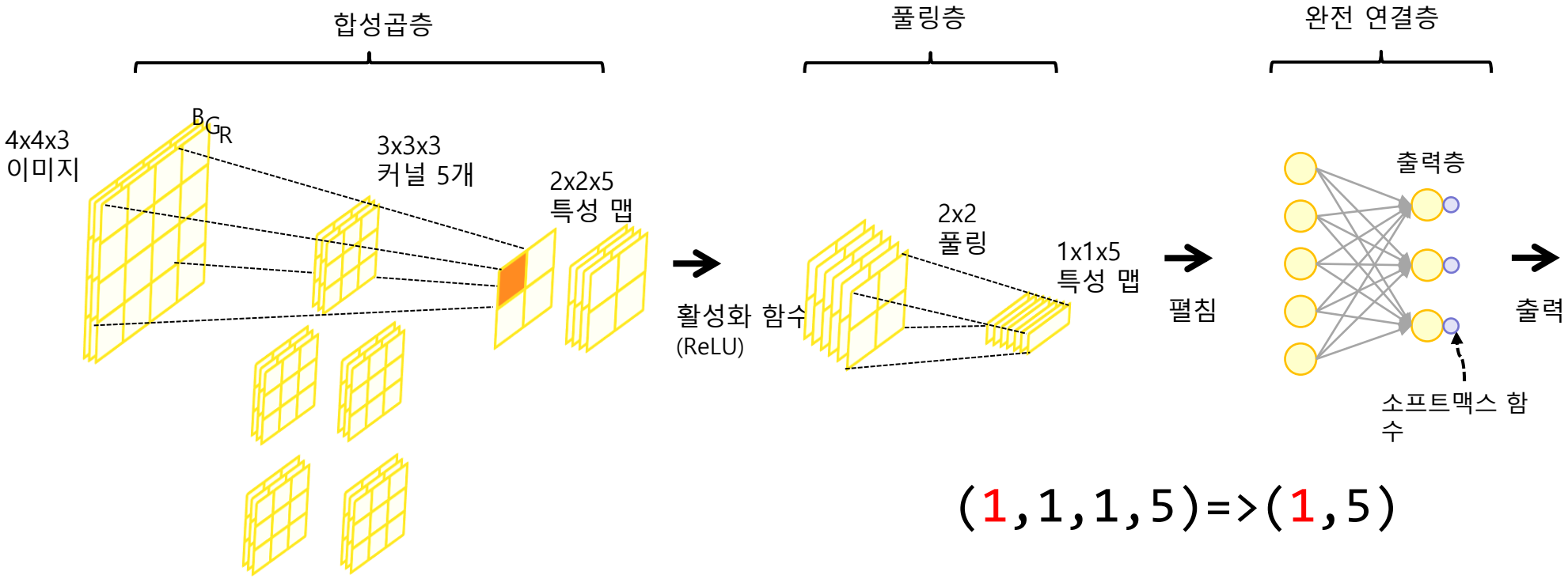
합성곱 층에서 일어나는 일



풀링 층에서 일어나는 일


합성곱층
풀링층
4x4x3
이미지
B
G
R
3x3x3
커널 5개
2x2x5
특성 맵
활성화 함수
(ReLU)
2x2x5
풀링
1x1x5
특성 맵

특성 맵을 펼쳐 완전 연결 신경망에 주입한다.



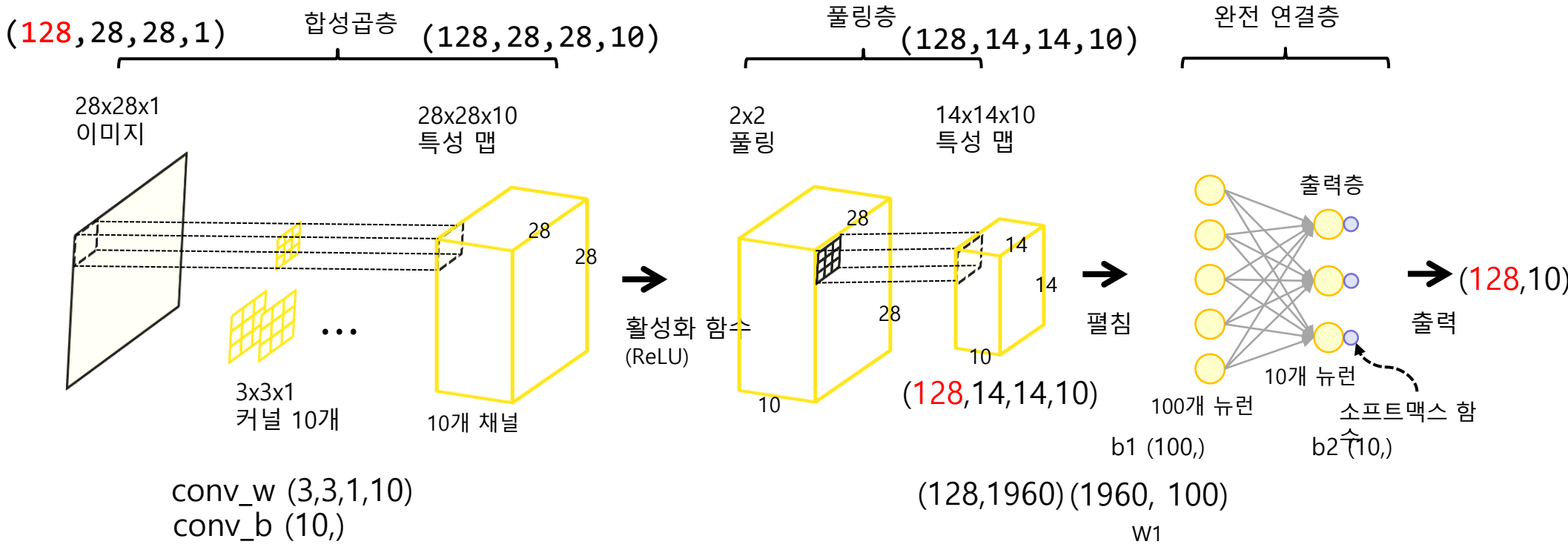
5. 합성곱 신경망 이해

5.1 합성곱 연산

5.2 합성곱 신경망 구현

5.3 케라스로 합성곱 신경망 구현

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.

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.
        f_out = tf.reshape(p_out, [x.shape[0], -1])
        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
```

- 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.

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.

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

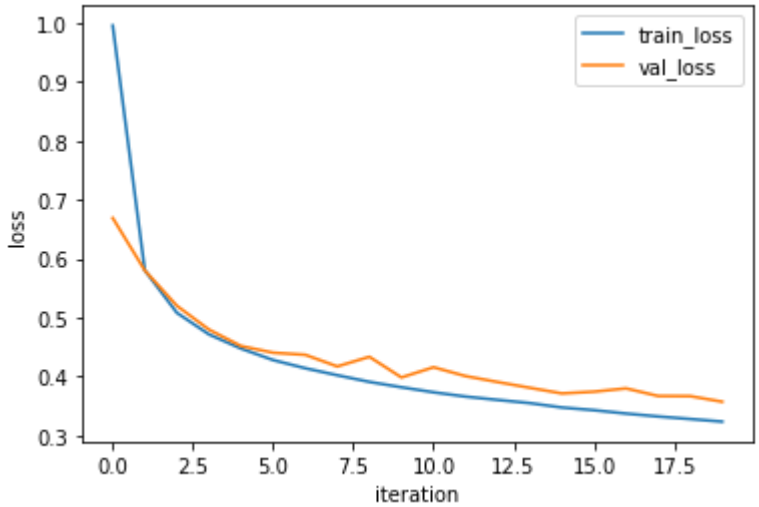
```
cn = ConvolutionNetwork(n_kernels=10, units=100, batch_size=128, learning_rate=0.01)
cn.fit(x_train, y_train_encoded,
       x_val=x_val, y_val=y_val_encoded, epochs=20)
```

```
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

5. 합성곱 신경망 이해

5.1 합성곱 연산

5.2 합성곱 신경망 구현

5.3 케라스로 합성곱 신경망 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

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
```

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

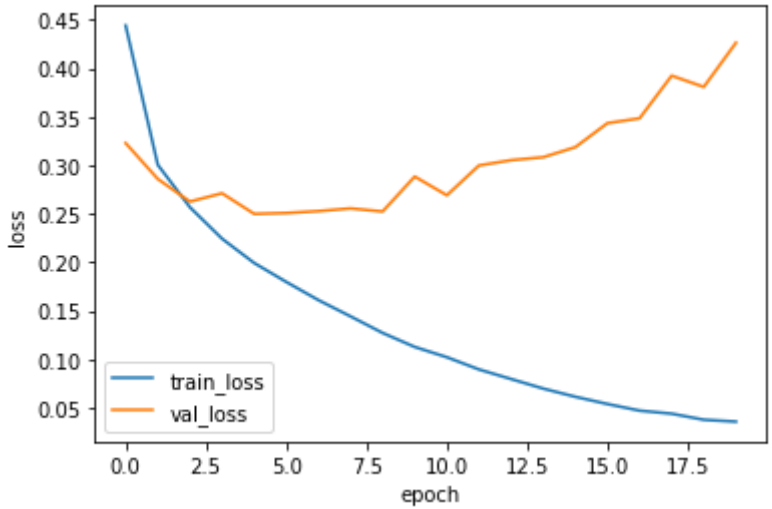
```
history = conv1.fit(x_train, y_train_encoded, epochs=20,
                    validation_data=(x_val, y_val_encoded))
```

```
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
```

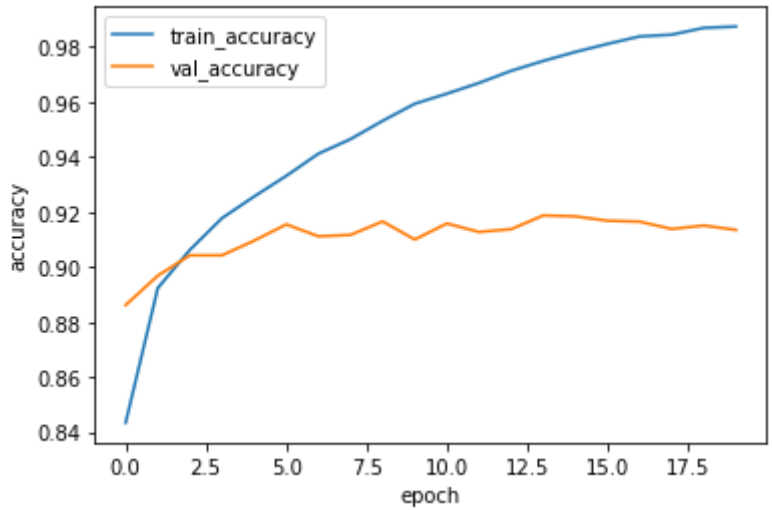
합성곱 신경망 모델 훈련하기

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()
```

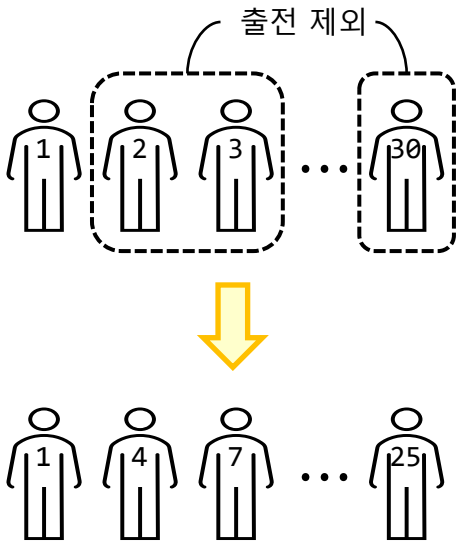


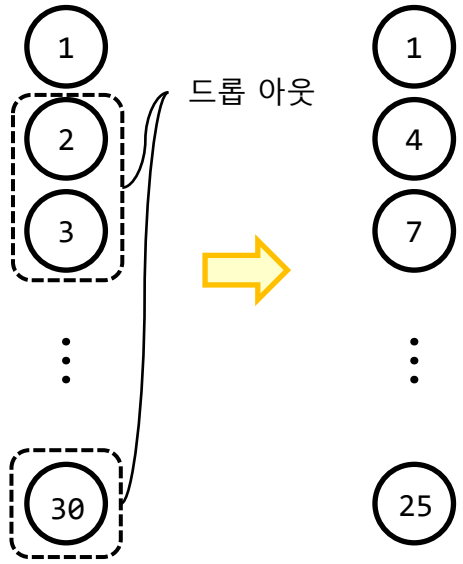

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



드롭아웃

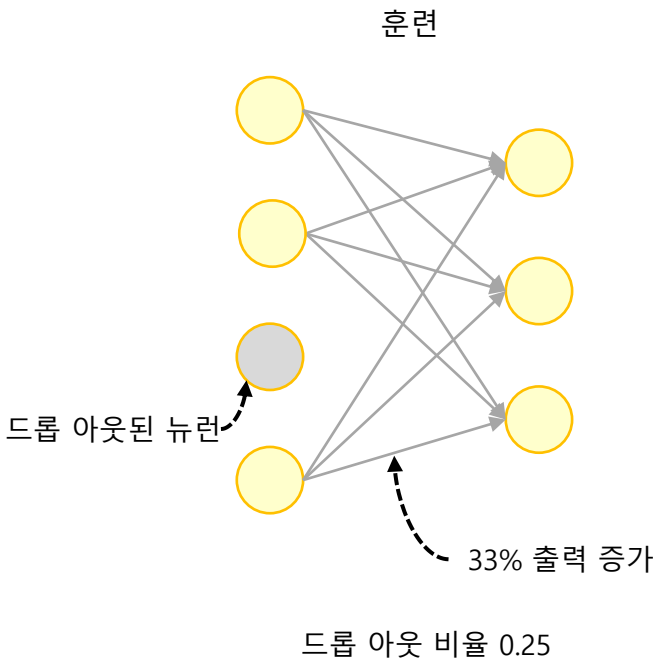
드롭아웃에 대하여

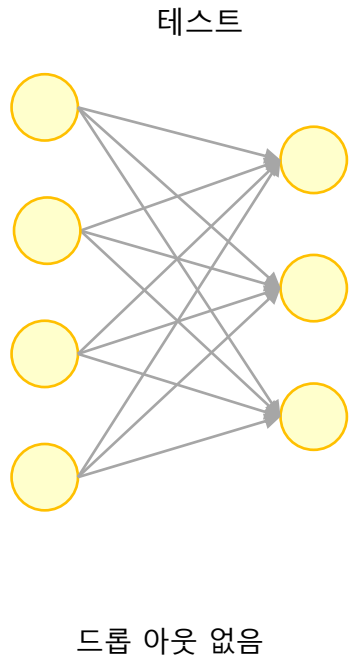


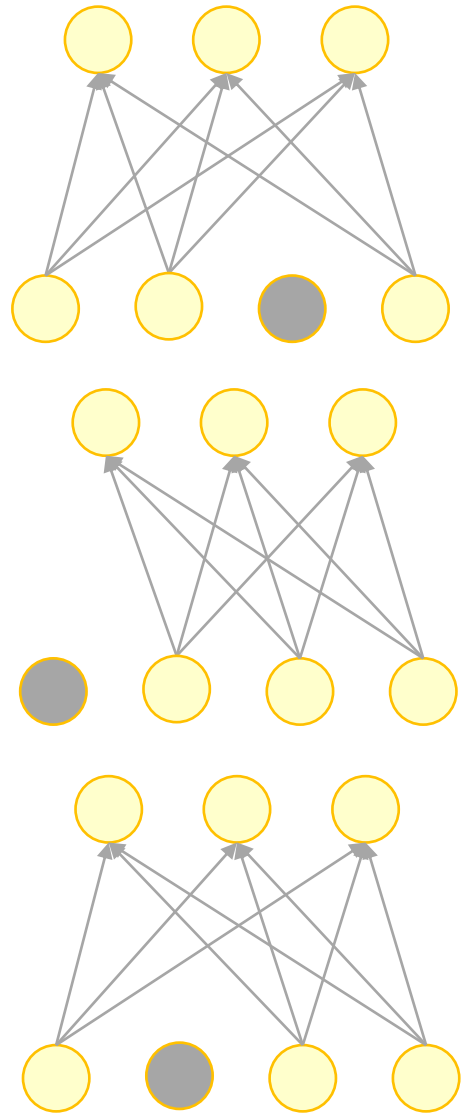


드롭아웃

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()
```

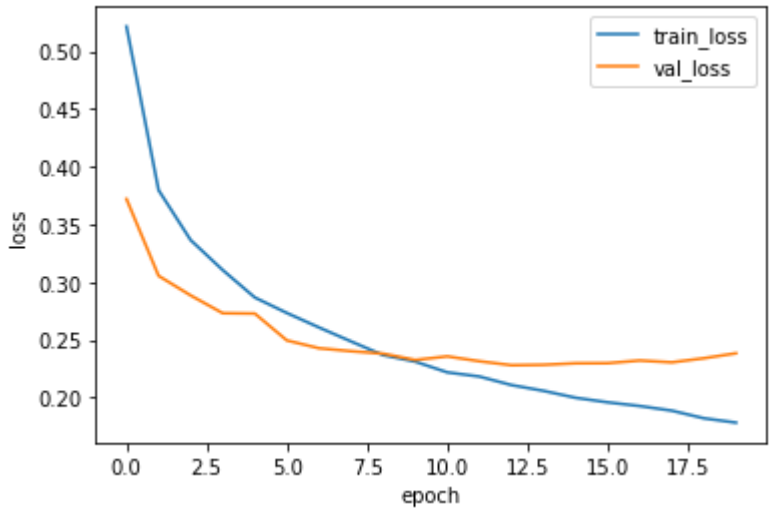
3. 훈련하기

```
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))
```

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()
```